

# ***Compilerbau***

Wintersemester 2010 / 2011

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

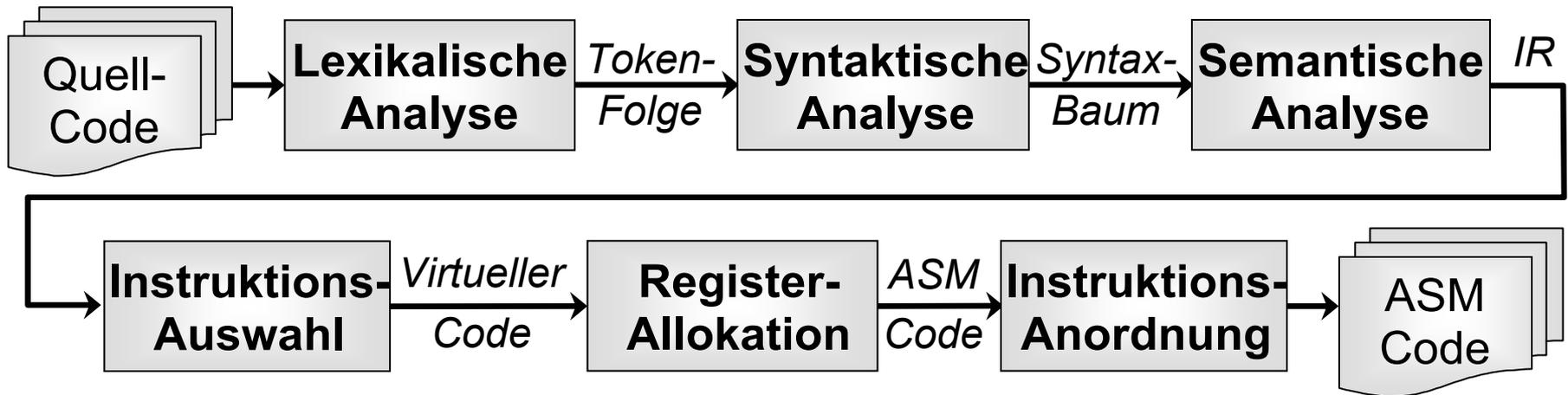
# ***Kapitel 6***

## ***Instruktionsauswahl***

# Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
  - Einführung
  - Baum-Überdeckung mit Dynamischer Programmierung
  - Diskussion
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

# Rolle der Instruktionauswahl



## Instruktionauswahl:

- Auswahl von Maschinenbefehlen zur Implementierung der IR
- „Herz“ des Compilers, das eigentliche Übersetzung von Quell- in Zielsprache vornimmt

# Funktion und Ziele

## Synonyme:

*Instruktionsauswahl*, *Code-Selektion* und *Code-Generierung* werden gleichbedeutend verwendet.

## Ein- & Ausgabe der Instruktionsauswahl:

- Eingabe: Eine zu übersetzende Zwischendarstellung *IR*
- Ausgabe: Ein Programm  $P(IR)$  (*meist in Assembler- oder Maschinencode, oft auch eine andere IR*)

## Randbedingungen der Instruktionsauswahl:

- $P(IR)$  muss semantisch äquivalent zu *IR* sein
- $P(IR)$  muss effizient hinsichtlich einer Zielfunktion sein

# Datenflussgraphen

## Was genau ist „semantische Äquivalenz zu IR“...?

$P(IR)$  muss zu  $IR$  äquivalenten Datenfluss haben, unter Berücksichtigung der durch den Kontrollfluss festgelegten Abhängigkeiten.

### Definition (*Datenflussgraph*):

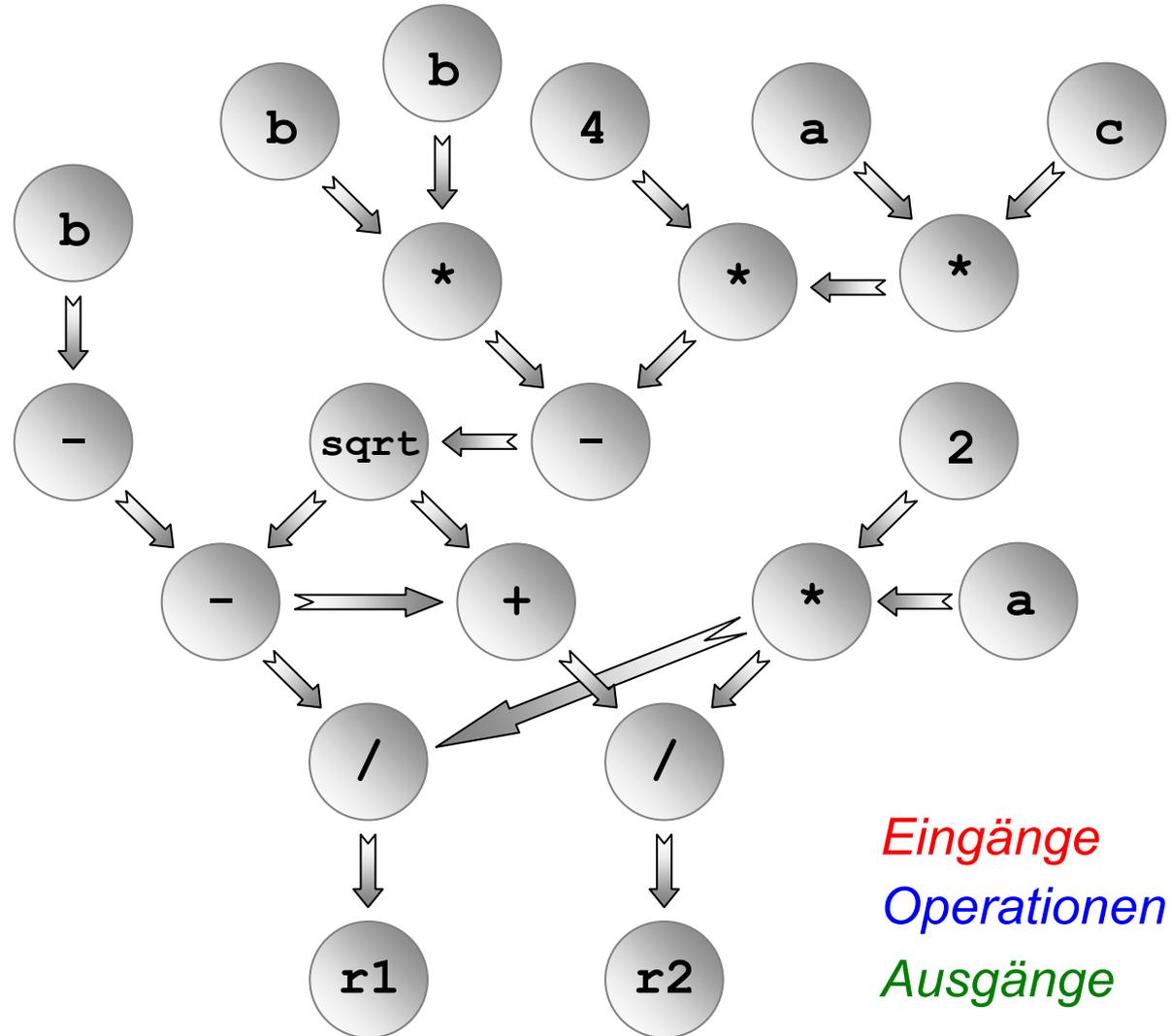
Sei  $B = (I_1, \dots, I_n)$  ein Basisblock ( Kapitel 5). Der *Datenflussgraph* (*DFG*) zu  $B$  ist ein gerichteter azyklischer Graph  $DFG = (V, E)$  mit

- Knoten  $v \in V$  repräsentiert entweder
  - einen Eingangswert in  $B$  (Eingangsvariable, Konstante)
  - oder eine einzelne Operation innerhalb von  $I_1, \dots, I_n$
  - oder einen Ausgangswert von  $B$
- Kante  $e = (v_i, v_j) \in E \Leftrightarrow v_j$  benutzt von  $v_i$  berechnete Daten

# Beispiel-DFG

```

t1 = a * c;
t2 = 4 * t1;
t3 = b * b;
t4 = t3 - t2;
t5 = sqrt( t4 );
t6 = -b;
t7 = t6 - t5;
t8 = t7 + t5;
t9 = 2 * a;
r1 = t7 / t9;
r2 = t8 / t9;
    
```



