5.3 Codegenerierung

Zur Vorlesung

Embedded Systems

WS 14/15

Reiner Kolla

5.3.1 Problemstellung

Wir gehen nun davon aus, dass Zwischencode in 3-Adressdarstellung schon gegeben und eine Registerzuweisung für die Variablen schon erfolgt ist, d.h. gewisse Variablen im 3 Adresscode sind Registerbezeichner, andere bezeichnen globale bzw. temporäre Variablen.

Für die Identifier im 3 Adresscode gelte also folgende Interpretation:

- Ri: Register Nummer *i*
- gSi: Globale Variable Nummer *i*. Wir assoziieren damit eine Konstante *#g_i*, die die Anfangsadresse der Variablen im Speicher angibt.
- tSi: Temporäre Variable Nummer *i*. Wir assoziieren mit jeder temporären Variablen eine Konstante *#t_i*, die die Anfangsadresse der temporären Variablen auf dem Stack relativ zum Stackpointer *SP* angibt.

Zur Vereinfachung wollen wir annehmen, dass die Zielmaschine nur Worte als Objekte kennt, wir lassen die Größenangabe also weg.

Eine einfache Zielmaschine

Zur Illustration wollen wir ferner eine einfache Zielmaschine (CISC) betrachten, die im Wesentlichen orthogonalen 2 Adresscode bereitstellt, d.h. jeder Befehl hat zwei Operanden, von denen der erste Ziel- und ggf. auch Quelloperand ist, der zweite nur Quelloperand.

Wir betrachten folgende Adressierungsarten:

- unmittelbar: #Konstante, nur beim 2. Operand
- direkt: Ri, $0 \le i \le$ Registerzahl-1 beide Operanden
- indirekt: *Ri, $0 \le i \le$ Registerzahl-1 beide Operanden
- indiziert: C(Ri), $0 \le i \le$ Registerzahl-1 beide Operanden

Mit diesen Adressierungsarten betrachten wir nun

- Transportbefehle: MV, Beispiel: MV R1,*R2 -- reg[1] = mem[reg[2]]
- Arithmetikbefehle: ADD, SUB, MUL, ...

  Beispiel: ADD C(R0),R1

  -- mem[reg[0]+C] = mem[reg[0]+C] + reg[1] ;

Eine einfache Zielmaschine -- ff

Wir werden uns bei der Codegenerierung im Wesentlichen auf Grundblöcke beschränken und lassen daher die Kontrollanweisungen, die solche Blöcke abschließen, zunächst mal außen vor.

Man erkennt:

Hätten wir eine 3 Adressmaschine, könnten wir mit diesen Adressierungsarten Befehl für Befehl direkt in einen Maschinenbefehl packen.

Bei unserer Beispielmaschine müssen wir manche Befehle durch Befehlsfolgen ersetzen und brauchen dazu auch mindestens ein freies Register (z.B. R0):

Beispiel:

Betrachte das Statement

t1 = g2 * t5;

Möglicher Code:

```
MV    R0,#g_2
MV    R0,*R0
MUL   R0,#t_5(SP)
MV    #t_1(SP),R0
```

Operatorbäume

Wir sehen, dass es gerade bei CISC Befehlssätzen nicht trivial ist, möglichst kompakte und zugleich effiziente Codesequenzen für Statements der Zwischensprache zu finden.

Wir sehen auch:

Es macht wenig Sinn, Statements der Zwischensprache Zeile für Zeile in Code umzusetzen, weil man dabei den Zusammenhang zwischen den Statements übersieht. Vor allem wenn komplexere Ausdrücke in Zwischencode aufgebrochen wurden, entstehen viele temporäre Variablen, die meist an einige, wenige Register gebunden werden können.

Besser:

Rekonstruiere aus dem Zwischencode **Operatorbäume**, die für Werte stehen, die an Register oder Variablen zugewiesen werden.

Sobald eine indizierte Zuweisung oder eine Zuweisung an eine Variable erfolgt, die bis zum Verlassen den Blockes lebt, liegt notwendigerweise die Wurzel eines Operatorbaumes vor.

Operatorbäume -- Definition

Ein **Operatorbaum** ist ein geordneter orientierter Baum mit einer Wurzel *r*, dessen Knoten *v* mit Operatoren *op(v)* markiert sind. Ist *op(u) = op(v)*, dann muss die Zahl der Kinder im Baum ebenfalls gleich sein. Die Blätter des Baumes stehen für Operatoren, die keine weiteren Operanden mehr haben. Sie tragen ggf. Attribute, wobei für zwei Blätter *u,v*, mit *op(u) = op(v),* die Namen der Attribute wieder übereinstimmen müssen.

