Inhalt

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- erklären können, welche Arten von Speicher man bei der Übersetzung imperativer Sprachen benötigt,
- den Zusammenhang zwischen der Lebensdauer von Variablen und der Stack-Speicherverwaltung darstellen können,
- erklären können, welchen Einfluss geschachtelte Prozedurdeklarationen auf die Speicherverwaltung haben,
- den Display-Mechanismus beschreiben können,
- die prinzipielle Struktur von Prozedurrahmen beschreiben können,
- wissen, welche Arten von Zwischendarstellungen es gibt,
- 3-Adress-Code und seine Befehlsklassen kennen,
- die Struktur der abstrakten Maschine zur Auswertung von 3-Adress-Code darstellen können,
- die Infrastruktur für die Übersetzung (Symboltabellen, Hilfsfunktionen) skizzieren können,
- die grundsätzlichen Probleme bei der Übersetzung von Deklarationen und ihre Lösungstechniken kennen,
- die Übersetzung von Ausdrücken und die Strategie für den Zugriff auf strukturierte Variablen beschreiben können,
- wissen, wie boolesche Ausdrücke im Kontext von Kontrollstrukturen übersetzt werden,
- einfache Übersetzungsschemata, etwa für Kontrollstrukturen, angeben können.

Kapitel 6

Übersetzung imperativer Programmiersprachen

In diesem Kapitel behandeln wir die klassische Anwendung des Übersetzerbaus, die Übersetzung imperativer Sprachen wie PASCAL, Modula-2, C usw. In Abschnitt 6.1 betrachten wir zunächst allgemein die Strategie zur Speicheraufteilung und -verwaltung für imperative Sprachen. In Abschnitt 6.2 führen wir eine konkrete Zwischensprache ein, die zunächst Ziel der Übersetzung sein soll und deren zugrundeliegendes Maschinenmodell die Strategie aus Abschnitt 6.1 widerspiegelt. Schließlich geben wir in Abschnitt 6.3 konkrete Übersetzungsschemata für die wesentlichen Konzepte imperativer Sprachen an.

6.1 Speicherorganisation und Laufzeitsystem

Wir betrachten zunächst die Anforderungen an die Speicherverwaltung, die sich aus den Konzepten imperativer Sprachen ergeben. Im zweiten Unterabschnitt entwerfen wir dann ein Konzept für die Speicherverwaltung, die teilweise zur Laufzeit durch das *Laufzeitsystem* der Programmiersprache vorgenommen werden muss.

6.1.1 Anforderungen

Die Aufgabe des Übersetzers besteht darin, für alle im Programm benutzten Variablen Speicherbereiche bereitzustellen und im Zielprogramm jede Benutzung der Variablen durch Zugriff auf die entsprechende Adresse zu realisieren. Wir betrachten folgende Aspekte von Variablen und ihre Konsequenzen für die Speicherverwaltung:

- Bedeutung verschiedener Datentypen,
- Lebensdauer von Variablen,
- Sichtbarkeit von Bezeichnern.

Bedeutung verschiedener Datentypen. Programmiersprachen bieten einen Satz *primitiver* Datentypen wie *integer*, *real*, *char*, *boolean* usw., außerdem Typkonstruktoren wie *record*, *array*. Mit Hilfe der Typkonstruktoren können komplexe Datentypen aus den primitiven Typen konstruiert werden. Schließlich kann man

Zeigertypen definieren, die es ermöglichen, auch dynamische Datenstrukturen (Listen, Bäume usw.) zu erzeugen.

Ganz allgemein gilt, dass der Übersetzer Werte jedes beliebigen Datentyps jeweils in einem zusammenhängenden Speicherbereich, also in einer Folge von Speicherzellen (Bytes oder Worten), darstellt. Für jede Variable ist daher ein solcher Speicherblock anzulegen. Primitive Datentypen werden in einer kleinen, konstanten Anzahl von Bytes dargestellt, etwa 1 Byte für *char*, 2 Bytes für *short integer*, 4 Bytes für *real* usw. Typkonstruktoren (*array*, *record*) erzeugen einen neuen Typ aus einer Menge von Argumenttypen. Die Darstellung des erzeugten Typs ergibt sich dabei jeweils durch sequentielle Anordnung (ggf. mit Lücken, wie wir noch sehen werden) der Darstellungen der Argumenttypen. Damit entsteht wieder ein zusammenhängender Speicherbereich für den neuen Typ. Alle Zeigertypen werden in einem Wort dargestellt; der dargestellte Wert ist die Adresse eines anderen Datenobjekts.

Lebensdauer von Variablen. Nicht alle in einem Programm vorkommenden Variablen sind während der gesamten Laufzeit eines Programms *gültig*. Das liegt vor allem an der Möglichkeit, Unterprogramme, also Prozeduren oder Funktionen, zu deklarieren (wir sprechen im Folgenden einheitlich von Prozeduren). Innerhalb von Prozeduren deklarierte Variablen heißen *lokale*, außerhalb von allen Prozeduren deklarierte Variablen *globale Variablen*. Lokale Variablen sind nur gültig, während eine Prozedurinkarnation aktiv ist. Globale Variablen sind gültig während der gesamten Laufzeit des Hauptprogramms. In manchen Sprachen kann man lokale Variablen auch innerhalb von *Blöcken* oder *Verbundanweisungen* (z. B. durch *begin* - *end* geklammert) deklarieren; eine solche Variable ist in allen eingeschlossenen Blöcken gültig. Den Ablauf eines Programms mit geschachtelten Prozeduraufrufen kann man als Aufrufbaum darstellen (Abb. 6.1):

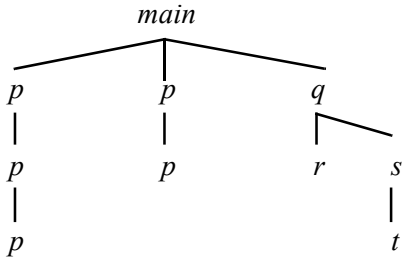


Abb. 6.1. Aufrufbaum für geschachtelte Prozeduraufrufe

Wenn wir für jede Prozedurinkarnation den Speicherbereich für ihre lokalen Variablen als Rechteck darstellen, erhalten wir Abb. 6.2.

Von zentraler Bedeutung ist die Tatsache, dass zu jedem Zeitpunkt nur die lokalen Variablen der Prozedurinkarnationen *entlang einem Pfad im Aufrufbaum* gültig sind und nur für sie Speicherbereiche existieren müssen, also etwa gerade die fett

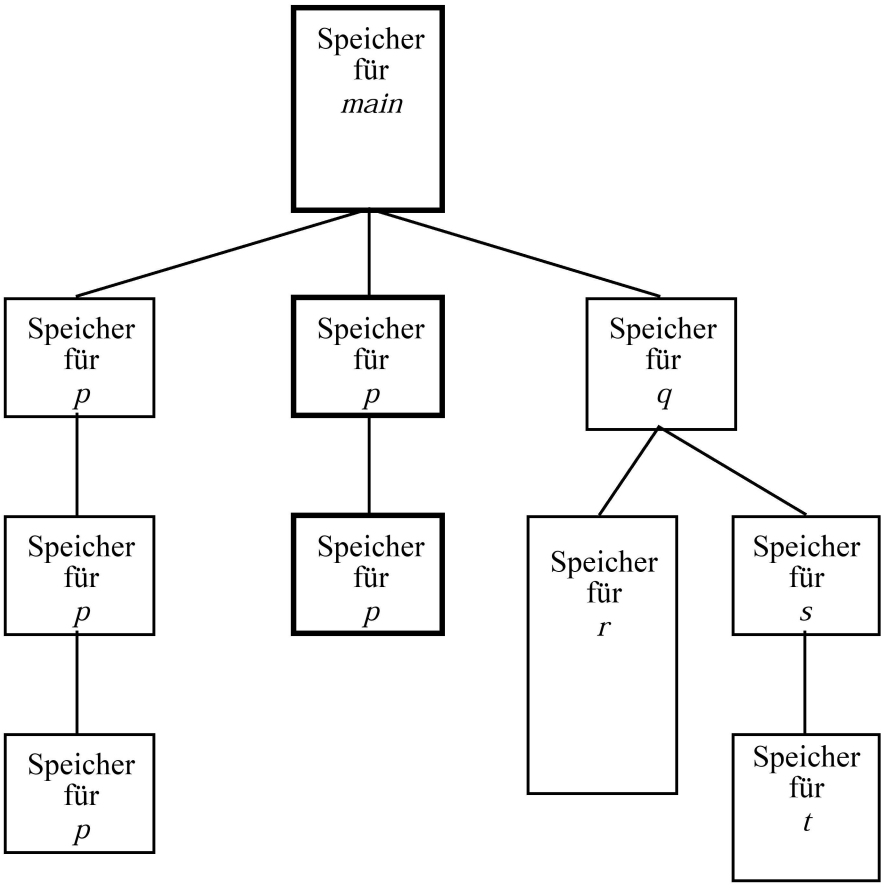


Abb. 6.2. Speicherbereiche für Aufrufbaum

gezeichneten Speicherbereiche in Abb. 6.2. Diese Überlegung führt unmittelbar zum Konzept einer *kellerartigen Speicherverwaltung*, wie in Abb. 6.3 gezeigt. Bei jedem Eintritt in eine neue Prozedurinkarnation wird der entsprechende Speicherbereich rechts an den Kellerspeicher angehängt (dieser Stack wächst also nach rechts); beim Verlassen der Prozedur wird der Speicher wieder freigegeben. Der dem Programm zur Verfügung stehende noch freie Speicher ist grau dargestellt.

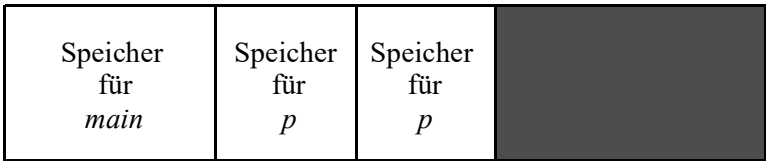


Abb. 6.3. Kellerartige Speicherverwaltung

Die bisherige Betrachtung gilt zunächst für Sprachen, in denen keine lokalen Variablen in Blöcken deklariert werden können (z. B. PASCAL). Wenn lokale Variablen in Blöcken möglich sind, so kann man beobachten, dass der Eintritt in Blöcke und das Verlassen von Blöcken genauso verläuft wie Eintritt und Verlassen geschachtelter Prozedurinkarnationen. Insofern kann man Speicherplatz für lokale Variablen von Blöcken vollkommen analog behandeln. Das heißt, beim Eintritt in den Block wird Speicherplatz für ihn auf dem Stack zugewiesen, beim Verlassen des Blockes freigegeben.

Es gibt allerdings doch einen Unterschied zwischen Blöcken und Prozeduren: Blöcke können nicht rekursiv durchlaufen werden – in dem Sinne, dass Block *A* noch einmal aktiviert würde, während man sich schon im Block *A* befindet. Es müssen also niemals gleichzeitig zwei oder mehr Versionen des Speicherplatzes für Block *A* existieren. Damit ist auch die alternative Strategie möglich, den Speicherplatz für alle Blöcke einer Prozedur jeweils auf einen Schlag zuzuweisen, was wieder der oben für Prozeduren beschriebenen Vorgehensweise entspricht.

Wenn die zu übersetzende Sprache *keine rekursiven Prozeduren* zulässt (z. B. FORTRAN), so kann man die gerade für Blöcke beschriebene Speicherzuordnung – nämlich Platz für alle Blöcke ist gleichzeitig, nebeneinander vorhanden – auch für Prozeduren verwenden. Jede Prozedur bekommt einen festen Speicherbereich zugewiesen, der über die gesamte Lebensdauer des Programms zur Verfügung steht. Die Kellerspeicherverwaltung ist dann nicht notwendig. Alle modernen Sprachen erlauben aber rekursive Prozeduren.

In manchen Sprachen ist es möglich, lokale Variablen in Prozeduren als *statisch* zu deklarieren, etwa in der Sprache C:

```
static int i;
```

Eine solche Variable hat die gleiche Lebensdauer wie globale Variablen; ihr Wert steht also über verschiedene Prozedurinkarnationen hinweg zur Verfügung. Solche Variablen müssen in einem statischen Speicherbereich untergebracht werden, z. B. in dem Bereich, in dem auch globale Variablen angelegt werden.

Eine wichtige Frage ist noch die, ob der Platzbedarf für alle Inkarnationen einer Prozedur *p* eigentlich gleich groß und zur Übersetzungszeit bekannt ist? Das ist dann der Fall, wenn die Sprache keine *dynamischen Arrays* zulässt. Das heißt, für jede Array-Variable sind die Indexgrenzen zur Übersetzungszeit bekannte Konstanten (z. B. PASCAL, Modula-2). Eine Sprache besitzt dynamische Arrays, wenn es möglich ist, in einer Prozedur oder einem Block einen Array zu deklarieren, dessen Indexgrenzen von den aktuellen Werten irgendwelcher Variablen abhängen, die damit erst zur Laufzeit bekannt sind. Wenn es dynamische Arrays gibt, so kann bzw. muss der Platz für sie auf dem Kellerspeicher zur Laufzeit bei Eintritt in die Prozedur oder den Block angelegt werden.

Neben *globalen* und *lokalen* Variablen gibt es Variablen (Behälter für Werte), deren Lebensdauer im Programm *speziell festgelegt* wird und zwar dadurch, dass Speicherplatz für sie explizit angefordert und ggf. wieder freigegeben wird. Diese

Variablen sind selbst anonym (besitzen keine Namen) und werden über Zeigervariablen angesprochen. Das Anfordern und Freigeben von Speicherplatz geschieht z. B. in PASCAL über Aufrufe von *new* und *dispose*. Mit solchen Variablen werden dynamische Datenstrukturen (Bäume usw.) aufgebaut.

Für die Speicherverwaltung ergibt sich als Konsequenz, dass ein gewisser Speicherbereich zur Verfügung stehen muss, aus dem Speicherblöcke unterschiedlicher Größe in nicht vorhersagbarer Reihenfolge angefordert und wieder freigegeben werden können. Ein solcher Speicherbereich wird üblicherweise als *Heap* bezeichnet.

Sichtbarkeit von Bezeichnern. Der gleiche Name oder Bezeichner kann in einem Programmtext durchaus für verschiedene Variablen verwendet werden. Eine Prozedur oder ein Block definiert einen *Sichtbarkeitsbereich* (*Scope*) für eine dort deklarierte Variable. Prozeduren und Blöcke treten sowohl im Programmtext als auch während der Ausführung des Programms geschachtelt auf. Dadurch sind auch die Sichtbarkeitsbereiche geschachtelt, und zwar einerseits *statisch* (im Programmtext), andererseits *dynamisch* (zur Laufzeit). Die Bindung von Namen an Variablen wird durch diese Schachtelung der Sichtbarkeitsbereiche bestimmt. Die meisten Sprachen benutzen die *statische* oder *lexikalische* Schachtelung (*statischer Scope*, z. B. PASCAL, Modula-2, C, Ada); es gibt aber auch Sprachen, die die dynamische Schachtelung zugrunde legen (*dynamischer Scope*, z. B. Lisp). Wir beschränken uns im Folgenden auf die Diskussion von Sprachen mit statischem Scope.

Die allgemeine und recht einfache Idee zur Bindung eines Namens x besteht darin, dass man die geschachtelten Scopes von innen nach außen durchläuft, bis man auf eine Deklaration für x stößt; x wird mit der ersten so gefundenen Deklaration gebunden. Die Frage ist dann nur noch, wie man Scopes lexikalisch in der gegebenen Sprache schachteln darf. Man kann unterscheiden:

(i) Prozedurdeklarationen dürfen geschachtelt werden oder nicht.
(ii) Blöcke können lokale Variablen haben oder nicht.

In PASCAL und Modula-2 z. B. kann man Prozedurdeklarationen schachteln, aber es gibt keine lokalen Variablen in Blöcken. In C gibt es lokale Variablen in Blöcken, aber man kann Prozedur- bzw. Funktionsdeklarationen nicht schachteln. Wir betrachten ein Beispiel mit geschachtelten Prozedurdeklarationen in Abb. 6.4. Die äußerste Ebene ist die des Hauptprogramms.

Eine Variable ist *sichtbar* in dem Scope, in dem sie deklariert wird, mit allen eingeschlossenen Scopes abzüglich derer, in denen es eine andere Deklaration mit demselben Namen gibt. Im Gegensatz dazu ist eine Variable *gültig* in dem Scope, in dem sie deklariert wird, und allen eingeschlossenen Scopes. Wenn sie in einem eingeschlossenen Scope gültig, aber nicht sichtbar ist, nennen wir sie dort *verdeckt*. Der Begriff der Gültigkeit entspricht der Lebensdauer der Variablen.

Im Beispiel in Abb. 6.4 ist die Variable s im gesamten Programm sichtbar, da es keine eingeschlossene Prozedur gibt, in der s erneut deklariert wird. Die in Prozedur A deklarierte Variable i ist im Scope von A sichtbar mit Ausnahme des Scopes von B. Denn in B wird erneut eine Variable i deklariert.

