Compilerkonstruktion

Wintersemester 2015/16

Prof. Dr. R. Parchmann

26. Januar 2016

Maschinencode-Erzeugung für Syntax-Bäume

In diesem Abschnitt wird ausgehend von baumartigen DAGs oder Syntaxbäumen Maschinencode erzeugt.
Wir wollen im folgenden annehmen, dass nur 4 Typen von Maschinenbefehlen erlaubt sind:

- `LD` *reg*, *mem*
- `ST` mem, *reg*
- `OP` *reg*, *reg*, *reg*
- `OP` *reg*, *reg*, *mem*

Ziel des Verfahren ist es, Maschinencode zu erzeugen, der bei einer vorgegebenen Zahl von verfügbaren Registern den Baum mit einer minimalen Zahl von Zwischenspeicherungen temporärer Ergebnisse auswertet.

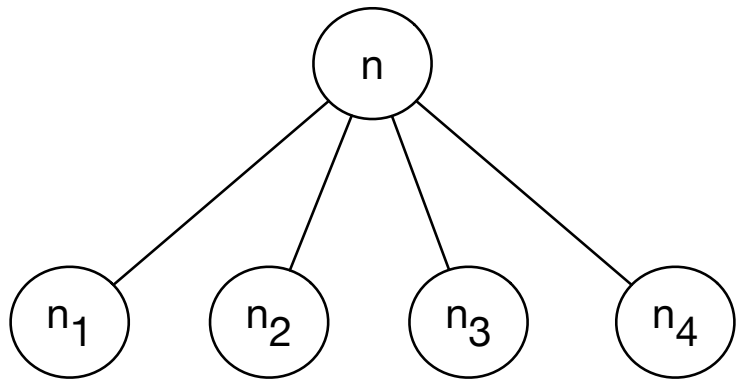
Die Idee besteht darin, zunächst den Baum einmal zu durchlaufen und Informationen über die Anzahl von Registern zu sammeln, die notwendig sind, um einen Teilbaum *ohne Zwischenspeicherungen* in den Speicher auszuwerten.
In einer zweiten Phase werden die so gewonnenen Informationen genutzt, um bei einem Knoten zu entscheiden, ob zuerst der linke oder der rechte Teilbaum ausgewertet werden soll.

Der Algorithmus läuft in zwei Phasen ab:

1. **Markierungsphase:** Für jeden Knoten n wird die minimale Anzahl $\mathrm{label}(n)$ von Registern (Ershov-Zahlen) bestimmt, die notwendig ist, um den entsprechenden Teilbaum ohne Speichern von Zwischenergebnissen auszuwerten. Diese Phase verläuft bottom-up.
2. **Code-Erzeugungsphase:** Jetzt werden die in 1) gefunden Werte benutzt, um einen möglichst effizienten Maschinencode zu erzeugen. Diese Phase verläuft top-down.

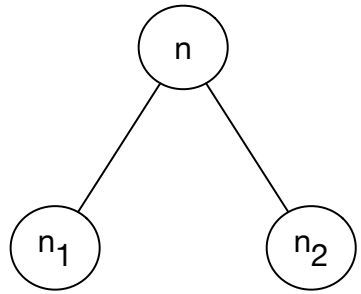
Die Markierungsphase

Betrachtet man z.B. einen Baum mit Wurzel n und Nachfolgerknoten n_1 bis n_4 und seien r_1 bis r_4 die jeweils minimal benötigten Register um den Teilbaum mit Wurzel n_1 bis n_4 auszuwerten.



Man benötigt für den Baum $\max\{r_1, r_2 + 1, r_3 + 2, r_4 + 3\}$ Register! Da die Reihenfolge der Auswertung der einzelnen Teilbäume aber frei wählbar ist, kann man diesen Wert möglichst klein halten.

Für den Spezialfall binärer Operatoren ergibt sich folgendes Bild



und damit die folgende Formel zur Berechnung der label-Funktion:

$$\text{label}(n) = \begin{cases} \max(\text{label}(n_1), \text{label}(n_2)) & \text{falls } \text{label}(n_1) \neq \text{label}(n_2) \\ \text{label}(n_1) + 1 & \text{falls } \text{label}(n_1) = \text{label}(n_2) \end{cases}$$

Bei unären Operatoren erhält man $\text{label}(n) = \text{label}(n_1)$.
Für die Blätter des Baumes gilt:

$$\text{label}(n) = \begin{cases} 1 & \text{falls n linkestes Blatt des Vorgängerknotens ist} \\ 0 & \text{sonst} \end{cases}$$

Eingabe: Ein Baum (Syntaxbaum)
Ausgabe: Derselbe Baum mit den Ershov-Zahlen an jedem Knoten.

Verfahren :
Sei n der aktuelle Knoten.

