Einfache Codegenerierung

- Idee: abstrakter Syntaxbaum → Zwischencodebaum
  - Transformation knotenweise
  - ein Übersetzungsschema pro Knotentyp
  - Zwischencode spezifisch für ein Maschinenmodell
  - abschließend Linearisierung des Zwischencodebaums
- zwei Maschinenmodelle:
  - Stackmaschine (Ursprung: PDP-11/VAX)
  - Registermaschine (Urspung: IBM 360/370)
- Effizienz:
  - Registermaschine besser als Stackmaschine
  - grobe Regel für Registermaschine: noch Effizienzsteigerung um ca. Faktor 3 möglich durch *fortgeschrittene* Codegenerierung

Stackmaschine

- Modell: Laufzeitstack als einziger Speicher
- Implementierung:
  - als erweiterbares Array (d.h. indizierbar)
  - zwei Zeiger (Abb. 4.20)
    - SP: zeigt auf den Top of Stack
    - BP: zeigt auf den Anfang der lokalen Variablen
- Befehlssatz (Abb. 4.21)
- Code-Beispiel: `p := p+5`

```
Push_Local  #p   push value of #p-th local
Push_Const   5   push value of 5
Add_Top2         add two elements
Store_Local #p   pop and store result into
                 #p-th local
```

Stackmaschine / Befehlssatz

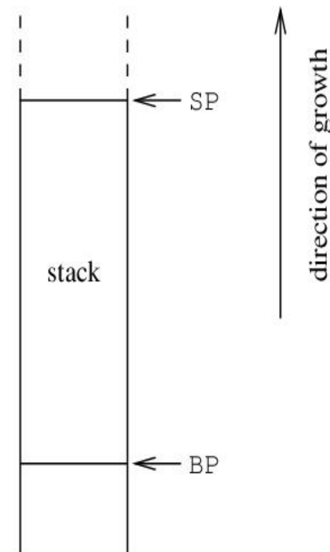


Figure 4.20 Data administration in a simple stack machine.

Instruction		Actions
`Push_Const`	*c*	`SP:=SP+1; stack[SP]:=`*c*`;`
`Push_Local`	*i*	`SP:=SP+1; stack[SP]:=stack[BP+`*i*`];`
`Store_Local`	*i*	`stack[BP+`*i*`]:=stack[SP]; SP:=SP-1;`
`Add_Top2`		`stack[SP-1]:=stack[SP-1]+stack[SP]; SP:=SP-1;`
`Subtr_Top2`		`stack[SP-1]:=stack[SP-1]-stack[SP]; SP:=SP-1;`
`Mult_Top2`		`stack[SP-1]:=stack[SP-1]*stack[SP]; SP:=SP-1;`

Figure 4.21 Stack machine instructions.

Registermaschine (1)

- reines Modell:
  - Rechenperationen nur auf Registern
    - Praxis (CISC): auch Speicherzelle als Operand/Ergebnis
  - Hauptspeicher zur Auslagerung momentan nicht gebrauchter Zwischenergebnisse
- zwei Befehlsarten:
  - Transfer zwischen Hauptspeicher und Registern
  - Operationen auf Registerinhalten

Registermaschine (2)

- zweigeteilte Befehlsnamen:

  1. Teil: spezifiziert Operation
     - `Load`, `Add`, `Subtr`, `Mult`: Register als Ziel
     - `Store`: Hauptspeicher als Ziel

  2. Teil: spezifiziert Operanden
     - `Const`: Konstante
     - `Reg`: Register
     - `Mem`: Speicherzelle

- Code-Beispiel: `p:=p+5`

```
Load_Mem     p, R1    (Dereferenzierung von p implizit)
Load_Const   5, R2
Add_Reg      R2,R1    (rechts: Ziel=1.Operand, links: 2. Op.)
Store_Reg    R1,p
```

Registermaschine / Befehlssatz

Instruction		Actions
Load_Const	c, R_n	$R_n := c;$
Load_Mem	x, R_n	$R_n := x;$
Store_Reg	R_n, x	$x := R_n;$
Add_Reg	R_m, R_n	$R_n := R_n + R_m;$
Subtr_Reg	R_m, R_n	$R_n := R_n - R_m;$
Mult_Reg	R_m, R_n	$R_n := R_n * R_m;$

Figure 4.22 Register machine instructions.

