Inhalt

Lehrziele

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- die verschiedenen Optimierungsverfahren erklären und klassifizieren können,
- das Prinzip der Datenflussanalyse verstanden haben,
- erklären können, wie man mit den aus der Datenflussanalyse gewonnenen Informationen globale bzw. Schleifenoptimierungen durchführen kann,
- die vier verschiedenen Grundformen der Datenflussanalyse gegeneinander abgrenzen können und jeweils Beispiele für deren Einsatz nennen können,
- Datenflussgleichungen aufstellen und lösen können,
- imstande sein, Maschinencode für Basisblöcke mit Hilfe der beiden Algorithmen *codegen* und *getreg* zu erzeugen.

Kapitel 8

Codeerzeugung und Optimierung

Mit den Mitteln der Syntaxanalyse und der syntaxgesteuerten Übersetzung sind wir bereits in der Lage, Übersetzer zwischen verschiedenen Sprachen zu implementieren, und was die Übersetzung auf der quellsprachlichen Ebene anbelangt, so ist dies eigentlich auch schon alles, da man für die Zielsprache eine Implementierung (einen Interpreter oder seinerseits einen weiteren Übersetzer) bereits gegeben hat oder annimmt.

Bei der Übersetzung von Programmiersprachen sind wir bisher allerdings lediglich zu Zwischensprachen, wie dem 3-Adress-Code (s. Abschnitt 6.2) oder dem SECD-Code (s. Abschnitt 7.4.1), vorgestoßen. Derartige abstrakte Programmrepräsentationen waren bei den bisherigen Betrachtungen äußerst hilfreich: Zum einen dienten sie als vereinfachendes Modell (z. B. eines realen Prozessors) und erlaubten so der Übersetzung, zunächst einmal von vielen Details abzusehen. Dies erleichterte die Beschreibung der Übersetzung, machte sie überschaubar und hoffentlich weniger fehlerbehaftet. Zum anderen kann man von einer Zwischensprache in verschiedene Maschinensprachen abbilden und unterstützt damit die Portierung von Sprachen auf verschiedene Architekturen.

Um ein Programm nun aber auf einem realen Prozessor ablaufen zu lassen, benötigt man entweder einen Interpreter für die Zwischensprache oder aber einen weiteren Übersetzungsschritt in die Maschinensprache des Prozessors. Diese Art der Übersetzung nennt man *Codeerzeugung*. Es sollte zunächst einmal nicht sehr schwierig sein, „irgendein“ (korrektes) Maschinenprogramm zu erzeugen, da beispielsweise die Befehle des 3-Adress-Codes relativ einfach sind und durch den Befehlsvorrat eines realen Prozessors sicherlich abgedeckt sein werden. Das Ziel ist aber vielmehr, ein möglichst *effizientes Maschinenprogramm* zu erhalten, d. h., es sollte möglichst schnell sein und möglichst wenig Speicher verbrauchen. Dies macht die Aufgabe der Codeerzeugung schon wesentlich anspruchsvoller: Man muss nun geschickt die speziellen Eigenschaften des Zielprozessors ausnutzen, wie z. B. bestimmte Adressierungsarten oder schnelle Register.

Die Schritte der Übersetzung, die mit der Auswahl möglichst guten Codes betraut sind, fasst man allgemein unter dem Begriff *Optimierung* zusammen. Demnach besteht ein wesentlicher Teil der Codeerzeugung selbst aus der Optimierung. Neben dieser *maschinenabhängigen Optimierung* gibt es aber auch eine Reihe von Verfahren, die Verbesserungen bereits auf der Ebene der Zwischensprachen durchführen können. Diese bezeichnet man entsprechend als *maschinenunabhängige Optimie-*

rungen. Darüber hinaus existieren weitreichende Optimierungsmöglichkeiten für Quellsprachen (rein technisch arbeiten diese Verfahren natürlich auf internen Repräsentationen wie Syntaxbäumen). Derartige Quellcodetransformationen bezeichnet man oft auch als *algebraische Optimierung*.

Wir wollen noch kurz anmerken, dass der Begriff „Optimierung“ eigentlich ein wenig hochgegriffen ist, denn die noch zu beschreibenden Methoden vermögen in der Regel nicht, optimalen Code zu erzeugen, sondern können lediglich Code verbessern.

In Abschnitt 8.1 geben wir zunächst einen Überblick über die verschiedenen Möglichkeiten zur Optimierung. Eine für sehr viele Optimierungsverfahren benötigte Analysemethode, die Datenflussanalyse, werden wir in Abschnitt 8.2 etwas genauer betrachten. In Abschnitt 8.3 beschreiben wir dann ein Verfahren zur Codegenerierung.

8.1 Ein Überblick über Optimierungsverfahren

Es gibt mittlerweile eine erstaunlich große Vielfalt an Optimierungsmethoden. Da es aus Platzgründen nicht möglich ist, sämtliche Verfahren zu beschreiben, wollen wir hier eine Übersicht geben, die einen Eindruck von den Möglichkeiten der Optimierung vermittelt.

Das allgemeine Ziel der Optimierung ist zum einen die *Elimination unnötiger Berechnungen*. Dies beinhaltet sowohl Berechnungen von Werten, die überhaupt nicht gebraucht werden, als auch von Werten, die bereits aufgrund einer Berechnung an anderer Stelle verfügbar sind. Zum anderen versucht die Optimierung, kostspielige (d. h. zeit- oder platzaufwendige) Berechnungen durch äquivalente *kostengünstigere Operationen* zu ersetzen.

Optimierungsmethoden kann man nach verschiedenen Gesichtspunkten klassifizieren. Zunächst kann man Verfahren nach dem Sprachniveau unterscheiden, auf denen sie Programme transformieren: Die *algebraische Optimierung* versucht, Programme auf der Ebene der Quellsprache zu transformieren, die *maschinenunabhängige Optimierung* arbeitet auf einer Zwischensprache wie z. B. dem 3-Adress-Code, und schließlich berücksichtigt die *maschinenabhängige Optimierung* die speziellen Eigenschaften des Zielprozessors wie z. B. besondere Adressierungsarten oder die Anzahl von Registern. Insbesondere lassen sich die maschinenunabhängigen Optimierungsmethoden nach ihrem „Blickwinkel“ unterscheiden: So betrachten Verfahren der *lokalen Optimierung* lediglich einen stark begrenzten Ausschnitt des Programms, im Gegensatz zur *globalen Optimierung*. Globale Verfahren sind in der Regel weitaus mächtiger, benötigen aber zusätzliche Informationen z. B. über die Gültigkeit von Variablen, die die sogenannte *Datenflussanalyse* liefert.

Im Folgenden wollen wir neben einer kurzen Beschreibung der jeweiligen Optimierungsmethode auch eine Art von Charakterisierung vornehmen, die den Vergleich der Methoden und deren Auswahl ein wenig erleichtern soll. Zunächst geben wir die Sprachebene(n) an, auf denen die Optimierung operiert. So bezeichnen wir z. B. ein Verfahren, das ausschließlich auf 3-Adress-Code operiert, mit „3AC→3AC“ (für die Quellsprache verwenden wir das Kürzel Q und für Maschinensprache M). In der Tat kann eine Optimierung durchaus zwei Ebenen betreffen, falls es sich um eine in die Übersetzung integrierte Optimierung handelt, wie z. B. die optimale Registerauswahl, die bei der Übersetzung von 3-Adress-Code in Maschinensprache einsetzt (3AC→M). Außerdem geben wir an, ob ggf. gewisse Zusatzstrukturen benötigt werden. Dies sind gerichtete azyklische Graphen (DAG) oder Flussgraphen (FG), die wir weiter unten noch genauer betrachten. Die Schleifenoptimierung wird beispielsweise mit „3AC+FG→3AC“ bezeichnet. Zur einfacheren Darstellung werden wir die Optimierungsideen teilweise auch auf der Quellsprachenebene illustrieren.

8.1.1 Basisblöcke und Flussgraphen

Viele Optimierungstechniken setzen eine Aufteilung des Programms in Basisblöcke voraus: Ein *Basisblock* ist eine maximale Folge von Anweisungen (in 3-Adress-Code), die in jedem Fall nacheinander ausgeführt werden, d. h., es gibt keine Verzweigungen in einen Basisblock hinein oder aus einem Basisblock heraus. Demnach besitzt jeder Basisblock einen eindeutig festgelegten Blockanfang und ein Blockende. Blockanfänge sind außer den ersten Anweisungen einer Prozedur oder des Programms Befehle, deren Adressen als Sprungmarken auftreten, sowie Befehle, die unmittelbar auf Verzweigungsanweisungen folgen. Ein Blockende ist jeweils der letzte Befehl vor einem Blockanfang sowie der letzte Befehl des Programms.

Als Beispiel betrachten wir die Prozedur *Selection Sort* in Abb. 8.1.

```
procedure ssort (n: integer; var x: array[1..n] of real);
var i,j,jmax,temp: integer;
begin
  for i:=n downto 2 do begin
      jmax := 1;
      for j:=2 to i do
          if x[j] > x[jmax] then jmax := j;
      if jmax <> i then begin
        temp := x[i];
        x[i] := x[jmax];
        x[jmax] := temp
      end
  end
end;
```

Abb. 8.1. Prozedur *Selection Sort* (PASCAL)

Die Übersetzung in 3-Adress-Code ist zusammen mit der Gruppierung in Basisblöcke in Abb. 8.2 gezeigt.