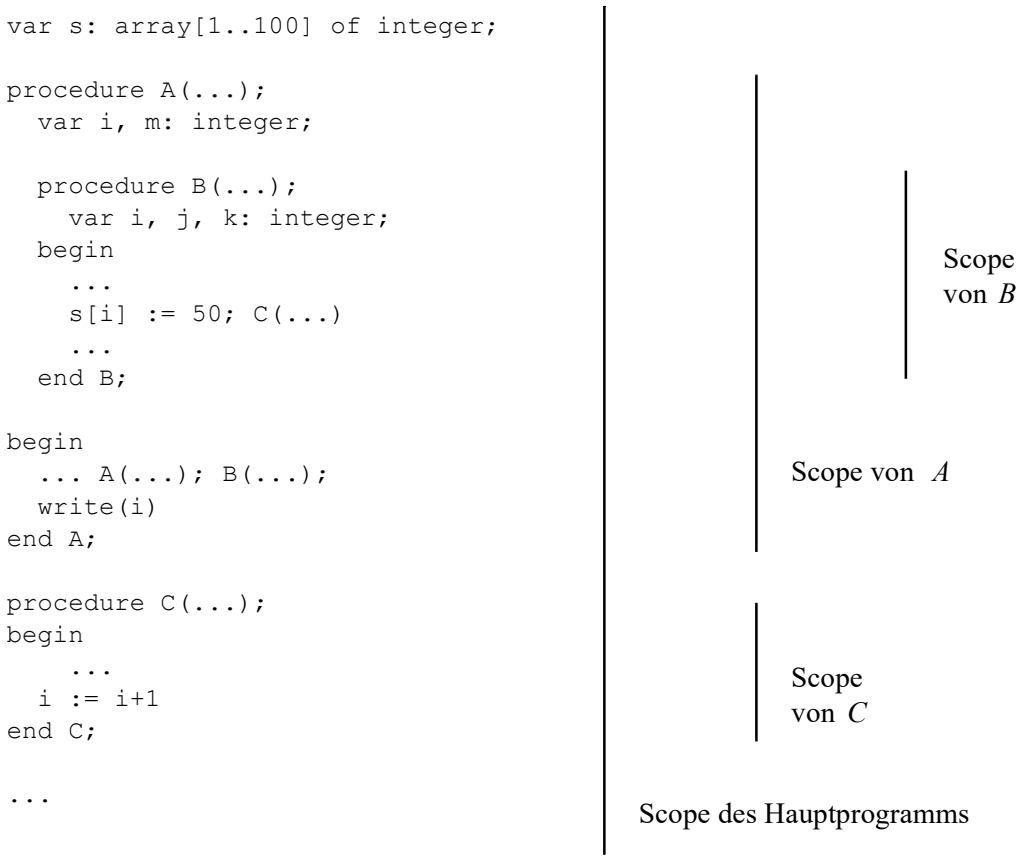


Abb. 6.4. Sichtbarkeitsbereiche (Scopes)

Das Laufzeitsystem muss so konstruiert werden, dass zur Laufzeit auf alle Variablen zugegriffen werden kann, die gerade gültig und darüber hinaus sichtbar sind. Insbesondere muss je nach Sichtbarkeitsregeln der Sprache die Verwaltung des Kellerspeichers ggf. so angelegt werden, dass auf Speicherblöcke von Prozeduren oberhalb des Speichers der gerade aktiven Prozedur zugegriffen werden kann. Die genauere Technik dazu besprechen wir in Abschnitt 6.1.2.

6.1.2 Speicheraufteilung

Aus den bisher diskutierten Anforderungen ergibt sich das in Abb. 6.5 dargestellte Konzept für die Speicheraufteilung.

Der gesamte einem Programm zur Verfügung stehende Speicherplatz wird aufgeteilt in die vier gezeigten Bereiche. Da Stack- und Heap-Bereiche dynamisch sind, d. h., ihre Größen variieren, lässt man diese Bereiche aufeinander zu wachsen. Dazwischen liegt der noch freie Speicher. Wenn zur Laufzeit Stack- und Heap-Grenze aneinanderstoßen, entsteht ein Fehler („Stack Overflow“ oder „Heap Overflow“). – Über den Bereich für den Programmcode ist nichts weiter zu sagen. Die Verwaltung

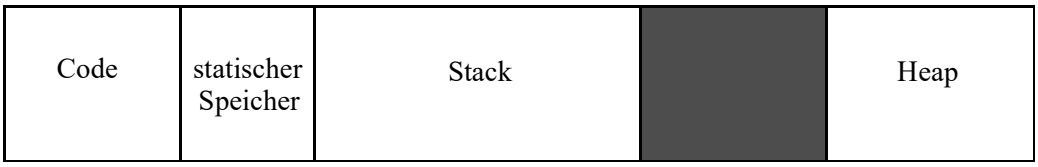


Abb. 6.5. Grundkonzept für die Speicherverwaltung

der drei anderen Bereiche diskutieren wir in den folgenden Abschnitten etwas genauer.

Statischer Speicher und Layout lokaler Daten

Der statische Speicher dient zur Verwaltung globaler Variablen und ggf. der Verwaltung als statisch deklarierter lokaler Variablen von Prozeduren. Hier besteht die Aufgabe eigentlich nur darin, Speicherbereiche für Variablen sequentiell aneinanderzureihen. Die gleiche Aufgabe entsteht an zwei anderen Stellen, nämlich beim Layout des Speichers für lokale Daten einer Prozedur (innerhalb des Speichers für die Prozedur auf dem Stack) sowie beim Anordnen der Komponenten einer Record-Variablen.

Der Übersetzer kann Platz fortlaufend vergeben beim Übersetzen der Variablendeklarationen. Analog dazu wird in einer Typdefinition für einen Record zwar noch kein Speicherplatz zugeordnet, aber mitgezählt, wieviel Platz innerhalb des Records vergeben ist, und es werden *Offsets* für die Feldnamen des Records gemerkt. Die jeweils benötigte Anzahl von Bytes liegt für primitive Typen fest; für im Programm definierte Typen wird sie einer Symboltabelle entnommen, in die die Größe bei der Übersetzung der Typdeklaration eingetragen wird.

Zu beachten bei diesem Aneinanderreihen ist lediglich das Problem des *Alignments*: Aufgrund der Maschinenarchitektur ist es gewöhnlich nötig, dass etwa 4-Byte Integer-Werte auf durch 4 teilbaren Speicheradressen beginnen müssen, 8-Byte-Darstellungen (z. B. *longreal*) auf durch 8 teilbaren Adressen usw. Das Einfügen von Lücken aus diesem Grund bezeichnet man als *Padding*.

Beispiel 6.1: Eine Folge von Deklarationen

```
char c;                 (1 Byte)
int i;                  (4 Bytes)
short int j;            (2 Bytes)
float f;                (4 Bytes)
```

würde etwa zu dem in Abb. 6.6 gezeigten Layout führen. □

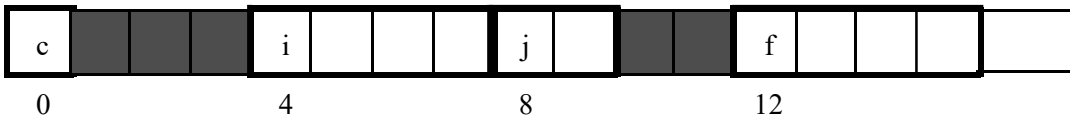


Abb. 6.6. Alignment für Variablen verschiedener Größe

Stack-Verwaltung

Wir betrachten nun die Verwaltung von Speicherbereichen für Prozedurinkarnationen auf dem Kellerspeicher genauer. Die Speicherbereiche werden *Prozedurrahmen* genannt. Eine mögliche Struktur von Prozedurrahmen ist in Abb. 6.7 gezeigt.

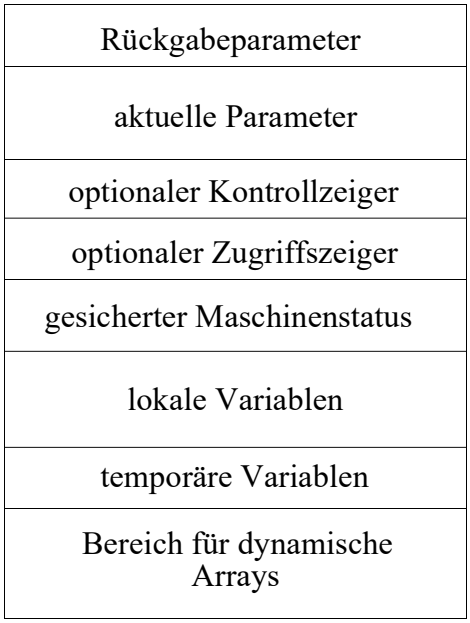


Abb. 6.7. Struktur von Prozedurrahmen

Die Felder haben folgende Bedeutung:

- Platz für *aktuelle Parameter* und *Rückgabeparameter* dienen der Parameterübergabe zwischen Aufrufer und aufgerufener Prozedur. Wenn möglich, wird man aus Effizienzgründen allerdings Parameter in Registern übergeben; dann entfallen diese Felder.
- Der *Kontrollzeiger* zeigt auf den Beginn des Prozedurrahmens des Aufrufers (der im Stack direkt oberhalb von diesem Prozedurrahmen liegt).
- Der *Zugriffszeiger* zeigt auf den Prozedurrahmen einer Vorgängerprozedur *p* auf dem Stack, deren lexikalischer Scope den Scope des Textes dieser Prozedur

q direkt einschließt. Das heißt, *p* ist der direkte Vorgänger von *q* in der statischen Schachtelung.
- Der *gesicherte Maschinenstatus* enthält Statusinformationen des Aufrufers wie Programmzähler und Registerinhalte; diese Umgebung muss bei der Rückkehr für den Aufrufer wiederhergestellt werden.
- Platz für *lokale Variablen* wird vergeben wie oben beschrieben; *temporäre Variablen* werden bei der Übersetzung von Ausdrücken erzeugt.
- Speicherbereiche für dynamische Arrays (soweit vorgesehen in der Sprache) werden beim Eintritt in die Prozedur am Ende des Prozedurrahmens angelegt.

Beim Aufruf einer Prozedur muss für den Aufgerufenen ein neuer Prozedurrahmen auf dem Stack angelegt und geeignet initialisiert werden. Bei der Rückkehr muss der Rahmen abgebaut und die Umgebung für den Aufrufer wiederhergestellt werden. Der Programmcode für die dabei durchzuführenden Aktionen kann im Aufrufer oder im Aufgerufenen plaziert werden; hier gibt es eine gewisse Freiheit bei der Aufgabenverteilung.

Zugriffszeiger werden in Sprachen mit geschachtelten Prozedurdeklarationen benötigt und dienen dem Zugriff auf nicht-lokale Variablen. In einer Sprache ohne geschachtelte Prozeduren, wie etwa C, entsteht dieses Problem nicht: Jede Variable, die nicht lokal zu einer Prozedur ist, ist global für alle Prozeduren und wird im statischen Speicherbereich untergebracht. In C beziehen sich Variablennamen also nur entweder auf den obersten Prozedurrahmen auf dem Stack oder auf den statischen Speicher.

Ein Beispiel für geschachtelte Prozedurdeklarationen haben wir in Abb. 6.4 gesehen. Wir betrachten eine Aufrufhierarchie *main – A – A – B – C*. Dann entsteht auf dem Stack die in Abb. 6.8 gezeigte Anordnung von Prozedurrahmen mit ihren Zugriffszeigern (hier wächst der Stack nach unten):

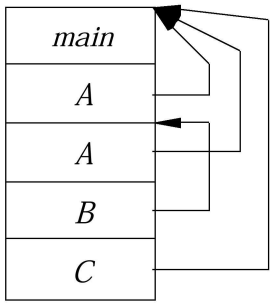


Abb. 6.8. Prozedurrahmen mit Zugriffszeigern

Die statische Schachtelung der Prozeduren in Abb. 6.4 ist

```
main          (Tiefe 0)
  A           (Tiefe 1)
    B         (Tiefe 2)
  C           (Tiefe 1)
```

Wie man sieht, zeigt im Stack der Zugriffszeiger jeweils auf den statischen Vorgänger, in *A* und *C* also auf *main*, in *B* auf die letzte Inkarnation von *A*. Beim Zugriff auf eine Variable muss lediglich ihre *relative Tiefe* beachtet werden. Wenn *B* auf die Variable *s* des Hauptprogramms zugreift, dann ist deren relative Tiefe 2, nämlich Tiefe(*B*) – Tiefe(*main*) . Das heißt, man muss zwei Zugriffszeigern folgen, um die Basisadresse des Prozedurrahmens für *s* zu erhalten. Variablen *i*, *j*, *k* in *B* haben relative Tiefe 0. Der Übersetzer kann die Tiefen und damit die relativen Tiefen aller Variablen bei der Analyse des Programmtextes bestimmen.

Es gibt noch eine etwas effizientere Alternative zur Verwendung von Zugriffszeigern, den sogenannten *Display*-Mechanismus. Dabei benutzt man ein separates Feld von Zeigern auf die Prozedurrahmen statischer Vorgänger der gerade aktiven Prozedur (Abb. 6.9).

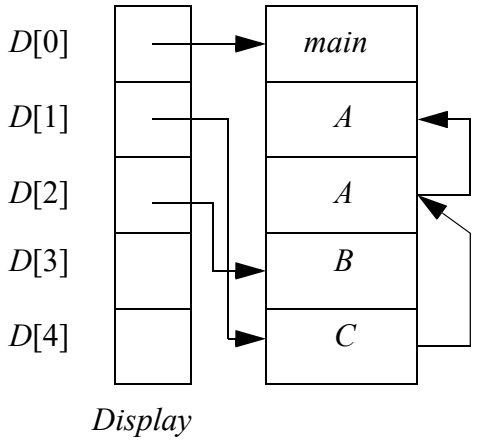


Abb. 6.9. Verwaltung von Prozedurrahmen mit dem Display-Mechanismus

Der Display-Zeiger *D*[*i*] zeigt auf den obersten Prozedurrahmen einer Prozedur der statischen Schachtelungstiefe *i* auf dem Stack. Um in einer Prozedur, die selbst Tiefe *k* hat, auf eine Variable der relativen Tiefe *j* zuzugreifen, folgt man hier nicht *j* Zugriffszeigern, sondern findet den richtigen Prozedurrahmen über den Display-Zeiger *D*[*k* – *j*]. Um z. B. in *B* (Tiefe 2) auf die Variable *s* zuzugreifen (relative Tiefe 2), benutzt man *D*[0]. Dies ist im Allgemeinen effizienter, da man stets nur einem Zeiger folgen muss.

Bei diesem Verfahren gibt es in den Prozedurrahmen anstelle der Zugriffszeiger Felder zur Sicherung von Display-Zeigern. Beim Aufruf einer Prozedur *p* der Schachtelungstiefe *i* wird ein neuer Prozedurrahmen für *p* angelegt und der Display-Zeiger *D*[*i*] auf diesen Rahmen umgesetzt. Der alte Wert von *D*[*i*] wird im Sicherungsfeld des Rahmens für *p* gemerkt. Bei Beendigung von *p* wird der gesi-

cherte Wert in $D[i]$ restauriert. Abb. 6.10 illustriert in 5 Schnappschüssen die Übergänge in der Verwaltung von Display und Prozedurrahmen bei sukzessivem Aufruf der Prozeduren *main – A – A – B – C*; der letzte Schnappschuss entspricht also Abb. 6.9. Man beachte, dass die Übergänge bei Beendigung der Prozeduren auch alle in umgekehrter Richtung ausführbar sind.

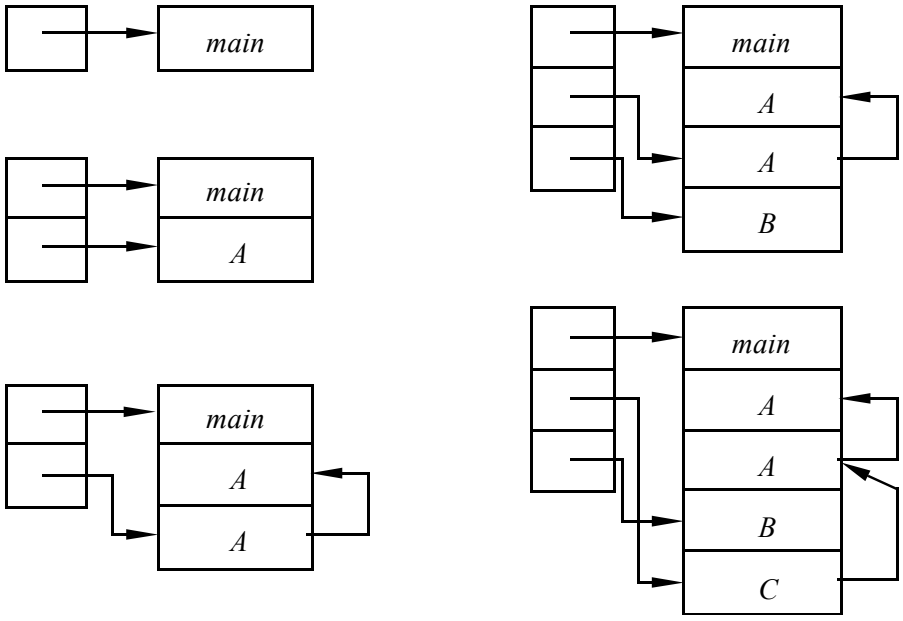


Abb. 6.10. Verwaltung von Prozedurrahmen mit dem Display-Mechanismus: 5 Schnappschüsse für die Aufruffolge *main – A – A – B – C*

Der Display selbst kann z. B. im statischen Speicherbereich untergebracht werden. Seine (Maximal-)Größe ist zur Übersetzungszeit bekannt; sie entspricht der maximalen statischen Schachtelungstiefe. Noch effizienter ist es, die Display-Felder in einer Folge von Registern unterzubringen.

