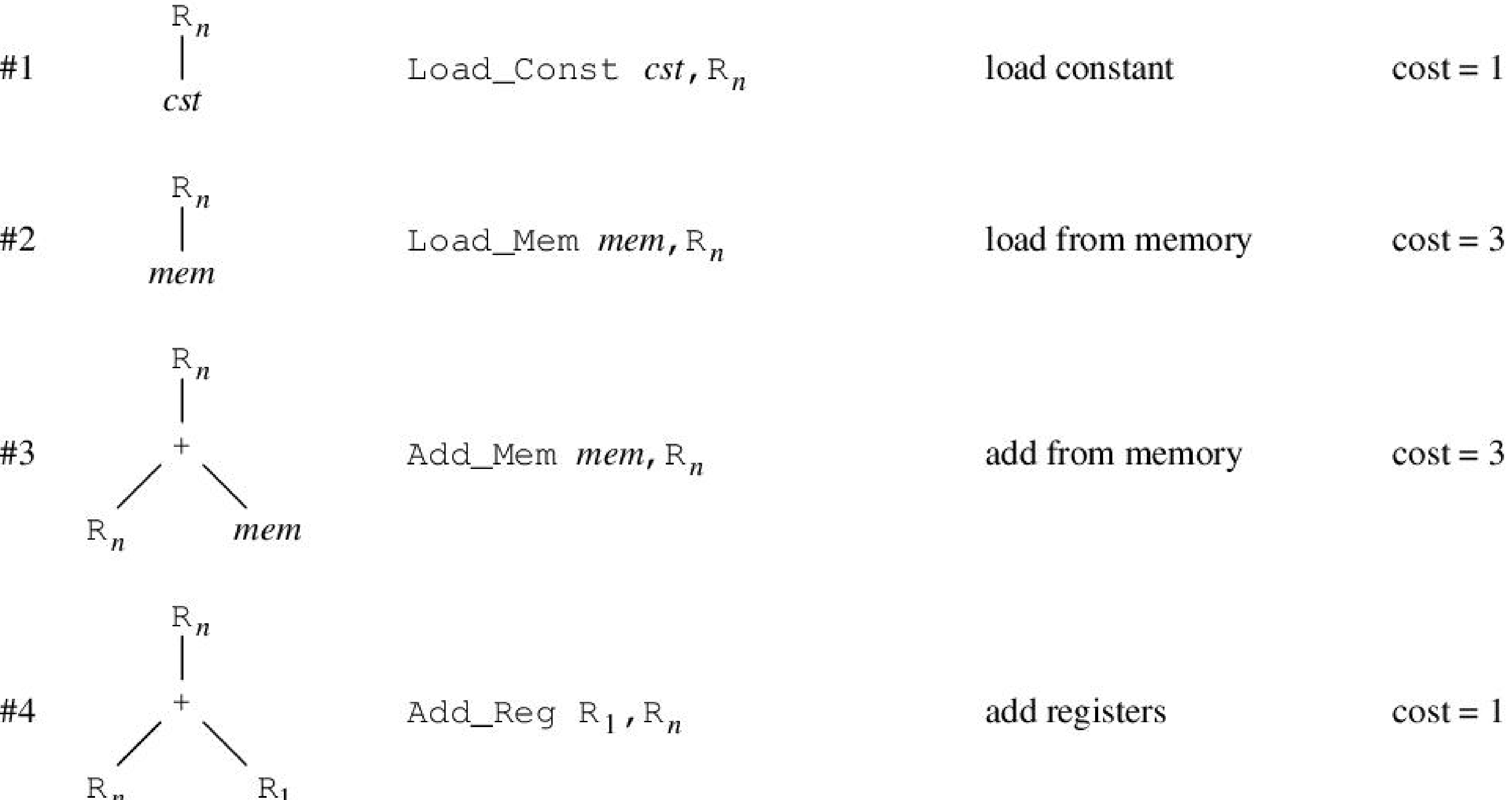
BURS-Codegenerierung

- bisher behandelte Codegenerierungs-Verfahren
  - Einschränkung (!) der Optimierungsdomäne: Übersetzung von IR(Zwischencode)-Knoten 1:1 in Maschinenbefehle
  - gewichtete Bäume (evtl. mit Spilling): exakt, aber nur für Bäume
  - Leitersequenzen: auch für andere DAGs, aber nur Heuristik
- neues Verfahren: BURS (Bottom-Up Rewrite Systems)
  - Abbildung mehrerer IR-Knoten auf einen Maschinenbefehl
  - Einschränkung auf Bäume
  - führt zu einer optimalen Befehlsfolge für Bäume, wenn keine Kompromisse wegen Komplexität gemacht werden
  - Komplexität bei CISC-Prozessoren durch
    - Kombinationen möglicher Befehle (einige hundert) und Adressierungsarten (10 und mehr)
    - weitere Vervielfachungen (z.B. durch Kommutativität)