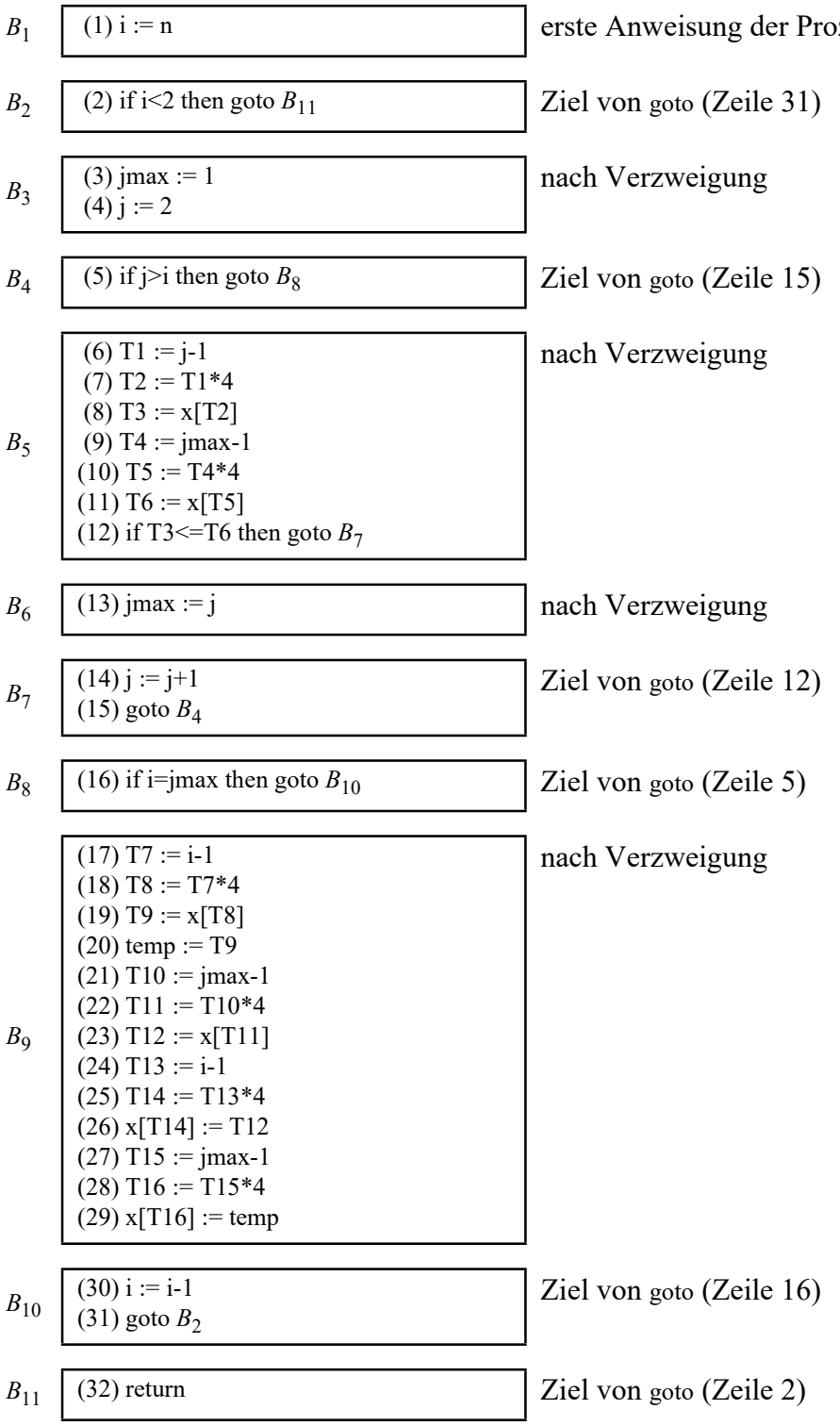


Abb. 8.2. Basisblöcke für *Selection Sort*

Zu jedem Block ist eine Begründung für den Blockanfang mit angegeben. Anstelle von Zeilennummern oder Marken verwenden wir im Folgenden auch Blocknamen als Adressen in goto-Befehlen.

Basisblöcke geben statische Information über den Programmfluss: Die Anweisungen in einem Basisblock werden ja auf jeden Fall alle nacheinander ausgeführt. Hingegen ist die Reihenfolge, in der Basisblöcke durchlaufen werden, im Allgemeinen für verschiedene Programmläufe mit verschiedenen Parametern unterschiedlich.

Ein *Flussgraph* gibt nun die Möglichkeiten an, in der Basisblöcke eines Programms durchlaufen werden können: Die Knoten dieses Graphen sind die Basisblöcke selbst, und eine Kante existiert genau dann von Block B_i zu B_j, wenn während eines Programmlaufs prinzipiell Block B_j direkt nach B_i ausgeführt werden kann. Das heißt, (i) B_j ist Ziel eines Sprungbefehls in B_i oder (ii) B_j folgt im Programmtext direkt auf B_i, und der letzte Befehl von B_i ist kein unbedingter Sprung. Der Flussgraph für unser Beispielprogramm hat demnach die in Abb. 8.3 gezeigte Struktur.

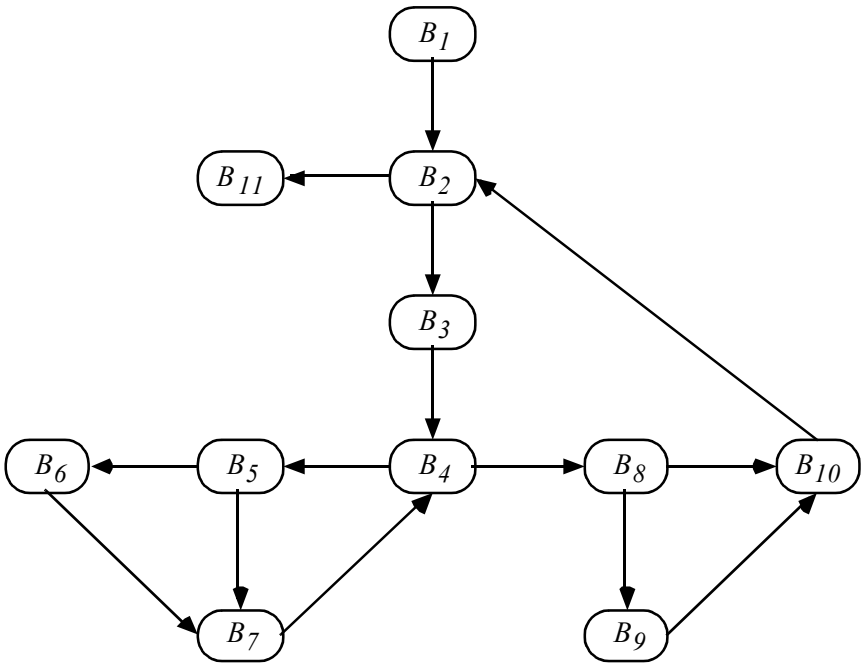


Abb. 8.3. Flussgraph zum Programm *Selection Sort*

Die Menge der *Nachfolger* (bzw. *Vorgänger*) eines Blocks B bezeichnen wir im Folgenden mit $suc(B)$ (bzw. $pred(B)$). Für den Flussgraphen aus Abb. 8.3 gilt z. B. $suc(B_2) = \{B_3, B_{11}\}$ und $pred(B_2) = \{B_1, B_{10}\}$.

Selbsttestaufgabe 8.1: Ermitteln Sie die Basisblöcke und den zugehörigen Flussgraphen für den folgenden 3-Adress-Code:

```
(1) x := 5
(2) z := 3
(3) m := x*z
(4) y := m+x
(5) if m>z then goto 8
(6) y := z-y
(7) if y<m then goto 14
(8) z := x+z
(9) i := m+x
(10) if i>z then goto 3
(11) y := x-z
(12) i := i*3
(13) goto 6
(14) x := z*y
(15) z := x+y
```

□

8.1.2 Algebraische Optimierung

Grundlage für die algebraische Optimierung ist eine mathematische Struktur der zu optimierenden Sprache. Eine solche Struktur existiert für imperative Sprachen meistens überhaupt nicht. Deshalb wird die algebraische Optimierung im „klassischen" Compilerbau so gut wie gar nicht thematisiert.[1] Für funktionale Sprachen gibt es dagegen eine Reihe von Transformationssystemen, mit denen man Programme auf der Quellsprachenebene bereits optimieren kann. Eine ganz wichtige Klasse von Verfahren versucht dabei, Datenstrukturen wie Listen oder Bäume zu eliminieren, die als „Zwischenergebnis" von Funktionen fungieren.[2]

Wir haben in Kapitel 7 Datenstrukturen mehr oder weniger ignoriert. Deshalb beschränken wir uns hier auf ein sehr einfaches Beispiel: Neben der Funktion *map*, die eine Funktion auf alle Elemente einer Liste appliziert, ist *reduce* eine wichtige Funktion auf Listen, die mittels einer binären Funktion die Elemente einer Liste aggregiert. Es gilt:

$$reduce \otimes u\ [x_1, \ldots, x_n] \;=\; x_1 \otimes \ldots \otimes x_n \otimes u$$

Dabei gibt *u* den Wert für die Aggregation einer leeren Liste an. So definiert z. B.

val *sum* = *reduce* + 0

eine Funktion zum Aufsummieren der Elemente einer Liste. Die Gesamtlänge einer Liste von Strings lässt sich also durch den Ausdruck

sum (*map size* ["Eine", "Liste", "von", "Strings"])

---

1 Eine Ausnahme bildet vielleicht die Reduktion der Stärke von Operatoren (s. u.).

2 In imperativen Sprachen ergibt sich dieses Problem zumeist überhaupt nicht, da dynamische Datenstrukturen nur sehr mühsam konstruiert werden können und sich daher die meisten Programmierer davor hüten, mehr Strukturen aufzubauen, als unbedingt nötig sind. In funktionalen Sprachen bilden dynamische Datenstrukturen dagegen ein natürliches Bindeglied zwischen Funktionsaufrufen.

sehr einfach berechnen. Ein Nachteil ist jedoch, dass der geklammerte Ausdruck zunächst die Liste der Zahlen [4, 5, 3, 7] als Zwischenergebnis konstruiert, bevor die Summe ermittelt wird. Dies verbraucht Rechenzeit und insbesondere Heap-Speicherplatz. Als Spezialfall eines sehr allgemeinen Gesetzes gilt nun die Gleichung:

$$\mathit{reduce}\ f\ u\ (\mathit{map}\ g\ l) \quad = \quad \mathit{reduce}\ (\mathbf{fn}\ (x, y) \Rightarrow f\ (g\ x, y))\ u\ l$$

Das heißt, die mittels *map* für jedes Element durchzuführende Berechnung *g* braucht nicht separat durchgeführt zu werden, sondern kann durchaus während der Aggregation der Liste erfolgen, so dass keine Liste als Zwischenergebnis anfällt. Auf das Beispiel angewandt ergibt sich also der Ausdruck:

$$\mathit{reduce}\ (\mathbf{fn}\ (x, y) \Rightarrow \mathit{size}\ x + y)\ 0\ [\text{"Eine"}, \text{"Liste"}, \text{"von"}, \text{"Strings"}]$$

Um den Blickwinkel über Programmiersprachen hinauszulenken, verweisen wir noch auf den Bereich der Datenbanken, in dem algebraische Optimierungen von Anfragen (z. B. in der Relationenalgebra) seit langem Standard sind.

Algebraische Optimierungsverfahren sind äußerst mächtig, setzen aber, wie schon gesagt, eine gewisse mathematische Struktur der zu optimierenden Sprache voraus, damit eine hinreichende Menge von algebraischen Gleichungen formuliert werden kann. Im Wesentlichen zielen algebraische Optimierungsverfahren auf die Ersetzung komplexer Operationen durch äquivalente aber effizientere sowie auf die Eliminierung von Zwischenergebnissen (wie oben) oder auf die Reduzierung von deren Größe (beispielsweise im Bereich der Datenbanken).

8.1.3 Maschinenunabhängige Optimierung

Wir unterscheiden hier lokale und globale Verfahren. *Lokale Optimierungen* beziehen sich auf relativ kleine zusammenhängende Codestücke, die Basisblöcke. Sie sind deshalb auch relativ einfach zu realisieren. Im Gegensatz dazu sind *globale Verfahren* mit erheblich mehr Aufwand verbunden, da sie Informationen z. B. über die Gültigkeit von Variablenwerten über große Programmteile hinweg berücksichtigen müssen. Derartige Informationen werden durch die z. T. recht aufwendige Datenflussanalyse (s. Abschnitt 8.2) bereitgestellt. Bei den globalen Verfahren unterscheidet man oftmals noch die Schleifenoptimierung von den übrigen globalen Verfahren, da diese eine ganz besonders wichtige Form darstellt.

Lokale Optimierung

Einige der lokalen Methoden können auch global eingesetzt werden. Im Unterschied zu globalen Verfahren kann man sie jedoch auch dann einsetzen, wenn man die Datenflussanalyse nicht implementiert hat.

(O1) Konstantenpropagation und Konstantenfaltung (3AC→3AC). Wenn man weiß, dass der Wert einer Variablen an einem bestimmten Auftreten immer konstant ist und man zudem diesen Wert kennt, so kann man an dieser Stelle die Variable

durch ihren Wert ersetzen. Diesen Vorgang nennt man *Konstantenpropagation*. (Typischerweise trifft dies auf Variablen zu, die aus der Übersetzung von Konstanten entstanden sind.)