# Instruktionsauswahl

## Ziel und Aufgabe:

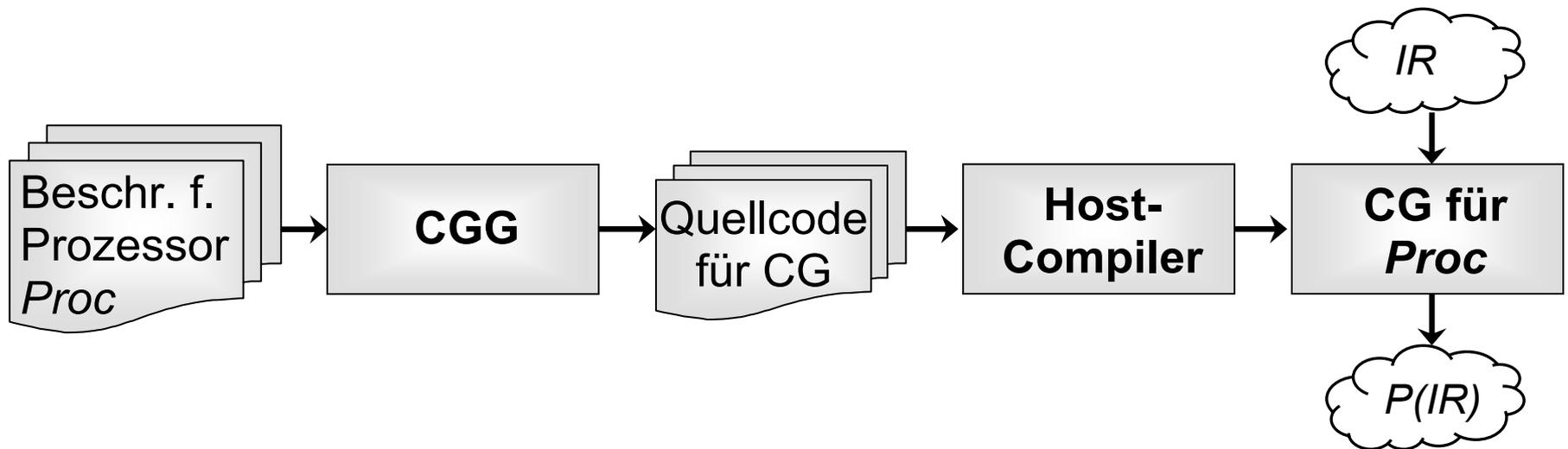
Überdecken der Knoten aller DFGs von *IR* mit semantisch äquivalenten Operationen der Zielsprache.

## Implementierung eines Code-Generators:

- Nicht-triviale, stark vom Ziel-Prozessor abhängige Aufgabe
- Per-Hand-Implementierung eines Code-Generators bei Komplexität heutiger Prozessoren nicht mehr vertretbar
- 👉 Statt dessen: Verwendung sog. *Code-Generator-Generatoren*

# Code-Generator-Generatoren

- Ein Code-Generator-Generator (CGG) erhält eine Prozessor-Beschreibung als Eingabe und erzeugt daraus einen Code-Generator (CG) für eben diesen Prozessor.



# Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
  - Einführung
  - Baum-Überdeckung mit Dynamischer Programmierung
  - Diskussion
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

# Tree Pattern Matching (TPM)

## Motivation:

- Überdeckung von Datenflussgraphen polynomiell reduzierbar auf 3-SAT

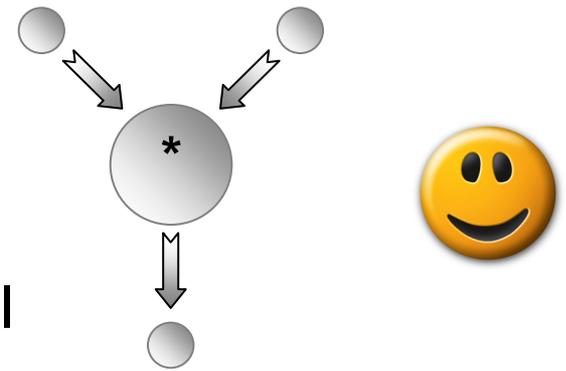
*[J. Bruno, R. Sethi, Code generation for a one-register machine, Journal of the ACM 23(3), Jul 1976]*

- Optimale Instruktionsauswahl ist NP-vollständig...

- **Aber:** Maschinenoperationen üblicher Prozessoren haben typischerweise baumförmigen Datenfluss:

### 👉 **Baum-basierte Instruktionsauswahl**

- Optimale baum-basierte Instruktionsauswahl effizient in polynomieller Laufzeit lösbar!



# Ablauf des Tree Pattern Matching

## Gegeben:

- Eine zu übersetzende Zwischendarstellung  $IR$

## Vorgehensweise:

- Programm  $P = \emptyset$ ;
- Für jeden Basisblock  $B$  aus  $IR$ :
  - Bestimme Datenflussgraph  $D$  von  $B$
  - Zerlege  $D$  in einzelne Datenflussbäume (DFTs)  $T_1, \dots, T_N$
  - Für jeden DFT  $T_i$ :
    - $P = P \cup \{ \text{Optimaler Code aus Baum-Überdeckung von } T_i \}$
- Gebe  $P$  zurück

# Zerlegung eines DFGs in DFTs

## Definition (**Gemeinsamer Teilausdruck**):

Sei  $DFG = (V, E)$  ein Datenflussgraph.

Ein Knoten  $v \in V$  mit mehr als einer ausgehenden Kante im DFG heißt *Gemeinsamer Teilausdruck (Common Subexpression, CSE)*.

## Definition (**Datenflussbaum**):

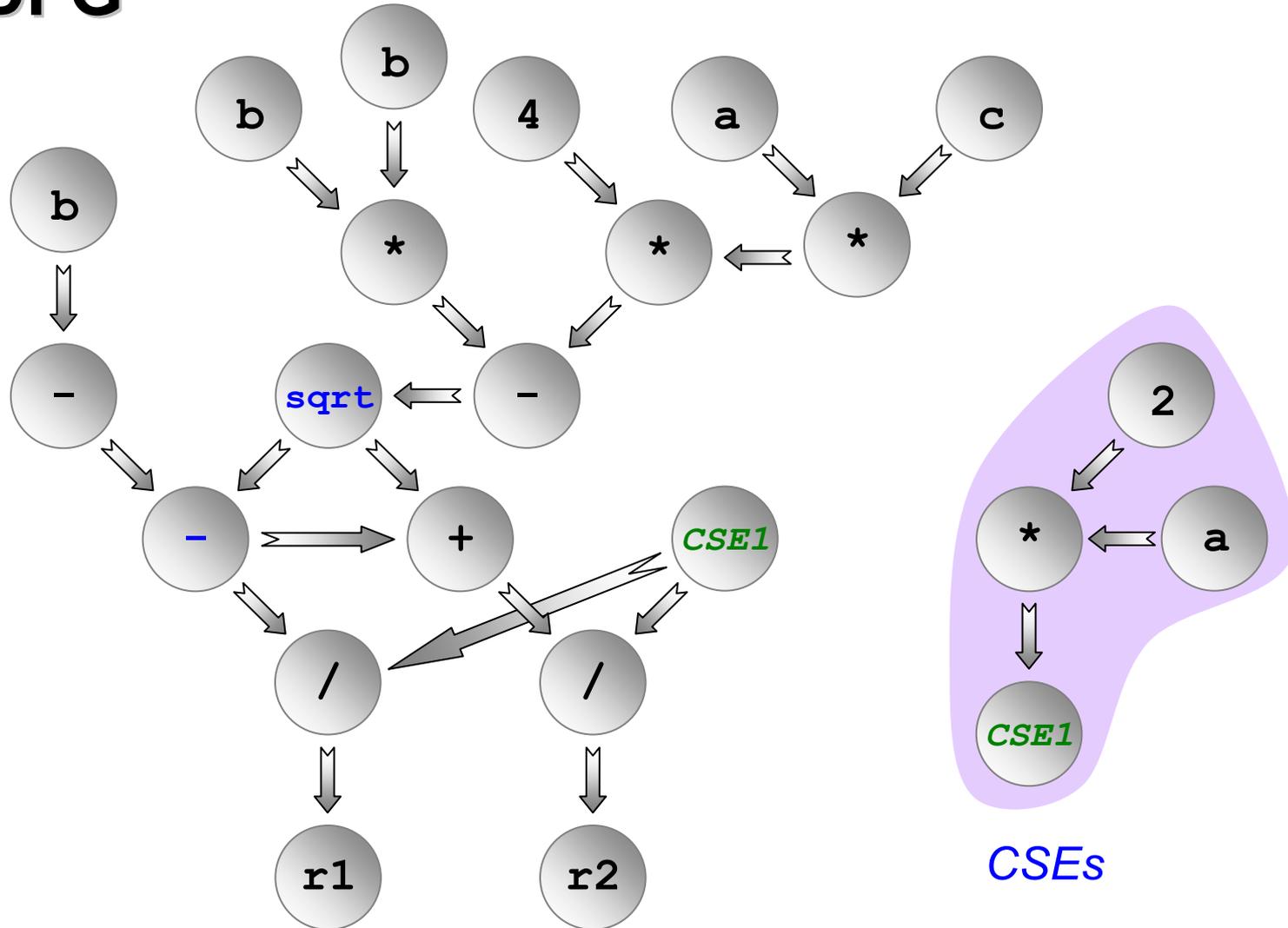
Ein Datenflussgraph  $DFG = (V, E)$  ohne CSEs heißt *Datenflussbaum (Data Flow Tree, DFT)*.

