Lexikalische Analyse

INHALT

(Dienstag, den 5. Mai 2003)

- Aufgabe
- Lexeme
- reguläre Ausdrücke
- Erzeugung nichtedeterministischer Austomaten aus REG
- Potenzmengenkonstruktion von deterministischen endlichen Automaten

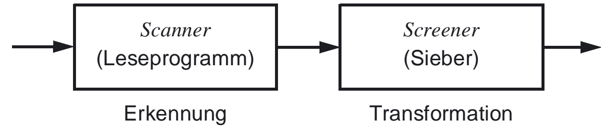
(Montag, den 12. Mai 2003)

- Programmierung endlicher Automaten
- Mehrdeutige Lexemfolgen und Vorschau
- Sieben (*screening*)
- Bezeichnerverwaltung mit Hashing

Aufgabe der lexikalischen Analyse

Erkennung
Zusammenfassung von Zeichen zu Zeichenketten
(*Lexeme* genannt)

Transformation
effiziente Codierung der Lexeme als Zahlen
insbesondere eindeutige Codierung der Bezeichner



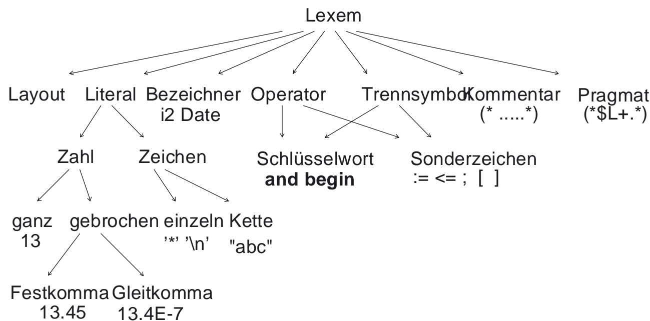
Diese Struktur haben viele Phasen eines Übersetzers
(Analyse — Synthese)

Erkennung
ist formal fundiert (Grammatiken, Automaten usw.)

Transformation
sorgt für die effiziente Organisation des Ablaufs und der Daten

Lexeme in Programmiersprachen

atomare Textbausteine der Sprache



Layout: Leerzeichen und -zeilen, Tabulatoren
Literal: Schreibweise für primitive eingebaute Werte
Bezeichner: vereinbare Namen
manchmal in verschiedene Klassen unterteilt
(z.B. Miranda: Konstruktornamen und andere)
Operator: Schreibweise für eingebaute Funktionen
manchmal auch vom Benutzer vereinbar, wie in Ada
Trennsymbol (*delimiter*): zur Lesbarkeit,
ohne Bedeutung

Was ist ein Lexem und was nicht?
Zeichenkette und Kommentar in Pascal?
Variable in Pascal?
die nicht selbsteinbetenden Anteile der Syntax

Beschreibung von Lexemen

reguläre Ausdrücke...

E =	ε	leere Zeichenfolge
|	‘Z’	Einzelzeichen
|	$E\ E$	Verkettung
|	E ‘?’	Option
|	E ‘+’	Wiederholung
|	E ‘*’	optionale Wiederholung
|	‘(’ E ‘)’	Klammerung
|	E ‘|’ E	Alternative

... definieren reguläre Sprachen

$$L(\varepsilon) = \{\varepsilon\}$$
$$L(‘Z’) = \{Z\}$$
$$L(E\ F) = L(E)\ L(F)$$
$$L(E\ ‘?’) = L(E \mid \varepsilon) = L(E) \cup \{\varepsilon\}$$
$$L(E\ ‘+’) = L(E \mid E\ E+) = \{v_1 \dots v_n \mid v_i \in L(E), n > 0, \}$$
$$L(E\ ‘*’) = L((E+)?) = \{v_1 \dots v_n \mid v_i \in L(E), n \geq 0, \}$$
$$L(‘(’\ E\ ‘)’) = L(E)$$
$$L(E\ ‘|’\ F) = L(E) \cup L(F)$$

Beispiel Binärzahlen

((‘0’ | ‘1’)* ‘**.**’)? (‘0’ | ‘1’)+

reguläre Definitionen: mit Namen und Abkürzungen
z. B. für den Teilausdruck ‘0’ | ‘1’

Beispiel Binärzahlen mit Abkürzungen

Bit = 0’ | ‘1’.   Bitnumber = (Bit* ‘**.**’)? Bit+

Übungen zu regulären Ausdrücken 1

Zahlen in Pascal und in PL0

Beispiele `1 0123 12.4 .45 12.45E+34`

Digit = “0” | ... | “9”
Nat = Digit+
Sign = ‘-’ | ‘+’
Exp = ‘E’ [Sign] Nat
Number = (Nat | Digit* ‘.’ Nat [Exp])