Beispiel: Wir betrachten folgende Operatoren:

⊙ arithmetische Operatoren	{+,-,*,/}	mit jeweils zwei Kindern,
⊙ Zuweisungsoperator	=	mit zwei Kindern,
⊙ Indirektionsoperator	ind	mit einem Kind
⊙ Konstante	C	Blatt mit Attribut <Wert>
⊙ Register	R	Blatt mit Attribut <Nummer>
⊙ Speicherplatz	M	Blatt mit Attribut <Adresse>

Beispiel: Extraktion von Operatorbäumen

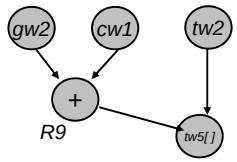
Wir betrachten folgende Sequenz in einem C - Programm

```
int b;
{       int a[]; int i;…
        …
        a[i] = b + 1;
};
```

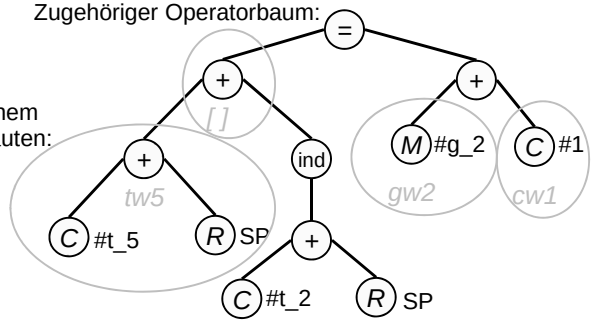
In A3 könnte dies in einem Grundblock wie folgt lauten:

```
R9 =  gw2 + cw1;
tw5[tw2] = R9;
```

Extrahierter Taskbaum:



Zugehöriger Operatorbaum:



5.3.2 Codeselektion

Die gebräuchlichste Methode zur Codeselektion sind sogenannte Baumtransformationen:

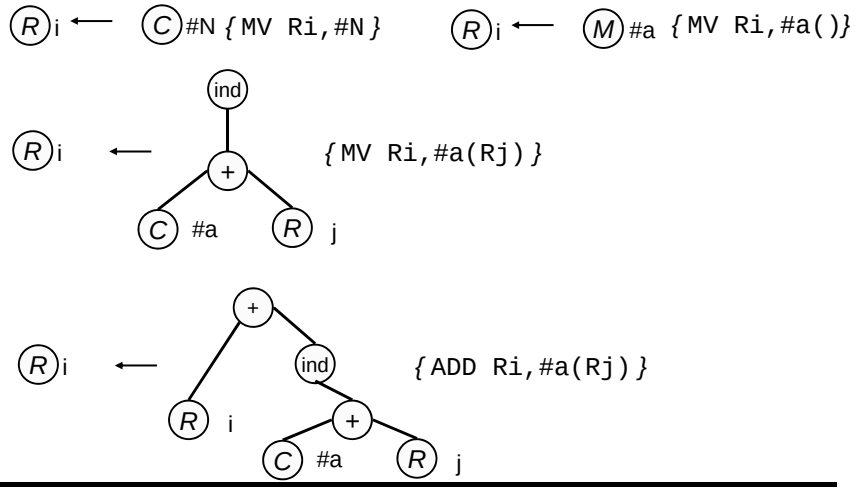
Definition

Eine Regel der Form *E(f(x)) ← T(x) { A }*, wobei *E* ein nullstelliger Operator ist, *f(x)* eine Berechnungsvorschrift für die Attribute von *E*, und *T* ein Operatorbaum mit Blattattributen *x* ist, heißt **Baumtransformationsregel**. *A* ist eine mit dieser Regel verbundene Aktion. Eine Menge *G* von Baumtransformationsregeln heißt **Baumübersetzungsschema**.

Baumtransformationsregeln lassen sich dazu benutzen, zu beschreiben, wie sich Teilbäume von Operatorbäumen durch Maschinenbefehle realisieren lassen. Man kann nun den kompletten Befehlssatz einer Maschine mit einer hinreichend großen Menge von Baumtransformationsregeln, versehen mit Kosten, beschreiben. In diesem Fall spricht man auch von einer „**Maschinengrammatik**“.

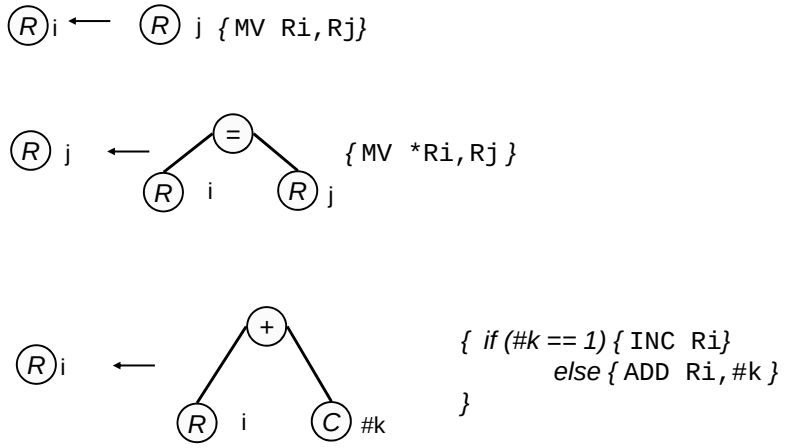
Beispiel: Baumtransformationsregeln

Für unsere einfache Zielmaschine lassen sich beispielsweise folgende Regeln angeben: (Nur eine Auswahl)



Beispiel: Baumtransformationsregeln ff

Für unsere einfache Zielmaschine lassen sich beispielsweise folgende Regeln angeben: (Nur eine Auswahl)



Codeselektion durch Baumtransformationen

Im Allgemeinen definiert man so mittels einer einfachen Sprache eine riesige Zahl von Regeln. Die Codeselektion erfolgt nun durch Transformation eines Operatorbaumes mittels dieser Regeln, bis der Baum auf einen einzigen Knoten reduziert ist.