## ☞ **DFG-Zerlegung:**

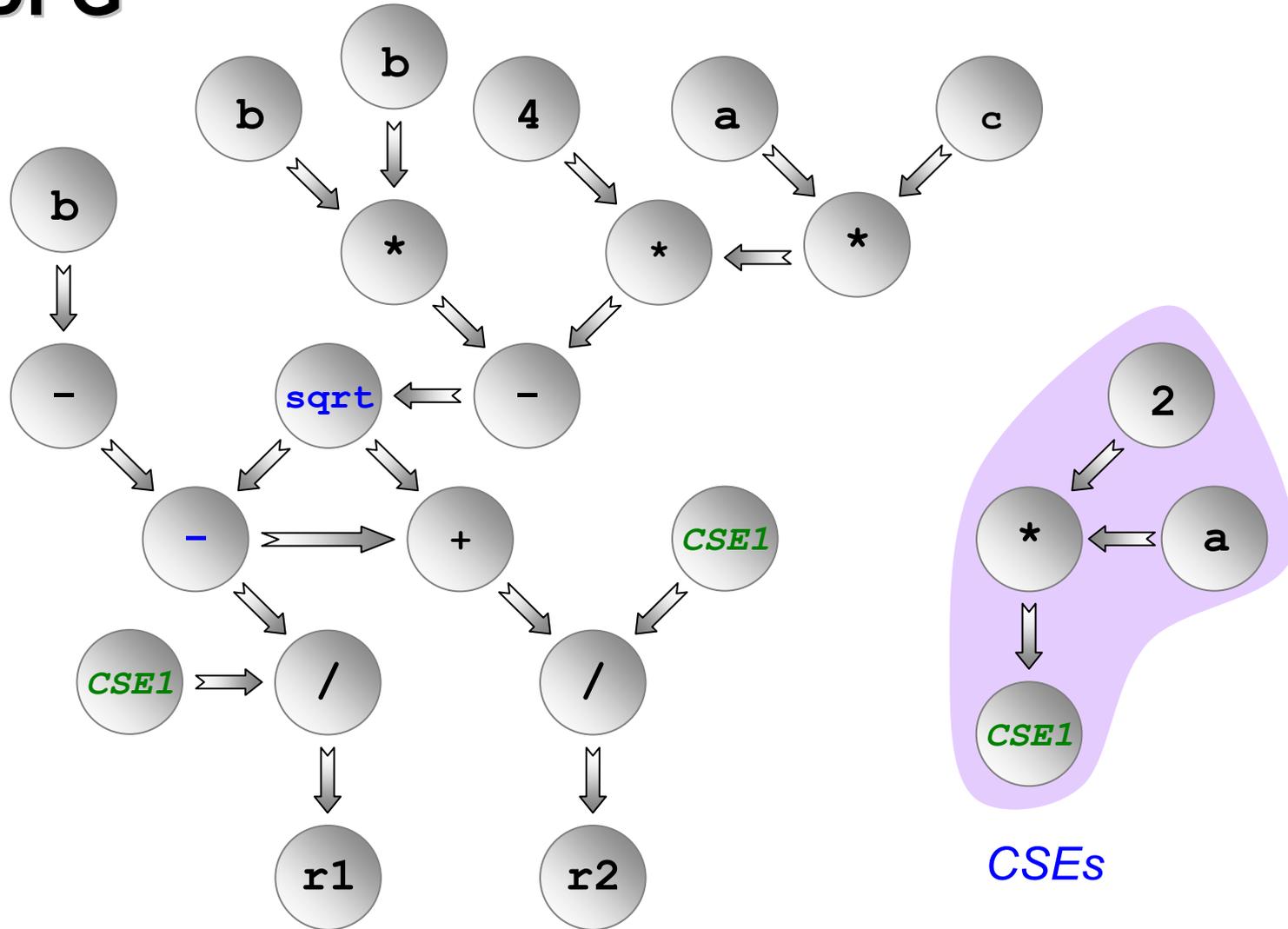
- Aufspaltung des DFGs in DFTs entlang der CSEs.
- Für jede CSE: Hilfsknoten in resultierenden Bäumen einfügen.