Kommentare in Pascal und in PL0

Begrenzung mit `(*...*)`
Beispiele `(* Hallo *) (**** hier ****) (**)`

Stars = ‘*’+
NoStar = **all but** ‘*’
NoStarClose = **all but** ‘*’ ‘)’
Comment = ‘(‘ ‘*’ (NoStar | Stars NoStarClose)* Stars ‘)’

Bemerkung

all but ‘*Z*’
definiert alle druckbaren Zeichen *ohne Z*
E / *F* definiert *E*, wenn danach *F* folgt (*Vorschau*)
Das vereinfacht Kommentare:

NoStar = **all but** ‘*’ | Stars / **all but** ‘)’
Comment = ‘(‘ ‘*’ { NoStar } ‘*’ ‘)’

Übungen zu regulären Ausdrücken 2

Zeichenketten in Pascal

Begrenzung mit "..."
Auftreten von " in der Zeichenkette
muß verdoppelt werden
Zeichenketten dürfen nicht leer sein (!)

Beispiele `"*" """" "Hallo $§??"`

Stringchar = **all but** ‘”’ | ‘”’ ‘”’
String = ‘”’ Stringchar+ ‘”’

Bezeichner in C und Ada

Buchstaben, Ziffern und “_”, nicht beginnend mit Ziffer
Ada: “_” nur einzeln und nicht am Anfang oder Ende

Beispiele `A a1 hallo_ihr`

Letter = “a” | “b” | ... | “Y” | “Z”
C-Letter = Letter | ‘_’
Pascal-Id = Letter { Letter | Digit }
C-Id = C-Letter { C-Letter | Digit }
Ada-Id = Letter { [‘_’] Letter | Digit }

Reguläre Definition der PL0-Lexeme

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter = a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z
| A | B | C | D | E | F | G | H | I | J | K | L
| M | N | O | P | Q | R | S | T | U | V | W
| X | Y | Z
Stars = ‘*’ { ‘*‘ }
comment = ‘(‘ ‘*’(**all but** ‘*’ | Stars (**all but** ‘*’ ‘)’))* Stars ‘)’
layout = { -- space, newline, tabulator-- }
number = digit { digit }
ident = letter { letter| digit }
keyword = **begin** | **call** | **const** | **do** | **end**
| **if** | **odd** | **procedure** | **then** | **var**
| **while**
delimiter = , | ; | . | (|)
operator = := | ? | ! | = | # | < | > | <= | >=
| + | - | * | /
PL0-Lexem = comment | layout | number | ident
| keyword | operator

Erkennung von Lexemen

Reguläre Sprachen können mit
***endlichen Automaten* erkannt werden**

endliche Zustandsmenge
Aktionen

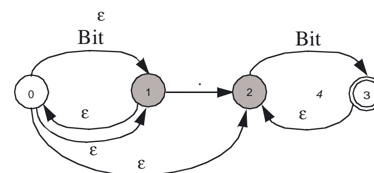
- ein Zeichen *lesen* und in einen Folgezustand übergehen
- Lexem in einem Endzustand *akzeptieren*
- in anderen Zuständen *Fehler* melden

der Automat ist *deterministisch*,
wenn unter einem Zeichen immer *höchstens ein* Übergang möglich ist.

Darstellung als *Zustandsübergangsdiagramme*

Knoten = Zustände
Kanten = Übergänge (markiert mit einer Zeichenmenge)
Endzustände = Doppelkreise
Startzustand ist *mit* * markiert

Beispiel DEA für Binärzahlen

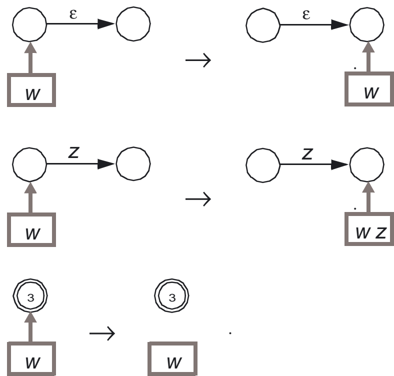


Erzeugen von Lexemen mit endl. Automaten

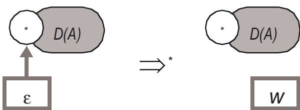
Das Prinzip

folge einem Pfad vom Startzustand in einen Endzustand und sammle die Markierungen der Pfeile auf