Ausdrücke mit Konstanten als Argumente kann man bereits zur Übersetzungszeit auswerten. Diesen Vorgang nennt man *Konstantenfaltung*. Man beachte, dass Konstantenfaltung u. U. erst nach vorangegangener Konstantenpropagation möglich ist und umgekehrt, so dass ein mehrfach iteriertes Anwenden beider Methoden z. T. sehr viele Berechnungen bereits zur Compilezeit vornehmen kann. Konstantenpropagation ergibt für das Programmstück

```
x := 3
y := 4
z := x*y
if i<z then goto 123
```

den Code

```
x := 3
y := 4
z := 3*4
if i<z then goto 123
```

Eine Verbesserung ist nun allein schon durch die Tatsache gegeben, dass die Argumente für die dritte Anweisung nicht mehr aus dem Speicher geladen werden müssen (d. h., die Codeerzeugung kann eine effizientere Adressierungsart verwenden). Die Konstantenfaltung bewirkt zudem, dass der dritte Befehl zu einer einfachen Zuweisung `z := 12` wird. Damit aber wird eine erneute Konstantenpropagation ermöglicht, so dass wir nach der dritten Iteration schließlich den folgenden Code erhalten:

```
x := 3
y := 4
z := 12
if i<12 then goto 123
```

Sollte sich nun noch herausstellen, dass die Variable eines propagierten Wertes an keiner anderen Stelle mehr benötigt wird (s. Abschnitt 8.2), so kann man die entsprechenden Initialisierungsanweisungen entfernen.

(O2) Kopierpropagation (3AC→3AC). Ähnlich wie Konstanten kann man auch Variablen propagieren. In einem Programmstück wie z. B.

```
x := y
…
z := a*x
```

kann man die letzte Zeile durch `z := a*y` ersetzen, d. h. also `y` propagieren, sofern bis dahin keine erneute Zuweisung an `x` erfolgt ist. Damit ist noch nicht viel gewonnen, jedoch kann es nun sein, dass – ebenso wie bei der Konstantenpropagation – die Variable `x` möglicherweise nach der Propagierung an keiner Stelle mehr benötigt wird, die Zuweisung kann dann gleichfalls eliminiert werden.

(O3) Reduktion der Stärke von Operatoren (Q→Q) (Q→3AC→M). Die Idee ist, komplexe Operationen in Spezialfällen durch einfachere zu ersetzen. Man kann diese Transformationen bei der Übersetzung oder aber auch im Sinne algebraischer Optimierung direkt auf der Quellsprache durchführen. Auch finden sich derartige Techniken bei der maschinenabhängigen *peephole optimization* (O13). Denkbare Transformationen sind beispielsweise:

```
x**y ⇒ exp(y*ln(x))
x**2 ⇒ x*x
2*x  ⇒ x+x
```

Man beachte, dass der Nutzen solcher Optimierungsregeln sehr stark vom Befehlssatz der Zielmaschine abhängt, so dass die Transformationen über einen evtl. erweiterten 3-Adress-Code eigentlich erst bei der Codeerzeugung vorgenommen werden (s. (O12)).

(O4) In-Line Expansion (Q→M). Bestimmte Anweisungen, die bei „normaler" Übersetzung als Funktionsaufruf realisiert würden, kann man oftmals besser direkt in eine einfache Folge von Maschinenbefehlen übersetzen. So gibt es in vielen Prozessoren beispielsweise einen Befehl, der das Vorzeichen eines Registers setzt oder löscht. Ein Ausdruck wie `abs(j)` könnte dann direkt in einen Maschinenbefehl übersetzt werden.

(O5) Elimination redundanter Berechnungen (3AC+DAG→3AC). Die Anweisungen eines Basisblocks kann man durch einen gerichteten azyklischen Graphen (DAG) darstellen (vgl. auch Abschnitt 6.2.1). Dazu wird jeder Operand und jede Operation als Knoten dargestellt, und es werden Kanten von Operatoren zu ihren Argumenten eingefügt. Der von einem Knoten aus erreichbare Teilgraph repräsentiert somit eine u. U. sehr komplexe Berechnung. Wiederholen sich nun Berechnungen, so wird dies bei der Konstruktion des DAGs erkannt und durch eine (zusätzliche) Kante auf den repräsentierenden Knoten dargestellt. Anschließend erzeugt man aus dem DAG für jeden Knoten eine Anweisung. Dies geschieht in umgekehrter topologischer Sortierreihenfolge, damit die Berechnung für ein Argument stets vor der Benutzung in einem anderen Ausdruck erfolgt.

Im Block B_9 des *Selection-Sort*-Programms (s. Abb. 8.2) wiederholt sich beispielsweise die Berechnung des Teilausdrucks `(i-1)*4` aus den Zeilen 17 und 18 in den Zeilen 24 und 25. Ebenso sind die Zeilen 27 und 28 redundant, da `(jmax-1)*4` bereits in den Zeilen 21 und 22 berechnet wurde. Über die DAG-Konstruktion kann man den folgenden Code erhalten (wir behalten die Originalzeilennummerierung zum einfacheren Vergleich mit dem Original bei):

```
(17) T7 := i-1
(18) T8 := T7*4
(19)
(20) temp := x[T8]
(21) T10 := jmax-1
(22) T11 := T10*4
(23) T12 := x[T11]
(24)
```

```
(25)
(26) x[T8] := T12
(27)
(28)
(29) x[T11] := temp
```

Schleifenoptimierung

Die Optimierung von Schleifen ist besonders attraktiv, da ca. 90% der Ausführungszeit eines Programms in Schleifen verbracht wird.

(O6) Verlagerung von Schleifeninvarianten (3AC+FG→3AC). Eine *schleifeninvariante Berechnung* enthält nur Operanden, deren Werte in der Schleife selbst nicht verändert werden. Im folgenden Programmstück ist z. B. der Teilausdruck `n/i` invariant bez. der inneren Schleife und braucht somit nur für jeden `i`-Wert neu berechnet zu werden. Die Berechnung der Quadratwurzel ist sogar invariant bez. beider Schleifen und braucht gar nur einmal berechnet zu werden.

```
for i:=1 to 1000 do
   for j:=1 to 500 do
      a[i,j] := n/i-j*sqrt(n)
```

Dementsprechend kann man die Schleife wie folgt transformieren:

```
r := sqrt(n);
for i:=1 to 1000 do begin
   q := n/i;
   for j:=1 to 500 do
      a[i,j] := q-j*r
end
```

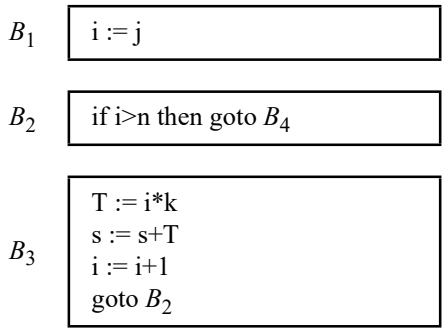
Die Ersparnis beträgt in diesem Beispiel 499999 Quadratwurzelberechnungen und 499000 Divisionen. Die obige Optimierung kann man natürlich als Programmierer selbst schon vornehmen. Aber abgesehen davon, dass der resultierende Code unübersichtlicher wird, gibt es für die Berechnungen z. B. von Array-Positionen ebenfalls Optimierungsmöglichkeiten, die nur dem Compiler zugänglich sind.

(O7) Vereinfachung von Berechnungen mit Schleifenvariablen (3AC+FG→ 3AC). Schleifenvariablen ändern sich kontinuierlich. Diese Eigenschaft kann man ausnutzen, um in Ausdrücken, die Schleifenvariablen benutzen, die Stärke von Operatoren zu reduzieren. Wir betrachten die folgende Schleife:

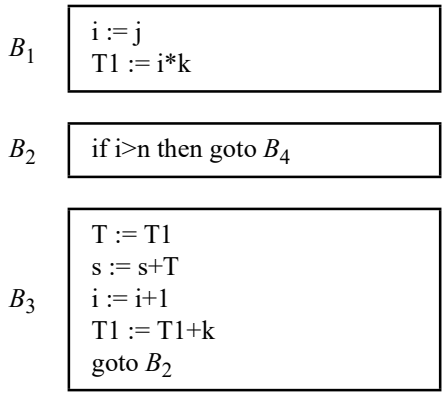
```
for i:=j to n do
   s := s + i*k
```

Die Übersetzung in 3-Adress-Code enthält vor dem Block für die Schleife eine Initialisierung `i := j` sowie einen bedingten Sprung zur Realisierung des Schleifenabbruchs. Der Schleifenblock enthält dann eine Anweisung zur Berechnung des Produkts, gefolgt von Anweisungen zur Berechnung von `s` und schließlich

der Inkrementierung von i und einem unbedingten Rücksprung zur Schleifenbedingung, also:



Man kann nun den Ausdruck i*k, der die Schleifenvariable i und ansonsten nur schleifeninvariante Argumente enthält, durch eine Addition ersetzen. Dazu führen wir eine Hilfsvariable T1 ein, die vor der Schleife zusammen mit i initialisiert und in der Schleife zusammen mit i inkrementiert wird. Ansonsten wird jedes Vorkommen des Ausdrucks i*k durch T1 ersetzt:



(O8) Schleifenentfaltung (3AC→3AC). Die Implementierung von Schleifen beinhaltet immer einen gewissen „Overhead“, z. B. die Initialisierung des Schleifenzählers, dessen Inkrementierung sowie das Überprüfen der Abbruchbedingung. Falls nun der Schleifenrumpf klein ist und die Schleife nicht oft durchlaufen wird, ist es u. U. günstiger, die Schleife durch eine entsprechende Anzahl von Kopien des Schleifenrumpfs zu ersetzen. Dies gilt insbesondere, wenn die zu entfaltende Schleife selbst in einer anderen Schleife enthalten ist und sich so die Einsparung des Overheads entsprechend multipliziert. Im folgenden Beispiel sollte man die innere Schleife entfalten.

```
for i:=1 to 50 do begin
   for j:=1 to 2 do
      write (a[i,j]);
   writeln
end;
```

Man erhält dann (wenn man die write-Befehle zusammenfasst):

```
for i:=1 to 50 do
  writeln (a[i,1], a[i,2]);
```

Globale Optimierung

Unter globaler Optimierung fasst man alle Methoden zusammen, die sich über mehr als einen Basisblock erstrecken und die keine Schleifenoptimierungen sind. Insbesondere sind auch die Konstanten- und Kopierpropagation unter Zuhilfenahme der Datenflussanalyse global durchzuführen.

(O9) Elimination toten Codes (3AC→3AC). Man nennt eine Anweisung *tot*, wenn sie im Laufe des Programmes niemals ausgeführt wird oder wenn deren Effekt an keiner Stelle sichtbar wird. Solche Anweisungen können entfernt werden. Eine Quelle für tote Anweisungen haben wir bereits bei der Kopierpropagation kennengelernt. Ansonsten entstehen tote Anweisungen z. B. beim Testen von Programmen, wenn man Kontrollausgaben erzeugt:

```
if debug then begin
  writeln (…);
  …
end
```

Die Unterdrückung der Kontrollausgaben erreicht man durch die Konstantendeklaration:

```
const
  debug = false;
```

Dadurch werden die Kontrollausgaben zu totem Code und können entfernt werden.

