9. Kapitel

klassische Codeerzeugung

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 BEG
5. Die letzten 10%

9. Die Synthesephase

Aufgabe: attributierter Strukturbaum → ausführbarer Maschinencode

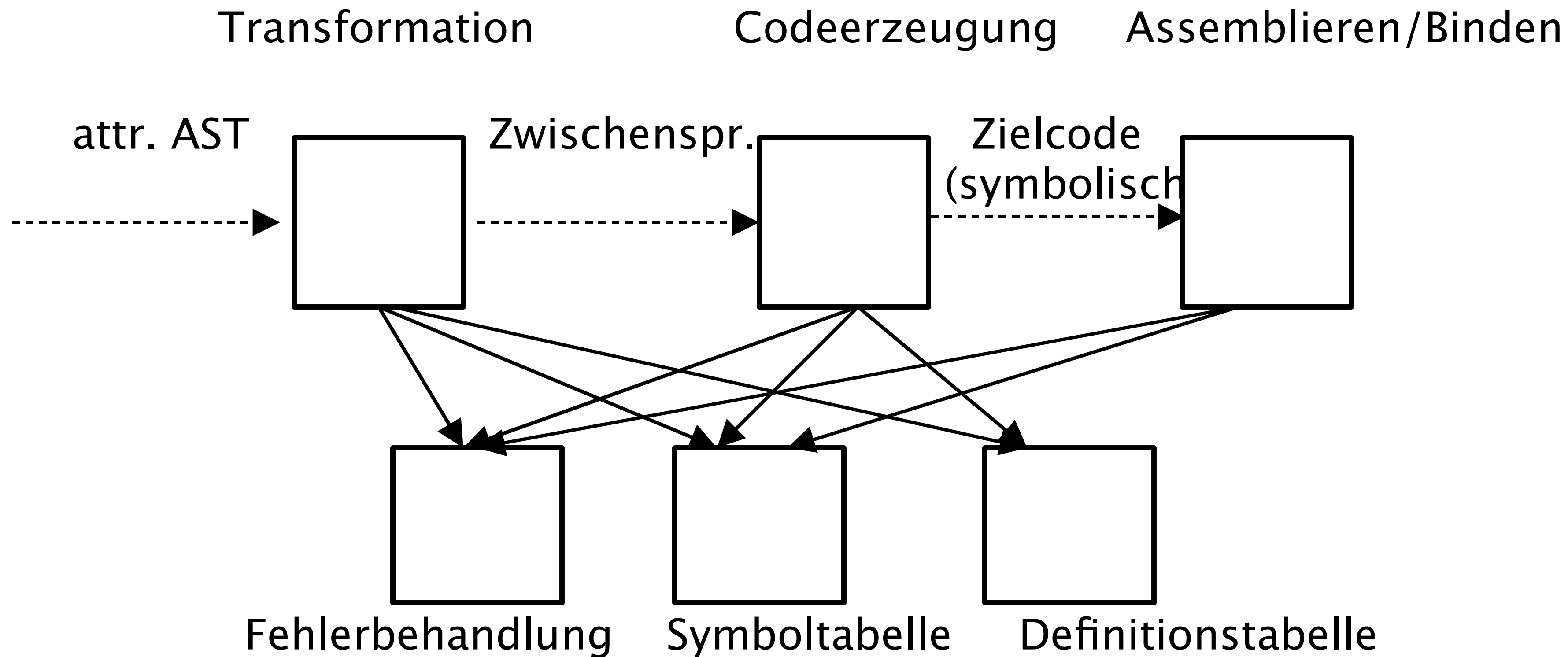
Problem:

- außer bei Codeerzeugung für die abstrakte Quellsprachenmaschine (*QM*), eine Kellermaschine, sind alle Aufgaben ,,guter" Codeerzeugung NP-vollständig
  - Qualität also nur näherungsweise erreichbar

Zerlegung der Synthese:

- Abbildung, d.h. Transformation/Optimierung: Code für abstrakte Zielmaschine *ZM* (ohne Ressourcenbeschränkung) herstellen und optimieren, Repräsentation als Zwischensprache *IL*
- Codeerzeugung: Transformation *IL* → symbolischer Maschinencode
  - unter Beachtung Ressourcenbeschränkungen
- Assemblieren/Binden: symbolische Adressen auflösen, fehlende Teile ergänzen, binär codieren

9. Codeerzeugung



9. Codeerzeugung – Aufgaben

Teilaufgaben müssen Gegebenheiten der Zielmaschine berücksichtigen

- Ausführungsreihenfolge
  - Anordnung der Zweige einer Ausdrucksberechnung im Hinblick auf Registerverbrauch
- Befehlsauswahl (code selection)
  - Bestimmung von konkreten Maschinen-Befehlen für die Operationen der Zwischensprache
  - Hinweis: Dieser Prozeß heißt auch Codeauswahl oder Codegenerierung
- Befehlsanordnung (scheduling)
  - Bestimmung der Ausführungsreihenfolge für Befehle
  - Festlegung einer Anordnung der Grundblöcke im Speicher
- Betriebsmittelzuteilung
  - im wesentlichen Registerzuteilung (register allocation)
- Cacheoptimierung (?)

9. Wiederholung: Zwischensprache *IL*

2 Klassen von Zwischensprachen:

- Code für Kellermaschine mit Halde, z.B. Pascal-P, ..., JVM
  - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
  - Datentypen und Operationen auf Daten entsprechen weitgehend der *QM*, zusätzlich Umfang und Ausrichtung im Speicher berücksichtigen
- Code für RISC-Maschine mit unbeschränkter Registerzahl und (stückweise) linearem Speicher
  - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
  - Datentypen entsprechen Zielmaschine einschl. Umfang und Ausrichtung im Speicher
  - Operationen der Zwischensprache entsprechen abstrakten Zielmaschinenbefehlen (Laufzeitsystem berücksichtigen!)
  - aber noch keine konkreten Befehle, keine Adressierungsmodi
  - Vorteil: fast alle Prozessoren auf dieser Ebene gleich

Kellermaschinencode gut für (Software-)Interpretation, schlecht für explizite Codeerzeugung, RISC-Maschine: umgekehrt

9. Wiederholung: Zwischensprache *IL* II

Im folgenden nur Code für RISC-Maschine mit unbeschränkter Registerzahl betrachtet

drei Darstellungsformen:

- keine explizite Darstellung: *IL* erscheint nur implizit bei direkter Codeerzeugung aus AST: höchstens lokale Optimierung, z.B. Einpaßübersetzung
- Tripel-/Quadrupelform: Befehle haben schematisch die Form
  - $t_1 := t_2$ t t_3 oder
  - $m\!: t_1 := t_2$ t t_3
- SSA-Form (Einmalzuweisungen, static single assignment): wie Tripelform, aber an jedes t_i kann nur einmal zugewiesen werden (gut für Optimierungsaufgaben)

9. Wiederholung: Zwischensprache *IL* III

Gesamtprogramm eingeteilt in Prozeduren, Prozeduren unterteilt in Grundblöcke, oder erweiterte Grundblöcke

- Grundblock: Befehlsfolge maximaler Länge mit: wenn ein Befehl ausgeführt wird, dann alle genau einmal, also
  - Grundblock beginnt mit einer Sprungmarke,
  - enthält keine weiteren Sprungmarken
  - endet mit (bedingten) Sprüngen
  - enthält keine weiteren Sprünge
  - entspricht einem Block im Flußdiagramm (dort nicht maximal)
  - Unterprogrammaufrufe zählen nicht als Sprünge!
- erweiterter Grundblock: wie Grundblock, aber kann mehrere bedingte Sprünge enthalten: ein Eingang, mehrere Ausgänge
  - Vorteil: nach Sprüngen ist die Registerbelegung bekannt

Beispiel (Wiederholung)

```
c=0;            s1: ST   >c<   0
if x > 0 {      s2: LD   <x>
                s3: GT   0
                s4: JMP  FALSE u1

a=2;            t1: ST   >a<   2
b=a*x+1;        t2: LD   <a>
                t3: LD   <x>
                t4: MUL  t2    t3
                t5: ADD  t4    1
                t6: ST   >b<   t5
a=2*x;          t7: LD   <x>
                t8: MUL   2    t7
                t9: ST   >a<   t8
```

```
c=a+1+b;        t10: LD   <a>
                t11: ADD  t10   1
                t12: LD   <b>
                t13: ADD  t11   t12
                t14: ST   >c<   t13
}               t15: JMP  u1

x=c;            u1: LD   <c>
                u2: ST   >x<   u1
```

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 BEG
5. Die letzten 10%

Die Abschnitte Grundlegendes und Optimierungen stellen verschiedene Verfahren dar, die nicht auf <u>einen</u> konsistenten Übersetzer abzielen.

9.1 Codeerzeugung

- Grundverfahren:

```
for   alle Grundblöcke
do    führe Grundblock sequentiell aus;
      gib nichtausführbare Operationen aus
end
```

Benutze dazu Maschinensimulation:

- abstrakte Interpretation des Programms
- Maschinenzustände bilden Vor- und Nachbedingungen für generierte Befehle
- Werte und allozierbare Resourcen werden durch Deskriptoren simuliert
- benutze Kostenfunktion zur Bestimmung der Befehle

Achtung: Dieses Konzept ist alt und erzeugt nicht den besten Code.
Code wird oft direkt aus dem AST erzeugt. Es gibt keine Trennung von Auswahl, Anordnung und Registerallokation.