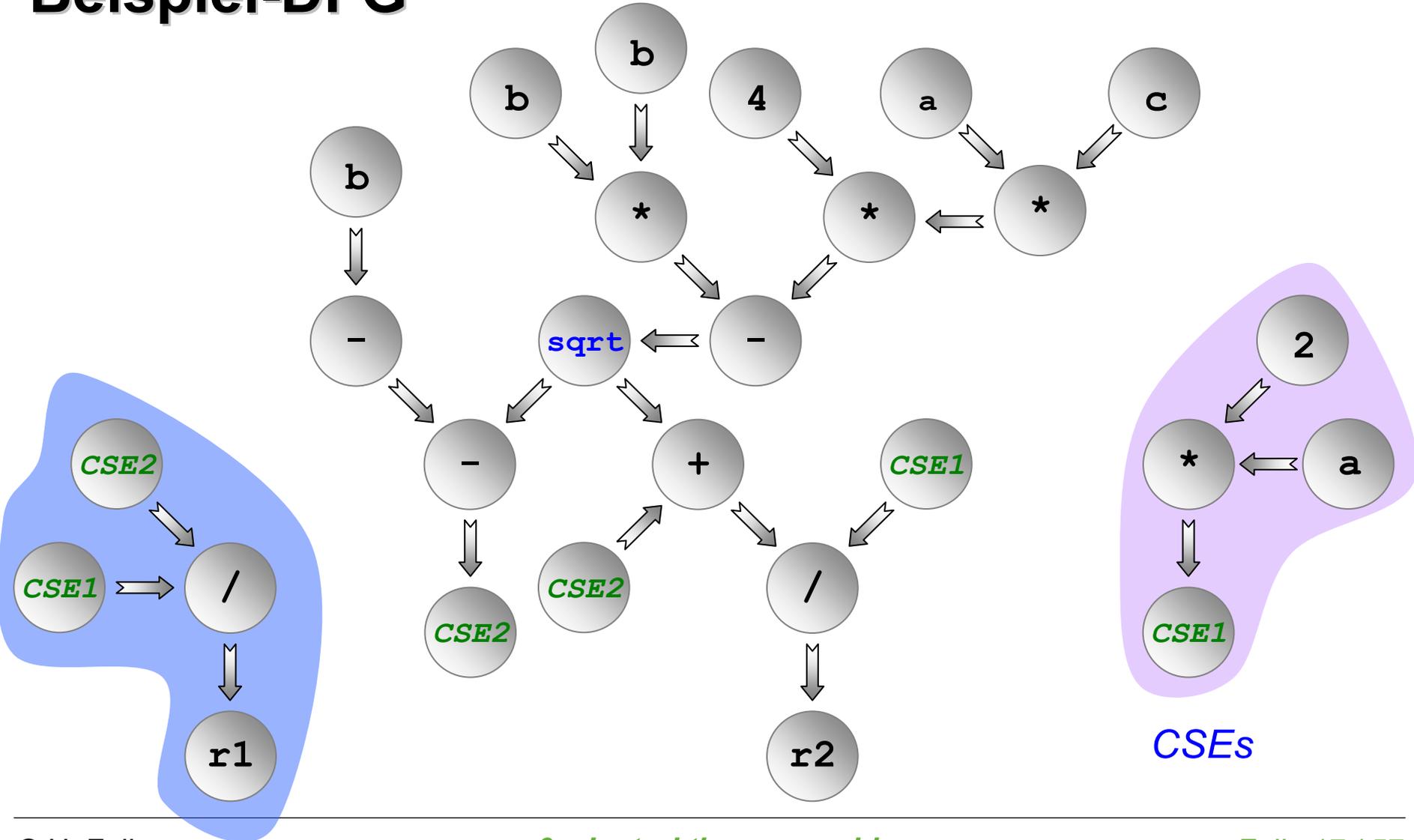
# Beispiel-DFG



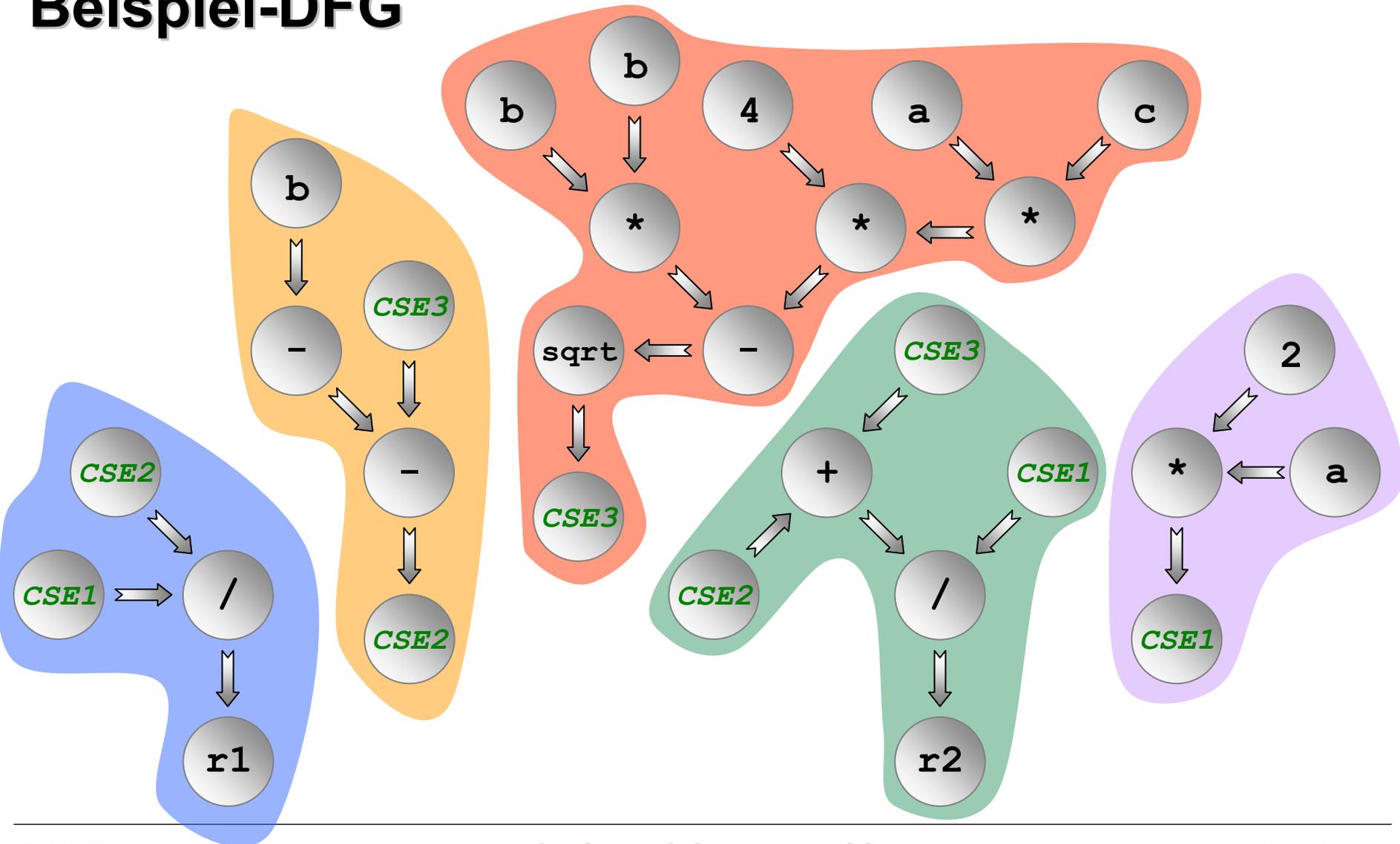
# Beispiel-DFG



# Beispiel-DFG



# Beispiel-DFG



# Baum-Überdeckung (*Tree Cover*)

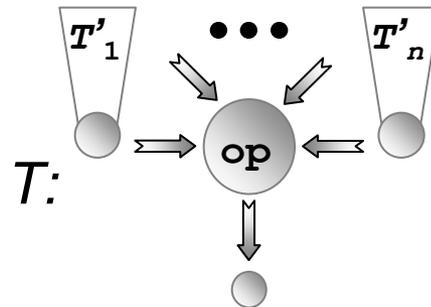
## Definition (*Baum-Überdeckung durch Operationsfolge*):

Sei  $T = (V, E)$  ein DFT,  $S = (o_1, \dots, o_N)$  eine Folge von Maschinenoperationen. Die letzte Operation  $o_N$  habe das Format

$d \leftarrow \text{op} (s_1, \dots, s_n)$ .  $S'_1, \dots, S'_n$  bezeichne die Teil-Folgen von  $S$ , die jeweils die Operanden  $s_1, \dots, s_n$  von  $O_N$  berechnen.

$S$  überdeckt  $T$  genau dann, wenn

- Operator  $\text{op}$  der Wurzel von  $T$  entspricht,  $T$  sich also wie folgt schreiben lässt:



- und wenn alle  $S'_i$  jeweils  $T'_i$  überdecken ( $1 \leq i \leq n$ ).

# Beispiel für Überdeckungen

## TriCore-Befehlssatz:

add Dc, Da, Db ( $Dc = Da + Db$ )

mul Dc, Da, Db ( $Dc = Da * Db$ )

madd Dc, Dd, Da, Db ( $Dc = Dd + Da * Db$ )

☞ Operation `add %d4, %d8, %d9` überdeckt *T1*

☞ Operation `mul %d10, %d11, %d12` überdeckt *T2*

☞ Auch klar: Operationsfolge

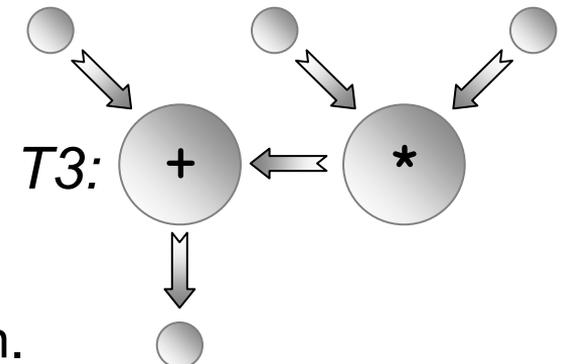
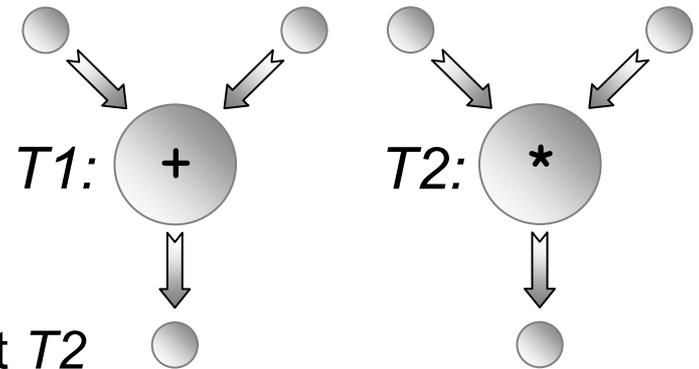
`mul %d10, %d11, %d12`

`add %d4, %d8, %d10` überdeckt *T3*

☞ Zusätzlich: Einelementige Folge

`madd %d4, %d8, %d11, %d12` überdeckt *T3* auch.

## Datenflussbäume:



# Tree Pattern Matching Algorithmus (1)

## Gegeben:

- DFT  $T = (V, E)$ , Knoten  $v_0 \in V$  sei Ausgang von  $T$
- Menge  $O$  aller generierbarer Maschinen-Operationen  $o$
- Kostenfunktion  $c: O \rightarrow \mathbb{N}$  (z.B. Größe jeder Operation in Bytes)
- Anzahl  $K \in \mathbb{N}$  aller Register des Ziel-Prozessors

## Datenstrukturen:

- Array  $C[j][v]$ : minimale Kosten pro Knoten  $v \in V$ , wenn zur Berechnung des Teilbaums mit Wurzel  $v$   $j$  Register zur Verfügung stehen.
- Array  $M[j][v]$ : optimale Maschinen-Operation und Operanden-Reihenfolge pro Knoten  $v \in V$  für  $j$  Register.