Rewriting für eine Stackmaschine

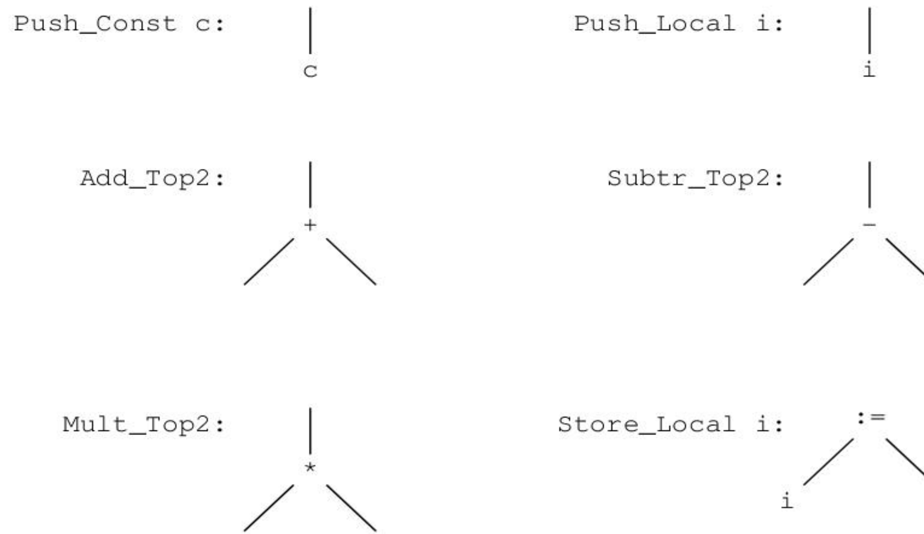


Figure 4.24 The abstract syntax trees for the stack machine instructions.

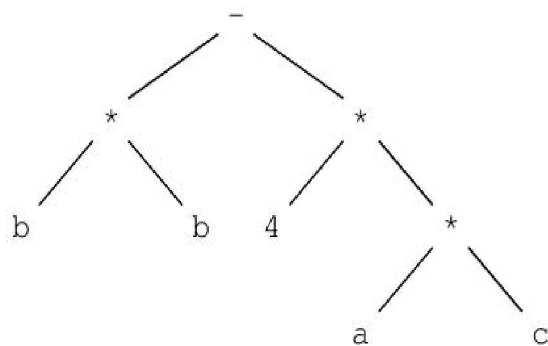


Figure 4.23 The abstract syntax tree for `b*b - 4*(a*c)`.

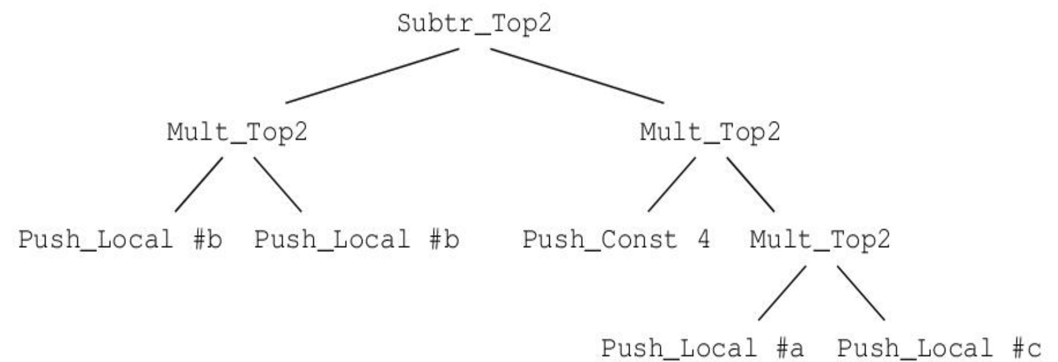


Figure 4.25 The abstract syntax tree for `b*b - 4*(a*c)` rewritten.

Codegenerierung für eine Stackmaschine (1)

- Bedingung zur Laufzeit:
  - Operanden der aktuellen Operation müssen oben auf dem Stack stehen
- Herstellung der Bedingung:
  - Stackverbrauch jedes Teilbaums wird zuerst vollständig zu einem Wert reduziert, bevor nächster Teilbaum oder Wurzeloperation ausgewertet wird
  - jede Rechenoperation poppt Operanden vom Stack und pusht das Ergebnis
- Umsetzung bei der Codegenerierung:
  - Postorder-Durchlauf des abstrakten Syntaxbaums

Codegenerierung für eine Stackmaschine (2)

```
PROCEDURE Generate code (Node):
    SELECT Node .type:
        CASE Constant type:   Emit ("Push_Const" Node .value);
        CASE LocalVar type:   Emit ("Push_Local" Node .number);
        CASE StoreLocal type: Emit ("Store_Local" Node .number);
        CASE Add type:
            Generate code (Node .left); Generate code (Node .right);
            Emit ("Add_Top2");
        CASE Subtract type:
            Generate code (Node .left); Generate code (Node .right);
            Emit ("Subtr_Top2");
        CASE Multiply type:
            Generate code (Node .left); Generate code (Node .right);
            Emit ("Mult_Top2");
```

Figure 4.26 Depth-first code generation for a stack machine.