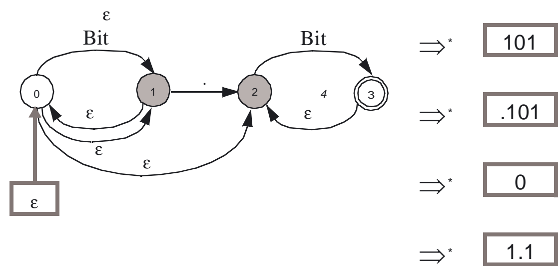
Regeln



Erzeugen



Beispiel: Erzeugung von Binärzahlen

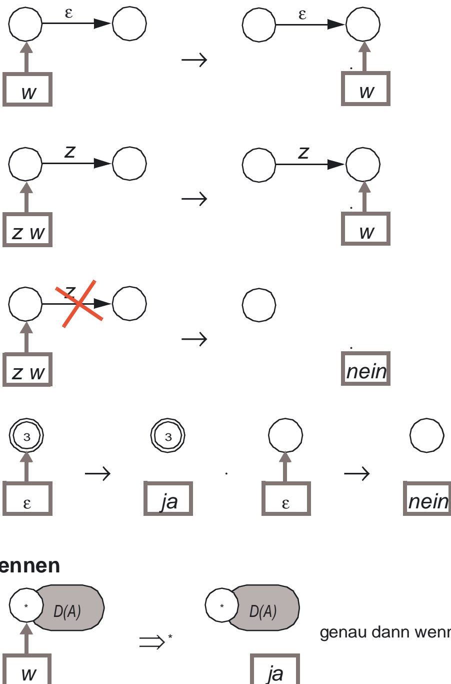


Erkennen von Lexemen mit endl. Automaten

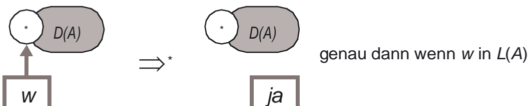
Das Prinzip

folge einem Pfad vom Starzustand in einen Endzustand, der mit dem zu erkennenden Wort markiert ist

Regeln

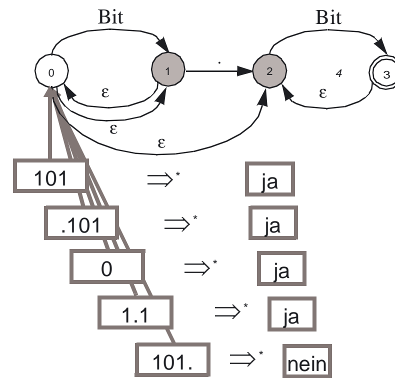


Erkennen



genau dann wenn *w* in *L*(*A*)

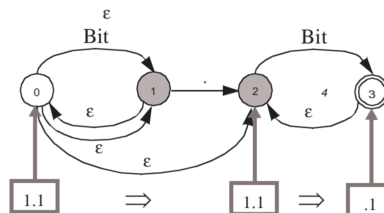
Beispiel: Erkennung von Binärzahlen



in nichtdeterministischen Automaten

kann die Erkennung steckenbleiben,

obwohl es einen (anderen) Weg der Erkennung gibt.



Endliche Automaten

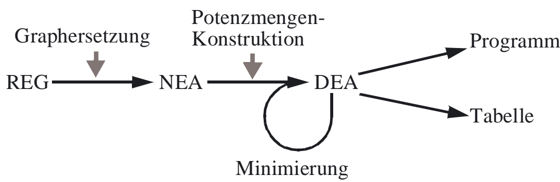
theoretische Ergebnisse

- jede reguläre Sprache
  kann mit einem endlichen Automaten erzeugt / erkannt werden.
- jeder reguläre Ausdruck
  kann in einen endlichen Automaten übersetzt werden
  (der im allgemeinen *nicht* deterministisch ist!).
- jeder nicht deterministische endliche Automat
  kann in einen deterministischen übersetzt werden
  (Potenzmengen-Konstruktion).
- jeder deterministische Automat
  kann bzgl. der Zustandsmenge minimiert werden.

Details in [ASU86] und [WM96]!