(O10) Code Hoisting (3AC→3AC). Die Elimination gemeinsamer Teilausdrücke mittels DAGs war auf nur einen Basisblock beschränkt. Insbesondere, wenn man gemeinsame Teilausdrücke in verschiedenen Zweigen von bedingten Anweisungen hat, greift diese Methode nicht mehr, da ja die Verzweigungen die Erzeugung mehrerer Basisblöcke bewirken. Wenn man aber mittels Datenflussanalyse feststellen kann, dass jede beteiligte Variable innerhalb des betrachteten Bereichs immer den gleichen Wert hat, so kann man Teilausdrücke auch über Basisblöcke hinweg faktorisieren. Beispiel:

```
if i<j then
  x := i*k+1
else
  if i>j then
    y := i*k+1
  else
    y := (x-i*k)/j
```

Man kann i*k wie folgt faktorisieren:

```
T := i*k
if i<j then
  x := T+1
else
  if i>j then
    y := T+1
  else
    y := (x-T)/j
```

Ausdrücke wie i*k, die in allen Zweigen benötigt werden, nennt man *very busy* (vielbeschäftigt).

Man beachte, dass in diesem Fall das Programm auf Speicherplatzeffizienz optimiert wurde. Die Ausführung wird nicht beschleunigt, da ja in jedem Fall nur ein Zweig mit der Multiplikation ausgeführt wird. Ebenso ist zu beachten, dass die Codeverschiebung eine Verschlechterung bedeutet, wenn der Teilausdruck nicht in allen Zweigen auftritt, denn dann wird ein Zwischenergebnis berechnet, das bei Auswahl des entsprechenden Zweiges gar nicht benötigt wird. Eine zusätzliche Berechnung T1 := T+1 sowie die Ersetzung der rechten Seiten der Zuweisungen an x und y durch T1 hat also zu unterbleiben.

8.1.4 Maschinenabhängige Optimierung

Maschinenabhängige Optimierungen können bereits während der Codeerzeugung erfolgen oder aber auch auf dem resultierenden Maschinenprogramm operieren. Sicherlich hängen die Optimierungsmethoden sehr stark von der Struktur der Zielmaschine ab (vgl. dazu das Maschinenmodell in Abschnitt 8.3.1). Wir nehmen im Weiteren an, dass die Zielmaschine folgende Befehle umfasst:

LOAD R,x	lädt die Konstante oder Variable x in das Register R.
STORE R,x	speichert Registerinhalt an der Adresse der Variablen x.
OP R,x	führt die Operation OP mit den Werten von R und x durch.

(O11) Anweisungsreihenfolge und Registerauswahl (3AC→M). Betrachtet man Folgen von 3-Adress-Code-Befehlen, die zur Berechnung von Ausdrücken dienen, so fällt auf, dass die Reihenfolge der Befehle nur zum Teil festgelegt ist. Die konkrete Reihenfolge hat aber einen wesentlichen Einfluss auf das Maschinenprogramm, das bei der Codeerzeugung daraus gewonnen wird.

Beispielsweise ergibt eine schematische Übersetzung des Ausdrucks (a+b)-(e-(c+d)) die Befehlsfolge:

```
T1 := a+b
T2 := c+d
T3 := e-T2
T4 := T1-T3
```

Man sieht sofort, dass die Zuweisung an T1 durchaus erst nach der für T2 oder auch T3 erfolgen kann. Zunächst aber wird für die gezeigte Folge unter der Annahme, dass

zwei Register R und S zur Verfügung stehen, folgender Maschinencode generiert (beispielsweise mit dem Algorithmus aus Abschnitt 8.3.2):

```
LOAD  R,a
ADD   R,b
LOAD  S,c
ADD   S,d
STORE R,T1
LOAD  R,e
SUB   R,S
LOAD  S,T1
SUB   S,R
STORE S,T4
```

Wenn man nun alternativ die Zuweisung an T1 hinter die an T3 verschiebt, so braucht man das Register R nicht zwischenzuspeichern und erhält den folgenden, um zwei Befehle kürzeren Code:

```
LOAD  R,c
ADD   R,d
LOAD  S,e
SUB   S,R
LOAD  R,a
ADD   R,b
SUB   R,S
STORE S,T4
```

Eine optimale Sortierung von 3-Adress-Code-Folgen (in dem Sinne, dass für eine begrenzte Anzahl von Registern die kürzeste mögliche Folge von Maschinenbefehlen gefunden wird) kann man effizient für Ausdrücke erreichen, die sich als Baum darstellen lassen: Jeder Teilbaum erhält eine Markierung, die angibt, wie viele Register zu seiner Berechnung notwendig sind. Dann wird 3-Adress-Code immer jeweils zuerst für den Teilbaum erzeugt, der mehr Register benötigt. (Für DAGs ist das Problem NP-vollständig, d. h., der Rechenaufwand wächst exponentiell mit der Größe der Eingabe.)

(O12) Befehlsauswahl (3AC→M) (M→M). Manchmal lassen sich bestimmte Spezialfälle von Anweisungen im 3-Adress-Code in spezielle Maschinenbefehle übersetzen. Üblicherweise wird eine arithmetische Berechnungen wie i := i+1 in eine Befehlsfolge wie die folgende übersetzt:

```
LOAD  R,i    Lade den Wert der Speicherzelle i in Register R
ADD   R,1    Addiere 1 zum Register R
STORE R,i    Speichere Register R in der Speicherzelle i
```

Nun werden die allermeisten Prozessoren einen sehr effizienten eingebauten Befehl zur Inkrementierung von Registerinhalten haben. Damit kann man die etwas langsamere Addition in der zweiten Zeile ersetzen. Manche Prozessoren (wie z. B. die der Intel 80*86-Reihe) besitzen sogar einen Befehl, um eine beliebige Speicherzelle zu inkrementieren. Damit kann man anstelle der obigen Befehlsfolge einfach den folgenden Befehl verwenden:

INC i Erhöhe den Wert der Speicherzelle i um 1.

Ein weiteres Beispiel für eine optimierende Befehlsauswahl ist die Ersetzung einer Integer-Multiplikation mit 2 durch einen Shift-Left-Befehl. Derartige Optimierungen kann man auch im Rahmen der *peephole optimization* (O13) durchführen.

Der Phantasie sind hier kaum Grenzen gesetzt, jedoch ist es fraglich, ob derartige, eher marginale, Verbesserungen überhaupt den Aufwand lohnen. Der Erfolg der RISC-Architekturen scheint diese Zweifel zu unterstützen. (RISC steht für **r**educed-**i**nstruction-**s**et **c**omputer; dieser Rechnertyp stellt nur einen sehr beschränkten Satz von Operationen zur Verfügung.)

(O13) Peephole Optimization (M→M). Nach der Codeerzeugung kann man kleine, zusammenhängende Gruppen von Befehlen einer nachträglichen Optimierung unterziehen. Man versucht dabei hauptsächlich, redundante Anweisungen zu eliminieren, die durch ein möglicherweise stereotypes Codeerzeugungsschema eingeführt worden sind.

Wir betrachten die folgenden Anweisungen im 3-Adress-Code:

```
x := y+2
z := x*3
```

Diese werden von naiven Codeerzeugern in den folgenden Maschinencode übersetzt:[3]

```
LOAD  R,y
ADD   R,2
STORE R,x
LOAD  R,x  ⇐
MULT  R,3
STORE R,z
```

Hier fällt sofort auf, dass der Ladebefehl in der vierten Zeile überflüssig ist, da x ja bereits im Register R steht. Man kann solche Situationen gut erkennen, wenn man eine Schablone, die einen Ausschnitt für nur wenige Befehlszeilen bietet, über die Befehlsfolge schiebt. Diese Vorstellung gibt der Methode ihren Namen *peephole optimization* („Guckloch-Optimierung“). Ein weiteres, sehr lohnendes Einsatzgebiet für die peephole optimization ist die Zusammenfassung von Folgen von Sprüngen. Man kann z. B. die Sprungfolge

```
    goto L1
    …
L1: goto L2
```

einfach ersetzen durch

```
    goto L2
    …
L1: goto L2
```

---

[3] Der Algorithmus aus Abschnitt 8.3.2 ist nicht ganz so naiv und vermeidet den überflüssigen Ladebefehl.

(Dies funktioniert natürlich auch mit einem bedingten Sprung.) Ebenso kann man auch Sprünge der Form

```
    goto L1
    …
L1: if a<b then goto L2
L3:
```

ersetzen durch

```
    if a<b then goto L2
    goto L3
    …
L3:
```

Eine zusammenfassende Übersicht über die beschriebenen Methoden findet sich in Abb. 8.4. Ein großer Teil der Verfahren basiert auf Informationen, die die Datenflussanalyse liefern kann. Wir werden diese daher im folgenden Abschnitt genauer untersuchen.

8.2 Datenflussanalyse

Ein Flussgraph repräsentiert die möglichen Abfolgen von Anweisungen, die während eines Programmablaufs entstehen können, d. h., eine Kante von Block B_i zu Block B_j besagt, dass die Anweisungen in B_i denen in B_j vorausgehen können. Das Gleiche gilt, wenn ein Pfad von B_i nach B_j existiert.

Die Datenflussanalyse liefert nun Informationen über die Verfügbarkeit von Variablen und Ausdrücken am Anfang oder Ende von Basisblöcken. Damit man diese Informationen bei der Übersetzung überhaupt ausnutzen kann, müssen die Aussagen unabhängig von einem konkreten Programmablauf sein, d. h., die Datenflussanalyse hat alle möglichen Programmabläufe zu berücksichtigen. Die Berechnung der Aussagen für einen Basisblock erfolgt nun über die entsprechenden Werte aus den Nachbarblöcken und der Information über deren Veränderung innerhalb des betrachteten Blocks. Man unterscheidet prinzipiell vier Arten der Datenflussanalyse, und zwar danach, ob sie vorwärts (zu den Nachfolgern im Flussgraphen) oder rückwärts gerichtet ist (*forward*/*backward analysis*) und ob sie alle oder nur einen benachbarten Block berücksichtigt (*all*/*any analysis*). Wir betrachten zunächst, wie man die gewünschte Information über sogenannte „Datenflussgleichungen“ ausdrükken kann und zeigen anschließend die Lösung der Gleichungen.

Im Weiteren benötigen wir die folgenden Begriffe für Programme im 3-Adress-Code: Eine *Definition* der Variablen x ist eine Zuweisung an x. Eine *Anwendung* der Variablen x ist das Auftreten von x als Operand eines Befehls. Eine *Stelle p* in einem Programm ist eine Position vor einem Befehl (also gegeben z. B. durch die Zeilennummer *p*) oder die Position nach dem letzten Befehl des Programms.