Stackkonfigurationen

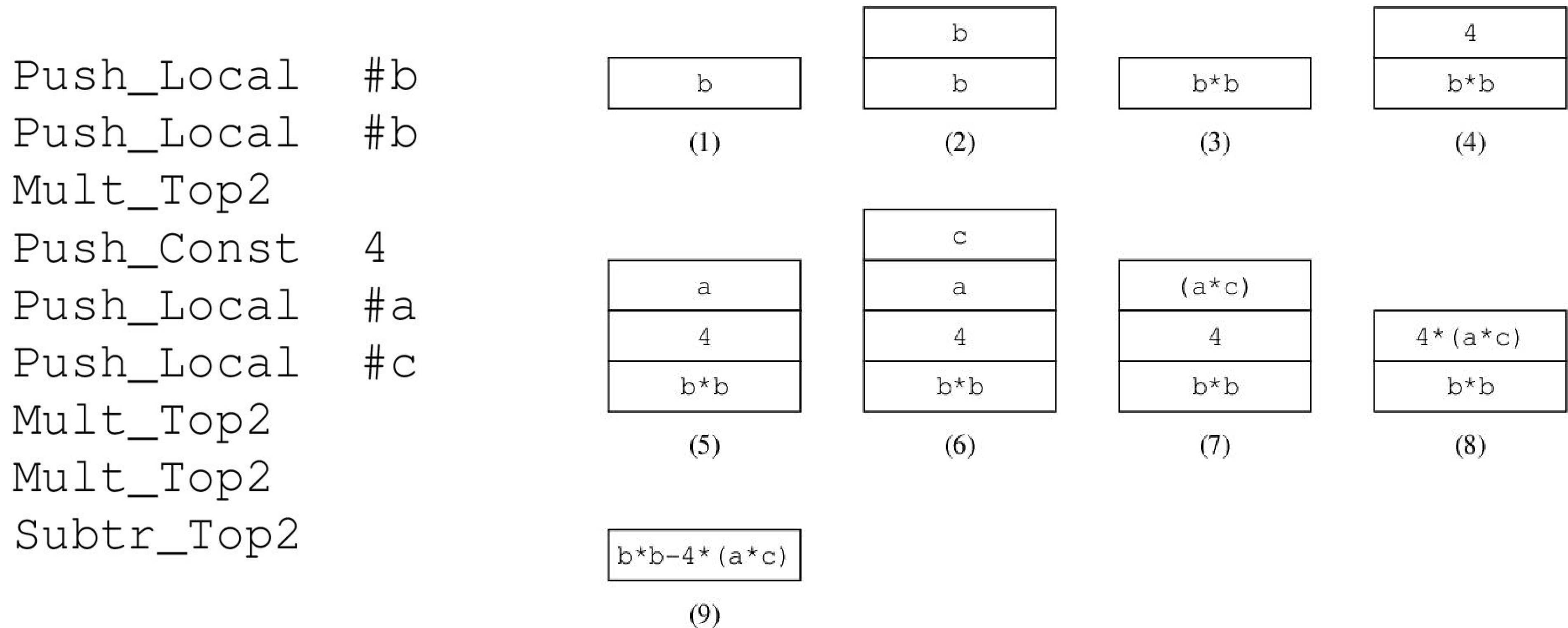
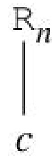


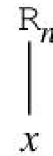
Figure 4.27 Successive stack configurations for `b*b - 4*(a*c)`.

Rewriting für eine Registermaschine

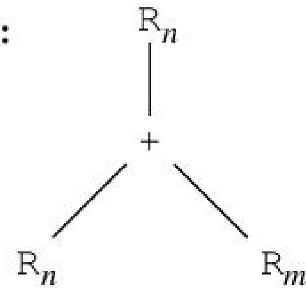
`Load_Const` c, R_n:



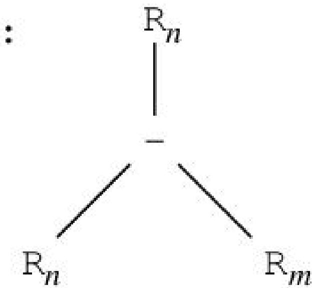
`Load_Mem` x, R_n:



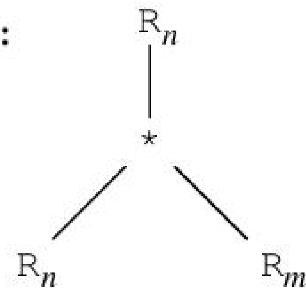
`Add_Reg` R_m, R_n:



`Subtr_Reg` R_m, R_n:



`Mult_Reg` R_m, R_n:



`Store_Reg` R_n, `x`:

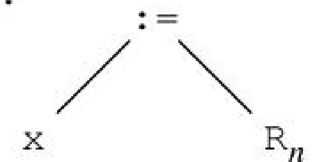


Figure 4.28 The abstract syntax trees for the register machine instructions.

Codegenerierung für eine Registermaschine

analog zur Stackmaschine (Postorder-Durchlauf des AST)

```
PROCEDURE Generate code (Node, a register Target, a register set Aux):
    SELECT Node .type:
        CASE Constant type:
            Emit ("Load_Const " Node .value ",R" Target);
        CASE Variable type:
            Emit ("Load_Mem " Node .address ",R" Target);
        CASE ...
        CASE Add type:
            Generate code (Node .left, Target, Aux);
            SET Target 2 TO An arbitrary element of Aux;
            SET Aux 2 TO Aux \ Target 2;
                // the \ denotes the set difference operation
            Generate code (Node .right, Target 2, Aux 2);
            Emit ("Add_Reg R" Target 2 ",R" Target);
        CASE ...
```

Figure 4.29 Simple code generation with register allocation.

