Aus Reinhard Wilhelm, Dieter Maurer
Übersetzerbau – Theorie,
Konstruktion, Generierung
2. Auflage, Springer 1997

Kapitel 2

Übersetzung imperativer Programmiersprachen

In diesem Kapitel wollen wir eine intuitive Vorstellung davon geben, **was** ein Übersetzer einer imperativen Programmiersprache tut, während wir später erst genau erklären, **wie** er es tut. Dazu definieren wir präzise, aber intuitiv die Korrespondenz zwischen Programmen einer imperativen Quellsprache und den durch Übersetzung gewonnenen Programmen eines Zielrechners.

Als Quellsprache wählen wir ein leicht abgemagertes Pascal. Als Zielrechner wählen wir eine abstrakte Maschine, die P-Maschine, deren Architektur so entworfen wurde, daß die Übersetzung von Pascal in ihre Maschinensprache möglichst einfach ist.

Die Definition der Korrespondenz erfolgt mit Hilfe von Übersetzungsfunktionen, die schrittweise für die behandelten Pascal-Konstrukte angegeben werden.

2.1 Sprachkonzepte und ihre Übersetzung

Imperative Programmiersprachen besitzen u.a. die folgenden Konstrukte und Konzepte, die auf Konstrukte, Konzepte und Befehlsfolgen abstrakter oder realer Rechner abgebildet werden müssen:

- **Variablen**, das sind Behälter für Datenobjekte, die ihren Inhalt (Wert) im Laufe der Programmausführung ändern können. Die Wertänderungen erfolgen durch die Ausführung von **Anweisungen** wie etwa Wertzuweisungen (assignments). Mehrere Variablen können in Aggregaten, Feldern (arrays) und Verbunden (records) zusammengefaßt werden. Die aktuellen Werte der Variablen zu irgendeinem Zeitpunkt machen einen Teil des **Zustands** des Programms zu diesem Zeitpunkt aus. Variablen werden in Programmen mit **Namen** bezeichnet. Da auch Konstanten, Prozeduren usw. mit Namen bezeichnet werden, sprechen wir von Variablenbezeichnungen, Konstantenbezeichnungen usw., wenn wir diese speziellen Arten von Namen unterscheiden wollen. Variablenbezeichnungen müssen Speicherzellen von Maschinen zugeordnet werden, die die jeweils aktuellen Werte enthalten. Enthält die Programmiersprache rekursive Prozeduren mit lokalen Namen, so entstehen durch den Aufruf einer Prozedur neue **Inkarnationen** der lokalen Variablenbezeichnungen; ihnen muß dann jeweils neuer Speicherplatz zugeordnet werden. Bei Verlassen der Prozedur werden die Zellen für diese Inkarnationen

wieder freigegeben. Deshalb werden solche Sprachen mit Hilfe einer kellerartigen Speicherverwaltung implementiert.

- **Ausdrücke**, das sind aus Konstanten, Namen und Operatoren zusammengesetzte Terme, die bei der Ausführung **ausgewertet** werden. Ihr Wert ist i.a. zustandsabhängig, da bei jeder Auswertung die aktuellen Werte der im Ausdruck enthaltenen Variablen zur Auswertung benutzt werden.

- **explizite Angabe des Kontrollflusses.** Der in den meisten imperativen Programmiersprachen existierende Sprungbefehl, **goto**, kann direkt in den unbedingten Sprungbefehl der Zielmaschine übersetzt werden. Höhere Kontrollkonstrukte wie bedingte (if) oder iterative (while, repeat, for) Anweisungen werden mit Hilfe bedingter Sprünge übersetzt. Ein bedingter Sprung folgt auf eine Befehlsfolge zur Auswertung einer Bedingung. Fallunterscheidungen (case) lassen sich für manche Quellsprachen auf manchen Zielmaschinen durch indizierte Sprünge effizient realisieren. Dabei wird die im Befehl angegebene Sprungadresse mit einem vorher berechneten Wert modifiziert. Prozeduren sind ein Mittel, eine Folge von Anweisungen von einer Programmstelle aus zu aktivieren und nach ihrer Abarbeitung dorthin zurückzukehren. Dazu muß die Maschine einen Sprungbefehl haben, der seine Herkunft nicht vergißt. Der Rumpf der Prozedur kann bei jeder Aktivierung (Aufruf) mit aktuellen Parametern versorgt werden. Dies, zusammen mit dem Kreieren von Inkarnationen lokaler Namen, erfordert eine komplexe Speicherorganisation, die häufig durch spezielle Maschineninstruktionen unterstützt wird.

2.2 Die Architektur der P-Maschine

Die (abstrakte) P-Maschine wurde entwickelt, um die Züricher Pascal-Implementierung portabel zu machen. Wollte jemand auf seinem Rechner Pascal implementieren, so mußte er „nur" einen Interpreter für die Instruktionen dieser abstrakten Maschine schreiben. Dann konnte er den in Pascal geschriebenen und in P-Code übersetzten Pascal-Übersetzer auf seinem realen Rechner zum Laufen bringen.

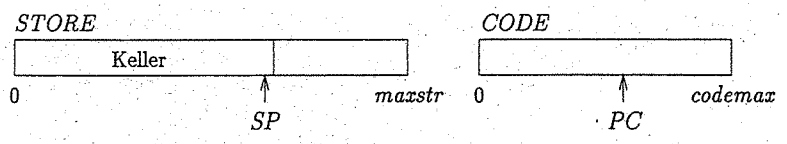


Abb. 2.1: Die Speicher der P-Maschine und einige Register

Die Architektur und die Instruktionen der P-Maschine werden schrittweise eingeführt, so wie wir sie für die Übersetzung der jeweils behandelten Konzepte in

der Quellsprache brauchen. Zunächst wollen wir nur die Speicher, einige Register und den Hauptzyklus (main cycle) der P-Maschine einführen. Die P-Maschine hat einen Datenspeicher *STORE* der Länge *maxstr* +1 und einen Programmspeicher *CODE* der Länge *codemax* +1 . Am unteren Ende des Datenspeichers, d.h. ab Adresse 0, liegt ein pulsierender Keller. Ein Register *SP* (stack pointer) zeigt auf die oberste belegte Zelle. Beachten Sie, daß später aus Gründen der Übersichtlichkeit in Abbildungen mit vertikal dargestellten Kellern die „oberste" Kellerzelle immer unten ist, während die tieferen Adressen von Speicherzellen am oberen Bildrand zu finden sind. Es gibt Instruktionen, die Inhalte von explizit adressierten Zellen oben auf dem Keller speichern und den Keller dabei verlängern, bzw. umgekehrt den Inhalt der obersten Kellerzelle in einer explizit adressierten Zelle abspeichern und den Keller verkürzen, siehe Tabelle 2.2. Die Befehle sind teilweise parametrisiert mit Typen; *N* steht für „numerischer" Typ, also integer, real oder Adresse, *T* für beliebigen einfachen Typ, also für numerische, logische, Zeichen- und Aufzählungstypen und für Adressen, i für integer, r für real, b für boolean, a für Adresse. Die arithmetischen Operationen auf Adressen sind die gleichen wie die auf integer-Operanden; aber die Wertebereiche sind i.a. verschieden. Die mit einem Typ indizierten Operatoren bezeichnen die entsprechenden Operationen auf den zugrundeliegenden Wertebereichen, $<_i$ etwa den Vergleich zweier ganzer Zahlen auf „kleiner".

Andere Instruktionen benutzen implizit den Inhalt von *SP*, indem sie auf dem Inhalt der **obersten belegten Kellerzelle** oder den Inhalten der **obersten Kellerzellen** operieren, evtl. den Inhalt von *SP* verändern und das Resultat der Operation in der (neuen) obersten Kellerzelle hinterlassen, siehe Tabelle 2.1.

Natürlich hat die P-Maschine auch einen Befehlszähler, das Register *PC*, dessen Inhalt die Adresse des nächsten auszuführenden Befehls ist. Alle Befehle der P-Maschine belegen jeweils eine Zelle im Programmspeicher, so daß außer bei Sprüngen der Inhalt des *PC*-Registers in Schritten von 1 hochgezählt wird. Also sieht der Hauptzyklus der P-Maschine folgendermaßen aus:

do
  $PC := PC + 1$;
  führe den Befehl in Zelle $CODE[PC - 1]$ aus
od

Anfangs ist das Register *PC* mit dem Programmanfang initialisiert, d.h. es ist $\emptyset$, da das Programm bei *CODE*[0] beginnt.

2.3 Wertzuweisungen und Ausdrücke

Die Tabellen 2.1 und 2.2 enthalten alle Befehle, die wir brauchen, um Wertzuweisungen und Ausdrücke in P-Maschinencode zu übersetzen. Welche Befehlsfolgen für eine Wertzuweisung bzw. einen Ausdruck erzeugt werden, wird durch die

Angabe von *code*-Funktionen spezifiziert. Diese Funktionen bekommen als Argument eine Wertzuweisung bzw. einen vollständig geklammerten Ausdruck. Sie zerlegen dieses Argument rekursiv und setzen die für die Komponenten jeweils erzeugten Befehlsfolgen zur gesamten Befehlsfolge zusammen.

Tabelle 2.1: P-Befehle für Ausdrücke. In der Bedin.-Spalte steht die erforderliche Situation am oberen Kellerende, in der Erg.-Spalte die resultierende Situation. Dabei beschreibt z.B. (N, N) in der Bedin.-Spalte, daß am oberen Ende des Kellers zwei numerische Werte gleichen Typs stehen müssen. Alle Vorkommen von N bzw. T in der Beschreibung einer Instruktion stehen jeweils für den gleichen Typ. Enthält die Bedin.-Spalte mehr Typzeichen als die Erg.-Spalte, so heißt das, daß durch die Ausführung dieses Befehls der Keller verkürzt wird.

Befehl	Bedeutung	Bedin.	Erg.
add N	$STORE[SP-1] := STORE[SP-1] +_N STORE[SP]$; $SP := SP-1$	(N,N)	(N)
sub N	$STORE[SP-1] := STORE[SP-1] -_N STORE[SP]$; $SP := SP-1$	(N,N)	(N)
mul N	$STORE[SP-1] := STORE[SP-1] *_N STORE[SP]$; $SP := SP-1$	(N,N)	(N)
div N	$STORE[SP-1] := STORE[SP-1] /_N STORE[SP]$; $SP := SP-1$	(N,N)	(N)
neg N	$STORE[SP] := -_N STORE[SP]$	(N)	(N)
and	$STORE[SP-1] := STORE[SP-1]$ *and* $STORE[SP]$; $SP := SP-1$	(b,b)	(b)
or	$STORE[SP-1] := STORE[SP-1]$ *or* $STORE[SP]$; $SP := SP-1$	(b,b)	(b)
not	$STORE[SP] :=$ *not* $STORE[SP]$	(b)	(b)
equ T	$STORE[SP-1] := STORE[SP-1] =_T STORE[SP]$; $SP := SP-1$	(T,T)	(b)
geq T	$STORE[SP-1] := STORE[SP-1] \geq_T STORE[SP]$; $SP := SP-1$	(T,T)	(b)
leq T	$STORE[SP-1] := STORE[SP-1] \leq_T STORE[SP]$; $SP := SP-1$	(T,T)	(b)
les T	$STORE[SP-1] := STORE[SP-1] <_T STORE[SP]$; $SP := SP-1$	(T,T)	(b)
grt T	$STORE[SP-1] := STORE[SP-1] >_T STORE[SP]$; $SP := SP-1$	(T,T)	(b)
neq T	$STORE[SP-1] := STORE[SP-1] \neq_T STORE[SP]$; $SP := SP-1$	(T,T)	(b)

Bei der Übersetzung von Wertzuweisungen muß man beachten, daß eine Variablenbezeichnung, die auf der linken Seite einer Wertzuweisung steht, anders übersetzt wird als eine Variablenbezeichnung, die auf der rechten Seite steht. Von der Variablenbezeichnung links benötigt man die Adresse der ihr zugeordneten Zelle, um deren Inhalt zu überschreiben, von der Variablenbezeichnung rechts

Tabelle 2.2: Instruktionen zum Laden und Speichern. **ldo** lädt aus einer absolut adressierten Zelle, **ldc** eine im Befehl angegebene Konstante, **ind** indirekt durch die oberste Kellerzelle. **sro** speichert in eine absolut adressierte Zelle, **sto** in eine über die zweitoberste Kellerzelle adressierte Zelle.

Befehl	Bedeutung	Bedin.	Erg.
ldo Tq	$SP := SP+1$; $STORE[SP] := STORE[q]$	$q \in [0, maxstr]$	(T)
ldc Tq	$SP := SP+1$; $STORE[SP] := q$	$Typ(q) = T$	(T)
ind T	$STORE[SP] := STORE[STORE[SP]]$	(a)	(T)
sro Tq	$STORE[q] := STORE[SP]$; $SP := SP-1$	(T) $q \in [0, maxstr]$	
sto T	$STORE[STORE[SP-1]] := STORE[SP]$; $SP := SP-2$	(a, T)	

ihren Wert, um den Wert des Ausdrucks zu berechnen. Man spricht deshalb von **Linkswerten** (L-Werten) und von **Rechtswerten** (R-Werten) von Variablen, wenn man ausdrücken will, daß man an der Adresse der Variablen bzw. ihrem aktuellen Wert interessiert ist.

Wir indizieren deshalb die *code*-Funktionen mit L bzw. R; $code_L$ erzeugt Befehle zur Berechnung des L-Wertes, $code_R$ zur Berechnung des R-Wertes. Die Funktion *code* (ohne Index) übersetzt Anweisungen, von deren Ausführung wir ja weder Adressen noch Werte erwarten.

Die Übersetzung einer Wertzuweisung $x := y$ mit integer-Variablen x und y liefert also folgende Befehlsfolgen:

Berechne L-Wert von x
Berechne R-Wert von y
sto i

Wir kümmern uns hier nicht um das Problem, wie ein Pascal-Programm syntaktisch analysiert wird, d.h. wie seine syntaktische Struktur entdeckt wird. Das bleibt dem Kapitel Syntaktische Analyse vorbehalten. Ebenfalls verlassen wir uns darauf, daß die Typkorrektheit des Eingabeprogramms bereits geprüft wurde. Wie das geschieht, zeigt das Kapitel Semantische Analyse. Die Bedingungen in den Fällen der *code*-Definition dienen lediglich dazu, die richtigen Typparameter für die zu erzeugenden Befehle zu bestimmen.

Die *code*-Funktionen benutzen als zweiten Parameter eine Funktion ρ, die allen deklarierten Variablen eine Adresse in *STORE* zuordnet. Später werden wir sehen, daß diese Adresse eine **Relativadresse** ist, d.h. eine konstante Differenz zwischen zwei absoluten Adressen in *STORE*, nämlich der tatsächlichen Adresse der Zelle für diese Variable und der Anfangsadresse eines ganzen Speicherbereichs für alle Variablen, Parameter usw. der Prozedur, in der die Variable deklariert ist. Für den Augenblick können wir $\rho(x)$ als die Adresse relativ zum Anfang von *STORE* auffassen.

Tabelle 2.3: Die Übersetzung von Wertzuweisungen

Funktion		Bedingung
$code_R(e_1 = e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **equ** T	$Typ(e_1) = Typ(e_2) = T$
$code_R(e_1 \neq e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **neq** T	$Typ(e_1) = Typ(e_2) = T$
$\vdots$		
$code_R(e_1 + e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **add** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 - e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **sub** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 * e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **mul** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1/e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **div** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(-e)\ \rho$	$= code_R\ e\ \rho;$ **neg** N	$Typ(e) = N$
$code_R\ x\ \rho$	$= code_L\ x\ \rho;$ **ind** T	x Variablenbezeichnung vom Typ T
$code_R\ c\ \rho$	= **ldc** T c	c Konstante vom Typ T
$code(x := e)\ \rho$	$= code_L\ x\ \rho;\ code_R\ e\ \rho;$ **sto** T	$Typ(e) = T$, x Variablenbezeichnung.
$code_L\ x\ \rho$	= **ldc** a $\rho(x)$	x Variablenbezeichnung.

Bemerkung:
Natürlich könnte man das Vorkommen einer Variablenbezeichnung x in einem Ausdruck in nur einem Befehl, nämlich **ldo** $T\ \rho(x)$, übersetzen. Wir haben das nicht getan, um mit dem später erweiterten *code*-Schema für Variablen Konsistenz zu erreichen.

Beispiel 2.3.1 Sei ein Programm mit den drei integer-Variablen a, b, c gegeben. Die Speicherbelegungsfunktion ρ bilde a, b, c auf die Adressen 5, 6 bzw. 7 ab. Die Übersetzung der Wertzuweisung $a := (b + (b * c))$ geschieht folgendermaßen:

$code(a := (b + (b * c)))\ \rho$
$= code_L\ a\ \rho;\ code_R\ (b + (b * c))\ \rho;$ **sto** i
$=$ **ldc** a 5; $code_R(b + (b * c))\ \rho;$ **sto** i
$=$ **ldc** a 5; $code_R(b)\ \rho;\ code_R(b * c)\ \rho;$ **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b * c)\ \rho;$ **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b)\ \rho;\ code_R(c)\ \rho;$ **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_R(c)\ \rho;$ **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; **ldc** a 7; **ind** i; **mul** i; **add** i; **sto** i
□

2.4 Bedingte und iterative Anweisungen, Anweisungsfolgen

Als nächstes wagen wir uns an die Übersetzung der bedingten und iterativen Anweisungen. Wir geben Übersetzungsschemata an für die zweiseitigen und einseitigen if-Anweisungen, **if** e **then** st_1 **else** st_2 **fi** und **if** e **then** st **fi**, und die while- und repeat- Schleifen, **while** e **do** st **od** und **repeat** st **until** e.

Die Syntax ist gegenüber Pascal etwas geändert, um die eindeutige Übersetzbarkeit der if-Anweisung zu erreichen, und um die **begin-end**-Klammerung loszuwerden.

In den Übersetzungsschemata der jetzt definierten Funktion *code* benutzen wir ein neues Hilfsmittel. Wir markieren Befehle oder auch das Ende des Schemas durch Namen, die wir in Sprungbefehlen verwenden. Die Bedeutung des Vorkommens einer solchen Marke als Ziel eines Sprungbefehls ist die folgende: man setze dort die Adresse ein, welche der Befehl erhält bzw. schon erhalten hat, der diese Marke trägt. Natürlich kann ein Codeschema bei der Übersetzung eines Pascal-Programms mehrfach und sogar rekursiv angewendet werden. Es ist aber klar, wie jeweils die Definition einer Marke und die Verwendung in einem Sprungbefehl korrespondieren. Um den in den jetzt behandelten Anweisungen beschriebenen Kontrollfluß auf der P-Maschine zu realisieren, gibt es dort bedingte und unbedingte Sprünge. Sie sind in Tabelle 2.4 beschrieben.

Tabelle 2.4: Unbedingter und bedingter Sprung

Befehl	Bedeutung	Kommentar	Bedin.	Erg.
ujp q	$PC := q$	unbedingter Sprung	$q \in [0, codemax]$	
fjp q	**if** $STORE[SP]$ = false **then** $PC := q$ **fi**; $SP := SP - 1$	bedingter Sprung	(b) $q \in [0, codemax]$	

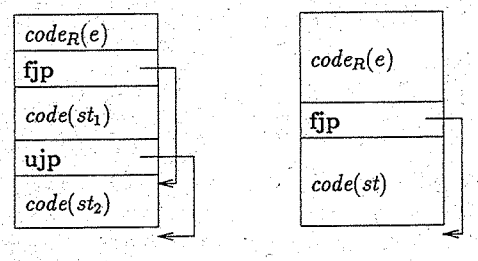
Jetzt können wir die Fälle der *code*-Funktion für bedingte Anweisungen und Schleifen angeben.

$code$(**if** e **then** st_1 **else** st_2 **fi**) $\rho =$
$code_R\ e\ \rho;$ **fjp** $l_1;$ $code\ st_1\ \rho;$ **ujp** $l_2;$ l_1: $code\ st_2\ \rho;$ l_2:

Nach der Auswertung der Bedingung e wird, falls sich als Wert *true* ergibt, der Code für st_1 ausgeführt, sonst wird über diesen samt dem abschließenden unbedingten Sprung hinweg auf den Code für st_2 gesprungen.

