7.3 Ein einfacher Maschinencode-Erzeuger

Im Folgenden soll ein Maschinencode-Erzeuger vorgestellt werden, der Code für einen einfachen Block erzeugt. Es wird angenommen, dass bis auf eventuell speziell benutzte Register (für den Frame-Pointer, Stack-Pointer usw.) für jeden einfachen Block die gleiche Zahl von freien Registern verfügbar ist.

Prinzip: Man belässt berechnete Werte so lange wie möglich in Registern und speichert sie erst zurück, wenn

- das Register für andere Werte benötigt wird
- das Ende des einfachen Blocks erreicht wird.

Beispiel 7.4:
Um möglichst kurzen Code für den Befehl `a := b + c` zu erzeugen, muss man wissen, wo sich die momentanen Werte von `b` und `c` befinden (im Register, im Speicher, etc.), und man muss wissen, ob `b` oder `c` nach diesem Befehl im einfachen Block noch gebraucht werden oder ob sie am Ende des einfachen Blocks lebendig sind.
Gilt

- `b` befindet sich in `Ri`, `b` ist nicht lebendig,
- `c` befindet sich in `Rj`, `c` wird danach noch gebraucht:

dann erzeuge `ADD Ri,Ri,Rj` und notiere, dass `a` in `Ri` ist. (Länge 1, Kosten 1)
Gilt

- `c` ist im Speicher,
- `Rj` ist frei,
- sonst wie oben:

dann erzeuge
```
LD  Rj,c
ADD Ri,Ri,Rj
```
und notiere, dass `a` in `Ri` und `c` in `Rj` ist. (Länge 3, Kosten 4)
Gilt

- `b` ist im Speicher,
- `Ri` ist frei,
- sonst wie oben:

dann erzeuge `ADD Ri,b,Rj` und notiere, dass `a` in `Ri` ist. (Länge 3, Kosten 4).
usw.

Also benötigt man zunächst einmal Informationen über den **nächsten Gebrauch** einer Variablen an einer Stelle im einfachen Block. Will man sich nicht auf einfache Blöcke beschränken, muss man Informationen aus der globalen Optimierung, in diesem Fall Informationen über die Lebendigkeit von Variablen, nutzen.

Für die Maschinencode-Erzeugung interessiert bei einem Befehl

```
x := y + z
```

der nächste Gebrauch von `x`, `y` und `z` (im einfachen Block).
Zu diesem Zweck geht man zunächst an das Ende eines einfachen Blockes. Man benötigt eigentlich jetzt die Information, welche Variablen an dieser Stelle lebendig sind (d.h. ein Gebrauch in einem anderen oder, bei Schleifen möglich, auch im selben Block haben). Entweder muss man das mit Datenfluss-Analyse bestimmen, oder aber die (konservative) Annahme treffen, dass alle nicht-temporären Namen am Ende des Blocks lebendig sind. Alle diese Variablen haben einen „unbestimmten nächsten Gebrauch".
Diese Information wird für jede Variable in die Symboltabelle eingetragen. Dann geht man rückwärts im einfachen Block voran.
Trifft man auf einen Befehl

```
(i) x := y op z
```

dann sind die folgenden Schritte auszuführen:

1) füge an den Befehl auf Platz `(i)` die Information aus der Symboltabelle über den nächsten Gebrauch von `x`, `y` und `z` an. Hat `x` keinen nächsten Gebrauch, d.h. ist `x` nicht lebendig, so kann dieser Befehl ersatzlos gestrichen werden!

2) Setze in der Symboltabelle `x` auf „keinen nächsten Gebrauch".

3) Setze in der Symboltabelle den nächsten Gebrauch von `y` und `z` auf `(i)`.

Dabei ist die Reihenfolge wichtig, man betrachte z.B. den Befehl `x := x * x`.