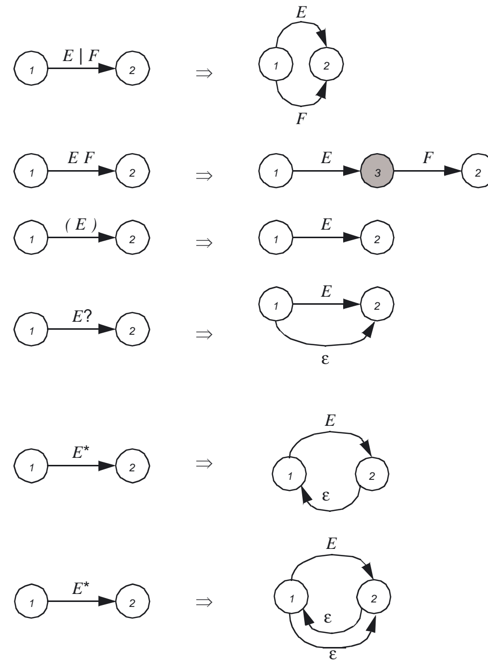
Folgerung

für jeden regulären Ausdruck kann *automatisch*
ein *effizienter* erkennender Automat erzeugt werden



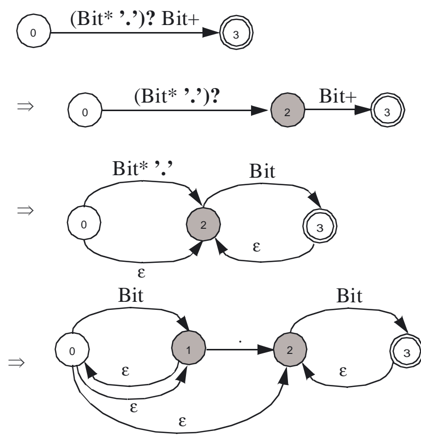
Graphersetzung: REG → NEA

Regeln

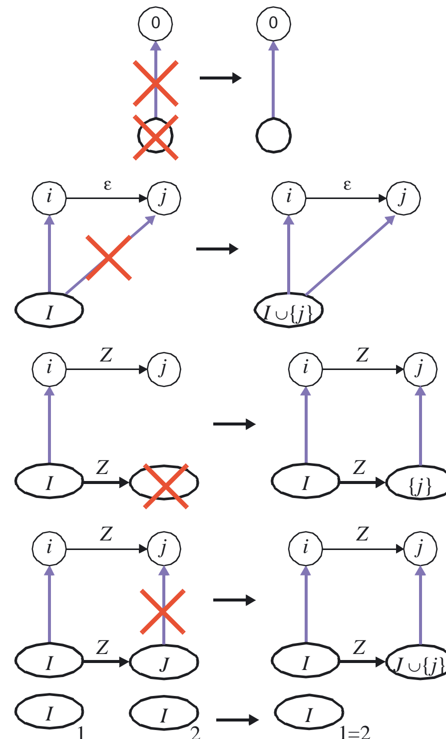


NEA für Binärzahlen (1)

Konstruktion des NEA



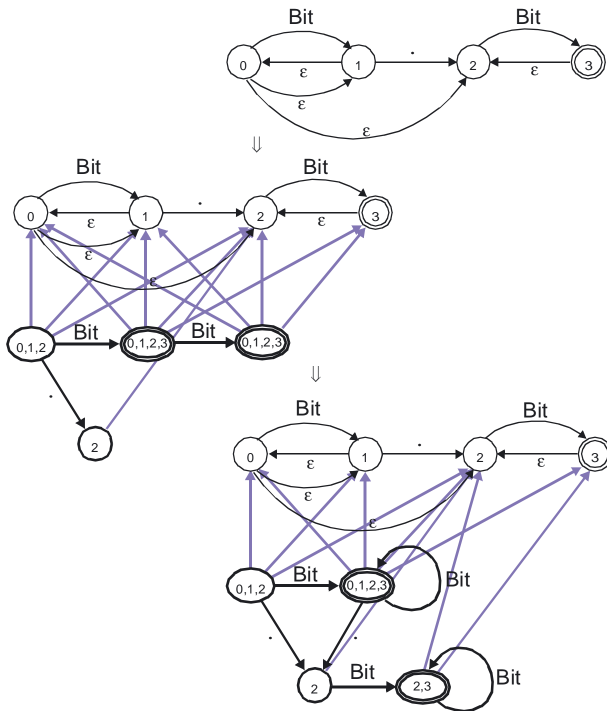
Graphersetzung: NEA → DEA



Beispiel Binärzahlen (2)

Potenzmengen-Konstruktion des DEA

Betrachte alle Zustände mit ε-Übergängen als äquivalent



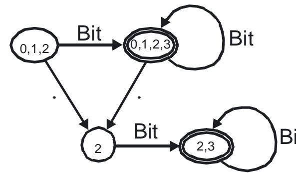
Minimierung von DEAs

Prinzip

- vereinige äquivalente Zustände
- Entferne nutzlose Zustände
  (ohne Verbindung zu Anfangs- oder Endzustand)

