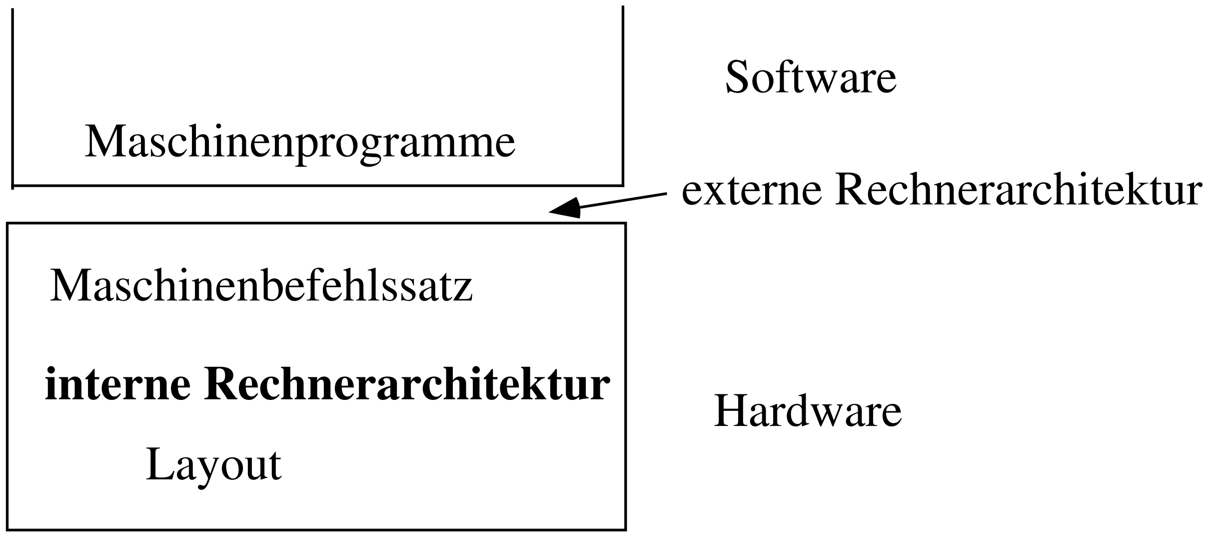
Kapitel 4

Übersetzer (Compiler): Kodegenerierung

4.1 Rechnerarchitekturen



- die externe Architektur legt durch den Maschinenbefehlssatz fest, wie ein Rechner von der Seite der Software her angesprochen werden kann. Dies ist die für den Kodegenerator massgebliche Schnittstelle.
- die interne Architektur bezieht sich auf den internen Aufbau und das Organisationsprinzip eines Rechners.

4.1.1 Klassen von Maschinenbefehlssätzen

Eine Klassifikation von Maschinenbefehlssätzen ist durch die Beantwortung der folgenden Fragen möglich:

- welche Operanden können - neben Hauptspeicheradressen angesprochen werden, insbesondere. Register in der CPU

- welche Anzahl von Operanden kann typischerweise in einem Befehl angesprochen werden?
- welche Operationen sind verfügbar?
- wie sind die Operanden einer Operation zwischen CPU und Hauptspeicher verteilt, müssen z.B. alle Operanden CPU-Register sein?
- wie ist der Typ und Größe der Operanden und wie wird er spezifiziert?

Typische externe Architekturen:

Stackmaschinen: Maschinenbefehle beziehen sich implizit auf einen Stack. Das Ergebnis von Operationen wird im Stack abgelegt. Dieser wird z.T. durch Register in der CPU nachgebildet.

Akkumulatormaschinen: impliziter Operand ist ein Akkumulator-Register. I.a. ein weiterer expliziter Operand. Ziel ist ebenfalls der Akkumulator.

Registersatzmaschinen: typisch sind zwei oder drei Operanden (Zwei- bzw. Dreiadressbefehle), die Register oder Hauptspeicheradressen sein können.

Beispiel 1. $a := b + c$

Stack	Akkumulator	Registersatz
PUSH b	LOAD c	LOAD R1,c
PUSH c	ADD b	ADD R1,b
ADD	STORE a	STORE a,R1

Abarbeitung:

1. **Stackmaschine**

Befehl	Stack
...	...,
`PUSH b`	..., b
`PUSH c`	..., b, c
`ADD`	..., b+c

2. **Akkumulatormaschine:**

Befehl	Zustand
...	Akku = ?
`LOAD c`	Akku = c
`ADD b`	Akku = c+b
`STORE a`	Akku = c+b, (a wird überschrieben