9.1 Kostenfunktionen

verschiedene Möglichkeiten:

- Anzahl bzw. Speicherumfang der Befehle
- Anzahl ausgeführter Befehle
- Zeitaufwand der Befehlsausführung
- Umfang benötigter Betriebsmittel (Register)
- Berücksichtigung Cache?

in der Praxis:

- Speicheraufwand/Energieverbrauch minimieren bei eingebetteten Systemen
- im allgemeinen Fall Laufzeit minimieren,
  aber: exakte Laufzeit oft nicht bestimmbar (Mangel an Dokumentation) oder sehr schwierig zu berechnen

Beachte: mit Kostenfunktion wird nur lokal optimiert, lokale Optimalität garantiert keine globale Optimalität!

9.1 Maschinensimulation

Jeder Zustand repräsentiert durch Register-, Wert- und Speicherdeskriptoren

- Welche (symbolischen) Werte sind derzeit identifiziert?
- Befinden sie sich im Speicher? wo?
- Befinden sie sich in Registern? mit Kopie im Speicher?

Symbolische sequentielle Ausführung des Zwischencodes eines Grundblocks:

- Operation im augenblicklichen Zustand statisch ausführbar, z.B. weil alle Operanden als Konstante bekannt: Operation ausführen, Ergebnis merken, keine Zustandsänderung
- Operation statisch nicht ausführbar: zur Operation äquivalente Befehlssequenz ausgeben, Zustand entsprechend ändern, gemerkte Ergebnisse berücksichtigen

Problem: Annotationen über gewünschte Registerbelegung, Ausdruckstransformation auf einfachste Form?

9.1 Register- und Speicherdeskriptoren (Pascal-Notation)

```
type   register-state = (free, copy, unique, locked);

  register-descriptor =
record
  state: register-state;
  content: ↑value-descriptor;
  memory-copy: ↑main-storage-access;
end;

  main-storage-access =
record
  base, index: ↑value-descriptor;
  displacement: internal-int;
end;
```

9.1 Wertdeskriptor

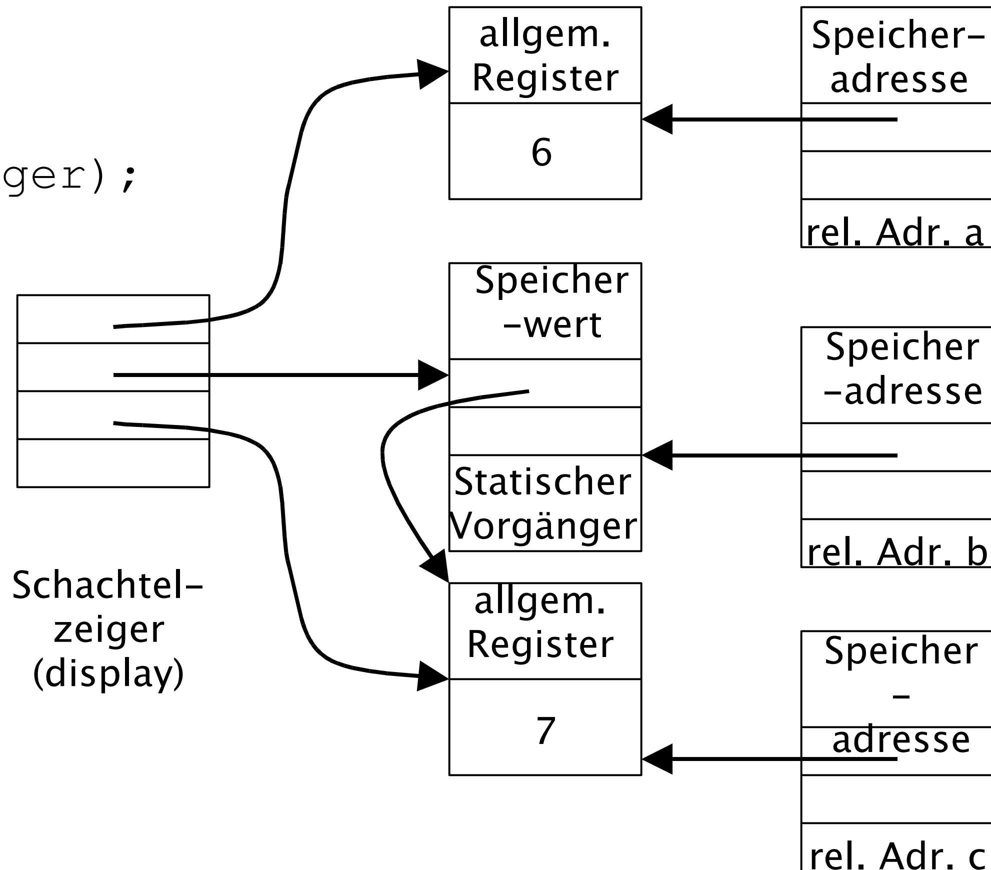
```
type value-descriptor =
record
  tmode: target-type;    (* Pointer to target
                            definition table *)
case class: value-class of
  literal-value:
    (lval: internal-int);
  label-reference, procedure-reference:
    (code: assembler-symbol;
     environment: ↑value-descriptor);
  general-register, register-pair, fp-register:
    (reg: ↑register-descriptor);
  memory-address, memory-value:
    (location: main-storage-access)
end;
```

9.1 Maschinensimulationszustand

```
var a: integer;
procedure p;
  var b: integer;
  procedure q(c: integer);
     a:=b+c;
  begin
    b:=1; q(2);
  end;
begin
  p
end
```

9.1 Maschinensimulationszustand

```
var a: integer;
procedure p;
  var b: integer;
  procedure q(c: integer);
     a:=b+c;
  begin
    b:=1; q(2);
  end;
begin
  p
end
```



9.1 Zielattributierung

Maschinensimulation und Befehlsauswahl abhängig von Eigenschaften (Attributen) des Zwischencodes und der Zielbefehle:

- Klassifikation der Register:
  - allgemeine, Gleitpunkt-, Adreßregister
  - reservierte Register für Rückkehradressen usw.
  - Doppelregister nur für gerade/ungerade Paare, z.B. (R2,R3)
  - Welche Operanden dürfen/müssen in welche Register?
- Umsetzung boolescher Ausdrücke mit Sprüngen in Kurzauswertung
- algebraische Vereinfachungen
- Nutzung der Adressierungspfade statt expliziter Berechnung

9.1 Gerade/ungerade Register

```
type register_class = (beliebig,gerade,ungerade,paar);
rule ausdruck ::= ausdruck operator ausdruck .
attribution
  ausdruck[2].wunsch :=
    case operator.operator of
      plus,minus:
        if ausdruck[1].wunsch=paar then gerade
        else ausdruck[1].wunsch;
      mal: ungerade;
      div: gerade
    end;
  ausdruck[3].wunsch :=
    case operator.operator of
      plus, minus:
        if ausdruck[1].wunsch=paar then gerade
        else ausdruck[1].wunsch;
      mal: ungerade;
      else beliebig
    end;
```

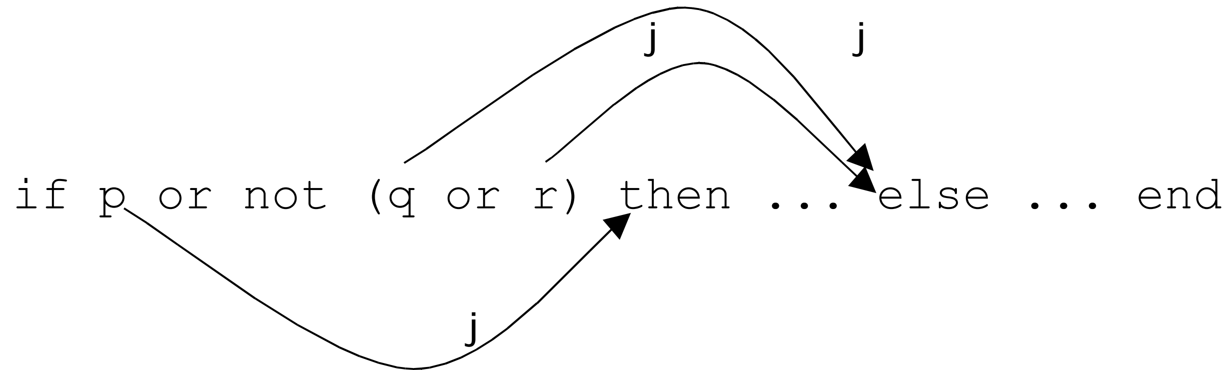
merke: Attribut nicht bindend, aber dann zusätzliche Kosten

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 BEG
5. Die letzten 10%

Die Abschnitte Grundlegendes und Optimierungen stellen verschiedene Verfahren dar, die nicht auf einen konsistenten Übersetzer abzielen.