Achtung:

Die Tatsache, ob jeder Operatorbaum reduzierbar ist, hängt von den Regeln ab. Die Regeln sollten so gestaltet sein, dass mindestens eine Reduktion auf einen Knoten möglich ist.

Definition -- Anwendung von Transformationsregeln

Eine Baumtransformationsregel $E(f(x)) \leftarrow T(x)\ \{ A \}$, heißt **anwendbar** auf einen Knoten *u* eines Operatorbaumes, genau dann, wenn *T* dem Unterbaum mit Wurzel *u* entspricht. Man erhält die Reduktion durch die Regel am Knoten *u*, indem man *u* durch *E* mit der Attributierung *f(x)* ersetzt. Als Codesegment nehme man die Codesegmente an den Blättern des ersetzten Unterbaumes „in geeigneter Reihenfolge“ (→ Ablaufplanung!) und füge *A* an.

Codeselektion durch Baumtransformationen

Wir können nun das Problem der Codeselektion als das Problem, eine kostengünstigste Transformation eines Operatorbaumes zu konstruieren, auffassen:

Problem:

Gegeben:

Ein Operatorbaum *T* zu einem (Teil) eines Grundblocks

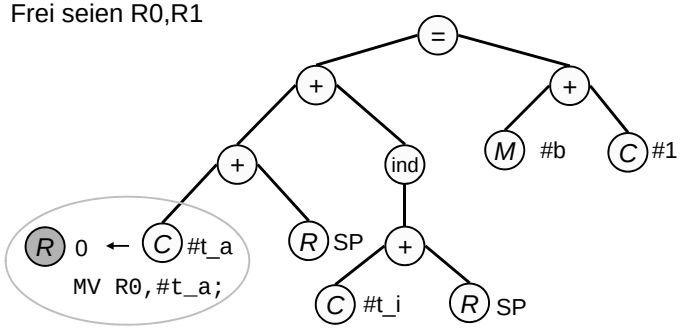
Eine Menge *R_free* von Registern, die für die Codegenerierung für *T* frei sind. Frei sind alle Register, die nicht als Operand in *T* vorkommen und bei Austritt aus dem zu *T* gehörigen A3-Codesegment nicht aktiv sind.

Gesucht:

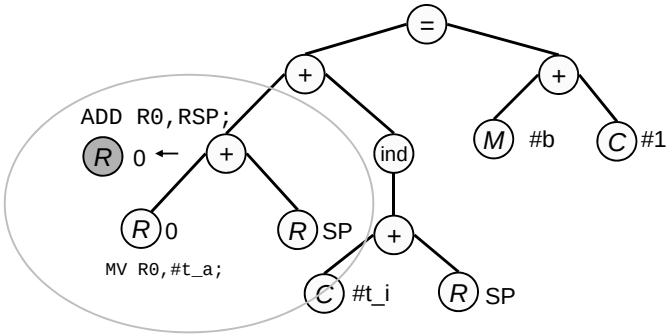
Eine Folge von Baumtransformationen, so dass *T* zu einem Knoten reduziert wird und der damit assoziierte Code kostenminimal ist.

Beispiel: Reduktion von Operatorbäumen

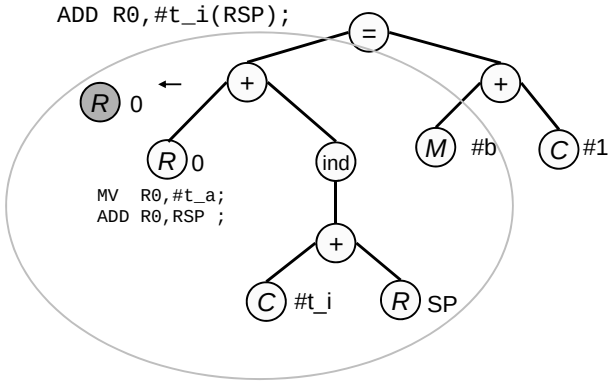
Wir hatten folgenden Operatorbaum zu a[i] = b + 1:
Frei seien R0,R1