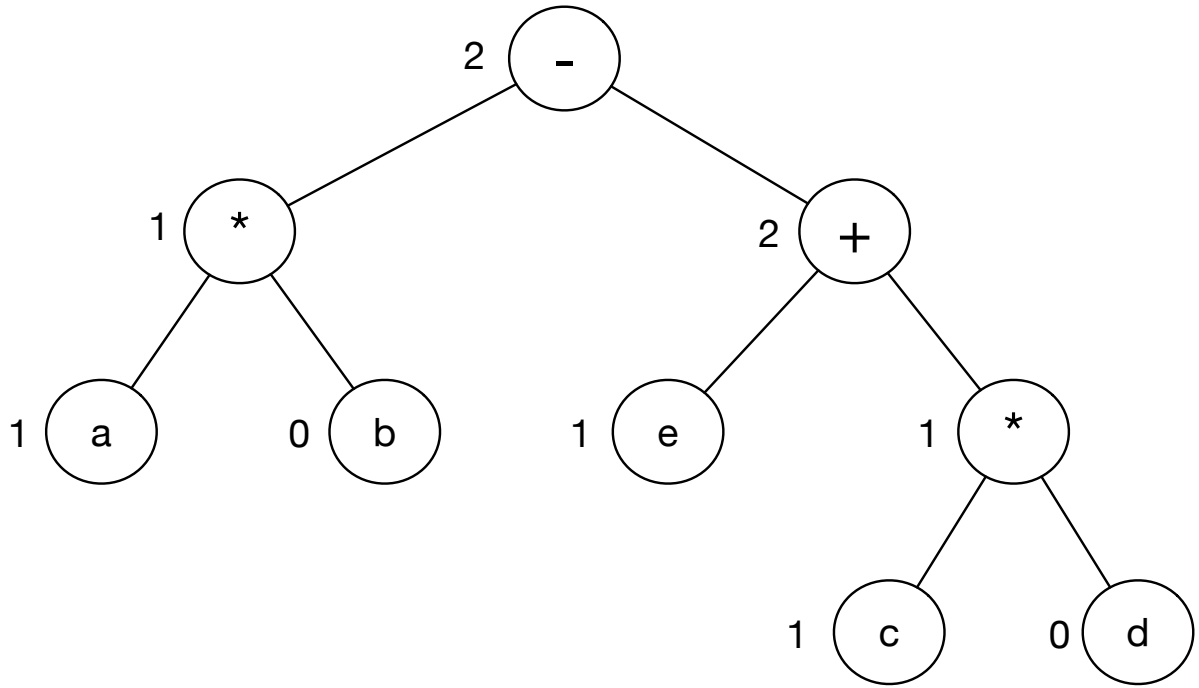
- Ist n ein Blatt und linkester Nachfolger des Vorgängerknotens, setze $\mathrm{label}(n) = 1$.
- Ist n ein Blatt, aber nicht linkester Nachfolger, setze $\mathrm{label}(n) = 0$.
- Ist n ein interner Knoten und seien $n_1, \ldots, n_k$ Nachfolgerknoten von n, wobei die Reihenfolge so gewählt sei, dass

  $$\mathrm{label}(n_1) \geq \mathrm{label}(n_2) \geq \ldots \geq \mathrm{label}(n_k)$$

  gilt.
  Dann setze $\mathrm{label}(n) = \max\{\mathrm{label}(n_i) + i - 1 \mid 1 \leq i \leq k\}$

Beispiel

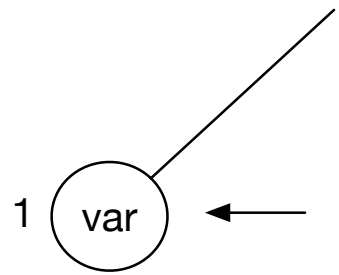
Man betrachte den arithmetischen Ausdruck `a*b - e + c*d`. Der zugehörige DAG ist der folgende Baum, wobei die Zahlen an den Knoten den Wert der label-Funktion angeben:



Die Code-Erzeugungsphase

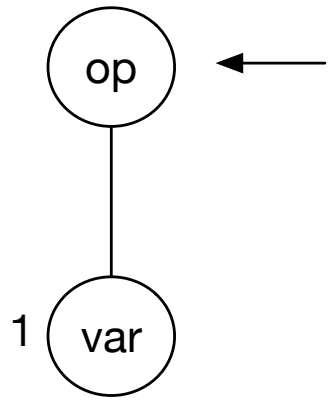
- Der Baum wird jetzt mit einer rekursiven Methode `gencode` top-down durchlaufen.
- Der Algorithmus verwaltet einen Stack, in dem die Namen `R0`, ..., $\texttt{R}_{r-1}$ der insgesamt r freien Register gespeichert sind.
- Bei einem Aufruf von `gencode(n)` erzeugt die Methode Maschinencode zur Auswertung des Teilbaums mit Wurzel `n`, wobei nur Register benutzt werden, deren Namen sich momentan im Stack befinden.
- Der Wert des Teilausdrucks befindet sich später im Register, dass oben im Stack steht.
- Beim Verlassen der Methode befindet sich der Stack wieder im gleichen Zustand wie beim Eintritt.
- Mit der Methode `swap` werden die beiden obersten Einträge im Stack vertauscht.