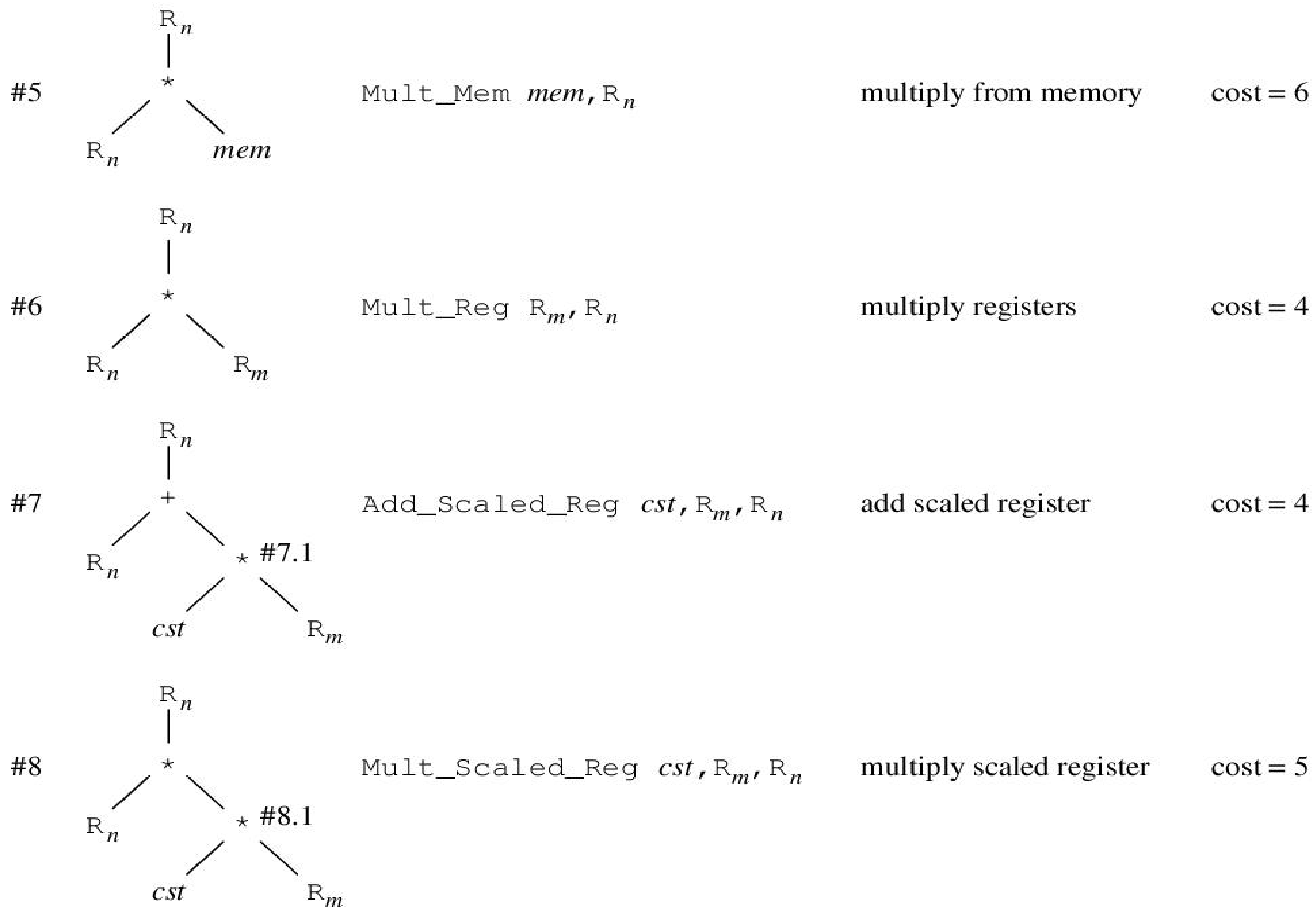
Pattern-Trees (1)

- jeder Befehlstyp (Kombination Operation, Operandentyp, etc.) modelliert durch Pattern-Tree (PT) und Kosten:



Pattern-Trees (2)

- Überdeckung des IR-Trees durch Pattern-Trees mit minimalen Gesamtkosten gesucht (Bsp.: #1/#5/#6 vs. #8)



Beispielbefehlssatz / realer Prozessor

- Beispielbefehlssatz
  - drei Operandentypen:

    `mem`: Speicher, `cst`: Konstante, `reg`: Register
  - ein Ergebnistyp:

    `reg`: Register
  - zwei Spezialbefehle mit mehr als einer Operation (scaled)
  - Knoten der Pattern-Trees enthalten Markierungen
- realer Prozessor
  - Kosten sind Funktion, nicht eine Konstante

    (z.B. geringere Kosten für #7, #8 bei `cst`=1,2,4)
  - Spezialbefehle billiger als Sequenz einfacher Befehle
  - in der Realität weit mehr Spezialbefehle

Eingabebaum / naive Übersetzung

- bisher: 1:1-Umsetzung der IR-Knoten:

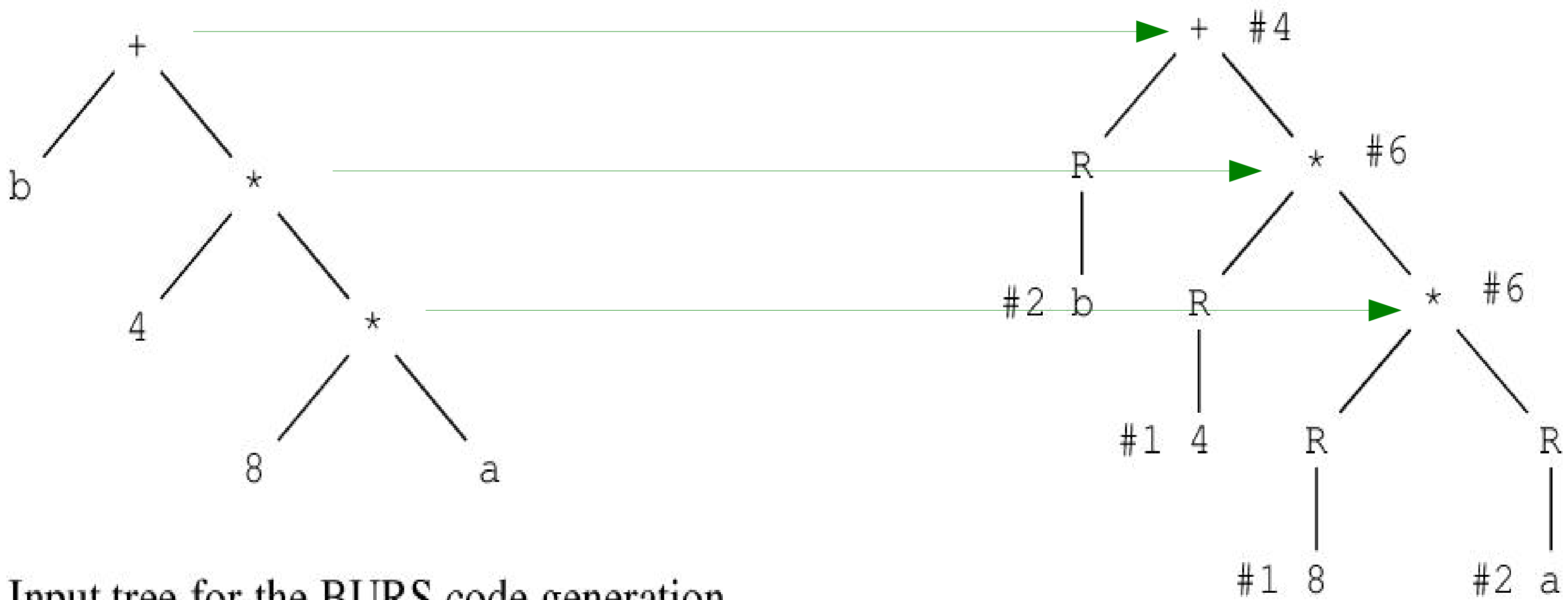


Figure 4.61 Input tree for the BURS code generation.

Figure 4.62 Naive rewrite of the input tree.

Kosten durch die naive Übersetzung

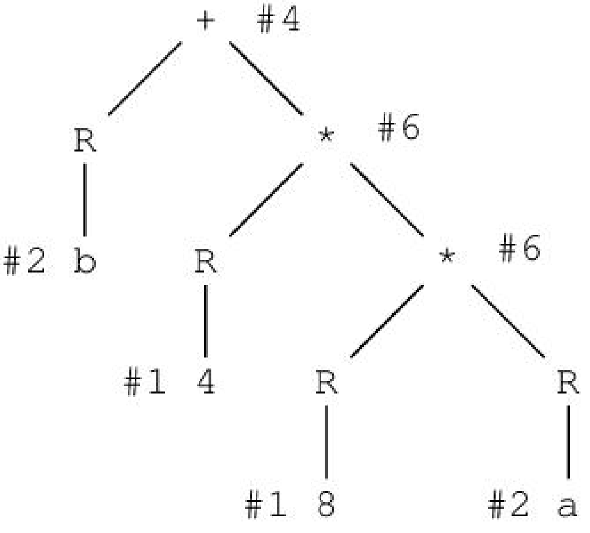


Figure 4.62 Naive rewrite of the input tree.

```
Load_Const   8,R1    ; 1 unit
Load_Mem     a,R2    ; 3 units
Mult_Reg     R2,R1   ; 4 units
Load_Const   4,R2    ; 1 unit
Mult_Reg     R1,R2   ; 4 units
Load_Mem     b,R1    ; 3 units
Add_Reg      R2,R1   ; 1 unit
             Total   = 17 units
```

Figure 4.63 Code resulting from the naive rewrite.

Verwendung komplexer Befehle

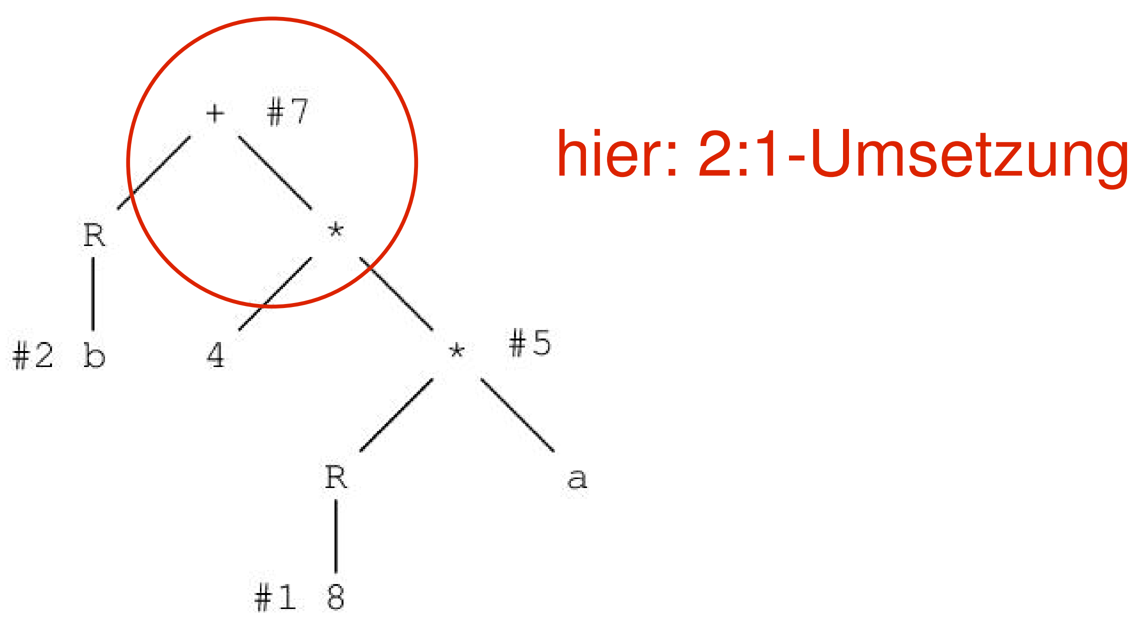


Figure 4.64 Top-down largest-fit rewrite of the input tree.

```
Load_Const        8,R1       ; 1 unit
Mult_Mem          a,R1       ; 6 units
Load_Mem          b,R2       ; 3 units
Add_Scaled_Reg    4,R1,R2    ; 4 units
                  Total      = 14 units
```

Figure 4.65 Code resulting from the top-down largest-fit rewrite.