Codegenerierung für Registermaschinen

- Baumkomponenten der Maschinenbefehle: (Abb. 4.28)
- Vorgaben: (vorläufig)
  - Ausgaberegister immer `R1`, alle Register außer `R1` Hilfsregister
  - Ausgaberegister muss im folgenden Befehl Eingaberegister sein
  - Eingaberegister eines festen Befehls unterschiedlich
- Registervergabealgorithmus (Abb. 4.29)
  - linker Teilbaum
    - kann alle Hilfsregister benutzen
    - speichert Ergebnis ins Ausgangsregister
  - rechter Teilbaum
    - ein Hilfsregister weniger (Ergebnis linker Teilbaum)
- serielle Registervergabe (Abb. 4.30)
  - arbeitet mit Registerstack (mit Zähler verwaltet)

```
PROCEDURE Generate code (Node, a register number Target):
    SELECT Node .type:
        CASE Constant type:
            Emit ("Load_Const " Node .value ",R" Target);
        CASE Variable type:
            Emit ("Load_Mem " Node .address ",R" Target);
        CASE ...
        CASE Add type:
            Generate code (Node .left, Target);
            Generate code (Node .right, Target+1);
            Emit ("Add_Reg R" Target+1 ",R" Target);
        CASE ...
```

Figure 4.30 Simple code generation with register numbering.

```
Load_Mem     b,R1
Load_Mem     b,R2
Mult_Reg     R2,R1
Load_Const   4,R2
Load_Mem     a,R3
Load_Mem     c,R4
Mult_Reg     R4,R3
Mult_Reg     R3,R2
Subtr_Reg    R2,R1
```

Figure 4.31 Register machine code for the expression `b*b - 4*(a*c)`.

Registerinhalte

```
Load_Mem     b,R1
Load_Mem     b,R2
Mult_Reg     R2,R1
Load_Const   4,R2
Load_Mem     a,R3
Load_Mem     c,R4
Mult_Reg     R4,R3
Mult_Reg     R3,R2
Subtr_Reg    R2,R1
```

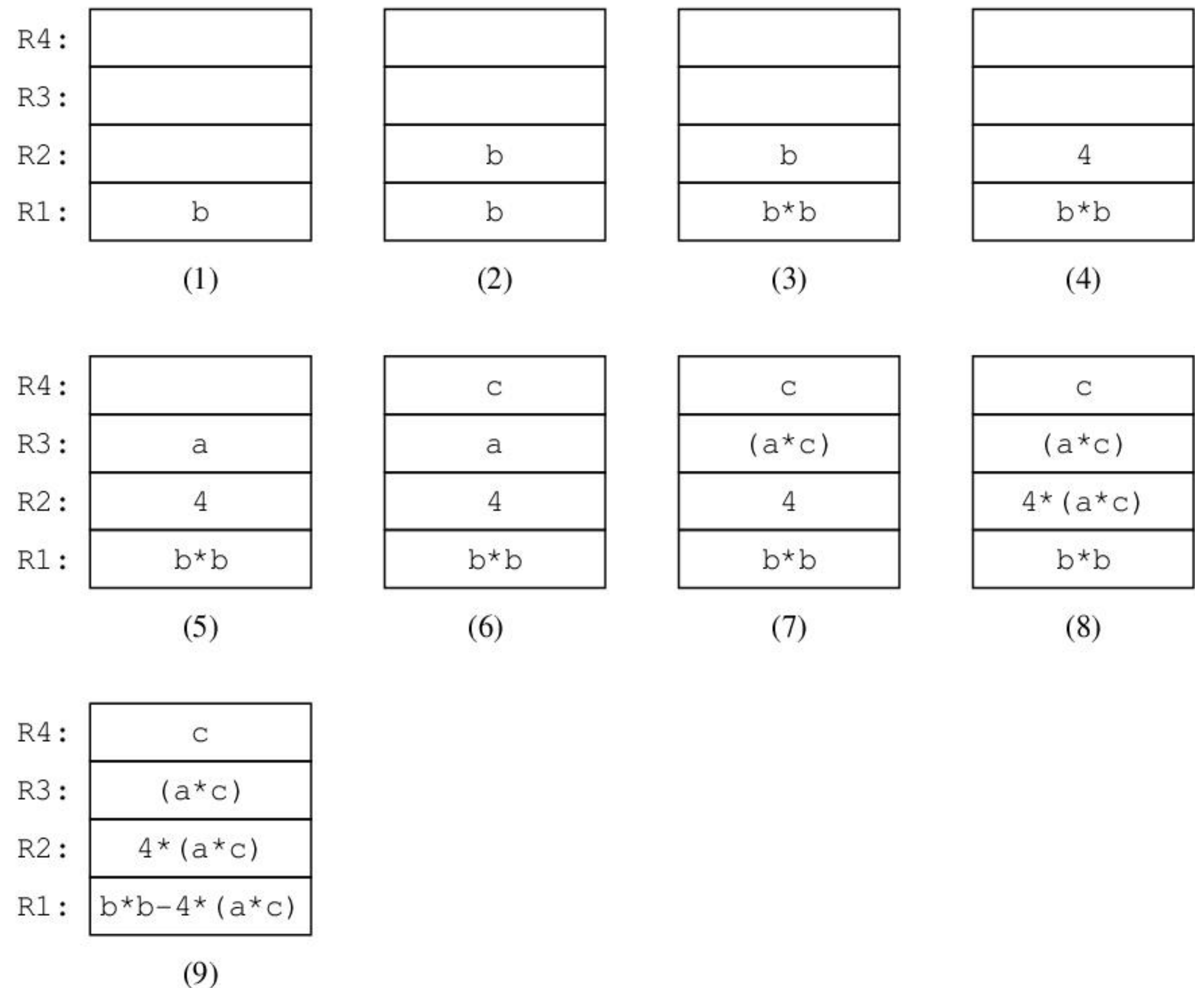


Figure 4.32 Successive register contents for `b*b - 4*(a*c)`.

