Compiler für Eingebettete Systeme (CfES)

Sommersemester 2009

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

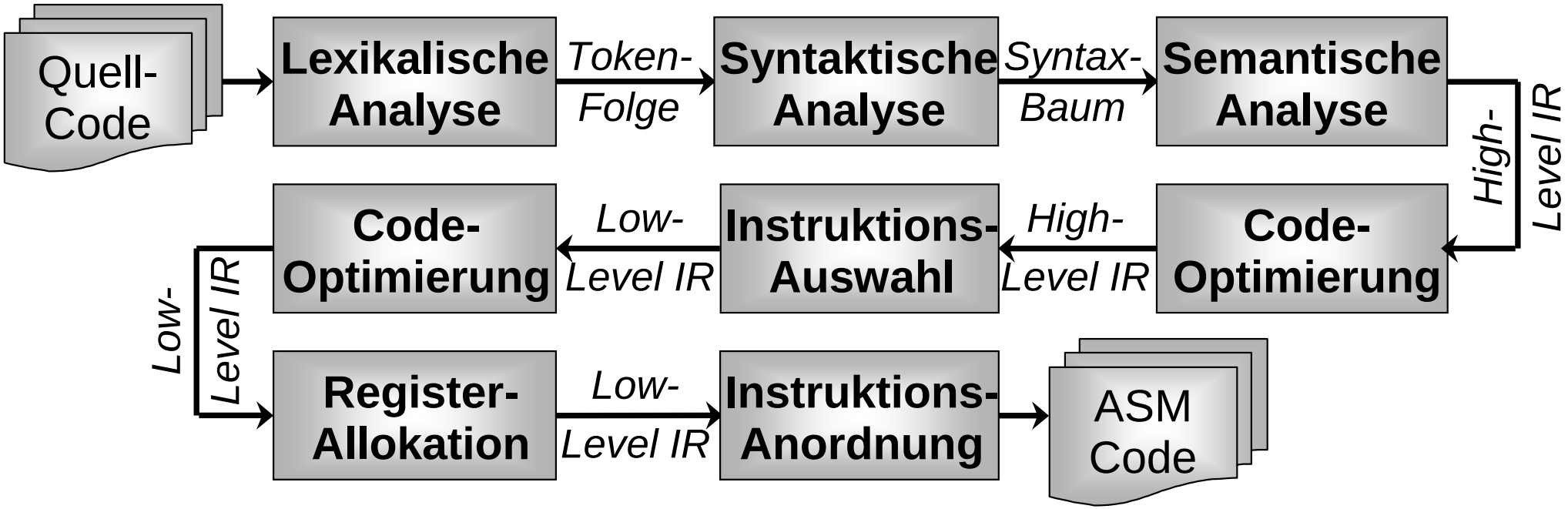
Kapitel 5

Instruktionsauswahl

Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- **Kapitel 5: Instruktionsauswahl**
  - Einführung
  - Baum-Überdeckung mit Dynamischer Programmierung
  - Graph-basierte Instruktionsauswahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung
- Kapitel 9: Ausblick

Rolle der Instruktionsauswahl



Instruktionsauswahl:

- 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.

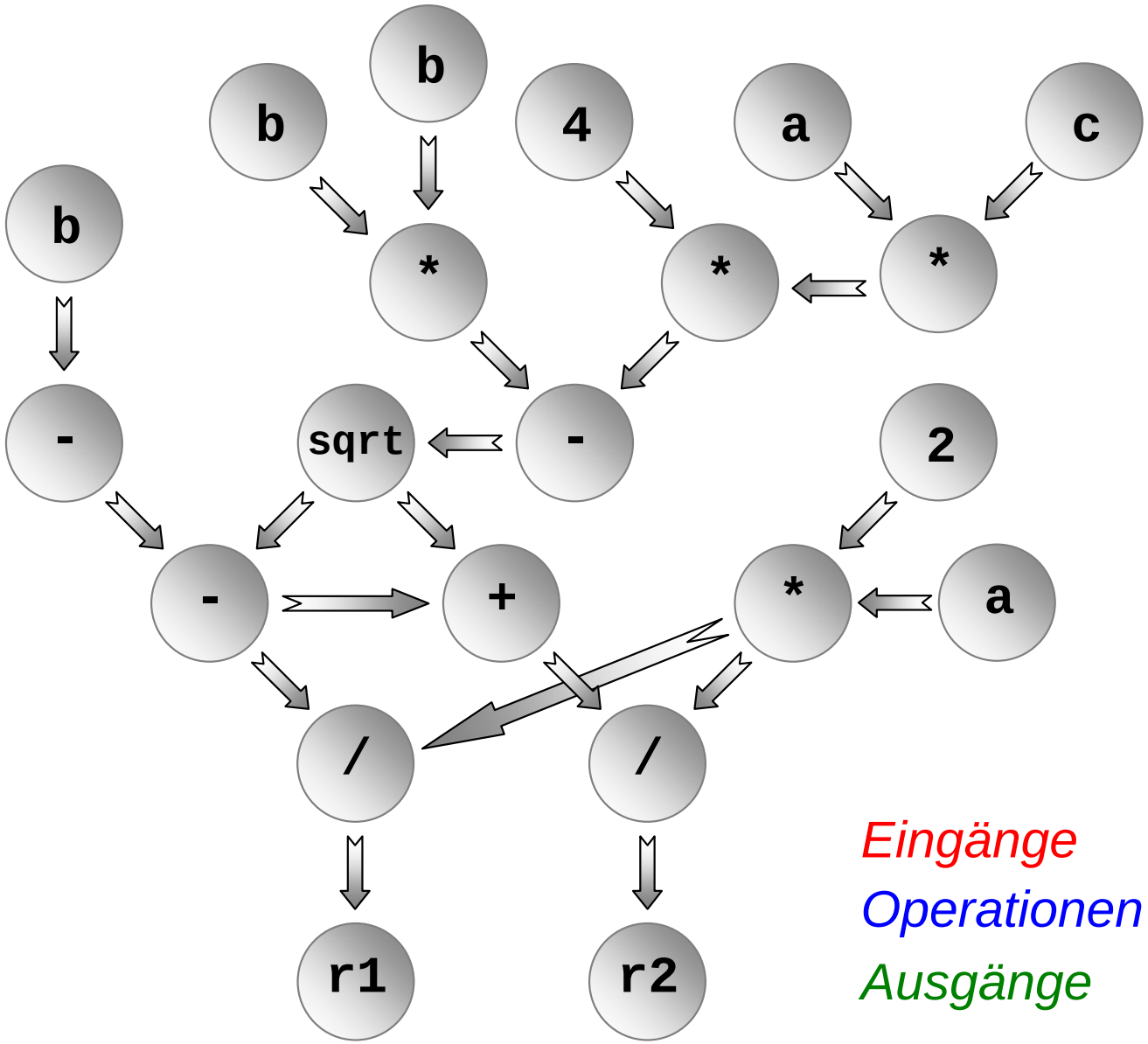
Definition *(Datenflußgraph)*:

Sei $B = (I_1, ..., I_n)$ ein Basisblock. 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 $I_1, ..., I_n$
  - 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.