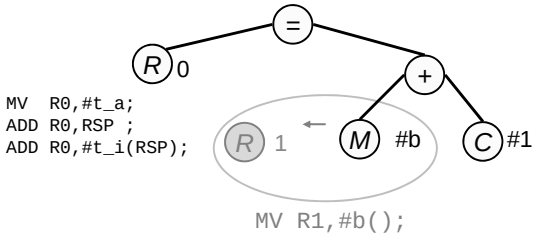
Beispiel: Reduktion von Operatorbäumen



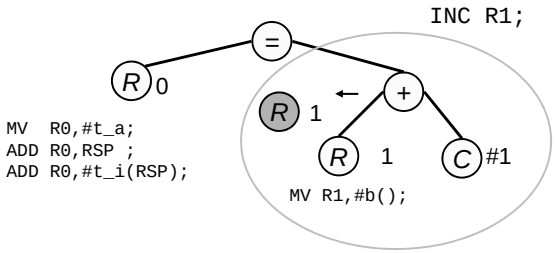
Beispiel: Reduktion von Operatorbäumen



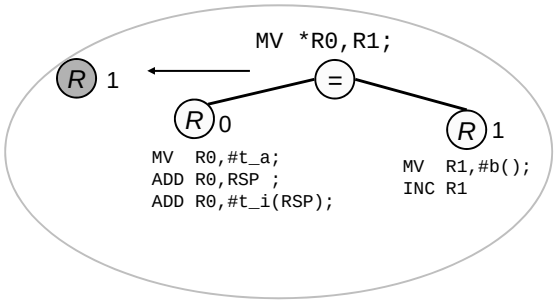
Beispiel: Reduktion von Operatorbäumen



Beispiel: Reduktion von Operatorbäumen



Beispiel: Reduktion von Operatorbäumen



Resultat:

R 1

```
MV  R0,#t_a;
ADD R0,RSP ;
ADD R0,#t_i(RSP);
MV  R1,#b();
INC R1;
MV  *R0,R1;
```

Registervergabe bei der Codeselektion

Problem:

Im Allgemeinen hat man nicht beliebig viele freie Register zur Verfügung, sondern nur die Register, die in der Umgebung des Operatorbaumes nicht aktiv sind, sind nutzbar.

Idee:

Modelliere die Nutzung des Speichers als Zwischenspeicher durch einfache Transformationsregeln. Sei #top die Konstante, die auf die erste freie Speicherzelle des Stacks zeigt.

S #top ← R ¡ { `MV #top(RSP),Rj;` #top = #top +1 }

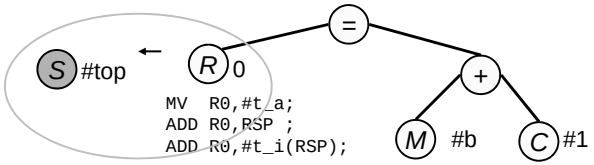
Nachführen von #top

Damit wird Rj frei, falls Rj als Hilfsregister benutzt wurde.

Eine Transformationsregel mit Ri **als Wurzel darf nur angewandt werden, wenn** Ri **frei ist!**

Beispiel: Reduktion von Operatorbäumen

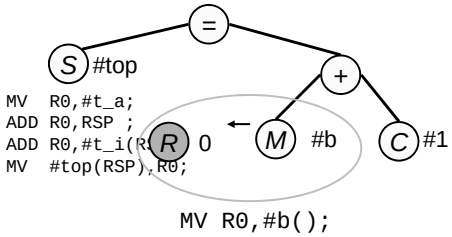
Wir nehmen an, dass nur R0 zur freien Verfügung stand:




```
MV #top(RSP),R0;
```

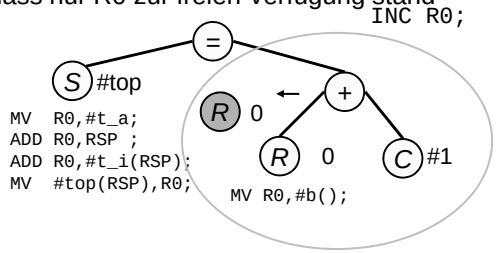
Beispiel: Reduktion von Operatorbäumen

Wir nehmen an, dass nur R0 zur freien Verfügung stand:

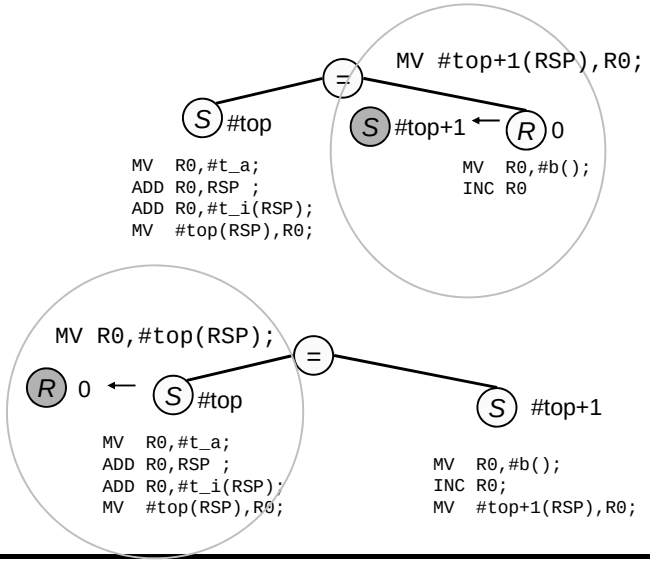


Beispiel: Reduktion von Operatorbäumen

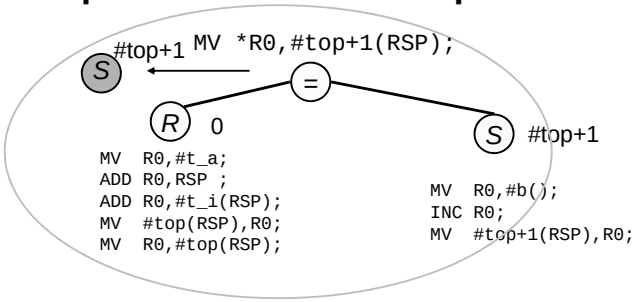
Wir nehmen an, dass nur R0 zur freien Verfügung stand