9.2 Kurzauswertung



9.2 Kurzauswertung II

```
type marken = record ja,nein:symb_adresse; nachf:Boolean end;

rule bed_anw ::= 'if' ausdruck 'then' anw 'else' anw 'end' .
attribution
  ausdruck.loc := neue_adresse;
  bed_anw.then_loc := neue_adresse;
  bed_anw.else_loc := neue_adresse;
  ausdruck.ziel :=
     neue_marken(bed_anw.then_loc, bed_anw.else_loc, true);
```

- neue_adresse: generiere neues Sprungziel für Zielcode
- neue_marken: neuer Verbund des Typs `marken`
- loc: Startadresse von Ausdruck/Anweisung
- ziel: Verbund der Sprungziele
- nachf: gibt an, welches Sprungziel unmittelbar folgt (Sprungbefehl nicht nötig, erweiterter Grundblock)

9.2 Kurzauswertung III

```
rule ausdruck := ausdruck operator ausdruck .
attribution
  ausdruck[2].loc := ausdruck[1].loc;
  ausdruck[3].loc := neue adresse;
  ausdruck[2].ziel :=
    if operator.operator = 'or'
    then neue_marken(ausdruck[1].ziel.ja,ausdruck[3].loc,false)
    else neue_marken(ausdruck[3].loc,ausdruck[1].ziel.nein,true)
    end;
  ausdruck[3].ziel := ausdruck[1].ziel;

rule ausdruck := 'not' ausdruck .
attribution
  ausdruck[2].loc := ausdruck[1].loc;
  ausdruck[2].ziel :=
    neue_marken(ausdruck[1].ziel.nein,ausdruck[1].ziel.ja,
      not ausdruck[1].ziel.nachf)
```

9.2 Algebraische Vereinfachungen

Ziel: Ausnutzung von Identitäten wie

- $x + y \quad = y + x$
- $x - y \quad = x + (-y) \quad = -(y - x)$
- $-(-x) \quad = x$ Vorsicht bei Zweierkomplement!
- $x * y \quad = y * x \quad = (-x) * (-y)$
- $-(x * y) = (-x) * y \quad = x * (-y)$

Beispiel: wie transformiert man

- $(-x) * (y - z)$ (5 Befehle) in
- $(z - y) * x$ (3 Befehle)

9.2 Algebraische Vereinfachungen II

Methode (gleiches Verfahren wie Operatoridentifizierung!):

- Berechne im Ausdrucksbaum von unten nach oben die Kosten (p,n) für (Ergebnis, negatives Ergebnis) unter Berücksichtigung Kommutativität.
- Dann entscheide von oben nach unten, welches Ergebnis benötigt wird, und ob Operanden vertauscht werden sollen.

9.2 Algebraische Identitäten

Baum Knoten	Resultat Form	Operanden Form	k	negative Operanden	Negieren	Eigentl. Operation	Methode
a+b	p	pp	1	false	false	+	a + b
		pn	1	false	false	-	a – (- b)
		np	1	true	false	-	b – (-a)
		nn	2	false	true	+	-(-a + (-b))
	n	pp	2	false	true	+	- (a + b)
		pn	1	true	false	-	-b - a
		np	1	false	false	-	-a – b
		nn	1	false	false	+	-a + (-b)
a – b	p	pp	1	false	false	-	a – b
		pn	1	false	false	+	a + (-b)
		np	2	false	true	+	- (-a + b)
		nn	1	true	false	-	-b – (-a)
	n	pp	1	true	false	-	b – a
		pn	2	false	true	+	- (-a + (-b))
		np	1	false	false	+	-a + b
		nn	1	false	false	-	-a – (-b)
a * b	p	pp	1	false	false	*	a * b
		pn	2	false	true	*	-(a * (-b))
		np	2	false	true	*	-(-a * b)
		nn	1	false	false	*	-a * (-b)
	n	pp	2	false	true	*	-(a * b)
		pn	1	false	false	*	a * (-b)
		np	1	false	false	*	-a * b
		nn	2	false	true	*	- (-a * (-b)

9.2 Registerverbrauch minimieren

Grundschema der Codeerzeugung für Ausdrücke:

- bringe Ausdruck in Postfixform *abc**+
- Lade Operanden in gegebener Reihenfolge in Register, wende Operation auf die zuletzt geladenen Operanden bzw. Zwischenergebnisse an
- Vereinfachung: kombiniere Operation mit Laden:

Statt

```
LD   a,R1
LD   b,R2
LD   c,R3
MUL  R3,R2
ADD  R2,R1
```

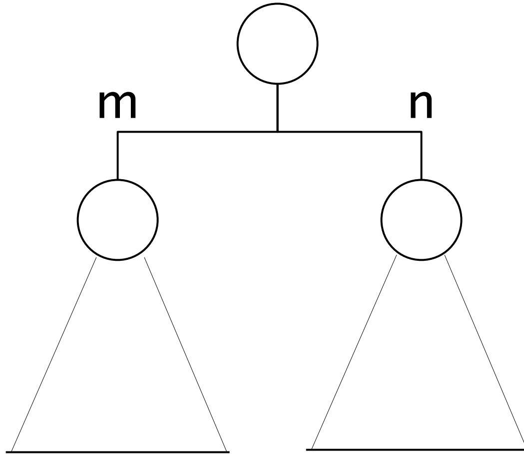
besser

```
LD   a,R1
LD   b,R2
MUL  c,R2
ADD  R2,R1
```

- Problem: Register für 1. Op. während Berechnung 2. Op belegt.
- Wann soll man den 2. Op. vor dem ersten berechnen?

```
LD   b,R1
MUL  c,R1
ADD  a,R1
```

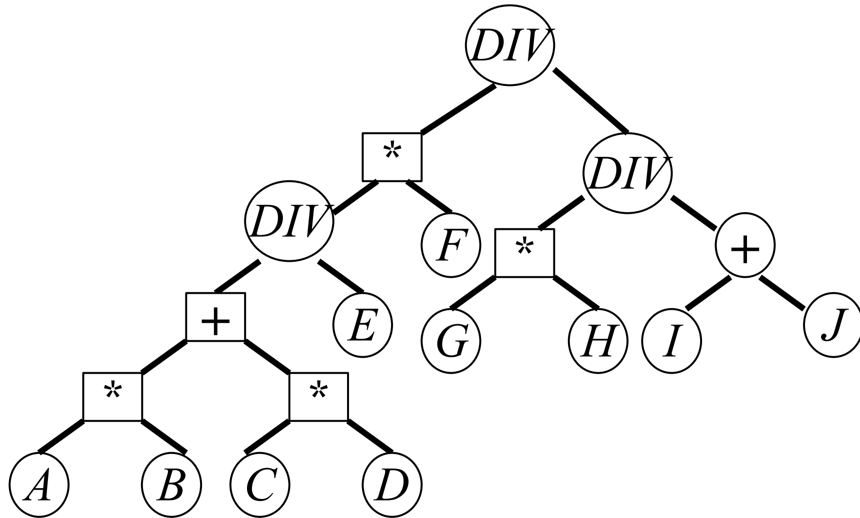
9.2 Registerverbrauch bei Ausdrücken: Reihenfolgebestimmung



r Anzahl verfügbarer Register

Verbrauch	Teilbäume	Baum	Reihenfolge
$n=m$,	$n+1 \leq r$	$n+1$ Register:	Reihenfolge gleichgültig
$n>m$,	$n \leq r$	n Register:	rechts, dann links auswerten, Zwischenergebnis in Register
$m>n$,	$m \leq r$	m Register:	links, dann rechts auswerten
$n=m=r$,	$max(n,m) > r$	r Register und Auslagern (spill code):	Reihenfolge gleichgültig

9.2 optimale Reihenfolge kann springen



Runde Ecken: einfach langes Resultat
Rechteckige Ecken: doppelt langes Resultat

MOV A, R0	*(R0, R1) := A*B*
MUL B, R0	
MOV C, R2	*(R2, R3):= C*D*
MUL D, R2	
ADD R3, R1	*(R0, R1):= (R0, R1)+(R2,R3)*
ADC R2	addiere Übertrag
ADD R2, R0	
DIV E, R0	*R0:= (R0, R1) DIV E*
MOV G, R2	*(R2, R3):= G*H*
MUL H, R2	
MOV I, R1	*R1:= I+J*
ADD J, R1	
DIV R1, R2	*R2:= (R2, R3) DIV R1*
MUL F, R0	*(R0, R1):= R0*F*
DIV R2, R0	*R0:= (R0, R1) DIV R2*

geschlossene Reihenfolge:
1 Register mehr

9.2 Wann ist Postfixform optimal? Starkes Normalformtheorem

Gegeben eine Maschine mit n identischen Registern r_i und Befehlen der Form:

- Register_i := Speicherplatz,
- Speicherplatz := Register_i,
- $\text{Register}_i := op(v_j, \ldots, v_k)$, v_h Register oder Speicherplatz.

Programm $P_1\, S_1\, P_2 \ldots P_{s\text{-}1}\, S_{s\text{-}1}\, P_s$ in Normalform: S_i – Speicheroperation, alle Register danach frei und P_i – Befehlsfolge ohne Speicheroperationen.

Programm in starker Normalform, wenn alle P_i stark zusammenhängend:

$\forall\, P_i = B_1 \ldots B_r$: B_j berechnet Operand für $B_l \Rightarrow \forall B_k$: $j \leq k < l$: B_k trägt zu Operanden für B_l bei.

Satz [Aho1976]: Wenn die Größe aller Operanden und Zwischenergebnisse eines Ausdrucks der Registergröße entspricht, gibt es ein optimales Programm in starker Normalform, das diesen Ausdruck berechnet.