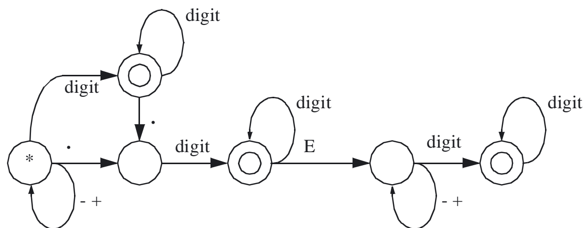
Algorithmus

vergleiche Aho-Sethi-Ullman 1988

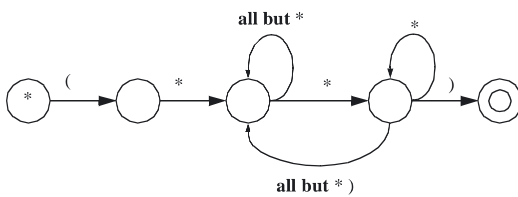


Übungen zu endlichen Automaten 1

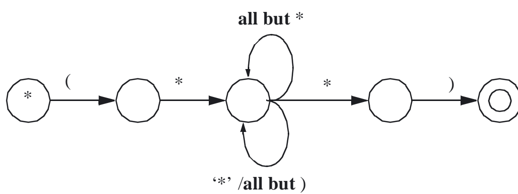
Zustandsübergangs-Diagramm für Zahlen



Zustandsübergangs-Diagramm für Kommentare



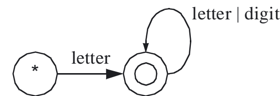
und mit Vorschau:



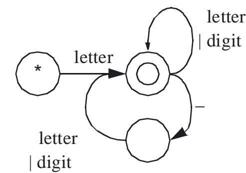
Übungen zu endlichen Automaten 2

Zustandsübergangs-Diagramm für Bezeichner

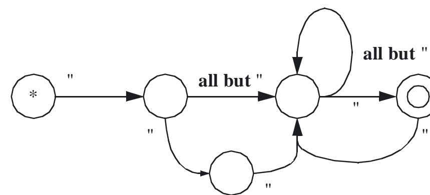
in Pascal



in Ada



Zustandsübergangs-Diagramm für Pascal-Zeichenketten



Sprünge

Jeder Zustand ist eine Fallunterscheidung
mit Sprüngen zu den Marken der Folgezustände

```
procedure BitNumber;
label 1,2,3;
begin
   1: case ch of
         '0','1': nextchar; goto 1
      |  '.':       nextchar; goto 2
      else (* kein Binärbruch *) end;
   2: case ch of
         '0','1': nextchar; goto 3
      else... { Fehler } end;
   3: case ch of
         '0','1': nextchar; goto 3
      else { fertig } end
end
```

Tabellen

Tabelle 1:

Zustände	0	1	.	...
1	1	1	2	-1
2	3	3	-1	-1
3	3	3	0	0

```
procedure BitNumber;
begin q := 1;{ Startzustand }
   while q > 0 do
      begin q := state[q]; nextchar end
   if q < 0 then ... ; { Fehler }
end
```

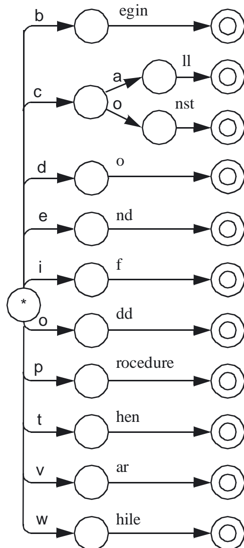

Schleifen

statt Sprüngen können auch Schleifen benutzt werden

```
procedure BitNumber;
begin
   while ch in ['0','1'] do nextchar;
   if ch /= '.' then ... { Fehler } else nextchar;
   if not (ch in ['0','1'])
   then ... { Fehler }
   else
      repeat nextchar;
      until not (ch in ['0','1']);
end
```


Erkennen der Schlüsselworte

Entscheidungsbaum, verwoben mit Bezeichnern



ziemlich großer Automat

Schlüsselworte werden als Bezeichner erkannt

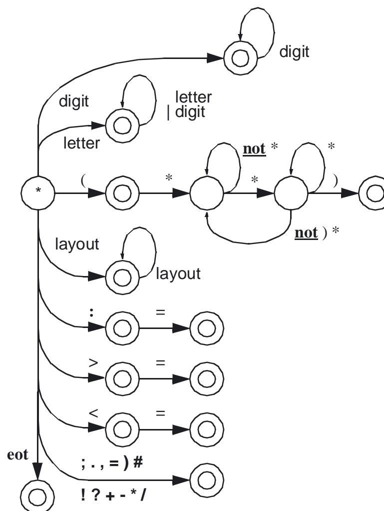
und hinterher aussortiert (im Sieber).

