Foraus

Programmdokumentation

Projektleitung:
Prof. Bernd Krieg-Brückner

Projektbetreuung:
Michael Fröhlich, Mattias Werner

Projektteilnehmer:
Achim Mahnke, Andreas Hinken, Andree Hähnel, Bernd Rattey, Christian Schaefer, Frank Hettling, Guido Frick, Gunnar Kavemann, Jan Hiller, Jens Warnken, Jens-Martin Brinkmann, Kai Hofmann, Michael Jäschke, Rainer Schmidtke, Tarik Ali, Trinh Le, Van Son Le, Volker Schmidtke

Universität Bremen

Inhaltsverzeichnis

Einleitung

Das vorliegende Schriftstück bildet die technische Dokumentation zum F^{or}a^{u}S–Textsystem, das im gleichnamigen Projekt entstanden ist. Vor der Verwendung dieser Dokumentation sollte der Bericht des Projektes gelesen werden, um ein Verständnis der Anforderungen und des Entwurfs zu erlangen.
Die Gliederung dieses Dokumentes orientiert sich größtenteils an der Modularisierung des Programms. Die Kapitel behandeln als gröbste Einteilung jeweils einen „Systemteil“, der auf der Entwurfsebene abgeschlossene Aufgabenblöcke umfaßt. Die Kapitel gliedern sich auf in eine kurze Übersicht über den jeweiligen Systemteil und eine Reihe von Beschreibungen der zugehörigen Module. Für die einzelnen Module wiederum besteht eine Auffächerung in öffentliche Schnittstellen und lokale Implementierungen; jeweils getrennt nach Sorten und Funktionen. Teilweise finden sich noch weitergehende Unterteilungen — diese wurden den einzelnen Autoren überlassen.
Jedes Modul enthält — wenn nötig — außerdem Angaben über Fehler, die in der Implementierung noch vorhanden sind sowie Hinweise auf bestehende Restriktionen.
Der Dokumentation liegt der Stand des Programms zu grunde, der nach Abschluß des Projektes im Dezember 1994 vorlag.
Falls der geneigte Leser Fehler im Programm findet, die im vorliegenden Papierberg nicht dokumentiert sind, gilt generell die Devise:
„It's not a bug, it's an feature!“

1. Interne Struktur

Achim Mahnke

1.1 Übersicht

1.1.1 Funktionalität

Die grundlegende Funktion des Systemteils *Interne Struktur* (IS) besteht in der dynamischen Speicherung von Daten, die auf einem externen Massenspeicher vorhanden sind und für die Zwecke des Systems benötigt werden. Daraus ergeben sich automatisch zwei Dienste, die angeboten werden müssen: Zum einen muß der Parser Funktionen vorfinden, um die eingelesenen Strukturen bekannt zu machen und so der Internen Struktur Gelegenheit zu geben, diese aufzubauen. Zum zweiten muß den anderen Systemteilen der Zugriff auf diese Strukturen gewährt werden, und zwar in einer für ihre Zwecke brauchbaren Form. Genau diese Dienste werden von der Internen Struktur wahrgenommen.
Zu den extern vorliegenden Daten gehört das Dokument in Form von ausgezeichnetem Text, die Dokument-Typ-Definition (DTD) und die Präsentationsregeln (PR).
Diese drei Datenstrukturen (`doc`, `dtd` und `pr`) sind in jeweils einem Modul realisiert. Dem Prinzip der abstrakten Datentypen folgend ist ihr konkreter Aufbau nur in dem entsprechenden Modul bekannt.
Zwei weitere Datentypen, die systemweit benötigt werden, sind in der IS (im Modul `ISTypes`) implementiert: `elementtype` und `errobject`. Ersterer wurde geschaffen, um die in der DTD definierten Namen der SGML–Elemente für den internen Gebrauch abzukürzen, weil sie für alle Objekte des Dokumentes gespeichert werden müssen. Stattdessen wird jedem Elementnamen genau ein Wert vom Typ `elementtype` zugeordnet, der fortan bei der Kommunikation zwischen Modulen stellvertretend für den Namen benutzt wird. Gleiches geschieht mit den in der PR definierten Objekten. Alle `elementtype`s werden in der Datenstruktur `elementtypeStruct` gespeichert.
Bei Werten des Typs `errobject` handelt es sich um Meldungen über Fehler, die während der Programmausführung auftreten, oder Hinweise, die dem Benutzer helfen könnten. Gespeichert werden sie in der Datenstruktur `err`. Die Bekanntgabe und Abfrage von Fehlersituationen wird systemweit mit diesen Datentypen und ihren Zugriffsoperationen abgewickelt.
Eine zentrale Aufgabe der Internen Struktur ist die Bereitstellung des Datentyps `env`. Dabei handelt es sich um das globale Environment, in dem die Datenstrukturen aller Systemteile enthalten sind. Der einzige zur Laufzeit des Programms existierende Wert dieses Typs stellt also praktisch einen Beutel für alle gespeicherten Daten dar und muß durch fast jede Funktion geschleift werden. Der Grund für diese Maßnahme liegt in der referenziellen Transparenz von „ASpecT“ und dem daraus resultierenden Verzicht auf globale Variablen.
Das Environment speichert die Datenstrukturen als Ganzes; die IS kann also nur Funktionen zum Lesen und Schreiben der Strukturen aus dem bzw. in das `env` zur Verfügung stellen.

1.1.2 Architektur

Das Gebäude der Internen Struktur besteht — wie jedes gute Haus — aus einem Fundament mit Grundmauern und einem Dach...
Wie in Abbildung 1.1 zu erkennen, bildet das Modul `ISTypes` das Fundament. Die beiden Datenstrukturen `elementtype` und `errobject` werden in nahezu allen anderen Modulen und Systemteilen benötigt und liegen deshalb ganz unten. In `ISTypes` sind auch Datentypen definiert, die von mehreren Systemteilen verwendet werden. Sie wurden hier aufgenommen, um zyklische Importe oder doppelte Deklarationen zu vermeiden.

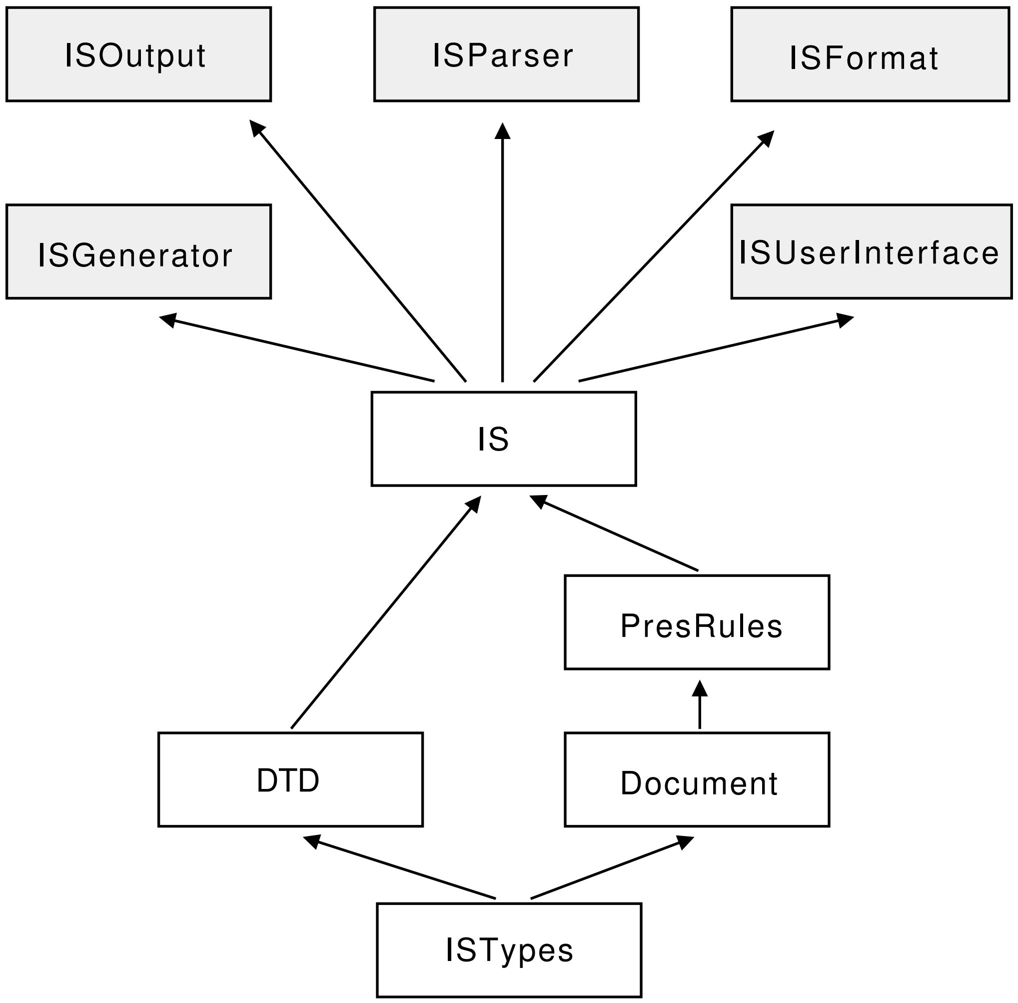

Abbildung 1.1: Modul–Struktur der IS — Die Pfeile stellen Importbeziehungen dar

Auf dem Fundament stehen als Grundmauern die drei Module `DTD`, `Document` und `PresRules`, in denen die Äquivalente der oben genannten externen Quelldaten liegen. `Document` wird von `PresRules` importiert, weil aufgrund des Vererbungsmechanismus bei der Bestimmung von Formatierungs-Attributen die Einbettung von Objekten in übergeordnete Objekte erfragt werden muß.
Das Modul `IS` setzt auf den darunterliegenden auf, indem es das `env` definiert und `dtd`, `doc` und `pr` darin ablegt. Die Datenstrukturen der anderen Systemteile werden hier natürlich ebenfalls importiert; sie sind in der Abbildung aus Platzgründen nicht dargestellt. Das gleiche Schicksal

widerfuhr dem Modul `ISPStructure`, das von `ISParser` zum Zwischenspeichern benutzt wird.
Das Dach wird von den grau unterlegten Schnittstellen–Modulen gebildet. In ihnen sind die Funktionen gebündelt, die das entsprechende Systemteil benötigt, um auf die IS zuzugreifen. Alle Funktionen arbeiten nach außen hin auf dem Environment `env` und setzen dieses nach innen hin — zu den Grundmauern und dem Fundament — auf die einzelnen Datenstrukturen um.
Die extern angebotene Funktionalität ist nicht immer unmittelbar auf Funktionen in den Basismodulen zurückzuführen; auf dem Dachboden muß also teilweise noch kräftig gewerkelt werden...
So ist z. B. die Vergabe und der Bezug auf die `elementtype`s in den Schnittstellenmodulen geregelt, da für einige Systemteile (Parser, Generator und Benutzungsoberfläche) die `elementtype`s nicht relevant sind. Sie arbeiten mit den SGML–Elementnamen und benutzen diese auch im Umgang mit der IS. Im jeweiligen Schnittstellenmodul muß also die Umsetzung auf `elementtype`s erfolgen.
In jedem dieser Module werden außerdem die Funktionen zum Erzeugen und Melden von Fehlern bereitgestellt. Realisiert ist dies über den transitiven Import aus dem Modul `IS`, wo sie eigentlich definiert sind.
Genau da ist auch der Schwachpunkt der Internen Struktur, den sie jedoch nicht zu verantworten hat. Da in `ASpecT` kein selektiver Import bzw. Export möglich ist, sind alle Typen und Funktionen, die ein Modul exportiert, dem importierenden Modul bekannt, obwohl es eigentlich nur einen Teil davon benötigt und deshalb auch nur diese kennen sollte. Verfolgt man den Pfad durch die IS und in die anderen Systemteile hinein, dann folgt daraus, daß eine Unmenge an Funktionen und Typen an Module exportiert werden, die damit überhaupt nichts zu tun haben. Dadurch muß bei der Benennung von Typen und Gestaltung von Importstrukturen zwar mehr Sorgfalt angewendet werden, aber nach anfänglichen Schwierigkeiten war dies im weiteren Verlauf unproblematisch.
In diesem Kapitel folgen nun die einzelnen Module der Internen Struktur — angefangen mit dem Fundament, fortgesetzt mit den Grundmauern und vervollständigt mit dem Dach.

1.2 ISTypes

1.2.1 Funktionalität

Dieses Modul hat zwei Aufgabenbereiche. Zum einen sind hier zwei Datenstrukturen zusammen mit ihren Zugriffsoperationen implementiert, die im gesamten System benötigt werden. Zum anderen bildet `ISTypes` einen Ort, an dem Datentypen definiert werden, die in mehreren Systemteilen Verwendung finden und hier plaziert wurden, um Probleme, die sich aus der Modul–Importstruktur ergeben, zu umgehen. So wäre es ohne diese Maßnahme wahrscheinlich zu doppelten Deklarationen oder zirkulären Importen gekommen.
Substanzielles bietet das Modul dagegen für die beiden Datenstrukturen `elementtypeStruct` und `err`. Ersterer bildet den Speicher für die Werte vom Typ `elementtype`. Dieser Typ wurde geschaffen, um die Namen der SGML–Elemente, wie sie in der eingelesenen DTD vorkommen, für den internen Gebrauch abzukürzen, weil sie für jedes Objekt des Dokumentes gespeichert werden müssen. Jedem in einer DTD definierten Elementnamen wird ein eindeutiger Wert vom Typ `elementtype` zugeordnet, der fortan bei der Kommunikation zwischen Modulen stellvertretend für den Namen benutzt wird.
Der Datentyp `err` stellt analog dazu einen Speicher für die Werte vom Typ `errobject` dar. Bei ihnen handelt es sich um Meldungen über Fehler, die während der Programmausführung auftreten, oder Hinweise, die dem Benutzer helfen könnten. Die Bekanntgabe und Abfrage von Fehlersituationen wird systemweit mit diesem Datentypen und seinen Zugriffsoperationen abgewickelt. Die zugehörigen Funktionen werden nur innerhalb der Internen Struktur verwendet. Alle Zugriffsfunktionen, die von anderen Systemteilen benötigt werden, sind in den entsprechenden Schnittstellen–Modulen definiert.

1.2.2 Entwurf

Die Datentypen `elementtype` und `errobject` sowie die zugehörigen Speicher–Strukturen `elementtypeStruct` und `err` sind opaque und wurden mit den benötigten Zugriffsfunktionen versehen.
Die `elementtype`s können in beliebiger Reihenfolge vergeben, in ihre Datenstruktur aufgenommen und angefragt werden. Die Fehler sind jedoch in einer LIFO–Schlange organisiert, deren Ordnung sich nach dem Zeitpunkt ihres Auftretens richtet. Der zuletzt gemeldete Fehler wird bei einer Anfrage als erster wieder ausgegeben, da er in der Regel die größte Bedeutung besitzt. Das Auslesen des jüngsten Fehlers führt nicht zum Löschen desselben; er muß explizit entfernt werden. Dies soll ein Nachschauen ermöglichen, ob ein Fehler für die gerade aktive Funktion von Bedeutung ist. Ist er es nicht, kann so die Fehlerbehandlung einer aufrufenden Funktion überlassen werden. Außerdem gibt es für die Fehler–Datenstruktur noch eine Funktion, die die Fehler mit der größten Relevanz (dem schwerwiegensten Fehlertyp) heraussucht, egal wann diese gemeldet wurden, um einen wichtigen, eventuell „verschütteten" Fehler ausfindig zu machen.
Zu guter letzt sind für die Datentypen `elementtype`, `errobject`, `objinfo`, `boxtype` und `finalstate` Funktionen vorhanden, die einen „Dummy–Wert" des entsprechenden Typs liefern, der keine Bedeutung hat, aber in manchen Situationen für Funktionsparameter und -ergebnisse benötigt wird. Alle Funktionen liefern stets denselben Wert zurück, so daß ein Vergleich mit == möglich ist (Die einzige Ausnahme bildet `mtObjinfo`).

1.2.3 Öffentliche Schnittstellen

1.2.3.1 Sorten

Sortenname:

elementtype elementtype

Signatur:

elementtype:: n (int :: integer).

Beschreibung:

Die Werte dieses Datentyps dienen der eindeutigen Bezeichnung von SGML–Elementen der eingelesenen DTD, sowie der physikalischen Elemente, die in den Präsentationsregeln zusätzlich definiert sind.

Sortenname:

elementtypelist elementtypelist

Signatur:

elementtypelist:: [elementtype].

Beschreibung:

Liste von `elementtype`s. Da dieser Typ häufig benötigt wird, ist er gleich hier definiert.

Sortenname:

elTypeList elTypeList

Signatur:

elTypeList:: [elementtypelist].

Beschreibung:

Wird für das `PresRules`–Modul benötigt.

Sortenname:

elementtypeStruct elementtypeStruct

Signatur:

elementtypeStruct:: Opaquer Typ. Siehe Lokalteil.

Beschreibung:

Die Datenstruktur, in der die `elementtype`s gespeichert werden. Zur Laufzeit gibt es nur einen Wert dieses Typs, da nur eine DTD mit den zugehörigen Präsentationsregeln im Speicher gehalten werden.

Sortenname:

err err

Signatur:

err:: Opaquer Typ. Siehe Lokalteil.

Beschreibung:

Die Datenstruktur, in der alle Werte vom Typ `errobject` gespeichert werden.

Sortenname:

errobject errobject

Signatur:

errobject:: Opaquer Typ. Siehe Lokalteil.

Beschreibung:

Stellt eine Fehlermeldung mit den vier Komponenten „modulName“, „errType“, „errNumber“ und „errTexts“ dar.

Sortenname:

errobjectList errobjectList

Signatur:

errobjectList:: [errobject].

Beschreibung:

Liste mit Fehlermeldungen.

Sortenname:

modulName modulName

Signatur:

modulName:: (string).

Beschreibung:

Ein String, der das Modul bezeichnet, in dem ein Fehler aufgetreten ist. Komponente eines `errobject`.

Sortenname:

errType errType

Signatur:

errType:: fatal | restriction | repair | warning | note | none.

Beschreibung:

Der Fehler–Typ; er kennzeichnet, wie schwerwiegend ein Fehler ist:

fatal Abbruch des Programms erforderlich.

restriction Aktion konnte nicht ausgeführt werden; das Programm kann jedoch weiterlaufen (z.B. falsch angegebener Dateiname für ein Dokument).

repair Fehler, der z.B. durch Einsetzen von Default–Werten repariert werden konnte.

warning Kein direkter Fehler, aber sehr bedenkenswert; z.B.: Zum höchsten Element einer DTD wurden für eine Sicht keine Präsentationsregeln definiert, so daß das gesamte Dokument in dieser Sicht ausgeblendet wird.

note Hinweis, z.B. wie etwas besser definiert oder benutzt werden sollte.

none Kein Fehler aufgetreten. Bei fehlerfreiem Ablauf einer Funktion braucht kein Fehlerobjekt mit diesem Typ erzeugt zu werden; die Funktion `getLastErr` gibt ein solches Fehlerobjekt von sich aus zurück, wenn kein Fehler gemeldet ist.

Sortenname:

errNumber errNumber

Signatur:

errNumber:: (integer).

Beschreibung:

Nummer des Fehlers im Modul, in dem er aufgetreten ist. Dient zur Unterscheidung von Fehlern, die in einem Modul auftreten können.

Sortenname:

errTexts errTexts

Signatur:

errTexts:: (strings).

Beschreibung:

Diese Zeichenketten können dafür benutzt werden, den Funktionen, die Fehler behandeln zusätzliche Informationen über den Fehler zu geben, z.B. auf welcher Seite des Dokumentes er aufgetreten ist.

Sortenname:

objID objID

Signatur:

objID:: (integer).

Beschreibung:

Jedes Objekt des Dokumentes bekommt einen eindeutigen Wert dieses Typs zugeordnet, anhand dessen die Bezugnahme auf dieses Objekt in der Kommunikation zwischen den Modulen stattfindet.

Sortenname:

objIDList objIDList

Signatur:

objIDList:: [objID].

Beschreibung:

Liste mit `objID`s.

Sortenname:

objinfo objinfo

Signatur:

objinfo:: obj (v_objID :: objID) (v_elementtype :: elementtype) (v_boxtype :: boxtype) (v_finalstate :: finalstate).

Beschreibung:

Dieser Typ dient zur Speicherung von Informationen über das mit `objID` benannte Objekt. Er enthält den Typ des SGML–Elementes, dessen Instanz das Objekt darstellt, den Formatierer–Boxtyp, der die Formatierung bestimmt und ein Kennzeichen, ob es sich um ein finales, nichtfinales oder physikalisches Objekt handelt.

Sortenname:

objinfolist objinfolist

Signatur:

objinfolist:: [objinfo].

Beschreibung:

Liste mit `objinfo`s.

Sortenname:

objtype objtype

Signatur:

objtype:: log | phy.

Beschreibung:

Dieser Typ wird vom Parser–Modul benutzt, um anzuzeigen, ob in den PR gerade Regeln für ein logisches oder ein physikalisches Objekt eingelesen werden.

Sortenname:

entitytype entitytype

Signatur:

entitytype:: views | objDef | objList | attrList | varObjList.

Beschreibung:

Dieser Typ wird vom Parser–Modul benutzt, um den gerade eingelesenen Teil einer Objekt–Definition in den PR anzuzeigen.

Sortenname:

attributetype attributetype

Signatur:

attributetype:: id | refid | none.

Beschreibung:

Dieser Typ bezeichnet die Art eines im Dokument befindlichen SGML–Attributes. `id` bedeutet, daß der Inhalt eines solchen Attributes eine eindeutige Bezeichnung für das Objekt, zu dem es definiert wurde darstellt. Mit dem Inhalt eines `refid`–Attributes wird auf eine solche Bezeichnung bezug genommen, und `none` deckt alle anderen Arten von SGML–Attributtypen ab.

Sortenname:

attributecontents attributecontents

Signatur:

attributecontents:: id objID | pid string.

Beschreibung:

Dieser Typ stellt den Inhalt eines SGML–ID–Attributes, also eine eindeutige Bezeichnung für ein Objekt im Dokument dar. Die erste Variante wird beim interaktiven Einfügen von Referenzen benutzt; in diesem Fall ist die eindeutige Kennzeichnung durch die programminterne Objekt–ID gegeben. Die zweite Möglichkeit ist beim Parsen des Dokumentes anzuwenden, weil hier die im Quell–File definierten Attributwerte in Form von Strings benutzt werden müssen.

Sortenname:

datatype datatype

Signatur:

datatype:: pcdata | cdata | ndata | empty.

Beschreibung:

Dieser Typ wird vom Parser beim Einlesen der DTD benutzt. Er bezeichnet die Art eines Element–Inhaltes.

Sortenname:

operator operator

Signatur:

operator:: (string).

Beschreibung:

Dieser Typ bezeichnet den beim Parsen einer DTD gelesenen Operator, welcher zur Spezifikation des legalen Aufbaus eines Elementes dient. Er kann folgende Werte annehmen:

'?' Optionales Element.

'+' Ein- oder mehrmaliges Wiederholen eines Elementes.

'*' Null- oder mehrmaliges Wiederholen eines Elementes.

',' Sequenz von Elementen.

'&' Sequenz von Elementen in beliebiger Reihenfolge.

'|' Wahl eines Elementes.

Sortenname:

listOfStrings listOfStrings

Signatur:

listOfStrings:: [strings].

Beschreibung:

Dieser Typ wird zwischen der IS und der Benutzungsoberfläche gebraucht, um eine Liste von möglichen Elementkombinationen zu übergeben, die an einer bestimmten Stelle im Dokument eingefügt oder gelöscht werden dürfen. Eine einzelne Kombination besteht aus einer Liste von Strings (also Typ `strings`), in der die Zeichenketten jeweils den Namen eines SGML-Elementes enthalten.

Sortenname:

finalstate finalstate

Signatur:

finalstate:: finalobj | nonfinalobj | physicalobj.

Beschreibung:

Dieser Typ gibt an, ob ein Objekt final ist (nur Text enthält), oder nichtfinal, dann beinhaltet es weitere Elemente. Ein Objekt vom Typ `physicalobj` ist ein vom Formatierer eingefügtes finales Objekt, das es im Quell–Dokument nicht gibt.

Sortenname:

boxtype boxtype

Signatur:

boxtype:: catalog | counter | index | layout | note | reference | text | head | foot | none.

Beschreibung:

Dieser Typ bezeichnet das Formatierer–Objekt, dem ein logisches Objekt zugeordnet wird. Jedes Formatierer–Objekt wird auf bestimmte Art und Weise, die in gewissem Umfang durch Parameter geändert werden kann, formatiert.

Sortenname:

docu docu

Signatur:

docu:: docu (atposition :: attrPos) (atkeepwith :: attrKeepwith).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt eine Gruppe von Attributwerten dar, die einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden sind.

Sortenname:

page page

Signatur:

page:: page (atevensidemargin :: attrNum) (atoddsidemargin :: attrNum) (atfootheight :: attrNum) (atfootsep :: attrNum) (atheadmargin :: attrNum) (atheadheight :: attrNum) (atheadsep :: attrNum) (attopmargin :: attrNum) (attextheight :: attrNum) (attextwidth :: attrNum).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt eine Gruppe von Attributwerten dar, die einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden sind.

Sortenname:

paragraph paragraph

Signatur:

paragraph:: paragraph (atbaselinediff :: attrNum) (athspace :: attrNum) (atlineskiplimit :: attrNum) (atleftindent :: attrNum) (atrightindent :: attrNum) (atparfillskip :: attrNum) (atparindent :: attrNum) (atpostparsep :: attrNum) (atpreparsep :: attrNum) (atalignment :: attrAln).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt eine Gruppe von Attributwerten dar, die einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden sind.

Sortenname:

counter counter

Signatur:

counter:: counter (atnumstart :: elementtype) (atnumincrement :: elementtype) (atnumstartvalue :: integer) (atnumincrementvalue :: integer) (atnumsymbol :: attrSym).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt eine Gruppe von Attributwerten dar, die einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden sind.

Sortenname:

catalogs catalogs

Signatur:

catalogs:: catalogs (atgroupflag :: attrFlg) (atpagenumflag :: attrFlg) (atitemizeflag :: attrFlg) (atlinkobject :: attrObj) (atnextobject :: attrObj) (atorder :: attrOrd) (atitemizesymbol :: attrIts) (atpagenumpos :: attrPps).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt eine Gruppe von Attributwerten dar, die einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden sind.

Sortenname:

notes notes

Signatur:

notes:: notes (atnotesymbol :: char) (atnotesymbolstart :: elementtype) (atnotestartvalue :: integer).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt eine Gruppe von Attributwerten dar, die einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden sind.

Sortenname:

attrPos attrPos

Signatur:

attrPos:: leftline | leftpage | rightline | rightpage | topofcolumn | topofpage | nextline | head | body | foot | lastpos | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrNum attrNum

Signatur:

attrNum:: atn (vReal :: real) (vAttrUnit :: attrUnit) (vPixel :: integer).

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrAln attrAln

Signatur:

attrAln:: left | right | centered | justified | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrSym attrSym

Signatur:

attrSym:: alph | alphl | arabic | roman | romanl | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrFlg attrFlg

Signatur:

attrFlg:: atf boolean | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrOrd attrOrd

Signatur:

attrOrd:: ascending | descending | prpage | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrPps attrPps

Signatur:

attrPps:: left | right | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrIts attrIts

Signatur:

attrIts:: ati char | elementtype | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrObj attrObj

Signatur:

attrObj:: elementtype | prnil | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrKeepwith attrKeepwith

Signatur:

attrKeepwith:: atk elementtype | prnil | prpage | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

attrUnit attrUnit

Signatur:

attrUnit:: mm | cm | in | pt | em | ex | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt die Einheit dar, in der ein numerischer Attributwert in der PR definiert wurde.

Sortenname:

fontstyle fontstyle

Signatur:

fontstyle:: bold | italic | plain | strikethrough | subscript | superscript | underline | undef.

Beschreibung:

Dieser Typ wird vom `PresRules`–Modul und dem Formatierer benutzt und stellt einen Attributwert dar, der einem logischen oder physikalischen Objekt durch die Präsentationsregeln zugeordnet worden ist.

Sortenname:

fontstyles fontstyles

Signatur:

fontstyles:: [fontstyle].

Beschreibung:

Liste mit `fontstyles`.

1.2.3.2 Funktionen

Funktionsname:

mtElementtypeStruct mtElementtypeStruct

Signatur:

mtElementtypeStruct:: elementtypeStruct.

elementtypeStruct: Leere Datenstruktur für `elementtypes`.

Beschreibung:

Die Funktion erzeugt eine leere Datenstruktur zur Aufnahme von maximal 500 Werten des Typs `elementtype`. Mit den nachfolgenden Funktionen können solche Werte bearbeitet und in die `elementtypeStruct`–Datenstruktur eingefügt und herausgenommen werden.

Implementierung:

Zur Definition der Sorte siehe Abschnitt „Lokale Implementation“ auf Seite 33. Das Array wird mit einer Größe von 500 Einträgen initialisiert, so daß maximal 500 logische und physikalische Elemente bzw. Objekte aufgenommen werden können. Dies begrenzt den Speicherplatz und sollte für alle Anwendungen reichen.

Funktionsname:

mtElementtype mtElementtype

Signatur:

mtElementtype:: elementtype.

elementtype: Spezieller Wert.

Beschreibung:

Diese Funktion liefert einen speziellen Wert, der keinem SGML–Elementnamen zugeordnet ist, sondern nur als Initial–Wert dient, wenn in irgendeiner Funktion ein `elementtype` als Parameter benötigt wird, obwohl er keine Bedeutung hat. Es wird bei jedem Aufruf derselbe Wert zurückgegeben, die Ergebnisse mehrerer Aufrufe sind also äquivalent.

Implementierung:

Als Wert wird `n 0` zurückgeliefert. Die richtigen `elementtype`s fangen bei 1 an.

Funktionsname:

equalElementtype equalElementtype

Signatur:

equalElementtype:: elementtype -> elementtype -> boolean.

elementtype: Irgendein `elementtype`.

elementtype: Irgendein `elementtype`.

boolean: Flag, ob die beiden `elementtype`s identisch sind, oder nicht.

Beschreibung:

Die Funktion liefert `true`, wenn die beiden `elementtype`s gleich sind, also das gleiche SGML– oder PR–Element bezeichnen, sonst `false`.

Implementierung:

Aus beiden Parametern wird der Integer–Wert herausgezogen und mit == verglichen.

Funktionsname:

getElementtype getElementtype

Signatur:

getElementtype:: elementtypeStruct -> string -> (boolean, elementtype).

elementtypeStruct: Struktur, in denen sich die `elementtype`s befinden.

string: Der Name eines logischen SGML–Elementes oder eines physikalischen PR–Objektes.

(boolean, elementtype): (Flag, ob es einen `elementtype` für dieses Element gibt, und der zugehörige `elementtype`, wenn dies der Fall ist.

Beschreibung:

Die Funktion prüft, ob ein `elementtype` für den übergebenen Elementnamen vorhanden ist und gibt diesen im positiven Fall mit `true` zurück, während im negativen ein `false` mit einem `mtElementtype` geliefert wird.

Implementierung:

Der übergebene String wird im Dictionary angefragt. Die Zugriffsfunktion ?! liefert genau die gewünschten Daten.

Funktionsname:

setElementtype setElementtype

Signatur:

setElementtype:: elementtypeStruct -> string -> (elementtypeStruct, elementtype).

elementtypeStruct: Struktur, in denen sich die `elementtype`s befinden.

string: Der Name eines logischen SGML–Elementes oder eines physikalischen PR–Objektes.

(elementtypeStruct, elementtype): Die neue Datenstruktur und der `elementtype`–Wert, der für das Element vergeben wurde.

Beschreibung:

Die Funktion vergibt für den übergebenen Elementnamen einen `elementtype` und speichert diesen in der Datenstruktur. Der vergebene `elementtype` und die neue Struktur werden zurückgeliefert. War der Name bereits registriert, bleibt die Struktur unverändert und der gespeicherte `elementtype` kommt zurück. Für den Aufrufer bleibt diese Tatsache transparent.

Implementierung:

Die `elementtype`s sind durch Integer–Werte voneinander unterschieden, die von 1 aufwärts an vergeben werden. Der zuletzt benutzte Wert wird in der `elementtypeStruct`–Struktur gespeichert, um in dieser Funktion die Zahl für den nächsten `elementtype` bestimmen zu können.

Kann mittels `getElementtype` kein Wert für den übergebenen Elementnamen gefunden werden, wird ein neuer vergeben, in das Dictionary eingefügt und die Angabe über den letzten `elementtype`–Wert in der Datenstruktur aktualisiert.

Funktionsname:

getElementName getElementName

Signatur:

getElementName:: elementtypeStruct -> elementtype -> string.

elementtypeStruct: Struktur, in denen sich die `elementtype`s befinden.

elementtype: Irgendein `elementtype`.

string: Der zu dem `elementtype` gehörige Elementname.

Beschreibung:

Diese Funktion gibt den zu dem `elementtype` gehörigen Elementnamen zurück. Ist dieser nicht registriert, wird ein Leerstring geliefert.

Implementierung:

Der Integer–Wert des `elementtype`s wird als Index für eine Anfrage im Array benutzt. Ist zu diesem Index kein String gespeichert, liefert die Zugriffsfunktion ! eine leere Zeichenkette.

Funktionsname:

removeElementtype removeElementtype

Signatur:

removeElementtype:: elementtypeStruct -> elementtype -> elementtypeStruct.

elementtypeStruct: Struktur, in denen sich die `elementtype`s befinden.

elementtype: Irgendein `elementtype`.

elementtypeStruct: Die neue Datenstruktur.

Beschreibung:

Diese Funktion soll den übergebenen `elementtype` aus der Datenstruktur löschen; sie ist jedoch *nicht implementiert* und gibt die Struktur unverändert zurück.

Die Funktion wurde aufgenommen, weil zur Definition einer Datenstruktur auch eine Operation zum Löschen seiner Inhalte gehört, aber sie wird nicht benötigt, weil zur Laufzeit kein Element gelöscht werden muß und beim Neuladen eines Dokumentes auch eine neue DTD und PR eingelesen wird, so daß die alte `elementtypeStruct` in diesem Fall ohnehin obsolet ist und eine ganz neue angelegt wird.

Funktionsname:

hashElementtype hashElementtype

Signatur:

hashElementtype:: elementtype -> integer.

elementtype: Irgendein `elementtype`.

integer: Hashwert für diesen `elementtype`.

Beschreibung:

Diese Funktion liefert für jeden `elementtype` einen *eindeutigen* Integer–Wert, so daß sie als Hashfunktion benutzt werden kann, wenn ein anderes Modul ein Dictionary mit `elementtype` als Schlüssel benötigt (das `PresRules`–Modul tut dieses).

Implementierung:

Ziemlich simpel: Jeder `elementtype` besteht praktisch aus nichts anderem, als einem eindeutigen Integer–Wert zur Unterscheidung von seinen Kollegen. Dieser wird einfach zurückgeliefert.

Funktionsname:

createError createError

Signatur:

createError:: modulName -> errType -> errNumber -> errTexts -> errobject.

modulName: Name des Moduls, in dem der Fehler aufgetreten ist.

errType: Art des Fehlers.

errNumber: Eindeutige Nummer des Fehlers im Modul, das den Fehler meldet.

errTexts: Strings, die genauere Informationen über den Fehler geben.

errobject: Aus den Angaben erzeugtes Fehlerobjekt.

Beschreibung:

Diese Funktion erzeugt aus den übergebenen Daten ein Fehlerobjekt, das mit `announceError` dem System gemeldet werden kann.

Implementierung:

Die Angaben werden einfach zu einem Fehlerobjekt zusammengefügt, so wie der Datentyp `errobject` definiert ist.

Funktionsname:

readError readError

Signatur:

readError:: errobject -> (modulName, errType, errNumber, ErrTexts).

errobject: Ein Fehlerobjekt.

(modulName, errType, errNumber, ErrTexts): Der Name des Moduls, in dem der Fehler aufgetreten ist, die Art des Fehlers, die eindeutige Nummer des Fehlers im Modul, das den Fehler meldet sowie Zeichenketten, die genauere Informationen über den Fehler enthalten.

Beschreibung:

Diese Funktion liest aus einem Fehlerobjekt dessen Komponenten und gibt sie zurück.

Implementierung:

Jede Komponente eines Fehlerobjektes hat einen Selektor/Modifikator, der hier zum Zugriff benutzt wird.

Funktionsname:

mtErr mtErr

Signatur:

mtErr:: err.

err: Leere Fehler–Datenstruktur.

Beschreibung:

Diese Funktion liefert eine leere Datenstruktur zur Aufnahme von Fehlerobjekten.

Implementierung:

Da die `err`–Struktur mit Hilfe einer Liste implementiert ist, wird hier eine leere Liste zurückgegeben.

Funktionsname:

announceError announceError

Signatur:

announceError:: errobject -> err -> err.

errobject: Ein Fehler.

err: Die Fehler–Datenstruktur.

err: Die neue Datenstruktur, mit dem gemeldeten Fehler.

Beschreibung:

Diese Funktion hängt den gemeldeten Fehler in die Datenstruktur ein, so daß er bei einem nachfolgenden `getLastError` als zuletzt aufgetretener Fehler angezeigt würde.

Implementierung:

Der Fehler wird vor dem ersten Element der Liste eingefügt.

Funktionsname:

getLastError getLastError

Signatur:

getLastError:: err -> errobject.

err: Die Fehler–Datenstruktur.

errobject: Der zuletzt gemeldete Fehler.

Beschreibung:

Diese Funktion gibt das jüngste Fehlerobjekt (also den zuletzt gemeldeten Fehler) zurück. Der Fehler bleibt in der Datenstruktur, bis er mit `removeError` gelöscht wurde. Ist kein Fehler vorhanden, dann wird ein `errobject` zurückgegeben, dessen `errType` 'none' ist; die anderen Komponenten sind mit belanglosen Werten gefüllt und sollten nicht ausgewertet werden.

Implementierung:

Wenn die Liste der Fehler leer ist, wird ein `mtError` zurückgeliefert, ansonsten das erste Objekt der Liste.

Funktionsname:

removeError removeError

Signatur:

removeError:: err -> err.

err: Die Fehler–Datenstruktur.

err: Die neue Datenstruktur, ohne den zuletzt gemeldeten Fehler.

Beschreibung:

Diese Funktion löscht das jüngste Fehlerobjekt in der Datenstruktur. Ist kein Fehler gemeldet, passiert nichts, was für den Aufrufer transparent bleibt.

Implementierung:

Ist die Fehler–Liste nicht leer, wird das erste Objekt entfernt.

Funktionsname:

maxErrObjects maxErrObjects

Signatur:

maxErrObjects:: err -> errobjectList.

err: Die Fehler–Datenstruktur.

errobjectList: Liste mit den Fehlern, die den höchsten `errType`–Wert haben.

Beschreibung:

Diese Funktion liefert die Fehler mit dem höchsten `errType` von allen gemeldeten Fehlern. Die Wertigkeit der Fehlertypen ergibt sich durch die Reihenfolge bei der Definition dieses Datentyps (siehe Seite 16): Am höchsten ist `fatal`, am niedrigsten `none`.

Alle Fehler der gelieferten Liste haben denselben `errType`–Wert. War kein Fehler vorhanden, dann wird eine Liste mit genau einem leeren Fehlerobjekt (`mtError`) zurückgegeben, dessen `errType` `none` ist. Seine restlichen Komponenten sind nicht auszuwerten.

Implementierung:

Die Liste der Fehler wird zunächst mit Hilfe von `maxLevel` auf den höchsten Fehlertyp hin durchsucht und anschließend eine Filter–Operation mit dem gefundenen Wert durchgeführt. Ist die Fehler–Liste leer, kommt ein `mtError` zurück.

Funktionsname:

mtObjinfo mtObjinfo

Signatur:

mtObjinfo:: objID -> objinfo.

objID: Irgendeine Objekt–ID.

objinfo: Ein „leerer“ Wert vom Typ `objinfo`.

Beschreibung:

Diese Funktion liefert einen Wert vom Typ `objinfo` zurück, der außer der übergebenen Objekt–ID nur Dummy–Werte enthält. Er stellt keinen sinnvollen Wert dar und wird benötigt, wenn ein `objinfo` als Parameter oder Ergebnis einer Funktion erforderlich ist, aber keine Bedeutung hat.

Implementierung:

Neben der Objekt–ID enthält der gelieferte Wert ein `mtElementtype`, ein `mtBoxtype` und ein `mtFinalstate`.

Funktionsname:

mtBoxtype mtBoxtype

Signatur:

mtBoxtype:: boxtype.

boxtype: Ein „leerer“ Wert vom Typ `boxtype`.

Beschreibung:

Diese Funktion liefert einen Wert vom Typ `boxtype` zurück, der nur einen Dummy–Wert enthält. Er stellt keinen sinnvollen Wert dar und wird benötigt, wenn ein `boxtype` als Parameter oder Ergebnis einer Funktion erforderlich ist, aber keine Bedeutung hat.

Implementierung:

Zurückgegeben wird der `boxtype` `none`.

Funktionsname:

mtFinalstate mtFinalstate

Signatur:

mtFinalstate:: finalstate.

finalstate: Ein „leerer“ Wert vom Typ `finalstate`.

Beschreibung:

Diese Funktion liefert einen Wert vom Typ `finalstate` zurück, der nur einen Dummy–Wert enthält. Er stellt keinen sinnvollen Wert dar und wird benötigt, wenn ein `finalstate` als Parameter oder Ergebnis einer Funktion erforderlich ist, aber keine Bedeutung hat.

Implementierung:

Zurückgegeben wird der `finalstate` `nonfinalobj`.

1.2.4 Lokale Implementation

1.2.4.1 Sorten

Sortenname:

err err

Signatur:

err:: List ACTUAL SORTS data = errobject. list = err. END+

Beschreibung:

Die Datenstruktur, in der alle gemeldeten Fehler gespeichert werden, ist eine Liste, die durch eine Parametrisierung des ASpecT–„List“–Moduls erzeugt wird.

Sortenname:

elementtypeStruct elementtypeStruct

Signatur:

elementtypeStruct:: et (lastType :: integer) (dictio :: eTypeDict) (table :: eTypeArray).

Beschreibung:

Die `elementtype`s werden in einem Dictionary (`eTypeDict`) gehalten, in dem zu jedem String, der mit `setElementtype` übergeben wurde, eine ganze Zahl gespeichert wird. Durch die Verwendung eines Dictionary mit dem Zugriff durch Hashing ist die Funktion `get-Elementtype` sehr effizient.

Gleichzeitig wird der Elementname in einem Array (`eTypeArray`) gespeichert, in dem der `elementtype` als Index dient. Damit ist zwar die Funktion `setElementtype` etwas aufwendiger und die gehaltene Datenmenge verdoppelt sich, aber das fällt nicht besonders ins Gewicht, wenn man bedenkt, daß eine DTD vielleicht 100 Elemente enthält (die Aufnahme von wesentlich mehr Elementen würde sie praktisch unlesbar und somit umbenutzbar machen). Dagegen kann so die Funktion `getElementName` sehr effizient gestaltet werden, weil der `elementtype` direkt als Array–Index verwendet wird, um den String zu finden.

Die Integer–Komponente enthält immer den letzten vergebenen `elementtype`–Wert, der bei jedem Aufruf von `setElementtype` um eins erhöht wird.

Sortenname:

errobject errobject

Signatur:

errobject:: er (mn :: modulName) (ty :: errType) (nu :: errNumber) (tx :: errTexts).

Beschreibung:

Werte dieser Sorte stellen Fehler dar, die in der Programmausführung auftreten. Die einzelnen Komponenten (genauere Beschreibung siehe Seite 15) sind mit Selektoren/Modifikatoren ausgestattet.

1.2.4.2 Funktionen

Funktionsname:

hashString hashString

Signatur:

hashString:: string -> integer.

string: Ein Elementname.

integer: Ein möglichst eindeutiger Wert für diesen Namen.

Beschreibung:

Diese Funktion bildet die Hash–Routine für das Dictionary, das die Elementnamen als Schlüssel zum schnellen Auffinden eines zugehörigen `elementtype`s verwendet. Die Funktion liefert *keine eindeutigen* Werte, aber sie dürften hinreichend verschieden sein, um das Dictionary nicht zu einer Anzahl von Listen verkommen zu lassen …

Implementierung:

Die Funktion `doHashString` wird mit dem String sowie dessen Länge gefüttert.

Funktionsname:

doHashString doHashString

Signatur:

doHashString:: chars -> integer -> integer.

chars: Liste der noch zu berechnenden Zeichen.

integer: Der Index des ersten Buchstabens in der Zeichenkette.

integer: Der Hashwert für das erste Zeichen in der Zeichenkette.

Beschreibung:

Diese Funktion rechnet für die übergebenen Zeichen einen Hashwert aus.

Implementierung:

Der ASCII–Wert jedes Buchstabens wird mit dem Index des betreffenden Zeichens innerhalb der ursprünglichen Zeichenkette multipliziert und diese Werte addiert. Die einzelnen Zeichen der Kette werden dabei der Einfachheit halber von Rechts nach Links durchgezählt, die Hashwerte aber von Links nach Rechts ermittelt; das spielt jedoch keine Rolle.

Die leere Zeichenkette bekommt immer den Wert 0.

Funktionsname:

mtError mtError

Signatur:

mtError:: errobject.

errobject: Ein „leeres“ Fehlerobjekt.

Beschreibung:

Diese konstante Funktion liefert ein Fehlerobjekt mit einem Fehlertyp `none`, der Fehlernummer 0, einem Leerstring als Modulnamen und einer leeren Liste als `errTexts`. Er stellt keinen Fehler dar und wird benötigt, wenn ein Wert vom Typ `errobject` als Parameter oder Ergebnis einer Funktion erforderlich ist, aber keine Bedeutung hat.

Implementierung:

Mit den Dummy–Werten wird ein Fehlerobjekt erzeugt.

Funktionsname:

maxLevel maxLevel

Signatur:

maxLevel:: err -> errType.

err: Die Fehler–Datenstruktur.

errType: Höchster `errType`–Wert aller gemeldeten Fehler.

Beschreibung:

Diese Funktion liefert den höchsten `errType`–Wert aller gemeldeten Fehler. Ist kein Fehler gemeldet, wird ein `none` zurückgegeben.

Implementierung:

Die Fehler–Liste wird von Vorne nach Hinten durchgegangen und dabei der `errType` des gerade betrachteten Fehlerobjektes mit dem höchsten Wert aller restlichen Objekte verglichen, der durch einen rekursiven Aufruf mit der Restliste ermittelt wird. Für den Vergleich wird den `errType`s durch die Funktion `errTypeToInt` jeweils ein Integer–Wert zugeordnet, so daß ein mathematisches > als Vergleichsoperation hergenommen werden kann.

Funktionsname:

errTypeToInt errTypeToInt

Signatur:

errTypeToInt:: errType -> integer.

errType: Irgendein `errType`–Wert.

integer: Bewertungszahl dieses `errType`s.

Beschreibung:

Diese Funktion ordnet den `errType`–Werten in absteigender Reihenfolge ihrer Bedeutung eine Integer–Zahl zu, so daß der schwerwiegendste Fehlertyp `fatal` die höchste Nummer (5) und der bedeutungsloseste Typ `none` die Nummer 0 bekommt.

Implementierung:

Über ein 'case'–Konstrukt werden den unterschiedlichen Fehlertypen ihre Wert zugeordnet.

Funktionsname:

isMaxError isMaxError

Signatur:

isMaxError:: errType -> errobject -> boolean.

errType: Irgendein `errType`–Wert.

errobject: Irgendein Fehler–Objekt.

boolean: Flag, ob der `errType`–Wert des Fehlerobjektes dem separat übergebenen Wert entspricht.

Beschreibung:

Diese Funktion überprüft, ob der `errType`–Wert des Fehlerobjektes dem separat übergebenen Wert entspricht. Trifft dies zu, wird `true`, sonst `false` geliefert.

Implementierung:

Der `errType`–Wert wird mittels seines Selektors aus dem Fehlerobjekt ausgelesen und verglichen.

Funktionsname:

convertErrLists convertErrLists

Signatur:

convertErrLists:: err -> errobjectList.

err: Liste mit Fehlern.

errobjectList: Liste mit denselben Fehlern.

Beschreibung:

Diese Funktion konvertiert eine Fehlerliste vom Typ `err` in eine Liste vom Typ `errobjectList`, die dieselben Fehlerobjekte in derselben Reihenfolge enthält. Sie wird benötigt, weil die Liste der Fehler, die `maxErrObjects` ermittelt, vom Typ `err` ist und als Liste vom Typ `errobjectList` zurückgegeben werden muß.

Implementierung:

Es wird die Funktion `workingConvErrLists` aufgerufen, die die eigentliche Arbeit macht.

Funktionsname:

workingConvErrLists workingConvErrLists

Signatur:

workingConvErrLists:: err -> errobjectList -> errobjectList.

err: Liste mit noch zu kopierenden Fehlern.

errobjectList: Liste mit bisher kopierten Fehlern.

errobjectList: Liste mit allen Fehlern, die beim ersten Aufruf der Funktion in `err` vorhanden waren.

Beschreibung:

Diese Funktion kopiert alle Objekte einer Fehlerliste vom Typ `err` in eine Liste vom Typ `errobjectList`, in derselben Reihenfolge.

Implementierung:

Durch rekursive Aufrufe wird jeweils das erste Objekt der ersten Liste vor das erste Objekt der zweiten Liste gehängt.

1.2.5 Fehler und Restriktionen

Eine Restriktion enthält dieses Modul: Die Anzahl der SGML–Element- bzw. PR–Objektnamen, für die ein `elementtype` vergeben werden kann, ist aus Effizienz– und Speicherplatzgründen auf 500 beschränkt. Allerdings dürfte es nur sehr wenige DTD geben, die mehr als 100 Elementnamen enthalten, und die Präsentationsregeln bestehen vorwiegend aus Definitionen zu den Elementen der DTD. Selbst wenn in den PR nochmal 100 Objekte definiert werden, ist die maximale Kapazität nicht einmal zur Hälfte erreicht.

1.3 DTD

Kai Siegele, Jens Warnken

Zu einem Dokument, das mit SGML ausgezeichnet ist, gehören immer mindestens zwei Dateien:

1. Die erste Datei enthält das Dokument, das mit SGML-Auszeichnungen durchsetzt ist.

2. Die zweite Datei ist die DTD. Diese enthält Regelbeschreibungen, nach denen die logischen Elemente in dem SGML-Text gesetzt sein müssen. So kann z.B. eine Regel besagen, daß das logische Element „Kapitel" sich aus dem mindestens einmaligen Vorkommen des Elementes „Absatz" definiert.

Das Modul `DTD.AS` verwaltet die DTD zu einem SGML-Dokument, das mit unserem System bearbeitet wird.

1.3.1 Funktionalität

In unserem System werden Dokumente mit SGML-Auszeichnung bearbeitet. Dabei müssen auch die von der DTD vorgegebenen Regeln beachtet werden. So kann z.B. nicht an jeder beliebigen Stelle jedwegiges logische Element eingefügt werden. Zudem kann die DTD das Löschen eines logischen Elementes verbieten. Um während der Bearbeitung in unserem System ein nach der DTD syntaktisch korrektes SGML-Dokument zu erhalten, ist es nötig, die Änderungen an dem Dokument auf ihre Ausführbarkeit zu überprüfen.
Dazu bietet unser Modul drei größere Funktionsgruppen an:

1. Funktionen zum Einlesen der DTD. Die DTD wird automatisch nach dem Laden des SGML-Dokumentes, das bearbeitet werden soll, eingelesen und in unserer Datenstruktur gespeichert.

2. Funktionen, die auf der Datenstruktur arbeiten. Diese gewährleisten die syntaktische Korrektheit aller Aktionen auf dem SGML-Dokument.

3. Funktionen zum Speichern der DTD. In späteren Versionen unseres Programmes wird es evtl. möglich sein, die DTD interaktiv zu ändern. Die Änderungen müssen gespeichert werden. Zur Zeit wird die DTD automatisch mit dem SGML-Dokument gespeichert, weil beim Einlesen der DTD verkürzte Schreibweisen aufgelöst werden, und diese Änderungen auch gespeichert werden müssen.

1.3.2 Entwurf

Sieht man sich eine DTD in einem der SGML-Fachbücher an, so fällt ihre Tabellenform ins Auge. Zur linken Seite der Name des Elementes, dessen Regeln definiert werden und zur rechten Seite die Regel selbst. Eine Regelbeschreibung ist eine Zeile aus dieser Tabelle. Ähnlich diesem Erscheinungsbild haben wir unsere Datenstruktur gewählt. Sie besteht aus Listen von Listen. Die eine Liste ist eine Zeile aus der DTD-Tabelle. Diese Listen sind in einer weiteren Liste zusammengefaßt und geben die Tabelle selbst an. In Abbildung 1.2 ist eine DTD angegeben.
Die Wahl zu dieser Datenstruktur war umso logischer, da wir in einer funktionalen Programmiersprache implementierten. Das Listenkonzept ist ein typisches Feature von funktionalen Programmiersprachen. Demzufolge werden auch mächtige Konstrukte zum Bearbeiten von Listen angeboten, die wir in unseren Algorithmen auch verwendeten.

Abbildung 1.2: Eine (unvollständige) Beispiel-DTD

```
<!--       ELEMENT   MIN    CONTENT -->
<!ELEMENT  textbook         (front?, body) >
<!ELEMENT  front            EMPTY >
                     ⋮
<!--       ELEMENT   NAME   VALUE -->
<!ATTLIST  textbook  refid  IDREF
                     ⋮
```

1.3.3 Öffentliche Schnittstellen

1.3.3.1 Sorten

Im folgenden sollten ursprünglich nur die Datentypen erklärt werden, die im Source-Code öffentlich gehalten sind. Zum besseren Verständnis der späteren Funktionsbeschreibungen, die auf diesen Datentypen arbeiten, werden hier auch Datentypen erklärt, die als `LOCAL` deklariert sind.

Sortenname:

dtd dtd

Signatur:

dtd:: union dtd_list attr_list integer dtd_saves

Beschreibung:

Hinter diesem polymorphen Datentyp verbirgt sich unsere Datenstruktur. Ihr eigentlicher Aufbau ist zwar erst im `LOCAL`-Teil des ASpecT-Codes deklariert (demzufolge ist er ganz und gar nicht öffentlich), wird hier aber zum besseren Verständnis der späteren Funktionenbeschreibungen schon erklärt. Die Datenstruktur besteht aus vier Elementen:

dtd_list Dieses Element beinhaltet die Regelbeschreibungen aus der DTD. In ihm werden logische Elemente benannt und ihr Aufbau festgelegt.

attr_list In diesem Datentyp sind die Attributbeschreibungen gespeichert.

integer Eine Hilfsvariable, die beim Laden der DTD die Schachtelungstiefe der nächsten Einfügeposition speichert. Ähnlich einem Stack wird beim Aufruf von `beginGroup` diese Hilfsvariable um eins erhöht und beim Aufruf von `endGroup` um eins erniedrigt.

dtd_saves Dies ist eine Hilfsstruktur, die zum Speichern der DTD aufgebaut und hier gehalten wird. Sie ist nötig, um das sequentielle Auslesen der DTD-Datenstruktur zu ermöglichen.

Der Konstruktor `union` hält diese vier Elemente zusammen.

Sortenname:

dtd_list dtd_list

Signatur:

dtd_list:: [dtd_element]

Beschreibung:

In dem Element `dtd_list` sind alle logischen Elemente der DTD und die Regelbeschreibungen zu ihrem Aufbau gespeichert.

Sortenname:

dtd_element dtd_element

Signatur:

dtd_element:: rule elementtype dtd_parts

Beschreibung:

Eine Zeile aus der DTD. Der Bezeichner `elementtype` benennt das logische Element, dessen Regelbeschreibungen in `dtd_parts` folgen. Der Konstruktor `rule` hält diese zwei Elemente zusammen. Z.B. bei der ersten Zeile aus Abb. 1.2 auf Seite 40 hätte `elementtype` den Wert `textbook` und in `dtd_parts` wäre die Regelbeschreibung `(front?, body)` gespeichert.

Sortenname:

dtd_parts dtd_parts

Signatur:

dtd_parts:: [dtd_part]

Beschreibung:

Dies ist eine Liste von DTD-Operatoren und Operanden. Hier wird der logische Aufbau eines DTD-Elementes beschrieben.

Sortenname:

dtd_part dtd_part

Signatur:

dtd_part:: opt dtd_parts | pos dtd_parts | mal dtd_parts | seq dtd_parts | xor dtd_parts | and dtd_parts | ele elementtype

Beschreibung:

Für jeden Operator in der DTD gibt es einen Konstruktor (`opt` z.B. für die optionalen Elemente; in der DTD mit '?' ausgezeichnet). Jeder Operator hat entweder eine Liste von weiteren Operatoren als Operand oder ein logisches Element, das in `ele elementtype` gespeichert wird. Die Regelbeschreibung aus der ersten Zeile aus Abb. 1.2 sähe folgendermaßen aus: `seq[opt[ele front], ele body]`.
Da jede Liste ohnehin aus einer Sequenz mehrerer Elemente besteht, dient der `seq`-Operator nur zur besseren Lesbarkeit. Bei vielen späteren Auswertungen wird er weggelassen.
Jeder `dtd_part` kann als regulärer Ausdruck interpretiert werden. Die oben beschriebene Regelbeschreibung stellt den regulären Ausdruck $front?, body$ dar.

Sortenname:

attr_list attr_list

Signatur:

attr_list:: [attr_list_part]

Beschreibung:

Hierin sind die Attributdefiniton zu logischen Elementen gespeichert.

Sortenname:

attr_list_part attr_list_part

Signatur:

attr_list_part:: al elementtype attr_defs

Beschreibung:

Der Konstruktor `al` hält zwei Elemente zusammen: `elementtype` bezeichnet das logische Element, dessen Attribute in `attr_defs` definiert sind. In dem Beispiel aus Abb. 1.2 hätte `elementtype` den Wert `textbook` und in `attr_defs` wäre die Attributdefinition gespeichert.

Sortenname:

attr_defs attr_defs

Signatur:

attr_defs:: [attr_def]

Beschreibung:

Die Attributdefinitionen zu einem logischen Element sind hierin gespeichert.

Sortenname:

attr_def attr_def

Signatur:

attr_def:: ad attributetype string

Beschreibung:

Der Konstruktor `ad` hält zwei Elemente zusammen: `attributetype` bezeichnet den Typ des Attributes; `string` beinhaltet den Namen des Attributes. In dem Beispiel aus Abb. 1.2 hätte `attributetype` den Wert `IDREF` und `string` hätte den Wert `refid`.

Sortenname:

dtd_saves dtd_saves

Signatur:

dtd_saves:: [dtd_save]

Beschreibung:

Eine Hilfsdatenstruktur zum Speichern der DTD. Die dabei auftretenden sequentiellen Anfragen nach DTD-Elementen werden aus dieser Datenstruktur bearbeitet. Mit der Funktion `getNextRule` wird angefragt, ob noch Einträge aus der DTD nicht ausgelesen worden sind. Wenn ja, gibt diese Funktion den Namen des logischen Elementes zurück, dessen Regelbeschreibung dann mit weiteren Funktionsaufrufen (s.u.) ausgelesen werden kann. Bezüglich unserer Datenstruktur bedeutet dies, daß aus `dtd_element` der `elementtype` das Ergebnis des Aufrufes von `getNextRule` ist und der zugehörige `dtd_parts`-Teil nach `dtd_saves` konvertiert wird. Dazu kommen noch die Attributdeklarationen, die zum Element `elementtype` gehören. Diese werden ebenfalls nach `dtd_saves` konvertiert und in dieser Hilfsdatenstruktur gespeichert.

Sortenname:

dtd_save dtd_save

Signatur:

dtd_save:: entry dtd_content string elementtype attributetype

Beschreibung:

Die in `dtd_parts` gehaltene Regelbeschreibung wird hierhin konvertiert. Jedes Element der Art `dtd_part` wird zu einem `dtd_save`. Dazu kommen noch die Attributdeklarationen. D.h. alle `attr_defs` werden ebenfalls nach `dtd_save` konvertiert.

Sortenname:

dtd_content dtd_content

Signatur:

dtd_content:: group | endgroup | element | attribute | data | nix

Beschreibung:

Dieser Datentyp ist ein Rückgabewert einer Funktion zum Speichern der DTD (s. `whatIsNext` in Kap. 1.3.3.2).

1.3.3.2 Funktionen zum Aufbauen der DTD-Datenstruktur

Es folgen die Funktionbeschreibungen zum Aufbauen der DTD-Datenstruktur. Der generelle Ablauf zum Aufbauen der DTD-Datenstruktur soll an einem Beispiel erläutert werden.
Eine Beispiel-DTD ist in Abbildung 1.2 auf Seite 40 gezeigt. Um diese DTD in unserer Struktur zu speichern sind folgende Aufrufe nötig:

beginDTD Initialisiert die DTD-Datenstruktur zum Aufnehmen einer neuen DTD.

beginRule textbook Beginnt die Regelbeschreibung zu dem logischen Element `textbook`.

beginGroup ',' Zu der zuletzt begonnenen Regelbeschreibung (in diesem Fall zu `textbook`) wird ein Operator eingefügt. Alle danach eingefügten Elemente werden diesem Operator zugeschrieben. Dazu wird die Integer-Hilfsvariable in der Datenstruktur `dtd` um eins erhöht.

beginGroup '?' Statt eines Elementes wird eine weitere Operatorgruppe begonnen. Die Integer-Hilfsvariable wird nochmals um eins erhöht (hat also den Wert 2).

insertElement front Das log. Element `front` wird in die Datenstruktur eingefügt. Dabei wird es der zuletzt begonnenen Operatorgruppe zugeordnet (also '?'). Dies gewährleistet der korrekte Wert der Integer-Hilfsvariable.

endGroup Die zuletzt begonnenen Operatorgruppe ('?') wird geschlossen (der Wert der Integer-Hilfsvariable um eins erniedrigt).

insertElement body Das log. Element `body` wird in die Datenstruktur eingefügt. Dabei wird es der Operatorgruppe ',' zugeordnet (die Integer-Hilfsvariable hat den Wert eins).

endGroup Die Operatorgruppe ',' wird beendet.

beginRule front Eine neue Elementregel wird in die Datenstruktur eingetragen.

insertElement EMPTY Diese Regelbeschreibung ist einfach.

beginAttrRule textbook Nach den Regelbeschreibungen folgen die Attributbeschreibungen der log. Elemente.

insertAttr REFID refid Das Attribut `refid` vom Typ `REFID` wird zu der zuletzt begonnenen Attributbeschreibung (also zu dem Element `textbook`) gefügt.

endDTD Das Aufbauen der DTD-Datenstruktur wird hiermit beendet.

Die Funktionen im einzelnen:

Funktionsname:

beginDTD beginDTD

Signatur:

beginDTD:: dtd

dtd: Initialisierte DTD-Datenstruktur

Beschreibung:

Der Rückgabewert dieser Funktion ist eine initialisierte (leere) DTD-Datenstruktur. Diese ist nun zum Aufnehmen einer neuen DTD vorbereitet.

Funktionsname:

endDTD endDTD

Signatur:

endDTD:: dtd -> dtd

dtd: Eingelesene DTD

dtd: Gespeicherte DTD

Beschreibung:

Beschließt das Einlesen einer DTD. Dabei wird die DTD nach Idempotenzen (wie z.B. (front*)*) durchsucht und diese aufgelöst.

Implementierung:

Nach dem letzten Aufruf zum Einfügen eines Elementes in die DTD ist unsere Datenstruktur fertig aufgebaut. Der Aufruf von `list_reform` zum Auflösen der Idempotenzen ist zur besonderen Vorsicht eingebaut worden.

Funktionsname:

clearDTD clearDTD

Signatur:

clearDTD:: dtd

dtd: Leere DTD-Datenstruktur

Beschreibung:

Löscht die DTD-Datenstruktur falls z.B. beim Einlesen der DTD ein Fehler aufgetreten ist.

Funktionsname:

beginRule beginRule

Signatur:

beginRule:: dtd -> elementtype -> dtd

dtd: Alte DTD-Datenstruktur

elementtype: Name des logischen Elementes, dessen Regelbeschreibung folgen wird.

dtd: Um oberen Regelnamen ergänzte DTD-Datenstruktur

Beschreibung:

Beginnt eine neue Regelbeschreibung. Zugleich wird die davor begonnene Regelbeschreibung abgeschlossen. Alle nach diesem Funktionsaufruf eingefügten Elemente oder Operatoren werden dieser Regel zugefügt.

Implementierung:

Eine neues Element `dtd_element` wird als erstes Element in die `dtd_list` eingefügt. Von dem Element `dtd_element` ist dabei `elementtype` auf den Regelnamen gesetzt und `dtd_parts` ist leer. Hiernach getätigte Funktionsaufrufe von Speichern von Gruppenoperatoren oder log. Elementen werden in `dtd_parts` gespeichert.

Funktionsname:

beginGroup beginGroup

Signatur:

beginGroup:: dtd -> string -> dtd

dtd: Alte DTD-Datenstruktur

string: Einzufügender DTD-Operator der Form '?', '*', '+', ',', '&', '|'

dtd: Ergänzte DTD-Datenstruktur

Beschreibung:

Zu der zuletzt begonnenen Regelbeschreibung wird ein Operator gefügt.

Vor- und Nachbedingungen:

Vor dem Aufruf dieser Funktion muß mindestens einmal ein `beginRule` getätigt worden sein.

Implementierung:

Das erste Element aus `dtd_list` wird abgetrennt. Dies ist gerade die zuletzt begonnene Regelbeschreibung (eine `dtd_parts`). Diese wird zusammen mit dem einzufügenden Operator in die Funktion `newGroup`[1] geschickt. `newGroup` nimmt die alte Regelbeschreibung `dtd_parts`, sucht rekursiv mit Zunahme der Integer-Hilfsvariablen die richtige Einfügeposition und fügt den Operator ein. Das Ergebnis von `newGroup` ist eine um den Operator ergänzte `dtd_parts`. Diese wird wieder als erstes Element in die `dtd_list` eingefügt. Danach muß noch die Integer-Hilfsvariable um eins erhöht werden, damit beim nächsten Einfügen von Elementen diese in die hier begonnene Operatorgruppe gesetzt werden.

Funktionsname:

insertElement insertElement

Signatur:

insertElement:: dtd -> elementtype -> dtd

dtd: Alte DTD-Datenstruktur

elementtype: Das einzufügende logische Element

[1] Näheres zu `newGroup` siehe Kap. 1.3.4.2

dtd: Ergänzte DTD-Datenstruktur

Beschreibung:

Zum zuletzt begonnenen Operator wird der im `elementtype` genannte Operand eingefügt.

Implementierung:

Die Funktion `insertElement` arbeitet analog zu `beginGroup`, nur daß statt `newGroup` die Hilfsfunktion `newElement` benutzt wird. Zudem wird die Integer-Hilfsvariable nicht erhöht.

Funktionsname:

endGroup endGroup

Signatur:

endGroup:: dtd -> dtd

dtd: Alte DTD-Datenstruktur

dtd: Neue DTD-Datenstruktur

Beschreibung:

Beschließt die zuletzt begonnene Regelbeschreibung.

Implementierung:

Die Integer-Hilfsvariable wird um eins erniedrigt.

Funktionsname:

beginAttrRule beginAttrRule

Signatur:

beginAttrRule:: dtd -> elementtype -> dtd

dtd: Alte DTD-Datenstruktur

elementtype: Name des logischen Elementes, dessen Attributbeschreibungen folgen werden.

dtd: Ergänzte DTD-Datenstruktur

Beschreibung:

Beginnt eine neue Liste von Attributen, die zum logischen Element `elementtype` gehören.

Implementierung:

Eine neue `attr_list_part`, in der nur der `elementtype` gesetzt ist (`attr_defs` ist hier noch leer), wird als erstes Element in die Liste `attr_list` eingefügt.

Funktionsname:

insertAttr insertAttr

Signatur:

insertAttr:: dtd -> attributetype -> string -> dtd

dtd: Alte DTD-Datenstruktur

attributetype: Typ des Attributes, das eingefügt wird

string: Name des Attributes

dtd: Ergänzte DTD-Datenstruktur

Beschreibung:

Fügt ein Attribut in die zuletzt begonnene Attributliste ein.

Vor- und Nachbedingungen:

Vorher ein Aufruf von `beginAttrRule`.

Implementierung:

Das erste Element von `attr_list` wird abgespalten, ein `attr_list_part`. Dies ist gerade die zuletzt begonnene Attributliste. Aus den übergebenen Parametern `attributetype` und `string` wird eine neue `attr_def` gebildet. Diese wird zusammen mit der `attr_list_part` an die Hilfsfunktion `newAttr` übergeben. Die Funktion `newAttr` filtert aus der `attr_list_part` die `attr_defs` heraus und fügt an diese die neue `attr_def` als erstes Element an. Das Ergebnis von `newAttr` ist eine ergänzte `attr_list_part`, die wieder als erstes Element an `attr_list` gehängt wird.

1.3.3.3 Funktionen zum Speichern der DTD-Datenstruktur

Es folgen die Funktionbeschreibungen zum Speichern der DTD-Datenstruktur. Der generelle Ablauf zum Speichern der DTD-Datenstruktur soll an einem Beispiel erläutert werden.
Eine Beispiel-DTD ist in Abbildung 1.2 auf Seite 40 gezeigt. Um diese DTD zu speichern sind folgende Aufrufe nötig:

beginDTDSave Initialisiert die DTD-Datenstruktur zum Speichern der DTD.

getNextRule Hiermit beginnt das Auslesen des ersten Elementes (in dem Beispiel `textbook`).

whatIsNext Diese Funktion gibt als Rückgabewert einen Konstruktor an, der anzeigt, welches Objekt einer DTD als nächstes ausgelesen werden kann. In diesem Fall würde er das Auslesen einer Operatorgruppe anzeigen.

getNextGroup Liest die durch `whatIsNext` angezeigte Operatorgruppe aus. Das Ergebnis ist nur der Operator (hier also ','). Die Elemente der Operatorgruppe müssen einzel durch eine Folge von `whatIsNext` und `getNextGroup` oder `getNextElement` ausgelesen werden.

whatIsNext In dem Beispiel folgt als nächstes wieder eine Operatorgruppe

getNextGroup Liest den Operator '?' aus.

whatIsNext Es folgt ein Element.

getNextElement Liest das Element `front` aus.

whatIsNext Die zuletzt begonnene Operatorgruppe ist vollständig ausgelesen. Die Funktion liefert ein `endgroup` als Ergebnis.

whatIsNext Was folgt nach der letzten Operatorgruppe? In dem Beispiel ein weiteres Element.

getNextElement Liest das Element `body` aus.

whatIsNext Auch die Operatorgruppe ',' ist vollständig ausgelesen. Die Funktion liefert ein `endgroup` als Ergebnis.

whatIsNext Jetzt folgt nach der Regelbeschreibung des Elementes `textbook` die Attributbeschreibung. Die Funktion liefert ein `attribute` als Ergebnis.

getNextAttr Liest das Attribut `refid` vom Typ `IDREF` aus.

whatIsNext Was folgt nach dem Attribut? In dem Beispiel nix mehr. Die Funktion liefert ein `nix` als Ergebnis. D.h. die DTD- und Attribut-Regeln zum Element `textbook` sind vollständig ausgelesen.

getNextRule Vielleicht gibt es noch mehr Regeln auslesen. In dem Beispiel würden die DTD- und Attribut-Regeln zum Element `front` folgen. Das Auslesen läuft analog zu oben, bis die Funktion `getNextRule` im Bool-Wert ein `false` angibt und damit das vollständige Auslesen der gesamten DTD anzeigt. Danach ist noch ein `endDTDSave` aufzurufen.

Die Funktionen im einzelnen:

Funktionsname:

beginDTDSave beginDTDSave

Signatur:

beginDTDSave:: dtd -> dtd

dtd: Alte DTD-Datenstruktur

dtd: Zum Speichern vorbereitete DTD-Datenstruktur

Beschreibung:

Diese Funktion muß vor allen anderen Funktionen zum Speichern der DTD aufgerufen werden. Sie initialisiert die Hilfsdatenstruktur.

Implementierung:

Die Hilfsdatenstruktur `dtd_saves` wird leer gesetzt. Zudem wird noch die Integer-Hilfsvariable aus dem Datentyp `dtd` auf 0 gesetzt. Die Integer-Hilfsvariable wird nach jedem Aufruf von `getNextRule` um eins erhöht. Durch diese Iteration werden sämtliche log. Elemente aus der DTD ausgelesen.

Funktionsname:

endDTDSave endDTDSave

Signatur:

endDTDSave:: dtd -> dtd

dtd: Alte DTD-Datenstruktur

dtd: DTD-Datenstruktur ohne Hilfsdatenstruktur

Beschreibung:

Die Hilfsdatenstruktur zum Speichern wird aufgelöst.

Funktionsname:

getNextRule getNextRule

Signatur:

getNextRule:: dtd -> (dtd, boolean, elementtype)

dtd: Alte DTD-Datenstruktur

dtd: Um die aktualisierte Hilfstruktur ergänzte DTD-Datenstruktur

boolean: Wahrheitswert, der noch auszulesende log. Elemente anzeigt. Nur wenn er `true` enthält, ist der folgende `elementtype` mit einem gültigen Wert gesetzt. Enthält er ein `false`, ist die DTD mit all ihren Elementen komplett ausgelesen worden.

elementtype: Name des logischen Elementes, dessen DTD- und Attributregeln im folgenden ausgelesen werden können.

Beschreibung:

Mit `getNextRule` beginnt das Auslesen der DTD-Definition eines log. Elementes. Der Name des log. Elementes ist dabei ein Rückgabewert dieser Funktion. Seine DTD- und Attributdefinitionen werden in die Hilfsdatenstruktur `dtd_saves` konvertiert, wo sie bereitliegen, um mit den unten beschriebenen Funktionen ausgelesen zu werden.

Implementierung:

Mit Hilfe der Integer-Hilfsvariablen wird überprüft, ob noch log. Elemente ausgelesen werden können. Dazu muß die Liste mit den DTD-Definitionen mehr Elemente enthalten, als die Integer-Hilfsvariable als Wert hat. Ist dies der Fall, kann mit dem ASpecT-Listenoperator '!' dasjenige `dtd_element` aus der Liste `dtd_list` herausgegriffen werden, dessen DTD- und Attributdefinitionen als nächsten auszulesen sind. Dazu werden Name und DTD-Definitionen dieses Elementes mit `splitDTDElement`[2] voneinander getrennt. Der Name wird ein unmittelbares Ergebnis des `getNextRule`-Aufrufes. Die Attribut-Definitionen dieses Elementes

[2] Näheres zu `splitDTDElement` siehe Kap. 1.3.4.2

müssen erst aus der eigenständigen Datenstruktur `attr_list` gesammelt werden. Dies geschieht mit der Funktion `getAllAttrs`. Danach werden Attribut- und DTD-Definitionen nach `dtd_saves` konvertiert. Dazu werden die Funktionen `mkDPSave` und `mkALSave` benutzt. Die konvertierten Definitionen werden in `dtd_save` gespeichert. Dort können sie dann mit folgenden Funktionen ausgelesen werden. Um iterativ alle DTD-Elemente auslesen zu können muß noch die Integer-Hilfsvariable um eins erhöht werden.

Funktionsname:

whatIsNext whatIsNext

Signatur:

whatIsNext:: dtd -> (dtd, dtd_content)

dtd: Alte DTD-Datenstruktur

dtd: Neue DTD-Datenstruktur

dtd_content: Ein Konstruktor, der angibt, was als nächstes aus der DTD-Hilfsdatenstruktur zum Speichern ausgelesen werden dann.

Beschreibung:

Diese Funktion ist alternierend mit den Funktionen zum Auslesen der DTD-Hilfsdatenstruktur zu benutzen. Sie gibt vor, welche der folgenden Funktionen zum Auslesen anzuwenden ist.

Implementierung:

Beim Aufruf von `getNextRule` werden die DTD- und Attributregeln des auszulesenden Elementes nach `dtd_saves` konvertiert. Der Datentyp `dtd_saves` ist eine Liste. Die Funktion `whatIsNext` nimmt das erste Element dieser Liste (ein `dtd_save`) und sieht in dem darin enthaltenen Datentyp `dtd_content` nach, um was es sich bei diesem Element handelt. Dies ist zugleich ein Rückgabewert dieser Funktion. Handelt es sich zudem um ein `endgroup`, wird das abgespaltene Element aus der Liste `dtd_saves` nicht wieder als erstes Element an die Liste gehängt, da es keine Funktion gibt, um dieses `endgroup` auszulesen. Bei allen anderen Elementtypen (`group, element, attribute, data, nix`) wird das abgespaltene Listenelement wieder als erstes Element an `dtd_saves` gehängt, damit mit einem der folgenden Funktionen dieses Element ausgelesen werden kann.

Funktionsname:

getNextGroup getNextGroup

Signatur:

getNextGroup:: dtd -> (dtd, string)

dtd: Alte DTD-Datenstruktur

dtd: Neue DTD-Datenstruktur

string: Einer der Gruppenoperatoren '?', '*', '+', ',', '&', '|'

Beschreibung:

Zeigt ein vorheriger Aufruf von `whatIsNext`, daß eine Operatorgruppe auszulesen ist, so kann dies mit dieser Funktion begonnen werden. Dabei liest der Aufruf dieser Funktion nur den Gruppenoperator aus. Die Operanden, die zu diesem Operator gehören, müssen mit einer alternierenden Folge von `whatIsNext` und dieser und den folgenden Funktionen ausgelesen werden.

Abhängigkeiten:

Ein vorheriger Aufruf von `whatIsNext` sollte das Auslesen eines Gruppenoperators anzeigen. Wird dies nicht gemacht, und ist das nächste auszulesende Objekt kein Gruppenoperator, so gibt der Aufruf von `getNextGroup` einen Leerstring im Gruppenoperator als Ergebnis zurück.

Implementierung:

Diese Funktion spaltet das erste Element aus der DTD-Hilfsdatenstruktur ab. Ist dieses Element ein Gruppenoperator (zu sehen am `dtd_content`-Feld im abgespalteten `dtd_save`-Element) wird der Gruppenoperator aus dem `dtd_save`-Element ausgegeben. Die `dtd_saves`-Liste verkürzt sich um dieses Element. Handelt es sich bei dem abgespaltenem `dtd_save`-Element nicht um einen Gruppenoperator, wird dieses abgespaltene Element wieder als erstes Element in die DTD-Hilfsdatenstruktur eingefügt, um noch korrekt ausgelesen werden zu können.

Funktionsname:

getNextElement getNextElement

Signatur:

getNextElement:: dtd -> (dtd, elementtype)

dtd: Alte DTD-Datenstruktur

dtd: Neue DTD-Datenstruktur

elementtype: Name des logischen Elementes, das ausgelesen wird

Beschreibung:

Liest ein log. Element aus der DTD-Hilfsdatenstruktur zum Speichern aus.

Abhängigkeiten:

Ein vorheriger Aufruf von `whatIsNext` sollte das Auslesen eines log. Elementes anzeigen. Wird dies nicht gemacht, und ist das nächste auszulesende Objekt kein log. Elemente, so gibt der Aufruf von `getNextElement` einen `mtElementtype` als Ergebnis zurück.

Implementierung:

Diese Funktion spaltet das erste Element aus der DTD-Hilfsdatenstruktur ab. Ist dieses Element ein log. Element (zu sehen am `dtd_content`-Feld im abgespalteten `dtd_save`-Element) wird das log. Element aus dem `dtd_save`-Element ausgegeben. Die `dtd_saves`-Liste verkürzt sich um dieses Element. Handelt es sich bei dem abgespaltenem `dtd_save`-Element nicht um ein log. Element, wird dieses abgespaltene Element wieder als erstes Element in die DTD-Hilfsdatenstruktur eingefügt, um noch korrekt ausgelesen werden zu können.

Funktionsname:

getNextAttr getNextAttr

Signatur:

getNextAttr:: dtd -> (dtd, attributetype, string)

dtd: Alte DTD-Datenstruktur

dtd: Neue DTD-Datenstruktur

attributetype: Typ des Attributes, das ausgelesen wird

string: Name des Attributes, das ausgelesen wird

Beschreibung:

Liest ein Attribut aus der DTD-Hilfsdatenstruktur zum Speichern aus.

Abhängigkeiten:

Ein vorheriger Aufruf von `whatIsNext` sollte das Auslesen eines Attributes anzeigen. Wird dies nicht gemacht, und ist das nächste auszulesende Objekt kein Attribut, so gibt der Aufruf von `getNextElement` einen `none` für den Typ und einen Leerstring für den Namen als Ergebnis zurück.

Implementierung:

Diese Funktion spaltet das erste Element aus der DTD-Hilfsdatenstruktur ab. Ist dieses Element ein Attribut (zu sehen am `dtd_content`-Feld im abgespalteten `dtd_save`-Element) werden der Typ und der Name des Attributes aus dem `dtd_save`-Element ausgegeben. Die `dtd_saves`-Liste verkürzt sich um dieses Element. Handelt es sich bei dem abgespaltenem `dtd_save`-Element nicht um ein Attribut, wird dieses abgespaltene Element wieder als erstes Element in die DTD-Hilfsdatenstruktur eingefügt, um noch korrekt ausgelesen werden zu können.

1.3.3.4 Funktionen, die auf der DTD-Datenstruktur arbeiten

Es folgen die Beschreibungen der Funktionen, die auf der DTD-Datenstruktur arbeiten.

Funktionsname:

getRootElement getRootElement

Signatur:

getRootElement:: dtd -> elementtype

dtd: Alte DTD-Datenstruktur

elementtype: Name des DTD-Wurzelelementes. Als Wurzelelement verstehen wir das log. Element in der DTD, das als einziges Element sich zwar aus anderen log. Elementen aus der DTD definiert, selbst aber nicht in der Regelbeschreibung eines anderen Elementes vorkommt.

Beschreibung:

Diese Funktion bestimmt das Wurzelelement einer DTD.

Abhängigkeiten:

Sollte mehr als ein Wurzelelement in der DTD vorkommen, ist nicht vorhersagbar, welches dieser Elemente als Ergebnis dieser Funktion ausgegeben wird.

Implementierung:

Diese Funktion bildet zwei Listen aus der DTD. Die erste Liste beinhaltet die Namen der logischen Elemente, deren Aufbau durch Regelbeschreibungen definiert sind. Diese log. Elemente stehen in einer DTD auf der linken Seite. In der Beispiel-DTD in Abbildung 1.2 auf Seite 40 wären dies die Elemente `textbook` und `front`. Diese Elemente sind mögliche Kandidaten für das Wurzelelement. In der zweiten Liste stehen die Namen der Elemente, die zur Definition des Aufbaus eines logischen Elementes verwendet werden. D.h sie werden in der Regelbeschreibung auf der rechten Seite einer DTD genannt. In der Beispiel-DTD wären dies `front` und `body`. Diese Elemente können keine Wurzelelemente darstellen. Mit Hilfe von `allDTDEls`[3] werden diese beiden Listen generiert. Nachdem in der zweiten Liste alle doppelten Elemente gelöscht wurden (`mkset`), wird die zweite Liste von der ersten Liste abgezogen. D.h. wir ziehen von allen möglichen Kandidaten für das Wurzelelement diejenigen ab, die zu einer Regelbeschreibung verwendet werden. Übrig bleibt eine Liste, die normalerweise genau ein Element, das Wurzelelement, beinhaltet. Ist diese Liste leer, ist die DTD fehlerhaft, und diese Funktion gibt einen `mtElementtype` zurück. Hat diese Liste mehr als ein Element, ist die DTD ebenfalls fehlerhaft. Allerdings wird das erste Element aus der Liste als Wurzelelement zurückgegeben.

Funktionsname:

insertables | insertables

Signatur:

insertables:: dtd -> elementtype -> elementtypelist -> elementtypelist -> elTypeList

dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes der DTD

elementtypelist: Liste von finalen Elementen, die vor der Einfügestelle steht

[3] Näheres zu `allDTDEls` siehe Kap. 1.3.4.2

elementtypelist: Liste von finalen Elementen, die hinter der Einfügestelle steht

elTypeList: Ergebnisliste

Beschreibung:

Die Liste der finalen Elemente, die vor der Einfügestelle steht, und die Liste der finalen Elemente, die hinter der Einfügestelle steht, können zusammengefügt werden, und die entstehende Liste ist nach der zum Bezeichener gehörigen Regelbeschreibung korrekt aufgebaut. Die Funktion ermittelt nun eine Liste von `elementtypelist`'s , wobei jede `elementtypelist` zwischen den beiden Listen eingefügt werden kann, ohne daß der korrekte syntaktische Aufbau zerstöret wird.

Implementierung:

Die Regelbeschreibung des Bezeichners muß zunächst aus der `dtd` herausgesucht werden. Bei sämtlichen `dtd_elementes` wird überprüft, ob der Name und der übergebene Bezeichner identisch sind. Ist dieses der Fall, wird durch die Funktion `moveto` eine Liste von `dtd_parts`'s gebildet. Jeder `dtd_parts` könnte von der übergebenen Regelbeschreibung übrigbleiben, nachdem die Liste von finalen Elementen erkannt wurde, die vor der Einfügestelle auftritt. In der Funktion `insertable_at` wird anschließend die weitere Verarbeitung durchgeführt. Falls bei der Suche kein `dtd_element` mit dem Bezeichner des logischen Elementes als Namen gefunden wurde, erzeugt die Funktion als Ergebnis eine leere Liste.

Funktionsname:

deleteable | deleteable

Signatur:

deleteable:: dtd -> elementtype -> elementtypelist -> elementtypelist -> boolean.

dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes der DTD

elementtypelist: Liste von finalen Elementen, die vor den gelöschten finalen Elementen steht

elementtypelist: Liste von finalen Elementen, die hinter den gelöschten finalen Elementen steht

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion überprüft, ob eine Liste von finalen Elementen noch einen korrekten Aufbau hat, nachdem einige finale Elemente aus der Liste gelöscht wurden.

Implementierung:

Zunächst wird die zum Bezeichner gehörige Regelbeschreibung aus der `dtd` ermittelt. Deshalb wird bei jedem `dtd_element` der Name mit dem übergebenen Bezeichner verglichen. Sind Name und Bezeichner gleich, wird festgestellt, ob die Liste von finalen Elementen ohne die gelöschten Elemente auch nach der gefundenen Regelbeschreibung korrekt aufgebaut ist. Zu diesem Zwecke wird zuerst die Liste von finalen Elementen, die vor den gelöschten finalen Elementen steht, mit der Liste von finalen Elementen, die hinter den gelöschten finalen Elementen steht, zusammengefügt. Mit dem Ergebnis und der Regelbeschreibung wird anschließend die Funktion `seq_parse` aufgerufen.

Funktionsname:

restruction restruction

Signatur:

restruction:: dtd -> elementtype -> elementtypelist -> elementtypelist -> elementtypelist -> elementtypelist

dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes der DTD

elementtypelist: Liste von finalen Elementen, die vor den markierten finalen Elementen steht

elementtypelist: Liste von markierten finalen Elementen

elementtypelist: Liste von finalen Elementen, die hinter den markierten finalen Elementen steht

elementtypelist: Ergebnisliste

Beschreibung:

Die drei übergebenen `elementtypelist`'s sind, wenn sie konkateniert werden, nach der zum Bezeichner gehörigen Regelbeschreibung korrekt aufgebaut. Die Funktion ermittelt nun eine Liste von finalen Elementen. Jedes dieser Elemente kann die Liste von markierten finalen Elementen ersetzen, ohne daß der korrekte Aufbau nach der Regelbeschreibung verletzt wird. Die Anwendung dieser Funktion ist vor allem dann sinnvoll, wenn nach einem Löschvorgang zwei gleiche Elemente nebeneinander stehen und entschieden werden soll, ob sie zu einem Element zusammengefaßt werden sollen.

Implementierung:

Aus der `dtd` muß zunächst die zum Bezeichner gehörige Regelbeschreibung herausgesucht werden. Zu diesem Zwecke wird bei jedem `dtd_element` der Name mit dem übergebenen Bezeichner verglichen. Wurde die gesuchte Regelbeschreibung gefunden, wird wie in der Funktion `pasteable` mit Hilfe der Funktion `moveto` eine Liste von `dtd_parts`'s gebildet. Jede Regelbeschreibung der ermittelten Liste kann entstehen, nachdem von der gefundenen Regelbeschreibung die Liste von finalen Elementen erkannt wurde, die vor den markierten Elementen auftritt. Danach wird die Funktion `restruct_at` aufgerufen, die die Verarbeitung fortführt. Wurde kein `dtd_element` mit dem übergebenen Bezeichner gefunden, wird eine leere Ergebnisliste zurückgeliefert.

Funktionsname:

pasteable pasteable

Signatur:

pasteable:: dtd -> elementtype -> elementtypelist -> elementtypelist -> elementtypelist -> elTypeList dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes der DTD

elementtypelist: Liste von finalen Elementen, die vor den markierten finalen Elementen steht

elementtypelist: Liste von markierten finalen Elementen

elementtypelist: Liste von finalen Elementen, die hinter den markierten finalen Elementen steht

elTypeList: Ergebnisliste

Beschreibung:

Werden die drei übergebenen `elementtypelist`'s konkateniert, entspricht ihr Aufbau der Regelbeschreibung des Bezeichners. Die Funktion ermittelt nun eine Liste von `elementtypelist`'s durch die die Liste von markierten finalen Elementen ersetzt werden kann, ohne den korrekten syntaktischen Aufbau zu zerstören.

Implementierung:

Zunächst wird aus der Datenstruktur `dtd` die zugehörige Regelbeschreibung herausgesucht. Die `dtd_list` der `dtd` wird sequentiell durchsucht. Sind Name des `dtd_elementes` und der übergebene Bezeichner identisch, wird zuerst durch die Funktion `moveto` eine Liste von `dtd_parts`'s gebildet, die von der gefundenen Regelbeschreibung übrigbleiben könnten, nachdem die Liste von finalen Elementen erkannt wurde, die vor den markierten Elementen auftritt. Die weiteren Verarbeitungsschritte werden in der Funktion `pasteable_at` durchgeführt. Falls kein `dtd_element` mit dem Bezeichner des logischen Elementes als Namen gefunden wurde, liefert die Funktion automatisch eine leere Ergebnisliste zurück.

Funktionsname:

is_final is_final

Signatur:

is_final:: dtd -> elementtype -> boolean.

dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion entscheidet, ob der übergebene Bezeichner, der ein logisches Element , final ist. Final bedeutet, daß das logische Element nicht mehr aus weiteren logischen Elementen bestehen kann, sondern z.B. `pcdata` ist.

Implementierung:

Zu jedem nichtfinalen logischen Element muß es ein `dtd_element` geben, in dessen `dtd_parts` beschrieben wird, aus welchen Bestandteilen das logische Element aufgebaut ist. Deshalb genügt es, die `dtd_list` nach einem Element zu durchsuchen, dessen Name gleich dem übergebenen Bezeichner ist. Leider enthält die Funktion noch einen kleinen Denkfehler, so daß falsche Ergebnisse auftreten können.

Funktionsname:

all_elements all_elements

Signatur:

all_elements:: dtd -> elementtype -> elTypeList

dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes

Beschreibung:

Die Funktion erzeugt eine Liste, die sämtliche Listen von finalen Elementen enthält, die nach einer Regelbeschreibung korrekt aufgebaut sind.

Implementierung:

Aus der `dtd` wird zunächst die zum Bezeichner gehörige Regelbeschreibung gesucht. Dazu wird der Bezeichner jedes Elementes der `dtd_list` mit dem übergebenen Bezeichner verglichen. Wurde das richtige `dtd_element` gefunden, wird aus der Regelbeschreibung die Ergebnisliste gebildet. Die folgenden Schritte werden dazu ausgeführt.

1. Aus den finalen Elementen, die in der Regelbeschreibung vorkommen, wird durch die Funktion `allEls` eine geordnete Liste von `elementtypelist`'s gebildet.
2. Die Funktion `building_elTypeList` bildet aus der geordneten Liste das „falsche Kartesische Produkt". Jede `elementtypelist` der Ergebnisliste ist mit gewisser Wahrscheinlichkeit nach der Regelbeschreibung korrekt aufgebaut.
3. Die Funktion `mkset` beseitigt zunächst die doppelten `elementtypelist`'s. Im letzten Teil werden durch die Funktion `insert_parse` die Listen von finalen Elementen ausgewählt, die der Regelbeschreibung entsprechen.

Funktionsname:

parse parse

Signatur:

parse:: dtd -> elementtype -> elementtypelist -> boolean

dtd: DTD-Datenstruktur

elementtype: Bezeichner eines logischen Elementes der DTD elementtypelist: Liste von finalen Elementen

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion `parse` überprüft, ob die übergebene `elementtypelist` einen korrekten syntaktischen Aufbau hat. Zum Vergleich wird die zum logischen Element, dessen Name mit dem 2. Parameter übergeben wird, gehörige Regelbeschreibung herangezogen.

Implementierung:

Zunächst muß aus der Datenstruktur `dtd` die zugehörige Regelbeschreibung herausgesucht werden. Dazu wird die `dtd_list` der `dtd` sequentiell durchsucht. Ist der Name des `dtd_elementes` gleich dem Bezeichner des logischen Elementes, wird mit dem Aufruf der Funktion `seq_parse` zusammen mit der Regelbeschreibung und der übergebenen `elementtypelist` überprüft, ob der Aufbau korrekt ist. Wurde kein `dtd_element` mit dem Bezeichner des logischen Elementes als Namen gefunden, liefert die Funktion den Wert `false` zurück.

Funktionsname:

count_elements — count_elements

Signatur:

count_elements:: dtd -> integer

dtd: DTD-Datenstruktur

integer: Anzahl der Elemente

Beschreibung:

Die Funktion emittelt, wieviele logische Elemente sich in der `dtd` befinden.

Implementierung:

Die Elemente der `dtd_list` werden gezählt.

1.3.4 Private Schnittstellen

1.3.4.1 Sorten

Die meisten der `LOCAL`-deklarierten Datentypen sind bereits in Kap. 1.3.3.1 dokumentiert worden. Dies geschah dort, um die danach dokumentierten Funktionen, die auf diesen Datentypen arbeiten, besser erläutern zu können. An dieser Stelle steht deshalb nur eine Referenz auf dieses Kapitel. Im folgenden werden die Sorten dokumentiert, die als `LOCAL`-deklariert und noch nicht im oben genannten Kapitel aufgeführt sind.

Sortenname:

poss_dtd_parts — poss_dtd_parts

. **Signatur:**

poss_dtd_parts:: [dtd_parts]

Beschreibung:

Der Datentyp `poss_dtd_parts` besteht ebenfalls aus einer Liste von `dtd_parts`'s, die durch die Bearbeitung eines `dtd_parts`'s durch eine entsprechende Funktion erzeugt wurde. Da die Semantik eine etwas andere ist als bei der Sorte `perm_dtd_parts` wurde eine neue Sorte definiert.

Sortenname:

perm_dtd_parts perm_dtd_parts

Signatur:

perm_dtd_parts:: [dtd_parts]

Beschreibung:

Der Datentyp besteht aus einer Liste von `dtd_parts`'s. Jedes Element dieses Datentypes soll alle Permutationen eines `dtd_parts`'s enthalten.

1.3.4.2 Funktionen

Im folgenden werden die im `LOCAL`-Teil des ASpecT-Codes deklarierten Funktionen beschrieben. Dabei handelt es sich um Hilfsfunktionen, die aus den in obigen Kapiteln genannten Funktionen aufgerufen wird. Zum besseren Verständnis der hier aufgeführten Funktionen ist daher der Verweis auf obiges Kapitel angebracht.

Funktionsname:

newGroup newGroup

Signatur:

newGroup:: integer -> dtd_parts -> string -> dtd_parts

integer: Wert der Integer-Hilfsvariablen zum Auffinden der richtigen Einfügeposition des einzusetzenden Gruppenoperators

dtd_parts: Alte DTD-Regelbeschreibung, die um den Gruppenoperator ergänzt werden soll

string: Der einzufügende Gruppenoperator

dtd_parts: Neue DTD-Regelbeschreibung, die um den Gruppenoperator ergänzt ist

Beschreibung:

Dies ist eine Hilfsfunktion zu `beginGroup`[4]. Sie wird benutzt, um einen neuen Gruppenoperator in die DTD-Datenstruktur einzutragen.

[4] Näheres zu `beginGroup` siehe Kap. 1.3.3.2

Implementierung:

Diese Funktion hat als ersten Parameter ein `dtd_parts`, das um den Gruppenoperator ergänzt werden soll. Nach jedem Aufruf von `beginGroup` wird die Integer-Hilfsvariable um eins erhöht. Ist beim Aufruf von `newGroup` die Integer-Hilfsvariable vom Wert 0, kann direkt hinter die übergebenen `dtd_parts` der neue Operator eingefügt werden.

Beispiel: die übergebene `dtd_parts` habe die Form [`ele front, pos`[`ele body`]]. Beim Einfügen des Gruppenoperators '`*`' und der Integer-Hilfsvariablen vom Wert 0 würde die Funktion `newGroup` folgendes Ergebnis liefern: [`ele front, pos`[`ele body`], `mal`[]]. Die Integer-Hilfsvariable hätte danach den Wert 1. Alle jetzt eingefügten Operatoren oder Elemente würden dem Operator `mal` zugeteilt werden.

Hätte dagegen die Integer-Hilfsvariable den Wert 1, würde die Funktion `newGroup` dieses Ergebnis liefern: [`ele front, pos`[`ele body, mal`[]]]. Dieses zweite Ergebnis würde dabei mit Hilfe der Funktion `newGroupDown` zustandekommen.

Funktionsname:

newGroupDown newGroupDown

Signatur:

newGroupDown:: integer -> dtd_parts -> string -> dtd_parts

integer: Wert der Integer-Hilfsvariablen zum Auffinden der richtigen Einfügeposition des einzusetzenden Gruppenoperators

dtd_parts: Alte DTD-Regelbeschreibung, die um den Gruppenoperator ergänzt werden soll

string: Der einzufügende Gruppenoperator

dtd_parts: Neue DTD-Regelbeschreibung, die um den Gruppenoperator ergänzt ist

Beschreibung:

Eine Hilfsfunktion zu `newGroup`. Diese Funktion wird benutzt, wenn die Einfügeposition des nächsten Operators nicht auf der ersten Ebene der an `newGroup` übergebenen `dtd_parts` liegt (die Integer-Hilfsvariable hat einen Wert größer 0, s. Bsp. in der Beschreibung von `newGroup`).

Implementierung:

Wird die Funktion `newGroup` mit einem Wert für die Integer-Hilfsvariablen größer 0 aufgerufen, so beinhaltet die übergebene `dtd_parts` nicht die Einfügeebene. Diese Funktion sucht nun die `dtd_parts`, die die Einfügeebene beinhaltet. Dabei ist die zu suchende `dtd_parts` das ganz rechte Element der übergebenen `dtd_parts`.

Ein Beispiel:
die übergebene `dtd_parts` habe die Form [`ele front, pos`[`ele body, mal`[`ele end`]]]. Die Integer-Hilfsvariable habe den Wert 2. Der einzufügende Operator sei die Sequenz. Der Aufruf von `newGroupDown` aus `newGroup` würde lauten:
`newGroupDown 2 [ele front, pos[ele body, mal[ele end]]] ','`.

Die Funktion `newGroupDown` sucht rekursiv den ganz rechten Gruppenoperator, den es hinabsteigen kann (im Source-Code von `newGroupDown` die letzte Zeile; der Catch-All-Fall; in diesem Fall `pos[...]`). Der rekursive Aufruf würde also lauten (das Element `front` wurde abgespalten):
`newGroupDown 2 [pos[ele body, mal[ele end]]] ','`. Diesmal paßt einer der ersten Funktionszeilen von `newGroupDown` und der weitere Aufruf würde lauten:
`newGroup 1 [ele body, mal[ele end]] ','`.

Die Funktionen `newGroup` und `newGroupDown` rufen sich verschränkt rekursiv nach obigen Schema auf, bis der Aufruf `newGroup 0 [ele end] ','` das gewünschte Ergebnis für die `dtd_parts` liefert (`[ele end, seq[]]`). Danach wird der durch die Rekursion aufgebaute Stack wieder abgebaut. Dadurch wird die durch das Suchen der Einfügeposition aufgebrochene Ursprungs-`dtd_parts` wieder rekonstruiert. Nur mit dem Unterschied, das in der ganz rechten Teil-`dtd_parts` der neue Gruppenoperator eingefügt ist. D.h. das Ergebnis aus dem Zusammenspiel der Funktionen `newGroup` und `newGroupDown` ist bei obigem Beispiel:
`[ele front, pos[ele body, mal[ele end, seq[]]]]`.

Funktionsname:

newElement newElement

Signatur:

newElement:: integer -> dtd_parts -> elementtype -> dtd_parts

integer: Wert der Integer-Hilfsvariablen zum Auffinden der richtigen Einfügeposition des einzusetzenden logischen Elementes

dtd_parts: Alte DTD-Regelbeschreibung, die um das log. Element ergänzt werden soll

elementtype: Das einzufügende log. Element

dtd_parts: Neue DTD-Regelbeschreibung, die um das log. Element ergänzt ist

Beschreibung:

Dies ist eine Hilfsfunktion zu `insertElement`[5]. Sie wird benutzt, um einen neues logischen Element in die DTD-Datenstruktur einzutragen.

Implementierung:

Die Implementation ist analog zu der Implementation der Funktion `newGroup`. Auch das Zusammenspiel der Funktionen `newElement` und `newEleDown` ist analog zu dem der Funktionen `newGroup` und `newGroupDown`. Der einzige Unterschied liegt in dem Parameter `elementtype` statt `string`.

Funktionsname:

newEleDown newEleDown

[5] Näheres zu `insertElement` siehe Kap. 1.3.3.2

Signatur:

newEleDown:: integer -> dtd_parts -> elementtype -> dtd_parts

integer: Wert der Integer-Hilfsvariablen zum Auffinden der richtigen Einfügeposition des einzusetzenden logischen Elementes

dtd_parts: Alte DTD-Regelbeschreibung, die um das log. Element ergänzt werden soll

elementtype: Das einzufügende log. Element

dtd_parts: Neue DTD-Regelbeschreibung, die um das log. Element ergänzt ist

Beschreibung:

Eine Hilfsfunktion zu `newElement`. Diese Funktion wird benutzt, wenn die Einfügeposition des nächsten log. Elementes nicht auf der ersten Ebene der an `newElement` übergebenen `dtd_parts` liegt (die Integer-Hilfsvariable hat einen Wert größer 0).

Implementierung:

Erfolgte analog zur Implementation von `newGroupDown`.

Funktionsname:

newAttr newAttr

Signatur:

newAttr:: attr_def -> attr_list_part -> attr_list_part

attr_def: Die einzufügende neue Attributdefinition

attr_list_part: Alte Attributregeln

attr_list_part: Um die neue Attributdefinition ergänzte Attributregeln

Beschreibung:

Dies ist eine Hilfsfunktion zu `insertAttr`[6]. Sie wird benutzt, um einen neues Attribut in die DTD-Datenstruktur einzutragen.

Implementierung:

Aus der übergebenen `attr_list_part` werden die Attributregeln (in `attr_defs` gehalten) von dem Namen des log. Elementes, zu dem diese Attributregeln gehören, getrennt. Die übergebene neue Attributdefinition wird als erstes Element an die alten `attr_defs` gehängt. Aus dem Namen und den ergänzten Attributregeln wird wieder eine `attr_list_part` erzeugt.

Funktionsname:

[6] Näheres zu `insertAttr` siehe Kap. 1.3.3.2

splitDTDElement splitDTDElement

Signatur:

splitDTDElement:: dtd_element -> (elementtype, dtd_parts)

dtd_element: Eine Regelbeschreibung

elementtype: Der Name des log. Elementes aus der Regelbeschreibung

dtd_parts: Die Aufbaubeschreibungen zu dem log. Element

Beschreibung:

Diese Funktion trennt den Namen eines logischen Elementes von seinen DTD-Definitionen.

Implementierung:

Dies ist durch einfaches Pattern-Matching realisiert.

Funktionsname:

getAllAttrs getAllAttrs

Signatur:

getAllAttrs:: elementtype -> attr_list_part -> attr_defs -> attr_defs

elementtype: Name des log. Elementes, dessen Attributregeln gesammelt werden sollen

attr_list_part: Eine Attributregel

attr_defs: Die bisher gesammelten Attributregeln

attr_defs: Die bisher gesammelten Attributregeln plus den aus der `attr_list_part` gewonnenen Attributregeln

Beschreibung:

Diese aus `getNextRule` aufgerufene Hilfsfunktion sammelt sämtliche zum übergebenen log. Element vorhandenen Attributdefiniton aus der DTD.

Implementierung:

Diese Hilfsfunktion wird in der Funktion `getNextRule` mit dem ASpecT-Listenoperator `foldr` über alle Attributregeln geschickt. Dabei wird überprüft, ob die übergebene `attr_list_part` Attributdefiniton zu dem gesuchten Element enthält. Dazu werden die Namen des übergebenen log. Elementes mit dem des in der `attr_list_part` genannten Elementes verglichen. Sind sie identisch, werden die Attributregeln aus der `attr_list_part` an die Ergebnis-`attr_defs` gehängt.

Funktionsname:

mkDPSave mkDPSave

Signatur:

mkDPSave:: dtd_parts -> dtd_saves

dtd_parts: Zu konvertierende DTD-Regeln

dtd_saves: Konvertierte DTD-Regeln

Beschreibung:

Diese Hilfsfunktion, die aus `getNextRule` aufgerufen wird, konvertiert die übergebenen, zu einem log. Element gehörende DTD-Regeln in die DTD-Hilfsdatenstruktur zum Speichern.

Implementierung:

Diese rekursive Funktion spaltet das erste Element aus der übergebenen `dtd_parts` ab und konvertiert es entsprechend seines Inhalts in ein `dtd_save`. Die einzeln konvertierten `dtd_save` werden in einer Liste, der `dtd_saves`, gesammelt. Die Liste `dtd_saves` ist nicht geschachtelt. Dadurch ist das Auslesen mit den in Kap. 1.3.3.3 beschriebenen Funktionen sehr einfach.

Funktionsname:

mkALSave mkALSave

Signatur:

mkALSave:: dtd_saves -> attr_def -> dtd_saves

dtd_saves: Bisher konvertierte Attributdefinitionen sind hierin gespeichert

attr_def: Zu konvertierende Attributdefinition

dtd_saves: Bisher konvertierte Attributdefinitionen plus der neu konvertierten Attributdefinition

Beschreibung:

Diese Hilfsfunktion, die ebenfalls aus `getNextRule` aufgerufen wird, konvertiert die übergebenen Attributdefinition in die DTD-Hilfsdatenstruktur zum Speichern.

Implementierung:

Diese Hilfsfunktion wird in der Funktion `getNextRule` mit dem ASpecT-Listenoperator `foldl` über alle mit Hilfe der Funktion `getAllAttrs` gesammelten Attributregeln geschickt. Jede einzelne Attributregel wird dabei in seine Bestandteile Attributname und Attributtype gespalten. Diese werden dann zu einem `dtd_save` wieder zusammengefaßt.

Funktionsname:

allDTDEls allDTDEls

Signatur:

allDTDEls:: dtd_list -> (elementtypelist, elementtypelist)

dtd_list: Alle in der DTD genannten log. Elemente mit ihren Aufbauregeln

elementtypelist: Hierin sind die Namen der log. Elemente gesammelt, deren Aufbau in der DTD definiert ist (die auf der linken Seite in der DTD stehen)

elementtypelist: Hierin sind die Namen der log. Elemente gesammelt, die in der Aufbaubeschreibung eines anderen log. Elementes verwendet wurden (die auf der rechten Seite einer DTD aufgeführt sind)

Beschreibung:

Diese Hilfsfunktion, die aus `getRootElement` aufgerufen wird, generiert zwei Listen von Namen log. Elemente, die in der DTD genannt sind. Die erste Liste beinhaltet die Namen der log. Elemente, deren Aufbau in der DTD vorgegeben ist. Es sind dies die Namen der log. Elemente, die in einer DTD auf der linken Seite einer Regelbeschreibung stehen. In der zweiten Liste sind die Namen der log. Elemente gesammelt, die in einer Regelbeschreibung eines anderen log. Elementes verwendet werden (also auf der rechten Seite einer DTD genannt werden). Oder anders gesagt: die erste Liste beinhaltet *alle* in der DTD genannten log. Elemente. In der zweiten Liste sind nur die aufgeführt, die zur Definition eines anderen Elementes verwendet werden.

Implementierung:

Diese rekursive Funktion spaltet von der übergebenen `dtd_list` das erste Element ab. Dies wird in seine Bestandteile Elementname und den dazugehörigen DTD-Regeln unterteilt. Der Elementname wird in der ersten Ergebnisliste gesammelt. Die in den DTD-Regeln verwendeten log. Elemente werden mit Hilfe der Funktion `allRuleEls` bestimmt und in der zweiten Ergebnisliste gesammelt.

Funktionsname:

allRuleEls allRuleEls

Signatur:

allRuleEls:: dtd_parts -> elementtypelist

dtd_parts: Die DTD-Regeln zu einem log. Element

elementtypelist: Die in den DTD-Regeln verwendeten log. Elemente

Beschreibung:

Diese Hilfsfunktion, die aus `allDTDEls` aufgerufen wird, generiert eine Listen von Namen log. Elemente, die in den übergebenen DTD-Regeln verwendet werden.

Implementierung:

Eine rekursive Funktion, die alle Gruppenoperatoren hinabsteigt, um die darin enthaltenen log. Elementnamen in der Ergebnisliste sammeln zu können.

Funktionsname:

seq_parse seq_parse

Signatur:

seq_parse:: dtd_parts -> elementtypelist -> boolean

dtd_parts: Regelbeschreibung für ein logisches Element

elementtypelist: Liste von finalen Elementen

boolean: Ergebnis des Vergleiches

Beschreibung:

Die Funktion `seq_parse` überprüft, ob die Liste von finalen Elementen nach den Regelbeschreibungen korrekt aufgebaut sind.

Implementierung:

Der Vergleich zwischen der Liste und der Regelbeschreibung, die die Form eines regulären Ausdrucks besitzt, wird in folgender Weise durchgeführt:

1. Durch Anwendung der Rechenregeln für reguläre Ausdrücke wird versucht, die Regelbeschreibung so umzuformen, daß diese mit einem finalen Element beginnt. Der Operator, der auf das erste Listenelement der Regelbeschreibung angewendet wird, bestimmt die Umformung.

   Opt R1 Da die Option das ein- oder nullmalige Auftreten von $R1$ anzeigt, gilt entweder $R1|\epsilon$. Die Funktion wird mit den entsprechend umgeänderten regulären Ausdrücken rekursiv aufgerufen.

   Pos R1 Der positive Abschluß kann ersetzt werden durch $R1, R1*$. Auch hier wird der reguläre Ausdruck umgeändert und die Funktion `seq_parse` erneut aufgerufen.

   Mal R1 Die n-malige Wiederholung kann auch ausgedrückt werden durch $R1, R1*|\epsilon$. Die Funktion `seq_parse` wird wieder mit den entsprechend umgeänderten regulären Ausdrücken rekursiv aufgerufen

   Seq R1 Da der Sequenz-Operator lediglich zur besseren Lesbarkeit dient, kann er entfernt werden.

   Xor R1 Beim Alternativoperator wird die Funktion `or_parse` aufgerufen, die anschließend entscheidet, ob die Liste von finalen Elementen nach den Regelbeschreibungen eines `dtd_part`'s aus $R1$ und des nachfolgenden regulären Ausdruckes korrekt aufgebaut ist.

   And R1 Beim `And`-Operator werden zunächst sämtliche Permutationen von $R1$ gebildet. Anschließend entscheidet die Funktion `and_parse`, ob die Bestandteile des logischen Elementes dem korrekten Aufbau einer Permutation und dem nachfolgenden Ausdruck entspricht.

2. Beginnt die Liste, die die Regel beschreibt, mit einem finalen Element, wird dieses mit dem ersten Element der `elementtypelist` verglichen. Sind die beiden Elemente gleich, muß der Rest der beiden Listen verglichen werden, was natürlich durch einen rekursiven Aufruf der Funktion `seq_parse` geschieht. Ansonsten wurde an dieser Stelle ein Fehler gefunden und die Verarbeitung wird abgebrochen.

3. Der Vergleich ist genau dann erfolgreich, wenn sowohl die `elementtypelist` als auch der reguläre Ausdruck komplett abgearbeitet wurden.

Funktionsname:

or_parse   or_parse

Signatur:

or_parse:: dtd_parts -> dtd_parts -> elementtypelist -> boolean

dtd_parts: Regelbeschreibung, die die verschiedenen Alternativen enthält

dtd_parts: Regelbeschreibung für den Teil, der der Alternative folgt

elementtypelist: Liste von finalen Elementen

boolean: Ergebnis des Vergleiches

Beschreibung:

Die Funktion entscheidet, ob die Liste von finalen Elementen nach den Regelbeschreibungen einer Alternative der `dtd_part`'s aus R1, an die der nachfolgende reguläre Ausdruck angehängt wird, korrekt aufgebaut sind.

Implementierung:

Jede Alternative wird mit dem Regelbeschreibungen für den nachfolgenden regulären Ausdruck konkateniert. Der so entstandene neue reguläre Ausdruck wird mit der `elementtypelist` verglichen, indem die Funktion `seq_parse` aufgerufen wird. Da es ausreicht, daß eine Alternative erfolgreich erkannt wird, erfolgt die Verknüpfung der Ergebnisse durch den Oder-Operator.

Funktionsname:

and_parse   and_parse

Signatur:

and_parse:: perm_dtd_parts -> dtd_parts -> elementtypelist -> boolean.

perm_dtd_parts: Regelbeschreibungen, die sämtliche Permutationen eines einzelnen regulären Ausdruckes enthält

dtd_parts: Regelbeschreibung für den Teil, der dem `And`-Operator folgt

elementtypelist: Liste von finalen Elementen

boolean: Ergebnis des Vergleiches

Beschreibung:

Die Funktion entscheidet, ob die Liste von finalen Elementen nach den Regelbeschreibungen einer Permutation, die zunächst mit dem nachfolgendem reguläre Ausdruck verbunden wird, korrekt aufgebaut sind.

Implementierung:

Jede Permutation des regulären Ausdruckes wird mit dem Regelbeschreibungen für den nachfolgenden regulären Ausdruck konkateniert. Der so entstandene neue reguläre Ausdruck wird mit der `elementtypelist` verglichen, indem die Funktion `seq_parse` aufgerufen wird. Da es ausreicht, daß eine Alternative erfolgreich erkannt wird, erfolgt die Verknüpfung der Ergebnisse durch den Oder-Operator.

Funktionsname:

cut_suffix cut_suffix

Signatur:

cut_suffix:: dtd_parts -> elementtypelist -> elementtypelist -> elementtypelist

dtd_parts: Regelbeschreibung eines logischen Elementes

elementtypelist: Liste von finalen Elementen

elementtypelist: Liste von finalen Elementen, die Zwischenergebnis aufnimmt

elementtypelist: Von der Funktion ermittelte Ergebnisliste

Beschreibung:

Die Funktion schneidet von der übergebenen `elementtypelist` den Teil ab, der nach der Regelbeschreibung einen korrekten Aufbau hat. Die Funktion war zunächst als Hilfsfunktion der Funktion `seq_parse` vorgesehen. Sie wurde jedoch, da sie bei bestimmten Ausdrücken fehlerhafte Ergebnisse liefert, in der Funktion `seq_parse` durch entsprechende Umformungen ersetzt.

Funktionsname:

pasteable_at pasteable_at

Signatur:

pasteable_at:: poss_dtd_parts -> elementtypelist -> elementtypelist -> elTypeList

poss_dtd_parts: Liste von Regelbeschreibungen, die in der aufrufenden Funktion gebildet wurde

elementtypelist: Liste von markierten finalen Elementen

elementtypelist: Liste von finalen Elementen, die hinter den markierten finalen Elementen steht

elTypeList: Ergebnisliste

Beschreibung:

Die Funktion ist eine Hilfsfunktion von `pasteable`. Sie führt die dort begonnene Verarbeitung weiter.

Implementierung:

Für jede Regelbeschreibung aus der übergebenen Liste wird versucht, die Ergebnisliste zu ergänzen, indem die folgenden Schritte durchgeführt werden:

1. Durch die Funktion `make_dtd` wird von der Regelbeschreibung ein Teil abgeschnitten, nach der die übergebene Liste von markierten finalen Elementen eventuell einen korrekten Aufbau haben könnte. Nach der so ermittelten Regelbeschreibung können ruhig einige `elementtypelist`'s zuviel korrekt aufgebaut sein. Diese werden von den nachfolgenden Phasen aussortiert.
2. Aus der ermittelten Regelbeschreibung wird mit Hilfe der Funktion `allEls` eine geordnete Liste von `elementtypelist`'s gebildet.
3. Durch Bildung des „falschen Kartesischen Produkteswird aus der geordneten Liste von `elementtypelist`'s eine Liste von `elementtypelist`'s erstellt, deren Listenelemente mit gewisser Wahrscheinlichkeit nach der abgeschnittenen Regelbeschreibung korrekt aufgebaut sind.
4. Nachdem mit der vordefinierten Funktion `mkset` die doppelten `elementtypelist`'s entfernt wurden, erfolgt eine abschließende Überprüfung der `elementtypelist`'s mit Hilfe der Funktion `insert_parse`. Wurden alle Regelbeschreibungen abgearbeitet, enthält die Ergebnisliste sämtliche `elementtypelist`'s, die anstelle der Liste der markierten finalen Elemente eingefügt werden können, ohne den korrekten syntaktischen Aufbau zu zerstören.

Funktionsname:

make_dtd make_dtd

Signatur:

make_dtd:: dtd_parts -> dtd_parts -> elementtypelist -> dtd_parts

dtd_parts: Regelbeschreibung, von deren Anfang eine Regelbeschreibung abgespaltet werden soll

dtd_parts: Regelbeschreibung, die das Zwischenergebnis aufnimmt

elementtypelist: Liste von finalen Elementen

dtd_parts: Ergebnis des Abspaltevorganges

Beschreibung:

Die Funktion spaltet von der übergebenen Regelbeschreibung eine Regelbeschreibung ab. Ziel dieses Vorganges ist es, eine Regelbeschreibung zu konstruieren, nach der die übergebene Liste von finalen Elementen hergeleitet worden sein könnte. Aus dieser Regelbeschreibung wird dann von den aufrufenden Funktionen eine Liste von Listen von finalen Elementen erzeugt, von denen jede mit gewisser Wahrscheinlichkeit die übergebene Liste von finalen Elementen ersetzen kann.

Implementierung:

Die Regelbeschreibung, die das Ergebnis der Funktion aufnimmt, ist am Anfang der Verarbeitung leer. Von der übergebenen Regelbeschreibung werden nun solange Ausdrücke abgeschnitten und der Ergebnisliste zugefügt, bis:

a) die übergebene `elementtypelist` nach den Regeln der Ergebnisliste korrekt aufgebaut ist. Sämtliche optionalen, sowie der erste nicht optionale Ausdruck müssen nun der Ergebnisliste zugefügt werden. Dieses geschieht durch Aufruf der Funktion `next_not_nullable`.

b) von der Ergebnisliste eine Regelbeschreibung abgespaltet werden kann, nach der die `elementtypelist` korrekt ist. Dieses wird mit der Funktion `moveto` festgestellt.

Funktionsname:

next_not_nullable next_not_nullable

Signatur:

next_not_nullable:: dtd_parts -> dtd_parts

dtd_parts: Regelbeschreibung von der Ausdrücke abgeschnitten werden sollen

dtd_parts: Ergebnis der Verarbeitung

Beschreibung:

Die Funktion spaltet von der übergebenen Regelbeschreibung alle optionalen sowie den ersten nicht optionalen Ausdruck ab.

Implementierung:

Bei jedem Ausdruck, der am Anfang der Regelbeschreibung steht, wird mit Hilfe der Funktion `nullable` festgestellt, ob er optional ist. Ist dieses der Fall wird der Ausdruck in die Ergebnisliste geschrieben und die Verarbeitung mit der restlichen Regelbeschreibung fortgesetzt. Die Funktion kann beendet werden, wenn entweder ein nichtoptionaler Ausdruck gefunden wurde, der der Ergebnisliste noch zugefügt wird, oder alle Ausdrücke verarbeitet wurden.

Funktionsname:

restruct_at restruct_at

Signatur:

restruct_at:: poss_dtd_parts -> elementtypelist -> elementtypelist -> boolean -> elementtypelist

poss_dtd_parts: Liste von Regelbeschreibungen, die in der aufrufenden Funktion gebildet wurde

elementtypelist: Liste von markierten finalen Elementen

elementtypelist: Liste von finalen Elementen, die hinter den markierten finalen Elementen steht

boolean: Wert der anzeigt, ob die Liste von markierten finalen Elementen nur aus einem Element besteht

elementtypelist: Ergebnisliste

Beschreibung:

Die Funktion führt die in der Funktion `restruction` begonnene Verarbeitung fort.

Implementierung:

Für jede Regelbeschreibung aus der übergebenen Liste werden die folgenden Schritte durchgeführt, um die Ergebnisliste zu ergänzen. Mit Hilfe der Funktion `seq_parse` wird zunächst überprüft, ob es sich um eine korrekte Regelbeschreibung handelt. Ist dieses der Fall, wird zunächst mit Hilfe der Funktion `firstpos` eine Liste mit finalen Elementen konstruiert, mit denen eine Liste von finalen Elementen beginnen kann, die der Regelbeschreibung entspricht. Dieses sind genau die finalen Elemente, die eventuell die markierte Liste von finalen Elementen ersetzen kann (Besteht die Liste der markierten Elemente nur aus einem Element, macht es wenig Sinn dieses Element durch sich selbst zu ersetzen. Es wird deshalb aus der durch `firstpos` gebildeten Liste entfernt.).

Funktionsname:

possible_alternatives possible_alternatives

Signatur:

possible_alternatives:: dtd_parts -> elementtypelist -> elementtypelist -> elementtypelist

dtd_parts: Regelbeschreibungen

elementtypelist: Liste mit finalen Elementen

elementtypelist: Liste von finalen Elementen, die hinter den markierten finalen Elementen steht

elementtypelist: Ergebnisliste

Beschreibung:

Die Funktion `possible_alternatives` ist eine Hilfsfunktion von `restruct_at`. Die Verarbeitung wird an dieser Stelle fortgeführt.