Für logische, Gleitpunkt- und Ganzzahloperationen erfüllt außer Ganzzahl-Multiplikation und Division.

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 BEG
5. Die letzten 10%

9.3 Befehlsauswahl

Verfahren:

- Makrosubstitution
- Entscheidungstabelle
- Programmierte Verfahren (Mixtur der anderen)
- Termersetzungsverfahren
- Graphersetzungsverfahren (Zukunft)

Voraussetzung: Spezifikation der schematischen Umsetzung von Zwischencodeoperationen in Befehlssequenzen liegt vor

- trivial für einfache arithmetische Operationen usw.
- schwierig für Operationen auf Teilwörtern u.ä.

9.3 Makrosubstitution

Fasse jede Operation als Prozeduraufruf auf, setze den Prozedurrumpf mit gleichzeitiger Substitution der Argumente offen in den Zielcode ein

- etwaige bedingte Anweisungen im Rumpf während der Substitution auswerten
- Gebe nicht auswertbare Anweisungen als Zielcode aus
- Schleifen im Rumpf bleiben erhalten

Berücksichtige dabei die Maschinensimulation

Bewertung:

- das einfachste und älteste Verfahren
- viele Fallunterscheidungen im Rumpf
- aufwendig zu programmieren
- Korrektheit des Ergebnisses erfordert aufwendigen Test

9.3 Entscheidungstabelle für jede Operation der Zwischensprache

Beispiel „plus integer integer“
(Code für IBM 370, Berücksichtigung Vorzeichen)

Ergebnis Vorz.	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
l Vorzeichen	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
r Vorzeichen	+	+	+	+	-	-	-	-	+	+	+	+	-	-	-	-
l in Register	j	j	n	n	j	j	n	n	j	j	n	n	j	j	n	n
r in Register	j	n	j	n	j	n	j	n	j	n	j	n	j	n	j	n
swap (l, r)			x				x		x		x	x			x	
lreg (l, desire)				x				x				x				x
gen (A l r)		x	x	x										x	x	x
gen (AR l r)	x												x			
gen (S l r)						x	x	x		x	x	x				
gen (SR l r)					x				x							
gen (LCR l l)							x			x			x	x	x	x
free (r)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
result (l store)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Ergebnis Vorz.	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
l Vorzeichen	+	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-
r Vorzeichen	+	+	+	+	-	-	-	-	+	+	+	+	-	-	-	-
l in Register	j	j	n	n	j	j	n	n	j	j	n	n	j	j	n	n
r in Register	j	n	j	n	j	n	j	n	j	n	j	n	j	n	j	n
swap (l, r)			x		x		x	x			x			x		
lreg (l, desire)				x				x				x				x
gen (A l r)		x	x	x										x	x	x
gen (AR l r)	x												x			
gen (S l r)						x	x	x		x	x	x				
gen (SR l r)					x				x							
gen (LCR l l)	x	x	x	x		x					x					
free (r)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
result (l store)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

vollständige Entscheidungstabelle

Auswertung: Bedingungen als Index

9.3 Bewertung Entscheidungstabellen

- gleiche Leistung wie Makrosubstitution
- Fallunterscheidungen systematisiert (weniger fehleranfällig)
- aufwendig zu spezifizieren
- automatische Verfahren zur optimalen Programmierung vollständiger Entscheidungstabellen verfügbar

9.3 Fazit Makroexpansion

- Sinnvoll, wenn Zielcode Hochsprache (*z.B. C*)
- Für Maschinensprachen: Aufwand hoch
- Korrektheit und Vollständigkeit schwierig zu erreichen
- wartungsfreundlich ??

9.3 Tripel/Quadrupel – Darstellung

(*ID:*) Operation,

(*ID:*) Operation Operand,

(*ID:*) Operation Operand1 Operand2.

Operationen sind

- Maschinenoperationen
- Adreßrechnungen (aus Reihungszugriffen, qualifizierten Zugriffen, Parameterzugriffen usw. übersetzt)

bisher: Codeerzeugung setzt diese Darstellung implizit voraus, kein expliziter Gebrauch

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 BEG
5. Die letzten 10%

9.4 Befehlsauswahl als Termersetzung

Voraussetzung: Fasse einen Grundblock als Folge von (Ausdrucks-) Bäumen (Termen) auf. Ecken sind die Tupel
ST >a< t´ , *LD <a>*, t *t´ t´´* , *Prozeduraufruf*(...);
auch die Bedingung der abschließenden bedingten Sprünge ist ein Baum

Beobachtung (Weingart, 1973): Jeder aus einem Ausdrucksbaum b erzeugte Befehl i deckt einen Teil dieses Baumes ab. Der Gesamtcode überdeckt den Gesamtbaum überlappungsfrei.

Idee: Jeder Ausdrucksbaum ist ein Term einer Termalgebra T. Wenn man auch die Maschinenbefehle als Terme einer Termalgebra T' beschreiben kann, dann kann man folgendermaßen Code erzeugen:
Ersetze den Ausdrucksbaum, einen Term $b \in T$ der Zwischensprache, durch einen Term $b' \in T'$ der Zielalgebra T'.

9.4 Einfache Termersetzung: kontextfreie Grammatiken

Einfache Fassung einer Termalgebra:
mit kontextfreien Grammatiken (Graham/Glanville 1978):

- schreibe alle Bäume in Präfixform (als Text, der zugleich die Baumstruktur wiedergibt) mit Hilfe der Grammatik G

  $Ausdruck ::= Operator\ Ausdruck\ Ausdruck \mid Operator\ Ausdruck \mid Konstante$

  $Operator ::= + \mid - \mid * \mid divmod \mid \ldots$

- definiere für jeden Maschinenbefehl Produktionen (Regeln), die den vom Befehl abgedeckten Baum beschreiben: Maschinengrammatik G'
  - linke Seite der Produktion: das Betriebsmittel, das das Ergebnis des Befehls enthält (Speicher, meist Register)
  - solche Betriebsmittel auch als Element der rechten Seite zulassen
  - Voraussetzung: jeder Befehl hat genau ein Ergebnis!
- zerteile den vorgegebenen Baum (Text in Präfixform) mit dieser Maschinengrammatik. Die dabei benutzten Produktionen ergeben zusammen die Befehle für den Baum.

9.4 LR–Zerteiler zur Codegenerierung

Cattell (1978):

Rekursiver Abstieg zur Zerteilung: nicht sehr erfolgversprechend

Graham und Glanville:

- LR–Zerteilung,
- Codegenerierung als Strukturanbindung,
- hochgradig indeterministisch,
- Kostenfunktion zur Auflösung der Mehrdeutigkeiten.

Karlsruher Implementierung 1980 (Jansohn/Landwehr): CGSS

- besser als die Berkeley–Implementierung
- bis 1990 in vielen Übersetzern eingesetzt
- Umfang der Maschinenbeschreibungen:
  ca. 1500 Zeilen (einfach) – 6000 Zeilen (mit allen Tricks)
- Hauptprobleme:
  - Nachweis der vollständigen Überdeckung $L(G) \subseteq L(G')$
  - effiziente Handhabung der Adressierungsmodi

9.4.1 Exkurs: Baumsprachen, Baumautomaten

Gegeben sei ein Alphabet S von Terminalen f mit Stelligkeit $s(f) = k$, $k \geq 0$

Die Menge B(S) der Bäume über S ist induktiv definiert durch

- $a \in B$(S), wenn $a \in$ S und $s(a) = 0$, d.h. $a \in$ S_0
- wenn $b_1,\ldots,b_k \in B$(S) und $f \in$ S, $s(f) = k$ dann $f(b_1,\ldots,b_k) \in B$(S)

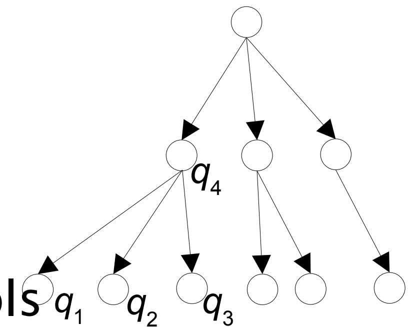
G=(N, S,P,Z) heißt eine (reguläre) Baumgrammatik mit der (regulären) Baumsprache $L(G) \subseteq B$(S$\cup N$) , wenn

- N ist eine endliche Menge von Nichtterminalen
- $Z \in N$ ist das Zielsymbol
- P ist eine Menge von Produktionen $X \rightarrow w$, $w \in B$(S$\cup N$), $X \in N$
- Der Typ $t(p) = (X_1,\ldots,X_k)$ einer Produktion p: $X \rightarrow w$ ist die Folge der Nichtterminale, die in w vorkommen.
- Ersetzt man alle diese X_k in w durch Variable x_k, so erhält man das Ersetzungsmuster $m(p)$. $m(p)$ heißt linear, wenn keine Variable mehrmals vorkommt.