einfache Mehrdeutigkeiten

Problem

ein Lexem besteht aus zwei anderen

Beispiel (PL0): "`:=`"



Dann nimmt man den längsten Präfix (das Prinzip *longest match*)
Es gibt allerdings kein PL0-Programm,
in dem "`:`" und "`=`" direkt aufeinander folgen dürfen

ernste Mehrdeutigkeiten

Problem

longest match versagt bei reellen Zahlen

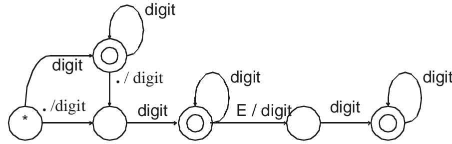
number = {digit} **[.** digit {digit} **[E** [+|-] digit {digits}**]]**

Beispiel Erkenne `1..10`

- *longest match* liefert einen Fehler (Nachkommastellen fehlen)
- Tatsächlich sollte nur eine ganze Zahl erkannt werden

Lösung

Der Übergang hängt vom nächsten Zeichen ab (Vorschau, *lookahead*)



manche Sprachen brauchen mehr Vorschau

FortranPascal

```
DO 5 I = 1,25 ...=for i := 1 to 25 do ...
DO 5 I = 1.25 ...=do5i := 25
```

Frage

der reguläre Ausdruck für *number* ist fehlerhaft. *Wo*?

Antwort: er läßt die leere Zeichenkette zu!

pathologische Mehrdeutigkeiten

mehr Vorschauzeichen

Fortran

`DO 5 I = 1.25` Zuweisung an `DO5I`
`DO 5 I = 1,25` Zählschleife

```
   for i := 1 to 25
   do begin ... 5: end;
```

PL / 1

`DECLARE (A, B, C, D);` Aufruf der Prozedur `DECLARE`
`DECLARE (A, B, C, D)=` Anfang einer Vereinbarung

Algol-68

Operatoren sind Folgen von Spezialzeichen, die neu vereinbart werden dürfen
(Leerzeichen werden immer ignoriert!)

`OP +:= = ( ... )` oder auch `OP+:==( ... )`

Nach `OP` muß von jedem Operatorzeichen hinten ein `=` abgespalten werden

lexikalische Fehlerbehandlung

mögliche lexikalische Fehler

ein unerwartetes Zeichen taucht auf

Beispiel `begin % end.`

rechter Begrenzer eines eingeschlossenen Lexems fehlt

Beispiel

```
begin "Hallo
end.
```

allgemeine Aktionen

jeweils ein Zeichen wird

- überlesen
- eingefügt
- ausgetauscht

am Beispiel PL0

nicht erwartete Zeichen (‘”&’ ‘§’) *überlesen*

fehlender Abschluß eines Kommentars (‘*)’) am Ende des *Programms* einfügen

und am Beispiel PL1

fehlender Abschluß einer Zeichenkette (‘”) am Ende der *Zeile* einfügen

für fehlende Nackommaziffern 0 einsetzen

Fehler-Behebung

Situation

die Syntaxanalyse erwartet ein *Schlüsselwort* K und der *Bezeichner* X wird erkannt

Wenn X dem Schlüsselwort K gleicht nach

- *Einfügen* eines Zeichens
- *Streichen* irgendeines Zeichens
- *Austauschen* eines Zeichens durch irgendein ein anderes
- *Vertauschen* zweier benachbarter Zeichen

dann wird K statt X abgeliefert.

Beispiel

Das Schlüsselwort **`begin`** wird erwartet
ein Bezeichner X wird gelesen

`begin` wird statt X abgeliefert, wenn

- X = `egin`, `bgin`, `bein`, `begn` oder `begi` (*vergessen*)
- X = `beginn`,... (*hinzugefügt*)
- X = `began`,... (*verwechselt*)
- X = `ebgin`, `bgein`, `beign` (*vertauscht*)

Problem

kurze Schlüsselworte wie **`if`** ähneln vielleicht anderen Schlüsselworten:
`i`, `f`, `iff`, **`of`**, `fi`

Transformation von Lexemen

Sieben (*screening*)