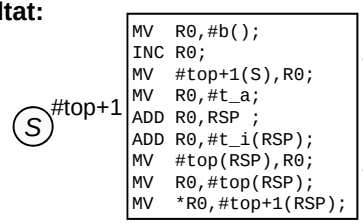
Beispiel: Reduktion von Operatorbäumen



Beispiel: Reduktion von Operatorbäumen



Resultat:



Beobachtungen:

- Der Code des rechten Unterbaums muss vor den des linken Unterbaums gelegt werden, d.h. Speicher-operanden vor Registeroperanden.
- Die Transformation erzeugt keinen optimalen Code !

Codegenerierungsverfahren

Das Problem, eine kostengünstigste Transformation eines Operatorbaumes zu konstruieren, kann mit dynamischem Programmieren gelöst werden.

Problem -- Codegenerierung

Gegeben:

Ein Operatorbaum *T* zu einem Teil eines Grundblocks.

Eine Menge *R_free* von Registern, die für die Codegenerierung für *T* frei sind.

Gesucht:

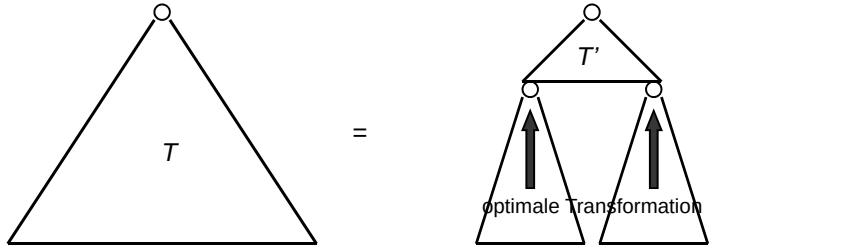
Eine Folge von Baumtransformationen, so dass *T* zu einem Knoten reduziert wird und der damit assoziierte Code kostenminimal ist.

Codegenerierungsverfahren -- ff

Beobachtung:

Eine kostenminimale Codesequenz für einen Operatorbaum *T* setzt sich stets zusammen aus

- einem Befehl zu einem Baummuster *T'*, das an die Wurzel des Baumes angelegt wird, und
- kostenminimalen Codesequenzen für die zu den Blättern von *T'* korrespondierenden Knoten $u_1,\ldots, u_k$ unter der Nebenbedingung, dass die Unterbäume $T(u_1),\ldots, T(u_k)$ auf die Blattsymbole von *T'* reduziert werden können.



Codegenerierungsverfahren -- ff

Idee:

Da offensichtlich die Menge der möglichen Transformationen eines Unterbaums *T(u)* keinen Einfluss auf die Menge der möglichen Transformationen *T(v)* hat, wenn *v* nicht in *T(u)* und *u* nicht in *T(v)* liegt, bietet es sich an, durch dynamisches Programmieren für jeden Knoten *u* bottom up die Liste

Sym(u) := { Symbole *x*, auf die *T(u)* reduziert werden kann *}*

sowie für jedes $x \in Sym(u)$ die Mengen

Pat(x,u) := { T', T' hat Wurzelmarkierung *op(u)* und kann zur Reduktion auf *x* durch $x \leftarrow T'$ benutzt werden *}*

cost(x,u) := Kosten einer kostenminimalen Reduktion von *T(u)* auf *x*

zu erzeugen.

Problem:

Wir haben die Registervergabe noch nicht berücksichtigt.

Dies lässt sich aber in den Prozess integrieren: Wir nehmen an, dass wir *k* freie Register haben und diese der Reihe nach vergeben.

Codegenerierungsverfahren -- ff

Idee:

Wir erweitern zunächst die bottom up zu berechnende Kostenfunktion zu

cost(x,j,u) := Kosten einer kostenminimalen Reduktion von *T(u)* auf *x,* unter der Bedingung, dass für *T(u) j* freie Register zur Verfügung stehen.

Problem:

Wir müssen letztlich auch in der Lage sein, den kostengünstigsten Code zu generieren:

Halte für jedes $x \in Sym(u)$

best_pattern(x,j,u) := Nummer der Regel $x \leftarrow T'$, so dass *T'* an die Wurzel von *u* gelegt werden muss, um eine kostenminimale Codesequenz für *T(u)* mit *j* freien Registern zu erzeugen.