9.4.1 Baumautomaten

Ein Baumautomat ist ein endlicher Automat, der Ableitungsbäume konstruiert bzw. analysiert:

- ein quellbezogener bottom-up (BU) Automat erreicht Zustände $q_1,\ldots,q_k$ für die k Unterbäume eines Terms $f(b_1,\ldots,b_k)$ und geht bei Erreichen von f in einen Zustand q über:
  $q_1 \ldots q_k f \rightarrow q$
- ein zielbezogener top-down Automat hat die umgekehrten Regeln $qf \rightarrow q_1 \ldots q_k$
- Baumautomaten analysieren/konstruieren den Baum während einer Tiefensuche:
  - zielbezogen: beim ersten
  - quellbezogen beim letzten Antreffen eines Symbols



9.4.1 Sätze über Baumsprachen und –automaten

Satz: Der Durchschnitt, die Vereinigung und das Komplement von regulären Baumsprachen sind ebenfalls reguläre Baumsprachen.

Satz: Gleichheit und Enthaltensein von Baumsprachen sind entscheidbar.

Satz: Zu jedem nicht-deterministischen BU-Baumautomaten existiert ein deterministischer BU-Baumautomat, der die gleiche Baumsprache akzeptiert. Für zielbezogene Baumautomaten gilt dies nicht.

Beweise: ganz ähnlich wie für reguläre Sprachen und endliche Automaten. Deterministisch-Machen funktioniert mit der Teilmengenkonstruktion.

9.4.1 Baumautomaten und Befehlsauswahl

Einsicht: sowohl die Termalgebra, mit der die Zwischensprachenbäume erzeugt sind, als auch die Termalgebra für die Maschinenbeschreibung sind Baumgrammatiken. Daher ist das Überdeckungsproblem $L(G) \subseteq L(G')$ lösbar.

Befehlsauswahl transformiert zwischen diesen Termalgebren. Dabei werden Ersetzungsmuster gemäß der Maschinenbeschreibung gesucht und durch entsprechende Terme ersetzt.

Problem: Termersetzungssystem ist mehrdeutig.

Ein Ausweg: Entscheidung mit Hilfe von Kostenmaßen

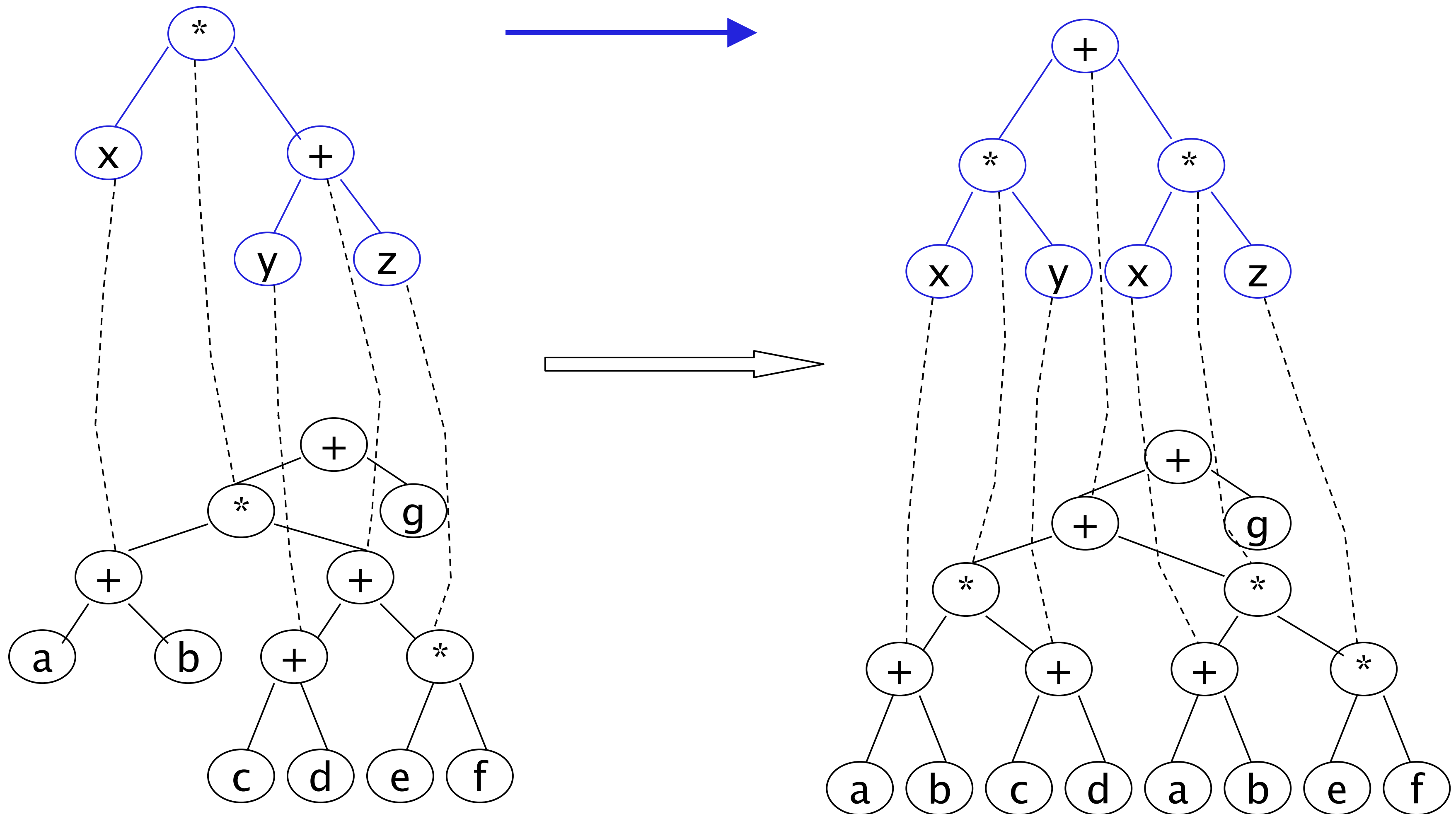
Problem: Termersetzung (mit Variablen) für einen kompletten Baum nicht effizient berechenbar: Ersetzung des Termersetzungssystems (TES) durch ein Grundtermersetzungssystem (GTES, enthält keine Variable), für das es effiziente Verfahren gibt.

9.4.1 Termersetzungssystem *TES*

- T sei Σ–Termalgebra mit Variablen V und Axiomen Q
- TES: Menge von Termersetzungsregeln $l \rightarrow r$, $l,r \in T$ für Termalgebra T
  - l,r können Variable enthalten
  - alle Variablen in l müssen auch in r vorkommen
- $l \rightarrow r$ beschreibt Ersetzung eines Unterterms $t´$ von Term t durch $s´$, falls Substitution s existiert mit $t´ = l$s und $s´ = r$s.
- $t \Rightarrow s$, wenn s durch Regelanwendung aus t entstanden

Beachte: in einem Term t kann eine Regel an mehreren Stellen anwendbar sein, es könnten auch verschiedene Regeln anwendbar sein; $t \Rightarrow s$ sagt nicht, welche Regel an welcher Stelle benutzt wurde

9.4.1 Beispiel: Distributivgesetz mittels Termersetzung



9.4.1 Ableitung mit festem Ziel Z

Gegeben:

- TES, Zielsymbol Z und Regeln $l \to r$
- Term $t \in T$

Gesucht:

- Ableitung $t \Rightarrow^* Z$

Sei $L(TES, Z) = \{t | t \Rightarrow^* Z\}$

9.4.1 Grundtermersetzungssystem *GTES*

Grundtermersetzungssystem: Termersetzungssystem, in dessen Regeln $l \rightarrow r$ keine Variablen vorkommen

- *GTES*: Ersetze in Termersetzungsregeln $l \rightarrow r$ von *TES* Variable durch Grundterme (Terme ohne Variablen)
- *GTES* ist Instanz von *TES*, wenn alle Ersetzungsregeln so entstanden sind
- Dann gilt $L(GTES, Z) \subseteq L(TES, Z)$
- Ableitung $t \Rightarrow^* Z$ effizient berechenbar für Grundtermersetzungssysteme
- Gesucht Instanz *GTES* von *TES* mit $L(GTES, Z) = L(TES, Z)$

9.4.1 Termersetzung → Grundtermersetzung

Konstruktion eines *GTES* aus *TES*:

- Prinzip: ersetze Regel $l \rightarrow r$ durch (potentiell unendlich viele) Regeln $ls \rightarrow rs$ für alle benötigten (!) Substitutionen *s*
  - Variable stellen Operanden (Unterbäume) dar, daher praktisch bei Befehlsauswahl nur endlich viele *s*, die die Register, Konstanten, Speicherplätze, ... für Operanden substitutieren
- Test auf Vollständigkeit $L(GTES) = L(TES)$ effizient möglich
- Konstruktion eines *GTES* mit $L(GTES) = L(TES)$ unentscheidbar, aber berechenbar:
  - Es gibt Algorithmen, die ein vollständiges *GTES* aus *TES* erzeugen, falls es existiert (sonst unendliche Laufzeit).

9.4.1 Ableitung $t \to^* Z$ für Grundtermersetzungssysteme

Satz: $L(GTES)$ ist reguläre Baumsprache – daher durch einen endlichen Baumautomaten akzeptierbar.

- Berechnen einer Ableitung (Überdeckung) durch einen endlichen Baumautomaten.
- Baumgrammatik $G = (T, N, Z, P)$ und Regeln P der Form $S \to K(L, R)$ wobei $S \in N, K \in T, L,R \in T \cup N$
- Wie bei regulären Sprachen und endlichen Automaten gilt:
  - Gleichheits-/Inklusions- und Akzeptionsproblem sind entscheidbar.
  - Konstruktion eines deterministischen und minimalen Baumautomaten möglich

9.4.1 Behandlung der Kosten

Term-Menge $L(A)$ – die Menge der Terme, die ein Baumautomat A akzeptiert.