Heap-Verwaltung

Die Aufgabe der Heap-Verwaltung besteht darin, explizite Speicheranforderungen des Programms zur Laufzeit zu erfüllen und für explizite Speicherfreigaben den Platz wieder verfügbar zu machen. Im Programm geschieht dies etwa durch Befehle *new* und *dispose* oder *allocate* und *deallocate*.

In den hier betrachteten imperativen Sprachen ist die Folge angeforderter Blockgrößen und freizugebender Blöcke nicht vorhersagbar (im Gegensatz zu Sprachen wie etwa LISP, in denen große Mengen gleich großer Blöcke benötigt werden). In dieser allgemeinen Situation gibt es zwei Hauptklassen von Strategien zur Speicherverwaltung, nämlich:

(i) Zuordnen von Blöcken exakt der angeforderten Größe,
(ii) Buddy-Verfahren.

Auf Buddy-Verfahren wollen wir hier nicht weiter eingehen. Bei Strategie (i) ist der verfügbare Speicherbereich nach einer Folge von Anforderungen und Freigaben zerlegt in eine alternierende Folge von belegten Blöcken und Lücken (Abb. 6.11).

Abb. 6.11. Zerlegung des Speichers in belegte und freie Blöcke

Bei Freigabe eines Blockes ist sein Speicherbereich mit den angrenzenden Lücken zu einer neuen Lücke zu verschmelzen. Bei Anforderung eines Blockes ist eine Lücke auszuwählen, aus der ein Block der gewünschten Größe entnommen werden kann. Bekannte Taktiken zur Auswahl dieser Lücke sind:

1. *First fit*. Wähle die erste Lücke, die groß genug ist.
2. *Rotating first fit.* Wie first fit, aber beginne die Suche dort, wo zuletzt ein Block gefunden wurde (damit nicht immer von Anfang des Speichers an gesucht wird).
3. *Best fit.* Wähle die kleinste Lücke, in die der Block hineinpasst.
4. *Worst fit*. Wähle die größte Lücke, damit noch etwas Brauchbares übrigbleibt.

Wenn keine geeignete Lücke mehr zu finden ist, wächst der Heap-Bereich insgesamt nach unten (= links). Wenn er dabei mit dem Stack-Bereich des Programms kollidiert, entsteht der Laufzeitfehler „Heap Overflow“.

Diese Verfahren und ihre Vor- und Nachteile werden in Büchern zu Datenstrukturen, z. B. (Aho, Hopcroft und Ullman 1983, Kap. 12), oder auch zu Betriebssystemen genauer diskutiert. In Betriebssystemen entsteht dieses Problem bei segmentierter Speicherung, wenn Programmteile variabler Größe eingelagert werden können.

6.2 3-Adress-Code: Eine Zwischensprache

6.2.1 Zwischendarstellungen

In der Einführung (Kapitel 1) wurde das *Erzeugen von Zwischencode* als eine der Übersetzungsphasen – nach lexikalischer und Syntaxanalyse – beschrieben. Vorteil gegenüber dem direkten Übersetzen in Maschinensprache ist das noch etwas höhere Sprachniveau und das Abstrahieren von Details der speziellen Zielmaschine. Darüber hinaus ist es möglich, ein einziges *Compiler-Frontend* zu schreiben, das Zwi-

schencode erzeugt, und dahinter verschiedene *Backends* zu schalten, die aus dem Zwischencode Code für unterschiedliche Zielmaschinen generieren.

Die Bezeichnung „Zwischencode“ oder „Zwischensprache“ ist eigentlich etwas zu speziell: Es geht um eine geeignete Darstellung des Programms „auf halbem Wege“ zwischen Quellprogramm und Zielprogramm, also als Ergebnis der lexikalischen und der Syntaxanalyse sowie ggf. daran angeschlossener weiterer Analyse- (semantische Analyse, Typprüfung) und Übersetzungsaktionen. Einige bekannte Darstellungen sind:

- (abstrakte) Syntaxbäume,
- gerichtete azyklische Graphen (*DAG = directed acyclic graph*),
- Postfix-Notation,
- 3-Adress-Code.

Syntaxbäume. Den Begriff des Syntaxbaums haben wir schon in Kapitel 3 (Definition 3.5) als Synonym zu Ableitungsbaum eingeführt. Diese müsste man nun als „konkrete“ Syntaxbäume bezeichnen. Als Zwischendarstellung verwendet man „abstrakte“ Syntaxbäume. Das sind etwas vereinfachte Darstellungen konkreter Syntaxbäume, bei denen die Struktur weniger an grammatischen Kategorien (Nichtterminalen) als an der Bedeutung des Konstrukts, d. h. den durchzuführenden Operationen orientiert ist.

Beispiel 6.2: Eine Zuweisung

```
x := (a + b) * ((a + b)/2)
```

würde so dargestellt, wie in Abb. 6.12 gezeigt. □

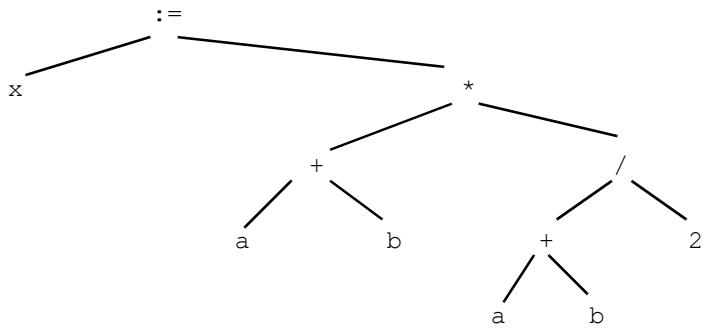


Abb. 6.12. Abstrakter Syntaxbaum für Zuweisung

DAGs, gerichtete azyklische Graphen. Diese sind eng verwandt mit Syntaxbäumen, unterscheiden sich aber darin von ihnen, dass im Syntaxbaum mehrfach auftretende Teilstrukturen nur einmal vorkommen. Ein DAG für die Zuweisung in Abb. 6.12 würde so aussehen (Abb. 6.13):

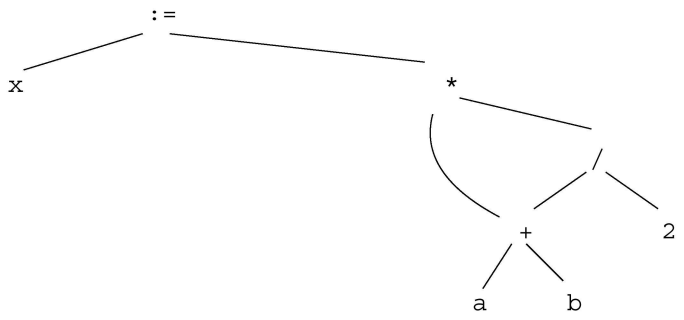


Abb. 6.13. DAG-Darstellung der Zuweisung

Der Vorteil liegt darin, dass am Ende weniger Code erzeugt wird. Normalerweise wird für jeden Teilbaum ein Code-Fragment erzeugt; bei mehrfach benutzten Teilstrukturen im DAG entsteht dann nur ein Code-Fragment.

Postfix-Notation. Wir kennen Postfix-Notation schon aus Beispiel 4.5 als Darstellung für arithmetische Ausdrücke. Die allgemeine Idee besteht darin, zunächst die Operanden und danach die auszuführende Operation hinzuschreiben. Die Zuweisung aus Beispiel 6.2 sähe damit so aus:

```
x a b + a b + 2 / * :=
```

Postfix-Notation lässt sich ausbauen zu einer Zwischensprache für eine abstrakte „Stack-Maschine“. Operanden werden jeweils auf den Stack geladen. Operationen entnehmen die obersten Elemente auf dem Stack als Argumente und legen ihr Ergebnis wieder auf den Stack. Code für eine Stack-Maschine für den obigen Ausdruck könnte dann so aussehen. Wir zeigen parallel zum Programm den Ablauf der Berechnung auf dem Stack.

Code	*Stack*	*Kommentar*
`lvar x`	`[x]`	lade die Adresse von *x*
`var a`	`[x] a`	lade den Wert von *a*
`var b`	`[x] a b`	
`+`	`[x] a+b`	
`var a`	`[x] a+b a`	
`var b`	`[x] a+b a b`	
`+`	`[x] a+b a+b`	
`const 2`	`[x] a+b a+b 2`	lade Konstante 2
`/`	`[x] a+b (a+b)/2`	

`*`	`[x]`	`(a+b)*(a+b)/2`
`:=`		speichere den Wert im obersten Element an die Adresse im zweitobersten Element

3-Adress-Code. Auch 3-Adress-Code ist uns schon begegnet, und zwar im Überblick im ersten Kapitel. 3-Adress-Code (kurz: 3AC) hat Befehle mit bis zu drei Argumenten, also lässt sich

```
x := a + b
```

in einem einzelnen Befehl darstellen. Die Argumente sind dabei (Adressen von) Variablen oder Konstanten. Wie in Assemblersprachen können Befehle mit Sprungmarken versehen werden, z. B.

```
L1:     x := y[i]
```

Gegenüber Postfix-Code hat 3AC den Vorteil, dass sich Befehlsfolgen leichter umordnen lassen, da mit expliziten Variablen gearbeitet wird, während man bei Postfix-Notation aufpassen muss, was gerade auf dem Stack steht. Darüber hinaus kann man sich auch nach Erzeugen des Codes noch entschließen, bestimmten Variablen Maschinenregister zuzuordnen und damit Speicherzugriffe einzusparen. Beide Aspekte zusammen machen 3AC besonders geeignet für eine anschließende Optimierungsphase (vgl. Kapitel 8).

Wir werden im Folgenden 3-Adress-Code genauer betrachten und ihn in Abschnitt 6.3 als Zielsprache benutzen. 3AC kann etwa folgende Klassen von Befehlen enthalten, die wir hier zunächst informal beschreiben (die gezeigte Wahl entspricht [Aho et al. 2006]).

`1. x := y` *op* `z`	Hier ist *op* ein binärer Operator, z. B. +, *, *and* usw. Diese Klasse enthält für jeden solchen Operator einen Befehl. Die Werte von *y* und *z* werden mit *op* verknüpft und der Variablen *x* zugewiesen.
`2. x :=` *op* `y`	Analog, nur ist *op* ein unärer Operator (z. B. *not*)
`3. x := y`	einfache Zuweisung
`4. goto L`	Sprung zum 3-Adress-Befehl mit Sprungmarke *L*
`5. if x` *cop* `y goto L`	Hier ist *cop* ein Vergleichsoperator, z. B. =, <, ≤, ≠ usw. Bedingter Sprung nach *L*, falls der Vergleich von *x* und *y* mit *cop true* ergibt. Auch hier gibt es einen Befehl für jeden Operator *cop*.

6. `x := y[i]` `x[i] := y`	Indizierte Zuweisungen zur Übersetzung von Array-Zugriffen. Der erste Befehl weist *x* den Wert der Speicherzelle zu, die *i* Speicherzellen hinter *y* liegt. Der zweite weist der Speicherzelle, die *i* Zellen hinter *x* liegt, den Wert von *y* zu.

7. `x := &y`
`x := *y`
`*x := y`
	Befehle zur Manipulation von Zeigervariablen. Der erste weist *x* die Adresse von *y* zu. Der zweite Befehl weist *x* den Wert der Speicherzelle zu, deren Adresse in *y* steht. Der dritte Befehl weist der Speicherzelle, deren Adresse in *x* steht, den Wert von *y* zu (Notationen wie in der Sprache C).

8. `param x`
`call p`
`return y`
	Diese Befehle dienen der Übersetzung von Prozeduraufrufen. Ein Aufruf $p(x_1, \ldots, x_n)$ wird übersetzt in eine Folge von Befehlen
	`param x1`
	`...`
	`param xn`
	`call p`
	Der *return*-Befehl steht im Code für die Prozedur und bewirkt Rückkehr aus der Prozedur. Das Argument *y* ist optional und dient der Rückgabe des Ergebnisses in Funktionsprozeduren.

6.2.2 Eine abstrakte Maschine für 3-Adress-Code

Ziel dieses Abschnitts ist es, für die Befehle des 3-Adress-Codes eine präzise Bedeutung festzulegen. Man muss sich darüber im Klaren sein, dass die Semantik von 3AC nicht allgemein fixiert ist. Denn beim Entwurf der Speicherverwaltung und des Laufzeitsystems hat man viele Freiheitsgrade, und man muss diese Entscheidungen schon in Abhängigkeit von der zu implementierenden Sprache treffen. Beispielsweise benötigt man zur Übersetzung der Sprache C den ganzen komplizierten Mechanismus zur Behandlung statischer Schachtelungen (z. B. Display) nicht. Selbst wenn man statische Schachtelungen unterstützen muss, hat man die Wahl, Zeiger auf statische Vorgänger zu verwalten oder die Display-Technik zu benutzen. Was genau bei der Auswertung eines Befehls im 3AC geschieht, insbesondere bei Prozeduraufrufen, hängt aber von diesen Entscheidungen ab.

Wir werden also jetzt eine Reihe solcher Entscheidungen treffen und damit eine konkrete Speicherorganisation festlegen. Anschließend beschreiben wir die Bedeutung der Befehle des 3AC in Bezug auf dieses Modell. Damit sollte es möglich sein, einen Interpreter für 3-Adress-Code zu schreiben. Wir bekämen dann eine abstrakte Maschine zur Auswertung von 3-Adress-Code (vgl. Abschnitt 1.4). Alternativ kann

man 3-Adress-Programme natürlich optimieren und daraus Maschinencode erzeugen, gemäß der in Kapitel 8 beschriebenen Vorgehensweise.

Das Maschinenmodell

Wir nehmen an, dass wir statisch geschachtelte Prozedurdeklarationen unterstützen müssen und wählen dazu die Display-Technik. Die Speicherorganisation sei so wie in Abb. 6.14 gezeigt.

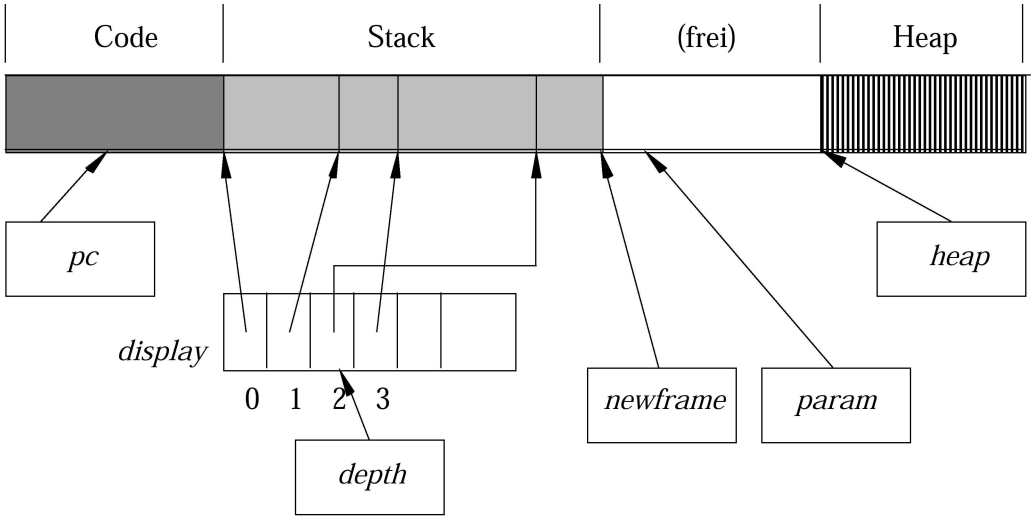


Abb. 6.14. Speicherverwaltung mit dazugehörigen Registern für die 3-Adress-Maschine

Wir verwenden eine Reihe von Registern, nämlich:

- *pc* zeigt auf die Darstellung des aktuellen 3AC-Befehls im Code-Speicher.
- *display* ist ein ganzer Satz von Registern, die auf die Anfänge von Prozedurrahmen zeigen.
- *depth* gibt die (statische) Tiefe der gerade aktiven Prozedur an.
- *newframe* und *param* zeigen auf den Beginn des freien Speichers bzw. in diesen hinein; die genaue Verwendung wird im Zusammenhang mit Prozeduraufrufen und Rückkehr daraus klar werden.
- *heap* zeigt auf den Anfang des Heap-Bereichs.

Es gibt keinen besonderen statischen Speicherbereich; seine Rolle wird vom mit *display*[0] adressierten Prozedurrahmen übernommen.

Den Speicher insgesamt bezeichnen wir mit *STORE*. Wir nehmen an, dass einzelne Bytes adressierbar sind; *STORE*[*i*] bezeichnet also das Byte an der Position *i*. Die größten Einheiten, für die – nach der Architektur der Zielmaschine – Alignment zu

beachten ist, seien 8 Bytes lang (z. B. *double* oder *longreal*-Typen). Um mit dem Alignment keine Probleme zu bekommen, lassen wir alle Speicherbereiche (Code, Prozedurrahmen, Heap) auf durch 8 teilbaren Speicheradressen beginnen. Bei der Vergabe relativer Adressen innerhalb dieser Speicherbereiche können wir dann das Alignment beachten (vgl. Abschnitt 6.1.2).

3-Adress-Programme

Programme im 3-Adress-Code sollen in diesem Modell interpretierbar, also durch einen Interpreter auswertbar, sein. Das hat zur Folge, dass ein Programm nicht nur eine Folge von 3AC-Befehlen ist, vielmehr sind einige Symboltabellen *integraler Bestandteil des Programms*. Der Interpreter benutzt diese Tabellen bei der Auswertung. Bei einer übersetzenden Weiterverarbeitung des 3-Adress-Programms würden diese Tabellen in der Optimierung und Codeerzeugung zur Verfügung stehen und die entsprechende Information in das Zielprogramm eingebaut werden, so dass dann zur Laufzeit natürlich keine Tabellen mehr benötigt würden.