Das dynamische Programm

In der Regel berechnet man nun den optimalen Code in mehreren Pässen:

Pass 1: Tree Matching

Berechnung aller möglichen Reduktionssymbole für alle Knoten *u*

(Berechnung von *Sym(u), Pat(x,u)*)

Pass 2: Codeselektion unter Registerbeschränkung

Berechnung der Kosten kostenminimaler Reduktionen unter Registerbeschränkung

➔ liefert *cost(x,j,u), best_pattern(x,j,u)* für alle *u*

Pass 3: Codegenerierung

Berechnung einer kostenminimalen Codesequenz für *k* Register auf der Basis von *cost(x,j,u), best_pattern(x,j,u)*

Das dynamische Programm -- Pass 1

Der erste Pass lässt sich naiv ganz leicht lösen:

```
Für alle Knoten u von den Blättern zur Wurzel
do      if u Blatt then Sym(u) := { op(u) } else Sym(u) := { };
        Für alle x do  Pat(x,u) = { };
          Für alle Regeln x ← T‘
              do if match(T‘, u ) then Sym(u) ∪ = {x};
                                       Pat(x,u) ∪ = {T‘};
              od
od;
```

```
match(T‘,u)
{ if u ist ein Blatt then return (T‘.root ist Blatt) and (op(T‘.root) ∈ Sym(u))
        else b = (op(T‘.root) == op(u)) ;
            for i = 1 to outdegree(u) while b
            do b = ( b and match(T‘(T‘.root.child(i)), u.child(i)) ) od;
 return b;
}
```

Das dynamische Programm -- Pass 1

Diese naive Lösung hat Laufzeit

Zahl der Knoten von *T* * Summe der Größen der Pattern *T‘*

weil man an jedem Knoten jedes Baumpattern anlegen muss. Dies liefert einen in der Praxis sehr hohen Zeitfaktor, weil die Maschinengrammatik i.A. sehr viele Regeln hat.

Verbesserung:

Man führe das Tree-Matching auf ein String Matching zurück, in dem man jedes Baumpattern durch die Menge seiner Operatorsymbole, gelesen von der Wurzel zu jedem Blatt, als Stringmenge auffasst. Matching von Stringmengen kann durch einen endlichen Automaten, der zur Stringmenge konstruiert wird, in Linearzeit auf einem String gelöst werden (Aho Korasik Algorithmus).

Wir wollen diese Technik hier nicht weiter vertiefen und verweisen dazu auf Compilerbauliteratur.

Das dynamische Programm -- Pass 2

Wir sind nun in der Lage, die Kostenfunktion zu berechnen. Wir nehmen dazu an, dass die Zielmaschine höchstens *M* freie Register zur Codeerzeugung für einen Operatorbaum bereitstellt.

Wir setzen ferner voraus, dass wir im Pass 1 auch für jedes *T‘ ∈ Pat(x,u)* eine Liste *leaves(T‘,u)* von Knoten in *T* angelegt haben, die bei Reduktion auf *x* zu Blättern des Patterns *T‘* mit Wurzelmarkierung *x* korrespondieren.

```
Für jeden Knoten u von den Blättern zur Wurzel
do for j = 0 to M
    do    Foreach x ∈ Sym(u)
          do best_match(x,j,u) = nil;
             if Pat(x,u) = {} then cost(x,j,u) = 0 else cost(x,j,u) = MAXINT;
             if  (x ist Registersymbol)  and  (j == 0) then continue;
             foreach T‘ ∈ Pat(x,u)
             do Sei leaves(T‘,u) = (u1,…,us,us+1,…,us+k) und seien
                      (u1,…,us) die Blätter, die Speicheroperanden entsprechen.
                cost = cost(T‘) + cost(u1.symb,j,u1 ) + … + cost(us.symb,j,us )
                      + cost(us+1.symb,j,us+1 ) + … + cost(us+k.symb,j-k+1,us+k );
                if cost < cost(x,j,u) then cost(x,j,u) = cost, best_match(x,j,u) = T‘;
             od;    ….
```

Das dynamische Programm -- Pass 2

Bemerkung:

in der Zeile

„Sei leaves(T‘,u) = (u_1,…,u_s,u_{s+1},…,u_{s+k}) und seien
(u_1,…,u_s) die Blätter, die Speicheroperanden entsprechen.
cost = cost(T‘) + cost(u_1.symb,j,u_1) + … + cost(u_s.symb,j,u_s)
+ cost(u_{s+1}.symb,j,u_{s+1}) + … + cost(u_{s+k}.symb,j-k+1,u_{s+k});“

berechnen wir die Kosten nur für eine feste Reihenfolge der Registeroperanden, nämlich die, die durch die Liste *leaves(T‘,u)* vorgegeben ist. Will man das wirkliche Optimum finden, muss man über alle Reihenfolgen von Unterbäumen, die auf Register reduziert werden, die Kosten betrachten und dazu das Kostenminimum berechnen, da der Erste *j*, der Letzte nur noch *j-k+1* Register zur Verfügung hat.

Da die Baumpattern i.A. nicht viele Blätter mit Registeroperanden haben, ist dies durchaus praktikabel. Man muss sich neben dem besten Match aber auch die Reihenfolge speichern.

Das dynamische Programm -- Pass 3

Die Codegenerierung ist nun denkbar einfach:

Sei *n* die Zahl der freien Register Ri_1, …, Ri_n :

Suche das x∈ Sym(u), mit cost(x,n,u) minimal.