- Bewertete Term-Menge – Jeder Term t liegt mit Kosten c in $L(A)$
- Kosten c ergeben sich aus der Summe der Einzelkosten bei der Ableitung (für nicht akzeptierte Terme ist $c = \infty$).

9.4.1 Akzeption mit Kosten

Berechne eine minimale Überdeckung durch einen endlichen Baumautomaten.

- Konstruktion eines deterministischen Baumautomaten nicht immer möglich, wenn Kosten berücksichtigt werden müssen,
- Konstruktion eines minimalen Baumautomaten mit Kosten effizient möglich

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 BEG
5. Die letzten 10%

9.4.2 Bottom-up Pattern Matching – *BUPM*

Hoffmann und O´Donnell (´82)

- Grundtermersetzungssystem,
- Zwischen- und Zielmaschinenprogramm als Bäume repräsentiert,
- Von unten werden Muster im Zwischensprachebaum gefunden,
- Musterabdeckung (mehrdeutig) hat Entsprechungen in Zielmaschinen-(unter-)bäumen,
- Von oben wird kostengünstigste Abdeckung selektiert.

Implementierung in Karlsruhe durch *BEG*-1 1988,

Entwicklung eines Codegenerators um eine Größenordnung schneller und zuverlässiger als handgeschrieben bei gleicher Qualität

9.4.2 Bottom-up Rewrite System – *BURS*

Graham und Pelegrini-Llopart (´88)

- Termersetzungssystem statt Grundtermersetzungssystem,
- Kleinere Spezifikation möglich,
- Findet theoretisch alle Abdeckungen
  - unendlich viele – exponentiell viele sinnvolle
  - Grenzen für Implementierung
- Anschließende Suche nach globalem Optimum (*NP* hart)
- Angenähert durch *A** Suche
  - In Karlsruhe implementiert in CGGG (Boesler ´98)

9.4.2 Back-End-Generator – *BEG-2*

Emmelmann (´94)

- Spezifikation von Termersetzungssystem,
- aus Termersetzungssystem wird Grundtermersetzungssystem erzeugt, wenn vorhanden,
- Implementierung wie für Grundtermersetzung,

9.4.2 Back-End-Generator – *BEG-2*

Aufteilung in Maschinenbeschreibung und Transformationsregeln.

- Maschinenbeschreibung
  - nur zielprozessorabhängig
  - keine Variablen in den Termen
  - Schablonen konkreter Maschinenbefehle und Resourcen angegeben

```
R := add R, c
R := add R, c(R)
```

- Transformationsregeln
  - zielprozessor- und zwischensprachenabhängig
  - Variable erlaubt, werden später durch Ressourcen ersetzt
  - bildet Zwischensprache auf Zielsprache ab, auch „optimierende" Transformationen auf der Zwischensprache

```
plus(A,B) --> add A, B
```

- Kostenfunktion
  - nur bei Maschinenbeschreibung zulässig
  - beschreibt Kosten der Maschinenbefehle bzgl. des Optimierungsziels

9.4.2 Back-End-Generator – *BEG-2*

Eingabe:

- Maschinenbeschreibung durch Baumgrammatik G mit Zielsymbol Z
- Transformationsregeln von Zwischen- in Maschinensprache durch Termersetzungssystem TES
- Kostenfunktion $c: Maschinenterm \rightarrow Integer$

Ausgabe:

- Grundtermersetzungssystem $GTES$ mit

$$L(GTES,Z) \subseteq L(TES \cup G^{-1},Z)$$

und minimalen Kosten gegeben durch Baumgrammatik, falls existent

9.4.2 Vergleich von Termersetzungs-Verfahren

- Graham-Glanville
  - Durch Hinzufügen von Regeln kann Code besser oder schlechter werden
  - Nicht immer optimale Überdeckung
    (durch Zerteilen von Links nach Rechts)
- BUPM
  - Durch Hinzufügen von Regeln kann Code nur besser werden
  - Findet optimale Überdeckung, aber nur GTES
- BURS
  - Nachweis für Vollständigkeit der Regelmenge nicht entscheidbar
  - Berechnet alle Überdeckungen, benötigt dazu eine Suchstrategie
- BEG-2
  - Durch Hinzufügen von Regeln kann Code nur besser werden
  - Spezifikation eines TES (mächtigere und kürzere Spezifikationen); automatischer Übergang auf GTES; Vollständigkeit entscheidbar

9.4.2 Vergleich: Makrosubstitution – Termersetzung

- **Makrosubstitution**
  - Generator leicht umzusetzen
  - Ablaufstrategie muss ausprogrammiert werden
  - Keine Kostensteuerung
  - Nur einstufige Ersetzungen
  - Nur geeignet nur wenn Zwischen- und Zielsprache sehr ähnlich sind
- **Termersetzung**
  - Generator enthält je nach Verfahren sehr komplizierte Algorithmen
  - Automatische Suchstrategie,

    durch Modularität des an den Regeln haftenden Codes
  - Es gibt Möglichkeit zur Kostensteuerung
  - Mehrstufige Ersetzungsschritte möglich
  - Spezifikation auch bei größeren Regelmengen traktabel

9.4.2 Praxis: Wer setzt welche Verfahren ein?

- Jikes-RVM: Bottom-Up Rewrite System (BURS)
  Abgeleitet von Iburg. Siehe: "Engineering a Simple, Efficient Code-Generator Generator" by Fraser, Hanson, and Proebsting, TOPLAS 1(3), Sept. 1992.
- Borland Pascal, Delphi: BEG
  http://www.hei.biz/Products.html
- Watcom (OpenWatcom compiler): sieht handgeschrieben aus
- Sun Hotspot JVM
  Für die Optimierung der Hotspots: BURS-ähnliches Verfahren
- GCC: ???
- Microsoft C: ???
- Intel C: ???

9.4.2 BEG – Generierte Code-Generatoren für ZS Mobil (1988)

Prozessor	Zeilen	Größe in *KB*	Anzahl der Regeln	Betriebssystem
Sparc	2000	50	140	Sun *OS*, Solaris
i386	4000	100	280	Linux, Free*BSD*
MIPS (*R*3000 – *R*10000)	1500	46	130	Ultrix, Irix
Power*PC*	3600	74	200	Parix
DEC-Alpha	1500	40	150	*OSF* / 1

9.4.2 Handgeschriebene Code-Generatoren

Prozessor	Zeilen	Größe in KB
68020	13000	450
VAX	15000	650

Kapitel 9: Codeerzeugung

0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 Beispiel: BEG
5. Die letzten 10%

9.4.3 BEG Beispiel – Spezifikation

Maschinenbeschreibung
Baumgrammatik

(1)	R ::=add (R, Ea)	4
(2)	R ::=mov (Ea)	2
(3)	R ::=bb	
(4)	Ea::=R	
(5)	Ea::=c	
(6)	Ea::=di (R, c)	

Abbildungsbeschreibung
Termersetzungssystem

(a1)	plus (A,B)	$\rightarrow$	add (A,B)
(a2)	A	$\rightarrow$	mov (A)
(a3)	cont(plus(A,B))	$\rightarrow$	di (A,B)
(a4)	plus(A,B)	$\rightarrow$	plus (B,A)

Initiales TES

Zum besseren Verständnis ist diese Spezifikation nur partiell und nicht in der BEG-Syntax verfasst.

9.4.3 Beispiel – Resultierendes TES

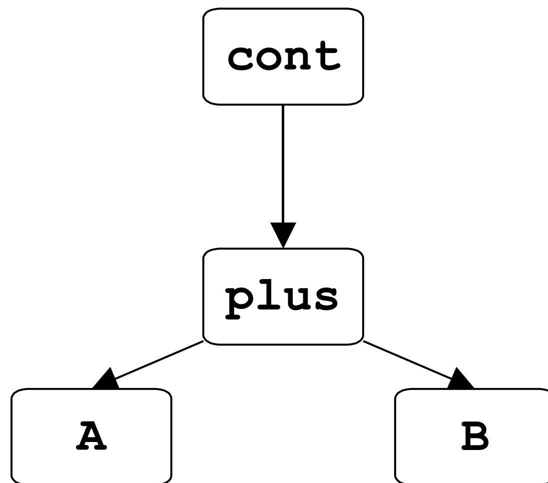
Entsteht aus der Spezifikation durch Umdrehen der Maschinenbeschreibung und Hinzufügen des initialen TES.