Die einseitige bedingte Anweisung wird folgendermaßen übersetzt:
$code$(**if** e **then** st **fi**) $\rho = code_R\ e\ \rho;$ **fjp** $l;$ $code\ st\ \rho;$ l:

Die Befehlsfolgen für die beiden Typen von bedingten Anweisungen lassen sich graphisch wie in Abb. 2.2 veranschaulichen.



(a) zweiseitige (b) einseitige

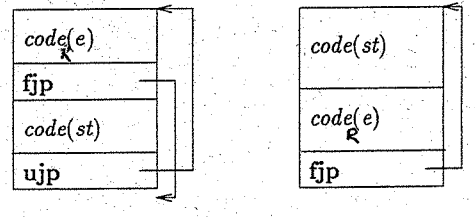
Abb. 2.2: Befehlssequenzen für bedingte Anweisungen

Die beiden iterativen Anweisungen werden folgendermaßen übersetzt:

code (**while** *e* **do** *st* **od**) $\rho = l_1$: $code_R\ e\ \rho$; **fjp** l_2; *code st* ρ; **ujp** l_1; l_2 :

code (**repeat** *st* **until** *e*) $\rho = l$: *code st* ρ; $code_R\ e\ \rho$; **fjp** l

Die Generierungsschemata für diese beiden Schleifentypen sind in Abb. 2.3 graphisch dargestellt.



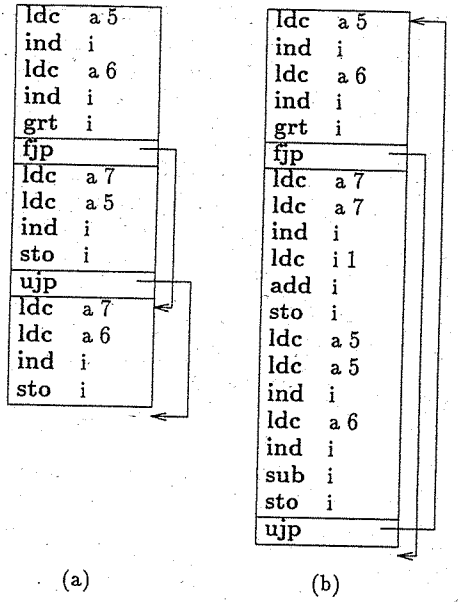
(a) while-Schleife (b) repeat-Schleife

Abb. 2.3: Codegenerierung für Schleifen **while** *e* **do** *st* **od** und **repeat** *st* **until** *e*

Beispiel 2.4.1 Wir nehmen wieder an, daß $\rho(a) = 5$, $\rho(b) = 6$ und $\rho(c) = 7$ sind. Dann übersetzen wir
die Anweisung **if** $a > b$ **then** $c := a$ **else** $c := b$ **fi** wie in Abb. 2.4 (a),
die Schleife **while** $a > b$ **do** $c := c + 1; a := a - b$ **od** wie in Abb. 2.4 (b). □

Das Übersetzungsschema für sequentiell aufeinanderfolgende Anweisungen ist sehr einfach:

code $(st_1;st_2)\ \rho$ = *code* $st_1\ \rho$; *code* $st_2\ \rho$



(a) (b)

Abb. 2.4: Übersetzung einer zweiseitigen bedingten Anweisung (a) und einer **while**-Schleife (b).

Betrachten wir eine gegenüber Pascal vereinfachte *case*-Anweisung. Nur Selektoren zwischen 0 und einer aus dem Programm ersichtlichen Konstanten k sind zugelassen. Es sind Komponenten für alle Selektoren 0, 1, ..., k vorhanden, und sie sind in aufsteigender Reihenfolge geordnet. Die *case*-Anweisungen haben also das Aussehen:

case *e* **of**
  0 : st_0 ;
  1 : st_1 ;
  ⋮
  k : st_k
end

Eine Möglichkeit, *case*-Anweisungen zu übersetzen, benutzt einen **indizierten Sprung**, das ist ein Sprung, bei dem die angegebene Zieladresse um einen unmittelbar vorher berechneten Wert erhöht wird. Die **ixj**-Instruktion der P-Maschine, siehe Tabelle 2.5, addiert zur Zieladresse den Wert in der obersten Kellerzelle hinzu.

Tabelle 2.5: indizierter Sprung

Befehl	Bedeutung	Bed.	Erg.
ixj q	$PC := STORE[SP] + q$; $SP := SP - 1$	(i)	

Eine graphische Darstellung der für *case*-Anweisungen erzeugten Befehlsfolgen sieht aus wie in Abb. 2.5.

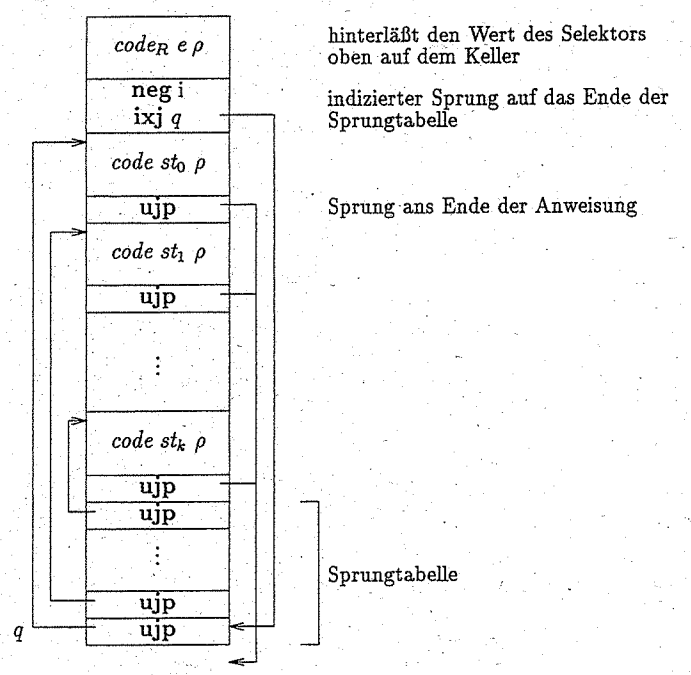


Abb. 2.5: Codegenerierung für case-Anweisung

Beachten Sie, daß in diesem Schema der Wert des Selektorausdrucks negiert wird. Im ixj-Befehl wird zu der Adresse q des letzten unbedingten Sprungs in der Sprungtabelle dieser negative Wert addiert. Deshalb sind die Sprünge in der Sprungtabelle in umgekehrter Reihenfolge der Komponenten angeordnet. Der letzte Sprung führt auf die Adresse der Befehlsfolge für st_0, der erste auf die für st_k. Dies wurde so gemacht, um eine rekursive Spezifikation der Übersetzung zu ermöglichen (siehe Übung 4.2).

2.5 Speicherbelegung für Variablen einfachen Typs

In diesem Abschnitt führen wir einige sehr wichtige Begriffe des Übersetzerbaus ein, nämlich die Begriffe **Übersetzungszeit** und **Laufzeit, statisch** und **dynamisch**. Zur **Übersetzungszeit** wird ein vorgelegtes Pascal-Programm in ein P-Programm übersetzt. Zur **Laufzeit** wird dieses erzeugte P-Programm mit Eingabedaten ausgeführt. **Statisch** sind alle Informationen über ein Pascal-Programm, die zur Übersetzungszeit allein aus diesem Programm ersichtlich sind oder aus ersichtlicher Information berechnet oder erzeugt werden können. **Dynamisch** sind all die Informationen, die erst zur Laufzeit durch das Ausführen des erzeugten P-Programms mit Eingabedaten verfügbar werden.

Wir haben schon einige Beispiele für statische und für dynamische Informationen über Pascal-Programme kennengelernt. Statisch etwa sind die Zieladressen von bedingten oder unbedingten Sprüngen, denn sie werden schließlich mithilfe der *code*-Funktion aus dem Quellprogramm berechnet. Dies gilt natürlich für das ganze für ein Pascal-Programm erzeugte P-Programm. Also ist dieses auch statisch. Dynamisch sind i.a. die Werte von Variablen, also auch die Werte von Ausdrücken, in denen Variablen auftreten. Diese Werte hängen i.a. von Eingabewerten des Programms ab, welche erst zur Laufzeit zur Verfügung stehen. Da also die Werte der Bedingungen in Anweisungen dynamisch sind, ist es auch dynamische Information, wie der Kontrollfluß nach der Auswertung der Bedingung ist.

Um die Zuordnung von im Quellprogramm deklarierten Variablen zu Speicherzellen vorzunehmen, müssen wir ein paar Annahmen über die Größe der Speicherzellen und unsere (Speicher-) Spendierfreudigkeit machen. Jeder Variablen der einfachen Typen integer, real, character, bool, sowie vom Aufzählungs- oder Mengentyp und Zeigervariablen werden wir eine Speicherzelle zur Aufnahme ihrer Werte zuordnen. Wir versuchen also nicht, wie es in realen Übersetzern geschieht, „kleine Werte" wie boolesche Größen oder Zeichen zu mehreren in ein Wort zu packen. Ebenfalls ignorieren wir temporär das Problem, daß Mathematiker, Physiker und Ingenieure häufig mit großer Genauigkeit, d.h. mit reellen Zahlen mit langer Mantisse rechnen möchten. Wir können also augenblicklich nur Programme übersetzen, in denen die Genauigkeit durch die Wortlänge der P-Maschine beschränkt ist. Diese Wortlänge ist dabei auch nicht festgelegt. Sie muß nur in der Lage sein, boolesche Werte, Zeichen eines hinreichend großen Alphabets und vor allen Dingen eine Instruktion pro Wort zu fassen.

Dann ergibt sich ein recht einfaches Schema zur Speicherbelegung. Den Variablen im Deklarationsteil des Programms – wir betrachten im Augenblick noch Programme ohne Blöcke und Prozeduren – ordnen wir in Reihenfolge ihres Auftretens konsekutive Adressen am Anfang des Kellerspeichers zu. Die erste zugeteilte Adresse ist aus später erklärlichen Gründen nicht 0 sondern 5. Die zugeteilten Adressen nennen wir aus Gründen, die ebenfalls klar werden, wenn wir Prozeduren und Blöcke betrachten, **relative Adressen**. Die (eigentlich) absolute Adresse 5 fassen wir also auf als die Adresse 5 relativ zur Basis 0. Die Funktion, die die Zuordnung von Variablen zu Relativadressen festhält, ist ρ: $Var \rightarrow \mathbb{N}_0$.

Betrachten wir den Variablendeklarationsteil eines Pascal-Programms, wobei die vorkommenden Typen alle einfach seien. Er hat dann die Form

var $n_1 : t_1; \ldots; n_k : t_k;$

Die oben geschilderte Speicherbelegungsstrategie würde dann die Funktion ρ folgendermaßen definieren.

$$\rho(n_i) = i + 5 \text{ für } 1 \leq i \leq k.$$

Man sieht leicht, daß die so zugeteilten relativen Adressen statische Größen sind; denn sie ergeben sich (auf sehr einfache Weise) aus dem Quellprogramm, nämlich aus den Positionen im Variablendeklarationsteil.

Diese Adressen liegen natürlich in dem Bereich des Speichers, den wir für den Keller der P-Maschine reserviert haben. Wenn Prozeduren und Funktionen behandelt werden, wird klar, daß wir eigentlich von zwei ineinander enthaltenen Kellern reden, nämlich einem „großen", der aus den Datenbereichen aller jeweils aktiven Prozeduren besteht und somit wächst bzw. schrumpft, wenn eine Prozedur betreten bzw. verlassen wird, und einem „kleinen" zu jeder aktiven Prozedur, der die während der Verarbeitung der Anweisungen des Rumpfes anfallenden Zwischenergebnisse aufnimmt. Nur mit diesem zweiten haben wir bisher Bekanntschaft gemacht, etwa bei der Verarbeitung von Wertzuweisungen. Die Zuteilung von Speicherzellen bzw. Adressen an deklarierte Variablen legt den Aufbau des Datenbereichs fest.

2.6 Speicherbelegung für Felder

Zuerst wird der Fall der statischen Felder, wie im ursprünglichen Pascal, behandelt, danach werden dynamische Felder, wie aus Algol60 oder ISO Pascal, betrachtet.

2.6.1 Statische Felder

Wieviel Speicherzellen wird ein Feld a belegen, wenn es wie folgt deklariert wird?

var a: **array**$[-5..5, 1..9]$ **of integer**

Jede Feldkomponente belegt nach unserer obigen Annahme eine Zelle. Offensichtlich gibt es 99 Komponenten,

$$\begin{array}{llll} a[-5,1], & a[-5,2], & \ldots & ,a[-5,9], \\ a[-4,1], & a[-4,2], & \ldots & ,a[-4,9], \\ \vdots & & & \\ a[5,1], & a[5,2], & \ldots & ,a[5,9], \end{array}$$

welche wir konsekutiv in dieser Reihenfolge im Speicher ablegen wollen.

Diese Art der Ablage heißt (in Anlehnung an Matrizen) **zeilenweise Ablage**. Sie ist durch das Prinzip „bei Durchlaufen der Feldkomponenten im Speicher variiert der letzte Index am schnellsten" intuitiv beschrieben. Präzise definiert ist

sie folgendermaßen: Sei ein k-dimensionales Feld gegeben durch die Deklaration:
var b : **array**$[u_1..o_1, \ldots, u_k..o_k]$ **of integer** (u_i, o_i ganzzahlige Konstanten)
Dann folgen im Speicher die beiden Feldkomponenten $b[i_1, \ldots, i_j, o_{j+1}, \ldots, o_k]$ und $b[i_1, \ldots, i_j + 1, u_{j+1}, \ldots, u_k]$ unmittelbar aufeinander, falls $u_j \leq i_j < o_j$ ist.

Bei obiger Deklaration des Feldes b belegen die Feldkomponenten $\prod_{i=1}^{k}(o_i - u_i + 1)$ Zellen in der Reihenfolge

$$\begin{array}{llll} b[u_1, \ldots, u_k], & b[u_1, \ldots, u_{k-1}, u_k + 1], & \ldots & ,b[u_1, \ldots, u_{k-1}, o_k], \\ b[u_1, \ldots, u_{k-1} + 1, u_k], & b[u_1, \ldots, u_{k-1} + 1, u_k + 1], & \ldots & ,b[u_1, \ldots, u_{k-1} + 1, o_k], \\ \vdots & & & \\ b[o_1, \ldots, o_{k-1}, u_k], & b[o_1, \ldots, o_{k-1}, u_k + 1], & \ldots & ,b[o_1, \ldots, o_{k-1}, o_k]. \end{array}$$

Die Adresse der ersten für das Feld belegten Zelle merken wir uns wieder in der Funktion ρ. Natürlich bekommen wir jetzt eine etwas kompliziertere Speicherbelegungsstrategie. Wir führen nämlich eine Funktion gr: *Typ* $\to \mathbb{N}$ ein, die die Größe eines Objekts dieses Typs angibt, d.h. die Anzahl der von der Variablen dieses Typs belegten Speicherzellen. Gemäß dem oben gesagten gilt:

$gr(t) = 1$ für Typen integer, real, char, bool, Zeiger und Aufzählungs- und Mengentypen.

$gr(b) = \prod_{i=1}^{k}(o_i - u_i + 1)$ für den Typ des oben deklarierten Feldes b.

Dann ergibt sich für einen Variablendeklarationsteil
var $n_1 : t_1; n_2 : t_2; \ldots; n_m : t_m;$
die Funktion ρ als

$$\rho(n_i) = 5 + \sum_{j=1}^{i-1} gr(t_j) \quad \text{für} \quad 1 \leq i \leq m.$$

Machen Sie sich an dieser Stelle klar, daß in dem bisher behandelten Fall statischer Felder die Grössen k, $u_1, u_2, \ldots, u_k$, $o_1, o_2, \ldots, o_k$ und entsprechend die davon abhängigen Funktionen gr und ρ zur Übersetzungszeit bekannt, d.h. aus einem vorliegenden Pascal-Programm abzulesen bzw. zu berechnen sind. Entsprechend könnten wir zum Abspeichern, sagen wir des Wertes 0, in die erste Zelle des obigen Feldes a die Befehlsfolge

ldc a $\rho(a)$;
ldc i 0;
sto i;

erzeugen, worin $\rho(a)$ eine Adresse ist, die zur Übersetzungszeit bekannt ist, da ρ statisch ist.

Interessant aber wird es, wenn wir etwa die Wertzuweisung $a[i, j] := 0$ übersetzen sollen, worin i und j integer-Variablen sind, welche ihren Wert erst zur Ausführungszeit des übersetzten Programms bekommen. Dann müssen wir

nämlich Befehle erzeugen, welche die aktuellen Werte $\overline{i}$ und $\overline{j}$ von i und j mithilfe der Adressen $\rho(i)$ und $\rho(j)$ laden und dann folgenden Ausdruck berechnen: $(\overline{i} - (-5)) * (9 - 1 + 1) + \overline{j} - 1 = (\overline{i} + 5) * 9 + \overline{j} - 1$.
Dieser Wert, addiert zur Anfangsadresse $\rho(a)$, ergibt dann die Adresse der Zelle $a[\overline{i}, \overline{j}]$. Bei diesem Vorgehen treffen wir also die Übersetzungszeitgrößen $-5, 1, 5$ und 9, die Grenzen des Feldes a, und $\rho(a)$, seine Anfangsadresse, an, dazu die Ausführungszeitgrößen $\overline{i}$ und $\overline{j}$, die Werte von i und j zur Zeit der Ausführung der Befehlsfolge für die Pascal-Anweisung $a[i, j] := 0$.

Betrachten wir unsere allgemeine Felddeklaration
var b: **array** $[u_1..o_1, u_2..o_2, \ldots, u_k..o_k]$ **of integer**

Seien $d_i = o_i - u_i + 1$ für $1 \leq i \leq k$ die **Spannen** in den einzelnen **Dimensionen** des Feldes b. Wie ergibt sich die Relativadresse der durch $b[i_1, i_2, \ldots, i_k]$ bezeichneten Komponente von b, relativ zur Anfangsadresse von b?