Wir benutzen folgende Symboltabellen:

Variables & Constants

	type	*name*	*s_depth*	*offset*	*size*	*value*	*alignm.*	*typeindex*
1	var	x	2	44	4	–	4	1
2	const	–	0	12	4	3	4	1
3	var	field	0	168	192		8	14
	…							

Diese Tabelle enthält eine Zeile für jede im 3-Adress-Programm vorkommende Variable oder Konstante. Wir haben oben gesagt, dass Argumente im 3AC Variablen oder Konstanten sein können. Wir können das jetzt präzisieren: Argumente sind Indizes in diese Tabelle der Variablen und Konstanten. Der Befehl

```
x := x + 3
```

würde also intern dargestellt als

```
([:= +], 1, 1, 2)
```

wobei [:= +] der Befehlscode sein soll (eine ganze Zahl). Tatsächlich behandeln wir Konstanten in diesem Modell wie Variablen. Das heißt, wir schreiben ihre Werte vor Beginn der Interpretation in den statischen Speicher unter *display*[0] und greifen darauf zu wie auf Variablen; die Werte ändern sich dann allerdings nicht zur Laufzeit. Die Einträge in der Tabelle haben folgende Bedeutung.

- *type*: Variable oder Konstante.
- *name*: Name der Variablen. Verschiedene Variablen im Programm können den gleichen Namen haben (wenn sie in unterschiedlichen Scopes deklariert sind), der Index in diese Tabelle identifiziert die Variable eindeutig.
- *s_depth*: Die Variable oder Konstante hat statische Tiefe *s_depth*, gehört also zu einem Prozedurrahmen, der über *display*[*s_depth*] adressiert wird.
- *offset*: Beginn der Darstellung innerhalb des Prozedurrahmens.
- *size*: Größe in Bytes.
- *value*: Wert für Konstante.
- *alignment*: Darstellung muss auf durch *alignment* teilbarer Position beginnen.
- *typeindex*: Index in eine Tabelle von Typbeschreibungen (die in Abschnitt 6.3.1 erklärt wird). Gibt den Typ für diese Variable oder Konstante an.

Während der Interpretation werden die Werte für *s_depth* und *offset* benötigt, um die Adresse zu ermitteln. Das geschieht mit der *Adressfunktion*:

```
adr(v) = display[s_depth(v)] + offset(v)
```

Dabei bezeichnet *v* den Index der Variablen oder Konstanten (die Zeile in der Tabelle), und wir notieren z. B. mit *offset*(*v*) den Zugriff auf den Wert in der Spalte *offset*.

Eine zweite Tabelle verwaltet Sprungmarken im 3-Adress-Programm:

Labels

	label	*index*
1	L	432
2	kappa	1318
	…	

Damit ist z. B. 432 die Adresse des Befehls mit Sprungmarke *L* im Code-Speicher, genauer die Adresse des ersten Bytes der Befehlsdarstellung. Schließlich verwaltet eine dritte Tabelle Information über Prozeduren:

Procedures

	name	*static_depth*	*static_size*	*start*	*typeindex*
1	p	2	48	624	4
	…				

Hier bezeichnet *static_depth* die statische Schachtelungstiefe der Prozedur, *static_size* die Größe ihres Prozedurrahmens einschließlich Verwaltungsinformation und Platz für lokale und temporäre Variablen, aber ohne ggf. noch anzulegende dynamische Arrays. *Start* ist die Anfangsadresse im Code-Speicher. *Typeindex* ist

ein Index in eine Typtabelle (Abschnitt 6.3.1) und beschreibt den Ergebnistyp für Funktionsprozeduren.

Semantik von 3-Adress-Befehlen

Damit sind wir in der Lage, die Ausführung von 3-Adress-Befehlen präzise zu beschreiben. Wir erweitern den Befehlssatz noch geringfügig gegenüber dem in Abschnitt 6.2.1 vorgestellten Konzept.

```
1. x := y op z
        STORE[adr(x)] := STORE[adr(y)] op STORE[adr(z)];
        next(pc)
   x :- y op z
        STORE[adr(x)] :- STORE[adr(y)] op STORE[adr(z)];
        next(pc)
```

Wir erlauben in allen Befehlen zwei Varianten von Zuweisungen von einer Speicheradresse an eine andere. Die erste Variante, notiert mit „:=“, kopiert wortweise, also 4 Bytes (das erste Byte ist das adressierte). Die zweite Variante, notiert mit „:–“, kopiert nur genau das adressierte Byte. Im Folgenden geben wir die zweite Variante des Befehls jeweils in eckigen Klammern an, notieren aber die Bedeutung nicht noch einmal dafür. – Die Operation *next* erhöht den Programmzähler um die Anzahl von Bytes, die zur Darstellung eines Befehls benutzt werden; diese ist für alle Befehle gleich.

```
2. x := op y    [x :- op y]
        STORE[adr(x)] := op STORE[adr(y)]; next(pc)
3. x := y       [x :- y]
        STORE[adr(x)] := STORE[adr(y)]; next(pc)
4. goto L
        pc := index(L)
```

Die Position des Zielbefehls wird der Tabelle *Labels* entnommen.

```
5. if x cop y goto L
        if STORE[adr(x)] cop STORE[adr(y)] then pc := index(L)
        else next(pc) end
6. x := y[i]    [x :- y[i]]
        STORE[adr(x)] := STORE[adr(y) + STORE[adr(i)]];
        next(pc)
   x[i] := y    [x[i] :- y]
        STORE[adr(x) + STORE[adr(i)]] := STORE[adr(y)];
        next(pc)
7. x := &y
        STORE[adr(x)] := adr(y); next(pc)
   x := *y      [x :- *y]
        STORE[adr(x)] := STORE[STORE[adr(y)]]; next(pc)
   *x := y      [*x :- y]
        STORE[STORE[adr(x)]] := STORE[adr(y)]; next(pc)
   x := *y[i]   [x :- *y[i]]
        STORE[adr(x)] := STORE[STORE[adr(y) + STORE[adr(i)]]];
        next(pc)
```

```
*x[i] := y    [*x[i] :- y]
     STORE[STORE[adr(x)] + STORE[adr(i)]] := STORE[adr(y)];
     next(pc)
```

Hier haben wir die Gruppe 7 erweitert um „indirekt indizierte“ Varianten der Befehle, die wir bei der Übersetzung von Zeigerdereferenzierungen benötigen. – Die nun folgenden Befehle der Gruppe 8 zur Behandlung von Prozeduraufrufen modifizieren wir etwas gegenüber dem allgemeinen Konzept. Es gebe folgende Befehle:

```
8. refparam x          return
   valparam x          freturn y
   call p
   getresult y
```

Neu ist, dass wir zwei Arten der Parameterübergabe unterscheiden, nämlich *call-by-reference* (mittels *refparam*) und *call-by-value* (mittels *valparam*). Weiterhin führen wir separate Befehle für Rücksprung aus einer Prozedur oder aus einer Funktion ein sowie einen Befehl zum Empfang des Ergebnisses einer Funktion.

Wir wählen die in Abb. 6.15 gezeigte Struktur von Prozedurrahmen:

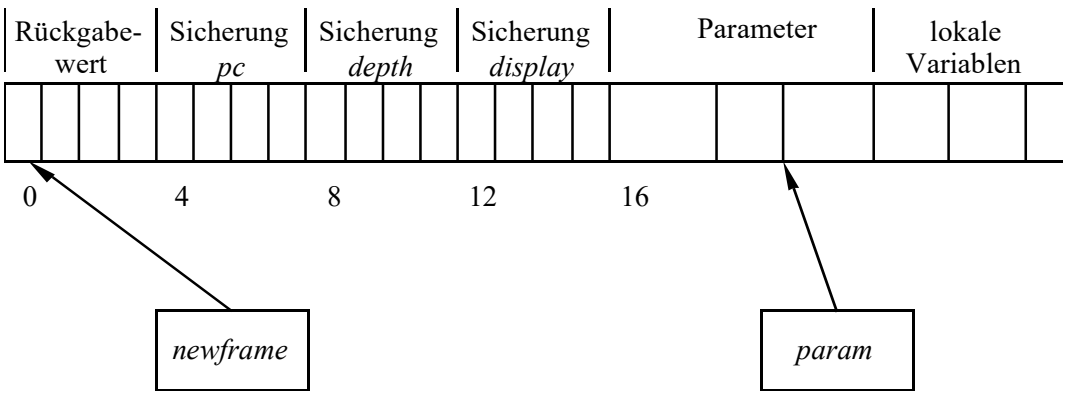


Abb. 6.15. Detaillierte Struktur von Prozedurrahmen für die 3-Adress-Maschine

Der Prozedurrahmen beginnt mit 4 Feldern zu jeweils 4 Bytes für den Rückgabewert und zur Sicherung von Registern. Es folgen dann Bereiche für Parameter und lokale Variablen. Der Zeiger (im Register) *newframe* zeigt jeweils auf den Anfang des freien Speichers oberhalb des Stacks, also an die Stelle, an der ein neuer Prozedurrahmen beginnen müsste, und innerhalb des neuen Prozedurrahmens damit auf den Rückgabewert. Hier wird angenommen, dass der Rückgabewert in einem Speicherwort (4 Bytes) darstellbar ist. Das ist eine in vielen Programmiersprachen übliche Annahme; damit kann man entweder eine Zahl oder einen Zeiger auf eine Struktur zurückgeben. Der Zeiger *param* zeigt vor Beginn der Parameterübergabe auf den Beginn des Parameterbereichs, also auf Relativposition 16. Mit jedem *valparam* oder *refparam*-Befehl wird dieser Zeiger auf die nächste mit einem Parameter zu

belegende Position weitergesetzt. – Damit können wir zunächst die Bedeutung der beiden *param*-Befehle angeben:

```
refparam x
     align(param, 4)
     STORE[param] := adr(x); param := param + 4; next(pc)
```

Adressen haben, wie schon erwähnt, Länge 4. Die Operation *align*(*param*, *j*) erhöht den Wert von *param* soweit, dass er durch *j* teilbar ist, ist also so definiert:

```
align(m, n) ==
        if (m mod n) ≠ 0 then m := m + n - (m mod n) end;
```

Bei *call-by-value* wird jeweils der Wert, nicht die Adresse der Variablen kopiert; hier muss die Länge der Darstellung beachtet werden.

```
valparam x
     align(param, alignment(x));
     for i := 0 to size(x) - 1 do
       STORE[param + i] :- STORE[adr(x) + i]
     end;
     param := param + size(x); next(pc)
```

Beachten Sie, dass in der Schleife die Darstellung des Parameters byteweise kopiert wird. Innerhalb der Prozedur wird auf die Parameter zugegriffen wie auf lokale Variablen. Beim Übersetzen der Prozedur werden entsprechende Einträge in der Variablentabelle erzeugt. Dabei müssen natürlich die gleichen Kriterien für das Alignment benutzt werden wie im hier gezeigten Code.

Die Situation vor der Ausführung eines *call*-Befehls ist in Abb. 6.16 gezeigt. Hier bezeichnen α, β und γ Adressen. Die Parameter sind bereits kopiert worden. Anschließend werden folgende Aktionen durchgeführt:

```
call p
        next(pc); STORE[newframe + 4] := pc;
        STORE[newframe + 8] := depth; depth := static_depth(p);
        STORE[newframe + 12] := display[depth];
        display[depth] := newframe;
        newframe := newframe + static_size(p);
        if newframe >= heap then error("stack overflow") end;
        param := newframe + 16; pc := start(p)
```

Beim Auftreten des Fehlers „stack overflow“ wird die Ausführung des gesamten Programms abgebrochen. Nehmen wir an, dass die aufgerufene Prozedur *p* statische Tiefe 2 hat. Dann erhalten wir als Ergebnis der Ausführung des *call*-Befehls die Situation in Abb. 6.17.

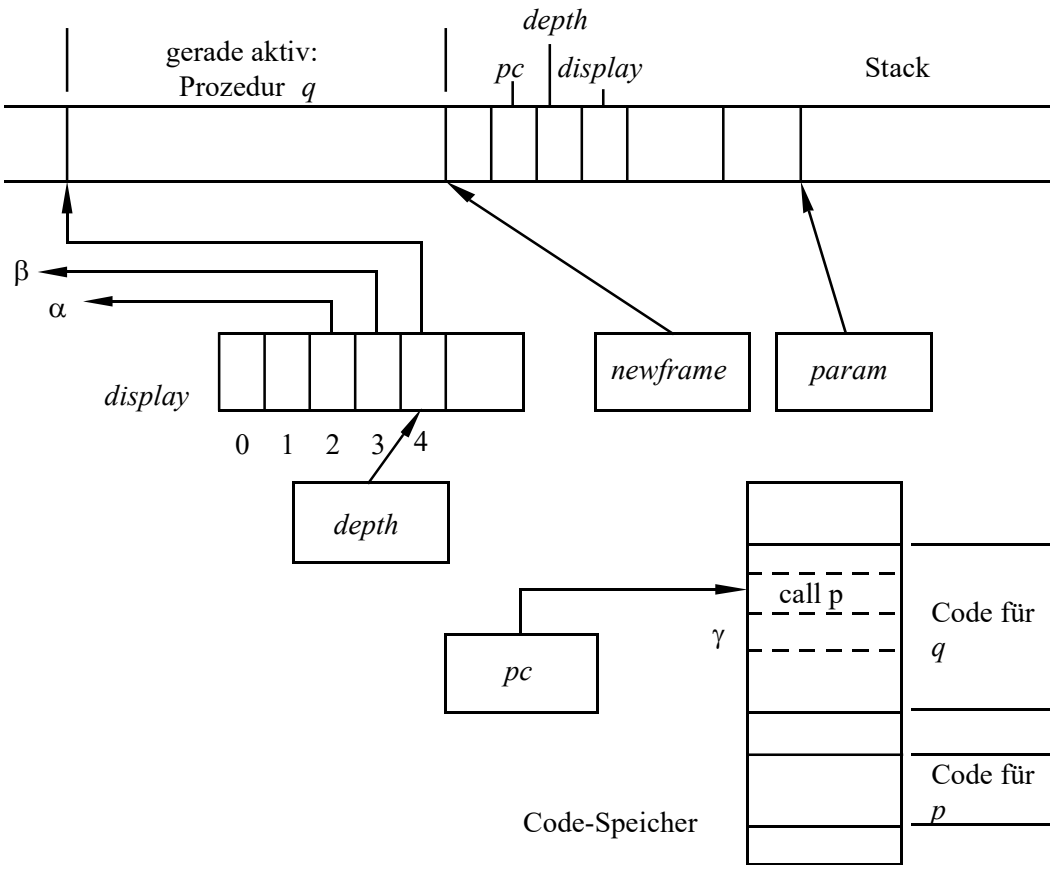


Abb. 6.16. Situation vor der Ausführung eines *call*-Befehls

Durch einen *return*-Befehl in *p* wird nun der vorherige Zustand wiederhergestellt. Wir betrachten zunächst die *freturn*-Variante:

```
freturn y
     STORE[display[depth]] := STORE[adr(y)];(*)
     newframe := display[depth]; param := newframe + 16;
     display[depth] := STORE[newframe + 12];
     depth := STORE[newframe + 8];
     pc := STORE[newframe + 4]
```

Überzeugen Sie sich, dass nun der Zustand in Abb. 6.16 wiederhergestellt ist, bis auf Folgendes: (i) Der von *p* berechnete Rückgabewert steht nun in *STORE*[*newframe*], und (ii) der Programmzähler zeigt auf Position γ, also den Befehl nach *call p*.