Im Algorithmus sind die folgenden Fälle zu unterscheiden:
(hier sei angenommen, dass der oberste Stackeintrag R_i ist und darunter R_j liegt. Der Pfeil zeigt immer auf den aktuellen Knoten.)
①



Das Blatt ist linker Nachfolger eines Knoten mit > 1 Nachfolgern
$\longrightarrow$ `LD R`i`,var`

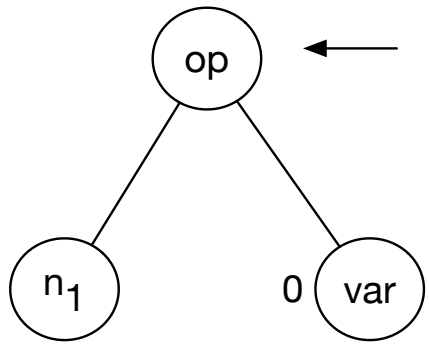
②



Das Blatt ist einziger Nachfolger des Vorgängerknotens, dann

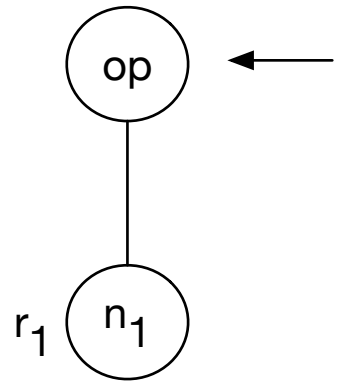
$\longrightarrow$ `op` $\mathtt{R}_i$`,var`

③



n_1 auswerten, das Ergebnis steht in R_i, dann $\longrightarrow$ `op R`$_i$`,R`$_i$`,var`

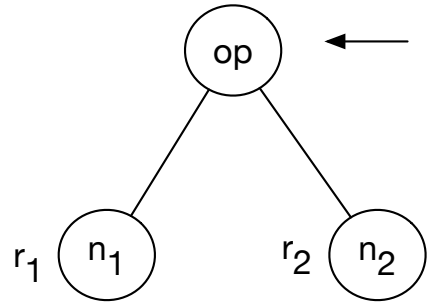
④



$r_1 > 1$

n_1 auswerten, das Ergebnis steht in R_i, dann $\longrightarrow$ op $\mathtt{R}_i\,,\mathtt{R}_i$

⑤

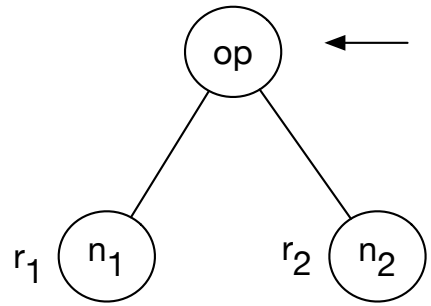


$$1 \leq r_1 < r_2$$

$$r_1 < r$$

`swap` ausführen, R_j ist oben auf dem Stack
n_2 auswerten $\rightarrow$ Ergebnis steht in R_j
R_j aus Stack entfernen, dann
n_1 auswerten $\rightarrow$ Ergebnis steht in R_i
$\longrightarrow$ op $\mathtt{R}_i, \mathtt{R}_i, \mathtt{R}_j$
R_j auf den Stack und `swap` ausführen

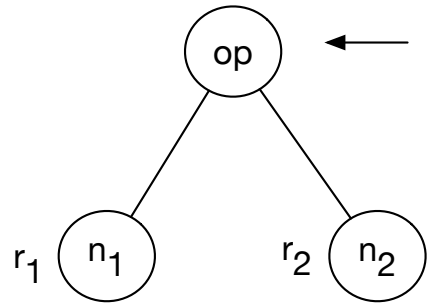
⑥



$$r_1 \geq r_2 \geq 1$$
$$r_2 < r$$

n_1 auswerten $\rightarrow$ Ergebnis steht in R_i
R_i aus dem Stack entfernen, dann
n_2 auswerten $\rightarrow$ Ergebnis steht in R_j
$\longrightarrow$ op $\mathtt{R}_i, \mathtt{R}_i, \mathtt{R}_j$
R_i auf den Stack legen.