3. **Registersatzmaschine**

Befehl	Zustand
...	R1 = ?
`LOAD R1,c`	R1 = c, R2 = ?,
`ADD R1,b`	R1 = c + b R2 = ?, ...
`STORE a,R1`	R1 = c + b R2 = ?, ...
	a wird mit dem Inhalt von R1 überschrieben

4.2 Kode-Erzeugung

Die Kodeerzeugung geschieht im allgemeinen in zwei Schritten: Man wählt eine Architektur aus, die bereits rechnernah ist, und modelliert danach den Zwischencode. Dies entspricht dem Entwurf einer **abstrakten Maschine**: Dazu gehört:

Modellierung des globalen Speichers : Dies ist i.a. eine Kombination von Hauptspeicher, Stack, Register, Akkumulator, usw. Auf jeden Fall mit dabei ist ein Programm und ein Programmzeiger.

abstrakte Maschinensprache

operationale Semantik D.h ein Interpreter für die abstrakte Maschinenbefehle ist spezifiziert.

4.2.1 Zwischencode-Erzeugung für eine Stackmaschine: (Kellermaschine)

Datenstrukturen, die den Zustand der Maschine ausmachen:

- Stack
- Programm,
- Programmzeiger

Der Keller kann nur von oben her angesprochen werden. D.h. er wird von oben her adressiert.

Für einen theoretischen Stack genügt es, nur die Operationen: `push, pop`, und die Frage `leerer-stack?` zuzulassen. Diese Datenstruktur ist sehr ähnlich zu einer Liste. Allerdings werden später auch andere Operationen dazu kommen. Z.B. Lesen des i-ten (von oben) Kellereintrags, Löschen mehrerer Kellereinträge.

Um mit einer einfachen Kellermaschine operieren zu können, nehmen wir zunächst an, daß auf dem Keller nur ganze Zahlen stehen dürfen:

Operationen, die im Kellerprogramm möglich sind:

pushk k k ist Zahlenkonstante.

pop

push i ; i ist konstante Stackadresse; 0 = oberstes, 1 = zweites, usw.

arithmetische Operationen: $+, -, *, /$. Jeweils die obersten beiden Elemente werden verknüpft. D.h. aus $\ldots, a, b$ wird $\ldots, a - b$.

print Gebe das Ergebnis aus: drucke das oberste Element des Kellers

Hier ist es wichtig auf die Beschränkungen zu achten: Z.B. , daß i, k konstant sind in den Befehlen [pushk k] [push i]. Es ist unproblematisch, eine entsprechende abstrakte Maschine für variable Werte zu entwerfen. Allerdings entfernt sich diese dann von einer echten Rechnerarchitektur, da dies Referenzen im Programmcode erfordert.

Beispiel 2. Damit kann man bereits ein Programm zur Berechnung von arithmetischen Ausdrücken schreiben: Berechne $1 + 3 - 5 * 6$

```
pushk 1
pushk 3
+
pushk 5
pushk 6
*
-
print
```

Die operationale Semantik dieser Kellerbefehle sieht formalisiert so aus: Es gibt einen Interpreter I mit folgenden Argumenten:

1. das aktuelle Programm als Liste, der nächste Befehl ist das erste Element der Liste. 2)
2. Den Keller als Liste.

Die operationale Semantik der jetzigen (rudimentären) Kellermaschine sieht dann so aus:

```
I  (pushk k : rest) stack    --> I rest   (k:stack)

I (+ : rest) (a:b: stackr)   --> I  rest  (b+a :  stackr)

I (- : rest) (a:b: stackr)   --> I  rest  (b-a :  stackr)

I (print:rest) (a:b: stackr) --> a
```

Es ist mit einer Argument- und Resultatsübergabekonvention auch möglich, Funktionsaufrufe durchzuführen: Man kann z.B. vereinbaren, daß die Argumente einer Funktion bei deren Aufruf sich auf dem Stack befinden, wobei das letzte Argument oben ist. Das Resultat kann z.B. nach Beendigung der Auswertung der Funktion oben auf dem Stack stehen, wobei die Argumente konsumiert wurden.

Beispiel 3. Berechne $x^2 + y^2$, wenn der Stack am Anfang $\ldots, x, y$ ist:

```
push 0
*
push 1
push 0
*
+
print
```

Diese Funktion hält nicht die Resultatsübergabekonvention ein, da auf dem Stack das Argument noch vorhanden ist.

Um auch Fallunterscheidungen zu programmieren, erlauben wir noch zusätzlich in der Kellermaschine:

slide m n Schiebe die obersten m Elemente um n nach unten. D.h. n Elemente unter den m oberen verschwinden.