Die Bedeutung des *return*-Befehls ist fast die gleiche:

```
return
     {wie freturn y, nur ohne die Anweisung (*)}
```

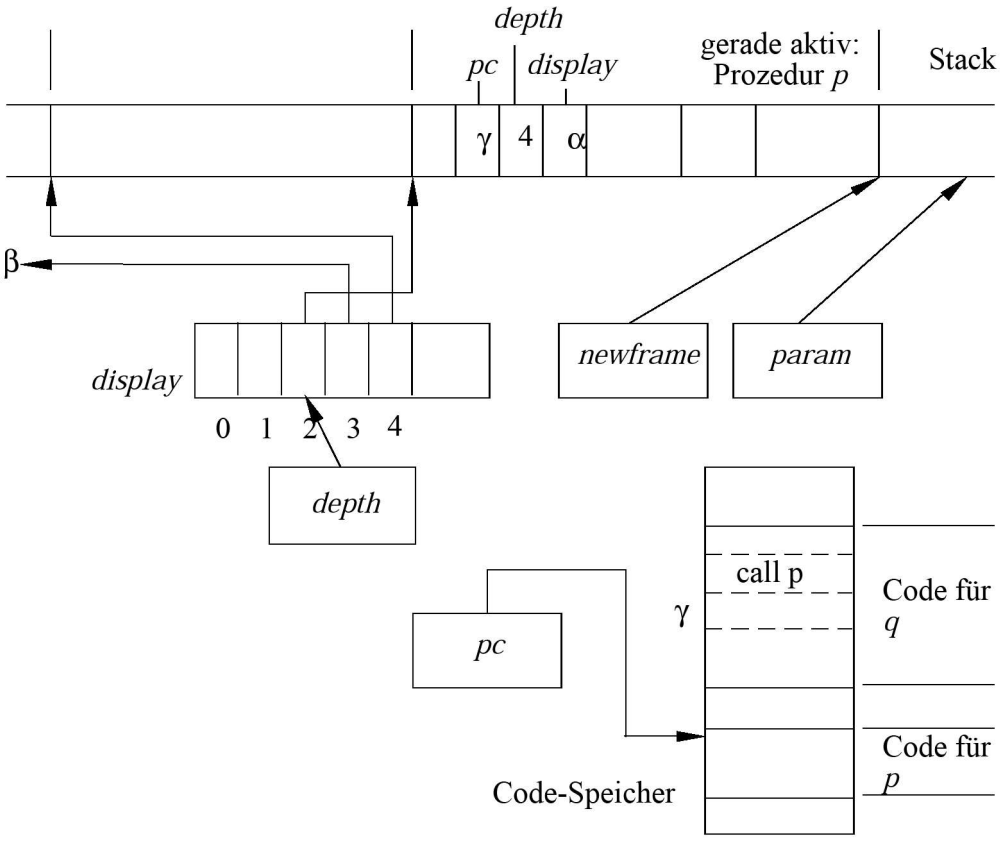


Abb. 6.17. Situation nach der Ausführung des *call*-Befehls

Der Befehl *getresult* wird im Code des Aufrufers plaziert, um das Ergebnis eines Funktionsaufrufs zu erhalten. Der Befehl kopiert 4 Bytes; ggf. muss im Aufrufer Code stehen, der ein einzelnes Byte extrahiert.

```
getresult y
     STORE[adr(y)] := STORE[newframe]; next(pc)
```

Damit haben wir eine vollständige, präzise Beschreibung aller mit Prozeduraufrufen und Prozedurrückkehr verbundenen Aktionen erhalten. Wir führen noch drei neue Gruppen von Befehlen ein. Die erste Gruppe dient der Vergabe von Speicherplatz auf dem Stack:

```
9. init_stack n
       newframe := display[0] + STORE[adr(n)];
       if newframe >= heap then error("stack overflow") end;
       param := newframe + 16; next(pc)
```

```
extend_stack x, n
    STORE[adr(x)] := newframe;
    newframe := newframe + STORE[adr(n)];
    if newframe >= heap then error("stack overflow") end;
    param := newframe + 16; next(pc)
```

Der Befehl *init_stack* erlaubt die Initialisierung der Register *newframe* und *param* gemäß der Größe des Hauptprogramms, die in der Variablen *n* angegeben wird. Der Befehl *extend_stack* allokiert Speicherplatz auf dem Stack (indem *newframe* und *param* „weitergeschoben“ werden). In *n* wird der Platzbedarf angegeben, *x* wird ein Zeiger auf den Speicherblock zugewiesen. Damit können lokale, dynamische Arrays in Prozeduren angelegt werden.

Gruppe 10 enthält Befehle für die dynamische Speichervergabe auf dem Heap:

```
10. alloc x, n
        STORE[adr(x)] := {Adresse eines Speicherblocks der
        Größe n im Heapbereich}; next(pc)
    dealloc x, n
        {der Speicherblock der Größe n, dessen Adresse in
        STORE[adr(x)] steht, wird freigegeben}; next(pc)
```

Dies sind die einzigen Befehle, bei denen wir die Bedeutung nicht vollständig festlegen, da wir sonst ein Programm für die Heap-Verwaltung angeben müssten, was hier zu aufwendig ist. – Schließlich enthält Gruppe 11 noch einen Befehl:

```
11. noop
        next(pc)
```

Dieser Befehl tut nichts. Bei der Übersetzung ist es aber praktisch, ihn zu haben; er wird beim Erzeugen von Sprungmarken benutzt, wenn der Folgebefehl noch nicht bekannt ist.

Schließlich müssen wir noch den Anfangszustand der 3-Adress-Maschine beim Start eines Programms angeben. Die Register haben folgende Werte: *pc* = 0, *depth* = 0, *display*[0] zeigt auf die erste durch 8 teilbare Adresse nach dem letzten Befehl des Code-Speichers, *newframe* und *param* sind undefiniert, und *heap* = *STOREMAX* (die letzte Adresse im Speicher). – Damit ist unsere abstrakte Maschine für 3-Adress-Code vollständig beschrieben.

6.3 Übersetzung in 3-Adress-Code

In diesem Abschnitt betrachten wir die Übersetzung der wichtigsten Sprachkonzepte imperativer Sprachen in 3-Adress-Code. Wir entwerfen dazu eine Beispielsprache, die sich im Wesentlichen an PASCAL oder Modula-2 anlehnt, in Einzelheiten aber davon abweicht. Die Abweichungen liegen darin begründet, dass wir die Übersetzung möglichst einfach halten wollen.

Wir zerlegen das Problem in vier Teilprobleme, nämlich Übersetzung von Deklarationen, von Zuweisungen und Ausdrücken, von Kontrollstrukturen und schließlich von Prozeduraufrufen. Dies sind die Themen der folgenden Unterabschnitte.

6.3.1 Deklarationen

Wir beginnen mit dem Entwurf der Grammatik für unsere Sprache. Die Sprache soll geschachtelte Prozedurdeklarationen erlauben und Arrays, Records und Zeigertypen besitzen.

```
program ->      types vars procs stmt .
```

Ein Programm ist eine Folge von Typdeklarationen, Variablendeklarationen, Prozedurdeklarationen und einer Anweisung, gefolgt von einem Punkt. Die strikte Reihenfolge der Deklarationen ist wie in PASCAL. Dort gibt es noch Konstantendeklarationen, auf die wir hier verzichten.

Zur Notation: Wir beschreiben die Grammatik und später Übersetzungsschemata angelehnt an Yacc-Notation, verzichten allerdings darauf, Terminalsymbole in Anführungszeichen zu setzen. Die Zeichenfolgen „->“ (nur direkt nach dem Nichtterminal der linken Seite) und „|“ gehören zur „Metasprache“ (Produktionspfeil und Trennung von Alternativen); alle Sonderzeichen auf rechten Seiten sind Terminalsymbole. Wörter beschreiben Symbole der Grammatik; Wörter für Terminalsymbole kennzeichnen wir durch Fettdruck.

```
types   ->      type id = typeexpr ; types
        |
```

Eine leere rechte Seite entspricht einer Ableitung ins leere Wort ε.

```
vars    ->      var decls ;
        |

decls   ->      decl
        |       decls ; decl

decl    ->      id : typeexpr

typeexpr ->     simpletype
        |       array [ intconst ] of typeexpr
        |       record decls end
        |       pointer to typeexpr
        |       typeident

simpletype ->   integer
        |       real
        |       boolean
        |       char

typeident ->    id
```

Die Sprache besitzt also vier atomare Datentypen, Zeiger, Records und der Einfachheit halber nur statische Arrays, d. h. die Arraygröße muss als Konstante zur Übersetzungszeit bekannt sein. Arrays werden jeweils eindimensional durch Angabe der Anzahl der Elemente deklariert (wie in C); es gibt folglich keine Untergrenzen (das erste Element wird mit Index 0 angesprochen). Wie man sieht, lassen sich aber Array- und Record-Deklarationen beliebig schachteln, so dass auch mehrdimensionale Arrays deklariert werden können.

```
procs    ->      proc procs
         |       function procs
         |

proc     ->      procedure id parameters ;
                 vars procs stmt ;

parameters ->    ( params )
         |

params   ->      param
         |       params ; param

param    ->      var id : paramtype
         |       id : paramtype

paramtype ->     simpletype
         |       typeident

function ->      function id ( params ) : paramtype ;
                 vars procs stmt ;
```

Eine Funktion muss Parameter haben; bei Prozeduren sind sie optional. Parameterübergabe erfolgt mit call-by-reference (mit **var**) oder call-by-value. Wenn man komplexe Typen als Parameter übergeben will, benötigt man eine explizite Typdeklaration. Typen können übrigens nur global deklariert werden, nicht innerhalb von Prozeduren.

Globale Symboltabellen

Bevor wir mit der Übersetzung beginnen, müssen wir Klarheit über die „Infrastruktur“ gewinnen, d. h. über vorhandene Symboltabellen, globale Variablen, Hilfsfunktionen usw. Es gebe vier globale Symboltabellen, nämlich:

- Variables & Constants
- Labels
- Procedures
- Types

Die ersten drei kennen wir schon, da sie in der abstrakten 3-Adress-Maschine auch zur Laufzeit benötigt werden. Die Tabelle über Typen braucht man nur während der Übersetzung. Dort soll für jede Variable und jede Typdeklaration der Typ in „aus-

wertbarer“ Form dargestellt sein. Auswertbar heißt, dass man etwa für eine Array-Variable nachsehen kann, wie viele Komponenten der Array hat, oder für einen Record einen Feld-Offset auffinden kann. Die Tabelle sieht so aus:

Types

	type	*name*	*nocomps*	*compsize*	*compindex*	*fieldtable*
1	integer					
2	real					
3	boolean					
4	char					
5	array		50	4	2	
6	array	matrix	20	200	5	
7	record					→
8	pointer	matpointer			6	
9	integer	number				
	…					

Mit Hilfe der Tabelle lässt sich jeder im Programm vorkommende Typ durch eine ganze Zahl darstellen, nämlich durch einen Index in diese Tabelle. Ein Typ kann dabei explizit deklariert sein oder implizit durch eine Variablendeklaration erzeugt werden. Jede Anwendung eines Konstruktors (array, record, pointer) auf einen gegebenen Typ erzeugt einen neuen Typeintrag in der Tabelle. Die Zeilen 5 und 6 in der obigen Tabelle sind z. B. durch eine Typdeklaration

```
type matrix = array [20] of array [50] of real;
```

zustandegekommen. Zeilen 8 und 9 entsprechen den darauffolgenden Deklarationen:

```
type matpointer = pointer to matrix;
type number = integer;
```

Die Felder der Tabelle haben folgende Bedeutung. *Type* enthält einen Grundtyp oder einen der Typkonstruktoren *array*, *record*, *pointer*. Die vier Grundtypen sind fest in die Tabelle eingetragen. Das Feld *name* enthält einen Typnamen, falls der Typ explizit deklariert wurde. *Nocomps* und *compsize* enthalten für Arrays die Anzahl der Komponenten sowie die Größe einer Komponente in Bytes. *Compindex* enthält für konstruierte Typen *array* und *pointer* den Typindex des Komponententyps. Schließlich enthält *fieldtable* für einen Recordtyp einen Verweis auf eine Tabelle von Feldnamen, deren Struktur wir unten beschreiben.

Für die drei Tabellen für Variablen, Prozeduren und Typen gibt es jeweils eine Funktion *enter_var*, *enter_proc*, bzw. *enter_type*, die einen neuen Eintrag in der Tabelle erzeugt. Die Funktion vergibt jeweils den nächsten freien Index und liefert diesen zurück. Felder, in die nichts einzutragen ist, werden beim Aufruf durch „–“ gekennzeichnet. Der Eintrag in Zeile 5 wäre z. B. erzeugt worden durch einen Aufruf:

```
index := enter_type(array, -, 50, 4, 2, -)
```

und hätte den Wert 5 zurückgegeben.

Namenstabellen

Neben den vier globalen Tabellen benutzen wir einen *Stack von Namenstabellen*. Jede Tabelle enthält Paare der Form (*name*, *index*), wobei *name* ein im Programm deklarierter Variablen- oder Prozedurname ist und *index* der Index für die entsprechende Tabelle. Der Stack von Tabellen beschreibt die statische Schachtelung der Deklarationen entsprechend dem in Abschnitt 6.1 beschriebenen Sichtbarkeitskonzept. Zum Beispiel beim Übersetzen der Deklaration der Prozedur *B* innerhalb der Deklaration der Prozedur *A* innerhalb des Hauptprogramms gemäß Abb. 6.4 gäbe es einen Stack von Tabellen (Abb. 6.18, dieser Stack wächst nach rechts):

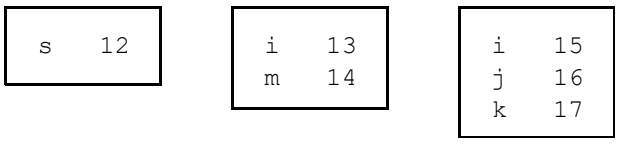


Abb. 6.18. Stack von Namenstabellen

Zum Arbeiten mit diesem Stack stellen wir folgende Prozeduren zur Verfügung:

```
pushnametable          enter_name(depth, name, index)
popnametable           v := lookup(name)
```

Die ersten beiden Prozeduren legen eine neue Tabelle auf den Stack bzw. löschen die oberste Tabelle. *Enter_name* trägt ein Paar in die Tabelle der Tiefe *depth* ein. Normalerweise ist *depth* die Tiefe der obersten (rechts außen stehenden) Tabelle; beim Übersetzen von Prozedurparametern benötigt man aber auch Einträge in die zweitoberste Tabelle. *Enter_name* löst automatisch eine Fehlerbehandlung aus, falls der einzutragende Name in dieser Tabelle bereits existiert. Um das Abfangen dieses Fehlers braucht man sich daher im Übersetzungsschema nicht zu kümmern. Die Funktion *lookup* durchsucht den Stack von Tabellen von oben nach unten, bis der Name *name* gefunden wird, und liefert dann den dazugehörigen Index zurück. Auch diese Funktion löst eine Fehlerbehandlung aus, falls der gesuchte Name nicht existiert.

Für die Verwaltung von Feldnamen in Records benutzen wir ebenfalls eine Namenstabelle, die genauso aussieht wie die Tabellen auf dem Stack. Diese Tabelle wird, wie oben erwähnt, über den *fieldtable*-Eintrag für den Recordtyp in der Typtabelle erreicht. Eintragen und Nachsehen von Namen erfolgt über Prozeduren

```
enter_fieldname(recordindex, name, index)
v := lookup_field(recordindex, name)
```

analog zu den obigen, wobei *recordindex* den Index in der Typtabelle bezeichnet. Auch diese Prozeduren sorgen selbst für Fehlerbehandlung. Man beachte, dass die zu einem Feldnamen zu verwaltenden Informationen wie für Variablen in der Variablentabelle vermerkt werden; deshalb genügen hier einfache Namenstabellen. In der Variablentabelle gibt es jetzt folgende mögliche Einträge in das *type*-Feld, wobei die letzten beiden zur Beschreibung von Prozedurparametern dienen:

```
const                   refparam
var                     valparam
recordfield
```

Globale Variablen und Hilfsfunktionen

Es gibt drei globale Variablen:

```
depth
prog_counter
temp_offset
```

Depth bezeichnet die aktuelle Schachtelungstiefe beim Übersetzen von Deklarationen. *Prog_counter* ist die Adresse des nächsten freien Speicherplatzes für einen 3-Adress-Befehl beim fortlaufenden Erzeugen und Eintragen von Befehlen. *Temp_offset* bezeichnet das aktuelle Offset in Prozedurrahmen beim Belegen von Speicherplatz für temporäre Variablen. Die globalen Variablen werden z. T. von folgenden Hilfsfunktionen manipuliert:

```
putcode(command)
v := newtemp(typeindex)
l := newlabel()
```

Putcode erhält als Parameter eine Beschreibung eines 3-Adress-Befehls und schreibt diesen an die Position *prog_counter* des (ggf. fiktiven) Befehlsspeichers. Falls der Befehl einen Label enthält, wird der aktuelle Wert des Programmzählers in die Labeltabelle eingetragen. Anschließend wird *prog_counter* um die zur Darstellung eines Befehls nötige Anzahl von Bytes erhöht. Die Funktion *newlabel*() liefert einfach den nächsten freien Index in der Labeltabelle und erhöht diesen. Zum Beispiel hat die Folge von Anweisungen

```
label := newlabel();
putcode(label ':' var1 ':=' var2 '+' '1');
```

bei aktuellem Stand 496 des Programmzählers und bei freiem Labelindex 70 folgenden Effekt: Zunächst wird in Zeile 70 der Labeltabelle ein Paar (L70, –) einge-

tragen.[1] Anschließend wertet der *putcode*-Befehl alle Argumente aus, die nicht in Anführungszeichen stehen, erhält also für *label* den Wert 70 und für *var1* und *var2* Indizes in die Variablentabelle (sagen wir 9 und 10). Nehmen wir weiterhin an, dass 110 der Befehlscode für den Befehl [:= +] ist. Dann wird zunächst 1 als Konstante in die Variablentabelle eingetragen (z. B. unter Index 22). Anschließend wird ein Befehl

```
(110, 9, 10, 22)
```

an Position 496 in den Befehlsspeicher geschrieben und *prog_counter* entsprechend erhöht. Schließlich wird in Zeile 70 der Labeltabelle das Paar (L70, 496) geschrieben.

Die Funktion *newtemp* erzeugt einen Eintrag für eine neue (temporäre) Variable in der Variablentabelle und gibt diesen zurück. Dabei wird Speicherplatz für die Variable an der Position *temp_offset* im gerade bearbeiteten Prozedurrahmen reserviert und *temp_offset* (unter Beachtung von Alignment, siehe unten) erhöht. Der Parameter *typeindex* gibt den Index des Typs in der Typtabelle an. Temporäre Variablen werden nur für die atomaren oder für Zeigertypen angelegt, belegen also entweder 1 Byte oder 4 Bytes.