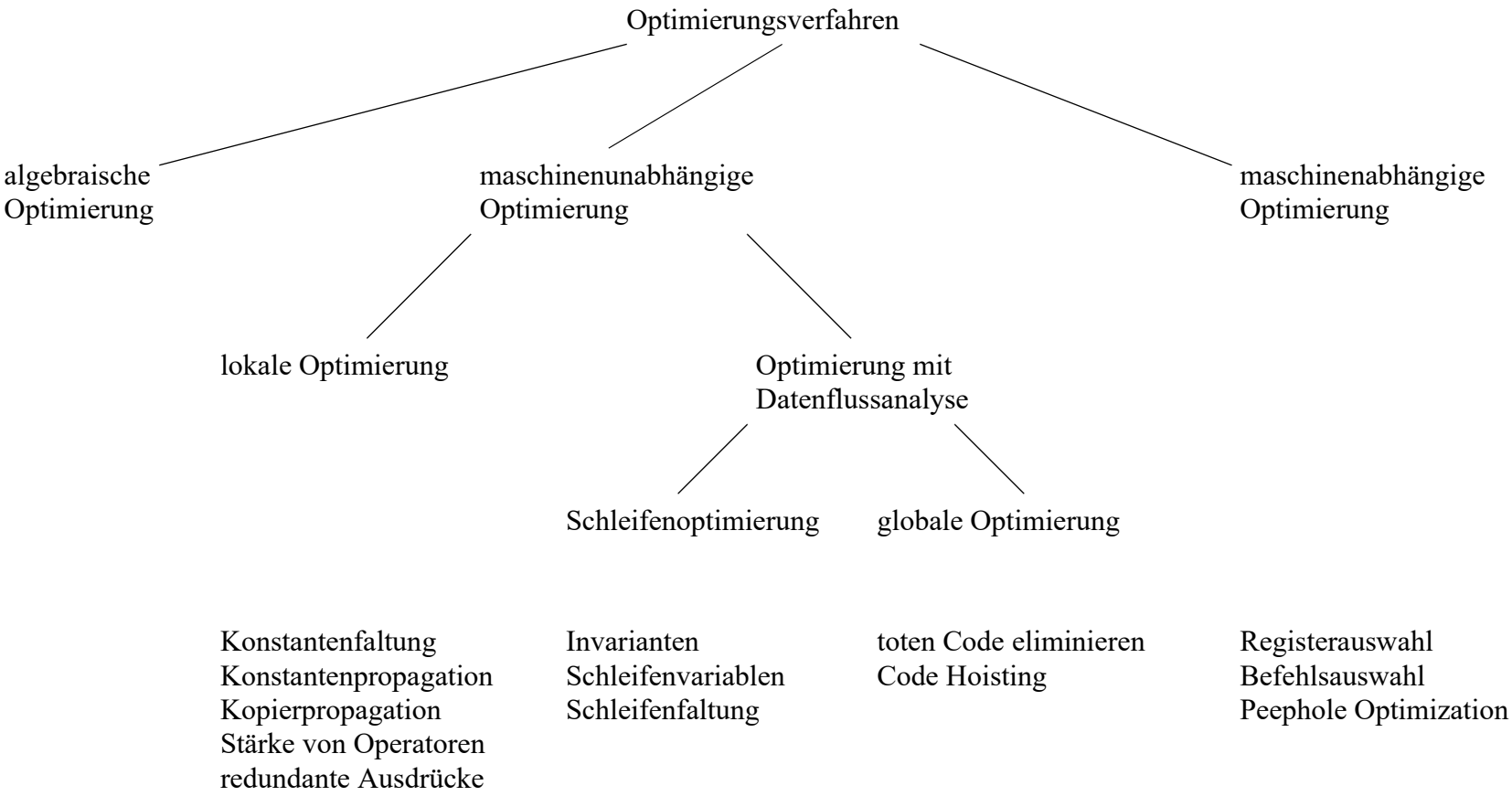


Abb. 8.4. Übersicht über Optimierungsverfahren

8.2.1 Datenflussgleichungen

Datenflussgleichungen beschreiben für jeden Basisblock eine Menge von Werten. Dies sind z. B. verfügbare Ausdrücke (für die Elimination gemeinsamer Teilausdrücke) oder definierende Stellen von Variablen (zur Erkennung von Schleifeninvarianten). Die allgemeine Form einer Datenflussgleichung besteht aus drei Teilen:

1. einer Initialisierung für die Wurzel (den letzten Block)
2. einer Gleichung zur Berechnung der Werte am Anfang (Ende) eines Blocks auf der Basis der Werte am Ende (Anfang) aller Vorgänger (Nachfolger)
3. einer Gleichung zur Berechnung der Werte am Ende (Anfang) eines Blocks auf der Basis der Werte am Anfang (Ende) des Blocks und der im Block selbst erzeugten und vernichteten Werte.

Dies beschreibt die Vorwärtsanalyse; in Klammern stehen die Angaben für die Rückwärtsanalyse. Mit den Gleichungen werden für jeden Block vier Mengen definiert: *in*(*B*) und *out*(*B*) enthalten die Mengen der am Anfang bzw. am Ende des Blocks *B* gültigen Informationen; *gen*(*B*) und *kill*(*B*) bezeichnen die im Block *B* erzeugten bzw. vernichteten Werte.

Vorwärtsanalyse

Als erstes Beispiel für die Vorwärtsanalyse beschreiben wir die Berechnung der sogenannten *UD-Verkettung* (*use/definition-chaining*); die gibt zu jeder Anwendung einer Variablen die Menge all der Definitionen an, die sie erreichen. Diese Information kann man dann beispielsweise zur Propagation von Konstanten oder Variablen benutzen.

Es sei *D* eine Definition der Variablen x an der Stelle *d*. Die in einer Definition definierte Variable (also x) bezeichnen wir mit *V*(*D*). Man sagt nun, *D erreicht* die Stelle *p* im Programm, falls es einen Pfad im Flussgraphen von *d* nach *p* gibt, auf dem keine andere Definition für x liegt. Definition *D* ist *hinter p gültig*, wenn sie *p* erreicht und wenn *p* keine Definition von x ist. *D erreicht das Ende von B*, wenn *D* hinter dem letzten Befehl des Blocks *B* gültig ist, und *D erreicht* (*den Anfang von*) *B*, wenn es einen Block *B*' $\in$ *pred*(*B*) gibt, so dass *D* das Ende von *B*' erreicht.

Wie sehen nun die Datenflussgleichungen für die UD-Verkettung aus? Für die Initialisierung beobachten wir, dass am Anfang von Block 1 keine Variablendefinition existiert, d. h.:[4]

$$in(B_1) = \varnothing$$

[4] Falls der Flussgraph eine Prozedur repräsentiert, kann man allerdings in $in(B_1)$ alle Parameter aufnehmen.

Des Weiteren sind die am Anfang eines Blocks B_i gültigen Definitionen gerade alle die, die am Ende irgendeines Vorgängerblocks gültig sind, d. h. also:

$$in(B_i) = \bigcup_{B_j \in pred(B_i)} out(B_j)$$

Dies bedeutet beispielsweise für den Anfang von Block B_2 aus dem *Selection Sort* Programm, dass dort die am Ende der Blöcke B_1 und B_{10} gültigen Definitionen (dies sind die Stellen 1 und 30) gültig sind.

Schließlich betrachten wir die Definitionen, die das Ende eines Blocks B_i erreichen. Dies sind zunächst alle Definitionen für Variablen in B_i, die in B_i selbst nicht mehr überschrieben werden, d. h., für jede Variable erreicht deren letzte Definition in B_i das Ende von B_i. Diese Menge wird mit $gen(B_i)$ bezeichnet. Zusätzlich erreichen aber auch alle Definitionen D das Ende von B_i, die den Anfang von B_i erreichen und für die es keine Definition für $V(D)$ in B_i gibt. $kill(B_i)$ bezeichnet nun alle Definitionen D, die außerhalb von B_i liegen und für die es eine Definition für $V(D)$ in B_i gibt. Damit gilt dann:

$$out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i))$$

Beispielsweise ist $gen(B_7) = \{14\}$, und da die Definition 4 den Anfang von B_7 erreicht, enthält $kill(B_7)$ die Definition 4.

Für einen Flussgraphen mit n Knoten erhalten wir also ein Gleichungssystem mit $2n$ Gleichungen. Wie man eine Lösung dafür berechnet, betrachten wir in Abschnitt 8.2.2.

Die UD-Verkettung ist ein Beispiel für eine *forward-any*-Analyse, da die Information entlang der Kanten von Vorgängern zu Nachfolgern propagiert wird und da für die Gültigkeit in einem Block die Gültigkeit in nur einem der Vorgänger gefordert wird. Als Beispiel für eine *forward-all*-Analyse betrachten wir nachfolgend die Berechnung verfügbarer Ausdrücke, die zur Elimination redundanter Berechnungen genutzt werden kann. Man beachte, dass die Werte in den Mengen *in*, *out*, *gen* und *kill* je nach Anwendung völlig verschieden sind. Waren es bei der UD-Verkettung einfach Zeilennummern, so sind es in der folgenden Anwendung Ausdrücke.

Am Anfang des Wurzelblocks sind noch keine Ausdrücke verfügbar:

$$in(B_1) = \varnothing$$

Ein Ausdruck steht nun nur dann am Anfang eines Blocks B_i zur Verfügung, wenn er am Ende eines jeden Vorgängerblocks zur Verfügung steht. Das heißt, da man den konkreten Programmablauf ja nicht kennt, muss man die Verfügbarkeit für alle Vorgänger fordern, um sicher zu sein. Die entsprechende Datenflussgleichung lautet:

$$in(B_i) = \bigcap_{B_j \in pred(B_i)} out(B_j)$$

Zur Notwendigkeit der *forward-all*-Analyse sei angemerkt: Würde man für einen Ausdruck in B_i die Verfügbarkeit annehmen, der in einem Vorgänger (z. B. B_j) nicht verfügbar ist, dann könnte eine Optimierung, die die Berechnung des Ausdrucks in B_i entfernt, zu einem undefinierten Verhalten des Programmes führen, da ja B_i über B_j erreicht werden kann.

Für die Ausdrücke, die am Ende eines Blocks zur Verfügung stehen, gilt das Gleiche wie bei der UD-Verkettung: Alle Ausdrücke, die in B_i berechnet werden (d. h. $gen(B_i)$), stehen am Ende von B_i zur Verfügung; darüber hinaus alle Ausdrücke, die bereits am Anfang von B_i verfügbar sind ($in(B_i)$) und deren Operanden in B_i nicht verändert werden. $kill(B_i)$ bezeichne alle Ausdrücke, deren Operanden verändert werden. Dann gilt:

$$out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i))$$

Wir sehen, dass die Mengen *in* und *out* im Wesentlichen als Schnittstelle zwischen Blöcken fungieren, die semantisch relevanten Informationen für eine spezielle Analyse kommen von den Funktionen *gen* und *kill*, für die man jeweils eigene Implementierungen angeben muss. Die in einem Block B_i erzeugten verfügbaren Ausdrücke berechnet man wie folgt: Am Anfang von B_i sei $gen(B_i) = \emptyset$. Nun durchläuft man den Block schrittweise, und für jede Anweisung x := y op z fügt man y op z zu $gen(B_i)$ hinzu und entfernt jeden Ausdruck, der x enthält. Die Menge $kill(B_i)$ enthält alle Ausdrücke der Form x op y, so dass es für x oder y in B_i eine Definition gibt und x op y $\notin gen(B_i)$.

Rückwärtsanalyse

Die Datenflussgleichungen für Rückwärtsanalysen sind denen der Vorwärtsanalyse sehr ähnlich, man bewegt sich allerdings im Flussgraphen entgegen den Kantenrichtungen von Nachfolgern zu Vorgängern.

Wir betrachten als Beispiel zunächst eine *backward-any*-Analyse, und zwar die Ermittlung von lebenden (und toten) Variablen. Die Werte in den Mengen *in*, *out*, *gen* und *kill* sind also nunmehr Variablen. Eine Variable *lebt* an einer Stelle, wenn ihr Wert möglicherweise noch benötigt wird; erfährt dagegen (von einer Stelle *p* aus betrachtet) eine Variable vor jeder möglichen Verwendung eine Zuweisung, ist sie tot, d. h., ihr Wert wird nicht mehr benötigt. Diese Information ist bei der Registerauswahl während der Codeerzeugung äußerst hilfreich: Register, die Werte von toten Variablen enthalten, können ohne Weiteres mit neuen Werten geladen werden; die Variablen brauchen zuvor nicht gespeichert zu werden.