Seien $\overline{i_1}, \ldots, \overline{i_k}$ wieder die aktuellen Werte von $i_1, \ldots, i_k$ zur Ausführungszeit der entsprechenden Anweisung. So, wie wir die Feldkomponenten im Speicher ausgelegt haben, bringt uns der Wert von

$$(\overline{i_1} - u_1) * gr(\textbf{array}[u_2..o_2, \ldots, u_k..o_k] \textbf{ of integer})$$

an den Anfang des $(k-1)$-dimensionalen Unterfeldes, in welchem die adressierte Komponente liegt. Addieren wir $(\overline{i_2} - u_2) * gr(\textbf{ array}[u_3..o_3, \ldots, u_k..o_k]$ **of integer**), so führt uns das zum Anfang des richtigen $(k-2)$-dimensionalen Unterfeldes usw. Der Ausdruck

$$\begin{aligned} r = \; & (\overline{i_1} - u_1) * gr(\textbf{array}[u_2..o_2, \ldots, u_k..o_k] \textbf{ of integer}) + \\ & (\overline{i_2} - u_2) * gr(\textbf{array}[u_3..o_3, \ldots, u_k..o_k] \textbf{ of integer}) + \\ & \vdots \\ & (\overline{i_{k-1}} - u_{k-1}) * gr(\textbf{array}[u_k..o_k] \textbf{ of integer}) + \\ & (\overline{i_k} - u_k) \end{aligned}$$

berechnet also genau die gewünschte Adresse relativ zum Anfang des Feldes. Wenn wir die (zur Übersetzungszeit bekannten) Werte der gr-Ausdrücke einsetzen, erhalten wir

$$\begin{aligned} r = \; & (\overline{i_1} - u_1) * d_2 * d_3 * \ldots * d_k + \\ & (\overline{i_2} - u_2) * d_3 * d_4 * \ldots * d_k + \\ & \vdots \\ & (\overline{i_{k-1}} - u_{k-1}) * d_k + \\ & (\overline{i_k} - u_k). \end{aligned}$$

Ausmultiplizieren und Aufspalten nach $\overline{i_j}$- und u_j-Ausdrücken ergibt

$$\begin{aligned} r = \; & \\ h \; \{ \; & \overline{i_1} * d_2 * d_3 * \ldots * d_k + \overline{i_2} * d_3 * d_4 * \ldots * d_k + \ldots + \overline{i_{k-1}} * d_k + \overline{i_k} - \\ g \; \{ \; & (u_1 * d_2 * d_3 * \ldots * d_k + u_2 * d_3 * d_4 * \ldots * d_k + \ldots + u_{k-1} * d_k + u_k) \end{aligned} \tag{2.1}$$

Man sieht, daß im zweiten Teilausdruck nur Größen auftreten, die zur Übersetzungszeit bekannt sind. Diesen Ausdruck kann der Übersetzer also zu einer Konstanten d auswerten. Auch den ersten Teil können wir noch vereinfachen. Da die Spannen im Falle statischer Felder bekannt sind, können wir alle auftretenden Produkte von Spannen zur Übersetzungszeit auswerten. Dann bleibt nur noch eine Summe der Form $h = \sum_{j=1}^{k} \overline{i_j} \cdot d^{(j)}$ übrig, wobei

$$d^{(j)} = \prod_{l=j+1}^{k} d_l \quad \text{ist.}$$

Jetzt verallgemeinern wir diese Adreßberechnung ein letztes Mal, bevor wir das Übersetzungsschema für die Komponentenadressierung in Feldern angeben. Unsere Deklaration für das Feld b legte den Komponententyp als integer fest. Damit war der Speicherbedarf für jede Komponente eine Zelle. Lassen wir jetzt beliebige, auch nicht einfache Komponententypen t mit bekanntem Speicherbedarf von $gr(t)$ Zellen zu, so benötigt das Feld c, mit Deklaration
var c: **array** $[u_1..o_1, u_2..o_2, \ldots, u_k..o_k]$ **of** t
$d_1 * d_2 * \ldots * d_k * g$ Speicherzellen, wobei g für $gr(t)$ stehe. Die Relativadresse der Feldkomponente $c[i_1, \ldots, i_k]$, immer relativ zum Anfang des Feldes, ist dann $h * g - d * g$, wobei der Teilausdruck $d * g$ in unserem Fall der statischen Felder wieder durch den Übersetzer ausgerechnet werden kann.

Tabelle 2.6 gibt eine P-Maschinen-Instruktion an, mit der man sich schrittweise durch immer kleiner dimensionierte Unterfelder „fortschalten" kann. Der Parameter q läßt sich dabei für die Faktoren $g \cdot d^{(j)}$ gebrauchen.

Tabelle 2.6: Berechne indizierte Adresse. $STORE[SP-1]$ enthält eine „Anfangsadresse", $STORE[SP]$ den Index des selektierten Unterfeldes, q die Größe des Unterfeldes.

Befehl	Bedeutung	Bedin.	Erg.
ixa q	$STORE[SP-1] := STORE[SP-1] +$ $\quad STORE[SP] * q;$ $SP := SP - 1$	(a,i)	(a)

Außerdem benötigen wir weitere Befehle zur Arithmetik auf Adressen. Wir kommen hier mit Inkrementier- und Dekrementierbefehlen aus (siehe Tabelle 2.7).

Tabelle 2.7: Inkrementierung und Dekrementierung in Pascal sind definiert für alle Typen, welche eine *succ*-Funktion haben.

Befehl	Bedeutung	Bedin.	Erg.
inc Tq	$STORE[SP] := STORE[SP] + q$	(T) und Typ(q) = i	(T)
dec Tq	$STORE[SP] := STORE[SP] - q$	(T) und Typ(q) = i	(T)

Zur Übersetzung der Indexfolge benutzen wir eine Funktion $code_I$, die als zweiten Parameter die Komponentengröße hat. Das Übersetzungsschema für die Berechnung der Adresse der Feldkomponente $c[i_1,\ldots,i_k]$ bei Komponentengröße g und Anfangsadresse $\rho(c)$ ist:

$$
\begin{array}{lcl}
code_L\ c[i_1,\ldots,i_k]\ \rho & = & \textbf{ldc a}\ \rho(c); code_I\ [i_1,\ldots,i_k]\ g\ \rho \\
code_I\ [i_1,\ldots,i_k]\ g\ \rho & = & code_R\ i_1\ \rho; \textbf{ixa}\ g \cdot d^{(1)}; \\
 & & code_R\ i_2\ \rho; \textbf{ixa}\ g \cdot d^{(2)}; \\
 & & \quad\vdots \\
 & & code_R\ i_k\ \rho; \textbf{ixa}\ g \cdot d^{(k)}; \\
 & & \textbf{dec a}\ g \cdot d;
\end{array}
$$

Dieses Übersetzungsschema leidet allerdings unter dem gleichen Mangel wie das früher angegebene Schema für die *case*-Anweisung; es wird nicht geprüft, ob die Werte der Indexausdrücke $i_1,\ldots,i_k$ innerhalb des erlaubten Bereichs liegen. Das wollen wir jetzt ändern. Die in Tabelle 2.8 angegebene **chk**-Instruktion überprüft, ob der ausgerechnete Indexwert innerhalb der Grenzen liegt.

Tabelle 2.8: Überprüfung, ob der oberste Kellerwert zwischen p und q einschließlich liegt. Halt mit Fehlermeldung, wenn nicht.

Befehl	Bedeutung	Bedin.	Erg.
chk p q	**if** $(STORE[SP] < p)$ **or** $(STORE[SP] > q)$ **then** *error*("value out of range") **fi**	(i,i)	(i)

Das verbesserte Übersetzungsschema ist dann (mit $fel = (g; u_1, o_1, \ldots, u_n, o_n)$)

$code_I\ [i_1,\ldots,i_k]\ fel\ \rho =$
$\quad code_R\ i_1\ \rho; \textbf{chk}\ u_1\ o_1; \textbf{ixa}\ g \cdot d^{(1)};$
$\quad code_R\ i_2\ \rho; \textbf{chk}\ u_2\ o_2; \textbf{ixa}\ g \cdot d^{(2)};$
$\quad\vdots$
$\quad code_R\ i_k\ \rho; \textbf{chk}\ u_k\ o_k; \textbf{ixa}\ g \cdot d^{(k)};$
$\quad \textbf{dec a}\ g \cdot d;$

Beispiel 2.6.1 Setzen wir die Deklarationen

var i,j: **integer**;
$\quad a$: **array**[−5..5, 1..9] **of integer**

voraus, so ergeben sich $\rho(i) = 5$; $\rho(j) = 6$; $\rho(a) = 7$; für das Feld a ergeben sich die Spannen $d_1 = 11$, $d_2 = 9$, die Komponentengröße $g = 1$ und $d = -44$. Die Übersetzung von $a[i+1,j] := 0$ ist dann

$code(a[i+1,j] := 0)\ \rho =$

```
ldc a 7        ldc a 6
ldc a 5        ind i
ind i          chk 1 9
ldc i 1        ixa 1
add i          dec a −44
chk −5 5       ldc i 0
ixa 9          sto i
```

□

In diesem Abschnitt wurde gezeigt, wie statische Felder implementiert werden können. Fassen wir die Besonderheiten noch einmal zusammen: Die Grenzen, die Spannen, die Komponentengröße und die Relativadresse jedes Feldes sind zur Übersetzungszeit bekannt. Alle Ausdrücke, die – etwa bei der Indizierung – nur aus ihnen bestehen, können schon durch den Übersetzer ausgewertet werden. Die Ergebnisse werden zu Operanden der erzeugten Befehle. Jedes Feld wird zusammen mit den anderen deklarierten Variablen abgelegt. Die ihm zugeordnete Relativadresse ist die Relativadresse seiner ersten Zelle.

2.6.2 Dynamische Felder

Der ISO-Standard für Pascal beseitigt einen Entwurfsfehler in der ursprünglichen Sprache, indem er Felder als formale Prozedurparameter erlaubt, die sich in ihrer Größe an die Größe der aktuellen Parameter anpassen. Bei gegebener Prozedurdeklaration **procedure** p (**value** a : **array**$[u_1..o_1,\ldots,u_k..o_k]$ **of** *Typ*,) ergeben sich die Grenzen von a jeweils erst beim Aufruf der Prozedur p. Ähnlich ist es bei einer Deklaration **var** a : **array**$[u_1..o_1,\ldots,u_k..o_k]$ **of** *Typ*, wenn nicht alle u_i, o_i konstant sind, sondern globale Variable oder formale Parameter der Prozedur. a ist also in beiden Fällen ein **dynamisches Feld**; seine Grenzen, die Spannen in den einzelnen Dimensionen und damit seine Größe sind dynamisch. Nur die Dimension selbst ist noch statisch. Damit lassen sich sowohl die bisherige Art der Speicherablage als auch die Komponentenadressierung nicht mehr benutzen. Deshalb wollen wir dies für den Fall der dynamischen Felder neu behandeln.

Die Ablage von dynamischen Feldern im Speicher macht deshalb Probleme, weil spätestens, wenn es zwei davon gibt, eine statische Zuteilung von Anfangsadressen nicht mehr möglich ist. Da der von einem Feld benötigte Speicherplatz erst zum Zeitpunkt des Eintritts in die Prozedur bekannt ist, kann die Speicherreservierung erst dann erfolgen. Wie wir später bei der Behandlung von Prozeduren sehen werden, werden dynamische Felder im dynamischen Teil des Prozedurrahmens hinter dem statischen Teil abgelegt.

Wie aber können wir Befehlsfolgen für die Komponentenadressierung erzeugen, ohne zumindest die Anfangsadresse zu kennen? Nun, wenn wir die Anfangsadresse nicht kennen, so können wir doch statisch festlegen, wo wir sie zur Laufzeit ablegen werden. Dazu benutzen wir einen **Felddeskriptor**, dem wir eine statische Anfangsadresse und auch Größe zuordnen. Seine Größe hängt nur von der Dimension des Feldes ab. Seine erste Zelle ist für die Anfangsadresse des Feldes vorgesehen. Diese wird dort abgespeichert, wenn das Feld angelegt wird,

also bei Eintritt in die Prozedur, bzw. bei Verarbeitung der Felddeklaration. Der weitere Aufbau wird sich noch ergeben.

Wenden wir uns wieder dem Problem zu, Befehlsfolgen für die Adressierung der Feldkomponente $b[i_1,\ldots,i_k]$, jetzt in einem dynamischen Feld b : **array**$[u_1..o_1,\ldots,u_k..o_k]$ **of integer**, zu erzeugen. Die $u_1,\ldots,u_k,o_1,\ldots,o_k$ erhalten ihre Werte erst zur Laufzeit. Betrachten wir wieder die Formel zur Berechnung der Relativadresse von $b[i_1,\ldots,i_k]$ in Gleichung (2.1) auf Seite 20. Mit Ausnahme von k, der Dimension des Feldes, sind jetzt alle in (2.1) auftretenden Größen dynamisch. Deshalb müssen wir zur Laufzeit ihre Werte berechnen und statisch Platz für sie vorsehen. Dabei läßt sich allerdings ein Unterschied zwischen den $i_1,\ldots,i_k$ und den $u_1,\ldots,u_k$, $d_2,\ldots,d_k$ feststellen. Die Werte der $i_1,\ldots,i_k$ sind abhängig von der jeweiligen Feldindizierung, während die Werte der $u_1,\ldots,u_k$ und der $d_2,\ldots,d_k$ sich einmal für die ganze Lebensdauer ergeben. Im Falle des formalen dynamischen Feldparameters werden die Werte der $u_1,\ldots,u_k$, $d_2,\ldots,d_k$ bei der Parameterübergabe vom aktuellen Parameter übernommen. Im Falle des deklarierten dynamischen Feldes werden die Werte der Grenzen und der $d_2,\ldots,d_k$ einmal bei der Anlage des Feldes berechnet. Der ganze zweite Teil der Gleichung (2.1), nämlich $(u_1 * d_2 * d_3 * \ldots * d_k + \ldots)$ ergibt für alle Feldindizierungen den gleichen Wert d, und dieser wird deshalb bei der Parameterübergabe übergeben bzw. beim Anlegen des Feldes einmal berechnet. Um uns auch noch die bei jeder Indizierung fällige Subtraktion von $d \cdot g$ zu ersparen, ziehen wir $d \cdot g$ gleich von der Anfangsadresse des Feldes ab; wir erhalten die Adresse der fiktiven Komponente $b[0,\ldots,0]$, die **fiktive** Anfangsadresse, und speichern sie statt der tatsächlichen Anfangsadresse im Felddeskriptor ab.

Nun muß zur Adressierung der Komponente $b[i_1,\ldots,i_k]$ nur noch zu dieser fiktiven Anfangsadresse der Wert von $h \cdot g$ mit $h = \overline{i_1} * d_2 * d_3 * \ldots * d_k + \overline{i_2} * d_3 * d_4 * \ldots * d_k + \ldots + \overline{i_{k-1}} * d_k + \overline{i_k}$ addiert werden. Diesen Ausdruck berechnen wir, um Multiplikationen zu sparen, mithilfe eines Hornerschemas

$$h = (\ldots((\overline{i_1} * d_2 + \overline{i_2}) * d_3 + \overline{i_3}) * d_4 + \ldots) * d_k + \overline{i_k}.$$

Zur Auswertung müssen wir auf die Werte der d_i zugreifen können. Deshalb sehen wir für sie Platz im Felddeskriptor vor. Er hat jetzt die in Abbildung 2.6 gezeigte Struktur.

Die zweite und dritte Zelle werden für das Kopieren von Feldern benötigt. Den Inhalt der dritten Zelle braucht man, um die fiktive Anfangsadresse der Kopie zu berechnen. Die Werte der Unter- und Obergrenzen wiederum sind für den Test auf das Einhalten der Bereichsgrenzen erforderlich.

Für die Indizierung in dynamischen Feldern benutzen wir zwei neue *code*-Funktionen, $code_{Ld}$ und $code_{Id}$. Dabei wird $code_{Ld}$ lediglich einen **ldc**-Befehl erzeugen, der die Adresse des Felddeskriptors oben auf den Keller lädt. Der von $code_{Id}$ erzeugte Code dupliziert diese Adresse, benutzt das obere Duplikat, um die fiktive Anfangsadresse (mit einem Indirektionsbefehl) zu laden, und greift auf die weiteren Inhalte des Felddeskriptors indirekt mithilfe des unteren Duplikates zu. Nach der Berechnung der Adresse wird das untere Duplikat nicht mehr

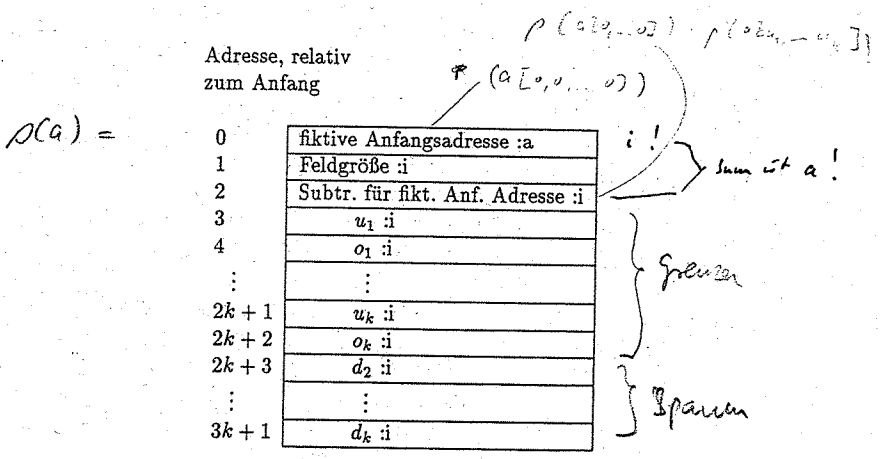


Abb. 2.6: Felddeskriptor für k-dimensionales Feld b mit Typindikatoren

benötigt und deshalb vom Keller entfernt, indem die Ergebnisadresse an seine Stelle geschoben wird. Im Augenblick könnten wir zwar noch die einzelnen Zellen des Felddeskriptors eines Feldes b durch $\rho(b)+j$ statisch adressieren, müßten dann aber später ein weiteres Codeschema angeben, wenn dies nicht mehr der Fall sein wird.

$code_{Ld}\ b[i_1,\ldots,i_k]\ \rho =$
  ldc a $\rho(b)$; Deskriptoradresse
  $code_{Id}\ [i_1,\ldots,i_k]\ g\ \rho$ g (statische) Komponentengröße

$code_{Id}\ [i_1,\ldots,i_k]\ g\ \rho\ =$
  dpl a; obersten Kellereintrag duplizieren
  ind a; fiktive Anfangsadresse
  ldc i 0;
  $code_R\ i_1\ \rho$; **add** i; **ldd** $2k+3$; **mul** i;
  $code_R\ i_2\ \rho$; **add** i; **ldd** $2k+4$; **mul** i;
  ⋮
  $code_R\ i_{k-1}\ \rho$; **add** i; **ldd** $3k+1$; **mul** i;
  $code_R\ i_k\ \rho$; **add** i;
  ixa g;
  sli a

Die neu eingeführten Instruktionen **dpl**, **ldd** und **sli** sind in Tabelle 2.9 definiert. **dpl** kopiert den obersten Kellereintrag, **ldd** greift indirekt auf Deskriptorfelder zu und **sli** schiebt den obersten Kellereintrag auf den zweitobersten.

Tabelle 2.9:

Befehl	Bedeutung	Bedin.	Erg.
dpl T	$SP := SP + 1;$ $STORE[SP] := STORE[SP-1]$	(T)	(T,T)
ldd q	$SP := SP + 1;$ $STORE[SP] := STORE[STORE[SP-3]+q]$	(a, T_1, T_2)	(a, T_1, T_2, i)
sli T_2	$STORE[SP-1] := STORE[SP];$ $SP := SP - 1$	(T_1, T_2)	(T_2)

Der allgemeine Fall dynamischer Felder beliebigen, auch dynamischen Typs wird in einer Übung behandelt, ebenso wie die notwendige Einführung der Bereichsprüfung.

2.7 Speicherbelegung für Verbunde

Jetzt werden wir das Problem der Speicherzuordnung und Adressierung von Verbunden (Records) behandeln. Wir vereinfachen gegenüber Pascal-Records etwas, indem wir keine Varianten zulassen und fordern, daß auch die Namen von Verbundkomponenten nicht mehrfach, etwa außerhalb des Verbundtyps, deklariert werden dürfen. Letzteres erlaubt es uns, unsere Funktion ρ auch für Namen von Verbunden und ihren Komponenten zu verwenden. Sei etwa die Verbundvariable v deklariert durch

var v : **record** a : **integer**; b : **bool end**.

Dann ordnen wir der Variablen v die Adresse der ersten freien Speicherzelle zu, wie es die bisherige Strategie festlegt, und den Komponentennamen, – aus Gründen, die im nächsten Abschnitt klar werden – Relativadressen innerhalb des Verbundes, also hier 0 und 1. Folgt die obige Deklaration also auf die Deklarationen **var** i, j : **integer**, so ergibt sich ρ zu $\rho(i) = 5$, $\rho(j) = 6$, $\rho(v) = 7$, $\rho(a) = 0$, $\rho(b) = 1$.

Die Berechnung der Relativadressen von Verbundkomponenten benutzt dabei die gr-Funktion analog zur Berechnung von Relativadressen in Deklarationsteilen. Allgemein ergibt sich also für eine Verbunddeklaration **var** v : r mit **type** r = **record** $c_1 : t_1; c_2 : t_2; \ldots; c_k : t_k$ **end**

$$\rho(c_i) = \sum_{j=1}^{i-1} gr(t_j)$$

Die Größe der Verbundvariablen v ergibt sich induktiv aus der Größe der Komponenten.

$$gr(r) = \sum_{i=1}^{k} gr(t_i)$$

Beachten Sie, daß diese Größe z.B. in Pascal statisch, d.h. zur Übersetzungszeit bekannt ist. Damit können wir ohne weiteres jetzt schon mit dem im letzten

Abschnitt Gelernten ein Feld von Verbunden behandeln und Befehlssequenzen zur Berechnung der Anfangsadresse einer Komponente in einem solchen Feld erzeugen.