Übersetzung von Deklarationen

Wir beginnen mit der Übersetzung von Typausdrücken. Für Typen sind drei Informationen relevant, nämlich die Größe der Typdarstellung (also der benötigte Speicherplatz), das Alignment (auf welchen Positionen darf eine Variable dieses Typs plaziert werden) und der Index des Typs in der Typtabelle. Daher verwalten wir drei Attribute *size*, *alignment* und *typeindex* für die entsprechenden Grammatiksymbole.

```
simpletype ->   integer          {simpletype.size := 4;
  size                           simpletype.alignment := 4;
  alignment                      simpletype.typeindex := 1}
  typeindex
        |       real             {simpletype.size := 4;
                                 simpletype.alignment := 4;
                                 simpletype.typeindex := 2}
        |       boolean          {simpletype.size := 1;
                                 simpletype.alignment := 1;
                                 simpletype.typeindex := 3}
        |       char             {simpletype.size := 1;
                                 simpletype.alignment := 1;
                                 simpletype.typeindex := 4}
```

Wir notieren in Übersetzungsschemata die zu einem Nichtterminal der linken Seite gehörenden Attribute, indem wir sie darunterschreiben. Im Folgenden fügen wir auch erklärende Absätze in Übersetzungsschemata ein.

---

[1] Die Darstellung „L70“ ist höchstens für eine Druckausgabe des 3-Adress-Programms von Bedeutung. Das *label*-Feld der Labeltabelle dient eigentlich der Verwaltung benutzerdefinierter Sprungmarken.

```
typeexpr ->     simpletype      {typeexpr.* := simpletype.*}
  size
  alignment
  typeindex
```

Die Notation *A*.* := *B*.* steht abkürzend dafür, dass alle Attributwerte von *B* den Attributen von *A* zugewiesen werden. Hier sind alle drei Attribute übrigens synthetisiert.

```
        |       pointer to typeexpr1
                {typeexpr.typeindex:=
                enter_type(pointer, -, -, -,
                  typeexpr1.typeindex, -);
                typeexpr.size := 4;
                typeexpr.alignment := 4;}
```

Wie in Kapitel 4 indizieren wir in einer Produktion mehrfach vorkommende Symbole, um sie zu unterscheiden. Beachten Sie, wie die Struktur in der Typtabelle aufgebaut wird.

```
        |       typeident       {typeexpr.* := typeident.*}
```

Das Übersetzungsschema zu *typeident*, auf dessen Angabe wir verzichten, hat die Attribute mit Hilfe der Typtabelle ermittelt.

```
        |       array [ intconst] of typeexpr1
                {index := enter_type(array, -,
                  intconst.value, typeexpr1.size,
                  typeexpr1.typeindex, -);
                typeexpr.size := intconst.value *
                  typeexpr1.size;
                align(typeexpr.size, 8);
                typeexpr.alignment := 8;
                typeexpr.typeindex := index}
```

Man sieht auch hier, wie die geschachtelte Struktur in der Typtabelle nachgebildet wird. Die Größen von Arrays und Records erhöhen wir grundsätzlich auf durch 8 teilbare Werte.

Bevor wir das Übersetzungsschema für Record-Typen betrachten, sehen wir uns die Übersetzung von Variablendeklarationen an, da diese im Record-Schema verwendet werden. Deklarationen legen Speicherplatz für Variablen an. Dabei ist das Offset in der aktuellen Umgebung (z. B. Prozedurrahmen) zu verwalten. Wir benutzen ein Attribut *offset* für jedes der Symbole *decls* und *decl*. *Decls* vererbt den Wert an *decl*. *Decl* ändert das Offset gemäß dem belegten Speicherplatz und reicht den geänderten Wert wieder hinauf an *decls*. Der Datenfluss im Ableitungsbaum ist in Abb. 6.19 dargestellt.

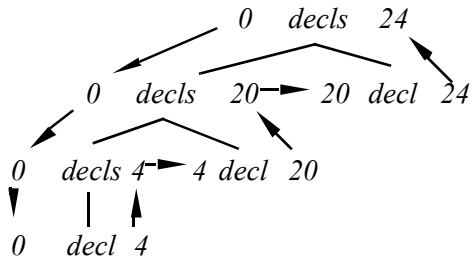


Abb. 6.19. Datenfluss für das Übersetzungsschema für Deklarationen

Dieser Datenfluss wird durch das folgende Übersetzungsschema realisiert:

```
decls   ->                      {decl.offset := decls.offset}
  offset        decl            {decls.offset := decl.offset}
        |                       {decls1.offset :=
                                  decls.offset}
                decls1 ;        {decl.offset := decls1.offset}
                decl            {decls.offset := decl.offset}
```

Tatsächlich ist das Übersetzungsschema für *decls* noch ein kleines bisschen komplizierter, da wir es auch für die Übersetzung von Felddeklarationen in Records mitbenutzen wollen. Wir benötigen dazu noch zwei zusätzliche Attribute: ein Attribut *type* mit möglichen Werten *var* oder *recordfield*, das festhält, ob diese Deklarationen Variablen oder Record-Felder beschreiben; weiterhin ein Attribut *recordindex*, das im zweiten Fall den Index des Recordtyps in der Typtabelle enthält. Dieses Attribut dient dazu, die Feldnamen in die richtige *fieldtable* eintragen zu können. Die beiden Attribute *type* und *recordindex* werden einfach nur von oben nach unten vererbt. Damit erhalten wir folgendes Schema für *decls*:

```
decls   ->                      {decl.* := decls.*}
  offset        decl            {decls.offset := decl.offset}
  type
  recordindex
        |                       {decls1.* := decls.*}
                decls1 ;        {decl.* := decls1.*}
                decl            {decls.offset := decl.offset}
```

Nun übersetzen wir die Deklarationen selbst:

```
decl    ->      id : typeexpr
  offset                {align(decl.offset,
                          typeexpr.alignment);
  type                  index := enter_var(decl.type, id.name,
  recordindex             depth, decl.offset, typeexpr.size,
                          -, typeexpr.alignment,
                          typeexpr.typeindex);
                        decl.offset := decl.offset +
                          typeexpr.size;
```

```
                                if decl.type = var then
                                  enter_name(depth, id.name, index)
                                else enter_fieldname(decl.recordindex,
                                  id.name, index)
                                end     }
```

Zunächst wird das aktuelle Offset anhand des Alignment-Wertes des Typausdrucks ggf. korrigiert. Dann wird die Variable in die Variablentabelle eingetragen. Alle dort benötigten Informationen stehen aus Attributen bzw. der globalen Variablen *depth* zur Verfügung. Das Offset wird erhöht. Schließlich wird der Name der Variablen entweder in der obersten Tabelle im Stack der Namenstabellen oder in der Tabelle des Record-Typs eingetragen.

Die drei Attribute erbt *decls* aus der Umgebung, nämlich entweder aus Variablendeklarationen oder aus einer Record-Deklaration:

```
vars    ->      var     {decls.offset := vars.offset;
  offset                  decls.type := var;
                          decls.recordindex := 0}
                decls ; {vars.offset := decls.offset;
                          align(vars.offset, 8)}
        |
```

Das Symbol *vars* erbt selbst das Attribut *offset* aus seiner Umgebung (Programm oder Prozedur). Auch für eine Gruppe von Variablen insgesamt sorgen wir abschließend für ein Alignment von 8. – Der noch ausstehende Fall der Record-Deklaration innerhalb von *typeexpr* sieht ähnlich aus wie das Schema für *vars*:

```
typeexpr ->…
  size  |       record  {decls.type := recordfield;
  alignment             decls.offset := 0;
  typeindex             index := enter_type(record,-,-,-,-,-);
                        decls.recordindex := index}
                decls
                end     {typeexpr.size := decls.offset;
                        align(typeexpr.size, 8);
                        typeexpr.alignment := 8;
                        typeexpr.typeindex :=
                          decls.recordindex}
```

In jeder Record-Deklaration wird das Offset zu 0 initialisiert, da nur relative Offsets zum Feldanfang von Interesse sind. Der Index des Record-Typs wird an die *decls* hinabgereicht, damit Feldnamen in die dazugehörige Namenstabelle eingetragen werden können. Der Gesamtplatzbedarf für den Record wird auf ein Vielfaches von 8 erhöht.

Selbsttestaufgabe 6.1: Geben Sie das Übersetzungsschema für explizite Typdeklarationen an. Einen Namen *alpha* kann man unter einem Index *v* der Typtabelle eintragen mit *name*(*v*) := *alpha*. Beachten Sie, dass der Typausdruck ein atomarer Typ sein kann!

```
types   ->      type id = typeexpr ; types
```

□

In den Übersetzungen für *program* und für Prozedur- bzw. Funktionsdeklarationen muss das richtige Offset für Variablendeklarationen berechnet werden. In Prozeduren muss außerdem der Stack der Namenstabellen und die globale Variable *depth* verwaltet werden.

```
program ->              {depth := 0; vars.offset := 0;
                        start := newlabel();
                        putcode('goto' start)}
                types vars procs
                        {putcode(start ':' noop);
                        temp_offset := vars.offset}
                stmt .  {mainsize := newtemp(integer);
                        align(temp_offset, 8);
                        putcode(mainsize ':='
                          const(temp_offset));
                        putcode('init_stack' mainsize)}
```

Da die 3-Adress-Maschine die Auswertung mit dem ersten Befehl im Befehlsspeicher beginnt, erzeugen wir als ersten Befehl (vor der Übersetzung von Prozeduren) einen Sprung zum Beginn des Hauptprogramms. Danach wird der Code für die am tiefsten geschachtelten Prozeduren zuerst plaziert werden, der für das Hauptprogramm zuletzt. Im Hauptprogramm wird weiterhin Speicherplatz für globale Variablen vergeben und anschließend für temporäre Variablen. Dazu wird nach der Übersetzung der Variablendeklarationen das erreichte Offset an die globale Variable *temp_offset* übergeben, die von da an die Vergabe von Platz für temporäre Variablen kontrolliert. Schließlich wird nach der Übersetzung der Anweisung(sfolge) die erreichte Gesamtgröße des Hauptprogramms zur Initialisierung des Stacks benutzt. Die Funktion *const* innerhalb des *putcode*-Befehls sorgt dafür, dass der Wert von *temp_offset* als Konstante in den Befehl eingetragen wird; andernfalls würde diese Zahl als Index in die Variablentabelle interpretiert.

```
procs   ->      proc procs
        |       function procs
        |

proc    ->      procedure
  start         id      {depth := depth + 1; pushnametable;
                        parameters.procname := id.name}
                parameters
                        {vars.offset := parameters.offset}
                vars
                procs   {temp_offset := vars.offset;
                        proc.start := prog_counter}
                stmt ;  {depth := depth -1; popnametable;
                        static_size := temp_offset;
                        align(static_size, 8);
                        index := enter_proc(id.name, depth,
                          static_size, proc.start, -);
                        enter_name(depth, id.name, index)}
```

Interessant ist hier die Vergabe von Speicherplatz innerhalb des Prozedurrahmens. Die Übersetzung der Parameter wird Platz ab Position 16 belegen und das am Ende

erreichte Offset an das Symbol *vars* übergeben. *Vars* belegt seinerseits Speicherplatz und übergibt das erreichte Offset vor der Übersetzung der Anweisung(sfolge) an *temp_offset*. Am Ende steht in *temp_offset* die erreichte Gesamtgröße des Prozedurrahmens, die nach Alignment in der Prozedurtabelle vermerkt wird.

Bei der Übersetzung von Parameterlisten von Prozeduren bzw. Funktionen ist einerseits Speicherplatz zu belegen analog zur Beschreibung von *refparam* und *valparam*-Befehlen der 3-Adress-Maschine. Darüber hinaus muss man sich für jeden Parameter merken, ob es ein Referenz- oder ein Wertparameter ist, damit beim Aufruf jeweils *refparam* oder *valparam*-Befehle erzeugt werden können. Und zwar muss man diese Information dort nachschlagen können, wo die Prozedur aufgerufen wird, also in demselben Scope, in dem auch der Prozedurname vermerkt wird. Beim Aufruf ist der jeweilige Name des formalen Parameters aber nicht bekannt. Man muss die Entscheidung zwischen *refparam* und *valparam* also anhand der Position in der Argumentliste treffen.

Wir denken uns dazu folgende Strategie aus: Ein Parameter wird zunächst in die Variablentabelle eingetragen, wobei als *type* entweder *refparam* oder *valparam* vermerkt wird. Anschließend wird ein Verweis auf diesen Eintrag in *zwei* Namenstabellen erzeugt. Der erste Eintrag erfolgt mit dem Namen des formalen Parameters, der in der Deklaration steht, in der Namenstabelle oben auf dem Stack. Dieser Eintrag wird innerhalb der Übersetzung der Prozedur benutzt (ebenso wie lokale Variablen der Prozedur). Der zweite Name wird vom Übersetzer erzeugt als Konkatenation des Prozedurnamens und der Parameternummer. Der zweite Parameter einer Prozedur *mergesort* bekäme damit den Namen *mergesort#2*. Dieser Name wird in die gleiche Namenstabelle eingetragen wie der Prozedurname selbst. Zum Zeitpunkt der Übersetzung der Parameter ist das die Namenstabelle der Tiefe *depth* – 1.

Die Ausführung dieser Idee ist etwas mühsam und technisch, und wir lassen sie hier weg. Bei Interesse können Sie sich die Übersetzung im Anhang A ansehen.

Die Übersetzung von Funktionsdeklarationen verläuft analog zu der von Prozeduren; lediglich der Rückgabeparameter muss zusätzlich beachtet werden. Wir lassen auch sie weg. Damit ist die Übersetzung von Deklarationen abgeschlossen.

6.3.2 Zuweisungen und Ausdrücke

Die Zuweisung ist die erste Art von Anweisung, die wir betrachten, und auch die zentrale. Alle anderen Arten von Anweisungen steuern ja nur den Kontrollfluss. Bei der Übersetzung von Zuweisungen gibt es folgende interessante Aspekte:

- Erzeugen temporärer Variablen zur Auswertung von Ausdrücken.
- Typüberprüfung (type checking) bei der Anwendung von Operatoren und bei der Zuweisung selbst – die Typen der Variablen und des Ausdrucks müssen kompatibel sein.
- Zugriff auf strukturierte Variablen, also Komponenten von Arrays und Records, und Verfolgen von Zeigern.

Wir erweitern zunächst die Grammatik:

```
stmt     ->      assignment
         |       cond
         |       loop
         |       proccall
         |       return
         |       block
```

Die anderen Arten von Anweisungen verfolgen wir weiter in Abschnitt 6.3.3.

```
assignment ->    var := expr

var      ->      id
         |       var [ intexpr ]
         |       var . id
         |       var ->

intexpr ->       expr
```

Diese Regeln beschreiben den Zugriff auf strukturierte Variablen. Die zweite Zeile definiert den Array-Zugriff, z. B. *x*[*i*]. Die dritte Zeile beschreibt Feldselektion in Records, z. B. *x.k*. Die vierte Zeile zeigt die Dereferenzierung von Zeigervariablen mit der Notation *x*->. Beachten Sie, dass man durch Aneinanderreihung Zugriffe beliebig tief schachteln kann, z. B.

```
x[i][j+1].tablepointer->[k].index
```

Beim Entwurf der Grammatik für Ausdrücke wollen wir boolesche von anderen Ausdrücken unterscheiden. Boolesche Ausdrücke werden meist zur Steuerung des Kontrollflusses benutzt und deshalb anders übersetzt. Das betrachten wir in Abschnitt 6.3.3.

```
expr     ->      simple
         |       boolexpr

simple   ->      term
         |       simple addop term

term     ->      factor
         |       term mulop factor

factor   ->      var
         |       ( expr )
         |       ( - factor )
         |       functioncall

boolexpr ->      boolterm
         |       boolexpr or boolterm

boolterm ->      boolfactor
         |       boolterm and boolfactor
```

```
boolfactor ->   not boolfactor
        |       simple
        |       simple cop simple
```

Diese Regeln erzeugen im Wesentlichen verschiedene Ebenen von Präzedenz bei der Auswertung arithmetischer Operatoren und sorgen dafür, dass Operationen gleicher Präzedenz linksassoziativ ausgewertet werden. Analog wird für boolesche Ausdrücke Vorrang zwischen *and*, *or* und *not* festgelegt. Zu den Terminalsymbolen gehören folgende Lexeme, die man jeweils durch ein Attribut *op* erhalten kann:

```
addop           +   -
mulop           *   /   div mod
cop             <   <=   =   #  >=   >
```

Dabei steht das Symbol # für „ungleich“.

Wir beginnen mit der Übersetzung der Zuweisung selbst, also der Regel

```
assignment ->var := expr
```

Auf der linken Seite kann ein beliebig tief geschachtelter Variablenzugriff stehen, auf der rechten Seite ebenso (*expr* leitet nach *var* ab) oder ein komplexerer Ausdruck. Die Frage ist, welche Informationen in Attributen von *var* und *expr* gesammelt sein müssen, damit die Zuweisung übersetzt werden kann.

Wir müssen zunächst einmal eine Entscheidung treffen, welche „Breite“ die Zuweisung haben darf. Damit ist Folgendes gemeint. Ein Befehl der 3-Adress-Maschine kopiert entweder 4 Bytes oder 1 Byte. Die Zuweisung in der Programmiersprache könnte hingegen einen ganzen Record oder Array kopieren. In diesem Fall müsste die Übersetzung eine Schleife im 3-Adress-Code generieren. Wir entscheiden uns hier zunächst für die etwas einfachere Variante und erlauben in der Programmiersprache nur Zuweisungen von atomaren Datentypen oder Zeigern. Damit ist ein Programmierer gezwungen, zum Kopieren eines Arrays eine Schleife zu schreiben bzw. einen Record feldweise zu kopieren. Die für den Programmierer komfortablere Variante (Kopieren größerer Breite als 4 Bytes) betrachten wir ggf. in den Übungen.