⑦



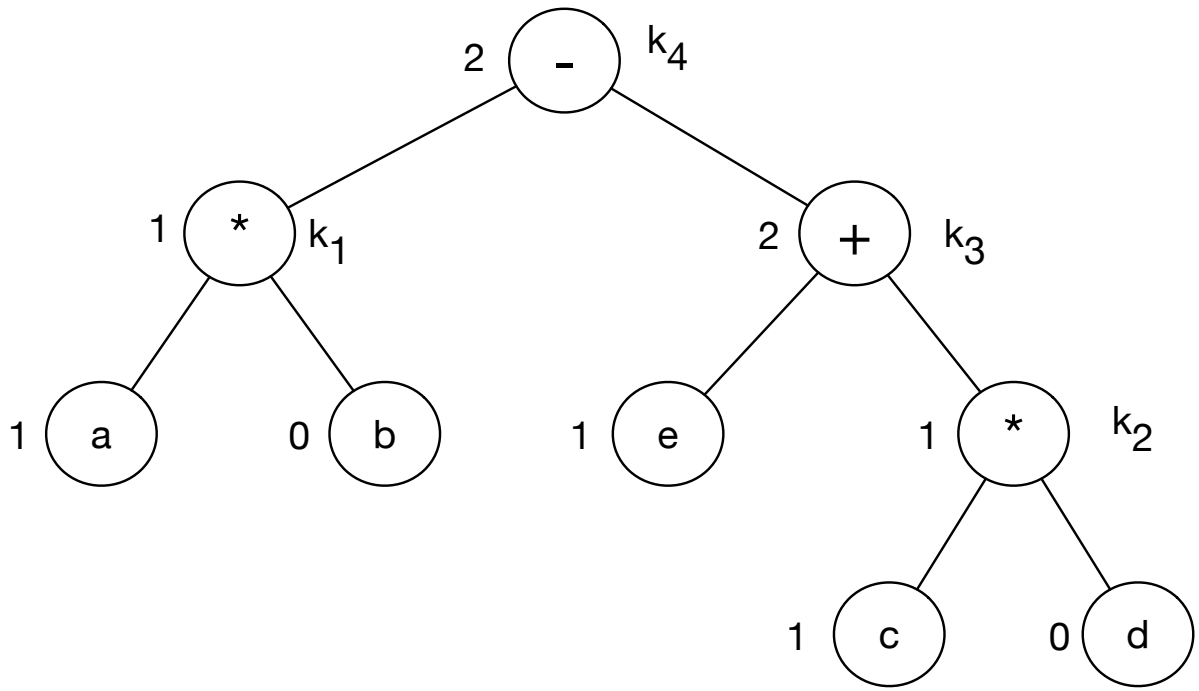
$$r_1 \geq r$$
$$r_2 \geq r$$

n_2 auswerten $\rightarrow$ Ergebnis steht in R_i
$\longrightarrow$ `ST Temp, R`$_i$
n_1 auswerten$\rightarrow$ Ergebnis steht in R_i
$\longrightarrow$ `op R`$_i$`,R`$_i$`,Temp`

Beispiel

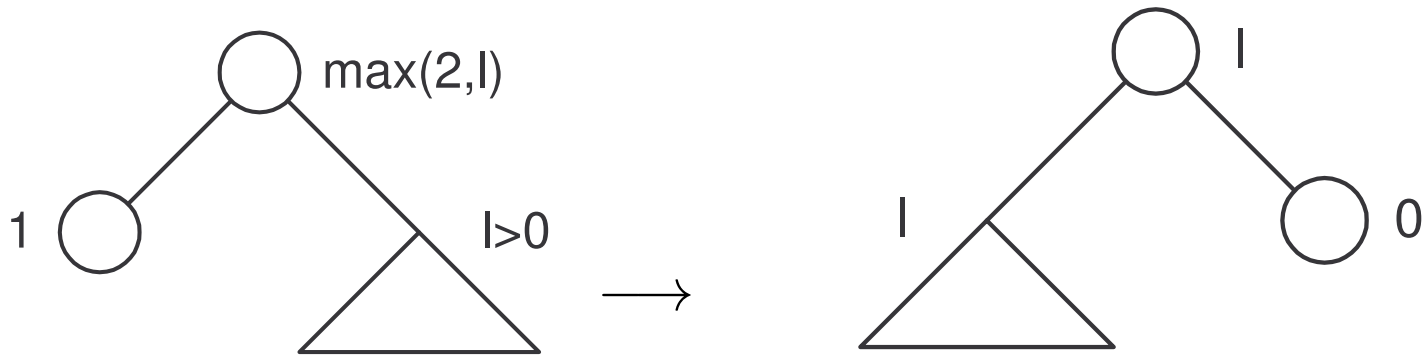
Es soll der folgende Syntaxbaum in Maschinencode übersetzt werden, wobei angenommen wird, dass die beiden Register `R0` und `R1` frei sind. Es ist also $r = 2$