Drei-Phasen Algorithmus

1. Bottom-Up-Befehlssammlung: (instruction-collecting scan)
   - bilde Menge möglicher Befehle für jeden Knoten
   - zwei Varianten: sammle alle möglichen Befehle

     (a) in Itemmengen (dynamisch)

     (b) mit einem Baumautomaten (statisch)

2. Top-Down-Befehlsauswahl: (instruction-selecting scan)
   - wähle pro Knoten einen Befehl aus seiner Menge
   - abhängig davon, ob bei (1.) Variante (a) oder (b) gewählt
   - Auswahl eines Befehls mit minimalen Kosten

3. Bottom-Up-Codeerzeugung: (code-generating scan)
   - gib die bei (2.) ausgewählten Befehle in korrekter Reihenfolge aus

Phase 1: Befehlssammlung

- Vorgehen
  - durchlaufe den Eingabebaum (IR-Tree) bottom-up
  - annotiere jeden besuchten Knoten mit den PTs der passenden Maschinenbefehle
- Match eines PTs gegen einen IR-Knoten N
  - Wurzel des PT passt auf N
  - Teilbäume des PTs passen auf Kinder von N
  - Blätter des PTs passen auf die verfügbaren Quellen aus:
    - `mem:` Speicher
    - `cst:` Konstante
    - `reg:` Register

Dotted Trees

- Visualisierung durch Markierung (“Dotted-Tree”)
  - Marke trennt erkannten von noch nicht erkanntem Teil des PT
- Verwaltung überlappender PTs
  - Unternummerierung (z.B. #7.1 für Teil von PT #7)
  - Unternummer in Menge des Kinds von N: Teilpattern hat gepasst

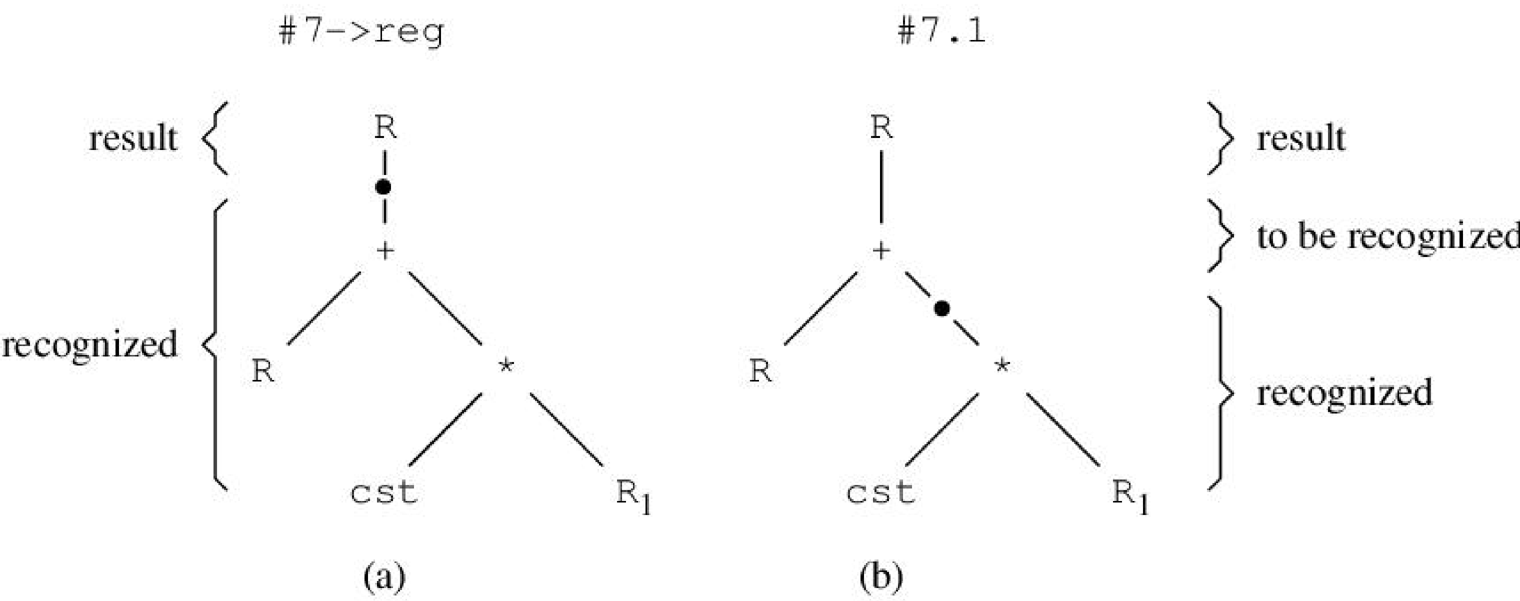


Figure 4.66 The dotted trees corresponding to #7→reg and to #7.1.