Maschinenvergleich: Stack- und Registerinhalte

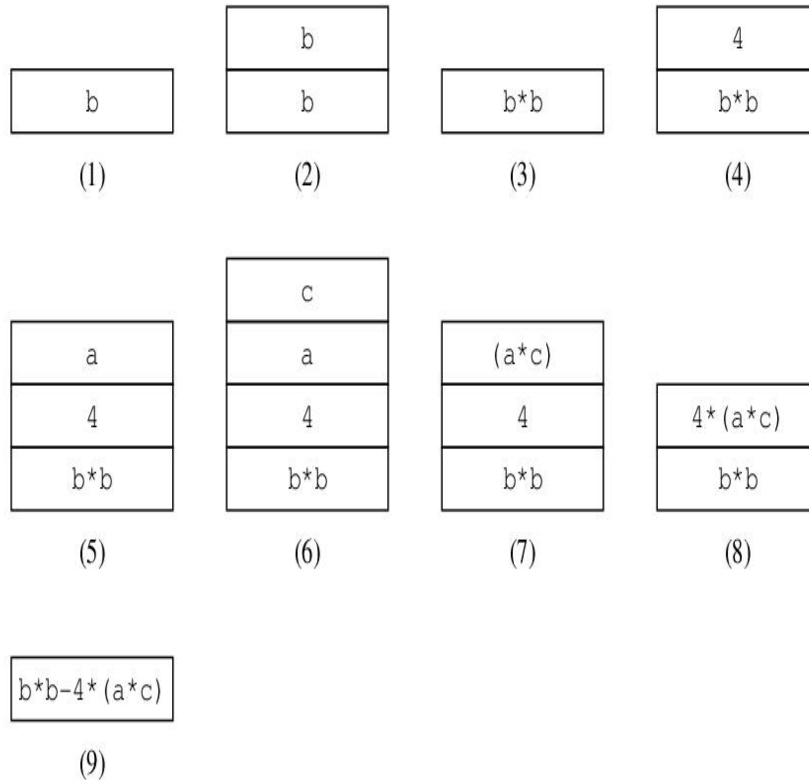


Figure 4.27 Successive stack configurations for `b*b - 4*(a*c)`.

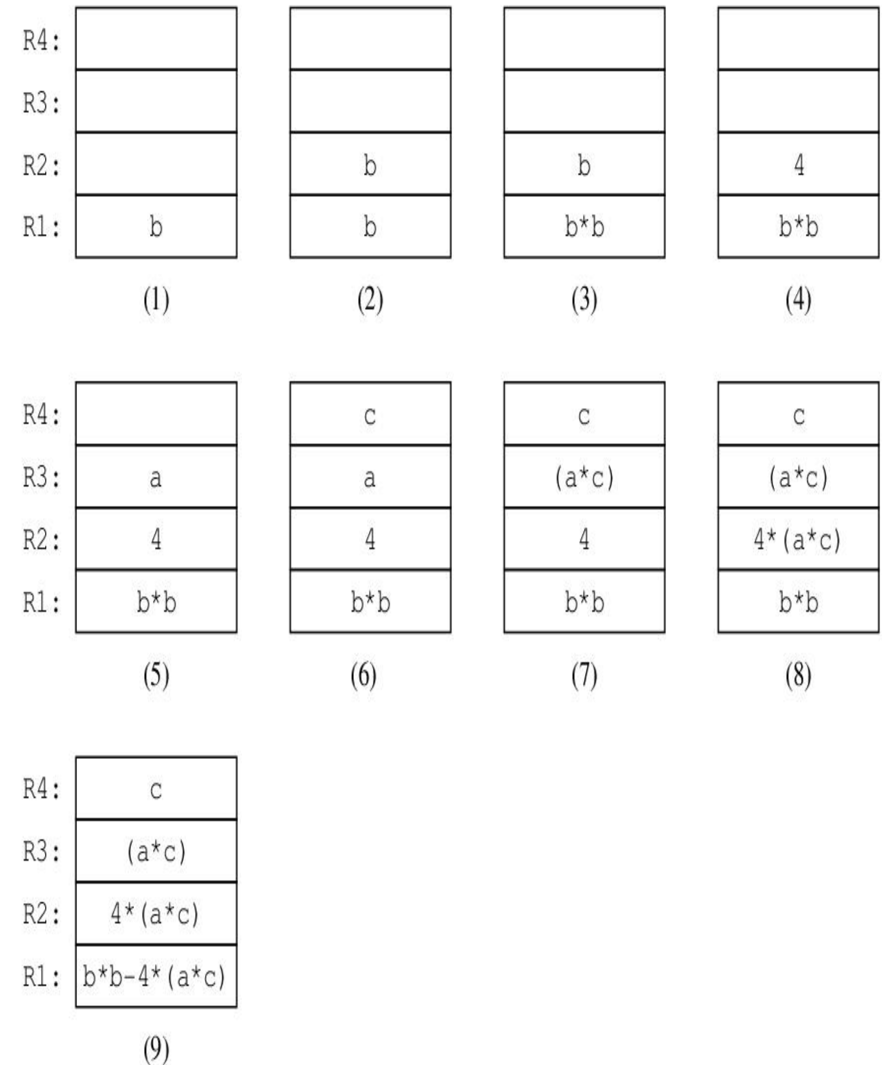


Figure 4.32 Successive register contents for `b*b - 4*(a*c)`.