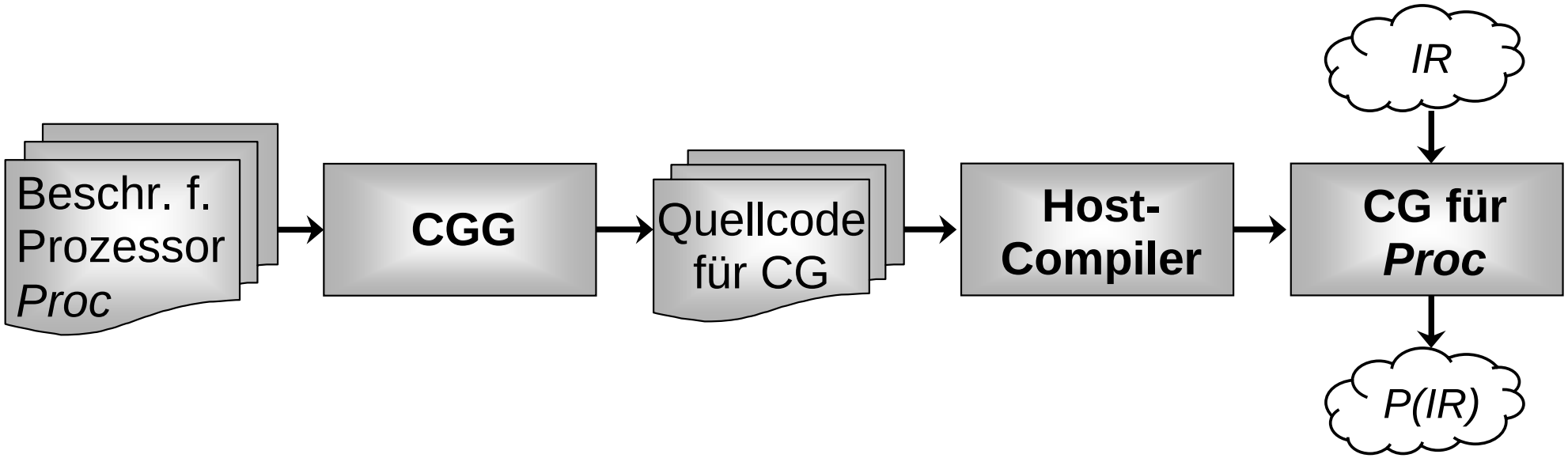
Implementierung eines Code-Generators:

- 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

- Sog. *Meta-Programme*, d.h. Programme, die Programme als Ausgabe erzeugen.
- 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 für Eingebettete Systeme
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- **Kapitel 5: Instruktionsauswahl**
  - Einführung
  - Baum-Überdeckung mit Dynamischer Programmierung
  - Graph-basierte Instruktionsauswahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung
- Kapitel 9: Ausblick

Tree Pattern Matching *(TPM)*

Motivation:

- Überdeckung von Datenflußgraphen polynomiell reduzierbar auf 3-SAT

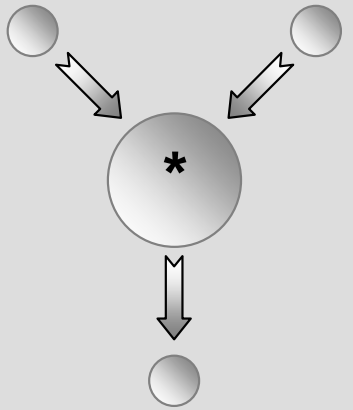
  *[**J. Bruno, R. Sethi,** Code generation for a one-register machine, Journal of the ACM 23(3), Jul 1976]*

- 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, ..., T_N
  - Für jeden DFT T_i:
    - $P = P \cup \{$ 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)*.

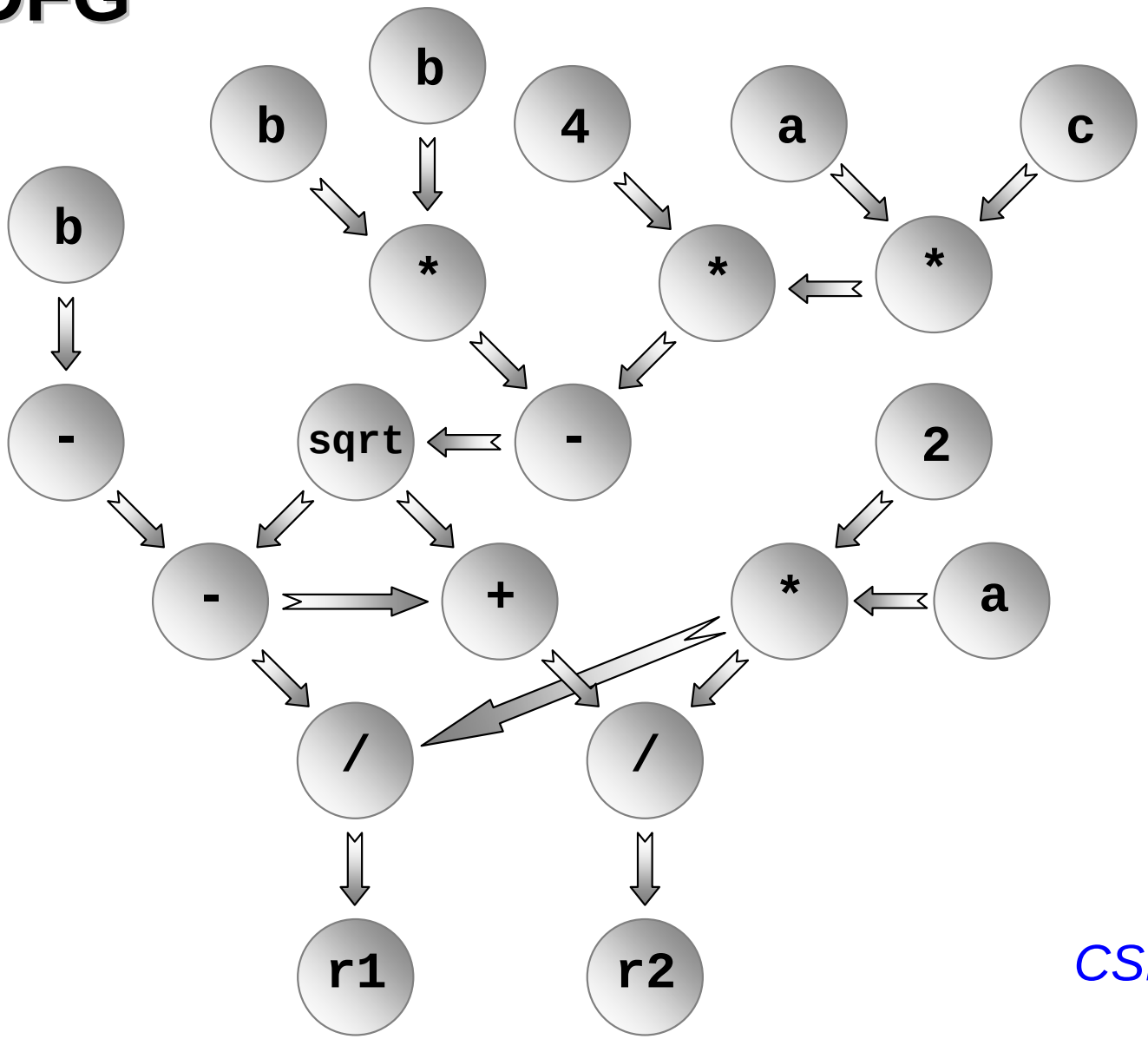
Definition *(Datenflußbaum)*:

Ein Datenflußgraph *DFG* = (*V*, *E*) ohne CSEs heißt *Datenflußbaum (Data Flow Tree, DFT)*.

☞ **DFG-Zerlegung:**

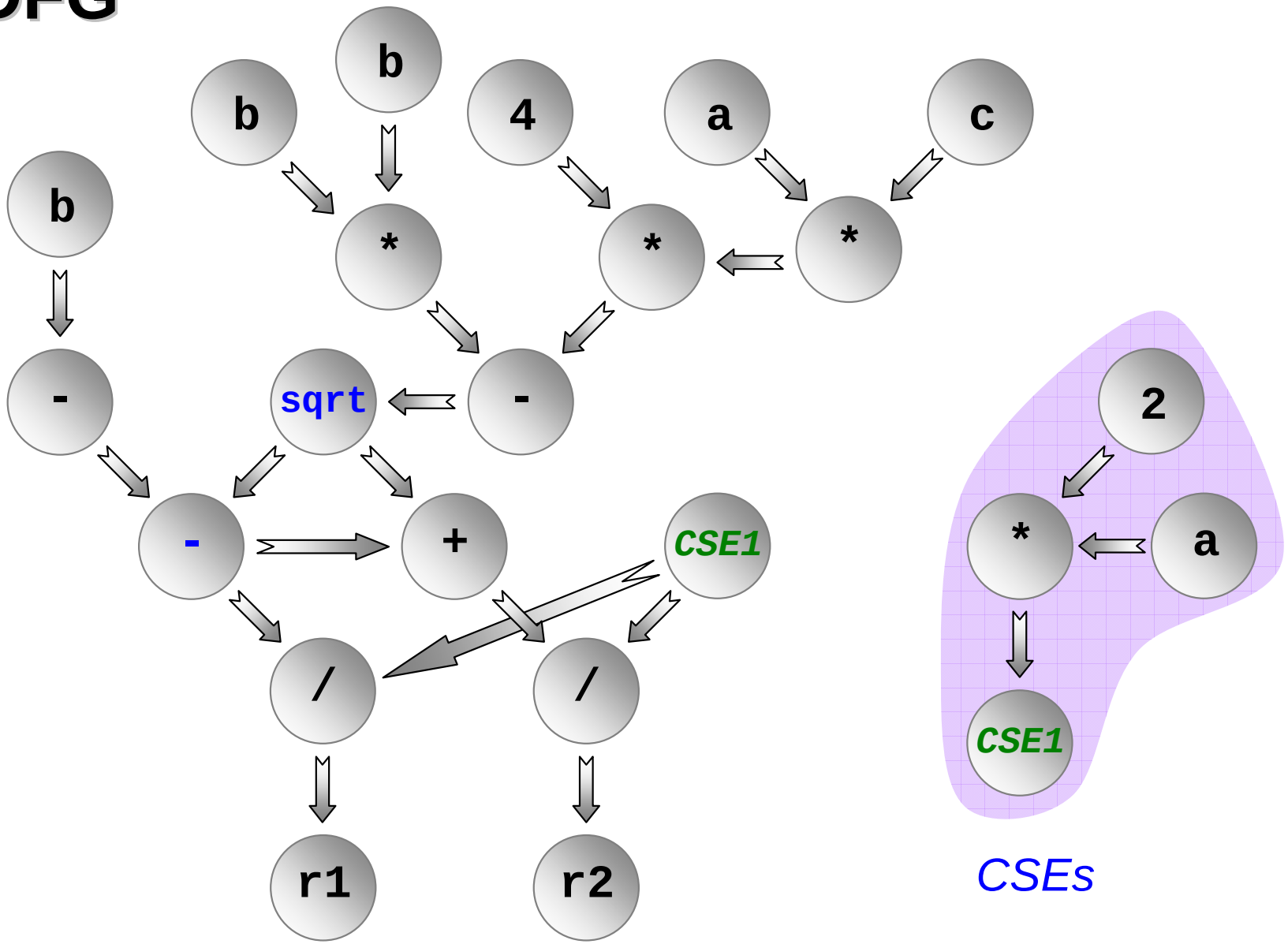
- Aufspaltung des DFGs in DFTs entlang der CSEs.
- Für jede CSE: Hilfsknoten in resultierenden Bäumen einfügen.