Bottom-Up Pattern-Matching (1)

- rekursiver Abstieg (Abb. 4.67)

```
PROCEDURE Bottom_up pattern matching (Node):
    IF Node is an operation:
        Bottom_up pattern matching (Node .left);
        Bottom_up pattern matching (Node .right);
        SET Node .label set TO Label set for (Node);
    ELSE IF Node is a constant:
        SET Node .label set TO Label set for constant ();
    ELSE Node is a variable:
        SET Node .label set TO Label set for variable ();
```

Bottom-Up Pattern-Matching (2)

- Matching von Operatorknoten

```
FUNCTION Label set for (Node) RETURNING a label set:
    SET Label set TO the Empty set;
    FOR EACH Label IN the Machine label set:
        FOR EACH Left label IN Node .left .label set:
            FOR EACH Right label IN Node .right .label set:
                IF Label .operator = Node .operator
                    AND Label .operand = Left label .result
                    AND Label .second operand = Right label .result:
                    Insert Label into Label set;
    RETURN Label set;
```

Bottom-Up Pattern-Matching (3)

- Matching von Konstanten

```
FUNCTION Label set for constant () RETURNING a label set:
    SET Label set TO
        { (No operator, No location, No location, "Constant") };
    FOR EACH Label IN the Machine label set:
        IF Label .operator = "Load" AND Label .operand = "Constant":
            Insert Label into Label set;
    RETURN Label set;
```

Bottom-Up Pattern-Matching (4)

- Matching von Speicherstellen

```
FUNCTION Label set for variable () RETURNING a label set:
    SET Label set TO
        { (No operator, No location, No location, "Memory") };
    FOR EACH Label IN the Machine label set:
        IF Label .operator = "Load" AND Label .operand = "Memory":
            Insert Label into Label set;
    RETURN Label set;
```

Bottom-Up Pattern-Matching (5)

- Typ der Knoteninformation

```
TYPE operator: "Load", '+', '*';
TYPE location: "Constant", "Memory", "Register", a label;

TYPE label:                          // a node in a pattern tree
    an operator FIELD operator;
    a location FIELD operand;
    a location FIELD second operand;
    a location FIELD result;
```

Figure 4.68 Types for bottom-up pattern recognition in trees.

Markierung des Beispiel-Eingabebaums (1)

- Aufsammeln aller Möglichkeiten

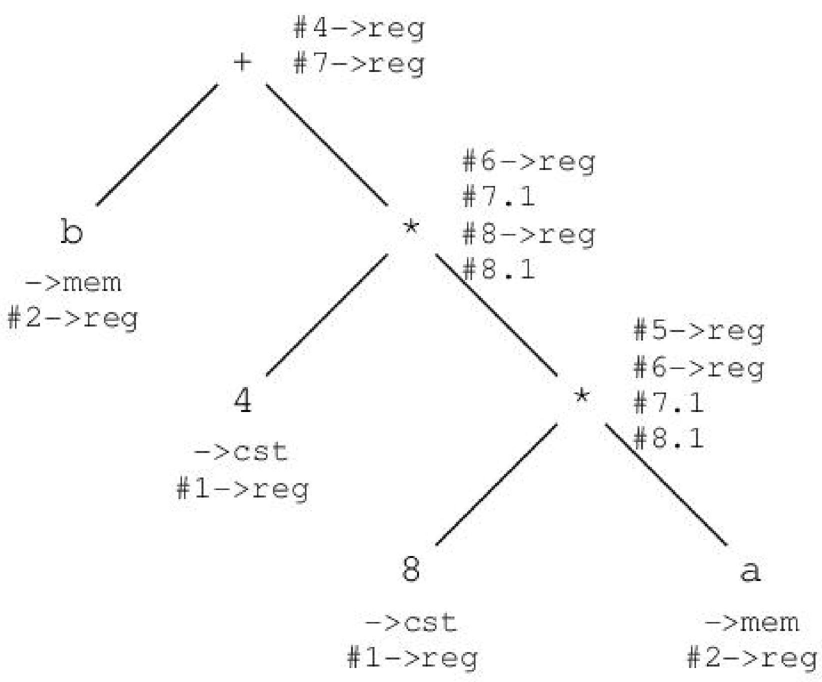


Figure 4.69 Label sets resulting from bottom-up pattern matching.

Markierung des Beispiel-Eingabebaums (2)

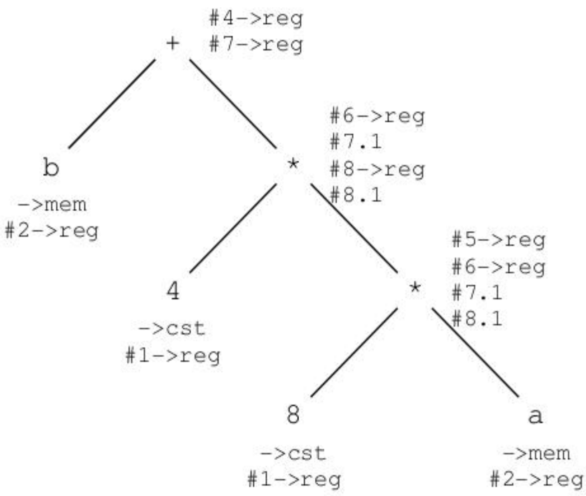


Figure 4.69 Label sets resulting from bottom-up pattern matching.

- Beispielbäume sind darin enthalten:

Figure 4.62 Naive rewrite of the input tree.

Figure 4.64 Top-down largest-fit rewrite of the input tree.