Marke. symbolische Sprungmarke

branchz marke Wenn auf dem Stack eine Null steht, springe zu marke. (Lösche das oberste Element)

jump marke Springe zu marke

Mit den Marken wurde jetzt eine symbolische Referenz wieder in den abstrakten Maschinenkode eingefügt. Man kann jetzt verschiedene Sichtweisen vertreten:
a) Die Symbole stehen für Adressen und werden nur für einen menschlichen Betrachter angezeigt. b) Die Symbole sind tatsächlich vorhanden. In diesem Fall müssen in einem anschließenden Kodeerzeugungsschritt diese symbolischen Referenzen wieder in eine explizite Adressierung umgewandelt werden. Dies geschieht einfach durch Angabe der Indizes in der Liste des Programms. Hier muß man darauf achten, daßman diese symbolischen Marken komplett aus einem Programm entfernen kann, und die Indizes als Ziele der Sprünge dem Programm nach Löschung angepaßt sein müssen.

Beispiel 4. Berechne $x^2 + y^2$, wenn der Stack am Anfang $\ldots, x, y$ ist. Diesmal mit richtiger Übergabekonvention des Resultats.

```
push 0
*
push 1
push 0
*
+
slide 1 1
print
```

Beispiel 5. Mit den obigen Kellermaschinenbefehlen kann man die Fakultätsfunktion definieren. Am Anfang steht das Argument auf dem Stack. Danach wird in der Schleife das Ergebnis oben auf dem Stack abgelegt, der Zähler ist tiefer.

```
anfang.
push 0
loop.
branchz ende.
push 1
```

```
pushk 1
-
push 1
push 1
*
slide 2 2
jump loop
ende.
slide 1 1
print
```

Das ergibt folgenden Programmablauf, wenn der Stack am Anfang die 3 enthält:

Stack	Befehl
3	push 0
3,3	branchz ende.
3,3	push 1
3,3,3	pushk 1
3,3,3,1	-
3,3,2	push 1
3,3,2,3	push 1
3,3,2,3,2	*
3,3,2,6	slide 2 2
2,6	jump loop.
2,6	branchz ende.
2,6	push 1
2,6,2	pushk 1
2,6,2,1	-
2,6,1	push 1
2,6,1,6	push 1
2,6,1,6,1	*
2,6,1,6	slide 2 2
1,6	
...	...

Beispiel 6. Wie kann man Kode für eine Kellermaschine erzeugen? Wir betrachten den Syntaxbaum für den Ausdruck $1 + 3 - 5 * 6$.

Der Syntaxbaum dazu in Präfixschreibweise ist:

minus (plus 1 3) (mal 5 6)

Daraus kann man mittels eines rekursiven Programms das Kellerprogramm erzeugen:

- minus s t $\rightarrow$ Code für s; Code für t; -
- mal s t $\rightarrow$ Code für s; Code für t; *
- plus s t $\rightarrow$ Code für s; Code für t; +
- Konstante $\rightarrow$ `pushk` Konstante

Dies ergibt: Code für (plus 1 3): Code für (mal 5 6); -
Das Kellermaschinenprogramm ist dann:
`Pushk` 1; `Pushk` 3; +; `Pushk` 5; `Pushk` 6; *; -

4.2.2 Speicheradressierung

Wir erweitern jetzt die Möglichkeiten der Kellermaschine, indem wir zusätzlich einen globalen Speicher (heap, Halde) erlauben, der Zahlen aufnehmen kann, pro Speicherplatz eine Zahl. Diese Zahlen sind adressierbar mittels einer Adresse (Index) die ebenfalls eine Zahl ist. Der Einfachheit halber erlauben wir, daß Zahlen und Indizes nicht unterscheidbar sind auf dem Stack.

Damit besteht der innere Zustand der Stackmaschine jetzt aus:

- Stack
- Programm,
- Programmzeiger
- globaler Speicher

Es gibt allerdings die Alternativen, diese Adressen erkennbar zu machen, entweder durch einen eigenen Stack oder durch eine extra Markierung als Adresse. Diese Markierung hat durchaus praktischen Nutzen. Hierdurch kann ein garbage collector bei jedem Zustand der Kellermaschine noch die referenzierten von unreferenzierten Adressen unterscheiden.

Folgende Befehle bewirken eine direkte Adressierung:

get a Der Wert SP[a] wird auf den oberste Platz des Stacks abgelegt.

put a Der Wert (Stack 0) wird an die Adresse a geschrieben.

Damit können wir jetzt Programme mit Zuweisungen realisieren:

Beispiel 7. Das Programm $x := x + 1$ kann jetzt folgendermaßen erzeugt werden. Allerdings muß vorher von anderer Stelle (Symboltabelle) eine Adresse für x zugewiesen worden sein.

```
get 1
pushk 1
+
put 1
```

Weitergehend will man auch mit Objekten im Hauptspeicher wie mit Paaren umgehen, die aus zwei Adressen bestehen und selbst wieder adressierbar sind.