Mit der Initialisierung beginnen wir am Ende des Programms, d. h., am Ende des Programms leben sicherlich keine Variablen mehr:

$$out(B_n) = \emptyset$$

Eine Variable lebt nun am Ende eines Blocks B_i, wenn sie in irgendeinem nachfolgenden Block benötigt wird, d. h., wenn sie insbesondere zu Beginn eines Nachfolgerblocks lebt. Die entsprechenden Datenflussgleichungen lauten:

$$out(B_i) = \bigcup_{B_j \in suc(B_i)} in(B_j)$$

Zu Beginn eines Blocks B_i leben zunächst all die Variablen $gen(B_i)$, deren erstes Vorkommen eine Anwendung ist. (Dabei geht in einer Anweisung x := x op y die Anwendung der Variablen x deren Definition voraus.) Jede Definition D in einem Block „verdeckt“ eine spätere mögliche Anwendung von $V(D)$ (in einem Nachfolgerblock), und so enthält $kill(B_i)$ alle Variablen aller Definitionen von B_i. Nun leben am Anfang von Block B_i auch alle Variablen, die am Ende von B_i leben und die nicht durch eine Definition in B_i „getötet“ werden, d. h. also:

$$in(B_i) = gen(B_i) \cup (out(B_i) - kill(B_i))$$

Schließlich ist ein Beispiel für die *backward-all*-Analyse die Berechnung von very-busy-Ausdrücken. Ein Ausdruck ist an einer Stelle p very busy, wenn er auf allen Wegen von dort aus benutzt wird, bevor eine Redefinition (eines seiner Operanden) erfolgt. Die Menge von very-busy-Ausdrücken am Ende des Programms ist leer:

$$out(B_n) = \varnothing$$

Ausdrücke sind very busy am Ende eines Blocks, wenn sie am Anfang eines jeden Nachfolgers auch very busy sind:

$$out(B_i) = \bigcap_{B_j \in suc(B_i)} in(B_j)$$

Am Anfang eines Blocks B_i sind diejenigen Ausdrücke very busy, die in B_i benutzt werden, sowie die, die am Ende von B_i bereits very busy sind und deren Operanden innerhalb des Blocks nicht redefiniert werden (letztere werden durch die Menge $kill(B_i)$ beschrieben):

$$in(B_i) = gen(B_i) \cup (out(B_i) - kill(B_i))$$

8.2.2 Das Lösen von Datenflussgleichungen

Für strukturierte Programme, d. h. für Programme, die lediglich aus Sequenzen, Fallunterscheidungen und Schleifen bestehen, kann man die *in*- und *out*-Mengen relativ einfach durch syntaxgesteuerte Übersetzung berechnen. Im allgemeinen Fall wendet man, ausgehend von der Initialisierung der Wurzel (bzw. des letzten Blocks bei Rückwärtsanalysen), die Gleichungen für jeden Block so lange an, bis sich in den Mengen *in* und *out* keine Änderungen mehr ergeben.

Wir berechnen beispielhaft die UD-Verkettung für das *Selection-Sort*-Programm, wobei wir Array-Variablen außer Acht lassen wollen. Die Definitionen für die übrigen (Nicht-Hilfs-)Variablen sind:

Variable	Zeile	Block
i	1	B_1
	30	B_{10}
j	4	B_3
	14	B_7
jmax	3	B_3
	13	B_6
temp	20	B_9

Zunächst ermitteln wir die *gen*- und *kill*-Mengen für alle Blöcke. Wir erinnern uns, dass $gen(B_i)$ die Menge der Stellen ist, die eine Variable das letzte Mal in B_i definieren, und dass $kill(B_i)$ die Stellen außerhalb von B_i enthält, die eine Variable definieren, die auch in B_i definiert wird; die Mengen $gen(B_i)$ kann man direkt aus der Tabelle entnehmen, und die Mengen $kill(B_i)$ enthalten für jede Variable, die in B_i definiert wird, alle Definitionen außerhalb von B_i.

Wir notieren die Mengen in einer Tabelle:

Block	*gen*	*kill*
B_1	{1}	{30}
B_2	{}	{}
B_3	{3, 4}	{13, 14}
B_4	{}	{}
B_5	{}	{}
B_6	{13}	{3}
B_7	{14}	{4}
B_8	{}	{}
B_9	{20}	{}
B_{10}	{30}	{1}
B_{11}	{}	{}

Für alle Blöcke B_i nehmen wir anfänglich an: $in(B_i) = \{\}$. Damit erhalten wir für die Mengen $out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i)) = gen(B_i)$:

Block	*in*	*out*
B_1	{}	{1}
B_2	{}	{}
B_3	{}	{3, 4}
B_4	{}	{}
B_5	{}	{}
B_6	{}	{13}
B_7	{}	{14}
B_8	{}	{}
B_9	{}	{20}
B_{10}	{}	{30}
B_{11}	{}	{}

Nun müssen wir iteriert $in(B_i)$ und $out(B_i)$ neu berechnen, bis sich keine Änderung mehr (in irgendeiner Menge $out(B_j)$) ergibt. Dabei ermittelt man $in(B_i)$ aus der Vereinigung der Mengen $out(B_j)$ aller Vorgänger B_j von B_i und $out(B_i)$ gemäß der obigen Gleichung. Für die Mengen $in(B_i)$ und $out(B_i)$ sind also stets die folgenden Gleichungen zu berechnen:

$$in(B_1) = \{\} \qquad out(B_1) = \{1\} \cup (\{\} - \{30\}) = \{1\}$$
$$in(B_2) = out(B_1) \cup out(B_{10}) \qquad out(B_2) = in(B_2)$$
$$in(B_3) = out(B_2) \qquad out(B_3) = \{3, 4\} \cup (in(B_3) - \{13, 14\})$$
$$in(B_4) = out(B_3) \cup out(B_7) \qquad out(B_4) = in(B_4)$$
$$in(B_5) = out(B_4) \qquad out(B_5) = in(B_5)$$
$$in(B_6) = out(B_5) \qquad out(B_6) = \{13\} \cup (in(B_6) - \{3\})$$
$$in(B_7) = out(B_5) \cup out(B_6) \qquad out(B_7) = \{14\} \cup (in(B_7) - \{4\})$$
$$in(B_8) = out(B_4) \qquad out(B_8) = in(B_8)$$
$$in(B_9) = out(B_8) \qquad out(B_9) = \{20\} \cup in(B_9)$$
$$in(B_{10}) = out(B_8) \cup out(B_9) \qquad out(B_{10}) = \{30\} \cup (in(B_{10}) - \{1\})$$
$$in(B_{11}) = out(B_2) \qquad out(B_{11}) = in(B_{11})$$

In der ersten Iteration berechnet man die Mengen wie folgt: Da B_1 keine Vorgänger im Flussgraphen hat, bleiben $in(B_1)$ und $out(B_1)$ unverändert. Für $in(B_2)$ berechnen wir $out(B_1) \cup out(B_{10}) = \{1\} \cup \{30\} = \{1, 30\}$. Da $gen(B_2) = kill(B_2) = \{\}$ ist, ergibt sich für $out(B_2)$ ebenfalls die Menge $\{1, 30\}$. Ebenso ist $in(B_3) = out(B_2) = \{1, 30\}$.

Da $gen(B_3) = \{3, 4\}$, ergibt sich für $out(B_3) = \{1, 3, 4, 30\}$. Führt man dies fort, so ergeben sich die folgenden Werte:

Block	*in*	*out*
B_1	{}	{1}
B_2	{1, 30}	{1, 30}
B_3	{1, 30}	{1, 3, 4, 30}
B_4	{1, 3, 4, 14, 30}	{1, 3, 4, 14, 30}
B_5	{1, 3, 4, 14, 30}	{1, 3, 4, 14, 30}
B_6	{1, 3, 4, 14, 30}	{1, 4, 13, 14, 30}
B_7	{1, 3, 4, 13, 14, 30}	{1, 3, 13, 14, 30}
B_8	{1, 3, 4, 14, 30}	{1, 3, 4, 14, 30}
B_9	{1, 3, 4, 14, 30}	{1, 3, 4, 14, 20, 30}
B_{10}	{1, 3, 4, 14, 20, 30}	{3, 4, 14, 20, 30}
B_{11}	{1, 30}	{1, 30}

In der nächsten Iteration legt man die neu berechneten Mengen $out(B_i)$ zur Bestimmung der $in(B_i)$ zugrunde. Zum Beispiel erhält man für $in(B_2) = out(B_1) \cup out(B_{10}) = \{1\} \cup \{3, 4, 14, 20, 30\} = \{1, 3, 4, 14, 20, 30\}$. Wir erhalten folgende Werte:

Block	*in*	*out*
B_1	{}	{1}
B_2	{1, 3, 4, 14, 20, 30}	{1, 3, 4, 14, 20, 30}
B_3	{1, 3, 4, 14, 20, 30}	{1, 3, 4, 20, 30}
B_4	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_5	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_6	{1, 3, 4, 13, 14, 20, 30}	{1, 4, 13, 14, 20, 30}
B_7	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 13, 14, 20, 30}
B_8	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_9	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_{10}	{1, 3, 4, 13, 14, 20, 30}	{3, 4, 13, 14, 20, 30}
B_{11}	{1, 3, 4, 14, 20, 30}	{1, 3, 4, 14, 20, 30}

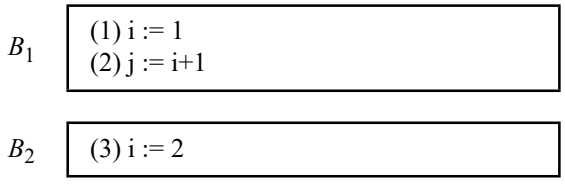
Die dritte Iteration ergibt:

Block	*in*	*out*
B_1	{}	{1}
B_2	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_3	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 20, 30}
B_4	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_5	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_6	{1, 3, 4, 13, 14, 20, 30}	{1, 4, 13, 14, 20, 30}
B_7	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 13, 14, 20, 30
B_8	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_9	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}
B_{10}	{1, 3, 4, 13, 14, 20, 30}	{3, 4, 13, 14, 20, 30}
B_{11}	{1, 3, 4, 13, 14, 20, 30}	{1, 3, 4, 13, 14, 20, 30}

Im folgenden Iterationsschritt ändern sich die Mengen $out(B_i)$ nicht mehr. In diesem Beispiel lassen sich mit den Mengen keine Optimierungen durchführen; im Allgemeinen helfen die Informationen darüber, welche Definitionen eine bestimmte Anwendung einer Variablen erreicht, z. B. bei der globalen Kopierpropagation. Um y aus der Zuweisung x := y (an der Stelle *d*) zu einer Anwendung z := a*x (an der Stelle *a*) propagieren zu können, d. h. diese durch z := a*y ersetzen zu können, muss *d* die einzige Definition von x in der UD-Verkettung an der Stelle *a* sein.

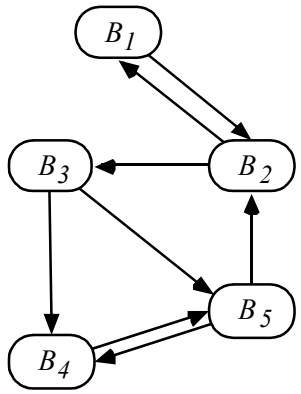
Wir wollen noch kurz begründen, warum das angegebene Berechnungsverfahren für Datenflussgleichungen terminiert: Da in jedem Schritt Informationen zu den Mengen hinzugefügt werden und diese niemals wieder entfernt werden, wachsen die Mengen stetig. Da die Menge aller Definitionsstellen endlich ist, wird zu irgendeinem Zeitpunkt keine Information mehr hinzuzufügen sein, d. h., es ergibt sich keine Änderung der Mengen $out(B_i)$. Dann aber bleiben auch die Mengen $in(B_i)$ unverändert, und in einem nächsten Schritt würden sich auch keine Änderungen mehr an den Mengen $out(B_i)$ ergeben. Deshalb ist das Abbruchkriterium korrekt, und der Algorithmus terminiert nach endlicher Zeit.