Implementierung:

Jedes der übergebenen fnalen Elemente wird vor die Liste von finalen Elementen gefügt, die hinter den markierten finalen Elementen steht. Durch Aufruf der Funktion `seq_pars` wird anschließend überprüft, ob die erzeugte Liste noch nach dem Aufbau der übergebenen Regelbeschreibung korrekt ist. Die finalen Elemente, für die dieses der Fall ist, können die markierte Liste von finalen Elementen ersetzen und werden in die Ergebnisliste eingefügt.

Funktionsname:

insertable_at insertable_at

Signatur:

insertable_at:: poss_dtd_parts -> elementtypelist -> elTypeList.

poss_dtd_parts: Liste von Regelbeschreibungen, die in der aufrufenden Funktion gebildet wurde

elementtypelist: Liste von finalen Elementen, die hinter den markierten finalen Elementen steht

elTypeList: Ergebnisliste

Beschreibung:

Die Funktion dient als Hilfsfunktion von `insertable`. Die dort begonnene Verarbeitung wird an dieser Stelle fortgeführt.

Implementierung:

Für jede Regelbeschreibung aus der übergebenen Liste wird durch Aufruf der Funktion `what_may_inserted` eine Liste von `dtd_parts`'s erzeugt. Jede `elementtypelist`, die einem `dtd_parts` der Liste entspricht, kann anschließend problemlos eingefügt werden. In der Funktion `expand` werden für jeden `dtd_parts` sämtliche `elementtypelist`'s gebildet, die dem `dtd_parts` entsprechen.

Funktionsname:

expand expand

Signatur:

expand:: poss_dtd_parts -> elTypeList.

poss_dtd_parts: Liste von Regelbeschreibungen, die in der aufrufenden Funktion gebildet wurde

elTypeList: Ergebnisliste

Beschreibung:

Die Funktion `expand` ermittelt für jeden `dtd_parts` der übergebenenen Liste, sämtliche `elementtypelist`'s , die nach den Regeln des `dtd_parts`'s korrekt aufgebaut sind.

Implementierung:

Für jede Regelbeschreibung aus der übergebenen Liste wird versucht, eine Liste zu erstellen, die alle Listen von finalen Elementen enthält, die dem Aufbau der Regelbeschreibung entsprechen. Dazu werden folgende Schritte durchgeführt:

1. Durch die Funktion `allEls` wird eine geordnete Liste von `elementtypelist`'s gebildet wird.
2. Von der geordneten Listen von elementtypelist's wird durch die Funktion `building_elTypeList` das „falsche Kartesische Produkt“ gebildet. Das Ergebnis ist eine Liste von `elementtypelist`'s, deren Listenelemente mit gewisser Wahrscheinlichkeit der Regelbeschreibung entsprechen.

3. Mit der vordefinierten Funktion `mkset` werden die doppelten `elementtypelist`'s entfernt. Die Funktion `insert_parse` wählt schließlich aus der Liste von Listen von finalen Elementen die Listen aus, die nach der Regelbeschreibung einen korrekten Aufbau haben.

Funktionsname:

insert_parse insert_parse

Signatur:

insert_parse:: dtd_parts -> elTypeList -> elementtypelist -> elementtypelist -> elTypeList

dtd_parts: Regelbeschreibung, mit der die Überprüfung durchgeführt wird

elTypeList: Liste von `elementtypelist`'s

elementtypelist: `elementtypelist`, die vor jeder `elementtypelist` angefügt wird

elementtypelist: `elementtypelist`, die hinter jeder `elementtypelist` angefügt wird

elTypeList: Ergebnisteliste

Beschreibung:

Die Funktion fügt jede `elementtypelist` der Liste, die, nachdem die beiden übergebenen `elementtypelist`'s vorne bzw. hinten angehängt wurden, der übergebenen Regelbeschreibung entsprechen, in die Ergebnisliste ein.

Implementierung:

Sämtliche `elementtypelist`'s der übergebenen Liste werden zunächst mit den beiden Listen von finalen Elementen konkateniert. Anschließend wird mit der Funktion `seq_parse` überprüft, ob die gebildete neue `elementtypelist` nach der Regelbeschreibung korrekt ist. Ist dieses der Fall, wird die `elementtypelist` in die Ergebnisliste eingefügt.

Funktionsname:

what_may_inserted what_may_inserted

Signatur:

what_may_inserted:: dtd_parts -> dtd_parts -> elementtypelist -> poss_dtd_parts

dtd_parts: Regelbeschreibung

dtd_parts: Regelbeschreibung zur Aufnahme der Zwischenergebnisse

elementtypelist: Bestandteile einer logischen Elementes

poss_dtd_parts: Ergenisliste von Regelbeschreibungen

Beschreibung:

Die Funktion ermittelt eine Liste von Regelbeschreibungen. An jede beliebige `elementtypelist`, die nach einer Regelbeschreibung aus der Ergebnisliste einen korrekten Aufbau hat, kann die übergebene `elementtypelist` angehängt werden. Das Ergebnis ist dann nach der übergebenen Regelbeschreibung korrekt aufgebaut. Voraussetzung für den korrekten Ablauf der Funktion ist allerdings, daß die übergebenene Liste von finalen Elementen nach der übergebenen Regelbeschreibung einen korrekten Aufbau hat.

Implementierung:

Aufgrund der Vorbedingung wird ersichtlicht, daß die Ergebnisliste nur mit den optionalen Bestandteilen der übergebenen Regelbeschreibung ergänzt werden kann. Deshalb kann die Verarbeitung auch beendet werden, wenn ein nicht optionaler Ausdruck, also ein finales Element, gefunden wurde. Ansonsten werden sämtliche Ausdrücke verarbeitet, aus denen die Regelbeschreibung besteht. Der Operator, der auf den ersten Ausdruck angewendet wird, bestimmt die anzuwendende Aktion:

Seq R1 Da der Sequenz-Operator lediglich zur besseren Lesbarkeit dient, kann er entfernt werden.

Xor R1 Beim Alternativoperator wird die Funktion `or_insert` aufgerufen, die anschließend die weiteren Regelbeschreibungen ermittelt.

And R1 Beim `And`-Operator wird die Funktion `perminsert` aufgerufen, nachdem von $R1$ durch Aufruf der Funktion `permutate` sämtliche Permutationen gebildet wurden.

Mal R1 Tritt der Operator für die n-malige Wiederholung auf, wird zunächst durch Aufruf der Funktion `option_may_be_inserted` überprüft, ob der optionale Ausdruck eventuell noch nicht vorhanden ist. Sicherheit bietet jedoch erst der Aufruf der Funktion `seq_parse`, die mit der Regelbeschreibung aufgerufen wird, die dem Ausdruck folgt, auf den der `Mal`-Operator angewendet wird. Dadurch wird festgestellt, ob der zu $R1$ gehörige Ausdruck in der `elementtypelist` vorkommt. Ist dieses der Fall, kann nur noch der Ausdruck `Mal R1` der Ergebnisliste zugefügt werden, da für reguläre Ausdrücke die Rechenvorschrift $a*, a* = a*$ gilt. Ansonsten wird ebenfalls der Ausdruck `Mal R1` der Ergebnisliste zugefügt und die Verarbeitung wird mit der restlichen Regelbeschreibung fortgesetzt.

Pos R1 Beim positiven Abschluß ist gewährleistet, daß der Ausdruck $R1$ in der `elementtypelist` mindestens einmal vorkommen muß. Er kann folglich in einer beliebigen `elementtypelist` noch 0 bis n-mal am Anfang dazugefügt werden, ohne daß der korrekte Aufbau nach der Regelbeschreibung verletzt wird.

Opt R1 Der Options-Operator läßt sich fast ebenso wie der Operator für die n-malige Wiederholung behandeln. Durch Aufruf der Funktionen `option_may_be_inserted` und `seq_parse`, wobei die Parameter jeweils die gleichen sind, wird überprüft, ob der zu $R1$ gehörige Ausdruck bereits in der `elementtypelist` vorkommt. Ist dieses der Fall kann nichts eingefügt werden, und die Verarbeitung wird abgebrochen. Ansonsten kann zur `elementtypelist` am Anfang $R1$ dazugefügt werden, ohne daß der korrekte Aufbau nach der Regelbeschreibung verletzt wird. Der Ausdruck `Opt R1` wird deshalb der Ergebnisliste zugefügt, und die Verarbeitung wird mit der restlichen Regelbeschreibung fortgesetzt.

Funktionsname:

permutate permutate

Signatur:

permutate:: dtd_parts -> perm_dtd_parts

dtd_parts: Regelbeschreibung die permutiert werden soll

perm_dtd_parts: Liste von sämtliche permutierten Regelbeschreibungen

Beschreibung:

Die Funktion bildet von der übergebenen Regelbeschreibung sämtliche Permutationen. Permutationen entstehen dadurch, daß die Ausdrücke aus denen die Regelbeschreibung besteht in beliebiger Weise umgestellt werden.

Implementierung:

Die Permutation einer leeren Regelbeschreibung ist definitionsgemäß die leere Liste. Von jeder anderen Regelbeschreibung wird das erste Element vom Anfang abgespaltet und die Permutation der Restliste gebildet. Mittels der Funktion `perm_insert_in_all` wird das abgespaltene Element anschließend zu den Permutationen der Restliste zugefügt.

Funktionsname:

perm_insert_in_all perm_insert_in_all

Signatur:

perm_insert_in_all:: dtd_part -> integer -> perm_dtd_parts -> perm_dtd_parts

dtd_part: Element, welches in jede permutierten Regelbeschreibungen eingefügt werden soll

integer: Numerischer Wert, der angibt aus wievielen Elementen, jede Regelbeschreibung aus `perm_dtd_parts` nach der Verarbeitung bestehen soll

perm_dtd_parts: Liste von sämtlichen permutierten Regelbeschreibungen

Beschreibung:

Die Funktion fügt das übergebene Element an jeder Stelle in jeder Regelbeschreibung der `perm_dtd_parts` ein.

Implementierung:

In die leere Liste muß das Element nur dann eingefügt werden, wenn bislang noch keine Permutationen gebildet wurden. N ist in diesem Fall gleich 1. Ansonsten wird durch Aufruf der Funktion `perm_insert_at_pos` gewährleistet, daß eine Liste erzeugt wird, in der das übergebene Element an jeder Stelle der ersten bislang erzeugten permutierten Regelbeschreibung eingefügt wird. Die Verarbeitung wird anschließend mit der Restliste der bislang erzeugten permutierten Regelbeschreibungen fortgeführt.

Funktionsname:

perm_insert_at_pos　　　perm_insert_at_pos

Signatur:

perm_insert_at_pos:: dtd_part -> integer -> dtd_parts->perm_dtd_parts

dtd_part: Element, daß in die folgende Regelbeschreibung eingefügt werden soll

integer: Aktuelle Position, an der das Element in die Regelbeschreibung eingesetzt werden soll

dtd_parts: Regelbeschreibung

perm_dtd_parts: Liste von permutierten Regelbeschreibungen

Beschreibung:

Die Funktion fügt das übergebene Element an jeder Postion in der Regelbeschreibung ein. Jder Einfügevorgang liefert dabei eine neue permutierte Regelbeschreibung, deren Länge um 1 größer ist als die alte Regelbeschreibung.

Implementierung:

Die Funktion wird am Anfang mit dem integer-Wert N aufgerufen, der gleich der Länge jeder Regelbeschreibung nach der Verarbeitung ist. Das neue Element wird durch Aufruf der Funktion `perm_new_at_pos` an der Stelle N eingefügt. Durch den rekursiven Aufruf der Funktion mit dem Wert N- 1 wird gewährleistet, daß das Element auch an jeder niedrigeren Stelle eingefügt wird. Ist N gleich 0 kann die Verarbeitung beendet werden.

Funktionsname:

perm_new_at_pos　　　perm_new_at_pos

Signatur:

perm_new_at_pos:: dtd_part -> integer -> dtd_parts -> dtd_parts

dtd_part: Element, daß in die folgende Regelbeschreibung eingefügt werden soll

integer: Position an der das Element in die Regelbeschreibung eingesetzt werden soll

dtd_parts: Regelbeschreibung

dtd_parts: Ergebnis der Einfügeoperation

Beschreibung:

Die Funktion fügt das übergebene Element an N-ter Stelle in die Regelbeschreibung ein.

Implementierung:

Durch rekursiven Aufruf der Funktion wird die richtige Einfügestelle in der Regelbeschreibung ermittelt. Diese ist genau dann gefunden, wenn N den Wert 1 annimmt. Ist die Liste vorher vollständig durchlaufen worden, wird das Element am Anfang des regulären Ausdruckes eingefügt. N muß an dieser Stelle ebenfalls den Wert 1 haben.

Funktionsname:

allEls allEls

Signatur:

allEls:: dtd_parts -> elTypeList

dtd_parts: Regelbeschreibung

elTypeList: Geordnete Liste von `elementtyplist`'s

Beschreibung:

Die Funktion erzeugt aus den finalen Elementen, die die übergebene Regelbeschreibung enthält, eine geordnete Liste von `elementtyplist`'s die folgende Eigenschaft besitzt: Tritt in einer Regelbeschreibung ein Ausdruck a vor dem Ausdruck b auf, sind die finalen Elemente von a in kleineren Listen als die finalen Elemente von b enthalten. Da bei Alternativen und Ausdrücken, auf denen der `And`-Operator angewendet wird, nicht bestimmt werden kann, welcher Ausdruck vor dem anderen auftritt, werden hier mehrere Listen zu einer zusammengefügt.

Implementierung:

Für jeden Ausdruck der Regelbeschreibung wird zunächst die geordnete Liste von `elementtypelist`'s bestimmt und anschließend mit den geordneten Listen von `elementtyplist`'s der restlichen Regelbeschreibung zusammengefaßt. Die Verarbeitung ist beendet, wenn die gesamte Regelbeschreibung abgearbeitet wurde. Die geordnete Liste von `elementtyplist`'s eines finalen Elementes ist eine Liste, die nur das finale Element enthält. Bei den unären Operatoren wird versucht, durch Entfernung des Operators und rekursiven Aufruf des Funktion mit dem Ausdruck, auf dem der Operator angewendet wurde, diesen Fall herbeizuführen. Auch bei den Operatoren, die auf mehrere Ausdrücke angewendet werden, wird versucht den Operator zu entfernen. Dazu bedarf es aber je nach Operator des Aufrufes einer der Funktionen `seqallEls`, `xorallEls` bzw. `andAllEls`.

Funktionsname:

seqallEls seqallEls

Signatur:

seqallEls:: dtd_parts -> elTypeList

dtd_parts: Regelbeschreibung, die mehrere aufeinanderfolgende Ausdrücke enthält

elTypeList: Geordnete Liste von `elementtyplist`'s

Beschreibung:

Die Funktion erzeugt für die übergebene Regelbeschreibung eine geordnete Liste von `elementtypelist`'s.

Implementierung:

Mit Hilfe der Funktion `allEls` wird für jeden Ausdruck eine geordnete Liste von `elementtypelist`'s erzeugt. Da die Ausdrücke hintereinander angeordnet sind, genügt es, die von der Funktion `allEls` erzeugten Ergebnislisten mit Hilfe der Konkatenation aneinanderzureihen.

Funktionsname:

xorallEls xorallEls

Signatur:

xorallEls:: dtd_parts -> elTypeList -> elTypeList.

Signatur:

dtd_parts:: Regelbeschreibung, die mehrere Alternativen enthält

elTypeList: Geordnete Liste von `elementtyplist`'s, die das Zwischenergebnis enthält

elTypeList: Geordnete Liste von `elementtyplist`'s

Beschreibung:

Die Funktion erzeugt für jede Alternative eine geordnete Liste von `elementtyplist`'s. Die entstehenden Listen werden anschließend zusammengefügt.

Implementierung:

Für jede der Alternativen wird zunächst durch Aufruf der Funktion `allEls` die geordnete Liste von `elementtyplist`'s gebildet. Das Ergebnis dieser Funktion wird mit Hilfe der Funktion `stick_together` der bisher ermittelten Ergebnisliste zugefügt.

Funktionsname:

andallEls andallEls

Signatur:

andallEls:: perm_dtd_parts -> elTypeList -> elTypeList.

perm_dtd_parts: Regelbeschreibungen, die sämtliche Permutationen eines einzelnen regulären Ausdruckes enthält

elTypeList: Geordnete Liste von `elementtyplist`'s, die das Zwischenergebnis enthält

elTypeList: Geordnete Liste von `elementtyplist`'s

Beschreibung:

Die Funktion erzeugt für jede der in `perm_dtd_parts` enthaltenen Permutationen eine geordnete Liste von `elementtyplist`'s. Auch in der Funktion `andallEls` werden die entstehenden Listen zu einer Liste zusammengefaßt.

Implementierung:

Die Funktion arbeitet wie die Funktion `xorallEls`. Statt mit einer Alternative wird die Funktion `allEls` mit einer Permutation der Regelbeschreibung aufgerufen. Die Permutationen wurden bereits in der aufrufenden Funktion gebildet.

Funktionsname:

stick_together stick_together

Signatur:

stick_together:: elTypeList->elTypeList->elTypeList

elTypeList: Erste geordnete Liste von `elementtyplist`'s

elTypeList: Zweite geordnete Liste von `elementtyplist`'s

elTypeList: Zusammengefügte geordnete Liste von `elementtyplist`'s

Beschreibung:

Die Funktion faßt zwei geordnete Listen von `elementtyplist`'s zusammen, indem je zwei `elementtyplist`'s zusammengefügt werden, die an der gleichen Position in den Listen auftreten.

Implementierung:

Die jeweils erste `elementtypelist` von beiden Listen wird abgespaltet und zusammengefügt. Danach wird das Ergebnis des Zusammenfügevorganges der beiden Restlisten angefügt. Ist eine der beiden Listen komplett abgearbeitet, wird der Rest der anderen Liste an das bisher ermittelte Ergebnis angehängt, und die Verarbeitung ist beendet.

Funktionsname:

building_elTypeList building_elTypeList

Signatur:

building_elTypeList:: elTypeList -> elTypeList

elTypeList: Geordnete Liste von `elementtyplist`'s, die von der Funktion `allEls` erzeugt wurde

elTypeList: Liste von `elementtyplist`'s

Beschreibung:

Die Funktion bildet aus den übergebenen `elementtyplist`'s, die für diese Funktion als Mengen interpretiert werden, das „falsche Kartesische Produkt“. Unter dem „falschen Kartesischen Produkt“ der Mengen $A\_1, \ldots, A\_n$ verstehen wir die Menge der n-stelligen Tupel $(a\_1, \ldots, a\_n)$ wobei gilt a_i ist Element A_i oder $a\_i = \epsilon$. In den folgenden Funktionen wird jedes Tupel als `elementtypelist` dargestellt. An dieser Stelle soll noch einmal das Zusammenwirken der Funktionen `allEls` und `building_elTypeList` erläutert werden: Durch die

Funktion `allEls` werden die finalen Elemente für eine Regelbeschreibung in einen geordnete Liste von `elementtypelist`'s gechrieben. Die Ordnung drückt dabei die Reihenfolge aus, in der die finalen Elemente in einer `elementtypelist`, die nach der Regelbeschreibung korrekt aufgebaut ist, auftauchen können. Bildet die Funktion `building_elTypeList` daraus nun das „falsche Kartesische Produkt“ entsteht eine Liste von `elementtypelist`'s, wobei jede `elementtypelist` mit gewisser Wahrscheinlichkeit nach der Regelbeschreibung korrekt aufgebaut ist.

Implementierung:

Das „falsche Kartesische Produkt“ der leeren Liste ist definitionsgemäß die leere Liste. Ansonsten wird vom Anfang der Liste eine `elementtypelist` abgespaltet und das „falsche Kartesische Produkt“ der Restliste gebildet. Durch Aufruf der Funktion `insert_list_in_elTypeList` wird gewährleistet, daß mit der abgespaltenen `elementtypelist` und dem „falschen Kartesischen Produkt“ der Restliste das „falsche Kartesische Produkt“ der Gesamtliste erzeugt wird.

Funktionsname:

insert_list_in_elTypeList insert_list_in_elTypeList

Signatur:

insert_list_in_elTypeList:: elementtypelist -> elTypeList -> elTypeList.

elementtypelist: Liste von elementtypes

elTypeList: Liste von `elementtypelist`'s, die im folgenden als Menge von Tupeln interpretiert wird

elTypeList: Menge von Tupeln, die als Liste von `elementtypelist`'s dargestellt wird

Beschreibung:

Die Funktion erzeugt aus der Menge A_i, die durch die übergebene `elementtypelist` repräsentiert wird und der Menge von Tupeln, die das „falsche Kartesische Produkt“ der Mengen $A\_i{+}1, \ldots, A\_n$ bildet, das „falsche Kartesische Produkt“ der Mengen $A\_i, \ldots, A\_n$.

Implementierung:

Durch den Aufruf der Funktion `insert_element_in_elTypeList` mit jedem Element der `elementtypelist` wird mit jedem Tupel des „falschen Kartesischen Produkt“ der Mengen $A\_i{+}1, \ldots, A\_n$ neue Tupel gebildet. Sämtliche Tupel werden konkateniert und bilden so das „falsche Kartesische Produkt“ der Mengen $A\_i, \ldots, A\_n$.

Funktionsname:

insert_element_in_elTypeList insert_element_in_elTypeList

Signatur:

insert_element_in_elTypeList:: elementtype -> elTypeList -> elTypeList.

elementtype: Element, mit dem neue Tupel gebildet werden sollen

elTypeList: Liste von `elementtypelist`'s, die im folgenden als Menge von Tupeln interpretiert wird

elTypeList: Menge von Tupeln, die als Liste von `elementtypelist`'s dargestellt wird

Beschreibung:

Die Funktion erzeugt mit dem übergebenen Element und der Menge von Tupeln, die das „falsche Kartesische Produkt“ der Mengen $A\_i{+}1, \ldots, A\_n$ bildet eine Menge von neuen Tupeln.

Implementierung:

Das übergebene Element wird im folgenden a_i genannt, jedes Tupel wird mit $(a\_j, \ldots, a\_n)$ bezeichnet. Für jedes Tupel aus der Menge werden nun die beiden Tupel $(a\_i, a\_j, \ldots, a\_n)$ und $(a\_j, \ldots, a\_n)$ (für den Fall, daß $a\_i = \epsilon$ ist) zugefügt. Um den Fall zu simulieren, daß $a\_j = \ldots = a\_n = \epsilon$ wird am Ende der Verarbeitung das Tupel (a_i) zur Erebnisliste zugefügt.

Funktionsname:

nullable nullable

Signatur:

nullable:: dtd_parts -> boolean.

dtd_parts: Regelbeschreibung

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion entscheidet, ob die leere Liste nach der übergebenen Regelbeschreibung korrekt aufgebaut ist.

Implementierung:

Nach einer Regelbeschreibung, die nur aus der leeren Liste besteht, ist die leere Liste natürlich korrekt aufgebaut. Besteht die Regelbeschreibung dagegen nur aus einem Ausdruck, der ein finales Element enthält, liefert der Vergleich mit der leeren Liste ein negatives Ergebnis. Ansonsten werden sämtliche Ausdrücke verarbeitet, aus denen die Regelbeschreibung besteht. Auch in dieser Funktion bestimmt der Operator, der auf den ersten Ausdruck angewendet wird, die anzuwendende Aktion:

Opt R1 Nach Ausdrücken, auf denen Options-Operator angewendet wird, kann die leere Liste immer korrekt hergeleitet werden. Das Ergebnis der Funktion hängt folglich nur vom Wert der restlichen Regelbeschreibung ab.

Mal R1 Was für den Options-Operator gesagt wurde, gilt auch für den Operator, der die n-malige Wiederholung anzeigt.

Seq R1 Aus einem Ausdruck $R1$, auf den der Sequenz-Operator angewendet wird, kann nur dann die leere Liste herleiten, wenn die leere Liste nach den Regelbeschreibungen sämtlicher Ausdrücke aus denen $R1$ besteht, korrekt aufgebaut ist. Dieses wird durch den Aufruf der Funktion `all_parts_are_nullable` überprüft.

And R1 Der `And`-Operator wird ebenso wie der Sequenz-Operator behandelt.

Pos R1 Aus dem Ausdruck `Pos R1` kann genau dann die leere Liste hergeleitet werden, wenn sie auch aus dem Ausdruck $R1$ hergeleitet werden kann. Entsprechend wird die Funktion `nullable` rekursiv aufgerufen.

Seq R1 Die leere Liste ist nach einem Ausdruck `Xor R1` genau dann korrekt aufgebaut, wenn eine der Alternativen aus denen $R1$ besteht die leere Liste herleitet. Die Funktion `one_part_is_nullable` stellt dieses fest.

Funktionsname:

one_part_is_nullable

one_part_is_nullable

Signatur:

one_part_is_nullable:: dtd_parts -> boolean

dtd_parts: Regelbeschreibung, die mehrere Alternativen enthält

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion stellt fest, ob die leere Liste nach einer der Alternativen, aus denen die übergebene Regelbeschreibung besteht, einen korrekten Aufbau hat.

Implementierung:

Durch Aufruf der Funktion `nullable` wird für jede Alternative überprüft, ob sie die leere Liste herleiten kann. Die Ergebnisse werden anschließend mit dem Oder-Operator verknüpft.

Funktionsname:

all_parts_are_nullable

all_parts_are_nullable

Signatur:

all_parts_are_nullable:: dtd_parts -> boolean

dtd_parts: Regelbeschreibung, die aus mehreren Ausdrücken besteht

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion stellt fest, ob die leere Liste nach jedem der Ausdrücke, aus denen die übergebene Regelbeschreibung besteht, einen korrekten Aufbau hat.

Implementierung:

Durch Aufruf der Funktion `nullable` wird für jeden Ausdruck überprüft, ob er die leere Liste herleiten kann. Die Ergebnisse werden anschließend mit dem Und-Operator verknüpft.

Funktionsname:

firstpos firstpos

Signatur:

firstpos:: dtd_parts -> elementtypelist.

dtd_parts: Regelbeschreibung

elementtypelist: Liste von finalen Elementen

Beschreibung:

Die Funktion erzeugt eine Liste von finalen Elementen mit denen eine elementtypelist, die nach der Regelbeschreibung einen korrekten Aufbau hat, beginnen kann.

Implementierung:

Definitionsgemäß ist die Liste mit denen ein finales Element beginnen kann, eine Liste die eben genau dieses finale Element enthält. Folglich wird versucht, durch Entfernung der Operatoren einen Ausdruck zu erzeugen, der mit einem finalen Element beginnt. Zwei Besonderheiten sind darüber hinaus zu berücksichtigen:

a) Kann ein Ausdruck weggelassen werden, was entweder per Definition der Fall ist oder durch den Aufruf der Funktion `nullable` festgestellt wird, müssen die finalen Elemente, mit denen `elementtypelist`'s beginnen können, die den restliche Regelbeschreibungen entsprechen, hinzugefügt werden.

b) Wird der `Xor`- oder der `And`-Operator auf einen Ausdruck angewendet, kann eine `elementtypelist` mit den finalen Elementen jeder Alternative bzw. jedes Ausdruckes beginnen, aus dem die Regelbeschreibung besteht. Diese werden durch den Aufruf der Funktion `all_firstpos` ermittelt.

Funktionsname:

all_firstpos all_firstpos

Signatur:

all_firstpos:: dtd_parts -> elementtypelist

dtd_parts: Regelbeschreibung, die aus mehreren Ausdrücken besteht

elementtypelist: Liste von finalen Elementen

Beschreibung:

Die übergebene Regelbeschreibung besteht aus mehreren Ausdrücken. Die Funktion erzeugt eine Liste von finalen Elementen, mit denen eine `elementtypelist` beginnen kann, die nach den Regelbeschreibungen eines Ausdruckes korrekt ist.

Implementierung:

Für jeden Ausdruck, aus denen die Regelbeschreibung besteht, wird durch den Aufruf der Funktion `firstpos` eine Liste der finalen Elemente gebildet. Die so entstehenden Listen werden anschließend durch Konkatenation zu einer Liste zusammengefaßt.

Funktionsname:

moveto moveto

Signatur:

moveto:: dtd_parts -> elementtypelist -> poss_dtd_parts.

dtd_parts: Regelbeschreibung für ein logisches Element

elementtypelist: Liste von finalen Elementen

poss_dtd_parts: Liste von Regelbeschreibungen, die von der Funktion ermittelt wurden

Beschreibung:

Von der übergebenen Regelbeschreibung wird vom Anfang eine Regelbeschreibung abgeschnitten, nach der übergene Liste von finalen Elementen einen korrekten Aufbau hat. Der von der Regelbeschreibung übriggebliebene Rest wird der Ergebnisliste zugefügt. Die so ermittelte Liste enthält sämtliche Regelbeschreibungen, die entstehen können, nachdem die übergebene Regelbeschreibung die Bestandteile der `elementtypelist` erkannt hat.

Implementierung:

Die Ermittlung der Ergebnisliste wird in folgender Weise durchgeführt:

1. Es wird zunächst mit Hilfe der Rechenregeln für reguläre Ausdrücke versucht, die Regelbeschreibung so umzuformen, daß diese mit einem finalen Element beginnt. Der Operator, der auf das erste Listenelement der Regelbeschreibung angewendet wird, bestimmt die Umformung. Außerdem wird versucht durch Anwendung der Funktionen `firstpos` und `nullable` einige Ergebnisse auszuschließen.

   Opt R1 Da die Option das ein oder nullmalige Auftreten anzeigt, gilt entweder $R1|\epsilon$. Die Funktion wird mit den entsprechend umgeänderten regulären Ausdrücken rekursiv aufgerufen. Durch die Bestimmung, ob das nächste Element in `firstpos` von $R1$ ist, wird versucht die Alternative $R1$ auszuschließen.

   Pos R1 Der positive Abschluß wird durch $R1, R1*$ ersetzt und die Funktion wird rekursiv aufgerufen.

   Mal R1 Ähnlich wie bei der Funktion `seq_parse` wird die n-malige Wiederholung durch $R1, R1 * |\epsilon$ ausgedrückt. Mit den umgeänderten Regelbeschreibungen wird anschließend die Funktion rekursiv aufzurufen, nachdem durch die Bestimmung, ob sich das nächste Element der `elementtypelist` in `firstpos` von $R1$ befindet, versucht wurde, die leere Alternative auszuwählen.

   Seq R1 Da der Sequenz-Operator lediglich zur besseren Lesbarkeit dient, kann er entfernt werden.

Xor R1 Beim Alternativoperator wird die Funktion `or_searchways` aufgerufen, die anschließend die weiteren Regelbeschreibungen ermittelt. Durch die Überprüfung, ob die Funktion `nullable` für den Ausdruck `Xor R1` den Wert `true` liefert, wird versucht auszuschließen, daß eine Verarbeitung mit dem Rest des regulären Ausdruckes durchgeführt werden muß.

And R1 Beim `And`-Operator wird die Funktion `and_searchways` aufgerufen, nachdem von $R1$ durch Aufruf der Funktion `permutate` sämtliche Permutationen gebildet wurden. `and_searchways` ergänzt danach die Ergebnisliste. Auch hier wird durch die Überprüfung mit Hilfe der Funktion `nullable` für den Ausdruck `And R1` versucht, die Verarbeitung mit dem Rest des regulären Ausdruckes zu unterbinden.

2. Wenn der reguläre Ausdruck mit einem finalen Element beginnt, wird dieses mit dem ersten Element der `elementtypelist` verglichen. Sind die beiden Elemente gleich, muß die Verarbeitung mit der Rest der beiden Listen fortgesetzt. Sind die beiden Elemente ungleich, so ist die `elementtypelist` nach der Regelbeschreibung nicht korrekt aufgebaut. Dieses kann zum Beispiel dann passieren, wenn versucht wird, die `elementtypelist` mit mehreren Alternativen zu vergleichen.
3. Eine Regelbeschreibung kann der Ergebnisliste, dann zugefügt werden, wenn die `elementtypelist` komplett abgearbeitet wurde.

Funktionsname:

or_searchways — or_searchways

Signatur:

or_searchways:: dtd_parts -> dtd_parts -> elementtypelist -> poss_dtd_parts.

dtd_parts: Regelbeschreibung, die die verschiedenen Alternativen enthält

dtd_parts: Regelbeschreibung für den Teil, der der Alternative folgt

elementtypelist: Liste von finalen Elementen

poss_dtd_parts: Liste von Regelbeschreibungen, die von der Funktion ermittelt wurden

Beschreibung:

Die Funktion versucht für jede der in der Regebeschreibung erhaltenen Alternative, eine Ergebnisliste zu ermitteln, die wie in der Funktion `moveto` beschrieben erzeugt wird.

Implementierung:

Durch Anwendung der Funktionen `firstpos` und `nullable` wird nach erfolgversprechenden Alternativen gesucht, d.h. Regelableitungen deren linke Seite mit den finalen Elementen der `elementtypelist` beginnen kann. Die erfolgversprechenden Alternativen werden anschließend mit der Regelbeschreibung für den nachfolgenden Ausdruck konkateniert und zusammen mit der `elementtypelist` wird die Funktion `moveto` aufgerufen, um die Ergebnisliste zu ergänzen.

Funktionsname:

and_searchways and_searchways

Signatur:

and_searchways:: perm_dtd_parts -> dtd_parts -> elementtypelist -> poss_dtd_parts.

perm_dtd_parts: Regelbeschreibungen, die sämtliche Permutationen eines einzelnen regulären Ausdruckes enthält

dtd_parts: Regelbeschreibung für den Teil, der dem And-Operator folgt

elementtypelist: Liste von finalen Elementen

poss_dtd_parts: Liste von Regelbeschreibungen, die von der Funktion ermittelt wurden

Beschreibung:

Die Funktion erzeugt für jede der in `perm_dtd_parts` enthaltenen Permutationen eine Ergebnisliste. Die Ergebnisliste wird durch die Funktion `moveto` ermittelt.

Implementierung:

Die Funktion arbeitet genau wie die Funktion `or_searchways`. Statt mit einer Alternative wird die Regelbeschreibung für den Teil, der dem And-Operator folgt, mit einer permutierten Regelbeschreibung konkateniert.

Funktionsname:

or_insert or_insert

Signatur:

or_insert:: dtd_parts -> dtd_parts -> dtd_parts -> elementtypelist -> poss_dtd_parts

dtd_parts: Regelbeschreibung, die die verschiedenen Alternativen enthält

dtd_parts: Regelbeschreibung für den Teil, der der Alternative folgt

dtd_parts: Regelbeschreibung, die das bisher ermittelte Zwischenergebnis enthält

elementtypelist: Liste von finalen Elementen

poss_dtd_parts: Liste von Regelbeschreibungen, die von der Funktion ermittelt wurden

Beschreibung:

Jede der in der Regelbeschreibung erhaltenen Alternativen wird mit der Regelbeschreibung konkateniert, die dem Ausdruck folgt, auf den der `Xor`-Operator angewendet wird. Eine so gebildete neue Regelbeschreibung wird als korrekt bezeichnet, wenn die übergebenen Bestandteile eines logischen Elementes nach der Regelbeschreibung korrekt aufgebaut sind. Für jede korrekte Regelbeschreibung wird anschließend versucht, eine Ergebnisliste zu ermitteln, die wie in der Funktion `what_may_inserted` beschrieben erzeugt wird.

Implementierung:

Sämtliche Alternativen der Regelbeschreibung werden nacheinander verarbeitet. Durch Aufruf der Funktionen `first_part_missing` wird überprüft, ob der erste Ausdruck der Alternative fehlt. Ist dieses der Fall, wird die Alternative mit der Regelbeschreibung konkateniert, die dem Ausdruck folgt, auf den der Xor-Operator angewendet wird. Ergibt der anschließende Aufruf der Funktion `seq_parse`, daß es sich um eine korrekte Alternative handelte, wird die Ergebnisliste durch den Aufruf der Funktion `what_may_inserted` ergänzt.

Funktionsname:

option_may_inserted option_may_inserted

Signatur:

option_may_inserted:: dtd_parts -> elementtypelist -> boolean

dtd_parts: Regelbeschreibung, die einem optionalen Ausdruck folgt

elementtypelist: Liste von finalen Elementen

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion ermittelt, ob ein optionaler Ausdruck eventuell in der `elementtypelist` vorkommt.

Implementierung:

Kommt ein optionaler Ausdruck nicht der übergebenen `elementtypelist` vor, muß entweder gelten:

a) die `elementtypelist` muß leer sein.

b) das erste Element der `elementtypelist` muß in der Liste der finalen Elemente sein, mit denen eine `elementtypelist` beginnen kann, die der Regelbeschreibung entspricht, die dem optionalen Ausdruck folgt.

Funktionsname:

perminsert perminsert

Signatur:

perminsert:: perm_dtd_parts -> dtd_parts -> dtd_parts -> elementtypelist -> poss_dtd_parts

perm_dtd_parts: Regelbeschreibungen, die sämtliche Permutationen eines einzelnen regulären Ausdruckes enthält

dtd_parts: Regelbeschreibung für den Teil, der der Alternative folgt

dtd_parts: Regelbeschreibung, die das bisher ermittelte Zwischenergebnis enthält

elementtypelist: Bestandteile eines logischen Elementes poss_dtd_parts: Liste von Regelbeschreibungen, die von der Funktion ermittelt wurden

Beschreibung:

Jede der in `perm_dtd_parts` enthaltenen Permutationen wird mit der Regelbeschreibung konkateniert, die dem Ausdruck folgt, auf den der `And`-Operator angewendet wird. Korrekte Regelbeschreibungen werden ebenso definiert, wie unter `or_insert` beschrieben. Für jede korrekte Regelbeschreibung wird anschließend versucht eine Ergebnisliste zu ermitteln, die wie in der Funktion `what_may_inserted` beschrieben erzeugt wird.

Implementierung:

Die Funktion arbeitet genau wie die Funktion `or_insert`. Statt mit verschiedenen Alternativen wird die Funktion mit sämtlichen Permutationen aufgerufen, die anschließend abgearbeitet werden.

Funktionsname:

first_part_missing first_part_missing

Signatur:

first_part_missing:: dtd_parts -> elementtypelist -> boolean

dtd_parts: Regelbeschreibung, die überprüft werden soll

elementtypelist: Liste von finalen Elementen

boolean: Ergebnis der Funktion

Beschreibung:

Die Funktion stellt fest, ob in der `elementtypelist` der erste Ausdruck der Regelbeschreibung nicht vorkommt.

Implementierung:

Das Ergebnis der Funktion wird durch entsprechende Aufrufe von `nullable` und `option_may_inserted` ermittelt. Die Funktion wurde vorwiegend zur besseren Lesbarkeit in den Quelltext eingefügt.

Funktionsname:

list_reform list_reform

Signatur:

list_ reform:: dtd_list -> dtd_list

dtd_list: Liste von `dtd_element`'s

dtd_list: Liste von `dtd_element`'s mit umgestalteten `dtd_parts`'s

Beschreibung:

Die Gestaltung der Regelbeschreibungen kann der Benutzer nach seinen Vorstellungen vornehmen, solange der syntaktische Aufbau nicht verletzt wird. Semantische Überprüfungen werden leider nicht vorgenommen. Die Funktion formt sämtliche zu den `dtd_element`'s gehörigen Regelbeschreibungen um, so daß sie von den restlichen Funktionen besser verarbeitet werden können.

Implementierung:

Mit dem `dtd_parts` jedes `dtd_element`'s wird die Funktion `reform_dtd` aufgerufen, der die Regelbeschreibung umformt.

Funktionsname:

reform_dtd reform_dtd

Signatur:

reform_dtd:: dtd_parts -> dtd_parts

dtd_parts: Regelbeschreibung

dtd_parts: Umgeformte Regelbeschreibung

Beschreibung:

Die Funktion formt eine Regelbeschreibung um, damit sie von den restlichen Funktionen besser bearbeitet werden kann. Bei der Umformung sollen besonders optionale Ausdrücke (Ausdrücke auf denen der `Opt`- bzw. der `Mal`-Operator angewendet wird), die wiederum aus optionalen Teilausdrücken bestehen, durch äquivalente Ausdrücke ersetzt werden. Dieses wollen wir als Beseitigung von Idempotenzen bezeichnen.

Implementierung:

Sämtliche Ausdrücke, aus denen die Regelbeschreibung besteht, werden verarbeitet. Der Operator, der auf den ersten Ausdruck angewendet wird, bestimmt die Umformungsschritte zur Neugestaltung der Regelbeschreibung. Die Frage, ob ein Ausdruck optional ist, wird durch Aufruf der Funktion `nullable` beantwortet. Bei folgenden Operatoren sind Umformungen durchzuführen:

Mal R1 Für die n-malige Wiederholung gilt für den Fall, daß $R1$ optional ist, $R1* = R1'*$. Dabei ist $R1'$ durch Anwendung der Funktion `reform` hervorgegangen, die einen optionalen Ausdruck in einen nichtoptionalen Ausdruck verwandelt.

Opt R1 Ist der Ausdruck, auf dem der `Opt`-Operator angewendet wird, ebenfalls optional, kann der Operator beseitigt werden. Dieses geschieht durch den rekursiven Aufruf der Funktion `reform_dtd` mit dem Ausdruck $R1$.

Pos R1 Auch beim positiven Abschluß wird zunächst festgestellt, ob $R1$ optional ist. Ist dieses der Fall gilt $R1+ = R1'*$. R1' wurde aus dem Ausdruck $R1$ durch Anwendung der Funktion `reform` gebildet.

Die Operatoren, die auf mehreren Ausdrücken angewendet werden, bedürfen zunächst keiner weiteren Umformung. Durch rekursiven Aufruf der Funktion `reform_dtd` mit $R1$ wird jedoch versucht, die Ausdrücke aus denen $R1$ besteht, weiter umzuformen. Bei Ausdrücken, die nur noch aus finalen Elementen bestehen, bedarf es keiner weiteren Umformung mehr.

Funktionsname:

reform reform

Signatur:

reform:: dtd_parts -> dtd_parts

dtd_parts: Regelbeschreibung, die optional ist

dtd_parts: Regelbeschreibung, die nicht mehr optional ist

Beschreibung:

Aufgabe der Funktion ist es, einen optionalen Ausdruck in einen nichtoptionalen Ausdruck umzuwandeln. Wird der Operator der n-maligen Wiederholung auf den optionalen Ausdruck und den nichtoptionalen angewendet, so sind die entstehenden Ausdrücke äquivalent.

Implementierung:

Da eine Liste als Sequenz aufgefaßt wird, muß von der übergebenenen Regelbeschreibung nur der erste Ausdruck bearbeitet werden. Eine Sequenz ist nur dann optional, wenn alle ihre Ausdrücke optional ist. Die Verarbeitung ist beendet, wenn der übergebene Ausdruck nicht mehr optional ist, was entweder bei Ausdrücken, die nur aus einem finalen Element bestehen per Definition der Fall ist, oder durch die Funktion `nullable` festgestellt wird. Die Operatoren, die per Definition optional sind, werden durch rekuriven Aufruf der Funktion `reform` mit dem Ausdruck, auf den sie angewendet werden, beseitigt. Bei den anderen Operatoren wird durch Umformung und Umgestaltung der Ausdrücke, auf denen sie wirken, mit Hilfe der Funktionen `reform` bzw. `or_reform` versucht, ein nicht optionales Äquivalent zu finden. Dabei werden die Rechenregeln für reguläre Ausdrücke verwendet. Bsp.: Der Ausdruck $(a*, b*, c*)*$ ist äquivalent zum Ausdruck $(a|b|c)*$.

Funktionsname:

or_reform or_reform

Signatur:

or_reform:: dtd_parts -> dtd_parts

dtd_parts: Regelbeschreibung, die aus mehreren Ausdrücken besteht

dtd_parts: Regelbeschreibung, die aus mehreren Ausdrücken besteht, die alle nicht mehr optional sind

Beschreibung:

Die Funktion wandelt jeden Ausdruck in einen nicht optionalen Ausdruck um, der die in der Funktion `reform` beschriebenen Eigenschaften besitzt.

Implementierung:

Mit jedem Ausdruck aus der übergebenenen Regelbeschreibung wird die Funktion `reform` aufgerufen und die Ergebnisse zu einer Regelbeschreibung zusammengefaßt.

1.3.5 Testmodul

Um unsere Funktionen unabhängig von anderen Moduln austesten zu können, wurde das Modul `DTDTest.AS` implementiert. Es enthält einige Funktionen, die zum Austesten späterer Ergänzungen oder zur Fehlersuche von Nutzen sein können. Deshalb werden diese Funktionen hier kurz aufgeführt.
Das Modul `DTDTest.AS` ist in der jetzigen Form nicht kompilierbar, da dazu die Datenstruktur `dtd` aus dem Modul `DTD.AS` und einige Funktionen, die im Quellcode als `LOCAL` deklariert sind, öffentlich gehalten werden müssen.

Funktionsname:

bspDTD bspDTD

Signatur:

bspDTD:: (elementtypeStruct, dtd)

Beschreibung:

Diese Funktion liefert als zweites Element im Ergebnistupel eine Beispiel-DTD, die in die Funktionen zum Austesten gestopft werden kann. Das erste Element des Ergebnistupels ist eine Datenstruktur aus der IS. Erläuterungen dazu siehe Kap. 1.2.1.

Implementierung:

Die Beispiel-DTD wird durch sequentielles Aufrufen der Funktionen aus Kap. 1.3.3.2 aufgebaut.

Funktionsname:

prtDTD prtDTD

Signatur:

prtDTD:: system -> elementtypeStruct -> dtd -> system

system: Übergebenes System

elementtypeStruct: Namensregister zu den DTD-Elementen

dtd: Auszugebende DTD

system: Verändertes System

Beschreibung:

Diese Funktion gibt eine in der Datenstruktur `dtd` gehaltenen DTD auf `stdout` aus.

Implementierung:

Rekursiv wird die `dtd`-Datenstruktur in ihre Einzelteile aufgebrochen und auf `stdout` ausgegeben. Dazu werden folgende Hilfsfunktionen aufgerufen:

prtRegel Gibt eine DTD-Regel mit Regelnamen und dazugehörenden Aufbaubeschreibungen aus.

prtElemente Gibt die Aufbaubeschreibungen einer DTD-Regel aus. Dazu werden die Funktionen `prtSeq`, `prtOr`, `prtAnd` benutzt.

prtAttrDefs Gibt die Attribut-Regel mit Regelnamen und dazugehörenden Attributdefinitionen aus. aus.

prtAttrList Gibt die Attributdefinitionen zu einem DTD-Element aus.

Funktionsname:

getDTDSave getDTDSave

Signatur:

getDTDSave:: dtd -> dtd_saves

dtd: DTD-Datenstruktur

dtd_saves: Hilfsdatenstruktur zum Speichern der DTD

Beschreibung:

Extrahiert aus der DTD-Datenstruktur die in ihr gehaltenen Hilfsdatenstruktur zum Speichern der DTD.

Implementierung:

Via Patternmatching

Funktionsname:

prtDTDSave prtDTDSave

Signatur:

prtDTDSave:: system -> elementtypeStruct -> dtd_saves -> system

system: Übergebenes System

elementtypeStruct: Namensregister zu den DTD-Elementen

dtd: Auszugebende Hilfsdatenstruktur zum Speichern der DTD

system: Verändertes System

Beschreibung:

Die zum Speichern der DTD aufgebaute Hilfsdatenstruktur wird ausgegeben.

Implementierung:

Rekursiv wird die `dtd-saves`-Datenstruktur in ihre Einzelteile aufgebrochen und auf `stdout` ausgegeben. Dazu werden folgende Hilfsfunktionen aufgerufen:

prtDC Gibt das Datenfeld mit dem `dtd_content` aus der Datenstruktur `dtd_save` aus.

prtAT Gibt den Attributtype des auszugebenden `dtd_save`-Elementes aus.

Funktionsname:

read read

Signatur:

read:: system -> (string, system)

system: Übergebenes System

string: Eingelesene Zeichenkette

system: Verändertes System

Beschreibung:

Diese Funktion liest eine Zeichenkette von `stdin`.

Implementierung:

Diese Funktion benutzt eine Hilfsfunktion gleichen Names (aber anderen Argumenten).

Funktionsname:

readStrings readStrings

Signatur:

readStrings:: system -> (strings, system)

system: Übergebenes System

strings: Liste von eingelesenen Zeichenketten

system: Verändertes System

Beschreibung:

Diese Funktion liest so viele Zeichenkette von `stdin` ein, bis die Eingabe eines leeren Zeichenkette erfolgt.

Implementierung:

Benutzt dazu obige `read`-Funktion.

Funktionsname:

strsToEl strsToEl

Signatur:

strsToEl:: strings -> elementtypeStruct -> elementtypelist

strings: Liste zu konvertierender Zeichenketten

elementtypeStruct: Namensregister zu den DTD-Elementen

elementtypelist: Liste der konvertierten Zeichenketten

Beschreibung:

Diese Funktion wandelt die übergebenen Zeichenketten in die entsprechenden `elementtype`s um.

Funktionsname:

prtElTypeList prtElTypeList

Signatur:

prtElTypeList:: system -> elementtypeStruct -> elTypeList -> system

system: Übergebenes System

elementtypeStruct: Namensregister zu den DTD-Elementen

elTypeList: Auszugebende Liste von `elementtypelist`

system: Verändertes System

Beschreibung:

Gibt die übergebenen Liste von `elementtypelist`s auf `stdout` aus.

Funktionsname:

prtEles prtEles

Signatur:

prtEles:: system -> elementtypeStruct -> elementtypelist -> system

system: Übergebenes System

elementtypeStruct: Namensregister zu den DTD-Elementen

elementtypelist: Auszugebende Liste von `elementtype`

system: Verändertes System

Beschreibung:

Gibt die übergebenen Liste von `elementtype`s auf `stdout` aus.

Funktionsname:

getDTDRule getDTDRule

Signatur:

getDTDRule:: dtd -> elementtype -> dtd_parts

dtd: DTD-Datenstruktur

elementtype: DTD-Element, zu dem seine Regelbeschreibung gesucht werden soll

dtd_parts: Zum `elementtype` gehörende Regelbeschreibung

Beschreibung:

Zu dem übergebenen DTD-Element wird die ihm zugehörige Regelbeschreibung als Ergebnis des Funktionsaufrufes zurückgegeben.

Funktionsname:

testXXXXXX testXXXXXX

Signatur:

testXXXXXX:: je nach Funktion unterschiedlich

Beschreibung:

Alle Funktionen, deren Name mit `test` beginnt (z.B. `testGetRootElement`) sind zum Testen der Funktionen aus dem Modul `DTD.AS` implementiert worden. Die getestete Funktion steht dabei als Ergänzung nach dem `test` (im obigen Beispiel wird die Funktion `getRootElement` aufgerufen).

1.3.6 SCCS

Das Hilfsprogramm `sccs`, das mit jedem OpenWindows-System geliefert wird, bietet die automatische Versionshaltung von Dateien an. Dazu wird ein Archiv der zu verwaltenden Dateien angelegt, in das von Zeit zu Zeit vom Benutzer eine neue Version einzuspielen ist. Dies hat den Vorteil, daß es immer eine Sicherheitskopie vorhanden ist. Zudem kann auf ältere Versionen einer Datei zurückgegriffen werden, falls z.B. nach Fehlern gesucht werden müssen, die in älteren Versionen nicht zum Vorschein kamen. Zu jeder eingespielten Version können Kommentare beigefügt werden, die z.B. die Fortentwicklung des Quelltextes dokumentieren (wie in unserem Fall geschehen).
Unsere beiden Module `DTD.AS` und `DTDTest.AS` haben wir mit `sccs` verwaltet. Durch Aufruf von `man sccs` in einer Shell bekommt man die Anleitung zu `sccs`. Wir empfehlen, sich in `sccs` einzuarbeiten, da damit die von uns schon begonnene Archivierung der Module aktuell gehalten werden kann.

1.4 Document

Guido Frick

1.4.1 Funktionalität

Das Modul enthält die Datenstrukturen, die einerseits die Repräsentierung des logischen Dokumentes darstellen, und desweiteren für die Verwaltung der Objekte (Objekt-IDs) notwendig sind. Hier sind alle Funktionen implementiert, die diese Strukturen aufbauen, manipulieren und auslesen.
Es werden Funktionen für unterschiedliche Module bereitgestellt:

- Parser: Der Parser erhält Funktionen, um das Dokument während des Einlesens in die internen Datenstrukturen zu schreiben und um für das Speichern diese Daten aus dieser Struktur wieder auszulesen.
- Formatierer: Der Formatierer erhält Funktionen, um Informationen zu einem bestimmten Objekt (z.B. den Elementtypen) zu ermitteln und die Liste von Objekten anzufragen, aus denen ein Objekt besteht.
- Benutzungsoberfläche: Die BO kann durch Funktionen Zeichen in Objekten einfügen oder löschen, Elemente einfügen oder löschen und anhand einer Cursorposition alle möglichen Einfügestellen ermitteln. Weiter kann die Dokumentenstruktur als Graphstruktur in der Notation des Programmes *daVinci* erzeugt werden.
- DTD: Das Modul `DTD` erhält die Möglichkeit, aus der Dokumentenstruktur den Kontext von Objekten und Cursorpositionen anzufragen.
- Tests: Es werden eine Vielzahl an Funktionen bereitsgestellt, mit denen Ergebnisstrukturen auf dem Bildschirm visualisiert und somit die Korrektheit von Funktionen kontrolliert werden können.

1.4.2 Entwurf

1.4.2.1 Daten und Struktur des Dokumentes

Die Aufgabe des Moduls ist es, einerseits den eingegebenen Text in Form von Zeichen oder Referenzen zu speichern und auf Abruf bereitszustellen, andererseits aber auch den logischen Aufbau des Dokumentes zu repräsentieren. Hierzu wird zwischen finalen und nicht-finalen Objekten unterschieden, bei denen die finalen Objekte den Text und die Referenzen sowie weitere Daten beinhalten (z.B. Grafiken). Die nicht-finalen Objekte ermöglichen die Abbildung von Objekt-Beziehungen einzelner finaler sowie nicht-finaler Objekte.
Es bietet sich an, eine solche hierarchische Struktur, wie es der Aufbau eines Dokumentes ist, als Baum darzustellen. Hier wurde die Implementierung der Baumstruktur in einem Array gewählt, deren Datenelemente die Blätter und Knoten, die in diesen Elementen gespeicherten Referenzen auf andere Datenelemente die Kanten darstellen.
Ein solches Datenelement besteht aus folgenden Komponenten:

- Objekt-ID des Vorgängers (Predecessor)
- Objekt-ID des Nachfolgers (Successor)
- Objekt-ID des ersten Objektes, aus dem das Objekt besteht (`Contents`)
- Daten bei finalen Elementen

Die Objekt-ID stellt einen eindeutigen Schlüssel für Objekte dar. Die Vergabe einer solchen Objekt-ID erfolgt ebenfalls in diesem Modul. Aus praktischen Gründen dient diese ID ebenfalls als Index für das Dokumenten-Array, sodaß ein Zugriff auf ein Objekt direkt mit dessen Objekt-ID erfolgen kann. Wird auf ein Objekt verwiesen, so erfolgt dies durch seine ID. Die Objekt-ID 0 signalisiert, daß kein Objekt z.B. als Nachfolger existiert.
Die Daten können einerseits Text, aber auch Referenzen enthalten. Bei nicht-finalen, also logischen Elementen, können keine Daten vorkommen.
Durch das Speichern der o.g. Komponenten ist es möglich, die gewünschten Beziehungen zwischen Objekten darzustellen. Zur Verdeutlichung folgendes Beispiel:

Dokument = Kapitel1, Kapitel2, Kapitel3
Kapitel1 = Text1
Text1 = "Dies ist ein Beispiel"
Kapitel2 = Text2, Referenz1
Text2 = "siehe dazu auch"
Referenz1 = Referenz auf Kapitel1
Kapitel3 = Unterkapitel1, Unterkapitel2, Unterkapitel3
Unterkapitel1 = Text3
Text3 = "U.a. kann man auch Referenzen darstellen..."
usw.

`Dokument` stellt das initiale Objekt (Root-Objekt) dar. Als Baum der o.g. Struktur sehe das folgendermaßen aus:

Bezeichnung	Objekt-ID	Vorgänger	Nachfolger	besteht aus	Daten
Document	1	0	0	2	-
Kapitel1	2	1	3	5	-
Kapitel2	3	2	4	6	-
Kapitel3	4	3	0	8	-
Text1	5	2	0	0	"Dies ist ein Beispiel"
Text2	6	3	7	0	"siehe dazu auch"
Referenz1	7	6	0	0	Ref. auf 2
Unterkapitel1	8	4	9	11	-
Unterkapitel2	9	8	10	...	-
Unterkapitel3	10	9	0	...	-
Text3	11	8	0	0	"U.a kann ..."
...					

1.4.2.2 Die Verwaltung der Objekt-IDs

Desweiteren müssen neben dem reinen Dokument noch weitere Daten funktionsübergreifend festgehalten werden. Es müssen in diesem Modul eindeutige Objekt-Nummern, sog. Objekt-IDs vergeben werden. Um den Nummernkreis so eng wie möglich zu halten, werden IDs von gelöschten Elementen neu vergeben. Es muß also nicht nur die zuletzt vergebene, sondern ebenfalls alle gelöschten Nummern, die noch nicht neu vergeben wurden, gespeichert werden. Dies geschieht mithilfe eines ADT, der einerseits die nächste, neu zu vergebende Nummer beinhaltet, und einer Liste der *alten*, wieder vergebbaren Nummern.

1.4.2.3 Das Einlesen eines Dokumentes

Beim Einlesen des Quelldokumentes werden sequentiell die Elemente in die interne Dokumentenstruktur eingetragen. Die Objektbeziehungen, die sich aus der Reihenfolge ihres Auftretens und den SGML-Schlüsselworten ergeben, werden durch die sequentiellen Aufrufe der Funktionen erzeugt. Hierfür wird ein Stack zu Hilfe genommen, der funktionsübergreifend Objektidentifikationen speichert.
Objekte, die auf andere Objekte verweisen, enthalten in der Komponente `data` des Dokumenten-Arrays die Objekt-ID des referenzierten Objektes. Da dieses Objekt u.U. erst später im Dokument auftritt, kann es vorkommen, daß die Objekt-ID noch gar nicht bekannt ist. Aus diesem Grund wird für das Einlesen eine Zuordnungstabelle verwendet, in der die Referenzidentifikationen aus dem SGML-Quelldokument und die korrespondierenden Objekt-IDs gegenübergestellt werden. Nach Abschluß des Einlesevorganges werden dann die SGML-Referenzen durch diese Objekt-IDs ersetzt.

1.4.2.4 Das Speichern eines Dokumentes

Beim Auslesen des Dokumentes fordert ebenfalls das Modul `Parser` nach und nach Objekte aus der Dokumentenstruktur an. Das hat den Vorteil, daß der Parser unter Berücksichtigung von Einstellungen und Vorgaben nur die Daten aus der Struktur ausliest, die er tatsächlich benötigt (z.B. werden andere Daten beim Speichern im ASCII-Format als beim Speichern im SGML-Format benötigt). Ein solches Verfahren setzt aber voraus, daß erkannt werden kann, welche Objekte und Daten bereits ausgelesen wurden. Dieses wird funktionsübergreifend in der Dokumentenstruktur festgehalten und erfolgt wiederum mithilfe eines Stacks, sowie eines Feldes, was das zuletzt ausgelesene Attribut eines Objektes festhält.

1.4.3 Öffentliche Schnittstellen

1.4.3.1 Sorten

Sortenname:

doc doc

Signatur:

doc:: Die Signatur der Sorte doc wird nicht öffentlich exportiert. Die einzelnen Komponenten werden später bei den lokalen Sorten aufgeführt.

Beschreibung:

Alle zu speichernden Datenstrukturen werden in dem Typen `doc` gespeichert. Diese ist die Grundlage aller Funktionen, auf die das Modul `Document.AS` basiert.

Sortenname:

insertPos insertPos

Signatur:

insertPos:: iPos objID objPosition

Beschreibung:

Diese Sorte gibt eine Einfügestelle in dem logischen Dokument an. Die Position bezieht sich auf ein Bezugsobjekt und eine Stelle, wo das Einfügen erfolgen soll (`objPosition`).

Sortenname:

objPosition objPosition

Signatur:

objPosition:: behind objID | before objID | inside integer | in

Beschreibung:

Die Einfügestelle kann sich an vier verschiedenen Positionen befinden:

1. behind objID
   Es soll in das Bezugsobjekt hinter das Objekt objID eingefügt werden.
2. before objID
   Es soll in das Bezugsobjekt vor das Objekt objID eingefügt werden.
3. inside integer
   Es soll in das (finale) Objekt eingefügt werden, d.h. das Bezugsobjekt wird in zwei Teilobjekte aufgespalten, zwischen die das neue Objekt eingefügt wird. Der Integerwert gibt den Buchstaben an, hinter dem das Objekt aufgetrennt werden soll.

4. in
   Hier soll das neue Objekt in das Bezugsobjekt eingefügt werden. Dieser Fall tritt auf, falls noch kein finales Objekt vorhanden ist und in das äußerste (nicht finale) Objekt eingefügt werden soll. Sind bereits Objekte in dem Bezugsobjekt vorhanden, so wird das neue Objekt an das *Ende* angehängt. Dies tritt während des Einlesens des Dokumentes auf, bei dem die Position `in` ebenfalls verwendet wird. In allen anderen Fällen werden 1 bis 3 benutzt.

Sortenname:

insertPosList insertPosList

Signatur:

insertPosList:: [insertPos]

Beschreibung:

Bei dieser Sorte handelt es sich um eine Liste von Elementen der Sorte `insertPos`.

Sortenname:

objContext objContext

Signatur:

objContext:: cont parentElemType predElemTypeList succElemTypeList

Beschreibung:

Die Sorte stellt den Kontext einer Stelle im logischen Dokument dar. Diese besteht einerseits aus dem Elementtypen des Vaterobjektes und andererseits aus den Elementlisten der Vorgänger und der Nachfolger.

Sortenname:

parentElemType parentElemType

Signatur:

parentElemType:: (elementtype)

Beschreibung:

Bei dieser Sorte handelt es sich lediglich um ein Synonym für `elementtype`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `objContext` verwendet.

Sortenname:

predElemTypeList predElemTypeList

Signatur:

predElemTypeList:: (elementtypelist)

Beschreibung:

Bei dieser Sorte handelt es sich lediglich um ein Synonym für `elementtypelist`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `objContext` verwendet.

Sortenname:

succElemTypeList succElemTypeList

Signatur:

succElemTypeList:: (elementtypelist)

Beschreibung:

Bei dieser Sorte handelt es sich lediglich um ein Synonym für `elementtypelist`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `objContext` verwendet.

1.4.3.2 Funktionen für den Parser

Funktionsname:

mtDoc mtDoc

Signatur:

mtDoc:: doc

doc: Dokumentenstruktur

Beschreibung:

Die Funktion legt ein leeres Dokument an. Alle internen Daten des Dokumentes werden initialisiert.

Funktionsname:

nameDocument nameDocument

Signatur:

nameDocument:: doc -> string -> doc

doc: Dokumentenstruktur

string: Name des aktuellen Dokumentes

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion speichert den übergebenen String als Name des Dokumentes. Der Name entspricht dem Filenamen.

Funktionsname:

endDocument endDocument

Signatur:

endDocument:: doc -> doc

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

Beschreibung:

Durch `endDocument` wird das Ende des Einlesens eines Dokumentes angezeigt.

Implementierung:

Nachdem das Ende des Dokumentes erreicht ist, werden alle Referenzen in den Objekten umgesetzt. Beim Einlesen wurden die in dem SGML-Dokument eingetragenen REFIDs in den Objekten gespeichert. Außerdem wurden die Objekt-IDs der referenzierten Objekte in eine Tabelle eingetragen, sodaß nun eine Ersetzung auf der Grundlage dieser Tabelle REFID-ObjID erfolgen kann. Der für das Einlesen benötigte Stack sowie die Referenztabelle werden im Anschluß gelöscht.

Funktionsname:

beginElement beginElement

Signatur:

beginElement:: doc -> elementtype -> doc

doc: Dokumentenstruktur

elementtype: Typ des einzufügenden neuen Elementes.

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion fügt ein neues SGML-Element in das Dokument ein, das den übergebenen Typ besitzt.

Implementierung:

Für das neue Element wird eine neue Objekt-ID ermittelt. Es wird als Bestandteil des zuletzt geöffneten (und nicht wieder geschlossenen) Elementes gespeichert. Sind hier bereits Elemente als *Inhalt* vorhanden, so wird es an das Ende gesetzt. Alle folgenden Elemente werden nun wiederum als Bestandteil dieses Objektes betrachtet und eingetragen, bis hierfür ein `endElement` erfolgt.

Funktionsname:

endElement endElement

Signatur:

endElement:: doc -> doc

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

Beschreibung:

Durch `endElement` wird das aktuelle SGML-Element geschlossen.

Implementierung:

Durch das Schließen des aktuellen Elementes werden folgende Elemente nicht mehr als Bestandteil dieses Elementes gesehen und eingetragen.

Funktionsname:

text text

Signatur:

text:: doc -> string -> doc

doc: Dokumentenstruktur

string: In das aktuelle Element einzutragender Text

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion trägt einen übergebenden Text in das aktuell geöffnete SGML-Element ein. Dies kann nur bei terminalen Elementen erfolgen.

Funktionsname:

attribute attribute

Signatur:

attribute:: doc -> (attributetype,attributecontents) -> doc

doc: Dokumentenstruktur

attributetype: Attributetyp

attributecontents: Attributinhalt

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion `attribute` setzt für das aktuelle Element ein durch `attributetype` festgelegtes Attribut mit dem Inhalt `attributecontents`.

Implementierung:

Die beiden Argumente `attributetype` und `attributecontents` können folgende Fälle reflektieren:

1. attributetype: `refid`
   attributecontents: `id objID`
   Es wird ein SGML-REFID-Attribut eingefügt, dessen Inhalt aus der Objekt-ID besteht, die in `attributecontents` übergeben wurde (entspricht der ID des referenzierten Objektes). Gleichzeitig wird für das referenzierte Objekt ein SGML-ID-Attribut mit demselben Inhalt eingefügt.

2. attributetype: `refid`
   attributecontents: `pid string`
   Dieser Fall wird vom Parser benutzt. `string` ist eine ID, mit der ein Elementt im Quelldokument ausgezeichnet ist. Die Funktion behandelt diese ID so, daß die Beziehung zwischen referenzierendem und referenziertem Objekt erhalten bleibt. Es bleibt ihr aber überlassen, welche Art von ID dafür benutzt wird. Auf jeden Fall wird hier nur das REFID-Attribut eingefügt, da das zugehörige ID-Attribut vom Parser mit dem dritten Fall (s.u.) eingefügt wird.
3. attributetype: `id`
   attributecontents: `pid string`
   Fügt ein SGML-ID-Attribut ein. Der Inhalt muß nicht aus dem übergebenen String in attributecontents bestehen, aber er ist der Schlüssel, um die Beziehung zum referenzierenden Objekt zu beschreiben.

Funktionsname:

beginDocSave — beginDocSave

Signatur:

beginDocSave:: doc -> doc

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion initialisiert die für das Auslesen der Dokumentenstruktur erforderlichen Datenstrukturen.

Funktionsname:

endDocSave — endDocSave

Signatur:

endDocSave:: doc -> doc

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion kennzeichnet das Beenden des Auslesens der Dokumentenstruktur.

Funktionsname:

getNextElement — getNextElement

Signatur:

getNextElement:: doc -> (doc, elementtype, objID, boolean)

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

elementtype: Elementtype des aktuell ausgelesenen Elementes

objID: Eindeutige Objekt-ID des aktuell ausgelesenen Elementes

boolean: Kennzeichen, ob für das zuletzt ausgelesene Element ein Sohn-Objekt existiert

Beschreibung:

Die Funktion `getNextElement` ermittelt das nächste auszugebene Element. Grundlage ist ein aktuelles Element. Von diesem aktuellen Element liefert die Funktion den ersten, noch nicht ausgelesenen Sohn als Ergebnis zurück und setzt dieses als neues aktuelle Element. Es ist zu beachten, daß dieses Element später als bereits ausgelesen erkannt und nicht noch einmal zurückgeliefert wird. Existiert für das akuelle Element kein Sohn, so wird `False`, ansonsten `True` zurückgeliefert.

Funktionsname:

getNextAttribute getNextAttribute

Signatur:

getNextAttribute:: doc -> (doc, attributetype, attributecontents, boolean)

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

attributetype: Typ des zurückgegebenen Attributes

attributecontents: Inhalt des zurückgegebenen Attributes

boolean: Kennzeichen, ob ein angefragtes, weiteres Attribut vorhanden ist

Beschreibung:

Die Funktion liefert das nächste Attribut des aktuellen Elementes. Es wird festgehalten, welche Attribute für das aktuelle Element ausgelesen wurden.

Implementierung:

Derzeit werden zwei Attribute unterstützt:

1. attributetype: `refid`
   attributecontents: `id objID`
   Es besteht eine Referenz auf das aktuelle Objekt. Das Objekt erhält nun eine eindeutige Identifikation, auf das sich das referenzierende Objekt beziehen kann. Diese eindeutige Identifikation besteht aus der eindeutigen Objekt-ID.

2. attributetype: `id`
   attributecontents: `id objID`
   Das aktuelle Objekt enthält eine Referenz auf ein anderes Objekt. Die eindeutige Identifikation des referenzierten Objektes erfolgt über die Objekt-ID.

In der Dokumentenstruktur wird das zuletzt von dem aktuellen Element ausgelesene Attribut gespeichert. Wurde noch kein Attribut angefordert, so hat LastAttribute den Wert `none`. Es wird als nächstes, falls vorhanden, `refid` und danach `id` ausgelesen.
Ist für das aktuelle Element kein weiteres Attribut vorhanden, so wird `False`, ansonsten `True` zusammen mit dem Attribut zurückgeliefert.

Funktionsname:

getElementText getElementText

Signatur:

getElementText:: doc -> (doc, string, boolean)

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

string: Text des aktuellen Elementes

boolean: Kennzeichen, ob Text in dem aktuellen Element enthalten ist

Beschreibung:

Die Funktion ermittelt (evtl. vorhandenen) Text des aktuellen Elementes. Ist kein Text vorhanden, so wird `False`, ansonsten `True` zusammen mit dem Text zurückgeliefert.

Funktionsname:

goParentElement goParentElement

Signatur:

goParentElement:: doc -> (doc, boolean)

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

boolean: Kennzeichen, ob das aktuelle Element bereits das *Root-Objekt* ist

Beschreibung:

Die Funktion setzt das Vaterobjekt des aktuellen Elementes als neues aktuelles Element. Handelt es sich bei dem vorherigen aktuellen Element um das *Root-Objekt*, so wird `False`, ansonsten `True` zurückgeliefert.

Funktionsname:

countObjects countObjects

Signatur:

countObjects:: doc -> integer

doc: Dokumentenstruktur

integer: Anzahl der Objekte im Dokument

Beschreibung:

Die Funktion liefert die momentane Anzahl von Objekten im Dokument.

1.4.3.3 Funktionen für den Formatierer

Funktionsname:

getFirstObj getFirstObj

Signatur:

getFirstObj:: doc -> objinfo

doc: Dokumentenstruktur

objinfo: Objekt-Info des *Root-Objektes*

Beschreibung:

Die Funktion liefert die Objekt-Info des äußersten Objektes (*Root-Objekt*).

Funktionsname:

getObjectInfo getObjectInfo

Signatur:

getObjectInfo:: doc -> objID -> objinfo

doc: Dokumentenstruktur

objID: Objekt-ID

objinfo: Objekt-Info des übergebenen Objektes

Beschreibung:

Die Funktion liefert die Objekt-Info eines Objektes.

Funktionsname:

getNextTextObject getNextTextObject

Signatur:

getNextTextObject:: doc -> objID -> objinfolist

doc: Dokumentenstruktur

doc: Geänderte Dokumentenstruktur

objID: Objekt-ID

objinfolist: Liste von Objekt-Infos der *Sohn-Objekte*

Beschreibung:

Die Funktion gibt eine Liste mit Objektinformationen aller Elemente zurück , die den Inhalt des angegebenen Objektes darstellen. Es ist zu beachten, daß nur die *1. Stufe* der Inhaltselemente aufgelöst wird.

Implementierung:

Ein Element der Objektliste ist ein Tripel. Beim Aufbauen der Liste werden die Listenelemente in umgekehrter Reihenfolge zusammengestellt. Um die Reihenfolge zu aendern wird die Funktion `reverse` angewendet.

Funktionsname:

getObjData getObjData

Signatur:

getObjData:: doc -> objID -> string

doc: Dokumentenstruktur

objID: Objekt-ID

string: Text des angegebenen Objektes

Beschreibung:

Die Funktion liefert zu einer Objekt-ID den zugehörigen Text. Diese Funktion ist nur sinnvoll für finale Textobjekte. Bei allen anderen wird ein leerer String zurückgeliefert.

1.4.3.4 Funktionen für die Benutzungsoberfläche

Funktionsname:

deleteChars deleteChars

Signatur:

deleteChars:: doc -> objID -> integer -> integer -> (doc, string)

doc: Dokumentenstruktur

objID: ID des Objektes, aus dem Zeichen gelöscht werden sollen

integer: Position des erstes zu löschenden Zeichens (erstes Zeichen im Wort ist Null)

integer: Position des letzten zu löschenden Zeichens

doc: Geänderte Dokumentenstruktur

string: Aus dem Objekt gelöschter String

Beschreibung:

Die Funktion entfernt aus dem durch die Objekt-ID bezeichneten Objekt eine Zeichenkette, die durch *Start* (erster Integer-Wert) und *Ende* (zweiter Integer-Wert) beschrieben wird. Die entfernte Zeichenkette wird zurückgegeben.

Funktionsname:

insertChars insertChars

Signatur:

insertChars:: doc -> objID -> integer -> string -> doc

doc: Dokumentenstruktur

objID: ID des Objektes, in dem Zeichen eingefügt werden sollen

integer: Position, vor dem die übergebene Zeichenkette eingefügt werden soll (erstes Zeichen im Wort ist Null)

string: Zeichenkette, die in dem angegebenen Objekt eingefügt werden soll

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion fügt in dem durch die Objekt-ID bezeichneten Objekt den übergebenen String ein. Die Position, ab welcher der String eingefügt werden soll, wird als Integer übergeben.

Funktionsname:

requestInsertPosition requestInsertPosition

Signatur:

requestInsertPosition:: doc -> objID -> integer -> insertPosList

doc: Dokumentenstruktur

objID: ID des Objektes, auf dem sich der Cursor momentan befindet

integer: Position des Cursors innerhalb des Objektes

insertPosList: Liste aller möglichen Einfügestellen

Beschreibung:

Die Funktion liefert eine Liste aller möglichen Einfügestellen bei einer konkreten Cursorposition. Der Cursor steht auf einem Objekt (objID) an der Position, die als Integer übergeben wurde. Diese Position wird durch die Angabe des Buchstabens des (finalen) Objektes angegeben. Null bedeutet hier, daß der Cursor vor dem ersten Zeichen des Objektes positioniert ist.

Implementierung:

Befindet sich der Cursor am Anfang oder am Ende eines Objektes, so ist nicht immer ableitbar, was bei dem Einfügen eines logischen Elementes zu erfolgen hat. Es sind drei Fälle zu unterscheiden:

1. Cursorposition nicht vor dem ersten und nicht hinter dem letzten Buchstaben:
   Das Einfügen eines logischen Elementes hat hier das Aufspalten des unterliegenden Objektes in zwei Teilobjekte zur Folge, zwischen denen das neue Element eingefügt wird.
2. Cursorposition vor dem ersten Buchstaben:
   Das Aufspalten des finale Objektes wie in 1. kann hier entfallen, da eines der beiden Objekte leer sein würde. Hier kann es aber noch dazu kommen, daß vor dem Objekt, auf dem der Cursor steht ebenfalls ein weiteres Objekt beginnt (Bsp. [Obj1 [Obj2 [Obj3]]]. Der Cursor befindet sich am Anfang von Objekt 3. Einfügen könnte nun bedeuten, in Objekt 2 vor Objekt 3. Aber auch in Objekt 1 vor Objkt 2). Es können also mehrere Möglichkeiten des Einfügens auftreten.
3. Cursorposition hinter dem letzten Buchstaben:
   Hier gilt analoges wie bei 2., mit dem Unterschied, daß nicht der Beginn sondern das Ende der Objekte relevant ist.
4. Cursorposition in einem nichtfinalen Objekt:
   Dies kann nur an einer Stelle auftreten, falls das Dokument noch leer ist. Die mögliche Einfügestelle ist in das logische Element.

Die Funktion liefert nun alle möglichen Einfügestellen in der Form (Bezugsobjekt-ID, Position). Die Bezugsobjekt-ID gibt an, in welches Objekt eingefügt werden kann. Die Position kann folgende Werte annehmen:

1. `inside integer`
   gibt an, an welcher Stelle des Bezugsobjektes das neue Element eingefügt werden kann. Das Bezugsobjekt wird an dieser Stelle aufgespalten (Fall 1).
2. `before objID`
   gibt an, vor welchem Objekt in dem Bezugsobjekt das neue Element eingefügt werden kann (Fall 2).
3. `behind objID`
   gibt an, hinter welchem Objekt in dem Bezugsobjekt das neue Element eingefügt werden kann (Fall 3).
4. `in`
   gibt an, daß das neue Objekt direkt in das (nichtfinale) Bezugsobjekt eingefügt werden kann (und muß) (Fall 4).

Für die Ermittlung der Einfügestellen kann es nun zu vier Fällen kommen (siehe auch oben):

1. Cursorposition nicht vor dem ersten oder hinter dem letzten Buchstaben:
   Dies stellt die einzige mögliche Einfügestelle dar.
2. Cursorposition vor dem ersten Buchstaben:
   Es kann in das Vaterobjekt des Bezugsobjektes *vor* dem Bezugsobjekt eingefügt werden. Hat das Bezugsobjekt keinen *Vorgänger* in dem Vaterobjekt, beginnt das Vaterobjekt also mit dem Bezugsobjekt, so ist es ebenfalls möglich in das übergeordnete Vater-Vaterobjekt *vor* dem Vaterobjekt einzufügen usw.
3. Cursorposition hinter dem letzten Buchstaben:
   Es kann in das Vaterobjekt des Bezugsobjektes *hinter* dem Bezugsobjekt eingefügt werden. Hat das Bezugsobjekt keinen *Nachfolger* in dem Vaterobjekt, endet das Vaterobjekt also mit dem Bezugsobjekt, so ist es ebenfalls möglich in das übergeordnete Vater-Vaterobjekt *hinter* dem Vaterobjekt einzufügen usw.
4. Cursorposition in einem nichtfinalen Objekt:
   Dies kann nur an einer Stelle auftreten, falls das Dokument noch leer ist. Die mögliche Einfügestelle ist in das logische Element.

Alle möglichen Einfügestellen werden in einer Liste zurückgegeben. Beispiele einer Einfügestelle:

- (100, inside 20): in Objekt 100 an Stelle 20
- (30, before 5): in Objekt 30 vor Objekt 5
- (45, behind 20): in Objekt 45 hinter Objekt 20
- (1, in): in Objekt 1.

Funktionsname:

insertRootElement insertRootElement

Signatur:

insertRootElement:: doc -> elementtype -> doc

doc: Dokumentenstruktur

elementtype: Elementtyp des *Root-Objektes*

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion fügt das erste Element in das Dokument ein.

Implementierung:

Diese Funktion darf nur aufgerufen werden, wenn noch keine Elemente vorhanden sind. Das *Root-Element* stellt das äußerste logische Element dar (z.B. Document). Da vorher noch keine Elemente vorhanden sind, hat das Root-Objekt immer die Objekt-ID 1. Um dies auf alle Fälle zu gewährleisten, werden die Objekt-IDs neu initialisiert!!!

Funktionsname:

insertElement insertElement

Signatur:

insertElement:: doc -> insertPos -> elementtype -> (doc, objinfo)

doc: Dokumentenstruktur

insertPos: Position, wo das neue Element eingefügt werden soll

elementtype: Typ des einzufügenden Elementes

doc: Geänderte Dokumentenstruktur

objinfo: Objekt-Info des neuen Elementes (Objekt-ID, Elementtyp)

Beschreibung:

Die Funktion fügt ein logisches Element vom Typ `elementtype` in das Dokument ein.

Implementierung:

Die Position, wo das neue Element eingefügt werden soll wird durch `insertPos` spezifiziert. Neben dem Bezugsobjekt (häufig das Vaterobjekt) wird die Position in Bezug auf dieses Objekt angegeben. Hier gibt es folgende Fälle:

1. `before objID`
   Das neue Objekt wird vor das angegebene Objekt eingefügt.
2. `behind objID`
   Das neue Objekt wird hinter das angegebene Objekt eingefügt.
3. `inside integer`
   Das neue Objekt wird in das Bezugsobjekt eingefügt.

4. `in`
   Das neue Objekt wird in das Bezugsobjekt eingefügt. Sind bereits Objekte als Bestandteile des Bezugsobjektes vorhanden, so wird das neue Objekt an das Ende gesetzt.

Funktionsname:

deleteElement deleteElement

Signatur:

deleteElement:: doc -> objID -> (boolean, doc)

doc: Dokumentenstruktur

objID: ID des zu löschenden Elementes

boolean: Kennzeichen, ob das Element gelöscht werden konnte

doc: Geänderte Dokumentenstruktur

Beschreibung:

Die Funktion entfernt das angegebene Objekt aus dem Dokument. Mit UNDO kann die Löschung rückgängig gemacht werden (noch nicht implementiert).

Implementierung:

`deleteElement` entfernt das angegebene Objekt aus dem Dokument. Der Nachfolger erhält einen neuen Vorgänger und der Vorgänger entweder einen neuen Nachfolger oder ein neues Anfangsobjekt, falls das zu löschende Element das Anfangsobjekt seines Vorgängers (Vaters) war. Die ID des gelöschten Objektes sowie alle folgenden, die in dem gelöschten Objekt enthalten sind, werden freigegeben.

1.4.3.5 Funktionen für die DTD

Funktionsname:

getInsertContext getInsertContext

Signatur:

getInsertContext:: doc -> insertPos -> objContext

doc: Dokumentenstruktur

insertPos: Einfügestelle, für die der Kontext ermittelt werden soll

objContext: Kontext der Einfügestelle

Beschreibung:

Die Funktion liefert den Kontext einer konkreten Einfügeposition im Dokument. Die Stelle wird als `insertPos` angegeben. Zurückgeliefert wird die Datenstruktur `objContext`, die sowohl den Elementtypen des Vaterobjekts, als auch die der Vorgänger und Nachfolger enthält.

Implementierung:

Die Einfügestelle wird als `insertPos` spezifiziert und kann folgende Zustände enthalten:

1. `FatherID before ChildID`
   Der Kontext ergibt sich folgendermaßen:
   - `FatherID`: Die Vater-ID wird mit übergeben.
   - `PredList`: Die Liste der Vorgänger ergibt sich aus allen Vorgängern des Child-Objektes, vor dem eingefügt werden soll.
   - `SuccList`: Die Nachfolgerliste beginnt mit dem Objekt, vor dem eingefügt werden soll. Hiernach kommen alle seine Nachfolger.
2. `FatherID behind ChildID`
   Der Kontext ergibt sich folgendermaßen:
   - `FatherID`: Die Vater-ID wird mit übergeben.
   - `PredList`: Die Vorgängerliste beginnt mit dem übergebenen Child-Objekt und nachfolgend mit allen Vorgängern.
   - `SuccList`: Die Nachfolgerliste besteht aus allen Nachfolgern des Child-Objektes.
3. `FatherID in`
   Der Kontext ergibt sich folgendermaßen:
   - `FatherID`: Die Vater-ID wird mit übergeben.
   - `PredList`: Die Vorgängerliste besteht aus allen Objekten des übergebenen Vaterobjektes (i.d.R. `[]`).
   - `SuccList`: Die Nachfolgerliste ist immer leer.
4. `InID inside Position`
   Der Kontext ergibt sich folgendermaßen:

- `FatherID`: Das Vaterobjekt entspricht dem des Objektes, in das eingefügt werden soll. Dieses muß ermittelt werden.
- `PredList`: Die Liste aller Vorgänger endet mit dem Objekt, in das eingefügt werden soll. Hiervor liegen alle Vorgänger dieses Objektes.
- `SuccList`: Die Liste der Nachfolger beginnt ebenfalls mit dem Objekt, in das eingefügt werden soll.

Daß das Objekt, in das eingefügt werden soll, einerseits in der Vorgänger- als auch in der Nachfolgerliste vorhanden ist, liegt an der Aufspaltung des Objekts.
Ausnahmen: Soll vor dem ersten Buchstaben eingefügt werden (Position = 0), dann handelt es sich hierbei um den Fall `before ID`. Soll hinter dem letzten Buchstaben eingefügt werden, so handelt es sich um `behind ID`.

Funktionsname:

getObjectContext getObjectContext

Signatur:

getObjectContext:: doc -> objID -> objContext

doc: Dokumentenstruktur

objID: ID des Objektes, für das der Kontext ermittelt werden soll

objContext: Kontext des Objektes

Beschreibung:

Die Funktion liefert den Kontext eines konkreten Objektes, das mit seiner Objekt-ID spezifiziert wird. Der Kontext besteht aus den selben Komponenten, wie in `getInsertContext` beschrieben. Er ergibt sich aus dem Vaterobjekt, sowie den Vorgängern und den Nachfolgern. Das Objekt selber taucht im Kontext nicht mehr auf.

1.4.3.6 Funktionen für daVinci

Funktionsname:

getDocumentGraph getDocumentGraph

Signatur:

getDocumentGraph:: doc -> elementtypeStruct -> objID -> nodeType -> graphtrees

doc: Dokumentenstruktur

elementtypeStruct: Struktur, die die Namen der Elementtypen beinhaltet

objID: ID des zur Zeit aktiven Objektes

nodeType: Art der Hervorhebung des aktiven Objektes in *daVinci*

graphtrees: Graphenstruktur für die Visualisierung in *daVinci*

Beschreibung:

Die Funktion liefert die logische Struktur des aktuellen Dokumentes als Graphen. Die Graphen-Datenstruktur entspricht der des Graphen-Visualisierungsprogrammes *daVinci*. Das aktive Objekt im Dokument (objID) wird hervorgehoben. Die Art der Hervorhebung kann gewählt werden (nodeType):

- colored
- ellipse
- none

1.4.3.7 Funktionen für diverse Module

Funktionsname:

getParent getParent

Signatur:

getParent:: doc -> objID -> objID

doc: Dokumentenstruktur

objID: ID des Objektes, für das das Vaterobjekt gesucht ist

objID: ID des Vaterobjektes

Beschreibung:

Die Funktion liefert zu dem angegebenen Objekt die Objekt-ID des Vaterobjektes. Die Vaterobjekt-ID des *Root-Objektes* ist 0.

Implementierung:

In der Dokumentenstruktur ist der Vorgänger nicht automatisch der Vater. Dies ist nicht der Fall, falls `Contents` des Vorgängers nicht das angegebene Objekt ist. Jetzt gilt aber, daß beide Objekte den selben Vater haben. Also kann nun der Vorgänger des Vorgängers der gemeinsame Vater sein. Dies wird solange fortgeführt, bis die o.g. Bedingung erfüllt ist (`Contents` des Vorgängers = ausgehender Nachfolger). Der so ermittelte Vorgänger ist nun der Vater des ausgehenden Nachfolgers und somit auch des angegebenen Objektes.

Funktionsname:

getObjElementtype getObjElementtype

Signatur:

getObjElementtype:: doc -> objID -> elementtype

doc: Dokumentenstruktur

objID: ID des Objektes, für das der Elementtyp gesucht ist

elementtype: Elementtyp des Objektes

Beschreibung:

Die Funktion liefert zu dem angegebenen Objekt den Elementtypen.

1.4.3.8 Funktionen zum Visualisieren der Datenstrukturen für Tests

Funktionsname:

showEnvironment showEnvironment

Signatur:

showEnvironment:: system -> doc -> string -> system

system: System

doc: Dokumentenstruktur

string: Als Überschrift auszugebender Text

system: Geändertes System

Beschreibung:

Die Funktion visualisiert fast alle Komponenten der Dokumentenstruktur, den Dokumentnamen, die aktuelle Objekt-ID-Verwaltungsstruktur, aufgebaute Referenzen nach dem Einlesen, den Ein- Ausgabestack und die Struktur/Inhalt des Dokumentes.

Funktionsname:

showGraph showGraph

Signatur:

showGraph:: system -> doc -> elementtypeStruct -> system

system: System

doc: Dokumentenstruktur

elementtypeStruct: Struktur, die die Namen der Elementtypen beinhaltet

system: Geändertes System

Beschreibung:

Die Funktion visualisiert die Struktur und den Inhalt des aktuellen Dokumentes in einfacher Graphenform.

Funktionsname:

showDocument showDocument

Signatur:

showDocument:: system -> doc -> system

system: System

doc: Dokumentenstruktur

system: Geändertes System

Beschreibung:

Die Funktion visualisiert den Inhalt des aktuellen Dokumentes.

Funktionsname:

showIDs showIDs

Signatur:

showIDs:: system -> doc -> system

system: System

doc: Dokumentenstruktur

system: Geändertes System

Beschreibung:

Die Funktion visualisiert die aktuellen Struktur für die Vergabe der Objekt-IDs.

Funktionsname:

showStack showStack

Signatur:

showStack:: system -> doc -> system

system: System

doc: Dokumentenstruktur

system: Geändertes System

Beschreibung:

Die Funktion visualisiert den aktuellen Zustand des Ein- Ausgabe-Stacks.

Funktionsname:

showReferences showReferences

Signatur:

showReferences:: system -> doc -> system

system: System

doc: Dokumentenstruktur

system: Geändertes System

Beschreibung:

Die Funktion visualisiert den Inhalt der Referenztabelle, die beim Einlesen eines Dokumentes der Umsetzung von Referenzen dient.

Funktionsname:

showDocName showDocName

Signatur:

showDocName:: system -> doc -> system

system: System

doc: Dokumentenstruktur

system: Geändertes System

Beschreibung:

Die Funktion visualisiert den Namen des Dokumentes.

Funktionsname:

showObjList showObjList

Signatur:

showObjList:: system -> objinfolist -> system

system: System

objinfolist: Darzustellende Liste von Objekt-Infos

system: Geändertes System

Beschreibung:

Die Funktion visualisiert eine Liste von Objekt-Infos, die Ergebnis von verschiedenen Funktionen sein kann.

Funktionsname:

showInsertPosList showInsertPosList

Signatur:

showInsertPosList:: system -> insertPosList -> system

system: System

objinfolist: Darzustellende Liste von Einfügepositionen

system: Geändertes System

Beschreibung:

Die Funktion visualisiert eine Liste von Einfügepositionen, die Ergebnis von verschiedenen Funktionen sein kann.

Funktionsname:

showContext showContext

Signatur:

showContext:: system -> objContext -> system

system: System

objContext: Darzustellender Kontext

system: Geändertes System

Beschreibung:

Die Funktion visualisiert eine übergebene Kontextstruktur, die Ergebnis von verschiedenen Funktionen sein kann.

Funktionsname:

showNextElement showNextElement

Signatur:

showNextElement:: system -> doc -> objID -> boolean -> system

system: System

doc: Dokumentenstruktur

objID: Auszugebene Objekt-ID

boolean: Kennzeichen, ob eine Objekt-ID ausgegeben werden soll. Ist dies nicht der Fall, so erscheint ein entsprechender Hinweis

system: Geändertes System

Beschreibung:

Die Funktion visualisiert eine übergebene Objekt-ID und den aktuellen Zustand des Stacks.

Funktionsname:

showNextAttribute showNextAttribute

Signatur:

showNextAttribute:: system -> doc -> attributetype -> attributecontents -> boolean -> system

system: System

doc: Dokumentenstruktur

attributetype: Typ des auszugebenen Attributes

attributecontents: Inhalt des auszugebenen Attributes

boolean: Kennzeichen, ob ein Attribut ausgegeben werden soll. Ist dies nicht der Fall, so erscheint ein entsprechender Hinweis

system: Geändertes System

Beschreibung:

Die Funktion visualisiert ein übergebenes Attribut, den aktuellen Zustand des Stacks und das in der Dokumentenstruktur gespeicherte Kenzeichen für das zuletzt ausgelesene Attribut.

1.4.4 Lokale Implementierung

1.4.4.1 Sorten

Sortenname:

doc doc

Signatur:

```
doc:: d (
    document :: document,
    ids :: ids,
    inOutStack :: inOutStack,
    lastAttribute :: attributetype,
    references :: references,
    documentName :: documentName)
```

Beschreibung:

Alle zu speichernden Datenstrukturen werden in dem Typen `doc` gespeichert. Diese ist die Grundlage aller Funktionen, auf die das Modul `Document` basiert.
Die Komponente `lastAttribute` wird während des Auslesens der Dokumentenstruktur zum Zwecke des Speichern benötigt. Hier werden nacheinander die Attribute eines Elementes angefordert. Es muß festgehalten werden, welches Attribut für ein Element bereits ausgelesen wurde. Wurden noch keine Attribute angefordert, so enthält `lastAttribute` den Wert `none`.
Das Attribute `refid` zeigt an, daß auf das aktuelle Element verwiesen wird, `id` hingegen identifiziert das aktuelle Element als Verweis auf ein anderes Element.
Alle anderen Komponenten werden im folgenden näher erläutert.

Sortenname:

document document

Signatur:

```
document:: Array ACTUAL
SORTS
    codom = docdata.
    array = document.
OPNS
    errorval = mtEntry.
END+
```

Beschreibung:

Die Struktur des Dokumentes wird in einem Array abgebildet. Dieses besteht aus folgenden Feldern:

- Objekt-ID des Vorgängers (Predecessor)
- Objekt-ID des Nachfolgers (Successor)
- Objekt-ID des ersten Objektes, aus dem das Objekt besteht (`Contents`)

- Daten bei terminalen Elementen

Die Objekt-ID 0 signalisiert, daß kein Objekt z.B. als Nachfolger existiert. Der Index des Arrays stellt die eigene Objekt-ID dar. Die Daten können einerseits Text, aber auch Referenzen sein. Bei nichtterminalen, also logischen Elementen, können keine Daten vorkommen. Beim Einlesen des Dokumentes kann es zu Referenzierungen kommen, d.h. ein Objekt verweist auf ein anderes. Im SGML-Dokument erhält das referenzierende Objekt ein Attribut `ID` mit einer eindeutigen Referenz-ID, die das Referenzobjekt unter dem Attribut `REFID` ebenfalls enthält. Beim Einlesen wird nun diese ID unter `pref` als String gespeichert. Wird das Referenzobjekt eingelesen, so wird in einer Tabelle dieser ID nun die tatsächliche Objekt-ID zugeordnet, und nachdem das gesamte Dokument eingelesen wurde in den Referenzobjekten unter `ref` eingesetzt.

Sortenname:

predecessor predecessor

Signatur:

predecessor:: (objID)

Beschreibung:

Bei dieser Sorte handelt es sich um ein Synonym für `objID`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `docdata` verwendet und beschreibt den Vorgänger eines Objektes.

Sortenname:

successor successor

Signatur:

successor:: (objID)

Beschreibung:

Bei dieser Sorte handelt es sich um ein Synonym für `objID`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `docdata` verwendet und beschreibt den Nachfolger eines Objektes.

Sortenname:

contents contents

Signatur:

contents:: (objID)

Beschreibung:

Bei dieser Sorte handelt es sich um ein Synonym für `objID`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `docdata` verwendet und beschreibt das erste Objekt, das den Inhalt des jeweiligen Objektes bildet.

Sortenname:

data data

Signatur:

data:: nonfinal | text string | ref objID | pref string

Beschreibung:

Die Sorte enthält die Daten eines Objektes. Handelt es sich bei dem Objekt um ein nichtfinales, das aus weiteren Objekten besteht, so enthält es als Kennzeichen hierfür `nonfinal`. Stellt das Objekt eine Referenz auf ein anderes Objekt dar, so enthält es die Objekt-ID des referenzierten Objektes (`ref objID`). Während des Einlesens kann diese Objekt-ID noch nicht bekannt sein, da das Objekt, auf das verwiesen wird, noch nicht eingelesen und mit einer eindeutigen ID versehen wurde. Deshalb wird vorerst die im Source-Text vorhandene Referenz als String gespeichert (`pref string`) und später durch `ref objID` ersetzt.

Sortenname:

docdata docdata

Signatur:

docdata:: element (predecessor, successor, contents, elementtype, data)

Beschreibung:

Diese Sorte enthält alle Daten eines Objektes und bildet somit ein Array-Element des Typs `document`. Sie beinhaltet den Vorgänger, den Nachfolger, das erste Objekt, das in dem jeweiligen Objekt enthalten ist, den Elementtypen und den Objekt-Inhalt.

Sortenname:

ids ids

Signatur:

ids:: id objID objIDList

Beschreibung:

Desweiteren müssen neben dem reinen Dokument noch weitere Daten funktionsübergreifend festgehalten werden. Es müssen in diesem Modul eindeutige Objekt-Nummern, sog. Objekt-IDs vergeben werden. Um den Nummernkreis so eng wie möglich zu halten, werden IDs von gelöschten Elementen neu vergeben. Es muß also nicht nur die zuletzt vergebene, sondern ebenfalls alle gelöschten Nummern, die noch nicht neu vergeben wurden, gespeichert werden. Dies geschieht mithilfe eines ADT, der einerseits die nächste, neu zu vergebende Nummer beinhaltet, und einer Liste der *alten*, wieder vergebbaren Nummern.

Sortenname:

inOutStack inOutStack

Signatur:

inOutStack:: [objID]

Beschreibung:

Während des Einlesens des Dokumentes muß festgehalten werden, welches Element derzeit aktiv ist. Weiter müssen auch die vorher aktuellen Elemente festgehalten werden, da diese nach einem `endElement` (siehe dort) wieder zu den aktiven Elementen werden. Ein solches Verhalten wird durch einen Stack realisiert. Jedes `beginElement` schreibt, und jedes `endElement` entfernt ein Element von dem Stack.
Beim Auslesen der Dokumentenstruktur wird festgehalten, welche Objekte bereits ausgelesen wurden. Die Stackelemente müssen im Gegensatz zum Einlesen etwas anders interpretiert werden: Das oberste Objekt stellt das zuletzt ausgelesene Sohnobjekt des aktuellen Elementes, das zweite Element das aktuelle Element selber dar. Die Handhabung dieser Interpretation ist den Kommentaren der entsprechenden Funktionen zu entnehmen.

Sortenname:

reference reference

Signatur:

reference:: refEntry refID objID

Beschreibung:

Das einzulesende SGML-Dokument kann Referenzen beinhalten. Hierzu kann einem Element eine REFID als Attribut gesetzt werden, auf das sich andere Elemente als REF beziehen. Es bietet sich an, diese REFID mit der Objekt-ID gleichzusetzen. Das bedeutet jedoch, daß während des Einlesens die REF-ID des ursprünglichen Dokumentes in die neue Objekt-ID, die mit Sicherheit nicht der REFID entspricht, umgesetzt werden muß. Bezieht sich nun ein Element mittels REF-Attribut auf ein solches Element, so wird stattdessen die tatsächlich Objekt-ID des referenzierten Objektes eingetragen.
Hier kann es nun dazu kommen, daß bevor das referenzierte Objekt eingelesen wurde und somit ein Eintrag in die Tabelle REFID-ObjID erfolgt ist, sich ein Element mittels REF darauf bezieht. Dieses Problem wird dadurch gelöst, indem obwohl das referenzierte Objekt noch nicht erzeugt wurde, hierfür trotzdem schon eine Objekt-ID vergeben wird. Wird dieses nun tatsächlich eingelesen, so ist durch die REFID erkennbar, ob sich bereits ein anderes Element hierauf bezogen hat und bereits eine Objekt-ID vergeben wurde. Ist das der Fall, so wird keine neue, sondern diese schon ermittelte ID eingesetzt.

Sortenname:

references references

Signatur:

references:: [reference]

Beschreibung:

Die Sorte ist eine Liste von `reference`.

Sortenname:

refID refID

Signatur:

refID:: (string)

Beschreibung:

Bei dieser Sorte handelt es sich um ein Synonym für `string`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `reference` verwendet.

Sortenname:

documentName documentName

Signatur:

documentName:: (string)

Beschreibung:

Bei dieser Sorte handelt es sich um ein Synonym für `string`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `doc` verwendet.
Das Dokument erhält selbstverständlich einen Namen, der zu speichern ist. Dies geschieht in dem Dokument selber. Der Name wird durch den Aufruf von `newDocument` gesetzt.

1.4.4.2 Elementare Zugriffsfunktionen auf die Dokumentenstruktur

Hier sind elementare Funktionen enthalten, die die einzelnen Komponenten aus einem Datenelement des Dokumenten-Arrays auslesen: den Predecessor, Successor, Contents, Elementtype und Data.

Funktionsname:

getDocData getDocData

Signatur:

getDocData:: document -> objID -> docdata

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

docdata: Datenelement des angegebenen Objektes

Beschreibung:

Die Funktion liefert das gesamte Datenelement des angegeben Objektes aus der Dokumentenstruktur (siehe auch Sorte `docdata`).

Funktionsname:

getPredecessor getPredecessor

Signatur:

getPredecessor:: document -> objID -> objID

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

objID: Vorgänger des angegebenen Elementes

Beschreibung:

Die Funktion liefert den Vorgänger des angegeben Objektes aus der Dokumentenstruktur.

Funktionsname:

getPredecessor2 getPredecessor2

Signatur:

getPredecessor2:: docdata -> objID

docdata: Datenelement aus dem der Vorgänger ausgelesen werden soll

objID: Vorgänger aus dem angegebenen Datenelement

Beschreibung:

Die Funktion liefert den Vorgänger aus einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getPredecessor` verwendet.

Funktionsname:

getSuccessor getSuccessor

Signatur:

getSuccessor:: document -> objID -> objID

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

objID: Nachfolger des angegebenen Elementes

Beschreibung:

Die Funktion liefert den Nachfolger des angegeben Objektes aus der Dokumentenstruktur.

Funktionsname:

getSuccessor2 getSuccessor2

Signatur:

getSuccessor2:: docdata -> objID

docdata: Datenelement aus dem der Nachfolger ausgelesen werden soll

objID: Nachfolger aus dem angegebenen Datenelement

Beschreibung:

Die Funktion liefert den Nachfolger aus einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getSuccessor` verwendet.

Funktionsname:

getContents getContents

Signatur:

getContents:: document -> objID -> objID document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

objID: Erstes Objekt, das in dem angegebenen Element enthalten ist

Beschreibung:

Die Funktion liefert das erste Objekt, das den Inhalt des angegebenen Objektes darstellt.

Funktionsname:

getContents2 getContents2

Signatur:

getContents2:: docdata -> objID

docdata: Datenelement aus dem das erste enthaltene Objekt ausgelesen werden soll

objID: Erstes Objekt, das den Inhalt aus dem angegebenen Datenelement darstellt

Beschreibung:

Die Funktion liefert den *Inhalt* aus einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getContents` verwendet.

Funktionsname:

getElementtype getElementtype

Signatur:

getElementtype:: document -> objID -> elementtype

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

elementtype: Elementtyp des angegebenen Elementes

Beschreibung:

Die Funktion liefert den Elementtypen des angegeben Objektes aus der Dokumentenstruktur.

Funktionsname:

getElementtype2 getElementtype2

Signatur:

getElementtype2:: docdata -> elementtype

docdata: Datenelement aus dem der Elementtyp ausgelesen werden soll

elementtype: Elementtyp aus dem angegebenen Datenelement

Beschreibung:

Die Funktion liefert den Elementtyp aus einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getElementtype` verwendet.

Funktionsname:

getData getData

Signatur:

getData:: document -> objID -> data

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

data: Daten des angegebenen Elementes

Beschreibung:

Die Funktion liefert die Daten des angegeben Objektes aus der Dokumentenstruktur.

Funktionsname:

getData2 getData2

Signatur:

getData2:: docdata -> data

docdata: Datenelement aus dem die Daten ausgelesen werden sollen

data: Daten aus dem angegebenen Datenelement

Beschreibung:

Die Funktion liefert die Daten aus einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getData` verwendet.

Funktionsname:

getText getText

Signatur:

getText:: document -> objID -> string

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

string: Text des angegebenen Objektes

Beschreibung:

Die Funktion liefert den Text eines finalen Textelementes. Handelt es sich bei dem angegebenen Objekt nicht um ein Textobjekt, so wird ein leerer String zurückgeliefert.

Funktionsname:

getText2 getText2

Signatur:

getText2:: docdata -> string

docdata: Datenelement aus dem der Text ausgelesen werden sollen

string: Text aus dem angegebenen Datenelement

Beschreibung:

Die Funktion liefert den Text aus einem Datenelement. Handelt es sich bei dem angegebenen Objekt nicht um ein Textobjekt, so wird ein leerer String zurückgeliefert.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getText` verwendet.

Funktionsname:

getLength getLength

Signatur:

getLength:: document -> objID -> integer

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

integer: Anzahl der Buchstaben eines finalen Objektes

Beschreibung:

Die Funktion ermittelt die Anzahl der Buchstaben in einem finalen Objekt. Die Länge des Textes bei nichtfinalen Objekten ist in allen Fällen 0.

Funktionsname:

getLength2 getLength2

Signatur:

getLength2:: docdata -> integer

docdata: Datenelement aus dem der Text ausgelesen werden sollen

integer: Anzahl der Buchstaben des angegebenen Datenelementes

Beschreibung:

Die Funktion liefert die Anzahl Buchstaben eines Datenelementes. Die Länge nicht-finaler Objekte ist immer 0.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getLength` verwendet.

Funktionsname:

getRefID getRefID

Signatur:

getRefID:: document -> objID -> objID

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

objID: ID eines referenzierten Objektes

Beschreibung:

Die Funktion liefert die Objekt-ID eines referenzierten Objektes. Enthält das aktuelle Objekt keine Referenz, so wird `mtObjID` zurückgeliefert.

Funktionsname:

getRefID2 getRefID2

Signatur:

getRefID2:: data -> objID

data: Daten eines Objektes, aus dem ein referenzirtes Objekt ausgelesen werden soll objID: ID eines referenzierten Objektes

Beschreibung:

Die Funktion liefert aus den Daten eines Objektes die ID eines Referenzobjektes. Enthält das angegebene Objekt keine Referenz, so wird `mtObjID` zurückgeliefert.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `getRefID` verwendet.

Funktionsname:

isFirstObj isFirstObj

Signatur:

isFirstObj:: document -> objID -> boolean

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

boolean: Kennzeichen, ob es sich bei dem angegebenen Objekt um das erste eines logischen Elementes handelt

Beschreibung:

Die Funktion prüft, ob das angegeben Objekt das erste Objekt in einem logischen Objekt ist. Ist es das erste, so besitzt es keinen weiteren Vorgänger mehr (bzw. der Vorgänger ist das Vaterobjekt).

1.4.4.3 Elementare Manipulierungsfunktionen des Dokumenten-Arrays

Hier sind die Funktionen enthalten, die die einzelnen Komponenten eines Datenelementes des Dokumenten-Arrays setzen. Der vorherige Inhalt der jeweiligen Teilkomponente geht verloren. Weiter sind hier die Initialisierungsfunktionen für ein Datenelement sowie das gesamte Dokumenten-Array enthalten.

Funktionsname:

mtData mtData

Signatur:

mtData:: data

data: Initiales Datenfeld

Beschreibung:

Die Funktionen liefert als initiales Datenfeld `nonfinal`.

Funktionsname:

mtEntry mtEntry

Signatur:

mtEntry:: docdata

docdata: Initiales Datenelement des Dokument-Arrays

Beschreibung:

Die Funktionen liefert einen initiales Datenelement des Dokumenten-Arrays.

Funktionsname:

mtDocument mtDocument

Signatur:

mtDocument:: document

document: Initiales Dokumenten-Array

Beschreibung:

Die Funktionen liefert ein initiales Dokumenten-Array.

Vor- und Nachbedingungen:

Derzeit werden nur 799 Objekte unterstützt. Diese Zahl ist willkürlich und kann ohne weiteres erhöht werden.

Funktionsname:

mtDocumentName mtDocumentName

Signatur:

mtDocumentName:: documentName

documentName: Name des Dokumentes

Beschreibung:

Die Funktion initialisiert den Namen des Dokumentes und setzt ihn auf `noname`.

Funktionsname:

setDocData setDocData

Signatur:

setDocData:: document -> objID -> docdata -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des zu ändernden Objektes

docdata: Einzutragendes Datenelement

document: Geändertes Dokument

Beschreibung:

Die Funktion setzt das gesamte Datenelement des angegebenen Objektes auf den übergebenen Inhalt.

Funktionsname:

setPredecessor setPredecessor

Signatur:

setPredecessor:: document -> objID -> objID -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des zu ändernden Objektes

objID: Einzutragender Vorgänger

document: Geändertes Dokument

Beschreibung:

Die Funktion setzt den Vorgänger des angegebenen Objektes auf die übergebene Objekt-ID.

Funktionsname:

setPredecessor2 setPredecessor2

Signatur:

setPredecessor2:: docdata -> objID -> docdata

docdata: Datenelement in dem der Vorgänger gesetzt werden soll

objID: ID des einzutragenden Vorgängers

docdata: Geändertes Datenelement

Beschreibung:

Die Funktion setzt den Vorgänger in einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `setPredecessor` verwendet.

Funktionsname:

setSuccessor setSuccessor

Signatur:

setSuccessor:: document -> objID -> objID -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des zu ändernden Objektes

objID: Einzutragender Nachfolger

document: Geändertes Dokument

Beschreibung:

Die Funktion setzt den Nachfolger des angegebenen Objektes auf die übergebene Objekt-ID.

Funktionsname:

setSuccessor2 setSuccessor2

Signatur:

setSuccessor2:: docdata -> objID -> docdata

docdata: Datenelement in dem der Nachfolger gesetzt werden soll objID: ID des einzutragenden Nachfolgers

docdata: Geändertes Datenelement

Beschreibung:

Die Funktion setzt den Nachfolger in einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `setSuccessor` verwendet.

Funktionsname:

setContents setContents

Signatur:

setContents:: document -> objID -> objID -> document

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

objID: ID des ersten Objektes, das in dem angegebenen Element enthalten ist

document: Geändertes Dokument

Beschreibung:

Die Funktion setzt die Objekt-ID, die das erste Element des Inhalts des angegebenen Elementes darstellt auf die übergebene Objekt-ID.

Funktionsname:

getContents2 getContents2

Signatur:

getContents2:: docdata -> objID -> docdata

docdata: Datenelement in dem das erste enthaltene Objekt gesetzt werden soll

objID: ID des ersten Objektes, das in dem angegebenen Element enthalten ist

Beschreibung:

Die Funktion setzt den *Inhalt* in einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `setContents` verwendet.

Funktionsname:

setElementType setElementType

Signatur:

setElementType:: document -> objID -> elementtype -> document

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

elementtype: Einzutragender Elementtyp des angegebenen Objektes

document: Geändertes Dokument

Beschreibung:

Die Funktion setzt in dem angegebenen Element den Typen.

Funktionsname:

setElementType2 setElementType2

Signatur:

setElementType2:: docdata -> elementtype -> docdata

docdata: Datenelement in dem der Elementtyp gesetzt werden soll

elementtype: Elementtyp des angegebenen Datenelementes

docdata: Geändertes Datenelement

Beschreibung:

Die Funktion setzt den Elementtyp in einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `setElementType` verwendet.

Funktionsname:

setData setData

Signatur:

setData:: document -> objID -> data -> document

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

data: Einzutragende Daten in dem angegebenen Element document: Geändertes Dokument

Beschreibung:

Die Funktion setzt die Daten des angegeben Objektes.

Funktionsname:

setData2 setData2

Signatur:

setData2:: docdata -> data -> docdata

docdata: Datenelement in dem die Daten gesetzt werden sollen

data: Daten des angegebenen Datenelementes

Beschreibung:

Die Funktion setzt die Daten in einem Datenelement.

Abhängigkeiten:

Die Hilfsfunktion wird nur von `setData` verwendet.

1.4.4.4 Höhere Zugriffsfunktionen auf die Dokumentenstruktur

Funktionsname:

getRoot getRoot

Signatur:

getRoot:: document -> objID

document: Dokument aus der Dokumentenstruktur

objID: ID des *Root-Objektes*

Beschreibung:

Ermittelt die Objekt-ID des äußersten Objektes (*Root-Objekt*).

Implementierung:

Durch die Strategie der Objekt-ID-Vergabe ergibt sich für dieses Objekt immer die ID 1. Für den Fall, daß sich die Vergabe der IDs ändert, wird aus Gründen der Erweiterbarkeit eine derartige Funktion bereitgestellt.

Funktionsname:

getParent2 getParent2

Signatur:

getParent2:: document -> objID -> objID

document: Dokument aus der Dokumentenstruktur

objID: Objekt-ID

objID: ID des Vaterobjektes

Beschreibung:

Die Funktion ermittelt den Vater eines angegebenen Objektes.

Abhängigkeiten:

Die Hilfsfunktion wird nur von der öffentlichen Funktion `getParent` verwendet. Weitere Informationen siehe dort (Seite 120).

1.4.4.5 Höhere Manipulierungsfunktionen des Dokumenten-Arrays

Funktionsname:

insertSuccessor insertSuccessor

Signatur:

insertSuccessor:: document -> objID -> objID -> elementtype -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, hinter dem das neue Objekt angehängt werden soll (*Vorgänger*)

objID: ID des Objektes, daß angefügt werden soll (*Nachfolger*)

elementtype: Elementtyp des neuen Elementes

document: Geändertes Dokument

Beschreibung:

Die Funktion fügt ein Objekt hinter ein angegebenes Objekt.

Implementierung:

Hat das Objekt, hinter dem das neue Objekt als Nachfolger eingetragen werden soll, bereits einen Nachfolger, so wird das neue Objekt hinter diesen Nachfolger angehängt. Dieses wird solange wiederholt, bis eines der Nachfolger-Objekte selber keinen Nachfolger mehr hat.
Als Nachfolger des betroffenen Objektes wird das neue Objekt eingetragen. Bei dem neuen Objekt wird entsprechend der Vorgänger gesetzt, der übergebene Elementtyp eingetragen, der eigene Nachfolger `successor` und das erste *Inhaltsobjekt* (`contents`) initialisiert (mtObjID). Als Dateninhalt wird `nonfinal` gesetzt.
Diese Funktion wird während des Einlesens eines Dokumentes verwendet.

Funktionsname:

insertContents insertContents

Signatur:

insertContents:: document -> objID -> objID -> elementtype -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, in das das neue Objekt eingefügt werden soll (*Vaterobjekt*)

objID: ID des Objektes, das eingefügt werden soll (*Kind- bzw. Sohnobjekt*)

elementtype: Elementtyp des neuen Elementes

document: Geändertes Dokument

Beschreibung:

Die Funktion trägt ein Objekt als Bestandteil eines anderen Objektes ein. D.h. das neue Objekt ist in dem bereits vorhandenen Objekt enthalten.

Implementierung:

Besitzt das Objekt, in dem das *Kindobjekt* enthalten sein soll, bereits ein enthaltenes Objekt, so wird das neue Objekt hinter das enthaltene Objekt angefügt (siehe `insertSuccessor`). Ist es jedoch das erste Objekt, das in dem *Vaterobjekt* enthalten ist, so wird bei dem Vaterobjekt `contents` auf das neue *Kindobjekt* gesetzt, beim Kindobjekt wird der Vorgänger auf das Vaterobjekt gesetzt, der Nachfolger und das erste *Inhaltsobjekt* initialisiert. Als Dateninhalt wird `nonfinal` und der übergebene Elementtyp gesetzt.
Diese Funktion wird während des Einlesens eines Dokumentes verwendet.

Funktionsname:

insertText insertText

Signatur:

insertText:: document -> objID -> string -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, in das der Text eingetragen werden soll

string: Einzutragender Text

document: Geändertes Dokument

Beschreibung:

Die Funktion setzt einen übergenen Text in einem Objekt ein.

Funktionsname:

insertPRef insertPRef

Signatur:

inserPRef:: document -> objID -> string -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, in das die Referenz eingtragen werden soll

string: Einzutragende Referenz

document: Geändertes Dokument

Beschreibung:

Die Funktion fügt eine Referenz in einem Objekt ein.

Implementierung:

Bei der Referenz handelt es sich um eine SGML-REFID, die eine beliebige Zeichenkette umfaßt. Diese Referenz wird später durch ein `ref` ersetzt, was der Objekt-ID des referenzierten Objektes darstellt. Für den Einlesevorgang muß aber vorerst die im SGML-Quelltextes eingetragene Referenz verwendet werden.

Funktionsname:

insertRef insertRef

Signatur:

inserRef:: document -> objID -> objID -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, in das die Referenz eingetragen werden soll

string: ID des Objektes, auf das referenziert werden soll

document: Geändertes Dokument

Beschreibung:

Die Funktion fügt eine Referenz in einem Objekt ein. Bei dieser Referenz handelt es sich nun um die Objekt-ID (siehe auch `insertPRef`).

Funktionsname:

insertAttribute insertAttribute

Signatur:

insertAttribute:: document -> references -> objID -> (attributetype, attributecontents) -> (document, references)

document: Dokument aus der Dokumentenstruktur

references: Referenz-Umsetztabelle bestehend aus REFID und ObjID

objID: ID des Objektes, in das das Attribut eingtragen werden soll

attributetype: Typ des Attributes (bisher nur REFID und ID)

attributecontents: Einzutragender Attributinhalt

references: Geänderte Referenzen-Umsetztabelle

document: Geändertes Dokument

Beschreibung:

Fügt das übergebene Attribut in das angegbene Objekt ein.

Implementierung:

Handelt es sich bei dem Attribut um ein `REF`-Attribut, so wird in der Zuordnungstabelle `REFID-ObjID` unter der `REFID` die tatsächliche Objekt-ID des aktiven Objektes eingetragen. Die `REFID` des SGML-Dokumentes wurde als `attributecontents` mit übergeben. Bei dem Attribut `ID` wird in dem aktiven Objekt, das auf ein anderes Objekt verweist, die `REFID` in `pref` eingetragen. Die Referenz-ID wurde als `attributecontents` übergeben. Nachdem das Dokument eingelesen wurde, werden alle Referenzen in `pref` bei den referenzierenden Objekten durch `ref` und der tatsächlichen Objekt-ID ersetzt. Die Ersetzung erfolgt auf der Grundlage der aufgebauten Tabelle `REFID-ObjID`.

Funktionsname:

getAContents getAContents

Signatur:

getAContents:: attributecontents -> string

attributecontents: Attributinhalt

string: Referenz als Zeichenkette

Beschreibung:

Die Funktion liefert eine enthaltene Referenz aus einem Attributinhalt als String. Bei der Referenz handelt es sich um die aus dem eingelesenen Quelldokument. Handelt es sich bei dem Attribut um keine Referenz, so wird `mt` zurückgeliefert.

Abhängigkeiten:

Diese Funktion wird nur von `insertAttribut` verwendet.

Funktionsname:

substRef substRef

Signatur:

substRef:: document -> references -> integer -> document

document: Dokument aus der Dokumentenstruktur

references: Referenz-Umsetztabelle bestehend aus REFID und ObjID

integer: Anzahl der noch zu durchlaufenden Objekte

document: Geändertes Dokument

Beschreibung:

Die Funktion ersetzt alle Referenzen in den Objekten nach dem Einlesen des Dokumentes.

Abhängigkeiten:

Diese Funktion wird von `endDocument` aufgerufen.

Implementierung:

Während des Einlesens werden Verweise auf Objekte mithilfe der `REFID` des SGML-Dokumentes gespeichert. Nach Abschluß werden diese Referenzen (`pref`) durch die Objekt-IDs der referenzierten Objekte ersetzt (`ref`). Dies wird für alle Objekte des Dokumentes durchgeführt. Die Anzahl der noch zu durchlaufenden Objekte wird als `integer` übergeben.

Funktionsname:

substRefObj substRefObj

Signatur:

substRefObj:: document -> objID -> references -> data -> document

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, bei dem die Referenz ersetzt werden soll

references: Referenz-Umsetztabelle bestehend aus REFID und ObjID

data: Daten des zu ändernden Objektes (`pref REFID`)

document: Geändertes Dokument

Beschreibung:

Die Funktion ersetzt bei dem übergebenen Objekt die Referenz.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `substRef` aufgerufen.

1.4.4.6 Funktionen zum Verwalten der Objekt-IDs

Funktionsname:

mtObjID mtObjID

Signatur:

mtObjID:: objID

objID: Leere Objekt-ID

Beschreibung:

Die Funktion liefert eine leere Objekt-ID (0).

Funktionsname:

mtIDs mtIDs

Signatur:

mtIDs:: ids

ids: Initialisierte Struktur zur Vergabe von Objekt-IDs

Beschreibung:

Die Funktion liefert eine initialisierte Struktur zur Vergabe der Objekt-IDs. Die nächste ID ist 1, die Liste neuzuvergebender IDs ist leer.

Funktionsname:

getNewID getNewID

Signatur:

getNewID:: ids -> (objID, ids)

ids: Struktur zur Vergabe von Objekt-IDs

objID: Neu vergebene Objekt-ID

ids: Geänderte Objekt-ID-Struktur

Beschreibung:

Die Funktion ermittelt eine neue Objekt-ID, die neu erzeugte Objekte erhalten.

Vor- und Nachbedingungen:

Es ist in allen Fällen sichergestellt, daß das *Root-Objekt* die Objekt-ID 1 erhält.

Implementierung:

Wird ein Objekt gelöscht, so wird die ID dieses Objektes festgehalten und geht nicht verloren. Bei der Ermittlung einer neuen Objekt-ID werden zuerst die Nummern vergeben, die von gelöschten alten Objekten bereits verwendet wurden. Erst wenn es keine solche ID mehr gibt, wird die nächste fortlaufende ID vergeben.

Funktionsname:

getBorderID getBorderID

Signatur:

getBorderID:: ids -> objID

ids: Struktur zur Vergabe von Objekt-IDs

objID: Zur Zeit größte Objekt-ID

Beschreibung:

Die Funktion liefert die zur Zeit größte, im Dokument gültige Objekt-ID.

Funktionsname:

getReleasedList getReleasedList

Signatur:

getReleasedList:: ids -> objIDList

ids: Struktur zur Vergabe von Objekt-IDs

objIDList: Liste aller neu zu vergebener Objekt-IDs

Beschreibung:

Die Funktion liefert die Liste aller Objekt-IDs, die von gelöschten Objekten verwendet wurden und nun neu vergeben werden können.