Die Befehle iget und iput bewirken eine indirekte Adressierung:

iget Die Zahl SP[a] wird auf den obersten Platz des Stacks abgelegt. a stand vorher auf dem Stack.

iput Die Zahl (Stack 1) wird an die Adresse (Stack 0) geschrieben. Die Zahl wird auf dem Stack gelöscht .

genpair Dies Operation nimmt an, daß auf dem Stack oben zwei Adressen stehen. Es wird ein Objekt erzeugt, das diese beiden Adressen enthält, und an einer neuen Adresse steht. Diese wird auf dem Stack abgelegt.

readpair Dies Operation nimmt die oberste Element des Stacks als Adresse, liest dort im Speicher das Paar und legt die beiden Adressen des Paars auf dem Stack ab.

getmem Legt einen Speicherplatz für eine Zahl an und legt die Adresse auf dem Stack ab.

Damit kann man z.B. Listen verwalten:

Beispiel 8. Die Liste $[1, 2]$ kann man jetzt erzeugen durch:

```
pushk 1
getmem
iput
pushk 2
getmem
iput
genpair
```

Bemerkungen Unsere Kellermaschine ist noch zu abstrakt. Folgende Dinge fehlen noch:

- Die Zahlen müssen von ihrem Speicherbedarf her festgelegt werden (4 byte, 8 byte, ??)
- Die Länge der Adressen muss festgelegt werden.
- Das Programm muß auf einen String abgebildet werden: jeder Befehl auf ein Halb-byte (Byte, o.ä).
- Die symbolischen Marken müssen in Adressen bzgl. des Programms umgewandelt werden.

Hat man diese Längen festgelegt, dann kann man das Programm mittels weiterer Schritte in einen Bitstring, d.h. in ein (Bytecode)-Maschinenprogramm umwandeln:

Jeder Stackmaschinenbefehl wird als ein String kodiert, der am Anfang einen Kode für den Namen des Befehls hat (z.B. push = X'01', pop = X'02', ...). Danach folgt das Argument, bzw. die Argumente, falls der Befehl welche benötigt.

Wenn die Länge der Konstanten + Adressen festgelegt ist und die Kodierung der Befehle als (Halbbyte oder Byte) bekannt ist, kann man schon Teilstrings des endgültigen Programms berechnen. Die Adressen der Sprungbefehle benötigt man als Abstand in Anzahl der Bytes vom Programmanfang. Die Berechnung dieser Adressen muß auf der Grundlage des endgültigen Programms erfolgen. Dies kann man z.B. durch zweifaches Durchgehen des Programms erreichen:

1. die endgültige Form und Länge des Programms wird berechnet, aber die internen Sprungadressen fehlen noch bzw. sind Zeiger in eine Tabelle. Man merkt sich in eine Sprungtabelle die endgültigen Adressen.

2. In diesem Durchgang werden die endgültigen Adressen eingesetzt.

Die gemischte Benutzung von Befehlen mit keinem, einem , zwei oder sogar mehr Argumenten führt zwar zu kurzem Maschinenkode, kann aber Nachteile bei RISC-Architekturen haben.

Die Implementierung beliebiger langer Zahlen wie integer in Haskell erfordert eine eigene Implementierung aller arithmetischen Operationen als Programm für die Kellermaschine.

4.2.3 Code-Optimierung durch den Compiler

- Zwischenrepräsentation

  Optimierung auf hoher Ebene:
  - Prozedurersetzung (procedure inlining)
  - Schleifentransformation,
  - partielle Auswertung

- Nach Kodeerzeugung
  - Gucklochoptimierung
  - Elimination redundanter Sprünge
  - Code-umstellungen

Gucklochoptimierungen.

Dies sind Optimierungen, die sich lokal bestimmte Sequenzen des erzeugten Codes anschauen und dies bei gleichem Effekt verkürzen. Z.B. kann man die Folge push i; pop ersatzlos streichen.

Es ist i.a. nicht möglich, optimalen Code zu erzeugen. Da man schematisch vorgehen muß, gibt es immer Folgen, die lokal unsinning aussehen und redundant sind.

Beispiel 9. Für das Polynom $f(x) = x$, oder auch die Identitätsfunktion, wird i.a. folgende Sequenz erzeugt:

```
push 0; slide 1 1
```

Diese Sequenz ist redundant und kann eliminiert werden.

Diese Optimierungen haben oft einen Satz von Mustern, nach dem sie den erzeugten Code durchforsten, und durch etwas besseres ersetzen. Im allgemeinen kann man durch Gucklochoptimierungen zwar leichte Verbesserungen erzielen, diese haben trotzdem nur den Status von nachträglichen Reparaturen. Zu diesem Zeitpunkt kann die Effizienz nur mit hohem Aufwand verbessert werden.