Stellen wir uns umgekehrt vor, daß eine Verbundkomponente ein dynamisches Feld ist. Auch in diesem Fall möchten wir die statische Adressierung aller Verbundkomponenten retten. Das erreichen wir dadurch, daß wir in dem Verbund wieder nur den Deskriptor des Feldes ablegen und nicht das Feld selbst.

Die Adressierung von Verbundkomponenten geschieht in folgenden Schritten:

Laden der Anfangsadresse des Verbundes;

Erhöhung der Adresse um die Relativadresse der Komponente.

Damit hat man die Adresse, also einen verwendbaren L-Wert. Benötigt man den Wert der (einfachen) Komponente, so muß man ihn unter der Benutzung der Adresse (indirekt) laden. Ist sogar die adressierte Komponente wieder ein Verbund oder ein Feld, so kann auf der Basis der errechneten Adresse weiter eine Verbundkomponente bzw. eine Feldkomponente selektiert werden.

Zur Erhöhung um eine Relativadresse verwenden wir den in Tabelle 2.7 eingeführten **inc**-Befehl. Zum Adressieren der Komponente c_i im Verbund v wird die folgende P-Befehlsfolge erzeugt:

ldc a $\rho(v)$; **inc** a $\rho(c_i)$

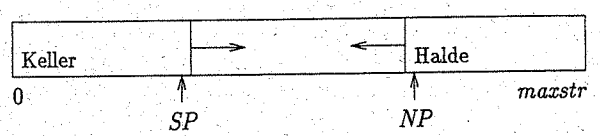
2.8 Zeiger und dynamische Speicherbelegung

Zeiger und die dynamische Speicherbelegung von anonymen Objekten sind zwei eng verwandte Konzepte in imperativen Programmiersprachen. Bisher haben wir nur die Speicherbelegung für Objekte betrachtet, die durch eine Deklaration eingeführt werden. In der Deklaration wird ein Name für das Objekt angegeben, und diesem Namen wird eine statische (Relativ-) Adresse zugeordnet. Kennt eine Programmiersprache Zeiger, so kann man diese benutzen, um auf namenlose Objekte zuzugreifen; mithilfe von Zeigern kann man dynamisch wachsende und schrumpfende verkettete Strukturen realisieren, wobei die einzelnen Objekte nicht durch eine Deklaration, sondern durch die Ausführung einer entsprechenden Anweisung, z.B. dem Pascal-**new**, kreiert werden. Die Semantik sowohl von Pascal wie auch von C ist bezüglich der Lebensdauer dynamisch kreierter Objekte nicht sehr präzise. Natürlich kann die Implementierung einer Programmiersprache schon vor dem Ende der Lebensdauer den durch ein solches Objekt belegten Speicher wieder freigeben, ohne gegen die Semantik zu verstoßen, wenn sichergestellt ist, daß es dem laufenden Programm nicht möglich ist, auf das Objekt noch zuzugreifen. Den Prozeß der Freigabe von Speicher, der von unerreichbaren Objekten belegt ist, nennt man **Speicherbereinigung** (engl. garbage collection).

Wie schon erwähnt, wird im unteren Teil des Datenspeichers bei Prozedureintritt ein Datenbereich für allen lokalen Speicherbedarf der Prozedur (und für organisatorische Zwecke) angelegt und bei Verlassen der Prozedur wieder freigegeben. Diese kellerartige Speicherbelegung und Freigabe paßt nicht zu der

Lebensdauer dynamisch kreierter Objekte; auch eine Speicherbereinigung wird i.a. Speicher nicht kellerartig und nicht synchronisiert mit Prozeduraustritten wieder freigeben.

Deshalb werden dynamisch kreierte Objekte in einem Speicherbereich, genannt **Halde** (engl. heap) am oberen Ende des Speichers untergebracht. Die Halde wächst bei dynamischer Anlage eines Objekts nach unten, also in Richtung auf den Keller zu. Auf die unterste belegte Zelle der Halde zeigt ein neues Register der P-Maschine, der *NP* (new pointer). Der Speicher der P-Maschine hat damit das folgende Aussehen:



Die Kreation eines neuen Objekts auf der Halde geschieht mithilfe der P-Instruktion **new** (siehe Tabelle 2.10 und Abb. 2.7).

Sie erwartet oben auf dem Keller die Größe des zu kreierenden Objekts, darunter die Adresse des Zeigers auf das zukünftige Objekt. Indirekt durch diese Adresse wird im Zeiger die Anfangsadresse des neuen Objekts gespeichert.

Tabelle 2.10: Der **new**-Befehl

Befehl	Bedeutung		Bedin.	Erg.
new	**if**	$NP - STORE[SP] \leq EP$	(a,i)	
	then	*error* ("store overflow")		
	else	$NP := NP - STORE[SP]$;		
		$STORE[STORE[SP-1]] := NP$;		
		$SP := SP-2$		
	fi;			

Wenn das neue untere Haldenende berechnet ist, wird geprüft, ob Keller und Halde zusammenstoßen würden. Dazu wird *EP* (extreme stack pointer) benutzt. Dieses Register der P-Maschine zeigt jeweils, wie wir im nächsten Abschnitt sehen werden, auf die oberste Kellerzelle, auf die *SP* bei der Auswertung von Ausdrücken im Anweisungsteil jemals zeigen kann. Wie in Übung 3.3 gezeigt werden soll, läßt sich für jeden Ausdruck die maximale Anzahl der zu seiner Auswertung benötigten Kellerzellen zur Übersetzungszeit vorberechnen. Deshalb kann man *EP* jeweils bei Eintritt in eine Prozedur, nach Anlegen etwaiger dynamischer Felder, um diese statische Größe erhöhen.

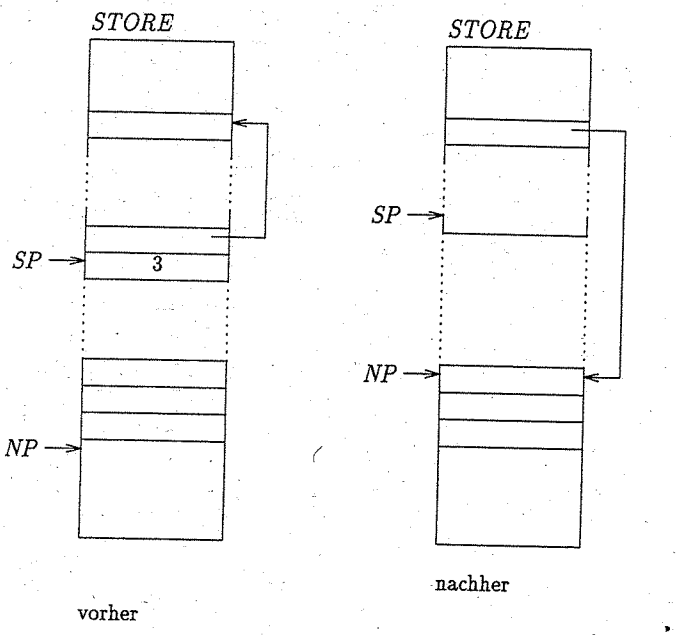


Abb. 2.7: Die Wirkung des **new**-Befehls der P-Maschine

Nur dann muß also überprüft werden, ob die Kellerverlängerung mit der Halde kollidieren würde. Sonst müßte bei jeder Erhöhung von *SP* dieser Test gemacht werden. Insofern kann die Benutzung des *EP*-Registers als eine Effizienzsteigerung gegenüber der prinzipiell auch möglichen Verwendung des *SP*-Registers betrachtet werden.

Jetzt können wir die Übersetzung einer Pascal-new-Anweisung in P-Maschinencode angeben, im wesentlichen natürlich in einen P-new-Befehl.

$code(\mathbf{new}(x))\ \rho\ =$
ldc a $\rho(x)$; **ldc** i $gr(t)$; **new** falls x Variable vom Typ $\uparrow t$ ist.

Zur **Dereferenzierung**, d.h. zur Adressierung eines Objekts über einen darauf weisenden Zeiger, verwendet man wieder den **ind**-Befehl.

Zum Abschluß der letzten drei Abschnitte betrachten wir das Problem der Adreßberechnung für beliebig kompliziert aufgebaute Bezeichnungen, das sind Bezeichnungen, die beliebig viele Indizierungen, Selektionen und Dereferenzierungen in beliebiger Reihenfolge enthalten. Es geht also um die Übersetzung

solcher Worte wie $x \uparrow [i+1,j].a \uparrow [i] \uparrow$ oder $y.a.b.c \uparrow [i,j+1].d$. Wir geben dafür einige neue Fälle der rekursiven *code*-Funktion an, die jeweils Adreßmodifikationen, d.h. Indizierungen, Selektionen bzw. Dereferenzierungen übersetzen. Zu beachten ist, daß die Funktion $code_{Id}$ zweistellig ist; sie bekommt außer der Liste der Indexausdrücke auch noch die (statisch feste) Feldkomponentengröße mit.
Das unten angegebene Übersetzungsschema zerlegt zusammengesetzte Bezeichnungen von links nach rechts, pflückt erst den führenden Namen, anschließend die darauf (evtl.) folgenden Selektionen, Dereferenzierungen bzw. Indizierungen ab.

$code_L(xr)\ \rho$	=	**ldc** a $\rho(x)$; $code_M(r)\ \rho$	für Namen x
$code_M(.xr)\ \rho$	=	**inc** a $\rho(x)$; $code_M(r)\ \rho$	für Namen x
$code_M(\uparrow r)\ \rho$	=	**ind** a; $code_M(r)\ \rho$	
$code_M([i]r)\ \rho$	=	$code_{Id}\ [i]\ g\ \rho$; $code_M(r)\ \rho$	falls g die Komponentengröße des indizierten Feldes ist
$code_M(\varepsilon)\ \rho$	=	ε	

Für diese *code*-Funktionen gilt die folgende Invariante: Wurde bei der Übersetzung einer zusammengesetzten Bezeichnung uv der Präfix u in eine Befehlsfolge b übersetzt, d.h. $code_L(uv)\ \rho = b$; $code_M\ v\ \rho$, dann gilt zur Laufzeit: Die Ausführung von b berechnet

- die Adresse eines Felddeskriptors in die oberste Kellerzelle, wenn v mit einer Indizierung beginnt, und
- die Anfangsadresse der von u bezeichneten Variablen in die oberste Kellerzelle, sonst.

Beispiel 2.8.1
Seien die folgenden Deklarationen gegeben.

```
type  t =  record
             a : array[−5.. + 5, 1..9] of integer;
             b : ↑t
           end;
var   i, j : integer;
      pt   : ↑t;
```

Unter der Voraussetzung, daß $\rho(i) = 5, \rho(j) = 6, \rho(pt) = 7$ ist, wird dann die Variablenbezeichnung $pt \uparrow .b \uparrow .a[i+1,j]$ folgendermaßen übersetzt:

ldc a 7;	Lade Adresse von pt
ind a;	Lade Anfangsadresse von Verbund
inc a 99;	Berechne Anfangsadresse von Verbundkomponente b
ind a;	Dereferenziere Zeiger
inc a 0;	Anfangsadresse von Komponente a
$code_{Id}[i+1,j]\ 1\ \rho$	ähnlich wie in Beispiel 2.6.1

□

Damit ist sowohl die Speicherbelegung durch Verbunde als auch die Adressierung von Verbundkomponenten abgeschlossen. Beachten Sie, daß diese Übersetzung nur für korrekt zusammengesetzte Variablenbezeichnungen richtige Befehlssequenzen liefert. Es werden also bei der Übersetzung nicht die Kontextbedingungen überprüft, etwa daß x in $x \uparrow$ auch eine Zeigervariable, daß x in $x[i_1,\ldots,i_k]$ ein k-dimensionales Feld, bzw. daß x in $x.a$ eine Verbundvariable mit einer Komponente namens a ist.

2.9 Prozeduren

Zur Vorbereitung der Übersetzung von Prozeduren wollen wir die zugehörigen Konzepte, Begriffe und Probleme kurz aufbereiten.
Die **Deklaration einer Prozedur** besteht aus

- einem Namen, unter dem sie aufrufbar ist,
- der Spezifikation der formalen Parameter, welche die Ein-/Ausgabeschnittstelle bilden,
- einer Folge von (lokalen) Deklarationen und
- einem Anweisungsteil, dem **Rumpf**.

Handelt es sich um eine Funktionsprozedur, so kommt noch die Angabe des Ergebnistyps hinzu.

Prozeduren werden **aufgerufen**, d.h. aktiviert, wenn ein Vorkommen ihres Namens im Anweisungsteil des Hauptprogramms oder einer Prozedur abgearbeitet wird. Eine aufgerufene Prozedur kann ihrerseits wieder eine andere Prozedur oder auch sich selbst aufrufen. Hat eine aufgerufene Prozedur ihren Anweisungsteil vollständig abgearbeitet, so wird sie „verlassen"und ihr Aufrufer, d.h. die Prozedur, die sie aktiviert hat, fährt in der Ausführung hinter dem Aufruf fort.

Betrachtet man die Ketten der Prozeduraufrufe, die während der Ausführung eines Programms entstehen, so bilden sie einen geordneten Baum, den **Aufrufbaum** des Programmlaufs. Die Wurzel des Aufrufbaums ist markiert mit dem Namen des Hauptprogramms. Jeder innere Knoten im Aufrufbaum ist markiert mit einem Prozedurnamen p[1], sein direkter Vorgänger mit dem Namen der Prozedur, welche diesen Aufruf von p ausgeführt hat; seine direkten Nachfolger bilden eine Liste von Prozeduren, geordnet in der Reihenfolge ihrer Aufrufe durch p.

[1] genauer mit einem (von evtl. mehreren) definierenden Vorkommen von p

Die Markierung p kann mehrfach im Aufrufbaum auftreten; genauer gesagt, sie tritt so oft auf, wie p im Laufe der Programmabarbeitung aufgerufen wurde. Jedes Auftreten von p nennen wir eine **Inkarnation** von p. Sie ist charakterisiert durch den Weg von der Wurzel des Baums – sie entspricht dem Hauptprogramm – bis zu diesem Knoten. Diesen Weg nennen wir den **Inkarnationsweg** dieser Inkarnation von p.

Betrachten wir den Zustand der Programmausführung, wenn eine bestimmte Inkarnation von p aktiv ist. Nach dem oben Gesagten sind alle Vorfahren dieser Inkarnation, also alle Knoten auf dem Inkarnationsweg, bereits aufgerufen aber noch nicht verlassen worden, bzw. im Falle des Hauptprogramms, gestartet, aber noch nicht beendet worden. Wir sagen, alle diese Inkarnationen sind zu diesem Zeitpunkt **lebendig**.

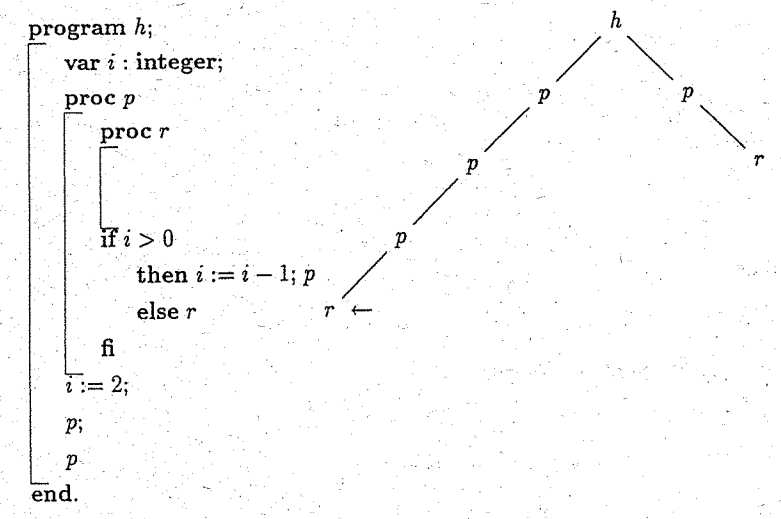


Abb. 2.8: Ein Programm und sein Aufrufbaum. Die Klammern umfassen jeweils den Deklarations- und den Anweisungsteil der Prozedur

Beispiel 2.9.1
Die Abb. 2.8 zeigt ein Programm und seinen Aufrufbaum. Der Pfeil zeigt auf eine Inkarnation von r, die nach den ersten drei rekursiven Aufrufen von p erreicht wird. Zu diesem Zeitpunkt leben das Hauptprogramm h, drei Inkarnationen von p und eine von r. □

Machen Sie sich an dieser Stelle klar,

- daß es zu einem Programm mehr als einen Aufrufbaum geben kann, und
- daß es auch unendliche Aufrufbäume gibt.

In einem Programm kann ein Name mehrmals vorkommen. Wir unterscheiden zwischen **definierenden Vorkommen** (binding occurrence) eines Namens, das sind diejenigen Vorkommen, bei denen ein Name in einer Deklaration definiert oder in einer formalen Parameterliste spezifiziert wird, und den andern, den sogenannten **angewandten Vorkommen** (applied occurrence).

Betrachten wir nun die in Prozeduren auftretenden Namen. Die als formale Parameter oder durch lokale Deklarationen eingeführten Namen nennen wir **lokale Namen**[2]. Wird eine Prozedur aufgerufen, so werden für alle lokalen Namen neue **Inkarnationen** kreiert. Dazu wird – entsprechend der Spezifikation bzw. Deklaration – Platz für einfache Variablen, Felder oder Verbunde angelegt und, im Falle von Parametern, gemäß den aktuellen Parametern besetzt. Die **Lebensdauer** der so kreierten Inkarnationen ist gleich der Lebensdauer der Prozedurinkarnation, d.h. der durch sie belegte Platz kann bei Verlassen der Prozedur wieder freigegeben werden[3]. Man sieht leicht, daß dies mit einer kellerartigen Speicherverwaltung realisiert werden kann. Bei Prozedureintritt wird Speicher für die formalen Parameter, die lokal deklarierten Variablen und für anfallende Zwischenergebnisse (siehe Abb. 2.10) belegt und bei Prozedurverlassen wieder freigegeben. Die Details werden wir später behandeln.

Nicht ganz so einfach ist die Behandlung von angewandt auftretenden Namen, die nicht lokal sind[4]. Wir bezeichnen sie als **globale Namen**, also als Namen, die zur betrachteten Prozedur global sind. Die **Sichtbarkeits-** und/oder **Gültigkeitsregeln** der Programmiersprache legen fest, wie das zu einem angewandten Auftreten eines Namens korrespondierende definierende Auftreten gefunden wird. Die umgekehrte, aber äquivalente Sicht geht von einem definierenden Vorkommen eines Namens aus und legt fest, in welchem Programmstück alle angewandten Vorkommen des Namens sich auf dieses definierende Vorkommen beziehen.

Aus Algol-ähnlichen Sprachen kennen wir folgende Sichtbarkeitsregel: Ein definierendes Auftreten eines Namens ist sichtbar in der Programmeinheit , in deren Deklarations- oder Spezifikationsteil die Definition steht, abzüglich aller von dieser Programmeinheit echt umfaßten Programmeinheiten, die eine neue Definition des Namens enthalten. Dabei steht „Programmeinheit“ für Prozedur und/oder Block.

Ausgehend von dieser gegebenen Sichtbarkeitsregel wollen wir die beiden obigen Sichten noch einmal aufgreifen. Suchen wir zu einem angewandten Vorkommen das zugehörige definierende Vorkommen, so finden wir es offensichtlich, wenn

[2]Im Abschnitt über funktionale Programmiersprachen werden sie uns als **gebundene** Namen wiederbegegnen.

[3]Wir ignorieren in dieser Einführung lokale Variablen, die gemäß einer Zusatzspezifikation (own in Algol60, STATIC in PL/I und in C) ihre Prozedurinkarnationen überleben.