Transformationsregeln

- Layout und Kommentare werden *unterdrückt*
- Pragmate werden auch *unterdrückt*, ändern aber den Zustand des Übersetzers
- Trennsymbole und Operatoren werden als Symbole *codiert*
- Literale werden zusätzlich in interne Werte *umgerechnet*
- Bezeichner werden zusätzlich eindeutig *verschlüsselt*

Beispiel PL0

Zeichenkette		(Symbol, Wert, Schlüssel)	
`‘ ’`	⇒	ε	
`(**** HALLO***)`	⇒	ε	
`(*$L-*)`	⇒	ε	`(Listing:= false;)`
‘+’	⇒	(plussym, 0 , 0)	
‘[‘	⇒	(sqbrsym, 0 , 0)	
`25E-2`	⇒	(numsym, 0.01, 0)	
`x`	⇒	(idsym, 0 , 7)	
`begin`	⇒	(beginsym, 0 , 0)	

Bezeichner-Verwaltung

allgemeine Anforderungen

an Bezeichner werden während der Übersetzung viele Informationen geknüpft

Wir brauchen eine effiziente Darstellung von *Mengen*

Insbesondere muß folgendes sehr effizient sein:

- Zuordnung von Attributen an Bezeichner
  Ident → Attribute
- Vergleich von Bezeichnern

konkreter

Bezeichner sollen als Zahlen *verschlüsselt* werden dann

- können Abbildungen als Felder dargestellt werden
- ist der Gleichheits-Test effizient

Konsequenz

von den vielen Möglichkeiten, Mengen darzustellen, wählen wir den Streuspeicher (*hashing*)

Streuspeicher (hashing)

Prinzip

Die Menge der Bezeichner wird *willkürlich*, aber *deterministisch* und *möglichst gleichmäßig* in Eimer (*buckets*) aufgeteilt

jeder Eimer wird als Menge dargestellt, z.B. als

- Liste
- geordnete Liste
- geordneter Baum

wenn es genügend Eimer gibt, enthält jeder Eimer wenige Bezeichner und der Suchaufwand ist annähernd konstant

Bezeichner als abstrakter Datentyp

Schnittstelle

```
type Id, Attr, Tab,
empty: void → Tab,
enter: String × ATTR × Tab → Id × Tab
attribute: Id × Tab → ATTR

hidden
key: String × Tab → Id
enterId: Id × ATTR × Tab → Tab

axioms

enter(s,a,t) = (key(s,t), enterId(key(s,t),a,t))

s = s’ ⇔ key(s,t) = key(s’,t)

i = j ⇒
enterId(i, a, enterId(j, b, t)) = enterId(i, a, t)
i ≠ j ⇒
enterId(i, a, enterId(j, b, t))
      = enterId(j, b, enterId(i, a, t))

enterId(i, a, empty) = enterId(i, a, empty)

i = j ⇒ attribute(i, enterId(j, a, t)) = a
i ≠ j ⇒
attribute(i, enterId(j, b, t)) = attribute(j, t)
attribute(i, a, empty) = unknown
```

Attribute sind beliebige Typen

von denen nur die Konstante `unknown` erwartet wird

Implementierung der Bezeichnertabelle

Realisierung mit Prozeduren

Die Tabelle selbst ist implizit

```
typeId= ...;
   Attr= ...;
   Tab= ...;
varIdTable: Tab;
procedure initIdTable ;
   (* setze die Tabelle auf “leer” *)
procedure enter(s: String, a: ATTR) : Id;
   var i: Id;
   begin
   i:= key(s); enterId(i,a,t));
   return i
   end;
(* hidden *)
procedure key (s:String): Id;
   (* verschlüssele s eindeutig *)
procedure enterId (i: Id, a: ATTR);
   (* trage a an der Stelle i ein *)
procedure attribute (i: Id): ATTR;
   (* liefere a für den Bezeichner i *)
```


Streuspeicher (*hashing*)

hash-Funktion

```
const hashmax = ...;
type BucketNo = 0..hashmax-1;
procedure hash(s: String) : BucketNo;
   begin ln:= length(s);
   return (ln+s[1]+s[ln])) mod hashmax
   end;
```

Die Tabelle ist ein Feld von Bezeichner-Listen

```
typeId= Bucket;
   Bucket= pointer to record
         spelling:String;
         attribute:ATTR;
         next:Bucket;
        end;
   Tab= array BucketNo of Bucket;
varIdTable: Tab;
procedure initIdTable ;
   var i: Integer;
   begin
   for i:= 0 to hashmax-1 do IdTable[i] := nil;
   end;
procedure enterId (i: Id, a: ATTR);
   begin
   i^.attribute := a
   end;
procedure attribute (i: Id): ATTR;
   begin
   if i # nil
   then return i^.attribute
   else return unknown
end;
```