Prozedurersetzung

aus `(sqrt (quadratsumme x y))`
wird durch Ersetzung: (sqrt (+ (* x x) (* y y)))

Dies erspart einen Prozeduraufruf, allerdings kann durch mehrfache Einsetzung der Kode des Gesamtprogramms vergrößert werden. In Haskell bleibt bei Prozedureinsetzung die Semantik erhalten.

Allerdings: Prozedurersetzung bzw. textuelle Ersetzung mittels einfacher Makros erhält nicht in jedem Fall die Semantik. Es gibt zwei Fälle:

1) Bei Funktionsaufrufen (Z.B. in LISP) kann sich die Auswertungsreihenfolge verändern, und auch das Terminierungsverhalten des Programms.

Beispiel 10. `f (x,y) = if y == 0 then x else 0`

```
 f (x := x+1, x:= 2*x)

  wird zu

if (x := 2*x) == 0 then (x := x+1) else 0
```

2) Zuweisungen und andere Effekte könnten verdoppelt werden. Dies passiert, wenn ein formaler Parameter mehr als einmal im Rumpf der Prozedur vorkommt. Dies kann auch passieren, wenn die Prozedur selbst keine Zuweisungen oder Seiteneffekte enthält, aber Zuweisungen (Seiteneffekt-ausdrücke) als Argumente.

4.2.4 partielle Auswertung

Das sind Berechnungen, die man schon zur Compilezeit machen kann. Im Prinzip ist dies auf jede Auswertung oder Berechnung anwendbar, wenn diese nicht von irgendeiner Eingabe abhängt.

statt	$(+\ (*\ 10\ 10)(*\ y\ y))$
kompiliert man:	$(+\ 100\ (*\ y\ y))$

Eine gute partielle Auswertung verkompliziert den Compiler erheblich. Eine Voraussetzung für partielle Auswertung ist eine festgelegte eindeutige Semantik für das Programm, die nur vom Programmtext abhängt, nicht vom Rechner. Die einfachste Variante ist, daß zur Kompilierzeit ein Interpreter vorhanden ist (d.h. die operationale Semantik ist spezifiziert), der allerdings keine Eingaben anfordern darf. Dieser Interpreter wird dann vom Compiler verwendet, um bestimmte Unterausdrücke auszuwerten. Es besteht hierbei immer die Gefahr, daß dieser Aufruf nicht terminiert, deshalb darf dieser Aufruf i.a. nur eine vorher festgelegte Anzahl von Berechnungschritten ausführen.

Z.B. kann man in einem funktionalen Programm im Prinzip jeden geschlossenen Ausdruck zur Compilezeit auswerten. Eine Verallgemeinerung ist die `Instantiierung`. Damit sind unerwartete Optimierungen möglich:

Beispiel 11. Sei die Definition von map

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

Dann kann man `map quadrat x` durch Instantiieren optimieren, indem man den Ausdruck `(map quadrat)` in etwas allgemeinerer Form partiell auswertet und eine neue instantiierte Funktionsdefinition erzeugt.

```
mapq  [] = []
mapq  (x:xs) = (quadrat x) : mapq xs
```

Analog kann man in einem prozeduralen Programm Z.B. ein Java-Programmfragment (hier eine Schleife) der Form:

```
for (int i =1; i > 20; i++) x += 1;

ersetzen durch:

x += 20
```

4.3 Kode-Erzeugung für Registersatzmaschinen

Die meisten Maschinen sind heutzutage Registersatzmaschinen, die einen Satz von 16-32 Universalregistern besitzen. Üblich sind **Dreiadreß-Befehle**

OP DEST, SRC1, SRC2 d.h. DEST:= SRC1 OP SRC2

oder **Zweiadreßbefehle**, bei denen ein Quelloperand mit dem Zieloperanden übereinstimmt:

OP DEST, SRC2 d.h. DEST:= DEST OP SRC2

ADD R5, R6 bedeutet z.B. R5:= R5+R6.

Man unterscheidet zwei Klassen von Registersatzmaschinen:

- die **CISC** (complex instruction set computer)-Architekturen besitzen sehr umfangreiche Befehlssätze mit komplizierten Adressierungsmodi. Die klassischen Registersatzmaschinen wie IBM 360, VAX, Intel 8086 sind solche Architekturen.

  Ziel und Quellen können entweder Universalregister oder Speicheradressen sein, z.B. ADD a, R6 bedeutet sp[a]:= sp[a]+R6.