# Tree Pattern Matching Algorithmus (2)

## Ablauf – TPM( DFT $T$ ):

- initialize(  $T$  );
- computeCosts(  $T$  );
- generateCode(  $T, K$  );

## Phase 1 – initialize( DFT $T$ ):

- Für alle verfügbaren Register  $1 \leq j \leq K$  und für alle Knoten  $v \in V$ :

$$C[j][v] = \begin{cases} 0, & \text{falls } v \text{ Eingang von } T \\ \infty & \text{sonst} \end{cases}$$

- Für alle verfügbaren Register  $1 \leq j \leq K$  und für alle Knoten  $v \in V$ :

$$M[j][v] = \emptyset$$

# Tree Pattern Matching Algorithmus (3)

## Phase 2 – computeCosts( DFT $T$ ):

- Für alle Knoten  $v \in V$  in Postorder-Folge, ausgehend von  $v_0$ :
  - Sei  $T'$  der Teilbaum von  $T$  mit aktuellem Knoten  $v$  als Wurzel
  - Für alle Operationen  $o \in O$ , die  $T'$  überdecken:
    - Zerlege  $T'$  anhand von  $o$  in Teilbäume  $T'_1, \dots, T'_n$  mit jeweiligen Wurzeln  $v'_1, \dots, v'_n$  gemäß Tree Cover-Definition
    - Für alle  $1 \leq j \leq K$  und alle Permutationen  $\pi$  über  $(1, \dots, n)$ :

- Berechne minimale Kosten für Knoten  $v$ :

$$C[j][v] = \min( C[j][v], \sum_{i=1}^n C[j-i+1][v'_{\pi(i)}] + c(o) )$$

(  dyn. Programmierung )

$M[j][v] = \text{Paar } (o, \pi), \text{ das zu minimalem } C[j][v] \text{ führt}$

# Tree Pattern Matching Algorithmus (4)

## Phase 3 – generateCode( DFT $T$ , int $j$ ):

- Sei Knoten  $v \in V$  Wurzel von  $T$
- Operation  $o$  = erstes Element von  $M[j][v]$
- Permutation  $\pi$  = zweites Element von  $M[j][v]$
- Zerlege  $T$  anhand von  $o$  in Teilbäume  $T_1, \dots, T_n$  gemäß Tree Cover-Definition
- Für alle  $i = 1, \dots, n$ : generateCode(  $T_{\pi(i)}, j-i+1$  )
- Generiere Code für Operation  $o$

*[A. Aho, S. Johnson, Optimal Code Generation for Expression Trees, Journal of the ACM 23(3), Jul 1976]*

## Bemerkungen (1)

- Postorder-Durchlauf: Für die Wurzel  $v$  von  $T$  besuche erst die Kinder  $v_1, \dots, v_n$  in Postorder-Folge, dann besuche  $v$ .
- Permutation  $\pi$ : Für den aktuellen Knoten  $v$  und Teilbäume  $T'_1, \dots, T'_n$  mit Wurzeln  $v'_1, \dots, v'_n$  beschreibt eine Permutation  $\pi$  eine mögliche Reihenfolge, in der die Teilbäume ausgewertet werden können.  $\pi = (2, 3, 1)$  besagt bspw., dass zuerst Teilbaum 2 auszuwerten ist, dann Teilbaum 3, dann Teilbaum 1.
- `computeCosts` berechnet für jeden Knoten  $v$  die minimalen Kosten, unter Berücksichtigung aller möglichen Auswertungsreihenfolgen der Kinder von  $v$  (d.h. alle Permutationen  $\pi$ ) und aller möglichen Anzahlen freier Register (d.h. alle Werte  $j \in [1, K]$ ).

## Bemerkungen (2)

- TPM-Algorithmus betrachtet für jeden Baum  $T$  stets, wie viele der  $K$  Register des Prozessors noch frei sind, d.h. es werden keine virtuellen Register verwendet.
- Dementsprechend werden die Kosten in Abhängigkeit von der Anzahl  $j$  verfügbarer Register berechnet.
- Für die Knoten  $v'_1, \dots, v'_n$  und einen festen Wert  $j$  variieren die Kosten allerdings, und zwar abhängig von der Permutation  $\pi$ !

## Bemerkungen (3)

**Beispiel:** zur Auswertung des aktuellen Knotens  $v$  habe man  $j = 3$  Register zur Verfügung. Um Teilbaum  $T'_1$  auszuwerten, brauche man 2 freie Register, für  $T'_2$  jedoch 3.

- $\pi = (1, 2)$ : Wird erst  $T'_1$  ausgewertet, werden zwischenzeitlich 2 Register benutzt, das Ergebnis von  $T'_1$  liegt danach dauerhaft in einem der 3 verfügbaren Register. Für  $T'_2$  stehen somit nur noch 2 Register bereit. Da  $T'_2$  aber 3 Register braucht, sind zur Auswertung von  $T'_2$  zusätzliche Speicher-Instruktionen zu generieren, die zu höheren Kosten führen.
- $\pi = (2, 1)$ :  $T'_2$  belegt während der Auswertung alle 3 verfügbaren Register, das Ergebnis von  $T'_2$  liegt danach dauerhaft in einem der 3 verfügbaren Register. Für  $T'_1$  stehen somit nur noch 2 Register bereit. Da  $T'_1$  aber nur 2 Register zur Auswertung braucht, fallen keine zusätzlichen Instruktionen an.

# Laufzeit-Komplexität von TPM

## Annahme:

- Befehlssatz eines Prozessors sei fest vorgegeben
- ☞ Größe der Menge  $O$  konstant
- ☞ Anzahl der Permutationen  $\pi$  ebenfalls konstant, da die Anzahl von Operanden pro Operation im Befehlssatz konstant ist  
(*typischerweise 2 bis 3 Operanden pro Operation*)

## Kostenberechnung:

- Da Schleifen über alle überdeckenden Operationen und über alle Permutationen nur konstanten Faktor beisteuern:
- ☞ Lineare Komplexität in Größe von  $T$ :  $O(|V|)$

## Code-Generierung:

- Offensichtlich auch lineare Komplexität in Größe von  $T$ :  $O(|V|)$

# Verbleibende Offene Fragen

**Präsentierter TPM-Algorithmus generisch formuliert.**

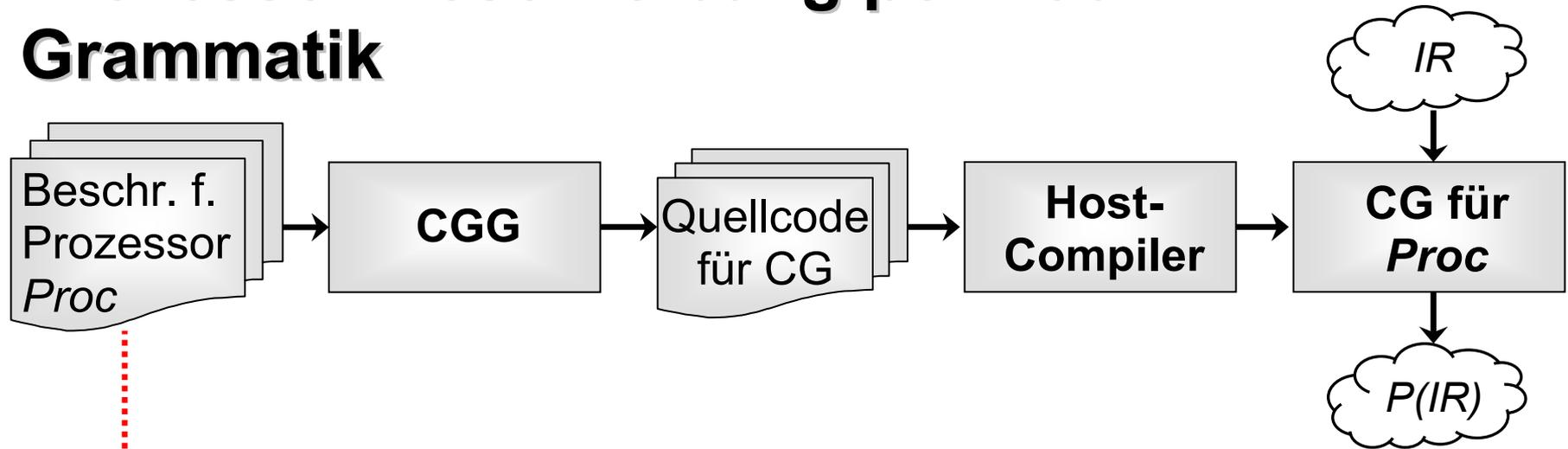
☞ ***Wie wird dieser Algorithmus für einen konkreten Prozessor adaptiert?***

**Zu klärende Detailfragen:**

- Wie ist die Entsprechung eines Operators  $op$  mit der Wurzel von  $T$  realisiert (vgl. *Tree Cover-Definition*)?
- In welcher Form erhält der TPM-Algorithmus die Menge  $O$  aller generierbarer Maschinen-Operationen und die Kostenfunktion  $c$ ?
- Wie handhabt TPM die Speicherung der optimalen Maschinen-Operation  $o$  in  $M$  + konkrete Code-Generierung für  $o$ ?