[4]In funktionalen Programmen heißen sie **frei** vorkommende Namen.

wir die Suche in dem Deklarationsteil der Programmeinheit beginnen, in der das angewandte Vorkommen steht. Eine dort vorhandene Deklaration oder Spezifikation ist die gesuchte. Ist keine solche vorhanden, dann setzen wir die Suche in der direkt umfassenden Programmeinheit fort usw. Findet sich in allen umfassenden Programmeinheiten einschließlich des Hauptprogramms kein definierendes Vorkommen, so liegt ein Programmierfehler vor.

Die andere Sicht – ausgehend von einem definierenden Vorkommen – überstreicht gewissermaßen die enthaltende Programmeinheit und ordnet alle angetroffenen angewandten Vorkommen des Namens diesen definierenden Vorkommen zu. An den Grenzen von Programmeinheiten, die eine Neudefinition des gleichen Namens enthalten, wird der überstreichende Strahl abgeblockt. Eine differenziertere Betrachtung dieser Begriffe findet sich im Kapitel Semantische Analyse.

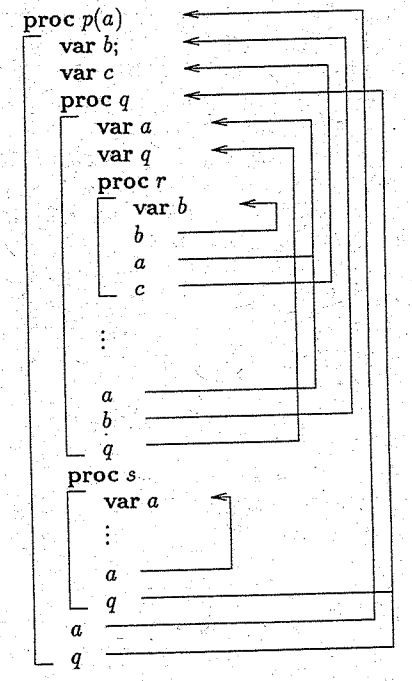


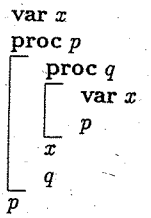
Abb. 2.9: Die Pfeile zeigen jeweils von angewandten Vorkommen auf die zugehörigen definierenden Vorkommen.

Beispiel 2.9.2
In Abbildung 2.9 wird ein Programm mit einigen geschachtelten Prozeduren und der Relation zwischen angewandten und definierenden Vorkommen von Variablen gezeigt. □

Durch die so beschriebene Sichtbarkeitsregel erhält man die sogenannte **statische Bindung**, d.h. globale Namen einer Prozedur werden definierenden Vorkommen in textlich umgebenden Prozeduren zugeordnet. Diese Zuordnung ist statisch, da sie nur auf dem Programmtext und nicht auf einer (dynamischen) Ausführung des Programms beruht. Jede Benutzung des globalen Namens zur Ausführungszeit trifft eine **Inkarnation des statisch zugeordneten definierenden Vorkommens.**

Im Gegensatz dazu besteht die **dynamische** Bindung[5] darin, daß ein Zugriff auf einen globalen Namen auf die letzte kreierte Inkarnation dieses Namens trifft, unabhängig davon, in welcher Prozedur er definierend auftrat.

Beispiel 2.9.3



1. Aufruf von p (außerhalb von p): Bei statischer und dynamischer Bindung bezieht sich das angewandte Vorkommen von x auf die äußere Deklaration.
2. Aufruf von p (in q): Bei statischer Bindung bezieht sich das angewandte Vorkommen von x auf die äußere Deklaration von x, bei dynamischer Bindung auf die Deklaration von x in q. Zu ihr gehört die letzte kreierte Inkarnation von x.□

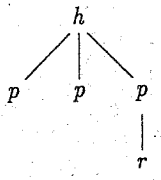
Der Unterschied zwischen statischer und dynamischer Bindung läßt sich gut anhand des Aufrufbaums verdeutlichen. Betrachten wir eine Inkarnation einer Prozedur p, welche auf eine globale Variable x zugreift. Bei dynamischer Bindung wird der aktuelle Inkarnationsweg angefangen bei der aktiven Inkarnation rückwärts durchlaufen, bis die erste Inkarnation einer Prozedur gefunden wird, welche eine Deklaration von x enthält.

Bei statischer Bindung wird auf dem Inkarnationsweg die letzte Inkarnation der innersten p umfassenden Prozedur gesucht, welche eine Definition von x

[5] Statische Bindung wird von allen Algol-ähnlichen Sprachen vorgeschrieben. Es gibt Lisp-Dialekte mit statischer, andere mit dynamischer Bindung.

enthält. Alle evtl. früher auf dem Inkarnationsweg vorkommenden Inkarnationen von x müssen „überschlagen" werden. Man sieht leicht ein, daß der Aufrufbaum zwar die Aufrufbeziehung zwischen Prozeduren widerspiegelt, aber wenig geeignet ist, effizient die richtige Inkarnation einer globalen Variablen zu finden. Bei genauer Betrachtung entdeckt man, daß jedem Inkarnationsweg bezüglich des Zugriffs auf globale Variablen eindeutig folgender Baum zugeordnet werden kann: Der direkte Vorgänger eines Knotens – also einer Prozedurinkarnation – in diesem Baum ist die letzte vor dieser kreierte Inkarnation der direkt umfassenden Prozedur. Diesen Baum nennen wir den Baum der **statischen Vorgänger** zu dem betrachteten Inkarnationsweg. Betrachten wir einen Weg von einer Inkarnation von p bis zur Wurzel in diesem Baum der statischen Vorgänger. Die für diese Inkarnation von p richtigen Inkarnationen von globalen Variablen liegen alle in einer der Inkarnationen auf diesem Weg. Nachfolgend betrachten wir nur noch die statische Bindung.

Beispiel 2.9.4
Der in Abb. 2.8 auftretende Inkarnationsweg $h-p-p-p-r$ hat den statischen Vorgängerbaum:



Denn alle Zugriffe auf globale Variable in allen Inkarnationen von p führen ins Hauptprogramm. Nur die Zugriffe aus r führen in die entsprechende Inkarnation von p. □

2.9.1 Speicherorganisation für Prozeduren

Machen wir uns noch einmal klar, daß

- alle zu einem Zeitpunkt lebenden Inkarnationen von Prozeduren (und dem Hauptprogramm) den aktuellen Inkarnationsweg bilden,
- beim Verlassen einer Prozedur ein Schritt von der aktuellen Inkarnation zu ihrem Vater im Aufrufbaum vorgenommen wird, und daß
- alle aus der aktuellen Inkarnation erreichbaren Variablen in Vorfahren der aktuellen Inkarnation im Baum der statischen Vorgänger zum aktuellen Inkarnationsweg liegen.

Die im folgenden vorgestellte Speicherorganisation, der sogenannte **Laufzeitkeller**, enthält zu jedem Zeitpunkt eine Folge von Speicherbereichen für die Menge der lebenden Inkarnationen, und zwar in der gleichen Reihenfolge, in der diese auf dem Inkarnationsweg auftreten. Um das Verlassen der Prozeduren effizient zu realisieren, sind die dynamischen Vorgänger miteinander verkettet. Um den Zugriff auf globale Variablen effizient zu realisieren, sind die statischen Vorgänger von „innen nach außen" verkettet. Der Laufzeitkeller enthält also gleichzeitig den aktuellen Inkarnationsweg und den aktuellen Baum der statischen Vorgänger. Dadurch wird zumindest das Verlassen von Prozeduren und der Zugriff auf globale Variablen unterstützt. Zu diesen beiden Zellen für Verweise kommen noch weitere organisatorische Zellen hinzu.

Im Speicher der P-Maschine wird für eine aufgerufene Prozedur, also eine Inkarnation, ein Kellerrahmen (engl. stack frame) angelegt, dessen Struktur in Abb. 2.10 dargestellt ist.

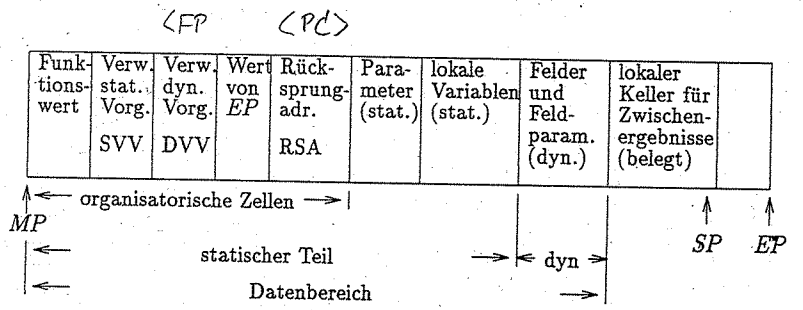


Abb. 2.10: Der oberste Kellerrahmen

Die Bestandteile eines Kellerrahmens und ihre Verwendung sind die folgenden:

- Eine Zelle für das Ergebnis, wenn es sich um eine Funktionsprozedur handelt. Dabei gehen wir davon aus, daß Ergebnisse von Funktionsprozeduren wie in Pascal nur einfachen Typ haben können.
- Eine Zelle, SVV, für einen Verweis auf den Kellerrahmen des statischen Vorgängers, also des direkten Vorgängers im statischen Vorgänger-Baum. Dies ist der Anfang der Kette, über die wir auf die richtigen Inkarnationen aller globalen Variablen zugreifen können. Diesen Verweis nennen wir im Folgenden den SV-Verweis.
- Eine Zelle, DVV, für einen Verweis auf den Kellerrahmen des dynamischen Vorgängers, das ist die Programmeinheit, welche die aktuelle Programmeinheit aktiviert hat. Dies ist der Anfang des aktuellen Inkarnationsweges,

auf dem alle lebenden Inkarnationen liegen. Dieser Verweis heißt jetzt DV-Verweis. In dieser Zelle wird, anders gesehen, der aktuelle Stand des *MP* gemerkt, damit dieser später restauriert werden kann.

- Eine Zelle, um sich den Stand von *EP* zu merken, damit dieser restauriert werden kann, wenn man aus einer aufgerufenen Prozedur in die aktuelle zurückkehrt.
- Die Rückkehradresse, RSA, für die aufrufende Programmeinheit, d.h. die Adresse des nächsten Befehls im Codespeicher, mit dem fortgefahren wird, wenn die aufgerufene Prozedur verlassen wird.

Diese fünf Zellen nennen wir die **organisatorischen Zellen**, weil sie dafür sorgen, daß die Prozeduranfangs- und -endorganisation und die globalen Zugriffe korrekt ablaufen.

Mit diesen organisatorischen Zellen beginnt der **Datenbereich** der Inkarnation. Er enthält außerdem noch

- Platz für die Werte, Adressen bzw. Deskriptoren der aktuellen Parameter.
- Platz für die lokalen Variablen der Prozedur und
- Platz für dynamische Felder.

Nach dem in den Abschnitten 2.5 - 2.7 Gesagten ist klar, daß außer dynamischen Feldern alle lokalen Variablen und Parameter von Pascal-Prozeduren statische Größe haben. Sie bilden zusammen mit den organisatorischen Zellen den statischen Teil des Datenbereichs der Prozedur. Ebenfalls wissen wir nach Abschnitt 2.6, daß für dynamische Felder und Feldparameter, die by-value übergeben werden, eine Kopie im Datenbereich angelegt werden muß, deren Platzbedarf erst dynamisch bekannt wird. Deshalb schließt sich an den statischen Teil des Datenbereichs ein dynamischer Teil an, in den die dynamischen Felder und Feldparameter der Parameterart by-value abgelegt werden. Ihre Größe ergibt sich bei Prozedureintritt bzw. der Übergabe des aktuellen Parameters. Im statischen Bereich stehen die Deskriptoren dieser Felder.
Als letztes folgt

- der lokale Keller, das ist der Keller, der uns in Abschnitt 2.3 bei der Auswertung von Ausdrücken begegnet ist. Wie man sich überlegen kann, ist auch seine maximale Länge statisch bestimmbar (siehe Aufgabe 3.3).

Die Register *MP* , *SP* und *EP* sind uns teilweise schon bekannt.

- *MP* (mark pointer) zeigt jeweils auf den Anfang des Kellerrahmens der aktuellen Inkarnation. Relativ zu dem Inhalt von *MP* werden die organisatorischen Zellen, die Zellen für die Parameter und die lokalen Variablen adressiert.
- *SP* zeigt auf die oberste belegte Zelle des lokalen Kellers. Bei leerem lokalem Keller zeigt *SP* auf die letzte Zelle des vorangehenden Bereichs.

- *EP* zeigt auf die oberste während der gesamten Ausführung der Prozedur belegte Zelle. *EP* wird dazu benutzt, eine eventuelle Kollision von Keller und Halde festzustellen. Deshalb muß der Test auf Kollision von Keller und Halde nicht bei jeder Erhöhung von *SP* gemacht werden.

Stellen wir uns vor, daß der Keller der *P*-Maschine während der Ausführung eines Programms von solchen Kellerrahmen belegt ist, und die Verweisketten entsprechend dem oben Gesagten gesetzt sind. Dann gilt: Die DV-Verweiskette verbindet die Kellerrahmen der Inkarnationen auf dem aktuellen Inkarnationsweg. Die SV-Verweise bilden den zu dem Inkarnationsweg gehörenden Baum der statischen Vorgänger.

Beispiel 2.9.5
Betrachten wir das Programm in Abb. 2.11(a) nach dem zweiten (rekursiven) Aufruf von *s*. Dann hat der Keller die in (b) gezeigte Gestalt. DV-Verweise sind links, SV-Verweise rechts von den Kellerrahmen gezogen. Der Baum der statischen Vorgänger ist in (c) gezeigt. □

2.9.2 Adressierung von Variablen

In Sprachen, die wie Pascal geschachtelte Sichtbarkeitsbereiche und dynamisch kreierte neue Inkarnationen von prozedurlokalen Namen kennen, können Variablennamen keine statischen, absoluten Speicheradressen mehr zugeordnet werden. Verschiedene Inkarnationen eines Namens werden i.a. verschiedene Speicherzellen belegen. Wie im vorigen Abschnitt beschrieben, wird für jede Prozedurinkarnation ein Kellerrahmen angelegt. Allen Größen, die im statischen Teil des Kellerrahmens abgelegt sind, ist eine statische Relativadresse relativ zum Anfang des Kellerrahmens zugeordnet. Alle Inkarnationen solcher Größen stimmen also, wenn schon nicht in ihren absoluten Adressen in *STORE*, so doch in ihren Relativadressen innerhalb der Kellerrahmen überein. Die Relativadressen werden zugeordnet, so wie das in den vorangehenden Abschnitten beschrieben wurde. Ein Zugriff auf eine lokale Variable läßt sich also realisieren, wenn man Befehle hat, die aus dem *MP*-Register die Anfangsadresse des aktuellen Kellerrahmens holen und die statische Relativadresse der Variablen darauf addieren. Dieses sind die später eingeführten Befehle **lod, lda** und **str.**

Jetzt wollen wir die Adressierung globaler Variablen behandeln. Dazu präzisieren wir den Begriff der **Schachtelungstiefe** (st) eines Programmkonstrukts.

Die Schachtelungstiefe des Hauptprogramms ist 0. Die Schachtelungstiefe eines definierenden (eines angewandten) Vorkommens eines Namens im Deklarations- bzw. Spezifikationsteil (Anweisungsteil) einer Programmeinheit mit der Schachtelungstiefe n ist $n + 1$.

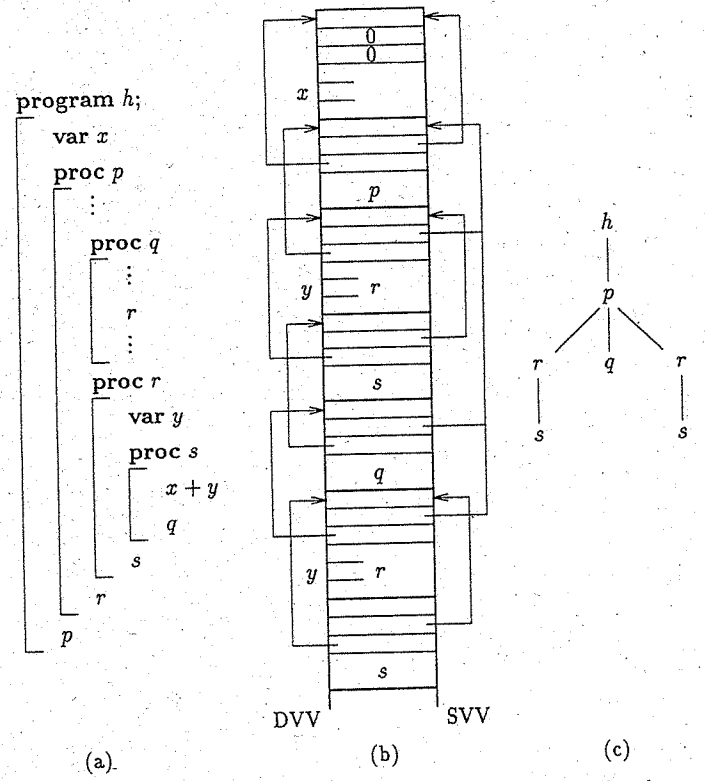


Abb. 2.11: Ein Programm (a), eine Kellerkonfiguration (b) und der dazu gehörige Baum der statischen Vorgänger (c)

Beispiel 2.9.6
Im Programm von Abbildung 2.11 sind die Schachtelungstiefen:

def.	Vorkommen	angew.	Vorkommen
p	1	p	1
q	2	q	4
r	2	r (in p)	2
s	3	r (in q)	3
		s	3

□

Zur Adressierung nichtlokaler Variablen aus einer Prozedurinkarnation wird der Verweis auf den statischen Vorgänger der Inkarnation benutzt. Wir gehen von folgender Invariante (ISV) aus:

(ISV) In jedem auf dem Keller für die Inkarnation einer Prozedur p angelegten Rahmen zeigt der SV-Verweis auf den Kellerrahmen der richtigen Inkarnation der p direkt umfassenden Programmeinheit. In Programmen ohne formale Prozeduren, also Prozeduren als Parameter, ist dies jeweils die jüngste noch lebende Inkarnation der p direkt umfassenden Programmeinheit. In Programmen mit Prozeduren ist dies nicht unbedingt der Fall. □

Betrachten wir jetzt zwei verschiedene Klassen von nichtlokalen Namen, Variablennamen und Prozedurnamen. Beim Zugriff auf nichtlokale Variablen, d.h. beim Berechnen ihrer Adresse, nutzen wir die Invariante (ISV) aus; beim Aufruf einer Prozedur (auch einer lokalen) nutzen wir (ISV) ebenfalls aus, um für den neu anzulegenden Kellerrahmen die Invariante (ISV) herzustellen.

Betrachten wir die Zugriffe auf die nichtlokalen Namen x und y aus der Prozedur s in Abbildung 2.11. Setzen wir voraus, das definierende Vorkommen von x im Hauptprogramm ist dort sichtbar. Dessen Schachtelungstiefe ist nach obiger Definition 1. Die Schachtelungstiefen der angewandten Vorkommen von x und y in s sind 4. Es ist in Abbildung 2.11 sichtbar, daß mit einem dreifachen Verfolgen der SV-Verweise der Kellerrahmen des Hauptprogramms, in dem x liegt, erreicht wird. Die richtige Inkarnation von y aus Prozedur r ist über einmaliges Verfolgen dieses Verweises zu erreichen; die Differenz der Schachtelungstiefen von angewandtem und definierendem Vorkommen ist 1. Tatsächlich ergibt sich, wenn wir (ISV) garantieren können, durch Induktion:

Ein Zugriff von einer Schachtelungstiefe n auf ein definierendes Vorkommen auf Schachtelungstiefe m mit $m \leq n$ erfordert ein $(n-m)$-maliges Verfolgen des SV-Verweises, um an den Kellerrahmen zu gelangen, in dem die richtige Inkarnation des Namens liegt. Auf dessen Anfangsadresse muß dann noch die (statische) Relativadresse addiert werden, um die Adresse der globalen Variablen zu erhalten. Lesende und schreibende Zugriffe und Adreßberechnung für globale Variablen werden mithilfe neuer P-Befehle realisiert, die in Tabelle 2.11 definiert sind. Zugriffe auf lokale Variable werden durch den Spezialfall $p = 0$ miterfaßt.