Funktionsname:

releaseID releaseID

Signatur:

releaseID:: objID -> ids -> ids

objID: Wieder freizugebende Objekt-ID

ids: Struktur zur Vergabe von Objekt-IDs

ids: Geänderte Objekt-ID-Struktur

Beschreibung:

Die Funktion gibt die ID eines gelöschten Objektes wieder frei. Diese kann für neue Objekte wieder vergeben werden.

1.4.4.7 Funktionen zum Verwalten des Stacks

Funktionsname:

mtInOutStack mtInOutStack

Signatur:

mtInOutStack:: inOutStack

inOutStack: Ein-Ausgabe-Stack

Beschreibung:

Die Funktion liefert einen leeren Stack.

Funktionsname:

mtAttribute mtAttribute

Signatur:

mtAttribute:: attributetype

attributetype: Leeres Attribut (none)

Beschreibung:

Die Funktion liefert ein leeres Attribut.

Funktionsname:

putStack putStack

Signatur:

putStack:: inOutStack -> objID -> inOutStack

inOutStack: Ein-Ausgabe-Stack

objID: Objekt-ID, die auf dem Stack gespeichert werden soll

inOutStack: Geänderter Ein-Ausgabe-Stack

Beschreibung:

Die Funktion legt eine übergebene Objekt-ID auf den Stack.

Funktionsname:

pushStack pushStack

Signatur:

pushStack:: inOutStack -> inOutStack

inOutStack: Ein-Ausgabe-Stack

inOutStack: Geänderter Ein-Ausgabe-Stack

Beschreibung:

Die Funktion entfernt das oberste Element aus dem Stack.

Funktionsname:

readStack readStack

Signatur:

readStack:: inOutStack -> objID

inOutStack: Ein-Ausgabe-Stack

objID: Oberste Objekt-ID

Beschreibung:

Die Funktion liest das oberste Element aus dem Stack.

Funktionsname:

read2Stack read2Stack

Signatur:

read2Stack:: inOutStack -> objID

inOutStack: Ein-Ausgabe-Stack

objID: Zweite Objekt-ID

Beschreibung:

Die Funktion entfernt das oberste Element und liest dann das neue oberste Element aus dem Stack. Diese Funktion wird für das Speichern des Dokumentes benötigt (siehe dort).

1.4.4.8 Funktionen zum Verwalten der Referenz-Tabelle beim Einlesen

Funktionsname:

mtReferences mtReferences

Signatur:

mtReferences:: references

references: Initialisierte Referenz-Tabelle (REFID-ObjID)

Beschreibung:

Die Funktion liefert eine leere Referenz-Tabelle.

Funktionsname:

writeReference writeReference

Signatur:

writeReference:: references -> refID -> objID -> references

references: Referenz-Tabelle (REFID-ObjID)

refID: Referenz-ID aus dem Quelldokument

objID: Korrespondierende Objekt-ID

references: Geänderte Referenz-Tabelle (REFID-ObjID)

Beschreibung:

Die Funktion fügt eine Referenz REFID-ObjID in die Referenz-Tabelle ein.

Funktionsname:

readReference readReference

Signatur:

readReference:: references -> refID -> objID

references: Referenz-Tabelle (REFID-ObjID)

refID: Referenz-ID aus dem Quelldokument

objID: Korrespondierende Objekt-ID

Beschreibung:

Die Funktion ermittelt anhand einer REFID die korrespondierende Objekt-ID.

1.4.4.9 Hilfsfunktionen für die Schnittstelle zum Formatierer

Hier sind die Hilfsfunktionen enthalten, die für die Implementierung der Öffentlichen Schnittstelle zum Formatierer erforderlich sind.

Funktionsname:

nextInList nextInList

Signatur:

nextInList:: document -> objinfolist -> objID -> objinfolist

document: Dokument aus der Dokumentenstruktur

objinfolist: Eingehende Liste von Objekt-Infos

objID: ID des Objektes, dessen Nachfolger an die übergene Liste als Objekt-Info angehängt werden soll

objinfolist: Erweiterte Liste von Objekt-Infos

Beschreibung:

Die Funktion hängt den Nachfolger eines Objektes an eine übergebene Liste von `objinfo` an.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `getNextTextObject` verwendet.

Implementierung:

Durch den rekursiven Aufruf werden alle Nachfolger eines Objektes an die Liste angehängt.

Funktionsname:

getFinalState getFinalState

Signatur:

getFinalState:: document -> objID -> finalstate

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, das auf final/nichtfinal geprüft werden soll

finalstate: Kennzeichen, ob das angegebene Objekt final oder nicht-final ist

Beschreibung:

Die Funktion liefert aus dem Dateninhalt eines Objektes die Information, ob es sich hierbei um ein finales oder nicht finales Objekt handelt.

1.4.4.10 Hilfsfunktionen für die Schnittstelle zur Benutzungsoberfläche

Hier sind die Hilfsfunktionen enthalten, die für die Implementierung der Öffentlichen Schnittstelle zur Benutzungsoberfläche erforderlich sind.

Funktionsname:

deleteChars2 deleteChars2

Signatur:

deleteChars2:: string -> integer -> integer -> string -> (string, string)

string: Zeichenkette, aus der Zeichen gelöscht werden sollen

integer: Position des erstes zu löschenden Zeichens (erstes Zeichen im Wort ist Null)

integer: Position des letzten zu löschenden Zeichens

string: Bereits gelöschte Zeichen (initial `mt`).

string: Zeichenkette, aus der die angegebenen Zeichen gelöscht wurden

string: Aus dem Objekt gelöschte Zeichenkette

Beschreibung:

Die Funktion entfernt aus einer Zeichenkette Zeichen, die durch *Start* (erster Integer-Wert) und *Ende* (zweiter Integer-Wert) beschrieben wird. Die entfernte Zeichenkette wird zurückgegeben.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `deleteChars` verwendet.

Funktionsname:

mergeString mergeString

Signatur:

mergeString:: string -> integer -> string -> string

string: Zeichenkette, in die Zeichen eingefügt werden soll

integer: Position, vor dem die Zeichen eingefügt werden sollen (erstes Zeichen im Wort ist Null)

string: Einzufügende Zeichenkette

string: Zeichenkette, in der die Zeichen eingefügt wurden

Beschreibung:

Die Funktion fügt eine Zeichenkette in einen String ab einer vorgegebenen Position ein.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `insertChars` verwendet.

Funktionsname:

reqInsIn reqInsIn

Signatur:

reqInsIn:: objID -> insertPosList

objID: Objekt-ID der Einfügestelle

insertPosList: Einelementige Liste von Einfügestellen

Beschreibung:

Die Funktion schreibt die Einfügestelle in die Positionsliste, die sich ergibt, wenn der Cursor in einem leeren Dokument steht.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `requestInsertPosition` verwendet.

Funktionsname:

reqInsInside reqInsInside

Signatur:

reqInsInside:: objID -> integer -> insertPosList

objID: Objekt-ID der Einfügestelle

integer: Position in dem Objekt, in das eingefügt werden kann

insertPosList: Einelementige Liste von Einfügestellen

Beschreibung:

Die Funktion schreibt die Einfügestelle in die Positionsliste, die sich ergibt, wenn der Cursor mitten in einem finalen Objekt steht.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `requestInsertPosition` verwendet.

Funktionsname:

reqInsBefore reqInsBefore

Signatur:

reqInsBefore:: document -> objID -> objID -> insertPosList -> insertPosList

document: Dokument aus der Dokumentenstruktur

objID: ID des Vaterobjektes

objID: ID des Objektes, vor dem eingefügt werden soll

insertPosList: Liste von bereits möglichen Einfügestellen (initial `mt`)

insertPosList: Liste aller möglichen Einfügestellen

Beschreibung:

Die Funktion ermittelt alle möglichen Einfügestellen, wenn der Cursor am Anfang eines finalen Elementes steht.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `requestInsertPosition` verwendet.

Implementierung:

Nähere Informationen zur Implementierung siehe auch Funktion `requestInsertPosition` auf Seite 113.

Funktionsname:

reqInsBehind reqInsBehind

Signatur:

reqInsBehind:: document -> objID -> objID -> insertPosList -> insertPosList

document: Dokument aus der Dokumentenstruktur

objID: ID des Vaterobjektes

objID: ID des Objektes, hinter dem eingefügt werden soll

insertPosList: Liste von bereits möglichen Einfügestellen (initial `mt`)

insertPosList: Liste aller möglichen Einfügestellen

Beschreibung:

Die Funktion ermittelt alle möglichen Einfügestellen, wenn der Cursor am Ende eines finalen Elementes steht.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `requestInsertPosition` verwendet.

Implementierung:

Nähere Informationen zur Implementierung sie auch Funktion `requestInsertPosition` auf Seite 113.

Funktionsname:

releaseContIDs releaseContIDs

Signatur:

releaseContIDs:: document -> objID -> ids -> ids

document: Dokument aus der Dokumentenstruktur

objID: Freizugebende Objekt-ID

ids: Struktur zur Vergabe von Objekt-IDs

ids: Geänderte Objekt-ID-Struktur

Beschreibung:

Die ID des angegebenen Objektes sowie alle IDs der Nachfolger und die IDs der Objekte, die den Inhalt des angegebenen Objektes und deren Nachfolger darstellen, werden freigegeben.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `deleteElement` verwendet.

Funktionsname:

insertElement2 insertElement2

Signatur:

insertElement2:: document -> objID -> objPosition -> objID -> ids -> elementtype -> (document, ids)

document: Dokument aus der Dokumentenstruktur

objID: ID des Bezugsobjektes der Objekt-Position (nur bei `in` und `inside`)

objPosition: Position, wo das neue Element eingefügt werden soll

objID: ID des einzufügenden Objektes

ids: Struktur zur Vergabe von Objekt-IDs

elementtype: Typ des einzufügenden Elementes

document: Geändertes Dokument

ids: Geänderte Objekt-ID-Struktur

Beschreibung:

Die Funktion fügt ein logisches Element vom Typ `elementtype` in das Dokument ein.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `insertElement` verwendet.

Implementierung:

Nähere Informationen zur Implementierung sie auch Funktion `insertElement` auf Seite 115.

1.4.4.11 Hilfsfunktionen für die Schnittstelle zur DTD

Hier sind die Hilfsfunktionen enthalten, die für die Implementierung der Öffentlichen Schnittstelle zur DTD erforderlich sind.

Funktionsname:

mtContext mtContext

Signatur:

mtContext:: objContext

objContext: Leerer Kontext

Beschreibung:

Die Funktion liefert einen leeren Kontext (siehe auch Sorte `objContext`).

Funktionsname:

getInsContext2 getInsContext2

Signatur:

getInsContext2:: document -> ids -> objID -> objPosition -> objContext

document: Dokument aus der Dokumentenstruktur

ids: Struktur zur Vergabe von Objekt-IDs

objID: ID des Vaterobjektes

objPosition: Einfügeposition bezogen auf das Vaterobjekt

objContext: Kontext der angegebenen Einfügestelle

Beschreibung:

Die Hilfsfunktion liefert den Kontext einer Einfügestelle bezogen auf ein Vaterobjekt und einer Objekt-Position. Der Kontext besteht aus dem Elementtypen des Vaterobjektes, alle *potentiellen* Vorgänger und Nachfolger.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `getInsertContext` verwendet.

Implementierung:

Nähere Informationen zur Implementierung siehe auch Funktion `getInsertContext` auf Seite 117.

Funktionsname:

getPredList getPredList

Signatur:

getPredList:: document -> objID -> predElemTypeList

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, deren Vorgänger ermittelt werden sollen

predElemTypeList: Liste von Elementtypen aller Vorgänger des angegebenen Objektes

Beschreibung:

Die Hilfsfunktion liefert eine Liste von Elementtypen der Elemente, die vor einem angegebenen Objekt liegen. Das Objekt ist selber noch in der Liste an letzter Position enthalten.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `getInsertContext` und `getObjectContext` verwendet.

Funktionsname:

getPredList2 getPredList2

Signatur:

getPredList2:: document -> predElemTypeList -> objID -> predElemTypeList

document: Dokument aus der Dokumentenstruktur

predElemTypeList: Bisherige Liste von Elementtypen von Vorgängern des angegebenen Objektes

objID: ID des Objektes, deren Vorgänger ermittelt werden sollen

predElemTypeList: Erweiterte Liste von Elementtypen von Vorgängern des angegebenen Objektes

Beschreibung:

Die Hilfsfunktion erzeugt die Elementtyp-Liste rekursiv.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `getPredList` verwendet.

Funktionsname:

getSuccList getSuccList

Signatur:

getSuccList:: document -> objID -> succElemTypeList

document: Dokument aus der Dokumentenstruktur

objID: ID des Objektes, deren Nachfolger ermittelt werden sollen

succElemTypeList: Liste von Elementtypen aller Nachfolger des angegebenen Objektes

Beschreibung:

Die Hilfsfunktion liefert eine Liste von Elementtypen der Elemente, die hinter einem angegebenen Objekt liegen. Das Objekt ist selber noch in der Liste an erster Position enthalten.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `getInsertContext` und `getObjectContext` verwendet.

Funktionsname:

getSuccList2 getSuccList2

Signatur:

getSuccList2:: document -> succElemTypeList -> objID -> succElemTypeList

document: Dokument aus der Dokumentenstruktur

succElemTypeList: Bisherige Liste von Elementtypen von Nachfolgern des angegebenen Objektes

objID: ID des Objektes, deren Nachfolger ermittelt werden sollen

succElemTypeList: Erweiterte Liste von Elementtypen von Nachfolgern des angegebenen Objektes

Beschreibung:

Die Hilfsfunktion erzeugt die Elementtyp-Liste rekursiv.

Abhängigkeiten:

Diese Hilfsfunktion wird nur von `getSuccList` verwendet.

1.4.4.12 Hilfsfunktionen für die Schnittstelle zu daVinci

Hier sind die Hilfsfunktionen enthalten, die für die Implementierung der Öffentlichen Schnittstelle zu *daVinci* erforderlich sind. Die verwendeten Sorten zur Erzeugung der notwendigen Datenstrukturen in *daVinci*-Notation werden im Modul `daVinci` definiert und an dieser Stelle importiert. Für weitere Informationen siehe dort.

Funktionsname:

makeGraphtree makeGraphtree

Signatur:

makeGraphtree:: doc -> elementtypeStruct -> objinfo -> objID -> nodeType -> graphtree

doc: Dokumentenstruktur

elementtypeStruct: Struktur, die die Namen der Elementtypen beinhaltet

objinfo: Objektinfo des zu bearbeitenden Objektes

objID: ID des zur Zeit aktiven Objektes

nodeType: Art der Hervorhebung des aktiven Objektes in *daVinci*

graphtree: Graphenstruktur für die Visualisierung in *daVinci*

Beschreibung:

Die Funktion liefert das aktuelle Objekt als Knoten-Datenstruktur der *daVinci*-Notation zurück.

Implementierung:

Durch die rekursiven Aufrufe zusammen mit `makeEdges` ergibt sich der gesamte Dokumentengraphen, wenn man die Funktion für das *Root-Objekt* aufruft.

Funktionsname:

makeEdges makeEdges

Signatur:

makeEdges:: doc -> elementtypeStruct -> objinfo -> objID -> nodeType -> edges

doc: Dokumentenstruktur

elementtypeStruct: Struktur, die die Namen der Elementtypen beinhaltet

objinfo: Objektinfo des zu bearbeitenden Objektes

objID: ID des zur Zeit aktiven Objektes

nodeType: Art der Hervorhebung des aktiven Objektes in *daVinci* edges: Liste von Kanten der *daVinci*-Struktur

Beschreibung:

Die Funktion liefert die Liste der Kanten aller Objekte, die die Söhne eines Objektes darstellen.

Implementierung:

Zusammen mit den rekursiven Aufrufen der Funktion `makeGraptree` ergibt sich die gesamte Struktur des Dokumentes als Graph.

Funktionsname:

makeEdge makeEdge

Signatur:

makeEdge:: doc -> elementtypeStruct -> objID -> nodeType -> objinfo -> edge

doc: Dokumentenstruktur

elementtypeStruct: Struktur, die die Namen der Elementtypen beinhaltet

objID: ID des zur Zeit aktiven Objektes

nodeType: Art der Hervorhebung des aktiven Objektes in *daVinci*

objinfo: Objektinfo des zu bearbeitenden Objektes

edge: Kante der *daVinci*-Struktur für das angegebene Objekt

Beschreibung:

Die Funktion liefert eine Kante zu dem aktuellen Objekt in der *daVinci*-Datenstruktur zurück.

Funktionsname:

makeRefEdges makeRefEdges

Signatur:

makeRefEdges:: doc -> objID -> edges

doc: Dokumentenstruktur

objinfo: Objektinfo des zu bearbeitenden Objektes

edges: Ein- oder nullelementige Liste von Kanten der `daVinci`-Struktur

Beschreibung:

Die Funktion liefert eine Kante zu einem Objekt, auf das das angegebene Objekt verweist. Enthält das Objekt keine Referenz, so wird [] zurückgeliefert.

Funktionsname:

getNodeAttributes getNodeAttributes

Signatur:

getNodeAttributes:: elementtypeStruct -> objinfo -> objID -> nodeType -> graphAttributes

elementtypeStruct: Struktur, die die Namen der Elementtypen beinhaltet

objinfo: Objektinfo des zu bearbeitenden Objektes

objID: ID des zur Zeit aktiven Objektes

nodeType: Art der Hervorhebung des aktiven Objektes in *daVinci*

graphAttributes: Liste aller Attribute eines Knoten

Beschreibung:

Die Funktion liefert die Liste aller Attribute eines Knoten zurück. Das aktive Objekt wird hervorgehoben.

Implementierung:

Die Art der Hervorhebung des aktiven Objektes ergibt sich aus dem übergebenen `nodeType`:

- colored: Der Knoten wird farbig hervorgehoben (Blau)
- ellipse: Der Knoten erhält als Umrandung keine Box, sondern eine Ellipse
- none: keine Hervorhebung des aktiven Knoten

Funktionsname:

getEdgeAttributes getEdgeAttributes

Signatur:

getEdgeAttributes:: graphAttributes

graphAttributes: Liste aller Attribute einer Kante

Beschreibung:

Die Funktion liefert eine einelementige Liste der Attribute einer normalen Kante (ungerichtet).

Funktionsname:

getRefAttributes getRefAttributes

Signatur:

getRefAttributes:: graphAttributes

graphAttributes: Liste aller Attribute einer Referenz-Kante

Beschreibung:

Die Funktion liefert eine einelementige Liste der Attribute einer Referenz-Kante (Farbe rot).

Funktionsname:

makeEmptyGraphtree makeEmptyGraphtree

Signatur:

makeEmptyGraphtree:: graphtree

graphtree: Graphenstruktur für die Visualisierung in *daVinci*

Beschreibung:

Die Funktion liefert einen Graphen, der angezeigt wird, falls noch kein aktuelles Dokument existiert.

1.4.4.13 Hilfsfunktionen zum Visualisieren der Datenstrukturen für Tests

Zur Implementierung der in den Öffentlichen Schnittstellen auf Seite 121 beschriebenen Testfunktionen zum Visualisieren der Datenstrukturen zu Testzwecken werden eine Reihe lokaler Funktionen verwendet. Auf die Beschreibung dieser Funktionen wird an dieser Stelle verzichtet.

1.5 PresRules

Gunnar Kavemann

Präsentationsregeln

1.5.1 Funktionalität

Das Modul `PresRules` (PR) hält zur Laufzeit die Präsentationsregeln für sämtliche im derweilgen Dokument vorhandenen Elemente vor.

1.5.2 Entwurf

Es wurde eine Datenstuktur entworfen, die geeignet war, die Präsentationsregeln in geeigneter Form zu speichern. Diese Struktur, die `pr`, wird nach ihrer Erzeugung komplett übergeben, und muß bei jedem neuerlichen Aufruf von Funktionen der `PresRules` wieder übergeben werden. Die Funktionen zum Schreiben und Lesen der `pr` können in beliebiger Reihenfolge aufgerufen werden.

1.5.3 Öffentliche Schnittstellen

1.5.3.1 Sorten

Sortenname:

pr. pr.

Beschreibung:

Die Sorte pr wird nur en bloc beim Aufruf der PR-Funktionen verwendet. Beim Neueinlesen eines Dokumentes wird zuerst mit mtPR eine leere pr erzeugt. Diese wird dann sukzessiv mit den zum Dokument gehörenden Bestandteilen der Präsentationsregeln vom Modul `PresRules` gefüllt.

Sortenname:

succObject succObject

Signatur:

succObject:: soObject (soImElementtype::elementtype)
(soImString::string) (soImElTypeList::elementtypelist).

Beschreibung:

Die Sorte `succObject` wird benötigt, um die einzelnen Informationen aus einem `succObject` zu erhalten. (Wird von den Formatierern benötigt.)

1.5.3.2 Funktionen

Funktionsname:

mtPR mtPR

Signatur:

mtPR:: pr.

Beschreibung:

Erzeugen einer leere `pr`. Die Funktion wird ohne Parameter aufgerufen. Beim Aufruf wird eine leere `pr` erzeugt.

Vor- und Nachbedingungen:

Eine vorher vorhandene `pr` wird durch die neue, leere `pr` überschrieben.

Funktionsname:

announceView announceView

Signatur:

announceView:: pr -> err -> string -> (pr,err).

pr: Alte `pr`

err: Alte `err`

string: Viewname

(pr,err): Antwort

Beschreibung:

Die Funktion dient zur Anmeldung einer Sicht. Als Argumente werden die alte `pr`, die alte `err` und der Name der Sicht übergeben. Das Ergebnis ist ein Tupel aus neuer `pr` und neuer `err`.

Vor- und Nachbedingungen:

Falls der Sichtname schon in der `pr` enthalten ist, sollte er nicht noch einmal angemeldet werden.

Implementierung:

Der neue View (string) wird mit `postfix` an die `prviewList` angehängt.

Funktionsname:

announceObject announceObject

Signatur:

announceObject:: pr -> err -> elementtype -> objtype -> (pr,err).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des neuen Objektes

objtype: Objekttyp des neuen Objektes

(pr,err): Antwort

Beschreibung:

Einfügen eines neuen Objektes in das Dictionary, in dem alle PR-Objekte gehalten werden.

Vor- und Nachbedingungen:

Jedes Objekt kann nur einmal angemeldet werden. Bei allen weiteren Versuchen wird eine Fehlermeldung erzeugt.

Abhängigkeiten:

getPrObject, makeErr

Implementierung:

Mit den übergebenen Argumenten wird erst ein neues Dictionary-Element kreiert. Danach wird mit dem übergebenen elementtype die Funktion `getPrObject` aufgerufen, um zu testen, ob eventuell dieses Objekt schon angemeldet wurde. Falls das der Fall ist, wird eine entsprechende Fehlermeldung in die `err` eingefügt und die alte `pr` unverändert gelassen. Falls das Objekt noch nicht bekannt ist, wird das neue Dictionary-Element in das Dictionary der `pr` eingefügt.

Funktionsname:

announceBoxType announceBoxType

Signatur:

announceBoxType:: pr -> err -> elementtype -> string -> string -> (pr , err).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die angemeldet wird

string: Boxtyp

(pr,err): Antwort

Beschreibung:

Anmelden des Formatobjekttyps für eine bestimmte Sicht.

Vor- und Nachbedingungen:

Für jedes Objekt kann pro Sicht nur ein Boxtyp angemeldet werden.

Abhängigkeiten:

getViewAttr, makeBoxType, setPrObject

Implementierung:

Zuerst wird mit `getViewAttr` der Attributblock für die entsprechende Sicht geholt. Im Fehlerfall wird mit `makeErr` eine entsprechende Fehlermeldung in `err` eingefügt, und diese zusammen mit der ursprünglichen `pr` zurückgegeben. Wenn die Aktion erfolgreich war, wird der Boxtyp-String mit `makeBoxType` zu einem `boxType` gewandelt und mit dem Modifikator `vatBoxType` in das aktuelle ViewAttr gesetzt. Aus diesem neuen ViewAttr und dem aktuellen Objekt wird nun mit dem Modifikator `obViewAtList` und der Listenoperation `exchange` ein neues Objekt kreiert. Dieses wird nun mit `setPrObject` in die `pr` eingefügt.

Funktionsname:

appendToSuccObjectList appendToSuccObjectList

Signatur:

appendToSuccObjectList:: pr -> err -> elementtype -> string -> succObject -> (pr,err).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die angemeldet wird

succObject: Anzufügendes Objekt

(pr,err): Antwort

Beschreibung:

Anhängen eines Folgeobjektes an die Liste der Folgeobjekte. Die Funktion bekommt als Eingabe die Alte `pr`, die alte err, das Objekt, die Sicht und das Folgeobjekt, das an die Liste angehängt werden soll. Zurückgegeben wird ein Tupel aus neuer pr und eventuell neuem Fehler.

Abhängigkeiten:

getViewAttr, makeErr, setPrObject

Implementierung:

Zuerst wird mit `getViewAttr` der Attributblock für die entsprechende Sicht geholt. Im Fehlerfall wird mit `makeErr` eine entsprechende Fehlermeldung in `err` eingefügt und diese zusammen mit der ursprünglichen `pr` zurückgegeben. Wenn die Aktion erfolgreich war, wird mit dem Modifikator `vatSuccObjectList` und der Listenoperation `postfix` das als Argument übergebene SuccObject in das aktuelle ViewAttr gesetzt. Aus diesem neuen ViewAttr und dem aktuellen Objekt wird nun mit dem Modifikator `obViewAtList` und der Listenoperation `exchange` ein neues Objekt kreiert. Dieses wird nun mit `setPrObject` in die `pr` eingefügt.

Funktionsname:

appendValToAttr appendValToAttr

Signatur:

appendValToAttr:: pr -> err -> elementtype -> string -> string -> string -> (pr,err).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die angemeldet wird

string: Value

string: Wert

(pr,err): Antwort

Beschreibung:

Einfügen eines 'Value-Attributes' in den Attributblock der angegebenen Sicht.

Abhängigkeiten:

getViewAttr, makeErr, setAttr, setPrObject

Implementierung:

Zuerst wird mit `getViewAttr` der Attributblock für die entsprechende Sicht geholt. Im Fehlerfall wird mit `makeErr` eine entsprechende Fehlermeldung in `err` eingefügt und diese zusammen mit der ursprünglichen `pr` zurückgegeben. Wenn die Aktion erfolgreich war, wird mit dem Selektor `vatAttr` der Attributblock aus dem ViewAttr geholt. Dann wird mit diesem Attributblock und den beiden als Argument übergebenen Strings für Attribut und Wert die Funktion `setAttr` aufgerufen. In Fehlerfall liefert diese eine um eine entsprechende Fehlermeldung bereicherte `err` und den ursprünglichen Attributblock zurück. Wenn kein Fehler aufgetreten ist, kommt eine ungeänderte `err` und ein neuer Attributblock zurück. Mit diesem Attributblock wird dann mit dem Modifikator `vatAttr` ein neues aktuelles ViewAttr erzeugt, aus welchem zusammen mit dem aktuellen Objekt nun mit dem Modifikator `obViewAtList` und der Listenoperation `exchange` ein neues Objekt kreiert wird. Dieses wird nun mit `setPrObject` in die `pr` eingefügt.

Funktionsname:

appendUnitToAttr appendUnitToAttr

Signatur:

appendUnitToAttr:: pr -> err -> elementtype -> string -> string -> real -> string -> (pr,err).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die angemeldet wird

string: Attributname

real: Wert

string: Einheit

(pr,err): Antwort

Beschreibung:

Einfügen eines 'Unit-Attributes' in den Attributblock der angegebenen Sicht.

Abhängigkeiten:

getViewAttr, makeErr, setUnit, setPrObject

Implementierung:

Zuerst wird mit `getViewAttr` der Attributblock für die entsprechende Sicht geholt. Im Fehlerfall wird mit `makeErr` eine entsprechende Fehlermeldung in `err` eingefügt und diese zusammen mit der ursprünglichen `pr` zurückgegeben. Wenn die Aktion erfolgreich war, wird mit dem Selektor `vatAttr` der Attributblock aus dem ViewAttr geholt. Dann wird mit diesem Attributblock und dem als Argument übergebenen String für Attribut, dem Real als numerischem Wert und dem String für die Einheit die Funktion `setUnit` aufgerufen. In Fehlerfall liefert diese den ursprünglichen Attributblock und eine um eine entsprechende Fehlermeldung bereicherte `err` zurück. Wenn kein Fehler aufgetreten ist, kommt ein neuer Attributblock und eine ungeänderte `err` zurück. Mit diesem Attributblock wird dann mit dem Modifikator `vatAttr` ein neues aktuelles ViewAttr erzeugt, aus welchem zusammen mit dem aktuellen Objekt nun mit dem Modifikator `obViewAtList` und der Listenoperation `exchange` ein neues Objekt kreiert wird. Dieses wird nun mit `setPrObject` in die `pr` eingefügt.

Funktionsname:

appendLogToAttr appendLogToAttr

Signatur:

appendLogToAttr:: pr -> err -> elementtype -> string -> string -> elementtype -> (pr,err).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die angemeldet wird

string: Wertname

elementtype: Logisches Element

(pr,err): Antwort

Beschreibung:

Einfügen eines 'Log-Attributes' in den Attributblock der angegebenen Sicht.

Abhängigkeiten:

getViewAttr, makeErr, setlog, setPrObject

Implementierung:

Zuerst wird mit `getViewAttr` der Attributblock für die entsprechende Sicht geholt. Im Fehlerfall wird mit `makeErr` eine entsprechende Fehlermeldung in `err` eingefügt und diese zusammen mit der ursprünglichen `pr` zurückgegeben. Wenn die Aktion erfolgreich war, wird mit dem Selektor `vatAttr` der Attributblock aus dem ViewAttr geholt. Dann wird mit diesem Attributblock und den als Argumenten übergebenen String für Attribut und dem dazugehörigen elementtype die Funktion `setUnit` aufgerufen. In Fehlerfall liefert diese den ursprünglichen Attributblock und eine um eine entsprechende Fehlermeldung bereicherte `err` zurück. Wenn kein Fehler aufgetreten ist, kommt ein neuer Attributblock und eine ungeänderte `err` zurück. Mit diesem Attributblock wird dann mit dem Modifikator `vatAttr` ein neues aktuelles ViewAttr erzeugt, aus welchem zusammen mit dem aktuellen Objekt nun mit dem Modifikator `obViewAtList` und der Listenoperation `exchange` ein neues Objekt kreiert wird. Dieses wird nun mit setPrObject in die `pr` eingefügt.

Funktionsname:

getPageLayout getPageLayout

Signatur:

getPageLayout:: pr -> err -> doc -> objID -> string -> (page).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Objekt, für das die Attribute erfragt werden

string: Sicht, für die die Attribute erfragt werden page: Antwort

Beschreibung:

Abfrage des Seitenlayouts für das durch die ObjectID bezeichnet Objekt und die angegebene Sicht.

Abhängigkeiten:

getPageL

Implementierung:

Die Funktion ist eine Vorschaltfunktion für `getPageL`. Mit `mtAtPage` wird ein leerer Seitenattributblock erzeugt. Damit und mit den anderen übergebenen Argumenten wird getPageL aufgerufen. Die `objID` wird hierbei zweimal als Argument übergeben, einmal als Ursprungs-ID und einmal als aktuelle ID. In der jetzigen Implementation werden die von den nachfolgend aufgerufenen Funktionen erzeugten Fehlermeldungen noch nicht ausgewertet. Deshalb wird die `err` auch nicht mit zurückgegeben.

Funktionsname:

getParagraphAttributes getParagraphAttributes

Signatur:

getParagraphAttributes:: pr -> err -> doc -> objID -> string -> (paragraph).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Objekt, für das die Attribute erfragt werden

string: Sicht, für die die Attribute erfragt werden

paragraph: Antwort

Beschreibung:

Abfrage der Absatzattribute für das angegebene Objekt in der angegeben Sicht.

Abhängigkeiten:

getParAttrs

Implementierung:

Die Funktion ist eine Vorschaltfunktion für `getParAttrs`. Mit `mtAtParagraph` wird ein leerer Absatzattributblock erzeugt. Damit und mit den anderen übergebenen Argumenten wird getParAttrs aufgerufen. Die ObjID wird hierbei zweimal als Argument übergeben, einmal als Ursprungs-ID und einmal als aktuelle ID. In der jetzigen Implementation werden die von den nachfolgend aufgerufenen Funktionen erzeugten Fehlermeldungen noch nicht ausgewertet. Deshalb wird die `err` auch nicht mit zurückgegeben.

Funktionsname:

getDocAttributes getDocAttributes

Signatur:

getDocAttributes:: pr -> err -> doc -> objID -> string -> docu.

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Objekt, für das die Attribute erfragt werden

string: Sicht, für die die Attribute erfragt werden

docu: Antwort

Beschreibung:

Abfrage der Dokumentattribute für das angegebene Objekt in der angegeben Sicht.

Abhängigkeiten:

getDocuAttrs

Implementierung:

Die Funktion ist eine Vorschaltfunktion für `getDocuAttrs`. Mit `mtAtDocument` wird ein leerer Dokumentattributblock erzeugt. Damit und mit den anderen übergebenen Argumenten wird getDocuAttrs aufgerufen. Die ObjID wird hierbei zweimal als Argument übergeben, einmal als Ursprungs-ID und einmal als aktuelle ID. In der jetzigen Implementation werden die von den nachfolgend aufgerufenen Funktionen erzeugten Fehlermeldungen noch nicht ausgewertet. Deshalb wird die `err` auch nicht mit zurückgegeben.

Funktionsname:

getFontInfo getFontInfo

Signatur:

getFontInfo:: pr -> err -> doc -> objID -> string -> font.

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

doc: Dokumentstruktur

objID: Objekt, für das die Attribute erfragt werden

string: Sicht, für die die Attribute erfragt werden

font: Antwort

Beschreibung:

Abfrage der Fontattribute für das angegebene Objekt in der angegeben Sicht.

Vor- und Nachbedingungen:

...

Abhängigkeiten:

getFontInf

Implementierung:

Die Funktion ist eine Vorschaltfunktion für `getFontInf`. Mit `mtAtFont` wird ein leerer Fontattributblock erzeugt. Damit und mit den anderen übergebenen Argumenten wird getFontInf aufgerufen. Die `objID` wird hierbei zweimal als Argument übergeben, einmal als Ursprungs-ID und einmal als aktuelle ID. In der jetzigen Implementation werden die von den nachfolgend aufgerufenen Funktionen erzeugten Fehlermeldungen noch nicht ausgewertet. Deshalb wird die `err` auch nicht mit zurückgegeben.

Funktionsname:

getCounterAttributes

getCounterAttributes

Signatur:

getCounterAttributes:: pr -> err -> doc -> objID -> string -> counter.

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Objekt, für das die Attribute erfragt werden

string: Sicht, für die die Attribute erfragt werden

counter: Antwort

Beschreibung:

Abfrage der Zählerattribute für das durch die objectID bezeichnet Objekt und die angegebene Sicht.

Abhängigkeiten:

getCounterAttrs

Implementierung:

Die Funktion ist eine Vorschaltfunktion für `getCounterAttrs`. Mit `mtAtCounter` wird ein leerer Zählerattributblock erzeugt. Damit und mit den anderen übergebenen Argumenten wird getCounterAttrs aufgerufen. Die `objID` wird hierbei zweimal als Argument übergeben, einmal als Ursprungs-ID und einmal als aktuelle ID. In der jetzigen Implementation werden die von den nachfolgend aufgerufenen Funktionen erzeugten Fehlermeldungen noch nicht ausgewertet. Deshalb wird die `err` auch nicht mit zurückgegeben.

Funktionsname:

getBoxtype getBoxtype

Signatur:

getBoxtype:: pr -> err -> elementtype -> string -> (pr,err,boxtype).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die der Boxtyp erfragt wird

(pr,err,boxtype): Antwort

Beschreibung:

Holt den Boxtyp für das durch den elementtype spezifizierte Objekt

Abhängigkeiten:

getViewAttr

Implementierung:

Mit `getViewAttr` wird der entsprechende Sicht-Attributblock geholt. Anschließend wird mit dem Separator `vatBoxType` der BoxTyp ausgelesen und als Tupel zusammen mit der `pr` und der `err` zurückgegeben.

Funktionsname:

getOrder getOrder

Signatur:

getOrder:: pr -> err -> elementtype -> string -> elementtypelist -> (err, elementtypelist).

pr: Alte `pr`

err: Alte `err`

elementtype: Elementtype des Objektes

string: Sicht, für die der Boxtyp angemeldet wird

elementtypelist: Vorhandene Liste

(err,elementtypelist): Antwort

Beschreibung:

Holt die richtige Reihenfolge der Folgeobjekte für ein Objekt in einer bestimmten Sicht.

Implementierung:

In der aktuellen Implementation ist diese Funktion noch lediglich als Dummy vorhanden, weil die Formatierer die zurückgegebene Liste nicht in befriedigender Form auswerten können. PR-seitig ist die Funktion jedoch bereits voll funktionsfähig.
Die dann gültige Implementation weicht allerdings erheblich von der momentan projektierten ab:

Abhängigkeiten:

getSuccObjectList, setOrderList, (mkSuccObjectList, fetchElTypes

Implementierung:

Zuerst wird mit `getSuccObjectList` die für das Objekt und die Sicht gültige Folgeobjektliste geholt. Anschließend wird mit `setOrderList` die übergebene Liste der vorhandenen Folgeobjekte umgeordnet und ggf. komplettiert. Die beiden Funktionen `mkSuccObjectList` und `fetchElTypes` dienen nur der Anpassung an die Formatierer, die im Moment nur in der Lage sind, mit Listen von elementtypes umzugehen.

1.5.4 Lokale Implementation

1.5.4.1 Sorten

Sortenname:

pr pr

Signatur:

pr:: prule (prviewList :: viewList) probjectDict :: objectDict).

Beschreibung:

Dies ist die Sorte, in der die Sichten und die Attribute aller Objekte gehalten werden.

Sortenname:

viewList viewList

Signatur:

viewList:: (strings)

Beschreibung:

Dies ist die Liste der Sichten

Sortenname:

view view

Signatur:

view:: string

Beschreibung:

Der Name einer einzelnen Sicht

Sortenname:

objectDictElement objectDictElement

Signatur:

objectDictElement:: object (obElementtype :: elementtype) (obObjecttype :: objtype) (obViewAttList :: viewAttrList) (obStat :: stat).

Beschreibung:

In dieser Sorte befinden sich alle Informationen über das einzelne Objekt.

Sortenname:

stat stat

Signatur:

stat:: complete — incomplete

Beschreibung:

Status des Objektes. Er zeigt an, ob aller erforderlichen Informationen gesetzt sind.

Sortenname:

viewAttrList viewAttrList

Signatur:

viewAttrList:: [viewAttr]

Beschreibung:

Dies ist die Liste der Attribute für die Sichten.

Sortenname:

viewAttr viewAttr

Signatur:

viewAttr:: vattr (vatView :: view) (vatBoxtype :: boxtype) (vatSuccObjectList :: succObjectList) (vatAttr :: attrs).

Beschreibung:

Diese Sorte enthält die Informationen über ein Objekt in einer einzelnen Sicht.

Sortenname:

succObjectList succObjectList

Signatur:

succObjectList:: [succObject]

Beschreibung:

Die Liste der Folgeobjekte

Sortenname:

attrs attrs

Signatur:

attrs:: attr (atDocument :: docu) (atPage :: page) (atParagraph :: paragraph) (atCounter :: counter) (atCatalogs :: catalogs) (atFont :: prFont) (atNotes :: notes).

Beschreibung:

Dies sind alle Attribute

Sortenname:

prfont prfont

Signatur:

prfont:: prfont (fontname :: string) (fontstyle :: string) (fontsize :: attrNum).

Beschreibung:

Dies sind die Fontattribute

1.5.4.2 Funktionen

Funktionsname:

getViewList getViewList

Signatur:

getViewList:: pr -> strings

pr: Alte `pr`

Beschreibung:

Holt die Liste der bereits angemeldeten Sichten aus der `pr`.

Implementierung:

Mit dem Separator `prviewList` wird die Liste der angemeldeten Sichten aus der `pr` geholt und zurückgegeben.

Funktionsname:

setAttr setAttr

Signatur:

setAttr:: attrs -> string -> string -> err -> (attrs,err).

attrs: aktueller Attributblock

string: Attributname

string: Wert

err: Alte `err`

(attrs,err): Antwort

Beschreibung:

Setzt für das angegebene Attribut den gewünschten Wert.

Abhängigkeiten:

stringEqual, posType, alnType, symType, ordType, ppsType, flgType

Implementierung:

Zuerst wird der Attributname mit dem ersten gültigen Attributstring verglichen. Bei Ungleichheit wird zum nächsten gültigen Attributnamen übergegangen, usw. Kann überhaupt keine Gleichheit festgestellt werden, wird mit `makeErr` eine entsprechende Fehlermeldung in die `err` eingefügt. Bei Gleichheit wird der String, der den Wert enthält, mit `posType` in den entsprecheden Typ gewandelt und mit dem Modifikator `atposition` in den Dokumentblock geschrieben. Dieser wird nun mit dem Modifikator `atDocument` in den neuen Attributblock gesetzt, der dann zusammen mit der `err` zurückgegeben wird. Das Verfahren ist bei den unterschiedlichen Attributen bis auf die Funktionen zur Wandlung von Strings in Typen und die Modifikatoren gleich.

Funktionsname:

setUnit setUnit

Signatur:

setUnit:: attrs -> string -> real -> string -> err -> (attrs,err).

attrs: aktueller Attributblock

string: Attributmane

real: Wert

string: Einheit

err: Alte `err`

(attrs,err): Antwort

Beschreibung:

Setzt für das angegebene Attribut den gewünschten Wert in der angegebenen Einheit.

Abhängigkeiten:

stringEqual, unitType, makePixel

Implementierung:

Zuerst wird der Attributname mit dem ersten gültigen Attributstring verglichen. Bei Ungleichheit wird zum nächsten gültigen Attributnamen übergegangen, usw. Kann überhaupt keine Gleichheit festgestellt werden, wird mit `makeErr` eine entsprechende Fehlermeldung in die `err` eingefügt. Bei Gleichheit wird der die Einheit enthaltende String mit `unitType` in den entsprecheden Typ gewandelt. Anschließend wird mit `makePixel` der real zusammen mit der Einheit in das passende Format gebracht und mit dem dem Attribut entsprechenden Modifikatoren in den Attributblock geschrieben, der dann zusammen mit der `err` zurückgegeben wird.

Funktionsname:

makePixel makePixel

Signatur:

makePixel:: real -> attrUnit -> integer.

real: Wert

attrUnit: Einheit

interger: Antwort

Beschreibung:

Wandelt real und Einheit in einen Pixelwert um

Abhängigkeiten:

mkVal

Implementierung:

Mit den übergebenen Argumenten wird `makeVal` aufgerufen. Anschließend wird das Ergebnis in einen integer gewandelt und zurückgegeben.

Funktionsname:

mkVal mkVal

Signatur:

mkVal:: attrUnit -> real -> real.

attrUnit: Einheit

real: Wert

real: Antwort

Beschreibung:

Errechnet aus Wert und Einheit einen Wert

Implementierung:

Der real wird mit einem der Einheit entsprechenden Faktor multipliziert und zurückgegeben.

Funktionsname:

setLog setLog

Signatur:

setLog:: attrs -> string -> elementtype -> err -> (attrs,err).

attrs: aktueller Attributblock

string: Attributname

elementtype: Logisches Element

err: Alte `err`

(attrs,err): Antwort

Beschreibung:

Setzt für das angegebene Attribut das gewünschte logische Element.

Abhängigkeiten:

stringEqual, posType, alnType, symType, ordType, ppsType, flgType,

Implementierung:

Zuerst wird der Attributname mit dem ersten gültigen Attributstring verglichen. Bei Ungleichheit wird zum nächsten gültigen Attributnamen übergegangen, usw. Kann überhaupt keine Gleichheit festgestellt werden, wird mit `makeErr` eine entsprechende Fehlermeldung in die `err` eingefügt. Bei Gleichheit wird mit dem dem Attribut entsprechenden Selektor der entsprechende Attributblock ausgewählt. In diesen wird dann das Logische Element mit dem entsprechenden Modifikator gesetzt und zuletzt wird der Attributblock wieder mit dem entsprechenden Modifikator in den Attributblock geschrieben, der seinerseits zusammen mit der `err` zurückgegeben wird.

Funktionsname:

getPageL getPageL

Signatur:

getPageL:: pr -> err -> doc -> objID -> objID -> view -> page -> (pr,err,page).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Identifikation des Ursprungsobjektes

objID: Identifikation des aktuellen Objektes

string: Sicht, für die abgefragt wird

page: Seitenattributblock, der aufgefüllt werden soll

(pr,err,page): Antwort

Beschreibung:

Mit dieser Funktion wird die Vererbung der Seitenattribute der Objekte realisiert, bzw. 'Aufgelöst'. Der Seitenattributblock für das Ursprungsobjekt wird aus der pr ausgelesen. Wenn der Attributblock nicht komplett ist, wird das Vorgängerobjekt bestimmt, und es werden durch rekursiven Aufruf der Funktion die Attribute aus dem Attributblock des Vorgängerobjektes ausgelesen.

Abhängigkeiten:

getObjElementtype, getViewattr, makeErr, getPageAT

Implementierung:

Als erstes wird die aktuelle ID geprüft. Wenn diese gleich 0 ist, ist die Wurzel des Dokumentes erreicht, und es gibt kein Vorgängerobjekt mehr. Das bedeutet, das mindestens ein Attribut durch keinen der in Frage kommenden Attributblöcke mit einem Wert belegt werden konnte. In diesem Fall wird geprüft, ob der Sichtname eventuell bereits auf 'ALL' gesetzt ist. Wenn nicht, wird `getPageL` mit der Sicht 'ALL', in deren Attributblock alle Werte geschrieben werden, für die keine spezielle Sicht angegeben wurden, aufgerufen. Wenn der Sichtname jedoch bereits auf 'ALL' gesetzt ist, wird als letzte Möglichkeit die Attribute nicht undefineiert zu lassen, die Funktion `getPageAt` mit `defPage`, einem mit Defaultwerten belegten Seitenattributblock, aufgerufen, und die `err` wird um eine entsprechende Fehlermeldung ergänzt.
Wenn die aktuellen ObjektID nicht 0 ist, wird zuerst der boolean, der anzeigt, ob der Attributblock komplett mit allen erforderlichen Werten belegt ist auf true gesetzt. Danach wird mit der über `ISTypes` transitiv aus Document importierten Funktion `getObjElementtype` der zu der aktuellen ObjektId gehörige, das entsprechender PR-Objekt bezeichnende elementtype bestimmt. Damit und anderen der Funktion übergebenen Argumenten wird mittels `getViewAttr` der gewünschte View-Attributblock aus der `pr` geholt. Aus diesem wird mit den Sepataroren `vatAttr` und `atPage` der dann aktuelle Seitenattributblock geholt. Mit diesem wird dann mittels der Funktion `getPageAt` versucht, die in Page noch undefinierten Attribute mit Werten zu belegen. Nach Aufruf von `getPageAt` wird über die Prüfung des boolean bestimmt, ob der Attributblock komplett ist. Ist das der Fall, wird er zusammen mit der `pr` und der `err` zurückgegeben. Falls Compl durch `getPageAt` auf false gesetzt wurde, wird `getPageL` mit einer neuen, durch die über `ISTypes` transitiv aus `Document` importierte Funktion `getParent` bestimmten aktuellen ID rekursiv aufgerufen.

Funktionsname:

getPageAt getPageAt

Signatur:

getPageAt:: page -> page -> boolean -> (page,boolean).

page: neuer Attributblock, in den ggf. geschrieben wird

page: vorhandener Attributblock, aus dem gelesen wird

boolean: Anzeige, ob alle Attribute gesetzt sind

(page,boolean): Antwort

Beschreibung:

Schreibt Attribute in den eneuen Attrbutblock, wenn diese dort noch nicht gesetzt sind

Abhängigkeiten:

undefNum

Implementierung:

Zuerst wird für das erste Attribut des Blocks im neuen Attributblock mit `undefNum` geprüft, ob es schon gesetzt ist, oder nicht. Wenn es gesetzt ist, wird es nicht verändert, und es wird zum zweiten Attribut gewechselt. Ist das erste Attribut noch nicht mit einem Wert belegt, wird als Nächstes mit `undefNum` geprüft, ob das erste Attribut im vorhandenen Attributblock mit einem Wert belegt ist. Ist das der Fall, wird dieser Wert in den neuen Block übernommen, der boolean bleibt unverändert, und es wird zum zweiten Attribut gewechselt. Wenn in der ActPage für das erste Attribut auch kein Wert vorhanden ist, wird der boolean auf false gesetzt. Anschließend wird zum zweiten Attribut gewechselt. Dieses Verfahren wird sooft wiederholt, bis alle Attribute des Blockes abgearbeitet sind.

Funktionsname:

undefNum undefNum

Signatur:

undefNum:: attrNum -> boolean.

attrNum: Zu prüfendes Attribut

(boolean): Antwort

Beschreibung:

Prüft, ob ein Num-Attribut noch undefiniert ist

Implementierung:

Beim erzeugen der Attributblöcke werden alle Num-Attribute zuerst auf 'undef' gesetzt. Hier geprüft, ob das Attribut diesen Wert hat. Wenn ja, wird der boolean auf true gesetzt.

Funktionsname:

getParAttrs getParAttrs

Signatur:

getParAttrs:: pr -> err -> doc -> objID -> objID -> view -> paragraph -> (pr,err,paragraph).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Identifikation des Ursprungsobjektes

objID: Identifikation des aktuellen Objektes

string: Sicht, für die abgefragt wird

paragraph: Absatzattributblock, der aufgefüllt werden soll

(pr,err,paragraph): Antwort

Beschreibung:

Mit dieser Funktion wird die Vererbung der Absatzattribute der Objekte realisiert, bzw. 'Aufgelöst'. Der Absatzattributblock für das Ursprungsobjekt wird aus der pr ausgelesen. Wenn der Attributblock nicht komplett ist, wird das Vorgängerobjekt bestimmt, und es werden durch rekursiven Aufruf der Funktion die Attribute aus dem Attributblock des Vorgängerobjektes ausgelesen.

Vor- und Nachbedingungen:

...

Abhängigkeiten:

getParAt

Implementierung:

Als erstes wird die aktuelle ID geprüft. Wenn diese gleich 0 ist, ist die Wurzel des Dokumentes erreicht, und es gibt kein Vorgängerobjekt mehr. Das bedeutet, das mindestens ein Attribut durch keinen der in Frage kommenden Attributblöcke mit einem Wert belegt werden konnte. In diesem Fall wird geprüft, ob der Sichtname eventuell bereits auf 'ALL' gesetzt ist. Wenn nicht, wird `getParAt` mit der Sicht 'ALL', in deren Attributblock alle Werte geschrieben werden, für die keine spezielle Sicht angegeben wurden, aufgerufen. Wenn der Sichtname jedoch bereits auf 'ALL' gesetzt ist, wird als letzte Möglichkeit die Attribute nicht undefineiert zu lassen, die Funktion `getParAt` mit `defParagraph`, einem mit Defaultwerten belegten Absatzattributblock, aufgerufen, und die `err` wird um eine entsprechende Fehlermeldung ergänzt.
Wenn die aktuellen ObjektID nicht 0 ist, wird zuerst der boolean, der anzeigt, ob der Attributblock komplett mit allen erforderlichen Werten belegt ist auf true gesetzt. Danach wird mit der über `ISTypes` transitiv aus Document importierten Funktion `getObjElementtype` der zu der aktuellen ObjektId gehörige, das entsprechender PR-Objekt bezeichnende elementtype bestimmt. Damit und anderen der Funktion übergebenen Argumenten wird mittels `getViewAttr` der gewünschte View-Attributblock aus der `pr` geholt. Aus diesem wird mit den Sepataroren `vatAttr` und `atParagraph` der dann aktuelle Absatzattributblock geholt. Mit diesem wird dann mittels der Funktion `getParAt` versucht, die in Paragraph noch undefinierten Attribute mit Werten zu belegen. Nach Aufruf von `getParAt` wird über die Prüfung des boolean bestimmt, ob der Attributblock komplett ist. Ist das der Fall, wird er zusammen mit der `pr` und der `err` zurückgegeben. Falls Compl durch `getParAt` auf false gesetzt wurde, wird `getParAttrs` mit einer neuen, durch die über `ISTypes` transitiv aus `Document` importierte Funktion `getParent` bestimmten aktuellen ID rekursiv aufgerufen.

Funktionsname:

getParAt getParAt

Signatur:

getParAt:: paragraph -> paragraph -> boolean -> (paragraph,boolean).

paragraph: neuen Attributblock, in den ggf. geschrieben wird

paragraph: vorhandener Attributblock, aus dem gelesen wird

boolean: Anzeige, ob alle Attribute gesetzt sind

(paragraph,boolean): Antwort

Beschreibung:

Schreibt Attribute in den neuen Attributblock, wenn diese dort noch nicht gesetzt sind

Abhängigkeiten:

undefNum, undefAln

Implementierung:

Zuerst wird für das erste Attribut des Blocks im neuen Attributblock mit `undefNum` geprüft, ob es schon gesetzt ist, oder nicht. Wenn es gesetzt ist, wird es nicht verändert, und es wird zum zweiten Attribut gewechselt. Ist das erste Attribut noch nicht mit einem Wert belegt, wird als Nächstes mit `undefNum` geprüft, ob das erste Attribut im vorhandenen Attributblock mit einem Wert belegt ist. Ist das der Fall, wird dieser Wert in den neuen Block übernommen, der boolean bleibt unverändert, und es wird zum zweiten Attribut gewechselt. Wenn im vorhandenen Attributblock für das erste Attribut auch kein Wert vorhanden ist, wird der boolean auf false gesetzt. Anschließend wird zum nächsten Attribut gewechselt. Dieses Verfahren wird sooft wiederholt, bis alle Attribute des Blockes abgearbeitet sind. Dabei wird einmal statt der Funktion `undefNum` die Funktion `undefAln` benutzt.

Funktionsname:

undefAln undefAln

Signatur:

undefAln:: attrAln -> boolean.

attrAln: Zu prüfendes Attribut

Beschreibung:

Prüft, ob ein Aln-Attribut noch undefiniert ist

Implementierung:

Beim erzeugen der Attributblöcke werden alle Aln-Attribute zuerst auf 'undef' gesetzt. Hier geprüft, ob das Attribut diesen Wert hat. Wenn ja, wird der boolean auf true gesetzt.

Funktionsname:

getDocuAttrs getDocuAttrs

Signatur:

getDocuAttrs:: pr -> err -> doc -> objID -> objID -> view -> docu -> (pr,err,docu).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Ursprungsobjekt

objID: aktuelles Objekt

string: Sicht, für die abgefragt wird

docu: Dokumentattributblock, der aufgefüllt werden soll

(pr,err,docu): Antwort

Beschreibung:

Mit dieser Funktion wird die Vererbung der Absatzattribute der Objekte realisiert, bzw. 'Aufgelöst'. Der Absatzattributblock für das Ursprungsobjekt wird aus der pr ausgelesen. Wenn der Attributblock nicht komplett ist, wird das Vorgängerobjekt bestimmt, und es werden durch rekursiven Aufruf der Funktion die Attribute aus dem Attributblock des Vorgängerobjektes ausgelesen.

Abhängigkeiten:

getObjectElementtype, getViewAttr, getDocAt, getParent

Implementierung:

Als erstes wird die aktuelle ID geprüft. Wenn diese gleich 0 ist, ist die Wurzel des Dokumentes erreicht, und es gibt kein Vorgängerobjekt mehr. Das bedeutet, das mindestens ein Attribut durch keinen der in Frage kommenden Attributblöcke mit einem Wert belegt werden konnte. In diesem Fall wird geprüft, ob der Sichtname eventuell bereits auf 'ALL' gesetzt ist. Wenn nicht, wird getDocuAttrs mit der Sicht 'ALL', in deren Attributblock alle Werte geschrieben werden, für die keine spezielle Sicht angegeben wurden, aufgerufen. Wenn der Sichtname jedoch bereits auf 'ALL' gesetzt ist, wird als letzte Möglichkeit die Attribute nicht undefineiert zu lassen, die Funktion `getDocAt` mitdefDocument, einem mit Defaultwerten belegten Dokumentattributblock, aufgerufen, und die `err` wird um eine entsprechende Fehlermeldung ergänzt.
Wenn die aktuellen ObjektID nicht 0 ist, wird zuerst der boolean, der anzeigt, ob der Attributblock komplett mit allen erforderlichen Werten belegt ist auf true gesetzt. Danach wird mit der über `ISTypes` transitiv aus `Document` importierten Funktion `getObjElementtype`

der zu der aktuellen ObjektId gehörige, das entsprechende PR-Objekt bezeichnende elementtype bestimmt. Damit und anderen der Funktion übergebenen Argumenten wird mittels `getViewAttr` der gewünschte View-Attributblock aus der `pr` geholt. Aus diesem wird mit den Sepataroren `vatAttr` und `atDocument` der dann aktuelle Dokumentattributblock geholt. Mit diesem wird dann mittels der Funktion `getDocAt` versucht, die im neuen Attributblock noch undefinierten Attribute mit Werten zu belegen. Nach Aufruf von `getDocAt` wird über die Prüfung des boolean bestimmt, ob der Attributblock komplett ist. Ist das der Fall, wird er zusammen mit der `pr` und der `err` zurückgegeben. Falls der boolean durch getDocAt auf false gesetzt wurde, wird `getDocuAttrs` mit einer neuen, durch die über `ISTypes` transitiv aus `Document` importierte Funktion `getParent` bestimmten aktuellen ID rekursiv aufgerufen.

Funktionsname:

getDocAt getDocAt

Signatur:

getDocAt:: docu -> docu -> boolean -> (docu,boolean).

docu: neuer Attributblock, in den ggf. geschrieben wird

docu: vorhandener Attributblock, aus dem gelesen wird

boolean: Anzeige, ob alle Attribute gesetzt sind

(docu,boolean): Antwort

Beschreibung:

Schreibt Attribute in den ProvParagraph, wenn diese dort noch nicht gesetzt sind

Abhängigkeiten:

undefKeepwith, undefPos

Implementierung:

Zuerst wird für das erste Attribut des Blocks im neuen Attributblock mit `undefKeepwith` geprüft, ob es schon gesetzt ist, oder nicht. Wenn es gesetzt ist, wird es nicht verändert, und es wird zum zweiten Attribut gewechselt. Ist das erste Attribut noch nicht mit einem Wert belegt, wird als Nächstes mit `undefKeepwith` geprüft, ob das erste Attribut im vorhandenen Attributblock mit einem Wert belegt ist. Ist das der Fall, wird dieser Wert in den neuen Block übernommen, der boolean bleibt unverändert, und es wird zum zweiten Attribut gewechselt. Wenn im vorhandenen Block für das erste Attribut auch kein Wert vorhanden ist, wird der boolean auf false gesetzt. Anschließend wird zum zweiten Attribut gewechselt und das Verfahren wird wiederholt, dabei wird diesmal statt der Funktion `undefKeepwith` die Funktion `undefPos` benutzt.

Funktionsname:

getFontInf getFontInf

Signatur:

getFontInf:: pr -> err -> doc -> objID -> objID -> view -> prfont -> (pr,err,prfont).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Ursprungsobjektes

objID: aktuelles Objektes

string: Sicht, für die abgefragt wird

prfont: Fontattributblock, der aufgefüllt werden soll

(pr,err,prfont): Antwort

Beschreibung:

Mit dieser Funktion wird die Vererbung der Fontattribute der Objekte realisiert, bzw. 'aufgelöst'. Der Fontattributblock für das Ursprungsobjekt wird aus der pr ausgelesen. Wenn der Attributblock nicht komplett ist, wird das Vorgängerobjekt bestimmt, und es werden durch rekursiven Aufruf der Funktion die Attribute aus dem Attributblock des Vorgängerobjektes ausgelesen.

Abhängigkeiten:

getObjElementtype, getViewattr, makeErr, getPageAT

Implementierung:

Als erstes wird die aktuelle ID geprüft. Wenn diese gleich 0 ist, ist die Wurzel des Dokumentes erreicht, und es gibt kein Vorgängerobjekt mehr. Das bedeutet, das mindestens ein Attribut durch keinen der in Frage kommenden Attributblöcke mit einem Wert belegt werden konnte. In diesem Fall wird geprüft, ob der Sichtname eventuell bereits auf 'ALL' gesetzt ist. Wenn nicht, wird getFontInf mit der Sicht 'ALL', in deren Attributblock alle Werte geschrieben werden, für die keine spezielle Sicht angegeben wurden, aufgerufen. Wenn der Sichtname jedoch bereits auf 'ALL' gesetzt ist, wird als letzte Möglichkeit die Attribute nicht undefiniert zu lassen, die Funktion getFinfo mit defFont, einem mit Defaultwerten belegten Seitenattributblock, aufgerufen, und die `err` wird um eine entsprechende Fehlermeldung ergänzt.
Wenn die aktuellen ObjektID nicht 0 ist, wird zuerst der boolean, der anzeigt, ob der Attributblock komplett mit allen erforderlichen Werten belegt ist auf true gesetzt. Danach wird mit der über `ISTypes` transitiv aus Document importierten Funktion `getObjElementtype` der zu der aktuellen ObjektId gehörige, das entsprechender PR-Objekt bezeichnende elementtype bestimmt. Damit und anderen der Funktion übergebenen Argumenten wird mittels `getViewAttr` der gewünschte View-Attributblock aus der `pr` geholt. Aus diesem wird mit den Sepataroren `vatAttr` und `atFont` der dann aktuelle Fontattributblock geholt. Mit

diesem wird dann mittels der Funktion `getFinfo` versucht, die im neuen Fontattributblock noch undefinierten Attribute mit Werten zu belegen. Nach Aufruf von `getFinfo` wird über die Prüfung des boolean bestimmt, ob der Attributblock komplett ist. Ist das der Fall, wird er zusammen mit der `pr` und der `err` zurückgegeben. Falls der boolean durch `getFinfo` auf false gesetzt wurde, wird `getFontInf` mit einer neuen, durch die über `ISTypes` transitiv aus `Document` importierte Funktion `getParent` bestimmten, aktuellen ID rekursiv aufgerufen.

Funktionsname:

getFinfo getFinfo

Signatur:

getFinfo:: prfont -> prfont -> boolean -> (prfont,boolean).

paragraph: neuer Attributblock, in den ggf. geschrieben wird

paragraph: vorhandener Attributblock, aus dem gelesen wird

boolean: Anzeige, ob alle Attribute gesetzt sind

(prfont,boolean): Antwort

Beschreibung:

Schreibt Attribute in den neuen Attributblock, wenn diese dort noch nicht gesetzt sind

Abhängigkeiten:

undefNum

Implementierung:

Zuerst wird für das erste Attribut des neuen Blocks mit einer Gleichheitsprüfung auf einen leeren String geprüft, ob es schon gesetzt ist, oder nicht. Wenn es gesetzt ist, wird es nicht verändert, und es wird zum zweiten Attribut gewechselt. Ist das erste Attribut noch nicht mit einem Wert belegt, wird als Nächstes mit mit Gleichheit auf einen leeren String geprüft, ob das erste Attribut im vorhandenen Attributblock mit einem Wert belegt ist. Ist das der Fall, wird dieser Wert in den neuen Attributblock übernommen, der boolean bleibt unverändert, und es wird zum zweiten Attribut gewechselt. Wenn im vorhandenen Attributblock für das erste Attribut auch kein Wert vorhanden ist, wird der boolean auf false gesetzt. Anschließend wird zum nächsten Attribut gewechselt. Dieses Verfahren wird sooft wiederholt, bis alle Attribute des Blockes abgearbeitet sind. Dabei wird einmal die Gleichheit mit der leeren Liste geprüft und einmal die Funktion `undefNum` benutzt.

Funktionsname:

convertFont convertFont

Signatur:

convertFont:: prfont -> font.

prfont: zu konvertierender Font

font: konvertierter Font

Beschreibung:

Wandelt einen FontAttributblock in für X verständliche Werte um.

Abhängigkeiten:

filterXFontnames, filterXStyles

Implementierung:

Zuerst wird mit den Separatoren `fontsize` und `vReal` der Real-Wert aus dem `attrNum` der `prfont` geholt und mit der Funktion `integer` in einen integer gewandelt. Dann wird mit dem Separator `fontname` der Fontname aus dem `prfont` geholt und mit `filterXFontnames` in einen für X verständlichen Wert umgewandelt. Mit dem Separator `fontstyle` und der Funktion `filterXStyles` passiert das gleiche für die Fontsytles.

Funktionsname:

filterXStyles filterXStyles

Signatur:

filterXStyles:: string -> string.

string: zu wandelnder String

string: gewandelter String

Beschreibung:

Wandelt einen Fontstylestring in das für X benötigte Format.

Implementierung:

Die Funktion prüft den eingegebenen String und setzt den entsprechenden Ausgabestring. Wenn für den Eingabestring keine Entsprechung gefunden werden kann, wird der Ausgabestring auf DEFAULT gesetzt.

Funktionsname:

filterXFontnames filterXFontnames

Signatur:

filterXFontnames:: string -> string.

string: zu wandelnder String

string: gewandelter String

Beschreibung:

Wandelt einen Fontnamestring in das für X benötigte Format.

Implementierung:

Die Funktion prüft den eingegebenen String und setzt den entsprechenden ausgabestring. Wenn für den Eingabestring keine Entsprechung gefunden werden kann, wird der Ausgabestring auf DEFAULT gesetzt.

Funktionsname:

getCountAttrs getCountAttrs

Signatur:

getCountAttrs:: pr -> err -> doc -> objID -> objID -> view -> counter -> (pr,err,counter).

pr: Alte `pr`

err: Alte `err`

doc: Dokumentstruktur

objID: Identifikation des Ursprungsobjektes

objID: Identifikation des aktuellen Objektes

string: Sicht, für die abgefragt wird

counter: Seitenattributblock, der aufgefüllt werden soll

(pr,err,counter): Antwort

Beschreibung:

Mit dieser Funktion wird die Vererbung der Zählerattribute der Objekte realisiert, bzw. 'Aufgelöst'.
Der Zählerattributblock für das Ursprungsobjekt wird aus der pr ausgelesen. Wenn der Attributblock nicht komplett ist, wird das Vorgängerobjekt bestimmt, und es werden durch rekursiven Aufruf der Funktion die Attribute aus dem Attributblock des Vorgängerobjektes ausgelesen.

Abhängigkeiten:

getObjElementtype, getViewAttr, makeErr, getPageAT

Implementierung:

Als erstes wird die aktuelle ID geprüft. Wenn diese gleich 0 ist, ist die Wurzel des Dokumentes erreicht, und es gibt kein Vorgängerobjekt mehr. Das bedeutet, das mindestens ein Attribut durch keinen der in Frage kommenden Attributblöcke mit einem Wert belegt werden konnte. In diesem Fall wird geprüft, ob der Sichtname eventuell bereits auf 'ALL' gesetzt ist. Wenn nicht, wird `getCountAttrs` mit der Sicht 'ALL', in deren Attributblock alle Werte geschrieben werden, für die keine spezielle Sicht angegeben wurden, aufgerufen. Wenn der Sichtname jedoch bereits auf 'ALL' gesetzt ist, wird als letzte Möglichkeit die Attribute nicht undefineiert zu lassen, die Funktion `getCountAt` mit `defCounter`, einem mit Defaultwerten belegten Zählerattributblock, aufgerufen, und die `err` wird um eine entsprechende Fehlermeldung ergänzt.
Wenn die aktuellen ObjektID nicht 0 ist, wird zuerst der boolean, der anzeigt, ob der Attributblock komplett mit allen erforderlichen Werten belegt ist auf true gesetzt. Danach wird mit der über `ISTypes` transitiv aus Document importierten Funktion `getObjElementtype` der zu der aktuellen ObjektId gehörige, das entsprechender PR-Objekt bezeichnende elementtype bestimmt. Damit und anderen der Funktion übergebenen Argumenten wird mittels `getViewAttr` der gewünschte View-Attributblock aus der `pr` geholt. Aus diesem wird mit den Sepataroren `vatAttr` und `atCounter` der dann aktuelle Zählerattributblock geholt. Mit diesem wird dann mittels der Funktion `getCountAt` versucht, die in Counter noch undefinierten Attribute mit Werten zu belegen. Nach Aufruf von `getCountAt` wird über die Prüfung des boolean, ob der Attributblock komplett ist. Ist das der Fall, wird er zusammen mit der `pr` und der `err` zurückgegeben. Falls ber boolean durch `getCountAt` auf false gesetzt wurde, wird `getCountAtttrs` mit einer neuen, durch die über `ISTypes` transitiv aus `Document` importierte Funktion `getParent` bestimmten aktuellen ID rekursiv aufgerufen.

Funktionsname:

getCountAt getCountAt

Signatur:

getCountAt:: counter -> counter -> boolean -> (counter,boolean).

counter: neuer Attributblock, in den ggf. geschrieben wird

counter: vorhandener Attributblock, aus dem gelesen wird

boolean: Anzeige, ob alle Attribute gesetzt sind

(counter,boolean): Antwort

Beschreibung:

Schreibt Attribute in den neuen Attributblock, wenn diese dort noch nicht gesetzt sind

Abhängigkeiten:

isMtElement, undefNum

Implementierung:

Zuerst wird für das erste Attribut des Blocks im neuen Attributblock mit `isMtElement` geprüft, ob es schon gesetzt ist, oder nicht. Wenn es gesetzt ist, wird es nicht verändert, und es wird zum zweiten Attribut gewechselt. Ist das erste Attribut noch nicht mit einem Wert belegt, wird als Nächstes mit `isMtElement` geprüft, ob das erste Attribut in dem vorhandenen Attributblock mit einem Wert belegt ist. Ist das der Fall, wird dieser Wert in den neuen Block übernommen, der boolean bleibt unverändert, und es wird zum zweiten Attribut gewechselt. Wenn in dem vorhandenen Block für das erste Attribut auch kein Wert vorhanden ist, wird der boolean auf false gesetzt. Anschließend wird zum zweiten Attribut gewechselt. Dieses Verfahren wird sooft wiederholt, bis alle Attribute des Blockes abgearbeitet sind. Für die Prüfung, ob Attribute schon gesetzt sind oder nicht, wird auch die Funktionen `undefSym` verwandt. Bei den integers wird mit einem Vergleich auf 0 ermittelt, ob sie schon gesetzt sind, oder nicht.

Funktionsname:

undefPos undefPos

Signatur:

undefPos:: attrPos -> boolean.

attrPos: Zu prüfendes Attribut

boolean: Antwort

Beschreibung:

Prüft, ob ein Pos-Attribut noch undefiniert ist

Implementierung:

Beim erzeugen der Attributblöcke werden alle Pos-Attribute zuerst auf 'undef' gesetzt. Hier geprüft, ob das Attribut diesen Wert hat. Wenn ja, wird der boolean auf true gesetzt.

Funktionsname:

undefKeepwith undefKeepwith

Signatur:

undefKeepwith:: attrKeepwith -> boolean.

attrKeepwith: Zu prüfendes Attribut

boolean: Antwort

Beschreibung:

Prüft, ob ein Keepwith-Attribut noch undefiniert ist

Implementierung:

Beim erzeugen der Attributblöcke werden alle Keepwith-Attribute zuerst auf 'undef' gesetzt. Hier geprüft, ob das Attribut diesen Wert hat. Wenn ja, wird der boolean auf true gesetzt.

Funktionsname:

undefSym undefSym

Signatur:

undefSym:: attrSym -> boolean.

attrSym: Zu prüfendes Attribut

boolean: Antwort

Beschreibung:

Prüft, ob ein Sym-Attribut noch undefiniert ist

Implementierung:

Beim erzeugen der Attributblöcke werden alle Sym-Attribute zuerst auf 'undef' gesetzt. Hier geprüft, ob das Attribut diesen Wert hat. Wenn ja, wird der boolean auf true gesetzt.

Funktionsname:

isMtElement isMtElement

Signatur:

isMtElement:: elementtype -> boolean.

elementtype: Zu prüfender Elementtype

Beschreibung:

Prüft, ob ein Element gleich dem mtElementtype ist.

Implementierung:

Wenn der übergebene elementtype gleich dem mtElementtype ist, wird true zurückgegeben, andernfalls false.

Funktionsname:

makeErr makeErr

Signatur:

makeErr:: integer -> string -> err -> err.

integer: Fehlernummer

string: Fehlertext

err: Alte `err`

Beschreibung:

Erzeugt eine Fehlermeldung und fügt sie in die `err` ein.

Abhängigkeiten:

announceError, crateError

Implementierung:

Abhängig von der Fehlernummer wird ein Fehlertext und eine Fehlerart generiert, die dann mittels der von `ISTypes` importierten Funktionen announceError und createError in die `err` eingefügt werden.

Funktionsname:

getSuccObjectList getSuccObjectList

Signatur:

getSuccObjectList:: pr -> err -> elementtype -> view -> (boolean, integer, viewAttr, viewAttrList, succObjectList, objectDictElement, err).

pr: Aktuelle `pr`

err: Aktuelle err

elementtype: Aktuelles Objekt

view: Aktuelle Sicht

(boolean,integer,viewAttr,viewAttrList,succObjectList,objectDictElement,err).: Antwort

Beschreibung:

Holt für das angegebene Objekt und die angegebene Sicht die Liste der Folgeobjekte aus der pr

Abhängigkeiten:

getViewAttr

Implementierung:

Mit getViewAttr wird aus der `pr` der dem Objekt und der Sicht entsprechende Attributblock geholt. Aus diesem wird anschließend mit dem Separator vatSuccObjectList die geltende Flgeobjektliste geholt und zurückgegeben.

Funktionsname:

mkSuccObjectlist mkSuccObjectlist

Signatur:

mkSuccObjectlist:: elementtypelist -> succObjectList.

elementtypelist: Liste, die zur SuccObjetList gewandelt werden soll

succObjectList: Antwort

Beschreibung:

Wandelt eine elementtypeList in eine SuccObjectList

Abhängigkeiten:

mSol

Implementierung:

Die Funktion ist mSol vorgeschaltet. Es wird eine leere Liste erzeugt. Mit der Inlist und der leeren Liste wird mSol aufgerufen.

Funktionsname:

mSol mSol

Signatur:

mSol:: elementtypelist -> succObjectList -> succObjectList.

elementtypelist: Vorhandene Liste mit elementtypes

succObjectList: neue Liste mit SuccObjects

succObjectList: Antwort

Beschreibung:

Wandelt eine elTypeList in eine SuccObjectList

Implementierung:

Zuerst wird geprüft, ob der Head der vorhandenen Liste gleich der leeren Liste ist. Ist das der Fall, so wird die `succObjectList` unverändert zurückgegeben. Wenn das Head der vorhandenen Liste nicht die leere Liste ist, daraus ein `succObject` kreiert, welches anschließend mit der Listenoperation `postfix` an die neue Liste angehängt wird. Danach wird `mSol` mit dem Tail der vorhandenen Liste und der neuen `succObjectList` rekursiv aufgerufen.

Funktionsname:

posType posType

Signatur:

posType:: string -> err -> (attrPos,err).

string: Eingabestring

err: Alte `err`

(attrPos,err): Antwort

Beschreibung:

Wandelt einen String in ein AttrPos

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

alnType alnType

Signatur:

alnType:: string -> err -> (attrAln,err).

string: Eingabestring

err: Alte `err`

(attrAln,err): Antwort

Beschreibung:

Wandelt einen String in ein attrAln

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

symType symType

Signatur:

symType:: string -> err -> (attrSym,err).

string: Eingabestring

err: Alte `err`

(attrSym,err): Antwort

Beschreibung:

Wandelt einen String in ein attrSym.

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

ordType ordType

Signatur:

ordType:: string -> err -> (attrOrd,err).

string: Eingabestring

err: Alte `err`

(attrOrd,err): Antwort

Beschreibung:

Wandelt einen String in einen attrOrd.

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

ppsType ppsType

Signatur:

ppsType:: string -> err -> (attrPps,err).

string: Eingabestring

err: Alte `err`

(attrPps,err): Antwort

Beschreibung:

Wandelt einen String in einen attrPos.

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

flgType flgType

Signatur:

flgType:: string -> err -> (attrFlg,err).

string: Eingabestring

err: Alte `err`

(attrFlg,err): Antwort

Beschreibung:

Wandelt einen String in einen attrFlg.

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

unitType unitType

Signatur:

unitType :: string -> err -> (attrUnit,err).

string: Eingabestring

err: Alte `err`

(attrUnit,err): Antwort

Beschreibung:

Wandelt einen String in einen attrUnit.

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

makeBoxType makeBoxType

Signatur:

makeBoxType :: string -> err -> (boxtype,err).

string: Eingabestring

err: Alte `err`

(boxtype,err): Antwort

Beschreibung:

Wandelt einen String in einen boxtype.

Abhängigkeiten:

stringequal

Implementierung:

Mit `stringequal` wird geprüft, ob der übergebene String mit einem der gültigen Schlüsselworte für die Attribute übereinstimmt. Ist das der Fall, wird der entsprechende Attributwert als Tupel mit der unveränderten `err` zurückgegeben. Ist das nicht der Fall, wird, nachdem alle Möglichkeiten geprüft worden sind, der Attributwert auf `undef` gesetzt, eine entsprechende Fehlermeldung in die `err` eingefügt und beides als Tupel zurückgegeben.

Funktionsname:

selectView selectView

Signatur:

selectView:: viewAttrList -> view -> integer -> (integer,viewAttr).

viewAttrList: Liste der Sicht-Attribute

view: ausgewählte Sicht

integer: Startindex

interger,viewAttr: Antwort

Beschreibung:

Holt aus der Liste aller Sicht-Attribute den für die ausgewählte Sicht gültigen Attributblock.

Abhängigkeiten:

mtViewAttr, stringequal

Implementierung:

Zuerst wird geprüft, ob die Liste leer ist. Ist das der Fall, wird der Index auf –1 gesetzt und ein mittels `mtViewAttr` erzeugter leerer Attributblock zurückgegeben. Wenn die Liste nicht leer ist, wird aus Head der Liste mit dem Selektor `vatView` der Sichtname genommen und mittels `stringEqual` mit dem als Argument übergebenen Sichtnamen verglichen. Sind diese gleich, wird Head der Liste zusammen mit dem aktuellen Index zurückgegeben. Sind die beiden Strings nicht gleich, wird `selectView` mit Tail der Liste rekursiv aufgerufen.

Funktionsname:

getViewAttr getViewAttr

Signatur:

getViewAttr:: pr -> err -> elementtype -> view ->
(boolean,integer,viewAttr,viewAttrList,objectDictElement,err).

pr: alte `pr`

err: alte `err`

elementtype: Objektbezeichner

view : Sicht

(boolean,integer,viewAttr,viewAttrList,objectDictElement,err) : Antwort

Beschreibung:

Holt für das angegebene Objekt und die angegebene Sicht den Block mit den entsprechenden Attributen aus der pr.

Abhängigkeiten:

getPrObject, selectView, mtViewAttr, mtobjectDictElement

Implementierung:

Zuerst wird mit `getPrObject` das angesprochene Dictionary-Element geholt. Wenn diese Aktion erfolgreich war — Ok ist true —, dann wird aus diesem mit `selectView` der Attributblock für die angegebene Sicht geholt. War diese Aktion ebenfalls erfolgreich – der Index ist nicht – 1 —, dann wird das Rückgabetupel zusammengestellt, und die entsprechende `viewAttrList` wird mittels des Selektors `obViewAtList` aus dem aktuellen Dictionary-Element herausgeholt. Wenn bei `selectView` ein Fehler auftritt, wird das Rückgabetupe ähnlich zusammengestellt, jedoch wird ein mittels `mtViewArttr` erzeugtes `viewAttr` zurückgegeben,und es wird mit `makeErr` eine entsprechende Fehlermeldung in die err eingefügt. Wenn bereits bei `getPrObject` ein Fehler auftrat, wird außer dem leeren `viewAttr` eine leere Liste als `viewAttrList` und ein leeres Dictionary-Element und die von `getPrObject` veränderte err zurückgegeben.

Funktionsname:

getPrObject getPrObject

Signatur:

getPrObject:: pr -> err -> elementtype -> (boolean,objectDictElement,err).

pr: Alte `pr`

err: Alte `err`

elementtype: akt. Objekt

(boolean,objectDictElement,err): Antwort

Beschreibung:

Holt aus dem Dictionary der vorhandenen Objekte ein durch den elementtype spezifiziertes Objekt. Die Funktion gibt ein true, das angesprochene Dictionary-Element und die unveränderte `err` zurueck, falls das Objekt existiert. Falls nicht, wird ein false, ein leeres Dictitionary-Element und ein um eine entsprechende Fehelermeldung ergänztes `err` zurückgegeben.

Abhängigkeiten:

mtobjectDictElement

Implementierung:

Zuerst wird das Dictionary aus der `pr` geholt. Dann wird geprüft, ob ein Dictionary-Eintrag, der mit dem als Argument übergebenen `elementtype` erkannt werden kann, existiert. Falls ja, wird ein true und das entsprechende `objectDictelement` zurückgegeben. Falls nein, kommt ein false und ein mit `mtobjectDictElement` erzeugtes leeres objectDictelement zurück.

Funktionsname:

selectView selectView

Signatur:

selectView:: viewAttrList -> view -> integer -> (integer,viewAttr).

viewAttrList: Liste der Sichtattribute

view: kat. Sicht

integer: Index

integer,viewAttr: Antwort

Beschreibung:

Holt die Attribute für die angegebene Sicht aus der Gesamtliste

Vor- und Nachbedingungen:

Außer der ViewAttrList und dem Viewnamen muß noch ein integer als Dummy übergeben werden.

Abhängigkeiten:

ismt, mtViewAttr

Implementierung:

Zuerst wird mit `ismt` geprüft, ob schon überhaupt Werte in der Gesamtattributliste stehen. Wenn nicht, wird der Index auf -1 gesetzt und ein mit `mtViewAttr` erzeugtes, leeres ViewAttr zurückgegeben. Falls die Gesamtattributliste nicht leer ist, werden die Elemente der Liste nach dem als Argument übergebenen string des Sichtnamens mit `selectView` rekursiv durchsucht. Wird ein dem Sichtnamen entsprechendes `viewAttr` in der Liste gefunden, so wird der Index und dieses `viewAttr` zurückgegeben.

Funktionsname:

mtAttr mtAttr

Signatur:

mtAttr:: attrs.

Beschreibung:

Erzeugt einen leeren Attributblock.

Funktionsname:

mtAtDocument mtAtDocument

Signatur:

mtAtDocument:: docu.

Beschreibung:

Erzeugt ein leeres Document-Attribut..

Funktionsname:

mtAtPage mtAtPage

Signatur:

mtAtPage:: page.

Beschreibung:

Erzeugt ein leeres Page-Attribut.

Funktionsname:

mtAtParagraph mtAtParagraph

Signatur:

mtAtParagraph:: paragraph.

Beschreibung:

Erzeugt ein leeres Paragraphen-Attribut.

Funktionsname:

mtAtCounter mtAtCounter

Signatur:

mtAtCounter:: counter.

Beschreibung:

Erzeugt ein leeres Counter-Attribut.

Funktionsname:

mtAtCatalogs mtAtCatalogs

Signatur:

mtAtCatalogs:: catalogs.

Beschreibung:

Erzeugt ein leeres Catalog-Attribut.

Funktionsname:

mtAtFont mtAtFont

Signatur:

mtAtFont:: prfont.

Beschreibung:

Erzeugt einen leeres Font-Attribut.

Funktionsname:

mtAtNotes mtAtNotes

Signatur:

mtAtNotes:: notes.

Beschreibung:

Erzeugt ein leeres Noten-Attribut.

Funktionsname:

mtViewAttr mtViewAttr

Signatur:

mtViewAttr:: viewAttr.

Beschreibung:

Erzeugt einen leeren Sicht-Attributblock.

Funktionsname:

mtobjectDictElement mtobjectDictElement

Signatur:

mtobjectDictElement:: objectDictElement.

Beschreibung:

Erzeugt ein leeres Dictionary-Element

Funktionsname:

defDocument defDocument

Signatur:

defDocument:: docu.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

Funktionsname:

defPage defPage

Signatur:

defPage:: page.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

Funktionsname:

defParagraph defParagraph

Signatur:

defParagraph:: paragraph.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

Funktionsname:

defCounter defCounter

Signatur:

defCounter:: counter.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

Funktionsname:

defCatalogs defCatalogs

Signatur:

defCatalogs:: catalogs.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

Funktionsname:

defFont defFont

Signatur:

defFont:: prfont.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

Funktionsname:

defNotes defNotes

Signatur:

defNotes:: notes.

Beschreibung:

Erzeugt einen mit Defaultwerten belegten Attributblock

1.6 IS

1.6.1 Funktionalität

Dieses Modul stellt das Environment `env` zur Verfügung. Es importiert die Definitionen der Datenstrukturen, die im Environment gespeichert werden sollen und exportiert Zugriffsfunktionen zum Lesen und Schreiben dieser Datenstrukturen ins Environment. Der Zugriff auf Teile der importierten Strukturen spielt hier keine Rolle; er wird den jeweiligen Modulen überlassen, in denen sie definiert sind.
Außerdem sind die Funktionen zur Fehlerbehandlung, die in jedem Schnittstellenmodul zu anderen Systemteilen gebraucht werden, hier implementiert.

1.6.2 Entwurf

Für jede im Environment gespeicherte Datenstruktur gibt es eine Funktion zum Auslesen und eine Funktion zum Hineinschreiben. Die Gestaltung der Operationen zur Fehlerbehandlung entspricht der schon im `ISTypes`–Modul erläuterten (siehe Seite 13).

1.6.3 Öffentliche Schnittstellen

1.6.3.1 Sorten

Sortenname:

env env

Signatur:

env:: Opaquer Typ. Siehe Lokalteil.

Beschreibung:

Dieser Typ stellt den Speicher für alle Daten des Programms dar. Zur Laufzeit existiert genau ein Wert dieses Typs.

1.6.3.2 Funktionen

Funktionsname:

mt mt

Signatur:

mt:: system -> env.

system: Das „System"; die Verbindung zur Programmumgebung.

env: Die leere `env`–Struktur.

Beschreibung:

Diese Funktion initialisiert sämtliche Datenstrukturen und speichert diese zusammen mit dem `system`, das beim Aufruf der 'goal'–Funktion im Hauptmodul kreiert wurde. Während des Programmlaufs gibt es nur einen Wert vom Typ `system`; es handelt sich dabei um eine ASpecT–eigene Struktur, die zur Verbindung mit dem Betriebssystem und anderen Prozessen benötigt wird.

Implementierung:

Die Initialisierungen werden mit Aufrufen an die Module durchgeführt, die die gespeicherten Strukturen definieren.

Alle nun folgenden Funktionen, deren Namen mit `getCookie` oder `setCookie` beginnen, sind über Selektoren/Modifikatoren realisiert. Die Beschreibung der Implementierung ist daher im einzelnen unterlassen worden.

Funktionsname:

getCookieFS getCookieFS

Signatur:

getCookieFS:: env -> fs.

env: Das Environment.

fs: Die Formatierer–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Formatierer–Datenstruktur zurück.

Funktionsname:

setCookieFS setCookieFS

Signatur:

setCookieFS:: env -> fs -> env.

env: Das Environment.

fs: Die Formatierer–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Formatierer–Datenstruktur durch die übergebene.

Funktionsname:

getCookieUI getCookieUI

Signatur:

getCookieUI:: env -> ui.

env: Das Environment.

ui: Die Benutzungsoberfläche–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Benutzungsoberfläche–Datenstruktur zurück.

Funktionsname:

setCookieUI setCookieUI

Signatur:

setCookieUI:: env -> ui -> env.

env: Das Environment.

ui: Die Benutzungsoberfläche–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Datenstruktur der Benutzungsoberfläche durch die übergebene.

Funktionsname:

getCookieOS getCookieOS

Signatur:

getCookieOS:: env -> os.

env: Das Environment.

os: Die Ausgabe–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Ausgabe–Datenstruktur zurück.

Funktionsname:

setCookieOS setCookieOS

Signatur:

setCookieOS:: env -> os -> env.

env: Das Environment.

os: Die Ausgabe–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Ausgabe–Datenstruktur durch die übergebene.

Funktionsname:

getCookieDOC getCookieDOC

Signatur:

getCookieDOC:: env -> doc.

env: Das Environment.

doc: Die Dokument–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Dokument–Datenstruktur zurück.

Funktionsname:

setCookieDOC setCookieDOC

Signatur:

setCookieDOC:: env -> doc -> env.

env: Das Environment.

doc: Die Dokument–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Dokument–Datenstruktur durch die übergebene.

Funktionsname:

getCookieDTD getCookieDTD

Signatur:

getCookieDTD:: env -> dtd.

env: Das Environment.

dtd: Die DTD–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte DTD–Datenstruktur zurück.

Funktionsname:

setCookieDTD setCookieDTD

Signatur:

setCookieDTD:: env -> dtd -> env.

env: Das Environment.

dtd: Die DTD–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte DTD–Datenstruktur durch die übergebene.

Funktionsname:

getCookiePR getCookiePR

Signatur:

getCookiePR:: env -> pr.

env: Das Environment.

pr: Die PR–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte PR–Datenstruktur zurück.

Funktionsname:

setCookiePR setCookiePR

Signatur:

setCookiePR:: env -> pr -> env.

env: Das Environment.

pr: Die PR–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte PR–Datenstruktur durch die übergebene.

Funktionsname:

getCookieElementtypeStruct getCookieElementtypeStruct

Signatur:

getCookieElementtypeStruct:: env -> elementtypeStruct.

env: Das Environment.

elementtypeStruct: Die Elementtype–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Elementtype–Datenstruktur zurück.

Funktionsname:

setCookieElementtypeStruct setCookieElementtypeStruct

Signatur:

setCookieElementtypeStruct:: env -> elementtypeStruct -> env.

env: Das Environment.

elementtypeStruct: Die Elementtype–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Elementtype–Datenstruktur durch die übergebene.

Funktionsname:

getCookieErr getCookieErr

Signatur:

getCookieErr:: env -> err.

env: Das Environment.

err: Die Fehler–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Fehler–Datenstruktur zurück.

Funktionsname:

setCookieErr setCookieErr

Signatur:

setCookieErr:: env -> err -> env.

env: Das Environment.

err: Die Fehler–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Fehler–Datenstruktur durch die übergebene.

Funktionsname:

getCookieISP getCookieISP

Signatur:

getCookieISP:: env -> isp.

env: Das Environment.

isp: Die ISParser–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte ISParser–Datenstruktur zurück. Sie wird vom Modul `ISParser` für temporäre Speicherungen über Funktionsaufrufe hinweg benutzt.

Funktionsname:

setCookieISP setCookieISP

Signatur:

setCookieISP:: env -> isp -> env.

env: Das Environment.

isp: Die ISParser–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte ISParser–Datenstruktur durch die übergebene.

Funktionsname:

i_getSystem i_getSystem

Signatur:

i_getSystem:: env -> system.

env: Das Environment.

system: Die „System“–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte „System“–Datenstruktur zurück.

Funktionsname:

i_setSystem i_setSystem

Signatur:

i_setSystem:: env -> system -> env.

env: Das Environment.

system: Die „System“–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte „System“–Datenstruktur durch die übergebene.

Funktionsname:

i_showGraphENV i_showGraphENV

Signatur:

i_showGraphENV:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion gibt auf „standard-out“ eine einfache graphische Darstellung der Struktur des momentan geladenen Dokumentes aus. Sie ist nur zu Testzwecken eingeführt worden. Näheres ist im `Document`–Modul auf Seite 119 zu finden.

Implementierung:

Hier wird nur eine entsprechende Funktion im `Dokument`–Modul — mit den benötigten Daten versorgt — aufgerufen.

Im folgenden sind Funktionen zur Behandlung von Fehlern aufgeführt, die auf der Datenstruktur `err` mit den im Modul `ISTypes` implementierten Zugriffsoperationen arbeiten. Alle anderen Systemteile benutzen nur die hier eingeführten Funktionen auf dem Environment. Verfügbar gemacht werden sie durch Import in die Globalteile der Schnittstellenmodule.
Ihre Implementierung folgt einem einheitlichen Muster: Es wird die `err`–Struktur aus dem Environment geholt, die entsprechende Funktion aus `ISTypes` aufgerufen und — bei Bedarf — `err` wieder im Environment gesichert.

Funktionsname:

i_createError i_createError

Signatur:

i_createError:: modulName -> errType -> errNumber -> errTexts -> errobject.

modulName: Name des Moduls, in dem der Fehler aufgetreten ist.

errType: Art des Fehlers.

errNumber: Eindeutige Nummer des Fehlers im Modul, das den Fehler meldet.

errTexts: Strings, die genauere Informationen über den Fehler geben.

errobject: Aus den Angaben erzeugtes Fehlerobjekt.

Beschreibung:

Diese Funktion erzeugt aus den übergebenen Daten ein Fehlerobjekt, das mit Hilfe von `i_announceError` dem System gemeldet werden kann.

Funktionsname:

i_readError i_readError

Signatur:

i_readError:: errobject -> (modulName, errType, errNumber, ErrTexts).

errobject: Ein Fehlerobjekt.

(modulName, errType, errNumber, ErrTexts): Der Name des Moduls, in dem der Fehler aufgetreten ist, die Art des Fehlers, die eindeutige Nummer des Fehlers im Modul, das den Fehler meldet sowie Zeichenketten, die genauere Informationen über den Fehler enthalten.

Beschreibung:

Diese Funktion liest aus einem Fehlerobjekt dessen Komponenten und gibt sie zurück.

Funktionsname:

i_announceError i_announceError

Signatur:

i_announceError:: errobject -> env -> env.

errobject: Ein Fehler.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion hängt den gemeldeten Fehler in die Datenstruktur ein, so daß er bei einem nachfolgenden `i_getLastError` als zuletzt aufgetretener Fehler angezeigt würde.

Funktionsname:

i_getLastError i_getLastError

Signatur:

i_getLastError:: env -> errobject.

env: Das Environment.

errobject: Der zuletzt gemeldete Fehler.

Beschreibung:

Diese Funktion gibt das jüngste Fehlerobjekt (also den zuletzt gemeldeten Fehler) zurück. Der Fehler bleibt in der Datenstruktur, bis er mit `i_removeError` gelöscht wurde. Ist kein Fehler vorhanden, dann wird ein `errobject` zurückgegeben, dessen `errType` 'none' ist; die anderen Komponenten sind mit belanglosen Werten gefüllt und sollten nicht ausgewertet werden.

Funktionsname:

i_removeError i_removeError

Signatur:

i_removeError:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion löscht das jüngste Fehlerobjekt in der Datenstruktur. Ist kein Fehler gemeldet, passiert nichts, was für den Aufrufer transparent bleibt.

Funktionsname:

i_maxErrObjects i_maxErrObjects

Signatur:

i_maxErrObjects:: env -> errobjectList.

env: Das Environment.

errobjectList: Liste mit den Fehlern, die den höchsten `errType`–Wert haben.

Beschreibung:

Diese Funktion liefert die Fehler mit dem höchsten `errType` von allen gemeldeten Fehlern. Die Wertigkeit der Fehlertypen ergibt sich durch die Reihenfolge bei der Definition dieses Datentyps: Am höchsten ist `fatal`, am niedrigsten `none`.

Alle Fehler der gelieferten Liste haben denselben `errType`–Wert. War kein Fehler vorhanden, dann wird eine Liste mit genau einem 'leeren' Fehlerobjekt zurückgegeben, dessen `errType` `none` ist. Seine restlichen Komponenten sind nicht auszuwerten.

1.6.4 Lokale Implementation

1.6.4.1 Sorten

Sortenname:

env env

Signatur:

env:: env ::= e (v_is :: is) (v_fs :: fs) (v_ui :: ui) (v_os :: os) (v_system :: system) (v_err :: err).

Beschreibung:

Das Environment enthält folgende Datenstrukturen, für die jeweils ein Selektor/Modifikator definiert wird:

is Alle Strukturen des Systemteils Interne Struktur.

fs Struktur des Formatierers (aus Modul `FormatStructure`).

ui Struktur der Benutzungsoberfläche (aus Modul `UIStructure`).

os Struktur der Ausgabe (aus Modul `OutputStructure`).

system ASpecT–Sorte zur Anbindung an die Programm–Umgebung.

err Struktur zur Speicherung der Fehler (aus Modul `ISTypes`).

Sortenname:

is is

Signatur:

is:: i (v_doc :: doc) (v_dtd :: dtd) (v_pr :: pr) (v_elts :: elementtypeStruct) (v_isp :: isp).

Beschreibung:

Diese Sorte bündelt sämtliche Datenstrukturen der Internen Struktur. Dazu gehören:

doc Struktur des Dokumentes (aus Modul `Document`).

dtd Struktur der DTD (aus Modul `DTD`).

pr Struktur der Präsentationsregeln (aus Modul `PresRules`).

elementtypeStruct Struktur zur Speicherung der `elementtype`s (aus Modul `ISTypes`).

isp Struktur zur Speicherung von temporären Werten während des Parse–Vorgangs (aus Modul `ISPStructure`).

1.7 ISFormat

1.7.1 Funktionalität

Dieses Modul stellt die Schnittstelle zwischen der Internen Struktur und dem Formatierer dar. Neben den Funktionen zum Lesen und Speichern der Formatierer–Datenstruktur werden im wesentlichen die vom `PresRules`– und `Document--Modul` zur Verfügung gestellten Funktionen auf das Environment umgesetzt.
Zusätzlich erhält der Formatierer durch transitiven Import Zugriff auf die Fehlerbehandlungsroutinen (aus dem Modul `IS`), die Datentypen für den Informationsaustausch mit den PR (aus dem Modul `ISTypes`) und zugehörige Funktionen, die Initial–Werte für diese Typen liefern (aus dem Modul `PresRules`).

1.7.2 Entwurf

Die vom `PresRules`– und `Document`–Modul definierte Funktionalität wird in seiner Konzeption und Namensgebung komplett übernommen und nur dahingehend modifiziert, daß die Funktionen auf dem Environment `env` arbeiten.
Informationen über die einzelnen Teile des Dokumentes bekommt der Formatierer über drei Funktionen. Zum einen kann das Wurzel–Objekt, das praktisch das gesamte Dokument darstellt ermittelt werden. Zum zweiten erlaubt eine Funktion die Abfrage der Elemente, aus denen ein Objekt besteht. Durch wiederholte Anwendung auf die direkten Nachfolger kann der Formatierer so bis zu den finalen Textobjekten hinunter alle Teile des Dokumentes mit deren Struktur erfassen. Zum dritten existiert eine Funktion zum Auslesen des Textinhaltes eines finalen Objektes.
Anfragen an die Präsentationsregeln folgen dem einfachen Muster der Übergabe einer Objekt–ID und der Herausgabe der zu diesem Objekt (bzw. dessen `elementtype`) gespeicherten Attribute.

1.7.3 Öffentliche Schnittstellen

1.7.3.1 Funktionen

Funktionsname:

i_getCookieFS i_getCookieFS

Signatur:

i_getCookieFS:: env -> fs.

env: Das Environment.

fs: Die Formatierer–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Formatierer–Datenstruktur zurück.

Funktionsname:

i_setCookieFS i_setCookieFS

Signatur:

i_setCookieFS:: env -> fs -> env.

env: Das Environment.

fs: Die Formatierer–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Formatierer–Datenstruktur durch die übergebene.

Alle nachfolgenden Funktionen sind auf gleiche Art und Weise implementiert: Sie holen die notwendigen Datenstrukturen aus dem Environment und rufen praktisch gleichlautende Funktionen in den Basismodulen auf. Dort, wo die Implementation mehr umfaßt, wird dies erläutert.

Die nächsten drei Funktionen erlauben dem Formatierer den Zugriff auf das Dokument.

Funktionsname:

i_getFirstObj i_getFirstObj

Signatur:

i_getFirstObj:: env -> objinfo.

env: Das Environment.

objinfo: Informationen über das Wurzel–Objekt.

Beschreibung:

Diese Funktion liefert Informationen über das Wurzel–Objekt des Dokumentes.

Implementierung:

Da das `Document`–Modul keine Information über den Boxtyp eines Objektes hat, wird der gelieferte `objinfo`–Wert nach einer entsprechenden Anfrage in den PR korrigiert.

Funktionsname:

i_getNextTextObject i_getNextTextObject

Signatur:

i_getNextTextObject:: env -> objID -> objinfolist.

env: Das Environment.

objID: Eine Objekt–ID.

objinfolist: Informationen über die „Kinder“ des Objektes.

Beschreibung:

Die Funktion gibt eine Liste mit Informationen über alle Objekte zurück, die den Inhalt des angegebenen Objektes bilden. Es ist zu beachten, daß nur die direkten Nachfolger einbezogen werden. Die Liste ist in derselben Reihenfolge geordnet, in der die Objekte im Text aufeinander folgen.

Implementierung:

Da das `Document`–Modul keine Information über den Boxtyp eines Objektes hat, werden die gelieferten `objinfo`–Werte nach entsprechenden Anfragen in den PR korrigiert.

Funktionsname:

i_getObjData i_getObjData

Signatur:

i_getObjData:: env -> objID -> string.

env: Das Environment.

objID: Eine Objekt–ID.

string: Der im Objekt enthaltene Text.

Beschreibung:

Diese Funktion liefert zu der Objekt–ID den zugehörigen Text. Ein Aufruf ist nur sinnvoll für finale Textobjekte. Bei allen anderen wird ein leerer String zurückgeliefert.

Es folgen Funktionen für die Abfrage von Präsentationsregeln.

Funktionsname:

i_getOrder i_getOrder

Signatur:

i_getOrder:: env -> elementtype -> elementtypelist -> elementtypelist.

env: Das Environment.

elementtype: Elementtyp des Objektes, das die fraglichen Objekte enthält.

elementtypelist: Liste mit Elementtypen von Objekten, deren korrekte Reihenfolge ermittelt werden soll.

elementtypelist: Laut Präsentationsregeln richtige Reihenfolge der fraglichen Objekte.

Beschreibung:

Diese Funktion bekommt eine Liste mit Objekten herein, so wie sie im Dokument vorkommen und korrigiert deren Reihenfolge nach den Vorgaben der Präsentationsregeln zu dem Vater–Objekt. Dabei kann es vorkommen, daß Objekte aus der Liste gelöscht werden, weil die Regeln sie ausblenden.

Es werden die zu den Objekten gehörenden `elementtype`s statt der IDs übergeben, weil sich die Präsentationsregeln auf SGML–Elemente beziehen und nicht auf spezielle Instanzen.

Funktionsname:

i_getDocumentAttributes　　　　i_getDocumentAttributes

Signatur:

i_getDocumentAttributes:: env -> objID -> docu.

env: Das Environment.

objID: Eine Objekt–ID.

docu: Dokument–Attribute zu diesem Objekt.

Beschreibung:

Diese Funktion liefert die zum Objekt gehörigen Dokument–Attribute.

Funktionsname:

i_getPageLayout　　　　i_getPageLayout

Signatur:

i_getPageLayout:: env -> objID -> page.

env: Das Environment.

objID: Eine Objekt–ID.

page: Layout–Attribute zu diesem Objekt.

Beschreibung:

Diese Funktion liefert die zum Objekt gehörigen Pagelayout–Attribute.

Funktionsname:

i_getParagraphAttributes　　　　i_getParagraphAttributes

Signatur:

i_getParagraphAttributes:: env -> objID -> paragraph.

env: Das Environment.

objID: Eine Objekt–ID.

paragraph: Paragraph–Attribute zu diesem Objekt.

Beschreibung:

Diese Funktion liefert die zum Objekt gehörigen Paragraph–Attribute.

Funktionsname:

i_getFontInfo i_getFontInfo

Signatur:

i_getFontInfo:: env -> objID -> font.

env: Das Environment.

objID: Eine Objekt–ID.

font: Font–Attribute zu diesem Objekt.

Beschreibung:

Diese Funktion liefert die zum Objekt gehörigen Font–Attribute.

Funktionsname:

i_getCounterAttributes i_getCounterAttributes

Signatur:

i_getCounterAttributes:: env -> objID -> counter.

env: Das Environment.

objID: Eine Objekt–ID.

counter: Zähler–Attribute zu diesem Objekt.

Beschreibung:

Diese Funktion liefert die zum Objekt (dessen Boxtyp 'counter' sein muß) gehörigen Zähler–Attribute.

Funktionsname:

i_equalElementtype i_equalElementtype

Signatur:

i_equalElementtype:: elementtype -> elementtype -> boolean.

elementtype: Ein `elementtype`–Wert.

elementtype: Noch ein `elementtype`–Wert.

boolean: Flag, ob die beiden Werte gleich sind, oder nicht.

Beschreibung:

Diese Funktion vergleicht zwei Werte des abstrakten und deshalb nach außen opaquen Datentyps `elementtype`. Sind die beiden gleich, wird `true`, sonst `false` geliefert.

1.7.4 Lokale Implementation

1.7.4.1 Funktionen

Funktionsname:

correctBoxtype correctBoxtype

Signatur:

correctBoxtype:: pr -> err -> objinfo -> objinfo.

pr: PR–Datenstruktur.

err: Fehler–Datenstruktur.

objinfo: Informationen über ein Objekt.

objinfo: Die korrigierten Informationen über das Objekt.

Beschreibung:

Diese Funktion korrigiert den im `objinfo` gespeicherten Boxtyp–Wert. Sie wird von `i_getFirstObj` und `i_getNextTextObject` benutzt.

Implementierung:

Zu dem im `objinfo` gespeicherten `elementtype` wird in den PR der richtige Boxtyp angefragt und in das `objinfo` hineingeschrieben.

1.7.5 Fehler und Restriktionen

Es besteht an dieser Stelle eine gravierende Einschränkung der Möglichkeiten, die die Präsentationsregeln bieten. Der Formatierer kann nach dem derzeitigen Stand der Implementation nur eine Sicht des Dokumentes speichern und bearbeiten. Aus diesem Grund wurde bei der Anfrage von PR–Attributen auf die Übergabe eines Sichtnamens durch den Formatierer verzichtet, weil dies keinen Sinn hätte. Stattdessen findet der Aufruf an das `PresRules`–Modul immer mit der Sicht 'ALL' statt, die alle Daten enthält, die in der PR–Datei ohne Angabe einer speziellen Sicht definiert wurden.
Der Schnittstellen–Entwurf beinhaltet eine weitere Restriktion. Wenn das `PresRules`–Modul Fehler oder Hinweise meldet — was aufgrund von falschen Angaben in der PR–Datei durchaus passieren kann, — dann gehen diese hier verloren, weil das geänderte Environment nicht an den Aufrufer zurückgegeben wird. Dies wurde beim ersten Entwurf übersehen und später aus Rücksicht auf sonst notwendige umfangreiche Änderungen im Formatierer so belassen.

1.8 ISOutput

1.8.1 Funktionalität

Dieses Modul stellt die Schnittstelle zwischen der Internen Struktur und dem Systemteil der Ausgabe dar.
Es werden die beiden Funktionen zum Lesen und Speichern der Ausgabe–Datenstruktur aus dem bzw. in das Environment zur Verfügung gestellt.
Zusätzlich erhält die Ausgabe durch transitiven Import aus dem Modul `IS` Zugriff auf die Fehlerbehandlungsroutinen.

1.8.2 Öffentliche Schnittstellen

1.8.2.1 Funktionen

Funktionsname:

i_getCookieOS i_getCookieOS

Signatur:

i_getCookieOS:: env -> os.

env: Das Environment.

os: Die Ausgabe–Datenstruktur.

Beschreibung:

Diese Funktion liefert die im Environment gespeicherte Ausgabe–Datenstruktur zurück.

Funktionsname:

i_setCookieOS i_setCookieOS

Signatur:

i_setCookieOS:: env -> os -> env.

env: Das Environment.

os: Die Ausgabe–Datenstruktur.

env: Das neue Environment.

Beschreibung:

Diese Funktion ersetzt die bisher im Environment gespeicherte Ausgabe–Datenstruktur durch die übergebene.

1.9 ISGenerator

1.9.1 Funktionalität

Dieses Modul stellt die Schnittstelle zwischen der Internen Struktur und dem `Generator`–Modul dar, das für die Speicherung der DTD und des Dokumentes verantwortlich ist.
Im wesentlichen werden die vom `DTD`– und `Document`–Modul zur Verfügung gestellten Funktionen auf das Environment umgesetzt.
Zusätzlich erhält der Generator durch transitiven Import aus dem Modul `IS` Zugriff auf die Fehlerbehandlungsroutinen.

1.9.2 Entwurf

Die vom `DTD`– und `Document`–Modul definierte Funktionalität wird in seiner Konzeption und Namensgebung komplett übernommen und nur dahingehend modifiziert, daß die Funktionen auf dem Environment `env` arbeiten.

1.9.3 Öffentliche Schnittstellen

1.9.3.1 Funktionen

Alle nachfolgenden Funktionen sind auf gleiche Art und Weise implementiert: Sie holen die notwendigen Datenstrukturen aus dem Environment, rufen praktisch gleichlautende Funktionen in den Basismodulen auf und speichern wenn nötig die modifizierte Datenstruktur wieder im Environment. Dort, wo die Implementation mehr umfaßt, wird dies erläutert.

Zunächst sind hier die Funktionen zum Auslesen des Dokumentes aufgeführt:

Funktionsname:

i_beginDocSave i_beginDocSave

Signatur:

i_beginDocSave:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion initialisiert die für das Auslesen der Dokumentenstruktur erforderlichen Datenstrukturen.

Funktionsname:

i_endDocSave i_endDocSave

Signatur:

i_endDocSave:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion kennzeichnet das Beenden des Auslesens der Dokumentenstruktur.

Funktionsname:

i_getNextElement i_getNextElement

Signatur:

i_getNextElement:: env -> (env, boolean, string).

env: Das Environment.

(env, boolean, string): Das neue Environment, ein Flag, ob ein Sohn für das aktuelle Element besteht und der SGML–Elementname des Sohns.

Beschreibung:

Diese Funktion ermittelt das nächste auszugebene Element innerhalb des Dokumentes. Grundlage ist ein aktuelles Element. Von diesem aktuellen Element liefert die Funktion den ersten, noch nicht ausgelesenen Sohn als Ergebnis zurück und setzt dieses als neues aktuelle Element. Es ist zu beachten, daß dieses Element später als bereits ausgelesen erkannt und nicht noch einmal zurückgeliefert wird. Existiert für das akuelle Element kein Sohn, so wird `false`, ansonsten `true` zurückgeliefert.

Funktionsname:

i_goParentElement i_goParentElement

Signatur:

i_goParentElement:: env -> (env, boolean).

env: Das Environment.

(env, boolean): Das neue Environment und ein Flag, ob zum Vaterobjekt gewechselt werden konnte.

Beschreibung:

Diese Funktion setzt das Vaterobjekt des aktuellen Elementes als neues aktuelles Element. Handelt es sich bei dem vorherigen aktuellen Element um das Root–Objekt, so wird `false`, ansonsten `true` zurückgeliefert.

Funktionsname:

i_getNextAttribute i_getNextAttribute

Signatur:

i_getNextAttribute:: env -> (env, boolean, attributetype, attributecontents).

env: Das Environment.

(env, boolean, attributetype, attributecontents): Das neue Environment, ein Flag, ob das aktuelle Element ein weiteres Attribut hatte sowie der Typ und Inhalt dieses Attributes.

Beschreibung:

Diese Funktion liefert das nächste Attribut des aktuellen Elementes im Dokument. Es wird festgehalten, welche Attribute für das aktuelle Element ausgelesen wurden. Derzeit werden zwei Attributtypen unterstützt:

- attributetype: `id`
  attributecontents: `id objID`
  Es besteht eine Referenz auf das aktuelle Objekt. Das Objekt erhält nun eine eindeutige Identifikation, auf das sich das referenzierende Objekt beziehen kann. Diese eindeutige Identifikation besteht aus der eindeutigen Objekt–ID.
- attributetype: `refid`
  attributecontents: `id objID`
  Das aktuelle Objekt enthält eine Referenz auf ein anderes Objekt. Die eindeutige Identifikation des referenzierten Objektes erfolgt über die Objekt–ID.

Funktionsname:

i_getElementText | i_getElementText

Signatur:

i_getElementText:: env -> (env, boolean, string).

env: Das Environment.

(env, boolean, string): Das neue Environment, ein Flag, ob das aktuelle Element Text enthält und wenn ja, den Text selbst.

Beschreibung:

Diese Funktion ermittelt den (evtl. vorhandenen) Text des aktuellen Elementes.

Funktionsname:

i_countObjects | i_countObjects

Signatur:

i_countObjects:: env -> integer.

env: Das Environment.

integer: Anzahl von Objekten im Dokument.

Beschreibung:

Diese Funktion liefert die momentane Anzahl von Objekten im Dokument.

Es folgen Funktionen für das Auslesen der DTD.

Funktionsname:

i_beginDTDSave i_beginDTDSave

Signatur:

i_beginDTDSave:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion bereitet die DTD–Datenstruktur zum Speichern vor.

Funktionsname:

i_endDTDSave i_endDTDSave

Signatur:

i_endDTDSave:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion signalisiert dem DTD–Modul, daß die Speicherung beendet ist.

Funktionsname:

i_getNextRule i_getNextRule

Signatur:

i_getNextRule:: env -> (env, boolean, string).

env: Das Environment.

(env, boolean, string): Das neue Environment, ein Flag, ob es noch eine weitere Regel in der DTD gibt und, wenn dies zutrifft, der Name des damit definierten SGML–Elementes.

Beschreibung:

Diese Funktion liefert, wenn es eine weitere DTD–Regel gibt (Boolwert ist `true`), den Namen des logischen Elementes, das mit dieser Regel definiert wird.

Funktionsname:

i_whatIsNext i_whatIsNext

Signatur:

i_whatIsNext:: env -> (env, dtd_content).

env: Das Environment.

(env, dtd_content): Das neue Environment sowie ein Indikator, welches Konstrukt als nächstes innerhalb der gerade ausgelesenen DTD–Regel folgt.

Beschreibung:

Diese Funktion informiert darüber, was als nächstes innerhalb der momentan bearbeiteten Regel kommt. Der zweite Parameter kann einen der folgenden Konstruktoren enthalten: `group`, `endgroup`, `element`, `data` oder `none`.

Implementierung:

Das `DTD`–Modul hat keine Möglichkeit, zwischen einem normalen Elementnamen und der Bezeichnung für einen erlaubten Elementinhalt („PCDATA“, „CDATA“, „NDATA“ oder „EMPTY“) zu unterscheiden, deshalb wird dies nach dem Aufruf des `DTD`–Moduls geprüft und ggf. `data` statt `element` als zweiter Parameter zurückgeliefert.

Funktionsname:

i_getNextGroup i_getNextGroup

Signatur:

i_getNextGroup:: env -> (env, string).

env: Das Environment.

(env, string): Das neue Environment und der Operator für die folgende Gruppe.

Beschreibung:

Diese Funktion gibt an, welche Art von Gruppe als nächste in der momentan bearbeiteten Regel folgt. Es kann eine der folgenden sein: „?“, „*“, „+“, „,“, „&“ oder „|“

Funktionsname:

i_getNextElement i_getNextElement

Signatur:

i_getNextElement:: env -> (env, string).

env: Das Environment.

(env, string): Das neue Environment und der Name des nächsten SGML–Elementes.

Beschreibung:

Mit dieser Funktion wird das nächste Element, das in der gerade „geöffneten" Gruppe folgt, ausgelesen.

Funktionsname:

i_getNextDatatype   i_getNextDatatype

Signatur:

i_getNextDatatype:: env -> (env, datatype).

env: Das Environment.

(env, datatype): Das neue Environment und ein Indikator, welcher Datentyp folgt.

Beschreibung:

Mit dieser Funktion wird der nächste Datentyp („PCDATA", „CDATA", „NDATA" oder „EMPTY"), der in der gerade „geöffneten" Gruppe folgt, ausgelesen.

Implementierung:

Das `DTD`–Modul hat keine Möglichkeit, zwischen einem normalen Elementnamen und der Bezeichnung für einen erlaubten Elementinhalt (Datentyp) zu unterscheiden, deshalb wird der vom `DTD`–Modul gelieferte Elementname mit den zulässigen Datentypen verglichen und der entsprechende Wert zurückgegeben. Entspricht der Name keinem Datentyp, wird ein Fehler der Stufe 'restriction' gemeldet und `empty` als Rückgabewert benutzt.

Funktionsname:

i_getNextAttr   i_getNextAttr

Signatur:

i_getNextAttr:: env -> (env, attributetype, string).

env: Das Environment.

(env, attributetype, string): Das neue Environment sowie der Typ und Inhalt des nächsten Attributes zur aktuellen Regel.

Beschreibung:

Mit dieser Funktion werden sukzessiv die zu dem momentanen SGML–Element gespeicherten Attribute ausgelesen.

Funktionsname:

i_countElements i_countElements

Signatur:

i_countElements:: env -> integer.

env: Das Environment.

integer: Anzahl der Regeln in der DTD.

Beschreibung:

Diese Funktion liefert die Anzahl der in der DTD definierten Regeln (also Elemente) zurück.

1.9.4 Lokale Implementation

1.9.4.1 Funktionen

Funktionsname:

makeError makeError

Signatur:

makeError:: env -> integer -> string -> env.

env: Das Environment.

integer: Fehlernummer.

string: Elementname.

env: Das neue Environment.

Beschreibung:

Diese Funktion übernimmt das Erzeugen des Fehlers, der in der Funktion `i_getNextDatatype` entdeckt wurde. Der String–Parameter enthält den von der DTD gemeldeten Elementnamen, der keinen Datentypen darstellt; er wird als zusätzliche Information im Fehlerobjekt gespeichert.

1.10 ISParser

1.10.1 Funktionalität

Dieses Modul bildet die Schnittstelle zwischen dem Parser und der internen Struktur. Es werden Funktionen zur Verfügung gestellt, mit dessen Hilfe der Parser den Aufbau und Inhalt des Dokumentes, der zugehörigen DTD und der Präsentationsregeln bekannt macht, damit in den entsprechenden Modulen der internen Struktur ein Abbild dieser externen Daten erzeugt werden kann.
Über transitiven Import werden die Routinen zur Fehlerbehandlung und zum Lesen und Speichern des `system` angeboten.
Darüber hinaus organisiert dieses Modul im Zuge des Aufbaus der DTD die Registrierung der `elementtype`s für die SGML–Elemente.

1.10.2 Entwurf

Die Funktions–Sätze für den Aufbau der DTD, des Dokumentes und der PR sind jeweils ähnlich strukturiert. Sie sind so gestaltet, daß der Parser parallel zum Lesen der entsprechenden Quelldatei Funktionen aufruft, um erkannte Konstrukte zu übermitteln. Sehr oft wird dabei zunächst der Beginn eines Konstruktes (z.B. einer Regel oder eines Textobjektes) gemeldet, dann dessen Inhalt und schließlich sein Ende; jeweils mit speziellen Funktionen. Die Reihenfolge der Funktionsaufrufe spielt also eine wichtige Rolle, aus ihr ergeben sich die Strukturen der einzelnen Teile.
Die korrespondierenden Schnittstellen zu den Basismodulen sind für die DTD und das Dokument ganz genauso aufgebaut, so daß die vom Parser kommenden Informationen meist direkt durchgereicht werden können. Im Gegensatz dazu sind die Zugriffsfunktionen für die PR weitestgehend von einer Aufrufreihenfolge unabhängig. Informationen zu den logischen oder physikalischen Objekten der PR werden durch die Übergabe des entsprechenden `elementtype`s zugeordnet. Eine gewisse Reihenfolge ist natürlich zu beachten; so muß ein Objekt erst gemeldet werden, damit das `PresRules`–Modul die Datenstruktur auf die Speicherung von Attributen vorbereiten kann.
Die Benutzung der Funktionsblöcke durch den Parser ist ebenfalls nicht ganz willkürlich zu handhaben. Da die Funktionen zum Aufbau der DTD die Vergabe der `elementtype`s vornehmen, muß die DTD auf jeden Fall vor dem Dokument und den Präsentationsregeln geparst werden.

1.10.3 Öffentliche Schnittstellen

1.10.3.1 Funktionen

Funktionen zum Aufbau des Dokumentes:
Da die Schnittstelle zum `PresRules`–Modul von einer Aufrufreihenfolge weitestgehend unabhängig gestaltet wurde, der Parser jedoch statt kompletter Konstrukte meist nur Informationen über deren Anfang und Ende liefert, müssen Daten zwischengespeichert werden, die sonst nach einem Funktionsaufruf verloren gehen würden. Dafür benutzt das `ISParser`–Modul eine Hilfs–Datenstruktur `ISP` von geringer Komplexität, zu der genaueres in der Beschreibung des Moduls `ISPStructure`, nachgelesen werden kann.
Funktionsname:

i_newDocument | i_newDocument

Signatur:

i_newDocument:: env -> string -> env.

env: Das Environment.

string: Name des Dokumentes.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt den Beginn des Einlesens eines Dokumentes mit dem in `string` übergebenen Namen an.

Implementierung:

Es wird eine leere Dokument–Datenstruktur erzeugt und in das Environment geschrieben, sowie die benötigte Hilfsstruktur `isp` initialisiert.

Funktionsname:

i_endDocument i_endDocument

Signatur:

i_endDocument:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt das Ende des eingelesenen Dokumentes an.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `Document--Modul` statt.

Funktionsname:

i_clearDocument i_clearDocument

Signatur:

i_clearDocument:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion löscht das zuletzt mit `i_newDocument` erzeugte Dokument unwiederbringlich.

Implementierung:

Es wird eine leere Dokument–Datenstruktur angefordert und in das Environment geschrieben. Gleichzeitig wird die ISP–Struktur neu initialisiert.

Funktionsname:

i_beginElement i_beginElement

Signatur:

i_beginElement:: env -> string -> env.

env: Das Environment.

string: SGML–Elementname.

env: Das neue Environment.