4 Register gebraucht:

```
Load_Mem      b,R1
Load_Mem      b,R2
Mult_Reg      R2,R1
Load_Const    4,R2
Load_Mem      a,R3
Load_Mem      c,R4
Mult_Reg      R4,R3
Mult_Reg      R3,R2
Subtr_Reg     R2,R1
```

Figure 4.31 Register machine code for the expression `b*b - 4*(a*c)`.

---

3 Register reichen:

```
Load_Mem      b,R1
Load_Mem      b,R2
Mult_Reg      R2,R1
Load_Mem      a,R2
Load_Mem      c,R3
Mult_Reg      R3,R2
Load_Const    4,R3
Mult_Reg      R3,R2
Subtr_Reg     R2,R1
```

Figure 4.33 Weighted register machine code for the expression `b*b - 4*(a*c)`.

Gewichtete Registervergabe (1)

- Ineffizienz:
  - 4 Register gebraucht
  - nur 3 nötig, wenn man zuerst den rechten Baum auswertet (Abb. 4.33, vgl. Abb. 4.31)
- Idee:
  - werte zuerst den Teilbaum mit dem höheren Registerbedarf aus
- Registerbedarf eines Teilbaums durch Gewichtsfunktion w
  - statisch bottom-up berechenbar:

    (i) $w(\texttt{Leaf}) = 1$

    (ii) $w(l)=w(r) \Rightarrow w(\texttt{Branch}(l,r)) = 1+w(l)$

    (iii) $w(l)\neq w(r) \Rightarrow w(\texttt{Branch}(l,r)) = \max(w(l),w(r))$
  - Programm (Abb. 4.34)

```
FUNCTION Weight of (Node) RETURNING an integer:
    SELECT Node .type:
        CASE Constant type: RETURN 1;
        CASE Variable type: RETURN 1;
        CASE ...
        CASE Add type:
            SET Required left TO Weight of (Node .left);
            SET Required right TO Weight of (Node .right);
            IF Required left > Required right: RETURN Required left;
            IF Required left < Required right: RETURN Required right;
            // Required left = Required right
            RETURN Required left + 1;
        CASE ...
```

Figure 4.34 Register requirements (weight) of a node.

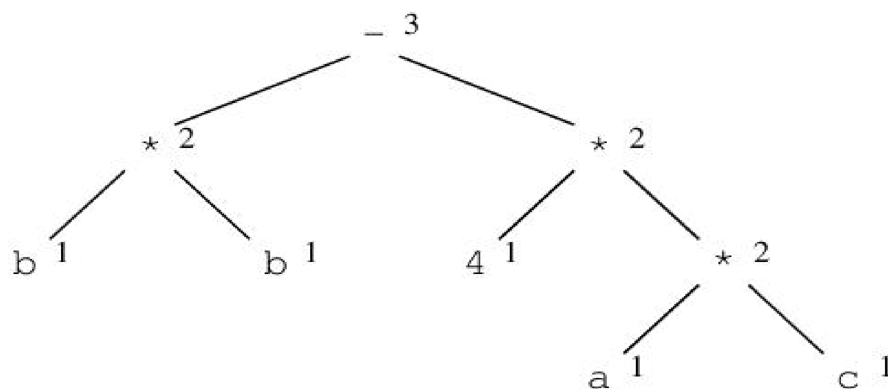


Figure 4.35 AST for `b*b - 4*(a*c)` with register weights.

Gewichtete Registervergabe (2)

- Erweiterung auf n Operanden (z.B. Routinenparameter):
  - sortiere Teilbäume nach absteigenden Gewichten $E_1,\ldots,E_n$
  - gesamter Registerbedarf: $(\max i: 0<i\leq n: E_i+i-1)$

    ($i-1$: Bedarf für die vorher ermittelten Zwischenwerte)

- Beispiel: Routine mit drei Parametern
  - Annahmen:
    - bereitgestellt in Registern `R1`, `R2`, `R3`
    - Ausdrucksbäume haben Gewichte 1, 4, 2
    - also: berechne erst Inhalt von `R2`, dann `R3`, dann `R1`
  - Bedarfsberechnung pro Parameter:

(a) Register-ID	2	3	1
(b) Gewicht des Baums	4	2	1
(c) besetzte Register vor Auswertung:	0	1	2
(d) Summe ((b)+(c))	4	3	3
(e) Maximum:	4		

Gewichtete Registervergabe (3)

- kein Registerstack: Register werden nicht in aufsteigender Reihenfolge benutzt
- muss mit Registermengen arbeiten