Im Folgenden: Annahme unendlich vieler virtueller Register

# Prozessor-Beschreibung per Baum-Grammatik



- Grammatik  $G$ , die für Teilbäume eines DFTs Maschinenoperationen generiert
- Jede einzelne Regel von  $G$  realisiert eine mögliche Überdeckung für einen DFT-Knoten
- Durch Anwendung von Regeln wird also Code abgeleitet
- Jede einzelne Ableitung/Regel verursacht Kosten

# Aufbau der Baum-Grammatik (1)

## Gemäß Code-Generator-Generator *icd-cg*:

- Baum-Grammatik  $G$  besteht aus Regeln  $R_1, \dots, R_r$
- Jede Regel  $R_i$  hat eine *Signatur*, bestehend aus Terminal- und Nichtterminal-Symbolen:
 
$$\langle \text{nonterminal}_{i,0} \rangle : \langle \text{terminal}_{i,1} \rangle ( \langle \text{nonterminal}_{i,2} \rangle , \dots , \langle \text{nonterminal}_{i,n} \rangle )$$

(Angabe von Nichtterminalen in (...) optional)  
 (Sog. Kettenregeln  $\langle \text{nonterminal}_{i,0} \rangle : \langle \text{nonterminal}_{i,1} \rangle$  auch legal)
- *Terminale*: mögliche Knoten im DFT  $T$  (z.B. `tpm_binaryExpPLUS`, `tpm_BinaryExpMULT`, ... in ICD-C)
- *Nichtterminale*: i.d.R. prozessorspezifische Speicher-Klassen, wo Quell- & Ziel-Operanden von Operationen abgelegt sein können (z.B. *Daten- & Adressregister, Immediate-Konstanten, ...*)

## Aufbau der Baum-Grammatik (2)

### Beispiel (anhand von ICD-C & TriCore 1.3):

- Regel

**dreg**: **tpm\_BinaryExpPLUS** ( **dreg**, **dreg** )

zuständig für Überdeckung des binären Operators + aus ANSI-C, mit beiden Summanden in Datenregistern und Summe in einem Datenregister.

- Regel

**dreg**: **tpm\_BinaryExpMULT** ( **dreg**, **const9** )

zuständig für Überdeckung des binären Operators \* aus ANSI-C, mit erstem Faktor in Datenregister, zweitem als vorzeichenbehaftetem 9-Bit Immediate-Wert und Produkt in einem Datenregister.

# Aufbau der Baum-Grammatik (3)

## Gemäß Code-Generator-Generator *icd-cg*: (Fortsetzung)

- Terminal- und Nichtterminal-Symbole müssen in Baum-Grammatik  $G$  deklariert sein
- Grundlegender Aufbau des Files zur Beschreibung von  $G$ :

```

%{          // Präambel
%}

%term <terminal1>
%term <terminal2>
...

%declare <nonterminal0>;

                                %declare <nonterminal1>;
                                ...
                                %%
                                Regel1;
                                Regel2;
                                ...
                                %%
  
```



## Aufbau der Baum-Grammatik (4)

### Gemäß Code-Generator-Generator *icd-cg*: (Fortsetzung)

- Die Spezifikation jeder Regel  $R_i$  in der Baum-Grammatik besteht aus Signatur, *Cost*-Teil und *Action*-Teil:

```

<nonterminali,0>: <terminali,1> ( <nonterminali,2> , ... ,
                                     <nonterminali,n> )
{
    // Code zur Kosten-Berechnung
}
=
{
    // Code für Action-Teil
};
  
```

## Aufbau der Baum-Grammatik (5)

### Gemäß Code-Generator-Generator *icd-cg*: (Fortsetzung)

- *Cost*-Teil von  $R_i$  weist **nonterminal** $_{i,0}$  Kosten zu, die entstehen, wenn  $R_i$  zur Überdeckung von **terminal** $_{i,1}$  benutzt wird.
- *Cost*-Teil kann beliebigen benutzerdefinierten C/C++-Code zur Kostenberechnung enthalten.
- Kosten können z.B. Anzahl erzeugter Maschinen-Operationen, Codegröße, ..., repräsentieren.
- Kosten von  $R_i$  können explizit auf  $\infty$  gesetzt werden, wenn  $R_i$  in speziellen Situationen keinesfalls zur Baum-Überdeckung verwendet werden soll.
- C/C++-Datentyp für Kosten, Kleiner-Als-Vergleich von Kosten und Null- & Unendlich-Kosten sind in Präambel von  $G$  zu deklarieren.

## Aufbau der Baum-Grammatik (6)

Beispiel (anhand von ICD-C & TriCore 1.3):

```

// Präambel
typedef int COST;
#define DEFAULT_COST 0;
#define COST_LESS(x, y) ( x < y )
COST COST_INFINITY = INT_MAX;
COST COST_ZERO = 0;
...
  
```

- Deklaration eines simplen Kostenmaßes – identisch mit Typ `int`
- Vergleich von Kosten mit `<` Operator auf Typ `int`
- Default-, Null und  $\infty$ -Kosten auf 0 bzw. max. `int`-Wert gesetzt

# Aufbau der Baum-Grammatik (7)

Beispiel (anhand von ICD-C & TriCore 1.3):

```

dreg: tpm_BinaryExpPLUS ( dreg, dreg )
{
  $cost[0] = $cost[2] + $cost[3] + 1;
} = {};
  
```

- Verwendung des feststehenden Schlüsselworts `$cost[j]` zum Zugriff auf Kosten von `nonterminali,j`
- Kosten für binäres `+` mit Summanden in Datenregistern (`$cost[0]`) gleich Kosten für ersten Operanden (`$cost[2]`) plus Kosten für zweiten Operanden (`$cost[3]`) plus eine weitere Operation (**ADD**)

## Aufbau der Baum-Grammatik (8)

### Gemäß Code-Generator-Generator *icd-cg*: (Fortsetzung)

- *Action*-Teil von  $R_i$  wird ausgeführt, wenn  $R_i$  die Regel mit minimalen Kosten zur Überdeckung von **terminal<sub>*i*,1</sub>** ist.
- *Action*-Teil kann beliebigen benutzerdefinierten C/C++-Code zur Code-Generierung enthalten.
- Verwendung des feststehenden Schlüsselwortes `$action[j]` zum Ausführen der *Action*-Teile für Operanden **nonterminal<sub>*i*,j</sub>**
- Nichtterminal-Symbole können mit Parametern und Rückgabewerten in  $G$  deklariert werden, um Werte zwischen *Action*-Teilen von Regeln auszutauschen.

## Aufbau der Baum-Grammatik (9)

Beispiel (anhand von *ICD-C* & *TriCore 1.3*):

```

dreg: tpm_BinaryExpPLUS ( dreg, dreg ) { }={
  if (target.empty()) target = getNewRegister();
  string r1($action[2] ("")), r2($action[3] (""));
  cout << "ADD " << target << ", " << r1
        << ", " << r2 << endl;
  return target;
};
  
```

- Zuerst Bestimmung des Registers für Ziel-Operanden
- Danach Code-Generierung für beide Operanden, inkl. Übergabe der Register `r1` und `r2`, in denen beide Operanden liegen
- Zuletzt Code-Generierung für Addition

# Aufbau der Baum-Grammatik (10)

**Beispiel (anhand von ICD-C & TriCore 1.3):**

```
%declare<string> dreg<string target>;
```

- Deklaration eines Nichtterminals für virtuelles Datenregister
- Ein string kann in Parameter **target** übergeben werden, um einem Action-Teil vorzugeben, in welchem Datenregister dieser seinen Ziel-Operanden abzulegen hat.
- Ein Action-Teil kann einen string zurückliefern, der das Datenregister bezeichnet, in dem dieser seinen Ziel-Operanden abgelegt hat.