Beschreibung:

Diese Funktion meldet den Beginn eines neuen Elementes im Quelldokument. Der String bezeichnet den SGML–Elementnamen.
Ist dieser Elementname in der geparsten DTD nicht vorhanden, wird ein Fehler vom Typ `restriction` gemeldet, für das Element aber ein `elementtype` angefordert und der Beginn eines neuen Elementes dem `Document`–Modul mitgeteilt, so daß dieses den Fehler nicht bemerkt. In die DTD wird dieser Elementtyp nicht aufgenommen, so daß ein interaktives Einfügen eines Objektes dieses Typs später nicht möglich ist. Da aber ein gravierender Fehler vorliegt, der möglichst sofort behoben werden sollte, ist eine Bearbeitungs des Textes sowieso nicht sinnvoll.
In jedem Fall werden die bis zu diesem Aufruf vom Parser bekanntgegebenen Attribute an das `Document`–Modul vermittelt, da der Parser die zu einem Element gehörigen Attribute vor dessen Anmeldung übergibt.

Implementierung:

Zunächst wird der zugehörige `elementtype` angefragt und dann das neue Element im `Dokument`–Modul angemeldet. Anschließend werden alle in der ISP–Struktur zwischengespeicherten Attribute übergeben und aus `isp` gelöscht.

Vor- und Nachbedingungen:

Vor dem Parsen des Dokumentes muß auf jeden Fall die DTD komplett vorhanden sein. Ist dies der Fall, und es wurde zu dem Element trotzdem kein `elementtype` registriert, liegt entweder in den Funktionen zum Aufbau der DTD in diesem Modul, oder in den `elementtype`–Funktionen im Modul `ISTypes` ein Fehler. Es könnte auch sein, daß die DTD und das Dokument nicht zusammenpassen, aber das sollte durch den Parser eigentlich vorher bemerkt worden sein.

Funktionsname:

i_endElement i_endElement

Signatur:

i_endElement:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion schließt das momentan offene Element.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im Document–Modul statt.

Funktionsname:

i_text i_text

Signatur:

i_text:: env -> string -> env.

env: Das Environment.

string: Text.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in das momentan offene Element den übergebenen Text ein.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im Document–Modul statt.

Funktionsname:

i_attribute i_attribute

Signatur:

i_attribute:: env -> attributetype -> attributecontents -> env.

env: Das Environment.

attributetype: SGML–Attributtyp.

attributecontents: Wert des Attributes.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in das SGML–Element, das als nächstes mittels `i_beginElement` geöffnet wird, ein SGML–Attribut vom Typ `attributetype` mit dem Wert `attributecontents` ein.
Es kann passieren, daß im Quelldokument ein ID– oder REFID–Attribut keinen Wert hat, weil dieser laut der DTD nicht gesetzt werden muß. In diesem Fall wird das Attribut ignoriert.

Implementierung:

Das Attribut wird in der ISP–Datenstruktur zwischengespeichert, wenn eine Prüfung des Wertes mittels `ismt` negativ ausgefallen ist.

Funktionen zum Aufbau der DTD:
Funktionsname:

i_beginDTD i_beginDTD

Signatur:

i_beginDTD:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt den Beginn des Einlesens einer DTD an.

Implementierung:

Es wird eine leere DTD–Datenstruktur erzeugt und in das Environment geschrieben.

Funktionsname:

i_endDTD i_endDTD

Signatur:

i_endDTD:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt das Ende des Parsevorgangs für die DTD an.
Außerdem wird im Dokument eine Instanz des obersten DTD–Elementes angelegt, um dem Benutzer eine interaktive Eingabe zu ermöglichen, wenn kein vorhandenes Dokument geladen, sondern ein neues erstellt werden soll. Es gibt dann zwar ein im Sinne der DTD ungültiges Dokument, aber so kann der Oberfläche eine erste Objekt–ID an die Hand gegeben werden. Das Dokument wird in der ersten Phase der interaktiven Erstellung sowieso

zunächst ungültig bleiben, weil dem Benutzer nicht zugemutet werden kann, alle notwendigen Auswahlen an Produktionen sofort vorzunehmen, nur um ein korrektes Dokument zu erhalten. Es wird also eine schrittweise Verfeinerung vorgenommen, indem nur auf der Ebene eines Elementtyps geblieben wird, und nicht Instanziierungen seiner Komponenten gefordert werden. Wichtig ist, daß das DTD–Modul eine Anfrage nach gültigen Manipulationen trotzdem richtig beantworten kann. Wenn ein Element, in das eingefügt werden soll, leer ist, obwohl laut DTD Elemente hierin zwingend erforderlich sind, dann müssen trotzdem alle Möglichkeiten ermittelt werden.
Wird nach dem Parsen der DTD ein Dokument geladen, löscht die Funktion `i_newDocument` durch das Ersetzen der alten Dokument–Struktur diese Manipulation.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im DTD–Modul statt. Anschließend wird das oberste Element erfragt und ein solches als Wurzelobjekt im Dokument angemeldet.

Funktionsname:

i_clearDTD — i_clearDTD

Signatur:

i_clearDTD:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion löscht die DTD–Datenstruktur. Das Dokument wird nicht gelöscht! Sie ist mit Vorsicht anzuwenden, da ein Dokument ohne die zugehörige DTD nicht bearbeitet werden kann.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im DTD–Modul statt.

Funktionsname:

i_beginRule — i_beginRule

Signatur:

i_beginRule:: env -> string -> env.

env: Das Environment.

string: SGML–Elementname.

env: Das neue Environment.

Beschreibung:

Diese Funktion legt eine neue Regel an, die den Aufbau des mit `string` bezeichneten Elementes beschreibt.

Implementierung:

Es wird ein `elementtype` für das mit dieser Regel definierte Element vergeben, falls das noch nicht geschehen ist. Anschließend wird das `DTD`–Modul über den Beginn der Definition unterrichtet.

Funktionsname:

i_beginGroup　　　　i_beginGroup

Signatur:

i_beginGroup:: env -> operator -> env.

env: Das Environment.

operator: Art der Gruppe.

env: Das neue Environment.

Beschreibung:

Diese Funktion legt eine neue Gruppe an, in die durch nachfolgende Aufrufe von `i_insertElement` die Elemente eingetragen werden. Als Operatoren sind zulässig: '?', '*', '+', ',', '&', '|'.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `DTD`–Modul statt.

Funktionsname:

i_insertElement　　　　i_insertElement

Signatur:

i_insertElement:: env -> string -> env.

env: Das Environment.

string: SGML–Elementname.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in die zuletzt geöffnete Gruppe das mit `string` angegebene SGML–Element ein.

Implementierung:

Zunächst wird überprüft, ob das einzutragende Element bereits einen `elementtype` hat; wenn nicht, wird einer vergeben und das Element auf jeden Fall dem `DTD`–Modul gemeldet.

Funktionsname:

i_insertDatatype i_insertDatatype

Signatur:

i_insertDatatype:: env -> datatype -> env.

env: Das Environment.

datatype: SGML–Datentyp.

env: Das neue Environment.

Beschreibung:

Mit dieser Funktion wird angezeigt, daß das zuletzt mit `i_beginRule` definierte Element auch terminal sein — also Text enthalten — kann. Der Parameter `datatype` kann einer der folgenden Konstruktoren sein: `PCDATA`, `CDATA`, `NDATA` oder `EMPTY`.

Implementierung:

Für den `datatype` wird ein `elementtype` vergeben und dieser so an die DTD gemeldet, als wäre er mit `i_insertElement` übergeben worden. Das `DTD`–Modul bemerkt also nicht, daß es sich nicht um ein normales Element handelt. Auf diese Weise wurde die Schnittstelle zum `DTD`–Modul einfach gehalten. Die notwendige Unterscheidung zwischen Elementen und `datatype`s wird in den Schnittstellenmodulen zu anderen Systemteilen vorgenommen.

Funktionsname:

i_endGroup i_endGroup

Signatur:

i_endGroup:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion schließt die zuletzt geöffnete Gruppe.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `DTD`–Modul statt.

Funktionsname:

i_endRule i_endRule

Signatur:

i_endRule:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion schließt die zuletzt eingeleitete Regel ab (also praktisch die Definition des entsprechenden Elementes).

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `DTD`–Modul statt.

Funktionsname:

i_beginAttrRule i_beginAttrRule

Signatur:

i_beginAttrRule:: env -> string -> env.

env: Das Environment.

string: SGML–Elementname.

env: Das neue Environment.

Beschreibung:

Diese Funktion signalisiert, daß zu dem mit `string` benannten Element mindestens ein Attribut in der DTD definiert wird und diese mit nachfolgenden Aufrufen von `i_insertAttr` bekannt gemacht werden.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `DTD`–Modul statt.

Vor- und Nachbedingungen:

Das Element muß vorher gemeldet worden sein. Kann kein zugehöriger `elementtype` gefunden werden, generiert die Funktion einen fatalen Fehler, da irgendetwas mit der DTD nicht stimmt.

Funktionsname:

i_insertAttr i_insertAttr

Signatur:

i_insertAttr:: env -> attributetype -> string -> env.

env: Das Environment.

attributetype: SGML–Attributtyp.

string: Attributwert.

env: Das neue Environment.

Beschreibung:

Diese Funktion fügt dem beim Aufruf von `i_beginAttrRule` benannten Element ein Attribut mit dem Namen `string` hinzu.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `DTD`–Modul statt.

Funktionsname:

i_endAttrRule i_endAttrRule

Signatur:

i_endAttrRule:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion schließt die Liste mit Attributen ab.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `DTD`–Modul statt.

Funktionen zum Aufbau der Präsentationsregeln:
Eine häufig vorkommende Fehlermöglichkeit innerhalb dieser Gruppe von Funktionen ist die Bezugnahme auf logische Elemente, für die kein `elementtype` registriert ist. Um in diesem Fall den Parsevorgang nicht abbrechen zu müssen, wird ein `elementtype` vergeben, obwohl dieses Element in der DTD nicht vorkommt. Anschließend wird die jeweilige Funktion normal fortgesetzt und zudem ein Fehler vom Typ `restriction` erzeugt. Ein solches Vorgehen kann Probleme verursachen, wenn das `DTD`– oder `Document--Modul` mit einem solchen Element arbeiten müssen — allerdings nur, wenn die Fehlermeldung ignoriert wird.

Funktionsname:

i_beginPR i_beginPR

Signatur:

i_beginPR:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt den Beginn des Einlesens einer PR an.

Implementierung:

Es wird eine leere PR– und ISP–Datenstruktur erzeugt und in das Environment geschrieben.

Funktionsname:

i_endPR i_endPR

Signatur:

i_endPR:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt das Ende des Parsevorgangs für die PR an.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_clearPR i_clearPR

Signatur:

i_clearPR:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion löscht die PR–Datenstruktur. Das Dokument und die DTD werden nicht gelöscht.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_beginEntity i_beginEntity

Signatur:

i_beginEntity:: env -> entitytype -> env.

env: Das Environment.

entitytype: Art des zu definierenden Konstrukts.

env: Das neue Environment.

Beschreibung:

Mit Hilfe dieser Funktion signalisiert der Parser den Beginn eines Blocks oder einer Liste in den PR. Worum es sich handelt, wird durch den Wert für `entitytype` festgelegt:

views Liste der Views.

objDef Definition des zuletzt mit `i_beginObjBlock` eingeleiteten Objektes.

objList Liste mit Folgeobjekten des zuletzt mit `i_beginObjBlock` eingeleiteten Objektes.

attrList Attributliste des zuletzt mit `i_beginObjBlock` eingeleiteten Objektes.

varObjList Liste mit Folgeobjekten, die in beliebiger Reihenfolge auftauchen dürfen.

Implementierung:

Diese Funktion ist eng mit der korrespondierenden Funktion `i_endEntity` verzahnt. Schwierigster Fall dabei ist die Folgeobjektliste (`objList`), da in ihr wiederum Listen mit Objekten auftauchen können, die im Dokument in beliebiger Reihenfolge im Vaterobjekt stehen dürfen (`varObjList`). Dabei wird folgendermaßen vorgegangen:
Der Parser meldet den Beginn der `objList` und anschließend Elemente daraus. Diese werden in der `isp`–Struktur in einer Liste zwischengespeichert. Findet der Parser nun eine `varObjList`, dann werden die bisher übergebenen Elemente an das `PresRules`–Modul weitergemeldet und in `isp` vermerkt, daß nun Elemente aus einer `varObjList` folgen. Nach deren Abschluß wird diese Liste als Ganzes dem `PresRules`–Modul gemeldet und der momentane Listentyp in `isp` wieder auf `objList` zurückgesetzt, da ja noch andere Elemente in der Folgeobjektliste kommen können.
Als Zwischenspeicher wird nur *eine* Listenstruktur in `isp` verwendet, die bei Bedarf gelöscht wird. Dies ist möglich, weil in einer `varObjList` nur einfache Elemente und keine weiteren Listen auftauchen dürfen, und die Funktion zur Aufnahme von Elementen einer Folgeobjektliste im `PresRules`–Modul so gestaltet ist, daß sie bei jedem Aufruf die neuen Elemente an die bestehende Liste anhängt.
Alle Entity-Typen außer `objList` und `varObjList` werden ignoriert, da in diesen Fällen keine besondere Vorbereitung auf nachfolgende Funktionsaufrufe nötig ist.

Funktionsname:

i_endEntity i_endEntity

Signatur:

i_endEntity:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion schließt den zuletzt mit `i_beginEntity` geöffneten Block bzw. die zuletzt geöffnete Liste ab.

Implementierung:

Die prinzipielle Vorgehensweise ist bei der Funktion `i_beginEntity` erläutert. Erwähnenswert ist noch, daß ein kleiner Fehler, der in der Quell–PR vorhanden sein könnte, beseitigt wird: Wurde in einer Folgeobjektliste eine Objektliste mit beliebiger Reihenfolge aufgeführt, die aber nur ein Element hat, dann wird dieses als normales Element der Folgeobjektliste an das `PresRules`–Modul weitergemeldet, da die Spezifizierung einer beliebigen Anordnung von Elementen nur bei mindestens zwei verschiedenen Typen Sinn macht.
Ist eine Objektliste mit beliebiger Reihenfolge leer, wenn der Parser deren Ende meldet, wird sie ignoriert.

Funktionsname:

i_newView — i_newView

Signatur:

i_newView:: env -> string -> env.

env: Das Environment.

string: Sichtname.

env: Das neue Environment.

Beschreibung:

Hiermit wird eine neue Sicht mit dem Namen `string` definiert.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_beginObjBlock — i_beginObjBlock

Signatur:

i_beginObjBlock:: env -> objtype -> string -> env.

env: Das Environment.

objtype: Typ des Objektes.

string: Objektname.

env: Das neue Environment.

Beschreibung:

Diese Funktion signalisiert den Beginn der Definition eines Objektes mit dem Namen `string`. `objtype` gibt an, ob es sich um ein logisches oder ein physikalisches Objekt handelt.

Implementierung:

Handelt es sich um ein physikalisches Objekt, wird ein `elementtype` vergeben, während bei einem logischen Objekt der registrierte angefragt wird. Anschließend wird das Objekt dem `PresRules`–Modul gemeldet und der Sichtname in der `isp`–Datenstruktur auf „ALL" gesetzt. Kommt hiernach kein `i_beginViewDef`, dann werden die zu diesem Objekt in der Folgezeit gemeldeten Attribute in der „ALL"–Sicht bei den PR gespeichert. Zusätzlich wird der ermittelte `elementtype` in der ISP–Struktur gespeichert, damit die folgenden Funktionsaufrufe ihn zur Hand haben.
Anmerkung: Für die Vergabe der `elementtype`s wird den Namen der physikalischen Objekte ein ‚!' vorangestellt, um Überschneidungen mit den logischen Elementen aus der DTD zu vermeiden.

Vor- und Nachbedingungen:

Sind in der DTD Elemente definiert, die mit einem ‚!' beginnen, kommt es zu Problemen, falls es gleichnamige physikalische Objekte in den PR gibt.

Funktionsname:

i_endObjBlock   i_endObjBlock

Signatur:

i_endObjBlock:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion signalisiert das Ende der Definition eines Objektes.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_formatType i_formatType

Signatur:

i_formatType:: env -> string -> env.

env: Das Environment.

string: Boxtyp.

env: Das neue Environment.

Beschreibung:

Diese Funktion übergibt den Typ des Formatierobjektes (Boxtyp), dem das zuletzt mit `i_beginObjBlock` geöffnete Element zugeordnet ist.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_beginViewDef i_beginViewDef

Signatur:

i_beginViewDef:: env -> string -> env.

env: Das Environment.

string: Sichtname.

env: Das neue Environment.

Beschreibung:

Hiermit wird innerhalb des zuletzt angekündigten Objektes der Beginn einer Definition zu der Sicht mit dem Namen `string` signalisiert.

Implementierung:

Der Sichtname wird in der `isp`–Struktur gespeichert.

Funktionsname:

i_endViewDef i_endViewDef

Signatur:

i_endViewDef:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Hiermit wird die zuletzt mit `i_beginViewDef` eingeleitete Sichtdefinition beendet.

Implementierung:

Als Sichtname wird „ALL“ in der `isp`–Struktur gespeichert.

Funktionsname:

i_insertObj i_insertObj

Signatur:

i_insertObj:: env -> objtype -> string -> env.

env: Das Environment.

objtype: Objekttyp.

string: Objektname.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in die mit `i_beginEntity` (`entitytype` = `objList` oder `entitytype` = `varObjList`) geöffnete Folgeobjektliste ein logisches oder physikalisches Objekt mit dem Namen `string` ein.

Implementierung:

Zu dem übergebenen Objektnamen wird dessen `elementtype` angefordert und an die Objektliste in `isp` angehängt.

Vor- und Nachbedingungen:

Zuvor muß vom Parser der Beginn einer Folgeobjektliste gemeldet worden sein. Ist dies nicht der Fall, wird ein Fehler vom Status `repair` erzeugt, der als zusätzliche Angaben den Namen des Objektes und der Sicht enthält, zu der dieses Folgeobjekt eingetragen werden sollte.

Funktionsname:

i_insertString i_insertString

Signatur:

i_insertString:: env -> string -> env.

env: Das Environment.

string: Zeichenkette.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in die mit `i_beginEntity` (`entitytype` = `objList`) geöffnete Liste ein Textobjekt mit dem Inhalt `string` ein.

Implementierung:

Das Textobjekt wird an die Objektliste in `isp` angehängt.

Vor- und Nachbedingungen:

Zuvor muß vom Parser der Beginn einer Folgeobjektliste (`objList`) gemeldet worden sein. Ist dies nicht der Fall, wird ein Fehler vom Status `repair` erzeugt, der als zusätzliche Angaben den Namen des Objektes und der Sicht enthält, zu der dieses Folgeobjekt eingetragen werden sollte.
Wurde vor diesem Funktionsaufruf eine `varObjList` geöffnet, wird ebenfalls ein ‚reparierter' Fehler gemeldet, da hier ein Textobjekt nicht erlaubt ist.

Funktionsname:

i_insertPagebreak — i_insertPagebreak

Signatur:

i_insertPagebreak:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in die mit `i_beginEntity` (`entitytype` = `objList`) geöffnete Liste ein `Pagebreak` ein.

Implementierung:

Es wird die Funktion `i_insertString` mit der Zeichenkette „Pagebreak" aufgerufen. Für mögliche Fehlersituationen gilt das dort gesagte entsprechend.

Funktionsname:

i_insertPCDATA — i_insertPCDATA

Signatur:

i_insertPCDATA:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in die mit `i_beginEntity` (`entitytype` = `objList`) geöffnete Liste ein „PCDATA“ ein.

Implementierung:

Es wird die Funktion `i_insertString` mit der Zeichenkette „PCDATA“ aufgerufen. Für mögliche Fehlersituationen gilt das dort gesagte entsprechend.

Funktionsname:

i_insertAttrUnit i_insertAttrUnit

Signatur:

i_insertAttrUnit:: env -> string -> real -> string -> env.

env: Das Environment.

string: Attributname.

real: Attributwert.

string: Maßeinheit.

env: Das neue Environment.

Beschreibung:

Hiermit wird innerhalb des zuletzt angekündigten Objektes ein PR–Attribut mit dem Namen `string` eingetragen, das als Wert eine Größenangabe (`real`) und eine Maßeinheit (zweiter `string`) hat.

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_insertAttrLog i_insertAttrLog

Signatur:

i_insertAttrLog:: env -> string -> string -> env.

env: Das Environment.

string: Attributname.

string: Attributwert: Name eines logischen Elementes.

env: Das neue Environment.

Beschreibung:

Hiermit wird innerhalb des zuletzt angekündigten Objektes ein PR–Attribut mit dem Namen `string` eingetragen, das als Wert den Namen eines logischen Elementes hat (zweiter `string`).

Implementierung:

Es findet ein Aufruf der entsprechenden Funktion im `PresRules`–Modul statt.

Funktionsname:

i_beginAttrVal　　　　i_beginAttrVal

Signatur:

i_beginAttrVal:: env -> string -> env.

env: Das Environment.

string: Attributname.

env: Das neue Environment.

Beschreibung:

Hiermit wird innerhalb des zuletzt angekündigten Objektes ein PR–Attribut mit dem Namen `string` gemeldet, das aus einer Liste mit Schlüsselworten besteht, die durch nachfolgende `i_insertAttrVal`–Aufrufe eingetragen werden.

Implementierung:

Der Attributname wird in der `isp`–Struktur gespeichert.

Funktionsname:

i_endAttrVal　　　　i_endAttrVal

Signatur:

i_endAttrVal:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion zeigt das Ende der mit `i_beginAttrVal` eingeleiteten Schlüsselwortliste an.

Implementierung:

Wenn es sich bei dem Attribut um `fontstyle` gehandelt hat, werden jetzt die in `isp` gespeicherten Fontstyle–Angaben im Stück an das `PresRules`–Modul gemeldet. Ansonsten findet keine Aktion statt.

Funktionsname:

i_insertAttrVal　　　i_insertAttrVal

Signatur:

i_insertAttrVal:: env -> string -> env.

env: Das Environment.

string: Schlüsselwort.

env: Das neue Environment.

Beschreibung:

Diese Funktion trägt in die geöffnete Schlüsselwortliste den übergebenen Wert ein.

Implementierung:

Fontstyle–Schlüsselworte werden in der `isp`–Datenstruktur gespeichert, während alle anderen direkt an das `PresRules`–Modul weitergehen.

1.10.4 Lokale Implementation

1.10.4.1 Funktionen

Funktionsname:

announceAttrs　　　announceAttrs

isp: ISP–Datenstruktur.

doc: Dokument–Datenstruktur.

(isp, doc): Die neue ISP– und Dokument–Struktur.

Beschreibung:

Diese Funktion wird von `i_beginElement` benutzt, um die in der ISP–Datenstruktur gespeicherten Attribute dem Dokument zu melden.

Funktionsname:

convertDatatype　　　convertDatatype

datatype: Datatype–Konstruktor.

string: Korrespondierender String.

Beschreibung:

Diese Funktion wird von `i_insertDatatype` benutzt, um die Konstruktoren in äquivalente Strings („#PCDATA“, „#CDATA“, ‘#NDATA“ und „EMPTY“) umzusetzen.

Funktionsname:

getElType getElType

env: Das Environment

objtype: Objekttyp.

string: Objektname.

integer: Fehlernummer.

strings: Fehlertext.

(env, elementtype): Das neue Environment und der zum Objekt gehörige `elementtype`.

Beschreibung:

Diese Funktion ermittelt für den übergebenen Elementnamen den zugehörigen `elementtype`, wenn es sich um ein logisches Element handelt (`objtype` = `log`). Ist hierfür keiner vergeben worden, wird ein Fehler erzeugt, der die übergebene Nummer und den übergebenen Text enthält. Anschließend wird ein `elementtype` vergeben; so wie für ein physikalisches Objekt.

1.11 ISUserInterface

1.11.1 Funktionalität

Dieses Modul stellt die Schnittstelle zwischen der Internen Struktur und der Benutzungsoberfläche (UserInterface, kurz „UI") dar.
Neben der Zugriffsmöglichkeit auf das Dokument, die DTD und das `system` werden die Fehlerbehandlungsfunktionen durch transitiven Import zur Verfügung gestellt.

1.11.2 Öffentliche Schnittstellen

1.11.2.1 Funktionen

Funktionsname:

i_init i_init

Signatur:

i_init:: system -> env.

system: Das „System".

env: Das initialisierte Environment.

Beschreibung:

Diese Funktion erzeugt ein Environment, dessen Datenstrukturen initialisiert sind und speichert das übergebene `system` darin.

Implementierung:

Das `system` wird an die Funktion `mt` aus dem Modul `IS` weitergereicht.

Funktionsname:

i_exit i_exit

Signatur:

i_exit:: env -> env.

env: Das Environment.

env: Das neue Environment.

Beschreibung:

Diese Funktion soll den relevanten Modulen die Gelegenheit geben, ihre Datenstrukturen auf das Programmende vorzubereiten, falls dies erforderlich ist.

Implementierung:

Das Environment wird unverändert zurückgegeben, weil keine Datenstruktur eine spezielle Behandlung vor dem Programmabschluß erfordert.

Funktionsname:

i_getCookieUI i_getCookieUI

Signatur:

i_getCookieUI:: env -> ui.

env: Das Environment.

ui: Datenstruktur der Benutzungsoberfläche.

Beschreibung:

Diese Funktion gibt die im Environment gespeicherte UI–Datenstruktur zurück.

Implementierung:

Es wird eine entsprechende Funktion im `IS`–Modul aufgerufen.

Funktionsname:

i_setCookieUI i_setCookieUI

Signatur:

i_setCookieUI:: env -> ui -> env.

env: Das Environment.

ui: Datenstruktur der Benutzungsoberfläche.

env: Das neue Environment.

Beschreibung:

Diese Funktion speichert die übergebene UI–Datenstruktur im Environment; die alte wird gelöscht.

Implementierung:

Es wird eine entsprechende Funktion im `IS`–Modul aufgerufen.

Funktionen für den Zugriff auf das Dokument:

Funktionsname:

i_insertChars i_insertChars

Signatur:

i_insertChars:: env -> objID -> integer -> string -> env.

env: Das Environment.

objID: Die ID des Objektes, in das Zeichen eingefügt werden sollen.

integer: Start–Index innerhalb des Objekt–Textes.

string: Einzufügende Zeichen.

env: Das neue Environment.

Beschreibung:

Diese Funktion fügt die übergebene Zeichenkette in den Text des spezifizierten Dokument–Objektes ein. Der `integer`–Parameter gibt die Nummer des Zeichens (Index) im bisherigen Text des Objektes an, *vor* dem der `string` eingefügt wird. Der Index startet bei Null für das erste Zeichen.

Implementierung:

Es wird eine entsprechende Funktion im `Document`–Modul aufgerufen.

Funktionsname:

i_deleteChars — i_deleteChars

Signatur:

i_deleteChars:: env -> objID -> integer -> integer -> (string, env).

env: Das Environment.

objID: Die ID des Objektes, aus dem die Zeichen gelöscht werden sollen.

integer: Start–Index innerhalb des Objekt–Textes.

integer: End–Index innerhalb des Objekt–Textes.

(string, env): Die gelöschte Zeichenkette und das neue Environment.

Beschreibung:

Diese Funktion löscht in dem spezifizierten Objekt alle Zeichen ab dem übergebenen Start–Index bis zum End–Index (einschließlich). Der Index startet bei Null für das erste Zeichen. Die gelöschte Zeichenkette wird mit zurückgegeben.

Implementierung:

Es wird eine entsprechende Funktion im `Document`–Modul aufgerufen.

Funktionsname:

i_getFirstObject — i_getFirstObject

Signatur:

i_getFirstObject:: env -> objinfo.

env: Das Environment.

objinfo: Informationen über das oberste Objekt.

Beschreibung:

Diese Funktion liefert Informationen über das Wurzelobjekt des Dokumentes. Wurde kein SGML–Quelldokument geparst, dann existiert dieses Objekt trotzdem, aber es ist leer.

Implementierung:

Es wird eine entsprechende Funktion im `Document`–Modul aufgerufen.

Funktionsname:

i_insertElement i_insertElement

Signatur:

i_insertElement:: env -> insertPos -> strings -> (objinfo, env).

env: Das Environment.

insertPos: Einfügeposition.

strings: Liste der Elemente, die einzufügen sind.

(objinfo, env): Informationen über das letzte eingefügte Objekt sowie das neue Environment.

Beschreibung:

Diese Funktion fügt „leere“ Objekte in das Dokument ein.[7] Der Parameter `strings` enthält die SGML–Elementnamen, deren Instanzen die Objekte darstellen. An der spezifizierten Stelle werden die Objekte in der Reihenfolge der Elementnamen eingefügt.
Da die DTD für jedes Element seinen möglichen Inhalt definiert, muß fast jedes Objekt weiter „heruntergebrochen“ werden, bis man auf Elemente stößt, die leer sein dürfen oder nur Text enthalten. In letztere wird als Text ihr Elementname eingetragen, damit sie auf dem Bildschirm überhaupt zu sehen sind. Den Namen muß der Benutzer allerdings manuell durch einfaches Löschen der Zeichen wieder entfernen.
Die DTD sieht in der Regel mehrere Varianten für den Aufbau eines Elementes vor. Damit der Benutzer nicht in einem langwierigen Prozeß für alle Objekte eine Entscheidung treffen muß, wird die erste vom `DTD`–Modul gefundene Variante ausgewählt.
Als Rückgabe erhält die aufrufende Funktion das geänderte Environment und Informationen über das letzte Objekt, das auf der mit `insertPos` angegebenen Ebene eingefügt wurde; dies ist stets ein Objekt vom Typ des letzten SGML–Elementes in `strings`.
Zur genaueren Information über die Einfügeposition kann im `Document`–Modul, auf Seite 100 nachgeschlagen werden.

Vor- und Nachbedingungen:

[7] Leer sind die Objekte meist nicht, da DTDs in vielen Fällen enthaltene Elemente vorschreiben.

Diese Funktion prüft **nicht**, ob die einzufügenden Objekte die Korrektheit des Dokumentes erhalten! Es wird davon ausgegangen, daß die Benutzungsoberfläche sich zuvor mit `i_requestInsertElements` die möglichen Elementkombinationen erfragt und daraus eine gewählt hat.

Implementierung:

Zunächst werden zu den Elementnamen die `elementtype`s angefordert. Ist eines davon nicht vorhanden, wird ein Fehler erzeugt und nichts eingefügt; `objinfo` ist leer.
Ansonsten werden mittels der Funktion `breakDownElement` die Objekte der Reihenfolge nach von links nach rechts in das Dokument eingefügt. Diese Funktion erzeugt durch rekursive Aufrufe auch die untergeordneten Objekte.

Funktionsname:

i_deleteElement i_deleteElement

Signatur:

i_deleteElement:: env -> objID -> (boolean, objID, env).

env: Das Environment.

objID: ID des zu löschenden Objektes.

(boolean, objID, env): Flag, ob das Objekt gelöscht werden konnte, die ID des Vaterobjektes und das neue Environment.

Beschreibung:

Diese Funktion löscht das mit `objID` bezeichnete Objekt aus dem Dokument und löscht dabei auch die in ihm enthaltenen Objekte, um die Korrektheit zu gewährleisten. Konnte das Objekt nicht gelöscht werden, weil dies nicht der DTD entsprechen würde oder das Wurzel-Objekt gelöscht werden sollte, ist das Flag auf `false` gesetzt. Die zurückgegebene Objekt–ID ist die ID des Vaters des zu löschenden Objektes. Ist der Boolwert `false`, darf dieser Parameter nicht ausgewertet werden.

Implementierung:

Handelt es sich bei dem zu löschenden Objekt nicht um die Wurzel des Dokumentes, wird mit Hilfe der Funktion `deleteable` aus dem `DTD`–Modul geprüft, ob das Löschen mit der DTD konform geht.

Funktionsname:

i_requestInsertElements i_requestInsertElements

Signatur:

i_requestInsertElements:: env -> insertPos -> listOfStrings.

env: Das Environment.

insertPos: Einfügeposition.

listOfStrings: Liste mit möglichen Elementkombinationen.

Beschreibung:

Diese Funktion fragt an, welche Elemente in, nach oder vor (`insertPos`) dem spezifizierten Objekt (`objID` in `insertPos`) laut DTD eingefügt werden dürfen. Die zulässigen Möglichkeiten werden in einer Liste zurückgegebenen, dessen einzelne Elemente jeweils eine Liste mit Strings darstellen. In jedem String steht der Name eines SGML-Elementes. Eine Liste von Elementnamen bedeutet, daß diese Elemente in der gegebenen Reihenfolge (von links nach rechts) an dieser Stelle in das Dokument eingefügt werden können.

Implementierung:

Aus dem `Document`–Modul wird der aktuelle Inhalt des Objektes, in das eingefügt werden soll, angefordert. Damit wird die Funktion `insertables` aus dem `DTD`–Modul „gefüttert".

Funktionsname:

i_requestInsertPosition i_requestInsertPosition

Signatur:

i_requestInsertPosition:: env -> objID -> integer -> insertPosList.

env: Das Environment.

objID: Objekt, auf dem der Cursor steht.

integer: Index des Zeichens, hinter dem der Cursor steht.

insertPosList: Liste der möglichen Einfügepositionen.

Beschreibung:

Diese Funktion liefert zu einer Cursorposition die Liste der möglichen Einfügestellen. Näheres dazu findet sich bei der entsprechenden Funktion im `Document`–Modul auf Seite 113.

Implementierung:

Es wird eine entsprechende Funktion im `Document`–Modul aufgerufen.

Funktionsname:

i_requestExchangeElement i_requestExchangeElement

Signatur:

i_requestExchangeElement:: env -> objID -> listOfStrings.

env: Das Environment.

objID: ID des auszutauschenden Objektes.

listOfStrings: Liste der Elemente, die den Platz des Objektes einnehmen dürften.

Beschreibung:

Diese Funktion überprüft, gegen welche Elemente das spezifizierte Objekt ausgetauscht werden kann. Jede Liste in `listOfStrings` stellt eine Möglichkeit dar, das ursprüngliche Element auszutauschen und enthält wiederum Strings, die jeweils Elementnamen bezeichnen.

Implementierung:

Die Funktion ist nicht implementiert.

Funktionsname:

i_exchangeElement   i_exchangeElement

Signatur:

i_exchangeElement:: env -> objID -> strings -> env.

env: Das Environment.

objID: ID des auszutauschenden Objektes.

strings: Stattdessen einzufügende Elemente.

env: Das neue Environment.

Beschreibung:

Das benannte Objekt wird durch die in `strings` übergebenen Elemente ausgetauscht. Die neuen Objekte werden von links nach rechts eingefügt.

Implementierung:

Die Funktion ist nicht implementiert.

Funktionsname:

i_requestElementname   i_requestElementname

Signatur:

i_requestElementname:: env -> objID -> string.

env: Das Environment.

objID: Objekt.

string: Dessen SGML-Elementname.

Beschreibung:

Diese Funktion gibt den SGML-Elementnamen des spezifizierten Objektes zurück.

Implementierung:

Der zugehörige `elementtype` wird im `Document`–Modul angefragt und anschließend dessen SGML–Elementname im `ISTypes`–Modul in Erfahrung gebracht.

Funktionsname:

i_getDocumentGraph i_getDocumentGraph

Signatur:

i_getDocumentGraph:: env -> objID -> nodeType -> (env, graphtrees).

env: Das Environment.

objID: Start–Objekt.

nodeType: Knotentyp.

(env, graphtrees): Das neue Environment und die Graphstruktur.

Beschreibung:

Diese Funktion liefert die logische Struktur des aktuellen Dokumentes als Graphen. Die Graphen–Datenstruktur entspricht der des Graphen–Visualisierungsprogrammes „daVinci" (TM). Näheres dazu auf Seite 119.

Implementierung:

Es wird eine entsprechende Funktion im `Document`–Modul aufgerufen.

Funktionsname:

i_getViewList i_getViewList

Signatur:

i_getViewList:: env -> strings.

env: Das Environment.

strings: Liste der Sichten.

Beschreibung:

Diese Funktion liefert eine Liste mit den Namen der in den Präsentationsregeln definierten Sichten.

Implementierung:

Es wird eine entsprechende Funktion im `PresRules`–Modul aufgerufen.

1.11.3 Lokale Implementation

1.11.3.1 Funktionen

Funktionsname:

breakDownElement breakDownElement

Signatur:

breakDownElement:: dtd -> elementtypeStruct -> insertPos -> (doc, objinfo) -> elementtype -> (doc, objinfo).

dtd: Die DTD-Datenstruktur.

elementtypeStruct: Die Datenstruktur der `elementtype`s.

insertPos: Einfügeposition.

(doc, objinfo): Das aktuelle Dokument und Informationen über das zuletzt eingefügte Objekt.

elementtype: SGML–Elementtyp des einzufügenden Objektes.

(doc, objinfo): Das geänderte Dokument und Informationen über das eingefügte Objekt.

Beschreibung:

Diese Funktion wird von `i_insertElement` in einem `foldl` aufgerufen, um eine Reihe von Objekten in das Dokument einzufügen. Sie wird mit der DTD, dem Speicher für `elementtype`s und der Einfügeposition teilweise parametrisiert, bekommt den Elementtyp des einzufügenden Objektes sowie das aktuelle Dokument und Infos über das zuletzt eingefügte Objekt herein[8] und liefert das neue Dokument mit Infos über das eingefügte Objekt zurück.
Nach Ende des gesamten `foldl`–Aufrufs sind also alle gewünschten Objekte eingefügt. Dabei wird berücksichtigt, daß die DTD für jeden Elementtyp dessen Aufbau vorschreibt und die Instanz einen gültigen Inhalt haben muß. Die dafür erforderlichen Objekte werden en passant erzeugt und ins Dokument eingesetzt.

Implementierung:

Zunächst wird überprüft, ob das einzufügende Element final ist — dies trifft zu, wenn sein Name „#PCDATA“, „#CDATA“, „#NDATA“ oder „#EMPTY“ ist. Dabei handelt es sich nach dem SGML–Standard nicht um Elemente, sondern um Bezeichner, die den Typ des Elementinhaltes angeben (nämlich verschiedene Sätze von ASCII–Zeichen bzw. Binärdaten). In ForauS haben wir jedoch entschieden, daß sie in der DTD als Elementtypen gespeichert werden, um die Schnittstelle zu diesem Modul einfach zu halten. Die Erkennung, daß es sich nicht um normale Elemente handelt, erfolgt in den Schnittstellenmodulen der IS, so wie hier.

In diesem Zusammenhang bedeutet das Auftreten eines solchen Elementnamens, daß es sich bei dem Objekt, in das eingefügt werden soll, bereits um ein Finales handelt. Eigentlich braucht nun nichts unternommen zu werden, da es ja kein Element gibt, das eingefügt

[8] Beim ersten Aufruf der Funktion ist dieses natürlich noch ein Dummywert.

werden müßte. Allerdings ist es sinnvoll, den Elementnamen als normalen Text in das Objekt zu schreiben, da sonst das umschließende Objekt nach dem Einfügen überhaupt nicht zu sehen wäre — dies wird auch getan.

Kennzeichnet der übergebene Elementtyp kein finales Element im eben beschriebenen Sinne, dann wird ein entsprechendes Objekt eingefügt und aus der DTD eine Liste mit möglichen, gültigen Inhalten dieses Elementes angefordert. Die erste Variante der Liste wird ausgewählt und das Einfügen zugehöriger Instanzen durch einen rekursiven Aufruf realisiert.

Abhängigkeiten:

Die Funktion verläßt sich darauf, daß `insertElement` aus dem `Document`–Modul bei der Angabe von `in` als Einfügeposition das einzufügende Objekt hinter die schon vorhandenen Objekte hängt. Dies ist nicht selbstverständlich, weil die Funktion eigentlich für das Einfügen in ein leeres Objekt gedacht ist.

1.11.4 Fehler und Restriktionen

Erst sehr spät haben wir erkannt, daß ein Austausch von Dokument–Objekten mit der ursprünglich zur Verfügung gestellten Funktionalität nicht durchgeführt werden kann, weil eine Realisierung durch Löschen des alten und Einfügen des neuen Objektes aufgrund der DTD unzulässig sein könnte. In diesem Fall hätte man nach dem Löschen einen nicht DTD-konformen Zustand des Dokumentes, was wir strikt ausgeschlossen haben.
Aus diesem Grund wurden die beiden Funktionen `i_requestExchangeElement` und `i_exchangeElement` erst in weit fortgeschrittenem Stadium der Implementierung eingeführt und nicht mehr ausprogrammiert.

1.12 ISPStructure

1.12.1 Funktionalität

Dieses Modul stellt eine Datenstruktur zur Verfügung, die das Schnittstellenmodul `ISParser` zum Speichern von Informationen benötigt.

1.12.2 Entwurf

Das Modul enthält keine expliziten Löschfunktionen für die gespeicherten Komponenten, weil das benutzende Modul `ISParser` über den Aufbau des Inhaltes sowieso informiert ist und deshalb selbst Werte in die Datenstruktur schreiben kann, die initialen Charakter haben. Einzige Ausnahme sind hier `clearFontStyleString` und `clearSuccList`.

1.12.3 Öffentliche Schnittstellen

1.12.3.1 Sorten

Sortenname:

isp isp

Signatur:

isp:: cisp
(v_attrs :: attributes)
(v_view :: string)
(v_element :: elementtype)
(v_attrname :: string)
(v_styles :: string)
(v_listtype :: objListType)
(v_succlist :: elementtypelist).

Beschreibung:

Diese Sorte nimmt die zu speichernden Informationen in getrennten Komponenten auf.

Sortenname:

objListType objListType

Signatur:

objListType:: ispObjList | ispVarObjList | ispNil.

Beschreibung:

Mit Werten dieses Typs merken sich die Funktionen im `ISParser`–Modul, welche Art von Folgeobjektliste vom Parser gerade bearbeitet wird. Dabei stellt `ispObjList` die normale Folgeobjektliste dar, während `ispVarObjList` für eine darin vorkommende Teilliste steht, die Elementtypen enthält, deren Instanzen im Dokument in beliebiger Reihenfolge vorkommen dürfen. `ispNil` bildet den Initialwert.

1.12.3.2 Funktionen

Funktionsname:

mtISP mtISP

Signatur:

mtISP:: isp.

isp: Leere Datenstruktur.

Beschreibung:

Diese Funktion gibt eine initialisierte Datenstruktur zurück.

Implementierung:

Als Werte für die einzelnen Komponenten werden leere Listen, Leerstrings sowie `mtElement-type` und `ispNil` eingesetzt.

Funktionsname:

appendAttribute appendAttribute

Signatur:

appendAttribute:: attributetype -> attributecontents -> isp -> isp.

attributetype: Attributtyp.

attributecontents: Attributwert.

isp: Alte ISP–Datenstruktur.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion speichert das übergebene Attribut in der Datenstruktur.
Sie wird zusammen mit `getAttribute` beim Parsen des Dokumentes benötigt, um die Attribute zwischenzuspeichern. Der Parser liefert die zu einem Objekt gehörigen Attribute nämlich vor dem Anmelden dieses Objektes, während das `Document`–Modul diese nach dem Anmelden haben möchte.

Implementierung:

Die Attribute werden in einer Liste gespeichert, an die das neue angehängt wird.

Funktionsname:

getAttribute getAttribute

Signatur:

getAttribute:: isp -> (boolean, isp, attributetype, attributecontents).

isp: ISP–Datenstruktur.

(boolean, isp, attributetype, attributecontents): Flag, ob ein Attribut vorhanden ist und wenn ja, deren Komponenten sowie die neue ISP–Struktur.

Beschreibung:

Diese Funktion gibt das am längsten gespeicherte Attribut zurück (FIFO-Methode). War noch eines vorhanden, ist der Boolwert `true` und das Attribut wird gelöscht. Ansonsten wird `false` zusammen mit Initialwerten für die Attributkomponenten zurückgeliefert.

Implementierung:

Das erste Element der Liste wird abgetrennt und zurückgegeben.

Die nachfolgenden Funktionen `setActViewName`, `getActViewName`, `setActElement` und `getAct-Element` werden beim Parsen der PR gebraucht, da eine Objekt- und Viewdefinition einmal eingeleitet wird, und sich alle in der Folge gemeldeten Attribute u.ä. darauf beziehen. Dem `PresRules`–Modul werden diese Daten jedoch separat jeweils mit der Angabe, zu welchem Objekt sie gehören, übermittelt.

Funktionsname:

setActViewName setActViewName

Signatur:

setActViewName:: isp -> string -> isp.

isp: Alte ISP–Datenstruktur.

string: Sichtname.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Der übergebene Sichtname wird gespeichert und der alte überschrieben.

Funktionsname:

getActViewName getActViewName

Signatur:

getActViewName:: isp -> string.

isp: ISP–Datenstruktur.

string: Der gespeicherte Sichtname.

Beschreibung:

Die Funktion liefert den gespeicherten Sichtnamen.

Funktionsname:

setActElement setActElement

Signatur:

setActElement:: isp -> elementtype -> isp.

isp: Alte ISP–Datenstruktur.

elementtype: Ein Elementtype.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion speichert einen `elementtype` in der Datenstruktur, der alte wird überschrieben.

Funktionsname:

getActElement getActElement

Signatur:

getActElement:: isp -> elementtype.

isp: ISP–Datenstruktur.

elementtype: Der gespeicherte Elementtype.

Beschreibung:

Diese Funktion gibt den gespeicherten `elementtype` zurück.

Funktionsname:

setActAttrName setActAttrName

Signatur:

setActAttrName:: isp -> string -> isp.

isp: Alte ISP–Datenstruktur.

string: Attributname.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Der übergebene Attributname wird gespeichert und der alte überschrieben.

Funktionsname:

getActAttrName getActAttrName

Signatur:

getActAttrName:: isp -> string.

isp: ISP–Datenstruktur.

string: Der gespeicherte Attributname.

Beschreibung:

Die Funktion liefert den gespeicherten Attributnamen.

Funktionsname:

appendFontStyle appendFontStyle

Signatur:

appendFontStyle:: isp -> string -> isp.

isp: Alte ISP–Datenstruktur.

string: Fontstyle–Angabe.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion hängt eine Fontstyle–Angabe an den FontStyle–String an.

Funktionsname:

getFontStyleString getFontStyleString

Signatur:

getFontStyleString:: isp -> string.

isp: ISP–Datenstruktur.

string: Der gespeicherte Fontstyle–String.

Beschreibung:

Diese Funktion gibt den gespeicherten FontStyle–String zurück.

Funktionsname:

clearFontStyleString clearFontStyleString

Signatur:

clearFontStyleString:: isp -> isp.

isp: Alte ISP–Datenstruktur.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion setzt den Fontstyle–String auf den initialen Wert (Leerstring) zurück.

Funktionsname:

setListtype setListtype

Signatur:

setListtype:: isp -> objListType -> isp.

isp: Alte ISP–Datenstruktur.

objListType: Listentyp.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion speichert die momentan vom Parser angemeldete Art von Folgeobjektliste.

Funktionsname:

getListtype getListtype

Signatur:

getListtype:: isp -> objListType.

isp: ISP–Datenstruktur.

objListType: Der gespeicherte Listentyp.

Beschreibung:

Diese Funktion liefert den momentan gespeicherten Objektlisten-Typ zurück.

Funktionsname:

appendElementToSuccList appendElementToSuccList

Signatur:

appendElementToSuccList:: isp -> elementtype -> isp.

isp: Alte ISP–Datenstruktur.

elementtype: Ein Elementtype.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion hängt den übergebenen `elementtype` an die Folgeobjektliste an.

Funktionsname:

getSuccList getSuccList

Signatur:

getSuccList:: isp -> elementtypelist.

isp: ISP–Datenstruktur.

elementtypelist: Die gespeicherte Folgeobjektliste.

Beschreibung:

Diese Funktion gibt die gespeicherte Folgeobjektliste zurück.

Funktionsname:

clearSuccList clearSuccList

Signatur:

clearSuccList:: isp -> isp.

isp: Alte ISP–Datenstruktur.

isp: Neue ISP–Datenstruktur.

Beschreibung:

Diese Funktion löscht die Folgeobjektliste.

Implementierung:

Speichert die leere Liste.

1.12.4 Lokale Implementation

1.12.4.1 Sorten

Sortenname:

attribute attribute

Signatur:

attribute:: attr (v_attrType :: attributetype) (v_attrCont :: attributecontents).

Beschreibung:

Diese Sorte stellt Werte für Dokument–Attribute zur Verfügung.

Sortenname:

attributes attributes

Signatur:

attributes:: [attribute].

Beschreibung:

Liste von Attributen.

2. Formatierer

2.1 Allgemeine Beschreibung

Das Formatierermodul besteht aus mehreren Moduln, die in diesem Kapitel beschrieben werden sollen. Es folgt zunächst ein Abschnitt über die Wirkungsweise des Formatierers. Im Weiteren werden die einzelnen Komponenten des Formatierers inkl. der Modulbeschreibungen vorgestellt.

2.1.1 Begriffe

Zunächst wollen wir einige grundlegende Begriffe definieren:

- **Logisches Element**
  Ein logisches Element ist ein Bestandteil der SGML-Struktur (z.B. „KAPITEL"). Jedem logischen Element ist eine eindeutige Typ-Id zugewiesen.

- **Textobjekt**
  Ein Textobjekt ist das Vorkommen eines logischen Elementes. Jedem Textobjekt ist eine eindeutige Objekt-Id zugewiesen.

- **Finales Element**
  Ein finales Element ist ein Textobjekt, das PCDATA enthält und explizit gesetzt werden muß. In unserer Datenstruktur sind die finalen Elemente die einzigen Elemente, denen intern Formatierungsattribute zugeordnet werden.

- **Nichtfinales Element**
  Ein nichtfinales Element ist ein Textobjekt, an dem in der Formatiererstruktur weitere Elemente angefügt sind. In einem Baum sind dies die Knoten, während die finalen Elemente die Blätter darstellen.

- **Physikalisches Element**
  Ein physikalisches Objekt ist ein durch den HLF erzeugtes Element. Es ist für die Einfachen Formatierer transparent. Physikalische Objekte enthalten Informationen wie z.B. die Kapitelnumerierung.

- **Seitenlayoutparameter**
  Dies sind Angaben zum Aussehen einer Seite, wie z.B. die Seitenränder, Textbreite, Abstand zum oberen und unteren Rand usw.

- **Textlayoutparameter**
  Dies sind Angaben, die das Aussehen eines Textelementes (z.B. ein Fließtextabsatz) bestimmen (z.B. Schriftart, -größe, -schnitt usw.).

- **PCDATA**
  Hiermit werden die Zeichen bezeichnet, die gesetzt werden sollen - also der Text an sich.

2.1.2 Beschreibung der Formatierer

Wir unterscheiden beim Vorgang des Formatierens drei verschiedene Formatierer:

1. Den High-Level-Formatierer innerhalb des Formatierers (HLF intern)
2. Die Einfachen Formatierer (EF – Das sind der Seitenformatierer und der Absatzformatierer)
3. Den High-Level-Formatierer als Schnittstelle zu den anderen Moduln. Interaktion mit Moduln außerhalb des Formatierers (HLF extern).

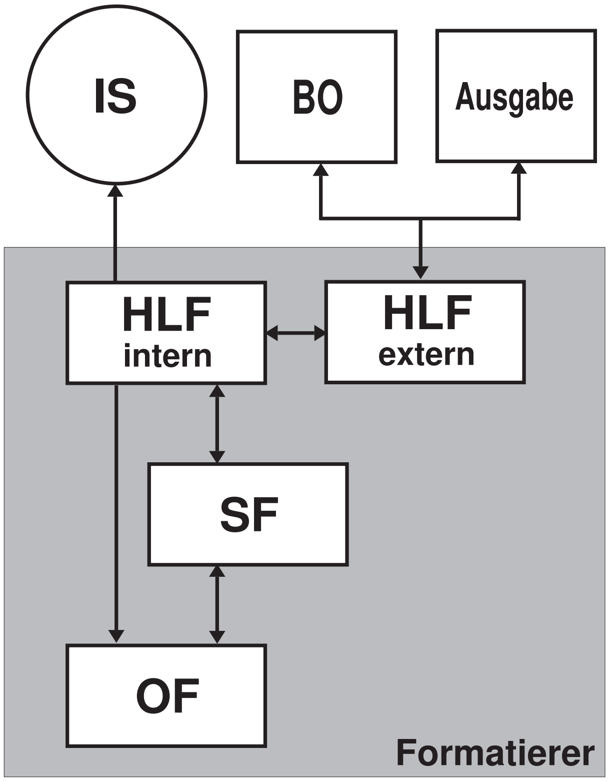

Abbildung 2.1: Gesamtübersicht: Die Interaktion des Formatierers mit anderen Moduln.

2.1.2.1 Der High-Level-Formatierer (HLF)

Da bei der inkrementellen Formatierung ein völlig anderer Modulaufruf als bei der ersten (initialen) Formatierung[1] abläuft, unterteilen wir die Beschreibung des HLFs in die Bereiche Intern und Extern.
Der HLF hat folgende Aufgaben zu erfüllen:

HLF intern

- Er erzeugt und verwaltet die Formatiererstruktur.
- Er ermöglicht den anderen Formatierern den Zugriff auf diese Struktur durch geeignete Funktionen.
- Er ist der Organisator innerhalb des Formatierers.
- Er führt den ersten Lauf der Formatierung durch.

HLF extern

- Er übernimmt die Kommunikation mit den Moduln außerhalb des Formatierers. Dies betrifft insbesondere das Ausgabemodul.

[1] siehe Abschnitt „Neuformatierung“

2.1.2.1.1 Der High-Level-Formatierer, intern Die Aufgabe des HLF ist es u.a. auf die Textobjekte der Internen Struktur zuzugreifen. Ebenso werden die Präsentationsregeln (PR) ausgelesen. Der HLF bestimmt die Reihenfolge der Textobjekte mit Hilfe der PR. Er liefert die finalen Elemente der so erzeugten Datenstruktur dann an die Einfachen Formatierer (EF). Eine solche Vorgehensweise impliziert, daß der HLF zunächst einmal das gesamte Dokument in die Formatierer-Struktur überträgt. Er führt einen ersten Lauf der Formatierung durch, bei dem die Kapitelnumerierungen u.ä. berechnet werden. Bei dieser initialen Formatierung erzeugt der HLF aus den Abhängigkeiten der Textobjekte aus der Internen Struktur eine eigene Struktur – die Formatiererstruktur. Hierbei handelt es sich um eine Baumstruktur. Auf diese Baumstruktur greifen die EF mittels geeigneter Funktionen zu.[2] Im Anschluß daran können die EF aufgerufen werden, um den Text zu setzen. In späteren Versionen soll der HLF zur Erzeugung von Verzeichnissen o.ä. aufgerufen werden können. Innerhalb des Formatierers sind nichtfinale Elemente nur dem HLF bekannt. Die Einfachen Formatierer bekommen vom HLF nur die finalen Elemente weitergereicht. Bis zu dieser Version werden Kapitelnumerierungen nicht von den EF sondern vom HLF selbst formatiert und gesetzt! Dies ist eine Ausnahme vom normalen Formatiervorgang.

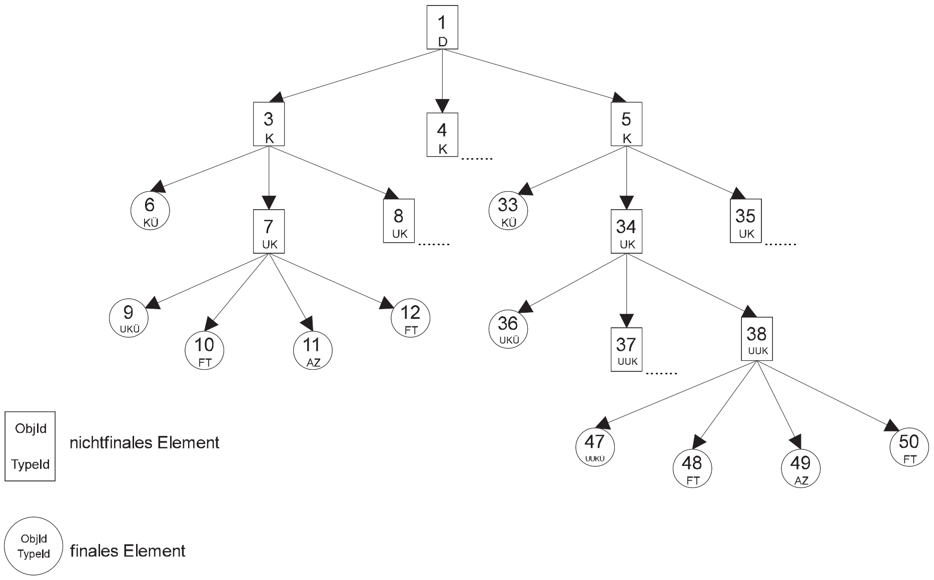

Abbildung 2.2: Ein Baum in der Formatiererstruktur.

Anhand dieses Beispiels wird die genaue Funktionsweise des HLFs erläutert. Zu Beginn teilt der HLF der Internen Struktur mit, daß er einen neuen Baum aufbauen möchte. Dann werden alle Textobjekte der ersten Ebene geliefert. In Abb. 2.2 entspricht dies dem „D“, was für Dokument steht. Falls mehrere Textobjekte vorliegen, muß mit Hilfe der Präsentationsregeln (PR) die Reihenfolge der Textobjekte festgelegt werden. Die Textobjekte werden nun in dieser Reihenfolge in den Baum eingefügt. Jedes neu eingefügte Textobjekt muß auf jeden Fall bei der späteren Formatierung direkt (finales Element) oder indirekt (nichtfinales Element) berücksichtigt werden. Daher enthält jeder Knoten und jedes Blatt ein Feld, das den Formatierstatus angibt. Für jedes gelieferte Textobjekt, das kein finales Element ist, werden wiederum rekursiv alle Textobjekte der Ebene (n+1) geliefert, was im Beispiel „K, K, K“ (was für Kapitel steht) ist.

[2] Die einzelnen Funktionen werden im Abschnitt „Modulbeschreibung Einfache Formatierer“ in der Dokumentation erläutert.

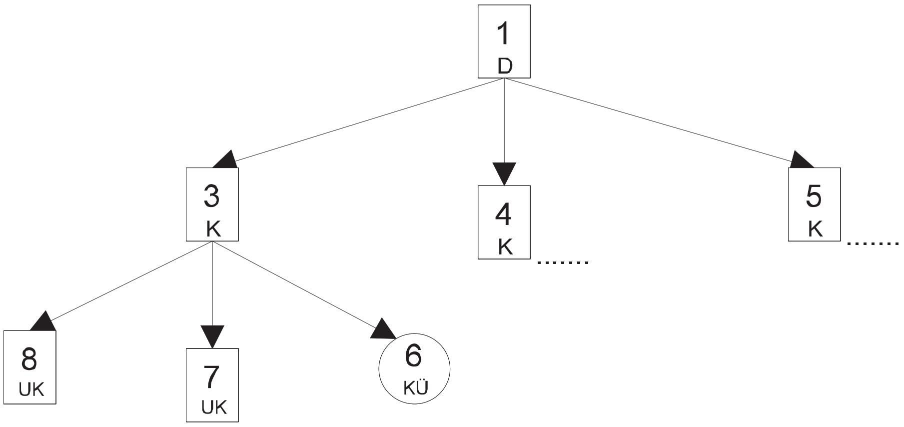

Abbildung 2.3: Eine Ebene des Baumes.

Im Beispiel wird für das Textobjekt „K“ mit der Objekt-Id 3 als Ergebnis aus der internen Struktur „UK, UK, KÜ“ geliefert (siehe Abb. 2.3). In den PR ist nun definiert, daß die Kapitelüberschrift vor dem ersten Unterkapitel erscheinen soll. Daher sind in der Formatiererstruktur diese Elemente anders sortiert, als in der Internen Struktur. Das gilt für alle Textobjekte K dieses Beispiels.
Stößt der HLF beim Aufbau der Formatiererstruktur auf ein finales Element (im Beispiel auf das Textobjekt „KÜ“ mit der Objekt-Id 6), so ist der entsprechende Ast komplett aufgebaut und der Rekursionsboden erreicht.
Die zu den finalen Elementen gehörigen Attribute, die das Aussehen des Textes bestimmen, werden nicht direkt in der Formatiererstruktur gespeichert. Sie lassen sich über die Typ-Id aber jederzeit aus den in der internen Struktur gespeicherten Präsentationsregeln (PR) auslesen. Für den HLF spielt das Layout (also das Aussehen) des Textes zu diesem Zeitpunkt noch keine Rolle. Erst die Einfachen Formatierer fordern sich die Layoutparameter zu den finalen Elementen bei der eigentlichen Wort- und Absatzformatierung an.
Der HLF erzeugt auch sogenannte „physikalische Objekte“. Dies sind z.B. die schon erwähnten Kapitelnumerierungen. Ein physikalisches Objekt kann vom Systembediener nicht direkt editiert werden. Es wird automatisch vom System durch den HLF erzeugt. Er reiht physikalische Objekte zwischen die anderen Textelemente ein.

2.1.2.1.2 Der High-Level-Formatierer, extern Während der interne HLF für den Aufbau und die Verwaltung der Formatiererdatenstruktur zuständig ist, organisiert der externe Teil das Zusammenspiel zwischen Moduln der Ausgabe und den Einfachen Formatierern.
Die Kernfunktionalität des externen HLF liegt in der Analyse der Dokumentveränderung durch den Benutzer und der Initierung entsprechender Formatierungsvorgänge durch die Einfachen Formatierer. Exemplarisch zeigen wir den Ablauf beim Einfügen eines Zeichens in den bestehenden, formatierten Text. Nachdem das Zeichen vom Benutzer entgegengenommen wurde, ruft die Benutzungsoberfläche den internen HLF auf. Dieser aktualisiert das sog. `changes`-Konstrukt, indem das Zeichen selbst und die aktuelle Cursorposition festgehalten wird. Anschließend fordert die Ausgabe vom externen HLF die Bildschirmveränderungen an. Hierzu analysiert der externe HLF durch das bereits gespeicherte `changes`-Konstrukt die angefallenen Änderungen: Ist ein Buchstabe oder Objekt eingefügt oder gelöscht worden? In diesem Fall kann die Position der Einfügemarke mit-

samt des Buchstabens ausgelesen werden. Jetzt wird die Formatierung der aktuellen Zeile durch die Einfachen Formatierer durchgeführt. Die als Ergebnis erhaltenen Parameter werden an die Ausgabe weitergeleitet und von ihr auf dem Bildschirm ausgegeben. Da die Einfachen Formatierer nur die Befugnis zur Veränderung der finalen Elemente haben, muß der externe HLF dafür sorgen, daß die nichtfinalen aktualisiert werden. Er aktualisiert in der Baumstruktur der Formatiererdaten evtl. geänderte Boxausmaße und Formatierungsstati.
Eine weitere häufig genutzte Funktionalität des externen HLF liegt in der Darstellung einer ganzen Seite. Da sich die Vorgänge vom oben beschriebenen unterscheiden, seien sie hier kurz umrissen: Zunächst wird überprüft, ob alle Objekte bis zum Seitenende formatiert sind. Ist dies nicht der Fall, wird eine entsprechende Formatierung veranlaßt. Wurde der Text bereits formatiert, wird eine Liste, die Wörter, deren Positionen und Auszeichnung enthält, aufgebaut und an die Ausgabe weitergereicht.
Schließlich sind im Modul des externen Formatierers noch einige Hilfsfunktionen für z.B. Postscript-Ausgabe und die Benutzungsoberfläche enthalten.

2.1.2.2 Die Einfachen Formatierer (EF)

Die Einfachen Formatierer (EF) unterteilen sich in drei Formatierer:

1. Objektformatierer (SimpleFormatter)
2. Seitenformatierer (PageFormatter)
3. Ausrichtungsformatierer (FormatAlignment)

2.1.2.2.1 Aufgabe der EF Der Objektformatierer nimmt sich ein finales Element und beginnt Wort für Wort einen Absatz zu formatieren[3]. Hierbei wird iterativ solange ein Wort hinter das andere „gesetzt“ (formatiert), bis das finale Element „leer“ ist. Die Layout-Parameter für ein zu setzendes (finales) Element bekommt der Objektformatierer über den HLF aus der Internen Struktur bzw. den Präsentationsregeln. Der Objektformatierer formatiert ohne Rücksicht auf Ausrichtung oder Seitenlängen. D.h. er formatiert ganze Boxen linksbündig, unabhängig davon, ob sie laut Präsentationsregeln in einer anderen Ausrichtung gesetzt werden sollen oder ob sie noch auf eine Seite passen.

Der Seitenformatierer hat nun die Aufgabe, die durch den Objektformatierer erzeugten Boxen auf einer Seite anzuordnen und im Bedarfsfall einen Seitenumbruch durchzuführen. Hierbei werden die Boxen ggf. in sog. Teil-/Subboxen aufgebrochen, wenn sie nicht komplett auf eine Seite passen oder die Seite schon vorher nahezu voll war. Die Teilboxen werden dann auf die nachfolgenden Seiten des Dokuments „verteilt“. Die beiden Einfachen Formatierer bekommen Textobjekte immer in der Reihenfolge, in der sie gesetzt werden. Im Speziellen sollte der Seitenformatierer die Kopf- und Fußzeilen sowie die Fußnoten setzen. Die hierfür notwendigen Informationen wären vom HLF ermittelt worden. Die Funktionalität ist allerdings noch nicht implementiert.

2.1.2.2.2 Ablauf der EF Die Einfachen Formatierer setzen alle finalen Elemente, die in der Formatiererstruktur des HLF gespeichert sind. Dazu muß der Seitenformatierer die Seitenlayoutparameter ermitteln. Der Objektformatierer erfragt dann für jedes finale Element über Funktionsaufrufe an den HLF die Objektparameter. Der Text des finalen Elements wird nun gemäß der

[3]Da unser System bis jetzt keine Silbentrennung beherrscht, ist das Wort die kleinste Einheit, die der Objektformatierer formatieren kann.

Text- und Seitenlayoutparameter auf mehrere Zeilen gebrochen und ausgerichtet. Die Umbruchstellen und die Größe und Lage der entstandenen Boxen werden dann gespeichert.
Ein Dokument besteht aus einer Liste von Textobjekten, wobei die Einfachen Formatierer nur die finalen Elemente setzen. Die nichtfinalen Elemente dienen lediglich der Repräsentation der Formatiererstruktur durch den HLF[4].

2.1.2.3 Ablauf des Formatierens

Wenn der Formatierer aufgerufen wird, gilt es zwei Zustände zu unterscheiden:

1. Neuformatierung
2. Inkrementelle Formatierung

Das Aufrufen des Formatierers geschieht jeweils durch das Ausgabemodul bzw. die Interne Struktur. Es/Sie stößt den Formatiervorgang durch den HLF (intern) an; dieser wiederum den Seitenformatierer. Letzterer ruft dann den Objektformatierer[5] und den Ausrichtungsformatierer auf.

2.1.2.3.1 Neuformatierung Bei der ersten Formatierung eines Dokuments ist die Datenstruktur des HLF leer. Daher wird zuerst der HLF aufgerufen, damit er ein Dokument initial formatiert und dabei die Formatiererstruktur aufbaut. Anschließend wird der Seitenformatierer aufgerufen, um die finalen Elemente vom Objektformatierer formatieren zu lassen. Die vom HLF erzeugte Datenstruktur wird dem Seitenformatierer dabei übergeben. Der Seitenformatierer richtet seine Anfragen nun an den HLF und übergibt jedesmal die komplette Datenstruktur an den HLF. Nach Beendigung der Formatierung durch den HLF und die Einfachen Formatierer wird die Struktur durch den HLF an die darüberliegende Funktion zurückgeliefert.

2.1.2.3.2 Inkrementelle Formatierung Die inkrementelle Formatierung soll das System richtig benutzbar machen. Selbstverständlich darf beim Einfügen eines Zeichens durch den Benutzer nicht jedesmal eine komplette Reformatierung des gesamten Dokuments durchgeführt werden. Das würde viel zu viel Zeit in Anspruch nehmen. Wir versuchen also nur die Teile neu zu formatieren, die von der Änderung (Einfügen eines einzelnen Zeichens) betroffen sind.

Die inkrementelle Formatierung geht wie folgt von statten:
Die Benutzungsoberfläche teilt dem internen HLF mit, was sich im Dokument in welchem Umfang wo geändert hat. Der interne HLF trägt diese Information in die Formatiererstruktur ein, reformatiert aber noch nichts. Im Anschluß ruft die Benutzungsoberfläche die Ausgabe zum Aufbau der veränderten Seite auf. Die Ausgabe leitet diese Anfrage sofort an ihre Schnittstelle zu den Formatierern (externer HLF) weiter. Hier wird überprüft, ob sich auf der entsprechenden Seite etwas verändert hat. Da dies der Fall ist, wird die Reformatierung des geänderten Bereichs (z.B. einer Zeile oder eines finalen Elementes) und der erneute Aufbau der Seite angestoßen.

Wir erhoffen uns dadurch, daß das System den Text während der Eingabe von Zeichen durch den Benutzer setzen und anzeigen kann, ohne daß unzumutbare Verzögerungen bei der Anzeige des Textes entstehen. Anderfalls wäre das System unbrauchbar.

[4] Nichtfinale Elemente werden im Abschnitt „HLF“ beschrieben.
[5] siehe auch „Die Einfachen Formatierer (EF)“

2.1.2.4 Das changes-Konstrukt

Das `changes`-Konstrukt dient der Ausgabe dazu, Bildschirmveränderungen effizient ausführen zu können. Es enthält drei Bereiche, die im Format des Dreiboxenkonzepts angegeben sind:

- den zu löschenden Bildschirmbereich,
- den einzufügenden Zeichenstrom,
- den zu verschiebenden Bereich.

Diesem Konzept liegt der Gedanke zugrunde, daß Bildschirmoperationen wie Block clear/move in grafischen Betriebssystemen viel schneller als ein komplettes Löschen mit anschließendem Neuzeichnen des Textes zu realisieren sind.

2.1.2.5 Das Boxenkonzept

Bei der Implementierung der Formatierermodule bemerkten wir, daß der geplante Ansatz zur Textrepräsentation in einer Box nicht ausreichend war, da innerhalb einer Zeile beginnender bzw. endender Text nicht beschrieben werden konnte. So wurde das Konzept um zwei, auf drei Boxen erweitert. Diese drei Boxen werden mit vier Koordinaten repräsentiert (siehe Abbildung 2.4).
Die erste Box beschreibt den Anfang eines Textblocks, der möglicherweise nur einen Teil einer Zeile belegt. Die zweite beschreibt den Kernteil, der vollständige Textzeilen enthält. Das Ende eines Textbereichs kann ebenfalls nur einen Teil einer Zeile belegen, deshalb ist eine dritte Box erforderlich.
Enthält die erste oder dritte Box keine Textinformationen, so liegen die Boxgrenzen der betroffenen Box auf denen der Hauptbox.

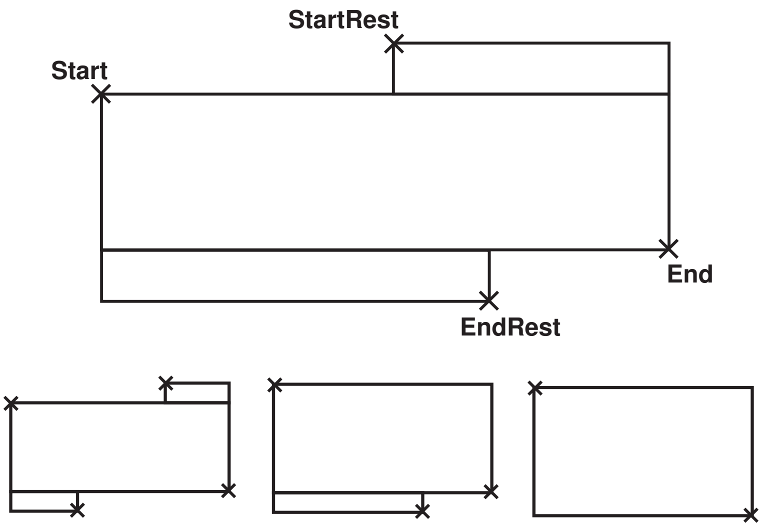

Abbildung 2.4: Die drei Boxen mit ihren Koordinaten.

2.1.3 Beschreibung eines finalen Elements

2.1.3.1 Datenfelder eines finalen Elementes

Den Unterschied zwischen der Datenstruktur der nichtfinalen und der finalen Elemente erläutern wir an dieser Stelle etwas ausführlicher. Anschließend folgt eine genaue Beschreibung der Sorten mit ihren Parametern.

Ein finales Element besteht aus einer Liste von Teilboxen. Die Teilboxen sind jeweils auf verschiedenen Seiten angeordnet. Findet ein finales Element auf nur einer Seite Platz, so gibt es auch nur eine Teilbox/Subbox.

2.1.3.2 Der Aufbau einer Teilbox

Eine Teilbox (`boxinfo`) besteht aus einer Liste von Zeilen (`[line]`). Die Anzahl der Zeilen der Teilbox werden in dem Attribut (`lastline`) abgelegt.
Zusätzlich enthält eine Teilbox die vier Koordinaten die zur Repräsentation der Box lt. Boxenkonzept 2.13.3.1 benötigt werden. Außerdem wird zu jeder Teilbox die Seitennummer gespeichert. Die Koordinaten einer Teilbox beziehen sich immer auf dieselbe Seite. Die Höhe der Teilbox läßt sich mit diesen Daten durch eine einfache Subtraktion bilden und wird daher nicht mit in die Datenstruktur aufgenommen.

2.1.3.3 Der Aufbau einer Zeile

Eine Zeile (`line`) besteht aus den Koordinaten der linken unteren Ecke des ersten Wortes, der Nummer des ersten Wortes und der Nummer des letzten Wortes der entsprechenden Zeile. Außerdem wird der in der Zeile noch zur Verfügung stehende Weißraum gespeichert. Diesen Wert bezeichnen wir als `emptySpace`.
In der Zeile werden weiterhin zwei Werte, die beim Setzen der Zeile in die Ausrichtung Blocksatz von nöten sind, abgelegt (`spacefactor` und `prevspaces`).

2.1.3.4 Ein einfaches Beispiel

Wir wollen nun anhand eines Beispiels das finale Element besser verdeutlichen.
Gegeben sei folgender Text, der einen Ausschnitt des Dokuments darstellt. Jede Zeile der Tabelle soll dabei eine Zeile des Dokuments entsprechen.

Objekt ID	**PCDATA**
8	... des Urwaldes.
9	Der Affe nahm die
	Banane vom Tisch, ohne
	rot zu werden.
10	Dabei...

Es ergibt sich nun folgende *(vereinfachte)* Datenstruktur für `textobj[9]`, also dem neunten Element von `document`.

line[1],line[2],line[3]	die einzelnen Zeilen
lastline=3	letzte Zeile des Objekts
(100,100)	linke obere Ecke des Objekts
(300,140)	rechte untere Ecke des Objekts

Die Zeilen haben dann folgenden Aufbau:

line	**1**	**2**	**3**
(x,y)	(100,100)	(100,118)	(100,136)
idfirst	1	5	9
idlast	4	8	11
emptyspace in pt	17	5	22

Wie aus dem Beispiel ersichtlich wird, ist das Komma ein Buchstabe des Wortes *Tisch,* der Punkt einer des Wortes *werden.*

Im folgenden wollen wir exemplarisch die Aufteilung eines Absatzes mit verschiedenen Textelementen darstellen.

2.1.3.5 Ein weiteres Beispiel

Dieses Beispiel soll zeigen, wie ein Absatz eines Dokumentes innerhalb der Formatiererstruktur repräsentiert wird.

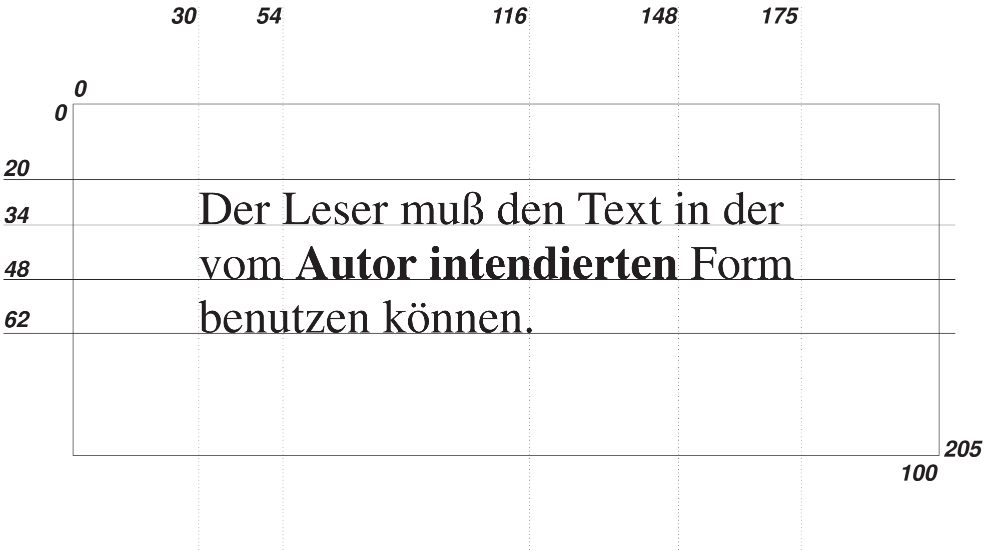

Abbildung 2.5: Text in Rasterbox mit Koordinaten.

Abbildung 2.5 zeigt den Absatz, wie er ausgegeben wird. Das hinterlegte Koordinatengitter bezeichnet die Aufteilung der aktuellen Seite in Pixel. Die auf den folgenden Abbildungen in den Boxinstanzen auftauchenden Koordinaten beziehen sich auf dieses Koordinatensystem. Die Koordinaten in den Zeileninstanzen beziehen sich jeweils auf die linke obere Ecke der Box, zu der die entsprechende Zeile gehört. Beginnt eine Box mitten auf einer Zeile, so beziehen sich diese Koordinaten auf die „gedachte“ linke obere Ecke der Box am Anfang dieser Zeile.

Der Absatz aus Abbildung 2.5 besteht aus drei finalen Elementen, die ihrer logischen Struktur folgend in einem nicht finalen Element „Absatz“ zusammengefaßt sind. Der Absatz kann nicht als ein finales Element behandelt werden, da Worte mit verschiedenem Layout in ihm vorkommen und ein finales Element immer nur ein Layout haben kann.

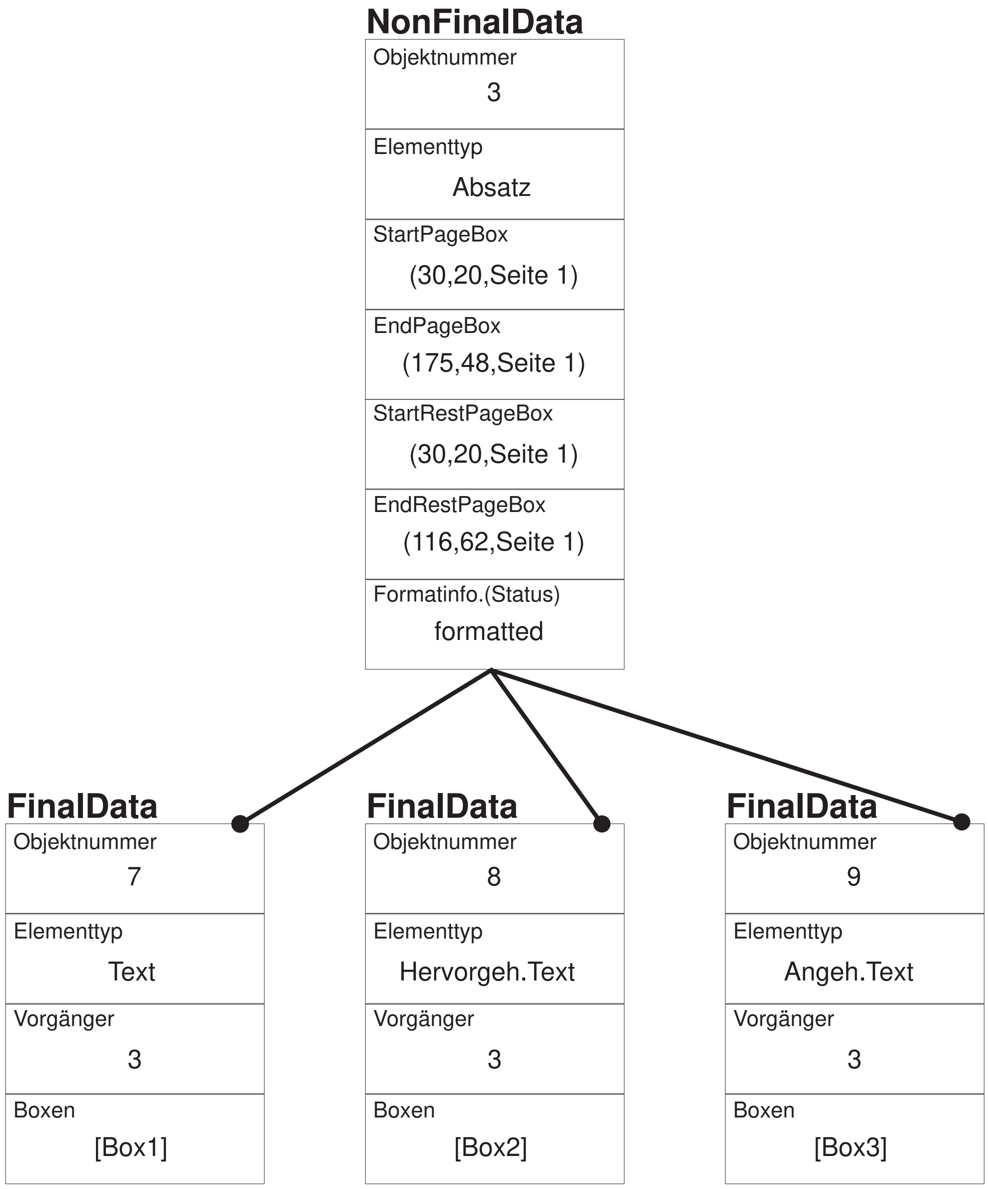

Abbildung 2.6: Dreigeteilte Säule.

Das erste finale Element ist vom Typ „Text“, das zweite vom Typ „HervorgehobenerText“ und das dritte vom Typ „Angehängter Text“. Das dritte finale Element kann nicht vom Typ „Text“ sein, da das Layout dieses Typs den Anfang eines neuen Absatzes induziert, was an dieser Stelle falsch wäre. Das nicht finale Element (siehe Abb. 2.6) beinhaltet weiterhin die Koordinaten mit den zugehörigen Seiten, die die Ausprägung dieses nicht finalen Elementes im Dokument darstellen.

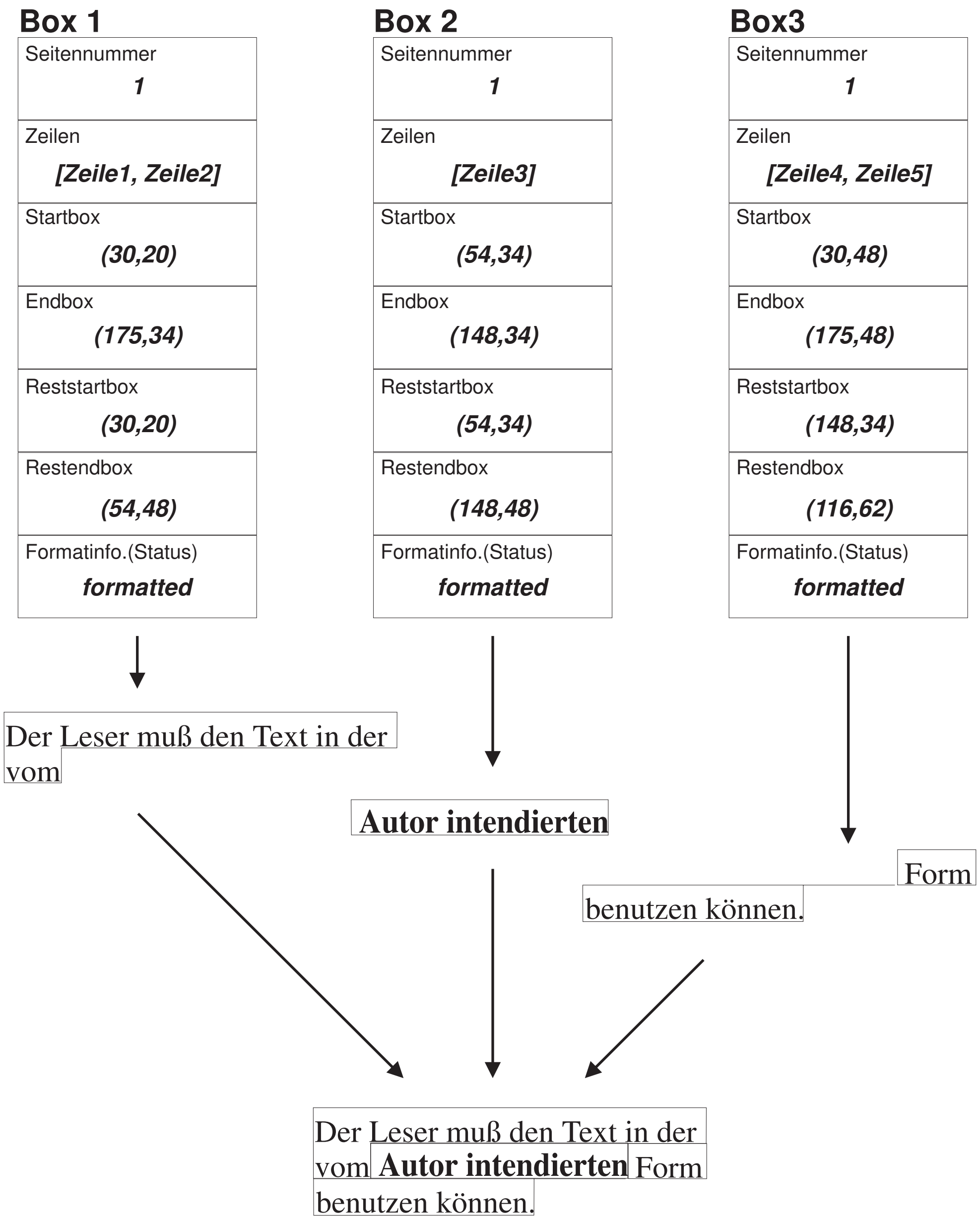

Abbildung 2.7: Drei Boxen bilden einen Absatz

Die drei finalen Elemente enthalten jeweils eine Box (ein finales Element enthält für jede Seite auf der sich Text dieses Elementes befindet genau eine Box). Diese drei Boxen zeigt Abb. 2.7. Die Boxen beinhalten jeweils die zu dem finalen Element gehörenden Zeilen dieser Seite und die Koordinaten, die die Ecken der Box definieren. Die drei Boxen zusammengesetzt ergeben den gesamten Absatz.

	Zeilenkoordinatden	erstes Wort	letztes Wort	Emptyspace	Spacefactor	Prevspaces	
Zeile 1:	*(0,14)*	*1*	*7*	*8*	*-*	*0*	Der Leser muß den Text in der
Zeile 2:	*(0,28)*	*9*	*9*	*131*	*-*	*0*	vom
Zeile 3:	*(24,14)*	*1*	*2*	*27*	*-*	*0*	**Autor intendierten**
Zeile 4:	*(118,14)*	*1*	*1*	*4*	*-*	*2*	Form
Zeile 5:	*(0,28)*	*2*	*3*	*59*	*-*	*0*	benutzen können

Abbildung 2.8: Die Zeilen des Absatzes

Die drei Zeilen des Absatzes werden innerhalb der Formatiererstruktur in fünf Zeilen abgelegt. Die erste und die letzte Zeile sind einfache Zeilen. Die mittlere wird aus drei Zeilen, die jeweils zu einer anderen Box (also auch zu einem anderen finalen Element) gehören, zusammengesetzt (siehe Abb. 2.8). Dieses Konzept ist nötig, um in einer Zeile Worte mit verschiedenem Layout hintereinander setzen zu können. Wenn z.B. jedes der zwanzig Worte einer Zeile ein verschiedenes Layout hätte, würde sie in der Formatiererstruktur durch zwanzig Boxen mit jeweils einer Zeile repräsentiert werden.

Die genaue Datenstruktur, die in der Formatiererstruktur gespeichert wird, ist auf den nächsten Seiten beschrieben (Finale Elemente).

2.1.3.6 Sortenbeschreibung der finalen Elemente

Sortenname:

fdata fdata

Signatur:

fdata:: fdata
(v_objID::objID)
(v_elementtype::elementtype)
(predecessor::integer)
(v_boxinfo::boxinfolist)

Beschreibung:

`(v_objID)` bezeichnet die Objekt-Identifikationsnummer eines finalen Elements, `(predecessor)` die Objekt-ID des Vorgängers des finalen Elements, und `(v_boxinfo)` beschreibt das Boxinfo/die Boxinfos, welche(s) weiter unten beschrieben ist

---

Sortenname:

line line

Signatur:

line:: line
(v_coordinates::coordinates)
(idfirst::integer)
(idlast::integer)
(emptyspace::integer)
(spacefactor::integer)
(prevspaces::integer)

Beschreibung:

`v_coordinates` definiert die linke untere Ecke des ersten Wortes der Zeile.
`idfirst` bezeichnet das erste Wort in der Zeile. Jedes Wort hat eingangs eine Wort-ID erhalten. Diese Wort-ID wird hier abgelegt. `idlast` bezeichnet das letzte Wort, das noch zur Zeile gehört. Der Freiraum, der noch für Buchstaben in der Zeile zur Verfügung steht ist in `emptyspace` gespeichert. `spacefactor` und `prevspaces` dienen dem Ausrichtungsformatierer zum Ablegen seiner Berechnungen (bei Ausrichtung Blocksatz).

Sortenname:

boxinfo boxinfo

Signatur:

boxinfo:: boxinfo
(pagenumber::integer)
(v_lines::lines) (lastline::integer)
(startbox::coordinates)

(endbox::coordinates)
(reststart::coordinates)
(restend::coordinates)
(pagebreakid::integer)
(baselinedistance::integer)
(verticalspace::integer)
(v_formatinfo::formatinfo)

Beschreibung:

`pagenumber` bestimmt die Seitennummer, auf der sich die Box befindet. `lastline` bezeichnet die Anzahl der Zeilen in einem Boxinfo. `startbox` ist ein Koordinatenpaar (x,y), welches die *linke obere Ecke der Grundbox* bezeichnet. `endbox` ist ein Koordinatenpaar (x,y), welches die *rechte untere Ecke der Grundbox* bezeichnet. `reststart` ist ein Koordinatenpaar (x,y), welches die *linke obere Ecke der berhangbox* (erste Zeile der Box beginnt evtl. nicht am linken Rand, wie die nachfolgenden Zeilen der Box) bezeichnet. `restend` ist ein Koordinatenpaar (x,y), welches die *rechte untere Ecke der unteren berhangbox* (letzte Zeile der Box endet i.d.R. nicht wie die vorausgegangenen Zeilen der Box) bezeichnet. Das Attribut `v_formatinfo` bezeichnet den Formatierungsstatus der Box.
Die Attribute `pagebreakid`, `verticalspace` und `baselinedistance` werden zur Zeit nicht genutzt.

2.2 HLF – Intern

2.3 FormatStructure

Bernd Rattey

2.3.1 Funktionalität

Das Modul definiert die Datentypen, die für die Formatierer benötigt werden. Das Modul stellt weiterhin die Routinen zum Zugriff auf die Formatierstruktur zur Verfügung. Zum Verständnis der einzelnen Datentypen und der darauf basierenden Funktionen dient der Abschnitt 2.1.3.

2.3.2 Entwurf

Das Modul muß grob gesagt fünf Aufgaben erfüllen.

Tabellen Zunächst werden zwei Grunddatentypen benötigt: `fdata` für finale Elemente und `nfdata` für nichtfinale Elemente. Die physikalischen Elemente sollen für die einfachen Formatierer transparent sein. Daher bietet es sich an, daß sie denselben internen Aufbau wie die finalen Elemente haben.
Die drei Daten werden in drei verschiedenen Arrays gespeichert. Als Index im Array dient die Objekt-ID. Das führt dazu, daß zu jedem Objekt höchstens eine der drei Tabellen mit einem sinnvollen Wert belegt ist.
Weiterhin liegt dem Dokument eine graphenähnliche Struktur zugrunde, daher wird der Graph auch in die drei Grunddatentypen mit aufgenommen, es werden zu jedem Objekt Nachfolger und Vorgänger gespeichert. Die Verweise können auch tabellenübergreifend sein,

ein finales Element hat beispielsweise immer einen nichtfinalen Vorgänger, der sich in einem anderen Array befindet. In der Nachfolger-Struktur (`successor`) ist daher u.a. auch der Status des Objekts gespeichert (`final`, ...). Auf die Weise ist feststellbar, in welchem Array man den Vorgänger oder Nachfolger finden kann.

Physikalische Objekte Alle Objekt-IDs physikalischer Elemente haben einen Wert, der eine bestimmte Grenze überschreitet. Alle finalen und nichtfinalen Elemente liegen unter der Grenze. Damit lassen sich die finalen und physikalischen Objekte leicht unterscheiden.
Für die Vergabe der Objekt-IDs der physikalischen Elemente ist der `HLF` verantwortlich. Daher wurde der Mechanismus in leicht veränderter Form aus der Internen Struktur übernommen. Es wird zum einen die nächste zu vergebende Nummer gespeichert. Angefangen wird mit `minPhysicalObjID`, dann wird aufsteigend fortgefahren. Zum anderen wird eine Freiliste geführt. In ihr werden wieder freigegebene IDs gespeichert. Bei der Vergabe der neuen IDs werden zunächst die aus der Freiliste genommen, dann neue gebildet.

Änderungen In der Formatierstruktur werden Art und Umfang inkrementeller Änderungen gespeichert. Hierzu dient der Datentyp `fschanges`.
Er wird von dem Teil des `HLF`, der die Daten für die Ausgabe bereitstellt (`FormatOutput`), benötigt. Da inkrementelle Änderungen von der `UI` nur dem Modul `FormatUI` mitgeteilt werden, müssen die Änderungen in `FormatUI` protokolliert werden. Dazu dient der Datentyp `fschanges`.
Im Modul `FormatStructure` muß der Datentyp definiert und die Zugriffsfunktionen realisiert werden.

Zugriff auf finale Elemente In der Formatierstruktur sind weiterhin alle Parameter gespeichert, die von den einfachen Formatierern gebraucht werden. Hierzu müssen Zugriffsfunktionen realisiert werden.
Wir haben uns dazu entschieden, den Zugriff auf die finalen Elemente hierarchisch aufzubauen, um den Umfang der zu übermittelnden Daten in den einzelnen Funktionsaufrufen gering zu halten. Der Aufbau der finalen Elemente ist in Abschnitt 2.1.3 auf S. 286 beschrieben. Dort sind auch die Bedeutungen der einzelnen Parameter zum Formatieren erklärt.
Soll nun beispielsweise der Wert der Seitennummer des finalen Elements ermittelt werden, so bedingt der hierarchische Aufbau folgende Vorgehensweise:

1. Ermitteln von `fdata` des Objekts
2. Ermitteln der `boxinfolist`
3. Ermitteln der ersten Box der Liste
4. Ermitteln der Seitennummer

Das sieht erst einmal umständlich aus. Der Vorteil liegt jedoch darin, daß der Formatierer in der Regel viele Daten braucht. Wenn er weitere Daten der Box braucht, muß er nicht das komplette `fdata` neu anfordern, sondern kann mit der erhaltenen Box arbeiten. Wenn die Funktion weitere Hilfsfunktionen aufruft, müssen die daher nicht mit der großen Struktur `fdata` sondern mit `boxinfo` gefüttert werden.
Diese Grundsatzentscheidung wurde zu Beginn des Programmierens getroffen, ob sie sich gelohnt hat, mag dahingestellt sein, funktionieren tut es aber.

Zugriff auf nichtfinale Elemente Hier wurde kein hierarchischer Zugriff gewählt, weil der Datentyp deutlich weniger Elemente enthält. Der Aufbau der nichtfinalen Elemente erfolgt auf der nächsten Seite.

2.3.2.1 Nichtfinale Elemente

In diesem Abschnitt folgt der Aufbau der nichtfinalen Elemente. Die Struktur ist soweit aufgeschlüsselt wie es hier notwendig ist. Der genaue Aufbau von `pagecoordinates` befindet sich in Abschnitt 2.1.3 auf S. 286. Der Datentyp `elementtype` stammt aus der Internen Struktur.

- (v_objID::objID)
- (v_elementtype::elementtype)
- (v_boxtype::boxtype)
- (v_graphinfo::graphinfo)
  - (predecessor::integer)
  - (v_successorlist::successorlist)
    - ▹ (v_ObjID::integer)
    - ▹ (v_final::finalstate)
- (startpagebox::pagecoordinates)
- (endpagebox::pagecoordinates)
- (startrestbox::pagecoordinates)
- (endrestbox::pagecoordinates)
- (v_formatinfo::formatinfo).
  - (status::formatstatus)
  - (pagelayout::boolean)
  - (visible::boolean)
  - (movement::integer).

Die nichtfinalen Elemente haben genau wie die finalen Elemente Informationen über Objekt-ID, Typ, Vorgänger und Nachfolger gespeichert. Die Daten dienen dem Realisieren des Graphen.
Die anderen Daten sind sehr ähnlich zu denen der Formatierdaten der finalen Elemente. Sie sind in den nichtfinalen Elementen enthalten, da ein nichtfinales Element mehrere finale Elemente enthalten kann, also einen größeren Bereich umspannen kann. Somit lassen sich mit einem nichtfinalen Element Aussagen über Bereiche machen. Falls beispielsweise das nichtfinale Objekt den Formatierstatus `formatted` hat, so sind alle darin befindlichen finalen Objekte ebenfalls formatiert.
Durch diese Lösung können viele Routinen effizienter werden.
Die Bedeutungen der einzelnen Formatierparameter sind in Abschnitt 2.1.3 auf S. 286 beschrieben.

2.3.3 Anmerkungen

Die Dokumentation der öffentlichen Schnittstellen gliedert sich in fünf Teile:

1. Die Sorten werden nicht alle detailliert beschrieben, da dies an anderer Stelle ausführlich dargestellt ist.[6] Die Sorten, die dort erklärt sind, werden hier nicht mehr dokumentiert. Es handelt sich um die Datentypen für die Parameter der einfachen Formatierer und den Aufbau der finalen und damit auch den der physikalischen Elemente.

2. Die Funktionen, die der internen Verwaltung der Tabellen dienen, sind detailliert ausgeführt.

3. Die Funktionen, die der Verwaltung der physikalischen Objekte dienen, sind detailliert ausgeführt.

4. Die Funktionen, die der Verwaltung der Änderungen dienen, sind detailliert ausgeführt.

5. Die Beschreibung der Zugriffsfunktionen auf die Daten der Formatierstruktur. Sie sind alle nach einem einheitlichen Schema aufgebaut und werden daher nicht einzeln in der Dokumentation ausgeführt, sondern in Gruppen zusammengefaßt. Der Aufbau der Hierarchie sollte dabei verstanden worden sein.

[6] s. S.286

2.3.4 Öffentliche Schnittstellen

2.3.4.1 Sorten

Die Sorten, bei denen die Signatur fehlt, sind an anderer Stelle ausführlich beschrieben (s.o).

Sortenname:

fs fs

Beschreibung:

Dies ist die gesamte Formatierstruktur. Sie enthält u.a. alle drei Arrays, also alle finalen, nichtfinalen und alle physikalischen Objekte.
Diese Sorte wird auch als `cookie` der Formatierer bezeichnet.
Eine genaue Beschreibung der Sorte erfolgt im lokalen Teil der Dokumentation, beginnend auf S.313.

Sortenname:

fdata fdata

Beschreibung:

In der Sorte sind sämtliche Formatierer-Informationen zu einem finalen Objekt gespeichert. Zusätzlich ist der Vorgänger als `integer` gespeichert.

Sortenname:

nfdata nfdata

Beschreibung:

In der Sorte sind sämtliche Formatierer-Informationen zu einem nichtfinalen Objekt gespeichert. Zusätzlich sind der Vorgänger und die Liste der Nachfolger in der geheimen Sorte `graphinfo` gespeichert (s.u.).

Sortenname:

boxinfo boxinfo

Beschreibung:

In der Sorte sind sämtliche Formatierer-Informationen zu einer `box` gespeichert. Eine `box` ist Teil eines finalen Elements.

Sortenname:

boxinfolist boxinfolist

Signatur:

boxinfolist:: [boxinfo]

Beschreibung:

Liste von `boxinfo`.

Sortenname:

line line

Beschreibung:

In der Sorte sind sämtliche Formatierer-Informationen zu einer `line` gespeichert. Eine `line` ist Teil einer `box`.

Sortenname:

lines lines

Signatur:

lines:: [line]

Beschreibung:

Liste von `line`.

Sortenname:

pagecoordinates pagecoordinates

Signatur:

pagecoordinates::
pagecoordinates (x::integer) (y::integer) (nr::integer)

Beschreibung:

In der Sorte sind die Seitenkoordinaten gespeichert, sie enthalten die Seitennummer, die x- und die y-Koordinate.

Sortenname:

coordinates coordinates

Signatur:

coordinates:: coordinates (x::integer) (y::integer)

Beschreibung:

In der Sorte sind die Koordinaten ohne die Seitennummern gespeichert. Sie enthalten die x- und die y-Koordinate.

Sortenname:

successor successor

Beschreibung:

In der Sorte werden die Nachfolger eines Objektes gespeichert. Der Datentyp dient der Realisierung des Graphen (s. 2.3.2). Eine genaue Beschreibung der Sorte erfolgt im lokalen Teil der Dokumentation.

Sortenname:

successorlist successorlist

Signatur:

successorlist:: [successor]

Beschreibung:

Liste von `successor`.

Sortenname:

formatstatus formatstatus

Signatur:

formatstatus:: formatted|nonformatted

Beschreibung:

Dieser Aufzählungsdatentyp beschreibt, ob das Objekt formatiert oder unformatiert ist.

Sortenname:

kindofchange kindofchange

Signatur:

kindofchange:: char|element

. **Beschreibung:**

Dieser Aufzählungsdatentyp beschreibt, ob beim inkrementellen Formatieren ein Zeichen oder ein Element eingefügt worden ist. Rein theoretisch können auch mehrere Zeichen gespeichert werden, dies wird jedoch von den anderen Modulen nicht unterstützt.

Sortenname:

fschange fschange

Signatur:

fschange:: fschange
(v_changeObjID::objID)
(v_kindofchange::kindofchange)
(v_firstcharacter::integer)
(v_changeChars::string)

Beschreibung:

Der Datentyp beschreibt, bei welchem Objekt (`v_changeObjID`) welche Art von Änderung (`v_kindofchange`) ab welchem Zeichen (`v_firstcharacter`) durchgeführt worden ist. Beim Einfügen von Text wird im letzten Parameter (`v_changeChars`) außerdem noch der Text gespeichert.

Sortenname:

fschanges fschanges

Signatur:

fschanges:: [fschanges]

Beschreibung:

Liste von `fschange`.

2.3.4.2 Funktionen

2.3.4.2.1 Basisfunktionen des HLF

2.3.4.2.1.1 Verwaltung der Tabellen

Funktionsname:

finaltablesize finaltablesize

Signatur:

finaltablesize:: integer

integer: Maximale Anzahl der finalen Elemente in der FS

Beschreibung:

Maximale Größe der Tabelle mit den finalen Elementen. Das Dokument darf auf keinen Fall mehr finale Elemente enthalten, sonst droht ein `Array out of index`.
Der momentane Wert beträgt 100.

Funktionsname:

nonfinaltablesize nonfinaltablesize

Signatur:

nonfinaltablesize:: integer

integer: Maximale Anzahl der nichtfinalen Elemente in der FS

Beschreibung:

Maximale Größe der Tabelle mit den nichtfinalen Elementen. Das Dokument darf auf keinen Fall mehr nichtfinale Elemente enthalten, sonst droht ein `Array out of index`.
Der momentane Wert beträgt 100.

Funktionsname:

mtNonFinalData mtNonFinalData

Signatur:

mtNonFinalData:: nfdata

nfdata: Nichtfinales Datenelement

Beschreibung:

Den Datensatz, den diese Funktion liefert, wird zurückgegeben, falls auf das Array über die Grenzen hinweg zugegriffen wird. Die Funktion wird von ASpecT zwingend bei der Definition der Tabelle benötigt.

Implementierung:

Die Funktion liefert `defaultNonFinalData`.

Funktionsname:

defaultNonFinalData defaultNonFinalData

Signatur:

defaultNonFinalData:: nfdata

nfdata: Standardwert für ein nichtfinales Datenelement

Beschreibung:

Liefert den Default-Datensatz für nichtfinale Elemente.

Funktionsname:

mtFinalData mtFinalData

Signatur:

mtFinalData:: fdata

nfdata: Finales Datenelement

Beschreibung:

Dieser Datensatz wird zurückgegeben, falls auf das Array über die Grenzen hinweg zugegriffen wird. Die Funktion wird von ASpecT zwingend bei der Definition der Tabelle benötigt.

Implementierung:

Die Funktion liefert `defaultFinalData`.

Funktionsname:

defaultFinalData defaultFinalData

Signatur:

defaultFinalData:: fdata

fdata: Standardwert für ein finales Datenelement

Beschreibung:

Liefert den Default-Datensatz für finale Elemente.

Funktionsname:

mt_fs mt_fs

Signatur:

mt_fs:: fs

fs: Formatierstruktur

Beschreibung:

Erzeugt die leere Formatierstruktur, indem die drei Tabellen, die Verwaltung der physikalischen Objekte und die Verwaltung der Änderungen initialisiert werden.

Funktionsname:

getStateOfObject getStateOfObject

Signatur:

getStateOfObject:: fs->objID->(finalstate,boolean)

fs: Formatierstruktur

objID: Objekt-ID des Objekts

finalstate: Status des Objekts

boolean: Flag, ob Objekt überhaupt vorhanden ist

Beschreibung:

Gibt den Typ des Objekts zurück: `FINALOBJ`, `NONFINALOBJ` oder `PHYSICALOBJ` und `TRUE`. Falls das Objekt undefiniert sein sollte, wird `NONFINALOBJ` und `FALSE` zurückgegeben.
Die Funktion gehört eigentlich in das Modul `FormatStrucUtil`, ist hier aber aus historischen Gründen hängengeblieben.

Implementierung:

Physikalische Objekte haben immer Objekt-IDs von mindestens `minPhysicalID`, daher ist die Unterscheidung zwichen finalen und physikalischen Objekten simpel. Bei der Unterscheidung zwischen finalen und nichtfinalen Objekten, wird geschaut, in welcher der beiden Tabellen im Index `objID` ein Wert steht, der ungleich dem Initialwert der Tabelle ist.

2.3.4.2.1.2 Verwaltung der physikalischen Elemente Eine Einordnung der Funktionen in das Gesamtkonzept befindet sich im Entwurf auf Seite 293. **Funktionsname:**

physicaltablesize physicaltablesize

Signatur:

physicaltablesize:: integer

integer: Maximale Anzahl der physikalischen Elemente in der FS

Beschreibung:

Maximale Größe der Tabelle mit den physikalischen Elementen. Das Dokument darf auf keinen Fall mehr physikalische Elemente enthalten, sonst droht ein `Array out of index`. Der momentane Wert beträgt 100 und wird von der Gruppe `IS` definiert.

Funktionsname:

minPhysicalID minPhysicalID

Signatur:

minPhysicalID:: integer

integer: Minimale Objekt-ID der physikalischen Elemente

Beschreibung:

Untere Grenze für die Objekt-ID der physikalischen Objekte. Alle IDs kleiner diesem Wert sind für finale und nichtfinale Elemente, alle IDs, die größer oder gleich sind, für physikalische Elemente vorgesehen.
Der momentane Wert beträgt 1000.

Funktionsname:

getPhysicalNewID getPhysicalNewID

Signatur:

getPhysicalNewID:: fs->(objID,fs)

fs [1]: Alte Formatierstruktur

obilD: Errechnete, neue ID des Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Für die Vergabe der Objekt-IDs der physikalischen Elemente ist der `HLF` verantwortlich. Daher wurde der Mechanismus in leicht veränderter Form aus der Internen Struktur übernommen.
Liefert ein Tupel, dessen erste Komponente die neue ID angibt, die dem nächsten physikalischen Objekt gegeben wird. In der zweiten Komponente wird die veränderte Formatierstruktur zurückgegeben. Es wurde dann entweder in der Freiliste ein Element entfernt (`objID`) oder aber die Grenze um Eins erhöht, falls die Liste leer war.

Funktionsname:

getPhysicalBorderID getPhysicalBorderID

Signatur:

getPhysicalBorderID:: fs->objID

fs: Formatierstruktur

objID: Größte Objekt-ID der physikalischen Elemente

Beschreibung:

Für die Vergabe der Objekt-IDs der physikalischen Elemente ist der `HLF` verantwortlich. Daher wurde der Mechanismus in leicht veränderter Form aus der Internen Struktur übernommen.
Die Funktion liefert die zur Zeit größte, gültige Objekt-ID eines physikalischen Elements.

Funktionsname:

releasePhysicalID releasePhysicalID

Signatur:

releasePhysicalID:: objID->fs->fs

objID: Objekt-ID

fs [1]: Alte Formatierstruktur

fs [2]: Neue Formatierstruktur

Beschreibung:

Für die Vergabe der Objekt-IDs der physikalischen Elemente ist der `HLF` verantwortlich. Daher wurde der Mechanismus in leicht veränderter Form aus der Internen Struktur übernommen.
Die Funktion trägt die `objID` in die Freiliste ein. Falls ein neues physikalisches Element angefordert wird, kann die ID wieder vergeben werden.

Funktionsname:

testPhysicalIDs testPhysicalIDs

Signatur:

testPhysicalIDs:: fs->(objID,objIDList)

fs: Formatierstruktur

objID: Ergebnis des Tests

objIDList: Ergebnis des Tests

Beschreibung:

Testet die Verwaltung der Vergabe der physikalischen Objekte.

Funktionsname:

mtString mtString

Signatur:

mtString:: string

string: Leerer Text

Beschreibung:

Liefert einen leeren String für die Tabelle der Strings der physikalischen Objekte.

Funktionsname:

setPhysicalString setPhysicalString

Signatur:

setPhysicalString:: fs->objID->string->fs

fs [1]: Alte Formatierstruktur

objID: Objekt-ID

string: Zu speichernder Text

fs [2]: Neue Formatierstruktur

Beschreibung:

Das physikalische Element `objID` besteht aus einem Text. Dieser wird mit der Funktion in der Tabelle der physikalischen Elemente gespeichert.

Implementierung:

Der Index im Array der Tabelle errechnet sich aus $objID - minPhysicalObjID$.

Funktionsname:

getPhysicalString getPhysicalString

Signatur:

getPhysicalString:: fs->objID->string

fs: Formatierstruktur

objID: Objekt-ID

string: Zu speichernder Text

Beschreibung:

Das physikalische Element `objID` besteht aus einem Text. Dieser wird mit der Funktion aus der Tabelle der physikalischen Elemente gelesen.

Implementierung:

Der Index im Array der Tabelle errechnet sich aus $objID - minPhysicalObjID$.

2.3.4.2.1.3 Verwaltung der der Änderungen ...

Eine Einordnung der Funktionen in das Gesamtkonzept befindet sich im Entwurf auf Seite 293.

Funktionsname:

mtFormatChanges mtFormatChanges

Signatur:

mtFormat:: fschanges

fschanges: Leere Struktur

Beschreibung:

Erzeugt eine leere Struktur für Änderungen in der Formatierstruktur.

Funktionsname:

getFormatChanges getFormatChanges

Signatur:

getFormatChanges:: fs->fschanges

fs: Formatierstruktur

fschanges: Änderungen

Beschreibung:

Liest alle Änderungen der Formatierstruktur aus.

Funktionsname:

setFormatChanges setFormatChanges

Signatur:

setFormatChanges:: fs->fschanges->fs

fs [1]: Alte Formatierstruktur

fschanges: Änderungen

fs [2]: Neue Formatierstruktur

Beschreibung:

Speichert alle Änderungen `fschanges` in der Formatierstruktur.

2.3.4.2.2 Zugriffsfunktionen

2.3.4.2.2.1 Nichtfinale Elemente ...

Funktionsname:

setNonFinalData setNonFinalData

Signatur:

setNonFinalData:: fs->objID->nfdata->fs

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des zu speichernden Objekts

nfdata: Zu speichernde Daten

fs [2]: Neue Formatierstruktur

Beschreibung:

Speichert ein nichtfinales Element in der Tabelle im Index `objID`.

Funktionsname:

getNonFinalData getNonFinalData

Signatur:

getNonFinalData:: fs->objID->nfdata

fs: Formatierstruktur

objID: Objekt-ID des zu lesenden Objekts

nfdata: Zu lesende Daten

Beschreibung:

Liest ein nichtfinales Element aus der Tabelle mit dem Index `objID`.

Es folgen die Gruppen für die nichtfinalen Elemente. Die Funktionsnamen beginnen entweder mit `getNonFinal` für das Lesen oder `setNonFinal` für das Schreiben von Parametern.

Funktionsname:

Gruppe ALLGEMEIN Gruppe ALLGEMEIN

Signatur:

Gruppe ALLGEMEIN:: nfdata->...

Beschreibung:

In die Gruppe fallen die Funktionen zum Lesen und Schreiben von:

- `objID`[7]
- `ElementType`
- `BoxType`
- `Predecessor`
- `SuccessorList`
- `Status`
- `PageLayout`
- `Visible`
- `Movement`
- `StartPageBox`
- `EndPageBox`
- `StartRestBox`
- `EndRestBox`

Funktionsname:

Gruppe SUCCESSORLIST Gruppe SUCCESSORLIST

Signatur:

Gruppe SUCCESSORLIST:: successor->...

Beschreibung:

In die Gruppe fallen die Funktionen zum Lesen und Schreiben der Operationen auf den Datentyp `successor` für die Nachfolger:

- `SuccessorKind`
- `SuccessorID`
- `mtSuccessor`

[7] Diese Funktion ist eigentlich überflüssig, da alle Zugriffe über die `objID` realisiert werden und die somit bekannt sein muß. Sie ist aus historischen Gründen enthalten.

2.3.4.2.2.2 Finale Elemente ...

Funktionsname:

setFinalData setFinalData

Signatur:

setFinalData:: fs->objID->fdata->fs

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des zu speichernden Objekts

fdata: Zu speichernde Daten

fs [2]: Neue Formatierstruktur

Beschreibung:

Speichert ein finales oder ein physikalisches Element in der Tabelle im Index `objID`.

Implementierung:

Falls das Objekt eine ID hat, die größer als `minPhysicalID` ist, handelt es sich um ein physikalisches Element. In dem Fall wird der Wert in die Tabelle der physikalischen Elemente geschrieben, sonst in die der finalen. Beim Schreiben in die physikalische Tabelle wird berücksichtigt, daß man `minPhysicalID` von der `objID` abziehen muß.

Funktionsname:

getFinalData getFinalData

Signatur:

getFinalData:: fs->objID->fdata

fs: Formatierstruktur

objID: Objekt-ID des zu lesenden Objekts

fdata: Zu lesende Daten

Beschreibung:

Liest ein finales oder ein physikalisches Element aus der Tabelle mit dem Index `objID`.

Implementierung:

Falls das Objekt eine ID hat, die größer als `minPhysicalID` ist, handelt es sich um ein physikalisches Element. In dem Fall wird der Wert aus der Tabelle der physikalischen Elemente gelesen, sonst aus der der finalen. Beim Schreiben in die physikalische Tabelle wird berücksichtigt, daß man `minPhysicalID` von der `objID` abziehen muß.

Es folgen die Gruppen für die finalen Elemente. Die Funktionsnamen beginnen entweder mit `getFinal` für das Lesen oder `setFinal` für das Schreiben von Parametern.

Funktionsname:

Gruppe ALLGEMEIN Gruppe ALLGEMEIN

Signatur:

Gruppe ALLGEMEIN:: fdata->...

Beschreibung:

In die Gruppe fallen die Funktionen zum Lesen und Schreiben von:

- `objID`[8]
- `ElementType`
- `Predecessor`
- `BoxInfoList`

Funktionsname:

Gruppe BOXINFO Gruppe BOXINFO

Signatur:

Gruppe BOXINFO:: boxinfo->...

Beschreibung:

In die Gruppe fallen die Funktionen zum Lesen und Schreiben von:

- `mtBoxInfoList`
- `splitBoxInfoList`
- `PageNumber`
- `Lines`
- `LastLine`
- `Start`
- `End`
- `RestStart`
- `RestEnd`
- `PageBreakID`
- `BaseLineDistance`
- `Status`
- `PageLayout`

[8] Diese Funktion ist eigentlich überflüssig, da alle Zugriffe über die `objID` realisiert werden und die somit bekannt sein muß. Sie ist aus historischen Gründen enthalten.

- `Visible`
- `VerticalSpace`
- `Movement`

Funktionsname:

Gruppe LINE Gruppe LINE

Signatur:

Gruppe LINE:: line->...

Beschreibung:

In die Gruppe fallen die Funktionen zum Lesen und Schreiben der Operationen auf den Datentyp `line`:

- `mtLine`
- `splitLineList`
- `Coordinates`
- `IDFirst`
- `IDLast`
- `EmptySpace`
- `SpaceFactor`
- `PrevSpaces`

2.3.5 Lokale Implementation

2.3.5.1 Sorten

Die Datentypen, die sich auf finale Elemente beziehen, werden nicht aufgeführt, da sie an anderer Stelle erklärt werden (s. Anmerkungen).
Ansonsten werden die bisher geheimen Sorten erklärt.

Sortenname:

fs fs

Signatur:

fs:: fs
(v_fdata::finaltable)
(v_nfdata::nonfinaltable)
(v_phydata::physicaltable)
(v_phyids::phyids)
(v_phystrings::physicalstringtable)
(v_changes::fschanges)

Beschreibung:

Dies ist die gesamte Formatierstruktur. Sie enthält u.a. alle drei Arrays, also alle finalen, nichtfinalen und alle physikalischen Objekte.
Diese Sorte wird auch als `cookie` der Formatierer bezeichnet.

Sortenname:

nfdata nfdata

Signatur:

nfdata:: nfdata
(v_objID::objID)
(v_elementtype::elementtype)
(v_boxtype::boxtype)
(v_graphinfo::graphinfo)
(startpagebox::pagecoordinates)
(endpagebox::pagecoordinates)
(startrestbox::pagecoordinates)
(endrestbox::pagecoordinates)
(v_formatinfo::formatinfo)

Beschreibung:

In der Sorte sind sämtliche Formatierer-Informationen zu einem nichtfinalen Objekt gespeichert. Zusätzlich sind der Vorgänger und die Liste der Nachfolger in der geheimen Sorte `graphinfo` gespeichert (s.u.).

Sortenname:

graphinfo graphinfo

Signatur:

graphinfo:: graphinfo
(predecessor::integer)
(v_successorlist::successorlist)

Beschreibung:

Der Datentyp speichert die Informationen, die zum Verwalten des Graphen in den Tabellen notwendig sind. Das sind der Vorgänger und die Liste der Nachfolger.

Sortenname:

successor successor

Signatur:

successor:: succ
(v_ObjID::integer)
(v_final::finalstate)

Beschreibung:

Der Datentyp speichert Objekt-ID und Status des Vorgängers.

Sortenname:

phyids phyids

Signatur:

phyids::
(v_objID::objID)
(v_objIDList::objIDList)

Beschreibung:

Es müssen für die physikalischen Elemente eindeutige Objekt-Nummern, sog. Objekt-IDs vergeben werden. Um den Nummernkreis so eng wie möglich zu halten, werden IDs von gelöschten Elementen neu vergeben. Es müssen also nicht nur die zuletzt vergebenen, sondern ebenfalls alle gelöschten Nummern, die noch nicht neu vergeben wurden, gespeichert werden. Dies geschieht mithilfe eines abstrakten Datentypen, der einerseits die nächste, neu zu vergebende Nummer, und einer Liste der alten, wieder vergebbaren Nummern beinhaltet.

2.3.5.2 Funktionen

Die folgenden lokalen Funktionen haben alle die Aufgabe, eine leere Tabelle zu erzeugen. Sie werden alle von der Funktion `mt_fs` benötigt.

Funktionsname:

mtNonFinalTable mtNonFinalTable

Signatur:

mtNonFinalTable:: nonfinaltable

nonfinaltable: Leere Tabelle

Beschreibung:

Funktion zum Erzeugen einer leeren nichtfinalen Tabelle mit `nonfinaltablesize` Einträgen.

Funktionsname:

mtFinalTable mtFinalTable

Signatur:

mtFinalTable:: finaltable

finaltable: Leere Tabelle

Beschreibung:

Funktion zum Erzeugen einer leeren finalen Tabelle mit `finaltablesize` Einträgen.

Funktionsname:

mtPhysicalTable mtPhysicalTable

Signatur:

mtPhysicalTable:: physicaltable

physicaltable: Leere Tabelle

Beschreibung:

Funktion zum Erzeugen einer leeren physikalischen Tabelle mit `physicaltablesize` Einträgen.

Funktionsname:

mtPhysicalStringTable mtPhysicalStringTable

Signatur:

mtPhysicalStringTable:: physicalstringtable

physicalstringtable: Leere Tabelle

Beschreibung:

Funktion zum Erzeugen einer leeren Tabelle mit `physicalstringtablesize` Einträgen. Jeder Eintrag ist ein String, der später den Text des physikalischen Elements enthält.

Funktionsname:

mtPhyIDs mtPhyIDs

Signatur:

mtPhyIDs:: ids

ids: Leere Tabelle

Beschreibung:

Erzeugt die leere Datenstruktur für das Verwalten der ObjIDs der physikalischen Objekte.

2.4 FormatStrucUtil

2.4.1 Funktionalität

Das Modul beinhaltet Routinen zum vereinfachten Umgang mit der `Formatierstruktur`, also häufig benutzte Hilfsfunktionen. Diese Routinen stellen eine zweite Schicht gegenüber `FormatStructure` dar.

2.4.2 Entwurf

Leider wurde dieses Modul erst recht spät geschaffen. Es wären sonst wohl erheblich mehr Funktionen in diesem Modul zu finden gewesen, so aber ist es eigentlich nur ein Versuch.

2.4.3 Öffentliche Schnittstellen

2.4.3.1 Sorten

Sortenname:

searchDirection searchDirection

Signatur:

searchDirection:: before | behind.

Beschreibung:

Diese Sorte ist für die Funktion `getNeighbour` (s.S. 2.4.3.2) eingeführt worden und dient zur Spezifikation einer Suchrichtung; `before` bedeutet in Richtung Textanfang, `behind` in Richtung Textende.

Sortenname:

searchModus searchModus

Signatur:

searchModus:: withPhysicals | withoutPhysicals.