(1)	add $(R,\ Ea)$	$\rightarrow$	R
(2)	mov (Ea)	$\rightarrow$	R
(3)	bb	$\rightarrow$	R
(4)	R	$\rightarrow$	Ea
(5)	c	$\rightarrow$	Ea
(6)	di $(R,$ c$)$	$\rightarrow$	Ea
(a1)	plus (A,B)	$\rightarrow$	add (A,B)
(a2)	A	$\rightarrow$	mov (A)
(a3)	cont(plus(A,B))	$\rightarrow$	di (A,B)
(a4)	plus(A,B)	$\rightarrow$	plus (B,A)

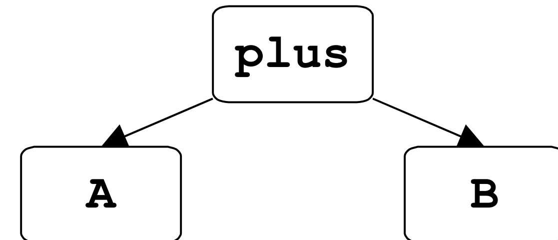
9.4.3 Beispiel – Regeln (a1) und (a3) von *TES*

Zwischensprachterme und Zielprogramm:

(a3)



(a1)



di A, B

add A, B

9.4.3 Beispiel – Resultierendes GTES

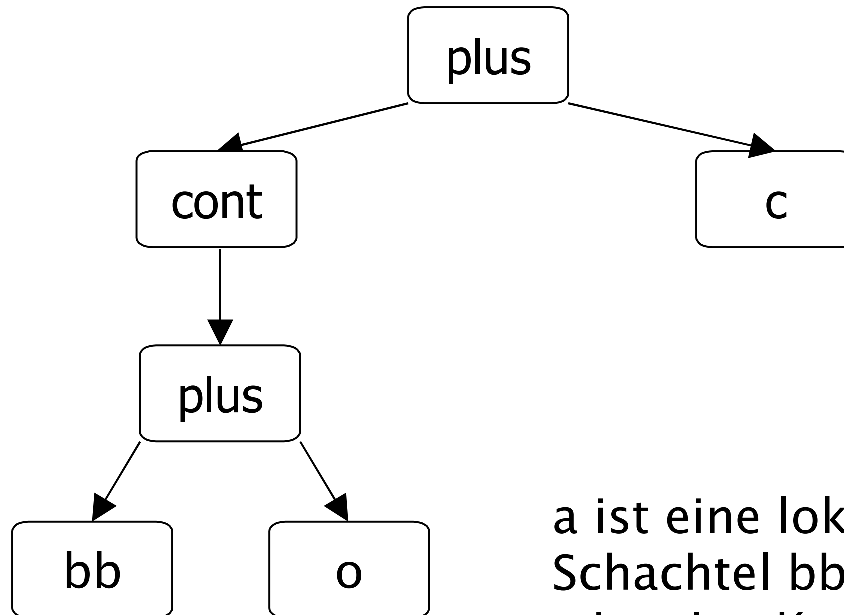
Anmerkung: offenbar werden alle Variablen mit den Ressourcen der Maschinenbeschreibung instantiiert.
Dieses GTES ist vollständig, aber nicht optimal bezüglich der Kosten.

(1)	add $(R,\ Ea)$	$\rightarrow$	R
(2)	mov (Ea)	$\rightarrow$	R
(3)	bb	$\rightarrow$	R
(4)	R	$\rightarrow$	Ea
(5)	c	$\rightarrow$	Ea
(6)	di $(R$, c$)$	$\rightarrow$	Ea
(g1)	plus (R,Ea)	$\rightarrow$	add (R,Ea)
(g2)	Ea	$\rightarrow$	mov (Ea)
(g3)	cont(plus(R,c))	$\rightarrow$	di (R,c)
(g4)	plus(c,R)	$\rightarrow$	plus (R,c)

9.4.3 Beispiel – Resultierender Baumautomat mit Kostenbewertung

Nr	Regel	Kosten	Aktion
(1)	$Ea \rightarrow R$	2	$R_1{:=}mov(Ea_1)$
(2)	$plus(R\ Ea) \rightarrow R$	4	$R_1^2{:=}add(R_1^1,Ea_1)$
(3)	$plus(Ea\ R) \rightarrow R$	4	$R_1^2{:=}add(R_1^1,Ea_1)$
(4)	$bb() \rightarrow R$	0	$R_1{:=}bb()$
(5)	$R \rightarrow Ea$	0	$Ea_1{:=}R_1$
(6)	$cont(P) \rightarrow Ea$	0	$Ea_1{:=}di(P_1,P_2)$
(7)	$c() \rightarrow Ea$	0	$Ea_1{:=}c()$
(8)	$c() \rightarrow Y$	0	$Y_1{:=}c()$
(9)	$plus(R\ Y) \rightarrow P$	0	$P_1{:=}R_1;\ P_2{:=}Y_1$
(10)	$plus(Y\ R) \rightarrow P$	0	$P_1{:=}R_1;\ P_2{:=}Y_1$

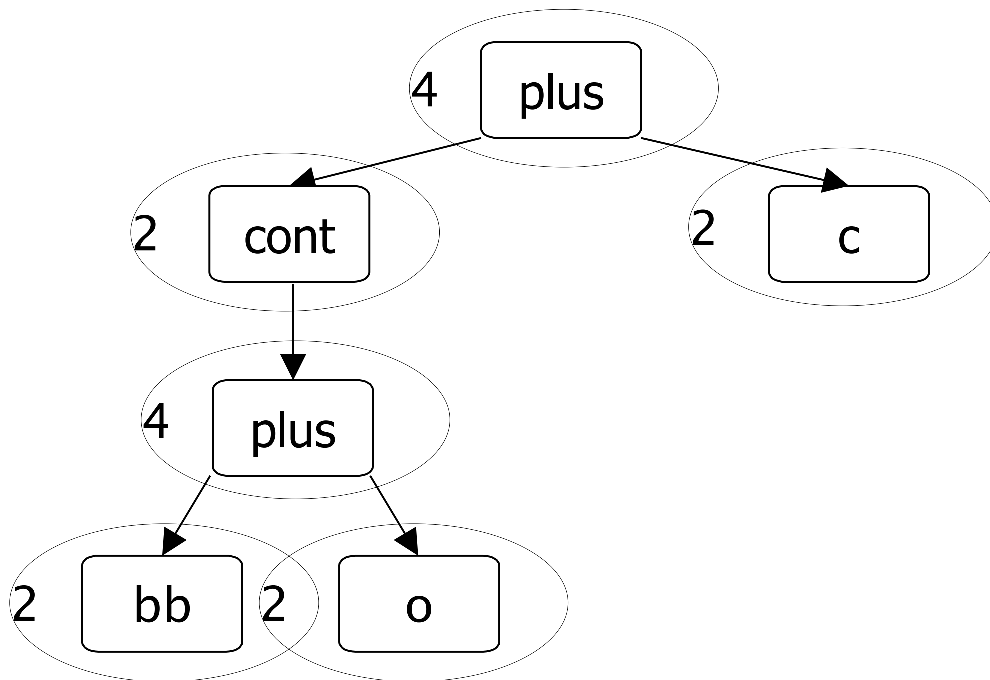
9.4.3 Beispiel – Zwischensprachterm für *a+c*



a ist eine lokale Variable in der Schachtel bb, wobei: o := offset(bb, a)
c ist eine Konstante

9.4.3 Beispiel – „dumme“ Überdeckung und Zielprogramm

Gesamtkosten 16:

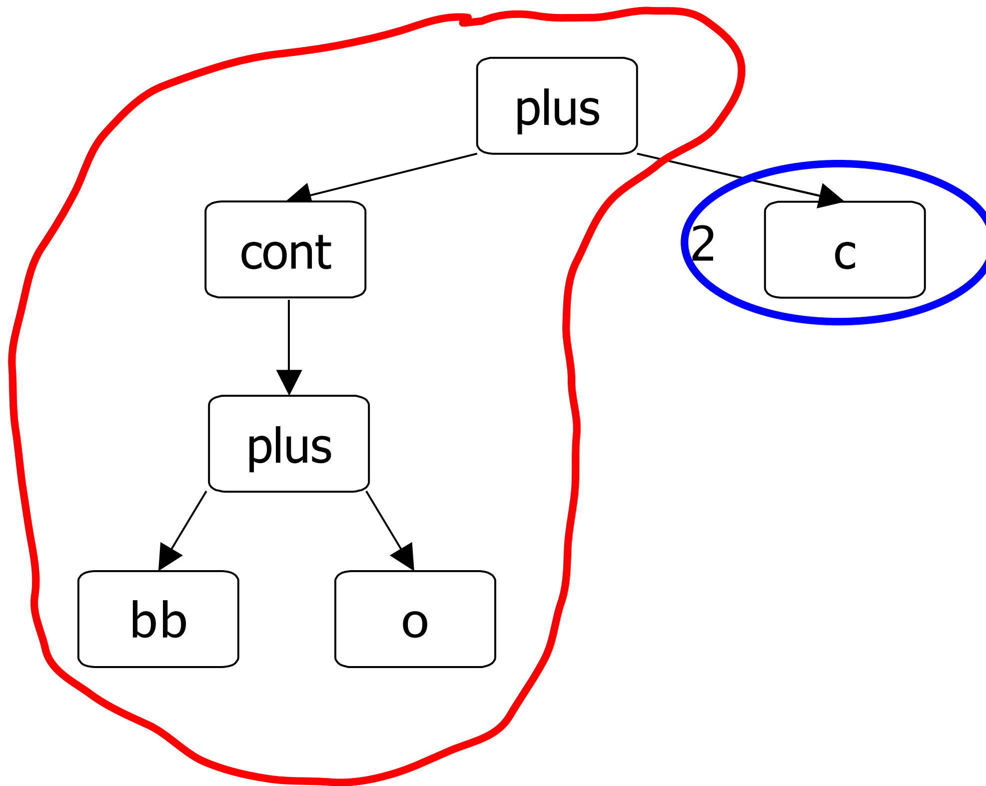


Maschinencode durch Makroxepansion:

```
R1 := mov bb
R2 := mov o
R2 := add R1, R2
R2 := mov di(R2, 0)
R1 := mov c
R1 := add R2, R1
```