Kollisionsbehandlung 1

hashing and chaining

lineare (ungeordnete) Liste von `Bucket`’s

```
procedure key (s:String): Id;
   begin
   return Idtable[hash(s)])
   end;
procedure enterBucket
         (s:String;b: var Bucket):Id;
  begin
  if b # nil
  thenif b.spelling = i
      then return b
      else return enterBucket (s,b^.next)
  elsebegin
      new(b);
      b.spelling := s;
      b.attribute := unknown;
      b.next := nil
      end
  end;
```


Kollisionsbehandlung 2

Variante ordererd chaining

```
procedure enterBucket
                  (s:String, b: var Bucket): Id;
  var n: Bucket;
  begin
  if b # nil
  then  if b.spelling = i
        then return b
        else if b.spelling  < i
             then return enterBucket (s,b^.next)
             else begin
                  n := b;
                  new(b);
                  b.spelling := s;
                  b.attribute := unknown;
                  b.next := n;
                  return b
                  end
  else  begin
        new(b);
        b.spelling := s;
        b.attribute := unknown;
        b.next := nil
        return b
        end
  end;
```

Kollisionsbehandlung 3

Variante `Bucket`'s als Überlauf im Feld `Tab`

```
typeId= 0 .. IdMax-1;
   Bucket= record
         spelling:String;
         attribute:ATTR;
         next:Id;
      end;
   Tab  = array Id of Entry;
```


Kollisionsbehandlung 4

Quadratisches Sondieren (*quadratic hashing*)

das gesamte Feld liegt im Bereich der *hash*-Funktion

bei Kollision an der Stelle `i` werden die Stellen

`i+1`, `i+2`, `i+4`, `i+8`,... (**mod** `hashmax = IdMax`) gesucht.

Wenn `hashmax` eine Primzahl ist,

werden so n/2 Stellen abgeklappert,

bevor i den alten Wert wieder erreicht.

Behandlung der Schlüsselworte

vor Einlesen des Programms wird aufgerufen:

```
i:= enter("begin", beginsym);
i:= enter("call", callsym);
...
i:= enter("type", typesym);
i:= enter("var", varsym);
```

wobei angenommen wird:

```
type ATTR = (beginsym, ..., eotsym);
```

Behandlung der anderen Bezeichner

```
sym := attribute(key(s,IdTable), IdTable);
```

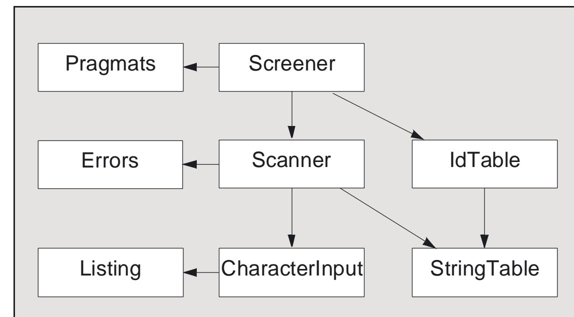

Schnittstelle zur syntaktischen Analyse

Der Datentyp Lexem

```
typePosition =record
         line : Integer;
         col  : 1..256;
      end;
   Lexem =record
         pos : Position;
         case sym: ATTR of
            idsym: ident: Id;
         | numsym: value;
         else ();
      end;
```


Struktur der lexikalischen Analyse

Moduln



Kommunikation mit der Syntaxanalyse

```
procedure NextSymbol;
```

PL0 in *lex* (= *flex* = *rex*)

```
layout[ \t\n]
whitespace{layout}+
letter[A-Za-z]
digit[0-9]
prchar[^\\\']|\\n|\\t|\\\'
noast [^\*]
noclose[^\)]
comment\(\*({noast}|\*{noclose})*\*\)
string\'({prchar})*\'
identifier{letter}({letter}|{digit})*
number{digit}+
{whitespace}{}
{comment}{}
CONST{return CONST;}
...
VAR{return VAR;}
{identifier}{... ; return ID;}
{number}{... ; return NUM;}
{string}{... ; return STR;}
"<="{return LE;}
">="{return GE;}
":="{return BECOMES;}
":"{return COLON;}
"("{return OPEN;}
")"{return CLOSE;}
"*"{return TIMES;}
"/"{return DIVIDES;}
"="{return EQ;}
"#"{return NE;}
"<"{return LT;}
">"{return GT;}
"?"{return IN;}
"!"{return OUT;}
```