Beschreibung:

Diese Sorte ist für die Funktion `getNeighbour` (s.S. 2.4.3.2) eingeführt worden und dient der Einschränkung auf bestimmte Objekte bei der Suche nach einem Vorgänger oder Nachfolger. Bei `withPhysicals` werden die physikalischen Objekte in die Suche einbezogen, bei `withoutPhysicals` nicht.

2.4.3.2 Funktionen

Funktionsname:

getObjIDSuccList getObjIDSuccList

Signatur:

getObjIDSuccList:: fs -> objID -> objIDList

fs: Formatierstruktur

objID: Objekt-ID des Objektes, für den alle Nachfolger bestimmt werden sollen

objIDList: Liste der Nachfolger des Objektes

Beschreibung:

Die Funktion liefert zu einer Objekt-ID die Objekt-IDs aller Nachfolger in einer Liste zurück. Falls `objID` ein finales Element identifiziert, wird die leere Liste zurückgeliefert.

Funktionsname:

getNeighbour getNeighbour

Signatur:

getNeighbour:: env -> objID -> searchDirection -> searchModus -> (env, boolean, objID).

env: Das Environment.

objID: Die ID des Objektes, dessen Nachbar ermittelt werden soll.

searchDirection: Richtung, in welcher der Nachbar gesucht werden soll (`before` oder `behind`).

searchModus: Angabe, ob bei der Suche physikalische Objekte miteinbezogen werden (`withPhysicals`) oder nicht (`withoutPhysicals`).

(env, boolean, objID): Das neue Environment, ein Flag für das Ergebnis der Suche und die ID des gefundenen Nachbar–Objektes.

Beschreibung:

Die Funktion bekommt die ObjID eines finalen oder nichtfinalen Objektes sowie eine Richtung übergeben und liefert das nächste finale Objekt, das in dieser Richtung liegt; also das unmittelbar davor- oder dahinterliegende. Bei Übergabe eines nichtfinalen Objektes wird das nächste finale Objekt geliefert, das außerhalb des übergebenen nichtfinalen liegt. Praktisch wird also das nächste nichtfinale Objekt gesucht und innerhalb dessen das am weitesten links oder rechts liegende finale Objekt.

Der `searchModus` gibt an, ob bei der Suche die physikalischen, vom Formatierer erzeugten Objekte übersprungen werden sollen (`withoutPhysicals`) oder nicht (`withPhysicals`). Das Überspringen der physikalischen Objekte wird z.B. bei der Cursorbewegung gebraucht.

Ist der zurückgegebene Boolwert `false`, dann liegt in dieser Richtung kein finales Objekt mehr. Vorsicht: Wenn ein nichtfinales Objekt übergegeben wird, dann heißt das nicht, daß in dieser Richtung überhaupt kein finales Objekt zu finden ist, sondern nur noch solche, die innerhalb dieses nichtfinalen Objektes liegen.

Implementierung:

Zunächst wird überprüft, ob das übergebene Objekt die Wurzel des Dokumentes ist; in diesem Fall ist die Suche nämlich ergebnislos abzubrechen, weil es auf dieser Ebene keine Nachbarn gibt. Anschließend werden die vorhergehenden bzw. nachfolgenden Objekte des übergebenen Objektes ermittelt und die Funktion `determinateFinal` auf diese angewendet. `determinateFinal` steigt selbständig in die tieferen Ebenen hinab, wenn sie auf ein nichtfinales Objekt stößt und liefert das gesuchte finale Objekt zurück, wenn es dies gibt. Da `determinateFinal` jedoch keine Überprüfung auf physikalische Objekte vornimmt, wird dessen Ergebnis entsprechend untersucht. Konnte kein finales Objekt gefunden werden, wird der Vater des übergebenen Objektes ermittelt und die Funktion rekursiv aufgerufen, da es ja möglich ist, daß man gerade den Nachbarn eines Objektes angefragt hat, hinter dem mehrere übergeordnete Objekte aufhören (oder vor ihm anfangen).

2.4.4 Lokale Implementation

2.4.4.1 Funktionen

Funktionsname:

determinateFinal determinateFinal

Signatur:

determinateFinal:: searchDirection -> (fs, boolean, objID) -> objID -> (fs, boolean, objID).

searchDirection: Richtung, in welcher der Nachbar gesucht werden soll (`before` oder `behind`).

(fs, boolean, objID): Ergebnisse des letzten Aufrufs (`foldl`).

objID: Die ID des zu untersuchenden Objektes.

(fs, boolean, objID): Formatierer–Datenstruktur, Flag für die erfolgreiche oder -lose Suche, ID des gefundenen finalen Objektes.

Beschreibung:

Diese Funktion dient dem Auffinden des linken oder rechten Nachbarn eines Objektes. Sie wird von `getNeighbour` im Zusammenhang mit einem `foldl` benutzt. Ist das übergebene Objekt (dritter Parameter) ein finales, dann wird es als das gefundene zurückgegeben (der Boolwert ist `true`); ist es jedoch nichtfinal, wird durch einen rekursiven Aufruf in diesem weitergesucht.

Implementierung:

Ein finales Objekt wird sofort mit `true` zurückgeliefert. Bei einem nichtfinalen werden die Objekte, die es beinhaltet, angefordert und bei der Suchrichtung `before` umgedreht. Anschließend findet mit dieser Liste ein Aufruf von foldl mit `determinateFinal` als Funktion statt, was ein Durchsuchen dieser Objekte auf das nächstliegende finale zur Folge hat.

2.4.4.2 Tests

Funktionsname:

testgetObjIDSuccList testgetObjIDSuccList

Signatur:

testgetObjIDSuccList:: system->system

system [1]: Altes System

system [2]: Neues System

Beschreibung:

Testet die Funktion `getObjIDSuccList` an einem Beispiel.

Vor- und Nachbedingungen:

Die Benutzungsoberfläche darf für den Test nicht aufgerufen werden.

Abhängigkeiten:

Zum Testen braucht man `FormatFirstPass`. Dies führt aber zu einem Zyklus bei den Importen, daher wird die Testfunktion ausgeklammert (9.6.1994).
Will man wieder testen, so muß man im Modul `FormatFirstPass` die Zeilen der Funktion `initFormatter` entfernen, die aus `FormatOutput` stammen und das Modul aus der Importliste entfernen.

2.5 FormatGraph

2.5.1 Funktionalität

Die gesamte, in `FormatStructure` definierte, Funktionalität setzt einen Graphen für das Verwalten der Daten voraus, dieser ist in der dort abgelegten Struktur impliziert. Das Modul verwaltet den Graphen, indem es Funktionen zum Aufbau, Löschen und Traversieren des Graphen bereitstellt.

2.5.2 Öffentliche Schnittstellen

2.5.2.1 Funktionen

Funktionsname:

insertNewNonFinalObject insertNewNonFinalObject

Signatur:

insertNewNonFinalObject:: fs->objID->objID->fs.

fs [1]: Alte Formatierstruktur

objID [1]: Objekt-ID des einzufügenden Objekts

objID [2]: Objekt-ID des Vorgängers des einzufügenden Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Setzt ein neues nichtfinales Objekt in den Graphen ein.

Vor- und Nachbedingungen:

Die Liste der Nachfolger vom Vorgänger muß korrekt sein, bereits entfernte Objekte tauchen dort also nicht mehr auf. Weiterhin darf die Funktion nicht mit der `objID=1` aufgerufen werden, da dieses Element keinen Vorgänger hat.

Implementierung:

Es werden bei jedem Aufruf 3 Arbeitsschritte durchgeführt:

1. Das Feld für das neue Objekt wird mit Default-Werten belegt.
2. In der Liste der Nachfolger beim Vorgänger wird das Objekt eingefügt. Im „Knoten" des Vorgängers wird der Nachfolger als nichtfinal markiert.
3. Der Vorgänger wird gesetzt.

Funktionsname:

insertNewFinalObject insertNewFinalObject

Signatur:

insertNewFinalObject:: fs->objID->objID->fs.

fs [1]: Alte Formatierstruktur

objID [1]: Objekt-ID des einzufügenden Objekts

objID [2]: Objekt-ID des Vorgängers des einzufügenden Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Setzt ein neues finales Objekt in den Graphen ein.

Vor- und Nachbedingungen:

Die Liste der Nachfolger vom Vorgänger muß korrekt sein, bereits entfernte Objekte tauchen dort also nicht mehr auf. Weiterhin darf die Funktion nicht mit der `objID=1` aufgerufen werden, da dieses Element immer nichtfinal ist.

Implementierung:

Es werden bei jedem Aufruf 3 Arbeitsschritte durchgeführt:

1. Das Feld für das neue Objekt wird mit Default-Werten belegt.
2. In der Liste der Nachfolger beim Vorgänger wird das Objekt eingefügt. Im „Knoten“ des Vorgängers wird der Nachfolger als final markiert.
3. Der Vorgänger wird gesetzt.

Funktionsname:

insertNewPhysicalObject insertNewPhysicalObject

Signatur:

insertNewPhysicalObject:: fs->objID->fs.

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Vorgängers des einzufügenden Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Setzt ein neues physikalisches Objekt in den Graphen ein.

Vor- und Nachbedingungen:

Die Liste der Nachfolger vom Vorgänger muß korrekt sein, bereits entfernte Objekte tauchen dort also nicht mehr auf. Weiterhin darf die Funktion nicht mit der `objID=1` aufgerufen werden, da dieses Element immer nichtfinal ist.

Implementierung:

Es werden bei jedem Aufruf 4 Arbeitsschritte durchgeführt:

1. Die Objekt-ID wird berechnet und als belegt gekennzeichnet.
2. Das Feld für das neue Objekt wird mit Default-Werten belegt.
3. In der Liste der Nachfolger beim Vorgänger wird das Objekt eingefügt. Im „Knoten“ des Vorgängers wird der Nachfolger als physikalisch markiert.
4. Der Vorgänger wird gesetzt.

Funktionsname:

deletePartOfGraph deletePartOfGraph

Signatur:

deletePartOfGraph:: fs->objID->fs.

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des zu löschenden Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Löscht einen Teilgraphen ab dem Element `objID`. Dabei kann es sich um ein einzelnes Element oder aber um Elemente handeln, an denen sich weitere Elemente befinden.

Implementierung:

Die Funktion ermittelt, ob es sich um ein nichtfinales oder ein finales Element handelt. Mit den finalen Elementen werden auch die physikalischen Elemente behandelt, da `FormatStructure` dies transparent behandelt.
Anschließend ruft die Funktion `deletePartOfGraphState` mit dem Typ und der Objekt-ID des Elements auf.

Funktionsname:

deleteSuccessorInList deleteSuccessorInList

Signatur:

deleteSuccessorInList:: fs->objID->objID->finalstate->fs.

fs [1]: Alte Formatierstruktur

objID [1]: Objekt-ID des zu löschenden Objekts

objID [2]: Objekt-ID des Vorgängers des zu löschenden Objekts finalstate: Status des zu löschenden Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Die Funktion löscht im Vorgängerobjekt (3. Parameter) die Objekt-Id des Nachfolgers (2. Parameter) aus der Liste der Nachfolger.

Implementierung:

Die Funktion setzt anstelle des alten Wertes einen Dummy-Wert. Dabei wird zwischen nichtfinalen und finalen Elementen unterschieden.

Funktionsname:

traverseGraph traverseGraph

Signatur:

traverseGraph:: env->fs->objID->(env->fs->objID->fs)->fs

env: Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Startobjekts

env->fs->objID->fs: Funktional

fs [2]: Neue Formatierstruktur

Beschreibung:

Traversiert den ganzen Graphen von der Wurzel hin zu den Blättern. Das Traversieren beginnt dabei beim Objekt `objID`. Die Funktion bekommt als letzten Parameter eine Funktion `f` vom Typ `env->fs-objID->fs`.
Das ist die Funktion, die auf jedes Element des Graphen angewendet wird. Es ist in ihr möglich, `fs` zu verändern, da sie vom Ergebnistyp `fs` ist. Die Funktion `f` wird auf jeden Fall auf das Element Objekt-ID angewandt. Falls es sich bei Objekt-ID um ein nichtfinales Element handelt, wird über die Interne Struktur die Liste der Nachfolger ermittelt und die Funktion `f` auf jedes Element angewendet.

Implementierung:

Bei nichtfinalen Elementen wird geprüft, ob es sich bei dem Objekt um die Wurzel des Graphen handelt. Ist dies der Fall, wird die Hilfsfunktion mit der Objekt-ID der Wurzel als einzigem Listenelement aufgerufen. Ist das Objekt nichtfinal und nicht die Wurzel des Baumes, wird die Hilfsfunktion mit allen Nachfolgern in einer Liste aufgerufen. Die Hilfsfunktion `traverseGraphHelp` übernimmt dann die eigentliche Arbeit.

Bei finalen Elementen wird nur die übergebene Funktion `f` ausgeführt.

Funktionsname:

traverseGraphEnv traverseGraphEnv

Signatur:

traverseGraphEnv:: env->fs->objID->(env->fs->objID->(env,fs))->(env,fs)

env [1]: Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Startobjekts

env->fs->objID->(env,fs): Funktional

env [2]: Neues Environment

fs [2]: Neue Formatierstruktur

Beschreibung:

Die Funktion arbeitet genauso wie `traverseGraph`, in ihr wird jedoch das Environment durchgereicht, um in ihm Fehler aufzunehmen.

Funktionsname:

countElementsInGraph countElementsInGraph

Signatur:

countElementsInGraph:: env->fs->objID->elementtype->
(env->fs->objID->elementtype->integer)->integer

env: Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Startobjekts

env->fs->objID->elementtype->integer: Funktional

integer: Anzahl der gefundenen Elemente

Beschreibung:

Die Funktion zählt die Anzahl der Objekte im Graphen, die den Typ `elementtype` haben. Das Traversieren beginnt dabei beim Objekt `objID`. Die Funktion bekommt als letzten Parameter eine Funktion `f` vom Typ `env->fs-objID->fs`.
Das ist die Funktion, die das eigentliche Zählen übernimmt. Falls es sich bei Objekt-ID um ein nichtfinales Element handelt, wird über die Interne Struktur die Liste der Nachfolger ermittelt und die Funktion `f` auf jedes Element angewendet.

Implementierung:

Bei nichtfinalen Elementen wird geprüft, ob es sich bei dem Objekt um die Wurzel des Graphen handelt. Ist dies der Fall, wird die Hilfsfunktion mit der Objekt-ID der Wurzel als einzigem Listenelement aufgerufen. Ist das Objekt nichtfinal und nicht die Wurzel des Baumes, wird die Hilfsfunktion mit allen Nachfolgern in einer Liste aufgerufen. Die Hilfsfunktion `countInGraphHelp` übernimmt dann die eigentliche Arbeit.
Bei finalen Elementen wird nur die übergebene Funktion `f` ausgeführt.

2.5.3 Tests

In das Modul sind vier Tests eingebaut.

Funktionsname:

smallphyx smallphyx

Signatur:

smallphyx:: system->system

system [1]: Altes System

system [2: Neues System

Beschreibung:

Das `x` steht für die Buchstaben 1 bis 4. Jede Funktion führt einen kleinen Test aus, indem ein Graph mit physikalischen Objekten erzeugt wird.

Vor- und Nachbedingungen:

Die Werte der Internen Struktur haben keinen Einfluß auf die Tests.

2.5.4 Lokale Funktionen

Funktionsname:

deletePartOfGraphState deletePartOfGraphState

Signatur:

deletePartOfGraphState:: fs->objID->finalstate->fs

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des zu löschenden Objekts

finalstate: Status des zu löschenden Objekts

fs [2]: Neue Formatierstruktur

Beschreibung:

Wird immer von `deletePartOfGraph` aufgerufen. Die Funktion hat im Unterschied dazu den Status des Objekts und veranlaßt dann das Löschen des Objekts abhängig vom Status. Im Falle nichtfinaler Elemente wird dazu die Hilfsfunktion `deletePartOfGraphHelp` benutzt.

Vor- und Nachbedingungen:

Es wird davon ausgegangen, daß das erste Objekt den Vorgänger mit dem Wert 0 hat.

Implementierung:

Die Funktion unterscheidet beim Löschen zwischen finalen und nichtfinalen Elementen. Jede der beiden Fälle wird mit Pattern Matching abgefangen:

NONFINALOBJ Die Funktion löscht alle Elemente aus dem Graphen, die am nichtfinalen Element `ObjId` hängen. Das nichtfinale Element `ObjId` selbst wird auch gelöscht. Dazu werden zunächst die Nachfolger des ersten Objekts ermittelt und dann die Hilfsfunktion `deletePartOfGraphHelp` aufgerufen, die das eigentliche Löschen übernimmt. Danach wird beim Vorgänger des Objekts dieses aus der Liste der Nachfolger gestrichen, falls es sich beim aktuellen Objekt nicht um die Wurzel des Graphen handelt (dann ist `Predecessor`=0). Abschließend wird das Objekt selbst gelöscht.

FINALOBJ Die Funktion löscht das Element und entfernt den Eintrag in der Liste der Nachfolger beim Vorgänger.

PHYSICALOBJ Die Funktion löscht das Element und entfernt den Eintrag in der Liste der Nachfolger beim Vorgänger.
Weiterhin wird aus der Liste der Objekt-IDs für physikalische Objekte die gelöschte ID in eine Freiliste eingetragen, damit sie wieder vergeben werden kann.

Funktionsname:

deletePartOfGraphHelp deletePartOfGraphHelp

Signatur:

deletePartOfGraphHelp:: fs->successorlist->fs

fs [1]: Alte Formatierstruktur

successorlist: Liste der zu löschenden Objekte

fs [2]: Neue Formatierstruktur

Beschreibung:

Die Funktion löscht die physikalischen, finalen bzw. nichtfinalen Elemente, die sich in der `successorlist` befinden.

Implementierung:

Dazu werden die Tabelleneinträge auf ihre Defaultwerte gesetzt.
Ist das Element nichtfinalen Typs, so werden danach rekursiv alle Nachfolger untersucht und dann mit dem Rest der Liste weitergemacht.

Funktionsname:

traverseGraphHelp traverseGraphHelp

Signatur:

traverseGraphHelp:: env->fs->(env->fs->objID->fs)->objinfolist->fs.

env: Altes Environment

fs [1]: Alte Formatierstruktur

env->fs->objID->fs: Funktional

objinfolist: Liste der zu traversierenden Objekte

fs [2]: Neue Formatierstruktur

Beschreibung:

Hilfsfunktion für `traverseGraph`. Realisiert das Anwenden von `f` auf jedes Element der darunterliegenden Ebenen bei nichtfinalen Elementen und für jedes Element der aktuellen Ebene.

Funktionsname:

traverseGraphEnvHelp traverseGraphEnvHelp

Beschreibung:

Funktioniert analog wie `traverseGraphHelp`, wird aber von `countElementsInGraph` aufgerufen.

2.6 FormatFirstPass

2.6.1 Funktionalität

Das Modul enthält die Funktionen, die notwendig sind, um den ersten Lauf der Formatierung zu übernehmen. Hierunter fällt das Aufbauen des Graphen, das Überprüfen der Reihenfolge der Elemente und das Numerieren der Kapitel. Außerdem enthält das Modul Funktionen, um die Objekte des Graphen zu markieren, z.B. um sie als `nonformatted` zu setzen.
Die Funktionen werden zum einen bei einer initialen Formatierung benötigt, sie werden aber auch beim inkrementellen Formatieren benötigt.

2.6.2 Öffentliche Schnittstellen

2.6.2.1 Funktionen

Funktionsname:

initFormatter initFormatter

Signatur:

initFormatter:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Die Funktion führt das initiale Formatieren durch. Hierunter fällt das Aufbauen des Graphen, das Überprüfen der Reihenfolge der Elemente und das Numerieren der Kapitel. Alle Änderungen werden in der Formatierstruktur gespeichert.

Vor- und Nachbedingungen:

Die Interne Struktur muß bereits initialisiert sein.

Implementierung:

Es werden folgende Dinge getan:

1. Aufbauen des Graphen
2. Ermitteln der korrekten Reihenfolge
3. Einfügen der physikalischen Objekte
4. Berechnen der Kapitelnumerierung
5. Initiale Formatierung
6. Abspeichern der Formatierstruktur in `env`

Funktionsname:

buildPartOfGraph buildPartOfGraph

Signatur:

buildPartOfGraph:: env->fs->objID->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Startobjekt des Graphenaufbaus

fs [2]: Neue Formatierstruktur

Beschreibung:

Baut den Graphen des Dokumenten in der Formatierstruktur ab dem Objekt `ObjId` komplett neu auf.

Vor- und Nachbedingungen:

Die Interne Struktur muß bereits initialisiert sein.
Außerdem muß der Graph für das Element mit der `ObjId=1` bereits erzeugt worden sein.

Implementierung:

Baut den Graphen ab dem Objekt `ObjId` auf, indem rekursiv für jedes nichtfinales Element `i_getNextTextObject` aufgerufen wird. Die Datenstruktur `fs` wird dabei jeweils mit durchgereicht.
Die Funktion benutzt die Hilfsfunktionen `buildID` und `buildLevelItem`, die den Rekursionsprozeß bilden. Die eigentliche Rekursion steckt in `buildID`, die über `buildLevelItem` für jedes Element aufgerufen wird. Um den Baum komplett aufzubauen, muß `buildPartOfGraph` mit der `ObjId` des Objekts der Wurzel des Dokuments (1) aufgerufen werden.
Der Graph muß für das Element mit der `ObjId`=1 bereits erzeugt worden sein.

Funktionsname:

buildGraph buildGraph

Signatur:

buildGraph:: env->fs->(env,fs)

env [1]: Altes Environment

fs [1]: Alte Formatierstruktur

env [2]: Neues Environment

fs [2]: Neue Formatierstruktur

Beschreibung:

Baut den Graphen des Dokumenten in der Formatierstruktur ab der Wurzel komplett neu auf.

Vor- und Nachbedingungen:

Die Interne Struktur muß bereits initialisiert sein.

Implementierung:

Die Funktion veranlaßt das Aufbauen des ganzen Graphen über die Funktion `buildPartOfGraph`. Zunächst wird das erste Element manuell eingefügt und dann `buildPartOfGraph` aufgerufen. Falls das erste Element (die Wurzel) final sein sollte oder gar nicht existiert, wird nichts gemacht.

Funktionsname:

buildOrder buildOrder

Signatur:

buildOrder:: env->fs->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

fs [2]: Neue Formatierstruktur

Beschreibung:

Prüft die Reihenfolge für alle Elemente des Graphen, falls das erste Element des Graphen nichtfinal ist. Falls das erste Element final oder physikalisch ist, wird nichts gemacht. Eigentlich kann letzterer Fall nie eintreten, da die Daten von der Internen Struktur kommen und diese keine physikalischen Objekte im Sinne der Formatierer kennt.

Vor- und Nachbedingungen:

Die Interne Struktur muß bereits initialisiert sein.

Implementierung:

Für nichtfinale Elemente ruft die Funktion `buildPartOfOrder` auf.

Funktionsname:

buildPartOfOrder buildPartOfOrder

Signatur:

buildPartOfOrder:: env->fs->objID->elementtype->finalstate->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Startobjekt des Prozesses

elementtype: Elemententyp des Startobjekt des Prozesses

finalstate: Status des Startobjekt des Prozesses

fs [2]: Neue Formatierstruktur

Beschreibung:

Veranlaßt das Setzen der korrekten Reihenfolge ab dem Objekt `objID`.

Vor- und Nachbedingungen:

Die Interne Struktur muß bereits initialisiert sein.

Implementierung:

Setzt für das Element mit den Werten `objID`, `elementtype` und `finalstate` die korrekte Reihenfolge in der aktuellen Ebene (`buildOrderOfLevel`) und arbeitet dann rekursiv für jedes Element der darunter liegenden Ebenen weiter (`buildPartOfOrderHelp`). Dies alles geschieht nur, wenn es sich jeweils um ein nichtfinales Element handelt. Bei andernen Elementen wird nichts gemacht.

Funktionsname:

buildPhysicalObjs buildPhysicalObjs

Signatur:

buildPhysicalObjs:: env->fs->objID->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Startobjekt des Prozesses

fs [2]: Neue Formatierstruktur

Beschreibung:

Ruft die Funktion zum Aufbauen der physikalischen Objekte auf. Es wird im Graphen mit dem Objekt `objID` begonnen.

Vor- und Nachbedingungen:

Die Interne Struktur muß bereits initialisiert sein.

Implementierung:

Ruft die Funktion `traverseGraph` mit dem Funktional `buildPhysicalSingle` auf.

Funktionsname:

buildAllCounter buildAllCounter

Signatur:

buildAllCounter:: env->fs->objID->(env,fs)

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Startobjekt des Prozesses

env [2]: Neues Environment

fs [2]: Neue Formatierstruktur

Beschreibung:

Bildet im Graphen ab dem Objekt `objID` alle Numerierungen für Kapitel, Unterkapitel, usw. .

Vor- und Nachbedingungen:

Die physikalischen Objekte müssen mit der Funktion `buildPhysicalObjs` erzeugt worden sein.

Implementierung:

Es werden zwei Läufe durchgeführt:

1. Ermitteln der Nummer des Objekts (`buildAllCounterSingle`).
2. Vor diesen Wert wird der Wert vorheriger Numerierungen konkateniert, falls es sich um ein Unterkapitel oder darunterliegendes Objekt handeln sollte (`buildComplete-CounterSingle`).

Man benötigt zwei Läufe, da es passieren kann, daß die Kapitelnummer nach der Unterkapitelnummer berechnet wird, da die Objekte in ihrer physikalischen und nicht in ihrer logischen Reihenfolge vorliegen.
Jeder der Läufe wird mit der Funktion `traverseGraphEnv` und dem entsprechenden Funktional ausgeführt.

2.6.2.2 Tests

Funktionsname:

initTestFormatter initTestFormatter

Signatur:

initTestFormatter:: system->(env,fs)

system: Altes System

env: Neues Environment

fs: Neue Formatierstruktur

Beschreibung:

Die Funktion führt das initiale Formatieren für einen Formatierertest durch. Hierunter fällt das Aufbauen des Graphen und das Überprüfen der Reihenfolge der Elemente. Die Änderungen werden nicht im Environment gespeichert, sondern in der Formatierstruktur zurückgegeben.

Vor- und Nachbedingungen:

Das Dokument muß in `ISFormat.AS` hartverdrahtet sein.

Implementierung:

Es werden folgende Dinge getan:

1. Aufbauen des Graphen
2. Ermitteln der korrekten Reihenfolge

Funktionsname:

withoutphysical withoutphysical

Signatur:

withoutphysical:: system->system

system [1]: Altes System

system [2]: Neues System

Beschreibung:

Baut den Baum komplett auf und gibt alles aus, physikalische Elemente werden dabei nicht berücksichtigt.

Vor- und Nachbedingungen:

Das Dokument muß in `ISFormat.AS` hartverdrahtet sein.

Funktionsname:

countChapter countChapter

Signatur:

countChapterx:: system->system

system [1]: Altes System

system [2]: Neues System

Beschreibung:

Es werden verschiedene Tests zum Überprüfen der Korrektheit der Kapitelnumerierung ausgeführt. Der Wert von `x` geht dabei von 1 bis 5. Ausführlichere Beschreibung im Quelltext.

Vor- und Nachbedingungen:

Das Dokument muß in `ISFormat.AS` hartverdrahtet sein.

2.6.2.3 Sonstiges

Die beiden Funktionen `setFollowingObjectsAsNonFormatted` und `setObjectAsNonFormatted` wurden entwickelt, aufgrund einer Spezifikationsänderung aber nie gebraucht. Ich habe sie nicht gelöscht, aber auch nicht in diese Dokumentation aufgenommen.

2.6.3 Lokale Funktionen

Funktionsname:

buildLevelItem buildLevelItem

Signatur:

buildLevelItem:: env->fs->objID->objinfolist->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Vorgängers

objinfolist: Liste der einzufügenden Objekte

fs [2]: Neue Formatierstruktur

Beschreibung:

Jedes der Objekte in `objinfolist` wird im Graphen an das Objekt `objID` gehängt. Dazu ruft die Funktion `buildID` für jedes Element der Liste auf und reicht den Graphen jeweils mit durch. Falls die Liste der Nachfolger leer ist, wird abgebrochen. Die Funktion hat den Sinn, daß der Graphaufbau für jedes Element der Nachfolgerliste aufgerufen wird.

Funktionsname:

buildID buildID

Signatur:

buildID:: env->fs->objID->objinfo->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Vorgängers

objinfo: Einzufügendes Objekt

fs [2]: Neue Formatierstruktur

Beschreibung:

Fügt das Element `objinfo` als Nachfolger von `objID` in den Baum ein.

Implementierung:

Setzt `elementtype` für `objinfo` und ruft `buildLevel` für `objinfo` auf (Rekursionsschritt).

Es wird beim Einsetzen zwischen finalen und nichtfinalen Elementen unterschieden und die jeweilige `insertNewXXXXObject` verwendet. Die Reihenfolge der eingesetzten Objekte ist noch dieselbe wie die aus der Internen Struktur.
Da `buildID` seine Daten aus der Internen Struktur bekommt und die physikalische Objekte im Sinne der Formatierer nicht kennt, können solche hier auch nicht auftauchen. Es wird daher in einem solchen Fall nichts gemacht.

Funktionsname:

filterObjId filterObjId

Signatur:

filterObjId:: objinfolist->objIDList

objinfolist: Liste der zu wandelnden Objekte

objidlist: Liste der Objekt-IDs

Beschreibung:

Filtert aus aus einer Liste von `objinfos` alle `objIDs` heraus.

Funktionsname:

filterElementType filterElementType

Signatur:

filterElementType:: objinfolist->elementtypelist

objinfolist: Liste der zu wandelnden Objekte

elementtypelist: Liste der Elemententypen

Beschreibung:

Filtert aus aus einer Liste von `objinfos` alle `elementtypes` heraus.

Funktionsname:

buildPartOfOrderHelp buildPartOfOrderHelp

Signatur:

buildPartOfOrderHelp:: env->fs->objinfolist->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objinfolist: Liste der prüfenden Objekte

fs [2]: Neue Formatierstruktur

Beschreibung:

Hilfsfunktion für `buildPartOfOrder`. Diese wendet `buildPartOfOrderHelp` rekursiv auf jedes Element aus `objinfolist` an. Die Variablen `env` und `fs` werden dabei mitgereicht und kommen verändert zurück.

Funktionsname:

buildOrderOfLevel buildOrderOfLevel

Signatur:

buildOrderOfLevel:: env->fs->objID->elementtype->objinfolist->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Vorgänger aller Objekte in der `objinfolist`

elementtype: Elementtyp des Vorgängers

objinfolist: Liste der prüfenden Objekte

fs [2]: Neue Formatierstruktur

Beschreibung:

Läßt für diese Ebene die Typen der Nachfolger ermitteln. Mit `buildOrderOfLevelHelp` werden die Nachfolger in der korrekten Reihenfolge ermittelt.
Diese werden dann via `setNonFinalSuccessors` gesetzt.

Funktionsname:

strippedObjList strippedObjList

Signatur:

strippedObjList:: objinfolist->objinfo->objinfolist

objinfolist [1]: Alte Liste der Objekt-Infos

objinfo: Zu löschendes Objekt-Info

objinfolist [2]: Neue Liste der Objekt-Infos

Beschreibung:

Entfernt das Objekt-Info `objinfo` aus der Liste.

Funktionsname:

buildOrderOfLevelHelp buildOrderOfLevelHelp

Signatur:

buildOrderOfLevelHelp:: objinfolist->objinfolist->elementtypelist->successorlist->(successorlist, objinfolist).

objinfolist [1]: Alte Liste der Objekt-Infos

objinfolist [2]: Temporäre Liste der Objekt-Infos

elementtypelist: Liste der Elementtypen

successorlist [1]: Alte Liste der Nachfolger

successorlist [2]: Neue Liste der Nachfolger

objinfolist [3]: Neue Liste der Objekt-Infos

Beschreibung:

Berechnet die korrekte Reihenfolge der Typen.

Implementierung:

Die Variable `elementtypelist` ist die Rekursionsvariable.
Die erste Variable speichert die Liste der noch nicht geordneten Objekte. Die Liste von Elementtypen wird Element für Element durchgearbeitet. Zu jedem Element wird das entsprechende Element in der `objinfolist [1]` gesucht, und dessen Wert `objID` in die Liste der Nachfolger aufgenommen. Danach wird die Liste `objinfolist[2]` wieder auf den Anfangswert der ersten Variablen Minus dem gefundenen und damit sortierten Element

gesetzt und der Prozeß für das nächste Element der `elementtyp`-Liste gestartet. Die erste Zeile der Funktion kann niemals eintreten, da jedes Element aus der Elementtype-Liste in der `objinfolist` sein muß.

Änderung (6.6.1994) :

Die Liste der erhaltenen Nachfolgeobjekte für die Reihenfolge kann weniger Elemente enthalten als die der tatsächlichen Nachfolger. In der Liste der tatsächlichen Nachfolger werden die Präsentationsregeln nicht berücksichtigt, während sie bei der Reihenfolge berücksichtigt werden. Daher muß der bereits aufgebaute Graph um die Elemente verringert werden, die zwar noch in der abzuarbeitenden `objinfolist` (2. Parameter) auftauchen, nicht aber in der Liste der Elementtypen.
Daher muß eine Liste mit den noch zu löschenden Elementen zurückgegeben werden. Das ist der letzte Parameter: `objinfolist`. Da nun nicht mehr alleine eine Liste zurückgegeben wird, wird das Ergebnis bis zum Erreichen des Rekursionsbodens in einem Parameter gespeichert und erst dann zurückgegeben.

Funktionsname:

buildPhysicalSingle buildPhysicalSingle

Signatur:

buildPhysicalSingle:: env->fs->objID->fs

env: Altes Environment

fs [1]: Alte Formatierstruktur

objID: Objekt-ID des Vorgängers

fs [2]: Neue Formatierstruktur

Beschreibung:

Die Funktion, die beim Bilden der physikalischen Elemente als Funktional von `buildAllPhysicalObjs` auf jedes Element des Graphen angewendet wird.
Falls das nichtfinale Element vom Typ `counter` ist, wird ein physikalisches Objekt eingefügt.

Funktionsname:

Kapitelnumerierung Kapitelnumerierung

Beschreibung:

Die Kapitelnumerierung setzt sich aus einer Vielzahl von Funktionen zusammen. Es macht hier wenig Sinn, jede Funktion nach Schema F aufzuführen, weil nur das Zusammenspiel der Funktionen interessant ist. Daher erfolgt hier der Versuch einer anderen Darstellung.
Es folgt zunächst eine Aufstellung aller beteiligten Funktionen und dann eine Übersicht.

Beteiligte Funktionen:

countElementTypeSingle::env->fs->objID->elementtype->integer

countElementType::env->fs->elementtype->objID->integer

getObjIDOfCountingElement::env->fs->objID->elementtype->(objID,env)

isObjNonFinal::fs->objID->boolean

getPreviousCountingObjIDs::env->fs->objID->elementtype->(objIDList,env)

getValueOfCounter::env->fs->objID->elementtype->elementtype->(env,integer)

buildCounterOfObject::env->fs->objID->(env,fs)

getObjIDOfPhysicalCounterInSuccList::fs->successorlist->objID

getObjIDOfCounterFatherType::env->fs->objID->elementtype->objID

getObjIDOfPhysicalCounterSingle::env->fs->objID->elementtype->integer

getObjIDOfPhysicalCounter::env->fs->elementtype->objID->integer

buildCompleteCounterSingle::env->fs->objID->(env,fs)

buildAllCounterSingle::env->fs->objID->(env,fs)

Der Zusammenhang der Funktionen ist leider nicht ganz trivial. Die Erklärung bezieht sich auf unser Beispiel 1, das sich im Quelltext befindet. Das Ermitteln besteht aus zwei Läufen: Im ersten wird die Nummer ermittelt, die nur für die Elemente direkt gilt. Der Wert ist für alle Kapitel korrekt, für alle Objekte ab *Unterkapitel* fehlt aber noch der erste Teil. Daher wird im zweiten Lauf der Text vom Vorgänger der Numerierung gebildet und davorgehängt. Im Beispiel betrifft das das Unterkapitel 1.1, der Vater ist Kapitel 1.

Die Funktionen: Die Funktion `buildAllCounter` ruft die beiden Läufe auf, indem ein Funktional mittels `traverseGraphEnv` auf jedes Element des Baums angewendet wird.

1. `buildAllCounterSingle`
2. `buildCompleteCounterSingle`

In jedem Lauf wird die jeweilige Funktion im Graphen von oben nach unten auf jedes Element angewendet. Dabei wird im ersten Lauf `buildAllCounterSingle` folgendes gemacht:

- Ermitteln der `objID` des zugehörigen physikalischen Objekts
- `buildCounterOfObject` des physikalischen Objekts
  - Berechne nichtfinalen Vorgänger
  - `getValueOfCounter` für den Vorgänger
    - ▷ Mit der Funktion werden alle links liegenden Objekte ermitteln
      Bsp.: links von 4 sind 2 und 3, links von 10 sind 16 3 2).
    - ▷ Nun werden in diesen Objekten die Objekte gezählt, die den gleichen Elemententyp wie das zu numerierende Objekt haben:
      Das macht die Funktion `countElementType`.
      Dazu wird wieder eine Funktion benutzt, die in einem Teilgraphen für jedes Element eine Funktion ausführt.
    - ▷ Bildung der Summe dieser gefundenen Elemente
  - Speichern des gewandelten Werts in physikalischer Stringtabelle.

Und für den zweiten Lauf `buildCompleteCounterSingle`:

- Das aktuelle nichtfinale Element ist vom BoxenTyp `counter`:
  - Ermitteln der Objekt-ID des Vaters der Numerierung:
    `getObjIDOfCounterFatherType`
  - Ermitteln der Objekt-ID des physikalischen Objekts, das die Numerierung des Vaters enthält:
    `getObjIDOfPhysicalCounter`
  - Lesen des Strings des Vorgängers.
  - Abspeichern des neuen Strings für das physikalische Objekt des aktuellen Elements.
- Das aktuelle nichtfinale Element ist vom BoxenTyp `counter`: Fertig

2.7 FormatSFInfo

2.7.1 Funktionalität

Dieses Modul ruft gleichnamige Funktionen der Internen Struktur aus dem Modul `ISFormat` auf. Es dient dazu, daß die einfachen Formatierer nicht direkt auf die Interne Struktur zugreifen, sondern über dieses Schnittstellenmodul. Die Funktionen liefern alle Werte, die die Präsentationsregeln betreffen. und den Text von Objekten (also den finalen und den phzsikalischen Objekten). Eine Erklärung der Inhalte der einzelnen Parameter befindet sich auf Seite 286.
Außerdem enthält das Modul die Funktion zum Ermitteln der finalen Elemente.

2.7.2 Entwurf

Anfänglich hielten wir es für praktisch, daß wir alle Anfragen der einfachen Formatierer über eine zentrale Stelle laufen ließen.
Im nachhinein war dies eine sehr glückliche Entscheidung, da wir damit zwei Probleme relativ leicht lösen konnten:

1. In der Internen Struktur gibt es keine physikalischen Objekte im Formatierersinn (z.B. Kapitelnummern). Die einfachen Formatierer kennen diese Objekte ebenfalls nicht, für sie erscheinen sie wie normale finale Objekte. Daher landen alle Anfragen der einfachen Formatierer die physikalischen Objekte betreffend auch in diesem Modul. Für solche physikalischen Objekte ist eine Abfrage eingebaut. Die Informationen wurden vorher vom `HLF` erzeugt und werden dann im Falle eines Aufrufs mit einem physikalischen Elementes direkt aus der Formatierstruktur statt aus `ISFormat` gelesen.

2. Die einfachen Formatierer haben sich zu Beginn der Implementation selbst die Datentypen `pagelayout` und `paragraphlayout` geschaffen. Diese sind aber von den entsprechenden Datentypen der `IS` verschieden, in diesem Modul ist daher an zentraler Stelle ein Konverter eingebaut.

2.7.3 Öffentliche Schnittstellen

2.7.3.1 Sorten

Sortenname:

pagelayout pagelayout

Signatur:

pagelayout:: pagelayout
(evensidemargin:: integer)
(oddsidemargin :: integer)
(footheight :: integer)
(topsep :: integer)
(headheight :: integer)
(headsep :: integer)
(topmargin :: integer)
(textheight :: integer)
(textwidth :: integer)

Beschreibung:

Datentyp der einfachen Formatierer für die Seitenlayout-Attribute (s. Entwurf).

Sortenname:

paragraphlayout paragraphlayout

Signatur:

paragraphlayout:: paragraphlayout
(baselinediff :: integer)
(hspace :: integer)
(lineskiplimit :: integer)
(leftindent :: integer)
(rightindent :: integer)
(parfillskip :: integer)
(parindent :: integer)
(postparsep :: integer)
(preparsep :: integer)
(alignment :: attrAln)

Beschreibung:

Datentyp der einfachen Formatierer für die Absatzlayout-Attribute (s. Entwurf).

2.7.3.2 Funktionen

Funktionsname:

getFinalElements getFinalElements

Signatur:

getFinalElements:: fs->integers

fs: Formatierstruktur

integers: Liste der Objekt-IDs aller finalen Elemente

Beschreibung:

Die Funktion ermittelt alle finalen Elemente des Dokuments im Graphen. Hierunter fallen sowohl finale, als auch physikalische Objekte. Ihre Objekt-IDs werden alle in einer Liste zurückgegeben.

Vor- und Nachbedingungen:

Die Funktion geht davon aus, daß das erste Element des Graphen nichtfinal ist und die `ObjId=1` hat.

Implementierung:

Dazu werden zunächst die Nachfolger des ersten Objekts ermittelt und dann die Hilfsfunktion `getFinalElementsHelp` aufgerufen, die das eigentliche Suchen übernimmt.

Funktionsname:

getObjData getObjData

Signatur:

getObjData:: env->objID->string

env: Environment

objID: Objekt-ID des Objekts, dessen Text gelesen werden soll

string: Text des Objekts

Beschreibung:

Ermittelt den Text des finalen oder physikalischen Objekts mit der ObjektID `objID`. Falls es sich um ein physikalisches Objekt handelt, werden die die errechneten Werte genommen.

Implementierung:

Falls es sich um ein physikalisches Objekt handelt, wird der Text aus der Formatierstruktur gelesen, ansonsten werden die Werte aus der gleichnamigen Funktion des Moduls `ISFormat` durchgereicht.

Funktionsname:

getPageLayout getPageLayout

Signatur:

getPageLayout:: env->objID->pagelayout

env: Environment

objID: Objekt-ID des Objekts, dessen Seitenlayout gelesen werden soll

pagelayout: Seitenlayout des Objekts

Beschreibung:

Ermittelt das Seitenlayout des finalen oder physikalischen Objekts mit der ObjektID `objID`. Falls es sich um ein physikalisches Objekt handelt, werden die Werte des nichtfinalen Vorgängers genommen.

Implementierung:

Falls es sich um ein physikalisches Objekt handelt, werden die Werte vom Vorgänger gelesen, ansonsten werden die Werte aus der gleichnamigen Funktion des Moduls `ISFormat` durchgereicht und mittels `convertPageLayout` in das Format der einfachen Formatierer konvertiert (s. Entwurf).

Funktionsname:

getParagraphLayout getParagraphLayout

Signatur:

getParagraphLayout:: env->objID->paragraphlayout

env: Environment

objID: Objekt-ID des Objekts, dessen Absatzlayout gelesen werden soll

paragraphlayout: Absatzlayout des Objekts

Beschreibung:

Ermittelt die Parameter für die Absätze des finalen oder physikalischen Objekts mit der ObjektID `objID`. Falls es sich um ein physikalisches Objekt handelt, werden die Werte des nichtfinalen Vorgängers genommen.

Implementierung:

Falls es sich um ein physikalisches Objekt handelt, werden die Werte vom Vorgänger gelesen, ansonsten werden die Werte aus der gleichnamigen Funktion des Moduls `ISFormat` durchgereicht und mittels `convertParagraphLayout` in das Format der einfachen Formatierer konvertiert (s. Entwurf).

Funktionsname:

getFontInfo getFontInfo

Signatur:

getFontInfo:: env->objID->font

env: Environment

objID: Objekt-ID des Objekts, dessen Fontinfo gelesen werden soll

font: Fontinfo des Objekts

Beschreibung:

Ermittelt die Parameter für den Zeichensatz des finalen oder physikalischen Objekts mit der ObjektID `objID`. Falls es sich um ein physikalisches Objekt handelt, werden die Werte des nichtfinalen Vorgängers genommen.

Implementierung:

Falls es sich um ein physikalisches Objekt handelt, werden die Werte vom Vorgänger gelesen, ansonsten werden die Werte aus der gleichnamigen Funktion des Moduls `ISFormat` durchgereicht.

Funktionsname:

getDocumentAttributes getDocumentAttributes

Signatur:

getDocumentAttributes:: env->objID->docu

env: Environment

objID: Objekt-ID des Objekts, dessen Dokumentenattribute gelesen werden sollen

docu: Dokumentenattribute des Objekts

Beschreibung:

Ermittelt die Dokumentenattribute des finalen oder physikalischen Objekts mit der ObjektID `objID`. Falls es sich um ein physikalisches Objekt handelt, werden die Werte des nichtfinalen Vorgängers genommen.

Implementierung:

Falls es sich um ein physikalisches Objekt handelt, werden die Werte vom Vorgänger gelesen, ansonsten werden die Werte aus der gleichnamigen Funktion des Moduls `ISFormat` durchgereicht.

Funktionsname:

getFirstObj getFirstObj

Signatur:

getFirstObj:: env->objinfo

env: Environment

objinfo: Objektinformation des ersten Objekts

Beschreibung:

Ermittelt die Objektinformationen des ersten Elements des Dokuments.

Implementierung:

Der Wert wird aus der gleichnamigen Funktion des Moduls `ISFormat` durchgereicht und mittels `convertParagraphLayout` in das Format der einfachen Formatierer konvertiert (s. Entwurf).

2.7.4 Lokale Implementation

2.7.4.1 Funktionen

Funktionsname:

getFinalElementsHelp getFinalElementsHelp

Signatur:

getFinalElementsHelp:: fs->successorlist->integers

fs: Formatierstruktur

successorlist: Liste der Nachfolger, die betrachtet werden sollen

integers: Liste der Objekt-IDs aller finalen Elemente

Beschreibung:

Hilfsfunktion für `getFinalElements`, die die eigentliche Rekursion durchführt.

Implementierung:

Die Funktion sucht die finalen Elemente; dazu wird das erste Element der `successorlist` untersucht. Ist es finalen Typs, wird die `ObjId` des Elements in die Ergebnisliste aufgenommen und der Rest betrachtet. Ist das Element nichtfinalen Typs, so werden zunächst rekursiv alle Nachfolger untersucht; danach wird mit dem Rest der Liste weitergemacht. Durch ein solches Vorgehen wird die Reihenfolge des Graphen gewahrt.

Funktionsname:

attrToInt attrToInt

Signatur:

attrToInt:: attrNum->integer

attrNum: Zu konvertierendes Attribut

integer: Ergebnis als ganze Zahl

Beschreibung:

Wandelt den Integer-Wert in `attrNum` in eine ganze Zahl um.

Funktionsname:

convertPageLayout convertPageLayout

Signatur:

convertPageLayout:: page-> pagelayout

page: Zu konvertierender Datentyp aus `IS`

pagelayout: Datentyp für `EF`

Beschreibung:

Wandelt die Seitenlayoutparameter vom Typ `page` aus der Internen Struktur in den Typen `pagelayout` für die Formatierstruktur um.

Funktionsname:

convertParagraphLayout convertParagraphLayout

Signatur:

convertParagraphLayout:: paragraph-> paragraphlayout

paragraph: Zu konvertierender Datentyp aus `IS`

paragraphlayout: Datentyp für `EF`

Beschreibung:

Wandelt die Absatzparameter vom Typ `paragraph` aus der Internen Struktur nach `paragraphlayout` (Formatierstruktur) um.

2.8 FormatUI

2.8.1 Funktionalität

Das Modul ist die Schnittstelle zwischen der Benutzungsoberfläche und den Formatierern. Das gilt sowohl für das inkrementelle als auch für das initiale Formatieren.

2.8.2 Entwurf

Dieses Modul stößt die einfachen Formatierer nur bei einer initialen Formatierung an. Bei einer inkrementellen Formatierung stößt die Ausgabe die einfachen Formatierer über ein anderes Modul (`FormatOutput`)an. In einem solchen Fall müssen die einfachen Formatierer aber wissen, was sich geändert hat. Dazu wird ein Datentyp namens `fschanges` verwendet, der die Art der Änderungen protokolliert.
In diesem Modul finden nur kleinere Berechnungen statt, in den meisten Fällen werden Funktionen aus anderen Modulen des `HLF` aufgerufen, die die eigentliche Arbeit machen.

2.8.3 Öffentliche Schnittstellen

2.8.3.1 Funktionen

Funktionsname:

f_initF f_initF

Signatur:

f_initF:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Nimmt einen initialen Lauf der Formatierung vor. Der `HLF` führt dazu seine Läufe durch (Graphenaufbau, Reihenfolge, Kapitelnumerierung), außerdem wird das komplette Dokument von den einfachen Formatierern formatiert.

Vor- und Nachbedingungen:

Dazu muß das Dokument mit allen Informationen geladen und in der Internen Struktur abgelegt sein.

Implementierung:

Es wird die Funktion `initFormatter` aus dem Modul `FormatFirstPass` aufgerufen. Sie macht folgendes:

1. Erzeugen einer leeren Formatierstruktur
2. Aufbauen des Graphen
3. Herstellen der Reihenfolge
4. Herstellen der Kapitelnumerierung

5. Speichern der Formatierstruktur in der `fs`
6. Initiales Formatieren des Dokuments.

Funktionsname:

f_DeleteChars f_DeleteChars

Signatur:

f_DeleteChars:: env->objID->integer->env

env [1]: Altes Environment

objID: Objekt-ID des Objektes, in dem Buchstaben gelöscht wurden

integer: Nummer des ersten zu löschenden Buchstabens

env [2]: Neues Environment

Beschreibung:

Protokolliert in `fschanges`, daß im Objekt `objID` ab dem Buchstaben `integer` ein oder mehrere Buchstaben gelöscht worden sind. Die einfachen Formatierer können später anhand dieser Änderung eine inkrementelle Formatierung durchführen.

Implementierung:

Die Werte werden in das `fschanges`–Konstrukt eingetragen.

Funktionsname:

f_InsertChars f_InsertChars

Signatur:

f_InsertChars:: env->objID->integer->string->env

env [1]: Altes Environment

objID: Objekt-ID des Objektes, in dem Buchstaben eingefügt wurden

integer: Nummer des ersten einzufügenden Buchstabens

string: Einzufügender Text

env [2]: Neues Environment

Beschreibung:

Protokolliert in `fschanges`, daß im Objekt `objID` ab dem Buchstaben `integer` der Text `string` eingefügt worden ist. Die einfachen Formatierer können später anhand dieser Änderung eine inkrementelle Formatierung durchführen.

Implementierung:

Die Werte werden in das `fschanges`–Konstrukt eingetragen.

Funktionsname:

f_DeleteElement f_DeleteElement

Signatur:

f_DeleteElement:: env->objID->objID->env

env [1]: Altes Environment

objID [1]: Objekt-ID des gelöschten Objektes

objID [2]: Objekt-ID des Vorgängers des gelöschten Objektes

env [2]: Neues Environment

Beschreibung:

Protokolliert in `fschanges`, daß das Objekt `objID` gelöscht worden ist. Die einfachen Formatierer können später anhand dieser Änderung eine inkrementelle Formatierung durchführen. Außerdem werden ab dem Vorgängerobjekt die Läufe des `HLF` (s. `f_initF`) erneut durchgeführt.

Implementierung:

Löscht ein logisches Element `objId` (2.Parameter) am Knoten `objID` inklusive aller daran befindlichen Elemente, falls es sich um ein nichtfinales Objekt handeln sollte. Handelt es sich um ein finales Objekt, so wird dieses manuell gelöscht. Anschließend wird ab dem Vorgängerobjekt die Reihenfolge der Objekte neu geprüft und die Numerierung für die Kapitel lokal erneuert.

Außerdem werden die Werte in das `fschanges`–Konstrukt eingetragen.

Funktionsname:

f_InsertElement f_InsertElement

Signatur:

f_InsertElement:: env->objID->objID->env

env [1]: Altes Environment

objID [1]: Objekt-ID des einzufügenden Objektes

objID [2]: Objekt-ID des Vorgängers des einzufügenden Objektes

env [2]: Neues Environment

Beschreibung:

Protokolliert in `fschanges`, daß das Objekt `objID` eingefügt worden ist. Die einfachen Formatierer können später anhand dieser Änderung eine inkrementelle Formatierung durchführen.
Außerdem werden ab dem Vorgängerobjekt die Läufe des `HLF` (s. `f_initF`) erneut durchgeführt.

Vor- und Nachbedingungen:

Annahmen:

- Das Einfügen des Objektes ist zulässig.
- Das Objekt ist in der Internen Struktur bereits eingefügt.
- Es handelt sich nicht um die Wurzel des Baumes (`type=document`)

Implementierung:

Fügt ein logisches Element `objID` (2. Parameter) am Knoten `objId` (3. Parameter) ein. Dazu wird der Baum ab dem Vorgängerobjekt neu aufgebaut. Anschließend wird ab dem Vorgänger die Reihenfolge der Objekte neu geprüft und die Numerierung für die Kapitel lokal erneuert.
Außerdem werden die Werte in das `fschanges`–Konstrukt eingetragen.

Funktionsname:

f_getFirstFinalObj f_getFirstFinalObj

Signatur:

f_getFirstFinalObj:: env->(env, objID)

env [1]: Altes Environment

env [2]: Neues Environment

objID: Objekt-ID des ersten finalen Elements

Beschreibung:

Die Funktion liefert die Objekt-ID des ersten finalen Elements des Dokuments oder `mtObjID`, falls kein solches vorhanden ist.

Implementierung:

Es wird mittels der Funktion `getFinalElements` aus dem Modul `FormatSFInfo` die Liste aller finalen Elemente bestimmt und hiervon der Kopf geliefert.

Funktionsname:

f_getLastPageNumber f_getLastPageNumber

Signatur:

f_getLastPageNumber:: env->(env, integer)

env [1]: Altes Environment

env [2]: Neues Environment

integer: Nummer der letzten Seite

Beschreibung:

Liefert die Seitennummer des letzten finalen Elements. Falls es ein solches nicht geben sollte, wird `0` geliefert.

Implementierung:

Es wird mittels der Funktion `getFinalElements` aus dem Modul `FormatSFInfo` die Liste aller finalen Elemente bestimmt und vom letzten Element die Seitennummer aus der letzten `box` ausgelesen.

Funktionsname:

f_getObjSize f_getObjSize

Signatur:

f_getObjSize:: env -> objID -> pagenumber -> boxList

env: Umgebungsvariable

objID: Objekt-ID

pagenumber: aktuelle Seitennummer

boxList: Liste der zu markierenden Boxen

Beschreibung:

Diese Funktion liefert zu einer eingegebenen Objekt-ID (final oder nicht final) das Konstrukt `boxlist`, welches die Ausdehnung des Objekts im Dokument (damit also auch auf dem Bildschirm) beschreibt.

Implementierung:

Zunächst wird zwischen finalen und nicht finalen Elementen unterschieden. In der Hilfsfunktion `getObjCoordinates` werden die entsprechenden Koordinaten ausgelesen.

2.8.4 Lokale Implementation

Funktionsname:

getObjCoordinates getObjCoordinates

Signatur:

getObjCoordinates:: env -> fs -> objIDList -> pagenumber -> boxList

env: Umgebungsvariable

fs: Formatiererstruktur

objIDList: Liste von Objekten

pagenumber: aktuelle Seitennummer

boxList: Liste von Boxen

Beschreibung:

Die Funktion liefert als Ergebnis eine Liste von Boxen zurück, die die Ausmaße der eingegebenen Objekte repräsentieren.

Implementierung:

Die Liste der eingegebenen Objekte wird rekursiv abgearbeitet. Handelt es sich beim jeweiligen Objekt um ein finales Element, werden die Koordinaten zusammengestellt (siehe Dreiboxenkonzept) und an die Rückgabeliste konkateniert. Bei einem nicht finalen Element wird die aufrufende Funktion `f_getObjSize` erneut aufgerufen, um die Ausmaße der darunterliegenden finalen Elemente zu bestimmen.

Funktionsname:

stripCoordList stripCoordList

Signatur:

stripCoordList:: boxList -> boxList

boxList: Liste von Boxen

boxList: bearbeitete Liste von Boxen

Beschreibung:

Die Funktion entfernt vorhandene neutrale Elemente `mtBox`, die lediglich als Abbruchbedingungen für die Funktion `getObjCoordinates` dienen aus der Liste.

Implementierung:

Die eingegebenene Liste wird rekursiv auf `mtBox` untersucht. Wird eine `mtBox` gefunden, wird dieses Element nicht an die Ergebnisliste angehängt.

Funktionsname:

getObjSize getObjSize

Signatur:

getObjSize:: boxList -> boxList

boxList: Liste von Boxen

boxList: bearbeitete Liste von Boxen

Beschreibung:

Die Funktion liefert ein `mtBox`-Element, wenn keine Listenelemente vorhanden sind. Andernfalls wird die Liste zurückgegeben.

2.9 FormatTest

2.9.1 Funktionalität

Das Modul stellt einige Funktionen für das Testen der Formatierstruktur bereit. Es ist nur für die Gruppe der Formatierer interessant, da das Hauptfenster der `UI` in diesen Tests nicht aufgerufen werden darf.
Die Tests gehen von einem festverdrahteten Modul `ISFormat` aus.

2.9.2 Öffentliche Schnittstellen

2.9.2.1 Funktionen

Funktionsname:

testState testState

Signatur:

testState:: fs->integers->integers

fs: Formatierstruktur

integers: Liste der Objekt-IDs der zu prüfenden Objekte

integers: Liste der Objekt-IDs der zu prüfenden Objekte

Beschreibung:

Testet den Status einer Liste von Objekten. Es wird eine Liste von Zahlen geliefert.[9]

0 final

1 physikalisch

2 nichtfinal

2 gefolgt von 500 Objekt nicht vorhanden

Vor- und Nachbedingungen:

Der Test geht von einem festverdrahteten Modul `ISFormat` aus.

[9] Dies ist ein „Hackertest“

Funktionsname:

printObjects printObjects

Signatur:

printObjects:: system->fs->integers->system

system [1]: Altes System

fs: Formatierstruktur

integers: Liste der Objekt-IDs des auszugebenden Objektes

system [2]: Neues System

Beschreibung:

Gibt alle `FS`-Informationen für die Objekte in einem Stream aus, die in der `objidlist` (3. Parameter) aufgeführt sind.

Implementierung:

Packt die Werte mittels der Hilfsfunktionen `stringObject` und `stringsToSystem` in einen Stream.

Funktionsname:

loadDocumentBernd loadDocumentBernd

Signatur:

loadDocumentBernd:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Lädt das Memo-Dokument, die Pfade sind auf `home|bernd|unifinal` eingestellt.

Funktionsname:

loadDocumentUni loadDocumentUni

Signatur:

loadDocumentUni:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Lädt das Memo-Dokument, die Pfade sind auf `foraus|final|SGMLtest` eingestellt.

2.9.3 Lokale Implementation

2.9.3.1 Funktionen

Funktionsname:

stringObject stringObject

Signatur:

stringObject:: fs->objID->string

fs: Formatierstruktur

objID: Objekt-ID des auszugebenden Objektes

string: Alle Daten des auszugebenden Objektes als Text

Beschreibung:

Gibt allen Daten eines Objekts in Abhängigkeit seines Typs in einen `string` aus.

Funktionsname:

stringsToSystem stringsToSystem

Signatur:

stringsToSystem:: system->strings->system

system [1]: Altes System

strings: Liste der auszugebenden Texte

system [2]: Neues System

Beschreibung:

Schreibt eine Liste von Strings in die Variable `system`.

2.10 HLF - Extern

Jan Hiller, Andreas Hinken

2.11 FormatOutput

2.11.1 Funktionalität

Im Modul `FormatOutput` werden Funktionen des „High-Level-Formatters“ (HLF), der das Bindeglied zwischen dem „Simple Formatter“ und der Bildschirmausgabe darstellt, zur Verfügung gestellt. Der HLF initiiert ggf. Formatierungsvorgänge und trägt die Verantwortung für die Aktualisierung der Daten in der Formatiererstruktur. Außerdem befinden sich einige Hilfsfunktionen für die Benutzungsoberfläche und die Postscript-Ausgabe in diesem Modul.

2.11.2 Entwurf

Das in die Konstruktion der Funktionen miteingeflossene „Boxenkonzept“ (siehe Kapitel 2.13.3.1) und die verwendeten inhaltsbeladenen Konstruktoren, wie z.B. `changes` und `wordAttributeList` werden im Typen-Modul `FormatOutputTypes.AS` eingehend erläutert.

2.11.3 Öffentliche Funktionen

2.11.3.1 Funktionen

Funktionsname:

getPageCoordinates getPageCoordinates

Signatur:

getPageCoordinates:: fs -> objID -> (pagecoordinates, pagecoordinates, pagecoordinates, pagecoordinates)

fs: Formatiererstruktur

objID: Objektidentifikationsnummer

pagecoordinates: Koordinaten der linken obersten Ecke

pagecoordinates: Koordinaten der linken oberen Ecke

pagecoordinates: Koordinaten der linken unteren Ecke

pagecoordinates: Koordinaten der linken untersten Ecke

Beschreibung:

Die Funktion liefert die vier Koordinaten (siehe Boxenkonzept in Kapitel 2.13.3.1) eines eingegebenen finalen oder nicht finalen Elements in Form von `pagecoordinates`.

Implementierung:

Zunächst wird der Status des eingegebenen Objekts ausgelesen. Es wird zwischen finalen und nicht finalen Elementen unterschieden. Handelt es sich um ein nicht finales Element, werden mithilfe der Funktionen `getNonFinal...Box` die charakteristischen Eckpunkte des Objekts ausgelesen und zurückgegeben. Bei einem finalen Element, welches aus mehreren Boxen bestehen kann, werden die oberen Koordinaten des Bereichs aus der ersten Box des Objekts und die unteren aus der letzten Box ausgelesen. Diese Werte sind als `coordinates` anders typisiert und müssen folglich in den `pagecoordinates`-Typ umgewandelt werden.

Funktionsname:

f_getPageSize f_getPageSize

Signatur:

f_getPageSize:: env -> pagenumber -> (env, pagesize)

env: Umgebungsvariable

pagenumber: aktuelle Seitennummer

env: neue Umgebungsvariable

pagesize: Ausmaße der Seite

Beschreibung:

Die Funktion ermittelt die Seitengröße der angegebenen Seite.

Vor- und Nachbedingungen:

Das Dokument muß vor dem Aufruf dieser Funktion bis einschließlich der betreffenden Seite vollständig formatiert worden sein.

Implementierung:

Die Funktion bestimmt zunächst mithilfe der Funktion `getObjIDFromPage` die Objekte auf der angegebenen Seite. Da alle Objekte einer Seite identische Seitenlayoutinformationen enthalten, werden diese Werte aus dem ersten Objekt ausgelesen. Die einzelnen Parameter der X- bzw. Y-Ausdehnung werden addiert und als `pagesize`-Typ zurückliefert.

Funktionsname:

f_getChanges f_getChanges

Signatur:

f_getChanges:: env -> pagecoordinates -> (env, changes)

env: Umgebungsvariable

pagecoordinates: aktuelle Cursorposition env: neue Umgebungsvariable

changes: zurückgelieferte Veränderungen

Beschreibung:

Die Funktion ermittelt nach einer ausgeführten Editierfunktion die Veränderungen der Formatierung im Dokument an der aktuellen Cursorposition. Herausgegeben wird ein Konstrukt `changes`, daß aus einem zu löschenden Bereich `clear`, einem einzufügenden Wortstrom mit Attributen `insert` und einem zu verschiebenden Bereich `move` besteht.

Vor- und Nachbedingungen:

Wenn ein Objekt gelöscht oder eingefügt worden ist, beinhaltet das `v_changeObjID` dasjenige Objekt, welches *vor* dem zu löschenden bzw. einzufügenden Objekt steht.

Implementierung:

Nach dem Auslesen der Formatiererstruktur und der Art der Veränderung (*Buchstabe* oder *Objekt*) wird die Objektidentifikationsnummer des bearbeiteten Objekts ermittelt und ggf. die eingefügten Zeichen festgestellt; wurde ein Zeichen gelöscht, sind keine Zeichen vorhanden.

Bei Operationen, die *Zeichen* betreffen, wird die Zeile der Box bestimmt, in der die Aktion stattgefunden hat. Anschließend wird mithilfe der Funktion `reformatLine` eine Neuformatierung ab dieser Zeile durchgeführt und die Veränderungen in der Formatierung bestimmt (siehe `changes`-Konstrukt in Kapitel 2.13.3.1).

Wurde ein *Objekt* geändert, wird mithilfe der Funktion `reformatObjekt` ab diesem Objekt eine Neuformatierung veranlaßt. Ebenso wie bei der zeichenweisen Veränderung wird auch hier aus den zurückgelieferten Parametern das `changes`-Konstrukt gebildet. In beiden Fällen folgt die Synchronisierung der Formatiererstruktur mit der Umgebungsvariable `env`.

Funktionsname:

f_getEntirePage f_getEntirePage

Signatur:

f_getEntirePage:: env -> pagenumber -> (env, wordattributelist)

env: Umgebungsvariable

pagenumber: aktuelle Seitennummer

env: neue Umgebungsvariable

wordattributelist: Liste von Wörtern bzw. Zeilen mit Koordinaten

Beschreibung:

Diese Funktion stellt eine Liste von Wörtern bzw. Zeilen mit Koordinaten und Schriftattributen zusammen, die dem Textinhalt des Dokumentes an der angegebenen Seite entspricht.

Vor- und Nachbedingungen:

Es wird davon ausgegangen, daß auf der darzustellenden Seite mindestens ein finales Objekt vorhanden ist.

Implementierung:

Zuerst wird aus der Formatiererstruktur das erste Element (die Wurzel) extrahiert und damit die Funktion `checkFormatStatus` angesteuert, die ggf. eine Formatierung dieses Bereiches vornimmt. Als nächstes wird aus der formatierten Struktur eine Liste von Objekten gebildet, die sich auf der Seite befinden. Die Elemente dieser Liste werden von der Funktion `setAttributes` rekursiv in die `wordAttributeList` eingelesen.

Funktionsname:

initNFStructure initNFStructure

Signatur:

initNFStructure:: env -> env

env: Umgebungsvariable

env: neue Umgebungsvariable

Beschreibung:

Die Funktion setzt Ausmaße der nicht finalen Elemente und deren Formatierstati in der gesamten Formatiererstruktur. Die Funktion wird in der Regel zur Initialisierung der Struktur verwendet.

Implementierung:

Die Formatiererstruktur wird aus der Umgebungsvariable extrahiert. Danach werden alle Formatierstati beginnend mit der Wurzel des Dokuments, mithilfe der Funktion `setFormattedFlags` gesetzt. Existieren finale Elemente, werden die Ausmaße dieser Objekte richtig gesetzt. Die neuen Werte werden abschließend in die Umgebungsvariable zurückgeschrieben.

2.11.4 Lokale Implementation

2.11.4.1 Funktionen

Funktionsname:

checkFormatStatus checkFormatStatus

Signatur:

checkFormatStatus:: env -> fs -> objIDList -> pagenumber -> fs

env: Umgebungsvariable

fs: Formatiererstruktur

objIDList: Liste von Objekten

pagenumber: aktuelle Seitennummer

fs: neue Formatiererstruktur

Beschreibung:

Die Funktion überprüft den Formatierstatus der eingegebenen und der im Baum der Formatierstruktur nachfolgenden Objekte. Unformatierte Objekte (Teile des Baumes) werden ggf. bis zur eingegebenen Seitennummer formatiert.

Implementierung:

Es werden rekursiv alle Objekte in der Liste abgearbeitet. Hierbei wird zwischen finalen bzw. physikalischen und nicht finalen Elementen unterschieden. Mithilfe der Funktionen `checkFFormatStatus` und `checkNFFormatStatus` werden die Objekte weitergehend analysiert und bearbeitet.

Funktionsname:

checkFFormatStatus checkFFormatStatus

Signatur:

checkFFormatStatus:: env-> fs -> objIDList -> boxinfolist -> pagenumber -> fs

env: Umgebungsvariable

fs: Formatiererstruktur

objIDList: Liste von Objekten

boxinfolist: Liste von Boxen eines Objekts

pagenumber: aktuelle Seitennummer

fs: neue Formatiererstruktur

Beschreibung:

Diese Funktion überprüft den Formatierstatus eines finalen Elements und seiner Boxen. Ist das finale Element nicht formatiert, so wird eine Formatierung ab diesem Element bis zur angegebenen Seitennummer ausgeführt.

Implementierung:

Bei der Untersuchung der Subboxen eines Elements können folgende Fälle auftreten:

- die Subbox ist formatiert und ihre Seitennummer ist größer der zu überprüfenden: dann ist die Struktur bis mindestens zu der darzustellenden Seiten formatiert. Die Formatiererstruktur wird unverändert zurückgegeben.

- die Subbox ist formatiert und ihre Seitennummer ist gleich der zu überprüfenden: in diesem Falle ist das betreffende Objekt ist zwar formatiert, da aber nicht auszuschließen ist, daß nicht noch andere (evtl. unformatierte!!) Objekte existieren, die auf dieser Seite liegen könnten, wird `checkFormatStatus` mit dem verbliebenen Rest der Objektliste aufgerufen.
- die Subbox ist formatiert, ihre Seitennummer ist jedoch kleiner als die zu überprüfende: dann wird die Funktion wird rekursiv mit dem nächsten Element der Boxliste aufgerufen.
- die Subbox ist nicht formatiert, es kann also keine Aussage darüber gemacht werden, ob die Seitennummer korrekt ist. Aus diesem Grund wird ab der aktuellen Subbox neu formatiert.

Entsprechend dieser Unterscheidung, werden die betreffenden Hilfsfunktionen ausgeführt.

Funktionsname:

checkNFFormatStatus checkNFFormatStatus

Signatur:

checkNFFormatStatus:: env -> fs -> objIDList -> pagenumber -> fs

env: Umgebungsvariable

fs: Formatiererstruktur

objIDList: Liste von Objekten

pagenumber: aktuelle Seitennummer

fs: neue Formatiererstruktur

Beschreibung:

Diese Funktion überprüft den Formatierstatus eines nicht finalen Elements und seiner Nachfolger.

Implementierung:

Bei der Untersuchung der nicht finalen Elemente können folgende Fälle auftreten:

- Das Element ist nicht formatiert: da es unbekannt ist, ab welchem Objekt die darunterliegende Formatiererstruktur nicht mehr formatiert ist, wird die Verteiler-Funktion `checkFormatStatus` mit den im Baum der Formatiererstruktur nachfolgenden Elementen zur weiteren Untersuchung aufgerufen.
- Das Element ist formatiert und die aktuelle Seitennummer ist kleiner als die des zu untersuchenden Objektes: in diesem Falle wird die Formatiererstruktur unverändert zurückgegeben, da der darzustellende Bereich folglich bereits formatiert ist.

- Das Element ist formatiert und die aktuelle Seitennummer ist gleich der des zu untersuchenden Objektes: dann wird die `checkFormatStatus`-Funktion noch einmal mit dem Rest der Objektliste aufgerufen. Hierbei wird dann festgestellt, ob die darzustellende Seite evtl. noch weitere Objekte besitzt, die im Baum der Formatiererstruktur nicht unter diesem nicht finalen Element „hängen“.
- Das Element ist formatiert und die aktuelle Seitennummer ist größer als die des zu untersuchenden Objektes: als Folge wird die `checkFormatStatus`-Funktion noch einmal mit dem Rest der Objektliste aufgerufen.

Funktionsname:

setFormattedFlags setFormattedFlags

Signatur:

setFormattedFlags:: fs -> objID -> fs

fs: Formatiererstruktur

objID: Objektidentifikationsnummer

fs: neue Formatiererstruktur

Beschreibung:

Die Funktion setzt ausgehend von einem nicht finalen Element den Formatierstatus dieses und allen im Baum der Formatiererstruktur darunterliegenden Elementen.

Abhängigkeiten:

Das eingegebene Objekt muß ein nicht finales Element sein.

Implementierung:

Zunächst werden die nachfolgenden Objekte des aktuellen Elements ermittelt. Mithilfe der Funktion `foldr` wird auf jeweils ein Element der Liste der Nachfolger die Funktion `ourand` angewendet, dabei wird der schon ermittelte Formatierstatus und die neue Formatiererstruktur zum nächsten Element weitergereicht. Die Funktion `ourand` kann ihrerseits zur Bestimmung des Formatierstatus eines unter dem ursprünglichen Objekt liegenden nicht finalen Elements diese Funktion aufrufen, so daß es zu einer verschränkten Rekursion kommt. Die neu gesetzte Formatierstatus wird in die Formatiererstruktur zurückgeschrieben.

Funktionsname:

ourand ourand

Signatur:

ourand:: objID -> (boolean, fs) -> (boolean, fs) objl: zu bearbeitendes Objekt

boolean: Wahrheitswert des Formatierstatus

fs: Formatiererstruktur

boolean: neuer Wahrheitswert des Formatierstatus

fs: neue Formatiererstruktur

Beschreibung:

Diese Funktion verknüpft den Formatierstatus des eingegebenen Objekts mit dem überlieferten Formatierstatus. Entsprechen beide Werte dem Status „formatiert", so ist auch das Ergebnis „formatiert"; wenn mindestens einer der Werte „nicht formatiert" ist, ist auch das Ergebnis „nicht formatiert" (entspricht der `and`-Verknüpfung).

Vor- und Nachbedingungen:

Diese Funktion dient als Verknüpfungsoperator für das `foldr` in der Funktion `setFormattedFlags`.

Implementierung:

Die Funktion unterscheidet zwischen finalen und nicht finalen Elementen. Handelt es sich um ein nicht finales Element, kann der Formatierstatus noch nicht ermittelt werden; zur näheren Bestimmung, wird die Funktion `setFormattedFlags` mit dem aktuellen Objekt rekursiv aufgerufen. Das Ergebnis von `setFormattedFlags` ist jedoch eine Formatiererstruktur, so daß zur Bestimmung des Status dieser erst wieder ausgelesen werden muß.
Handelt es sich beim aktuellen Objekt um ein finales Element, müssen zur Bestimmung des Formatierstatus die Boxen des Objekts untersucht werden. Dies wird mithilfe der Funktion `getFormatStatus` realisiert.

Funktionsname:

getFormatStatus getFormatStatus

Signatur:

getFormatStatus:: boxinfolist -> boolean

boxinfolist: Liste der Boxen eines Objekts

boolean: Wahrheitswert des Formatierstatus

Beschreibung:

Die Funktion ermittelt den Formatierstatus einer eingegeben Liste von Boxen eines Objekts und liefert diesen als Ergebnis zurück.

Abhängigkeiten:

In der Liste der Boxen dürfen nur Boxen eines Elements enthalten sein. Die Reihenfolge dieser Boxen muß der standardmäßigen Anordnung (Reihenfolge im Dokument) entsprechen.

Implementierung:

Die Funktion verknüpft zur Ermittlung des Ergebnisses den Formatierstatus des ersten und letzten Elements der Boxenliste. Dabei wird das Wissen über die Semantik dieser Liste ausgenutzt. Ein finales Element ist nur dann formatiert, wenn auch alle Elemente der Boxliste formatiert sind. Aufgrund der Konzeption des Einfachen Formatierers kann innerhalb der Liste keine unformatierte Box vorhanden sein, sofern die letzte Box formatiert ist.

Funktionsname:

setBoxExtend setBoxExtend

Signatur:

setBoxExtend:: fs -> objIDList -> fs

fs: Formatiererstruktur

objIDList: Liste von Objekten

fs: neue Formatiererstruktur

Beschreibung:

Die Funktion setzt neu ermittelte Werte des Boxumfangs in die darübergelegenen Knoten des Baumes ein, bis keine Veränderung im Baum mehr nötig ist.

Abhängigkeiten:

Die nicht finalen Objekte werden im Einfachen Formatierer mit Koordinaten von „0 0 0“ für alle Startkoordinaten und „9999 9999 9999“ für alle Endkoordinaten versehen, damit ein Arbeiten im Baum mit dem Vergleich größer/kleiner möglich ist. Bei der eingegebenen Liste von Objekten handelt es sich um die veränderten finalen Elemente.

Implementierung:

Die Funktion trennt rekursiv jeweils ein Element aus der eingegebenen Liste ab und bearbeitet dieses mithilfe der Funktion `setBoxExtendHelp`.

Funktionsname:

setBoxExtendHelp setBoxExtendHelp

Signatur:

setBoxExtendHelp:: fs -> objID -> fs

fs: Formatiererstruktur

objID: Objektidentifikationsnummer

fs: neue Formatiererstruktur

Beschreibung:

Die Funktion `setBoxExtendHelp` schreibt die Werte eines Objekts solange im Baum der Formatiererstruktur weiter nach oben, bis an einem nicht finalen Element keine Veränderungen mehr erforderlich sind.
Eine Anpassung der Werte eines nicht finalen Elements ist dann erforderlich, wenn das aktuelle Objekt der linke oder rechte Nachfolger seines Vorgängers ist.

Vor- und Nachbedingungen:

Die Wurzel des Baumes der Formatiererstruktur trägt die Objektidentifikationsnummer „1“.

Implementierung:

Zunächst wird der Vorgänger des aktuellen Elements ermittelt. Im Anschluß wird die eigene Position in der Liste der Nachfolger des Vorgängers analysiert. Ist das aktuelle Objekt der linke Nachfolger, bestimmen seine Werte die Anfangskoordinaten des Vorgängers, wenn es sich um den rechten Nachfolger handelt, werden dadurch die Endkoordinaten definiert. Liegt das aktuelle Element in der Mitte, sind die umspannenden Koordinaten bereits durch den rechten und linken Nachfolger des Vorgängers bestimmt.
Die rekusive Funktion terminiert, sobald die Wurzel (das erste Element des Dokuments) erreicht ist.

Funktionsname:

getPredecessor getPredecessor

Signatur:

getPredecessor:: fs -> objID -> objID

fs: Formatiererstruktur

objID: aktuelle Objektidentifikationsnummer

objID: Objektidentifikationsnummer des Vorgängers

Beschreibung:

Die Funktion ermittelt die Objektidentifikationsnummer des Vorgängers des eingegebenen Objekts unabhängig vom finalen oder nicht finalen Objektstatus.

Implementierung:

Nachdem festgestellt wurde, ob das aktuelle Objekt final oder nicht final ist, wird mithilfe der Funktionen `getFinalPredecessor` und `getNonFinalPredecessor` der Vorgänger ermittelt und zurückgeliefert.

2.12 FormatOutputCursor

Achim Mahnke

2.12.1 Funktionalität

Dieses Modul berechnet Positionen für Cursor-Bewegungen. Die Benutzungsoberflächen- und Ausgabe-Module besitzen keine Informationen über die konkrete Position von Worten oder Buchstaben auf einer formatierten Seite; dieses Wissen steckt nur im Systemteil des Formatierers. Bei Cursor-Bewegungen muß demzufolge die genaue Positionierung hinter einem Buchstaben aufgrund der Formatiererdaten vorgenommen werden; genau dafür ist dieses Modul gedacht.
Zu beachten ist, daß der Cursor nicht explizit positioniert wird, sondern nur Koordinaten ausgerechnet werden, an die er nach der Bewegung gesetzt werden muß. Die physikalische Umsetzung selbst verbleibt — durch den Aufruf der Benutzungsoberfläche — dem Ausgabe–Modul.
Über die reinen Koordinaten der neuen Cursorposition hinaus werden zusätzliche Informationen zurückgegeben: Das finale Objekt, das sich unter dem Cursor befindet und der Index des Buchstabens, hinter dem er steht. Zu diesem Zweck sind sämtliche Buchstaben innerhalb des Textes eines finalen Objektes durchnumeriert; dies geschieht im Modul `FormatTextConvert`.

2.12.2 Entwurf

Die angebotene Funktionalität wurde auf drei Funktionen aufgeteilt. Die am häufigsten benötigte ist sicherlich `f_moveCursor`. Sie bekommt die Daten der letzten Cursorposition sowie eine gewünschte Bewegungsrichtung und Schrittweite herein und gibt die neue Position zurück. Mit ihr sind alle Anforderungen abgedeckt, bei der sich der Cursor relativ zu seinem alten Ort bewegen soll.
Die zweite Funktion — `f_getCursorPos` — ermittelt die Position nicht anhand der vorherigen Cursor-Daten, sondern aufgrund von übergebenen Koordinaten, die einen Punkt auf einer Seite darstellen. Sie liefert die Position hinter dem nächstliegenden Buchstaben zurück. Diese Anforderung tritt auf, wenn der Benutzer mit der Maus auf eine Seite klickt und der Cursor möglichst nahe an dieser Stelle korrekt im Text positioniert werden muß.
Die Funktion `f_getObjID` ermittelt für einen übergebenen Punkt auf einer Seite das darunterliegende finale Objekt. Diese Funktion wurde zu Anfang der Entwicklung von der Benutzungsoberfläche benötigt, ist nun jedoch überflüssig. Ihre korrekte Funktionsweise ist nicht sichergestellt, da die Ausprogrammierung nicht an die letzten Änderungen innerhalb des Formatierers angepaßt wurde.
Zu guter letzt werden auch zwei Funktionen exportiert, die für den modulinternen Gebrauch geschrieben wurden, für andere Module jedoch auch benötigt und deshalb in die öffentliche Schnittstelle aufgenommen wurden: `determinateLine` und `determinateSubBox`.

2.12.3 Öffentliche Schnittstellen

2.12.3.1 Funktionen

Funktionsname:

f_moveCursor f_moveCursor

Signatur:

f_moveCursor:: env -> objID -> pagecoordinates -> integer -> direction -> stepwidth -> (env, objID, pagecoordinates, integer).

env: Das Environment.

objID: Die Objekt–ID der letzten Cursorposition.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition.

integer: Der Index des Zeichens innerhalb des Textes hinter dem der Cursor zuletzt stand.

direction: Gewünschte Bewegungsrichtung (`left`, `right`, `up` oder `down`).

stepwidth: Gewünschte Schrittweite (`single`, `word` oder `page`).

(env, objID, pagecoordinates, integer): Das Environment, die Objekt–ID, die Seitenkoordinaten und der Text–Index der neuen Cursorposition nach der Bewegung.

Beschreibung:

Anhand der alten Cursor-Daten und den gewünschten Bewegungsparametern wird die neue Position des Cursors ausgerechnet. Bei Bewegungen nach links oder rechts stehen als Schrittweiten `single` (zeichenweise) und `word` (wortweise) zur Verfügung. Letztere ist nicht korrekt implementiert und getestet.

Für die Richtungen `up` und `down` werden die Schrittweiten `single` und `page` akzeptiert. Bei ihnen ist die Positionierung nicht von vornherein so klar wie beim Sprung nach links oder rechts, weil dort die Reihenfolge der Zeichen das nächste Sprungziel deterministisch bestimmt[10], während hier eine Definition erfolgen muß. Es wurde folgendes festgelegt: Beim zeilenweisen Bewegen (`single`) nach oben oder unten wird in die nächste tiefer- bzw. höherliegende Zeile gesprungen, und zwar möglichst an der horizontalen Position des letzten Cursor-Ortes. Hat die entsprechende Zeile dort noch nicht angefangen, oder schon geendet, wird der Cursor an deren Anfang (vor den ersten Buchstaben) bzw. Ende (hinter den letzten Buchstaben) gesetzt.

Bei der Schrittweite `page` wird versucht, den Cursor an dieselbe (X,Y)-Position, nur auf der nächsten bzw. vorigen Seite zu setzen. Ist dort nichts, dann wird die nächste, tieferliegende Zeile angesteuert.

Konnte der Cursor nicht gemäß der gewünschten Richtung bewegt werden, weil kein Zeichen, keine Zeile oder keine Seite mehr folgt bzw. vorausgeht, dann wird die alte Cursorposition wieder zurückgegeben, ohne einen Fehler zu melden. Waren die übergebenen Daten über die alte Position fehlerhaft und konnte diese somit nicht innerhalb des angegebenen Objektes ermittelt werden, wird der Cursor auf den Anfang des betreffenden Objektes gesetzt.

Implementierung:

Das seitenweise Bewegen wird an die Funktion `f_getCursorPos` delegiert, indem die alten (X,Y)-Koordinaten kombiniert mit einer neuen Seitennummer als Eingangsparameter genommen werden.

[10] Das trifft zumindest zu, wenn man festgelegt hat, wie Worte abgegrenzt sind und die Objekte eine explizite Sequenz haben, was bei unserem Modell der Fall ist. Bei Tabellen-Elementen wäre das nicht ohne weiteres klar.

Für alle anderen Bewegungen werden zunächst die Formatierer-Daten über das bisherige Objekt angefordert und in ihnen mittels der beiden Funktionen `determinateSubBox` und `determinateLine` die Zeile gesucht, in der der Cursor bisher stand. Wurde diese nicht gefunden, tritt die Funktion `getNewPosInBoxes` mit einem auf den Ursprung der Seite „geeichten" Cursor in Kraft, um die erste mögliche Position innerhalb des Objektes zu suchen; diese wird dann zurückgegeben.

Ist die alte Line gefunden worden, macht die Funktion `doStep` weiter und liefert die Rückgabewerte.

Vor- und Nachbedingungen:

Alle Objekte innerhalb des möglichen Sprungbereichs müssen korrekt formatiert sein, sonst wird von falschen Daten ausgegangen. Insbesondere muß also beim Einfügen oder Löschen von Zeichen oder Objekten zunächst formatiert werden, bevor der Cursor neu gesetzt wird.

Funktionsname:

f_getCursorPos f_getCursorPos

Signatur:

f_getCursorPos:: env -> pagecoordinates -> (env, objID, pagecoordinates, integer).

env: Das Environment.

pagecoordinates: Seitenkoordinaten, in dessen Nähe der Cursor gesetzt werden soll.

(env, objID, pagecoordinates, integer): Das neue Environment, die neue Objekt–ID, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursorbewegung.

Beschreibung:

Diese Funktion rechnet die nächstmögliche Position aus, die den spezifizierten Seitenkoordinaten nahekommt und gleichzeitig eine korrekte Position — hinter einem Zeichen — darstellt. Sie wird verwendet, wenn der Benutzer auf eine Stelle der angezeigten Seite klickt und dort der Cursor angezeigt werden soll, oder wenn seitenweise gesprungen wird. Befindet sich unter den angegebenen Koordinaten keine Line, wird die nächste tieferliegende Line gewählt. Liegt der Punkt rechts oder links neben einer Line, wird an den Anfang bzw. das Ende derselben gesprungen.

Praktisch immer kommen als Rückgabewerte die genauen Seitenkoordinaten, auf denen der neue Cursor stehen muß, die ID des unterliegenden finalen Objektes sowie der Index des Buchstabens, hinter dem der Cursor nach der Bewegung steht zurück. Konnte der Cursor auf der gewünschten Seite jedoch gar nicht untergebracht werden, kommt eine `mtObjID` sowie der Index 0 im Ergebnistupel zurück und es wird ein Fehler gemeldet; die aufrufende Funktion muß den Erfolg der Aktion also überprüfen. Dieser Fall tritt dann ein, wenn auf der Seite entweder gar kein finales Objekt, oder nur physikalische Objekte, die der Benutzer nicht editieren soll, vorhanden sind.

Implementierung:

Es wird das erste im Dokument vorkommende finale Objekt ermittelt und dann durch die Funktion `getFirstFinalOnPage` nach dem ersten finalen Objekt auf der gewünschten Seite gesucht. Gibt es dies nicht, wird ein Fehler erzeugt und der Aufruf kehrt mit Dummy-Werten zurück. War die Suche erfolgreich, erledigt die Funktion `roundPosInObjects` die eigentliche Arbeit, indem sie die auf der Seite liegenden finalen Objekte von oben nach unten auf die gewünschte Position hin durchgeht.

Vor- und Nachbedingungen:

Wenn auf der Seite mindestens ein finales Objekt — oder genauer gesagt, eine Box mit Text liegt, die angesprungen werden darf (also kein physikalisches Objekt darstellt), dann wird eine neue Position für den Cursor berechnet. Ansonsten wird ein Fehler gemeldet und eine leere Objekt–ID zurückgeliefert; dies muß nach dem Aufruf überprüft werden.

Funktionsname:

determinateLine determinateLine

Signatur:

determinateLine:: integer -> line -> (boolean, line).

integer: Y-Koordinate des Cursors; relativ zum Anfang der Box, in der sich der Cursor befindet.

line: Zu überprüfende Zeile.

(boolean, line): Das Anzeichen für eine erfolgreiche oder -lose Suche, die überprüfte Line.

Beschreibung:

Diese Funktion wird im Zusammenhang mit einem ‚select' von `f_moveCursor` benutzt, um die Line zu finden, in der der Cursor bisher stand. `determinateLine` wird von oben nach unten über die Lines einer Box geschickt und liefert für die Lines `true` zurück, deren Y-Koordinate für den unteren Rand größer oder gleich dem übergebenen Y-Wert ist. Die erste solche Line wird durch das aufrufende ‚select' ausgewählt.

Implementierung:

Der mit der Line gespeicherte Y-Wert wird mit dem übergebenen verglichen. Ist die Line leer — was eigentlich nicht vorkommen sollte —, wird `false` zurückgeliefert.

Funktionsname:

determinateSubBox determinateSubBox

Signatur:

determinateSubBox:: pagecoordinates -> boxinfo -> (boolean, boxinfo).

pagecoordinates: Koordinaten des Cursors.

boxinfo: Zu überprüfende Box.

(boolean, boxinfo): Das Flag für eine erfolgreiche oder -lose Suche, die überprüfte Box.

Beschreibung:

Diese Funktion wird im Zusammenhang mit einem ‚select' von `f_moveCursor` benutzt, um die Box zu finden, in der der Cursor bisher stand. `determinateSubBox` wird von oben nach unten über die Boxen eines Objektes geschickt und liefert für die Box `true` zurück, in der die übergebenen Koordinaten liegen.

Implementierung:

Zunächst wird geprüft, ob die Box auf der richtigen Seite liegt, dann werden drei Rechtecke gebildet, die den Teilen einer Box entsprechen (näheres dazu in Kapitel 2.13.3.1) und getestet, ob die Koordinaten in einem dieser Rechtecke liegen. Nur in diesem Fall wird `true` zurückgeliefert.

2.12.4 Lokale Implementation

Funktionsname:

doStep doStep

Signatur:

doStep:: env -> objID -> boxinfolist -> boxinfo -> lines -> line -> pagecoordinates -> integer -> direction -> stepwidth -> (env, objID, pagecoordinates, integer).

env: Das Environment.

objID: Die Objekt–ID der letzten Cursorposition.

boxinfolist: Liste der Boxen innerhalb des übergebenen Objektes.

boxinfo: Die Box, in der sich der Cursor bisher befunden hat.

lines: Liste der Lines innerhalb der übergebenen Box.

line: Line in der sich der Cursor bisher befunden hat.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der übergebenen Box vorliegt).

integer: Der Index des Zeichens hinter dem der Cursor zuletzt stand.

direction: Gewünschte Bewegungsrichtung (`left`, `right`, `up` oder `down`).

stepwidth: Gewünschte Schrittweite (`single`, `word` oder `page`).

(env, objID, pagecoordinates, integer): Das Environment, die Objekt–ID, die Seitenkoordinaten und der Text–Index der neuen Cursorposition nach der Bewegung.

Beschreibung:

Diese Funktion steuert das Auffinden der neuen Position auf hoher Ebene. Die Formatierer–Daten sind in einer Hierarchie abgelegt, die sich von der untersten Ebene, der Line, über die Box bis zum finalen und nichtfinalen Objekt erstreckt. Der Zugriff auf die Lines ist nicht linear, man muß sich durch die Hierarchie bewegen. Genau hier liegt die Verantwortung von `doStep`.

Zunächst wird versucht, innerhalb der bisherigen Line zu suchen, wenn die Bewegung nach rechts oder links gehen soll (`getNewPosInLineByIndex`). Verläßt der Cursor mit dieser Bewegung jedoch die Line, oder wurde als Richtung `up` oder `down` angegeben, wird in den angrenzenden Lines gesucht (`getNewPosInLines`). Es kann allerdings sein, daß der Cursor die umgebende Box oder sogar das bisherige Objekt verläßt, dann muß natürlich auf diesen Ebenen weitergemacht werden (`getNewPosInBoxes` und `getNewPosInObject`).

Konnte keine neue Position ermittelt werden, gehen die alten Cursor-Daten zurück.

Implementierung:

Für jede Ebene von Formatierer–Elementen steht eine Funktion bereit, die die Suche im nächstliegenden Element durchführt. Jede dieser Funktionen „weiß", daß, wenn sie aufgerufen wird, die Suche in der darunterliegenden Ebene erfolglos war und arbeitet entsprechend. So gibt `getNewPosInLines` bei der Bewegung nach links oder rechts die Position des Anfangs oder Endes der nächsten Line zurück, weil klar ist, daß der Cursor da landen muß, wenn die Positionierung in der bisherigen Line fehlgeschlagen ist. Ebenso holt sich `getNewPosInBoxes` in diesem Fall die erste bzw. letzte Line und beauftragt dann `getNewPosInLines` entsprechend.

Für entgegengesetzte Richtungen werden dieselben Funktionen aufgerufen, bei Bewegungen nach links oder oben nur mit umgedrehter Reihenfolge der Elemente.

Funktionsname:

getNewPosInLine getNewPosInLine

Signatur:

getNewPosInLine:: env -> objID -> wordlist -> line -> pagecoordinates -> direction -> stepwidth -> integer -> integer -> (env, boolean, pagecoordinates, integer).

env: Das Environment.

objID: Die ID des Objektes, in dem gesucht wird.

wordlist: Liste der Worte innerhalb des übergebenen Objektes.

line: Line, in der gesucht werden soll.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der umgebenden Box vorliegt).

direction: Gewünschte Bewegungsrichtung (`left` und `right` werden hier unterstützt).

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

integer: Der Index des Zeichens, hinter dem der Cursor zuletzt stand.

integer: Die Y-Koordinate des Anfangs der Box, in der gesucht wird.

(env, boolean, pagecoordinates, integer): Das neue Environment, das Anzeichen für eine erfolgreiche oder -lose Suche, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion versucht, die neue Cursorposition innerhalb der übergebenen Line anhand der X-Koordinate der alten Cursor-Daten zu finden. Gelingt dies, wird als zweite Tupel-Komponente `true` zurückgeliefert. Wandert der Cursor jedoch mit dieser Bewegung aus der Line heraus, kommt hier ein `false` zurück.

Implementierung:

Zunächst wird geprüft, ob die X-Koordinate des Anfangs der Line korrigiert werden muß. Dies ist notwendig, wenn mehrere Lines auf einer Höhe liegen, dann sind diese optisch durch ein virtuelles Leerzeichen getrennt, dessen Breite zu dem gespeicherten Anfang der Line hinzuaddiert werden muß.

Anschließend wird mit einer der beiden Funktionen `determinatePosLeft` und `determinatePosRight` die Line durchgegangen, um die neue Position zu finden, indem das nächste Zeichen mit einer kleineren bzw. größeren X-Koordinate gesucht wird.

Anmerkung: In der Programmierphase dieses Moduls war die Funktion lange Zeit für die Bewegung innerhalb einer Zeile zuständig. Dies erwies sich jedoch als falsch, da die Heranziehung der alten X-Koordinate zur Berechnung der neuen Position keine geeignete Vorgehensweise ist. Ein neuer Umbruch nach Einfügen oder Löschen eines Zeichens kann beim Blocksatz z.B. erhebliche Verschiebungen innerhalb der Zeile hervorrufen, und der alte X-Wert des Cursors ist somit unbrauchbar.

Aus diesem Grund wurde die Funktion `getNewPosInLineByIndex` eingeführt, die sich an dem Index des alten Cursors innerhalb des Textes orientiert, der bei geändertem Umbruch weiterhin gültig ist.

Diese Funktion wird nur noch zum Auffinden einer Position möglichst nahe an einer bestimmten X-Koordinate gebraucht, wie es beim Wechsel der Zeile mit `up` oder `down` erforderlich ist. Da sie hierfür von der Funktion `getNewPosInLines` nur mit der Richtung `right` aufgerufen wird, könnte man die Implementierung vereinfachen und insbesondere die Funktion `determinatePosLeft` streichen.

Funktionsname:

getNewPosInLineByIndex getNewPosInLineByIndex

Signatur:

getNewPosInLineByIndex:: env -> objID -> wordlist -> line -> pagecoordinates -> direction -> stepwidth -> integer -> integer -> (env, boolean, pagecoordinates, integer).

env: Das Environment.

objID: Die Objekt–ID der letzten Cursorposition.

wordlist: Liste der Worte innerhalb des übergebenen Objektes.

line: Line, in der sich der Cursor bisher befunden hat.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der umgebenden Box vorliegt).

direction: Gewünschte Bewegungsrichtung (`left` und `right` werden hier unterstützt).

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

integer: Der Index des Zeichens, hinter dem der Cursor zuletzt stand.

integer: Die Y-Koordinate des Anfangs der Box, in der der Cursor bisher stand.

(env, boolean, pagecoordinates, integer): Das neue Environment, das Anzeichen für eine erfolgreiche oder -lose Suche, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion versucht, die neue Cursorposition innerhalb der übergebenen Line — in der sich der Cursor vorher auch befunden hat — zu finden. Sie benutzt den alten Index innerhalb des Textes, um die vorherige Position innerhalb der Line zu lokalisieren. Kann ein neuer Ort für den Cursor gefunden werden, gibt die Funktion als zweite Tupel-Komponente `true` zurück. Wandert der Cursor jedoch mit dieser Bewegung aus der Line heraus, kommt hier ein `false` zurück.

Implementierung:

Zunächst wird geprüft, ob die X-Koordinate des Anfangs der Line korrigiert werden muß. Dies ist notwendig, wenn mehrere Lines auf einer Höhe liegen, dann sind diese optisch durch ein virtuelles Leerzeichen getrennt, dessen Breite zu dem gespeicherten Anfang der Line hinzuaddiert werden muß.

Anschließend wird mit einer der beiden Funktionen `determinatePosLeftByIndex` und `determinatePosRightByIndex` die Line durchgegangen, um die neue Position zu finden, indem das nächste Zeichen mit einem kleineren bzw. größeren Index gesucht wird.

Vor- und Nachbedingungen:

Die übergebene Line muß das Zeichen enthalten, hinter dem der Cursor vorher gestanden hat, sonst wird keine korrekte Position gefunden.

Funktionsname:

getNewPosInLines getNewPosInLines

Signatur:

getNewPosInLines:: objID -> integer -> pagecoordinates -> direction -> wordlist -> stepwidth -> (env, boolean, pagecoordinates, integer) -> line -> (env, boolean, pagecoordinates, integer).

objID: Die ID des Objektes, in dem gesucht wird.

integer: Die Y-Koordinate des Anfangs der Box, in der gesucht wird.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der umgebenden Box vorliegt).

direction: Gewünschte Bewegungsrichtung (`left`, `right`, `up` oder `down`).

wordlist: Liste der Worte innerhalb des übergebenen Objektes.

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

(env, boolean, pagecoordinates, integer): Die Ergebnisse des Aufrufs dieser Funktion mit der vorherigen Line (foldl).

line: Zu untersuchende Line.

(env, boolean, pagecoordinates, integer): Das neue Environment, das Anzeichen für eine erfolgreiche oder -lose Suche, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion wird mittels ‚foldl' über die sich der bisherigen Cursor-Line anschließenden bzw. vorgehenden Lines geschickt, um dort eine neue Cursorposition zu finden, wenn feststeht, daß der Cursor aus seiner bisherigen Line herausbewegt wird. Soll nach rechts oder links gegangen werden, dann sorgt diese Funktion dafür, daß der Anfang bzw. das Ende der nachfolgenden oder vorhergehenden Line als neue Position für den Cursor genommen wird. Soll nach unten oder oben gegangen werden, wird der Cursor entsprechend seiner alten X-Koordinate in die nachfolgende oder vorausgehende Line bewegt. Die Funktion ist sehr komplex, da auch mehrere Lines auf einer Höhe liegen können und dies bei der Bewegung nach unten oder oben beachtet werden muß.

Implementierung:

Grundsätzliches: Wird als zweiter Parameter des Ergebnis-Tupels `true` zurückgeliefert, dann sorgt die erste Gleichung der Funktion dafür, daß die nachfolgenden Aufrufe durch das ‚foldl' gleich wieder beendet werden, so daß das Ergebnis über alle noch folgenden Lines bis zum Ende durchgereicht wird. Das `true` wird gesetzt, wenn in der untersuchten Line eine neue Position gefunden wurde. Im negativen Fall sorgt das Setzen eines `false`-Wertes für eine weitere Nachforschung in der nächsten Line.

Bei der Bewegung nach links wird die Länge der gesamten Line ermittelt und zum Anfangs-X-Wert der Line hinzuaddiert, um die neue X-Position zu errechnen. Der Index wird auf die Nummer des letzten Zeichens im letzten Wort der Line gesetzt.

Bei der Bewegung nach rechts werden einfach die Koordinaten des Line-Anfangs als neue Koordinaten des Cursors gesetzt. Der Index entspricht der Nummer des ersten Zeichens im ersten Wort der Line.

Bei der Bewegung nach unten wird darauf geachtet, daß die gerade untersuchte Line nicht auf derselben Höhe wie die ursprüngliche liegt, denn in diesem Fall muß die nächste untersucht werden. Außerdem muß geprüft werden, ob die gewünschte X-Position des neuen Cursors über das Ende der Line hinausgeht. Tut sie es, gibt es zwei Möglichkeiten: Wenn eine Line auf derselben Höhe mit der jetzigen folgt, dann muß in dieser weitergesucht werden (also wird `false` für die gerade untersuchte Line zurückgeliefert). Gibt es jedoch keine auf derselben Höhe, dann wird der Cursor ans Ende der jetzigen gesetzt. Paßte der Cursor von vornherein in die untersuchte Line, dann wird mittels `getNewPosInLine` eine neue Position in ihr ausfindig gemacht.

Die Bewegung nach oben wird im Prinzip genauso behandelt, nur daß hier danach geschaut werden muß, ob die gewünschte X-Koordinate nicht vor der zu untersuchenden Line liegt, da die Lines bei dieser Richtung von unten nach oben (und somit Lines, die auf einer Höhe liegen von rechts nach links) durchsucht werden.

Funktionsname:

getNewPosInBoxes getNewPosInBoxes

Signatur:

getNewPosInBoxes:: objID -> pagecoordinates -> direction -> wordlist -> stepwidth -> (env, boolean, pagecoordinates, integer) -> boxinfo -> (env, boolean, pagecoordinates, integer).

objID: Die ID des Objektes, in dem gesucht wird.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition.

direction: Gewünschte Bewegungsrichtung (`left`, `right`, `up` oder `down`).

wordlist: Liste der Worte innerhalb des übergebenen Objektes.

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

(env, boolean, pagecoordinates, integer): Die Ergebnisse des Aufrufs dieser Funktion mit der vorherigen Box (foldl).

boxinfo: Zu untersuchende Box.

(env, boolean, pagecoordinates, integer): Das neue Environment, das Anzeichen für eine erfolgreiche oder -lose Suche, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion wird mittels ‚foldl' über die anschließenden bzw. vorgehenden Boxen geschickt, um dort eine neue Cursorposition zu finden, wenn feststeht, daß der Cursor aus seiner bisherigen Box herausbewegt wird. Die eigentliche Suche wird auch hier in den Lines durchgeführt, nur müssen diese eben aus den angrenzenden Boxen ausgelesen werden. Außerdem kann ein Wechsel der Box ein Sprung auf eine neue Seite bedeuten; dies muß überprüft werden.

Implementierung:

Bei Bewegungen nach rechts oder unten werden die Lines der übergebenen Box in ihrer Original-Reihenfolge an die Funktion `getNewPosInLines` übergeben, bei Bewegungen nach links oder unten wird ihre Reihenfolge vorher umgedreht. Anschließend wird die Seitennummer der Box in die ermittelten Koordinaten eingefügt.

Funktionsname:

getNewPosInObject getNewPosInObject

Signatur:

getNewPosInObject:: env -> objID -> pagecoordinates -> direction -> stepwidth -> (env, boolean, objID, pagecoordinates, integer).

env: Das Environment.

objID: Die ID des Objektes, in dem der Cursor bisher stand.

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition.

direction: Gewünschte Bewegungsrichtung (`left`, `right`, `up` oder `down`).

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

(env, boolean, objID, pagecoordinates, integer): Das neue Environment, das Flag für eine erfolgreiche oder -lose Suche, die neue ObjID, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion sucht die neue Cursorposition in den, dem übergebenen Objekt nachfolgenden bzw. vorhergehenden Objekten; abhängig davon, welche Richtung angegeben wurde. Ist die Suche erfolgreich, wird `true` als zweiter Tupel-Parameter zurückgegeben, ansonsten `false`. Im Fall der erfolglosen Suche gehen die übergebenen Daten der alten Cursorposition wieder zurück. Die physikalischen (also erst beim Formatieren eingefügten Objekte) werden bei der Suche nicht berücksichtigt, da ein Editieren nicht möglich sein soll und deshalb der Cursor auch nicht in diese Objekte hineingelangen darf.

Implementierung:

Zunächst holt sich die Funktion mittels `getNeighbour` die ID des benötigten Nachbar-Objektes, um dann dessen Boxliste auszulesen und in die Funktion `getNewPosInBoxes` zu stecken, die die eigentliche Suche durchführt. Wurde keine neue Position gefunden, folgt ein rekursiver Aufruf, um in dem nächsten Objekt nachzusehen.

Funktionsname:

determinatePosLeft determinatePosLeft

Signatur:

determinatePosLeft:: pagecoordinates -> font -> stepwidth -> integer -> integer -> (env, boolean, integer, integer, integer, integer, integer) -> wordlistelement -> (env, boolean, integer, integer, integer, integer, integer).

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der umgebenden Box vorliegt).

font: Der Font, in dem der Text des Objektes gesetzt ist, in dem gesucht wird.

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

integer: Text–Index der letzten Cursorposition.

integer: Spacefactor; der verbleibende Weißraum in der Zeile, in der das übergebene wordlistelement steht.

(env, boolean, integer, integer, integer, integer, integer): Ergebnis des vorherigen Aufrufs (foldl).

wordlistelement: Das nach einer neuen Position zu durchsuchende Wort.

(env, boolean, integer, integer, integer, integer, integer): Neues Environment, das Flag für die erfolgreiche- oder lose Suche, die Nummer des Leerzeichens, das vor dem übergebenen Wort liegt, die X-Position, bis zu der bisher gesucht wurde, die nächste zu überprüfende X-Position, der Index, bis zu dem bisher gesucht wurde, der nächste zu überprüfende Index.

Beschreibung:

Diese Funktion wird von `getNewPosInLine` mittels ‚foldl' über die Wörter einer Line geschickt, um von links nach rechts nach einer neuen Position für den Cursor zu suchen, der ein Zeichen oder ein Wort nach *links* gehen soll. Wird diese gefunden, sorgt ein `true` als zweiter Parameter des Ergebnistupels dafür, daß in allen folgenden Aufrufen durch ‚foldl' nicht mehr weitergesucht, sondern das Ergebnis weitergereicht wird, so daß es als Ergebnis des gesamten ‚fold' dient. Ein `false` hat das Weitersuchen im nächsten Wort zur Folge.

Implementierung:

In der vierten Komponente des Ergebnistupels (XPosL) wird über die wiederholten Aufrufe hinweg die X-Position gespeichert, bis zu der keine neue Position gefunden wurde. In der fünften Komponente ist die nächste X-Position gespeichert, die untersucht werden muß. Ist diese größer als der alte X-Wert des Cursors, wird XPosL als neue Position zurückgeliefert, da dies vom Anfang der Zeile aus gesehen der letze X-Wert ist, der vor dem alten Cursor-X-Wert liegt. Der zu den jeweiligen Positionen gehörige Index wird in den letzten beiden Komponenten weitergereicht und der vorletzte ist der zu XPosL gehörige Index.

Ist der zu untersuchende X-Wert nicht größer als der alte Cursor-Wert, wird in dem übergebenen Wort weitergesucht. Ist auch dort noch nicht der alte Wert erreicht, wird die Breite des Wortes sowie des anschließenden Leerzeichens ermittelt und in dem Ergebnistupel als neue zu untersuchende X-Position an den nächsten Aufruf weitergegeben. Als bisher untersuchte Position wird der X-Wert aus dem vorherigen Aufruf plus der Breite des übergebenen Wortes weitergereicht.

Um die Suche zu beschleunigen, wird der alte Index mit dem Index des letzten Zeichens im zu untersuchenden Wort verglichen. Ist er kleiner, steht fest, daß dieses Wort übersprungen werden kann.

Die Suche muß von links nach rechts durchgeführt werden, weil die Berechnung der X-Positionen nur mit Informationen über die vorhergehenden Wörter und Zwischenräume erfolgen kann. Näheres dazu findet sich im Modul *SimpleFormatter* (Seite 290).

Der Spacefactor wird als Parameter für die Funktion benötigt, die die Breite eines Leerzeichens ausrechnet. Der Font ist für die Berechnung der Ausmaße des Wortes notwendig.

Anmerkung: Der Teil des Funktionsrumpfes, der für das wortweise Springen verantwortlich ist, wurde nicht an die letzten Änderungen angepaßt und kann fehlerhaft sein.

Vor- und Nachbedingungen:

Zu beachten ist, daß die Funktion auch `true` zurückliefert, wenn der Cursor ganz am Anfang der Line stand. Der Aufrufende muß also prüfen, ob der X-Wert der gelieferten Position wirklich kleiner als der alte Wert ist.

Funktionsname:

determinatePosRight determinatePosRight

Signatur:

determinatePosRight:: pagecoordinates -> font -> stepwidth -> integer -> integer -> (env, boolean, integer, integer, integer, integer, integer) -> wordlistelement -> (env, boolean, integer, integer, integer, integer, integer).

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der umgebenden Box vorliegt).

font: Der Font, in dem der Text des Objektes gesetzt ist, in dem gesucht wird.

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

integer: Text–Index der letzten Cursorposition.

integer: Spacefactor; der verbleibende Weißraum in der Zeile, in der das übergebene wordlistelement steht.

(env, boolean, integer, integer, integer, integer, integer): Ergebnis des vorherigen Aufrufs (foldl).

wordlistelement: Das nach einer neuen Position zu durchsuchende Wort.

(env, boolean, integer, integer, integer, integer, integer): Neues Environment, das Flag für die erfolgreiche- oder lose Suche, die Nummer des Leerzeichens, das vor dem übergebenen Wort liegt, die X-Position, bis zu der bisher gesucht wurde, die nächste zu überprüfende X-Position, der Index, bis zu dem bisher gesucht wurde, der nächste zu überprüfende Index.

Beschreibung:

Diese Funktion wird von `getNewPosInLine` mittels ‚foldl‘ über die Wörter einer Line geschickt, um von links nach rechts nach einer neuen Position für den Cursor zu suchen, der ein Zeichen oder ein Wort nach *rechts* gehen soll. Wird diese gefunden, sorgt ein `true` als zweiter Parameter des Ergebnistupels dafür, daß in allen folgenden Aufrufen durch ‚foldl‘ nicht mehr weitergesucht, sondern das Ergebnis weitergereicht wird, so daß es als Ergebnis des gesamten ‚foldl‘ dient. Ein `false` hat das Weitersuchen im nächsten Wort zur Folge.

Implementierung:

Im Prinzip arbeitet die Funktion genauso, wie `determinatePosLeft`, nur daß der Vergleich der alten X-Koordinate des Cursors mit denen der zu untersuchenden Zeichen so erfolgt, daß die nächstfolgende Position rechts vom alten Cursor gefunden wird.

Funktionsname:

determinateWordPos determinateWordPos

Signatur:

determinateWordPos:: pagecoordinates -> font -> (integer -> integer -> integer) -> (integer -> integer -> boolean) -> (env, boolean, integer, integer, integer, integer) -> char -> (env, boolean, integer, integer, integer, integer).

pagecoordinates: Die Seitenkoordinaten der letzten Cursorposition (Die Y-Komponente wurde vorher korrigiert, so daß sie relativ zum Anfang der umgebenden Box vorliegt).

font: Der Font, in dem der Text des Objektes gesetzt ist, in dem gesucht wird.

(integer -> integer -> integer): Funktion, die zwei Integer übernimmt und einen zurückgibt. Wird bei den Aufrufen mit der Funktion + aktualisiert.

(integer -> integer -> boolean): Funktion, die zwei Integer übernimmt und einen Boolwert zurückgibt. Wird bei den Aufrufen mit der Funktion > oder >= aktualisiert.

(env, boolean, integer, integer, integer, integer): Ergebnis des vorherigen Aufrufs (foldl).

char: Das nach einer neuen Position zu überprüfende Zeichen.

(env, boolean, integer, integer, integer, integer): Neues Environment, das Flag für die erfolgreiche- oder lose Suche, die X-Position, bis zu der bisher gesucht wurde, die nächste zu überprüfende X-Position, der Index, bis zu dem bisher gesucht wurde, der nächste zu überprüfende Index).

Beschreibung:

Die Funktion wird von `determinatePosLeft` und `determinatePosRight` mittels ‚foldl‘ über die Buchstaben eines Wortes geschickt, um von links nach rechts nach einer neuen Position für den Cursor zu suchen, der ein Zeichen nach links oder rechts gehen soll.

Die Funktion wird mit zwei Funktionen vorparametrisiert: Die erste dient zum Berechnen der X-Position, die hinter einem Zeichen gilt und wird immer mit + aktualisiert. Als `determinatePosLeft` in einem frühen Stadium der Implementierung noch von rechts nach links gearbeitet hat, wurde hier ein – benutzt, um die Breite eines Zeichens von dem bisherigen X-Wert abzuziehen, statt sie aufzuaddieren. Daher rührt auch der ursprüngliche Gedanke für die Verwendung eines Funktionals. Die zweite Funktion vergleicht zwei Integerzahlen und liefert einen Boolwert. Sie wird zur Überprüfung verwendet, ob die neue Position gefunden wurde und ist im Falle von `determinatePosLeft` $>=$ und im Falle von `determinatePosRight` $>$. Erstere muß ein Größergleich benutzen, weil die Position, die identisch mit der des alten Cursors ist, bei der Bewegung nach links schon nicht mehr in Frage kommt, während sie bei der Bewegung nach rechts noch nicht ausreicht, um als neue Position zu gelten.

Wird eine neue Position gefunden, sorgt ein `true` als zweiter Parameter des Ergebnistupels dafür, daß in allen folgenden Aufrufen durch ‚foldl' nicht mehr weitergesucht, sondern das Ergebnis weitergereicht wird, so daß es als Ergebnis des gesamten ‚foldl' dient. Ein `false` hat das Weitersuchen beim nächsten Zeichen zur Folge.

Implementierung:

Die Breite des übergebenen Zeichens wird ermittelt und durch die ebenfalls übergebene Funktion zu dem bisherigen X-Wert addiert, der im ersten Integer-Parameter des Eingangs-Tupels zwischengespeichert wird. Anschließend wird das Ergebnis der Vergleichsfunktion zusammen mit den neuen Werten für die bisher betrachtete X-Position und die Indices an den nächsten Aufruf durch ‚foldl' weitergereicht.

Funktionsname:

determinatePosLeftByIndex determinatePosLeftByIndex

Signatur:

determinatePosLeftByIndex:: env -> font -> stepwidth -> integer -> integer -> integer -> wordlist -> (env, boolean, integer, integer).

env: Das Environment.

font: Der Font, in dem der Text des Objektes gesetzt ist, in dem gesucht wird.

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

integer: Text–Index des alten Cursors.

integer: Spacefactor; der verbleibende Weißraum in der Zeile, in der die übergebenen Wörter stehen.

integer: Anzahl der vor der zu untersuchenden Line befindlichen Spaces, die auf derselben Höhe mit dieser liegen.

wordlist: Liste der Worte innerhalb der zu untersuchenden Line.

(env, boolean, integer, integer): Das neue Environment, das Anzeichen für eine erfolgreiche oder -lose Suche, die neue X-Koordinate und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion durchsucht eine Line nach der neuen Cusorposition links von der alten, indem sie sich an dem Text–Index des alten Cursors orientiert. Im Prinzip liefert sie die X-Koordinate des Zeichens zurück, das den nächstkleineren Index hat. Ist dieses jedoch ein Leerzeichen zwischen zwei Worten (im Text können mehrere Spaces zwischen zwei Wörtern liegen; sie werden bei der Indizierung mitgerechnet), dann wird die Position hinter dem vorhergehenden Wort genommen. Liegt die neue Position innerhalb der Line, wird `true` zurückgeliefert, sonst `false`.

Implementierung:

Die Wörter der Line werden zunächst in zwei Listen aufgeteilt, wobei die erste Liste (L) die Wörter enthält, die komplett links von dem angenommenen neuen Index (alter Index minus 1) liegen, während die zweite alle anderen enthält (R). Dann werden eine Reihe von Möglichkeiten überprüft:

Liegt der neue Index links vom ersten Wort in der Liste R, dann ist er genau im Zwischenraum zwischen diesem und dem letzten Wort der Liste L, und die neue Position liegt genau hinter diesem. Ist die Liste L allerdings leer, wandert der Cursor mit dieser Bewegung in die vorausgehende Line und es wird `false` zurückgeliefert.

Die anderen Möglichkeiten bestehen darin, daß der Index am Anfang des ersten Wortes der Liste R oder in diesem Wort drin liegt. Dann wird die korrekte Breite der Zeichenkette, die links vom neuen Cursor liegt, errechnet und als neue X-Koordinate zusammen mit dem Index zurückgegeben.

Vor- und Nachbedingungen:

Wird fälschlicherweise ein Index übergeben, dessen Zeichen nicht innerhalb der zu durchsuchenden Wörter zu finden ist, dann kommt ein falsches Ergebnis zurück. Speziell wird, wenn der Index größer als alle in den Wörtern vorkommenden ist, das Ende der Line als neue Position gemeldet, obwohl dies falsch sein könnte.

Funktionsname:

determinatePosRightByIndex determinatePosRightByIndex

Signatur:

determinatePosRightByIndex:: env -> font -> stepwidth -> integer -> integer -> integer -> wordlist -> (env, boolean, integer, integer).

env: Das Environment.

font: Der Font, in dem der Text des Objektes gesetzt ist, in dem gesucht wird.

stepwidth: Gewünschte Schrittweite (`single` und `word` werden hier unterstützt).

integer: Text–Index des alten Cursors.

integer: Spacefactor; der verbleibende Weißraum in der Zeile, in der die übergebenen Wörter stehen.

integer: Anzahl der vor der zu untersuchenden Line befindlichen Spaces, die auf derselben Höhe mit dieser liegen.

wordlist: Liste der Worte innerhalb der zu untersuchenden Line.

(env, boolean, integer, integer): Das neue Environment, das Anzeichen für eine erfolgreiche oder -lose Suche, die neue X-Koordinate und der neue Text–Index nach der Cursor-Bewegung.

Beschreibung:

Diese Funktion durchsucht eine Line nach der neuen Cusorposition rechts von der alten, indem sie sich an dem Text–Index des alten Cursors orientiert. Im Prinzip liefert sie die X-Koordinate des Zeichens zurück, das den nächsthöheren Index hat. Ist dieses jedoch ein Leerzeichen zwischen zwei Worten (im Text können mehrere Spaces zwischen zwei Wörtern liegen; sie werden bei der Indizierung mitgerechnet), dann wird die Position vor dem nachfolgenden Wort genommen. Liegt die neue Position innerhalb der Line, wird `true` zurückgeliefert, sonst `false`.

Implementierung:

Die Wörter der Line werden zunächst in zwei Listen aufgeteilt, wobei die erste Liste (L) die Wörter enthält, die komplett links von dem angenommenen neuen Index (alter Index plus 1) liegen, während die zweite alle anderen enthält (R). Dann werden eine Reihe von Möglichkeiten überprüft:

Liegt der neue Index links vom ersten Wort in der Liste R und hinter dem Ende des letzten Wortes der Liste L, dann muß die neue Position am Anfang des ersten Wortes in R sein. Ist R jedoch leer, geht der Cursor über die Zeile nach rechts hinaus und es wird `false` zurückgegeben.

Die anderen Möglichkeiten bestehen darin, daß der Index am Ende des letzten Wortes der Liste L oder innerhalb des ersten Wortes aus R liegt. Dann wird die neue X-Koordinate errechnet und zusammen mit dem Index zurückgegeben.

Vor- und Nachbedingungen:

Wird fälschlicherweise ein Index übergeben, dessen Zeichen nicht innerhalb der zu durchsuchenden Wörter zu finden ist, dann kommt ein falsches Ergebnis zurück. Speziell wird, wenn der Index kleiner als alle in den Wörtern vorkommenden ist, der Anfang der Line als neue Position gemeldet, obwohl dies falsch sein könnte.

Funktionsname:

isAtEndOfWord isAtEndOfWord

Signatur:

isAtEndOfWord:: integer -> wordlist -> boolean.

integer: Ein Index.

wordlist: Liste mit Worten.

boolean: Ergebnis.

Beschreibung:

Es wird überprüft, ob der übergebene Index mit dem Index des letzten Buchstabens innerhalb des ersten Wortes der Liste übereinstimmt. Ist dies der Fall, kommt `true`, sonst `false` zurück. Ist die Liste leer, kommt `false` zurück.

Implementierung:

Zu dem Index des ersten Zeichens, der mit dem Wort gespeichert ist, wird die Länge des Wortes hinzuaddiert und eins abgezogen.

Funktionsname:

shallBeAtEndOfWord shallBeAtEndOfWord

Signatur:

shallBeAtEndOfWord:: integer -> wordlist -> boolean.

integer: Ein Index.

wordlist: Liste mit Worten.

boolean: Ergebnis.

Beschreibung:

Es wird überprüft, ob der übergebene Index kleiner als der Index des letzten Zeichens vor dem ersten Wort der Liste ist. Ist dies der Fall, kommt `true`, sonst `false` zurück. Ist die Liste leer, kommt `true` zurück!