Zur besseren Übersicht sind hier die Knoten zusätzlich nummeriert



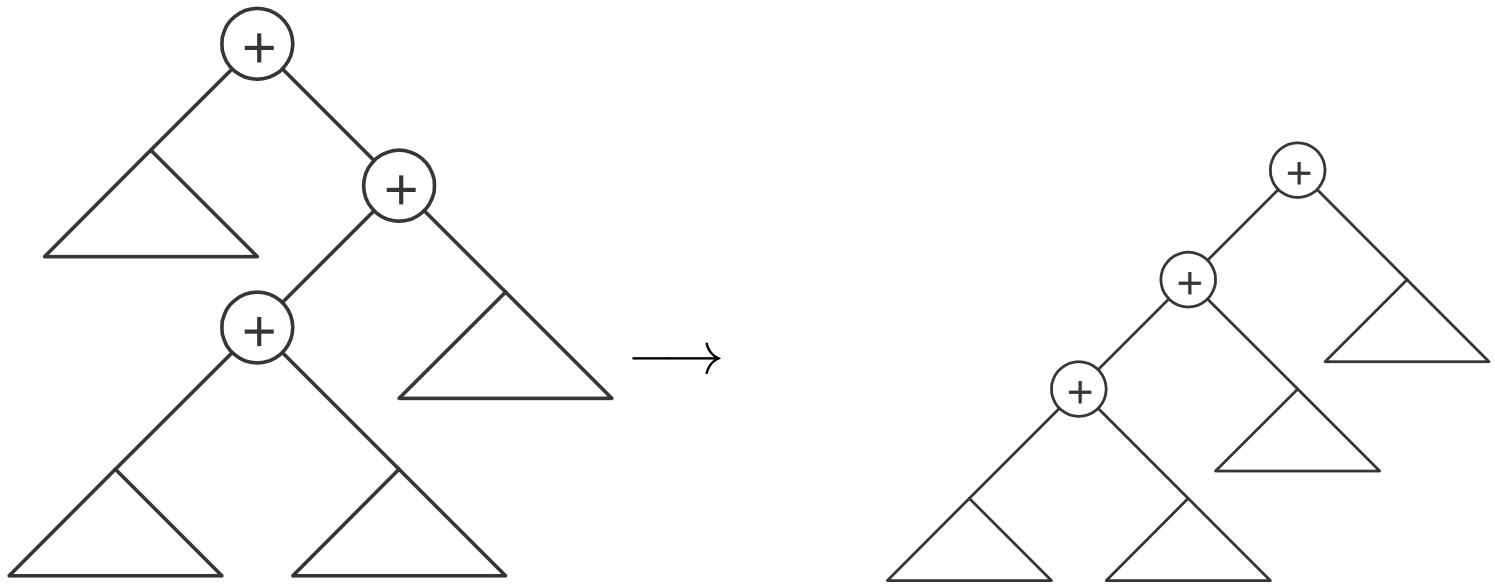
Aufrufe	Registerstack	Fall	Maschinencode
gencode(k_4)	R1 R0	5	
gencode(k_3)	R0 R1	6	
gencode(e)	R0 R1	1	
		1	LD R1,e
gencode(k_2)	R0	3	
gencode(c)	R0	1	
		1	LD R0,c
		3	MUL R0,R0,d
	R0 R1	6	ADD R1,R1,R0
gencode(K_1)	R0	3	
gencode(a)	R0	1	
		1	LD R0,a
		3	MUL R0,R0,b
		5	SUB R0,R0,R1
	R1 R0		

Man kann zeigen, dass der Algorithmus `gencode` unter den gemachten Voraussetzungen (keine algebraischen Umformungen, keine gemeinsamen Teilausdrücke, betrachtete Maschinenbefehle) optimalen Code, d.h. Code mit der kürzesten Befehlssequenz, erzeugt.
Viele Variationen dieses Verfahrens sind denkbar. So kann man z.B. algebraische Umformungen direkt auf den Baum anwenden, um eventuell die Zahl der benötigten Register zu verringern, etwa:



So wird bei kommutativen Operatoren ein `LD`- Befehl gespart.