Nun betrachten wir die Sicherstellung der Invariante (ISV) beim Abarbeiten eines Prozeduraufrufs.

Tabelle 2.11: Laden und Speichern bei angegebener Differenz der Schachtelungstiefen und Relativadresse. **lod** lädt Werte, **lda** Adressen

Befehl	Bedeutung	Kommentar
lod T p q	$SP := SP + 1$; $STORE[SP] := STORE[base(p, MP) + q]$	p Differenz der Schachtelungstiefen q Relativadresse
lda p q	$SP := SP + 1$; $STORE[SP] := base(p, MP) + q$	
str T p q	$STORE[base(p, MP) + q] := STORE[SP]$; $SP := SP - 1$	

$$base(p, a) = \textbf{if } p = 0 \textbf{ then } a \textbf{ else } base(p-1, STORE[a+1])$$

Ist eine Prozedur q auf Schachtelungstiefe n deklariert, dann kann sie aufgerufen werden

1. im Anweisungsteil der sie direkt umfassenden Programmeinheit r; das wäre auf Schachtelungstiefe n,
2. in Prozeduren, deren Deklarationen (beliebig tief geschachtelt) von r umfaßt werden, wenn q nur nicht durch eine Deklaration verdeckt wird. Diese Aufrufe befinden sich auf einer Schachtelungstiefe größer n.

Abb. 2.12 zeigt Beispiele für diese Situationen: p ist jeweils aufrufende, q aufgerufene Prozedur. d sei die Differenz der Schachtelungstiefen von Aufruf und Deklaration der Prozedur.

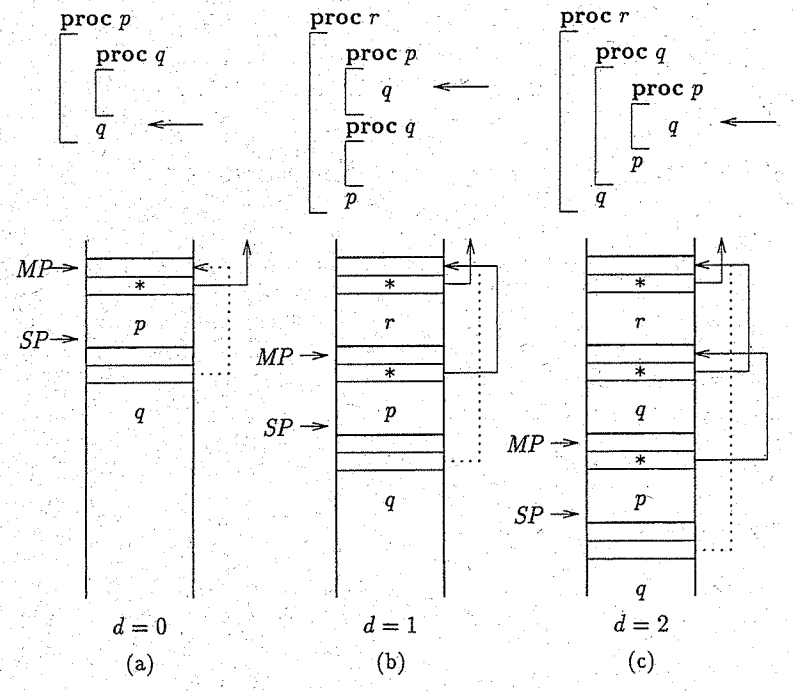


Abb. 2.12: Drei verschiedene Aufruf- und die Kellersituationen. Der aktuelle Aufruf ist jeweils durch den Pfeil markiert. Die gestrichelte Linie zeigt jeweils den neu zu setzenden SV-Verweis. $*$ = SV-Verweis

Betrachten wir die Situation, daß die Prozedur q in Prozedur p aufgerufen wird und daß die Differenz d der Schachtelungstiefen von Aufruf und Deklaration größer oder gleich 1 ist; d.h. wir schließen den Fall (a) von Abb. 2.12 momentan

aus. Der zu findende statische Vorgänger von q ist natürlich auch ein direkter oder indirekter statischer Vorgänger von p und deshalb über die Kette der SV-Verweise erreichbar. Wieweit müssen wir dieser Kette, angefangen mit dem Verweis im Kellerrahmen von p, nachgehen? d-mal, wie man sich leicht überlegen kann.

Im Falle (a) aus Abb. 2.12 wäre $d = 0$. Tatsächlich brauchen wir in diesem Fall dem statischen Verweis in p nicht nachzugehen, da der statische Verweis im Kellerrahmen von q auf den Kellerrahmen von p zeigen muß.

Dieser letzte Fall sorgt insbesondere dafür, daß die Invariante (ISV) bei Aufrufen von Prozeduren im Anweisungsteil des Hauptprogramms hergestellt werden kann. Diese Aufrufe können nur im Hauptprogramm deklarierte Prozeduren betreffen. Der SV-Verweis für diese muß auf den Kellerrahmen des Hauptprogramms zeigen. Auf diesen zeigt das MP-Register. Die Differenz der Schachtelungstiefen von Aufruf und Deklaration ist 0. Man muß also keinem SV-Verweis nachlaufen. Ein solcher existierte ja auch noch gar nicht.

Damit sollte intuitiv klar geworden sein, wie der Zugriff auf globale Namen, seien es Prozedur- oder seien es Variablennamen, realisiert wird. Der Zugriff auf globale Variablen wird formal in Abschnitt 2.9.6, das Setzen des Verweises auf den statischen Vorgänger bei Prozedureintritt in Abschnitt 2.9.4 beschrieben.

2.9.3 Berechnung der Adreßumgebungen

In Abschnitt 2.5 wurde beschrieben, wie man den in einem Deklarationsteil definierten Namen Speicherzellen bzw. Adressen zuordnen kann. Da wir dort nur Programme mit einem Deklarations- und einem Anweisungsteil, also ohne Prozeduren betrachteten, gab es das Problem der globalen Variablen noch nicht. Die in einem Deklarationsteil berechnete Adreßumgebung ρ ordnete Namen (Relativ-) Adressen zu. Wir müssen jetzt ρ und seine Berechnung in zweierlei Richtung erweitern:

ρ muß zu jedem definierenden Vorkommen eines **Variablennamens** außer einer Relativadresse auch noch seine Schachtelungstiefe enthalten; denn wie wir im vorhergehenden Abschnitt gesehen haben, erfordert die Adressierung globaler Variablen genau diese beiden Informationen.

Woran sollte ρ einen Prozedurnamen binden? Natürlich an alles, was man braucht, um Prozeduraufrufe zu übersetzen. Wie bei den Variablennamen gehört die Schachtelungstiefe der Prozedurdeklaration dazu. Außerdem muß der Übersetzer den Sprung an den Anfang der Übersetzung der Prozedur erzeugen, deshalb bindet ρ jeden Prozedurnamen an eine symbolische Marke, deren Wert die Anfangsadresse des Prozedurcodes sein wird.

Für **Adreßumgebungen** wie ρ führen wir deshalb den Bereich

$$Adr\_Umg = Name \rightarrow Adr \times ST \text{ mit } Adr = \mathbb{N}_0 \text{ und } ST = \mathbb{N}_0$$

ein. Variablennamen werden dabei an Relativadressen in Kellerrahmen und Prozedurnamen an CODE-Adressen gebunden.

Die Art der Berechnung von ρ muß dafür sorgen, daß an jeder Anwendungsstelle ρ genau die dort sichtbaren Namen kennt. Das können wir erreichen, indem wir bei der Verarbeitung des Deklarationsteils einer Prozedur in dem von außen

erhaltenen ρ die Einträge für solche Namen „überschreiben", die durch eine Neudeklaration verdeckt werden. Außerhalb der Prozedur müssen wir allerdings mit dem äußeren ρ weitermachen.

Die nun definierten Funktionen *elab_specs*, *elab_vdecls* und *elab_pdecls* zerlegen rekursiv Spezifikationslisten formaler Parameter und Variablen- bzw. Prozedurdeklarationsteile. Dabei konstruieren sie, ausgehend von einer äußeren Adreßumgebung ρ, die lokale Adreßumgebung für den Anweisungsteil der Prozedur. Treffen sie auf eine neue Deklaration eines außen bereits bekannten Namens, so wird dessen Eintrag in der Adreßumgebung durch den Neueintrag überschrieben. Damit ist der äußere Name verdeckt. Die Funktion $f' : A \to B$, die auf $a \in A$ den Wert $b \in B$ hat und sonst mit der Funktion $f : A \to B$ übereinstimmt, bezeichnen wir mit $f[b/a]$.

elab_specs nimmt eine Liste von Parameterspezifikationen, eine Adreßumgebung, eine nächste zu vergebende Relativadresse und eine Schachtelungstiefe und produziert eine neue Adreßumgebung und die Relativadresse der nächsten freien Zelle hinter dem Platz für die formalen Parameter. Die Funktionalität von *elab_specs* ist also

$$elab\_specs : Spec^* \times Adr\_Umg \times Adr \times ST \to Adr\_Umg \times Adr.$$

elab_specs (**var** $x : t$; *specs*) $\rho\ n\_a\ st =$
  elab_specs specs $\rho[(n\_a, st)/x]\ (n\_a + 1)\ st$
elab_specs (**value** x: **array**$[u_1..o_1, \ldots, u_k..o_k]$ **of** t'; *specs*) $\rho\ n\_a\ st =$
  elab_specs specs $\rho'\ (n\_a + 3k + 2)\ st$ *mit*
    $\rho' = \rho[(n\_a, st)/x][(n\_a + 2i + 1, st)/u_i]_{i=1}^{k}[(n\_a + 2i + 2, st)/o_i]_{i=1}^{k}$
elab_specs (**value** $x : t$; *specs*) $\rho\ n\_a\ st =$
  elab_specs specs $\rho[(n\_a, st)/x]\ (n\_a + gr(t))\ st$ für statische Typen t
elab_specs ( ) $\rho\ n\_a\ st = (\rho, n\_a)$

Für var-Parameter (Referenz-Parameter) wird eine Zelle reserviert, in der der Aufrufer den Verweis auf den aktuellen Parameter abspeichert. Für k-dimensionale value-Feldparameter wird ein Deskriptor aus $3k + 2$ Zellen vorgesehen. Darin liegen später die Anfangsadresse, die Feldgröße, der Subtrahend zur Berechnung der fiktiven Anfangsadresse, die Unter- und die Obergrenzen (für die Bereichsüberprüfung) und die Spannen. Man beachte, wie die formalen Grenzen, $u_1, \ldots, u_k, o_1, \ldots, o_k$ des formalen value-Feldparameters an ihre entsprechenden Positionen im Deskriptor gebunden werden. Der zweite Fall der statischen value-Parameter deckt die anderen Fälle ab. Der letzte Fall betrifft abgearbeitete Spezifikationslisten.

Es folgt die Definition der Funktion *elab_vdecls*. Sie besorgt die Speicherzuteilung für Variablendeklarationen. Hierbei müssen wir die Behandlung statischer und dynamischer Objekte unterscheiden. Die ersten drei Fälle behandeln die Deklaration statischer Variablen. Die Adressierung von Verbundkomponenten wird aus Abschnitt 2.7 übernommen.

Die Funktionalität von *elab_vdecls* ist

$$elab\_vdecls : Vdecl^* \times Adr\_Umg \times Adr \times ST \to Adr\_Umg \times Adr$$

elab_vdecls (**var** $x : t$; *vdecls*) $\rho\ n\_a\ st =$ für nicht-Feld-Typen t
  elab_vdecls vdecls $\rho[(n\_a, st)/x]\ (n\_a + gr(t))\ st$
elab_vdecls (**var** x : **array**$[u_1..o_1, \ldots, u_k..o_k]$ **of** t; *vdecls*) $\rho\ n\_a\ st =$

$$elab\_vdecls\ vdecls\ \rho[(n\_a, st)/x]\ \underbrace{(n\_a + 3k + 2)}_{\substack{\text{Platz für} \\ \text{den Deskriptor}}}\ + \underbrace{\prod_{i=1}^{k}(o_i - u_i + 1) \cdot gr(t))}_{\substack{\text{Platz für die} \\ \text{Feldkomponenten}}}\ st$$

falls x statisches Feld ist.
elab_vdecls ( ) $\rho\ n\_a\ st = (\rho, n\_a)$

Das ist die schon in Abschnitt 2.9.2 beschriebene Speicherzuordnung für lokale Variablen. Geändert hat sich lediglich die Speicherablage von Feldern. Für jedes deklarierte Feld wird ein Deskriptor angelegt; denn es könnte als aktueller Parameter einer Prozedur auftreten. In diesem Fall benötigen wir die im Deskriptor enthaltene Information. Bei statischen Feldern werden Deskriptor und Feld aufeinanderfolgend abgelegt. Die Anfangsadresse des Feldes sowie die Grenzen und die Spannen werden durch eine Befehlsfolge in den Deskriptor eingetragen. Die Erzeugung dieser Befehlsfolge wird einer Übung überlassen. Im Falle dynamischer Felder hat lediglich der Deskriptor eine statische Länge. Er wird im statischen Teil des Kellerrahmens abgelegt. Der Feldname wird an die Anfangsadresse des Deskriptors gebunden. Bei Eintritt in die Prozedur wird dann eine Befehlsfolge ausgeführt, die das Feld selbst im dynamischen Teil des Kellerrahmens ablegt und die Einträge im Deskriptor besetzt.

Die Funktion *elab_pdecls* verarbeitet Prozedurdeklarationsteile.

$$elab\_pdecls : Pdecl^* \times Adr\_Umg \times ST \to Adr\_Umg \times Code$$

elab_pdecls (**proc** $p_1(\ldots); \ldots;$
  $\vdots$
  proc $p_k(\ldots); \ldots;$) $\rho\ st =$

$(\rho',\quad l_1 : code\ (\mathbf{proc}\ p_1(\ldots); \ldots)\ \rho'\ st + 1;$
  $\vdots$
  $l_k : code\ (\mathbf{proc}\ p_k(\ldots); \ldots)\ \rho'\ st + 1)$

wobei $\rho' = \rho[(l_1, st)/p_1, \ldots, (l_k, st)/p_k]$

elab_pdecls ( ) $\rho\ st = (\rho, (\ ))$

Die Funktion *elab_pdecls* hat eine bisher nicht angetroffene Funktionalität; sie erzeugt als Ergebnis sowohl eine Adreßumgebung als auch die Übersetzung des Prozedurdeklarationsteils. Ein solcher kann simultan rekursive Prozeduren enthalten, wenn man nicht wie in Pascal forward-Deklarationen verlangt; dann trifft der Übersetzer auf Prozedurnamen, die er noch nicht kennt und muß Unterprogrammsprünge auf Adressen erzeugen, die noch nicht festliegen. Wir helfen uns wieder, indem wir symbolische Marken für die Übersetzung der Prozeduren

einführen, die Prozedurnamen daran binden und die Prozeduren in dieser Umgebung übersetzen[6]. Beachten Sie, daß alle Prozeduren in der gleichen Adreßumgebung übersetzt werden. Die im Innern der Prozeduren berechneten Adreßumgebungen werden außerhalb der Prozeduren nicht benutzt.

2.9.4 Prozedureintritt und Prozedurverlassen

Betrachten wir die wichtigen Aktionen bei der Implementierung von Prozeduren, den Aufruf und damit den Eintritt in die Prozedur und das Verlassen der Prozedur nach Abarbeitung ihres Rumpfes.

Zuerst betrachten wir den Aufruf der Prozedur. Sei p die gegenwärtig aktive Prozedur. Ihr Kellerrahmen ist der oberste auf dem Keller. Die Verweise im Kellerrahmen seien richtig gesetzt, d.h. der SV-Verweis zeigt auf die richtige Inkarnation der p direkt umfassenden Prozedur, der DV-Verweis auf die Inkarnation, aus der p aufgerufen wurde. Jetzt rufe p eine Prozedur q auf.

Welche Aktionen müssen durchgeführt werden, um die Abarbeitung von q starten zu können?

1. Der SV-Verweis auf den Rahmen der richtigen Inkarnation der q direkt umfassenden Prozedur muß gesetzt werden.
2. Der DV-Verweis auf den Anfang des Rahmens von p muß gesetzt werden.
3. Der aktuelle Stand des EP-Registers muß gerettet werden.
4. Die Werte bzw. Adressen der aktuellen Parameter müssen bestimmt und abgespeichert werden. Dabei müssen insbesondere für formale value-Feldparameter Deskriptoren angelegt werden. Die Kopien der zugehörigen aktuellen Parameter werden später angelegt. (Die Behandlung der Parameterübergabe verschieben wir noch etwas).
5. Das MP-Register muß hochgesetzt werden, damit es auf den Anfang des neuen Kellerrahmens zeigt.
6. Die Rückkehradresse muß abgespeichert werden.
7. Der Sprung auf die erste Instruktion der Übersetzung von q muß ausgeführt werden.
8. SP muß auf das obere Ende des vom statischen Teil der Parameter und der lokalen Variablen belegten Bereiches gesetzt werden.
9. Die Kopien der aktuellen value-Feldparameter müssen angelegt werden. Dabei wird die Größe jedes solchen Parameters mithilfe seines Deskriptors berechnet und das SP-Register um diese Größe hochgesetzt.

[6] Die gleiche Technik der „zirkulären Umgebung" verwendet man auch in funktionalen Sprachen zur Implementierung simultaner Rekursion, siehe Kapitel 3.

10. EP muß gesetzt werden, um den Platzbedarf für den lokalen Keller zu berücksichtigen. Dabei wird auf eine eventuelle Kollision von Keller und Halde geprüft.

Tabelle 2.12: Befehle für Prozeduraufruf und -Eintritt, **mst** (mark stack), **cup** (call user procedure), **ssp** (set stack pointer), **sep** (set extreme stack pointer).

Befehl	Bedeutung	Kommentar
mst p	$STORE[SP+2] := base(p, MP)$;	Verweis auf stat. Vorgänger
	$STORE[SP+3] := MP$;	Verweis auf dyn. Vorgänger
	$STORE[SP+4] := EP$;	Retten von EP
	$SP := SP + 5$	Ab $STORE[SP+1]$ können jetzt die Parameter ausgewertet werden.
cup p q	$MP := SP - (p+4)$;	p ist Speicherplatzbedarf für die Parameter
	$STORE[MP+4] := PC$;	Retten der Rücksprungadresse
	$PC := q$	Sprung zur Anfangsadresse q der Prozedur
ssp p	$SP := MP + p - 1$	p Größe des stat. Teils des Datenbereichs
sep p	$EP := SP + p$;	p max. Tiefe des lok. Kellers
	if $EP \geq NP$	Prüfe auf Kollision
	then *error* ("store overflow")	von Keller und Halde
	fi	

$$base(p, a) \;=\; \textbf{if } p \;=\; 0 \textbf{ then } a \textbf{ else } base(p-1, STORE[a+1])\textbf{fi}$$

Die Aktionen (1) bis (3) werden mithilfe der **mst**-Instruktion (mark stack) durch den Aufrufer ausgeführt. Die Parameterauswertung erfolgt natürlich auch durch den Aufrufer, denn die Variablen auf aktueller Parameterposition müssen in der Umgebung der Aufrufstelle ausgewertet werden. (5) bis (7) erledigt die **cup**-Instruktion (call user procedure), ebenfalls ausgeführt durch den Aufrufer. Danach nimmt die aufgerufene Prozedur ihre Arbeit auf. (8) führt sie mittels der **ssp**- (set stack pointer) Instruktion aus[7], (9) führt sie durch eine Initialisierungssequenz aus, die wir später behandeln, (10), die Erhöhung von EP, mittels des Befehls **sep** (set extreme stack pointer). Die verwendeten Befehle finden sich in Tabelle 2.12.

[7] In der orginal P-Maschine, die nur statische Felder kannte, wurden die Schritte (8) und (10) zusammen von dem **ent**-Befehl erledigt. Das Kopieren von statischen value-Feldparametern konnte später ausgeführt werden.

ent p q	$SP := MP + q - 1$;	q Größe des Datenbereichs
	$EP := SP + p$;	p max. Tiefe des lok. Kellers
	if $EP \geq NP$	Kollision von Keller
	then *error*(*"store overflow"*)	und Halde
	fi	

Tabelle 2.13: Rückkehr aus Funktionsprozedur und eigentlicher Prozedur.