Beispiel 7.5:
Betrachtet werden soll der Drei-Adress-Code für das Programmfragment

```
a := (-c)*b;
b := b*c + a;
```

Am Blockende seien `a` und `b` lebendig, während `c` und die temporären Variablen t_1, ..., t_4 nicht lebendig sind.
An jeden Befehl wird jetzt die Information über den nächsten Gebrauch der im Befehl auftretenden Variablen angefügt. Dabei stehe

(i) für einen nächsten Gebrauch beim Befehl `(i)`,

? für unbestimmten nächsten Gebrauch und

- für keinen nächsten Gebrauch (nicht lebendig).

An die Drei-Adress-Befehle des Programms werden die folgenden Informationen angehängt:

Befehl	nächster Gebrauch		
(1) t_1 := -c	(2)	(4)	
(2) t_2 := t_1 * b	(3)	-	(4)
(3) a := t_2	(5)	-	
(4) t_3 := b * c	(5)	-	-
(5) t_4 := t_3 + a	(6)	-	?
(6) b := t_4	?	-	

Beim Befehl auf Platz (4) steht zum Beispiel, dass t_3 nach diesem Befehl lebendig ist und einen nächsten Gebrauch im Befehl (5) hat, während `b` und `c` an dieser Stelle nicht lebendig sind.
Die Symboltabelle entwickelt sich dabei wie folgt:

Variable		nächster Gebrauch vor Befehl					
		(6)	(5)	(4)	(3)	(2)	(1)
t_1	-	-	-	-	-	(2)	-
t_2	-	-	-	-	(3)	-	-
t_3	-	-	(5)	-	-	-	-
t_4	-	(6)	-	-	-	-	-
a	?	?	(5)	(5)	-	-	-
b	?	-	-	(4)	(4)	(2)	(2)
c	-	-	-	(4)	(4)	(4)	(1)

Für die Maschinencode-Erzeugung benötigt man weiterhin zwei Tabellen, in denen festgehalten wird, welche Variablenwerte sich momentan in den Registern befinden und wo sich die Variablenwerte zur Zeit befinden:

1) Die Registertabelle:
   Zu Beginn eines einfachen Blocks soll angenommen werden, dass jedes der verfügbaren Register leer ist. Später ist in dieser Tabelle verzeichnet, welche Variablenwerte sich in einem Register befinden.

Register	Variable
`R0`	`a`,`b`
`R1`	`-`
`R2`	$\texttt{t}_1$,`d`
⋮	⋮

2) Die Adresstabelle:
   Diese Tabelle gibt an, an welchen Stellen sich der Wert einer Variablen zur Zeit befindet. Diese Tabelle könnte etwa in die Symboltabelle integriert werden.

Variable	Wert befindet sich zur Zeit in
`a`	`R0`, Speicher (Stack)
`b`	`R0`
`d`	`R3`, Speicher (statisch)
⋮	⋮

7.3.1 Maschinencode-Erzeugung für einfache Blöcke

Eingabe: Ein einfacher Block mit Informationen über den nächsten Gebrauch von Variablen

Ausgabe: Übersetzung des einfachen Blocks in Maschinencode

Methode:

- Durchlaufe den einfachen Block von Anfang bis Ende.
- Für jeden Drei-Adress-Befehl der Form `x := y op z` führe die folgenden Schritte aus:

1) Bestimme mit Hilfe der Adresstabelle, wo sich momentan der Wert von `y` befindet. Ist der Wert in einem Register `Ri`, dann setze `y':=Ri`. Befindet sich y nicht in einem Register und hat y einen nächsten Gebrauch im einfachen Block und ist ein Register `Rj` frei, so erzeuge den Befehl `LD R,y` und setze `y':=Rj`. Andernfalls setze `y':=y`. Aktualisiere die Adress- und Registertabelle.
2) wie unter 1) bestimme mit Hilfe der Adresstabelle einen Platz `z'`, wo sich momentan der Wert von `z` befindet.
3) Bestimme mit Hilfe der Funktion `getreg` ein Register `R`, in dem das Ergebnis der Berechnung `y op z` abgelegt werden soll. Erzeuge den Befehl `OP R,y',z'`. Vermerke in der Registertabelle, dass in `R` der Wert von `x` steht.
4) Hat `y` und/oder `z` keinen nächsten Gebrauch im einfachen Block und ist nicht lebendig am Ende des Blocks und befindet sich in einem Register, dann lösche `y` und/oder `z` aus dem entsprechenden Eintrag der Registertabelle und setze die Adresstabelle entsprechend um.