Implementierung:

Von dem Index des ersten Zeichens, der mit dem Wort gespeichert ist, wird eins abgezogen (= Index des Zeichens unmittelbar vor diesem Wort) und dieser mit dem übergebenen Index verglichen.

Funktionsname:

getSpaceWidth getSpaceWidth

Signatur:

getSpaceWidth:: env -> font -> integer -> integer -> integer.

env: Das Environment.

font: Ein Font.

integer: Spacefactor; der verbleibende Weißraum einer Zeile.

integer: Die Nummer des Leerzeichens innerhalb der Zeile.

integer: Breite des Leerzeichens in Pixeln.

Beschreibung:

Die Funktion berechnet die Breite eines Leerzeichens im angegebenen Font. Wenn es sich um ein Leerzeichen innerhalb eines Blocksatzes handelt, werden der Spacefactor und die Spacenummer benötigt, um die Breite auszurechnen.

Implementierung:

Handelt es sich um ein Leerzeichen im Blocksatz (Spacefactor <> 0), werden der Spacefactor und die Spacenummer an die Funktion `calculateWord2WordAddSpace` aus dem Modul „FormatAlignment" übergeben, um dessen Breite auszurechnen, sonst wird einfach die Breite eines Leerzeichens („ ") durch eine Funktion des Moduls „OutputFormat" berechnet. Wird beim Blocksatz als Spacenummer eine 0 übergeben, wird auch als Breite 0 zurückgeliefert, weil die Spacenummern nach Definition bei 1 anfangen.

Funktionsname:

nextLineAtSameY

nextLineAtSameY

Signatur:

nextLineAtSameY:: env -> objID -> integer -> direction -> line.

env: Das Environment.

objID: Die ID des finalen Objektes, in dem die aufrufende Funktion nach der neuen Cursorposition sucht.

integer: Y-Koordinate der Line, in der gesucht wird (absolut zum Seitenanfang).

direction: Gewünschte Bewegungsrichtung (`left` oder `right`).

line: Die Line, die im übergebenen Objekt auf der angegebenen Höhe liegt. Ist dort keine vorhanden, wird eine ‚mtLine' zurückgegeben.

Beschreibung:

Die Funktion überprüft, ob im angegebenen Objekt eine Line auf der spezifizierten Höhe liegt. Sie wird benutzt, um herauszufinden, ob noch mehr Zeilen auf der Höhe einer Line liegen, die gerade nach einer neuen Cursorposition durchsucht wird. Die Bewegungsrichtung wird gebraucht, um die Suche auf einen Teil des Objektes einschränken zu können.

Wird eine Line auf gleicher Höhe gefunden, wird diese zurückgeliefert, ansonsten eine ‚mtLine' (leere Line).

Implementierung:

Die Überprüfung braucht nur für die erste bzw. letzte Line des übergebenen Objektes durchgeführt zu werden. Dies erklärt sich wie folgt: Wird `right` angegeben, soll überprüft werden, ob rechts neben der gerade durchsuchten Line eine weitere existiert, die gleichauf mit dieser liegt. Da in einem Objekt jede Line eine andere Y-Position hat, kann sich die nächste Line auf derselben Höhe also nur im nächsten Objekt (dem übergebenen) befinden und dort muß sie die erste Line überhaupt sein. Ähnliches gilt für das Nachschauen nach links; hier muß die Line die letzte innerhalb des vorhergehenden Objektes sein. Es wird also einfach die entsprechende Line herausgesucht und ihre Y-Koordinate mit der übergebenen verglichen.

Funktionsname:

getTextWidth getTextWidth

Signatur:

getTextWidth:: env -> font -> wordlist -> integer -> integer -> (integer, env).

env: Das Environment.

font: Der Font, in dem die übergebenen Worte gesetzt sind.

wordlist: Liste mit Wörtern.

integer: Spacefactor; der verbleibende Weißraum in der Zeile, in der die übergebenen Wörter stehen.

integer: Anzahl der vor der zu untersuchenden Line befindlichen Spaces, die auf derselben Höhe mit dieser liegen.

(integer, env): Die Gesamtbreite der übergebenen Wörter und das neue Environment.

Beschreibung:

Die Funktion errechnet die Gesamtbreite der übergebenen Worte einschließlich der Leerräume.

Implementierung:

Die Breite der Wörter mit ihren Zwischenräumen wird durch eine Anfrage im „Output-Format“–Modul bestimmt. Anschließend werden die zusätzlichen Zwischenräume, die beim Blocksatz hinzukommen, mit Hilfe der Funktion `collectAddSpace` ermittelt und addiert.

Funktionsname:

mygetLineWords mygetLineWords

Signatur:

mygetLineWords:: wordlist -> integer -> integer -> wordlist.

wordlist: Liste mit Wörtern.

integer: Start–Index.

integer: Schluß–Index.

wordlist: Liste der Wörter, die durch die beiden Indices begrenzt werden.

Beschreibung:

Die Wörter eines Textobjektes sind durchnummeriert; diese Funktion liefert aus der übergebenen Wortliste die Teilliste von Wörtern, deren Indices größergleich dem Start–Index und kleinergleich dem Schluß–Index sind.

Implementierung:

Die Funktion geht die Wörter von links nach rechts durch und konstruiert rekursiv die Rückgabeliste mit den entsprechenden Wörtern.

Funktionsname:

mymakeLine mymakeLine

Signatur:

mymakeLine:: wordlist -> string.

wordlist: Eine Liste mit Worten.

string: Die daraus resultierende Zeichenkette.

Beschreibung:

Die Funktion erzeugt aus der Wortliste eine Zeichenkette mit Leerzeichen zwischen den Worten.

Implementierung:

Die Worte werden aneinandergehängt und für alle, bis auf das letzte, Leerzeichen eingesetzt.

Funktionsname:

collectAddSpace collectAddSpace

Signatur:

collectAddSpace:: integer -> integer -> integer -> integer.

integer: Ein Spacefactor.

integer: Eine Spacenummer.

integer: Eine abschließende Spacenummer.

integer: Gesamtbreite der angefragten Spaces.

Beschreibung:

Die Funktion bekommt den Spacefactor einer Line sowie eine Anfangs– und End–Spacenummer herein, die Spaces innerhalb dieser Line bezeichnen. Zurückgegeben wird der Gesamtwert, der zu den normalen Space–Breiten hinzugerechnet werden muß, um auf die tatsächlich gesetzte Breite einer Zeichenkette zu kommen.

Implementierung:

Durch Rekursion wird die zusätzliche Breite jedes einzelnen Spaces zwischen den übergebenen Nummern (Anfangs- und Endnummer eingeschlossen) angefragt und aufaddiert.

Funktionsname:

getFirstFinalOnPage getFirstFinalOnPage

Signatur:

getFirstFinalOnPage:: env -> pagenumber -> objID -> (boolean, objID).

env: Das Environment.

pagenumber: Eine Seitennummer.

objID: Die ID eines finalen Objektes.

(boolean, objID): Flag, ob ein Teil des übergebenen Objektes auf der Seite liegt sowie das übergebene Objekt selbst.

Beschreibung:

Die Funktion durchsucht von dem spezifizierten Objekt an alle nachfolgenden Objekte daraufhin, ob eine ihrer Boxen auf der angegebenen Seite liegt und liefert das erste Objekt, auf das dieses zutrifft zurück. Wird kein solches gefunden, kommt ein `false` zurück.

Implementierung:

Zunächst wird das übergebene Objekt überprüft. Fällt dies negativ aus, wird das nächste im Dokument folgende finale Objekt ermittelt und ein rekursiver Aufruf getätigt, um dort weiterzusuchen. Ist man beim letzten Objekt angekommen und auch dieses liegt nicht auf der entsprechenden Seite, wird ein `false` im Verbund mit dem letzten finalen Objekt des Dokumentes zurückgereicht, ansonsten ein `true` zusammen mit dem Objekt, das die Bedingung erfüllt.

Funktionsname:

roundPosInObjects roundPosInObjects

Signatur:

roundPosInObjects:: env -> pagecoordinates -> objID -> (env, objID, pagecoordinates, integer).

env: Das Environment.

pagecoordinates: Die Seitenkoordinaten, in dessen Nähe der Cursor gesetzt werden soll.

objID: Die ID des zu untersuchenden Objektes.

(env, objID, pagecoordinates, integer): Das neue Environment, die neue Objekt–ID, die neuen Seitenkoordinaten und der neue Text–Index nach der Cursorbewegung.

Beschreibung:

Diese Funktion führt die Suche nach der korrekten Cursorposition durch, die möglichst nahe an den angegebenen Seitenkoordinaten liegt. Sie überprüft dabei von der übergebenen Objekt–ID an alle nachfolgenden finalen Objekte. Wird keine entsprechende Position gefunden, tritt eine ‚mtObjID' an die zweite Stelle des Rückgabe–Tupels.

Implementierung:

Es wird überprüft, ob sich die Seitenkoordinaten in einer der Boxen des Objektes befinden und wenn ja, in welcher Line. Als Anhaltspunkt dient dabei die Y-Koordinate. Ist die Line gefunden, wird die korrekte Position durch `getNewPosInLines` ermittelt, indem dieselben Seitenkoordinaten allerdings mit einer um einen Pixel geringeren Y-Koordinate übergeben werden, und als Suchrichtung `down` angegeben wird. Durch diesen Trick findet `getNewPosInLines` die vorher herausgesuchte Line als neue Line für den Cursor und ermittelt die korrekte X-Position. Liegen die gewünschten Seitenkoordinaten vor oder nach dem Beginn der Line, sorgt die aufgerufene Funktion selbst dafür, daß der Anfang oder das Ende der Line als neue Position vorgeschlagen werden.

Ist im aktuellen Objekt keine passende Box oder Line zu finden, wird das nächste finale Objekt daraufhin getestet, ob es auch auf dieser Seite liegt und — wenn das der Fall ist — durch einen rekursiven Aufruf in diesem weitergesucht. War das aktuelle Objekt bereits das letzte auf der Seite, wird die Position hinter dem letzten Zeichen der letzten Box auf der Seite als neue Position zurückgeliefert.

Funktionsname:

setCursorToEndOfBox setCursorToEndOfBox

Signatur:

setCursorToEndOfBox:: env -> objID -> boxinfo -> (pagecoordinates, integer).

env: Das Environment.

objID: Die ID des Objektes, in dem die Box liegt.

boxinfo: Eine Box innerhalb des Objektes.

(pagecoordinates, integer): Die Position hinter dem letzten Zeichen der Box, sowie der Index dieses Zeichens.

Beschreibung:

Diese Funktion liefert die Position hinter dem letzten Zeichen der Box sowie den Index dieses Zeichens.

Implementierung:

Es werden die Wörter der letzten Line innerhalb der Box geholt und deren Breite berechnet.

Funktionsname:

determinateSubboxLoosely determinateSubboxLoosely

Signatur:

determinateSubboxLoosely:: integer -> boxinfo -> (boolean, boxinfo).

integer: Ein Y-Wert.

boxinfo: Eine Box.

(boolean, boxinfo): Flag, ob das Ende der Box unterhalb des Y-Wertes liegt, sowie die Box selbst.

Beschreibung:

Die Funktion gibt `true` zurück, wenn die Y-Koordinate des Box–Endes größer als der spezifizierte Y-Wert ist, sonst `false`.

Implementierung:

Offensichtlich.

Funktionsname:

getFirstNonEmptyLine getFirstNonEmptyLine

Signatur:

getFirstNonEmptyLine:: line -> (boolean, line).

line: Eine Line.

(boolean, line): Flag, ob es sich um eine leere Line gehandelt hat, oder nicht, sowie die übergebene Line selbst.

Beschreibung:

Diese Funktion wird im Zusammenhang mit einem ‚select' verwendet, um die erste Line innerhalb einer Box zu finden, die nicht leer ist. Sie liefert `false`, wenn die übergebene Line leer ist und `true`, wenn dies nicht der Fall ist.

Implementierung:

Entspricht die Line dem Wert von ‚mtLine', dann wird sofort `false` zurückgegeben, ansonsten wird geprüft, ob der Index des ersten und letzten Zeichens in dieser Line Null ist.

Funktionsname:

notequalLines notequalLines

Signatur:

notequalLines:: line -> line -> boolean.

line: Eine Line.

line: Eine Line.

boolean: Flag, ob die beiden Lines gleich sind, oder nicht.

Beschreibung:

Die Funktion liefert `true`, wenn die beiden Lines nicht identisch sind, ansonsten `false`. Sie wird als Testfunktion für ‚dropwhile' bzw. ‚takewhile' benutzt.

Implementierung:

Benutzt den eingebauten Test auf strukturelle Gleichheit.

Funktionsname:

notequalBoxes notequalBoxes

Signatur:

notequalBoxes:: boxinfo -> boxinfo -> boolean.

boxinfo: Eine Box.

boxinfo: Eine Box.

boolean: Flag, ob die beiden Boxen gleich sind, oder nicht.

Beschreibung:

Die Funktion liefert `true`, wenn die beiden Boxen nicht identisch sind, ansonsten `false`. Sie wird als Testfunktion für ‚dropwhile' bzw. ‚takewhile' benutzt.

Implementierung:

Benutzt den eingebauten Test auf strukturelle Gleichheit.

Funktionsname:

notequalObjIDs notequalObjIDs

Signatur:

notequalObjIDs:: objID -> objID -> boolean.

objID: Eine Objekt–ID.

objID: Eine Objekt–ID.

boolean: Flag, ob die beiden Objekt–IDs gleich sind, oder nicht.

Beschreibung:

Die Funktion liefert `true`, wenn die beiden IDs nicht identisch sind, ansonsten `false`. Sie wird als Testfunktion für ‚dropwhile' bzw. ‚takewhile' benutzt. Da die Zuordnung zwischen Objekten und ihren IDs eineindeutig ist, bedeutet dies gleichzeitig, daß es sich um dieselben Objekte handelt, wenn ein `false` geliefert wird.

Implementierung:

Benutzt den eingebauten Test auf strukturelle Gleichheit.

Funktionsname:

isBoxOnPage isBoxOnPage

Signatur:

isBoxOnPage:: pagenumber -> boxinfo -> boolean.

pagenumber: Eine Seitennummer.

boxinfo: Eine Box.

boolean: Flag, ob die Box auf der entsprechenden Seite liegt.

Beschreibung:

Überprüft, ob die Box auf der durch die übergebene Seitennummer spezifizierten Seite liegt. Ist dies der Fall, wird `true`, sonst `false` zurückgeliefert.

Implementierung:

Vergleicht die übergebene Seitennummer mit der in der Box gespeicherten.

Funktionsname:

isObjectPartOnPage isObjectPartOnPage

Signatur:

isObjectPartOnPage:: fs -> objID -> pagenumber -> boolean.

fs: Die Formatierer–Datenstruktur.

objID: Die ID des zu überprüfenden Objektes.

pagenumber: Eine Seitennummer.

boolean: Flag, ob ein Teil des Objektes auf der entsprechenden Seite liegt.

Beschreibung:

Die Funktion prüft, ob eine Box des übergebenen Objektes auf der spezifizierten Seite liegt; in diesem Fall liefert sie `true`, sonst das Gegenteil.

Implementierung:

Jede Box des Objektes wird mit Hilfe der Funktion `isBoxOnPage` überprüft.

Funktionsname:

isLeftOfIndex isLeftOfIndex

Signatur:

isLeftOfIndex:: integer -> wordlistelement -> boolean.

integer: Ein Index.

wordlistelement: Ein Wort.

boolean: Flag, ob die Indices aller Zeichen des Wortes kleiner oder höchstens gleich dem separat übergebenen sind.

Beschreibung:

Überpüft, ob die Indices aller Zeichen des Wortes kleiner oder höchstens gleich dem separat übergebenen sind. Dies bedeutet, daß das Wort links neben diesem Index liegt; dann wird `true` zurückgeliefert.

Implementierung:

Es wird der Index des letzten Buchstabens des Wortes auf die geforderte Eigenschaft hin überprüft.

Funktionsname:

makeError makeError

Signatur:

makeError:: env -> pagecoordinates -> env.

env: Das Environment.

pagecoordinates: Seitenkoordinaten.

env: Das neue Environment.

Beschreibung:

Die Funktion schreibt einen Fehler ins Environment, mit der Information, daß es auf der übergebenen Seite keine finalen Objekte gibt und deshalb der Cursor nicht auf ihr plaziert werden kann. Sie wird von `f_getCursorPos` benutzt.

2.12.5 Fehler und Restriktionen

Zwei größere Fehler sind im Modul vorhanden. Zum einen wird der Cursor falsch positioniert, wenn das Wort, in dem sich der Cursor befindet, beim Umbruch auf eine neue Zeile rutscht. In diesem Fall wandert der Cursor an den Anfang bzw. das Ende der betreffenden Zeile, was offensichtlich falsch ist. Dies dürfte an der Methode liegen, mit der die alte Cursorposition aufgefunden wird. Die alte Y–Position ist nach dem Umbruch nicht mehr gültig und dadurch wird die falsche Line herausgesucht. Dieser Fehler dürfte einigermaßen einfach zu beheben sein.
Die Quelle des zweiten Fehlers ist dagegen nicht ohne weiteres auszumachen: Wenn am Ende eines Objektes mit „Backspace“ das letzte Zeichen gelöscht wird, springt der Cursor zum Anfang des Objektes. Der Fehler kann sogar außerhalb des Moduls, im inkorrekten Aufruf liegen, doch das ist reine Spekulation.
Neben den Fehlern müßte auch noch die korrekte Implementierung des wortweisen Bewegens stattfinden. Sie war bereits ausprogrammiert und die letzten Änderungen waren nicht so schwerwiegend, daß die Anpassung zu umfangreich wäre.

2.13 FormatOutputTypes

2.13.1 Funktionalität

Im Modul „FormatOutputTypes“ werden die grundlegenden Typen für die Verwendung in den Moduln des Formatierers definiert. Ferner werden für fast alle hier definierten Objekte Initialisierungfunktionen zur Verfügung gestellt.

2.13.2 Entwurf

In diesem Modul werden die Typen für zwei zentrale Konzepte der Formatierer definiert:

1. Das „Dreiboxenkonzept“ (siehe dazu Kapitel 2.13.3.1), welches grundlegend von allen Formatierern dazu genutzt wird, Ausdehnungen von Textbereichen einzugrenzen.
2. Das `changes`-Konstrukt“ (siehe Kapitel 2.13.3.1), welches vor allem für die Ausgabe entwickelt worden ist, um auf effiziente Weise Bildschirmveränderungen zu beschreiben.

Beide Konzepte werden bei ihrer Definition weitergehend erläutert.

2.13.3 Öffentliche Schnittstellen

2.13.3.1 Typdefinitionen

Sortenname:

pagenumber pagenumber

Signatur:

pagenumber:: (integer)

Beschreibung:

Beschreibt die Seitenummer im Dokument.

Sortenname:

lineposition lineposition

Signatur:

lineposition:: (integer)

Beschreibung:

Dient zur Beschreibung der aktuellen Cursorposition innerhalb einer Zeile.

Sortenname:

direction direction

Signatur:

direction:: left — right — up — down

Beschreibung:

Der Aufzählungstyp `direction` beschreibt die möglichen Cursorbewegungen.

Sortenname:

stepwidth stepwidth

Signatur:

stepwidth:: single — word — page

Beschreibung:

Der Aufzählungstyp `stepwidth` definiert die Schrittweite, die in Verbindung mit Cursorbewegungen angewandt wird.

Sortenname:

attributes attributes

Signatur:

attributes:: attributes
(v_xattrib :: integer)
(v_yattrib :: integer)
(v_font :: font)

Beschreibung:

Der Konstruktor `attributes` beinhaltet drei Angaben zur Positionierung von Text:

- `v_xattrib` enthält die X-Postition auf der darzustellenden Seite
- `v_yattrib` enthält die Y-Postition auf der darzustellenden Seite
- `v_font` enthält die Schriftarteninformationen des darzustellenden Textes

---

Sortenname:

wordattributes wordattributes

Signatur:

wordattributes:: wordattributes
(words :: string)
(v_attributes :: attributes)

Beschreibung:

Der Konstruktor `wordattributes` beinhaltet alle Angaben, die benötigt werden, um einen Text an einer bestimmten Stelle im Dokument zu positionieren. Dazu werden folgende Werte gebraucht:

- `words` beschreibt den darzustellenden Text
- `v_attributes` verweist auf den o.g. Typ `attributes`

Sortenname:

wordattributelist wordattributelist

Signatur:

wordattributelist:: [(wordattributes)]

Beschreibung:

Der Typ `wordattributelist` definiert eine Liste von `wordattributes`.

Sortenname:

box box

Signatur:

box:: box
(startrestbox :: pagecoordinates)
(startpagebox :: pagecoordinates)
(endpagebox :: pagecoordinates)
(endrestbox :: pagecoordinates)

Beschreibung:

Der Konstruktor `box` definiert die Ausmaße eines Textbereichs, in der Form des „Dreiboxenkonzepts". Dies geht davon aus, daß ein Textblock eine Fläche benötigt, die sich durch drei Boxen beschreiben läßt. Die erste Box beschreibt den Anfang eines Textblocks, der u.U. nur einen Teil einer Zeile belegt. Die zweite beschreibt den Kernteil, der vollständige Textzeilen enthält. Das Ende eines Textes kann wiederum nur einen Teil einer Zeile belegen, deshalb ist die dritte Box erforderlich.
Enthält die erste oder dritte Box keine Textinformationen, so liegen die Boxgrenzen der betroffenen Box auf denen der Hauptbox.

- `startrestbox` definiert die linke obere Ecke des ersten Textblocks
- `startpagebox` definiert die linke obere Ecke des zweiten Textblocks
- `endpagebox` definiert die rechte untere Ecke des zweiten Textblocks
- `endrestbox` definiert die rechte untere Ecke des dritten Textblocks

Mithilfe dieser vier Koordinaten lassen sich die drei Boxen vollständig beschreiben.
Die erste Box enthält bereits den linken oberen Eckpunkt. Der rechte untere Eckpunkt kann aus dem X-Wert der rechten unteren Ecke des zweiten Textblocks und dem Y-Wert der linken oberen Ecke des zweiten Blocks zusammengesetzt werden. Die Seitennummer des zweiten Eckpunktes entspricht der Nummer, der linken oberen Ecke der ersten Box, da alle drei Boxen nur auf jeweils einer Seite liegen können.
Für die Bestimmung der zweiten Textbox können die Koordinaten `startpagebox` und `endpagebox` herangezogen werden. Die Berechnung der dritten Box erfolgt analog zum oben beschriebenen Verfahren für die erste Box.

Sortenname:

clear clear

Signatur:

clear:: clear
(block :: box)

Beschreibung:

Der Konstruktor `clear` beschreibt die Ausmaße des zu löschenden Bildschirmbereichs in Form des „Dreiboxenkonzepts“.

Sortenname:

insert insert

Signatur:

insert:: insert
(walist :: wordattributelist)

Beschreibung:

Der Konstruktor `insert` enthält den einzufügenden Textstrom inklusive der benötigten Attribute und Koordinaten.

Sortenname:

move move

Signatur:

move:: move
(block :: box)
(movement :: integer)

Beschreibung:

Der Konstruktor `move` beschreibt die Ausmaße des zu bewegenden Bildschirmbereichs in Form des „Dreiboxenkonzepts“. Der Wert `movement` repräsentiert dabei das Maß der Verschiebung in Pixeln, wobei ein positiver Wert einen Versatz nach unten und ein negativer eine Bewegung nach oben beschreibt.

Sortenname:

boxList boxList

Signatur:

boxList:: [box]

Beschreibung:

Der Typ definiert eine Liste von Boxen.

Sortenname:

changes changes

Signatur:

changes:: changes
(v_clear :: clear)
(v_insert :: insert)
(v_move :: move)

Beschreibung:

Das `changes`-Konstrukt dient der Ausgabe dazu, Bildschirmveränderungen effizient ausführen zu können. Nachdem ein Text verändert und neu formatiert wurde, werden folgende drei Angaben überliefert:

- `v_clear` enthält den zu löschenden Bildschirmbereich
- `v_insert` enthält den einzufügenden Zeichenstrom
- `v_move` enthält den zu verschiebenden Bereich

Sortenname:

pagesize pagesize

Signatur:

pagesize:: pagesize
(v_xsize::integer)
(v_ysize::integer)

Beschreibung:

Der Konstruktor `pagesize` enthält die maximale Ausdehnung der Seite.

- `v_xsize` enthält die X-Ausdehung der darzustellenden Seite
- `v_ysize` enthält die Y-Ausdehung der darzustellenden Seite

2.13.3.2 Funktionen

Funktionsname:

mtCoord mtCoord

Signatur:

mtCoord:: coordinates

Beschreibung:

Die Funktion `mtCoord` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtPageCoord mtPageCoord

Signatur:

mtPageCoord:: pagecoordinates

Beschreibung:

Die Funktion `mtPageCoord` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtCursorPos mtCursorPos

Signatur:

mtCursorPos:: pagecoordinates

Beschreibung:

Die Funktion `mtCursorPos` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtWordAttribute mtWordAttribute

Signatur:

mtWordAttribute:: wordattributes

Beschreibung:

Die Funktion `mtWordAttribute` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtWordAttributeList mtWordAttributeList

Signatur:

mtWordAttributeList:: wordattributelist

Beschreibung:

Die Funktion `mtWordAttributeList` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtObjID mtObjID

Signatur:

mtObjID:: objID

Beschreibung:

Die Funktion `mtObjID` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtObjIDList mtObjIDList

Signatur:

mtObjIDList:: objIDList

Beschreibung:

Die Funktion `mtObjIDList` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtBox mtBox

Signatur:

mtBox:: box

Beschreibung:

Die Funktion `mtBox` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtClear mtClear

Signatur:

mtClear:: clear

Beschreibung:

Die Funktion `mtClear` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtMove mtMove

Signatur:

mtMove:: move

Beschreibung:

Die Funktion `mtMove` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtInsert mtInsert

Signatur:

mtInsert:: insert

Beschreibung:

Die Funktion `mtInsert` initialisiert ein Objekt vom entsprechenden Typen.

Funktionsname:

mtChanges mtChanges

Signatur:

mtChanges:: changes

Beschreibung:

Die Funktion `mtChanges` initialisiert ein Objekt vom entsprechenden Typen.

2.14 Absatzformatierer

Tarik Ali, Andree Hähnel

2.14.1 Entwurf

Das Modul `SimpleFormatter` (Absatzformatierer) ist für das Formatieren von finalen Elementen zuständig. Dies wird bewerkstelligt, indem der Text, der zu dem jeweiligen Objekt gehört, unter Berücksichtigung der Absatz-, Zeichen- und Seitenattribute, wie Einrückungen, Schriftgröße und Textbreite, in Zeilen aufgeteilt wird.
Da der Text finaler Elemente von beliebiger Länge sein darf, bilden diese Zeilen einen Absatz der sich unter Umständen über mehrere Seiten erstrecken kann, aber auch nur aus einer Zeile bestehen könnte.
Der Seitenformatierer (PageFormatter) muß diese Absätze im Anschluß so umgebrochen, daß sie auf die Seiten des aktuellen Dokumentes passen.

2.14.2 Importschnittstelle

Der Absatzformatierer importiert Funktionen zum Zugriff auf die Formatiererstruktur aus dem Modul `FormatStructure`. Die Funktionen zum Zugriff auf die Layoutparameter und sämtliche anderen Daten aus der internen Struktur werden aus dem formatiererinternen Schnittstellenmodul `FormatSFInfo` importiert. Die für das Formatieren so wichtige Datenstruktur `wordlist` und die zugehörigen Methoden werden vom Modul `FormatTextConvert` bereitgestellt. Die den Zeichensatz betreffenden Funktionen und Datenstrukturen werden aus den Modulen `OutputFormat` und `FontTypes` importiert.

2.14.3 Exportschnittstelle

Funktionsname:

formatObject formatObject

Beschreibung:

Die Funktion `formatObject` ist die Schnittstelle zwischen dem Objekt- und dem Seitenformatierer. Hiermit wird das Formatieren eines finalen Elementes angestoßen.

Signatur:

formatObject:: env -> objID -> integer -> integer -> (env, lines)

env: die Umgebungsvariable hält jegliche Daten für die Formatierung bereit

objID: die Objektidentifikationsnummer des zu formatierenden finalen Elementes

integer: die Größe des Freiraums der in der ersten Zeile noch vorhanden ist (Absätze können auch mitten auf einer Zeile beginnen)

integer: die Nummer des Wortes im Absatz, bei dem mit der Formatierung begonnen werden soll env: die modifizierte Umgebungsvariable

lines: die formatierten Zeilen

functionimplementation Die Arbeitsweise der Funktion ist folgende. Zunächst werden die zur Formatierung notwendigen Daten des finalen Elementes wie der Text selbst und die Zeichen-, Absatz- und Seitenattribute aus der Umgebungsvariablen ausgelesen.
Im folgenden wird der Wert, der den Restfreiraum auf der aktuellen Zeile angibt, analysiert, denn dieser fungiert als Schalter und muß falls er gleich Null ist auf die aktuelle Textbreite gesetzt werden, da in diesem Falle die zu formatierende Zeile nicht an eine andere angehängt werden soll, sondern es sich um eine neue Zeile handelt, die die volle Zeilenbreite haben soll.
Dann überführen wir mit Hilfe der Funktion `makeWordlist` aus dem Modul `FormatTextConvert` den zum aktuellen Objekt gehörenden Zeichenstrom in eine `wordlist`. Diese Struktur erleichtert die Arbeit des Absatzformatierers erheblich, da hier unter anderem alle Worte mit ihrer Position relativ zum Absatzanfang in einer Liste enthalten sind. Die Wortliste ist die Grunlage der späteren Formatierung in den Unterfunktionen und wird nun noch mithilfe des vierten Parameters, der Nummer des Wortes im Absatz bei dem mit der Formatierung begonnen werden soll, verkürzt. (Bei der initialen Formatierung wird diese Nummer immer gleich eins sein.)
Zuletzt ruft `formatObject` die Unterfunktion `formatBox` mit den ermittelten Werten auf, um dann die von dieser Funktion ermittelten Zeilen des finalen Objektes als ihren eigenen Funktionswert zurückliefern.

2.14.4 lokale Funktionen

Funktionsname:

formatBox formatBox

Beschreibung:

Die rekursive Hilfsfunktion `formatBox` teilt eine Wortliste in Zeilen auf. Diese Zeilen werden hier auch gesetzt und die Liste dieser ermittelten Zeilen wird als Ergebnis zurückgeliefert.

Signatur:

formatBox:: env -> lines -> wordlist -> integer -> integer -> integer -> integer -> font -> integer -> integer -> (env, lines)

env: die Umgebungsvariable

lines: die schon formatierten Zeilen

wordlist: die zu bearbeitende Wortliste

integer: die Textbreite

integer: die Zeilenhöhe

integer: der linke Einzug

integer: der rechte Einzug

font: der aktuelle Zeichensatz

integer: die laufende Nummer der aktuellen Zeile

integer: der in der ersten Zeile noch zur Verfügung stehende Freiraum

env: die modifizierte Umgebungsvariable

lines: die formatierten Zeilen

Implementierung:

Zunächst wird mithilfe der Unterfunktion `formatLine` ein Sechs-Tupel berechnet. Dieses Tupel representiert die gerade formatierte Zeile, denn es enthält die Position (rel. im Absatz) des ersten Wortes der Zeile, die Position (rel. im Absatz) des letzten Wortes der Zeile und den Freiraum der hinter dem letzten Wort bis zum Zeilenende noch vorhanden ist (Emptyspace). Außerdem ist noch die Restliste der Worte im Absatz, die noch formatiert werden müssen und eine neue Umgebungsvariable im Tupel enthalten. Der letzte Wert des Tupels sollte den Wert, der auf jedes Leerzeichen der Zeile aufgerechnet werden muß, um Blocksatz zu erhalten, enthalten (Spacefactor). Hiervon wurde aber Abstand genommen, da dies mit dem Konzept von ïn einer Zeile ineinandergreifenden Absätzenüicht harmonierte. Somit wird hier zunächst jede Zeile nur Linksbündig formatiert.
Mithilfe der Vorgaben für die Zeile, wie linker Einzug, Zeilenabstand, Zeilennummer usw. wird nun, durch die Hilfsfunktion `calcLineCoo`, die Anfangskoordinate (linke untere Ecke des ersten Wortes) der Zeile berechnet. Diese Koordinate gilt natürlich nur bei linksbündigem Satz, für zentrierten und rechtsbündigen Satz muß sie vom Ausrichtungsformatierer noch angepaßt werden.
Die für die Zeile ermittelten Werte werden nun von der Funktion `setLine` in Instanz des Datentyps `line` geschrieben, die so entstandene Zeile wird an die vorher schon ermittelten Zeilen angehängt und die Funktion `formatBox` ruft sich selbst mit den der noch zu formatierenden Restwortliste auf, um die nächste Zeile zu ermitteln.
Ist die Wortliste irgendwann leer, werden die ermittelten Zeilen als Ergebnis zurückgeliefert.

Funktionsname:

setLine setLine

Beschreibung:

Die Funktion `setLine` ist eine einfach Hilfsfunktion, die ihre Eingabeparameter in eine Zeile schreibt und diese dann zurückliefert.

Signatur:

setLine:: coordinates -> integer -> integer -> integer -> integer -> line

coordinates: die Anfangskoordinaten der Zeile

integer: die Position des ersten Wortes der Zeile relativ im Absatz

integer: die Position des letzten Wortes der Zeile relativ im Absatz integer: der Restfreiraum in der Zeile

integer: der Spacefactor

line: die gesetzte Zeile

Implementierung:

Zunächst wird eine leere Instanz des Datentyps `line` erzeugt. Im Anschluß werden, mithilfe der jeweiligen Zugriffsfunktionen aus dem Modul `FormatStructure`, die Attribute in diese anfangs leere Zeile geschrieben. Die so aufgefüllte Zeile wird am Ende zurückgegeben.

Funktionsname:

calcLineCoo calcLineCoo

Beschreibung:

Die Hilfsfunktion `calcLineCoo` berechnet die Koordinate des ersten Wortes einer Zeile, die sich links unten auf der Grundlinie dieser Zeile befindet.

Signatur:

calcLineCoo:: integer -> integer -> integer -> integer -> coordinates

integer: die Zeilenhöhe der Zeile

integer: der linke Einzug der Zeile

integer: die Zeilennummer im Absatz

integer: die schon von einem anderen finalen Element in dieser Zeile belegte Breite

coordinates: die berechneten Zeilenkoordinaten

Implementierung:

Anhand der gegebenen Layoutparameter wird die Koordinate der Zeile relativ zur linken oberen Ecke der Absatzbox berechnet (Koordinate (0,0)).
Die vertikale Y-Koordinate ergibt sich aus der Zeilenhöhe multipliziert mit der Zeilennummer. Dies ist nicht ganz korrekt, den eigendlich sollte der Zeilenabstand durch den Zeichensatz vorgegeben sein. Wir benutzen hier den Durchschuß (Baselineskip) als Zeilenhöhe und überlassen so dem Benutzer die Verantwortung darüber, daß der Durchschuß, also die Zeilenhöhe, zur gewählten Zeichengröße paßt (siehe Presentationsregeln).
Die horizontale X-Koordinate ergibt sich aus der Addition von linkem Einzug und schon belegtem Raum in der Zeile.Diese beiden Werte werden zu einer Koordinate zusammengefaßt.
Der schon belegte Raum in der Zeile kann nur in der ersten Zeile eines finalen Elementes groesser als Null sein.

Funktionsname:

formatLine formatLine

Beschreibung:

Die Funktion `formatLine` wird von der Funktion `formatBox` aufgerufen und formatiert genau eine Zeile.

Signatur:

formatLine:: env -> wordlist -> integer -> font -> integer -> (env, integer, integer, integer, wordlist, integer)

env: die Umgebungsvariable

wordlist: die zu formatierende Wortliste

integer: die Textbreite

font: der aktuelle Zeichensatz

integer: der Restfreiraum in der aktuellen Zeile

env: die modifizierte Umgebungsvariable

integer: die Nummer des ersten Wortes der Zeile

integer: die Nummer des letzten Wortes der Zeile

integer: der Restfreiraum in der aktuellen Zeile

wordlist: die restlichen Worte des Absatzes

integer: der SpaceFactor

Implementierung:

Aus der Wortliste wird zunächst das erste Wort abgetrennt.
Nun gibt es zwei Möglichkeiten um fortzufahren. Handelt es sich um eine neue Zeile, soll also auf die gesamte Textbreite formatiert werden, so wird zunächst mithilfe der Funktion `o_getTextDims` aus dem Modul `Output` die Breite des ersten Wortes bestimmt und abgeschätzt, ob es in die Zeile paßt. Paßt es nicht in die Zeile wird ein Fehler erzeugt (noch nicht implementiert). Anderenfalls muß überprüft werden ob es sich vielleicht um das letzte Element der Wortliste, also um das letzte Wort im finalen Element handelt. Dann besteht die aktuelle Zeile nur aus diesem Wort und es werden die Position dieses Wortes im Absatz als Anfangs- und Endposition der Zeile, die Textbreite verringer um die Wortlänge als Restfreiraum der Zeile und die leere Wortliste an die übergeordnete Funktion zurückgeliefert.
- Ist die Wortliste noch nicht leer so wird die Hilfsfunktion `getMoreWords` aufgerufen, die dann die nächsten Worte der Wortliste in die aktuelle Zeile formatiert bis die Zeile voll ist. Das Rückgabetupel von `getMoreWords` beinhaltet dann wiederum die Restwortliste, denn es können ja immernoch zu formatierende Worte übrig sein, die Position des Wortes im Absatz, das als letztes in die Zeile Formatiert wurde, und den Restfreiraum in der Zeile. Diese Werte werden dann in das Rückgabetupel von `formatLine` übernommen und zusammen mit der Position des ersten Wortes der Zeile (im Absatz) an die übergeordnete Funktion

`formatBox` zurückgegeben.
Die zweite Möglichkeit um fortzufahren tritt in Kraft, wenn in die aktuelle Zeile schon Worte eines anderen finalen Elementes formatiert wurden und das aktuelle finale Element direkt an das vorherige angehängt werden soll. In diesem Fall muß ein Leerzeichen vor das erste Wort gesetzt werden, da sonst das letzte Wort des voherigen finalen Elementes ohne Zwischenraum vor dem ersten Wort des aktuellen finalen Elementes stehen würde. Die nachfolgende Formatierung der Zeile erfolgt ansonsten wie in Möglichkeit eins.

Funktionsname:

getMoreWords getMoreWords

Beschreibung:

Die rekursive Hilfsfunktion `getMoreWords` wird von der Funktion `formatLine` aufgerufen. `formatLine` formatiert jeweils das erste Wort (normalerweise ohne Leerzeichen s.o.) einer Zeile selbst und läßt dann den Rest der Zeile von `getMoreWords` (mit Leerzeichen) formatieren.

Signatur:

getMoreWords:: env -> wordlist -> integer -> font -> (env, wordlist, integer, integer)

env: die Umgebungsvariable

wordlist: die zu formatierende Wortliste

integer: die restliche Textbreite in der aktuellen Zeile

font: der aktuelle Zeichensatz

env: die modifizierte Umgebungsvariable

wordlist: die restlichen Worte des Absatzes

integer: die Nummer des letzten Wortes der Zeile

integer: der Restfreiraum in der aktuellen Zeile

Implementierung:

Das erste Element der Wortliste wird ausgelesen. Dann wird vor dieses Wort ein Leerzeichen gesetzt und die Breite des Wortes mit Leerzeichen bestimmt. Nun entscheidet der Vergleich dieser Breite mit der restlichen Textbreite der Zeile darüber, ob das Wort noch in die Zeile paßt oder nicht. Paßt das Wort nicht mehr in die Zeile so wird es wieder vor die Wortliste gehängt und die Position des vorherigen Wortes, das also das letzte Wort in dieser Zeile ist, und der Restfreiraum in dieser Zeile zurückgeliefert. - Paßt das Wort noch in die Zeile, so ruft sich `getMoreWords` mit der nicht leeren Restwortliste, der neuen Resttextbreite, die die Differenz der alten Resttextbreite und der Breite von Leerzeichen und Wort ist, und dem gleichen Zeichensatz selbst auf. Ist die Restwordliste doch leer, so handelt es sich um das letzte Wort im finalen Element und die Position dieses Wortes im Absatz und die neue Restbreite werden zurückgegeben.

Funktionsname:

checkdiv checkdiv

Beschreibung:

Die Funktion `checkdiv` ist eine Hilfsfunktion die das Dividieren zweier ganzer Zahlen durchführt und hierbei darauf achtet, daß nicht durch Null geteilt wird.

Signatur:

checkdiv:: integer -> integer -> integer

integer: der Zähler

integer: der Nenner

integer: die differenz

Implementierung:

Diese Funktion liefert jeweils die Differenz der beiden Eingabeparameter zurück. Ist der zweite Parameter gleich Null wird eins zurückgeliefert.

2.15 Seitenformatierer

Tarik Ali, Andree Hähnel

2.15.1 Importschnittstelle

Der Seitenformatierer importiert Funktionen zum Formatieren und Ausrichten von Absätzen aus den Modulen `SimpleFormatter` und `FormatAlignment`. Die Layoutparameter werden für ihn im Modul `FormatSFInfo` umgesetzt.

2.15.2 Exportschnittstelle

Funktionsname:

formatPage formatPage

Beschreibung:

Die Funktion `formatPage` ist die Funktion zum initialen Formatieren von allen finalen Elementen. Ihr Name ist somit sehr schlecht gewählt. `formatDocument` wäre ein prägnanterer Name.

Signatur:

formatPage:: env -> fs -> (env, fs)

env: akluelle Umgebungsvariable

fs: Formatiererstruktur

env: modifizierte Umgebungsvariable

fs: modifizierte Formatiererstruktur

Implementierung:

Zunächst werden alle finalen Elemente des gesamten Dokumentes ausgelesen. Ist die Liste dieser Objekte leer, so liegt ein Fehler vor und die Funktion wird abgebrochen. Anderenfalls wird ein leeres Pagelayout erzeugt und die rekursiv arbeitende Hilfsfunktion `arrangeFinalElements`, die das wirkliche Formatieren (bis auf Ausrichtung) der finalen Elemente durchführt, aufgerufen.
Aus der Instanz des letzten finalen Elementes wird die höchste im Dokument vorkommende Seitenzahl ausgelesen. Mit dieser Seitenzahl wird nun die Ausrichtung des gesamten Dokumentes angestoßen. Die sich hieraus ergebende Umgebungsvariable und Formatiererstruktur werden als Ergebnis zurückgeliefert.

Funktionsname:

reformatObject reformatObject

Beschreibung:

Die Funktion `reformatObject` ist neben `reformatLine` eine Funktion zum Reformatieren von modifizierten Objekten. Im Gegensatz zu letzterer Funktion, die lediglich in einer Zeile reformatiert, reformatiert `reformatObject` ab einem angegebenen finalen Element bis zum Seitenende.
Um eine schnellere Ausgabe des reformatierten Textes zu ermöglichen werden auch die Liste der reformatierten Zeilen (bzw. Worte bei Blocksatz) mit ihren Koordinaten sowie der zu löschende und zu verschiebende Bildschirmbereich berechnet. Diese können dann direkt an die Äusgabeẅeitergereicht und von dieser interpretiert und angezeigt werden.

Signatur:

reformatObject:: env -> fs -> objID -> integer -> (env, wordattributelist, clear, move, objIDList, objIDList)

env: die aktuelle Umgebungsvariable

fs: die aktuelle Formatiererstruktur

objID: Identifikationsnummer des finalen Elementes ab dem reformatiert werden soll

integer: aktuelle Seite, bis zu derern Ende reformatiert werden soll

env: modifizierte Umgebungsvariable

wordattributelist: Liste der reformatierten Zeilen (bzw. Worte bei Blocksatz) mit ihren Koordinaten

clear: auf dem Bildschirm zu löschender Bereich

move: auf dem Bildschirm zu verschiebender Bereich

objIDList: reformatierte Objekte

objIDList: nicht formatierte restliche Objekte des Dokumentes (das Dokument ist ab dieser Seite nicht mehr formatiert)

Vor- und Nachbedingungen:

Das Objekt (finale Element) mit dem diese Funktion zum reformatieren aufgerufen wird, dient u.U. als Ansatzpunkt auf der Seite und darf nicht modifiziert worden sein. Es muß also im Text des Dokumentes vor dem finalen Element stehen in dem es eine Veränderung gegeben hat, denn eine Veränderung kann auch das Löschen eines Elementes sein. Wenn dann das gelöschte Objekt als Ansatzpunkt genommen würde wären die entsprechenden Daten ebenfalls nicht mehr vorhanden und das System geriete in einen inkonsistenten Zustand.
Nach Beendigung der Funktion ist die aktuelle Seite sowie die erste Box der nächsten Seite formatiert (diese Box ist bis auf Ausrichtung formatiert). Alle weiteren finalen Elemente die noch hinter dieser Box im Textfluß des Dokumentes vorkommen sind ab jetzt nicht formatiert.

Abhängigkeiten:

Diese Funktion wird im Modul `FormatOutput`, der Schnittstelle zwischen Formatierer und Ausgabe, benutzt.

Implementierung:

Zunächst werden mit Hilfe der Funktion `getRelatedBoxInfos` diejenigen finalen Elemente ermittelt, die mit dem eingegebenen Objekt in Beziehung stehen. Diese Objekte zusammengenommen ergeben einen Absatz, d.h. da jeweils ein nächstes Objekt direkt hinter dem vorherigen Objekt ansetzt (also in der gleichen Zeile) und nicht in einer neuen Zeile.
Dies ist nötig, weil für in Beziehung stehende Objekte immer die Ausrichtung (links, rechts, Blocksatz) des ersten dieser Objekte gilt, weil es in einem Absatz natürlich nur eine Ausrichtung geben kann. Die Ausrichtungen der anderen finalen Elemente dieser Gruppe (die den Absatz bildet) werden unterdrückt. Man muß also auch beim reformatieren die Ausrichtung des ersten Objektes der Gruppe verwenden, deswegen wird dieses Objekt nun als Ansatzpunkt zum reformatieren benutzt (Spezialfall: das eingegebene Objekt ist gerade das erste einer solchen Gruppe).
Falls die Nummer des ersten Elementes dieser Gruppe gleich Null ist liegt ein Fehler vor und die Funktion wird abgebrochen. Anderenfalls kann reformatiert werden.
Aus der Box des Ansatzobjektes werden nun die alten Zeilen und aus diesen dann die Nummer des ersten zu reformatierenden Wortes ausgelesen. Dies ist nötig, weil Textteile dieses finalen Elementes (Ansatzobjekt) auf der vorherigen Seite stehen können und nicht reformatiert werden müssen.
Nun wird mit Hilfe der Funktion `getObjsWithObjID` die Liste aller finalen Elemente des Dokumentes ab dem Ansatzobjekt ermittelt. Dies sind dann die Objekte die reformatiert werden, bzw wenn das Seitenende erreicht wurde auf n̈icht formatiertg̈esetzt werden.
Des weiteren werden einige Layoutattribute des Ansatzobjektes ausgelesen, die einerseits beim Aufruf der Hilfsfunktion zum reformatieren `arrangeChangedObs` benötigt werden und andererseits das Ermitteln der Lösch- und Verschiebebereiche unterstützen.
Aus der alten (irgendwann vorher formatierten) Box des Ansatzobjektes wird die Anfangskoordinate, die sich nicht verändert haben kann, ausgelesen. Mit dieser Koordinate und der Texthöhe wird zunächst die Resttexthöhe auf der aktuellen Seite ermittelt, damit bekannt ist wieviel vertikaler Freiraum zum Reformatieren auf dieser Seite zur Verfügung steht. Außerdem wird die Anfangskoordinate als Endkoordinate in eine leere Box geschrieben, die dann ebenfalls ein Eingabeparameter der Funktion `arrangeChangedObs` ist. Dies ist etwas umständlich, aber diese rekursive Funktion holt sich beim reformatieren einer neuen Box die Anfangskoordinaten aus den Endkoordinaten der gerade vorher reformatierten Box. Soll also auch die erste Box richtig reformatiert werden muß die vorhergehende Box also simuliert und mit der richtigen Endkoordinate ausgestattet werden.
Nun muß noch der Freiraum der hinter dem vorherigen finalen Element zu setzen ist ausgelesen werden, denn dieser wird zusammen mit dem Freiraum der vor das aktuelle finale Element gehört gesetzt.
Das wirkliche Reformatieren, also das Aufteilen der entsprechenden finalen Elemente auf die Seite übernimmt die Funktion `arrangeChangedObs`. Sie wird mit den richtigen Parametern angesteuert und liefert die Liste der reformatierten Objekte, die Liste der nicht formatierten Obkjekte, die modifizierte Umgebungsvariable und die neue Formatiererstruktur zurück.
Mit Hilfe der Funktion `setFinalsNotFormatted` werden der Status der nicht mehr formatierten finalen Elemente auf `nonformatted` gesetzt.
Die Funktion `setPartlyPageAlignment` aus dem Modul `FormatAlignment` hat die Aufgaben die gerade reformatierten Objekte auszurichten und die `wordattributelist`, also die Liste der reformatierten Zeilen (bzw. Worte bei Blocksatz) mit ihren Koordinaten, für diese Objekte zu ermitteln (Dies sind also gerade die Zeilen auf der aktuellen Seite, die sich verändert haben).

Nun muß noch der zu löschende Bereich auf dem Bildschirm ermittelt werden, damit die gerade reformatierten Zeilen angezeigt werden können und es nicht zu überschneidungen mit den vorher auf dem Bildschirm befindlichen Zeilen kommt.
Da im Moment noch immer bis zum Ende der Seite reformatiert wird gibt es noch keinen Bereich zum Verschieben. Es wird immer der leere Verschiebebereich zurückgeliefert.
Das Ergebnistupel dieser Funktion besteht aus der manipulierten Umgebungsvariable, der neuen Liste von Zeilen und Koordinaten, dem Lösch - und dem Verschiebebereich und den reformatierten und nicht formatierten finalen Elementen.

Funktionsname:

reformatLine reformatLine

Beschreibung:

Die Funktion `reformatLine` ist für das inkrementelle, also auf den vorher formatierten Werten aufsetzendes, reformatieren einer Zeile zuständig. Dies geht nur solange wie keine zusätzlichen Worte in die Zeile eingefügt werden und die einzufügenden Buchstaben noch in die Zeile passen. Anderenfalls wird sofort die Funktion zum Reformatieren ganzer finaler Elemente (`reformatObject`) aufgerufen.
Auch diese Funktion liefert, um eine schnellere Ausgabe zu ermöglichen, direkt die Liste der reformatierten Worte und deren Koordinaten, sowie den Lösch- und den Verschiebebereich auf dem Bildschirm.

Signatur:

reformatLine:: env -> fs -> objID -> boxinfo -> line -> string -> (env, wordattributelist, clear, move, objIDList, objIDList)

env: eingegebene Umgebungsvariable

fs: eingegebene Formatiererstruktur

objID: Identifikationsnummer des aktuellen finalen Elementes

boxinfo: die aktuelle Box

line: die zu verändernde Zeile

string: der veränderte Zeichenstrom

env: die modifizierte Umgebungsvariable

wordattributelist: die Liste der reformatierten Worte und deren Koordinaten

clear: der zu löschende Bildschirmbereich

move: der zu verschiebende Bildschirmbereich

objIDList: die reformatierten Objekte

objIDList: die nicht formatierten Objekte

Implementierung:

Zunächst wird aus der aktuellen Box, die die veränderte Zeile beinhaltet, die aktuelle Seitennummer ausgelesen.
Der eingegebene Zeichenstrom, der die Veränderungen in der Zeile beinhaltet, wird dahingehend überprüft, ob er Trennzeichen enthält (neues Wort) oder leer ist (bei Löschaktionen). In diesen beiden Fällen kann diese Funktion das Reformatieren nicht übernehmen und die Funktion `reformatObject` wird aufgerufen.
Anderenfalls werden die mit dem aktuellen Objekt in Beziehung stehenden Objekte (die mit ihm einen Absatz bilden) mit Hilfe der Funktion `getRelatedBoxInfos` bestimmt. Diese Objekte haben an ihren Schnittstellen jeweils gleiche Zeilen. Die Reformatierung dieser Zeilen kann ebenfalls nicht von dieser Funktion geleistet werden ,da die Veränderungen in einer Zeile eines Objektes auch Veränderungen in einem anderen Objekt bedingen können. Diese Zusammenhänge sind aber nicht ohne weiteres absehbar. Deshalb werden diese Zeilen herausgefiltert und für sie die Funktion `reformatObject` aufgerufen.
Jetzt kann das eigendliche Reformatieren beginnen. - Die Ausrichtung in der die aktuelle Zeile gesetzt werden muß wird ausgelesen (immer die Ausrichtung des ersten Elementes im Absatz). Mit Hilfe der Fontinformationen des aktuellen Objektes werden die Höhe und die Breite des eingegebenen Zeichenstroms bestimmt. Diese Parameter werden in die Funktion `reformatLineAlign` aus dem Modul `FormatAlignment`, die das wirkliche Reformatieren der Zeile unter Berücksichtigung der entsprechenden Ausrichtung durchführt, eingegeben. Deren Rückgabeparameter sind ein Wahrheitswert und eine Zeile. Ist der Wahrheitswert gleich `false`, so war der Eingabezeichenstrom zu groß für die Zeile und es muß ein neuer Zeilenumbruch stattfinden. Zu diesem Zweck wird wiederum die Funktion `reformatObject` aufgerufen. - Ist der Wahrheitswert gleich `true` so konnte das Reformatieren erfolgreich durchgeführt werden und der zweite Parameter enthält die neue Zeile. Diese Zeile wird nun anstelle der alten in die Box eingetragen. Die so entstandene neue Box wird in die Boxliste des finalen Elementes eingeordnet und diese dann in die Instanz des finalen Elementes geschrieben.
Die Liste der reformatierten Worte der Zeile und deren Koordinaten wird durch den Aufruf der Funktion `wordReformattedLine` aus dem Modul `FormatAlignment` bestimmt.
Nun werden noch Layoutattribute des aktuellen Objektes und Koordinate der Box und der Zeile ausgelesen, um den Lösch und den Verschiebebereich auf dem Bildschirm berechnen zu können. Diese beiden Bereiche werden ermittelt.
Als Funktionsergebnis wird das Tupel aus manipulierter Umgebungsvariable, der Liste der reformatierten Worte der Zeile und deren Koordinaten, dem Lösch- und dem Verschiebebereich zurückgeliefert.

Funktionsname:

objIDs objIDs

Beschreibung:

Die Funktion `objIDs` ist eine einfache Konvertierungsfunktion, die aus einer Liste von ganzen Zahlen eine Liste von Objekten macht. Diese Funktion wurde nötig, weil der Compiler eine Liste ganzer Zahlen nicht als Objektliste erkennt, obwohl der Typ `objIDList` als Liste ganzer Zahlen definiert ist.

Signatur:

objIDs:: integers -> objIDList

integers: eingegebene Liste ganzer Zahlen

objIDList: ausgegebene Liste von Objekten

Implementierung:

Die Liste ganzer Zahlen wird rekursiv Element für Element abgebaut und als Objektliste wieder aufgebaut.

Funktionsname:

checkdivbase checkdivbase

Beschreibung:

Die Hilfsfunktion `checkdivbase` dividiert zwei ganze Zahlen. Hierbei wird überprüft, ob der Teiler den Wert Null hat. In diesem Fall wird nicht dividiert, sondern Vereinbarung der Wert eins zurückgeliefert. Anderenfalls wird der Quotient der beiden Zahen zurückgeliefert.

2.15.3 lokale Funktionen

Signatur:

checkdivbase:: integer -> integer -> integer

integer: Zähler

integer: Nenner

integer: Ergebnis der Division

Implementierung:

Es wird überprüft, ob der zweite Eingabeparameter den Wert Null hat. Trifft dies zu wird der Wert eins als Funktionswert zurückgegeben. Anderenfalls werden die beiden Argumente mittels des Operators / aus dem Modul `integer` dividiert, und der sich hieraus ergebende Wert wird zurückgeliefert.

Funktionsname:

fitLines fitLines

Beschreibung:

Die Funktion `fitLines` trennt vom Anfang einer Liste von Zeilen eine bestimmte Anzahl von Zeilen ab. Der Rückgabewert dieser Funktion ist ein Tupel, das sich aus zwei Listen von Zeilen zusammensetz. Die erste Liste des Tupels enthält die abgetrennten Zeilen, und die zweite Liste des Tupels enthält die restlichen Zeilen. Diese Hilfsfunktion wird von der Funktion `fromLinestoBoxes` benutzt, um die Zeilen zu bestimmen, die in einen bestimmten Bereich noch noch hineinpassen.

Signatur:

fitLines:: lines -> integer -> (lines, lines)

lines: die zu bearbeitende List von Zeilen

integer: Anzahl abzutrennender Zeilen

lines: die abgetrennten Zeilen

lines: die restlichen Zeilen

Vor- und Nachbedingungen:

Die beiden Ergebniszeilen konkateniert ergeben in jedem Falle wieder die Eingabeliste.

Implementierung:

Ist die Anzahl der abzutrennenden Zeilen schon zu Begin größer als die Länge der Liste von Zeilen, gibt es also gar nicht so viele Zeilen wie abgetrennt werden sollen, so wird die gesamte Liste von Zeilen als abgetrennte Zeilen und die leere Liste als Restzeilen zurückgeliefert. Anderenfalls wird die rekursive Hilfsfunktion `fitLinesHelp` mit der leeren Liste, der zu bearbeitenden Liste und der Anzahl der abzutrennenden Zeilen als Argumente aufgerufen. Diese Funktion führt dann gerade das Abtrennen der Zeilen durch und liefert als Ergebnis das Tupel der beiden Listen von Zeilen (abgetrennt,Rest).

Funktionsname:

fitLinesHelp fitLinesHelp

Beschreibung:

Die rekursive Hilfsfunktion `fitLinesHelp` unterstützt die Funktion `fitLines` bei ihrer Arbeit. Sie ist für das wirkliche Abtrennen von Zeilen aus einer Liste von Zeilen zuständig und liefert ein Tupel von abgetrennten - und restlichen Zeilen zurück.

Signatur:

fitLinesHelp:: lines -> lines -> integer -> (lines, lines)

lines: Liste von Zeilen zur Ablage abgetrennter Zeilen

lines: zu bearbeitende Liste

integer: Anzahl der abzutrennenden Zeilen

lines: die abgetrennten Zeilen

lines: die restlichen Zeilen

Vor- und Nachbedingungen:

Beim initialen Aufruf der Funktion sollte das erste Argument die leere Liste von Zeilen sein, da diese Funktion hier (im ersten Argument) die Liste der abgetrennten Zeilen aufbaut und somit alle sich hierin vorher schon befindlichen Zeilen auch als abgetrennte Zeilen zurückgeliefert würden.

Implementierung:

Solange noch Zeilen abzutrennen sind werden diese aus der zu bearbeitenden Liste abgetrennt und an das erste Argument, also die schon abgetrennten Zeilen angehängt. Die Zahl abzutrennender Zeilen wird jeweils um eins verringert. Die Funktion ruft sich mit den neuen Argumenten selbst wieder auf.
Die Rekursion bricht in zwei Fällen ab. Erstens: Wenn die zu bearbeitende Liste leer ist werden die schon abgetrennten Zeilen und die leere Liste in einem Tupel zurückgeliefert. Dies kann nicht vorkommen, da in diesem Fall die Funktion `fitLines` die Hilfsfunktion `fitLinesHelp` gar nicht aufgerufen hätte. Zweitens: Wenn die Anzahl abzutrennender Zeilen den Wert Null hat, also keine Zeilen mehr abgetrennt werden sollen, wird das Tupel der abgetrennten Zeilen und der Restzeilen als Ergebnis zurückgegeben.

Funktionsname:

setCorrectYCoords setCorrectYCoords

Beschreibung:

Die rekursive Hilfsfunktion `setCorrectYCoords` wird von der Funktion `fromLinesToBoxes` aufgerufen. Sie hat die Aufgabe die Zeilenkoordinaten einer Liste von Zeilen neu zu setzen. Dies ist nötig, weil die Funktion `fromLinesToBoxes` die Liste von Zeilen, die zu einem finalen Element gehören, in Subboxen aufteilt. D.h. die Zeilenkoordinaten müssen jeweils so gesetzt werden, da sie relativ zu dem Ursprung (0,0) der jeweiligne Subbox sind, zu der sie gehören. Diese Funktion manipuliert nur die vertikalen Y-Koordinaten, die horizontalen X-Koordinaten bleiben unverändert

Signatur:

setCorrectYCoords:: lines -> integer -> integer -> integer -> lines

lines: die zu manipulierenden Zeilen

integer: vertikaler Freiraum vor dem Absatz

integer: Zeilenhöhe

integer: Anzahl der schon in diesem Absatz befindlichen Zeilen

lines: die manipulierten Zeilen

Vor- und Nachbedingungen:

Beim initialen Aufruf sollte das letzte Argument der Funktion, also der Zeilenzähler, den Wert eins haben. Dies ist wichtig, damit die Funktion die erste Zeile des Absatzes auch als diese erkennt.

Implementierung:

Falls die zu bearbeitende Liste von Zeilen leer ist, wird die leere Liste als Funktionswert zurückgeliefert. Anderendfalls wird die alte Koordinate der aktuell ersten Zeile ausgelesen. Der X-Wert dieser Koordinate wird bestimmt. Im Anschluß wird aus diesem X-Wert und dem neuen Y-Wert, der sich aus der Zeilenhöhe multipliziert mit der Zeilennummer im Absatz und dem vertikalen Freiraum zusammensetzt, die neue Koordinate der Zeile gebildet. Diese neue Koordinate wird dann noch in die Zeile geschrieben. Zum Schluß wird diese neue Zeile in die Rückgabeliste eingehängt und für den Rest der noch zu bearbeitenden Zeilen ruft sich die Funktion mit dem um eins erhöhten Zeilenzähler wieder auf.

Funktionsname:

setNewBoxInfo setNewBoxInfo

Beschreibung:

Die Hilfsfunktion `setNewBoxInfo` wird von der Funktion `fromLinesToBoxes` benutzt. Diese Funktion hat die Aufgabe eine Box vom Typ `boxinfo` zu setzen. Hierbei wird nichts berechnet, alle zu setzenden Werte stehen schon vorher fest und werden beim Aufruf übergeben.

Signatur:

setNewBoxInfo:: coordinates -> coordinates -> coordinates -> coordinates -> integer -> lines -> formatstatus -> boxinfo

coordinates: die Startbox-Koordinate

coordinates: die Reststartbox-Koordinate

coordinates: die Endbox-Koordinate

coordinates: die Restendbox-Koordinate

integer: aktuelle Seite

lines: die zur zu setzenden Box gehörenden Zeilen

formatstatus: Formatstatus der zu setzenden Box

boxinfo: die gesetzte Box

Implementierung:

Zunächst wird mit Hilfe der generischen Funktion `mtBoxInfo` aus dem Modul `Format-Structure` eine leere Instanz des Datentyps `boxinfo`, der Boxen repräsentiert, erzeugt. Anschlie end wird für jedes Attribut dieser Sorte die Methode (aus dem Modul `Format-Structure`) zum Eintragen des Attributes in eine Instanz dieses Datentyps aufgerufen und jeweils der bergabewert (aus den Argumenten) dieses Attributes in die Box eingetragen. Die so entstandene Box wird dann zurückgeliefert.

Funktionsname:

fromLinesToBoxes fromLinesToBoxes

Beschreibung:

Die Funktion `fromLinesToBoxes` ist für das Aufteilen des Zeilenstroms eines finalen Elementes in verschiedene Subboxen, die dann auf hintereinanderfolgenden Seiten liegen, verantwortlich. Hierbei werden die durch den Typ des finalen Elementes vorgegebenen Seiten- und Absatzattribute berücksichtigt.
Diese für das Seitenformatieren wichtige Funktion wird von verschiedenen Funktionen dieses Moduls, die ihrerseits Hilfsfunktionen für das initiale und das inkrementelle Formatieren sind, wie z.B. `arrangeFinalElements` und `arrangeChangedObs`, benutzt.

Signatur:

fromLinesToBoxes:: lines -> integer -> pagelayout -> integer -> integer -> integer -> boxinfo -> paragraphlayout -> boolean -> (boxinfolist, integer, integer)

lines: die in Boxen aufzuteilenden Zeilen

integer: die aktuelle Seitenzahl

pagelayout: das aktuelle Seitenlayout

integer: die auf dieser Seite noch zur Verfügung stehende Resttexthöhe (vertikal)

integer: der vor den nächsten Absatz zu setzende vertikale Freiraum

integer: die aktuelle Zeilenhöhe

boxinfo: die zuletzt gesetzte Box

paragraphlayout: das aktuelle Absatzlayout

boolean: Wahrheitswert, der anzeigt, ob der Absatz der erste zu reformatierende Absatz ist

boxinfolist: Liste der berechneten Boxen in der Reihenfolge ihres Auftretens im Dokument

integer: die zuletzt (re-)formatierte Seite

integer: die auf der zuletzt (re-)formatierten Seite noch zur Verfügung stehende Resttexthöhe (vertikal)

Vor- und Nachbedingungen:

Die eingegebenen Zeilen müssen mit dem Objektformatierer formatiert worden sein, dürfen abernoch nicht ausgerichtet sein.

Implementierung:

Zunächst werden die einzelnen Seiten- und Absatzattribute, wie Texthöhe und Textbreite aus den Layoutparametern ausgelesen. Im Anschluß werden in Abhängigkeit davon, ob der zu setzende Absatz an den vorherigen Absatz angehängt werden soll, oder in einer neuen Zeile beginnen soll, die Anfangskoordinaten des zu setzenden Absatzes und ein Zeilenoffset, der bei hintereinanderhängenden Absätzen berücksichtigt werden muß, berechnet.
Nun beginnt das eigendliche Setzen der Box (des Absatzes). Mithilfe der Division des auf der aktuellen Seite verbleibenden vertikalen Freiraums durch die Zeilenhöhe des finalen Elements wird die Anzahl der Zeilen des finalen Elements bestimmt, die noch auf diese Seite passen.
Die Hilfsfunktion `fitLines` teilt dann die Zeilen in auf diese Seite passende Zeilen (FrontLines) und nicht auf diese Seite passende Restzeilen (RestLines) auf.
Im folgenden werden die relative und die reale Boxhöhe, die zum weiteren Setzen der Boxkoordinaten vonnöten sind, berechnet. Diese unterscheiden sich dadurch, da bei der relativen Boxhöhe, falls der Absatz an einen anderen angehängt wird, die erste Zeile, in die auch Worte des vorhergehenden Absatzes formatiert wurden, nicht mitgezählt wird.
Nun werden die vier Eckkoordinaten der Box gesetzt. Im Anschlu werden mithilfe der Funktion `setCorrectYCoords` die Y-Koordinaten der zu dieser Box gehörenden Zeilen so umgesetz, daß sie relativ zur linken oberen Ecke dieser Box sind.
Schließlich werden mit der Funktion `setNewBoxInfo` die berechneten Informationen in einen Instanz des Typs `boxinfo` geschrieben.
Sind nun die restlichen Zeilen leer, d.h. das finale Element enthält keinen weiteren Text, ist die Funktion fertig und das 3-Tupel bestehend aus der errechneten Box, der aktuellen Seite und dem auf dieser Seite verbleibenden vertikalen Freiraum wird zurückgeliefert.
Existieren noch restliche Zeilen, so müssen auch diese noch in Boxen aufgeteilt werden. Da diese Seite aber keine Zeilen mehr aufnehmen kann liegt die nächste Box auf der nächsten Seite.
Also ruft sich die Funktion `fromLinesToBoxes` mit den restlichen Zeilen und den entsprechenden Parametern, die das weitere Aufteilen in Boxen auf den nächsten Seiten bewirken, rekursiv auf. Beim Rückweg aus der Rekursion wird die Liste der berechneten Boxen aufgebaut. Hierbei werden die Nummer der Seite auf die als letztes formatiert wurde und der auf ihr noch zur Verfügung stehende Freiraum durchgereicht.

Funktionsname:

equalLayout equalLayout

Beschreibung:

Die Funktion `equalLayout` überprüft zwei Parameter der Sorte `pagelayout` auf Gleichheit. Bei bereinstimmung der Eingabeparameter wird `true` zurückgeliefert, sonst `false`.

Signatur:

equalLayout:: pagelayout -> pagelayout -> boolean

pagelayout: erster Vergleichsparameter

pagelayout: zweiter Vergleichsparameter

boolean: zurückgegebener Wahrheitswert

Implementierung:

Jeweils gleichartige Attribute der beiden Layouts, wie `topmargin`, `headheight` usw. werden ausgelesen und miteinander verglichen. Diese Vergleiche sind insgesamt mit dem UND-Operator verknüpft, somit wird also nur `true` zurückgeliefert, wenn alle gleichartigen Attribute der beiden Layouts gleich sind.

Funktionsname:

arrangeFinalElements arrangeFinalElements

Beschreibung:

Die Funktion `arrangeFinalElements` übernimmt eine der Hauptaufgaben beim initialen Formatieren des gesamten aktuellen Dokumentes. Sie wird unter anderem von der Funktion `formatPage` aufgerufen und sorgt dafür, das alle finalen Elemente, die im Dokument vorkommen, in der richtigen Reihenfolge und mit dem richtigen Layout formatiert und auf aufeinanderfolgenden Seiten angeordnet werden. Außerdem wird die initiale Seitennummerrierung durchgeführt.

Signatur:

arrangeFinalElements:: env -> fs -> integers -> pagelayout -> integer -> integer -> integer -> boxinfo -> (env, fs)

env: aktuelle Umgebungsvariable

fs: die aktuelle Formatiererstruktur

integers: die Liste aller finalen Elemente des Dokumentes in der richtigen Reihenfolge

pagelayout: das Seitenlayout des vorherigen finalen Elementes (beim ersten Aufruf leeres Pagelayout)

integer: Seitenzähler (initial 0)

integer: restliche Texthöhe auf der aktuellen Seite

integer: der Freiraum, der noch hinter den letzten Absatz gesetzt werden muß

boxinfo: die zuletzt formatierte Box (mit ihren Koordinaten)

env: die neue Umgebungsvariable

fs: die veränderte Formatiererstruktur

Vor- und Nachbedingungen:

Beim ersten Aufruf der Funktion muß der Seitenzähler gleich Null und das Seitenlayout leer sein.

Implementierung:

Im ersten Abschnitt der Funktion werden die drei Layoutparameter (PageLayout, ParagraphAttributes und DokumentAttributes) des ersten finalen Elementes der eingegebenen Liste aus der Umgebungsvariablen ausgelesen. Aus diesen Layout-Attributsammlungen werden die hier benötigten Attribute ausgelesen (`textheight`, `BaseLineSkip` usw.).
Eines der wichtigsten Attribute in diesem Zusammenhang ist das `atposition` Attribut der `DokumentAttributes`, das falls es gleich `nextline` ist anzeigt, daß das aktuelle finale Element in der nächsten Zeile anfangen soll, also einen neuen Absatz bildet und nicht an das letzte finale Element in der letzten Zeile angehängt werden soll.
Im nächsten Abschnitt der Funktion werden drei für die Formatierung wichtige Variablen berechnet.

- FirstLineEmptyspace: Diese Variable gibt den noch zur Verfügung stehenden Freiraum in der ersten Zeile des nun zu formatierenden finalen Elementes an (also den Teil der Zeile, der noch nicht vom vorherigen finalen Element belegt wurde). Sie wird mithilfe der Funktion `calculateFirstLineEmptyspace` berechnet und ist nach Definition gleich Null, wenn das nun zu formatierende finale Element in eine neue Zeile formatiert werden soll.
- NewLastBoxInfo: Diese Variable ist später einer der Eingabeparameter der Funktion `fromLinesToBoxes`. Sie soll, falls das nun zu formatierende Element an das letzte angehängt werden soll die Box des letzten finalen Elementes enthalten, soll das nun zu formatierende finale Element aber in eine neue Zeile formatiert werden gleich `mtBoxInfo` sein. Die richtige Belegung dieser Variablen übernimmt die Funktion `calculateWhichBoxinfo`.
- VerticalSpace: Falls das nun zu formatierende finale Element in einer neuen Zeile beginnen soll, also einen neuen Absatz bildet, wird hier der durch die Layoutattribute vorgegebene Abstand zwischen den Absätzen ermittelt. Soll nicht in eine neue Zeile formatiert werden ist dieser Abstand gleich Null.

Im weiteren Verlauf der Funktion wird überprüft, ob das Seitenlayout des vorherigen mit dem Seitenlayout des aktuellen finalen Elementes übereinstimmt. Ist dies nicht der Fall, wird ein Seitenumbruch erzeugt, denn es können nicht zwei verschiedene Layouts auf einer Seite gelten. Dies geschieht, indem sich die Funktion mit der gleichen Objektliste wie vorher, dem neuen Seitenlayout, dem um eins erhöhten Seitenzähler und der neuen Texthöhe rekursiv aufruft.
Sind die beiden Seitenlayouts gleich, so kann das aktuelle finale Element auf die aktuelle Seite formatiert werden. Zu diesem Zweck wird zunächst mit Hilfe der Funktion `formatObjekt` aus dem Modul `SimpleFormatter` der Text des aktuellen finalen Elementes in formatierte Zeilen umgewandelt, die dann von der Hilfsfunktion `fromLinesToBoxes` auf der aktuellen und eventuell auch auf den nächsten Seiten angeordnet und in Boxen eingeteilt werden. Anschließend wird die so ermittelte Liste von Boxen in der Instanz des aktuellen finalen Elementes in der Formatiererstruktur abgelegt.
Nun ruft sich diese Funktion mit der neuen Formatiererstruktur, den restlichen finalen Elementen, dem Seitenlayout des gerade formatierten finalen Elementes, der aktuellen Seite und der Resttexthöhe auf dieser Seite (beide Rückgabewerte von `fromLinesToBoxes`), dem `Postparsep` des gerade formatierten Elementes (hinter den Absatz zu setzender vertikaler Freiraum, über das setzen dieses Freiraums kann erst beim Setzen des nächsten finalen Elementes entschieden werden) und der letzten Box des gerade formatierten finalen Elementes rekursiv auf, um das Setzen des nächsten finalen Elementes anzustoßen. Diese

Rekursion bricht ab, wenn die Liste der finalen Elemente leer ist, es also keine Objekte mehr zu formatieren gibt. In diesem Fall wird das zwei-Tupel aus der Umgebungsvariablen und der aktuellsten Formatiererstruktur zurückgeliefert.

Funktionsname:

arrangeFEuntilPage arrangeFEuntilPage

Beschreibung:

Die Funktion `arrangeFEuntilPage` ist eine Hilfsfunktion für die Funktion `reformatFromObjectToPage`, wird wie diese im Moment nicht mehr benutzt und soll deswegen nur kurz beschrieben werden. Sie arbeitet im großen und ganzen wie die Funktion `arrangeFinalElements` mit dem Unterschied, daß sie nur bis zu einer bestimmten Seite formatiert und den Rest unformatiert läßt. Was für die Benutzung dieser Funktion noch fehlt, ist ein Mechanismus, der die mit ihr vorformatierten finalen Element es im Moment so noch nicht. - Dies ist aber nicht schlimm, denn die Funktion zum inkrementellen Formatieren `reformatObject`, für die ein solcher Mechanismus existiert, wird nun an der Stelle eingesetzt, wo vorher `reformatFromObjectToPage` benutzt wurde. Dies hat den Nachteil, daß man in unserer Textverarbeitung nicht beliebig von einer Seite zu einer anderen springen kann, sondern sich langsam hinblättern muß, weil `reformatObject` eben immer nur eine Seite neu formatieren kann. Wollte man ein beliebiges Springen im Dokument ermöglichen, mü te man im Modul `FormatAlignment` eine Funktion implementieren, die das Ausrichten von mit `arrangeFEuntilPage` reformatierten Objekten durchführt.

Signatur:

arrangeFEuntilPage:: env -> fs -> integers -> pagelayout -> integer -> integer -> integer -> boxinfo -> integer -> (env, fs).

env: die aktuelle Umgebungsvariable

fs: die aktuelle Formatiererstruktur

integers: die Liste aller finalen Elemente des Dokumentes in der richtigen Reihenfolge

pagelayout: das Seitenlayout des vorherigen finalen Elementes (beim ersten Aufruf leeres Pagelayout)

integer: Seitenzähler (initial 0)

integer: restliche Texthöhe auf der aktuellen Seite

integer: der Freiraum, der noch hinter den letzten Absatz gesetzt werden mu

boxinfo: die zuletzt formatierte Box (mit ihren Koordinaten)

integer: Seite bis zu der formatiert werden soll

env: die neue Umgebungsvariable

fs: die veränderte Formatiererstruktur

Funktionsname:

getPredecAndObjs getPredecAndObjs

Beschreibung:

Die Funktion `getPredecAndObjs` bekommt eine `ObjectID` und die Liste aller im Dokument vorhandenen finalen Elemente herein. Zurückgeliefert wird die ID des Vorgängers des eingegebenen Objektes, sowie die Restliste der Objekte beginnend mit dem eingegebenen Objekt.

Signatur:

getPredecAndObjs:: integer -> integers -> (integer, integers)

integer: eingegebenes finales Element

integers: Liste aller im Dokument vorhandenen finalen Elemente

integer: Vorgänger des eingegebenen finalen Elementes

integers: Restliste beginnend mit dem eingegebenen finalen Element

Vor- und Nachbedingungen:

Es darf sich bei dem eingegebenen Objekt nicht um das erste in der Liste der finalen Elemente handeln, da dieses ja keinen Vorgänger hat.

Implementierung:

Bei jedem Funktionsaufruf wird zunächst das erste Element der Eingabeliste abgetrennt. Das erste Element der Restliste wird nun mit dem eingegebenen Objekt verglichen. Stimmen diese überein, so ist das zuerst abgetrennte Objekt der Vorgänger des gesuchten Objektes und das gesuchte Objekt selbst ist erstes Objekt in der Restliste. Somit werden das zuerst abgetrennte Objekt und die Restliste in einem Tupel als Ergebnis zurückgeliefert.
Stimmen die verglichenen Objekte nicht überein wird rekursiv in der Restliste weitergesucht. Falls das gesuchte Objekt nicht in der Liste enthalten ist, werden nach Definiton das Tupel aus Null und der leeren Liste zurückgeliefert.

Funktionsname:

containsDelimiter containsDelimiter

Beschreibung:

Die rekursive Hilfsfunktion `containsDelimiter` wird von der Funktion `reformatLine` benutzt und prüft, ob in einer Zeichenkette ein Trennzeichen wie Leerzeichen, Tabulator oder Zeilenumbruch enthalten ist. Ist dies der Fall wird `true` geliefert, sonst `false`.

Signatur:

containsDelimiter:: chars -> boolean

chars: eingegebener Zeichenstrom

boolean: zurückgegebener Wahrheitswert

Implementierung:

Das erste Element des Zeichenstroms wird abgetrennt. Mit der Hilfsfunktion `is_delimiter` aus dem Modul `FormatTextConvert` wird überprüft, ob es sich bei dem Zeichen um ein Trennzeichen handelt. Ist dies der Fall wird `true` geliefert, anderenfalls wird rekursiv im Restzeichenstrom weitergesucht.
Das Ergebnis beim Aufruf der Funktion mit der leeren Liste ist `false`.

Funktionsname:

calculateFirstLineEmptyspace calculateFirstLineEmptyspace

Beschreibung:

Die Funktion `calculateFirstLineEmptyspace` ermittelt den Freiraum, der in einer Zeile noch zum Formatieren zur Verfügung steht, also von keinem anderen finalen Element belegt wurde. Die Funktion bezieht ihre Information über den Freiraum aus der zuletzt gesetzten Box, die Eingabeparameter ist. Der zweite Parameter bietet die Möglichkeit in besonderen Fällen, z.B. falls das nächste finale Element in einer neuen Zeile beginnen soll, das Auslesen dieses Freiraums aus der zuletzt gesetzte Box zu unterdrücken. In diesem Fall und wenn die zuletzt gesetzte Box leer ist (dies bedeutet ebenfalls, da eine neue Zeile begonnen werden soll) wird nach Vereinbarung Null zurückgeliefert. Dies ist leider logisch falsch, was uns zu spät aufgefallen ist (Null heißt eigendlich, da die Zeile voll ist). Dieser Fehler wird im `SimpleFormatter` durch das Einsetzen der entsprechenden Textbreite (bei Null) behoben. Dies ist natürlich unschön und muß noch berichtigt werden.

Signatur:

calculateFirstLineEmptyspace:: boxinfo -> boolean -> integer

boxinfo: die zuletzt gesetzte Box

boolean: Wahrheitswert der einen neuen Absatz anzeigt (bei `true`)

integer: Restfreiraum in der letzten Zeile (oder Null)

Implementierung:

Es wird überprüft ob die eingegebene Box leer ist oder ob ein neuer Absatz beginnen soll. Geht dieser Vergleich positiv aus wird Null zurückgeliefert. Anderenfalls werden die Zeilen der Box ausgelesen, die letze Zeile bestimmt und ihr Restfreiraum abgefragt.

Funktionsname:

calculateWhichBoxinfo calculateWhichBoxinfo

Beschreibung:

Die Funktion `calculateWhichBoxinfo` wird von den Funktionen benutzt die auch die Funktion `fromLinesToBoxes` benutzen. Letztere bekommt in ihren Parametern unter anderem eine Box vom Typ `boxinfo` übergeben. Diese Box dient einerseits als Schalter um anzuzeigen, da nun ein neuer Absatz begint (wenn Box gleich `mtBoxInfo`), andererseits enthält diese Box, falls kein neuer Absatz beginnt, also an den alten angehängt werden soll, auch gleich die Koordinaten des alten Absatzes, die die Ansatzstelle repräsentieren. Es ist also wichtig, daß hier die richtige Box übergeben wird. Dies ist die Aufgabe der Funktion `calculateWhichBoxinfo`. Das Ermitteln der richtigen Box wird durch die eingegebenen Wahrheitswerte gesteuert. Hierbei zeigt der erste Wahrheitswert den Beginn eines neuen Absatzes an. Dies kann durch den zweiten Wahrheitswert unterdrückt werden, da beim Reformatieren auch wenn ein neuer Absatz beginnt die Koordinaten des vorherigen Absatzes als Ansatzpunkt benötigt werden.

Signatur:

calculateWhichBoxinfo:: boxinfo -> boolean -> boolean -> boxinfo

boxinfo: zuletzt formatierte Box

boolean: Wahrheitswert der einen neuen Absatz anzeigt (bei `true`)

boolean: Wahrheitswert der anzeigt, ob es sich um das erste finale Element beim Reformatieren handelt (bei `true`)

boxinfo: ermittelte Box

Implementierung:

Falls es sich bei der vorherigen Box um eine leere handelt oder bei dem jetzt zu formatierenden Absatz um einen neu beginnenden, und es sich bei dem jetzt zu formatierenden Absatz nicht um den ersten beim Reformatieren handelt, wird die leere Box zurückgeliefert. Anderenfalls ist das Ergebnis die eingegebene Box.

Funktionsname:

calculateVerticalSpace calculateVerticalSpace

Beschreibung:

Die Funktion `calculateVerticalSpace` berechnet den Freiraum, der vor eine neu zu setzende Subbox gesetzt werden muß.
Der Parameter `LastPostparsep` enthält den Wert des Postparsep, des zuletzt gesetzten finalen Elementes. Dieser muß natürlich nur gesetzt werden, wenn das aktuelle Element nicht an das letzte direkt angehängt werden soll, also wenn das aktuelle Element ein neuer Absatz sein soll. Ob dies der Fall ist sagt der Wahrheitswert `NewParaBool`, der für einen neuen Absatz `true` ist. ähnliches gilt für das `Preparsep` des aktuellen Elementes, das auch nur für neue Absätze gesetzt werden muß.
Der von dieser Funktion zurückgelieferte Wert `VerticalSpace` setzt sich (gegebenenfalls) aus den Parametern `LastPostparsep` und `Preparsep` zusammen. Dies ist eigentlich nicht ganz richtig, den das Postparsep sollte nach Definition zu dem vorherigen Absatz gehören. Da aber erst beim Formatieren des aktuellen Absatzes klar wird ob es sich hier um einen neuen oder einen angehähgten handelt, haben wir es der Einfachheit halber in den aktuellen Absatz hineingenommen.

Signatur:

calculateVerticalSpace:: integer -> integer -> boolean -> integer

integer: der Freiraum, der hinter den letzten Absatz gesetzt werden mu

integer: der Freiraum, der vor den jetzt zu formatierenden Absatz gesetzt werden mu

boolean: Wahrheitswert der einen neuen Absatz anzeigt (bei `true`)

integer: sich ergebender vertikaler Freiraum

Implementierung:

Soll ein neuer Absatz gesetzt werden wird die Summe der eingegebenen Einzelfreiräume zurückgeliefert, sonst Null.

Funktionsname:

getObjsWithObjID getObjsWithObjID

Beschreibung:

Die Funktion `getObjsWithObjID` bekommt ein finales Element (Objekt) und die Liste aller im Dokument vorhandenen finalen Elemente herein. Zurückgeliefert wird die Restliste der Objekte beginnend mit dem eingegebenen Element.

Signatur:

getObjsWithObjID:: objID -> objIDList -> objIDList

objID: zu suchendes, finales Element

objIDList: Liste aller finalen Elemente des aktuellen Dokumentes

objIDList: Liste der restlichen Objekte beginnend mit dem eingegebenen Element

Implementierung:

Das erste Element der eingegebenen Liste wird abgetrennt und mit dem zu suchenden finalen Element verglichen. Stimmen diese überein wird das gerade abgetrennte Objekt wieder vor die Liste gehängt und diese so entstandene Liste zurückgeliefert.
Wird das eingegebene Element nicht gefunden ist die leere Liste das Ergebnis.

Funktionsname:

setFinalsNotFormatted setFinalsNotFormatted

Beschreibung:

Die Funktion `setFinalsNotFormatted` setzt eine Liste von finalen Elementen auf `nonformatted`.

Signatur:

setFinalsNotFormatted:: fs -> objIDList -> fs

fs: eingegebene Formatiererstruktur

objIDList: auf nicht formatiert zu setzende finale Elemente

fs: modifizierte Formatiererstruktur

Implementierung:

Die Funktion arbeitet rekursiv jedes Element der eingegebenen Liste ab. Das ïicht formatiert"Setzen eines finalen Elementes besteht aus dem Herausschreiben der Instanz des finalen Elementes aus der Formatiererstruktur, dem Auslesen der Boxen aus dieser Instanz, dem auf ïicht formatiert"Setzen jeder dieser gelesenen Boxen (mithilfe der Funktion `setBoxesNotFormatted`) und dem Rückschreiben dieser Instanzen in die Formatiererstruktur.
Ist die Liste der finalen Elemente leer wird die aktuelle Formatiererstruktur als Ergebnis zurückgeliefert.

Funktionsname:

setBoxesNotFormatted setBoxesNotFormatted

Beschreibung:

Die Funktion `setBoxesNotFormatted` setzt eine Liste von Boxen auf `nonformatted`.

Signatur:

setBoxesNotFormatted:: boxinfolist -> boxinfolist

boxinfolist: eingegebene Boxenliste

boxinfolist: modifizierte Boxenliste

Implementierung:

Rekursiv wird für jede eingegebene Box der Boxstatus auf `notformatted` gesetzt.

Funktionsname:

arrangeChangedObs arrangeChangedObs

Beschreibung:

Die Funktion `arrangeChangedObs` wird von `reformatObject` aufgerufen und sorgt für eine richtige Neuformatierung (Reformatierung), falls Objekte eingefügt, gelöscht oder in einem umfangreicheren Ma e als in einer Zeile verändert wurden.
Ihre Arbeitsweise ist ähnlich der von `arrangeFinalElements`, mit dem Unterschied, daß hier Mitten im Dokument mit der Formatierung begonnen wird und beim Seitenende aufgehört wird.
Es wird keine Rücksicht auf ehemalige Formatierungen genommen. Die zurückgegebene Liste der Integer beinhaltet die restlichen nicht formatierten Objekte, die sich auf den nächsten Seiten befinden und auf `nonformatted` gesetzt werden müssen.

Signatur:

arrangeChangedObs:: objIDList -> env -> fs -> objIDList -> pagelayout -> integer -> integer -> integer -> boxinfo -> boolean -> integer -> (objIDList, env, fs, objIDList)

objIDList: da dies eine rekursive Funktion ist werden hier die gerade reformatierten Objekte in einer Liste abgelegt

env: eingegebene Umgebungsvariable

fs: eingegebene Formatiererstruktur

objIDList: Liste der zu reformatierenden finalen Elemente

pagelayout: Das Seitenlayout des zuvor reformatierten Objektes

integer: der vertikale Freiraum, der hinter das letzte finale Element zu setzen ist

integer: aktuelle Seitenzahl

integer: aktuelle Resttexthöhe

boxinfo: die zuletzt reformatierte Box

boolean: Schalter der angibt, ob das gerade zu reformatierende Objekt das erste ist, also ob dies der erste aufruf von `arrangeChangedObs` ist

integer: das alte erste Wort, damit auch teile von finalen Elementen reformatiert werden können

objIDList: die reformatierten finalen Elemente

env: die manipulierte Umgebungsvariable

fs: die neue Formatiererstruktur

objIDList: die nicht formatierten Objekte

Vor- und Nachbedingungen:

Die nicht reformatierten Objekte müssen auf `nonformatted` gesetzt werden.

Implementierung:

Zunächst wird das Seitenlayout des aktuell zu reformatierenden finalen Elemente ausgelesen und mit dem des vorherigen verglichen. Handelt es sich um verschiedene Seitenlayouts so gibt es an dieser Stelle einen Seitenumbruch und die Funktion ist beendet.
Anderenfalls kann das aktuelle Objekt reformatiert werden. Zu diesem Zweck werden parallel zu `arrangeFinalElements` die Variablen zum Ansteuern der Funktion `fromLinesToBoxes`, die dann das Setzen des finalen Elementes auf der aktuellen Seite übernimmt, bestimmt. Dies sind die Absatzattribute, der restliche Freiraum auf der letzten Zeile, die letzte gesetzte Box (wobei natürlich berücksichtigt werden muß, ob es sich um die erste zu reformatierende Box handelt), der vertikale Freiraum vor dem finalen Element und die neu formatierten Zeilen des finalen Elementes.

Die Funktion `fromLinesToBoxes` liefert nun die zu dem aktuellen Objekt gehörenden reformatierten Boxen, die neue Seitenzahl und die neu Resttexthöhe auf dieser aktuellen Seite zurück. Die so reformatierten Boxen (bis auf Ausrichtung reformatiert) müssen nun noch in die Formatiererstruktur zurückgeschrieben werden. Hierbei muß berücksichtigt werden, daß keine Boxen des aktuellen Objektes, die vor dieser Seite liegen und noch formatiert sind überschrieben werden. Diese Aufgabe übernimmt die Funktion `setNewFinalBoxInfoList`. Im Weiteren wird abhängig davon, ob die reformatierte Boxliste einelementig ist oder nicht rekursiv das nächste Element weiter formatiert oder die Rekursion abgebrochen. Dies findet seine Begründung darin, daß es zu einem finalen Element immer nur eine Box auf einer Seite geben kann und somit mehrere Boxen hier einen Seitenumbruch implizieren. Ein Seitenumbruch bedeutet aber den Abbruch der Funktion. Nur bei einer einelementigen Boxliste bleibt man auf der gleichen Seite und muß hier weiter reformatieren.
Wenn die Liste der zu reformatierenden Objekte leer ist wird natürlich auch abgebrochen.

Funktionsname:

setNewFinalBoxInfoList setNewFinalBoxInfoList

Beschreibung:

Die Funktion `setNewFinalBoxInfoList` schreibt die zu einem finalen Element gehörenden Boxen in die Instanz des finalen Elementes. Im Gegensatz zu der Funktion `setFinalBoxInfoList`, die hierbei alle vorher in der Instanz abgelegten Boxen überschreibt, gibt es hier die Möglichkeit die Boxen die eine geringere Seitennummer haben als die erste einzuschreibende Box in der Instanz des finalen Elementes zu belassen. In diesem Fall muß aber der erste Eingabeparameter vom Typ `boolean` den Wert `true` haben.

Signatur:

setNewFinalBoxInfoList:: boolean -> fdata -> boxinfolist -> fdata

boolean: dieser Schalter gibt an, ob alle vorherigen Boxen überschrieben werden sollen

fdata: alte Instanz des finalen Elementes

boxinfolist: einzusetzende Boxen

fdata: modifizierte Instanz des finalen Elementes

Implementierung:

Ist der erste Eingabeparameter gleich `false` sollen die alten Boxen überschrieben werden und die Funktion `setFinalBoxInfoList` wird aufgerufen.
Anderenfalls werden die alten Boxen aus der Instanz ausgelesen und mit Hilfe der Funktion `calcNewBoxes` wird eine Liste von Boxen erzeugt, die die alten und die neuen Boxen enthält (natürlich nicht alle alten). Diese Liste von Boxen wird dann wiederum durch die Funktion `setFinalBoxInfoList` in die Instanz des finalen Elementes geschrieben.

Funktionsname:

calcNewBoxes calcNewBoxes

Beschreibung:

Die Funktion `calcNewBoxes` bestimmt eine Liste von Boxen. Diese Liste setzt sich zusammen aus alten Boxen und neuen reformatierten Boxen. Die Liste der neuen Boxen wird dort eingehängt, ab wo sich etwas geändert hat.

Signatur:

calcNewBoxes:: boxinfolist -> boxinfolist -> boxinfolist -> boxinfolist

boxinfolist: hier wird die Rückgabeliste bei jedem rekursiven Aufruf aufgebaut (muß beim ersten Aufruf die leere Liste sein)

boxinfolist: die Liste der alten Boxen

boxinfolist: die Liste der reformatierten Boxen

boxinfolist: die Rückgabeliste

Implementierung:

Die beiden erste Elemente der Boxlisten werden abgetrennt. Von jeder dieser Boxen wird die Nummer des ersten Wortes dieser Box bezüglich des logischen Elementes bestimmt. Solange wie die Wortnummern ungleich sind wird die alte Box im ersten Parameter abgelegt und die Funktion rekursiv aufgerufen. Sind die Nummern gleich, werden die Liste im ersten Parameter und die Liste der reformatierten Boxen konkateniert und als Ergebnis zurückgeliefert.

Funktionsname:

storeLineInBox storeLineInBox

Beschreibung:

Die Funktion `storeLineInBox` ordnet eine Zeile an der richtigen Stelle in die zugehörige Box ein. Die vorher an dieser Stelle befindliche Zeile wird überschrieben.

Signatur:

storeLineInBox:: boxinfo -> line -> boxinfo

boxinfo: eingegebene Box

line: einzufügende Zeile

boxinfo: manipulierte Box mit Zeile

Vor- und Nachbedingungen:

Die Funktion wird benutzt um eine reformatierte Zeile in eine Box einzutragen. Hierbei wird die ehemalige Instanz der Zeile überschrieben. Gibt man nun eine andere Box ein, die die ehemalige Instanz der Zeile nicht beinhaltet, kommt es zu Fehlern.

Implementierung:

Die Zeilen der Box werden ausgelesen. Die Hilfsfunktion `storeLineHelp` wird benutzt um die Zeile einzuordnen. Die sich so ergebenden Zeilen werden wieder in die Box geschrieben.

Funktionsname:

storeLineHelp storeLineHelp

Beschreibung:

Die Hilfsfunktion `storeLineHelp` durchläuft eine Liste von Zeilen und hängt bei gleichen Wortnummern eine neue Zeile ein.

Signatur:

storeLineHelp:: lines -> line -> integer -> lines

lines: zu durchlaufende Zeilen

line: einzuhängende Zeile

integer: Nummer des ersten Wortes der einzuhängenden Zeile

lines: Zeilen mit eingehängter Zeile

Implementierung:

Die rekursive Funktion durchläuft die Liste der Zeilen. Die Nummer des ersten Wortes der aktuellen Zeile wird ausgelesen und mit der Nummer des ersten Wortes der einzuhängenden Zeile verglichen. Bei gleichheit wird die neue Zeile eingehängt und die Funktion abgebrochen. Bei ungleichheit wird die Rekursion weitergeführt.

2.16 FormatTextConvert

Tarik Ali, Andree Hähnel

2.16.1 Entwurf

2.16.1.1 Das Modul

Das Modul `FormatTextConvert` ist eine Ressource für die einfachen Formatierer, die aus Absatz- (SimpleFormatter), Ausrichtungs- (FormatAlignment) und Seitenformatierer (PageFormatter) bestehen.
Die Aufgaben dieses Moduls lassen sich in zwei Bereiche aufteilen. Zunächst stellt es den Datentyp `wordlist` und die zugehörigen Funktionen zum Erzeugen und manipulieren dieser Wortlisten zur Verfügung. Der zweite Aufgabenbereich ist der eines Werkzeugkastens von Funktionen, die nur die einfachen Formatierer benutzen.
Das Konzept der `wordlist` ist folgendes: Der zu einem Absatz gehörende Text, der in der internen Struktur (Modul `IS`) gespeichert wird, liegt dort als Zeichenstrom vor. Das Format des Zeichenstroms ist kein Format, das eine schnelle und effiziente Formatierung in einer funktionalen Sprache unterstützt, da hier eventuell Leerzeichen, Absatzumbrüche und Tabulatoren hintereinander vorkommen und eine rekursive Bearbeitung eines solchen unsauberen Zeichenstroms sehr uneffizient ist. Aus diesem Grund wird dieser vor der Formatierung zunächst zu einer `wordlist` konvertiert, die jedes Wort des Zeichenstroms, mit der Nummer bzw. Position dieses Wortes im Zeichenstrom und der Nummer des ersten Buchstaben des Wortes im Zeichenstrom enthält. Die Wortliste enthält keine Leerzeichen mehr. Diese Wortliste kann nun rekursiv vom Formatierer bearbeitet werden.

2.16.1.2 Anmerkung

Der Formatierer speichert in seiner Struktur beim Formatieren nur die Nummern der Worte. Das heißt, daß bei der Ausgabe des formatierten Absatzes erneut die Wortliste erzeugt werden muß, da diese nicht gespeichert wird und nur sie die Information über die Wortpositionen enthält.

2.16.2 Importschnittstelle

Das Modul `FormatTextConvert` importiert Funktionen aus dem Modul `ISFormat` und dem heir schon importierten Modul `FormatStructure`.

2.16.3 Exportschnittstelle

2.16.3.1 Sorten

Sortenname:

wordlistelement　　　　wordlistelement

Signatur:

wordlistelement:: wordlistelem (word :: string) (id :: integer) (firstcharid :: integer)

Beschreibung:

Die Sorte `wordlistelement` repräsentiert ein einzelnes Wort, das in einem Absatz vorkommt. Sie hat den Konstruktor `wordlistelem` und besteht aus drei Attributen. Das erste Attribut bezeichnet das jeweilige Wort, das zweite die Wortnummer im Absatz und das dritte die Nummer des ersten Buchstabens des Wortes bzgl. aller Buchstaben im Absatz (-Zeichenstrom).

Sortenname:

wordlist wordlist

Signatur:

wordlist:: [wordlistelement]

Beschreibung:

Die Sorte `wordlist` repräsentiert einen ganzen Absatz oder Teile eines Absatzes, also eine Liste von Worten. Sie ist implementiert als einfache Liste des Datentyps `wordlistelement`.

2.16.3.2 Funktionen

Funktionsname:

is_delimiter is_delimiter

Beschreibung:

Die Funktion `is_delimiter` analysiert, ob es sich bei einem Zeichen um ein Leerzeichen, ein Tabulatorzeichen oder einen Zeilenumbruch handelt.

Signatur:

is_delimiter:: char -> boolean

char: das zu prüfende Zeichen

boolean: ist `true` falls eingegebenes Zeichen Worttrenner

Abhängigkeiten:

Innerhalb dieses Moduls wird diese Funktion von der Funktion `norm_string` benutzt.

Funktionsname:

makeWordlist makeWordlist

Beschreibung:

Die Funktion `makeWordlist` ist eine Schnittstellenfunktion, die von den einfachen Formatierern benutzt wird, um aus dem in der `Internen Struktur` zu jedem finalen Element vorliegenden Text (`string`, Zeichenstrom) eine Wortliste zu erzeugen, die alle in diesem

Text vorkommenden Worte in der Reihenfolge ihres Auftretens, sowie die jeweilige Wortnummer und die Nummer des ersten Buchstabens dieses Wortes beinhaltet.
Anmerkung:
Diese Wortliste ist die Datenstruktur die beim Formatieren bearbeitet wird, d.h. der Formatierer kennt nur Worte, die von ihm gesetzt werden. Falls in dem Text in der `Internen Struktur` zwei Worte durch mehr als ein Trennzeichen getrennt werden, weiß der Formatierer hiervon nichts. Für ihn gibt es nur ein Trennzeichen. Dies erleichtert die Arbeit des Formatierers erheblich. Das sich hierbei ergebende Problem ist, daß das, was nach der Formatierung auf dem Bildschirm als Text steht, bezüglich der Trennzeichen nicht mit dem Text der `Internen Struktur` übereinstimmen muß, daß also z.B. der erste Buchstabe des zweiten Wortes auf dem Bildschirm die laufende Nummer fünf hat, in der `Internen Struktur` aber als Nummer sechs gehandelt wird, weil vor ihm zwei Leerzeichen stehen. - Man muß also z.B. beim Löschen dieses Buchstabens darauf achten, daß mithilfe der Wortliste seine wirkliche Position ermittelt wird und nicht seine Position nach der Formatierung als Position in der `Internen Struktur` angesehen wird.

Signatur:

makeWordlist:: chars -> wordlist

chars: eingegebener Zeichenstrom

wordlist: die erzeugte Liste von Worten

Implementierung:

Zunächst wird mit Hilfe der Funktion `norm_string` aus dem Eingabezeichenstrom ein normierter Zeichenstrom erzeugt. Hierbei muß initial das Leerzeichen und als erste laufende Zeichennummer die eins übergeben werden.
Aus diesem normierten Zeichenstrom wird mit dem Aufruf der Funktion `createWordlist` die Wortliste erzeugt, die dann auch der Rückgabeparameter der Funktion `makeWordlist` ist. Hierbei müssen initial als erster Parameter die leere Liste, als dritter die laufende Wortnummer eins und als vierter die laufende Zeichennummer eins übergeben werden.

Funktionsname:

getWordlistFromWordID getWordlistFromWordID

Beschreibung:

Die Funktion `getWordlistFromWordID` verkürzt eine Wortliste bis zu einer bestimmten Wortnummer.

Signatur:

getWordlistFromWordID:: wordlist -> integer -> wordlist

wordlist: die Eingabewortliste

integer: die Wortnummer bis zu der die Wortliste verkürzt werden soll

wordlist: die erzeugte Liste von Worten

Implementierung:

Das erste Element der Wortliste wird solange abgeschnitten, bis seine Wortnummer gleich der Eingabenummer ist. Dann wird die Eingabenummer mit dem Rest der Liste zurückgegeben.

Funktionsname:

makeLine makeLine

Beschreibung:

Die Funktion `makeLine` erzeugt mit Hilfe einer Wortliste, der Nummer des ersten Wortes einer Zeile und der Nummer des letzten Wortes einer Zeile den zugehörigen Zeichenstrom dieser Zeile. Diese Funktion wird im Modul `FormatAlignment` benutzt, um die in der Formatiererstruktur abgelegten Informationen zur Ausgabe auf den Bildschirm aufzubereiten.

Signatur:

makeLine:: wordlist -> integer -> integer -> string

wordlist: die die Zeile beinhaltende Liste von Worten

integer: Anfang der Zeile

integer: Ende der Zeile

string: der ermittelte Zeichenstrom der Zeile

Implementierung:

Das erste Wort der Wortliste wird abgespalten. Ist die laufende Nummer dieses Wortes kleiner als die Nummer des ersten Wortes der Zeile, so wurde der Anfang der Zeile noch nicht erreicht, und es wird rekursiv mit `makeLine` im Rest der Wortliste weitergesucht.
Ist die laufende Nummer des abgetrennten Wortes größer als die Nummer des letzten Wortes der Zeile, so ist das Ende der Zeile erreicht, und es wird der leere Zeichenstrom zurückgeliefert.
In jedem anderen Fall handelt es sich bei dem abgetrennten Wort um ein Wort, das in der betreffenden Zeile liegt und somit in den Rückgabestring eingehängt werden muß. Damit dieses Wort nicht direkt vor dem nächsten steht, muß hier noch ein Leerzeichen eingehängt werden. Mit dem erneuten Aufruf von `makeLine` wird nun das nächste Wort der Zeile gesucht.

Funktionsname:

makeWordLine makeWordLine

Beschreibung:

Die Funktion `makeWordLine` erzeugt mit Hilfe einer Wortliste, der Nummer des ersten Wortes einer Zeile und der Nummer des letzten Wortes einer Zeile, die Wortliste, die gerade die Worte dieser Zeile enthält.

Signatur:

makeWordLine:: wordlist -> integer -> integer -> wordlist

wordlist: die die Zeile beinhaltende Liste von Worten

integer: Anfang der Zeile

integer: Ende der Zeile

wordlist: der ermittelte Wortliste der Zeile

Implementierung:

Die Implementierung dieser Funktion ist mit der Implementierung der Funktion `makeLine` identisch. Sie unterscheiden sich lediglich bezüglich ihrer Rückgabedatenstruktur, die hier eine `wordlist` ist.

2.16.3.3 Allgemeine Hilfsfunktionen für die einfachen Formatierer

---

Funktionsname:

getLastBox getLastBox

Beschreibung:

Die einfache Hilfsfunktion `getLastBox` dient den einfachen Formatierern dazu aus einer Subboxliste die letzte Box auszulesen. Hierbei wird berücksichtigt, daß diese Subboxliste auch leer sein kann. In diesem Fall wird die leere Box zurückgeliefert. Diese Funktion wurde nötig, weil die Funktion `last` aus dem Modul `ListOps` auf eine leere Liste angewendet einen Programmfehler erzeugt.

Signatur:

getLastBox:: boxinfolist -> boxinfo

boxinfolist: eingegebene Liste von Boxen

boxinfo: die letzte Box der eingegebenen Liste

Implementierung:

Mit þattern-matchingẅird überprüft, ob die eingegebene Liste leer ist. Für diesen Fall wird mit Hilfe der Funktion `mtBoxinfo` aus dem Modul `FormatStructure` eine leere Instanz des Datentyps `boxinfo` erzeugt. Diese wird dann zurückgegeben. Anderenfalls liefert die Funktion `last` die gewünschte letzte Box der Subboxliste.

Funktionsname:

getFirstBox getFirstBox

Beschreibung:

Diese Hilfsfunktion ist die Inverse zu der Funktion `getLastBox`. Auch beim Ermitteln des ersten Elementes einer Subboxliste ergibt sich das Problem, daß diese leer sein kann. Dies wird wiederum von der Funktion `hd` aus dem Modul `ListOps` nicht berücksichtigt und führt so zu einem Programmfehler. Die Funktion `getFirstBox` liefert die erste Box einer Subboxliste. Für den Fall einer leeren Liste wird die leere Box geliefert.

Signatur:

getFirstBox:: boxinfolist -> boxinfo

boxinfolist: eingegebene Liste von Boxen

boxinfo: die erste Box der eingegebenen Liste

Implementierung:

Mit "pattern-matching" wird überprüft, ob es sich bei der Subboxliste um eine leere Liste handelt. Ist dies der Fall, so liefert die Funktion `mtBoxinfo` aus dem Modul `FormatStructure` die leere Box, die dann auch das Ergebnis dieser Funktion ist. In jedem anderen Fall wird mit der Funktion `hd` das erste Element der Liste ermittelt.

Funktionsname:

getLastLine getLastLine

Beschreibung:

Die Hilfsfunktion `getLastLine` ist der Funktion `getLastBox` sehr ähnlich und hilft den einfachen Formatierern dabei aus einer Liste von Zeilen die letzte Zeile auszulesen. Hierbei wird berücksichtigt, daß diese Liste auch leer sein kann. In diesem Fall wird die leere Zeile zurückgeliefert.

Signatur:

getLastLine:: lines -> line

lines: eingegebene Liste von Zeilen

line: die letzte Zeile der eingegebenen Liste

Implementierung:

Mit "pattern-matching" wird überprüft, ob die eingegebene Liste leer ist. Für diesen Fall wird mit Hilfe der Funktion `mtLine` aus dem Modul `FormatStructure` eine leere Instanz des Datentyps `line` erzeugt. Diese wird dann zurückgegeben. Anderenfalls liefert die Funktion `last` die gewünschte letzte Zeile der Liste.

Funktionsname:

getFirstLine getFirstLine

Beschreibung:

Diese Hilfsfunktion ist die Inverse zu der Funktion `getLastLine` und der Funktion `getFirstBox` sehr ähnlich.
Die Funktion `getFirstLine` liefert die erste Zeile einer Liste von Zeilen. Für den Fall einer leeren Liste wird die leere Zeile geliefert.

Signatur:

getFirstLine:: lines -> line

lines: eingegebene Liste von Zeilen

line: die erste Zeile der eingegebenen Liste

Implementierung:

Mit "pattern-matching" wird überprüft, ob es sich bei der Liste von Zeilen um eine leere Liste handelt. Ist dies der Fall, so liefert die Funktion `mtLine` aus dem Modul `FormatStructure` die leere Zeile, die dann auch das Ergebnis dieser Funktion ist. In jedem anderen Fall wird mit der Funktion `hd` das erste Element der Liste ermittelt.

Funktionsname:

getFirstOfObjIDs getFirstOfObjIDs

Beschreibung:

Die Hilfsfunktion `getFirstOfObjIDs` liefert das erste Objekt einer Objektliste. Ist die Objektliste leer so wird nach Vereinbarung der Wert Null geliefert. Auch diese Funktion wurde implementiert um einen Programmfehler beim Aufruf der Funktion `hd` mit einer leeren Liste zu umgehen.

Signatur:

getFirstOfObjIDs:: objIDList -> objID

objIDList: die aktuelle Objektliste

Implementierung:

Für den Fall einer leeren Liste wird Null als Ergebnis zurückgeliefert. Anderenfalls ergibt sich das Ergebnis aus der Anwendung der Funktion `hd` auf die Objektliste.

2.16.4 Lokaler Teil

2.16.4.1 Sorten

Sortenname:

specialchar specialchar

Signatur:

specialchar:: spechar (spchar :: char) (charid :: integer)

Beschreibung:

Diese lokale Sorte steht in engem Zusammenhang mit der Funktion `norm_string` und beinhaltet ein Zeichen und die Nummer dieses Zeichens in dem gerade zur Bearbeitung anstehenden Zeichenstrom. Der Datentyp `specialchar` hat den Konstruktor `spechar` besteht aus einem Zeichen und einer Zahl. Diese Zahl bezeichnet die Nummer des Zeichens im Absatz.

Sortenname:

specialchars — specialchars

Signatur:

specialchars:: [specialchar]

Beschreibung:

Diese Sorte repräsentiert einen Zeichenstrom, bei dem jedem Zeichen noch eine bestimmte Nummer zugeordnet ist. Dieser Datentyp wird benötigt, um bei der Konvertierung eines Zeichenstroms in eine Wortliste die Positionen der Buchstaben in dem ehemaligen Zeichenstrom nicht zu verlieren, damit in der Wortliste zu jedem Wort auch wirklich die Position des ersten Buchstaben dieses Wortes im Absatz abgelegt werden kann. (Siehe auch `makeWordlist` und `norm_string`). Der Datentyp `specialchars` ist als einfache Liste der Sorte `specialchar` implementiert.

2.16.4.2 Funktionen

Funktionsname:

norm_string — norm_string

Beschreibung:

Die rekursive Hilfsfunktion `norm_string` wird von der Funktion `makeWordlist` benutzt und hat die Aufgabe aus einem Zeichenstrom alle mehrfach hintereinander auftretenden Worttrennzeichen, wie Leerzeichen, Tabulatoren oder Zeilenumbrüche herauszufiltern. So liefert diese Funktion einen Zeichenstom zurück, der aus Worten besteht, die jeweils nur durch ein Leerzeichen getrennt sind. Jedem dieser Zeichen im Zeichenstrom ist die Position im Eingabezeichenstrom zugeordnet. D.h. kommt im Eingabezeichenstrom an den Positionen drei und vier ein Leerzeichen und an Position fünf der Buchstabe 'a' vor, so würde im Ausgabestrom an Position vier der Buchstabe 'a' stehen, dem aber als Position die Zahl fünf zugeordnet wurde.

Signatur:

norm_string:: char -> chars -> integer -> specialchars

char: das aktuelle Zeichen

chars: der zu bearbeitende Restzeichenstrom integer: die Nummer (Position) des aktuell ersten Zeichens im Restzeichenstrom in Bezug auf den gesamten (Anfangs-) Zeichenstrom (beim ersten Aufruf der Rekursion).

specialchars: der ausgezeichnete Zeichenstrom

Implementierung:

Durch ‚pattern-matching' werden hier drei Fälle unterschieden. - Falls die zu bearbeitende Liste leer ist, ist die Bearbeitung beendet und die leere Liste wird zurückgeliefert.
Für die nächsten beiden Fälle wird der erste Eingabeparameter, das Zeichen, überprüft. Handelt es sich hierbei um ein Leerzeichen, bedeutet dies, daß das letzte Zeichen dieses Zeichenstroms ein Worttrennzeichen war. Ist dies der Fall, so wird geprüft, ob das jetzt erste Zeichen des Zeichenstrom auch ein Worttrennzeichen ist. Dieses würde dann gelöscht werden, indem wiederum die Funktion `norm_string` mit dem Restzeichenstrom und dem um eins erhöhten Wortzähler aufgerufen würde. - Handelt es sich bei dem aktuellen Zeichen nicht um ein Worttrennzeichen, so würde es mit seiner Position zurückgeliefert werden und die Funktion `norm_string` würde wiederum mit dem aktuellen Zeichen (das ja kein Trennzeichen ist), dem Restzeichenstrom und dem um eins erhöhten Wortzähler aufgerufen werden.
Handelte es sich bei dem vorherigen Zeichen des Zeichenstroms nicht um ein Worttrennzeichen, so wird, falls das jetzige Zeichen ein Worttrennzeichen ist, ein Leerzeichen in den Zeichenstrom eingehängt, ansonsten das jeweils gelesene Zeichen. Für den Rest des Zeichenstroms wird wiederum die Funktion `norm_string` rekursiv aufgerufen.
Anmerkung
Beim ersten Aufruf von `norm_string` sollte der erste Parameter ein Leerzeichen sein, damit der Wortanfang des ersten Wortes des Zeichenstroms auch als solcher behandelt wird.

Funktionsname:

createWordlist createWordlist

Beschreibung:

Die Hilfsfunktion `createWordlist` erzeugt aus einem Zeichenstrom, der vorher mit der Fuktion `norm_string` normiert wurde, eine Wortliste und wird von der exportierten Funktion `makeWordlist` aufgerufen.

Signatur:

createWordlist:: chars -> specialchars -> integer -> integer -> wordlist

chars: das Wort, das gerade gelesen wird

specialchars: der Zeichenstrom (als `specialchars`) mit den Zeichen und ihren Positionen (in der Internen Struktur)

integer: die laufende Nummer des Wortes, das gerade gelesen wird

integer: die Nummer des zuletzt gelesenen Zeichens

wordlist: die ermittelte liste von Worten

Implementierung:

Zunächst wird mit þattern-matchingünterschieden, ob der zu bearbeitende Zeichenstrom leer ist. In diesem Fall, ist das im ersten Parameter stehende Wort das letzte Wort des bearbeiteten Absatzes (Zeichenstroms) und muß nur noch in die Wortliste eingehängt werden. Da eine Wortliste aus Elementen der Sorte `wordlistelement` besteht, muß dieses Wort also noch zu einer Instanz dieses Datentyps überführt werden. Hierzu benötigt man zunächst das Wort selbst als `string`, die laufende Nummer des Wortes bezüglich des Absatzes und die laufende Nummer des ersten Buchstaben des Wortes bezüglich des Absatzes (so wie er als Zeichenstrom in der Internen Struktur vorliegt). - Die laufende Nummer des Wortes ist der dritte Eingabeparameter der Funktion, da die Worte mitgezählt wurden. Die laufende Nummer des ersten Buchstaben des Wortes ergibt sich aus der laufenden Nummer des zuletzt gelesenen Zeichens (vierter Parameter), verringert um eins weniger als die Länge des Wortes.
Ist der zubearbeitende Zeichenstrom noch nicht leer, so wird das erste Zeichen abgetrennt und überprüft, ob es sich um ein Leerzeichen handelt. Ist dies der Fall, so bedeutet dies, daß das Wort, das gerade gelesen wird, zuende ist und somit das gesamte Wort im ersten Eingabeparameter steht. Dieses Wort wird nun wie oben beschrieben zu einer Instanz der Sorte `wordlistelement` überführt und in die Rückgabewortliste eingehängt. Um nun das nächste Wort zu lesen, wird für den Restzeichenstrom rekursiv die Funktion `createWordlist` mit der leeren Liste als ersten Parameter (damit hier dann das nächste Wort hineingeschrieben werden kann), der laufenden Nummer des nächsten Wortes und der Nummer des zuletzt gelesenen Zeichnes (also des Leerzeichens) aufgerufen.
Handelt es sich bei dem gelesenen Zeichen nicht um ein Leerzeichen, dann ist das Wort, das gerade gelesen wird noch nicht zuende. In diesem Fall wird sofort rekursiv die Funktion `createWordlist` aufgerufen. Der erste Aufrufparameter setzt sich zusammen aus dem vorher schon gelesenen Wort, an das das gelesene Zeichen angehängt wurde, weil es ein Bestandteil dieses Wortes ist. Der zweite Parameter ist der Restzeichenstrom, der Dritte die aktuelle Wortnummer und der Vierte die laufende Nummer des gelesenen Zeichens.

Vor- und Nachbedingungen:

Beim ersten Aufruf von `createWordlist` sollte der erste Parameter die leere Liste sein, weil hier das erste Wort hineingeschrieben werden soll. Jeder andere `string` an dieser Stelle würde mit zum ersten Wort gezählt werden und zu Fehlern führen.
Es ist wichtg, daß der Eingabezeichenstrom von der Funktion `norm_string` normiert wurde, da hintereinanderstehende Leerzeichen leere Worte in der Ausgabewortliste erzeugen würden.

3. Benutzungsoberfläche

3.1 Das ASpecT-Modul UIStructure

Jäschke

3.1.1 Funktionalität

In diesem Modul wird die Struktur samt Zugriffsfunktionen definiert, die später mit set- und getCookieUI-Funktionen der internen Struktur in das Environment von ForauS geschrieben bzw. daraus gelesen werden kann.

3.1.2 Entwurf

Die zentrale Struktur der Benutzungsoberfläche ist ein Dictionary aus der ui Struktur. Darin werden all diejenigen Werte gespeichert, die sich als Integer darstellen lassen. Das heißt in diesem Fall, daß hauptsächlich Zeiger des C-Moduls UI.xc darin aufgehoben werden. Auf das Dictionary gibt es zwei Zugriffsfunktionen: put_c_pointer und get_c_pointer. Des weiteren gibt es eine Funktion mt_ui, die eine leere ui-Struktur liefert.
An vielen Stellen im ASpecT-Modul der Benutzungsoberfläche wäre es sinnvoll, wenn nicht der Umweg über das Dictionary gegangen würde. Besonders das Zwischenspeichern der aktuellen Cursorposition oder des aktuellen Objekts ist somit sehr umständlich und vor allem zeitraubend. Eine entsprechende Umstrukturierung ist möglich und wurde und bisher nur deshalb nicht verwirklicht, weil andere davon betroffene Gruppen noch nicht soweit waren, daß sie die Umstellung hätten vornehmen können.

3.1.3 Öffentlich Schnittstellen

3.1.3.1 Sorten

Sortenname:

c_pointer c_pointer

Signatur:

c_pointer:: (integer).

Beschreibung:

c_pointer ist Bestandteil des Dictionarys in ui. Darin werden alle wichtigen Zeiger des C-Teils des UI Moduls aufbewahrt.

Sortenname:

ui ui

Signatur:

ui:: uiStruct (cpointer :: uiDictCPointer).

Beschreibung:

Dies ist die Struktur, die im Environment von ASpect von der internen Struktur für die Benutzungsoberfläche aufgehoben wird. uiDictCPointer steht für ein Dictionary, welches wie folgt aufgebaut ist:

```
 Dict ACTUAL
SORTS dom = string.
codom = c_pointer.
dictionary = uiDictCPointer.
OPNS errorval = initial_c_pointer.
hash = global_vars_hash.
END
```

Das Dictionary ist wie ein Wörterbuch aufgebaut. In der Domäne string steht das Wort, welches man sucht, und in der Co-Domäne c_pointer die Übersetzung. Diese ist in diesem Wörterbuch ein Integerwert.

Um einen Eintrag schnell wiederzufinden wird ein sogenannter Hash-Wert verwendet. Dieser muß durch eine benutzerdefinierte Funktion bereitgestellt werden, in diesem Fall ist es global_vars_hash. Für den Fall, daß eine dem Dictionary unbekannte Domäne übergeben wird, muß eine Funktion angegeben werden, die diesen Fehler abfängt. In diesem Dictionary erledigt daß initial_c_pointer. Sie gibt einfach minus eins zurück.

Zwei Funktion unterstützen die Arbeit mit dem Dictionary: zum Hineinschreiben wird die Funktion put_c_pointer und zum Lesen get_c_pointer verwendet.

3.1.3.2 Funktionen

Funktionsname:

get_c_pointer get_c_pointer

Signatur:

get_c_pointer:: ui -> string -> (boolean, c_pointer).

ui: Die Struktur, in dem das Dictionary steht, aus dem eine Variable gelesen werden soll.

string: Die Domäne, zu dem die entsprechende Co-Domäne gelesen werden soll.

boolean: Gibt an, ob die Operation erfolgreich war.

c_pointer: Die gefundene Co-Domäne, hier ein Integerwert.

Beschreibung:

Mit dieser Funktion kann ein zuvor mit put_c_pointer im Dictionary plazierter Wert wieder ausgelesen werden.

Funktionsname:

mt_ui mt_ui

Signatur:

mt_ui:: ui.

ui: Diese Struktur wird leer erzeugt.

Beschreibung:

Mit dieser Funktion wird die Struktur ui leer erzeugt. Benötigt wird diese Funktion von der internen Struktur, wenn sie das Environment initialisiert und den Cookie für die Benutzungsoberfläche anlegt.