Selbsttestaufgabe 8.2: Gegeben seien die folgenden Basisblöcke, in denen lediglich Variablendefinitionen (fortlaufend nummeriert) dargestellt sind:



B_3	(4) j := j+1
B_4	(5) j := j-4 (6) i := 3*j (7) j := 0
B_5	

Dazu sei der folgende Flussgraph gegeben:



(a) Ermitteln Sie die Mengen $gen(B_i)$ und $kill(B_i)$ $(1 \le i \le 5)$ zur Bestimmung der UD-Verkettung.

(b) Berechnen Sie die Mengen $in(B_i)$ und $out(B_i)$ $(1 \le i \le 5)$. □

8.3 Codeerzeugung

Wir beschreiben im Folgenden ein sehr einfaches Verfahren, das 3-Adress-Code schrittweise in eine Maschinensprache übersetzt, die in Abschnitt 8.3.1 kurz beschrieben ist. Dabei wird versucht, Werte so lange wie möglich in Registern zu halten und das Wissen über Operanden in Registern zum Einsparen von Ladebefehlen auszunutzen. In der Beschreibung der Codeerzeugung beschränken wir uns auf Basisblöcke und klammern somit die Erzeugung von Sprungbefehlen der Einfachheit halber aus.

8.3.1 Maschinenmodell

Anstelle eines konkreten Prozessors wählen wir ein sehr einfaches, aber dennoch realistisches Maschinenmodell. Der imaginäre Prozessor besteht aus einem Datenspeicher und einem Programmspeicher, und er enthält n Register. Es gibt vier Arten

von Befehlen: Ladebefehle, Speicherbefehle, (arithmetische) Operationen auf Registern und Sprungbefehle (die wir nicht weiter betrachten werden).

Bei der Bezeichnung von Befehlsargumenten verwenden wir die folgenden Konventionen: R, S, T usw. bezeichnen Register, C bezeichnet Konstanten, V, W, U usw. bezeichnen Speicheradressen, und X, Y, Z usw. bezeichnen Register, Speicheradressen oder Konstanten. OP steht für eingebaute Befehle, die auf einem Register und evtl. einem zusätzlichen Argument operieren. Wir nehmen vereinfachend an, dass es für jede Operation op des 3-Adress-Codes einen entsprechenden Maschinenbefehl OP gibt.

LOAD R,X	lädt X in das Register R. Falls X eine Konstante ist, so wird eben dieser Wert geladen; ist X eine Speicheradresse oder ein anderes Register, so wird der an dieser Stelle gespeicherte Wert geladen.
STORE R,V	speichert den Wert des Registers R an der Stelle V, d. h. in einer Speicherzelle. (Registertransfers können durch LOAD realisiert werden).
OP R,X (OP R)	führt die binäre (unäre) Operation OP auf dem Wert des Registers R (und dem Wert von X) durch und speichert das Ergebnis anschließend in R.

Üblicherweise geht man davon aus, dass ein Registertransfer (d. h. ein Befehl LOAD R,S) schneller abläuft als das Laden einer Speicherzelle (LOAD R,V).

8.3.2 Codeerzeugung für Basisblöcke

In der Codeerzeugung wird zu jedem 3-Adress-Code-Befehl eine Folge von Maschinenbefehlen generiert. Dabei müssen Variablenbezeichner durch Speicheradressen oder Register ersetzt werden. Zur Vereinfachung bezeichnen wir Speicheradressen mit symbolischen Namen. Dabei verwenden wir gerade den Namen der Variablen, die an der betreffenden Adresse gespeichert ist, und zwar als Großbuchstaben, damit im jeweiligen Kontext immer deutlich wird, ob die Variable oder die Speicheradresse gemeint ist. Also bezeichnet z. B. V die Speicheradresse der Variablen v.

Der nachfolgend beschriebene Algorithmus zur Codeerzeugung verwaltet zur effektiven Ausnutzung der Register die folgenden Informationen:

Registerinhalte. Für jedes Register R bezeichnet *Con*(R) (*contents*) zu jedem Zeitpunkt die Menge von Variablennamen, deren Wert im Register R gespeichert ist. Zu Beginn eines Basisblocks sind alle Register leer, d. h. *Con*(R) = ∅. Jeder Befehl, der auf dem Register R operiert, bewirkt eine Änderung von *Con*(R). In der Tat können durch Kopierbefehle verschiedene Variablennamen gleichzeitig einem Register zugeordnet sein.

Variablenpositionen. Für jede Variable v gibt *Pos*(v) die Stelle an, an der sich der aktuelle Wert von v gerade befindet. Dies ist im Allgemeinen ein Regis-

ter oder eine Speicheradresse; *Pos*(v) kann aber auch undefiniert sein, wenn v keine Anwendung mehr im aktuellen Block hat und an dessen Ende nicht mehr lebt. Zu Beginn eines Blocks befinden sich alle Variablenwerte im Speicher, d. h. *Pos*(v) = V. Prinzipiell könnte man auch in *Pos* Mengen verwalten, da durch Kopieranweisungen Werte von Variablen an mehreren Stellen gleichzeitig auftreten können. Allerdings wird dadurch sowohl die Registerverwaltung als auch die Codeerzeugung erheblich komplizierter.

Darüber hinaus benötigen wir für jede Variable v Informationen darüber, ob sie (i) am Ende des Blocks noch lebt und (ii) ob sie noch eine weitere Anwendung im Block hat. Diese Aussagen bezeichnen wir im Folgenden mit *live*(v) bzw. *used*(v).

Im Weiteren benutzen wir eine Hilfsfunktion *getreg*(x), die ein geeignetes Register für die Ausführung eines Maschinenbefehls auf der Basis des Wertes x (Konstante oder Variable im 3-Adress-Code) auswählt. *getreg*(x) wird jeweils bei der Übersetzung eines 3-Adress-Code-Befehls aufgerufen. *getreg*(x) liefert ein Register, in dem sich x bereits befindet (sofern der Wert von x überschrieben werden darf) oder ein neues Register, in das der Wert von x zu laden ist.

Algorithmus *getreg*(x).

1. Falls (i) *Pos*(x) = R und *Con*(R) = {x} (d. h., R enthält nur den Wert von x),
   (ii) ¬*used*(x) und (iii) ¬*live*(x), dann:
   Con(R) := ∅; **return** R.
2. Anderenfalls wähle ein freies Register R; **return** R.
3. Gibt es kein freies Register mehr, so wähle ein belegtes Register R aus.
   Für alle y ∈ *Con*(R) mit *Pos*(y) ≠ Y:
   Generiere einen Befehl STORE R,Y; *Pos*(y) := Y;
   return R.

Für die Auswahl eines belegten Registers im dritten Schritt gibt es kein anerkannt bestes Verfahren. In den Beispielen werden wir immer versuchen, anstelle eines beliebigen Registers ein solches auszuwählen, das im konkreten Beispiel die Anzahl der LOAD/STORE-Befehle minimiert.

Im Folgenden bezeichnet $\underline{x}$ die bei der Codeerzeugung bevorzugte Adresse von x, d. h.

$\underline{x}$ = X, falls: (i) x ist eine Konstante oder (ii) *Pos*(x) ist kein Register
$\underline{x}$ = S, falls *Pos*(x) = S (S sei ein Register)

Der Algorithmus zur Codeerzeugung hat als Eingabe die Folge von 3-Adress-Befehlen eines Basisblocks sowie für jede Variable Informationen darüber, ob sie am Ende des Blocks lebt und ob sie noch eine Anwendung hat. Die Anweisungen des Basisblocks werden nacheinander bearbeitet, und wir beschreiben mit dem Algorithmus *codegen* die Übersetzung einer 3-Adress-Code-Anweisung *stat* abhängig von deren konkreter Gestalt (A)-(D):

Algorithmus *codegen*(*stat*:3-Adress-Code).

(A) *stat* = v := x op y (x, y: Konstanten oder Variablen)
(B) *stat* = v := op x (x: Konstante oder Variable)
(C) *stat* = v := x (x: Variable)

1. R := *getreg*(x).
2. Falls *Pos*(x) ≠ R, erzeuge den Befehl LOAD R,x.
3. (A): Erzeuge den Befehl OP R,y.
   (B): Erzeuge den Befehl OP R.
   Pos(v) := R.
   (A, B): *Con*(R) := {v}
   (C): *Con*(R) := {v, x}
4. Für alle Register S: x ∈ *Con*(S) ∧ ¬*used*(x) ∧ ¬*live*(x): *Con*(S) := *Con*(S) – {x}
   (A): Für alle Reg. S: y ∈ *Con*(S) ∧ ¬*used*(y) ∧ ¬*live*(y): *Con*(S) := *Con*(S) – {y}
   Für alle Register S ≠ R mit v ∈ *Con*(S): *Con*(S) := *Con*(S) – {v}

(D) *stat* = v := c (c: Konstante)

1. R := *getreg*(v).
2. Erzeuge den Befehl LOAD R,C.
   Pos(v) := R; *Con*(R) := {v}

Als Beispiel wollen wir Maschinencode für den folgenden Block von 3-Adress-Code-Befehlen erzeugen. Wir nehmen an, dass drei Register R, S und T zur Verfügung stehen und dass am Ende des Blocks nur die Variablen z und v leben.

```
y := 1
z := 2
x := y+z
v := x-y
v := z*v
z := y
y := z*v
v := y+z
```

Zu Beginn sind alle drei Register frei, und die Position jeder Variablen ist ihre Speicheradresse. Im Folgenden notieren wir *Con* und *Pos* als partielle Abbildungen, d. h. *Con* = {} und *Pos* = {v ↦ V, x ↦ X, y ↦ Y, z ↦ Z}.

Zunächst erzeugt *codegen* zweimal Code gemäß Fall (D). *getreg* liefert dabei jeweils ein freies Register:

```
LOAD  R,1
LOAD  S,2
```

Danach gilt *Con* = {R ↦ {y}, S ↦ {z}}, *Pos* = {v ↦ V, x ↦ X, y ↦ R, z ↦ S}. Als nächstes wird *codegen* auf x := y+z angewandt, d. h. also eine Anweisung vom Typ (A). Im ersten Schritt liefert *getreg*(y) das nächste freie Register T (und nicht etwa R, da es im Block noch weitere Anwendungen von y gibt). Im zweiten Schritt muss dann ein Ladebefehl generiert werden (genauer gesagt, ein Registertransfer, da y = R). Danach wird der dem 3-Adress-Befehl + entsprechende Maschinenbefehl ADD erzeugt, wobei der Operand ebenfalls ein Register ist:

```
LOAD  T,R
ADD   T,S
```

Für die Register und Variablen ergibt sich nun: *Con* = {R ↦ {y}, S ↦ {z}, T ↦ {x}} und *Pos* = {v ↦ V, x ↦ T, y ↦ R, z ↦ S}. Nun ruft *codegen*(v := x-y) die Funktion *getreg*(x) auf. Da x als einzige Variable im Register T steht, im Rest des Blocks keine Anwendung mehr hat und an dessen Ende auch nicht lebt, kann x überschrieben werden, d. h., die Berechnung kann in T stattfinden. Es braucht dann im zweiten Schritt kein Ladebefehl erzeugt zu werden, sondern es erfolgt direkt eine Subtraktion auf T:

```
SUB   T,R
```

Damit gilt: *Con* = {R ↦ {y}, S ↦ {z}, T ↦ {v}}, *Pos* = {v ↦ T, x ↦ T, y ↦ R, z ↦ S} (*Pos* gibt für x zwar noch ein Register an, dieser Eintrag ist aber nicht mehr von Bedeutung und wird auch im Folgenden nicht mehr benutzt, da x im Rest des Blocks keine Anwendung mehr hat). Bei der Übersetzung des nächsten Befehls v := z*v wird in *getreg*(z) der dritte Fall relevant, da alle Register belegt sind und die enthaltenen Werte noch gebraucht werden. Wir wählen als zu speicherndes Register S aus, da wir dann im zweiten Schritt innerhalb von *codegen* einen Ladebefehl einsparen können. (Diese bewusste Auswahl ist in der Funktion *getreg* allerdings nicht formalisiert.) *getreg* erzeugt also den Code:

```
STORE S,Z
```

und anschließend liefert der dritte Schritt von *codegen*:

```
MULT  S,T
```

Da der Wert von v überschrieben wurde, muss im vierten Schritt das Register T jetzt freigegeben werden: *Con* = {R ↦ {y}, S ↦ {v}}, *Pos* = {v ↦ S, x ↦ T, y ↦ R, z ↦ Z}. Für den folgenden Kopierbefehl liefert *getreg*(y) das Register R. In *codegen* wird dann gar kein Befehl erzeugt, da y bereits in R ist. Der durch den Kopierbefehl bezweckte Effekt wird durch Setzen von *Con*(R) := {z, y} erreicht. Es gilt somit: *Con* = {R ↦ {z, y}, S ↦ {v}} und *Pos* = {v ↦ S, x ↦ T, y ↦ R, z ↦ R}.