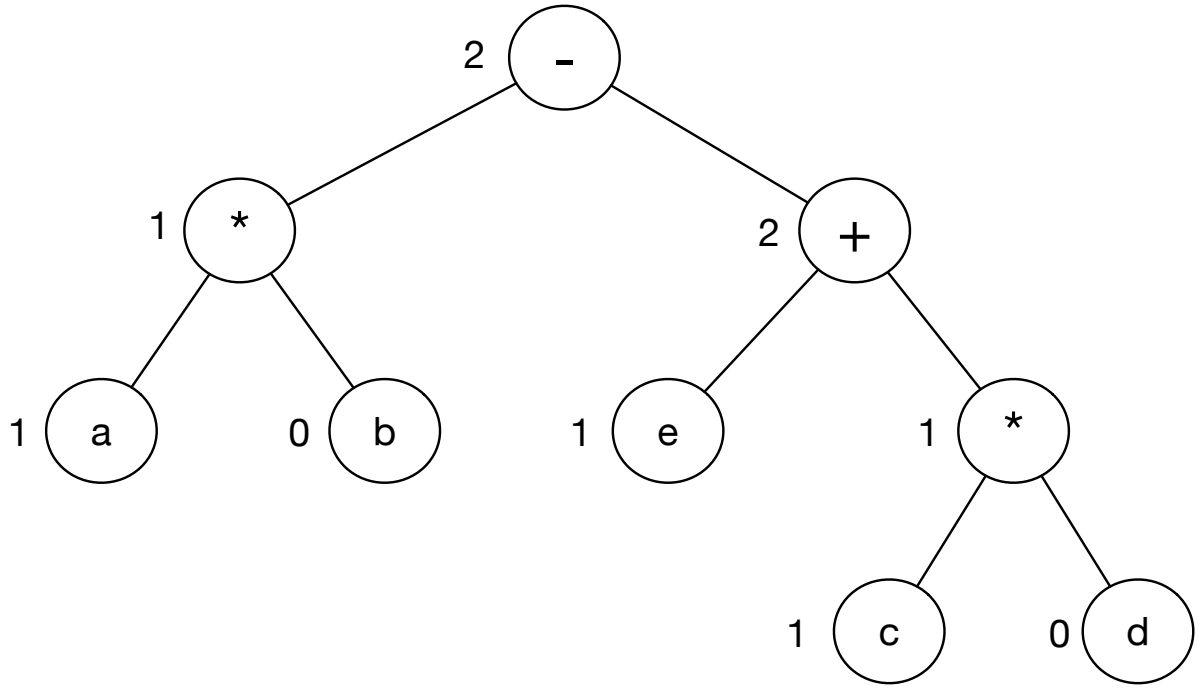
Bei assoziativen Operatoren kann man den Baum ebenfalls umordnen, um die Zahl der benötigten Register eventuell zu verringern:



Das Verfahren liefert natürlich nur für die dort eingeführte Maschine optimalen Code. Erweitert man z.B. die Maschinensprache, dann erhält man mit diesem Verfahren unter Umständen nicht mehr optimale Maschinencode-Sequenzen.

Beispiel

Unser Beispiel `a*b - e + c*d` mit dem Baum



aus vorigem Beispiel ergibt unter der Annahme, das nur ein Register `R0` frei ist den folgenden Maschinencode mit Kosten = 27:

```
LD  R0,c
MUL R0,R0,d
ST  Temp,R0
LD  R0,e
ADD R0,R0,Temp
ST  Temp,R0
LD  R0,a
MUL R0,R0,b
SUB R0,R0,Temp
```

Hat die Maschine andererseits Befehle, bei denen nicht mindestens einer der beiden Operanden in einem Register liegen muss, wäre auch das folgende Programm mit Kosten = 21 möglich:

```
MUL R0,c,d
ADD R0,e,R0
ST  Temp,R0
MUL R0,a,b
Sub R0,R0,Temp
```

Im allgemeinen Fall wird man speziell bei *heterogeneren* Befehlssätzen einen anderen Ansatz probieren müssen. Außerdem müssen natürlich auch die Kosten der evtl. vielen anwendbaren Befehle in Betracht gezogen werden.
Eine Möglichkeit, unter diesen Nebenbedingungen optimalen oder fast optimalen Code zu erzeugen, besteht in der Anwendung des Prinzips der dynamischen Programmierung

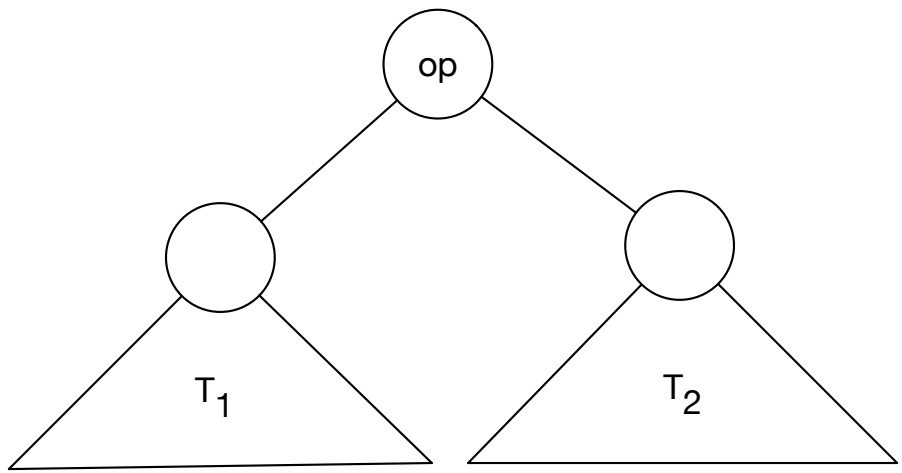
Code-Erzeugung mit dynamischer Programmierung

In diesem Abschnitt wird eine Verallgemeinerung des Verfahrens aus vorigem Abschnitt vorgestellt, die unter gewissen Annahmen optimalen Code liefert, jedoch auch für komplexere und umfangreichere Sätze von Maschinenbefehlen gute Übersetzungen liefert. Das Verfahren beruht auf dem Prinzip der dynamischen Programmierung und ist in der Lage, auch die Kosten der einzelnen Maschinenbefehle zu berücksichtigen.