Befehlssammlung / (Un-)Vollständigkeit

- Erfolg nicht garantiert
  - unmarkierte Knoten ⇒ keine Ersetzungsalternative
  - BURS nicht vollständig: findet nicht solche Lösungen, bei denen ein IR-Knoten durch mehrere Maschinenbefehle implementiert werden muss
- in der Praxis kein Problem
  - Existenz vieler kleiner Maschinenbefehle (für 1:1-Umsetzung von Operatoren in Ausdrücken)
  - komplizierte AST-Knoten (z.B. Matrixmultiplikation) werden in Routinenaufruf umgewandelt ⇒ erscheinen nicht in IR-Trees
  - bei extrem simplem Prozessor: Assemblerroutine

Befehlssammlung / Komplexität

- Speicher-/Zeitkomplexität
  - linear in der Knotenzahl des Eingabebaums (unkritisch)
  - linear in der Anzahl aller möglichen und unmöglichen Kombinationen Befehle / Adressierungsarten (sehr ineffizient)
- Effizienzsteigerung
  - Ziel: Komplexität unabhängig vom Befehlssatz
  - Erstellung eines Baum(!)-Automaten
  - jede Kombination ergibt einen Zustand
  - dreidimensionale Zustandsübergangstabelle
    1. IR-Tree-Operator
    2. Zustand des linken Teilbaums
    3. Zustand des rechten Teilbaums
  - Kompression der Übergangstabelle: wichtig und gut möglich

Verwendung von Zustandsmengen

- Befehlssammlung durch Baumautomaten
- Berechnung jeder Knotenmenge einfach durch Tabellen-Lookup

```
PROCEDURE Bottom_up pattern matching (Node):
    IF Node is an operation:
        Bottom_up pattern matching (Node .left);
        Bottom_up pattern matching (Node .right);
        SET Node .state TO Next state
            [Node .operand, Node .left .state, Node .right .state];
    ELSE IF Node is a constant:
        SET Node .state TO State for constant;
    ELSE Node is a variable:
        SET Node .state TO State for variable;
```

Figure 4.70 Outline code for efficient bottom-up pattern matching in trees.

Drei-Phasen Algorithmus, Phasen 2 und 3

1. Bottom-Up-Befehlssammlung: (instruction-collecting scan)
   - bilde Menge möglicher Befehle für jeden Knoten
   - zwei Varianten: sammle alle möglichen Befehle
     (a) in Itemmengen (dynamisch)
     (b) mit einem Baumautomaten (statisch)
2. Top-Down-Befehlsauswahl: (instruction-selecting scan)
   - wähle pro Knoten einen Befehl aus seiner Menge
   - abhängig davon, ob bei (1.) Variante (a) oder (b) gewählt
   - Auswahl eines Befehls mit minimalen Kosten
3. Bottom-Up-Codeerzeugung: (code-generating scan)
   - gib die bei (2.) ausgewählten Befehle in korrekter Reihenfolge aus

Phase 2: Top-Down Befehlsauswahl

- Entscheidung
  (a) zwingend in einem Pattern (z.B. Wahl #8 bedingt #8.1)
  (b) sonst: wähle minimale *zulässige* Alternative