Funktionsname:

put_c_pointer put_c_pointer

Signatur:

put_c_pointer:: ui -> string -> c_pointer -> ui.

ui: Die Struktur, in dem das Dictionary steht

string: Die Domäne, zu dem die Co-Domäne c_pointer geschrieben werden soll.

c_pointer: Die im Dictionary abzulegende Co-Domäne, in diesem Fall ein Integerwert.

Beschreibung:

Mit dieser Funktion wird ein c_pointer Wert im Dictionary der Struktur ui abgelegt. Dieser kann später mit der Funktion get_c_pointer und dem gleichen string wieder ausgelesen werden.

3.1.4 Lokale Funktionen

Funktionsname:

initial_c_pointer initial_c_pointer

Signatur:

initial_c_pointer:: c_pointer

c_pointer: Immer minus eins.

Beschreibung:

Diese Funktion ist Teil des Dictionarys uiDictCPointer der Struktur ui. Sie wird immer aufgerufen, wenn die angefragte Domäne nicht gefunden wurde. Als Ergebnis liefert sie immer den Integerwert minus eins.

Funktionsname:

global_vars_hash global_vars_hash

Signatur:

global_vars_hash:: string -> integer.

string: Domäne, für die ein Hashwert berechnet werden soll.

integer: Der Hashwert für den string.

Beschreibung:

Aus dem an die Funktion übergebenen string wird ein Hashwert berechnet, mit dem das Dictionary ein schnelles auffinden der Domänen ermöglicht wird. Dazu wird jedes Zeichen des strings genommen und dessen ASCII-Wert über die Funktion sum_ascii zusammengezählt.

Funktionsname:

sum_ascii sum_ascii

Signatur:

sum_ascii:: char -> integer -> integer.

char: Ein Buchstabe einer Zeichenkette.

integer: Die Summe der bisher zusammengezählten ASCII-Werte.

integer: Die neue Summe der zusammengezählten ASCII-Werte.

Beschreibung:

Die Funktion nimmt das übergebene Zeichen, wandelt es in einen Integerwert entsprechend den ASCII-Konventionen um und addiert diesen Wert dann auf den ersten übergebenen Integerwert auf. Die Funktion wird zur Berechnung des Hashwertes des Dictionarys uiDict-CPointer benötigt.

3.2 Das ASpecT-Modul UI

Jäschke

3.2.1 Funktionalität

Das Modul User Interface dient der Kommunikation zwischen dem Anwendendem und dem Programm ForauS . Dabei bedient es sich dem in C programmierten Modul UI.xc, welches die Verbindung zu X-Windows herstellt. Ein paar Funktionen wurden von den Gebrüdern Schmidtke beschrieben.

3.2.2 Entwurf

Die komplette Bedienung des ForauS -Editors geschieht über die Benutzungoberfläche die X-Windows zur Verfügung stellt. Da es bisher keine vernünftige Möglichkeit gibt mit ASpecT auf das X System zuzugreifen, wurden alle zur Verwaltung nötigen Routinen in C formuliert. X-Windows selbst ist ein ereignisorientiertes Betriebssystem. Führt der Benutzende eine Aktion aus, wird dies an das entsprechende C-Modul signalisiert [1]. Dieses muß nun entsprechend reagieren, und das heißt in unserem Fall zumeist, daß eine Funktion aus dem Modul UI.AS gerufen wird. Diese Funktion entscheidet dann, was zu tun ist, um das gewünschte Ergebnis zu bekommen. Fast immer greift sie dazu auf Funktionen zu, die in den anderen Modulen von ForauS definiert wurden. Um nur ein simples Beispiel zu nennen: Für das Einfügen eines einzelnen Buchstaben im Text (Ereignis „Taste gedrückt“) werden Funktionen aus den Modulen Interne Struktur, Formatierer und Ausgabe benötigt.
Zusammenfassend kann man also sagen, daß das Modul UI die Verbindung zwischen X-Windows bzw. dem Benutzer und den restlichen Modulen von ForauS herstellt. Es ist zugleich eine Klammer für diese Module, die erst durch die Verknüpfung im User Interface sinnvoll werden.
Alle im Kapitel „Externe Funktionen“ angegebenen Routinen entstammen dem Modul UI.xc.

[1] Konkret heißt das, daß eine zuvor vom Programmierer für diese Aktion vorgegebene Funktion, ein sogenannter Notifier oder Callback, aufgerufen wird.

3.2.3 Öffentlich Schnittstellen

3.2.3.1 Sorten

Sortenname:

prozent prozent

Signatur:

prozent:: (integer).

Beschreibung:

Diese Variable enthält in Prozent den Grad der Verarbeitung in einer Funktion. Jener kann dann in regelmäßigen Abständen an die Funktion zur Auffrischung des Zeitbalkens (u_ActTimereq) weitergeleitet werden. Da aber diese Funktion zur Zeit in diesem Modul noch nicht genutzt wird, bleibt diese Variable noch ungenutzt.

Sortenname:

status status

Signatur:

status:: (boolean).

Beschreibung:

Diese Variable empfängt von der externen Funktion zur Auffrischung des Timerequesters (u_ActTimereq) einen boolschen Wert. Standardmäßig ist diese Variable auf logisch falsch gesetzt. Sobald jedoch ein Benutzer diese Funktion abbrechen will, wird sie auf wahr gesetzt. Derzeitig wird der Zeitbalken von keiner Routine aus diesem Moduls gerufen, so daß diese Variable unbenutzt bleibt.

Sortenname:

statustext statustext

Signatur:

statustext:: (string).

Beschreibung:

statustext enthält einen Text, der beim Öffnen des Timerequester oberhalb des Prozentbalkens gedruckt wird. Wie auch für die Sorten status und prozent gilt, daß der Timerequester in diesem Modul noch nicht im Einsatz ist und die Variablen deshalb auch nicht verwendet werden.

3.2.3.2 Funktionen

Funktionsname:

afterLoad afterLoad

Signatur:

afterLoad:: env -> env.

env: ASpecT-Environment des $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$ -Systems.

Beschreibung:

Die Funktion erledigt alle nötigen Operationen nach dem Laden einer Datei: Neusetzen des Cursors, Bestimmung des ersten finalen Elementes, Aktualisieren der Menüs und des Autosave-Zeitgebers, Anzeige der Seitennummerierung und des Status der Lade-Operation.

Implementierung:

Der Aufruf erfolgt von der C-Seite, siehe Datei UI.xc.

Funktionsname:

event_KeyAscii event_KeyAscii

Signatur:

event_KeyAscii:: env -> integer -> env.

env: ASpecT-Environment des $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$ -Systems.

integer: Scanwert der gedrückten Taste.

Beschreibung:

Wird eine Taste gedrückt, deren Bedeutung einem Zeichen aus dem ASCII-Zeichensatz entspricht, dann wertet diese Funktion den übergeben Integerwert aus. Handelt es sich um ein druckbares Zeichen, z.B. den Buchstaben „A“, wird es an der aktuellen Cursorposition eingefügt. Die Seite wird neu formatiert und angezeigt, der Cursor um ein Zeichen weiterbewegt.

Implementierung:

Diese Funktion wird aus dem C-Modul UI.xc gerufen. Die Funktion heißt canvas_event_proc.

Funktionsname:

event_KeyCopy event_KeyCopy

Signatur:

event_KeyCopy:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Kopiert den markierten Block in den Blockpuffer. Von dort aus kann er mittels der Paste-Taste an anderer Stelle wieder eingefügt werden.

Implementierung:

Diese Funktion ist noch nicht implementiert worden.

Funktionsname:

event_KeyCut event_KeyCut

Signatur:

event_KeyCut:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Ein zuvor markierter Block wird in den Blockpuffer geschrieben. Der Block im Text wird anschließend gelöscht. Ist kein Text markiert, wird das Element, auf dem der Cursor steht, gelöscht.

Implementierung:

Die Funktion unterstützt noch keine Blockoperationen. Das heißt, daß markierte Texte oder Elemente zwar aus dem laufenden Text ausgeschnitten aber nicht in den Blockpuffer kopiert werden.

Funktionsname:

event_KeyDeleteElement event_KeyDeleteElement

Signatur:

event_KeyDeleteElement:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Löschen des logischen Elements, das per Block markiert wurde. War das Löschen erfolgreich, wird die Seite neu formatiert und dargestellt. Die Aktion wird über Ctrl+Backspace ausgelöst.

Implementierung:

Vor dem Löschen wird zunächst in der internen Struktur angefragt, ob ein Löschen möglich ist. Wenn ja, werden über Formatiererbefehle die betroffen Elemente entfernt. Danach wird der Cursor gelöscht, die Seite neu formatiert und dann wieder neu aufgebaut.

Funktionsname:

event_KeyDeleteLast event_KeyDeleteLast

Signatur:

event_KeyDeleteLast:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Löscht das Zeichen links vom Cursor (Backspace-Taste). Die Seite wird inkremental neu formatiert und dargestellt.

Funktionsname:

event_KeyDeleteNext event_KeyDeleteNext

Signatur:

event_KeyDeleteNext:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Funktion für die Delete-Taste. Es wird das Zeichen rechts vom Cursor gelöscht (s.a. event_-KeyDeleteLast). Die aktuelle Seite wird inkremental neu formatiert und angezeigt.

Funktionsname:

event_KeyDocStart event_KeyDocStart

Signatur:

event_KeyDocStart:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Mit dieser Funktion wird an den Anfang eines Dokuments gesprungen. Ein markierter Block wird demarkiert. Wird die Seite gewechselt, wird die neue, anzuzeigende Seite formatiert und dargestellt. Der Cursor steht jetzt in der ersten Zeile des Dokuments. Diese Funktion wird zur Zeit vom Formatierer nicht unterstützt.

Funktionsname:

event_KeyDocEnd event_KeyDocEnd

Signatur:

event_KeyDocEnd:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Es wird die letzte Seite des Dokuments angezeigt. Ist ein Block markiert, wird dieser zunächst demarkiert. Die Seite wird angezeigt und der Cursor steht in der letzten Zeile.

Funktionsname:

event_KeyDown event_KeyDown

Signatur:

event_KeyDown:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Cursorbewegung abwärts. Noch vor jeder anderen Aktion wird ein möglicherweise markierter Block demarkiert. Erst dann wird per f_moveCursor der Cursor nach unten bewegt. Sollte die Schreibmarke dabei die aktuelle Seite verlassen, wird die nächste Seite formatiert und angezeigt.

Funktionsname:

event_KeyFind event_KeyFind

Signatur:

event_KeyFind:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Suchen von markierten Textstücken oder logischen Elementen.

Implementierung:

Diese Funktion ist noch nicht implementiert worden.

Funktionsname:

event_KeyLeft event_KeyLeft

Signatur:

event_KeyLeft:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Wird die Pfeiltaste links gedrückt, so wird der Cursors um ein Zeichen nach links bewegt. Die Markierung eines Block wird entfernt (falls vorhanden) und dann werden die aktuellen Koordinaten der f_moveCursor-Funktion übergeben. Wird die aktuelle Seite verlassen, erfolgt das Kommando zum Neuzeichnen.

Funktionsname:

event_KeyLineStart　　event_KeyLineStart

Signatur:

event_KeyLineStart:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Diese Funktion setzt den Cursor an den Anfang der gerade aktuellen Zeile.

Funktionsname:

event_KeyLineEnd　　event_KeyLineEnd

Signatur:

event_KeyLineEnd:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Es wird mit dem Cursor an das Ende der aktuelle Zeile gesprungen.

Funktionsname:

event_KeyPageUp　　event_KeyPageUp

Signatur:

event_KeyPageUp:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Seitenweises Blättern vorwärts. Bei Aufruf der Funktion wird ein markierter Block demarkiert und danach auf die nächste Seite gesprungen. Wird dabei das Objekt gewechselt, wird die Anzeige im Fußbereich des Fensters aufgefrischt.

Funktionsname:

event_KeyPageDown event_KeyPageDown

Signatur:

event_KeyPageDown:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Die Funktion blättert auf die vorhergehende Seite. Dazu wird erst ein markierter Block demarkiert. Die anzuzeigende Seite wird formatiert und dargestellt.

Funktionsname:

event_KeyPaste event_KeyPaste

Signatur:

event_KeyPaste:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Es wird der mit event_KeyCopy festgehaltene Blockpuffer an der Stelle eingefügt, an der der Cursor steht. Ist ein Block markiert, wird dieser vorher gelöscht. Der Inhalt des Blockpuffers ersetzt praktisch den im Text markierten Block.

Vor- und Nachbedingungen:

Der Blockpuffer muß zunächst mit event_KeyCopy eingefangen werden.

Implementierung:

Diese Funktion ist noch nicht implementiert worden.

Funktionsname:

event_KeyReturn event_KeyReturn

Signatur:

event_KeyReturn:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Diese Funktion nimmt das Einfügen eines neuen logischen Elements vor und setzt dann den Cursor auf auf den Anfang des neuen Elements. Das neue Element hat den gleichen Typ wie das Element, auf dem der Cursor zuvor stand. Es wird, wenn möglich, automatisch eingefügt.

Implementierung:

Diese Funktion ist noch nicht implementiert worden.

Funktionsname:

event_KeyRight event_KeyRight

Signatur:

event_KeyRight:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Genau wie bei der Funktion event_KeyRight wird beim Drücken der Pfeiltaste rechts die Position des Cursors um ein Zeichen geändert, jedoch dieses mal nach rechts. Der alte Block wird demarkiert (wenn vorhanden), der Cursor mit der f_moveCursor-Funktion bewegt. Wird die aktuelle Seite verlassen, erfolgt das Kommando zum Neuzeichnen.

Funktionsname:

event_KeyUp event_KeyUp

Signatur:

event_KeyUp:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Diese Funktion führt die Cursorbewegung aufwärts aus. Dazu wird als erstes ein markierter Block demarkiert. Mit der f_moveCursor-Funktion wird dann der Cursor tatsächlich bewegt. Verläßt der Cursor die aktuelle Seite, wird der Bildschirm neu aufgebaut.

Funktionsname:

event_KeyWordLeft event_KeyWordLeft

Signatur:

event_KeyWordLeft:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Diese Funktion bewegt den Cursor, im Gegensatz zu KeyLeft, gleich ein ganzes Wort nach links. Nach dem Demarkieren eines eventuell markierten Blocks wird mit der f_moveCursor-Funktion der Cursor um ein Wort nach links bewegt. Sollte dadurch die Seite wechseln wird neu formatiert und die vorherige Seite angezeigt.

Funktionsname:

event_KeyWordRight event_KeyWordRight

Signatur:

event_KeyWordRight:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Cursorbewegung um ein Wort nach rechts. Nach dem Demarkieren eines markierten Blocks wird der Cursor auf das nächste Wort bewegt. Springt er dabei auf die nächste Seite, wird diese formatiert und angezeigt.

Funktionsname:

event_Menu_Select_Element event_Menu_Select_Element

Signatur:

event_Menu_Select_Element:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Die Funktion markiert das Objekt, auf dem der Cursor steht. Ein eventuell schon markierter Block wird vorher demarkiert.

Funktionsname:

event_MouseLeft event_MouseLeft

Signatur:

event_MouseLeft:: env -> integer -> integer -> env.

env: ASpecT-Environment des ForauS -Systems.

integer: Horizontale Position des graphischen Mauszeigers.

integer: Vertikale Position des graphischen Mauszeigers.

Beschreibung:

Die Funktion wird aufgerufen, wenn mit der linken Maustaste der Textcursor neu gesetzt wird. Die Mauszeigerposition wird dazu in interne Koordinaten umgerechnet und daraus dann die neue Cursorposition berechnet. Der alte Cursor wird gelöscht, eine eventuell vorhandene Blockmarkierung gelöscht und der Cursor anschließend wieder gesetzt.

Funktionsname:

event_MouseMiddle event_MouseMiddle

Signatur:

event_MouseMiddle:: env -> integer -> integer -> env.

env: ASpecT-Environment des ForauS -Systems.

integer: Horizontale Position des graphischen Mauszeigers.

integer: Vertikale Position des graphischen Mauszeigers.

Beschreibung:

Die mittlere Maustaste erweitert die Blockmarkierung. Zunächst werden die Parameter, die die Mausposition auf dem Bildschirm repräsentieren, in interne Koordinaten umgerechnet. Der neu zu invertierende Bereich geht von der alten Block-Start-Position bis zur neuen Cursorposition, falls diese sich unterhalb der alten befindet (Bereichserweiterung bzw. Verkleinerung nach unten, Block-Ende-Marke verschiebt sich). Ansonsten (der Cursor steht oberhalb der Block-Ende-Position) geht er von der neuen Cursorposition bis zur alten Block-Ende-Position (Bereichserweiterung bzw. Verkleinerung nach oben).

Funktionsname:

event_Print event_Print

Signatur:

event_Print:: env -> integer -> string -> string -> integer -> integer -> integer -> integer -> env.

env: ASpecT-Environment des ForauS -Systems.

integer: Bestimmt die Ausgabeeinheit:

Wert	Bedeutung
0	Ausdruck auf dem Postscript-Drucker
1	Ausgabe in eine Druckdatei.

string: Pfad für Druckdatei (unbenutzt).

string: Name der Druckdatei (unbenutzt).

integer: Gibt Auskunft darüber, welcher Seitenbereich gedruckt werden soll.

Wert	Bedeutung
0	In diesem Fall bestimmen die nachfolgenden zwei Parameter (Anfangsseite und Endseite) von welcher Seite bis zu welcher Seite gedruckt werden soll.
1	Drucken aller Seiten des Dokuments.

integer: Erste zu druckende Seite.

integer: Letzte zu druckende Seite.

integer: Anzahl der Kopien, die von jeder Seite gemacht werden sollen (unbenutzt).

Beschreibung:

Aufruf der Postscript-Druckfunktion. Die Parameter kommen von den X-Objekten und werden bei Betätigung der Print-Taste hierher durchgereicht.

Funktionsname:

event_KeyToggleInsertMode event_KeyToggleInsertMode

Signatur:

event_KeyToggleInsertMode:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Umschalten des Einfügemodus. Ist das Überschreiben aktiv, wechselt der Aufruf dieser Funktion auf Einfügen und vice versa.

Implementierung:

Diese Funktion ist noch nicht implementiert worden.

Funktionsname:

event_KeyUndo event_KeyUndo

Signatur:

event_KeyUndo:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Zurücknehmen einer oder mehrer Aktionen des Benutzers.

Implementierung:

Diese Funktion ist noch nicht implementiert worden.

Funktionsname:

get_global_var get_global_var

Signatur:

get_global_var:: string -> env -> c_pointer.

string: Name der Variable, die aus dem Dictionary für Zeiger im C-Code gelesen werden soll.

env: ASpecT-Environment des ForauS -Systems.

c_pointer: Das Ergebnis ist ein Integer-Wert, der mit Casting unter C als Zeiger verwendet werden kann.

Beschreibung:

get_global_var ist immer im Zusammenhang mit put_global_var zu sehen. Ein mit letzterer Funktion in das Dictionary geschriebener C-Pointer kann mit dieser Funktion wieder ausgelesen werden. Durch das Lesen wird der Wert im Dictionary nicht gelöscht! Er bleibt solange unverändert, bis ein erneutes put_global_var mit string einen anderen Wert hineinschreibt.

Vor- und Nachbedingungen:

Einem Lesen mit get_global_var muß mindestens ein put_global_var vorausgehen. Das System gerät sonst in einen undefinierten Zustand, bishin zum Absturz. Ist die mit string beschriebene Variable noch nicht in das Dictionary eingetragen worden, wird eine Fehlermeldung auf dem Standardfehlerkanal ausgegeben.

Funktionsname:

is_crash is_crash

Signatur:

is_crash:: daVinci_answer -> boolean.

daVinci_answer : Meldung von daVinci. Eine Auflistung aller möglichen Antworten findet man im Modul daVinci.AS bzw. in der zugehörigen Dokumentation.

boolean : Ergebnis der Auswertung. Ist das Ergebnis wahr, bedeutet das, daß ein Fehler auftrat.

Beschreibung:

Die Funktion analysiert die daVinci-Nachricht (genauer die mitgelieferten Statusinformationen) um herauszufinden, ob sich bei der Übertragung ein Fehler ereignet hat. Das Ergebnis wird als Wahrheitswert zurückgegeben.

Funktionsname:

process_answer process_answer

Signatur:

process_answer:: (daVinci_answer, env) -> env.

daVinci_answer: Zeichenkette, die von daVinci über die Interprozess-Kommunikation an FORAUS gesendet wird und ausgewertet werden soll.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Die Funktion ueberprüft die Nachrichten von daVinci daraufhin, ob sie das Schlüsselwort „node_selections_labels“ enthalten (d.h. der Benutzer hat einen Knoten im daVinci-Graphen selektiert). Ist das der Fall, wird das gewählte Element unter ForauS markiert und dieser Vorgang durch eine Meldung dem Benutzer mitgeteilt. Bei anderen Nachrichtentypen gibt ForauS eine Warnmeldung aus.

Funktionsname:

put_global_var　　　　put_global_var

Signatur:

put_global_var:: string -> c_pointer -> env -> env.

string: Eine eindeutige Zeichenkette, unter deren Namen c_pointer abgelegt wird.

c_pointer: Ein Integer-Wert, bzw. ein C-Pointer, der ja letztlich nichts anderes ist.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

c_pointer wird in einem Dictionary unter der Bezeichnung string abgelegt und aufgehoben. Mit der Umkehrfunktion get_global_var kann die Variable wieder ausgelesen werden.

Vor- und Nachbedingungen:

put_global_var für eine Variable string muß mindestens einmal vor dem ersten Aufruf von get_global_var mit der gleichen Variablen string aufgerufen werden.

Implementierung:

Das oben angesprochene Dictionary wird in UISTRUCT.AS definiert und heißt ui. Aufbewahrt wird es im Environment env. Mit Funktionen der internen Struktur wird es zunächst ausgelesen (getCookieUI) und nach der Modifikation wieder zurückgeschrieben (i_setCookieUI).

Funktionsname:

read_answer　　　　read_answer

Signatur:

read_answer:: string -> env -> (daVinci_answer, env).

string: Zeichenkette, die von daVinci über die Interprozess-Kommunikation an ForauS gesendet wird und ausgewertet werden soll.

env: ASpecT-Environment des ForauS -Systems.

daVinci_answer : Meldung von daVinci ohne zusätzliche Statusinformationen.

Beschreibung:

Liest Daten von stdin und überprüft sie auf Richtigkeit durch Auswertung der zusätzlichen Statusinformationen von daVinci. Im Fehlerfall wird das Programm verlassen, da sonst eine instabile Prozesskommunikation aufrechtgehalten würde.

Vor- und Nachbedingungen:

Die Prozesskommunikation zu daVinci muß vorher korrekt aufgebaut worden sein.

Implementierung:

Die Funktion wird von der C-Seite (Modul UI.xc) aufgerufen. Sie beinhaltet nur die Fehlerüberprüfung, nicht die Auswertung der eigentlichen Nachrichten. Siehe auch Funktion process_answer.

Funktionsname:

set_ViewMenu set_ViewMenu

Signatur:

set_ViewMenu:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Auf der Benutzungsoberfläche wird im Sichtenmenü eine Auswahl aller möglichen Sichten angezeigt. Mit dieser Funktion wird die Auswahl aktualisiert. Die Informationen dazu kommen aus der internen Struktur.

Implementierung:

Um die Liste anzeigen zu können wird zunächst aus der internen Struktur eine Liste der gegenwärtig erlaubten Sichten angefordert. Diese wird dann in eine Subfunktion gesteckt: set_ViewMenu2.

Funktionsname:

show_possibleElements show_possibleElements

Signatur:

show_possibleElements:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Zeigt für die aktuelle Cursorposition an, an welcher Position (davor, dahinter, dazwischen, ...) welche logischen Objekte der DTD eingefügt werden können. Der Benutzer kann nun eines wählen und an der entsprechende Stelle einfügen.

Funktionsname:

stdin_handler stdin_handler

Signatur:

stdin_handler:: string -> env -> env.

env: ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

string: Antwort nach dem Schema von daVinci.AS.

Beschreibung:

Der stdin_handler empfängt die Nachrichten, die von daVinci kommen und wertet sie aus. Siehe auch read_answer und process_answer.

Funktionsname:

test_error test_error

Signatur:

test_error:: env -> (env, integer).

env: ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

integer: Nummer des Fehlertyps.

Beschreibung:

Testet, ob ein Fehler in der Fehlerliste steht und gibt ihn als Meldung an den Benutzer weiter. Mit der zurückgehenden Fehlertypnummer kann ein Programm auf den Fehler mit dem höchsten Fehlertyp reagieren.

Implementierung:

Die Funktion verwendet zum Anzeigen des Fehlers test_error2.

Funktionsname:

u_insert_element u_insert_element

Signatur:

u_insert_element:: env -> integer -> env.

env: ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

integer: Nummer des gewählten Listeneintrags und damit indirekt das einzufügende Element.

Beschreibung:

Einfügen eines logischen Elements an der Cursorposition im aktuellen Element, vor dem Element oder dahinter.

Implementierung:

Wenn der Benutzer auf der Benutzungsoberfläche die Liste aller an der aktuellen Cursorposition einfügbaren Elemente aufgeklappt, ein Element selektiert und die Auswahl bestätigt hat, dann wird diese Routine aufgerufen. Die rufende Routine im Modul UI.xc heißt ui_which_element().

Neben dem obligatorischen Environment bekommt u_insert_element den Index des Listeneintrages in der Elementauswahlliste als Parameter rein. Mit i_requestInsertPosition besorgt sich die Funktion nun die Liste der möglichen Einfügepositionen und bestimmt damit und mit der Nummer des gewählten Listeneintrages das einzufügende logische Element. Dann endlich kann das Element eingefügt und die aktuelle Seite neu dargestellt werden.

Funktionsname:

u_writeDocumentGraph u_writeDocumentGraph

Signatur:

u_writeDocumentGraph:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Es wird die graphische Visualisierung des Dokuments in daVinci aufgefrischt.

Implementierung:

Immer wenn ein Dokument geladen oder das logische Element gewechselt wird sollte diese praktische kleine Routine gerufen werden. Sie fordert von der internen Struktur einen Dokumentengraphen an, versieht ihn mit einem Befehl, der daVinci sagt, daß sich die Struktur des Graphen (dazu gehört auch die Hervorhebung des gerade aktiven Elements!) geändert hat und gibt dann dieses Ergebnis mit dem ASpecT-Befehl write als Zeichenkette aus. Daran schließt sich die Ausgabe einer leeren Zeichenkette mit `@` an. Damit wird ein LineFeed-Zeichen gesendet, welches daVinci zum Erkennen des Zeilenendes benötigt. Da unglücklicherweise durch einen neuen Graphen auch automatisch die Fensterüberschrift im daVinci-Fenster gelöscht wird, muß eine weitere Ausgabe diesen wieder setzen.

Anmerkung: es wird nicht sichergestellt, ob daVinci bereits geladen wurde. Sollte dies nicht der Fall sein, werden im Normalfall bei Aufruf dieser Routine einige wilde Zeichen im Shell-Fenster erscheinen. Dies hat keine Auswirkung auf den Programmablauf von ForauS .

Funktionsname:

write_command write_command

Signatur:

write_command:: daVinci_command -> env -> env.

daVinci_command : Abzusendendes Kommando für daVinci.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Es wird ein daVinci-Kommando über die stdout-Pipe gesendet.

Vor- und Nachbedingungen:

Die Prozesskommunikation zu daVinci muß vorher korrekt aufgebaut worden sein.

Implementierung:

Momentan wird die Funktion nicht verwendet, da die Kommunikation mit daVinci meist über C-Schnittstellen erfolgt.

3.2.4 Lokale Funktionen

Funktionsname:

arrangeElements arrangeElements

Signatur:

arrangeElements:: env -> listOfStrings -> insertPos -> env.

env: ASpecT-Environment des ForauS -Systems.

listOfStrings : Liste mit Objektnamen, die die genaue Einfügeposition des einzufügenden Objekts näher beschreiben.

insertPos : Position, an der das selektiert Objekte eingefügt werden kann.

Beschreibung:

arrangeElements dient zur Erzeugung eines zusammenhängenden Textes, der dem Benutzer angezeigt wird, wenn ein logisches Element eingefügt werden soll. Die Einfügeposition (davor, Cursorpositiontion, ...) wird dadurch dem Benutzer genauer beschrieben.

Funktionsname:

arrangeElements2 arrangeElements2

Signatur:

arrangeElements2:: env -> listOfStrings -> insertPos -> integer ->(boolean, strings, env).

env: ASpecT-Environment des ForauS -Systems.

listOfStrings : Liste mit Objektnamen, die die genaue Einfügeposition des einzufügenden Objekts näher beschreiben.

insertPos : Position, an der das selektiert Objekte eingefügt werden kann.

integer : Listenindex des vom Benutzer ausgewählten Menüeintrags.

Beschreibung:

Dient zur Erzeugung eines zusammenhängenden Textes, der die Einfügeposition eines Elementes dadurch genauer beschreibt. Anders als die Funktion arrangeElements gelangt der Text nicht zur Anzeige, sondern dient der Überprüfung und Rückverfolgung der vom Benutzer getätigten Eingabe zum tatsächlich intern repräsentierten Objekt.

Funktionsname:

bit_is_set bit_is_set

Signatur:

bit_is_set:: integer -> integer -> boolean.

integer : Wert, dessen einzelne Bits untersucht werden soll.

integer : Zweierpotenz des zu untersuchenden Bits.

boolean : Boolsches Ergebnis: TRUE, falls das ausgewählte Bit gesetzt ist, ansonsten FALSE.

Beschreibung:

Testet, ob ein bestimmtes Bit eines binären Wertes gesetzt ist.

Funktionsname:

change_marked_Block change_marked_Block

Signatur:

change_marked_Block:: env -> integer -> integer -> env.

env: ASpecT-Environment des ForauS -Systems.

integer : Aktuelle X-Koordinate des Grafik-Cursors (der Mauszeiger), wie vom X-System (relativ zum umgebenden Fenster) gemeldet.

integer : Aktuelle Y-Koordinate des Grafik-Cursors (der Mauszeiger), wie vom X-System (relativ zum umgebenden Fenster) gemeldet.

Beschreibung:

Markiert einen Textbereich auf dem Bildschirm unter Berücksichtigung der jetzigen Cursorposition und der alten Blockkoordinaten. Bei gedrückter linker Maustaste wird der Blockbereich erweitert. Unterschieden werden folgende Fälle:

- Der Grafikcursor wird hinter den Blockstart gesetzt, d.h. das Blockende verschiebt sich.
- Der Grafikcursor wird vor den Blockstart gesetzt, d.h. der Blockstart verschiebt sich. Blockende ist Blockstart.
- Der Grafikcursor wird auf den Blockstart gesetzt, d.h. das Zeichen unter dem Cursor wird markiert.

Vor- und Nachbedingungen:

Blockstart-Koordinaten müßen vorher gesetzt werden.

Implementierung:

Die (lokale) Funktion findet keine Verwendung (mehr), da die Blockmarkierung auf Textebene aufgrund von Schwierigkeiten im Formatierer- / Ausgabe-Modul nicht mehr vollständig entwickelt wurde. Im wesentlichen liegt dies an der Unterscheidung der Zeilenbreite und der Spaltenbreite bei variablen Größen, sowie der problematischen Erkennung von Elementgrößen / Blockgrößen über mehrere Seiten, als auch das Überschneiden von verscheidenen Element-Blcken in der gleichen Zeile. Die Funktion wird von C-Seite aufgerufen.

Funktionsname:

clear_marked_Block clear_marked_Block

Signatur:

clear_marked_Block:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Hebt die Markierung eines Blocks, sofern vorhanden, auf. Folgende Situationen werden von der Funktion erkannt und unterstützt:

- Blockstart und Blockende befinden sich auf der aktueller Seite.
- Der Block liegt nicht auf der sichtbaren Seite.
- Ein markierter Block läuft über mehrere Seiten. In diesem Fall gibt es nochmal vier verschiedene Situationen:
  - der Blockstart liegt auf der sichtbaren Seite;
  - das Blockende liegt auf der sichtbaren Seite;
  - der Blockstart liegt vor der sichtbaren Seite;
  - das Blockende liegt hinter der sichtbaren Seite;

Implementierung:

Das Aufheben einer Blockmarkierung ist einfach zu bewerkstelligen. Es wird nämlich nichts anderes gemacht als bei der vorhergehenden Markierung eines Blocks. Durch das wiederholte Invertieren derselben Passage wird die Markierung aufgehoben.

Funktionsname:

concatElementNames concatElementNames

Signatur:

concatElementNames:: string -> strings -> string.

string : Ausgangs-Zeichenkette, woran rechtbündig weitere Zeichenketten angefügt werden.

strings : Liste von Zeichenketten, die rechtbündig an die Ausgangs-Zeichenkette angefügt werden.

string : Resultierende Zeichenkette.

Beschreibung:

Zusammenfügen von Zeichenketten mit der Eigenschaft, daß die einzelnen Zeichenketten durch ein „and“ verbunden werden. Dies dient der Generierung einer Meldungs-Zeile für den Benutzer beim Einfügen bzw. Löschen von Elementen innerhalb eines Dokumentes. Beim Einfügen eines Elementes E1, womit gleichzeitig auch die Elemente E2 und E3 zwangsweise (aufgrund der logischen Struktur eines Dokumentes) erzeugt werden müssen, wird zum Beispiel die Ausgabe „E1 and E2 and E3“ generiert.

Implementierung:

Die Meldung wird auch dann richtig erstellt, wenn keine Zeichenkette konkateniert werden konnte (bei nur einem möglichen Element). Die Funktion ist nur lokal verfügbar.

Funktionsname:

concatStrings concatStrings

Signatur:

concatStrings:: string -> strings -> string.

string: Basiszeichenkette.

strings: Liste von string-Argumenten, die mit der Basiszeichenkette verbunden werden.

Beschreibung:

Die Funktion verbindet Zeichenketten miteinander. Das Ergebnis ist eine einzige neue Zeichenkette. Wird strings als leere Liste übergeben erhält man als Ergebnis den Eingabewert von string zurück.

Funktionsname:

convertErrType convertErrType

Signatur:

convertErrType:: errType -> string.

errType: Fehlertyp wie im Modul Error beschrieben.

string: Der Typ des Fehler im Klartext.

Beschreibung:

Es wird ein Fehlertyp in eine Klartextzeichenkette übersetzt. Der Aufruf von convertErrType mit dem Fehlertyp fatal als Parameter beispielsweise würde den String „fatal“ zurückliefern.

Funktionsname:

convTypeToNum convTypeToNum

Signatur:

convTypeToNum:: errType -> integer.

errType: Der umzuwandelnde Fehlertyp.

integer: Numerischer Äquivalenzwert zum Fehlertyp.

Beschreibung:

Der Fehlertyp wird in einen Integerwert konvertiert. Je schwerwiegender der Fehler, desto höher der Wert:

Wert	Fehlertyp
0	fatal
1	note
2	warning
3	repair
4	restriction
5	fatal

Funktionsname:

create_pagelist create_pagelist

Signatur:

create_pagelist:: integer -> integer -> integers -> integers.

integer: Erste zu druckende Seite.

integer: Letzte zu druckende Seite.

integers: Liste von Seitenzahlen

Beschreibung:

Erzeugt eine Liste die alle Seitenzahlen enthält, die gedruckt werden sollen.

Funktionsname:

doitforallPositions doitforallPositions

Signatur:

doitforallPositions:: env -> insertPosList -> env.

env: ASpecT-Environment des ForauS -Systems.

insertPosList : Liste mit Positionen, an denen Objekte eingefügt werden können.

Beschreibung:

Dient zur Ermittlung derjenigen Objekte, die an den bestimmten Positionen eingefügt werden dürfen. Sie werden angezeigt und gelangen später zur Auswahl für den Benutzer. Siehe auch doitforallPositions2.

Implementierung:

Dies ist eine lokale Hilfsfunktion der Funktion show_possibleElements.

Funktionsname:

doitforallPositions2 doitforallPositions2

Signatur:

doitforallPositions2:: env -> insertPosList -> integer -> (insertPos, strings, env).

env: ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

insertPosList : Liste mit Positionen, an denen Objekte eingefügt werden können.

integer : Listenindex des Eintrages der Auswahlliste, die zuvor dem Benutzer zur Auswahl angezeigt wurde.

insertPos : Position, an der das selektiert Objekte eingefügt werden kann.

strings : Nähere Angaben zum Objekt (Text aus der Auswahlliste).

Beschreibung:

Die Funktion dient zur Ermittlung derjenigen Objekte, die an den bestimmten Positionen eingefügt werden dürfen. Sie werden im Gegensatz zur Funktion doitforallPositions nicht angezeigt und gelangen später nicht zur Auswahl für den Benutzer. doitforallPositions2 dient ausschließlich zur Rückverfolgung der ausgewählten Elemente aus der Auswahlliste, die der Benutzer zuvor angezeigt bekommen hat. Siehe auch doitforallPositions.

Implementierung:

Dies ist eine lokale Hilfsfunktion der Funktion show_possibleElements.

Funktionsname:

errornumToString errornumToString

Signatur:

errornumToString:: modulName -> errNumber -> errTexts -> string -> string.

modulName : Das $F^{or}a^{u}S$ -Modul, in dem ein Fehler aufgetreten ist, welcher dem Benutzer gemeldet werden soll.

errNumber : Fehlernummer des Moduls.

errTexts : Liste mit Fehlertexten, die der genaueren Erläuterung der Fehlernummer dienen.

string : Zeichenkette, die als Maske für die Ausgabe dient, falls kein anderer Fehlertext gefunden werden konnte.

string : Ergebnis der Funktion.

Beschreibung:

Stellt eine Fehlermeldung zusammen. Je nach Modul und Fehlernummer wird eine Fehlertext-Maske mit den kontextspezifischen Fehler Zusatzangaben aufgefüllt. Gibt es keine passende Maske, wird der übergebene Vorgabewert zurückgeliefert.

Implementierung:

Die Funktion kann an verschiedenen Stellen weiterentwickelt werden, wenn die anderen ForauS -Module bessere und mehr Fehlerangaben machen. Bislang ist dies nicht für alle ForauS -Module gelungen.

Funktionsname:

get_parentid get_parentid

Signatur:

get_parentid:: insertPos -> objID.

insertPos : insertPos-Struktur des zu untersuchenden Obejktes.

objID : Resultierendes Vorgänger-Objekt.

Beschreibung:

Es wird aus der insertPos-Struktur eines Objekts dessen Vaterobjekt isoliert.

Funktionsname:

init_timer init_timer

Signatur:

init_timer:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Initialisiert die Timer-Funktion, die für die automatische Speicherung zuständig ist.

Implementierung:

Diese Funktion ist ein Ersatz für die inkorrekten Timerfunktionen des ASpecT Systems. Siehe auch afterLoad.

Funktionsname:

is_odd is_odd

Signatur:

is_odd:: integer -> boolean.

integer : Wert, der untersucht werden soll.

boolean : Boolsches Ergebnis: TRUE, falls der Integerwert ungerade ist, ansonsten FALSE.

Beschreibung:

Testet, ob die Integerzahl ungerade ist.

Funktionsname:

mark_element_block mark_element_block

Signatur:

mark_element_block:: env -> objID -> env.

env: ASpecT-Environment des ForauS -Systems.

objID: Identifikationsnummer des betroffenen Elements.

Beschreibung:

Markiert im Textfenster ein logisches Element oder hebt dessen Markierung wieder auf. Ob bereits markiert wurde, erkennt die Funktion selbständig.

Implementierung:

Das vollautomatische Erkennen einer Markierung wird durch eine globale Variable im Dictionary ui erreicht. Sie heißt foraus_Block_is_marked und wird immer dann auf eins gesetzt, wenn ein Block markiert wird. Ein Wert Null sagt aus, daß kein Block spezifiziert wurde. Das Invertieren der gewünschten Passage wird von Ausgabe realisiert. Die Funktion dazu heißt o_MarkText.

Funktionsname:

mustIshow mustIshow

Signatur:

mustIshow:: integer -> errType -> boolean.

integer : Bit-Maske, die die Auswahl des Benutzer repräsentiert.

errType : Typ der Nachricht, die überprüft werden soll.

boolean : Boolsches Ergebnis: TRUE, falls die Nachricht dargestellt werden soll, ansonsten FALSE.

Beschreibung:

Testet, ob der Benutzer eine Nachricht eines bestimmten Typs angezeigt haben will.

Vor- und Nachbedingungen:

Die Bit-Maske muß einer festgelegten Struktur entsprechen. Dabei entspricht

- Bit 1 fatalen Fehlern,
- Bit 2 Einschränkungs-Warnungen,
- Bit 3 Reparatur-Warnungen,

- Bit 4 allgemeinen Warnhinweisen,
- Bit 5 sonstigen Bemerkungen.

Funktionsname:

pos2string pos2string

Signatur:

pos2string:: env -> insertPos -> string.

env: ASpecT-Environment des ForauS -Systems.

insertPos : Positionsangabe, an denen das selektierte Objekte eingefügt/gelöscht werden kann.

string : Resultierende Zeichenkette.

Beschreibung:

Dient zur Generierung einer (Teil-) Meldung für den Benutzer, aus der hervorgeht, an welcher genauen (logischen) Position im Dokument ein Element eingefügt beziehungsweise gelöscht werden kann. Dies ist besonders bei nicht eindeutigen Cursor-Psoitionierungen im WYSIWYG-Modus wichtig, da der genaue Platz für das Einfügen/Löschen von entscheidender Bedeutung für die logische Struktur eines Dokumentes ist.

Implementierung:

Die Positionsangaben in Form des Parameters insertPos geschieht nicht in dieser (lokalen) Funktion, sondern schon zwei Funktionshierarchien höher, in der Funktion show_possible-Elements.

Funktionsname:

printmsg printmsg

Signatur:

printmsg:: modulName -> errType -> errNumber -> errTexts -> env -> env.

modulName: Name des Moduls, in dem der Fehler auftrat.

errType: Fehlertyp.

errNumber: Numerische Kennung des Fehlers.

errTexts: Die Beschreibung des Fehlers im Klartext.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Die Funktion gibt in das von der Benutzungsoberfläche geöffnete Nachrichtenfenster eine Fehlermeldung aus. Die Meldung wird aus den oben genannten Komponenten zusammengesetzt.

Funktionsname:

repaint_marked_Block repaint_marked_Block

Signatur:

repaint_marked_Block:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Zeichnet die alte Markierung eines Textbereichs neu auf dem Bildschirm. Dies geschieht, wenn ein Seitenwechsel zurück zu einer Seite mit einer Blockmarkierung stattgefunden hat, die Blockmarkierung über mehrere Seiten geht, oder ein Window-Refresh das Neuzeichnen einer Seite erforderlich macht.

Vor- und Nachbedingungen:

Blockstart-Koordinaten müßen vorher gesetzt werden.

Implementierung:

Die Funktion findet keine Verwendung (mehr), da die Blockmarkierung auf Textebene aufgrund von Schwierigkeiten im Formatierer- / Ausgabe-Modul nicht mehr vollständig entwickelt wurde. Im wesentlichen liegt dies an der Unterscheidung der Zeilenbreite und der Spaltenbreite bei variablen Größen, sowie der problematischen Erkennung von Elementgrößen / Blockgrößen über mehrere Seiten, als auch das Überschneiden von verscheidenen Element-Blcken in der gleichen Zeile. Sie ist deshalb nicht funktionstüchtig!

Funktionsname:

set_ViewMenu2 set_ViewMenu2

Signatur:

set_ViewMenu2:: env->strings->env.

env: ASpecT-Environment des ForauS -Systems.

strings : Liste mit Namen von Sichten, die in das Sichtenmenü eingetragen werden sollen.

Beschreibung:

Dient zum Eintragen der aktuell gültigen Sichtennamen in das Menü der Sichten der Oberfläche .

Vor- und Nachbedingungen:

Es muß zuvor sichergestellt werden, daß ein Menü für Sichten vom X-System erzeugt worden ist und eventuell noch vorhandene Sichten-Menüeinträge entfernt werden, da immer nur hinzugefügt wird.

Implementierung:

Diese lokale Funktion arbeit sehr eng mit den Funktionen afterLoad und zahlreichen C-Funktionen zusammen. Vor einer Änderung bitte genau die Implementierung studieren, da hier schnell Fehler entstehen, die schlecht aufgefunden werden.

Funktionsname:

strConcat strConcat

Signatur:

strConcat:: strings -> string.

strings: Liste von Zeichenketten.

string: Konkatenierte Zeichenkette.

Beschreibung:

Konkateniert eine Liste von Zeichenketten zu einer einzigen Zeichenkette, indem stets rechts der Ergebnis-Zeichenkette angefügt wird.

Implementierung:

Die Funktion wurde hauptsächlich zur Optimierung des Übersetzungsvorganges von ASpecT eingesetzt.

Funktionsname:

strConcat1 strConcat1

Signatur:

strConcat1:: strings -> string -> string.

strings: Liste von Zeichenketten.

string: Zwischenergebnis der Funktion.

strings: Liste von Zeichenketten.

Beschreibung:

Hilfsfunktion zum Konkatenieren einer Liste von Zeichenketten zu einer einzigen Zeichenkette. Es wird stets rechts der Ergebnis-Zeichenkette angefügt.

Implementierung:

Es handelt sich bei strConcat1 um eine Hilfsfunktion zu strConcat (s.dort).

Funktionsname:

string2int string2int

Signatur:

string2int:: string -> integer.

string: Die in einen Integer umzuwandelnde Zeichenkette.

integer: Die aus dem String gewonnene Ganzzahl.

Beschreibung:

Umwandlung einer Zeichenkette in einen Integerwert. string2int ist als Ersatz für die fehlerhafte long-Funktion der Standardbibliothek gedacht. Siehe auch string2int2.

Funktionsname:

string2int2 string2int2

Signatur:

string2int2:: string -> integer -> integer -> integer.

string: Zeichenkette, die in ein Integer umgewandelt werden soll. Sie muß vorher mittels der Funktion reverse umgekehrt werden.

integer: Zwischensumme (immer mit null (0) initiieren).

integer: Potenz der ersten Stelle im dezimalen Zahlensystem (immer mit eins (1) initiieren).

integer: Ergebniszahl.

Beschreibung:

Die Funktion liefert den dezimalen Wert, der der (verdrehten) Zeichenkette entspricht. Falls die Zeichenkette Zeichen enthält, die keine Ziffern sind, wird der Wert 0 (null) zurückgeliefert.

Implementierung:

Es handelt sich um eine Hilfsfunktion von string2int. Sie ist als Ersatz für die fehlerhafte Funktion der Standardbibliothek gedacht.

Funktionsname:

test_error2 test_error2

Signatur:

test_error2:: env -> integer -> (env, integer).

env: ASpecT-Environment des ForauS -Systems.

integer: Nummer des bisherigen Fehlertyps.

integer: Nummer des bisherigen Fehlertyps, wenn der aktuelle Fehler nicht einen Fehler mit einer noch höheren Wertigkeit darstellt.

Beschreibung:

Prüft den aktuellen Fehler, falls vorhanden, daraufhin ab, ob er dem Benutzer angezeigt werden muß und gibt nötigenfalls eine Meldung auf dem Nachrichtenfenster aus.

Implementierung:

Über die interne Struktur besorgt sich die Funktion den aktuellen Fehler und konvertiert ihn per convTypeToNum in einen Integerwert (siehe dort). Er wird mit dem bisherigen Fehler verglichen und löst diesen ab, wenn er größer ist. Gegebenenfalls wird der Fehler dann angezeigt. Zum Schluß wird der Fehler mit i_removeError entfernt.

Funktionsname:

ui_get_vars ui_get_vars

Signatur:

ui_get_vars:: env -> (integer, integer, integer, integer, integer, env).

env: ASpecT-Environment des ForauS -Systems.

integer: Cursorposition X.

integer: Cursorposition Y.

integer: Seitennummer.

integer: Objektindex.

integer: ObjektID.

Beschreibung:

Diese Funktion dient zur Unterstützung der Cursorsteuerung. Sie holt mithilfe der Funktion get_global_var aktuelle Koordinaten- und Objektdaten aus dem Dictionary.

Vor- und Nachbedingungen:

Die in der Beschreibung genannten Variablen müssen vor dem Aufruf dieser Funktion mit ui_put_vars oder einzeln mit put_global_var initialisiert werden.

Implementierung:

Die Funktion besteht im Grunde nur aus einer Aneinanderreihung von get_global_var Befehlen. Diese werden mit den Zeichenketten foraus_CursorPositionX, foraus_CursorPositionY, foraus_CurrentPageNumber, foraus_CurrentObjIndex und foraus_CurrentObjID gefüttert und ihr Ergebnis zurückgeliefert.

Funktionsname:

ui_put_vars ui_put_vars

Signatur:

ui_put_vars:: integer -> integer -> integer -> integer -> integer -> env -> env.

env: ASpecT-Environment des ForauS -Systems.

integer: Cursorposition X.

integer: Cursorposition Y.

integer: Seitennummer.

integer: Objektindex.

integer: ObjektID.

Beschreibung:

Hier liegt das Gegenstück zu ui_get_vars vor. Die oben angeführten Integerwerte werden in das Dictionary der Benutzungsoberfläche geschrieben.

Funktionsname:

ui_showObjInfo ui_showObjInfo

Signatur:

ui_showObjInfo:: env -> objID -> env.

env: ASpecT-Environment des ForauS -Systems.

objID: Kennung des aktuellen Objekts.

Beschreibung:

Schreibt den Namen des aktuellen logischen Elements in den Fußbereich des Textfensters. Wenn daVinci aktiviert wurde, wird außerdem noch die graphische Anzeige des Dokuments aktualisiert und das Element, auf dem sich der Cursor befindet, hervorgehoben.

Funktionsname:

ui_showPage ui_showPage

Signatur:

ui_showPage:: env -> pagecoordinates -> pagecoordinates -> env.

env: ASpecT-Environment des ForauS -Systems.

pagecoordinates: Alte Seitenkoordinaten.

pagecoordinates: Neue Seitenkoordinaten.

Beschreibung:

Bei Seitenwechsel wird die neue Seite initialisiert (d.h. ggf. neu formatiert) und angezeigt.

Funktionsname:

ui_showPage2 ui_showPage2

Signatur:

ui_showPage2:: env -> pagecoordinates -> pagecoordinates -> env.

env: ASpecT-Environment des ForauS -Systems.

pagecoordinates: Alte Seitenkoordinaten.

pagecoordinates: Neue Seitenkoordinaten.

Beschreibung:

Bei Seitenwechsel wird die neue Seite initialisiert (d.h. ggf. neu formatiert) und angezeigt. Diese Funktion ist speziell für Seitenwechsel die durch Cursorbewegungen erzeugt werden.

Funktionsname:

ui_showPage3 ui_showPage3

Signatur:

ui_showPage3:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Bei Seitenwechsel wird die neue Seite initialisiert (d.h. ggf. neu formatiert) und angezeigt. Diese Funktion ist speziell für Seitenwechsel bei denen sich die Cursorposition nicht ändert.

Funktionsname:

watch_timer watch_timer

Signatur:

watch_timer:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Schaut auf der Uhr nach, ob das aktuelle Dokument gespeichert werden muß. Die vollautomatische, periodische Speicherung kann im Dialogfenster der Voreinstellungen deaktiviert werden.

Vor- und Nachbedingungen:

Diese Funktion liefert nur bei eingebauter Uhr korrekte Ergebnisse.

Implementierung:

Diese Funktion ist ein lokaler Ersatz für die inkorrekten ASpecT Timerfunktionen. Sie sollte in möglichst gleichmäßigen Abständen aufgerufen werden. Bisher wird sie bei jedem Einfügen von Zeichen und Elementen aufgerufen.

3.2.5 Externe Funktionen

Funktionsname:

change_ViewMenuC change_ViewMenuC

Signatur:

change_ViewMenuC:: env -> string -> boolean

env: ASpecT-Environment des ForauS -Systems.

string: Zeichenkette mit einem Menüeintrag für das Sichtenmenü.

boolean: Das Ergebnis ist immer logisch wahr.

Beschreibung:

Die Funktion fügt dem Menü für die Sichten den angegebenen Menüpunkt hinzu.

Funktionsname:

create_new_MenuItemC create_new_MenuItemC

Signatur:

create_new_MenuItemC:: env -> string -> env.

env: ASpecT-Environment des ForauS -Systems.

string: Zeichenkette mit einem Menüeintrag für das Sichten-Menü.

Beschreibung:

Die Funktion fügt der Liste der Elemente, die an der Cursorposition eingefügt werden können, einen Eintrag hinzu.

Funktionsname:

init_user_interface init_user_interface

Signatur:

init_user_interface:: boolean -> boolean -> string -> env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Initialisiert die Benutzungsoberfläche. Zur Zeit wird hier allerdings nur das ASpecT-Environment für ForauS in die globale Variable FORAUSenv kopiert, damit es allen C-Funktionen zur Verfügung steht. Genau das gleiche wird aber auch in u_Main gemacht, so daß diese Funktion z.Z. ohne Verwendung ist.

Funktionsname:

printAsMsgC printAsMsgC

Signatur:

printAsMsgC:: env -> string -> env.

env: ASpecT-Environment des ForauS -Systems.

string: Zeichenkette mit Nachrichtentext.

Beschreibung:

Die Funktion schreibt eine Nachricht in das Nachrichtenfenster der Benutzungsoberfläche von ForauS .

Funktionsname:

print2rightFooterC print2rightFooterC

Signatur:

print2rightFooterC:: env -> integer -> integer.

env: ASpecT-Environment des ForauS -Systems.

integer: Aktuell angezeigte Seitenummer.

integer: Gesamtseitenzahl des Dokuments.

Beschreibung:

Im rechten Fußbereich wird ein Text in der Form „Seite X von Y“ angezeigt.

Funktionsname:

printToFileC printToFileC

Signatur:

printToFileC:: env -> string.

env: ASpecT-Environment des ForauS -Systems.

string: Die auszudruckende Zeichenkette.

Beschreibung:

Diese Funktion druckt die mit string übergebene Zeichenkette in die Datei, die vom Anwender im Properties Menü eingetragen wurde. Siehe dazu auch im Modul UI.xc nach.

Funktionsname:

printTofooterC printTofooterC

Signatur:

printTofooterC:: env -> string -> env.

env: ASpecT-Environment des ForauS -Systems.

string: Zeichenkette mit einer einzeiligen Meldung.

Beschreibung:

Der an die Funktion übertragene Text wird linksbündig in den Fußbereich des ForauS Hauptfensters geschrieben.

Funktionsname:

printToPrinterC printToPrinterC

Signatur:

printToPrinterC:: env -> string.

env: ASpecT-Environment des ForauS -Systems.

string: Die auszudruckende Zeichenkette.

Beschreibung:

Diese Funktion druckt die übergebene Zeichenkette auf dem Drucker aus. Wie der Drucker angesprochen wird legt der Anwender im ForauS Properties Menü fest. Siehe dazu aber auch im Modul UI.xc nach.

Funktionsname:

u_ActTimereq u_ActTimereq

Signatur:

u_ActTimereq:: env -> prozent -> (env,status).

env: ASpecT-Environment des ForauS -Systems.

prozent: Fortschritt in Prozent. (Numerisch [1..100])

status: Status für Benutzerinteraktion.

Beschreibung:

Es wird die Anzeige des Timerequesters aktualisiert. Für das Auffrischen muß in Prozent v.H. angegeben werden, wieweit die Aufgabe bereits abgearbeitet wurde. Ein vollständig abgearbeiteter Vorgang entspricht 100 Prozent. Als Rückgabewert wird das ASpecT-Environment sowie ein Statuswert geliefert, anhand dessen man feststellen kann, ob der Vorgang abgebrochen werden soll.

Vor- und Nachbedingungen:

Bevor diese Funktion verwendet werden kann muß mit u_OpenTimereq das Anzeigefenster initialisiert werden. Nach der Beendigung des Vorganges, muß das Anzeigefenster mit u_CloseTimereq wieder geschlossen werden.

Implementierung:

In UI.AS wird die Funktion nur als externer Verweis geführt. Die eigentliche Funktionalität ist im Modul UI.xc verankert worden.

Funktionsname:

u_CloseTimereq u_CloseTimereq

Signatur:

u_CloseTimereq:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Nach Verwendung des Timerequesters muß dessen Anzeigefenster wieder geschlossen werden. Dies geschieht mit u_CloseTimereq.

Vor- und Nachbedingungen:

Bevor diese Funktion verwendet werden kann muß mit u_OpenTimereq das Anzeigefenster initialisiert worden sein.

Implementierung:

In UI.AS wird die Funktion nur als externer Verweis geführt. Die eigentliche Funktionalität ist im Modul UI.xc verankert worden.

Funktionsname:

u_Main u_Main

Signatur:

u_Main:: env -> env.

env: ASpecT-Environment des ForauS -Systems.

Beschreibung:

Mit u_Main startet die Benutzungsoberfläche und gibt die Kontrolle von ASpecT-generierten Code an den C-Code bzw. das X System ab. Die Benutzungsoberfläche ist die Schnittstelle zwischen ForauS und dem Benutzer.

Vor- und Nachbedingungen:

Vor dem Aufruf muß die Benutzungsoberfläche mit init_user_interface initialisiert werden.

Implementierung:

In UI.AS wird die Funktion nur als externer Verweis geführt. Die eigentliche Funktionalität ist im Modul UI.xc verankert worden.

Funktionsname:

u_OpenTimereq u_OpenTimereq

Signatur:

u_OpenTimereq:: env -> statustext -> env.

env: ASpecT-Environment des ForauS -Systems.

statustext: Erläuternder Text zum Vorgang. (String)

Beschreibung:

Es wird auf dem Bildschirm ein Fenster geöffnet, in welchem der Fortschritt eines Vorganges angezeigt wird. Dargestellt wird das durch einen thermometerähnlichen horizontalen Balken, der mit null Prozent beginnt und mit hundert Prozent endet. An die Funktion wird ein Text übergeben, der Auskunft darüber erteilt, um was für einen Vorgang es sich handelt.

Vor- und Nachbedingungen:

u_OpenTimereq sorgt nicht selbständig für die Aktualisierung der Anzeige. Dazu muß bei einer Änderung des Zustandes u_ActTimereq aufgerufen werden.

Implementierung:

In UI.AS wird die Funktion nur als externer Verweis geführt. Die eigentliche Funktionalität ist im Modul UI.xc verankert worden.

Funktionsname:

ui_getTimeC ui_getTimeC

Signatur:

ui_getTimeC:: integer.

integer: Minuten.

Beschreibung:

Die Funktion liefert die Zahl der Minuten, die seit Mitternacht vergangen sind.

3.3 Das C-Modul UI

Jäschke

3.3.1 Funktionalität

Mit dem C-Modul der Benutzungsoberfläche wird die Verbindung zwischen dem in ASpecT programmierten Teil von ForauS und der Benutzungsoberfläche XView hergestellt. Dazu wird eine graphische Oberfläche mit den dafür üblichen Fenstern und Menüs generiert. Aktionen des Benutzers werden ausgewertet und dann weitergeleitet.

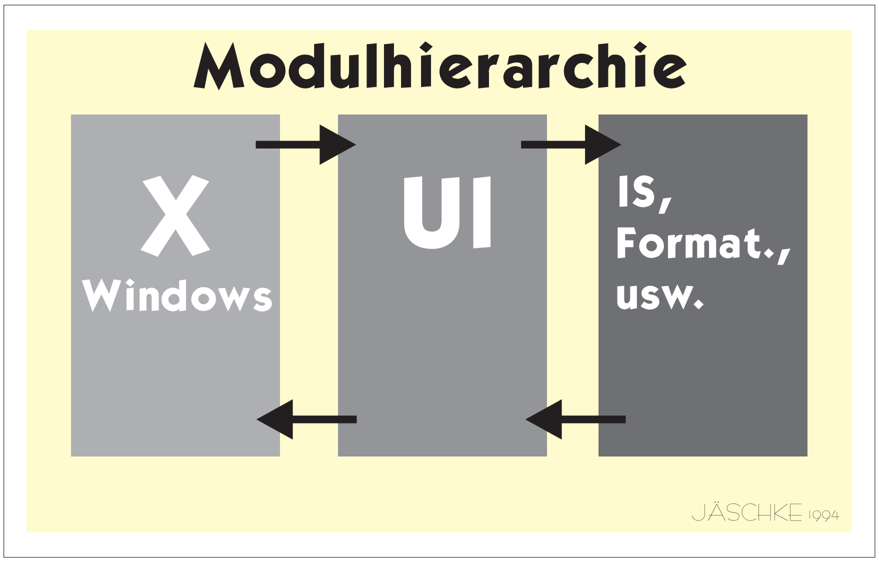

Abbildung 3.1: Zusammenspiel der Module.

3.3.2 Entwurf

Zunächst einmal stellt sich natürlich die Frage, warum ForauS nicht ohne in C programmierte Funktionen auskommt. Die Antwort darauf ist, wie so oft, relativ einfach. Es gibt für ASpecT, der „Muttersprache" von ForauS , keine praktisch verwertbaren Hilfsmittel, die eine Verbindung zur Fensteroberfläche herstellen. So wurde mit C unsere Schnittstelle zwischen XView (der Mensch-Maschine-Schnittstelle) und dem in ASpecT geschriebenen ForauS -Programm, der eigentlichen „Intelligenz", hergestellt.
Sämtliche C-Funktionen wurden in einem einzigen Modul untergebracht — UI.xc. Dazu kommen noch einige kleinere ForauS -spezifische Includedateien, als da wären UI.xh, UIEnglish.xh und ../pixel/UI.icon. Letztere enthält das kleine Bildchen, welches immer dann angezeigt wird, wenn das ForauS -Hauptfenster ikonisiert wird. Für die Includedatei UIEnglish.xh, welche die auf dem Bildschirm dargestellten Texte in englischer Fassung enthält, gibt es auch eine deutsche Version. Sie heißt UIGerman.xh. Leider wurde die Trennung von Programm und Texten nicht immer konsequent durchgehalten. Es gibt noch eine Hand voll Funktionen, welche keine Konstanten für die Bildschirmtexte enthalten.
Die Schnittstelle zum Windowssystem X wurde in erster Linie so programmiert, daß sie mit dem Betriebssystem Solaris Version 2.3 und höher von SUN zusammenarbeitet. Wird das System auf einem solchen System kompiliert, sollte der Schalter SOLARIS definiert sein. Dadurch wird es

möglich, die Dokumente per Dateiauswahlbox (auch File_chooser genannt) bequem mit der Maus auszuwählen. Andernfalls kann lediglich eine „fest eingebrannte“ Datei (durch Druck auf den Quickloader-Knopf) geladen werden. Eine sinnvolle Erweiterung an dieser Stelle wäre es also, wenn es neben der Dateiauswahlbox auch einen einfachen Requester gäbe, in dem der Dateiname des zu ladenden Dokuments direkt eingetippt werden kann. Trotz der genannten Einschränkungen kann das System auch auf einem Linux-Rechner gefahren werden. In diesem Fall ist der Schalter LINUX zu definieren.

3.3.3 Funktionen

Funktionsname:

allpages_selected allpages_selected

Signatur:

int allpages_selected(Panel_item item, int value, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

int value: Eins aktiviert die Eingabefelder für Seitenzahlen, jeder andere Wert deaktiviert sie.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Rückgabewert: Die Funktion liefert als festes Ergebnis die X-Konstante `XV_OK`.

Beschreibung:

Die Funktion ist ein Notifier der in der Funktion ui_drucken Verwendung findet. Genauer heißt das, daß er dort in eine X-Struktur eingebunden wurde und jetzt immer dann gerufen wird, wenn der Benutzer den Knopf für „Alle Seiten drucken" betätigt. Damit schaltet er immer zwischen zwei Zuständen hin und her: wenn ein Häkchen erscheint, gibt das Programm später alle Seiten des Dokuments aus, andernfalls kann er einen von ihm frei gewählten Seitenbereich ausdrucken.

Implementierung:

Ist „Alle Seiten drucken" ausgewählt, werden über die xv_set() Betriebssystemfunktion die Eingabefelder für die Seitenzahlen deaktiviert und umgekehrt.

Funktionsname:

answer answer

Signatur:

int ui_ask_for_quit()

Rückgabewert: Ein Integerwert im Bereich von eins bis drei mit folgender Bedeutung:

answer	Bedeutung
1	alle Änderungen im Dokument seit dem letzten Speichern verwerfen
2	alle Änderungen im Dokument vor dem Verlassen des Programms speichern
3	das Programm nicht zu beenden und mit dem Edieren fortzufahren

Beschreibung:

Diese Funktion wird aufgerufen, wenn im ForauS -Fenster Quit angewählt wird. Dabei wird die Funktion quit() (s. dort) aufgerufen, die widerum ui_ask_for_quit() ruft. Ein Fenster öffnet sich, und es kommt ein Hinweis darauf, daß das Programm jetzt beendet wird. Der Anwender bekommt nun drei Möglichkeiten zur Auswahl: alle Änderungen seit dem letzten Speichern zu verwerfen, alle Änderungen vor dem Verlassen des Programms zuvor zu speichern oder das Programm nicht zu beenden und mit dem edieren fortzufahren.

Funktionsname:

apply_einstellungen apply_einstellungen

Signatur:

void apply_einstellungen(Panel_item item, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Beschreibung:

Immer wenn im Fenster der Voreinstellungen der Knopf zur Bestätigung der Eingabe angeklickt wird ruft das Betriebssystem diese Funktion, die dann die Einstellungen ins Environment überträgt und somit wirksam macht. Der Menüpunkt zum Speichern der Einstellungen wird freigegeben.

Funktionsname:

apply_print apply_print

Signatur:

void apply_print(Panel_item item, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Beschreibung:

Die Funktion ist ein Notifier der in der Funktion ui_drucken Verwendung findet. Sie wird immer dann gerufen, wenn die Eingabe bestätigt wird. Es werden die aktuellen Einstellungen für den Dokumentendruck aus den X-Objekten gelesen und in das ForauS Dictionary übertragen. Zum Schluß wird dann der Ausdruck gestartet.

Implementierung:

Für das Drucken wird die Funktion ASpecT-Funktion event_Print aufgerufen, die dann die zu druckenden Seiten aufbereitet und ausgibt.

Funktionsname:

autosaveselect autosaveselect

Signatur:

int autosaveselect(Panel_item item, int value, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

int value: Gibt an, ob das automatische periodische abspeichern des Dokuments ein oder ausgeschaltet ist. Der Wert eins deaktiviert die Funktion, jeder andere Wert schaltet sie ein.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Rückgabewert: Die Funktion immer den Integerwert XV_OK zurück.

Beschreibung:

Dieser Notifier schaltet das automatische Abspeichern, das wiederkehrend nach einer bestimmten Zeit durchgeführt wird, ein oder aus.

Implementierung:

Diese Funktion schaltet das automatische Speichern nicht direkt um. Es wird eigentlich nur das Objekt auf dem Bildschirm beeinflußt. Wirksam wird die Änderung erst, wenn der Benutzer seine Eingaben im Properties Fenster bestätigt (s.a. apply_einstellungen()).

Funktionsname:

canvas_event_proc canvas_event_proc

Signatur:

void canvas_event_proc(Xv_Window window, Event* event, Notify_arg arg)

Xv_Window window: Wird vom X-Betriebssystem übergeben.

Event *event: Wird vom X-Betriebssystem übergeben.

Notify_arg arg: Wird vom X-Betriebssystem übergeben.

Beschreibung:

Bei jeder Aktion die auf der Canvas-Oberfläche, also dem Bereich des Fensters in dem das Dokument ediert wird, durchgeführt wird, ruft XView diese Routine auf. Ein Verteiler springt dann die Unterroutinen an, die nötig sind um die Aufgaben zu erledigen. Als Ereignisse werden das Drücken einer Taste und Aktionen mit der Maus erkannt.

Implementierung:

Viele der Aufgerufenen Routinen wurden in ASpecT programmiert und finden sich deshalb im Modul UI.AS wieder. Der Grund dafür ist, daß die grundsätzliche Programmiersprache für ForauS ASpecT ist. Daraus ergibt sich fast automatisch, daß sich die meisten Probleme am elegantesten in genau dieser Sprache realisieren lassen.

Funktionsname:

check_scanf_error check_scanf_error

Signatur:

void check_scanf_error(int err_number)

int err_number: Nummer des Fehlers, der beim Einlesen mit der scanf() Funktion aufgetreten ist.

Beschreibung:

Wenn die Fehlernummer gleich null oder gleich der Konstante EOF ist, wird die Fehlermeldung „Error while reading data from property file.“ auf der Konsole ausgegeben.

Funktionsname:

clear_ViewMenu clear_ViewMenu

Signatur:

void clear_ViewMenu()

Beschreibung:

Entfernt aus dem Sichtenmenü alle Menüpunkte, mit denen man unterschiedliche Sichten anwählen kann. Dies ist immer dann nötig, wenn ein neues Dokument geladen wird, da sich dann die möglichen Sichten ändern können.

Funktionsname:

connect_daVinci connect_daVinci

Signatur:

void connect_daVinci()

Beschreibung:

Diese Funktion verbindet ForauS mit daVinci. Nach ihrem Aufruf kann über die stdin- und stdout-Pipe eine Kommunikation zwischen den beiden Programmen stattfinden. Wenn die Verbindung nicht aufgebaut werden kann, wird eine Fehlermeldung auf die Konsole ausgegeben.

Implementierung:

Um daVinci mit ForauS zu verbinden wird mit der C-Funktion fork() ein sogenannter Child-Prozess gestartet. Die stdin-, stdout- und stderr Kanäle des Elternprozess werden nun so auf zuvor generierte Pipes umgelenkt, daß darüber Befehle zwischen den beiden Programmen/Prozessen ausgetauscht werden können. Eine genaue Beschreibung zur Verbindung von Programmen mit daVinci findet sich im Manual zu daVinci.

Die ForauS -C-Funktion zum Senden von Nachrichten an daVinci heißt send_message_to_daVinci(). Für Nachrichten, die von daVinci an ForauS gesendet werden (das passiert z.B. immer dann, wenn dort ein Element des Graphen markiert wird), wird zudem ein Handler installiert. Sein Name ist stdin_notify_for_daVinci().

Funktionsname:

destroy_func_for_daVinci destroy_func_for_daVinci

Signatur:

Notify_value destroy_func_for_daVinci(Notify_client client, Destroy_status status)

Notify_client client: Client-Struktur des X-Systems.

Destroy_status status: Status-Informationen von X.

Rückgabewert: Status des Notifiers nach Beendigung dieser Funktion.

Beschreibung:

Diese Funktion wird vom XView Runtime System immer dann gerufen, wenn der Elternprozess, also ForauS , „stirbt“. Sie entfernt dann den Kindprozess daVinci. Wie der Prozess zu entfernen ist wird über den von X übergebenen status-Wert entschieden. Weiterführende Informationen finden sich in der Beschreibung zu Unix und daVinci.

Funktionsname:

discard_einstellungen discard_einstellungen

Signatur:

void discard_einstellungen(Panel_item item, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Beschreibung:

Sollen die im Fenster für die Einstellungen gemachten Änderungen nicht wirksam werden, kann der Benutzer einen Knopf zum Abbrechen betätigen. Damit löst er diese Routine aus, die dem System mitteilt, daß die Einstellungen nicht geändert wurden und eine entsprechende Meldung im Fußbereich des Hauptfenster plaziert.

Implementierung:

Die Mitteilung an das System sieht so aus, daß im Environment das flag foraus_properties_changed auf null gesetzt wird.

Funktionsname:

discard_print discard_print

Signatur:

void discard_print(Panel_item item, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Beschreibung:

Dieser Notifier wird von ui_drucken() gerufen, wenn der Knopf zum Abbrechen angeklickt wurde. Es wird „vergessen", daß der Benutzer Änderungen vorgenommen hat und eine kurze, entsprechende Meldung im Fußbereich des Hauptfensters ausgegeben.

Implementierung:

Das „Vergessen" sieht so aus, daß die bei Änderungen gesetzte Environmentvariable foraus_properties_changed auf null zurückgesetzt wird.

Funktionsname:

insert_Element_Cancel_Notify_Proc insert_Element_Cancel_Notify_Proc

Signatur:

void insert_Element_Cancel_Notify_Proc(Panel_item item, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Beschreibung:

Es handelt sich hierbei um einen Notifier, der immer dann angesprungen wird, wenn im Fenster für das Einfügen eines Elements der Knopf zum Abbrechen gedrückt wird. Das Auswahlfenster wird, ungeachtet vom Status des Push-Pins, geschlossen und das Hauptfenster wieder aktiviert. Im Fußbereich des Hauptfenster wird der Abbruch der Funktion kurz kommentiert.

Implementierung:

Das Schliessen des Einfügefensters geschieht durch ein simples xv_set(), mit dem der Parameter XV_SHOW auf FALSE gesetzt wird.

Funktionsname:

printAsMsgC2 printAsMsgC2

Signatur:

void printAsMsgC2(char* String)

char* String: Zeichenkette mit Nachrichtentext. Zeilenumbrüche werden mit \ n gekennzeichnet.

Beschreibung:

Schreiben einer Nachricht in das Nachrichtenfenster von ForauS.

Funktionsname:

Quickloader Quickloader

Signatur:

void Quickloader()

Beschreibung:

Der Quickloader ist speziell zum Testen unter Linux integriert worden, da unter diesem Betriebssystem die Dateiabfragebox (File_chooser package unter Solaris) nicht läuft. Es wird das SGML-Dokument `/home/foraus/final/SGMLtest/memo.sgm` geladen.

Funktionsname:

quit quit

Signatur:

Notify_value quit(Notify_client my_client, Destroy_status my_status)

Notify_client my_client: Wird vom X-Betriebssystem übergeben.

Destroy_status my_status: Wird vom X-Betriebssystem übergeben.

Rückgabewert: Ergebnis der Behandlung des Notifiers, also der quit-Routine.

Beschreibung:

Wird im ForauS -Fenster Quit gewählt, springt XView automatisch diese Routine an. Es wird abgefragt, ob der Benutzer das Programm wirklich verlassen möchte, ggf. vorher das Dokument abgespeichert oder wieder zur Bearbeitung des Textes zurückgekehrt.

Implementierung:

Die Quit-Routine wird in xx_UIu_Main_0() im Basisrahmen, dem foraus_base_frame, eingehängt. Die Funktion zur Abfrage des Benutzers ist ui_ask_for_quit()

Funktionsname:

send_message_to_daVinci send_message_to_daVinci

Signatur:

void send_message_to_daVinci(char* cMsgString)

char* cMsgString: Kommando, welches an daVinci gesendet werden soll.

Beschreibung:

Mit dieser Funktion werden Befehle an daVinci übermittelt. Das Kommando zum Ändern der Fensterüberschrift des daVinci-Fensters würde z.B. so aussehen: `set_window_title( "ForauS : Document structure")`. Eine vollständige Auflistung aller Kommandos und ihrer Syntax findet sich im Manual zu daVinci.

Implementierung:

Es wird zunächst über einen Eintrag im Dictionary geprüft, ob das daVinci System bereits über connect_daVinci() (s.a. dort) an ForauS angesschlossen wurde. Ist dies der Fall, wird über ein simples printf() der Kommandostring auf die stdout-Pipe geschrieben. Der daVinci-Prozess fängt die Nachricht auf und reagiert dementsprechend.

Funktionsname:

sigchldcatcher_for_daVinci sigchldcatcher_for_daVinci

Signatur:

Notify_value sigchldcatcher_for_daVinci(Notify_client client, int pid, int* status, struct rusage* rusage)

Notify_client client: Client-Struktur von XView.

int pid: Prozess-ID

int* status: status des Prozess.

struct rusage *rusage: rusage-Struktur von XView.

Rückgabewert: `NOTIFY_DONE`, wenn das eingehende Signal behandelt werden konnte, sonst `NOTIFY_IGNORED`.

Beschreibung:

Diese Funktion wird vom XView Runtime-System aufgerufen. Sie behandelt das Ende des daVinci-Prozesses. Wenn dieser beendet wird, generiert dieser ein Signal. Es wird von dieser Funktion abgefangen, die dann die Leitungen zwischen den Prozessen kappt. Näheres darüber in der Dokumentation zu daVinci.

Funktionsname:

stdin_notify_for_daVinci stdin_notify_for_daVinci

Signatur:

Notify_value stdin_notify_for_daVinci(Notify_client client, int fd)

Notify_client client: Client-Struktur von X.

int fd: fd-Struktur von X.

Rückgabewert: Ist immer `NOTIFY_DONE`.

Beschreibung:

Diese Funktion wird immer dann vom Betriebssystem aktiviert, wenn daVinci über die aufgebauten Kommunikationsleitungen eine Nachricht an ForauS sendet. Die eingehende Nachricht wird ausgelesen und ausgewertet.

Implementierung:

Wenn die Nachricht gelesen werden konnte und es sich nicht um eine „System warning“ handelte, dann wird der in ASpecT geschriebene Handler stdin_handler aufgerufen. Dieser wertet die eingehenden Daten weiter aus.

Funktionsname:

tofile_toggle tofile_toggle

Signatur:

int tofile_toggle(Panel_item item, int value, Event* event)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

int value: Gibt an, wohin der Druck erfolgen soll. Ist value null, erfolgt die Ausgabe auf den angegebenen Drucker, andernfalls in eine Datei.

Event* event: Zeiger auf eine Ereignisstruktur von X.

Rückgabewert: Die Funktion liefert als festes Ergebnis die X-Konstante `XV_OK`.

Beschreibung:

Die Funktion ist ein Notifier der in der Funktion ui_drucken Verwendung findet. Genauer heißt das, daß er dort in eine X-Struktur eingebunden wurde und jetzt immer dann gerufen wird, wenn der Benutzer zwischen der Ausgabe in eine Datei oder auf den Drucker umschaltet. Die Funktion selbst kopiert jetzt je nach Auswahl die nötigen aktuellen Parameter, Druckername und Druckeroptionen oder Dateipfad und Dateiname, in die zugehörigen Felder.

Implementierung:

Die konkrete Implementierung sieht so aus, daß es vier Dictionary-Einträge gibt, in denen die Informationspaare aufgehoben werden: foraus_print_directory und foraus_print_filename bewahren die aktuellen Parameter für die Ausgabe in eine Datei auf, wenn der Drucker aktiv ist und foraus_print_printername und foraus_print_printeroptions heben die Druckparameter im umgekehrten Fall auf.

Funktionsname:

UIu_ActTimereq UIu_ActTimereq

Signatur:

void UIu_ActTimereq(TERM Env, TERM Wert, TERM* EnvBack, TERM* Status))

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM Wert: Enthält den neuen Prozentwert als numerische Zahl. Diese darf 100 Prozent nicht überschreiten!

TERM *EnvBack: In diese Variable wird nach der Ausführung das modifiziert ForauS -Environment geschrieben.

TERM *Status: Wenn der Abbruchknopf gedrückt wurde (s.a. UIu_OpenTimereq()) ändert sich hier der Statuswert auf eins. Ansonsten enthält status den Wert Null.

Beschreibung:

Aktualisiert die Anzeige des Prozentbalkens entsprechend der übergebenen Prozentzahl. Es wird geprüft, ob der Abbruchknopf gedrückt wurde und das Ergebnis der Prüfung in der Variablen status abgelegt.

Vor- und Nachbedingungen:

Vor Verwendung dieser Funktion muß das Anzeigefenster mit UIu_OpenTimereq() geöffnet werden. Nach Beendigung des Vorgangs kann die Anzeige mit UIu_CloseTimereq() wieder entfernt werden.

Funktionsname:

UIu_CloseTimereq UIu_CloseTimereq

Signatur:

TERM UIu_CloseTimereq(TERM Env)

TERM Env: ASpecT-Environment des ForauS -Systems.

Rückgabewert: Das modifizierte ASpecT-Environment des ForauS -Systems.

Beschreibung:

Schließt ein mit UIu_OpenTimereq() geöffnetes Fenster wieder.

Vor- und Nachbedingungen:

Das Fenster muß mit UIu_OpenTimereq() geöffnet worden sein.

Funktionsname:

UIu_OpenTimereq UIu_OpenTimereq

Signatur:

TERM UIu_OpenTimereq(TERM Env, TERM Text)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM Text: Beschreibender Text für den Vorgang.

Rückgabewert: Das modifizierte ASpecT-Environment des ForauS -Systems.

Beschreibung:

Die Funktion öffnet ein Fenster, in dem ein Prozentbalken dargestellt wird. Die Anzeige wird mit null Prozent initialisiert. Der angegebene Text wird hinzugeschrieben. Außerdem enthält das Fenster einen Knopf, mit dem vom Benutzer signalisiert werden kann, daß die Operation abgebrochen werden soll.

Vor- und Nachbedingungen:

Der Prozentbalken wird mit UIu_ActTimereq() aktualisiert. Wenn die Prozentanzeige nicht mehr benötigt wird, kann das Fenster mit UIu_CloseTimereq() wieder geschlossen werden.

Funktionsname:

UI_Xinitialize UI_Xinitialize

Signatur:

unsigned UI_Xinitialize(unsigned MODE)

unsigned MODE: Unbekannter Inhalt — wird von ASpecT belegt.

Rückgabewert: Unbekannt (ASpecT-intern).

Beschreibung:

Die genaue Aufgabe dieser Funktion ist unbekannt. Nur soviel ist sicher: jede C-Erweiterung zu einem ASpecT Programm muß sie in dieser Form besitzen. Der Name der Funktion ist immer der Name des Moduls plus „_Xinitialize()“. Die verwendete Variable sollte „__XINIT_“ plus ebenfalls den Modulnamen heißen.

Funktionsname:

ui_ausschneiden ui_ausschneiden

Signatur:

void ui_ausschneiden(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Es wird markierter Text oder ein logisches Element ausgeschnitten und auf dem Clipboard abgeheftet.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_keyCut gerufen. In der jetzigen Phase der Implementierung wird der ausgeschnittene Text noch nicht auf dem Clipboard abgelegt.

Funktionsname:

ui_auswaehlen ui_auswaehlen

Signatur:

void ui_auswaehlen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Auswählen des logischen Elements unter dem Cursor und das Markieren als Block. Ein bisher markierter Block wird deselektiert.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_Menu_Select_Element gerufen.

Funktionsname:

ui_create_callback ui_create_callback

Signatur:

void ui_create_callback(File_chooser foraus_file_chooser_create, char* cPath_ptr, char* cFile_ptr, Xv_opaque client_data)

File_chooser foraus_file_chooser_create: Die File_chooser Struktur.

char *cPath_ptr: Enthält den im File_chooser gewählten Pfad inklusive dem Dateinamen, z.B. `/foraus/doc/demo.sgm`.

char *cFile_ptr: Enthält nur den Dateinamen, z.B. `demo.sgm`.

Xv_opaque client_data: XView Daten.

Beschreibung:

Nach Eingabe eines Dateinamen im File_chooser werden von dieser Funktion zunächst im Menü Document die Menüpunkte zum Speichern und Schliessen aktiviert. Danach wird der Parser initialisiert, die DTD und die zugehörigen Präsentationsregeln geladen und abschließend das Menü der Sichten auf den neuesten Stand gebracht.

Implementierung:

Diese Routine wird im File_chooser zum Erstellen neuer Dateien eingehängt (ui_neu_erstellen()). Sie wird immer dann angesprungen, wenn der Benutzer einen Dateinamen, und vielleicht auch Pfad, eingibt und die Eingabe bestätigt. Wird der File_chooser abgebrochen, wird diese Routine nicht angesprungen!

Vor- und Nachbedingungen:

Dieser Funktion wird nur aktiv (kompiliert), wenn SOLARIS als Konstante definiert wurde.

Funktionsname:

ui_DocumentStructureCallBack ui_DocumentStructureCallBack

Signatur:

void ui_DocumentStructureCallBack(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Mit dieser Funktion wird eine Verbindung zwischen ForauS und daVinci geschaffen. Sobald das System installiert ist wird die Struktur des Dokuments in daVinci graphisch dargestellt.

Implementierung:

Da das Anbinden von daVinci sehr zeitkritisch ist und auf alle Fälle nur einmal aufgerufen werden darf, wird zuerst der Menüpunkt für die Anzeige von Dokumentengraphen ausgeblendet. connect_daVinci() stellt dann die gewünschte Verbindung her. Danach erfolgt die Aktualisierung des Graphen automatisch.

Funktionsname:

ui_drucken ui_drucken

Signatur:

void ui_drucken(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

ui_drucken öffnet ein Dialogfenster auf dem Bildschirm. Es werden darin die folgenden Druckparameter abgefragt:

- Ausgabeeinheit (Drucker, Datei, ...)
- Druckername
- Druckeroptionen
- Drucken von Seite bis Seite, alternativ alle Seiten
- Anzahl anzufertigender Kopien

Die Auswahl kann mit einem Apply-Button bestätigt oder jederzeit mit dem Abort-Button zurückgenommen werden.

Implementierung:

Die vom Benutzer gewählte Konfiguration wird im Dictionary abgelegt und steht somit für andere Funktionen bereit.

Funktionsname:

ui_einfuegen ui_einfuegen

Signatur:

void ui_einfuegen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Wenn sich ein Text auf dem Clipboard befindet wird dieser an der Cursorposition einkopiert.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_keyPaste gerufen. Da das Clipboard noch nicht unterstützt wird läuft diese Funktion zur Zeit noch nicht.

Funktionsname:

ui_einstellen_oberflaeche ui_einstellen_oberflaeche

Signatur:

void ui_einstellen_oberflaeche(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Es öffnet sich ein Dialogfenster in dem der Benutzer Einstellungen modifizieren kann, die das ganze ForauS System betreffen: Einfüge- oder Überschreibmodus, automatisches Speichern vor dem Laden eines Dokuments, Backup beim Abspeichern erzeugen, periodisches, automatisches Speichern in frei wählbaren Zeitabständen, daVinci Pfad usw.

Implementierung:

Wenn das Fenster nach Programmstart zum ersten mal aufgerufen wird, wird ein neuer Rahmen erzeugt (foraus_properties_frame). Auf dem zugehörigen Panel werden dann die einzelnen Objekte plaziert, mit denen der Benutzer die Einstellungen verändern kann. Wird das Fenster aber zum wiederholten male aufgeklappt, werden nur die aktuellen Einstellungen aus dem Environment in die Objekte der noch vorhandenen Struktur übertragen. Zum Schluß wird das Fenster mit dem Umschalten von XV_SHOW auf TRUE sichtbar gemacht und aktiviert.

Funktionsname:

ui_element_einfuegen ui_element_einfuegen

Signatur:

void ui_element_einfuegen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Es wird an der Cursorposition ein logisches Element eingefügt. Dazu wird erst eine Liste aller an dieser Stelle möglichen Elemente ermittelt und diese dem Benutzer in einem Fenster zur Auswahl bereitgestellt. Elemente können vor, hinter oder in dem aktuellen logischen Element plaziert werden.

Implementierung:

Als erstes wird das Fenster erzeugt, vorausgesetzt es existiert noch nicht. Dann wird eine leere Liste erzeugt. Diese wird dann per show_possibleElements (Modul UI.AS) mit allen möglichen Elementen und den Stellen, an denen sie eingefügt werden dürfen, gefüllt. Gibt es an dieser Stelle keine Möglichkeit, ein Element einzufügen, wird am Fuße des Hauptfensters eine Meldung ausgegeben. Das eigentliche Einfügen nimmt dann der Listennotifier ui_which_element() vor.

Funktionsname:

ui_kopieren ui_kopieren

Signatur:

void ui_kopieren(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Es wird markierter Text oder ein logisches Element kopiert und auf dem Clipboard abgeheftet.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_keyCopy gerufen. In der jetzigen Phase der Implementierung wird der Text noch nicht auf dem Clipboard abgelegt.

Funktionsname:

ui_load_document ui_load_document

Signatur:

int ui_load_document(char* cPath, char* cName)

char* cPath: Der Pfad inklusive dem Dateinamen, z.B. `/foraus/doc/demo.sgm`.

char* cName: Nur der Dateiname, z.B. `demo.sgm`.

Rückgabewert: Null im Fehlerfall, sonst eins.

Beschreibung:

Die Funktion stößt das Lesen eines Dokuments vom permanenten Datenspeicher an. Es werden in der Reihenfolge die DTD, die Präsentationsregeln und zum Schluß das SGML-Dokument geladen. Traten dabei keine Fehler auf, wird der Formatierer initialisiert und die erste Seite formatiert. Diese wird sofort angezeigt und, sofern daVinci eingehängt und vom Benutzer aktiviert wurde, graphisch visualisiert.

Funktionsname:

ui_loeschen ui_loeschen

Signatur:

void ui_loeschen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Löscht einen markierten Block oder ein logisches Element. Der entfernte Text wird in den Undo-Puffer geschrieben.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_keyCut gerufen. Ein Undo-Puffer wurde bisher nicht eingerichtet, so daß ein einmal gelöschter Text bisher nicht wieder hergestellt werden kann.

Funktionsname:

ui_neues_fenster_oeffnen ui_neues_fenster_oeffnen

Signatur:

void ui_neues_fenster_oeffnen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Öffnet ein neues Fenster zur Darstellung einer anderen, möglichen Sicht.

Implementierung:

Das Fenster wird zwar schon geöffnet, doch viel mehr passiert noch nicht. Dies liegt an der bisherigen Gesamtkonzeptzion des Systems (Oberfläche, Ausgabe, Formatierung, Interne Struktur...).

Funktionsname:

ui_neu_erstellen ui_neu_erstellen

Signatur:

void ui_neu_erstellen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Diese Funktion ist mit dem Menüpunkt für das Erstellen eines neuen Dokuments verknüpft. Es wird ein entsprechender File_chooser erzeugt und aktiviert.

Vor- und Nachbedingungen:

Dieser Funktion wird nur aktiv (kompiliert), wenn SOLARIS als Konstante definiert wurde.

Funktionsname:

ui_oeffnen ui_oeffnen

Signatur:

void ui_oeffnen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Diese Funktion ist mit dem Menüpunkt für das Laden eines bereits existierenden Dokuments verknüpft. Es wird ein entsprechender File_chooser erzeugt und aktiviert.

Vor- und Nachbedingungen:

Dieser Funktion wird nur aktiv (kompiliert), wenn SOLARIS als Konstante definiert wurde.

Funktionsname:

ui_oeffnen_callback ui_oeffnen_callback

Signatur:

void ui_oeffnen_callback(File_chooser foraus_file_chooser_open, char* cPath_ptr, char* cFile_ptr, Xv_opaque client_data))

File_chooser foraus_file_chooser_open: Die File_chooser Struktur.

char *cPath_ptr: Enthält den im File_chooser gewählten Pfad inklusive dem Dateinamen, z.B. `/foraus/doc/demo.sgm`.

char *cFile_ptr: Enthält nur den Dateinamen, z.B. `demo.sgm`.

Xv_opaque client_data: XView Daten.

Beschreibung:

Wird im Dokumentenmenu „Open...“ angewählt, erscheint unter SOLARIS eine Dateiauswahlbox, aus der eine Datei zum Einlesen ausgewählt werden kann. Wird die Eingabe bestätigt, ruft eben diese Auswahlbox ui_oeffnen_callback(), um das Dokument tatsächlich zu laden. Zuvor jedoch wird entsprechend den Einstellungen der Properties geprüft, ob ein modifiziertes Dokument erst gespeichert werden muß und dies ggf. auch getan. Nachdem das Dokument geladen wurde, wird das Sichten-Menü den neuen Präsentationsregeln angepaßt und der Titel des Textfensters aktualisiert.

Implementierung:

Diese Routine wird im File_chooser zum Laden bereits existierender Dateien eingehängt (ui_load_document()). Sie wird immer dann angesprungen, wenn der Benutzer einen Dateinamen, und vielleicht auch Pfad, eingibt und die Eingabe bestätigt. Wird der File_chooser abgebrochen, wird diese Routine nicht angesprungen!

Vor- und Nachbedingungen:

Dieser Funktion wird nur aktiv (kompiliert), wenn SOLARIS als Konstante definiert wurde.

Funktionsname:

ui_rueckgaengig ui_rueckgaengig

Signatur:

void ui_rueckgaengig(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Die zuletzt durchgeführte Aktion wird zurückgenommen.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_keyUndo gerufen, welche allerdings zum jetzigen Zeitpunkt noch ohne Funktion ist.

Funktionsname:

ui_save_document ui_save_document

Signatur:

int ui_save_document(char* cPath, char* cName)

char* cPath: Der Pfad inklusive dem Dateinamen, z.B. `/foraus/doc/demo.sgm`.

char* cName: Nur der Dateiname, z.B. `demo.sgm`.

Rückgabewert: Immer numerisch eins.

Beschreibung:

Die Funktion schreibt ein Dokument auf den permanenten Datenspeicher. Es wird erst das SGML-Dokument geladen und dann die DTD gespeichert. Siehe dazu auch in der Dokumentation des Generators nach.

Funktionsname:

ui_schliessen ui_schliessen

Signatur:

void ui_schliessen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Es wird ein im Speicher stehendes Dokument geschlossen. Die Menüs zum Speicher und Bearbeiten sowie die Sichten werden deaktiviert.

Funktionsname:

ui_set_zoom ui_set_zoom

Signatur:

void ui_set_zoom(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Schaltet auf die gewünschte Vergrößerung bzw. Verkleinerung um.

Implementierung:

Aus dem entsprechenden X-Objekt (s.a Hauptfunktion) wird der Vergößerungs- bzw. Verkleinerungsfaktor ausgelesen und das Ausgabemodul übergeben. Die aufgerufene Funktion ist o_SetZoom.

Funktionsname:

ui_sichten_notifier ui_sichten_notifier

Signatur:

void ui_sichten_notifier(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Dieser Notifier realisiert das Umschalten auf eine andere, in den Präsentationsregeln beschriebene, Sicht. Die Seite wird dementsprechend neu aufgebaut.

Implementierung:

An Pfad und Dateinamen des geladenen Dokuments wird, durch einen Punkt getrennt, der Name der gewünschten Sicht angehängt. Dieser entspricht dem Namen im Fenster des Sichten Menüs. Mit diesem neuen String wird die Parserfunktion p_loadPR aufgerufen. Danach baut der Formatierer die Seite neu auf und ein Aufruf von o_InitPage aus der Ausgabe zeigt die Seite neu auf dem Bildschirm an.

Funktionsname:

ui_speichern ui_speichern

Signatur:

void ui_speichern(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Diese Funktion ist mit dem Menüpunkt für das Speichern eines im Speicher stehenden Dokuments verknüpft. Es wird ui_save_document() aufgerufen. Wenn das Dokument seit dem letzten Speichern bzw. Laden nicht modifiziert wurde, wird dies als Hinweis gemeldet und das Dokument nicht gesichert. Soll trotzdem gesichert werden, muß dies unter einem anderen Dateinamen geschehen.

Funktionsname:

ui_speichern_unter ui_speichern_unter

Signatur:

void ui_speichern_unter(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Diese Funktion ist mit dem Menüpunkt für das Speicher eines Dokuments unter einem neuen Namen verknüpft. Es wird ein entsprechender File_chooser erzeugt und aktiviert.

Vor- und Nachbedingungen:

Dieser Funktion wird nur aktiv (kompiliert), wenn SOLARIS als Konstante definiert wurde.

Funktionsname:

ui_speichern_unter_callback ui_speichern_unter_callback

Signatur:

void ui_speichern_unter_callback(File_chooser foraus_file_chooser_save_as,
char* cPathAndFile_ptr, struct stat* sFile_status)

File_chooser foraus_file_chooser_save_as: Die File_chooser Struktur

char *cPathAndFile_ptr: Enthält den im File_chooser gewählten Pfad plus den Dateinamen, z.B. `/foraus/doc/demo.sgm`.

struct stat* sFile_stats: X-Struktur.

Beschreibung:

Wird im Dokumentenmenu „Save As...“ angewählt, erscheint, unter SOLARIS, eine Dateiauswahlbox, in der ein Dateiname angegeben werden kann, unter dessen Namen das aktuelle Dokument gespeichert werden soll. Wird die Eingabe bestätigt, ruft die Auswahlbox ui_speicher_unter_callback() auf, um das Dokument tatsächlich zu laden. Zuvor jedoch wird entsprechend den Einstellungen der Properties geprüft, ob ein modifiziertes Dokument erst als Backup gespeichert werden muß und dies ggf. auch getan. Nachdem das Dokument gespeichert wurde, wird der Titel des Textfensters aktualisiert.

Implementierung:

Diese Routine wird im File_chooser zum Speicher von Dokumenten unter einem neuen Namen eingehängt (ui_speichern_unter()). Sie wird immer dann angesprungen, wenn der Benutzer einen Dateinamen, und vielleicht auch einen Pfad, eingibt und die Eingabe bestätigt. Wird der File_chooser abgebrochen, wird diese Routine nicht angesprungen!

Die Backup-Funktion ist noch nicht implementiert worden.

Vor- und Nachbedingungen:

Dieser Funktion wird nur aktiv (kompiliert), wenn SOLARIS als Konstante definiert wurde.

Funktionsname:

ui_timereq_abortnotify ui_timereq_abortnotify

Signatur:

int ui_timereq_abortnotify(Panel_item item, int value, Event* event))

Panel_item item: Die Struktur des Knopfs, mit dem im Timereq-Fenster ein Vorgang abgebrochen werden kann.

int value: Wertigkeit des Schalters.

Event* event: Eine Ereignisstruktur.

Rückgabewert: Die Konstante XV_ERROR oder XV_OK. Letztere, wenn der Abbruchknopf gedrückt wurde.

Beschreibung:

Diese Funktion gehört zum Funktionsblock des Prozentbalkens. Dieser wird in einem separaten Fenster angezeigt und mit einem Knopf zum Abbruch der gerade laufenden Aktion ausgestattet. Wird nun dieser Knopf gedrückt, verzweigt das Programm automatisch in die hier beschrieben Funktion. Dies wird in UIu_OpenTimereq() so festgelegt. Die Funktion selbst schreibt nun in ein Merkfeld (foraus_timereq_status im Dictionary) den Statuswert eins, der dann von der Funktion UIu_ActTimereq() zurückgegeben wird.

Funktionsname:

ui_toggleMsgWindow ui_toggleMsgWindow

Signatur:

void ui_toggleMsgWindow(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Die Funktion blendet das Nachrichtenfenster von ForauS ein oder aus.

Funktionsname:

ui_ueber_foraus ui_ueber_foraus

Signatur:

void ui_ueber_foraus(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Die Funktion öffnet ein Fenster, in dem Information über das Projekt ForauS angezeigt werden. Der Text dazu kann maximal 19 Zeilen umfassen und kommt aus dem Includefile UIyyyyyyy.xh. yyyyyyy muß durch die entsprechende Sprachkennung, z.B. English, ersetzt werden. Die defines mit dem Text haben die Namen ui_txt_about_string_xx, wobei xx die Zahlen [01..19] darstellen soll.

Funktionsname:

ui_umgebung_laden ui_umgebung_laden

Signatur:

void ui_umgebung_laden(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Installiert und aktiviert einen Dateiauswahlkasten zum Laden der Umgebungsvariablen, sprich Voreinstellungen.

Implementierung:

Die File_chooser Struktur wird nur dann neu erzeugt, wenn die Funktion nach Programmstart das erste mal aufgerufen wird.

Funktionsname:

ui_umgebung_laden_callback ui_umgebung_laden_callback

Signatur:

void ui_umgebung_laden_callback(File_chooser foraus_file_chooser_properties_open, char* cPath_ptr, char* cFile_ptr, Xv_opaque client_data)

File_chooser foraus_file_chooser_properties_open: Die File_chooser Struktur

char *cPath_ptr: Enthält den im File_chooser gewählten Pfad inlusive Dateinamen für die Einstellungsdatei.

char *cFile_ptr: Enthält nur den Dateinamen. Namen der Einstellungsdateien enden immer mit .env, z.B. `privat.env`.

Xv_opaque client_data: XView Daten.

Beschreibung:

Es handelt sich hierbei um die eigentliche Laderoutine für die Voreinstellungen. Die Daten werden aus der gewählten Datei gelesen und in die entsprechenden Variablen im ForauS Environment geschrieben. Danach wird der File_chooser geschlossen. Wenn die Einstellungen nicht gelesen werden können erscheint im Fußbereich des ForauS Hauptfensters eine dementsprechende Meldung.

Implementierung:

Wannimmer im Dateiauswahlkasten zum Laden der Einstellungsdatei die Auswahl bestätigt wird, wird diese Funktion angesprungen (s.a. ui_umgebung_laden()). Sie öffnet die gewählte Datei binär mit fopen(), liest mit fscanf() die Einstellungen aus der Datei und trägt sie im Environment/Dictionary ein.

Funktionsname:

ui_umgebung_speichern ui_umgebung_speichern

Signatur:

void ui_umgebung_speichern(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Installiert und aktiviert einen Dateiauswahlkasten zum Speichern der Umgebungsvariablen, sprich Voreinstellungen.

Implementierung:

Die File_chooser Struktur wird nur dann neu erzeugt, wenn die Funktion nach Programmstart das erste mal aufgerufen wird.

Funktionsname:

ui_umgebung_speichern_callback ui_umgebung_speichern_callback

Signatur:

void ui_umgebung_speichern_callback(File_chooser foraus_file_chooser_properties_save_as, char* cPath_ptr, struct stat* sFile_stats)

File_chooser foraus_file_chooser_properties_open: Die File_chooser Struktur

char *cPath_ptr: Enthält den im File_chooser gewählten Pfad inlusive Dateinamen für die Einstellungsdatei.

struct stat* sFile_stats: XView Daten.

Beschreibung:

Die Routine speichert die Umgebungsvariablen binär in der Datei ab, die durch cPath_ptr spezifiziert wurde. Kann die Daten nicht gesichert werden wird im Fußbereich des Hauptfensters eine Warnung ausgegeben. Die Auswahlbox wird am Ende noch geschlossen.

Implementierung:

Diese Prozedur wird immer dann angesprochen, wenn der Benutzer in dem von ui_umgebung_speichern() initialisierten Dateiauswahlkasten die Wahl bestätigt. Mit fopen() wird dann die Datei binär zum Schreiben geöffnet und die Daten dann nacheinander mit fprintf() gesichert.

Funktionsname:

ui_what_to_do_docload ui_what_to_do_docload

Signatur:

int ui_what_to_do_docload(Panel foraus_panel, int nModified)

Panel foraus_panel: Panel-Struktur des ForauS Hauptfensters bzw. des foraus_base_frame. An dieses Panel wird ggf. (je nach Situation) eine Abfragebox geheftet. Wenn es mehrere Sichten gibt, von denen jede ihr eigenes Panel hat, dann muß hier immer das aktuelle Panel eingetragen werden.

int nModified: Ein Flag das anzeigt, ob das aktuell geladene Dokument modifiziert wurde. Null bedeutet, daß das Dokument nicht verändert wurde, der Wert eins steht für eine Modifikation.

Rückgabewert: Ein Integerwert, der Auskunft darüber gibt, wie weiter zu verfahren ist. Die Bedeutung der Ergebnisse ist weiter unten beschrieben.

Beschreibung:

Die Funktion entscheidet, wie die Funktion ui_open_callback() vorzugehen hat. Zur Findung der Lösung wird der Wert foraus_properties_load_document_value herangezogen, in dem der Anwender festgelegt hat, wie sich das System beim Laden zu verhalten hat. Dieser kann in den Properties vom Benutzer beeinflußt werden. Außerdem wird beachtet, ob das Dokument überhaupt verändert wurde. Wenn nicht, wird die Funktion auf keinen Fall einen Rückgabewert von 1001 liefern (s.u.). Folgende drei Werte kann die Funktion als Ergebnis liefern:

answer	Bedeutung
1001	altes Dokument erst speichern, dann laden
1002	altes Dokument nicht speichern, nur laden; wird auch als Ergebnis geliefert, wenn nModified null ist.
1003	Ladevorgang abbrechen

Funktionsname:

ui_what_to_do_docsave ui_what_to_do_docsave

Signatur:

int ui_what_to_do_docsave(Panel NoticePanel, int nModified)

Panel NoticePanel: Panel, an welches ggf. eine Abfragebox geheftet werden soll. Derzeitig ist dies immer das foraus_base_panel. Wenn es später einmal mehrere Sichten geben sollte, die jede ihr eigenes Panel haben, muß immer das Panel der aktuellen Sicht hier eingetragen werden.

int nModified: Ein Flag das anzeigt, ob das aktuell geladene Dokument modifiziert wurde. Null bedeutet, daß das Dokument nicht verändert wurde, der Wert eins steht für ein Modifikation.

Rückgabewert: Ein Integerwert der Auskunft darüber gibt, wie weiter zu verfahren ist. Die Bedeutung der Ergebnisse sind weiter unten beschrieben.

Beschreibung:

Entscheidet, wie die Funktion ui_save_callback() vorzugehen hat. Zur Findung der Lösung wird der Wert foraus_properties_save_document_value herangezogen, in dem der Anwender festgelegt hat, wie sich das System beim Speichern zu verhalten hat. Dieser kann in den Properties vom Benutzer beeinflußt werden. Als Ergebnis liefert die Funktion drei mögliche Werte:

answer	Bedeutung
1001	alte Datei erst in Backup-Datei kopieren, dann speichern
1002	alte Datei überschreiben
1003	Speichervorgang abbrechen

Funktionsname:

ui_which_element ui_which_element

Signatur:

void ui_which_element(Panel_item item, char* string, Xv_opaque client_data, Panel_list_op op, Event* event, int row)

Panel_item item: Zeiger auf das Objekt, das diesen Notifier aktiviert hat.

char* string: Zeichenkette mit dem Text des gewählten Listeneintrages, in diesem Fall dem Elementnamen im Klartext.

Xv_opaque client_data: Weitere X-Information.

Panel_list_op op: Beschreibt, durch welche Benutzeraktion dieser Notifier ausgelöst wurde

Event* event: Zeiger auf eine Ereignisstruktur von X.

int row: Die gewählte Zeilennummer innerhalb der Liste der Elemente.

Beschreibung:

Wird vom Benutzer ein Element zum Einfügen ausgewählt (s.a. ui_element_einfuegen()), wird diese Funktion aktiviert. Das Element wird eingesetzt, die Seite neu formatiert und angezeigt.

Implementierung:

Als erstes schließt die Funktion das Auswahlfenster. Das eigentliche Einfügen, Reformatieren und Anzeigen der Seite übernimmt dann u_insert_element aus dem ASpecT-Modul UI.AS.

Funktionsname:

ui_wiederholen ui_wiederholen

Signatur:

void ui_wiederholen(Menu menu, Menu_item menu_item)

Menu menu: Das gewählte Menü. Wird vom X-Betriebssystem übergeben.

Menu_item menu_item: Der gewählte Menüpunkt innerhalb des Menüs. Wird vom X-Betriebssystem übergeben.

Beschreibung:

Die zuletzt durchgeführte Aktion wird nochmal ausgeführt.

Implementierung:

Die Funktion ist lediglich ein X Notifier, der die Aktion an das UI ASpecT-Modul weitergibt. Es wird die Funktion event_keyUndo gerufen.

Funktionsname:

xxUIui_getTimeC_0 xxUIui_getTimeC_0

Signatur:

TERM xxUIui_getTimeC_0()

Rückgabewert: Zahl der Minuten, die seit Mitternacht (Null Uhr) vergangen sind.

Beschreibung:

Die Funktion liefert die Zahl der Minuten die seit Mitternacht vergangen sind.

Funktionsname:

xxUIui_getTimeStringC_0 xxUIui_getTimeStringC_0

Signatur:

TERM xxUIui_getTimeStringC_0()

Rückgabewert: Ein String mit Datum und aktueller Zeit.

Beschreibung:

Die Funktion liefert das aktuelle Datum und die Zeit in Form einer Zeichenkette.

Funktionsname:

xx_UIchange_ViewMenuC_0 xx_UIchange_ViewMenuC_0

Signatur:

TERM xx_UIchange_ViewMenuC_0(TERM Env, TERM menuName)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM menuName: Zeichenkette mit einem Menüeintrag für das Sichten-Menü.

Rückgabewert: Das Ergebnis ist immer logisch wahr.

Beschreibung:

Die Funktion fügt dem Menü für die Sichten den angegebenen Menüpunkt hinzu. Verwendung findet sie im ASpecT Modul UI.AS.

Funktionsname:

xx_UIcreate_new_MenuItemC_0 xx_UIcreate_new_MenuItemC_0

Signatur:

TERM xx_UIcreate_new_MenuItemC_0(TERM Env, TERM NameASP)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM NameASP: Zeichenkette mit einem Menüeintrag für das Sichten-Menü.

Rückgabewert: Das modifizierte ASpecT-Environment des ForauS -Systems.

Beschreibung:

Die Funktion fügt der Liste der Elemente, die an der Cursorposition eingefügt werden können, einen Eintrag hinzu.

Funktionsname:

xx_UIinit_user_interface xx_UIinit_user_interface

Signatur:

TERM xx_UIinit_user_interface(TERM Env)

TERM Env: ASpecT-Environment des ForauS -Systems.

Rückgabewert: Das modifizierte ASpecT-Environment des ForauS -Systems.

Beschreibung:

Initialisiert die Benutzungsoberfläche. Zur Zeit wird hier allerdings nur das ASpecT-Environment für ForauS in die globale Variable FORAUSenv kopiert, damit es allen C-Funktionen zur Verfügung steht. Genau das gleiche wird aber auch in u_Main gemacht, so daß diese Funktion z.Z. ohne Verwendung ist.

Funktionsname:

xx_UIprint2rightFooterC_0 xx_UIprint2rightFooterC_0

Signatur:

TERM xx_UIprint2rightFooterC_0(TERM Env, TERM Page, TERM LastPage)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM Page: Aktuell angezeigte Seitenummer.

TERM LastPage: Gesamtseitenzahl des Dokuments.

Das modifizierte ASpecT-Environment des ForauS -Systems.:

Beschreibung:

Im rechten Fußbereich wird ein Text in der Form „Seite X von Y“ angezeigt.

Funktionsname:

xx_UIprintAsMsgC_0 xx_UIprintAsMsgC_0

Signatur:

TERM xx_UIprintAsMsgC_0(TERM Env, TERM Text)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM Text: Zeichenkette mit Nachrichtentext. Zeilenumbrüche werden mit \ n gekennzeichnet.

Das modifizierte ASpecT-Environment des ForauS -Systems.:

Beschreibung:

Schreiben einer Nachricht in das Nachrichtenfenster von ForauS .

Implementierung:

Die Funktion bereitet die Ausgabe nur vor. Die eigentliche Arbeit erledigt printAsMsgC2().

Funktionsname:

xx_UIprintToFileC_0 xx_UIprintToFileC_0

Signatur:

TERM xx_UIprintToFileC_0(TERM Env, TERM PSstring)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM PSstring: Die auszudruckende Zeichenkette.

Das modifizierte ASpecT-Environment des ForauS -Systems.:

Beschreibung:

Diese Funktion druckt die mit PSstring übergebene Zeichenkette in die Datei, die im Environment eingetragen wurde (s.a. ui_drucken()).

Implementierung:

Es wird mit den Standard C-Funktionen fopen(), fprintf() und fclose() gearbeitet.

Funktionsname:

xx_UIprintTofooterC_0 xx_UIprintTofooterC_0

Signatur:

TERM xx_UIprintTofooterC_0(TERM Env, TERM Text)

TERM Env: ASpecT-Environment des ForauS -Systems.

TERM Text: Zeichekette mit einer einzeiligen Meldung.

Das modifizierte ASpecT-Environment des ForauS -Systems.:

Beschreibung:

Der an die Funktion übertragene Text wird in den Fußbereich des $F^{or}a^{u}S$ Hauptfensters geschrieben, und zwar ausgerichtet am linken Rand (am rechten Rand befindet sich die Anzeige für die gerade angezeigt Seite). Siehe auch xx_UIprint2rightFooterC_0().

Funktionsname:

xx_UIprintToPrinterC_0 xx_UIprintToPrinterC_0

Signatur:

TERM xx_UIprintToPrinterC_0(TERM Env, TERM PSstring)

TERM Env: ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

TERM PSstring: Die auszudruckende Zeichenkette.

Das modifizierte ASpecT-Environment des $F^{or}a^{u}S$ -Systems.:

Beschreibung:

Diese Funktion druckt die mit PSstring übergebene Zeichenkette auf dem Drucker aus, der im Environment eingetragen wurde (s.a. ui_drucken()).

Implementierung:

Für die Ausgabe wird zunächst eine Temporärdatei geschaffen, in die die Zeichenkette PS-string geschrieben wird. Dann wird aus dem Druckernamen und der Optionszeile ein Kommandostring zusammengebaut (z.B. `lpr /home/foraus/Output.ps 1>NULL 2>NULL`) und dieser mit dem system() Befehl ausgeführt. Nach Beendigung des Drucks wird die Temporärdatei wieder gelöscht.

Funktionsname:

xx_UIu_Main_0 xx_UIu_Main_0

Signatur:

TERM xx_UIu_Main_0(TERM Env)

TERM Env: ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

Rückgabewert: Das modifizierte ASpecT-Environment des $F^{or}a^{u}S$ -Systems.

Beschreibung:

u_Main baut die für die Benutzungsoberfläche benötigten Strukturen auf und übergibt anschließend die Kontrolle an das Betriebssystem. Tritt dann ein Ereignis ein, das von $F^{or}a^{u}S$ behandelt werden muß (z.B. Mausklick, gedrückte Taste, ...), kehrt die Kontrolle für die Zeit der Abarbeitung der Aufgabe zurück.

Implementierung:

Es wird als erstes das von ASpecT übergebene Environment in eine globale Variable kopiert. Damit steht es jederzeit den anderen C-Funktionen zur Verfügung. Als nächstes werden die meisten der Variablen, die im Dictionary ui (s.a. UIStruct.AS) aufgehoben werden, vorbelegt. Es werden die Module Ausgabe und Generator initialisiert. Danach werden die Struktur für das Hauptfenster und sein Panel aufgebaut und die einzelnen Menüs damit verknüpft. Außerdem wird das Nachrichtenfenster, in dem unter anderem die Fehlermeldungen erscheinen sollen, erstellt und die Standard Quit-Routine derart ergänzt, daß der Benutzer vor dem Verlassen des Programms gefragt wird, ob er das denn auch wirklich will. Wenn all dies erledigt ist, wird die xv_main_loop gerufen. Diese terminiert, wenn das ForauS -Programm vom Benutzer beendet wird. Nun werden noch einige allokierte Speicherbereich freigegeben und ganz zum Schluß die zuvor aufgebauten Strukturen wieder freigegeben.

3.4 Das Dictionary uiDictCPointer

Jäschke

Da für den reibungslosen Ablauf der Benutzungsoberfläche unter X ständig eine ganze Reihe von Zeigern parat gehalten werden muß, ist die nachfolgende Tabelle recht lang geworden. Es handelt sich dabei trotzdem nicht um eine ausführliche Beschreibung der einzelnen Variablen mit Verwendungsnachweis, sondern eher um Nachschlagewerk mit kurzer Erläuterung. Die Angabe des Typs ist besonders dann nützlich, wenn mit UI_GET_GLOBAL einer der Werte zurückgelesen werden soll. Dann nämlich muß der Typ für das sogenannte Casting mit angegeben werden. Ein solcher Aufruf könnte so aussehen: `UI_GET_GLOBAL( Frame, foraus_base_frame )`.
Neben den von X benötigten Zeigern werden auch Werte in dieser Tabelle geführt, die dem Austausch von Daten zwischen ASpecT und C dienen oder sogar ausschließlich nur auf ASpecT-Seite verwendet werden. Auf diese Variablen wird in der Dokumentation der entsprechenden Module genauer eingegangen.

Eintrag	**Typ**	**Inhalt**
foraus_base_frame	Frame	Der wichtigste Zeiger im ganzen User Interface. Er verweist auf den Basisrahmen. Das ist grob gesagt das Hauptfenster auf dem alles andere aufbaut.
foraus_base_panel	Panel	Dieses Paneel sitzt direkt auf dem foraus_base_frame. Daran angeheftet werden die Menüknöpfe des Hauptfenster (foraus_base_panel_button Variablen).
foraus_base_panel_button_view	Panel_item	Zeiger auf den Knopf des Sichtenmenüs. Dieses wird je nach geladenen Präsentationsregeln vom Programm versorgt. Nur der Menüpunkt für die graphische Anzeige des Dokumentengraphen ist fest integriert.
foraus_menu_view	Menu	Zeiger auf das Menü der Sichten.
foraus_menu_view_structure_document	Menu_item	Menüpunkt zur Anzeige des Dokumentengraphen. Löst das Laden von daVinci aus.

Eintrag	Typ	Inhalt
foraus_base_panel_button_document	Panel_item	Zeiger auf den Knopf des Dokumentenmenüs. Wird er gedrückt, öffnet sich das Menü, auf das foraus_menu_document zeigt.
foraus_menu_document	Menu	Zeiger auf das Dokumentenmenü.
foraus_menu_document_create	Menu_item	Zeiger auf den ersten Menüpunkt des Dokumentenmenüs: Neuerstellen eines Dokuments.
foraus_menu_document_open	Menu_item	Menüpunkt Dokument öffnen.
foraus_menu_document_close	Menu_item	Menüpunkt Dokument schliessen.
foraus_menu_document_save	Menu_item	Menüpunkt Dokument speichern.
foraus_menu_document_save_as	Menu_item	Menüpunkt Dokument speichern unter.
foraus_menu_document_open_window	Menu_item	Menüpunkt Neue Sicht auf Dokument öffnen.
foraus_menu_document_print	Menu_item	Menüpunkt Dokument drucken.
foraus_base_panel_button_edit	Panel_item	Zeiger auf den Knopf für das Ediermenü.
foraus_menu_edit	Menu	Menü des Edierknopfs.
foraus_menu_edit_canvas	Menu	Ediermenü. Dieses Menü ist nicht mit ein einem Knopf am Hauptfenster befestigt. Es wird immer dann aufgerufen, wenn im Textbereich (Canvas) die rechte Maustaste gedrückt wird.
foraus_base_panel_button_properties	Panel_item	Knopf für das Menü der Voreinstellungen.
foraus_menu_properties	Menu	Menü für die Voreinstellungen.
foraus_document_path	char*	Pfad und Dateiname des geladenen Dokuments.
foraus_document_name	char*	Dateiname des geladenen Dokuments.
foraus_document_modified	int	Gibt an, ob das Dokument bearbeitet wurde. Null wenn nicht, sonst eins.
foraus_file_chooser_create	File_chooser	Dateiauswahldialog für das Neuerstellen eines Dokuments.
foraus_file_chooser_open	File_chooser	Dateiauswahldialog für das Öffnen eines vorhandenen Dokuments.

Eintrag	**Typ**	**Inhalt**
foraus_file_chooser_save_as	File_chooser	Dateiauswahldialog für das Speichern des geladenen Dokuments unter einem neuen Namen.
foraus_file_chooser_properties_open	File_chooser	Dateiauswahldialog für das Öffnen einer Voreinstellungen Datei.
foraus_file_chooser_properties_save_as	File_chooser	Dateiauswahldialog für das Speichern einer Voreinstellungen Datei.
foraus_properties_frame	Frame	Rahmenumgebung für das Fenster des Voreinstellungen Dialogs
foraus_properties_panel	Panel	Diese Panel ist mit dem foraus_properties_frame verbunden. Es werden darauf die Schalter gesetzt, mit denen man die Voreinstellungen verändern kann.
foraus_properties_refeshwindow_object	Panel_item	Schalter für Voreinstellung: Immer alle Fenster auffrischen oder nur aktives.
foraus_properties_exitforausselect_object	Panel_item	Schalter für Voreinstellung: Was tun, wenn ein Fenster geschlossen wird.
foraus_properties_docsaveselect_object	Panel_item	Schalter für Voreinstellung: Vorgehen beim Speichern eines Dokuments.
foraus_properties_docloadselect_object	Panel_item	Schalter für Voreinstellung: Vorgehen beim Laden eines Dokuments.
foraus_properties_textmodeselect_object	Panel_item	Schalter für Voreinstellung: Einfüge- oder Überschreibmodus.
foraus_properties_cursor_positioning_object	Panel_item	Schalter für Voreinstellung: Cursorpositionierung.
foraus_properties_minutes_object1	Panel_item	Schalter für Voreinstellung: Ein- bzw. Ausschalten des automatischen Abspeicherns.
foraus_properties_minutes_object2	Panel_item	Schalter für Voreinstellung: Abstand in Minuten, nachdem automatisch gespeichert werden soll.
foraus_properties_errorlevel_object	Panel_item	Schalter für Voreinstellung: Errorlevel.
foraus_properties_daVinci_Path_object	Panel_item	Schalter für Voreinstellung: Pfad für das daVinci Programm.

Eintrag	Typ	Inhalt
foraus_properties_daVinci_Filename_object	Panel_item	Schalter für Voreinstellung: Name des daVinci Programms.
foraus_properties_cursor_positioning_value	int	Aktueller Wert des Schalters Cursorpositionierung.
foraus_properties_text_mode_value	int	Aktuelle Einstellung des Textmodus.
foraus_properties_load_document_value	int	Aktuelle Einstellung für das Laden eines Dokuments.
foraus_properties_save_document_value	int	Aktuelle Einstellung für das Speichern eines Dokuments.
foraus_properties_close_window_value	int	Aktueller Wert für das Vorgehen beim Schließen eines Fensters.
foraus_properties_refresh_window_value	int	Aktuelle Einstellung für das Auffrischen der Fenster.
foraus_properties_automatic_save_value	int	Abstand für das automatische Speichern in Minuten.
foraus_properties_automatic_save_is_on	int	Enthält die Information darüber, ob das automatische Speichern ein- oder ausgeschaltet ist.
foraus_errorlevel	int	Einstellungswert des Errorlevels
foraus_properties_changed	int	Gibt Auskunft darüber, ob die Voreinstellungen geändert wurden.
foraus_timereq_status	int	Statuswert der darüber Auskunft gibt, ob der Anwender im Fenster des Timerequesters die Taste zum Abbrechen des Vorganges gedrückt hat.
foraus_timereq_frame	Frame	Basisrahmen für den Timerequester. Alle weiteren Teile werden daran angefügt.
foraus_timereq_panel	Panel	Panel, an dem die Bestandteile des Timerequesters befestigt werden.
foraus_timereq_text	Panel_item	Text für den Timerequester.
foraus_timereq_gauge	Panel_item	Anzeigebalken des Timerequesters.
foraus_timereq_button	Panel_item	Knopf auf dem Fenster des Timerequester, mit dem die laufende Aktion beendet werden kann. Wird er betätigt, wird foraus_timereq_status auf einen Wert ungleich null gesetzt.