Die Idee zur Realisierung des Zugriffs auf strukturierte Variablen besteht nun darin, dem Symbol *var* auf der linken Seite *zwei* Variablen zuzuordnen, die in Attributen *var.var* und *var.offset* verwaltet werden. Die erste Variable *var.var* ist normalerweise die im Programm deklarierte; sie beschreibt den *Anfang* des Speicherbereichs anhand der Einträge in der Variablentabelle. Die zweite Variable *var.offset* wird vom Übersetzer erzeugt, und ihr Wert wird entsprechend den übersetzten Zugriffen zur Laufzeit erhöht. Wenn also auf der linken Seite z. B. $x[3*i]$ steht, so wird $var.var = x$ sein, und der Übersetzer wird eine Variable *var.offset* sowie Code erzeugt haben, so dass *offset* den richtigen Wert hat, um auf die Komponente $3*i$ zuzugreifen. Diese Situation ist in Abb. 6.20 gezeigt. Hier sind die Speicherbereiche der Variablen *var.var* und *var.offset* grau dargestellt; die gerade durch diese beiden Variablen adressierte Komponente ist schwarz gezeichnet.

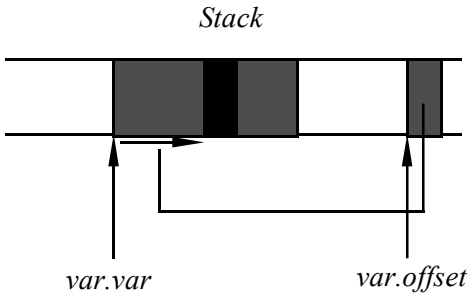


Abb. 6.20. Adressierung über die Variablen *var.var* und *var.offset*

Es gibt noch eine Variante dieser Situation, die dadurch entsteht, dass Speicherplatz auf dem Heap allokiert worden ist (Abb. 6.21).

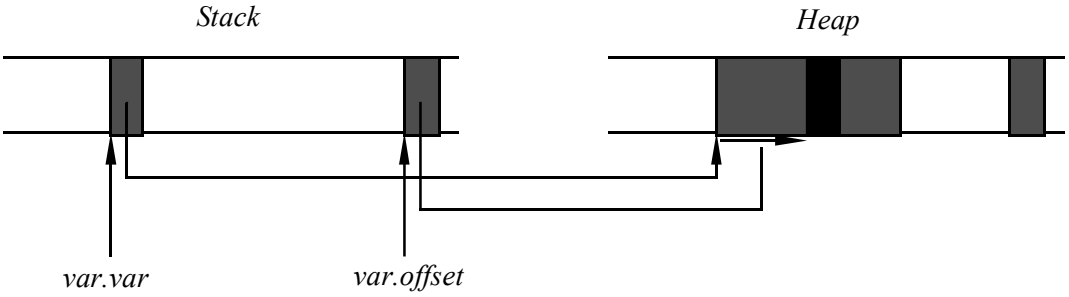


Abb. 6.21. Adressierung von Heap-Speicher mittels *var.var* und *var.offset*

Das heißt, *var.var* ist selbst eine Zeigervariable, die auf den Anfang eines Speicherbereichs im Heap zeigt, und wir haben die gerade adressierte Komponente (schwarz gezeichnet) durch bereits mindestens einmaliges Dereferenzieren erreicht. Die Strategie, durch Erhöhen des Wertes von *var.offset* auf Komponenten strukturierter Variablen zuzugreifen, gilt ebenfalls für Heap-Speicherbereiche. Wir merken uns in einem dritten Attribut *var.indirect*, ob *var.var* eine Zeigervariable ist (entsprechend Abb. 6.21).

Der Ausdruck auf der rechten Seite besitzt als Attribut *expr.var* eine temporäre Variable, in der der Wert des Ausdrucks berechnet worden ist. Die Symbole auf beiden Seiten der Zuweisung besitzen weiterhin ein Attribut *type*, das den jeweiligen Typ als Index in die Typtabelle enthält. Variablen sind ebenfalls als Indizes in die Variablentabelle dargestellt. Insgesamt gibt es also Attribute *var.var*, *var.offset*, *var.indirect* und *var.type* sowie *expr.var* und *expr.type*.

```
assignment ->   var := expr
                      {if compatible(var.type, expr.type) then
                          if width(var.type) = 1 then
                            ass := ':-'
                          else ass := ':=' end;
                          if not var.indirect then
                            if var.offset = 0 (* no variable
                              "offset" exists *) then
                              putcode(var.var ass expr.var)
                            else putcode(var.var '['
                              var.offset ']' ass expr.var)
                            end
                          else
                            if var.offset = 0 then
                              putcode('*' var.var ass
                                expr.var)
                            else putcode('*' var.var '['
                              var.offset ']' ass expr.var)
                            end
                          end
                        else error end}
```

Die Funktion *compatible* ist für die Typüberprüfung zuständig. Funktion *width* ermittelt aus dem Typindex die Größe 1 oder 4 der Darstellung. Bei der Übersetzung der Zuweisung wird diese Größe beachtet und der entsprechende 3-Adress-Befehl erzeugt. Das Attribut *var.indirect* steuert, ob ggf. indirekte oder indirekt indizierte Zugriffe benutzt werden.

```
expr    ->      simple  {expr.* := simple.*}
  var   |       ...
  type
```

Die Alternative *boolexpr* betrachten wir erst in Abschnitt 6.3.3.

```
simple  ->      term    {simple.* := term.*}
  var   |       simple1 addop term
  type                  {simple.type := typemap(simple1.type,
                          addop.op, term.type);
                        if simple.type ¦ 0 then
                          simple.var := newtemp(simple.type);
                          putcode(simple.var ':='
                            simple1.var addop.op term.var)
                        else error
                        end     }
```

Die Funktion *typemap* berechnet aus den Typen der Argumente und dem Operator den Ergebnistyp (ebenfalls als Index in die Typtabelle) und gibt im Fehlerfall den Wert 0 zurück.

```
term    ->      factor  {term.* := factor.*}
  var   |       term1 mulop factor
  type                  {...}
```

Diese Übersetzung ist analog zu der von *simple*.

```
factor  ->      var     {if width(var.type) = 1 then
                           ass := ':-'
  var                    else ass := ':=' end;
  type                   if not var.indirect then
                           if var.offset = 0
                             (* no "offset" var. *)
                           then factor.var := var.var
                           else factor.var :=
                             newtemp(var.type);
                             putcode(factor.var ass var.var
                               '[' var.offset ']')
                           end
                         else factor.var := newtemp(var.type);
                           if var.offset = 0 then
                             putcode(factor.var ass '*'
                               var.var)
                           else putcode(factor.var ass
                             '*' var.var '[' var.offset ']')
                           end
                         end;
                         factor.type := var.type}
```

Falls auf der rechten Seite eine einfache Variable steht, wird sie für *factor* übernommen. Falls dort Zugriff auf eine strukturierte Variable erfolgt, kopieren wir an dieser Stelle in eine einfache Variable (unter Beachtung der Breite).

```
        |       ( expr )         {factor.* := expr.*}
        |       ( - factor1 )   {if type(factor1.type) =
                                    integer or
                                    type(factor1.type) = real
                                  then factor.* := factor1.*;
                                    putcode(factor.var ':=' '-'
                                      factor1.var)
                                  else error
                                  end     }
        |       functioncall    {factor.* := functioncall.*}
```

Das Symbol *functioncall* hat ebenfalls Attribute *var* und *type* (vgl. Abschnitt 6.3.4).

Es fehlt noch die Übersetzung von Zugriffen auf strukturierte Variablen:

```
var     ->      id      {v := lookup(id.name); var.var := v;
  var                   var.offset := 0; var.indirect := false;
  offset                var.type := typeindex(v)}
  indirect
  type
```

Hierbei ist der Wert 0 für das Attribut *offset* so zu interpretieren, dass (noch) keine *offset*-Variable existiert. – Spannend wird nun die Übersetzung von Array-Zugriffen. Einerseits muss eine Variable *offset* den richtigen Wert bekommen. Darüber hinaus muss Code erzeugt werden, der *zur Laufzeit* überprüft, ob der berechnete Index innerhalb der Arraygrenzen liegt. Wenn das nicht der Fall ist, muss dann zu einer Adresse *RangeErr* gesprungen werden, die eine Fehlermeldung „range error“ ausgibt.

```
|       var1 [ intexpr ]
              {if type(var1.type) = array then
                 (* range checking *)
                 y := intexpr.var;
                 x := newtemp(integer);
                 putcode(x ':='
                   const(nocomps(var1.type)));
                 putcode('if' y '<' '0' 'goto'
                   RangeErr);
                 putcode('if' y '>=' x 'goto'
                   RangeErr);

                 if var1.offset = 0 then
                   var.offset := newtemp(integer);
                   putcode(var.offset ':=' '0')
                 else var.offset := var1.offset
                 end;
                 z := newtemp(integer);
                 putcode(z ':='
                   const(compsize(var1.type)) '*' y);
                 putcode
                   (var.offset ':=' var.offset '+' z);
                 var.var := var1.var;
                 var.indirect := var1.indirect;
                 var.type := compindex(var1.type)
               else error
               end     }
```

Dabei sind *type*(*var*$_1$.*type*), *nocomps*(…) usw. Zugriffe auf die Typtabelle. Beachten Sie, dass bei einem Array mit *n* Komponenten diese mit den Indizes 0, …, *n*–1 angesprochen werden.

Selbsttestaufgabe 6.2: Übersetzen Sie den Zugriff auf Felder von Record-Variablen, also die rechte Seite

```
|       var1 . id
```

Benutzen Sie die Funktion *lookup_field*, um den Eintrag für den Feldselektor in der Variablentabelle zu finden. □

Es folgt die Dereferenzierung. Das bisher mit *var*$_1$.*var* und *var*$_1$.*offset* erreichte Feld (auf dem Stack oder auf dem Heap) sollte einen Zeiger auf den Heap-Bereich enthalten, der nun dereferenziert wird. Die Idee ist, eine neue temporäre Variable für *var*.*var* zu erzeugen und ihr diesen Zeiger auf den Heap-Bereich zuzuweisen (nun wird also *var*.*indirect* = *true* sein). Anschließend können wir mit einer neuen Variablen *var*.*offset* bei Bedarf weiter in den Heap-Bereich hineingreifen.

```
        |       var1 ->
                      {if type(var1.type) = pointer then
                         z := newtemp(pointer);
                         if not var1.indirect then
                           if var1.offset = 0
                             (* no offset var.*)
                           then putcode(z := var1.var)
                           else putcode(z :=
                             var1.var '[' var1.offset ']')
                           end
                         else
                           if var1.offset = 0
                             (* no offset var.*)
                           then putcode(z := '*' var1.var)
                           else putcode(z :=
                             '*' var1.var '[' var1.offset ']')
                           end;
                         var.var := z; var.offset := 0;
                         var.indirect := true;
                         var.type := compindex(var1.type)
                       else error
                       end     }
```

Das Symbol *intexpr* in der Regel für den Arrayzugriff dient nur der Typprüfung:

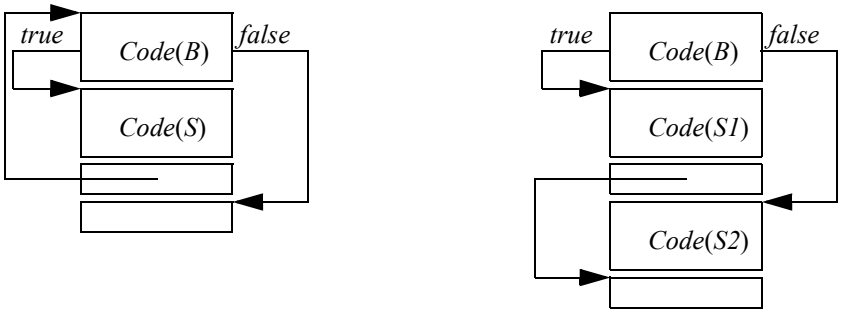
```
intexpr ->      expr    {if type(expr.type) = integer then
                           intexpr.* := expr.*
                           else error end}
```

6.3.3 Kontrollstrukturen und boolesche Ausdrücke

Wir vervollständigen zunächst die Grammatik, wobei die Produktionen für *stmt* ja schon in Abschnitt 6.3.2 eingeführt wurden:

```
stmt    ->      assignment
        |       cond
        |       loop
        |       proccall
        |       return
        |       block

block   ->      begin stmts end

stmts   ->      stmt
        |       stmts ; stmt

cond    ->      if boolexpr then stmt else stmt end
        |       if boolexpr then stmt end

loop    ->      while boolexpr do stmt end
```

Prozeduraufrufe und *return*-Anweisungen sind Thema des Abschnitts 6.3.4. Natürlich gibt es weitere interessante Kontrollstrukturen wie *for*-Schleifen, *case*-Anweisungen usw., auf deren Realisierung wir hier verzichten. Auch Sprünge lassen wir weg. Zur Übersetzung von Kontrollstrukturen sind grundsätzlich nur einige Sprunganweisungen zu generieren. Für *while*-Schleifen und bedingte Anweisungen ist die Struktur des zu erzeugenden Codes in Abb. 6.22 gezeigt.



while *B* **do** *S* **if** *B* **then** *S1* **else** *S2*

Abb. 6.22. Codeanordnung für Kontrollstrukturen

In dieser Darstellung zeigt jeweils das letzte, leere Rechteck die Folgeanweisung. Streng genommen ist der Sprung aus *Code*(*B*) bei Auswertung zu *true* nicht nötig. Es ist aber für die Übersetzung ganz praktisch, ihn vorzusehen. In der anschließenden Optimierungsphase können solche überflüssigen Sprünge sicherlich entfernt werden.

Die Auswertung boolescher Ausdrücke hat im Kontext von Kontrollstrukturen also das Ziel, einen von zwei Sprüngen auszuführen. Dabei ist es wichtig, zu beobachten, dass boolesche Ausdrücke oft nicht vollständig ausgewertet werden müssen, da das Ergebnis nach partieller Auswertung bereits feststeht. Zum Beispiel kann die Auswertung des Ausdrucks

```
A and (B or C or (x < 5))
```

sofort abgebrochen werden, wenn *A* den Wert *false* hat.

Die folgende Technik zur Übersetzung boolescher Ausdrücke, auch *Kurzschließen boolescher Ausdrücke* genannt, nutzt diese Beobachtungen aus. Die Idee ist, an das Symbol *boolexpr* in zwei Attributen *true* und *false* die Sprungmarken zu vererben, an die bei Auswertung zu *true* bzw. *false* gesprungen werden soll. Die Übersetzungsschemata für *and*, *or* und *not* vererben diese oder neu generierte Sprungmarken weiter an die jeweiligen Teilausdrücke. Abb. 6.23 illustriert die Übersetzungsstrategie.

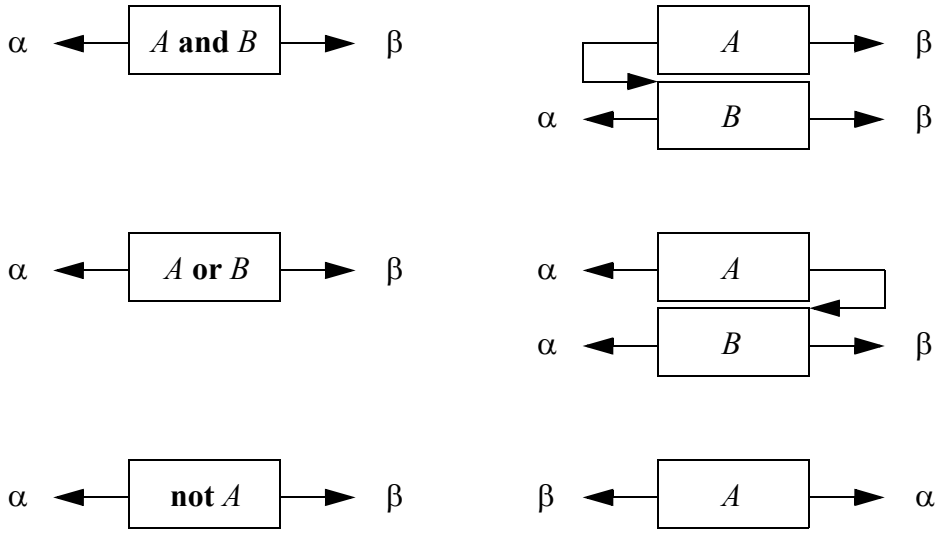


Abb. 6.23. Übersetzung boolescher Ausdrücke mit Vererbung von Sprungzielen

Hier steht jeweils ein Kästchen für den Code eines Teilausdrucks, α und β sind die bei Auswertung zu *true* bzw. zu *false* anzuspringenden Marken. Für jedes Kästchen zeigt der linke Pfeil, wohin bei Auswertung zu *true*, der rechte, wohin bei Auswertung zu *false* gesprungen wird. Damit ergeben sich folgende Übersetzungsschemata für boolesche Ausdrücke:

```
boolexpr ->              {boolterm.* := boolexpr.*}
  true           boolterm
  false
        |                {boolexpr1.true := boolexpr.true;
                         boolexpr1.false := newlabel()}
                 boolexpr1
                         {putcode(boolexpr1.false ':' noop);
                         boolterm.* := boolexpr.*}
                 or boolterm

boolterm ->      boolfactor
  true   |       boolterm1 and boolfactor
  false
```

Dieses Übersetzungsschema geht völlig analog zu dem für *boolexpr*.

```
boolfactor ->            {boolfactor1.true := boolfactor.false;
  true                   boolfactor1.false := boolfactor.true}
  false          not boolfactor1
        |        simple  {if type(simple.type) = boolean then
                            putcode('if' simple.var '=' '0'
                            'goto' boolfactor.false);
                            putcode('goto' boolfactor.true)
                          else error end}
```

```
        |       simple1 cop simple2
                        {if typemap(simple1.type, cop.op,
                           simple2.type) = boolean
                         then putcode('if' simple1.var cop.op
                           simple2.var 'goto'
                           boolfactor.true);
                           putcode('goto' boolfactor.false)
                         else error
                         end    }
```

Die Übersetzung von Kontrollstrukturen muss die Attribute *true* und *false* geeignet initialisieren. Das Übersetzungsschema für die *while*-Schleife ist dann (vgl. Abb. 6.22):

```
loop    ->      while   {start := newlabel();
                        boolexpr.true := newlabel();
                        boolexpr.false := newlabel();
                        putcode(start ':' noop);}
                boolexpr
                do      {putcode(boolexpr.true ':' noop)}
                stmt
                end     {putcode('goto' start);
                        putcode(boolexpr.false ':' noop)}
```

Selbsttestaufgabe 6.3: Geben Sie das Übersetzungsschema an für eine der Alternativen von

```
cond    ->      if boolexpr then stmt else stmt end
        |       if boolexpr then stmt end
```

□

Schließlich müssen wir die Übersetzung boolescher Ausdrücke noch einbauen in die Auswertung beliebiger Ausdrücke. Man kann ja auch den Wert eines booleschen Ausdrucks einer booleschen Variablen zuweisen.

```
expr    ->      simple  {expr.* := simple.*}
        |               {boolexpr.true := newlabel();
                        boolexpr.false := newlabel()}
                boolexpr
                        {expr.var := newtemp(boolean);
                        expr.type := boolean;
                        after := newlabel();
                        putcode(boolexpr.true ':'
                          expr.var ':-' '1');
                        putcode('goto' after);
                        putcode(boolexpr.false ':'
                          expr.var :- '0');
                        putcode(after ':' noop)}
```

Boolesche Werte werden in einem Byte dargestellt, *true* durch 1, *false* durch 0.