- Für jeden Drei-Adress-Befehl der Form `x := op y` führe die folgenden Schritte aus:
  1) Bestimme mit Hilfe der Adresstabelle, wo sich momentan der Wert von `y` befindet. Ist der Wert in einem Register `Ri`, dann setze `y':=Ri`. Befindet sich y nicht in einem Register und hat y einen nächsten Gebrauch im einfachen Block und ist ein Register `Rj` frei, so erzeuge den Befehl `LD Rj,y` und setze `y':=Rj`. Andernfalls setze `y':=y`. Aktualisiere die Adress- und Registertabelle.
  2) Bestimme mit Hilfe eine der Funktion `getreg` ein Register `R`, in dem das Ergebnis der Berechnung `op y` abgelegt werden soll. Erzeuge den Befehl `OP R,y'`. Vermerke in der Registertabelle, daß in `R` der Wert von `x` steht.
  3) Hat `y` keinen nächsten Gebrauch im einfachen Block, ist nicht lebendig am Ende des Blocks und befindet sich in einem Register, dann lösche `y` aus dem entsprechenden Eintrag der Registertabelle und setze die Adresstabelle entsprechend um.
- Für jeden Drei-Adress-Befehl der Form `x := y` führe die folgenden Schritte aus:
  1) Ist der Wert von `y` im Register, ändere die Eintragungen in der Register- und der Adresstabelle so, dass der Wert von `x` nur im Register auftritt, das `y` enthält. Hat `y` keinen nächsten Gebrauch und ist nicht lebendig am Ende des Blocks, lösche `y` aus der Registertabelle.
  2) Ist der Wert von `y` im Speicher, dann bestimme mit `getreg` ein Register `R` und erzeuge den Befehl `LD R,y`. Danach müssen die beiden Tabellen wieder aktualisiert werden.
- Am Ende des einfachen Blocks müssen für die Werte alle Variablen, die an dieser Stelle lebendig sind und sich noch nicht im Speicher befinden, `ST`-Befehle erzeugt werden, um die entsprechenden Werte zurück zu speichern.

7.3.2 Mögliche Implementation der Funktion `getreg`

Eingabe: Ein Drei-Adress-Befehl `x := y op z` oder `x := op y` oder `x := y` mit angehefteten Informationen über den nächsten Gebrauch der Variablen.

Ausgabe: Ein Register `R`, in dem das Ergebnis der Operation abgelegt wird.

Methode:

1) Ist `y` oder `z` in einem Register, das nicht auch den Wert anderer Variablen enthält und hat `y` oder `z` keinen nächsten Gebrauch nach diesem Befehl, dann wähle dieses Register für `R`.
2) Trifft 1) nicht zu, wähle ein freies Register für `R`.
3) Trifft 2) nicht zu, hat `x` aber einen nächsten Gebrauch in diesem Block, dann suche ein besetztes Register `R`.
   (Hier gibt es viele Möglichkeiten, mit Hilfe der Informationen über den nächsten Gebrauch von Variablen, der Registertabelle und der Speichertabelle ein Register zu suchen, das mit minimalem Aufwand „geleert“ werden kann.)
   Erzeuge für jede Variable, deren Wert sich momentan in `R` und nicht im Speicher befindet, einen `ST`-Befehl, der diesen Wert in den Speicher befördert.

Diese Funktion könnte noch signifikant verbessert werden, wenn man z.B. die mögliche Kommutativität der Operation `op` bei der Wahl eines Registers ausnutzt .