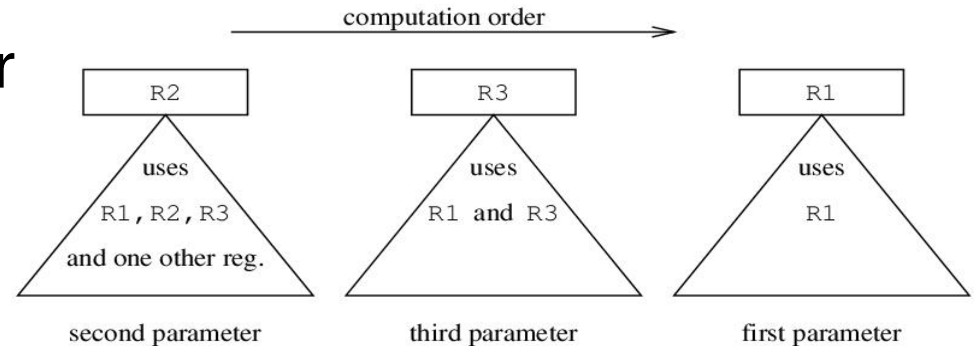


Figure 4.36 Evaluation order of three parameter trees.

```
PROCEDURE Generate code (Node, a register Target, a register set Aux):
    SELECT Node .type:
        CASE Constant type:
            Emit ("Load_Const " Node .value ",R" Target);
        CASE Variable type:
            Emit ("Load_Mem " Node .address ",R" Target);
        CASE ...
        CASE Add type:
            Generate code (Node .left, Target, Aux);
            SET Target 2 TO An arbitrary element of Aux;
            SET Aux 2 TO Aux \ Target 2;
                // the \ denotes the set difference operation
            Generate code (Node .right, Target 2, Aux 2);
            Emit ("Add_Reg R" Target 2 ",R" Target);
        CASE ...
```

Figure 4.29 Simple code generation with register allocation.

Register-Spilling (1)

- Anzahl verfügbarer Register ist kleiner als nötig:
  (a) Aufteilung des Baums in Teilbäume
  (b) temporäre Speicherung der Teilbaumergebnisse im Hauptspeicher ⇒ alle Register frei für andere Teilbäume
- Ziel
  - möglichst wenig Auslagerungen
  - Strategie: bearbeite zuerst Teilbaum mit maximalem erfüllbaren Registerbedarf

Register-Spilling (2)

- Algorithmus: (Abb. 4.37)
  - Menge verfügbarer Register und Menge von Hilfsvariablen
  - iterative Hauptroutine, wiederhole ...
    1. Hilfsfunktion: suche Unterbaum T maximal erfüllbaren Bedarfs
       - falls Registerbedarf des Teilbaums gedeckt: gefunden
       - sonst: suche in einem Teilbaum, der mehr Register braucht
    2. generiere Code für T mit Ergebnis im Ausgaberegister
    3. falls Wurzel von T nicht der Wurzel des Gesamtausdrucks
       - erzeuge Code, der Ergebnis von T in Speicherzelle S kopiert
       - ersetze T im Gesamtbaum durch eine Referenz auf S

    ... solange ein Unterbaum mit nicht erfüllbarem Registerbedarf existiert
- Bsp. für generierten Code: (Abb. 4.38, vgl. Abb. 4.33)
  - Inhalt von Register `R3` wird in `T1` gespeichert
  - Änderung der Auswertungsreihenfolge

```
PROCEDURE Generate code for large trees (Node, Target register):
    SET Auxiliary register set TO
        Available register set \ Target register;

    WHILE Node /= No node:
        Compute the weights of all nodes of the tree of Node;
        SET Tree node TO Maximal non_large tree (Node);
        Generate code
            (Tree node, Target register, Auxiliary register set);

        IF Tree node /= Node:
            SET Temporary location TO Next free temporary location();
            Emit ("Store R" Target register ",T" Temporary location);
            Replace Tree node by a reference to Temporary location;
            Return any temporary locations in the tree of Tree node
                to the pool of free temporary locations;
        ELSE Tree node = Node:
            Return any temporary locations in the tree of Node
                to the pool of free temporary locations;
            SET Node TO No node;

FUNCTION Maximal non_large tree (Node) RETURNING a node:
    IF Node .weight <= Size of Auxiliary register set: RETURN Node;
    IF Node .left .weight > Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .left);
    ELSE Node .right .weight >= Size of Auxiliary register set:
        RETURN Maximal non_large tree (Node .right);
```

Figure 4.37 Code generation for large trees.

Register-Spilling (3), Beispiel mit 2 Registern

4*(a*c)→ T1

```
Load_Mem      a,R1
Load_Mem      c,R2
Mult_Reg      R2,R1
Load_Const    4,R2
Mult_Reg      R2,R1
Store_Reg     R1,T1
```

Auslagerung

b*b → R1

```
Load_Mem      b,R1
Load_Mem      b,R2
Mult_Reg      R2,R1
```

R1, R2 frei

```
Load_Mem      T1,R2
Subtr_Reg     R2,R1
```

Wiederherstellung

Figure 4.38 Code generated for `b*b - 4*(a*c)` with only 2 registers.