Beispiel-DFG

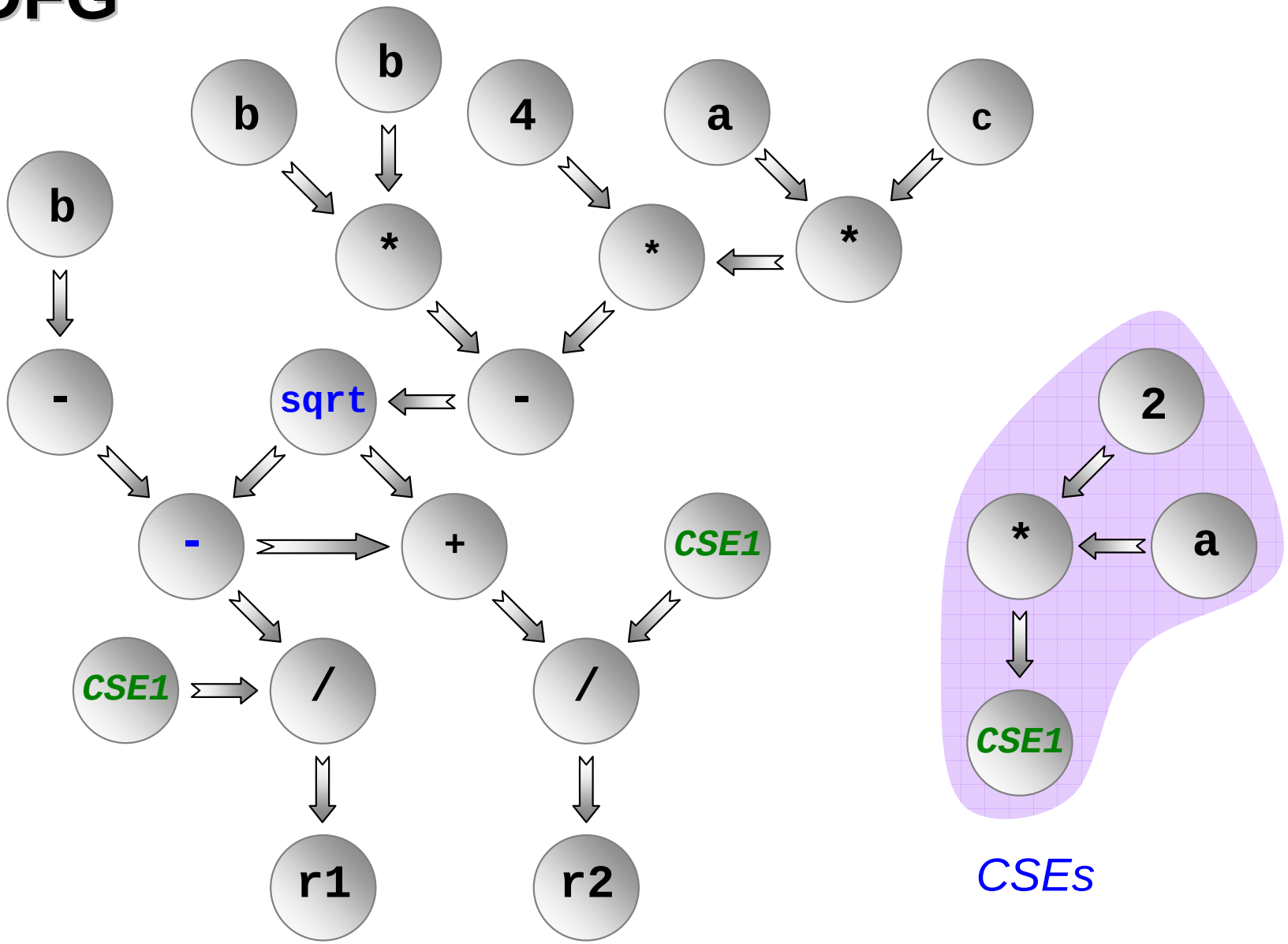


CSEs

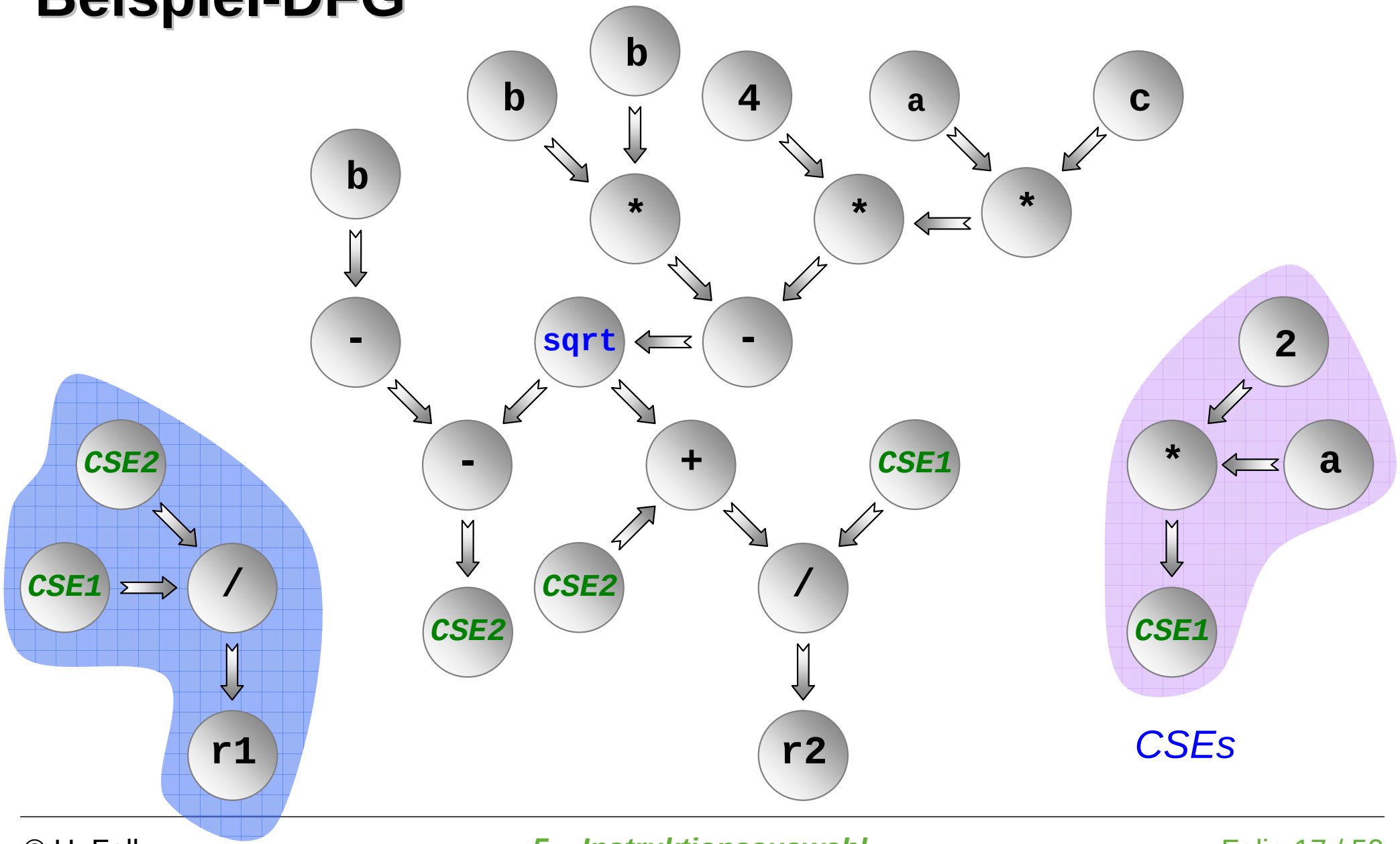
Beispiel-DFG



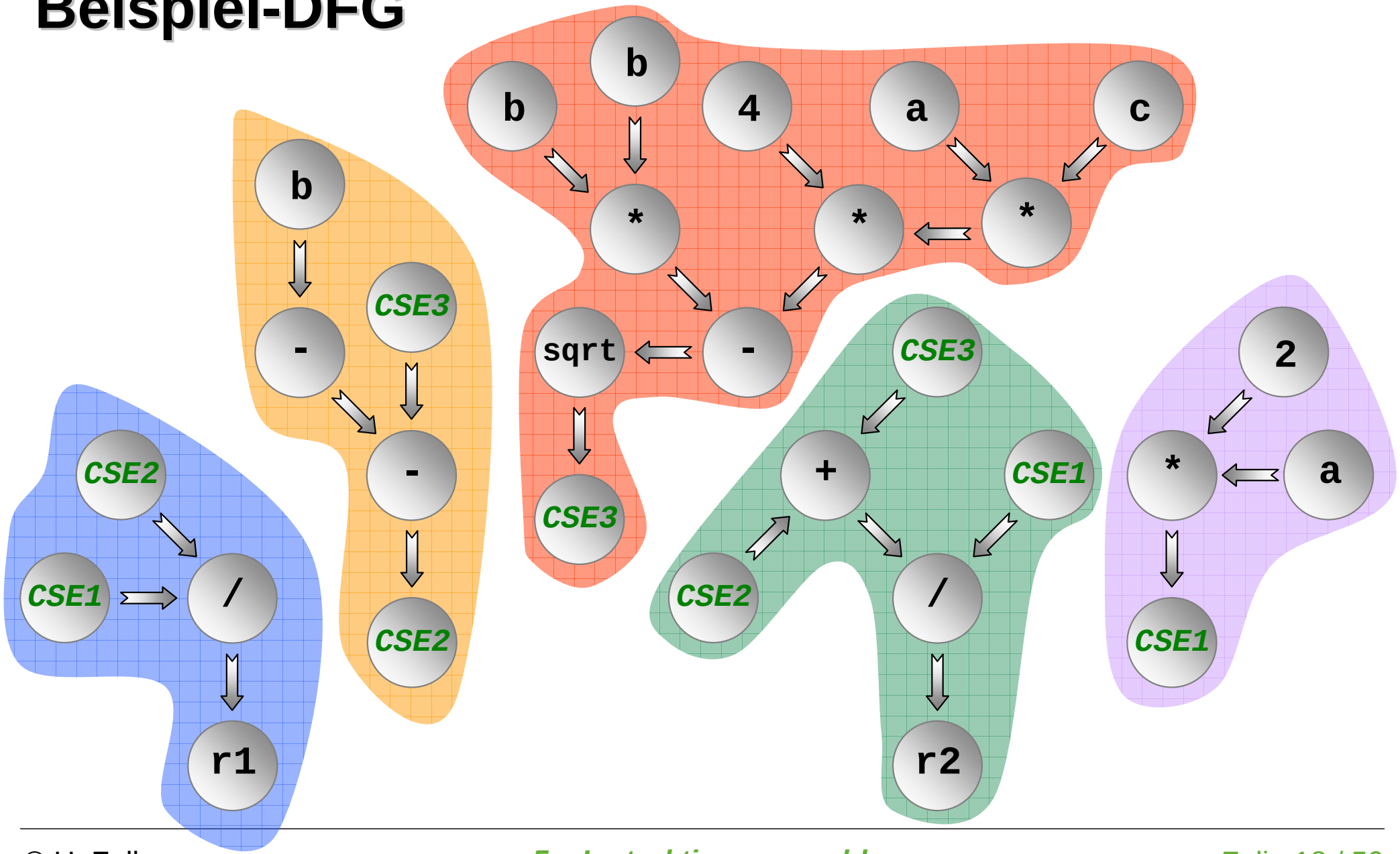
Beispiel-DFG



Beispiel-DFG



Beispiel-DFG



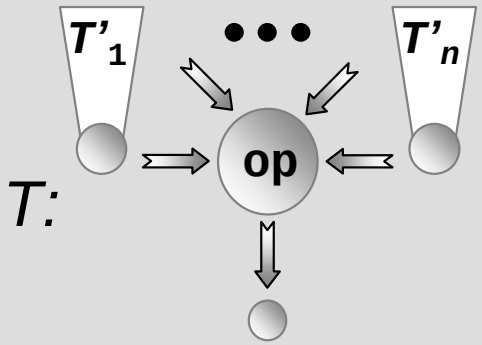
Baum-Überdeckung *(Tree Cover)*

Definition *(Baum-Überdeckung durch Operationsfolge)*:

Sei $T = (V, E)$ ein DFT, $S = (o_1, ..., o_N)$ eine Folge von Maschinenoperationen. Die letzte Operation o_N habe das Format $\mathbf{d} \leftarrow \mathbf{op}(\mathbf{s_1}, ..., \mathbf{s}_n)$. $S'_1, ..., S'_n$ bezeiche die Teil-Folgen von S, die jeweils die Operanden $\mathbf{s_1}, ..., \mathbf{s}_n$ von O_N berechnen.

S überdeckt T genau dann, wenn

- Operator **op** der Wurzel von T entspricht, T sich also wie folgt schreiben läßt:



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

Beispiel für Überdeckungen

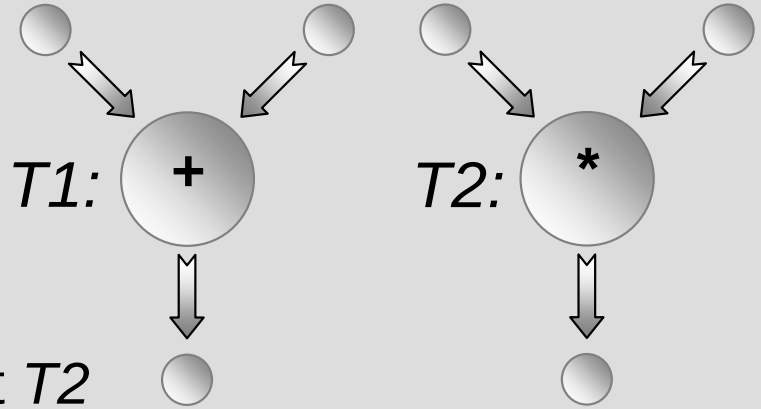
- **TriCore-Befehlssatz:**

```
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*

- **Datenflußbäume:**



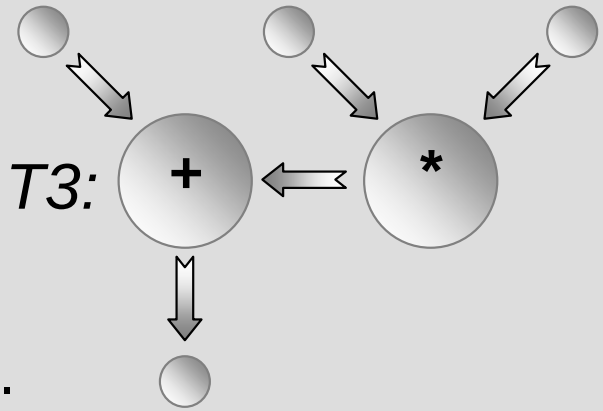
☞ *Auch klar:* Operationsfolge

`mul %d10, %d11, %d12`

`add %d4, %d8, %d10` überdeckt *T3*

☞ *Zusätzlich:* Einelementige Folge

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



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)

Datenstrukturen:

- Array C: minimale Kosten pro Knoten $v \in V$
- Array M: optimale Maschinen-Operation und optimale Operanden-Reihenfolge pro Knoten $v \in V$

Phase 1 – Initialisierung:

- Für alle Knoten $v \in V$: $C[v] = \begin{cases} 0, \text{ falls } v \text{ Eingang von } T \\ \infty \text{ sonst} \end{cases}$
- Für alle Knoten $v \in V$: $M[v] = \emptyset$

Tree Pattern Matching Algorithmus (2)

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 $T'_1, ..., T'_n$ mit jeweiligen Wurzeln $v'_1, ..., v'_n$ gemäß Tree Cover-Definition
    - Für alle Permutationen π über (1, ..., *n*):
      - Berechne minimale Kosten für Knoten *v*:

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

        (☞ dyn. Programmierung)

        $M[v]$ = Paar (o, π), das zu minimalem $C[v]$ führt

Tree Pattern Matching Algorithmus (3)

Phase 3 – generateCode(DFT *T*):

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

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

Laufzeit-Komplexität von TPM

Annahme:

- Befehlssatz eines Prozessors sei fest vorgegeben
- ☞ Größe der Menge *O* konstant
- ☞ Anzahl der Permutationen π 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

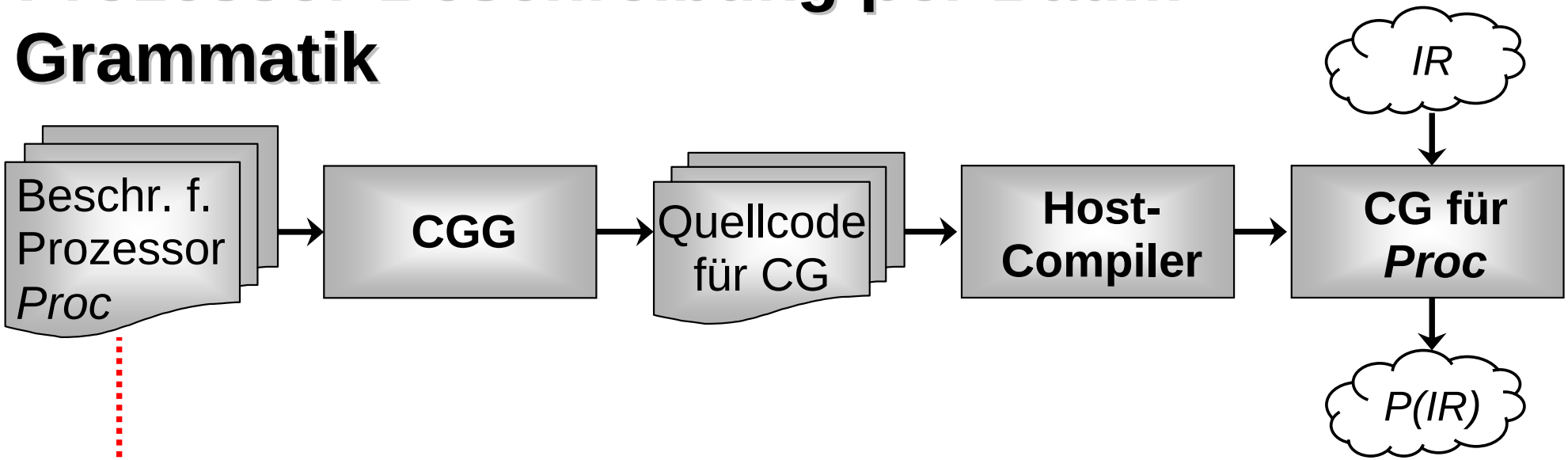
Präsentierter TPM-Algorithmus generisch formuliert.

☞ ***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*?

Prozessor-Beschreibung per Baum-Grammatik



- Grammatik *G*, die für Teilbäume eines DFTs Maschinen-operationen generiert
- Jede einzelne Regeln 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, ..., R_r
- Jede Regel R_i hat eine *Signatur*, bestehend aus Terminal- und Nichtterminal-Symbolen:

  <nonterminal$_{i,0}$>**:** **<terminal$_{i,1}$>(** <nonterminal$_{i,2}$>**, …,** <nonterminal$_{i,n}$> **)**

  (Angabe von Nichtterminalen in (...) optional)

  (Sog. Kettenregeln <nonterminal$_{i,0}$>**:** <nonterminal$_{i,1}$> *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-Symbold müssen in Baum-Grammatik *G* deklariert sein
- Grundlegender Aufbau des Files zur Beschreibung von *G*:

```
%{          // Präambel
%}

%term <terminal_1>
%term <terminal_2>
...

%declare <nonterminal_0>;
```

```
%declare <nonterminal_1>;
...

%%

Regel_1;
Regel_2;
...