Beispiel 7.6:
Die Übersetzung von

```
a := (-c) * b
b := b * c + a
```

ergibt folgenden Drei-Adress Code (siehe Beispiel 7.5)

```
t1 := -c
t2 := t1 * b
a  := t2
t3 := b * c
t4 := t3 + a
b  := t4
```

Dabei seien $t_1,\ldots,t_4$ temporäre Dateien, `a` und `b` am Blockende lebendig und `c` nicht lebendig. Damit gelten die Informationen über den nächsten Gebrauch aus Beispiel 7.5.
Weiter sei angenommen, dass drei Register `R0`, `R1` und `R2` frei sind. Zu Beginn befinden sich `a`, `b` und `c` im Speicher, $t_1,\ldots,t_4$ befinden sich *nicht* im Speicher.
Unter diesen Voraussetzungen ergibt sich folgende Übersetzung in Maschinencode:

Befehl	Maschinencode	Registertabelle	Adresstabelle
-	-	Alle Register sind frei	s.o.
t_1 := -c	LD R0,c NEG R1,R0	R0 : c R1 : t_1	c in R0 und im Speicher t_1 in R1
t_2 := t_1 * b	LD R2,b MUL R1,R1,R2	R2 : b R1 : t_2	b in R2 und im Speicher t_2 in R1 t_1 –
a := t_2		R1 : a	a in R1, nicht im Speicher t_2 –
t_3 := b * c	MUL R2,R2,R0	R2 : t_3	t_3 in R2 b im Speicher
t_4 := t_3 + a	ADD R2,R2,R1	R2 : t_4	t_4 in R2 t_3 –
b := t_4		R2 : b	b in R2, nicht im Speicher
	ST a,R1 ST b,R2		a in R1 und im Speicher b in R2 und im Speicher

Damit erhält man ein Programm mit Kosten 16.

Unser Algorithmus zur Maschinencode-Erzeugung ist natürlich noch unvollständig, da nicht alle Typen von Drei-Adress-Befehlen behandelt werden. Um einen Eindruck von der zunehmenden Komplexität zu geben, soll nun beispielhaft die Behandlung eines Drei-Adress-Befehls der Form `a := b[c]` gezeigt werden.
Als Ziel sei ein Register `Rj` gewählt worden. Wieder gibt es eine große Fallunterscheidung:

- Ist `c` in `Ri`, `b` eine Konstante, dann erzeuge

  ```
  LD Rj,b(Ri)
  ```
  mit Kosten 3,

- ist `c` im Speicher, dann erzeuge

  ```
  LD Rj,c
  LD Rj,b(Rj)
  ```
  mit Kosten 6,

- ist `c` im Speicher,
  `c` wird im selben Block noch gebraucht und es sind noch
  ein weiteres Register `R1` frei, dann erzeuge

  ```
  LD R1,c
  LD Rj,b(Rj)
  ```
  mit Kosten 6,

- ist `c` im Aktivierungs-Record (AR), FP der Frame-Pointer, dann erzeuge

  ```
  LD Rj,c(FP)
  LD Rj,b(Rj)
  ```
  mit Kosten 6,

- `c` im AR und ist auch das Array `b` im AR und
  ist $\#b_{offset}$ der Abstand des Arrays `b` vom FP, dann erzeuge

  ```
  ADD Rj,FP,#boffset
  ADD Rj,c(FP),Rj
  LD  Rj,*Rj
  ```
  mit Kosten 7

- usw.

Fazit: Dieser Ansatz der Maschinencode-Erzeugung funktioniert zwar, wird aber bei der großen Zahl von Maschinenbefehlen und Adressierungsarten moderner Rechner zu einem Programmteil führen, das äußerst komplex und undurchsichtig ist! Die relevante Information ist gegen alle Regeln des Software Engineering zerstreut und teilweise in der Programmlogik eingearbeitet. Das führt zu einem fehleranfälligen, schwer zu modifizierenden und schwierig zu übertragenden Programmcode!