- die RISC (reduced instruction set computer)-Architekturen versuchen, Befehle möglichst einfach, einheitlich in der Länge (sowohl zeitlich als auch als Codierung im Speicher), usw. zu halten. Die Verarbeitung solcher Befehle kann leichter durch Pipelining (Fließbandprinzip) beschleunigt werden. RISC-Architekturen werden besser als **Load/Store**-Architekturen bezeichnet, da sich die Einfachheit des Befehlssatzes darin ausdrückt, daß Speicheradressen Ziel und Quelle nur von Lade- und Speicherbefehlen sein können. Die Operationen wie Addition können ausschließlich mit Registerinhalten ausgeführt werden. Beispielsweise muß die o.a. Operation $sp[a] := sp[a] + R6$ durch drei Befehle realisiert werden:

  LOAD R0,a ADD R0,R6 STORE R0,a

  Moderne Workstations sind fast immer mit RISC-Prozessoren ausgestattet.

4.3.1 Codegenerierung für Registersatzmaschinen

Wir nehmen an: Es stehen Dreiadreßbefehle zur Verfügung, mit denen Operationen der Art

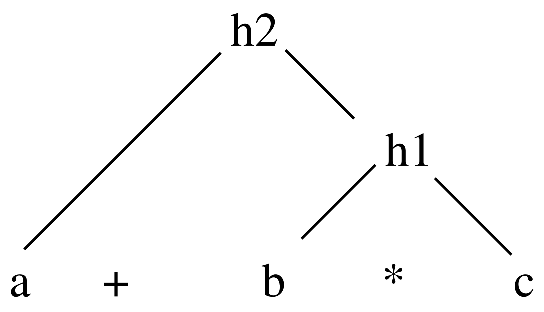
DEST:= SRC1 op SRC2

ausgeführt werden können. Bei einer CISC-Architektur sind DEST, SRC1 und SRC2 entweder Speicheradressen oder Register.

1. Schritt: Einführung von Hilfsvariablen h_i für jeden Knoten im Syntaxbaum. Diese Hilfsvariablen entsprechen den Registern. Dies nennt man auch Überführung in SSA-Form (static single assignment: statische einmalige Zuweisung). Ein Ausdruck wie

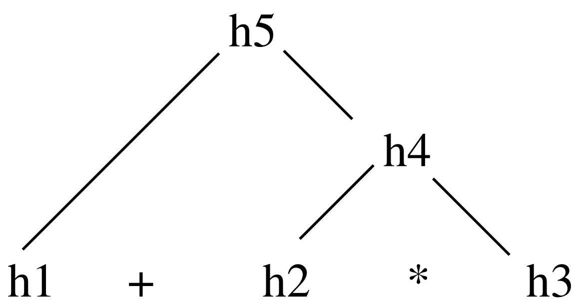
$$a + b * c$$

wird dann wie folgt verarbeitet, wobei für jeden Knoten eine eigene Hilfsvariable benutzt wird:



$$h_1 := b * c; h_2 := h_1 + a;$$

Bei einer Load/Store-(RISC)-Architektur können nur Register miteinander verknüpft werden; Variable müssen vorher in Register geladen werden. Der Ausdruck a + b*c wird daher wie folgt übersetzt:



$$\begin{aligned} h_1 &:= a; \\ h_2 &:= b; \\ h_3 &:= c; \\ h_4 &:= h_2 * h_3; \\ h_5 &:= h_4 + h_1; \end{aligned}$$

2ter Schritt Lebenszeitanalyse der Hilfsvariablen

Grundblock: linear verarbeitete Sequenz von Anweisungen. Die Lebenszeit einer Hilfsvariablen wird relativ zu Anweisungen angegeben. Eine Variable x heißt tot (T) für eine Sequenz von Anweisungen A, wenn diese zwischen Benutzung und Zuweisung , d.h. genauer zwischen 1) und 2) liegt:

1.) der Anweisung mit dem letzmaligem Vorkommen von x in einem Ausdruck auf der rechten Seite einer Zuweisung

2.) der nächsten Anweisung einschließlich, die x einen neuen Wert zuweist. Sonst heißt eine Variable lebendig (L).

Zur Lebenszeitanalyse wird eine Sequenz von Anweisungen rückwärts von der letzten zur ersten Anweisung durchlaufen. Beim Analysieren einer Anweisung

$$h_1 := h_2 \ op\ h_3$$

wird

- die Variable h_1 als tot und
- die Variablen h_2 und h_3 als lebendig markiert.