Befehl	Bedeutung	Kommentar
retf	$SP := MP$;	Fktserg. liegt oben im lokalen Keller
	$PC := \text{STORE}[MP+4]$;	Rücksprung
	$EP := \text{STORE}[MP+3]$;	Restaurieren von EP
	if $EP \geq NP$	
	then *error*(*"store overflow"*)	
	fi	
	$MP := \text{STORE}[MP+2]$	DV-Verweis
retp	$SP := MP - 1$;	eigentliche Prozedur ohne Ergebnis
	$PC := \text{STORE}[MP+4]$;	Rücksprung
	$EP := \text{STORE}[MP+3]$;	Restaurieren von EP
	if $EP \geq NP$	
	then *error*(*"store overflow"*)	
	fi	
	$MP := \text{STORE}[MP+2]$	DV-Verweis

Ist der Rumpf einer Prozedur oder Funktion vollständig abgearbeitet, so muß der für sie angelegte Kellerrahmen wieder freigegeben werden und der Rücksprung in den Aufrufer ausgeführt werden. Dies besorgen die P-Befehle **retf** für Funktionsprozeduren bzw. **retp** für eigentliche Prozeduren (siehe Tabelle 2.13).

Damit sind wir in der Lage, bis auf die Parameterübergabe sowohl Prozedurdeklaration als auch Prozeduraufruf zu übersetzen.
Das Übersetzungsschema für eine Prozedurdeklaration ist:

$code$ (**procedure** p ($specs$); $vdecls$; $pdecls$; $body$) ρ st =

ssp n_a'';	Speicherbedarf statischer Teil
$code_S$ $specs$ ρ' st;	Speicherbedarf dynamischer Teil
$code_P$ $vdecls$ ρ'' st;	anlegen und initialisieren
sep k;	k max. Tiefe des lok. Kellers
ujp l;	
$proc\_code$;	Code für die lokalen Prozeduren
l : $code$ $body$ ρ''' st;	Code für Prozedurkörper
retp	

wobei $(\rho', n\_a') =$ $elab\_specs$ $specs$ ρ 5 st
$(\rho'', n\_a'') =$ $elab\_vdecls$ $vdecls$ ρ' n_a' st
$(\rho'''$, $proc\_code) =$ $elab\_pdecls$ $pdecls$ ρ'' st

Die Anwendung der $code_S$-Funktion auf den Spezifikationsteil behandelt eventuell spezifizierte value-Feld-Parameter. Für sie muß Code zum Kopieren des Feldes erzeugt werden (vgl. Seite 51). Für die im Variablendeklarationsteil vereinbarten Felder müssen ebenfalls Befehlsfolgen erzeugt werden, die die Deskriptoren besetzen und Platz für dynamische Felder durch Hochsetzen des SP-Registers reservieren (siehe Aufgabe 9.2).

Die Übersetzung der Deklaration einer Funktionsprozedur erzeugt statt der **retp**-Instruktion die **retf**-Instruktion. Diese hinterläßt das Ergebnis der Funktionsprozedur oben auf dem Keller.
Das Übersetzungsschema für einen Prozeduraufruf $p(e_1, \ldots, e_k)$ ist:

$$
\begin{aligned}
code\ p(e_1,\ldots,e_k)\ \rho\ st \quad = \quad & \textbf{mst}\ st - st'; \\
& code_A\ e_1\ \rho\ st; \\
& \vdots \\
& code_A\ e_k\ \rho\ st; \\
& \textbf{cup}\ s\ l
\end{aligned}
$$

wobei $\rho(p) = (l, st')$ ist und s der Platzbedarf für die aktuellen Parameter ist.

2.9.5 Parameterübergabe

Pascal, wie auch eine Reihe anderer imperativer Sprachen, kennt zwei verschiedene Arten von Parametern: *value*- und *var*-Parameter (Referenz-Parameter). Dazu kommen noch Prozeduren als Parameter, sogenannte **formale Prozeduren**.

Die Übersetzung eines aktuellen Parameters erfordert Informationen über den zugehörigen formalen Parameter. Solche Informationen stellt die semantische Analyse zur Verfügung, die vor der Codeerzeugung abläuft. Die für einen aktuellen Parameter erzeugten Befehlsfolgen laufen in folgendem Kontext ab: Die aufrufende Prozedur hat bereits den **mst**-Befehl ausgeführt; das Register SP zeigt bereits auf die letzte unterhalb des Parameterbereichs gelegene Zelle. Die Parameterübergabe belegt jetzt die nächsten Zellen mit Adressen, Werten bzw. Deskriptoren. Dabei wird SP jeweils erhöht, so daß es immer auf die letzte belegte Zelle zeigt.

Ist der formale Parameter y einer Prozedur als *var*-Parameter spezifiziert, so muß der zu y korrespondierende aktuelle Parameter x eine Variable oder ein formaler *var*-Parameter sein. Die Parameterübergabe besteht in diesem Fall darin, daß die Adresse, also der L-Wert, des aktuellen Parameters berechnet und unter der y zugeordneten Relativadresse abgelegt wird. Für einen Feldparameter ist der L-Wert die Adresse seines Deskriptors. Jeder Zugriff auf den formalen Parameter im Rumpf der Prozedur erfordert dann eine Indirektion durch seine Relativadresse (siehe Abschnitt 2.9.6).

$code_A\ x\ \rho\ st =$	falls der zu x korrespondierende formale Parameter y
$code_L\ x\ \rho\ st$	*var*-Parameter ist.

Die $code_L$-Funktion, ebenso die $code_R$-Funktion, werden im nächsten Abschnitt neu definiert.

Ist der formale Parameter y einer Prozedur p als *value*-Parameter skalaren Typs spezifiziert, so darf als zu y korrespondierender aktueller Parameter in einem Aufruf von p ein Ausdruck e dieses Typs auftreten. Bei diesem Aufruf geschieht dann folgendes: der Ausdruck e wird ausgewertet, so daß sein Wert im

Parameterbereich der aufgerufenen Prozedur unter der für y festgelegten Relativadresse liegt. y fungiert während der Abarbeitung der Prozedur als eine lokale Variable, die mit dem Wert des aktuellen Parameters initialisiert wurde.

$code_A\ e\ \rho\ st =$ falls der zu e korrespondierende formale Parameter y *value*-Parameter ist.
$\quad code_R\ e\ \rho\ st$

Aktuelle *value*-Parameter vom Verbundtyp werden behandelt, indem Platz für die zu ihnen korrespondierenden formalen Parameter reserviert wird und ihr Wert, das ist das strukturierte Objekt bestehend aus den Komponentenwerten, in den reservierten Speicherplatz kopiert wird. Dazu benutzt man die statische Variante der move-Befehle[8] (siehe Tabelle 2.14).

$code_A\ x\ \rho\ st =$ falls der zu x korrespondierende formale *value*-Parameter strukturierten Typs t und statischer Größe $gr(t) = g$ ist.
$\quad code_L\ x\ \rho\ st;$
$\quad$ **movs** g

Dieses letzte Schema kopiert z.B. Verbunde oder Felddeskriptoren. g ist dabei der Speicherplatzbedarf für den Verbund bzw. Deskriptor. Die Zieladresse ergibt sich aus dem Inhalt der obersten Kellerzelle.

Bei dynamischen *value*-Feldparametern spaltet sich die Behandlung, wie wir gesehen haben, in mehrere Teile auf;

- das Anlegen des Deskriptors, wobei alle Einträge – inklusive der Anfangsadresse – aus dem Deskriptor des aktuellen Parameters entnommen werden; dazu muß allerdings ein solcher Deskriptor (auch für statische Felder) existieren;
- das Kopieren des Feldinhalts,
- das Eintragen der fiktiven Anfangsadresse in den Deskriptor.

Diese Schritte wollen wir noch detaillierter behandeln und am Ende ein Übersetzungsschema für die Anlage von value-Feldparametern angeben. Den ersten Schritt, das Kopieren des Deskriptors des aktuellen Feldparameters in den Parameterbereich der aufgerufenen Prozedur, besorgt noch der Aufrufer. Er setzt voraus, daß ein solcher Deskriptor existiert, auch im Falle eines statischen Feldes. Wir erinnern uns, daß bei der Behandlung der statischen Felder in Abschnitt 2.6 die Deskriptorinformation bei der Übersetzung von Feldindizierungen in Form von Konstanten in die Befehlsfolgen eingefügt wurde. Jetzt, da unsere Quellsprache dynamische Felder hat, muß für jedes statische Feld, welches als aktueller Feldparameter auftritt, ein Deskriptor angelegt werden (oder eine äquivalente Vorsorge getroffen werden). Der Aufbau des Deskriptors ist der in Abbildung 2.13 gezeigte.

[8] Die Orginal-P-Maschine benötigte nur einen move-Befehl und zwar für den Fall von Speicherbereichen statischer Länge. Er hieß **mov** und sein Effekt war gleich dem von **movs**.

Tabelle 2.14: Die Blockkopierbefehle; **movs** q, für den Fall statischer Größe des zu kopierenden Bereichs, kopiert q Zelleninhalte ab Adresse $STORE[SP]$ in den Bereich, auf den SP gerade zeigt. Das Kopieren geschieht rückwärts, damit $STORE[SP]$ erst als letztes überschrieben wird.
movd, im dynamischen Fall, erwartet im obersten Kellerrahmen ab Relativadresse q den Deskriptor eines Feldes. Er kopiert dieses Feld, dessen Größe ja im Deskriptor steht, in den Bereich oberhalb des SPs. Danach wird die fiktive Anfangsadresse im Deskriptor auf die Kopie des Feldes gesetzt, so daß der Deskriptor sich nun nicht mehr auf das Originalfeld bezieht, sondern auf das kopierte Feld.

Befehl	Bedeutung	Bedin.	Erg.
movs q	**for** $i := q-1$ **down to** 0 **do** $\quad STORE[SP+i] := STORE[STORE[SP]+i]$ **od**; $SP := SP + q - 1$	(a)	
movd q	**for** $i := 1$ **to** $STORE[MP+q+1]$ **do** $\quad STORE[SP+i] :=$ $\quad\quad STORE[STORE[MP+q]$ $\quad\quad + STORE[MP+q+2]+i-1]$ **od**; $STORE[MP+q] := SP + 1 - STORE[MP+q+2]$ $SP := SP + STORE[MP+q+1]$		

Das Kopieren des Felddeskriptors geschieht mittels der gleichen Codesequenz wie sie für das Kopieren von strukturierten aktuellen Parametern statischer Größe angegeben wurde. Der Deskriptor wird also durch den statischen move-Befehl, **movs**, kopiert. Dabei wird insbesondere auch die fiktive Anfangsadresse des aktuellen Feldparameters mitkopiert. Sie wird von der aufgerufenen Prozedur benötigt, um das Feld selbst zu kopieren. Dieses Kopieren geschieht mittels des dynamischen move-Befehls, **movd**. Dieser wird mit der Relativadresse des Deskriptors versorgt. Die Feldgröße und die fiktive Quelladresse aus dem Deskriptor und der Inhalt des SP +1 als Zieladresse werden vom **movd**-Befehl benutzt.

Nachdem die Kopie des aktuellen Feldes angelegt ist, wird die fiktive Anfangsadresse der Kopie berechnet und in den Deskriptor eingetragen.

Das folgende Code-Schema übersetzt die Spezifikation eines formalen value-Feldparameters x. Diese Befehlsfolge wird beim Eintritt in eine Prozedur ausgeführt, um das dynamische Feld lokal anzulegen. Die angegebene Codesequenz setzt voraus, daß ein Aufrufer den Deskriptor des aktuellen Feldparameters bereits kopiert hat. Dieser liegt also ab Relativadresse ra im Rahmen der Prozedur, wenn $\rho(x) = (ra, st)$ ist. Eine weitere Befehlsfolge, die den Deskriptor und den Speicherbereich eines lokalen dynamischen Feldes anlegt, verbleibt als Übungsaufgabe.

$code_P$ (**value** x : **array**$[u_1..o_1, \ldots, u_k..o_k]$ **of** t;sp) $\rho\ st =$
$\quad$ **movd** ra; Kopieren des Feldes
$\quad code_P\ (sp)\ \rho\ st$

Adresse, relativ
zum Anfang

0	fiktive Anfangsadresse :a
1	Feldgröße :i
2	Subtr. für fikt. Anf. Adresse :i
3	u_1 :i
4	o_1 :i
⋮	⋮
$2k+1$	u_k :i
$2k+2$	o_k :i
$2k+3$	d_2 :i
⋮	⋮
$3k+1$	d_k :i

Abb. 2.13: Aufbau eines Felddeskriptors für ein k-dimensionales Feld.

Für anders spezifizierte formale Parameter erzeugt $code_P$ keinen Code.
$code_P\ (s;sp)\ \rho\ st = code_P\ (sp)\ \rho\ st$
$code_P\ (\,)\ \rho\ st = (\,)$

Beachten Sie, daß sowohl die Auswertung des aktuellen *value*-Parameters, als auch die Adreßberechnung für aktuelle *var*-Parameter durch die aufrufende Prozedur erfolgt. Diese kennt die richtigen statischen Vorgänger der Aufrufstelle, um korrekt auf globale Variablen zugreifen zu können.

2.9.6 Zugriff auf Variablen und formale Parameter

Abschließend wollen wir die Übersetzungsfunktion $code_L$, die die Adressierung von Variablen besorgt, so umdefinieren, daß sie sowohl den Zugriff auf lokale wie globale Variablen, als auch auf formale Parameter übersetzen kann.

Wie schon im einzelnen erklärt, gibt es dabei die folgenden Fälle:

- lokale Variablen und formale *value*-Parameter werden mit ihrer bekannten Relativadresse relativ zum Inhalt des Registers MP adressiert;
- formale *var*-Parameter werden indirekt adressiert, und zwar durch ihre Zelle, die ebenfalls relativ zum Inhalt von MP adressiert wird.
- globale Variablen werden adressiert, indem erst die Basisadresse des richtigen Kellerrahmens mittels der SV-Verweiskette berechnet und dann zu dieser ihre Relativadresse addiert wird.

Die Adressierung lokaler und globaler Variablen, sowie formaler *value*-Parameter können wir zusammenfassen; denn für den Fall $d = 0$, d.h. Zugriff auf eine lokale Größe, benutzt **lda** als Basisadresse den Inhalt von MP.

Das Übersetzungsschema $code_L$ wird jetzt für den Fall der angewandt auftretenden Namen neu definiert.

$code_L(x\ r)\ \rho\ st =$ **lda** a d ra;
$code_M\ r\ \rho\ st$,
wobei $\rho(x) = (ra, st')$ und $d = st - st'$, falls x Variable oder formaler *value*-Parameter,

$code_L(x\ r)\ \rho\ st =$ **lod** a d ra;
$code_M\ r\ \rho\ st$
wobei $\rho(x) = (ra, st')$ und $d = st - st'$, falls x formaler *var*-Parameter ist,

r ist jeweils ein eventuell leeres Wort, welches Indizierungen, Selektionen bzw. Dereferenzierungen beschreibt.

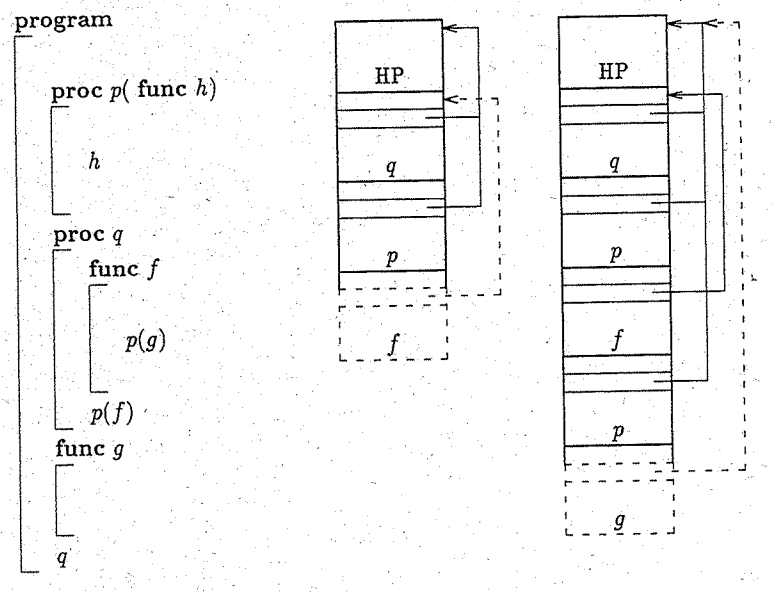
2.9.7 Formale Prozeduren als Parameter

Pascal erlaubt wie Algol60 die Übergabe von Prozeduren als Parameter. Die entsprechenden formalen Parameter nennt man **formale Prozeduren**. Die Übergabe solcher Parameter und den Aufruf formaler Prozeduren wollen wir jetzt behandeln. Betrachten wir dazu das Beispiel in Abbildung 2.14.

Wird eine formale Prozedur, im Beispiel h, aufgerufen, so wirkt das wie der Aufruf der korrespondierenden aktuellen Prozedur f bzw. g. Dazu muß dem Aufrufer der formalen Prozedur, p, die Anfangsadresse der Übersetzung der aktuellen Prozedur bekannt sein. Diese muß deshalb beim Aufruf an p übergeben werden.

Wie aber setzt man beim Anlegen des Kellerrahmens für die aktuellen Prozeduren zu h den Verweis auf den statischen Vorgänger? Statische Bindung besagt, daß beim Aufruf $p(f)$ die globalen Variablen von f zuerst in q und dann im Hauptprogramm gesucht werden, beim Aufruf $p(g)$ die globalen Variablen von g direkt im Hauptprogramm. Für den jeweiligen Aufrufer von p muß die aktuelle Prozedur sichtbar sein. Er kann sich also wie üblich durch Verfolgen der Kette der statischen Vorgänger die Anfangsadresse des Kellerrahmens der richtigen Inkarnation der Programmeinheit besorgen, in der die aktuelle Prozedur deklariert oder spezifiziert wurde. Diese Anfangsadresse wird also als weitere Information über eine aktuelle Prozedur beim Aufruf mitgegeben.

Damit wissen wir, daß bei der Übergabe einer aktuellen Prozedur die Anfangsadresse ihrer Übersetzung und die Anfangsadresse des Kellerrahmens ihres statischen Vorgängers im Parameterbereich der aufgerufenen Prozedur abgelegt werden. Wir müssen allerdings die zwei Fälle unterscheiden, daß die aktuelle Prozedur deklariert ist, oder daß sie ihrerseits als formale Prozedur spezifiziert ist.



(a) Programm mit formalen Prozeduren

(b) Kellersituation nach Aufruf $p(f)$ und (gestrichelt) nach Aufruf von h

(c) Kellersituation nach Aufruf $p(g)$ und (gestrichelt) nach Aufruf von h

Abb. 2.14: Beispiel für formale Prozeduren

$code_A\ f\ \rho\ st =$ falls f dekl. Proz. mit $\rho(f) = (adr,\ st')$ und $d = st - st'$ ist
  ldc p *adr;* Anfangsadresse der Übersetzung
  lda a *d* 0*;* SV-Verweis für späteren Aufruf

$code_A\ f\ \rho\ st =$ falls f formale Proz. mit $\rho(f) = (ra, st')$ und $d = st - st'$ ist
  lda a *d ra;* Lade Deskriptoradresse (ra ist Rel.adresse d.Deskr.)
  movs 2*;* Kopiere Deskriptor

Im ersten Fall wird ein „Prozedurdeskriptor“ [9] angelegt. Dazu muß die Anfangsadresse der Übersetzung der Prozedur geladen werden. Die P-Maschine hat keinen Typ „Programmspeicheradresse“. Dafür wurde p als Typ eingeführt. Im letzten Fall wird der Inhalt des Deskriptors der formalen Prozedur f kopiert.