Danach ist Code für die Multiplikation y := z*v zu erzeugen. *getreg*(z) liefert das (wieder) freie Register T (und nicht R, da R auch noch y enthält). *codegen* erzeugt dann, wie schon gesehen, einen Registertransferbefehl und eine Multiplikation:

```
LOAD  T,R
MULT  T,S
```

Die dritte Zeile im vierten Schritt von *codegen* entfernt y aus *Con*(R). Es gilt somit: *Con* = {R ↦ {z}, S ↦ {v}, T ↦ {y}} und *Pos* = {v ↦ S, x ↦ T, y ↦ T, z ↦ R}. Der letzte Befehl wird im Register T bearbeitet (y ist bereits dort und wird nicht mehr benötigt). Also:

```
ADD   T,R
```

Es gilt schließlich: *Con* = {R ↦ {z}, T ↦ {v}} und *Pos* = {v ↦ T, x ↦ T, y ↦ T, z ↦ R}.

Am Ende des Basisblocks müssen dann noch die lebenden Variablen gespeichert werden. Für unser Beispiel erhalten wir:

```
STORE R,Z
STORE T,V
```

Nicht benötigt wurden in diesem Beispiel die ersten beiden Zeilen des vierten Schrittes. Sie dienen dazu, Register frühzeitig freizumachen, damit *getreg* so oft wie möglich freie Register zur Verfügung stehen.

Selbsttestaufgabe 8.3: Erzeugen Sie Code für den folgenden Basisblock unter der Annahme, dass zwei Register R und S zur Verfügung stehen und dass lediglich x am Ende des Blocks zu speichern ist.

```
T1 := d+e
T2 := a+b
T3 := T2-c
T4 := T3*T1
T5 := T4-e
T6 := T2+T4
x  := T6*T5
```

□

8.4 Literaturhinweise

Eine der ausführlichsten und umfangreichsten Darstellungen der Themen Codeerzeugung und Optimierung bietet sicherlich das Standardwerk von Aho et al. (2006). Dort werden viele der hier nur angedeuteten Verfahren z. T. sehr detailliert beschrieben.

Es gibt eine Vielzahl von Verfahren zur algebraischen Optimierung in funktionalen Sprachen; das in Abschnitt 8.1.2 skizzierte Verfahren ist in (Wadler 1990) beschrieben.

Die Technik der Datenflussanalyse geht auf Vyssotsky zurück (Vyssotsky und Wegner 1963) und wurde intensiv von Allen und Cocke untersucht (Allen und Cocke 1976). Einen Überblick über die verschiedenen Datenflussanalysetechniken gibt Kennedy (1981). Die Schleifenoptimierung wird eingehend von Allen (1969) behandelt. Zwei Übersichtsartikel zu dem Thema sind (Allen und Cocke 1972) sowie (Waite 1976).

Der Algorithmus zur Codeerzeugung aus Abschnitt 8.3 stimmt im Wesentlichen mit dem aus (Aho et al. 2006) überein. Neben der in der Funktion *getreg* realisierten sehr einfachen Idee zur Registerauswahl existieren sehr anspruchsvolle Verfahren. Darunter fallen insbesondere Ansätze, die die Registerauswahl auf ein Graphfärbungsproblem zurückführen (Chaitin et al. 1981, Chaitin 1982).

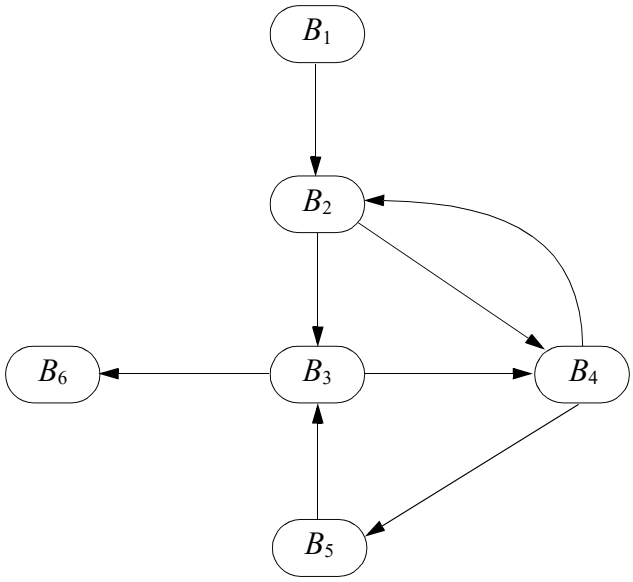
Lösungen zu den Selbsttestaufgaben

Aufgabe 8.1

Zunächst ermitteln wir die Basisblöcke:

B_1	(1) x := 5 (2) z := 3	erste Anweisung der Prozedur
B_2	(3) m := x*z (4) y := m+x (5) if m>z then goto B_4	Ziel von goto (Zeile 10)
B_3	(6) y := z-y (7) if y<m then goto B_6	nach Verzweigung
B_4	(8) z := x+z (9) i := m+x (10) if i>z then goto B_2	nach Verzweigung
B_5	(11) y := x-z (12) i := i*3 (13) goto B_3	nach Verzweigung
B_6	(14) x := z*y (15) z := x+y	Ziel von goto (Zeile 7)

Daraus ergibt sich der folgende Flussgraph:



Aufgabe 8.2

(a) Wir notieren die Mengen in einer Tabelle:

Block	*gen*	*kill*
B_1	{1, 2}	{3, 4, 5, 6, 7}
B_2	{3}	{1, 6}
B_3	{4}	{2, 5, 7}
B_4	{6, 7}	{1, 2, 3, 4}
B_5	{}	{}

(b) Zunächst initialisieren wir die Mengen $in(B_i)$ mit {} und entsprechend $out(B_i)$ mit $gen(B_i)$:

Block	*in*	*out*
B_1	{}	{1, 2}
B_2	{}	{3}
B_3	{}	{4}
B_4	{}	{6, 7}
B_5	{}	{}

Für die weitere Berechnung der Mengen $in(B_i)$ und $out(B_i)$ sind die folgenden Gleichungen iteriert anzuwenden:

$in(B_1) = out(B_2)$ $\quad out(B_1) = \{1, 2\} \cup (in(B_1) - \{3, 4, 5, 6, 7\})$
$in(B_2) = out(B_1) \cup out(B_5)$ $\quad out(B_2) = \{3\} \cup (in(B_2) - \{1, 6\})$
$in(B_3) = out(B_2)$ $\quad out(B_3) = \{4\} \cup (in(B_3) - \{2, 5, 7\})$
$in(B_4) = out(B_3) \cup out(B_5)$ $\quad out(B_4) = \{6, 7\} \cup (in(B_4) - \{1, 2, 3, 4\})$
$in(B_5) = out(B_3) \cup out(B_4)$ $\quad out(B_5) = in(B_5)$

Im ersten Durchlauf erhält man die folgenden Werte:

Block	*in*	*out*
B_1	{3}	{1, 2}
B_2	{1, 2}	{2, 3}
B_3	{2, 3}	{3, 4}
B_4	{3, 4}	{6, 7}
B_5	{3, 4, 6, 7}	{3, 4, 6, 7}

Der zweite Durchlauf ergibt:

Block	*in*	*out*
B_1	{2, 3}	{1, 2}
B_2	{1, 2, 3, 4, 6, 7}	{2, 3, 4, 7}
B_3	{2, 3, 4, 7}	{3, 4}
B_4	{3, 4, 6, 7}	{6, 7}
B_5	{3, 4, 6, 7}	{3, 4, 6, 7}

Nach der dritten Iteration lauten die Mengen:

Block	*in*	*out*
B_1	{2, 3, 4, 7}	{1, 2}
B_2	{1, 2, 3, 4, 6, 7}	{2, 3, 4, 7}
B_3	{2, 3, 4, 7}	{3, 4}
B_4	{3, 4, 6, 7}	{6, 7}
B_5	{3, 4, 6, 7}	{3, 4, 6, 7}

Da sich keine Änderungen mehr and den Mengen $out(B_i)$ ergeben, können wir die Berechnung hier beenden.

Aufgabe 8.3

Es ergeben sich unterschiedliche Maschinenprogramme in Abhängigkeit davon, welches Register zum Zwischenspeichern ausgewählt wird. Wählt man in der

fünften Zeile R für die Berechnung von T3 aus, so ergibt sich die kürzeste mögliche Folge:

```
LOAD  R,d
ADD   R,e
LOAD  S,a
ADD   S,b
STORE R,T1      (T1 und T2 haben noch weitere Anwendungen)
LOAD  R,S
SUB   R,c
MULT  R,T1      (T3 kann überschrieben werden)
STORE R,T4      (T4 hat noch eine weitere Anwendung)
SUB   R,e
ADD   S,T4      (T2 kann nun überschrieben werden)
MULT  S,R       (T6 kann überschrieben werden)
STORE S,x
```

Literatur

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

Allen, F.E (1969). Program Optimization. *Annual Review in Automatic Programming 5*, 239-307.

Allen, F.E. und Cocke, J. (1972). A Catalogue of Optimizing Transformations. In: (Rustin 1972), S. 1-30.

Allen, F.E. und Cocke, J. (1976). A Program Data Flow Analysis Procedure. *Communications of the ACM 19* (3), 137-147.

Chaitin, G.J. (1982). Register Allocation and Spilling via Graph Coloring. *ACM SIGPLAN Notices 17* (6), 201-207.

Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E. und Markstein, P.W. (1981). Register Allocation via Coloring. *Computer Languages 17*, 47-57.

Kennedy, K. (1981). A Survey of Data Flow Techniques. In: (Muchnik und Jones 1981), S. 5-54.

Muchnik, S.S. und Jones, N.D. (1981). *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ.

Rustin, R. (1972). *Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, NJ.

Vyssotsky, V. und Wegner, P. (1963). A Graph Theoretical Fortran Source Language Analyzer. Manuscript, AT & T Bell Laboratories, Murray Hill, N.J.

Wadler, P. (1990). Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science 73*. 231-284.

Waite,W.M. (1976). Optimization. *Compiler Construction: An Advanced Course*, LNCS 21, S. 549-602.

VI Literatur

Index