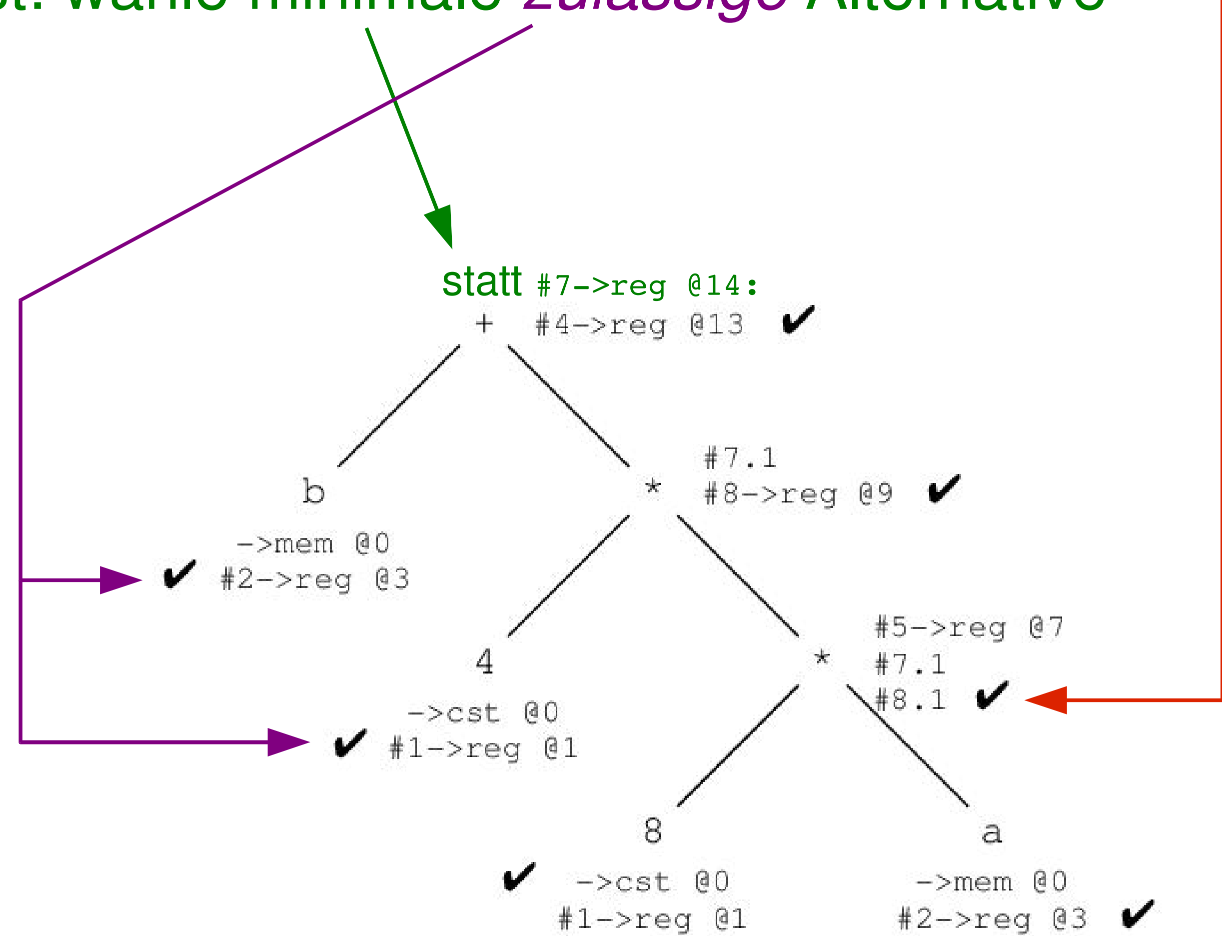


Figure 4.71 Bottom-up pattern matching with costs.

Phase 3: Bottom-Up Codegenerierungs-Scan

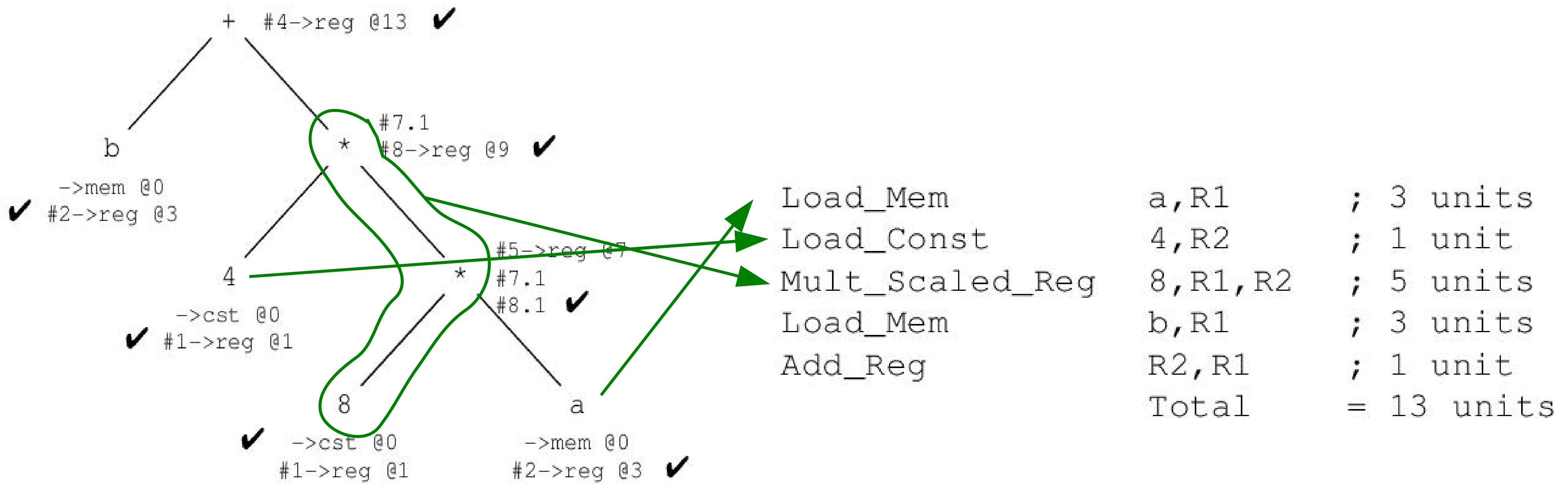


Figure 4.71 Bottom-up pattern matching with costs.

Figure 4.72 Code generated by bottom-up pattern matching.

Ausnutzung von Kommutativität

```
Load_Mem          a,R1       ; 3 units
Load_Const        4,R2       ; 1 unit
Mult_Scaled_Reg   8,R1,R2    ; 5 units
Load_Mem          b,R1       ; 3 units
Add_Reg           R2,R1      ; 1 unit
                  Total      = 13 units
```

Figure 4.72 Code generated by bottom-up pattern matching.

```
Load_Mem          a,R1       ; 3 units
Load_Const        4,R2       ; 1 unit
Mult_Scaled_Reg   8,R1,R2    ; 5 units
Add_Mem           b,R12      ; 3 units
                  Total      = 12 units
```

Figure 4.73 Code generated by bottom-up pattern matching, using commutativity.

Effiziente Befehlsauswahl

- Komplexitätsproblem bei der Top-Down Befehlsauswahl
  - rekursive Suche nach minimaler Lösungskombinationen für Teilprobleme
  - exponentielle Komplexität in der Baumtiefe
  - ineffiziente Mehrfachberechnungen
- Lösung
  - Tabulierung (Wiederverwendung berechneter Teilergebnisse)
  - bewährtes Programmierparadigma: Dynamic Programming
    - Bottom-Up-Vorberechnung (Integration in Befehlssammlung)
    - schnelle lokale Befehlsauswahl

Integration: Optimierung in Befehlssammlung

- Idee
  - integriere Optimierung in den Baumautomaten für die Befehlssammlung
  - Zustände müssen Kosteninformation tragen
  - erwartete Beschleunigung: Faktor 10-100
  - Voraussetzung: Kosten der Maschinenbefehle sind konstant
- Probleme
  1. absolute Kosten sind unbeschränkt
     - ⇒ unbeschränkte Markierungsmengen
     - ⇒ unbeschränkte Anzahl von Zuständen
     - ⇒ kein endlicher Automat möglich
  2. welche Kosten teilweise gematchten Pattern-Trees zuweisen?