Beispiel 12. Für eine Load/Store-(RISC)-Architektur wird die Anweisung

$$y := x - z + w;$$

wie folgt in eine Sequenz mit Hilfsvariablen und angegebenen Lebenszeiten übersetzt:

Zuweisung			h_1	h_2	h_3	h_4	h_5
h_1	$:=$	x	T	T	T	T	T
h_2	$:=$	z	L	T	T	T	T
h_3	$:=$	$h_1 - h_2$	L	L	T	T	T
h_4	$:=$	w	T	T	L	T	T
h_5	$:=$	$h_3 + h_4$	T	T	L	L	T
y	$:=$	h_5	T	T	T	T	L

3. Schritt: Zuordnung von Registern zu Hilfsvariablen

Zwei Hilfsvariable können gemeinsam in einem Register gehalten werden, wenn sich ihre Lebenszeiten nicht überschneiden. Sie heißen dann **verträglich.** Im Beispiel oben sind h_1, h_3 und h_5 miteinander sowie h_2 und h_4 verträglich.

Im Spezialfall der Sequenz ist es möglich, in polynomialer Zeit eine minimale Anzahl von Registern zu finden, indem man die Tabelle von oben nach unten bearbeitet

Zuweisung			h_1	h_2	h_3	h_4	h_5	R_1	R_2
h_1	$:=$	x	T	T	T	T	T		
h_2	$:=$	z	L	T	T	T	T	h_1	
h_3	$:=$	$h_1 - h_2$	L	L	T	T	T	h_1	h_2
h_4	$:=$	w	T	T	L	T	T	h_3	
h_5	$:=$	$h_3 + h_4$	T	T	L	L	T	h_3	h_4
y	$:=$	h_5	T	T	T	T	L	h_5	

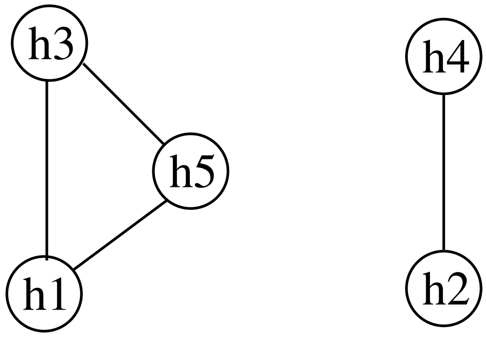
Werden die zugeordneten Register für die Hilfsvariablen eingesetzt, erhält man:

$$\begin{array}{lcl} R_1 & := & x; \\ R_2 & := & z; \\ R_1 & := & R_1 - R_2; \\ R_2 & := & w; \\ R_1 & := & R_1 + R_2; \\ y & := & R_1; \end{array}$$

Im allgemeineren Fall, z.B. wenn Verzweigungen vorkommen, und das Programmfragment ein gerichteter Graph mit einem Anfang und einem Ende ist (d.h. ohne Rücksprünge), wird ein **Kompatibilitätsgraph** aufgestellt.

In unserem Fall werden die Hilfsvariablen als Knoten dargestellt; eine Kante $h_i - h_j$ bedeutet, daß zwei Hilfsvariablen miteinander verträglich sind. Die neu eingeführten Hilfsvariablen in verschiedenen Verzweigungen sind unabhängig. Das Problem, eine Zuordnung von Hilfsvariablen zu Registern so zu finden, daß eine minimale Anzahl von Registern benötigt wird, läßt sich als Cliquenpartitionierungsproblem verstehen. Eine Clique ist ein Graph, bei dem es eine Kante von jedem Knoten zu jedem anderen gibt (ein vollständiger Graph). (Eine Clique entspricht dabei einem Register.) Beim Cliquenpartitionierungsproblem versucht man, die Menge der Knoten eines Graphen so zu partitionieren, d.h. in disjunkte Mengen zu unterteilen, daß jede Teilmenge eine Clique bildet und die Anzahl der Cliquen minimal wird.

In dem o.a. Beispiel erhält man folgenden einfachen Kompatibilitätsgraphen:

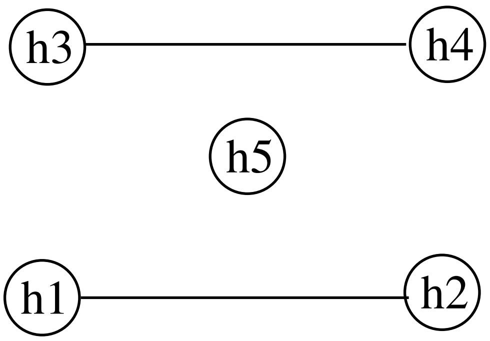


Die beiden Cliquen werden durch die Knotenmengen $\{h_1, h_3, h_5\}$ und $\{h_2, h_4\}$ gebildet.

Die Lösung des Cliquenpartitionierungsproblem verlangt i.a. exponentiellen Aufwand; es wurden jedoch eine Reihe guter Heuristiken entwickelt, mit denen man zu nicht-optimalen, aber befriedigenden Ergebnissen in akzeptabler Laufzeit kommt.