%%
```

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:

<nonterminal$_{i,0}$>: **<terminal$_{i,1}$>(** <nonterminal$_{i,2}$>, …, <nonterminal$_{i,n}$> **)**

```
{
        // 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 $\text{nonterminal}_{i,0}$ Kosten zu, die entstehen, wenn R_i zur Überdeckung von $\texttt{terminal}_{i,1}$ 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 ∞ 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 ∞–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 **`$cost[j]`** zum Zugriff auf Kosten von nonterminal$_{i,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`**$_{i,1}$ ist.
- *Action*-Teil kann beliebigen benutzerdefinierten C/C++-Code zur Code-Generierung enthalten.
- Verwendung des feststehenden Schlüsselwortes **`$action[`*`j`*`]`** zum Ausführen der *Action*-Teile für Operanden nonterminal$_{i,j}$
- Nichtterminal-Symbole können mit Parametern und Rückgabe-werten 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 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

<nonterminal$_{i,0}$>: **<terminal$_{i,1}$>(** <nonterminal$_{i,2}$>, **…,** <nonterminal$_{i,n}$> **)**

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 <nonterminal$_{i,j}$> erzeugen ($1 \leq j \leq n$), und
- die Kosten von R_i kleiner als ∞ 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;
};
```

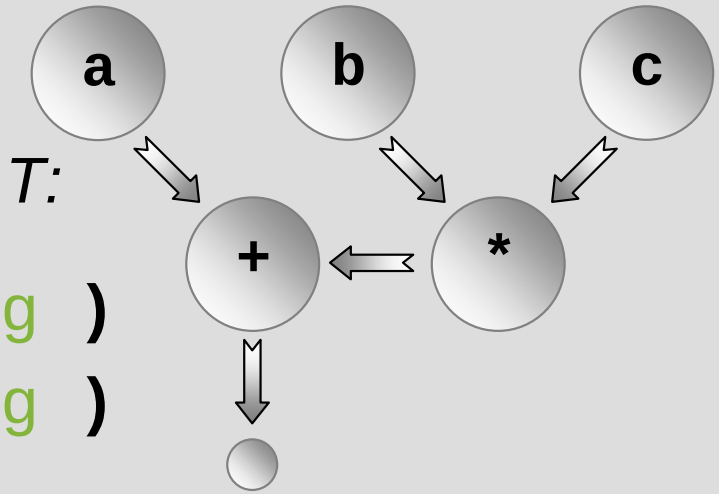
Komplexeres Beispiel (2)

```
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->getExp()->getSymbol()** liefert das Symbol / die Variable der IR
- Im Fall globaler Variablen liefert diese Regel Kosten ∞ 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*:

$C[\mathbf{+}] = C[\mathbf{a}] + C[\mathbf{*}] + 1 = C[\mathbf{a}] + (C[\mathbf{b}] + C[\mathbf{c}] + 1) + 1 = 2$

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*] = Summe d. Operanden-Kosten

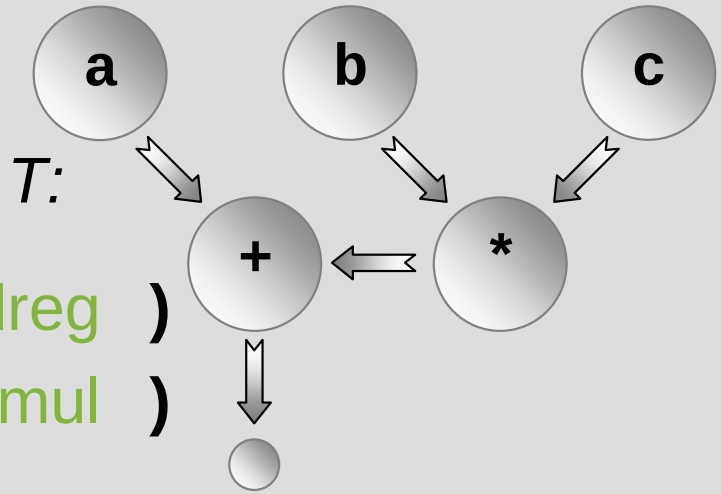
Komplexeres Beispiel (5)

```
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 wird Register-Paar des Nichtterminals virtmul angefordert und eine kombinierte **MADD**-Operation generiert

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*:

$C[\mathbf{+}] = C[\mathbf{a}] + C[\mathbf{*}] + 1 = C[\mathbf{a}] + (C[\mathbf{b}] + C[\mathbf{c}]) + 1 = 1$

Generierter Code für *T*:

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

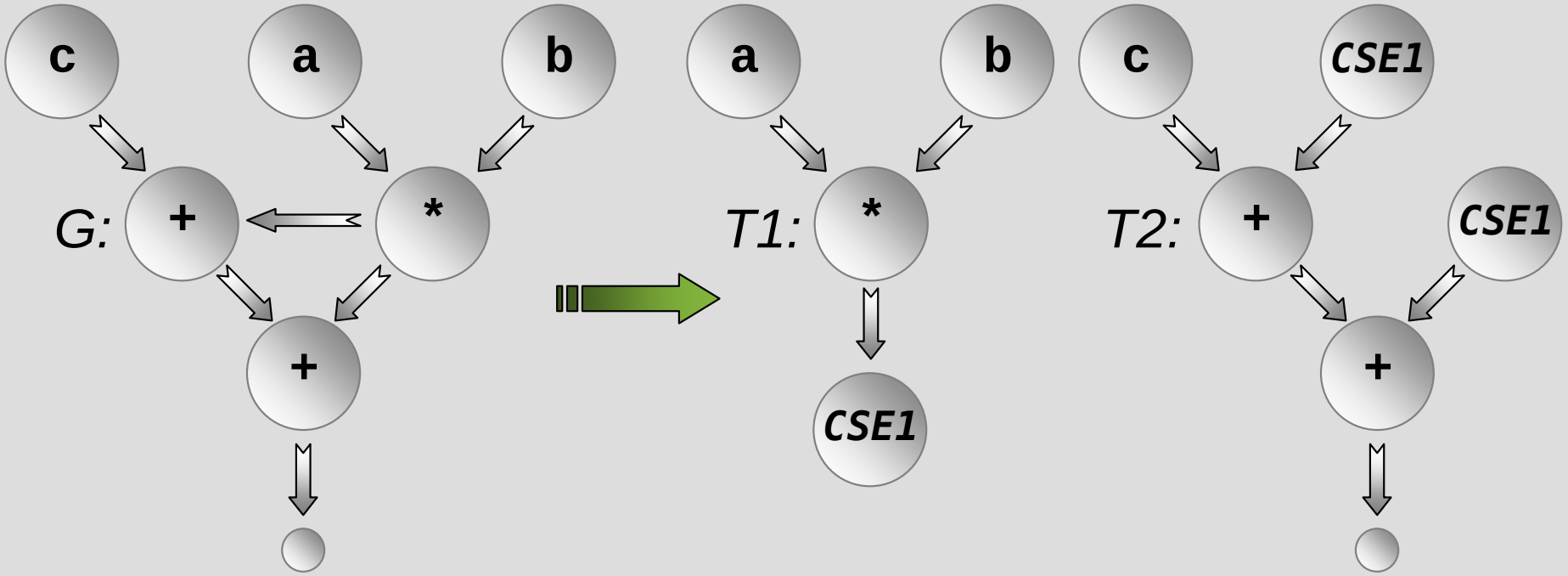
Gliederung der Vorlesung

- Kapitel 1: Compiler für Eingebettete Systeme
- Kapitel 2: Interner Aufbau von Compilern
- Kapitel 3: Prepass-Optimierungen
- Kapitel 4: HIR Optimierungen und Transformationen
- **Kapitel 5: Instruktionsauswahl**
  - Einführung
  - Baum-Überdeckung mit Dynamischer Programmierung
  - Graph-basierte Instruktionsauswahl
- Kapitel 6: LIR Optimierungen und Transformationen
- Kapitel 7: Register-Allokation
- Kapitel 8: Compiler zur $WCET_{EST}$-Minimierung
- 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; … (c + e) + e …`**?



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
```