Register-Spilling (4)

- Anzahl verfügbarer Register ist kleiner als nötig:
  (a) Aufteilung des Baums in Teilbäume
  (b) Temporäre Speicherung der Teilbaumergebnisse im Hauptspeicher ⇒ alle Register frei für andere Teilbäume
- Ziel
  - möglichst wenig Auslagerungen
  - Strategie: bearbeite zuerst Teilbaum mit maximalem erfüllbaren Registerbedarf
- Erfahrungswerte
  - handgeschriebener Code kommt oft mit 4-5 Registern aus
  - Reservierung von 4-5 Registern auch gute Strategie für Ausdrücke, die mehr brauchen (z.B. automatisch erzeugte)

Register/Speicher-Operationen (1)

- Beispieloperation: `Add_Mem X, R1`
  addiere den Inhalt der Speicherstelle `X` zum Inhalt des Registers `R1`
- Folgen
  - es werden weniger Register gebraucht
  - Änderung der Gewichtsberechnung im AST
  - Beispiel:

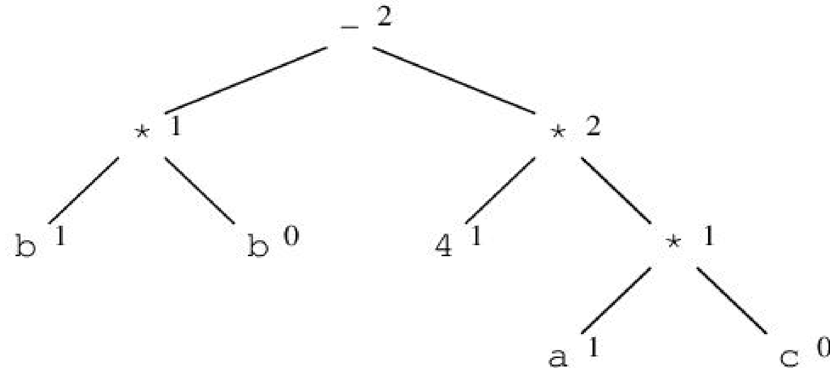


Figure 4.39 Register-weighted tree for a memory-register machine.

Register/Speicher-Operationen (2)

Bsp.: Code für Ausdruck `b*b-4*(a*c)`:

```
Load_Const   4,R2
Load_Mem     a,R1
Mult_Mem     c,R1
Mult_Reg     R1,R2
Load_Mem     b,R1
Mult_Mem     b,R1
Subtr_Reg    R2,R1
```

Figure 4.40 Code for the register-weighted tree for a memory-register machine.

Beobachtung:
zwei Register reichen ohne Auslagerung oder Anwendung von Kommutativität

Registerbelegung für eine Prozedur

- Reservierung eine bestimmten Anzahl für lokale Variablen
  - vorzugsweise: häufig verwendete Variablen, z.B. Schleifenzähler, bestimmt durch Analyse oder Profiling
  - Compiler-Direktive in der Sprache C: `register int x`
    gibt Compiler die Empfehlung, `x` in einem Register zu speichern
- generelles Problem der Prozedur-weiten Belegung
  - Register verschwendet wo die Variable nicht verwendet wird
  - Lösung durch Aufteilung der Bereiche (später: Interferenzgraph)

Profiling

- Ziel: Gewinnung statistischer Informationen über das Programm
- dynamisches Profiling
  - Anreicherung des Programms mit Code zum Sammeln der gewünschten Information (Instrumentierung)
  - viele Ausführungen des Programms mit repräsentativen Eingabedaten
- statisches Profiling
  - Analyse des Kontrollflussgraphen
    - Anwendung von Flussgleichungen (Kirchhoffsche Gesetze)
  - Annahmen über Verzweigungswahrscheinlichkeiten
    - `if-then-else`: 70% für Wahl des `then`-Teils
    - Schleifen: 90% für weiteren Durchlauf
  - Lösen des Gleichungssystems

Symbolische Interpretation

- in SIPS I verwendet zur Programmanalyse:
  - lässige/einfache Alternative zur Methode “abstrakte Interpretation”
  - symbolische Interpretation benutzt zur Berechnung von Bedingungen, die bereits zur Übersetzungszeit bekannt sind
    - Bsp.: Variable ist definiert / hat bestimmten Wert
  - Verfolgen des Kontrollflusses, Aufteilung bei Verzweigungen
  - Stack (Environment) von Bedingungen für lokale Variablen
  - Approximation, partielle Information über das Programm
- hier, für die Verwaltung von Variablen:
  - exakte Behandlung
    - zur Compilezeit, soweit möglich, z.B. `x=3`
    - generiere Code für die Laufzeit sonst, z.B. `x=read_real()`
  - REGVAR-Deskriptor
    - enthält Informationen über Register und Variablen

REGVAR-Deskriptor

- drei Teile, die entlang des/der Kontrollflusspfade(s) aktualisiert werden:

1. Register-Deskriptor-Tabelle
   - adressiert mit Registernummern
   - Eintrag enthält Information, was das Register enthält
2. Variablen-Deskriptor-Tabelle
   - adressiert mit Variablennamen
   - Eintrag enthält Information, wo die Variable gespeichert ist
3. Menge freier Register