# Baum-Überdeckung & Baum-Grammatik

## Baum-Überdeckung:

Eine Regel  $R_i$  aus  $G$  mit Signatur

$$\langle \text{nonterminal}_{i,0} \rangle : \langle \text{terminal}_{i,1} \rangle ( \langle \text{nonterminal}_{i,2} \rangle , \dots , \langle \text{nonterminal}_{i,n} \rangle )$$

überdeckt einen DFT  $T$  genau dann, wenn

- es Regeln in  $G$  gibt, die jeweils Teil-Baum  $T'_j$  überdecken und jeweils ein Nichtterminal-Symbol der Klasse  $\langle \text{nonterminal}_{i,j} \rangle$  erzeugen ( $2 \leq j \leq n$ ), und
- die Kosten von  $R_i$  kleiner als  $\infty$  sind.

# TPM-Algorithmus & Baum-Grammatik

**Phase 1 – Initialisierung:** *Unverändert*

**Phase 2 – Kostenberechnung:**

- Anstatt alle Operationen  $o \in O$  zu bestimmen, die Teilbäume  $T'$  überdecken:
- ☞ Bestimme Menge  $R'$  aller Regeln  $R_i \in G$ , die  $T'$  überdecken
- ☞ Berechne  $C[v]$  wie ursprünglich, lediglich unter Ausführung des Codes der *Cost*-Teile aller Regeln aus  $R'$
- ☞ Speichere in  $M[v]$  die Regel  $R^{opt} \in R'$  mit minimalem  $C[v]$

**Phase 3 – Code-Generierung:**

- Für Wurzel  $v_0 \in T$ : Rufe *Action*-Teil der optimalen Regel  $M[v_0]$  auf
- In *Action*-Teile eingebettete `$action [ ]`-Aufrufe beziehen sich stets auf die *Action*-Teile der jeweils kostenoptimalen Regel  $R^{opt}$

# Komplexeres Beispiel (1)

```

dreg: tpm_BinaryExpPLUS ( dreg, dreg ) {
    $cost[0] = $cost[2] + $cost[3] + 1;
}={
    if ( target.empty() ) target = getNewRegister();
    string r1( $action[2]("") ), r2( $action[3]("") );
    cout << "ADD " << target << ", " << r1 << ", " << r2 << endl;
    return target;
};

dreg: tpm_BinaryExpMULT ( dreg, dreg ) {
    $cost[0] = $cost[2] + $cost[3] + 1;
}={
    if ( target.empty() ) target = getNewRegister();
    string r1( $action[2]("") ), r2( $action[3]("") );
    cout << "MUL " << target << ", " << r1 << ", " << r2 << endl;
    return target;
};
  
```

## Komplexeres Beispiel (2)

```

dreg: tpm_SymbolEXP {
    $cost[0] = $1->getExp()->getSymbol().isGlobal() ?
        COST_INFINITY : COST_ZERO;
}= {
    target = "r_" + $1->getExp()->getSymbol().getName();
    return target;
};
  
```

- Regel weist lokalen Variablen im DFT  $T$  ein virtuelles Register zu
- $\$1$  ist der durch Terminal-Symbol zu überdeckende Knoten von  $T$
- $\$1 \rightarrow \text{getExp}() \rightarrow \text{getSymbol}()$  liefert das Symbol / die Variable der IR
- Im Fall globaler Variablen liefert diese Regel Kosten  $\infty$  zurück
- Für lokale Variablen: Kosten 0, da kein Code erzeugt wird.

## Komplexeres Beispiel (3)

C-Fragment  $a + (b * c)$  mit DFT  $T$   
wird durch Regeln

dreg: `tpm_SymbolExp`

dreg: `tpm_BinaryExpPLUS ( dreg, dreg )`

dreg: `tpm_BinaryExpMULT ( dreg, dreg )`

überdeckt.

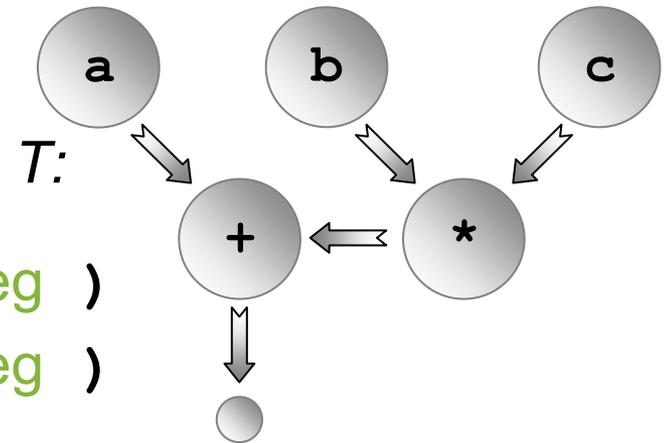
Kosten für  $T$ :

$$C[+] = C[a] + C[*] + 1 = C[a] + (C[b] + C[c] + 1) + 1 = 2$$

Generierter Code für  $T$ :

```
MUL r_0, r_b, r_c
```

```
ADD r_1, r_a, r_0
```



## Komplexeres Beispiel (4)

```

typedef pair<string, string> regpair;
%declare<regpair> virtmul;

virtmul: tpm_BinaryExpMULT( dreg, dreg ) {
    $cost[0] = $cost[2] + $cost[3];
}={
    string r1( $action[2]("") ), r2( $action[3]("") );
    return make_pair( r1, r2 );
};
  
```

- Neues Nichtterminal `virtmul` repräsentiert Multiplikation in  $T$ , für die aber nicht direkt Code generiert werden soll
- Statt Code-Generierung wird ein Register-Paar zurückgegeben, das speichert, wo die Operanden der Multiplikation liegen
- Mangels Code-Generierung:  $C[v] = \text{Summe d. Operanden-Kosten}$

## Komplexeres Beispiel (5)

```

dreg: tpm_BinaryExpPLUS ( dreg, virtmul ) {
    $cost[0] = $cost[2] + $cost[3] + 1;
}={
    if ( target.empty() ) target = getNewRegister();
    string r1( $action[2]("") );
    regpair rp( $action[3]() );
    cout << "MADD " << target << ", " << r1 << ", " << rp.first
        << ", " << rp.second << endl;
    return target;
};
  
```

- Regel aktiv, falls zweiter Summand virtuelle Multiplikation ist
- *Dann:* Register-Paar des Nichtterminals **virtmul** anfordern, eine Multiply-Accumulate-Operation **MADD** (☞ Kapitel 1) generieren

## Komplexeres Beispiel (6)

C-Fragment  $a + (b * c)$  mit DFT  $T$   
 wird nun zusätzlich durch Regeln

dreg: `tpm_SymbolExp`

virtmul: `tpm_BinaryExpMULT ( dreg, dreg )`

dreg: `tpm_BinaryExpPLUS ( dreg, virtmul )`

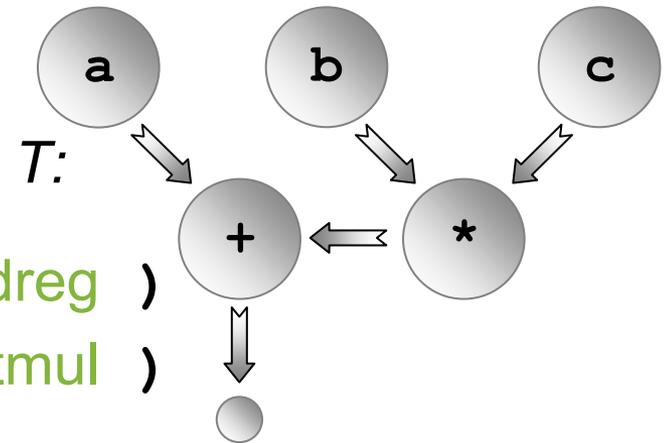
überdeckt.

Kosten für  $T$ :

$$C[+] = C[a] + C[*] + 1 = C[a] + (C[b] + C[c]) + 1 = 1$$

Generierter Code für  $T$ :

```
MADD r_0, r_a, r_b, r_c
```



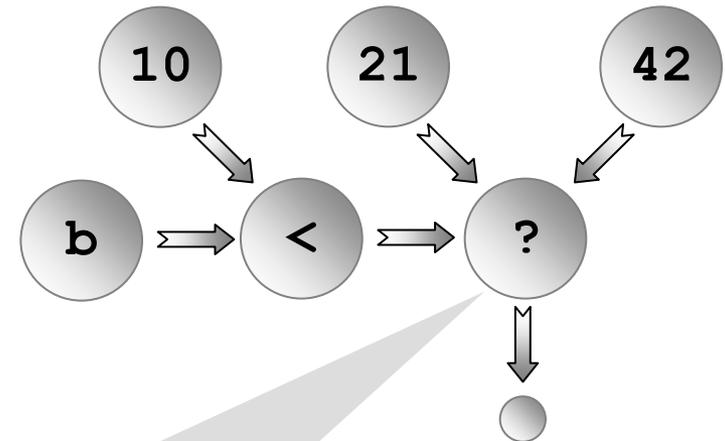
# Aufruf von Action-Teilen mit Parametern

```
%declare<string> dreg<string target>;
```

- Ein string kann in Parameter **target** übergeben werden, um einem Action-Teil vorzugeben, in welchem Datenregister dieser seinen Ziel-Operanden abzulegen hat.