Gegeben ist wieder ein baumartiger DAG bzw. ein Syntaxbaum T und eine Anzahl r von frei verfügbaren Registern.

Bei der dynamischen Programmierung wird das Problem der Erzeugung optimalen Codes für einen arithmetischen Ausdruck E zerlegt in das Problem, optimale Übersetzungen für die Teilausdrücke zu finden.

Man betrachte etwa einen Ausdruck $E = E_1 \,\mathrm{op}\, E_2$. Dann wird die optimale Übersetzung für E dadurch gebildet, dass man die optimalen Übersetzungen für E_1 und E_2 mit dem Code zur Auswertung von op kombiniert.
Sei T der Syntaxbaum für E und T_1 bzw. T_2 die Teilbäume für E_1 bzw E_2



Ein Programm P wertet einen Baum T *kontinuierlich* aus, wenn zuerst die Teilbäume von T auswertet werden, deren Ergebnisse in den Speicher geschrieben werden. Anschließend wird der Rest von T ausgewertet, entweder in der Reihenfolge T_1, T_2 oder in der Reihenfolge T_2, T_1, wobei die im Speicher abgelegten Werte mit benutzt werden. Abschließend wird dann der Wert für die Operation an der Wurzel von T berechnet.

Für die in diesem Abschnitt betrachteten Maschinenbefehle kann man zeigen, dass für jede Übersetzung P von T ein äquivalentes Programm P' existiert mit:

1. P' hat Kosten kleiner gleich den Kosten von P
2. P' nutzt nicht mehr Register als P
3. P' wertet den Baum kontinuierlich aus.

Daraus folgt, dass jeder Syntaxbaum eines arithmetischen Ausdrucks durch ein optimales Programm kontinuierlich ausgewertet werden kann.

Bemerkung

Dies gilt jedoch nur, wenn die erlaubten Maschinenbefehle vom beschriebenen Typ sind. Zum Beispiel für Maschinen, die Registerpaare für Multiplikation und Division benutzen gilt dies nicht. Man kann zeigen, dass es Beispiele optimaler Programme für derartige Maschinen gibt, in denen erst Teile von T_1, dann Teile von T_2, dann wieder Teile von T_1 usw., ausgeführt werden müssen.

Der Algorithmus

Der Algorithmus verläuft in drei Phasen:

1. In der ersten Phase wird für jeden Knoten n des Syntaxbaumes ein $r+1$-stelliger Kostenvektor C berechnet.
   - $C[0]$ enthält die minimalen Kosten eines Maschinenprogramms zur Berechnung des Teilbaums mit Wurzel n, wobei das Ergebnis im Speicher abgelegt wird.
   - $C[i]$ enthält die minimalen Kosten eines Maschinenprogramms zur Berechnung dieses Teilbaums unter Verwendung von i Registern, wobei das Ergebnis in einem Register gehalten wird.

   Zu jedem Eintrag des Kostenvektors merkt man sich ausserdem den Maschinenbefehl, der zu den minimalen Kosten geführt hat, um später das optimales Programm zu konstruieren.

2. Dann bestimmt man mit Hilfe der Kostenvektoren, welche Teilbäume S von T so ausgewertet werden müssen, dass die Ergebnisse im Speicher abgelegt werden.
3. Man erzeugt für jeden unter 2) gefundenen Teilbaum S mit Hilfe der Kostenvektoren und der zugeordneten Maschinenbefehle die Übersetzung von S. Dann erzeugt man, wieder unter Benutzung der Kostenvektoren und der zugeordneten Befehle, die Übersetzung des restlichen Baums.

Um die Kosten $C[i]$ für den Knoten n zu bestimmen, betrachtet man alle Maschinenbefehle, die die Operation in n realisieren.

- Für jeden möglichen Maschinenbefehl B berechnet man mit Hilfe der Kostenvektoren der Nachfolgerknoten von n die Kosten von n.
- Für die Register-Operanden von B betrachtet man dabei alle möglichen Reihenfolgen der Auswertung. Der erste Teilbaum kann mit i Registern ausgewertet werden, der nächste mit $i - 1$ und so weiter.
- Hinzu kommen noch die Kosten für den Befehl B.
- Der Wert von $C[i]$ ergibt sich dann als das Minimum aller Möglichkeiten.

Üblicherweise speichert man zum Wert $C[i]$ auch noch den Befehl B, der zum Minimum führte.