Eintrag	Typ	Inhalt
foraus_print_frame	Frame	Basisrahmen für das Fenster zum Ausdrucken eines Dokuments.
foraus_print_panel	Panel	Das zum Druckfenster zugehörige Panel
foraus_print_copies_object	Panel_item	Schalter für den Ausdruck: Eingabe der Anzahl der anzufertigenden Kopien.
foraus_print_startpage_object	Panel_item	Schalter für den Ausdruck: Eingabe der Seitenummer, mit der der Ausdruck begonnen werden soll.
foraus_print_endpage_object	Panel_item	Schalter für den Ausdruck: Eingabe der letzten zu druckenden Seite.
foraus_print_allpages_object	Panel_item	Schalter für den Ausdruck: Sorgt für den Ausdruck aller Seiten.
foraus_print_tofile_object	Panel_item	Schalter für den Ausdruck: Ausgabe auf den Drucker oder in eine Druckdatei.
foraus_print_text1_object	Panel_item	Schalter für den Ausdruck: Eingabe des Druckerbefehls/Verzeichnis.
foraus_print_text2_object	Panel_item	Schalter für den Ausdruck: Eingabe der Optionen/des Dateinamens
foraus_print_filename_object	Panel_item	Unbenutzt.
foraus_print_copies_value	int	Anzahl der Kopien.
foraus_print_startpage_value	int	Startseite
foraus_print_endpage_value	int	Endseite
foraus_print_allpages_is_on	int	Ist dieser Wert eins, werden die Schalter für die Seitenzahlen deaktiviert und beim Druck alle Seiten ausgegeben. Sonst ist der Wert null.
foraus_print_printername	char*	Enthält den Druckeraufruf.
foraus_print_printeroptions	char*	Enthält die Druckoptionen.
foraus_print_tofile_is_on	int	Ist dieser Wert null, wird auf den Drucker ausgegeben. Bei eins wird die Ausgabe in eine Datei umgeleitet.
foraus_print_filename	char*	Enthält das Verzeichnis für die Druckdatei.
foraus_print_directory	char*	Enthält den Namen der Druckdatei.
foraus_MaxPages	int	Unbenutzt.
foraus_subframe_created	int	Gibt an, ob ein Subfenster geöffnet wurde.
foraus_CursorPositionX	int	Aktuelle X Position des Cursors.
foraus_CursorPositionY	int	Aktuelle Y Position des Cursors.
foraus_CurrentPageNumber	int	Aktuelle Seitennummer.
foraus_OldCursorPositionX	int	Unbenutzt.
foraus_OldCursorPositionY	int	Unbenutzt.
foraus_OldPageNumber	int	Unbenutzt.
foraus_LastAction	int	Unbenutzt.
foraus_LastDocIsSavedTime	int	Unbenutzt.

Eintrag	**Typ**	**Inhalt**
foraus_CurrentObjIndex	int	Aktueller Objektindex.
foraus_CurrentObjID	int	Aktuelle ObjektID.
foraus_BlkStartX	int	Aktuelle Blockmarkierung, X Startposition.
foraus_BlkStartY	int	Aktuelle Blockmarkierung, Y Startposition.
foraus_BlkStartPage	int	Seite, auf der der Block beginnt.
foraus_BlkStartIdx	int	Objektindex des Objekts, mit dem der Block beginnt.
foraus_BlkStartObjID	int	ObjektID des Objekts, mit dem der Block beginnt.
foraus_BlkEndX	int	Aktuelle Blockmarkierung, X Endposition.
foraus_BlkEndY	int	Aktuelle Blockmarkierung, Y Endposition.
foraus_BlkEndPage	int	Seite, auf der die Blockmarkierung endet.
foraus_BlkEndIdx	int	Objektindex des Objekts, mit dem der Block endet.
foraus_BlkEndObjID	int	ObjektID des Objekts, mit dem der Block endet.
foraus_Block_is_marked	int	Wenn null, dann ist kein Block markiert. Bei eins ist ein Element markiert und bei zwei ein Textabschnitt.
foraus_message_frame	Frame	Basisrahmen für das Fenster, in dem die Nachrichten/Fehlermeldung von ForauS angezeigt werden.
foraus_error_textsw	Textsw	Textfeld, auf dem die Nachrichten angezeigt werden. Ist mit dem foraus_message_frame verbunden.
foraus_insert_element_frame	Frame	Basisrahmen für das Fenster, in dem die logischen Objekte angeboten werden, die an der aktuellen Cursorposition hinzugefügt werden können.
el_einf_panel	Panel	Panel für die Schalter des Fenster zum Einfügen von Elementen.
foraus_element_einfuegen_panel_list	Panel_item	Auswahlliste im Element-Einfügen-Fenster.
daVinci_installed	int	Diese Variable ist null, solange daVinci vom Benutzer nicht aufgerufen wurde.
daVinci_Path	char*	Pfad für die daVinci Applikation.
daVinci_Filename	char*	Name des daVinci Programms.
daVinci_process_id	int	Nummer des Prozesses, der beim Starten der daVinci Applikation entsteht.

3.5 Das Modul daVinci

Jäschke

3.5.1 Funktionalität

Das Modul daVinci wurde aus dem daVinci-Paket Version 1.3 entnommen. Es enthält die Strukturbeschreibung für Graphen, die mit diesem System dargestellt werden sollen. Für eine genauere Beschreibung wird hier nur auf die Dokumentation zum System daVinci verwiesen. Neu hinzugefügt wurde nur eine einzige Zeile:

```
outGraphtrees ::= new_term2 (graphtrees).
```

4. Ausgabe

4.1 Output

Bernd Rattey

4.1.1 Funktionalität

Das Modul `Output` ist die unterste Schicht der Gruppe `Ausgabe`. Es nimmt die eigentlichen Zeichen, Kopier- und Verschiebeoperationen unter X11 vor. Es wird daher in der Regel auch nur von den anderen Modulen der Gruppe `Ausgabe` aufgerufen.
Das Modul bietet folgende Grundfunktionalitäten:

- Initialisieren der Grafik der Ausgabe
- Zeichnen von Text
- Verschieben von Bereichen
- Invertieren von Bereichen
- Zeichnen des Cursors
- Ermitteln der Ausmaße der Grafik
- Ermitteln der Ausmaße von Text

Die Aufstellung dient nur einem ersten Eindruck, das Modul enthält noch weitere Funktionen.

4.1.2 Entwurf

Mehrere Probleme kennzeichneten den Entwurf zu diesem Modul:

1. Das Hauptproblem bei der Ausgabe ist, daß in einer funktionalen Sprache programmiert wird. In einer solchen sind globale Variablen nicht vorgesehen. Sie werden aber benötigt, da bei einem Refresh des Bildschirms keine Informationen darüber vorliegen, was neu gezeichnet werden soll.

2. Ein weiterer Aspekt ist, daß die eigentlichen Routinen, die die Grafik manipulieren, in der Programmiersprache C erfolgen müssen. Das Modul besteht daher aus zwei Dateien, einer AspecT-Datei und einer C-Datei. Die AspecT-Funktionen rufen in der Regel gleichnamige C-Funktionen auf, ein tieferer Sinn steckt in ihnen nicht. **Daher werden in dieser Dokumentation auch beide gleichzeitig dokumentiert.**

3. Zum Testen mußte zunächst eine Oberfläche simuliert werden. In der Testumgebung findet jedoch keine Interaktion statt, das Modell unterscheidet sich deswegen von dem Modell, das auf die Benutzungsoberfläche aufsetzt.

4. Da keines der Gruppenmitglieder vorher unter X11 programmiert hatte und keinerlei Dokumentation vorlag, mußten die Manual-Pages herhalten. Aus dem Grundwären einige Lösungen wahrscheinlich etwas eleganter realisierbar, sie funktionieren aber auch so.

4.1.3 Das Modell

4.1.3.1 Speicherung der statischen Variablen

Die Gruppe braucht einige statische Variablen, die gespeichert werden müssen. Das sind ausnahmslos Pointer bzw. Integer-Werte. Sie werden mit dem Mechanismus gespeichert, den auch die Gruppe `UI` verwendet und der von der Gruppe stammt. Eine Beschreibung erfolgt dort.

4.1.3.2 Art der statischen Variablen

Da alle Zeichenoperationen ständig wiederholbar sein müssen, um einen Refresh zu erlauben, kann nicht direkt in den `canvas` gezeichnet werden, da die Informationen sonst nicht erhalten blieben. Wir führen daher folgende Variablen ein:

- Es wird ein eigenständiger `canvas` für die Gruppe `Ausgabe` geschaffen, der initial leer bleibt.

- Es wird eine sog. `pixmap` geschaffen, die die komplette aktuelle Seite des Bildschirms der Ausgabegruppe speichert. Alle Zeichenoperationen, die später erfolgen, beziehen sich auf diese `pixmap`. Die Zeichenoperationen sind damit in einer Variablen gespeichert, allerdings noch nicht auf dem Bildschirm sichtbar.

- Es werden zwei Rollbalken geschaffen, die für das Scrollen verantwortlich sind.

- Es werden zwei sogenannte `gc` (graphics context) geschaffen. Sie sind notwendig, da X11 erst mit ihrer Hilfe Zeichenoperationen vornehmen kann. In ihnen sind beispielsweise die Farben von Vorder- und Hintergrund gespeichert.
  Warum man zwei Vertreter der Gattung und nicht einen benötigt, ist uns nicht klar geworden. Mit einem hat es jedoch nicht funktioniert.

- Es wird ein `display` geschaffen. Das wird benötigt, wenn die `pixmap` in den `canvas` kopiert und damit auf dem Bildschirm sichtbar wird.

4.1.3.3 Vorgehensweise

Mit den oben aufgeführten Variablen wird die Ausgabe recht simpel: Jede von den oberen Schichten der Gruppe `Ausgabe` kommende Anforderung einer Zeichenoperation wird in der `pixmap` gespeichert. Falls die aktuelle Seite dann auf dem Bildschirm dargestellt werden soll, wird die `pixmap` in den `canvas` kopiert.
Es werden zwei Funktionen zum Initialisieren implementiert: Eine für die spätere Benutzungsoberfläche und eine zum Testen. Weiterhin gibt es zwei Funktionen zum Löschen aller Variablen. In der Testversion wird die eigentliche X11-Testanwendung erst in der Funktion zum Löschen der angelegten Daten aufgebaut. Ansonsten arbeiten alle Funktionen sowohl als Test- als auch als Endversion.

4.1.4 Öffentliche Schnittstellen

4.1.4.1 Funktionen

Funktionsname:

initOutput initOutput

Signatur:

initOutput:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Die Funktion initialisiert die Ausgabe. Dazu wird die C-Funktion `initFontsC` aufgerufen. In der werden die Variablen erzeugt und gespeichert, die zur Darstellung der Ausgabe notwendig sind. Die gespeicherten Variablen befinden sich dann im zurückgegebenen Environment.

Vor- und Nachbedingungen:

Die X11-Anwendung muß von der `UI` erzeugt worden sein. Es muß einen `frame` namens `foraus_base_frame` geben.

Implementierung:

Zunächst werden die benötigten statischen Variablen gelesen. Dann wird in den `foraus-baseframe` der `canvas` der Ausgabegruppe eingehängt. Dieser wird mit zwei Rollbalken und drei Prozeduren versehen. Die ereignisgesteuerten Prozeduren sind für das Neuzeichnen, das Verschieben des `canvas` und für den Abbau des `canvas` notwendig.
Eine Erklärung hierzu findet sich im *XView Programming Manual*.
Dann werden die `gc` und die Zeichensätze initialisiert. Danach wird die Variable `display` dem `canvas` zugeordnet und die `pixmap` mit weißer Farbe ausgefüllt, also gelöscht.
Abschließend werden alle geschaffenen Variablen im Environment gespeichert.

Funktionsname:

drawText drawText

Signatur:

drawtext:: env->font->string->integer->integer->env

env [1]: Altes Environment

font: Zeichensatzinformation, die den Zeichensatz für den auszugebenden Font spezifiziert

string: Auszugebender Text

integer [1]: x-Koordinate

integer [2]: y-Koordinate

env [2]: Neues Environment

Beschreibung:

Gibt den Text `string` an der Position `(x,y)` aus. Der Text wird nicht direkt ausgegeben, sondern in die `pixmap` kopiert.

Implementierung:

Die Funktion ermittelt das Fonthandle des Zeichensatzes `font` mit der Funktion `getFont-Handle` und ruft die Funktion `drawTextC` auf, der auch die Länge des Textes übergeben wird.
In ihr werden die statischen Variablen gelesen, der `gc` auf KOPIEREN gestellt, der X-Font erzeugt und der Text dann mittels der X-Funktion `XDrawString` ausgegeben. Der `gc` muß geändert werden, da der Text sonst evtl. invertiert dargestellt werden würde, wenn direkt davor ein solcher Zeichenbefehl ausgeführt worden wäre (mit der Funktion `invertAreaC`).
Abschließend wird der Speicherplatz für die Zeichenkette freigegeben und die veränderten Variablen im Environment gespeichert.

Funktionsname:

getTextDims getTextDims

Signatur:

getTextDims:: env->font->string->(fontdims, env)

env [1]: Altes Environment

font: Zeichensatzinformation, die den Zeichensatz für den auszugebenden Font spezifiziert

string: Auszugebender Text

fontdims: Ausmaße des Textes

env [2]: Neues Environment

Beschreibung:

Die Funktion liefert die Größe einer Zeichenkette `string` in einem ausgewählten Zeichensatz `font`. Die Ausmaße der Zeichenkette werden als Datentyp `fontdims` (Höhe und Breite) in Pixel zurückgegeben.

Vor- und Nachbedingungen:

Die `initFontsC`- oder `initFontsTestingC`-Funktion muß bereits aufgerufen worden sein.

Implementierung:

Die Funktion ermittelt das Fonthandle des Zeichensatzes `font` mit der Funktion `getFontHandle` und ruft die C-Funktionen `getTextHeightC` und `getTextWidthC` auf.
In ihr werden jeweils die statischen Variablen gelesen und der X-Font erzeugt. Dann wird in `getTextHeightC` mit der X-Funktion `XTextExtents` die Höhe des Textes und in `getTextWidthC` mit der X-Funktion `XTextWidth` die Breite des Textes ermittelt.
Es müssen zwei Funktionen aufgerufen werden, damit wir unter C kein Tupel sondern jeweils einen Integer-Wert zurückgeben können.

Funktionsname:

redisplayArea redisplayArea

Signatur:

redisplayArea:: env->integer->integer->integer->integer->env

env [1]: Altes Environment

integer [1]: x-Koordinate linke obere Ecke

integer [2]: y-Koordinate linke obere Ecke

integer [3]: x-Koordinate rechte untere Ecke

integer [4]: y-Koordinate rechte untere Ecke

env [2]: Neues Environment

Beschreibung:

Stellt den Bereich `(x1,y1)` bis `(x2,y2)` im `canvas` neu dar, in dem die `pixmap` mit der aktuellen Seite in den `canvas` kopiert wird.

Implementierung:

Die Funktion ruft die C-Funktion `redisplayAreaC` auf. Diese Funktion ermittelt das `window` des `canvas`, das für die nachfolgende C-Funktion `XCopyPlane` gebraucht wird.
In ihr wird die `pixmap` in den `canvas` kopiert, wobei nur der angegebene Bereich kopiert wird. Wichtig für die Parameter von `XCopyPlane` ist, daß zwischen `pixmap` und `canvas` eine 1:1-Beziehung besteht, da beide dieselben Ausmaße haben.

Funktionsname:

redisplayPage redisplayPage

Signatur:

redisplayPage:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Stellt den kompletten `canvas` neu dar, in dem die `pixmap` mit der aktuellen Seite in den `canvas` kopiert wird.

Implementierung:

Ruft die Funktion `redisplayArea` mit den Ausmaßen des `canvas`, also dem Bereich von `(0,0)` bis `(1000,1000)` auf.

Funktionsname:

clearFontsC clearFontsC

Signatur:

clearFontsC:: env -> env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Löscht die `pixmap`, `gc` und weitere, von der Ausgabe erzeugte dynamische Daten.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden die Variablen gelesen und die Daten freigegeben.

Funktionsname:

clearAreaC clearAreaC

Signatur:

clearAreaC:: env->integer->integer->integer->integer->env

env [1]: Altes Environment

integer [1]: x-Koordinate linke obere Ecke

integer [2]: y-Koordinate linke obere Ecke

integer [3]: x-Koordinate rechte untere Ecke

integer [4]: y-Koordinate rechte untere Ecke

env [2]: Neues Environment

Beschreibung:

Löscht den Bereich der aktuellen Seite, der durch die Koordinaten `x1` (1. Integer), `y1` (2. Integer), `x2` (3. Integer) und `y2` (4. Integer) umspannt wird. Es handelt sich bei den Koordinaten um die linke obere Ecke und die rechte untere Ecke der Seite in absoluten Pixelwerten.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden die Variablen gelesen, der `gc` auf LÖSCHEN gesetzt und ein Rechteck der Größe gezeichnet, das den angegebenen Bereich umspannt. Durch das Setzen des Attributs für `gc` bewirkt das Zeichnen des Rechtecks das Löschen des Bereichs.
Abschließend wird `gc` wieder auf KOPIEREN gesetzt. Danach werden alle Änderungen gespeichert.

Funktionsname:

invertAreaC invertAreaC

Signatur:

invertAreaC:: env->integer->integer->integer->integer->env

env [1]: Altes Environment

integer [1]: x-Koordinate linke obere Ecke

integer [2]: y-Koordinate linke obere Ecke

integer [3]: x-Koordinate rechte untere Ecke

integer [4]: y-Koordinate rechte untere Ecke

env [2]: Neues Environment

Beschreibung:

Invertiert den Bereich der aktuellen Seite, der durch die Koordinaten `x1` (1. Integer), `y1` (2. Integer), `x2` (3. Integer) und `y2` (4. Integer) umspannt wird. Es handelt sich bei den Koordinaten um die linke obere Ecke und die rechte untere Ecke der Seite in absoluten Pixelwerten.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden die Variablen gelesen, der `gc` auf EXKLUSIV-ODER gesetzt und ein Rechteck der Größe gezeichnet, das den angegebenen Bereich umspannt. Durch das Setzen des Attributs für `gc` bewirkt das Zeichnen des Rechtecks das Löschen des Bereichs. Abschließend wird `gc` wieder auf KOPIEREN gesetzt und alle Änderungen gespeichert.

Funktionsname:

moveAreaC moveAreaC

Signatur:

moveAreaC:: env->integer->integer->integer->integer->integer->integer->env

env [1]: Altes Environment

integer [1]: x-Koordinate linke obere Ecke

integer [2]: y-Koordinate linke obere Ecke

integer [3]: x-Koordinate rechte untere Ecke

integer [4]: y-Koordinate rechte untere Ecke

integer [5]: Verschiebung in x-Richtung

integer [6]: Verschiebung in y-Richtung

env [2]: Neues Environment

Beschreibung:

Verschiebt den Bereich der aktuellen Seite, der durch die Koordinaten `x1` (1. Integer), `y1` (2. Integer), `x2` (3. Integer) und `y2` (4. Integer) umspannt wird. Es handelt sich bei den Koordinaten um die linke obere Ecke und die rechte untere Ecke der Seite in absoluten Pixelwerten.
Es wird um den Betrag `xmove` (5. Integer), und `ymove` (6. Integer) verschoben. Dabei handelt es sich um vorzeichenbehaftete Pixelwerte.

Positiver Wert : Verschiebung nach unten/rechts.

Negativer Wert : Verschiebung nach oben/links.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In der werden zunächst die Variablen gelesen. Danach wird mit `XGetImage` der umspannte Bereich ausgeschnitten und mit `XPutImage` an der neuen Position wieder eingefügt.
Danach werden alle geänderten Variablen gespeichert.

Funktionsname:

getViewOriginXC getViewOriginXC

Signatur:

getViewOriginXC:: env->(integer, env)

env [1]: Altes Environment

integer: x-Koordinate linke obere Ecke

env [2]: Neues Environment

Beschreibung:

Auf dem Bildschirm ist immer nur ein Teil des `canvas` sichtbar. Diese Funktion liefert die X-Koordinate des Pixels zurück, der in der linken oberen Ecke des sichtbaren Bereichs liegt.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen. Diese Funktion ermittelt das `window` des `canvas`, das wird für die nachfolgende C-Funktion zum Ermitteln der Ausmaße des sichtbaren Rechtecks gebraucht wird.
Vom gewonnenen Ausmaß wird dann die linke obere Ecke als Ergebnis geliefert und das neue Environment zurückgegeben.

Funktionsname:

getViewOriginYC getViewOriginYC

Signatur:

getViewOriginYC:: env->(integer, env)

env [1]: Altes Environment

integer: y-Koordinate linke obere Ecke

env [2]: Neues Environment

Beschreibung:

Auf dem Bildschirm ist immer nur ein Teil des `canvas` sichtbar. Diese Funktion liefert die Y-Koordinate des Pixels zurück, der in der linken oberen Ecke des sichtbaren Bereichs liegt.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen. Diese Funktion ermittelt das `window` des `canvas`, das wird für die nachfolgende C-Funktion zum Ermitteln der Ausmaße des sichtbaren Rechtecks gebraucht.
Davon wird dann die linke obere Ecke als Ergebnis geliefert und das neue Environment zurückgegeben.

Funktionsname:

getViewLastXC getViewLastXC

Signatur:

getViewLastXC:: env->(integer, env)

env [1]: Altes Environment

integer: x-Koordinate rechte untere Ecke

env [2]: Neues Environment

Beschreibung:

Auf dem Bildschirm ist immer nur ein Teil des `canvas` sichtbar. Diese Funktion liefert die X-Koordinate des Pixels zurück, der in der rechten unteren Ecke des sichtbaren Bereichs liegt.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen. Diese Funktion ermittelt das `window` des `canvas`, das wird für die nachfolgende C-Funktion zum Ermitteln der Ausmaße des sichtbaren Rechtecks gebraucht wird.
Davon wird dann die rechte untere Ecke als Ergebnis geliefert und das neue Environment zurückgegeben.

Funktionsname:

getViewLastYC getViewLastYC

Signatur:

getViewLastYC:: env->(integer, env)

env [1]: Altes Environment

integer: y-Koordinate rechte untere Ecke

env [2]: Neues Environment

Beschreibung:

Auf dem Bildschirm ist immer nur ein Teil des `canvas` sichtbar. Diese Funktion liefert die Y-Koordinate des Pixels zurück, der in der rechten unteren Ecke des sichtbaren Bereichs liegt.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen. Diese Funktion ermittelt das `window` des `canvas`, das wird für die nachfolgende C-Funktion zum Ermitteln der Ausmaße des sichtbaren Rechtecks gebraucht wird.
Davon wird dann die rechte untere Ecke als Ergebnis geliefert und das neue Environment zurückgegeben.

Funktionsname:

setCursorC setCursorC

Signatur:

setCursorC:: env->integer->integer->env

env [1]: Altes Environment

integer [1]: x-Koordinate der Cursorposition

integer [2]: y-Koordinate der Cursorposition

env [2]: Neues Environment

Beschreibung:

Setzt den Cursor an den Koordinaten `x` (1. Integer) und `y` (2. Integer). Es handelt sich bei den Koordinaten um die linke untere Ecke auf der aktuellen Seite in absoluten Pixelwerten. Der Cursorbereich in Form eines Dreiecks wird invers gezeichnet.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen.
Die Darstellung des Cursors erfolgt durch die inverse Darstellung des Cursorbereichs. Er kann nicht schwarz gezeichnet werden, da man sonst beim Löschen des Cursors nicht mehr den alten Hintergrund rekonstruieren könnte, er wäre in der `pixmap` gelöscht. So würden Teile des Textes verloren gehen.
Daher wird mit der X-Funktion `XFillPolygon` ein Dreieck gezeichnet, nachdem der `gc` zuvor auf EXKLUSIV-ODER (für Invertieren) gesetzt wurde. Das Dreieck wird so gezeichnet, daß seine Spitze die Koordinaten `(x,y)` hat. Die Breite beträgt `CURSORWIDTH`, die Höhe `CURSORHEIGHT`. Die beiden Werte sind Konstanten, die in `Output.xc` definiert werden.

Funktionsname:

clearCursorC clearCursorC

Signatur:

clearCursorC:: env->integer->integer->env

env [1]: Altes Environment

integer [1]: x-Koordinate der Cursorposition

integer [2]: y-Koordinate der Cursorposition

env [2]: Neues Environment

Beschreibung:

Löscht den Cursor an den Koordinaten `x` (1. Integer) und `y` (2. Integer). Es handelt sich bei den Koordinaten um die linke untere Ecke auf der aktuellen Seite in absoluten Pixelwerten. Um ihn zu löschen, wird der Cursor ein zweites Mal invers gezeichnet.

Implementierung:

Die Funktion `setCursorC` wird aufgerufen.

Funktionsname:

scrollWindowXC scrollWindowXC

Signatur:

scrollWindowXC:: env->integer->env

env [1]: Altes Environment

integer: x-Koordinate, zu der gescrollt werden soll

env [2]: Neues Environment

Beschreibung:

Scrollt das sichtbare Fenster so, daß die linke obere Ecke des sichtbaren Bereichs die erste sichtbare X-Koordinate in ihrem Tupel `(X,Y)` ist.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen.
Das Scrollen erfolgt in Einheiten zu x Pixeln, wobei x in der Konstanten `SCROLL_X_PIXEL` definiert ist. Der Wert ist in `Output.xc` definiert.
Beim Scrollen muß der Pixelbetrag `integer` in solche Einheiten umgerechnet werden. Dabei muß gerundet werden: Falls der Rest größer als 0,5 ist, wird aufgerundet, ansonsten wird der Scrollbar um den errechneten Betrag verschoben.

Funktionsname:

scrollWindowYC scrollWindowYC

Signatur:

scrollWindowYC:: env->integer->env

env [1]: Altes Environment

integer: y-Koordinate, zu der gescrollt werden soll

env [2]: Neues Environment

Beschreibung:

Scrollt das sichtbare Fenster so, daß die linke obere Ecke des sichtbaren Bereichs die erste sichtbare Y-Koordinate in ihrem Tupel `(X,Y)` ist.

Implementierung:

Die Funktion ruft die gleichnamige C-Funktion auf. In ihr werden zunächst die Variablen gelesen.
Das Scrollen erfolgt in Einheiten zu y Pixeln, wobei y in der Konstanten `SCROLL_Y_PIXEL` definiert ist. Der Wert ist in `Output.xc` definiert.
Beim Scrollen muß der Pixelbetrag `integer` in solche Einheiten umgerechnet werden. Dabei muß gerundet werden: Falls der Rest größer als 0,5 ist, wird aufgerundet, ansonsten wird der Scrollbar um den errechneten Betrag verschoben.

4.1.5 Lokale Funktionen

Funktionsname:

getFontHandle getFontHandle

Signatur:

getFontHandle:: env->font->(integer,env)

env [1]: Altes Environment

font: Zeichensatzinformation, die den Zeichensatz für den auszugebenden Font spezifiziert

integer: Fonthandle

env [2]: Neues Environment

Beschreibung:

Die Funktion ermittelt den von X11 erzeugten Handle für einen gewünschten Font. Wurde der Font noch nicht registriert, dann existiert ein solches Handle noch nicht. In dem Fall wird es mittels der C-Funktion `registerFont` kreiert.

Implementierung:

Es wird zunächst geprüft, ob der Font bereits registriert ist. Wurde der Font noch nicht registriert, existiert noch kein Fonthandle. Es wird dann mit der C-Funktion `registerFont` erzeugt und im Dictionary gespeichert.

4.1.6 Tests

Funktionsname:

initOutputTesting initOutputTesting

Signatur:

initOutputTesting:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Die Funktion initialisiert die Datenstrukturen der Ausgabe als *Testversion*. Nach Aufruf der Funktion ist noch nichts auf dem Bildschirm sichtbar, dazu dient die Funktion `clearFonts-CTesting`, die die X-Applikation aufruft. Das Ermitteln von Zeichensatzinformationen ist jedoch möglich.

Implementierung:

Zunächst wird die gleichnamige C-Funktion aktiviert.
Sie leistet annähernd dasselbe wie die Funktion `initOutputC`, allerdings baut sie den `frame` selber auf.

Funktionsname:

initFontsCTesting initFontsCTesting

Signatur:

initFontsCTesting:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Die Funktion ruft `initOutputTesting` auf.
Sie ist noch aus historischen Gründen erhalten geblieben.

Funktionsname:

clearFontsCTesting clearFontsCTesting

Signatur:

clearFontsCTesting:: env->env

env [1]: Altes Environment

env [2]: Neues Environment

Beschreibung:

Zunächst wird die gleichnamige C-Funktion aktiviert.
Diese ruft in der *Testversion* die X11-Applikation auf und löscht anschließend alle dynamisch angelegten Variablen der Ausgabegruppe.

4.2 Fonttypes

Guido Frick

4.2.1 Funktionalität

Das Modul stellt die Typen bereit, die für die Behandlung von Fonts benötigt werden. Dies sind neben dem Fontnamen, dem Fontstil und der Fontgröße ebenfalls eine Sorte, die alle drei Komponenten zusammenfaßt. Weiter werden Funktionen zur Verfügung gestellt, die Listen aller möglichen Fontnamen, -stile und -größen generieren.

4.2.2 Entwurf

Die Sorten wurden vorwiegend aufgrund benötigter Daten des Moduls `Output` definiert, das die tatsächliche Koordination mit dem X-System übernimmt und für die Ausgabe sorgt.

4.2.3 Öffentliche Schnittstellen

4.2.3.1 Sorten

Sortenname:

font font

Signatur:

font:: font
(fontname :: fontnamestring)
(fontstyle :: fontstylestring)
(fontsize :: fontsize)

Beschreibung:

Die Sorte faßt die drei Komponenten zusammen, die eindeutig eine bestimmte Schriftformatierung definiert.

Sortenname:

fontnamestring fontnamestring

Signatur:

fontnamestring:: (string)

Beschreibung:

Diese Sorte beinhaltet den Namen eines Fonts. Es handelt sich lediglich um ein Synonym für `string`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `font` verwendet.

Sortenname:

fontnameList fontnameList

Signatur:

fontnameList:: [fontnamestring]

Beschreibung:

Die Sorte definiert eine Liste von `fontnamestring`.

Sortenname:

fontstylestring fontstylestring

Signatur:

fontstylestring:: (string)

Beschreibung:

Diese Sorte beinhaltet einen Fontstil. Es handelt sich lediglich um ein Synonym für `string`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `font` verwendet.

Sortenname:

fontstyleList fontstyleList

Signatur:

fontstyleList:: [fontstylestring]

Beschreibung:

Die Sorte definiert eine Liste von `fontstylestring`.

Sortenname:

fontsize fontsize

Signatur:

fontsize:: (integer)

Beschreibung:

Diese Sorte beinhaltet die Größe eines Fonts. Es handelt es sich lediglich um ein Synonym für `integer`. Dieses wird zur Verbesserung der Lesbarkeit der Sorte `font` verwendet.

Sortenname:

fontsizeList fontsizeList

Signatur:

fontsizeList:: [fontsize]

Beschreibung:

Die Sorte definiert eine Liste von `fontsize`.

Sortenname:

fontdims fontdims

Signatur:

fontdims:: fontdims
  (dimwidth :: integer)
  (dimheight:: integer)

Beschreibung:

Die Sorte beschreibt die Ausmaße eine Textes in Pixel (Höhe und Breite).

4.2.3.2 Exportierte Funktionen

Funktionsname:

fontnames fontnames

Signatur:

fontnames:: fontnameList

fontnameList: Liste aller möglichen Fontfamilien

Beschreibung:

Die Funktion liefert die Liste aller möglichen Fontfamilien, die das System unterstützt.

Funktionsname:

fontstyles fontstyles

Signatur:

fontstyles:: fontstyleList

fontstyleList: Liste aller möglichen Fontstile

Beschreibung:

Die Funktion liefert die Liste aller möglichen Fontstyles, die das System unterstützt.

Funktionsname:

fontsizes fontsizes

Signatur:

fontsizes:: fontsizeList

fontsizeList: Liste aller möglichen Fontgrößen

Beschreibung:

Die Funktion liefert die Liste aller möglichen Fontgrößen, die das System unterstützt.

4.2.4 Lokale Implementierung

Das Modul stellt lediglich Sorten öffentlich bereit, die öffentlichen Funktionen selber benötigen aufgrund ihrer mangelnden Komplexität keine lokalen Unterfunktionen, sodaß diese hier nicht anfallen.

4.3 OutputStructure

4.3.1 Funktionalität

Das Modul enthält die Datenstrukturen der Ausgabe, die zum Verwalten der Fonts benötigt werden. Das Font-Management erfolgt in dem Modul `Output`.

4.3.2 Entwurf

Es gibt verschiedene Komponenten, aus denen sich ein Font zusammensetzt. Diese sind:

- der Fontname (z.B. Roman, Helvetica)
- der Fontstil (z.B. Bold, Italic)
- die Fontgröße (z.B. 12pt, 18pt)

Möchte man alle Kombinationen der Komponenten unterstützen, so ergeben sich beispielsweise bei jeweils 6 möglichen Werten jeder Komponente 6 hoch 3, also 216 verschiedene Kombinationen. Es ist ersichtlich, daß von diesen Kombinationen nur ein Bruchteil tatsächlich verwendet werden wird. Trotzdem muß im Extremfall die Unterscheidung aller dieser Möglichkeiten gewährleistet sein.

Um eine möglichst effiziente Speicherung dieser vielen Kombinationen zu erreichen, wird hier ein Dictionary verwendet, dessen Schlüssel sich aus dem Produkt aller drei Komponenten zusammensetzen. Da man kein Produkt aus "Roman", "Bold" und "12pt" bilden kann, werden hier die Positionen der einzelnen Komponenten in den Listen aller möglichen Werte herangezogen. Diese Listen werden in dem Modul `Fonttypes` definiert.

Wird ein Font im X-System initialisiert, so wird ein eindeutiger Integerwert für diesen Font vergeben (*Fonthandle*). Dieser Wert wird in das Dictionary geschrieben. Wird der Font wieder angesprochen, so wird aus diesem Dictionary dieser *Fonthandle* ausgelesen, mit dem das X-System sofort den bereits initialisierten Font verwenden kann (siehe hierzu auch das Modul `Output`).

4.3.3 Öffentliche Schnittstellen

4.3.3.1 Sorten

Sortenname:

fontHandle fontHandle

Signatur:

fontHandle:: (integer)

Beschreibung:

Das X-System identifiziert einen Font anhand eines Integerwertes, der zum Zeitpunkt der Initialisierung des Fonts durch das X-System selber vergeben wird. Dieser Integerwert wird *Fonthandle* genannt.

Sortenname:

fontDict fontDict

Signatur:

fontDict:: Die Signatur der Sorte `fontDict` wird nicht öffentlich exportiert. Die einzelnen Komponenten werden später bei den lokalen Sorten aufgeführt.

Beschreibung:

Die Zuordnung Font-Fonthandle wird in der Datenstruktur `fontDict` festgehalten.

Sortenname:

os os

Signatur:

os:: Die Signatur der Sorte os wird nicht öffentlich exportiert. Die einzelnen Komponenten werden später bei den lokalen Sorten aufgeführt.

Beschreibung:

Die Datenstruktur `os` faßt alle Datenstrukturen zusammen, die funktionsübergreifend gespeichert werden müssen.

4.3.3.2 Exportierte Funktionen

Funktionsname:

mtOS mtOS

Signatur:

mtOS::

os: Ausgabe-Struktur

Beschreibung:

Die Funktion liefert eine initialisierte Datenstruktur, in der alle für die Ausgabe benötigten Daten gespeichert werden.

Funktionsname:

mtFontHandle mtFontHandle

Signatur:

mtFontHandle:: fontHandle

fontHandle: Leerer *Fonthandle*

Beschreibung:

Die Funktion liefert einen leeren *Fonthandle*, der vom X-System zur Identifizierung von Fonts vergeben wird. Bei diesem handelt es sich um einen Wert vom Typ Integer.

Funktionsname:

getDictFontHandle getDictFontHandle

Signatur:

getDictFontHandle:: fontDict -> font -> (boolean, fontHandle)

fontDict: Dictionary der bereits erzeugten Fonts

font: Gewünschter Font

boolean: Kennzeichen, ob der Font bereits initialisiert war

fontHandle: Vom X-System vergebener Integerwert zur Identifizierung des Fonts

Beschreibung:

Die Funktion liefert zu einem gegebenen Font den vom X-System benötigten *Fonthandle*. Die Zuordnung Font-Fonthandle erfolgt in einem Dictionary.

Funktionsname:

setDictFontHandle setDictFontHandle

Signatur:

setDictFontHandle:: fontDict -> font -> fontHandle -> fontDict

fontDict: Dictionary der bereits erzeugten Fonts

font: Einzutragender Font

boolean: Kennzeichen, ob der Font bereits initialisiert war

fontHandle: *Fonthandle* des neuen Fonts

fontDict: Erweitertes Dictionary

Beschreibung:

Die Funktion schreibt eine Zuordnung Font-Fonthandle. Die Ermittlung des Fonthandles für einen bestimmten Font wird vom X-System durchgeführt und in dieser Funktion lediglich festgehalten.

Funktionsname:

getFontDict getFontDict

Signatur:

getFontDict:: os -> fontDict

os: Ausgabe-Struktur

fontDict: Dictionary der bereits erzeugten Fonts

Beschreibung:

Die Funktion liefert aus der Struktur der Ausgabe das Dictionary, in dem die Zuordnung von Fonts zu deren `Fonthandle` festgehalten werden.

Funktionsname:

setFontDict setFontDict

Signatur:

setFontDict:: os -> fontDict -> os

os: Ausgabe-Struktur

fontDict: Dictionary der bereits erzeugten Fonts

os: Geänderte Ausgabe-Struktur

Beschreibung:

Die Funktion schreibt ein verändertes Dictionary zurück in die Struktur der Ausgabe.

4.3.4 Lokale Implementierung

4.3.4.1 Sorten

Sortenname:

fontDict fontDict

Signatur:

fontDict:: Dict ACTUAL
SORTS
  dom = fontkey.
  codom = fontHandle.
  dictonary = fontDict.
OPNS
  hash = hashFontKey.
  errorval = mtFontHandle.
END+

Beschreibung:

Die Zuordnung Font-Fonthandle wird in der Datenstruktur `fontDict` festgehalten. Bei dieser handelt es sich um ein Dictionary, dessen Schlüssel sich aus dem Fontnamen, dem Fontstil und der Fontgröße zusammensetzt.

Sortenname:

fontkey fontkey

Signatur:

fontkey:: fontkey
  (nameidx :: integer)
  (styleidx :: integer)
  (sizeidx :: integer)

Beschreibung:

Als Schlüssel des Dictionary dient der Fontname, der Fontstil und die Fontgröße. Der Typ `fontkey` faßt die Indize aller drei Komponenten zusammen. Die Indize ergeben sich aus der Position der jeweiligen Komponente in einer Liste aller möglichen Werte. Die Listen sind in dem Modul `Fonttypes` definiert.

Sortenname:

os os

Signatur:

os:: os (v_fontDict :: fontDict)

Beschreibung:

Die Datenstruktur `os` faßt alle Datenstrukturen zusammen, die funktionsübergreifend gespeichert werden müssen. Derzeit handelt es sich hierbei nur um `fontDict`, aus Gründen der Erweiterbarkeit wird aber eine höhere Struktur verwendet, die noch weitere Datenstrukturen aufnehmen kann.

4.3.4.2 Lokale Funktionen

Funktionsname:

hashFontKey hashFontKey

Signatur:

hashFontKey:: fontkey -> integer

fontkey: Schlüsselstruktur eines Fonts

integer: Eindeutiger Zugriffsschlüssel für das Dictionary

Beschreibung:

Die Funktion ermittelt aus einem `Fontkey` einen Integer-Wert, der als Schlüssel für das Dictionary dient. Dieser Wert ist das Produkt aller drei Komponenten des `Fontkeys` (siehe dort).

Funktionsname:

getFontKey getFontKey

Signatur:

getFontKey:: font -> fontkey

font: Zu verschlüsselnder Font

fontkey: Zugehöriger `Fontkey`

Beschreibung:

Die Funktion ermittelt aus einem Font, bestehend aus dem Fontnamen, dem Fontstil und der Fontgröße den korrespondierende `Fontkey`. Dieser setzt sich aus den Indize der drei Komponenten zusammen. Die Indize spiegeln die Positionen der jeweiligen Komponenten in den Listen aller möglichen Werte wider. Die Listen werden in dem Modul `Fonttypes` definiert.

Abhängigkeiten:

Es werden die Funktionen `fontnames`, `fontstyles` und `fontsizes` des Moduls `Fonttypes` verwendet.

4.4 OutputFormat

4.4.1 Funktionalität

Das Modul enthält die Funktionen, die dem Formatierer von der Ausgabe bereitgestellt werden. Es handelt sich nur um zwei Funktionen, die in dem Schnittstellenmodul `OutputFormat` direkt zwei Funktionen des Moduls `Output` aufrufen.

4.4.2 Entwurf

Ziel des Moduls ist es, dem Formatierer eine klare Schnittstelle zu liefern, in der er die von ihm benutzbaren Funktionen der Ausgabe bereitgestellt bekommt. Es soll das Geheimnis-Prinzip wahren helfen.

4.4.3 Öffentliche Schnittstellen

Funktionsname:

o_initOutput o_initOutput

Signatur:

o_initOutput:: env -> env

env: Environment

env: geändertes Environment

Beschreibung:

Die Funktion initialisiert die Datenstrukturen der Ausgabe.

Implementierung:

Die Funktion leitet den Aufruf direkt an das Modul `Output` durch die Funktion `initOutput` weiter.

Funktionsname:

o_getTextDims o_getTextDims

Signatur:

o_getTextDims:: env -> font-> string -> (fontdims, env)

env: Environment

font: Zeichensatzinformation, die den Zeichensatz für den auszugebenden Text spezifiziert

string: Auszugebender Text

fontdims: Ausmaße des Textes

env: Geändertes Environment

Beschreibung:

Die Funktion liefert die Größe einer Zeichenkette (string) in einem ausgewählten Font. Die Ausmaße der Zeichenkette wird als Datentyp `fontdims` (Höhe und Breite in Pixel) zurückgegeben.

Implementierung:

Die Funktion leitet den Aufruf direkt an das Modul `Output` durch die Funktion `getTextDims` weiter.

4.4.4 Lokale Implementierung

Das Modul verwendet Funktionen des Moduls `Output` und enthält selber keine lokalen Sorten oder Funktionen.

4.5 OutputUI

4.5.1 Funktionalität

Das Modul enthält die Funktionen, die der Benutzungsoberfläche bereitgestellt werden, um alle Aufgaben der Bildschirmausgabe erledigen zu können.

4.5.2 Entwurf

Die Benutzungsoberfläche ruft Funktionen auf, die direkt Einfluß auf die Ausgabe auf dem Bildschirm haben. Hierbei handelt es sich um Funktionen um

- eine komplette formatierte Seite darzustellen
- Änderungen auf einer Seite auf dem Bildschirm zu aktualisieren
- den Cursor zu setzen bzw. zu löschen
- Textteile hervorzuheben (markieren).

Diese Aufgaben der Ausgabe übernimmt dieses Modul. Es stützt sich hierbei vorwiegend auf elementare Funktionen des Moduls `Output`, die die tatsächlichen Änderungen in den Strukturen der Ausgabe vornehmen. Es handelt sich hierbei um das Initialisieren der Ausgabe-Strukturen, Funktionen zum Ausgeben von Text, Löschen, Invertieren und Verschieben von Bereichen einer Seite, Funktionen zum Setzen und Löschen eines Cursors und um Funktionen zum Ändern des aktuellen Fensterausschnittes.

4.5.3 Öffentliche Schnittstellen

4.5.3.1 Sorten

Sortenname:

pixelcoordinates pixelcoordinates

Signatur:

pixelcoordinates:: (v_Xcoordinate :: integer, v_Ycoordinate :: integer)

Beschreibung:

Für die Darstellung von Objekten auf dem Bildschirm ist eine Spezifizierung der Koordinate erforderlich. Die Position wird durch `pixelcoordinates` angegeben. Dies ist das Pendant zu den `pagecoordinates` der Formatiererstruktur, nur daß man auf dem Bildschirm nicht zwischen verschiedenen Seiten zu unterscheiden braucht. Die Umrechnung einer `pagecoordinate` in eine `pixelcoordinate` erfolgt mittels der Funktion `getPixelCoord`.

4.5.3.2 Exportierte Funktionen

Funktionsname:

o_Init o_Init

Signatur:

o_Init:: env -> env

env: Environment

env: Geändertes Environment

Beschreibung:

Die Funktion initialisiert alle Datenstruturen der Ausgabe zum Programmstart.

Abhängigkeiten:

Es wird die Funktion `initOutput` aus dem Modul `Output` verwendet.

Funktionsname:

o_Exit o_Exit

Signatur:

o_Exit:: env -> env

env: Environment

env: Geändertes Environment

Beschreibung:

Die Funktion setzt alle Datenstrukturen der Ausgabe in einen Zustand, der das korrekte beenden des Programmes ermöglicht.

Implementierung:

Die Funktion ist für den Fall vorgesehen, daß durch Erweiterungen Abschlußfunktionen aufgerufen werden müssen. Dies ist bei dem derzeitigen Programmstand nicht erforderlich, sodaß die Funktion keinerlei Wirkung hat.

Funktionsname:

o_GetObjID o_GetObjID

Signatur:

o_GetObjID:: env -> pagecoordinates -> (env, objID)

env: Environment

pagecoordinates: aktuelle Cursorposition

env: Geändertes Environment

objID: Objekt-ID, auf dem sich der Cursor momentan befindet

Beschreibung:

Die Funktion ermittelt die eindeutige Objekt-ID des Objektes, auf dem sich der Cursor aktuell befindet. Die genaue Cursorposition wird durch die Seite und den X-Y-Koordinaten angegeben (`pagecoordinates`).

Abhängigkeiten:

Es wird die Funktion `f_getObjID` aus dem Modul `FormatOutput` verwendet.

Funktionsname:

o_InitPage o_InitPage

Signatur:

o_InitPage:: env -> integer -> env

env: Environment

integer: Darzustellende Seitennummer

env: Geändertes Environment

Beschreibung:

Die Funktion baut eine durch die Seitennummer spezifizierte Seite auf. Es wird unterschieden, ob die aktuelle Seite komplett neu aufgebaut werden soll, oder ob nur Änderungen an einer bereits aufgebauten Seite vorgenommen werden müssen. Die Funktion übernimmt das Neuaufbauen.

Abhängigkeiten:

Es wird die Funktion `f_getEntirePage` aus dem Modul `FormatOutput` sowie aus dem Modul `Output` die Funktionen `clearAreaC` und `redisplayPage` verwendet.

Implementierung:

Das Aufbauen der Seite geschieht in der Datenstruktur der Ausgabe. Die Funktion fordert alle Daten der Seite von dem Formatierer an und baut diese im ganzen auf, nachdem der gesamte *Canvas* gelöscht wurde. Der derzeitige sichtbare Ausschnitt wird im Anschluß in den *Canvas* kopiert.

Funktionsname:

o_UpdatePage o_UpdatePage

Signatur:

o_UpdatePage:: env -> pagecoordinates -> env

env: Environment

pagecoordinates: Aktuelle Position des Cursors

env: Geändertes Environment

Beschreibung:

Die Funktion führt alle angefallenen Änderungen auf der aktuellen Seite durch.

Vor- und Nachbedingungen:

Die zu aktualisierende Seite muß vorher bereits mit `o_InitPage` komplett aufgebaut worden sein.

Abhängigkeiten:

Es wird die Funktion `f_getChanges` aus dem Modul `FormatOutput` sowie `redisplayPage` aus dem Modul `Output` verwendet.

Implementierung:

Die Änderungen setzen sich aus 3 Komponenten zusammen:

- Clear
- Insert
- Move

`Clear` beschreibt einen Block eines zu löschenden Bereiches, `Move` einen Block und ein Verschiebevektor eines zu verschiebenden Bereiches und `Insert` einen einzufügenden Zeichenstrom.
Die Änderungen werden auf der gesamten Seite durchgeführt. Der derzeitig sichtbare Ausschnitt wird im Anschluß in den *Canvas* kopiert.

Funktionsname:

o_GetXYPos o_GetXYPos

Signatur:

o_GetXYPos:: env -> integer -> integer -> (env, integer, integer)

env: Environment

integer: X-Koordinate X-System

integer: Y-Koordinate X-System

env: Geändertes Environment

integer: X-Koordinate interne Pixeldarstellung

integer: Y-Koordinate interne Pixeldarstellung

Beschreibung:

Die Funktion rechnet von den Window-Koordinaten des X-Systems in die interne Pixeldarstellung der aktuellen Seite um. Die Information, um welche Seite es sich bei der aktuellen handelt, wird in dem Modul der Benutzungsoberfläche gespeichert.

Implementierung:

Die von der BO übergebenen Koordinaten stellen bereits absolute Werte dar. Die Veränderung des Fensterausschnittes durch die Scrollbars werden bereits mit eingerechnet. Die Funktion bleibt aber für spätere Erweiterungen bestehen, bei denen die Ermittlung der Koordinaten ausgehend von den X-System-Koordinaten in die interne Pixeldarstellung eine Umrechnung bedarf.

Funktionsname:

o_ClearCursor o_ClearCursor

Signatur:

o_ClearCursor:: env -> pagecoordinates -> env

env: Environment

pagecoordinates: Aktuelle Position des Cursors

env: Geändertes Environment

Beschreibung:

Die Funktion löscht den Cursor an der durch `pagecoordinates` beschriebene Stelle im Dokument.

Abhängigkeiten:

Es werden die Funktionen `clearCursorC` und `redisplayPage` aus dem Modul `Output` verwendet.

Funktionsname:

o_SetCursor o_SetCursor

Signatur:

o_SetCursor:: env -> pagecoordinates -> (env, pagecoordinates, objID, integer)

env: Environment

pagecoordinates: Gewünschte Position des Cursors

env: Geändertes Environment

pagecoordinates: Aktuelle Position des Cursors

objID: ID des Objektes, auf dem sich der Cursor befindet

integer: Position des Zeichen, vor dem sich der Cursor im aktuellen Objekt befindet

Beschreibung:

Die Funktion setzt den Cursor an eine durch `pagecoordinates` beschriebene Stelle im Dokument. Die tatsächlich neue Cursorposition, die aktuelle Objekt-ID und der Index der Stelle im aktuellen Objekt wird zurückgeliefert.

Abhängigkeiten:

Es werden die Funktionen `setCursorC` und `redisplayPage` aus dem Modul `Output` sowie `f_getCursorPos` aus dem Modul `FormatOutput` verwendet.

Implementierung:

Die tatsächlich neue Cursorposition ergibt sich aus der Rundung der aktuellen Cursorposition durch `f_getCursorPos`. Befindet sich der Cursor außerhalb des aktuell sichtbaren Bereiches, so wird dieser entsprechend angepaßt.

Funktionsname:

o_DrawCursor o_DrawCursor

Signatur:

o_DrawCursor:: env -> pagecoordinates -> env

env: Environment

pagecoordinates: Neue Position des Cursors env: Geändertes Environment

Beschreibung:

Die Funktion setzt den Cursor an der übergebenen Position.

Abhängigkeiten:

Es werden die Funktionen `setCursorC` und `redisplayPage` aus dem Modul `Output` verwendet.

Funktionsname:

o_MarkText o_MarkText

Signatur:

o_MarkText:: env -> box -> env

env: Environment

box: Zu markierender Bereich

env: Geändertes Environment

Beschreibung:

Die Funktion markiert einen Bereich im Text. Dieser Bereich wird durch 4 Koordinaten beschrieben (`box`).

Abhängigkeiten:

Es werden die Funktionen `invertAreaC` und `redisplayPage` aus dem Modul `Output` verwendet.

Implementierung:

Das Markieren erfolgt durch Invertieren des übergebenen Bereiches durch Aufrufe der Funktion `invertAreaC` des Moduls `Output`, der durch den Datentypen `box` beschrieben wird.

4.5.4 Lokale Implementierung

Funktionsname:

drawWords drawWords

Signatur:

drawWords:: env -> integer -> wordattributelist -> env

env: Environment

integer: Aktuelle Seitennummer

wordattributelist: Liste von auszugebenen Zeichenketten

env: Geändertes Environment

Beschreibung:

Die Funktion gibt Zeichenketten an bestimmten Koordinaten in einem spezifizierten Font aus.

Abhängigkeiten:

Es wird die Funktion `drawText` aus dem Modul `Output` verwendet.

Implementierung:

Die auszugebenen Zeichenketten werden durch eine `wordattributelist` beschrieben, dessen Listenelemente neben dem Text ebenfalls die Koordinaten und den zu verwendenden Font enthalten. Die tatsächliche Ausgabe geschieht in der Funktion `drawText` des Moduls `Output`, in der die auzugebenen Texte vorerst in die `pixmap` kopiert werden (siehe dort).

Funktionsname:

clearBox clearBox

Signatur:

clearBox:: env -> box -> env

env: Environment

box: Zu löschender Bereich

env: Geändertes Environment

Beschreibung:

Die Funktion löscht einen Bereich auf der aktuellen Seite. Dieser Bereich wird durch 4 Koordinaten beschrieben (`box`).

Abhängigkeiten:

Es wird die Funktion `clearAreaC` aus dem Modul `Output` verwendet.

Funktionsname:

moveBox moveBox

Signatur:

moveBox:: env -> box -> integer -> env

env: Environment

box: Zu verschiebender Bereich

integer: Anzahl Pixel, um die nach oben (negativ) bzw. nach unten (positiv) verschoben werden soll

env: Geändertes Environment

Beschreibung:

Die Funktion verschiebt einen Bereich vertikal auf der aktuellen Seite. Dieser Bereich wird durch 4 Koordinaten beschrieben (`box`).

Abhängigkeiten:

Es wird die Funktion `moveAreaC` aus dem Modul `Output` verwendet.

Implementierung:

Ist die Pixelzahl negativ, so bedeutet dies eine Verschiebung nach oben, ansonsten eine nach unten. Soll nach unten verschoben werden, dann muß zuerst der dritte, der zweite und dann der erste Block verschoben werden. Bei einer Verschiebung nach oben hat dies in umgekehrter Reihenfolge zu erfolgen.
Das Löschen erfolgt durch die Funktion `moveAreaC` des Moduls `Output`.

Funktionsname:

getPixelCoord getPixelCoord

Signatur:

getPixelCoord:: env -> pagecoordinates -> pixelcoordinates

env: Environment

pagecoordinates: Koordinaten des Formatierers

pixelcoordinates: Koordinaten der Ausgabe

Beschreibung:

Die Funktion errechnet anhand übergebener Koordinaten des Formatierers (`pagecoordinates`) die Koordinaten auf der aktuell dargestellten Seite (`pixelcoordinates`).

Implementierung:

Derzeit entsprechen sich beide Koordinaten.

Funktionsname:

setWindowCoord setWindowCoord

Signatur:

setWindowCoord:: env -> pixelcoordinates -> env

env: Environment

pixelcoordinates: Koordinaten der Ausgabe

env: Geändertes Environment

Beschreibung:

Die Funktion errechnet anhand der übergebenen Koordinaten des Cursors die neue Startkoordinate des angezeigten Fensters. Wird der Cursor außerhalb des sichtbaren Bereiches bewegt, so muß dieser Bereich entsprechend angepaßt werden. Ändert sich der sichtbare Bereich, so wird das Fenster neu gesetzt.

Abhängigkeiten:

Aus dem Modul `Output` werden die Funktionen `getViewOriginXC`, `getViewOriginYC`, `getViewLastXC`, `getViewLastYC`, `scrollWindowXC` und `scrollWindowYC` verwendet.

Implementierung:

Der Algorithmus arbeitet wie folgt:

X1, Y1.... Anfangskoordinate des sichtbaren Bereiches
X2, Y2.... Endkoordinate des sichtbaren Bereiches
Xc, Yc.... Koordinate des gesetzten Cursors
Xw, Yw.... neue Anfangskoordinate des sichtbaren Bereiches (Window)

Lh........ Lineheight (Höhe der aktuellen Zeile, hochscrollen)
Ch........ Cursorheight (Höhe des Cursors, runterscrollen)
Cw........ Cursorwidth (Breite des Cursors, links-rechts scrollen)

1. Xc < X1 ==> Xw = Xc
2. Xc > X2 ==> Xw = X1 + (Xc - X2)
3. sonst ==> Xw = X1

Damit der Cursor immer komplett zu sehen ist, wird zu der Cursorposition immer noch ein entsprechender Umgebungsbereich addiert bzw. abgezogen. Für das Bewegen nach links, rechts und unten muß lediglich die Größe des Cursors berücksichtigt werden (hier werden fest 5 Punkt angenommen). Für das Bewegen nach oben muß die Höhe der aktuellen Zeile berücksichtigt werden, da diese noch komplett sichtbar sein soll. Daraus ergeben sich folgende geänderten Beziehungen:

1. (Xc - Cw) < X1 ==> Xw = (Xc - Cw)
2. (Xc + Cw) > X2 ==> Xw = X1 + (Xc - X2) + Cw
3. sonst ==> Xw = X1

Zu beachten ist, daß Xw nicht kleiner als 0 werden kann! Analoges gilt für die Y-Koordinaten:

1. (Yc - Lh) < Y1 ==> Yw = (Yc - Lh)
2. (Yc + Ch) > Y2 ==> Yw = Y1 + (Yc - Y2) + Ch
3. sonst ==> Yw = Y1

Funktionsname:

getFirstBoxCoord getFirstBoxCoord

Signatur:

getFirstBoxCoord:: env -> box -> (integer, integer, integer, integer)

env: Environment

box: Gesamte Box

integer: X-Koordinate der oberen linken Ecke

integer: Y-Koordinate der oberen linken Ecke

integer: X-Koordinate der unteren rechten Ecke

integer: Y-Koordinate der unteren rechten Ecke

Beschreibung:

Die Funktion ermittelt die Koordinaten des ersten Blocks der 3 Blöcke, die durch den Typen box beschrieben werden. Um die Überschneidung mit der zweiten Boxkomponente zu vermeiden, endet die Boxkomponente eine Zeile höher als angegeben.

Funktionsname:

getSecondBoxCoord getSecondBoxCoord

Signatur:

getSecondBoxCoord:: env -> box -> (integer, integer, integer, integer)

env: Environment

box: Gesamte Box

integer: X-Koordinate der oberen linken Ecke integer: Y-Koordinate der oberen linken Ecke

integer: X-Koordinate der unteren rechten Ecke

integer: Y-Koordinate der unteren rechten Ecke

Beschreibung:

Die Funktion ermittelt die Koordinaten des zweiten Blocks der 3 Blöcke, die durch den Typen box beschrieben werden.

Funktionsname:

getThirdBoxCoord getThirdBoxCoord

Signatur:

getThirdBoxCoord:: env -> box -> (integer, integer, integer, integer)

env: Environment

box: Gesamte Box

integer: X-Koordinate der oberen linken Ecke

integer: Y-Koordinate der oberen linken Ecke

integer: X-Koordinate der unteren rechten Ecke

integer: Y-Koordinate der unteren rechten Ecke

Beschreibung:

Die Funktion ermittelt die Koordinaten des dritten Blocks der 3 Blöcke, die durch den Typen box beschrieben werden. Um die Überschneidung mit der zweiten Boxkomponente zu vermeiden, beginnt die Boxkomponente eine Zeile tiefer als angegeben.

5. CTools

Kai Hofmann

Bei den CTools handelt es sich um Hilfsmodule für die C-Programmierung des ForauS-Systems unter ASpecT und im Umgang mit der Karlsruher-Compiler-Bau-Toolbox.

5.1 AviseC

Dieses Modul besteht aus den Files `AviseC.h`, `AviseC.c`.

5.1.1 Funktionalität

Das Modul AviseC stellt in erster Linie Funktionen zum Konvertieren verschiedener Datentypen zwischen C und ASpecT zur Verfügung. Zusätzlich auch noch eine Funktion, die die Fehlerübergabe an das ForauS-System von C-Seite aus stark vereinfacht.

5.1.2 Öffentliche Schnittstellen

5.1.2.1 Globale Variablen

Sortenname:

FORAUSenv FORAUSenv

Beschreibung:

Globaler Pointer, der auf das aktuelle ForauS-Environment zeigt.

5.1.2.2 Funktionen

Funktionsname:

SHORTtoTERM SHORTtoTERM

Signatur:

TERM SHORTtoTERM(short x)

x: `short int` der nach ASpecT zu konvertieren ist.

Rückgabewert: ASpecT Integer.

Beschreibung:

Wandelt einen C `short` int Wert in ein ASpecT Integer.

Vor- und Nachbedingungen:

`x` muß vom Typ `short int` sein.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

INTtoTERN INTtoTERN

Signatur:

TERM INTtoTERM(int x)

x: Integer der nach ASpecT zu konvertieren ist.

Rückgabewert: ASpecT Integer.

Beschreibung:

Wandelt einen C `int` Wert in ein ASpecT Integer.

Vor- und Nachbedingungen:

`x` muß vom Typ `int` sein.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

BOOLtoTERM BOOLtoTERM

Signatur:

TERM BOOLtoTERM(bool x)

x: Boolean der nach ASpecT zu konvertieren ist.

Rückgabewert: ASpecT Boolean.

Beschreibung:

Wandelt einen C bool Wert in ein ASpecT Boolean.

Vor- und Nachbedingungen:

`x` muß vom Typ bool sein.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

CHARtoTERM CHARtoTERM

Signatur:

TERM CHARtoTERM(char x)

x: Char der nach ASpecT zu konvertieren ist.

Rückgabewert: ASpecT Char.

Beschreibung:

Wandelt einen C `char` Wert in ein ASpecT Character.

Vor- und Nachbedingungen:

`x` muß vom Typ char sein.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

STRINGtoTERM STRINGtoTERM

Signatur:

TERM STRINGtoTERM(char *x)

x: String der nach ASpecT zu konvertieren ist.

Rückgabewert: ASpecT String.

Beschreibung:

Wandelt einen C String in einen ASpecT String.

Vor- und Nachbedingungen:

`x` muß ein Pointer auf eine String sein.

Abhängigkeiten:

ASpecT-Runtime-Bibliothek

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

REALtoTERM REALtoTERM

Signatur:

TERM REALtoTERM(float x)

x: Float der nach ASpecT zu konvertieren ist.

Rückgabewert: ASpecT Real.

Beschreibung:

Wandelt einen C `float` Wert in ein ASpecT Real.

Vor- und Nachbedingungen:

`x` muß vom Typ `float` sein.

Abhängigkeiten:

ASpecT Bibliothek `REAL.AS`

Implementierung:

Es ist nur ein einfaches Macro. Als Ergänzung zu den Macros von Michael Fröhlich und Mattias Werner. Ruft die ASpecT-Funktion `toTERM` auf.

Funktionsname:

TERMtoSHORT TERMtoSHORT

Signatur:

short int TERMtoSHORT(TERM x)

x: TERM der von ASpecT in ein C short konvertiert werden soll.

Rückgabewert: C short.

Beschreibung:

Wandelt einen ASpecT Integer in ein C `short`.

Vor- und Nachbedingungen:

`x` muß vom Typ `TERM` sein bzw. ein ASpecT Integer.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

TERMtoINT TERMtoINT

Signatur:

int TERMtoINT(TERM x)

x: TERM der von ASpecT in ein C int konvertiert werden soll.

Rückgabewert: C int.

Beschreibung:

Wandelt einen ASpecT Integer in ein C `int`.

Vor- und Nachbedingungen:

`x` muß vom Typ `TERM` sein bzw. ein ASpecT Integer.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

TERMtoBOOL TERMtoBOOL

Signatur:

bool TERMtoBOOL(TERM x)

x: TERM der von ASpecT in ein C bool konvertiert werden soll.

Rückgabewert: C == Wert (bool)

Beschreibung:

Wandelt einen ASpecT Boolean in ein C bool.

Vor- und Nachbedingungen:

`x` muß vom Typ `TERM` sein bzw. ein ASpecT Boolean.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

TERMtoCHAR TERMtoCHAR

Signatur:

char TERMtoCHAR(TERM x)

x: TERM der von ASpecT in ein C char konvertiert werden soll.

Rückgabewert: C char.

Beschreibung:

Wandelt einen ASpecT Char in ein C `char`.

Vor- und Nachbedingungen:

`x` muß vom Typ `TERM` sein bzw. ein ASpecT Char.

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

TERMtoSTRING TERMtoSTRING

Signatur:

char * TERMtoSTRING(TERM x)

x: TERM der von einem ASpecT-String in einen C String konvertiert werden soll.

Rückgabewert: C char*

Beschreibung:

Wandelt einen ASpecT String in einen C String. Auf keinen Fall vergessen den erhaltenen String nach Gebrauch mit `free()` wieder zu befreien!

Vor- und Nachbedingungen:

`x` muß vom Typ `TERM` sein bzw. ein ASpecT String.

Abhängigkeiten:

`TERM_to_HEAP_STRING(x)`

Implementierung:

Es ist nur ein einfaches Macro. Von Michael Fröhlich und Mattias Werner übernommen.

Funktionsname:

TERMtoREAL TERMtoREAL

Signatur:

float TERMtoREAL(TERM x)

x: TERM der von ASpecT-Real in einen C float konvertiert werden soll.

Rückgabewert: C float

Beschreibung:

Wandelt einen ASpecT Real in ein C `float`.

Vor- und Nachbedingungen:

`x` muß vom Typ `TERM` sein bzw. ein ASpecT Real.

Abhängigkeiten:

ASpecT Bibliothek `REAL.AS`

Implementierung:

Es ist nur ein einfaches Macro. Als Ergänzung zu den Macros von Michael Fröhlich und Mattias Werner. Ruft die ASpecT-Funktion `toREAL` auf.

Funktionsname:

announceError announceError

Signatur:

TERM announceError(TERM env, char *modulename, TERM errtype, int errnum,
char *string[])

env: Aktuelles $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$-Environment.

modulename: Name des Modules, welches einen Fehler meldet.

errtype: Typ des Fehlers, siehe hierzu: `ISTypes.AS`

errnum: Modulinterne Nummer des Fehlers.

string: Stringliste mit den Argumenten der Fehlermeldung.

Rückgabewert: Neues $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$-Environment

Beschreibung:

Vereinfacht die Übergabe von Fehlern an das $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$-System.

Vor- und Nachbedingungen:

Die Liste von Strings muß mit einem `NULL` Pointer abgeschlossen werden!

Abhängigkeiten:

`STRINGtoTERM(x)`, `INTtoTERM(x)`, `IS.AS`, ASpecT-Bibliothek: `Strings.AS`

Implementierung:

Diese Funktion wurde notwendig, weil die ASpecT Funktion `i_createError`, welche die IS zur Verfügung stellt, eine Liste von Strings benötigt. Diese Liste ist in C aber nur mit etwas Aufwand aufbaubar! Hierzu wird unter anderem der ASpecT : Operator benötigt, welcher ein Element vor eine Liste hängt! Da der Funktion ein Array von Pointern auf Strings übergeben wird, wird zunächst einmal die Anzahl der übergebenen Strings mittels einer einfachen Schleife ermittelt. Sollte kein String übergeben werden, wird bei `i_createError` eine leere Liste benutzt! Ansonsten wird zunächst eine Liste mit nur einem Element erzeugt (aus einer leeren Liste und dem letzten String). Sollten nun noch mehr Strings vorhanden sein, so werden diese entsprechend in die Liste eingefügt. Das Einfügen geschieht rückwärts, da der : Operator die Elemente jeweils an den Anfang der Liste hängt! Ist dies nun geschehen, so wird mittels der ASpecT Funktion `i_createError` ein Fehler erzeugt, und auch gleich mit Hilfe der ASpecT Funktion `i_announceError`, welche ebenfalls von der IS stammt, dem ForauS-System bekannt gemacht.

5.1.3 Lokale Implementation

5.1.3.1 Funktionen

Funktionsname:

TERM_to_HEAP_STRING TERM_to_HEAP_STRING

Signatur:

char * TERM_to_HEAP_STRING(TERM String)

String: Nach C zu transferierender ASpecT String.

Rückgabewert: C char*.

Beschreibung:

Legt eine Kopie eines ASpecT Strings im C Format an.

Vor- und Nachbedingungen:

Auf keinen Fall vergessen den erhaltenen String nach Gebrauch mit `free()` wieder zu befreien!

Abhängigkeiten:

ASpecT-Bibliothek `String.AS`, ASpecT-Runtime-Bibliothek, `stdlib.h`

Implementierung:

Von Michael Fröhlich und Mattias Werner übernommen.

5.1.4 Restriktionen, Fehler

Die Funktionen `REALtoTERM(x)` und `TERMtoREAL(x)` arbeiten nur auf Basis des C-Typs `float`, da aufgrund von ASpecT nicht ermittelt werden kann, ob ASpecT nun für `float` oder `double` compiliert wurde!

5.2 TreasureLib

Das Modul TresureLib besteht aus den Files `TreasureLib.h`, `TreasureLib.c`.

5.2.1 Funktionalität

Dieses Modul enthält Hilfsroutinen, die in erster Linie für das Modul Parser benötigt wurden. Teilweise werden diese Routinen jedoch auch von anderen Modulen (UI und Output) benutzt.

5.2.2 Öffentliche Schnittstellen

5.2.2.1 Globale Variablen

Sortenname:

maxzeilen maxzeilen

Beschreibung:

Diese Variable wird vom Parser benötigt, um den Zeitverbrauchsrequester aktuell halten zu können. Sie darf deshalb von keinem anderen Modul verwendet werden!

5.2.2.2 Funktionen

Funktionsname:

xprozent xprozent

Signatur:

int xprozent(long int aktuellezeile, long int gesamtzeilen)

aktuellezeile: Aktuelle Zeile, für die ein Prozentwert berechnet werden soll.

gesamtzeilen: Gesamte Anzahl von Zeilen, die das File hat.

Rückgabewert: Prozentwert (0-100). Im Fehlerfall wird 0 zurückgegeben.

Beschreibung:

Diese Funktion berechnet einen Prozentwert, der angibt, wieviel von einem gerade laufenden Prozess schon abgearbeitet wurde.

Vor- und Nachbedingungen:

`0 <= aktuellezeile <= gesamtzeilen`

Implementierung:

Die Implementierung verwendet die umgeformte und allgemein bekannte mathematische Gleichung: $\frac{GW}{100} = \frac{PW}{PS}$

Funktionsname:

countlines countlines

Signatur:

long int countlines(char *filename)

filename: Name des PR-Files, für welches die Zeilen zu zählen sind.

Rückgabewert: Anzahl der Zeilen im entsprechenden File, oder `-1` wenn das File nicht geöffnet werden konnte.

Beschreibung:

Diese Routine zählt, wieviele Textzeilen im angegebenen File vorhanden sind.

Abhängigkeiten:

`stdio.h`

Implementierung:

Es wird zunächst versucht das File zu öffnen. Sollte dies fehlschlagen, wird `-1` als Fehlerwert zurückgegeben. Konnte das File hingegen geöffnet werden, wird bis zum Fileende jedes Zeichen gelesen, und auf ein $\backslash n$ hin überprüft. Ist dies der Fall, so wird ein Zähler erhöht, der nach dem Durchlesen des Files als Rückgabewert zurückgegeben wird.

Funktionsname:

shiftstrleft shiftstrleft

Signatur:

void shiftstrleft(char *str, unsigned long x)

str: String, welcher nach links zu shiften ist.

x: Anzahl der Zeichen, die geshiftet werden sollen.

Beschreibung:

Diese Funktion schiebt einen String um `x` Zeichen nach links, wobei die ersten `x` Zeichen verloren gehen.

Vor- und Nachbedingungen:

Der String mu mindestens `x` Zeichen enthalten und mit einem $\backslash x0$ abgeschlossen sein!

Implementierung:

Diese Funktion ist als eine kleine Schleife implementiert, welche den String durchläuft, und dabei die Zeichen nach vorn kopiert. Dies geschieht, bis ein $\backslash x0$ erkannt wurde.

Funktionsname:

appendtobuffer appendtobuffer

Signatur:

char * appendtobuffer(char *blk, char *blkbuffer)

blk: Speicherblock, welcher an `blkbuffer` angehängt werden soll.

blkbuffer: Speicherblock, an den `blk` angehängt werden soll.

Rückgabewert: Neuer Speicherblock, der aus `blkbuffer` und `blk` besteht, oder im Fehlerfalle nur `blkbuffer`.

Beschreibung:

Diese Funktion hängt den Speicherblock blk hinter den Speicherblock blkbuffer. Das bedeutet, daß beide Blöcke zusammengefügt werden.

Vor- und Nachbedingungen:

Das Argument `blk` muß ggf. nach dem Aufruf dieser Funktion selbst befreit werden.

Abhängigkeiten:

`stdlib.h`, `string.h`

Implementierung:

Zunächst einmal wird geprüft, ob blkbuffer ein `NULL` Pointer ist, und somit nur neuer Speicher angefordert und `blk` kopiert werden muß. Wird beim Anfordern von Speicher ein Fehler gemeldet, so wird dieser als Fehler vom Typ fatal mit der Nummer 17 an das ForauS-System gemeldet. Ist nun aber `blkbuffer` schon ein gültiger Speicherblock, so wird dieser um die Länge von `blk` vergrößert, und anschließend `blk` mittels `strcat` angehängt. Die Fehlerbehandlung erfolgt genau wie im oben schon erwähnten Fall.

Funktionsname:

sepfwd sepfwd

Signatur:

char * sepfwd(char *pathfilename, char *cmd)

pathfilename: Kompletter Filepfad mit Filename.

cmd: Kommando, das angibt, was aus dem Filenamen zu extrahieren ist.
„PATH“ – Den Pfadnamen ohne abschließendes '/' extrahieren.
„NAME“ – den Filenamen ohne Fileextension extrahieren.
„NAME.“ – den Filenamen mit Fileextension extrahieren.

Rückgabewert: Pointer auf einen neuen String.

Beschreibung:

Diese Funktion extrahiert aus einem String, der einen kompletten Filenamen mit Pfadangabe enthält entweder den Pfad, oder den Filenamen (mit oder ohne Fileextension).

Vor- und Nachbedingungen:

Der Rückgabestring muß nach Benutzung frei gegeben werden!

Abhängigkeiten:

`stdlib.h`, `string.h`

Implementierung:

Zuerst wird überprüft, ob überhaupt ein Pfadname vorhanden ist (Falls dies nicht der Fall ist, wird ein `NULL` Pointer zurückgegeben). Nun wird der Pfadname von hinten nach dem ersten auftretenden / durchsucht. Ist als Kommando „PATH“ angegeben, so wird nun Speicher bereit gestellt (tritt hierbei ein Fehler auf, wird das dem ForauS-System mit der Fehlernummer 17 gemeldet), und der Pfadname kopiert. Weiterhin wird der String korrekt mit einem $\backslash x0$ beendet. Wenn das Kommando aber „NAME“ ist, wird ebenfalls wieder Speicher bereitgestellt (Fehlerbehandlung wie oben). In diesen Speicher wird nun der Filename kopiert. Nun muß noch der neue String von hinten nach einem . durchsucht werden. Wird ein . gefunden, so wird an dessen Stelle der String mit einem $\backslash x0$ beendet. Als letztes Kommando könnte noch „NAME.grqq vorkommen, hier wird ebenfalls wieder Speicher bereitgestellt (Fehlerbehandlung siehe oben) und der Filename in diesen hinein kopiert.

Funktionsname:

connect_file connect_file

Signatur:

int connect_file(char *pathfilename, char *command)

pathfilename: Pfad für das Dokument, inklusive Filename und Fileextension.

command: Wird hier „SGMLS“ angegeben, so wird der SGMLS-Parser benutzt, um ein SGML-Dokument einzulesen. Wird statt dessen aber „DTDEL“ angegeben, so wird mit Hilfe des DTDEL-Parsers versucht eine DTD einzulesen.

Rückgabewert: Nummer des geöffneten Streams, oder `-1` im Fehlerfall.

Beschreibung:

Diese Funktion baut eine Pipe zum SGMLS bzw. DTDEL Parser auf, um so ein SGML-Dokument bzw. eine DTD einzulesen.

Vor- und Nachbedingungen:

Diese Funktion darf nur einmal zur gleichen Zeit arbeiten! Der geöffnete Stream muß auf jeden Fall mittels `close_file` wieder geschlossen werden.

Abhängigkeiten:

`stdio.h`, `stdlib.h`, `string.h`

Implementierung:

Zunächst wird das aktuelle Unix-Working-Directory geholt, sollte keines vorhanden sein wird der Fehler Nummer 20 an das Foraus-System gemeldet und `-1` als Rückgabewert geliefert. Jetzt wird in das Verzeichnis gewechselt, in dem das zu parsende SGML-Dokument bzw. die DTD liegt (dies ist notwendig, da der SGMLS/DTDEL nur mit Files arbeitet, die im aktuellen Verzeichnis liegen!). Sollte dieser Verzeichniswechsel nicht möglich sein, wird die Fehlernummer 19 an das FORAUS-System gemeldet. Jetzt wird Speicher für die Kommandozeile des SGMLS/DTDEL Aufrufes bereitgestellt (1024 Bytes). Sollte die Speicherreservierung fehlschlagen wird Fehlernummer 17 gemeldet. Nun wird entsprechend dem Kommando die Befehlszeile für den SGMLS bzw. DTDEL Aufruf aufgebaut. Zuerst wird hier von dem UI der Pfad, in dem der SGMLS/DTDEL zu finden ist, geholt. Hiernach wird nun der Befehl (sgmls oder dtdel) in den String eingefügt, danach noch der Name der DTD und ggf. der Name des SGML-Dokumentes. Weiterhin werden noch stderr und stdout zusammengelegt, damit beide gemeinsam über die Pipe kommen. Bleibt nun nur noch das erzeugte Kommando mittels `popen` zu starten und somit die Pipe zum Scanner zu erzeugen. Sollte hierbei ein Fehler auftreten, wird wieder `-1` als Rückgabewert geliefert, und Fehlernummer 18 an das System gemeldet. Konnte die Pipe aber aufgebaut werden, so wird noch die Filenummer des erzeugten Streams geholt und als Rückgabewert benutzt.

Funktionsname:

readfrom_file　　　　readfrom_file

Signatur:

int readfrom_file(int filenumber, char *filebuff, int size)

filenumber: Filenummer, die von `connect_file` erhalten wurde.

filebuff: Puffer in den Zeichen gelesen werden sollen.

size: Anzahl der Zeichen, die gelesen werden sollen.

Rückgabewert: Anzahl der tatsächlich gelesenen Zeichen.

Beschreibung:

Diese Funktion liest eine vorgegebene Anzahl von Zeichen in einen Puffer ein.

Vor- und Nachbedingungen:

Mit `connect_file` mußte vorher ein Stream zum SGMLS oder DTDEL geöffnet werden.

Abhängigkeiten:

`stdio.h`

Implementierung:

Es wird solange ein Zeichen aus dem globalen Stream gelesen, bis entweder die gewünschte Anzahl an Zeichen gelesen wurde, oder aber das Ende des Files erreicht ist. Anschliessend wird der Puffer mit einem $\backslash x0$ abgeschlossen und die tatsächlich gelesene Anzahl Zeichen zurückgegeben.

Funktionsname:

close_file　　　close_file

Signatur:

void close_file(int filenumber)

filenumber: Filenummer, die von `connect_file` erhalten wurde.

Beschreibung:

Diese Funktion schließt einen von `connect_file` geöffneten Stream.

Vor- und Nachbedingungen:

Mit connect_file mußte vorher ein Stream zum SGMLS oder DTDEL geöffnet werden.

Abhängigkeiten:

`stdio.h`, `stdlib.h`

Implementierung:

Der offene globale Stream wird mittels `pclose` geschlossen, und anschliessend wird versucht, den aktuellen Unix-Pfad wiederherzustellen (sollte dies nicht möglich sein, so wird dies an das FORAUS-System mit der Fehlernummer 22 gemeldet).

Funktionsname:

countdtdelements　　　countdtdelements

Signatur:

long int countdtdelements(char *filename)

filename: Name des DTD-Files, für welches die Element-Definitionen zu zählen sind.

Rückgabewert: Anzahl der Element-Definitionen im entsprechenden File, oder `-1` wenn das File nicht geöffnet werden konnte.

Beschreibung:

Diese Routine zählt, wieviele Element-Definitionen im angegebenen File vorhanden sind.

Abhängigkeiten:

`stdio.h`

Implementierung:

Es wird zunächst versucht das File zu öffnen. Sollte dies fehlschlagen, wird `-1` als Fehlerwert zurückgegeben. Konnte das File hingegen geöffnet werden, wird bis zum Fileende jedes Zeichen gelesen, und bei jedem Auftreten der Zeichensequenz „<!EL“ wird ein Zähler erhöht, der nach dem Durchlesen des Files als Rückgabewert benutzt wird.

Funktionsname:

countelements countelements

Signatur:

long int countelements(char *filename)

filename: Name des SGML-Files, für welches die Elemente zu zählen sind.

Rückgabewert: Anzahl der Elemente im entsprechenden File, oder `-1` wenn das File nicht geöffnet werden konnte.

Beschreibung:

Diese Routine zählt, wieviele Elemente im angegebenen File vorhanden sind.

Abhängigkeiten:

`stdio.h`

Implementierung:

Es wird zunächst versucht das File zu öffnen. Sollte dies fehlschlagen, wird `-1` als Fehlerwert zurückgegeben. Konnte das File hingegen geöffnet werden, wird bis zum Fileende jedes Zeichen gelesen, und bei jedem Auftreten der Zeichensequenz „<„ wird ein Zähler erhöht, der nach dem Durchlesen des Files als Rückgabewert benutzt wird.

5.2.3 Lokale Implementation

5.2.3.1 Globale Variablen

Sortenname:

streamhandle streamhandle

Beschreibung:

Pointer auf den aktuellen Stream (siehe hierzu `connect_file`, `readfrom_file`, `close_file`).

Sortenname:

currentpath currentpath

Beschreibung:

Pointer auf den aktuellen Unix-Pfad (siehe hierzu `connect_file`, `close_file`).

5.2.3.2 Funktionen

Um das Debuggen dieser Routinen zu vereinfachen, wurden einige `printf` Ausgaben eingebaut, die mit Hilfe eines Define des Preprozessorsymbols `LDEBUG` eingeschaltet werden können.

5.3 Integration ins FORAUS System

5.3.1 Makefile

Das Makefile wird benötigt, um die Hilfsbibliotheken „AviseC"und „TreasuresLib" zu übersetzen, sowie mit Hilfe der Toolbox die Scanner und Parser zu generieren und zu übersetzen. In der Compilerzeile (.c.o) können ggf. auch die folgenden Optionen angegeben werden:
-`DNOTIMEREQ` - Diese muß dann aber auch im Foraus.cmd File gesetzt werden!
-`DSDEBUG` - Um das Scanner debugging einzuschalten
-`DPDEBUG` - Um das Parser debugging einzuschalten
-`DLDEBUG` - Um das Library debugging einzuschalten

5.3.2 Foraus.cmd

Das Foraus.cmd File enthält Informationen für das ASpecT Kommando „gen", welches dafür sorgt, daß das gesamte ForauS-System korrekt übersetzt wird. Abweichungen vom Standard „default.cmd" File wurden vorgenommen, um die Hilfsbibliotheken „AviseC" und „TreasuresLib" sowie die generierten Scanner und Parser hinzu zu linken. Weiterhin werden auch die benötigten X11 Bibliotheken gelinkt. Unter „Compiling" kann ggf. die Option -`DNOTIMEREQ` mit angegeben werden, diese muß dann aber auch im Makefile eingefügt werden!

5.3.3 mak

Das Shell Macro „mak" enthält jeweils 2 mal hintereinander einen Aufruf für das „make" und das „gen". Dieses ist notwendig, da die Abhängigkeiten zwischen den zu generierenden Parsern und ASpecT-Programmen so groß sind, daß bei einem Makelauf, welcher das System von grundauf neu schaffen soll, nicht alles korrekt erzeugt und gelinkt werden kann.

6. Parser

Kai Hofmann

6.1 Parser

Das Parser Modul besteht aus folgenden Files:
`Parser.AS`, `Parser.xh`, `Parser.xc`, `SGML.rex`, `DTD.rex`, `DTD.lalr`, `PR.rex`, `PR.lalr`, `SGML-ScannerSourceReplace.c`, `DTDScannerSourceReplace.c`

6.1.1 Funktionalität

Das Parser Modul hat die Aufgabe extern in Dateien vorliegende Daten einzulesen und an die IS weiterzuleiten. Hierzu gehört das Einlesen einer DTD, einer PR und natürlich das Einlesen von SGML-Dokumenten.

6.1.2 Entwurf

Das Parser Modul hat die Aufgabe externe Daten an die IS zu leiten. Hierbei sind zunächst als Grundfunktionalitäten das Einlesen einer DTD, eines SGML-Dokumentes und einer PR vorgesehen. Später wäre es evtl. auch noch wünschenswert die gespeicherten default Werte des GUI's wieder einzulesen, und falls möglich auch diverse Fremdformate von anderen Textsystemen (z.B. LaTeX), wobei allerdings eine Textstruktur erzeugt werden muß. Die Schnittstellen für alle Funktionen sind so zu gestalten, daß sie für das System transparent und möglichst flexibel sind.

6.1.2.1 DTD Schnittstelle

Gefordert wird hier, daß es möglich ist eine komplette DTD einzulesen. Das Gewicht liegt hierbei jedoch zunächst nur auf den Element-Definitionen und den dazugehörigen Attribut-Deklarationen. Weitere SGML-Besonderheiten wie Entities können in späteren Versionen nachgetragen werden.

6.1.2.2 PR Schnittstelle

Mit Hilfe dieser Schnittstelle soll eine PR eingelesen werden können, wie sie von der PR-Arbeitsgruppe erarbeitet wurde.

6.1.2.3 SGML-Dokument Schnittstelle

Es soll hier möglich sein ein komplettes SGML-Dokument einzulesen, auf DTD-Konsistenz zu prüfen und an die IS weiterzuleiten. Hierbei ist zunächst wieder besonderes Augenmerk auf die Textelemente selbst (evtl. rekursiv geschachtelt!) und deren Attribute zu legen.

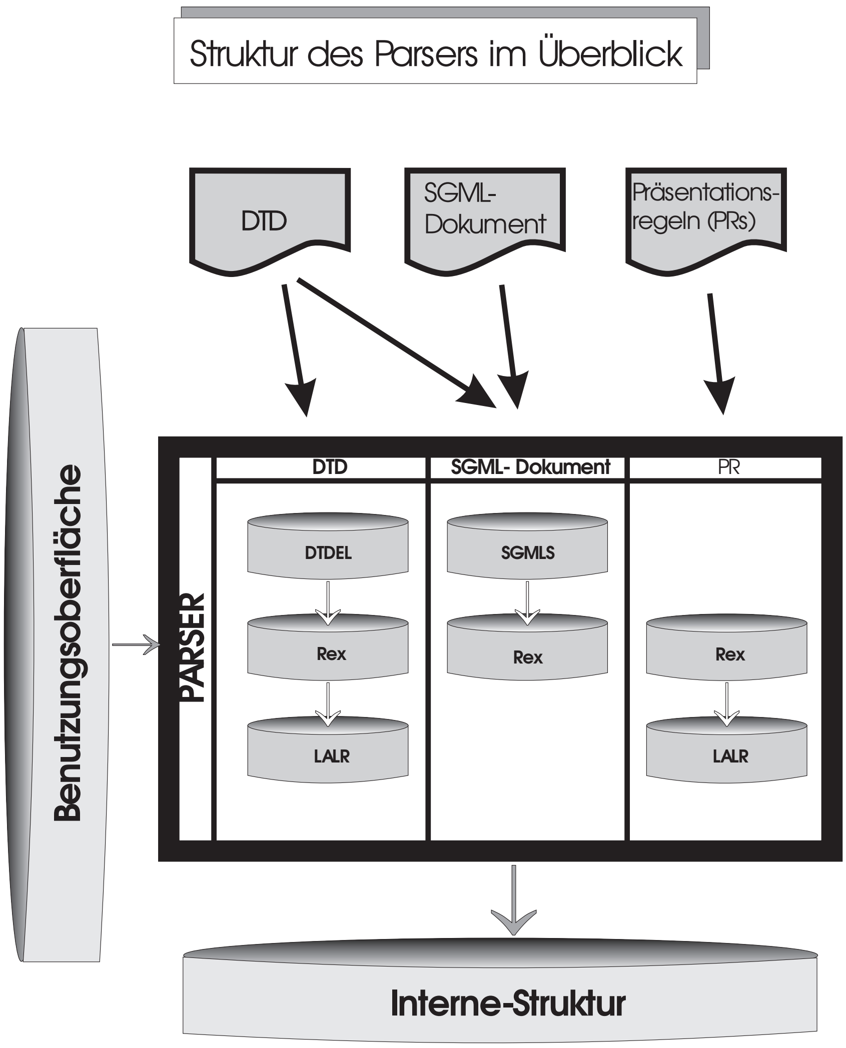

Abbildung 6.1: Übersicht Parser

6.1.3 Öffentliche Schnittstellen

6.1.3.1 Funktionen

Funktionsname:

p_init p_init

Signatur:

p_init:: env -> env.

env: Aktuelles globales ForauS-Environment.

Beschreibung:

`p_init` dient zur Initialisierung der vom Parser benötigten Strukturen und wird beim Programmstart durch das UI aufgerufen.

Implementierung:

Unbenutzt, dh. liefert nur das aktuelle Environment zurück.

Funktionsname:

p_exit p_exit

Signatur:

p_exit:: env -> env

env: Aktuelles globales $F^{or}a^{u}S$-Environment.

Beschreibung:

`p_exit` dient zur Löschung der vom Parser aufgebauten Strukturen und wird beim Programmende durch das UI aufgerufen.

Implementierung:

Unbenutzt, dh. liefert nur das aktuelle Environment zurück.

Funktionsname:

p_loadDTD p_loadDTD

Signatur:

p_loadDTD:: env -> string -> env.

env: Aktuelles globales $F^{or}a^{u}S$-Environment.

string: Pfadname der zu parsenden DTD.

Beschreibung:

Die Funktion `p_loadDTD` startet mit dem Parameter `DTD_PATH_FILENAME` den Freeware SGMLS-Parser. Dieser kontrolliert die bezeichnete DTD auf Vollständigkeit und Korrektheit. War die Überprüfung erfolgreich, so wird die DTD an die IS mittels dortiger Routinen durchgereicht. Die Namenskonvention des SGMLS-Parsers sind auch in dieser Schnittstelle einzuhalten. Sollten Fehler beim Parsvorgang auftreten, so wird eine entsprechende Fehlernummer in dem dafür vorgesehenen Bereich der Systemumgebung abgelegt. Weiterhin wird veranlasst, daß die bis zu diesem Zeitpunkt aufgebauten Strukturen der IS wieder abgebaut werden.

Abhängigkeiten:

ASpecT-Runtime-Bibliothek, `IS.AS`, `ISTypes.AS`, `UI.AS`, `AviseC.h`, `TreasureLib.h`, `DTDScanner.h`, `DTDParser.h`

Implementierung:

Zunächst einmal werden in dem File, welches einzulesen ist mit Hilfe der Funktion `countdtdelements` die Element- Definitionen durchgezählt, um später den Zeitverbrauchsrequester aktuell halten zu können. Sollte hierbei ein Fehler auftreten, dh. das DTD File konnte nicht geöffnet werden, so wird Fehlernummer 16 an das ForauS-System gemeldet. Sollte jedoch alles in Ordnung sein, so wird nun der Zeitverbrauchsrequester geöffnet (sofern dieser nicht mittels der Compileroption `NOTIMEREQ` ausgeblendet wurde). Jetzt wird dem generierten Scanner mitgeteilt (`BeginFile`), daß er seine Eingabe aus einem File lesen soll. Weiterhin wird eine evtl. in der IS vorhandene DTD nun gelöscht, und angemeldet, daß eine neue DTD übergeben wird. Nach dem Start des generierten Parsers wird der IS bekanntgegeben, daß die DTD komplett eingelesen wurde, weiterhin wird der Parser geschlossen und ggf. auch der Zeitverbrauchsrequester.

Funktionsname:

p_loadPR p_loadPR

Signatur:

p_loadPR:: env -> string -> env.

env: Aktuelles globales ForauS-Environment.

string: Pfadname der zu parsenden PR.

Beschreibung:

Die Funktion `p_LoadPR` leitet den Parsvorgang der durch den Parameter `PR_PATH_FILENAME` bezeichneten Präsentionsregeln ein. Sobald ein Objekt der PR erkannt wird, wird dies durch die Schnittstelle zur IS durchgereicht. Sollten Fehler beim Parsvorgang auftreten, so wird eine entsprechende Fehlernummer in den dafür vorgesehenen Bereich der Systemumgebung abgelegt. Weiterhin wird veranlasst, daß die bis zu diesem Zeitpunkt aufgebauten Strukturen der IS wieder abgebaut werden.

Abhängigkeiten:

ASpecT-Runtime-Bibliothek, `IS.AS`, `ISTypes.AS`, `UI.AS`, `AviseC.h`, `TreasureLib.h`, `PRScanner.h`, `PRParser.h`

Implementierung:

Zunächst einmal werden in dem File, welches einzulesen ist mit Hilfe der Funktion `countlines` die Zeilen durchgezählt, um später den Zeitverbrauchsrequester aktuell halten zu können. Sollte hierbei ein Fehler auftreten, dh. das PR File konnte nicht geöffnet werden, so wird Fehlernummer 16 an das ForauS-System gemeldet. Sollte jedoch alles in Ordnung sein, so wird nun der Zeitverbrauchsrequester geöffnet (sofern dieser nicht mittels der Compileroption `NOTIMEREQ` ausgeblendet wurde). Jetzt wird dem generierten Scanner mitgeteilt (`BeginFile`), daß er seine Eingabe aus einem File lesen soll. Nach dem Durchlauf des generierten Parsers wird dieser noch wieder geschlossen und ggf. auch der Zeitverbrauchsrequester.

Funktionsname:

p_loadSgmlDoc p_loadSgmlDoc

Signatur:

p_loadSgmlDoc:: env -> string -> env.

env: Aktuelles globales $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$-Environment.

string: Pfadname des zu parsenden SGML-Dokumentes.

Beschreibung:

Die Funktion `p_loadSgmlDoc` startet mit dem Parameter `SGML_PATH_FILENAME` den Freeware SGMLS-Parser. Dieser kontrolliert die bezeichnete DTD auf Vollständigkeit und Korrektheit, sowie das dazu gehörige SGML-Dokument. War die Überprüfung des Dokumentes (DTD & SGML-Text) erfolgreich, so wird der ausgezeichnete SGML-Text an die IS mittels dortiger Routinen durchgereicht. Die Namenskonvention des SGMLS-Parser sind auch in dieser Schnittstelle einzuhalten. Sollten Fehler beim Parsvorgang auftreten, so wird eine entsprechende Fehlernummer in den dafür vorgesehenen Bereich der Systemumgebung abgelegt. Weiterhin wird veranlasst, daß die bis zu diesem Zeitpunkt aufgebauten Strukturen der IS wieder abgebaut werden.

Abhängigkeiten:

`AviseC.h`, `TreasureLib.h`, `IS.AS`, `ISTypes.AS`, `UI.AS`, `SGMLScanner.h`, ASpecT-Runtime-Bibliothek

Implementierung:

Zunächst einmal werden in dem File, welches einzulesen ist mit Hilfe der Funktion `countsgmlelements` die SGML-Elemente durchgezählt, um später den Zeitverbrauchsrequester aktuell halten zu können. Sollte hierbei ein Fehler auftreten, dh. das SGML File konnte nicht geöffnet werden, so wird Fehlernummer 16 an das $\mathrm{F}^{\mathrm{or}}\mathrm{a}^{\mathrm{u}}\mathrm{S}$-System gemeldet. Sollte jedoch alles in Ordnung sein, so wird nun der Zeitverbrauchsrequester geöffnet (sofern dieser nicht mittels der Compileroption `NOTIMEREQ` ausgeblendet wurde). Jetzt wird dem generierten Scanner mitgeteilt (`BeginFile`), daß er seine Eingabe aus einem File lesen soll. Weiterhin wird ein evtl. in der IS vorhandenes Dokument nun gelöscht, und angemeldet, daß ein neues Dokument übergeben wird. Nach dem Start des generierten Scanners wird der IS bekanntgegeben, daß das Dokument komplett eingelesen wurde, weitehrin wird der Scanner geschlossen und ggf. auch der Zeitverbrauchsrequester.

6.2 SGML Scanner

6.2.1 SGMLScannerSourceReplace.c

Dieses File ist ein Ersatz für das von der Toolbox zur Verfügung gestellte File `SGMLScannerSource.c`. Die Schnittstelle des Files ist vollkommen identisch zu der des Toolbox Files, deshalb kann dieses beim Linken ganz einfach das Originalfile ersetzen! Zweck ist es normalerweise ein File zu öffnen, und aus diesem die Eingabedaten für den Scanner zu lesen. Die entsprechenden Routinen wurden mit Hilfe der „TreasuresLib“ so verändert, daß der SGMLS gestartet wird, und über eine Pipe die Daten an den Scanner weitergereicht werden.

6.2.2 SGML.rex

Zum Verständnis dieses Scanners werden grundlegende Kenntnisse über die Karlsruher Toolbox und den Umgang mit dieser vorausgesetzt!
Erstes Problem beim Einbinden von Toolbox generiertem Code war es, daß Includes von ASpecT benötigt wurden, um z.B. die Schnittstellen zur IS ansprechen zu können. Hierbei wurde festgestellt, daß `true` und `false` sowohl von der Toolbox als auch von ASpecT verwendet werden! Um dieses Problem zu handhaben, müssen vor den entsprechenden Includes undef's ausgeführt werden, und nach ihnen wieder die für die Toolbox benötigten defines durchgeführt werden.
Die `DEFAULT` Definition wurde so verändert, daß der Fehler direkt an das ForauS-System gemeldet wird.
Die Grammatik aus der der Scanner selbst generiert wird, wurde so gestaltet, daß sie hoffentlich auf sämtliche Ausgaben des SGMLS korrekt eingeht. Hierbei tritt jedoch ein kleines Problem auf: Da `stderr` und `stdout` zusammen auf eine Pipe gelegt wurden, und der SGMLS scheinbar nicht darauf achtet, daß Fehlermeldungen nur am Anfang einer neuen Zeile erscheinen, sondern auch mitten im Text auftauchen können, kann dies zu Problemen führen, für die in der vorhandenen Zeit noch keine Lösung gefunden werden konnte! Der auftretende Fehler selbst wird noch korrekt erkannt und an das ForauS-System gemeldet. Allerdings gibt es bei den nachfolgenden Eingaben, welche vom SGMLS kommen dann zunächst kleine Probleme, bis der Scanner zurück in die Grammatik gefunden hat!
Bis jetzt werden folgende SGMLS Eigenschaften unterstützt:
Attribute, Elementstart, Elementende und Elementinhalte inklusive oktaler Zahlen, die z.B. beim Gebrauch von Umlauten auftreten. Alle weiteren SGMLS Features sind bereits angedacht, werden aber bisher übergangen, da sie vom ForauS-System nicht unterstützt werden.
Auf den Toolbox-Code soll hier nicht weiter eingegangen werden, da er leicht verständlich (bei Kenntnis der Toolbox und C) ist, und auch viel mit Hilfe der „TreasureLib" arbeitet.

6.2.3 SGMLS

Christian Schaefer

Der Freeware SGML-Parser namens SGMLS, welcher von James Clark weiterentwickelt wird, stammt ursprünglich vom ARCSGML-Parser ab, welcher von Charles Goldfarb entwickelt wurde. Der SGMLS-Parser ist direkt von James Clark per FTP über die Internetadresse ftp.jclark.com beziehbar, und scheinbar Weltweit der einzige SGML-Parser, der den SGML-Standard fast komplett in sich vereint.
Zuerst erhielten wir die Version 1.0 von Dr. Ing. Carsten Borman, mit welcher wir erste Erfahrungen sammelten. Nach eigenen Nachforschungen fanden wir dann die etwas aktuellere 1.1 Version, mit der wir die meiste Zeit arbeiteten. Mit dieser Version wurde auch der später gebrauchte DTDEL erzeugt. Gegen ende des Projektes erhielten wir von James Clark noch die Version 1.1.92, welche eine Beta-Version der zukünftigen 1.2 Version ist. Mit dieser Version ließ sich allerdings leider nicht mehr der DTDEL erzeugen.

6.3 DTD Parser

6.3.1 DTDScannerSourceReplace.c

Dieses File ist ein Ersatz für das von der Toolbox zur Verfügung gestellte File `DTDScanner-Source.c`. Die Schnittstelle des Files ist vollkommen identisch zu der des Toolbox Files, deshalb kann dieses beim Linken ganz einfach das Originalfile ersetzen! Zweck ist es normalerweise ein File zu öffnen, und aus diesem die Eingabedaten für den Scanner zu lesen. Die entsprechenden Routinen wurden mit Hilfe der „TreasuresLib“ so verändert, daß der DTDEL gestartet wird, und über eine Pipe die Daten an den Scanner weitergereicht werden.

6.3.2 DTD.rex

Zum Verständnis dieses Scanners werden grundlegende Kenntnisse über die Karlsruher Toolbox und den Umgang mit dieser vorausgesetzt!
Erstes Problem beim Einbinden von Toolbox generiertem Code war es, daß Includes von ASpecT benötigt wurden, um z.B. die Schnittstellen zur IS ansprechen zu können. Hierbei wurde festgestellt, daß `true` und `false` sowohl von der Toolbox als auch von ASpecT verwendet werden! Um dieses Problem zu handhaben, müssen vor den entsprechenden Includes undef's ausgeführt werden, und nach ihnen wieder die für die Toolbox benötigten defines durchgeführt werden.
Die `DEFAULT` Definition wurde so verändert, daß der Fehler direkt an das F^{or}a^{u}S-System gemeldet wird.
Die Grammatik aus der der Scanner selbst generiert wird, wurde so gestaltet, daß sie hoffentlich auf sämtliche Ausgaben des DTDEL korrekt eingeht. Hierbei tritt jedoch ein kleines Problem auf: Da `stderr` und `stdout` zusammen auf eine Pipe gelegt wurden, und der DTDEL scheinbar nicht darauf achtet, daß Fehlermeldungen nur am Anfang einer neuen Zeile erscheinen, sondern auch mitten im Text auftauchen können, kann dies zu Problemen führen, für die in der vorhandenen Zeit noch keine Lösung gefunden werden konnte! Der auftretende Fehler selbst wird noch korrekt erkannt und an das F^{or}a^{u}S-System gemeldet. Allerdings gibt es bei den nachfolgenden Eingaben, welche vom DTDEL kommen dann zunächst kleine Probleme, bis der Scanner zurück in die Grammatik gefunden hat!
Auf den Toolbox-Code soll hier nicht weiter eingegangen werden, da er leicht verständlich (bei Kenntnis der Toolbox und C) ist, und auch viel mit Hilfe der „TreasureLib“ arbeitet. Der grösste Teil der erkannten Tokens wird sowieso an den Parser weitergeleitet, der ja auch die Hauptarbeit leistet.

6.3.3 DTD.lalr

Zum Verständnis dieses Parsers werden grundlegende Kenntnisse über die Karlsruher Toolbox und den Umgang mit dieser vorausgesetzt!
Erstes Problem beim Einbinden von Toolbox generiertem Code war es, daß Includes von ASpecT benötigt wurden, um z.B. die Schnittstellen zur IS ansprechen zu können. Hierbei wurde festgestellt, daß `true` und `false` sowohl von der Toolbox als auch von ASpecT verwendet werden! Um dieses Problem zu handhaben, müssen vor den entsprechenden Includes undef's ausgeführt werden, und nach ihnen wieder die für die Toolbox benötigten defines durchgeführt werden.
Die verwendete Grammatik entspricht dem Ausgabeformat des DTDEL. Hierbei ist folgendes zu beachten: Da die Grammatik wie ein Baum organisiert ist, ist es wichtig, bei Aufrufen an die IS darauf zu achten, daß zunächst einmal der C Code der tieferliegenden Äste abgearbeitet wird, und erst nach Abarbeitung sämtlicher zu einem Knoten gehörender Äste, der Code dieses Knotens abgearbeitet wird! Dies bedeutet auch, daß die Grammatik an einigen Stellen etwas glqqzerflückt“ aussieht, da es hier notwendig war, die Grammatik aufzuteilen, um Code auch an der entsprechend richtigen Stelle ausführen zu lassen! Andere Knoten hingegen sind auch leer, da diese für Code nicht benötigt werden.

6.3.4 DTDEL

Christian Schaefer

DTDEL ist ein von James Clark zu Testzwecken entwickeltes Tool, welches es ermöglicht, die syntaktische Beschreibung (DTD) eines SGML-Dokumentes auszugeben. Es basiert zu 99% auf dem von ihm weiterentwickelten SGML-Parser (SGMLS) und besteht aus zwei in C geschriebenen Modulen. Das Hauptmodul bildet das File: `dtdel.c`, dieses stellt eine Kommandozeilenoberfläche, das Filehandling und den Aufbau der Datenstrukturen sowie die Kommunikation mit den benötigten SGMLS-Modulen zur Verfügung. Das zweite Modul wird durch das File `dump.c` gebildet. In ihm sind die Routinen zur Traversierung der Datenstrukturen und der Ausgabe von DTD-Elementen enthalten. Das Ausgabeformat von DTDEL entspricht einer GNU-Emacs-Lisp-Repräsentation der Document-Type-Definition. Da dieses Ausgabeformat für unsere Zwecke nicht brauchbar war und wir nur einen kleinen Teil der Informationen aus der DTD für unser $\text{F}^{\text{or}}\text{a}^{\text{u}}\text{S}$-System nutzen können, beschlossen wir das Ausgabeformat des DTDEL für unsere Zwecke zu modifizieren. Unsere Änderungen beziehen sich nur auf das Modul `dump.c`. In ihm sind die Aufrufe der Routine `SGMLDUMP` zur Ausgabe von Attributen, Shortreferenzen und Entitäten auskommentiert worden. Weiterhin ist die Routine `ELEMDUMP` dahingehend modifiziert worden, daß sie nur die Operatoren und die Datentypen zu einer Elementdeklaration ausgibt. In diesem Zusammenhang steht auch die Abänderung der Routine `TOKDUMP`, so daß die Operatoren (Sequenz, Optional, Ein- und Mehrfach, Und, Oder) bezüglich einer Elementdeklaration, in der für SGML genormten Schreibweise ausgegeben werden.

6.4 PR Parser

6.4.1 PR.rex

Zum Verständnis dieses Scanners werden grundlegende Kenntnisse über die Karlsruher Toolbox und den Umgang mit dieser vorausgesetzt!
Erstes Problem beim Einbinden von Toolbox generiertem Code war es, daß Includes von ASpecT benötigt wurden, um z.B. die Schnittstellen zur IS ansprechen zu können. Hierbei wurde festgestellt, daß `true` und `false` sowohl von der Toolbox als auch von ASpecT verwendet werden! Um dieses Problem zu handhaben, müssen vor den entsprechenden Includes undef's ausgeführt werden, und nach ihnen wieder die für die Toolbox benötigten defines durchgeführt werden.
Die `DEFAULT` Definition wurde so verändert, daß der Fehler direkt an das $\text{F}^{\text{or}}\text{a}^{\text{u}}\text{S}$-System gemeldet wird.
Die Grammatik selbst entspricht den Präsentationsregeln in der Fassung vom 28.04.1994 (siehe entsprechendes Dokument). Die entsprechenden Token werden auch hier wieder an den Parser weitergereicht, so daß hier nicht weiter auf den Code eingegangen werden muß, da dieser trivial ist.

6.4.2 PR.lalr

Zum Verständnis dieses Parsers werden grundlegende Kenntnisse über die Karlsruher Toolbox und den Umgang mit dieser vorausgesetzt!
Erstes Problem beim Einbinden von Toolbox generiertem Code war es, daß Includes von ASpecT benötigt wurden, um z.B. die Schnittstellen zur IS ansprechen zu können. Hierbei wurde festgestellt, daß `true` und `false` sowohl von der Toolbox als auch von ASpecT verwendet werden! Um dieses Problem zu handhaben, müssen vor den entsprechenden Includes undef's ausgeführt

werden, und nach ihnen wieder die für die Toolbox benötigten defines durchgeführt werden.
Die verwendete Grammatik entspricht den Präsentationsregeln in der Fassung vom 28.04.1994 (siehe entsprechendes Dokument). Hierbei ist folgendes zu beachten: Da die Grammatik wie ein Baum organisiert ist, ist es wichtig bei Aufrufen an die IS darauf zu achten, daß zunächst einmal der C Code der tieferliegenden Äste abgearbeitet wird, und erst nach Abarbeitung sämtlicher zu einem Knoten gehörender Äste der Code dieses Knotens abgearbeitet wird! Dies bedeutet auch, daß die Grammatik an einigen Stellen etwas „zerflückt" aussieht, da es hier notwendig war, die Grammatik aufzuteilen, um Code auch an der entsprechend richtigen Stelle ausführen zu lassen! Andere Knoten hingegen sind auch leer, da diese für Code nicht benötigt werden. An zwei Stellen ist der Code dennoch etwas komplizierter (`ValueKeyWord` und `LogObj`), da hier mit Hilfe der Toolbox nicht entschieden werden konnte, in welchem Zustand sich der Parser genau befindet. Um dies dennoch zu ermöglichen mussten zwei Variablen (`VarLogFlag` und `AttrValFlag`) zu Hilfe genommen werden, um den Zustand eindeutig zu definieren. Der gesamte Code mag zunächst sehr verwirrend aussehen, vorallem, da dieser ja der Baumart der Grammatik angepasst ist, die Hilfsausgaben des Debugmodes können hier aber doch schnell sehr hilfreich sein, wenn es um das Verständnis des vom Parser erkannten Codes geht.

7. Generator

Kai Hofmann

7.1 Generator

Das Modul Generator besteht aus den Files `Generator.AS`, `Generator.xh` und `Generator.xc`!

7.1.1 Funktionalität

Das Modul `Generator.AS` stellt Funktionen zum Speichern der internen Datenstrukturen zur Verfügung. Bisher wurden Funktionen zum Speichern der DTD und des SGML-Dokumentes realisiert.

7.1.2 Entwurf

Dieses Modul soll es ermöglichen, die internen Datenstrukturen des F$^{\text{or}}$a$^{\text{u}}$S-Systems zu speichern. Hierzu erscheint es am besten, wenn fr jede mögliche Struktur eine Schnittstelle zum Speichern bereitgestellt wird. Diese Schnittstellen sind so zu wählen, daß sie für das Gesamtsystem vollkommen transparent sind und der unterliegende Code leicht austauschbar bzw. erweiterbar ist. Im jetzigen Stadium werden Schnittstellen zum Speichern der DTD und des SGML-Dokumentes benötigt. In späteren Stadien wären auch noch Schnittstellen zum Speichern der Präsentations-Regeln und der default Werte des GUI's denkbar. Weiterhin wären noch Schnittstellen zum Speichern des Dokumentes als ASCII und falls möglich auch als TeX Files wünschenswert. Denkbar sind jedoch auch alle anderen Fileformate von gängigen Textverarbeitungssystemen sowie PostScript.

7.1.2.1 SGML-Dokument Schnittstelle

Die Schnittstelle zum Speichern des SGML-Dokumentes hat als Aufgabe dafür zu sorgen, daß das intern vorliegende Dokument in der SGML-Form gespeichert wird. Hierbei sind sowohl die einzelnen Elemente (ggf. auch geschachtelt!) sowie deren Attribute zu speichern.

7.1.2.2 DTD Schnittstelle

Mit der DTD Schnittstelle wird dafür gesorgt, daß einerseits sehr komplizierte fremde DTD's in einfacherer Form (durch den SGMLS aufgelöst!) gespeichert werden, andererseits wird hier aber auch schon die Möglichkeit des Speicherns einer in der Zukunft interaktiv erstellbaren DTD berücksichtigt. Es ist hier also erforderlich die einzelnen Element-Definitionen, aber auch deren Attribut-Deklarationen zu speichern.

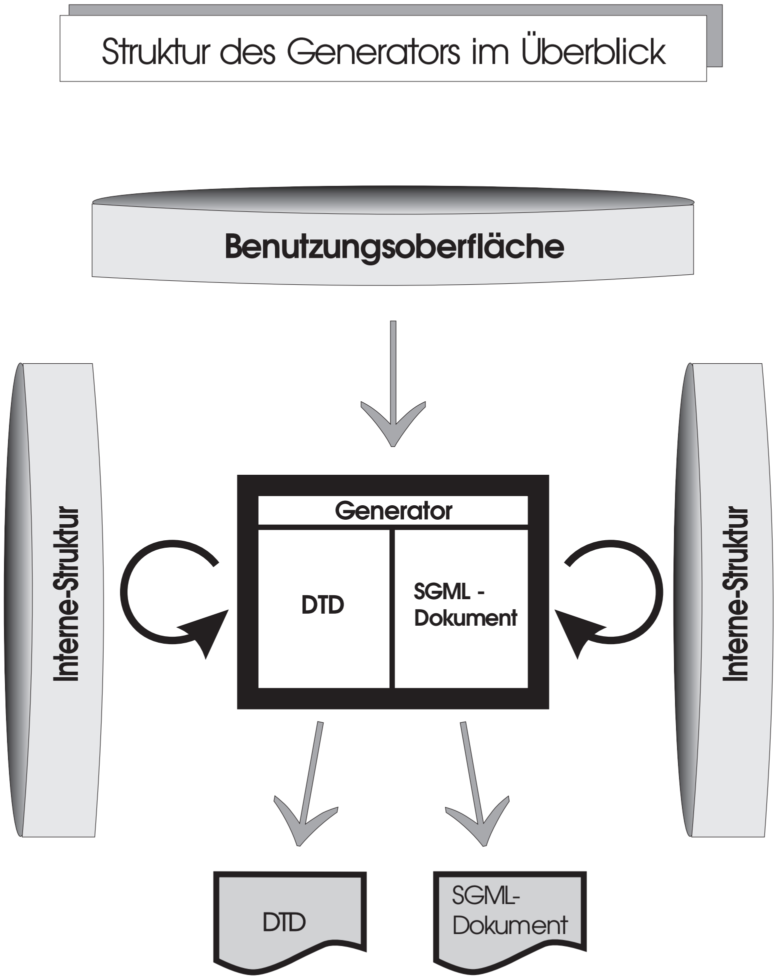

Abbildung 7.1: Übersicht Generator

7.1.3 Öffentliche Schnittstellen

7.1.3.1 Funktionen

Funktionsname:

g_init g_init

Signatur:

g_init:: env -> env.

env: Aktuelles globales ForauS-Environment.

Beschreibung:

`g_init` dient zur Initialisierung der vom Generator benötigten Strukturen und wird beim Programmstart durch das UI aufgerufen.

Implementierung:

Unbenutzt, dh. liefert nur das aktuelle Environment zurück.

Funktionsname:

g_exit g_exit

Signatur:

g_exit:: env -> env

env: Aktuelles globales ForauS-Environment.

Beschreibung:

`g_exit` dient zur Löschung der vom Generator aufgebauten Strukturen und wird beim Programmende durch das UI aufgerufen.

Implementierung:

Unbenutzt, dh. liefert nur das aktuelle Environment zurück.

Funktionsname:

g_saveDTD g_saveDTD

Signatur:

g_saveDTD:: env -> string -> env.

env: Aktuelles globales ForauS-Environment.

string: Pfadname der zu speichernden DTD.

Beschreibung:

Die Funktion `p_saveDTD` speichert die von der IS gehaltene DTD als Textfile.

Abhängigkeiten:

`UI.AS`, `ISGenerator.AS`, `IS.AS`, `ISTypes.AS`, `AviseC.h`, `stdio.h`, `stdlib.h`, `string.h`, `SaveDTDElements()`, ASpecT-Runtime-Bibliothek

Implementierung:

Zuerst wird der IS mitgeteilt, daß eine DTD gespeichert werden soll. Weiterhin wird nun ggf. ein Zeitverbrauchsrequester geöffnet, sowie daß File, in welches die DTD zu speichern ist. Sollte beim Öffnen dieses Files ein Fehler auftreten, so wird dem ForauS-System die Fehlernummer 1 übergeben. Die DTD Definition wird nun eingeleitet, wobei als DOCTYPE der Filename ohne Pfad und Fileextension verwendet wird. Nun werden noch 2 Variablen für den Zeitverbrauchsrequester initialisiert, bevor damit begonnen wird von der IS die erste DTD-Element-Definition abzufragen. Das Abfragen der Elemente von der IS erfolgt innerhalb einer Schleife, die solange läuft, bis die IS meldet, daß keine weiteren Definitionen mehr folgen. Innerhalb dieser Schleife wird nun zunächst der Zeitverbrauchsrequester aktuallisiert, und danach die Definition eines DTD-Elementes, wobei dessen Abhängigkeiten mit Hilfe der Funktion `SaveDTDElements` gespeichert werden.
Nachdem nun alle Daten gespeichert sind, wird das File geschlossen, sowie ggf. auch der Zeitverbrauchsrequester. Weiterhin wird der IS noch mitgeteilt, daß das Speichern beendet ist.

Funktionsname:

g_saveSgmlDoc g_saveSgmlDoc

Signatur:

g_saveSgmlDoc:: env -> string -> env.

env: Aktuelles globales ForauS-Environment.

string: Pfadname des zu speichernden SGML-Dokumentes.

Beschreibung:

Die Funktion `p_saveSgmlDoc` speichert das von der IS gehaltene Dokument als SGML Textfile.

Abhängigkeiten:

ASpecT-Runtime-Bibliothek, `UI.AS`, `ISGenerator.AS`, `AviseC.h`, `stdio.h`, `SaveSGMLElement()`

Implementierung:

Zuerst wird der IS mitgeteilt, daß ein SGML-Dokument gespeichert werden soll. Weiterhin wird nun ggf. ein Zeitverbrauchsrequester geöffnet, sowie daß File, in welches die internen Daten zu speichern sind. Sollte beim Öffnen dieses Files ein Fehler auftreten, so wird dem ForauS-System die Fehlernummer 2 übergeben. Nun wird die Funktion `SaveSGMLElement` aufgerufen, wobei als Parameter die Anzahl vorhandener SGML-Elemente von der IS übergeben wird. Nachdem nun alle Daten gespeichert sind, wird das File geschlossen, sowie ggf. auch der Zeitverbrauchsrequester. Weiterhin wird der IS noch mitgeteilt, daß das Speichern beendet ist.

7.1.4 Lokale Implementation

7.1.4.1 Globale Variablen

Sortenname:

file file

Beschreibung:

Globaler Pointer, der auf das File zeigt, welches gerade gespeichert wird.

Funktionsname:

SaveSGMLElement SaveSGMLElement

Signatur:

void SaveSGMLElement(unsigned long maxelem, unsigned long aktelem)

maxelem: Anzahl in der IS vorhandener Elemente

aktelem: Aktuelles Element, welches gerade gespeichert wird.

Beschreibung:

Speichert die SGML-Text-Elemente

Vor- und Nachbedingungen:

Wird von `g_saveSgmlDoc` Aufgerufen.

Abhängigkeiten:

`ISGenerator.AS`, `UI.AS`, `AviseC.h`, `stdio.h`

Implementierung:

Aufgabe dieser rekursiven Routine ist es, die SGML-Elemente zu speichern. Hierzu wird zunächst der Zeitverbrauchsrequester aktuallisiert, und der Begin eines neuen Elementes geschrieben. Nun folgt die bereits angedachte, aber noch nicht funktionsfähige Schleife zum Speichern der Attribute dieses Elementes. Es folgt die Abfrage des Textes, welcher dem Element zugeordnet ist. Ist dieser vorhanden, wird er gespeichert. Ist aber kein Text vorhanden, handelt es sich um ein Nicht-Finales-Element, und somit um rekursiv geschachtelte Unterelemente. Diese werden nun durch einen rekursiven Selbstaufruf von `SaveSGMLElement` bearbeitet. Nach diesen Schritten wird nun das Ende des SGML-Elementes geschrieben, und zum Vater-Element zurückgekehrt, da es sich ja um eine Baum-Struktur handelt, die abgearbeitet wird. Nun wird mittels Rekursion ab diesem Element mit dem Speichern fortgefahren.

Funktionsname:

SaveDTDElements SaveDTDElements

Signatur:

void SaveDTDElements(void)

Beschreibung:

Speichert die einzelnen DTD Element Definitionen.

Vor- und Nachbedingungen:

Wird von `g_saveDTD` aufgerufen.

Abhängigkeiten:

ASpecT-Runtime-Bibliothek, `ISGenerator.AS`, `ISTypes.AS`, `DTD.AS`, `AviseC.h`, `stdio.h`

Implementierung:

Beim Speichern der Abhängigkeiten eines Elementes wird zuerst einmal eine Gruppe geöffnet, in die alles weiter geschrieben wird. Nun wird eine Schleife begonnen, die solange läuft, bis von der IS gemeldet wird, daß keine weiteren Abhängigkeiten dieses Elementes bestehen. Innerhalb dieser Schleife geschieht nun folgendes: Zunächst wird ermittelt, was die IS als nächstes anzubieten hat (`nix`, `group`, `endgroup`, `element`, `attribute`, `data`). Sollte es sich um eine `group` handeln, wird diese mittels Rekursion gespeichert. Handelt es sich aber um ein `endgroup`, so wird die Gruppe geschlossen und ggf. noch mit einem *, + oder ? versehen (näheres hierzu siehe SGML-Norm). Falls aber ein Element gemeldet wird, so wird dieses geholt und gespeichert. Wird ein Attribute gemeldet, so wird noch nichts vorgenommen. Bei einem `data` hingegen wird geprüft, ob es sich um ein pcdata, cdata, ndata oder ein empty handelt. Bisher wird nur ein pcdata unterstütz, und auch entsprechend gespeichert. Noch ein Wort zu der Variable `ngb`: Diese ist für die Trennzeichen der einzelnen Elemente einer Gruppe zuständig, und zwar im Zusammenhang mit der Variable `ngstrptr`. Am Ende wird nun noch die aktuelle Gruppe geschlossen und eine rekursions Ebene beendet.

7.1.5 Restriktionen, Fehler

Das Speichern von Attributen wird noch nicht unterstützt, ist aber schon angedacht worden (siehe Quellcode). Die Funktion `SaveSGMLElement` ist stark rekursiv gehalten, dies bedeutet jedoch, daß sehr viel Speicher verbraucht wird. Die Funktionen zum Speichern sollten bei Gelegenheit in ASpecT implementiert werden.