C-Fragment (**b** < 10) ? 21 : 42

- Das Ergebnis des ?-Operators muss in einem **dreg** liegen.
- Beide Teilbäume links und rechts von „:“ sind in das selbe **dreg** auszuwerten.
- Die Regel für ? muss beiden Teilbäumen das gleiche Zielregister vorgeben!



```
target = getNewRegister();
$action[3](target);
$action[4](target);
```

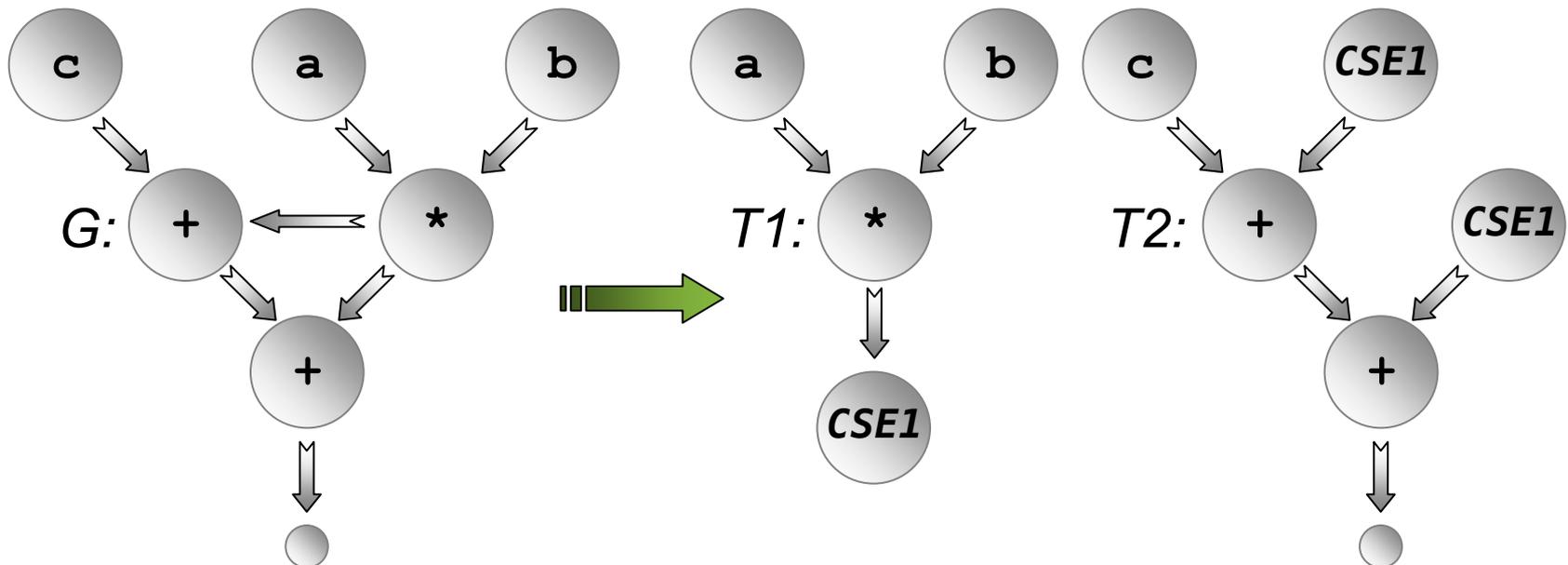
# Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
  - Einführung
  - Baum-Überdeckung mit Dynamischer Programmierung
  - Diskussion
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

# Nachteile von Tree Pattern Matching

## Zerlegung von DFGs in DFTs führt zu sub-optimalem Code:

- Beispiel  $a + (b * c)$  aus vorigem Abschnitt wird durch TPM optimal auf **MADD**-Operation abgebildet.
- Aber was passiert z.B. bei  $e = a*b; \dots (c + e) + e \dots ?$



# Nachteile von Tree Pattern Matching

## Optimale Baum-Überdeckung von $T1$ und $T2$ :

- Würde in insgesamt *drei* Maschinen-Operationen resultieren.
- 1 Multiplikation zur Überdeckung von  $T1$ , 2 Additionen für  $T2$

```
MUL r_0, r_a, r_b  
ADD r_1, r_c, r_0  
ADD r_2, r_1, r_0
```

## Optimale Graph-Überdeckung von $G$ :

- Würde in insgesamt *zwei* Maschinen-Operationen resultieren.
- 2 Multiply-Additionen für  $G$

```
MADD r_0, r_c, r_a, r_b  
MADD r_1, r_0, r_a, r_b
```

# Diskussion von Tree Pattern Matching

## Vorteile:

- Lineare Laufzeit-Komplexität
- Optimalität für Datenflussbäume
- „Leichte“ Realisierung mit Hilfe von Baum-Grammatiken und Code-Generator-Generatoren

## Nachteile:

- TPM schlecht geeignet für Prozessoren mit sehr heterogenen Registersätzen
- TPM ungeeignet für Prozessoren mit Parallel-Verarbeitung

# TPM und Heterogene Registersätze

## Zerlegung von DFGs in DFTs:

- Sei  $T$  ein DFT, der eine CSE  $C$  berechnet;  $T'$  die DFTs, die  $C$  benutzen.
- Nach Überdeckung von  $T$  muss der für  $T$  generierte Code  $C$  irgendwo zwischenspeichern, und alle  $T'$  müssen  $C$  zur Benutzung aus dem Zwischenspeicher laden.
- Da  $T$  und  $T'$  völlig unabhängig voneinander überdeckt werden, kann während der Code-Generierung von  $T$  nicht berücksichtigt werden, wo die  $T'$   $C$  optimalerweise erwarten.
- Wenn  $T$   $C$  in einem Teil des heterogenen Registerfiles speichert,  $T'$   $C$  aber in einem anderen Teil erwartet, sind zusätzliche Register-Transfers notwendig!

# TPM und Parallele Prozessoren

## Additives Kostenmaß von Tree Pattern Matching:

- Kosten eines DFTs  $T$  mit Wurzel  $v$  sind Summe der Kind-Kosten plus Kosten für  $v$  selbst.
- *Action*-Teil für  $T$  erzeugt i.d.R. eine Maschinen-Operation
- *Erinnerung*: Parallele Prozessoren führen mehrere Maschinen-Operationen, die in einer Maschinen-Instruktion gebündelt sind, parallel aus.
- ☞ Additives TPM-Kostenmaß geht implizit davon aus, dass alle erzeugten Operationen rein sequentiell ausgeführt werden!
- ☞ Da TPM bei der Kostenberechnung nicht berücksichtigt, dass Operationen parallel zu Instruktionen gruppiert werden können, wird erzeugter Code schlechte parallele Performance haben!

# Literatur

## Tree Pattern Matching:

- A. Aho, S. Johnson, *Optimal Code Generation for Expression Trees*, Journal of the ACM 23 (3), 1976.
- A. Aho, M. Ganapathi, S. Tjiang, *Code Generation Using Tree Matching & Dynamic Programming*, ACM ToPLaS 11 (4), 1989.

## Code-Generator-Generatoren:

- *ICD-CG code generator generator*,  
<http://www.icd.de/es/icd-cg>, 2010.
- *iburg. A Tree Parser Generator*,  
<http://code.google.com/p/iburg>, 2010. inkl.  
C. W. Fraser, D. R. Hanson, T. A. Proebsting, *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems 1 (3), Sep 1992.

# Zusammenfassung

- **Instruktionsauswahl**
  - Umsetzung eines DFGs durch zu erzeugendes Programm
  - Code-Generator-Generatoren
- **Tree Pattern Matching**
  - Zerlegung von DFGs in Datenfluss-Bäume
  - Linearzeit-Algorithmus zur optimalen Überdeckung von DFTs
  - Aufbau und Struktur von Baum-Grammatiken
- **Diskussion**
  - Tree Pattern Matching nur für reguläre Prozessoren gut
  - Nachteilig für Architekturen mit heterogenen Registern
  - Nachteilig für Prozessoren mit Parallel-Verarbeitung