[9] Diese werden uns im Kapitel Übersetzung funktionaler Sprachen als *Abschluß* wiederbegegnen.

Als letztes müssen wir noch den Aufruf einer formalen Prozedur behandeln. Gegenüber dem Schema für den Aufruf einer deklarierten Prozedur ändert sich nur folgendes:

- der Verweis auf den statischen Vorgänger wird aus der zweiten Zelle der formalen Prozedur geladen. Dazu braucht man eine modifizierte mst-Instruktion, siehe Tabelle 2.15.
- der Unterprogrammsprung erfolgt indirekt durch die erste Zelle für die formale Prozedur. Dazu benötigen wir noch eine neue Instruktion **cupi** (call user procedure indirectly), die in Tabelle 2.15 definiert ist.

Tabelle 2.15: Die Befehle **smp** (set MP) und **cupi** (call user procedure indirectly) und **mstf** (mark stack for formal procedure)

Befehl	Bedeutung	Kommentar
smp p	$MP := SP - (p+4)$;	
cupi $p\ q$	$STORE[MP+4] := PC$;	Rücksprung-
	$PC := STORE[base(p, STORE[MP+2]) + q]$	adresse
mstf $p\ q$	$STORE[SP+2] := STORE[base(p, MP) + q + 1]$;	
	$STORE[SP+3] := MP$;	
	$STORE[SP+4] := EP$	
	$SP := SP + 5$;	

Dabei werden die Funktionen der alten **cup**-Instruktion auf zwei Instruktionen aufgeteilt, da sonst die Zahl der Parameter zu groß würde.

Der Code für den Aufruf einer formalen Prozedur f ergibt sich aus dem folgenden Schema:

$code\ f(e_1, \ldots, e_k)\ \rho\ st =$
  mstf $(st - st')\ ra$
  $code_A\ e_1\ \rho\ st$
  $\vdots$
  $code_A\ e_k\ \rho\ st$
  smp s
  cupi $(st - st')\ ra$

wobei $\rho(f) = (ra, st')$ und s der Platzbedarf der aktuellen Parameter ist. An dieser Stelle steht die deklarative Information für die formalen Parameter der formalen Prozedur f zur Verfügung. Sie wird allerdings erst im $code_A$-Schema verwendet. Wie die Verwaltung der deklarativen Information geschieht, wird im Kapitel 9 beschrieben.

Nachträglich müssen wir noch die Definition von *elab_specs* um die Fälle formaler Prozeduren und formaler Funktionen erweitern. Dabei werden deren Namen an die Relativadressen der (zwei) Speicherzellen für ihre Deskriptoren gebunden.

2.10 Hauptprogramm

Als letztes wird beschrieben, wie ein Hauptprogramm übersetzt wird. Dabei gehen wir davon aus, daß das Register *SP* mit -1, das Register *NP* mit *maxstr* und alle anderen Register der P-Maschine mit 0 initialisiert sind. Insbesondere heißt das, daß die Ausführung des P-Programms mit dem Befehl in *CODE*[0] beginnt.

code (**program** *vdecls; pdecls; stats*) $\emptyset$ 0 =
    ssp *n_a;*
    code$_P$ *vdecls* ρ 1; erzeugt Code für die Besetzung von Felddeskriptoren
    sep *k;* max. Tiefe des lok. Kellers.
    ujp *l;*
    proc_code;
l: *code stats* ρ' 1;
    stp
    wobei $(\rho, n\_a) =$ *elab_vdecls vdecls* $\emptyset$ 5 1 und
    $(\rho',$ *proc_code*$) =$ *elab_pdecls pdecls* ρ 1

Der die Übersetzung beendende Befehl **stp** hat den Effekt, die Maschine anzuhalten. Die Befehle **ssp** und **sep** sind in Tabelle 2.12 beschrieben.

Warnungen

In diesem jetzt abzuschließenden Kapitel wurde um der Verständlichkeit willen stark vereinfacht. Einige harte Realitäten wurden ignoriert. Insbesondere birgt die Begrenzung durch die festzulegende Wortgröße schwere Probleme in sich und erzwingt Modifikationen in der Architektur und den Befehlen der P-Maschine.

Greifen wir das Problem der Behandlung von Konstanten heraus. Wir haben Konstanten immer als Operanden in P-Befehlen untergebracht, im **chk**-Befehl sogar zwei auf einmal. Es ist aber klar, daß, wie groß man die Wortlänge auch wählt, immer mal zwei Konstanten als Operanden der **chk**-Instruktion auftreten werden, die nicht zusammen mit dem Befehlscode in ein Wort passen werden. Deswegen differenziert die tatsächliche P-Maschine zwischen den Fällen, in denen die konstanten Operanden in den Befehl hineinpassen, und solchen, in denen das nicht der Fall ist. Dann werden die großen Konstanten in einer am oberen Speicherende, oberhalb der Halde liegenden Konstantentabelle abgelegt. In den Instruktionen stehen dann nur Verweise in diese Tabelle. Es gibt dann verschiedene Instruktionen für eine Operation, die sich darin unterscheiden, ob sie ihre Operanden im Befehl oder in der Konstantentabelle erwarten. Im letzteren Fall stehen Adressen im Befehl selbst. Dazu wird am oberen Kellerende, oberhalb der Halde, eine Konstanten-Tabelle abgelegt, in der die „großen" im Programm auftretenden Konstanten abgespeichert werden. Der Übersetzer, bzw. im Falle des

züricher P4-Übersetzers ein nachgeschalteter Assembler, generiert dann abhängig von der Größe der Konstanten unterschiedliche Befehle und legt entsprechend die Konstanten in den Befehlen oder der Konstanten-Tabelle ab.

Offene Fragen und Vorwärtsverweise

In diesem Kapitel wurde intuitiv erklärt, was die Übersetzung einer Pascal-ähnlichen Sprache in die Sprache einer geeigneten abstrakten Maschine ist. Beim geneigten Leser sollte jetzt der massive Wunsch entstanden sein, zu erfahren, wie die angegebenen Übersetzungsschemata realisiert werden.

- Die Schemata sind definiert über die syntaktische Struktur von Programmen. Sie setzen diese syntaktische Struktur bereits als bekannt voraus. Prinzipiell ist es zwar möglich, die Schemata als rekursive Funktionen aufzufassen, die die Struktur von Programmen erkennen und diese anschließend übersetzen. Allerdings wäre dies kein effizientes Verfahren; denn die Schablonen für die Ausdrücke oder auch die beiden bedingten Anweisungen wüßten lokal nicht immer, welches das richtige als nächstes anzuwendende Schema wäre. Sie müßten deshalb ausprobieren und falsche Auswahl später rückgängig machen. Deshalb werden in den Kapiteln lexikalische und syntaktische Analyse effiziente Methoden zur Erkennung der Struktur von Programmen vorgestellt.

- Bei der Übersetzung von Wertzuweisungen und Ausdrücken wurden die Typen von Variablen und Ausdrücken benutzt. Die Art ihrer Berechnung wurde nicht angegeben. Das ist eine Teilaufgabe der semantischen Analyse, welche in Kapitel 9 behandelt wird.

- In den Beispielen konnte man leicht sehen, daß die Verwendung der angegebenen Übersetzungsschemata nicht immer zu den bestmöglichen, d.h. kürzesten P-Code-Befehlssequenzen führte. Die Frage, wie man Übersetzungsschemata angeben kann, welche zu besseren, vielleicht sogar zu optimalen Befehlssequenzen führen, wird im Kapitel 12 als das Problem der Codeerzeugung behandelt. Wird dabei nichtlokale Information über dynamische, d.h. Laufzeiteigenschaften von Programmen benutzt, so muß diese durch abstrakte Interpretation berechnet werden. Grundlagen, Algorithmen und Anwendungen der abstrakten Interpretation werden in Kapitel 10 dargestellt.

- Die Verwendung von abstrakten Maschinen hat das Übersetzungsproblem etwas entschärft. Es ergeben sich neue große Probleme, wenn man Zielprogramme für reale Maschinen erzeugen und dabei deren Architektur möglichst gut nutzen will. Die Probleme bestehen darin, die Registersätze der Zielmaschinen optimal auszunutzen und unter den eventuell vielen möglichen Befehlssequenzen für Quellprogrammstücke die besten auszuwählen. Auch diese Probleme werden in Kapitel 12, Code-Erzeugung behandelt.

2.11 Übungen

3.1: Sei $\rho(a) = 5$, $\rho(b) = 6$, $\rho(c) = 7$. Bestimmen Sie
(a) $code\ (a := ((a + b) = c))\ \rho$
(b) $code_R\ (a + (a + (a + b)))\ \rho$
(c) $code_R\ (((a + a) + a) + b)\ \rho$
(d) Wieviele Kellerzellen werden bei der Ausführung der im Fall (b) und der im Fall (c) erzeugten Befehlsfolgen belegt?

3.2: Man sieht sofort, daß es in Spezialfällen „bessere" Befehlsfolgen für die Übersetzung der Wertzuweisung gibt. Modifizieren Sie das entsprechende Schema.

3.3: Wie hängt die Anzahl der für die Auswertung eines Ausdrucks maximal benötigten Kellerzellen von der Struktur des Ausdrucks ab?
Hinweis: Betrachten Sie die beiden „Extremfälle" $a + (a + (a + \ldots))$ und $(\ldots(((a + a) + a) + a) + \ldots)$!

4.1: Übersetzen Sie die Anweisungsfolge

$a := 1; b := 0;$
repeat
$b := b + c;$
$a := a + 1$
until $a = 10$

unter der Voraussetzung $\rho(a) = 5, \rho(b) = 6, \rho(c) = 7$.

4.2:
(a) Geben Sie ein rekursives Übersetzungsschema für die vereinfachte case-Anweisung an.
(b) Das angegebene Übersetzungsschema für die case-Anweisung enthält noch einen kleinen Kunstfehler. Überlegen Sie sich, was bei der Ausführung des erzeugten Codes schiefgehen kann. Beseitigen Sie diesen Fehler.
(c) Überlegen Sie sich, wie die Übersetzung „realistischer" case-Anweisungen aussähe, d.h. lassen Sie zu, daß die case-Komponenten in beliebiger Reihenfolge auftreten können, daß mehrere Selektoren die gleiche Anweisungsfolge auswählen können und daß nicht alle Selektoren des Intervalls $[0, k]$ auftreten müssen.

6.1:
(a) Definieren Sie, was die spaltenweise Ablage von Feldern heißt.
(b) Entwickeln Sie ein Codegenerierungsschema für den Zugriff auf Feldkomponenten bei spaltenweiser Ablage.

6.2: Entwerfen Sie die Übersetzung des Zugriffs auf dynamische Felder mit Komponententyp dynamischer Größe. Geben Sie ein Übersetzungsschema für die Feldindizierung und die notwendigen Erweiterungen für die Felddeskriptoren an.

6.3: Definieren Sie eine neue **chk**-Instruktion, nennen wir sie **chd**, für die Bereichsprüfung bei dynamischen Feldern.

8.1: Sei der Deklarationsteil aus Beispiel 2.8.1 gegeben. Übersetzen Sie die Wertzuweisung

$$pt\uparrow.a[i, i] := pt\uparrow.b\uparrow.b\uparrow.a[j, j] + 1$$

8.2: Es seien die folgenden Typ - und Variablendeklarationen gegeben:

type t = **record**
a: **array**$[-5..5, 1..9]$ **of integer**;
b: $\uparrow t$
end;
var i, j: **integer**;
box: t;
pt: $\uparrow t$;

Die erste zu vergebende Relativadresse ist 5, d.h $\rho(i) = 5$. Übersetzen Sie unter dieser Voraussetzung die Wertzuweisung:

$$pt\uparrow.a[i, j] := pt\uparrow.b\uparrow.b\uparrow.a[i, j] + box.a[0, 0];$$

8.3: Im Deklarationsteil einer Prozedur werde ein dynamisches Feld deklariert.

proc p;
var b: **array**$[u1..o1, u2..o2]$ **of integer**;
i, j: **integer**;
begin
$b[i, j] := b[i - 1, j + 1] + 1;$
end;

(a) Bei Eintritt in die Prozedur haben die globalen Variablen $u1$, $o1$, $u2$ und $o2$ die folgenden Werte: $u1 = -4$, $o1 = 5$, $u2 = 1$ und $o2 = 10$. Die Relativadresse von b sei $\rho(b) = 5$. Geben Sie den Felddeskriptor für b an.
(b) Übersetzen Sie die Zuweisung im Anweisungsteil der Prozedur.

9.1:
Betrachten Sie folgendes Programmfragment (b sei eine boolesche Variable):

```
program p;
  proc q
    proc t
      proc v
      proc u
      u
    t
    if b then q fi
  proc r
    if b then q else s fi
  proc s
  s
  q
  r
  s
end.
```

(a) Geben Sie zwei Aufrufbäume für das Programm p an.
(b) Die Struktur eines Programms läßt sich durch seinen Schachtelungsbaum darstellen. Die Knoten dieses Baumes sind Prozeduren bzw. das Hauptprogramm. Die Wurzel ist das Hauptprogramm. Die Söhne eines Knotens sind die von ihm direkt umfaßten Prozeduren in der Reihenfolge des Programmtextes. Geben Sie den Schachtelungsbaum für das Programm p an.
(c) Überlegen Sie sich, wie man aus dem Aufrufbaum und dem Schachtelungsbaum zu einem gegebenen Inkarnationspfad den statischen Vorgängerbaum bestimmen kann. Betrachten Sie zwei verschiedene Inkarnationspfade in den Rekursionsbäumen aus (a) und geben Sie ihre statischen Vorgängerbäume an.

9.2:
(a) Geben Sie das $code_p$-Übersetzungsschemat für den Aufbau der Felddeskriptoren für statische und dynamische Felder beim Eintritt in eine Prozedur oder in das Hauptprogramm an.
(b) Erzeugen Sie eine Befehlsfolge für den Aufbau des Deskriptors für das (statische) Feld

$$\textbf{var } a : \textbf{array}[1..7, -5..+5] \textbf{ of integer},$$

wenn $\rho(a) = 5$ ist.

(c) Erzeugen Sie eine Befehlsfolge für den Aufbau des Deskriptors für das dynamische Feld

$$\textbf{var } b : \textbf{array}[1..x, y..z] \textbf{ of integer},$$

wenn x, y formale Parameter der aktuellen Prozedur mit $\rho(x) = 5$, $\rho(y) = 6$ sind, und z eine globale Variable, in der umgebenden Prozedur deklariert, mit $\rho(z) = 7$ ist.

9.3: Zum Setzen des *EP* benötigt der Übersetzer die maximale Tiefe des lokalen Kellers. Wie berechnet sich dieser (statische) Wert?

Hinweis: Beachten Sie, was bei der Auswertung aktueller Parameter geschieht, und benutzen Sie die Ergebnisse von Übung 3.3.

9.4: Übersetzen Sie das folgende Programm:

```
program test;
var i, faktor, sum: integer;
proc q(x: integer; var y: integer);
begin
    y := faktor * x + y
end;
begin
    i := 1; sum := 0; faktor := 2;
    while i ≤ 10
    do q(i + 1, sum);
        i := i + 1
    od
end.
```

9.5: In dieser Aufgabe betrachten wir einige einfache Erweiterungen unserer Pascal-ähnlichen Sprache. Überlegen Sie Sich, wie man sie auf der P-Maschine implementieren könnte.
(a) Eine Zuweisung $v := e$ soll, neben dem Ändern des Inhalt von v, einen Wert bezeichnen, nämlich gerade den Wert von e. Damit sind etwa Ausdrücke der Gestalt $(x := a + b) > 0$ möglich.
(b) Funktionen konnten bisher nur Werte von skalarem Typ als Ergebnis zurückliefern. Jetzt lassen wir zu, daß ihr Ergebnistyp ein beliebiger statischer Typ, etwa ein Verbundtyp, sein kann.
(c) Eine prozedurlokale Variable heißt own-Variable (in C static-Variable), wenn sie Prozeduraufrufe überlebt. D.h. bei einem erneuten Aufruf der Prozedur hat sie den gleichen Wert, den sie beim Verlassen des vorhergehenden Aufrufs ihrer Prozedur hatte.

2.12 Literaturhinweise

Sprachorientierte abstrakte Maschinen gibt es schon recht lange. Sie wurden benutzt, um die Übersetzung zu vereinfachen und die Portierung auf andere Maschinen zu erleichtern. In [RR64] wurde eine abstrakte Maschine für Algol60, der Algol Object Code (AOC), beschrieben.

Die in diesem Abschnitt beschriebene P-Maschine wurde für den weltweit verbreiteten züricher P4-Übersetzer für Pascal verwendet. Sie ist beschrieben in [Amm81] und in [PD82], worin sich die Quellen des P4-Übersetzers, des Assemblers und des P-Maschineninterpreters finden.

Sowohl der Algol Object Code als auch die P-Maschine sind für spezielle Quellsprachen entworfen. In den 50erJahren hat man versucht, das Problem, daß man für m Quellsprachen und n Zielmaschinen $m \cdot n$ Übersetzer braucht, durch die Suche nach einer universellen Zwischensprache, genannt UNCOL (UNiversal Communication Oriented Language) zu lösen. Diese hätte dann nur noch m Übersetzer in UNCOL und n Übersetzer von UNCOL in Maschinensprachen erfordert. Diese Suche erbrachte Vorschläge, z.B. [Ste61]. Es hat sich aber gezeigt, daß die Universalität zusammen mit den anderen Entwurfszielen nicht zu erreichen ist. Gesucht wird ein UNCOL aber immer wieder.

Kapitel 3

Übersetzung funktionaler Programmiersprachen

3.1 Sprachtyp und einleitendes Beispiel

Funktionale Programmiersprachen haben ihren Ursprung in Lisp, gehen also bis zum Anfang der 60er Jahre zurück. Aber erst seit Ende der 70er Jahre hat sich diese Sprachklasse von der Dominanz von Lisp befreit und durch die Entwicklung neuer Konzepte und Implementierungsmethoden zu dem Sprachtyp entwickelt, den wir – repräsentiert durch Miranda[1] – hier behandeln wollen.

Imperative Sprachen kennen (mindestens) zwei Welten, die Welt der Ausdrücke und die Welt der Anweisungen. Ausdrücke liefern Werte; Anweisungen verändern den Zustand von Variablen oder steuern den Kontrollfluß. Bei entsprechender Programmierung kann auch bei der Auswertung von Ausdrücken als Seiteneffekt der Zustand von Variablen verändert werden. In funktionalen Sprachen gibt es nur Ausdrücke, und die Ausführung eines funktionalen Programms besteht in der Auswertung des zugehörigen „Programmausdrucks", der das Programmergebnis beschreibt. Zu seiner Auswertung müssen evtl. noch viele weitere Ausdrücke ausgewertet werden, wenn ein Ausdruck durch Funktionsanwendung andere Ausdrücke „aufruft", aber es gibt keinen explizit durch Anweisungen angegebenen Kontrollfluß. Eine Variable in einem funktionalen Programm ist Bezeichner für einen Ausdruck, nicht Bezeichner für eine oder mehrere Speicherstellen wie in imperativen Sprachen. Ihr Wert kann sich durch die Ausführung des Programmes nicht ändern; lediglich die „Reduktion" des von ihr bezeichneten Ausdrucks auf seinen Wert ist möglich.

Eine moderne funktionale Programmiersprache enthält die meisten der folgenden Konzepte:

Funktionsdefinitionen. Dies geschieht entweder über **λ-Abstraktion** oder über **Rekursionsgleichungen.** Ein Ausdruck $\lambda x_1 \ldots x_n.E$ definiert eine Funktion in den n Argumenten $x_1 \ldots x_n$ mit dem definierenden Ausdruck E. Damit hat man eine anonyme, d.h. namenlose Funktion definiert, die man auf Argumente anwenden kann, indem man $\lambda x_1 \ldots x_n.E$ vor die Argumentausdrücke hinschreibt. Das ist zum Programmieren aber nicht attraktiv. Man möchte Funktionen Namen geben, die man dann anstatt der Funktionsdefinition mit Argumenten

[1] Miranda ist eingetragenes Warenzeichen von Research Software, Canterbury, Großbritannien