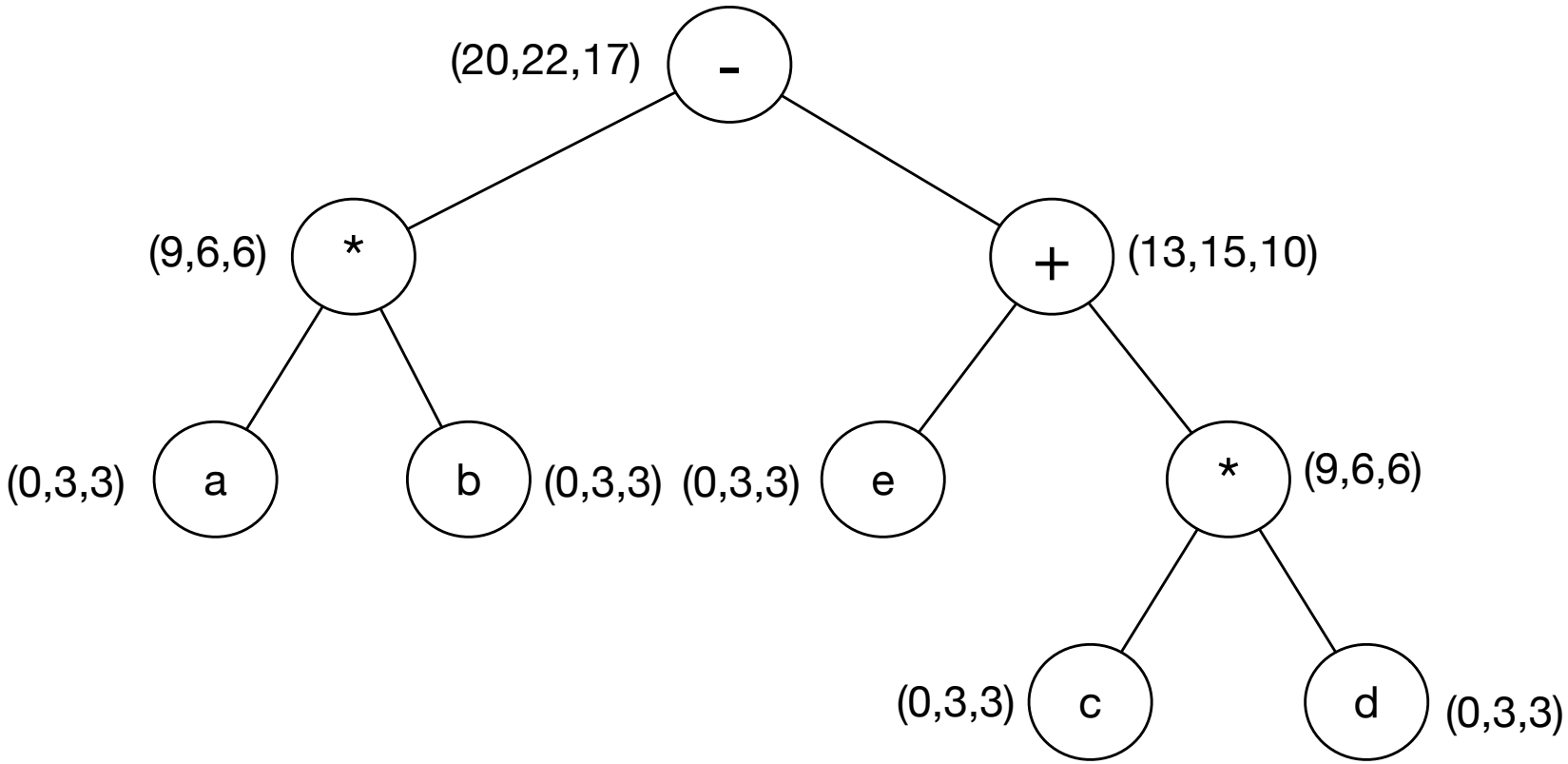
Letztlich gibt der minimale Wert im Kostenvektor der Wurzel von T die Kosten des optimalen Programms zur Auswertung von T an.

Beispiel

Wir betrachten eine Maschine, die zwei Register `R0` und `R1` frei hat und den folgenden Befehlssatz besitzt:

- `LD` *reg*, *mem*
- `ST` mem, *reg*
- `OP` *reg*, *reg*, *reg*
- `OP` *reg*, *reg*, *mem*

Übersetzt werde soll der Syntaxbaum vom vorigen Beispiel. Zunächst bestimmt man die Kostenvektoren für alle Knoten im Baum. Man erhält:



Nach der Berechnung der Kostenvektoren kann man nun den Baum ein weiteres mal durchlaufen und die Befehle, die zu den jeweils minimalen Kosten führten, zu einem optimalen Programm zusammensetzen. Man erhält:

```
LD  R0,c
MUL R0,R0,d
LD  R1,e
ADD R1,R1,R0
LD  R0,a
MUL R0,R0,b
SUB R0,R0,R1
```