Ein äquivalente Methode ist das Problem als Färbung eines Graphen mit möglichst wenig Farben zu betrachten:

Der Interferenz-Graph hat als Knoten die Hilfsvariablen. Zwei Hilfsvariablen h_i und h_j sind verbunden, wenn sich ihre Lebenszeiten überschneiden (sie nicht kompatibel sind)



Man sieht: der Graph läßt sich mit zwei Farben einfärben, wobei h_3 und h_4 unterschiedliche Farben bekommen, ebenso h_1 und h_2.

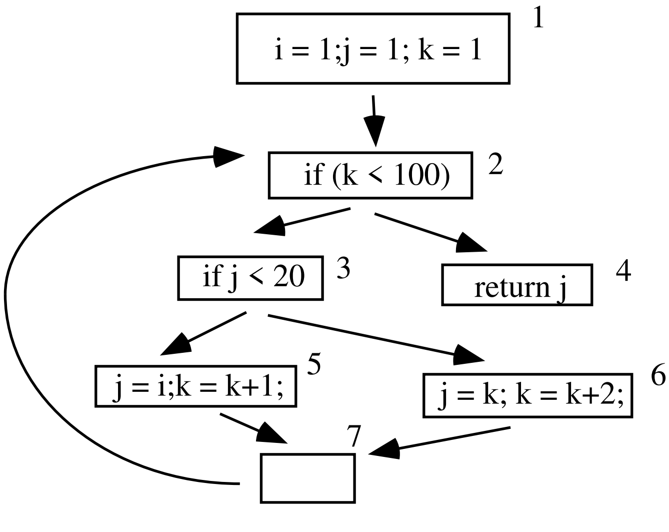
Dieses Verfahren ist günstiger, wenn die Anzahl der inkompatibel-Kanten klein ist. Auch dieses Problem ist $\mathcal{NP}$-vollständig, d.h. die Algorithmen haben im schlechtesten Fall exponentielle Laufzeit.

Wir betrachten ein Beispiel von Appel:

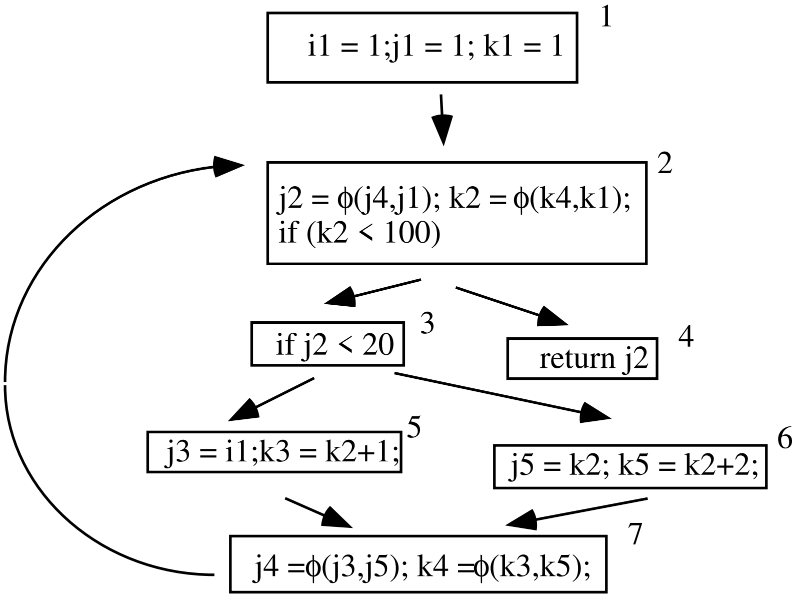
Beispiel 13.

```
i = 1; j := 1; k := 1;
while (k < 100)
   if (j < 20) {j = i; k = k+1;}
   else {j = k; k = k+2;};
 return j
```

Der Graph des Steuerungsflusses (ein Flußdiagramm) ist:



Wir verwenden im Bild das Symbol Φ. Dies soll wie eine Funktion wirken: wenn man in einen Block von links oben kommt, dann wird der linke Wert der Variablen genommen, andernfalls der rechte. Dies ist ein sogenanntes SSA-Formular:



Die Φs lassen sich in Funktionen übersetzen, so daß man sieht, daß auch hier funktionales Programmieren die Struktur des Programmes am deutlichsten macht:

```
let i1 = 1;j1 = 1; k1 = 0
in let f2 j2 k2 =
     if k2 < 100
     then let f7 j4 k4 = f2 j4 k4
           in if j2 < 20
               then let j3 = i1; k3 = k2+1
                     in  f7 j3 k3
               else let j5 = k2; k5 = k2+2
                     in   f7 j5 k5
```