6.3.4 Prozedur- und Funktionsaufrufe

Die Grammatik dazu ist recht einfach:

```
functioncall -> id ( args )

args     ->     arg argrest

argrest  ->     , arg argrest
         |

arg      ->     expr

proccall ->     id ( args )

return   ->     return
         |      return expr
```

Zur Beschreibung von Argumentlisten haben wir diesmal rechtsrekursive Regeln gewählt (im Gegensatz z. B. zur Beschreibung von Parameterlisten oben). Der Grund dafür wird sofort klar werden.

Prozedur- und Funktionsaufrufe sehen völlig gleich aus; sie lassen sich daran unterscheiden, ob sie im Kontext von Anweisungen oder von Ausdrücken auftreten. Auch die Übersetzung ist weitgehend die gleiche. Wir betrachten im Folgenden nur die Übersetzung von Funktionsaufrufen.

Die Übersetzung von $p(e_1, \ldots, e_n)$ muss für die Auswertung jedes Argumentausdrucks Code generieren und danach jeweils einen *refparam*- oder *valparam*-Befehl erzeugen. Dann ist ein *call p*-Befehl zu erzeugen und schließlich (nur für Funktionen) ein *getresult*-Befehl. Dabei sind zwei Probleme zu lösen:

Erstens muss für jeden Argumentausdruck nachgesehen werden, ob ein *refparam*- oder ein *valparam*-Befehl zu erzeugen ist. Dazu haben wir bei der Übersetzung der Deklaration die Einträge *p*#1, *p*#2 usw. in die Namenstabelle vorgenommen. Technisch bedeutet das, dass an jedes *arg*-Symbol der Name der Prozedur sowie die Position in der Argumentliste übergeben werden muss. Wir haben die rechtsrekursive Form der Regeln gewählt, da damit die Nummerierung sehr leicht durch Vererbung erreicht werden kann (Abb. 6.24). Die aktuelle Position muss nur beim Übergang zum rechten Sohn um 1 erhöht werden.

Das zweite Problem besteht darin, dass in den Argumentausdrücken wiederum Funktionsaufrufe enthalten sein könnten, die ihrerseits *param*-Befehle generieren würden. Wenn wir die Argumentausdrücke sequentiell übersetzen, würden aber dann *param*-Befehle für *p* mit denen für solche Funktionen durcheinandergeraten. Zum Beispiel würde für einen Aufruf $p(a, f(x))$ eine Folge

```
valparam a
valparam x
call f
getresult y
call p
```

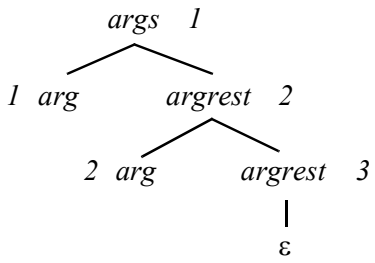


Abb. 6.24. Nummerierung von Funktionsargumenten mit Vererbung mittels rechtsrekursiver Regeln

erzeugt werden. Dabei würde *a* in den Prozedurrahmen für *f* kopiert werden, da *f* zuerst aufgerufen wird.

Um dies zu vermeiden, müssen wir dafür sorgen, dass die *param*-Befehle für *p* erst *nach* der Auswertung aller Argumentausdrücke von *p* erzeugt bzw. in den Befehlsspeicher geschrieben werden. Dazu benutzen wir einen *temporären Befehlsspeicher*, den wir als ein Attribut *commandlist* verwalten. Zum Bearbeiten einer solchen Liste von Befehlen gebe es folgende Operationen:

newcommandlist(). Liefert eine leere Liste.

appendcode(*commandlist*, *command*). Hängt einen 3-Adress-Befehl an die Liste an. Das Argument *command* wird genauso spezifiziert wie in *putcode*-Befehlen.

flush(*commandlist*). Schreibe alle in der *commandlist* gesammelten Befehle in den Befehlsspeicher und lösche die Liste.

Damit können wir die Übersetzungsschemata angeben. Argumente haben also drei Attribute, nämlich den Prozedurnamen, die Nummer des Arguments und eine Befehlsliste:

```
args    ->              {arg.* := args.*; argrest.* := args.*;
  procname              argrest.no := argrest.no + 1}
  no            arg argrest
  commandlist

argrest ->              {arg.* := argrest.*;
  procname              argrest1.* := argrest.*;
  no                    argrest1.no := argrest1.no + 1}
  commandlist   , arg argrest1
        |
```

```
arg     ->      expr    {v := lookup(arg.procname'#'arg.no);
  procname                if compatible(typeindex(v), expr.type)

  no                      then
  commandlist               if type(v) = refparam then
                              appendcode(arg.commandlist,
                                'refparam' expr.var)
                            else appendcode(arg.commandlist,
                              'valparam' expr.var)
                            end
                          else error
                          end     }

functioncall -> id      {args.procname := id.name;
  var                   args.no := 1;
  type                  args.commandlist := newcommandlist()}
                ( args )
                        {p := lookup(id.name);
                        functioncall.var :=
                          newtemp(typeindex(p));
                        functioncall.type := typeindex(p);
                        flush(args.commandlist);
                        putcode('call' p);
                        putcode('getresult' functioncall.var)}
```

Das Attribut *commandlist* enthält dabei jeweils einen Zeiger auf eine Befehlsliste, so dass die Übersetzungsschemata tatsächlich alle die gleiche Liste bearbeiten.

Das Übersetzungsschema für *proccall* geht analog, das für *return* kann man sich leicht selbst überlegen.

Damit haben wir die Übersetzung aller wesentlichen Konzepte imperativer Programmiersprachen präzise beschrieben.

6.4 Literaturhinweise

Gute Darstellungen der in diesem Kapitel behandelten Themen finden sich z. B. in den Büchern von Aho et al. (2006), Wilhelm und Maurer (2007) oder Alblas und Nymeyer (1996).

Speicherplatzverwaltung auf einem Stack wurde im Zusammenhang mit Programmiersprachen erfunden, die Rekursion erlauben, wie etwa Lisp oder Algol 60. Das Buch von Randell und Russell (1964) beschreibt bereits detailliert die Verwaltung von Prozedurrahmen auf einem Stack, einschließlich des Display-Mechanismus, im Kontext der Implementierung von Algol 60. Diese Techniken wurden für die Implementierung von Algol 68 angepasst (Hill 1976). Der Display-Mechanismus selbst stammt von Dijkstra (1960, 1963).

Die Idee der Verwendung von Zwischensprachen kam sehr früh auf, schon in den 50er Jahren. Man beobachtete damals, dass man für die Implementierung von m Programmiersprachen auf n verschiedenen Maschinen $m \cdot n$ Compiler benötigt, was sich bei Verwendung einer einheitlichen Zwischensprache auf $m+n$ reduzieren ließe. Daraus resultierte die Suche nach einer „universellen" Zwischensprache, der sogenannte UNCOL (*Universal Communication Oriented Language*), vorgeschlagen im Bericht eines Komitees (Strong et al. 1958). Ein konkreter Vorschlag für eine solche Sprache stammt von Steel (1961). Obwohl Zwischensprachen in vielen Übersetzern benutzt werden, hat sich die Idee einer einzigen universellen Sprache nicht durchgesetzt, wohl deshalb, weil doch immer wieder Anpassungen an spezielle Maschinenarchitekturen und Eigenschaften der Quellsprache nötig sind, wenn man nicht zuviel an Effizienz verlieren will.

Ein bekanntes Beispiel für die Verwendung einer stack-orientierten Zwischensprache ist die Übersetzung von PASCAL in sogenannten P-Code, der auf einer abstrakten Maschine, der P-Maschine, interpretiert wird (Nori et al. 1981, Pemberton und Daniels 1982). Eine portable C-Implementierung wird in (Johnson 1979) beschrieben. Portable Übersetzer, d. h. die Auswechselbarkeit von Frontend oder Backend, werden auch in (Johnson 1978) und (Tanenbaum et al. 1983) diskutiert.

Ein neueres bekanntes Beispiel für Übersetzung in eine Zwischensprache ist Java (Gosling, Joy und Steele 1996). Hier wird in Java-Bytecode übersetzt, der effizient über das Internet übertragen werden kann und am Ziel von der „Java Virtual Machine" (Lindholm und Yellin 1996) durch Interpretation ausgeführt werden kann. Anstelle der Interpretation ist auf dem Zielrechner auch eine „just-in-time"-Übersetzung möglich.

Anhang A

Übersetzung von Parameterlisten in Prozedurdeklarationen

Die Übersetzung von Parameterlisten in Prozedurdeklarationen hatten wir in Abschnitt 6.3.1 weggelassen. Hier sind die Übersetzungsschemata dazu.

```
parameters ->           {params.procname :=
  offset                   parameters.procname}
  procname      ( params )
                        {parameters.offset := params.offset;
                        align(parameters.offset, 8)}
        |               {parameters.offset := 16}

params  ->              {param.procname := params.procname;
  offset                param.offset := 16; param.no := 1}
  no            param
  procname              {params.offset := param.offset;
                        params.no := 1}
        |               {params1.procname := params.procname}
                params1 ;
                        {param.procname := params.procname;
                        param.offset := params1.offset;
                        param.no := params1.no + 1}
                param
                        {params.offset := param.offset;
                        params.no := param.no}
```

Hier wird das Attribut *procname* von oben nach unten vererbt und die Attribute *offset* und *no* von unten nach oben gereicht (ähnlich der Strategie für *decls* in Abbildung 6.19).

```
param   ->      var id : paramtype
  offset                {align(param.offset, 4);
  no                    index := enter_var(refparam, id.name,
  procname                depth, param.offset, 4, -, 4,
                          paramtype.typeindex);
                        enter_name(depth - 1,
                          param.procname'#'param.no, index);
                        enter_name(depth, id.name, index);
                        param.offset := param.offset + 4}
        |       id : paramtype
                        {align(param.offset,
                          paramtype.alignment);
```

```
                                    index := enter_var(valparam, id.name,
                                      depth, param.offset, paramtype.size,
                                      -, paramtype.alignment,
                                      paramtype.typeindex);
                                    enter_name(depth - 1,
                                      param.procname'#'param.no, index);
                                    enter_name(depth, id.name, index);
                                    param.offset := param.offset +
                                      paramtype.size}
```

So werden also *call-by-reference* und *call-by-value* übersetzt. Man beachte, wie gemäß der in Abschnitt 6.3.1 beschriebenen Strategie jeder Parameter unter zwei verschiedenen Namen in Namenstabellen der Tiefe *depth* und *depth* – 1 eingetragen wird.

```
paramtype ->    simpletype      {paramtype.* := simpletype.*}
  size
  alignment
  typeindex
        |       typeident        {paramtype.* := typeident.*}
```

Lösungen zu den Selbsttestaufgaben

Aufgabe 6.1

Beschreibt der Typausdruck einen atomaren Typ, so müssen wir einen neuen Typeintrag in die Typtabelle einfügen. In allen anderen Fällen wurde der Typeintrag bereits bei Abarbeitung der *typeexpr*-Produktion vorgenommen, und wir müssen hier nur noch den Typnamen nachtragen.

```
types   ->      type id = typeexpr
                        {if typeexpr.typeindex > 4 then
                           name(typeexpr.typeindex) :=
                             id.name;
                         else enter_type(
                           type(typeexpr.typeindex),
                           id.name,-,-,-,-);
                         end }
                ; types
        |
```

Aufgabe 6.2

```
var     ->      …
  var   |       var1 . id
  offset                {if type(var1.type) = record then
  indirect                 if var1.offset = 0 then
  type                       var.offset := newtemp(integer);
                             putcode(var.offset ':=' '0')
                           else var.offset := var1.offset
                           end;
                           z := lookup_field(var1.type,
                             id.name);
                           putcode(var.offset ':='
                             var.offset '+' const(offset(z)));
                           var.var := var1.var;
                           var.indirect := var1.indirect;
                           var.type := typeindex(z);
                         else error
                         end    }
```

Dabei ist *z* eine Variable vom Typ *integer*, die noch irgendwo außerhalb des Übersetzungsschemas deklariert werden muss.

Aufgabe 6.3

Wir geben das Übersetzungsschema für beide Alternativen an.

```
cond    ->      if      {boolexpr.true := newlabel();
                        boolexpr.false := newlabel();
                        continue := newlabel();}
                boolexpr
                then    {putcode(boolexpr.true ':' noop)}
                stmt    {putcode('goto' continue)}
                else    {putcode(boolexpr.false ':' noop)}
                stmt
                end     {putcode(continue ':' noop)}
        |       if      {boolexpr.true := newlabel();
                        boolexpr.false := newlabel();}
                boolexpr
                then    {putcode(boolexpr.true ':' noop)}
                stmt
                end     {putcode(boolexpr.false ':' noop)}
```

Literatur

Aho, A.V., Hopcroft, J.E. und Ullman, J.D. (1983). *Data Structures and Algorithms*. Addison-Wesley, Reading, MA.

Aho, A.V., Lam, M.S., Sethi, R. und Ullman, J.D. (2006). *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley, Reading, MA.

Alblas, H. und Nymeyer, A. (1996). *Practice and Principles of Compiler Building with C*. Prentice-Hall International, London, UK.

Dijkstra, E.W. (1960). Recursive Programming. *Numerische Mathematik 2*, 312-318.

Dijkstra, E.W. (1963). An Algol 60 Translator for the X1. *Annual Review in Automatic Programming 3*, Pergamon Press, New York, S. 329-345.

Gosling, J., Joy, B. und Steele, G. (1996). *The Java™ Language Specification*. Addison-Wesley, Reading, MA.

Hill, U. (1976). Special Run-Time Organization Techniques for ALGOL 68. In: Bauer, F.L. und Eickel, J. (Hrsg.), *Compiler Construction – An Advanced Course*. Lecture Notes in Computer Science 21, Springer-Verlag, Berlin.

Johnson, S.C. (1978). A Portable Compiler: Theory and Practice. 5th ACM Symposium on Principles of Programming Languages, S. 97-104.

Johnson, S.C. (1979). A Tour through the Portable C Compiler. AT & T Bell Laboratories, Murray Hill, N.J.

Lindholm, T. und Yellin, F. (1996). *The Java™ Virtual Machine Specification*. Addison-Wesley, Reading, MA.

Nori, K.V., Ammann, U., Jensen, K., Nägeli, H.H., und Jacobi, C. (1981). Pascal P Implementation Notes. In: Barron, D.W. (Hrsg.), *Pascal – The Language and its Implementation.* John Wiley & Sons, New York, S. 125-170.

Pemberton, S. und Daniels, M. (1982). *Pascal Implementation, The P4 Compiler*. Ellis Horwood Publ. Co.

Randell, B. und Russell, L.J. (1964). *Algol 60 Implementation*. Academic Press, New York.

Steel, T.B., Jr. (1961). A First Version of Uncol. Proc. Western Joint Computer Conference, S. 371-378.

Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O. und Steel, T. (1958). The Problem of Programming Communication with Changing Machines: A Proposed Solution. Report of the Share Ad-Hoc Committee on Universal Languages. *Communications of the ACM 1:8*, 12-18 (Part 1) und *1:9*, 9-15 (Part 2).

Tanenbaum, A.S., van Steveren, H., Keizer, E.G. und Stevenson, J.W. (1983). A Practical Tool Kit for Making Portable Compilers. *Communications of the ACM 26*, 654-660.

Wilhelm, R. und Maurer, D. (2007). *Übersetzerbau: Theorie, Konstruktion, Generierung*. 2. Aufl., Springer-Verlag, Berlin.

VI Literatur

Index