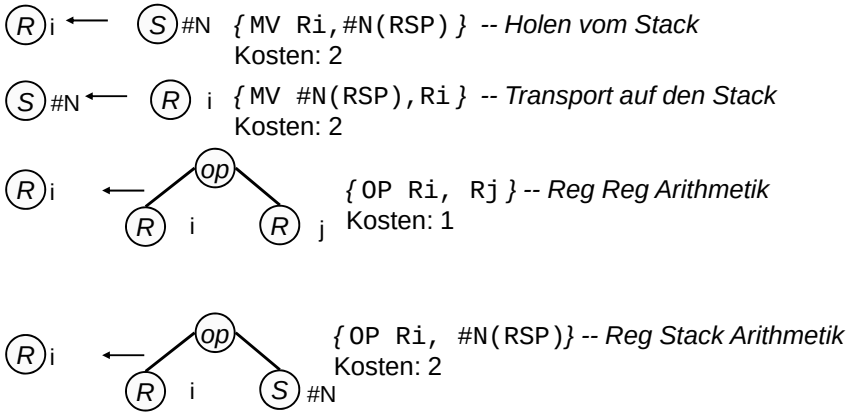
Generate_code(u,x,n);

Mit

```
Generate_code(u,x,n) {
    if (u Blatt and Best_match(x,n,u) == nil) then return;
    T‘ = best_match(x,n,u);
    Sei wieder leaves(T‘,u) = (u1,…,us,us+1,…,us+k):
    For i = 1 to s do Generate_code(ui,ui.symb,n) od;
    For i = 1 to k do Generate_code(us+i, us+i.symb, n-i+1) od;
    If (x ist Registersymbol) then setze Registerattribut auf in;
    Code << Aktion(x ← T‘);
}
```

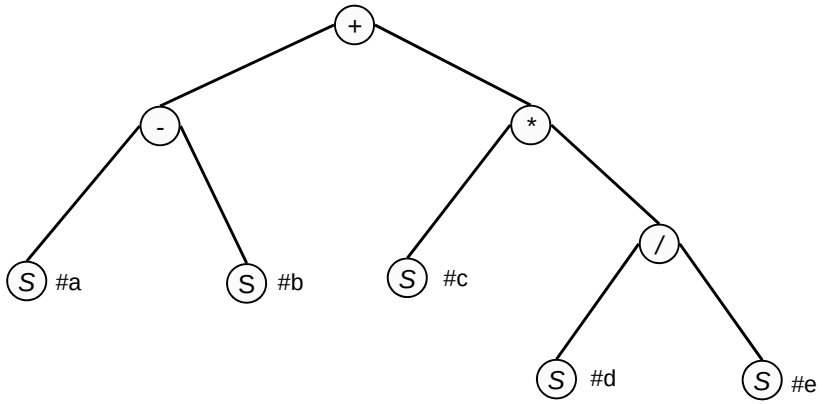
Beispiel: Codeselektion

Um zu demonstrieren, wie das dynamische Programm läuft, betrachten wir folgende Teilmenge von Transformationsregeln, die wir zur Codeselektion eines Ausdrucks ausschließlich benutzen wollen:



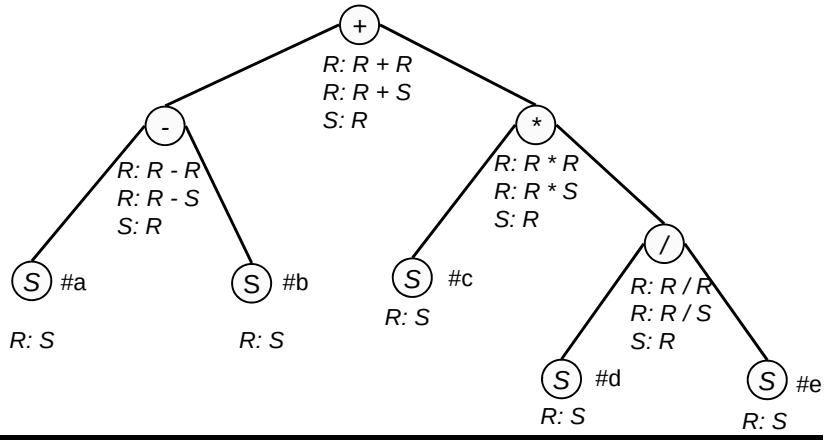
Beispiel: Codeselektion -- der Baum

Wir betrachten nun folgenden Operatorbaum zu (a-b)+c*(d/e), wobei a, b, c, d und e temporäre Variablen auf dem Stack sind:



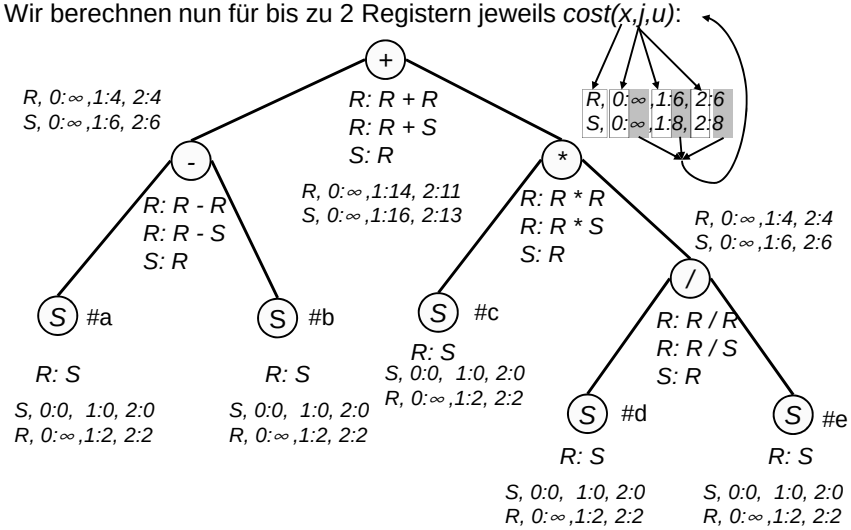
Beispiel: Codeselektion -- Pass 1

Wir berechnen nun bottom up die Symbolmengen, und die zugehörigen Baummuster:



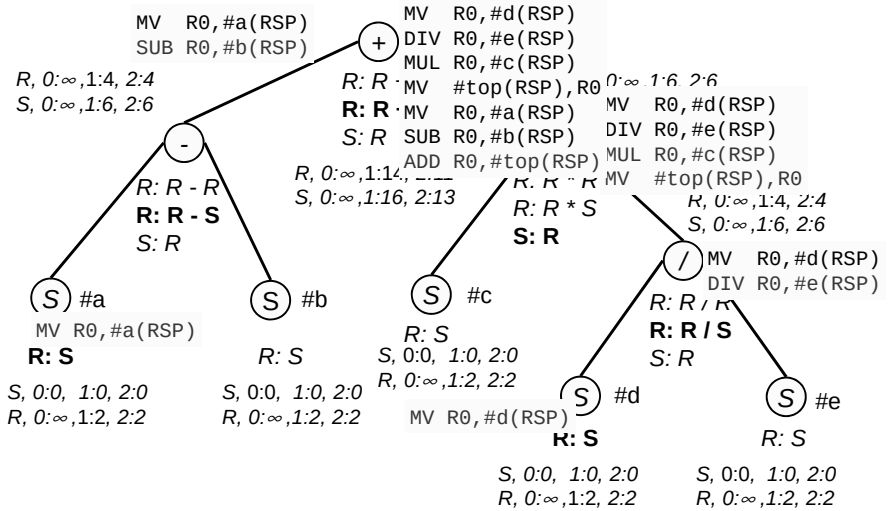
Beispiel: Codeselektion -- Pass 2

Wir berechnen nun für bis zu 2 Registern jeweils *cost(x,j,u)*:



Beispiel: Codegenerierung -- Pass 3

Wir wollen nun optimalen Code für 1 Register generieren:



Beispiel: Codegenerierung -- Pass 3

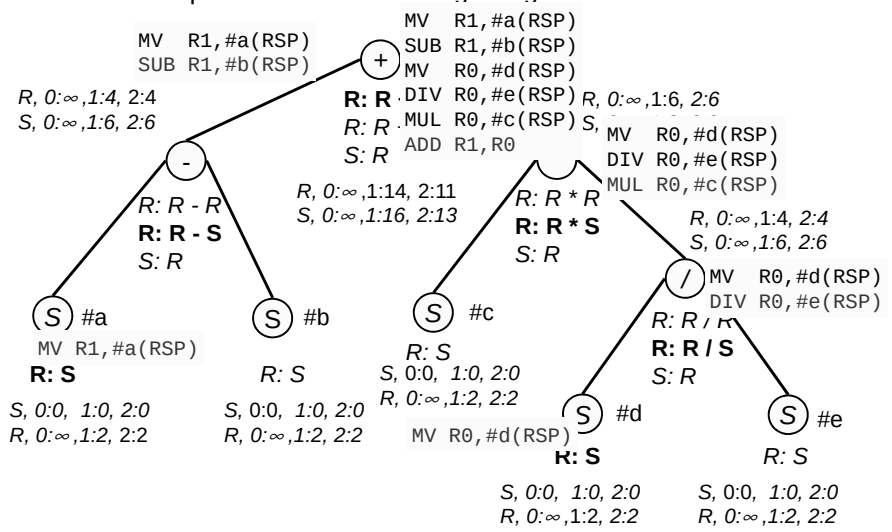
Ein optimaler Code für 1 Register lautet also:

```
MV  R0,#d(RSP)     -- 2
DIV R0,#e(RSP)     -- 2
MUL R0,#c(RSP)     -- 2
MV  #top(RSP),R0   -- 2
MV  R0,#a(RSP)     -- 2
SUB R0,#b(RSP)     -- 2
ADD R0,#top(RSP)   -- 2
```

Kosten 14

Beispiel: Codegenerierung -- Pass 3

Wir wollen nun optimalen Code für 2 Register generieren:



Beispiel: Codegenerierung -- Pass 3

Ein optimaler Code für 2 Register lautet also:

```
MV  R1,#a(RSP)     -- 2
SUB R1,#b(RSP)     -- 2
MV  R0,#d(RSP)     -- 2
DIV R0,#e(RSP)     -- 2
MUL R0,#c(RSP)     -- 2
ADD R1,R0                -- 1
Kosten                   11
```