Ansatz 1: Erweiterung von TPM

Grundlegende Vorgehensweise:

- Beibehaltung von Baum-Grammatiken zur Prozessor-Spezifikation
- Grundlegende Beibehaltung des TPM-Algorithmus mit beiden Phasen der Kostenberechnung und Code-Generierung
- Erweiterung beider Phasen, so daß beim Durchlauf zu überdeckender DFGs jeder Knoten nur einmal besucht wird

*[**M. Ertl,** Optimal Code Selection in DAGs, Symposium on the Principles of Programming Languages, Jan 1999]*

Ansatz 1: TPM für DFGs

Erweiterte Kostenberechnung:

- Durchlauf des DFGs *G* von Quellen zu Senken
- Verwaltung eines Flags **`visited[`***v***`]`** pro Knoten von *G*
- Rekursiver Besuch nachfolgender Knoten *v'* nur dann, wenn **`visited[`***v'***`]`** nicht gesetzt

Erweiterte Code-Generierung:

- Durchlauf des DFGs *G* von allen (potentiell vielen) Senken zu Quellen
- Verwaltung eines Flags **`visited[`***v***`]`** pro Knoten von *G*
- Rekursiver Besuch nachfolgender Knoten *v'* nur dann, wenn **`visited[`***v'***`]`** nicht gesetzt

Diskussion von Ansatz 1

Vorteile:

- Nach wie vor lineare Laufzeit-Komplexität
- Beibehaltung bisheriger Baum-Grammatiken
- Optimalität für DFGs und gewisse Klassen von Baum-Grammatiken

Nachteile:

- Verfahren (ebenso wie reines TPM) schlecht geeignet für Prozessoren mit sehr heterogenen Registersätzen
- Verfahren (ebenso wie reines 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!

Ansatz 2: Genetische Code-Selektion

Mögliche Alternativen bei Code-Generierung für DFGs:

- Ein DFG-Knoten kann durch mehrere alternative Maschinen-Operationen implementiert werden.
- Je nach gewählter Maschinen-Operation kann ein DFG-Knoten auf alternativen Funktionseinheiten ausgeführt werden.
- Je nach gewählter Operation und Funktionseinheit können Quell- und Ziel-Operanden des DFG-Knotens in verschiedenen heterogenen Registern liegen.
- Je nach gewählter Operation und Funktionseinheit können DFG-Knoten einzelnen Zeiteinheiten *(Kontrollschritten)* zugeordnet werden. Alle Operationen eines Kontrollschritts werden zu einer Instruktion gebündelt und parallel ausgeführt.

Ansatz 2: Chromosom-Repräsentation

- **Jedes Gen eines Chromosoms repräsentiert genau einen DFG-Knoten *v*.**
- **Der Inhalt eines Gens codiert genau die möglichen Alternativen zur Code-Generierung für *v*:**
  - Die für *v* gewählte Maschinen-Operation
  - Die für *v* gewählte Funktionseinheit
  - Die für *v* gewählten Register
  - Der für *v* gewählte Kontrollschritt
  - ...

*[**M. Lorenz,** Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen, Jan 2003]*

Ansatz 2: Variation

- **Crossover-Operator kann ungültige Individuen erzeugen**
- **Nach Crossover stattfindende Mutation führt ggfs. Reparaturen durch:**
  - Probabilistische Neubelegung einzelner Gene ungültiger Individuen
  - Herstellen der *Kantenkonsistenz*: Gewährleisten, daß Knoten v_1 nur Register definiert, die datenabhängige Knoten v_2 auch als Argument benutzen können.
  - Herstellen der *Knotenkonsistenz*: Gewährleisten, daß für jeden Knoten nur solche Quell- und Zielregister gewählt werden, die gemäß Prozessor-Befehlssatz legale Maschinen-Operationen darstellen.

Ansatz 2: Fitness

- **Mehrkriterielle Fitnessfunktion: Parallele Ausführungszeit und Energieverbrauch**
- **Bewertung der parallelen Ausführungszeit:**
  - Zähle Anzahl verschiedener Kontrollschritte eines Individuums
- **Bewertung des Energieverbrauches eines Individuums:**
  - Summiere Basis- und Interinstruktionskosten aller Maschinen-Instruktionen eines Individuums
- **Fitness eines Individuums *I*:**
  - Gewichtete Summe von Ausführungszeit (Kontrollschritte) und Energieverbrauch:

$$fitness(I) = \omega_{cs} * cs_I + \omega_{en} * en_I$$

Gewichtung ω gibt an, ob für Zeit oder Energie optimiert wird.

Diskussion von Ansatz 2

Vorteile:

- Ebenfalls lineare Laufzeit-Komplexität
  (Anzahl der Iterationen des Genetischen Algorithmus konstant, Komplexität der Fitness-Funktion linear in Chromosom-Länge)
- *Phasengekoppelte* Durchführung von Code-Selektion, Registerallokation und Scheduling
- Multikriterielle Optimierung (Ausführungszeit vs. Energie)

Nachteile:

- Keine Aussage bezüglich Optimalität möglich
- Prozessor-Beschreibung nicht mehr in Form übersichtlicher Baum-Grammatiken möglich

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.
  - *ICD-CG code generator generator*, www.icd.de/es/icd-cg, 2009.
- **Graph-basierte Instruktionsauswahl:**
  - M. Ertl, *Optimal Code Selection in DAGs*, Symposium on the Principles of Programming Languages, Jan 1999.
  - M. Lorenz, *Performance- und energieeffiziente Compilierung für digitale SIMD-Signalprozessoren mittels genetischer Algorithmen*, Dissertation, Dortmund 2003.

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
- **Graph-basierte Instruktionsauswahl**
  - Tree Pattern Matching nur für reguläre Prozessoren gut
  - Erweiterungen betrachten irreguläre Architekturen
  - Phasenkopplung