Lösung der Probleme

1. unbeschränkte absolute Kosten
   - absoluter Wert ist für Auswahl irrelevant
   - Verwendung relativer Kosten,

     ⇒ Normalisierung der kleinsten Kosten auf 0
2. Kosten für teilweise gematchten Pattern-Trees
   - Operatorkosten für Teilbefehl unsinnig

     ⇒ werden als 0 festgelegt, d.h., es gehen nur die Kosten der Teilbäume in die Normalisierung ein

Tabellenerstellung am Beispiel

- Anfangszustände
  - S_1: { → cst @0 ,
#1 → reg @1 }
  - S_2: { → mem @0 ,
#2 → reg @3 }
- Zustandsübergänge (Auswahl, mit normalisierten Kosten)
  - state['+', S_1, S_1] = S_3: { #4 → reg @0 }
  - state['+', S_1, S_2] = S_5: { #3 → reg @0 }
  - state['+', S_1, S_4] = S_9: { #7 → reg @0 }
  - state['*', S_1, S_2] = S_6: { #5 → reg @4 ,
#7.1 @0 ,
#8.1 @0 }

Zustandsmenge des Automaten

S_{00} = { }
S_{01} = {→cst@0, #1→reg@1}
S_{02} = {→mem@0, #2→reg@3}
S_{03} = {#4→reg@0}
S_{04} = {#6→reg@5, #7.1@0, #8.1@0}
S_{05} = {#3→reg@0}
S_{06} = {#5→reg@4, #7.1@0, #8.1@0}
S_{07} = {#6→reg@0}
S_{08} = {#5→reg@0}
S_{09} = {#7→reg@0}
S_{10} = {#8→reg@1, #7.1@0, #8.1@0}
S_{11} = {#8→reg@0}
S_{12} = {#8→reg@2, #7.1@0, #8.1@0}
S_{13} = {#8→reg@4, #7.1@0, #8.1@0}

Figure 4.74 States of the BURS automaton for Figure 4.60.

cst: S_{01}
mem: S_{02}

Figure 4.75 Initial states for the basic operands.

- 3D-Übergangstabelle für Befehlssammlung mit Optimierung:

'+'	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
S_{01} – S_{13}	S_{03}	S_{05}	S_{03}	S_{09}	S_{03}	S_{09}	S_{03}	S_{03}	S_{03}	S_{03}	S_{03}	S_{03}	S_{09}

'*'	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
S_{01}	S_{04}	S_{06}	S_{04}	S_{10}	S_{04}	S_{12}	S_{04}	S_{04}	S_{04}	S_{04}	S_{04}	S_{13}	S_{12}
S_{02} – S_{13}	S_{07}	S_{08}	S_{07}	S_{11}	S_{07}	S_{11}	S_{07}	S_{07}	S_{07}	S_{07}	S_{07}	S_{11}	S_{11}

Figure 4.76 The transition table `Cost conscious next state[]`.

- Tabelle für Befehlsauswahl:

	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
cst		--	--	--	--	--	--	--	--	--	--	--	--
mem	--		--	--	--	--	--	--	--	--	--	--	--
reg	#1	#2	#4	#6	#3	#5	#6	#5	#7	#8	#8	#8	#8

Figure 4.77 The code generation table.

- annotierter IR-Tree:

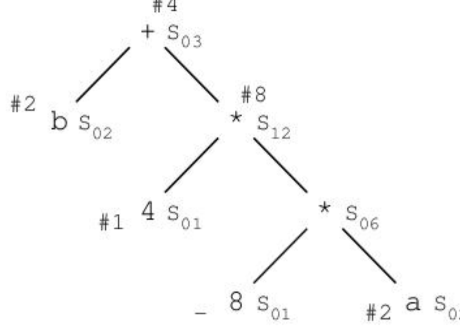


Figure 4.78 States and instructions used in BURS code generation.

Adaptionen des BURS-Verfahrens

- mehrere Registerklassen
  - z.B. `A`-Register für Adressberechnungen, `B`-Register nicht
  - differenziere Marken nach `regA` und `regB`
- Codegröße entscheidet bei gleichen (Laufzeit-)Kosten
  - Alternativen mit gleichen Kosten kommen oft vor
  - neue Kostenfunktion: das Paar (Laufzeit,Codegröße) mit der lexikographischen Ordnung
- Codegröße wichtiger als Laufzeit
  - invertiere Paar: (Codegröße,Laufzeit)

Zusammenfassung der Entwicklung

- 3-Phasen-Verfahren
  1. Bottom-Up Befehlssammlung
  2. Top-Down Befehlsauswahl, Kosten-minimierend
  3. Bottom-Up Codegenerierungs-Scan
- Optimierung A: Tabelle
  - Bottom-Up Befehlssammlung als Zustandsübergangsfunktion
- Optimierung B: Dynamic Programming
  - Reduktion der Befehlssammlung an jedem Knoten auf den Befehl kleinster Kosten plus evtl. Teilpattern
  - Bottom-Up-Prozess zusammen mit der Sammlung
- Optimierungen A+B:
  - Integration der Kosten in die Zustände
  - notwendig: Normalisierung der absoluten Kosten