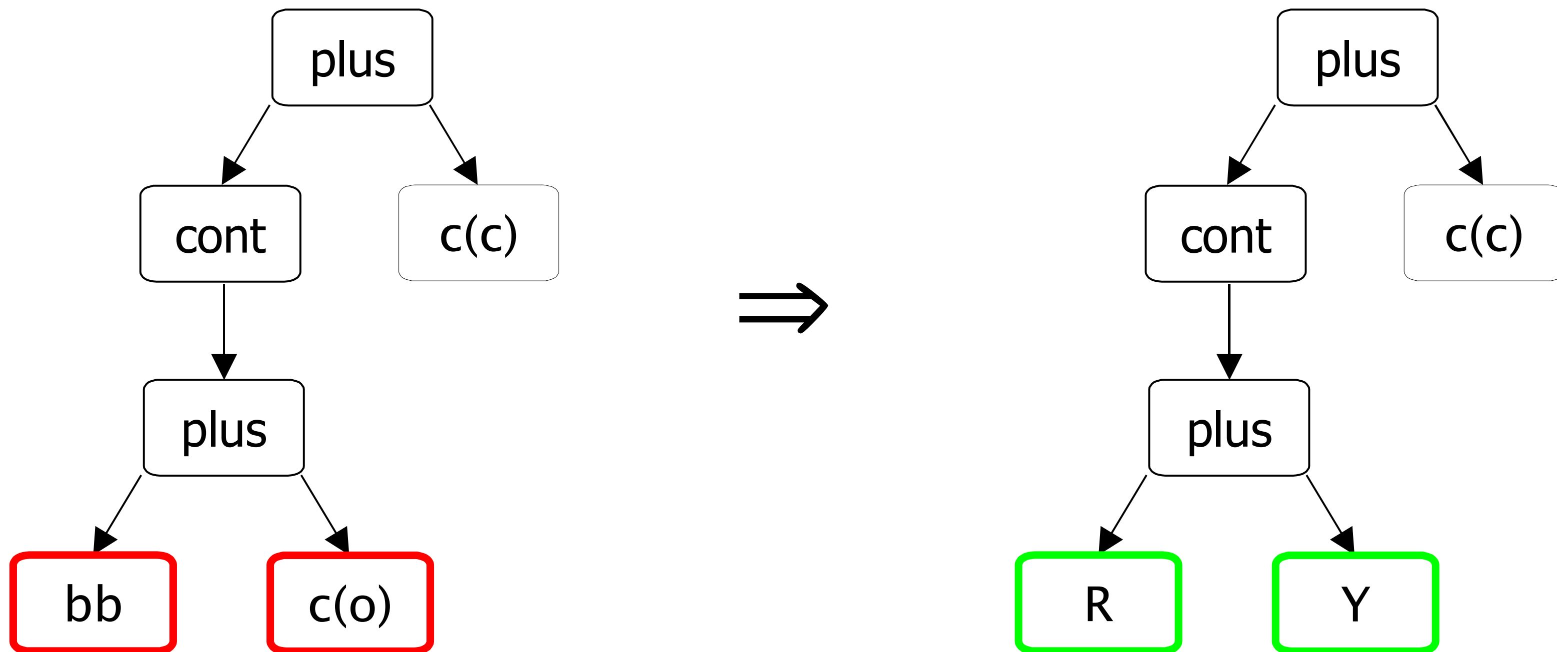
9.4.3 Beispiel – Effizientere Überdeckung und Programm

Gesamtkosten 6:

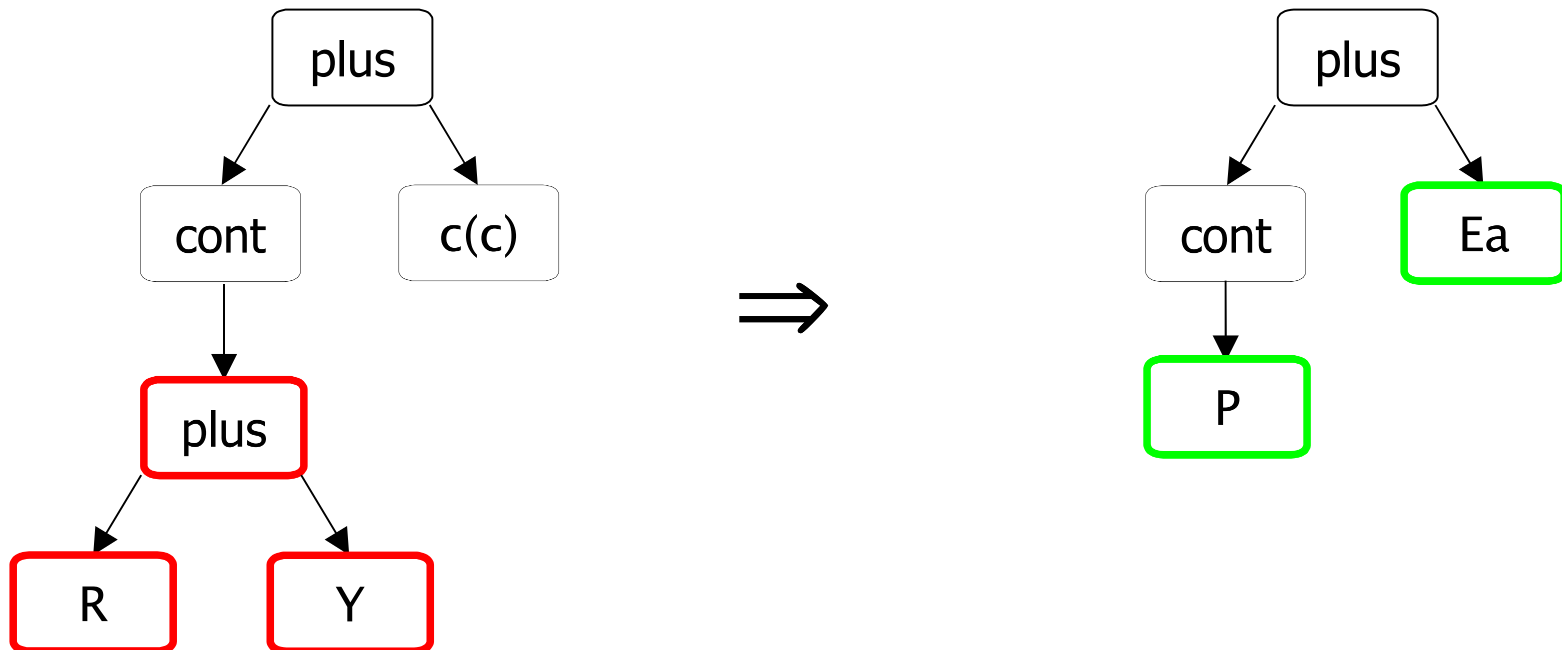



```
R1 := mov c
R1 := add R1, di(bb, o)
```

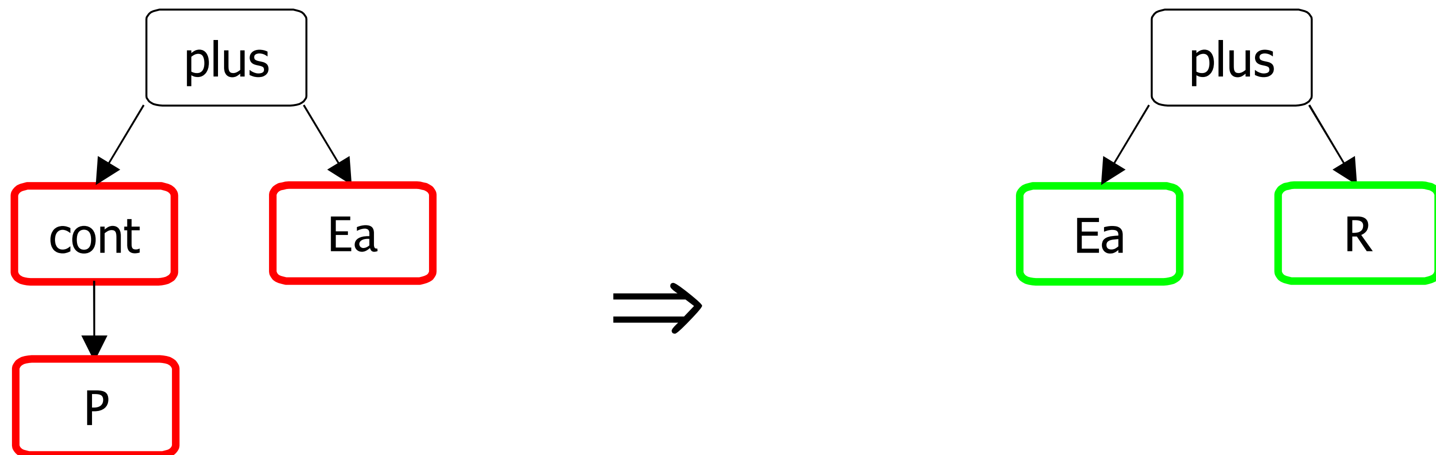
9.4.3 Beispiel – Regel 4 & 8



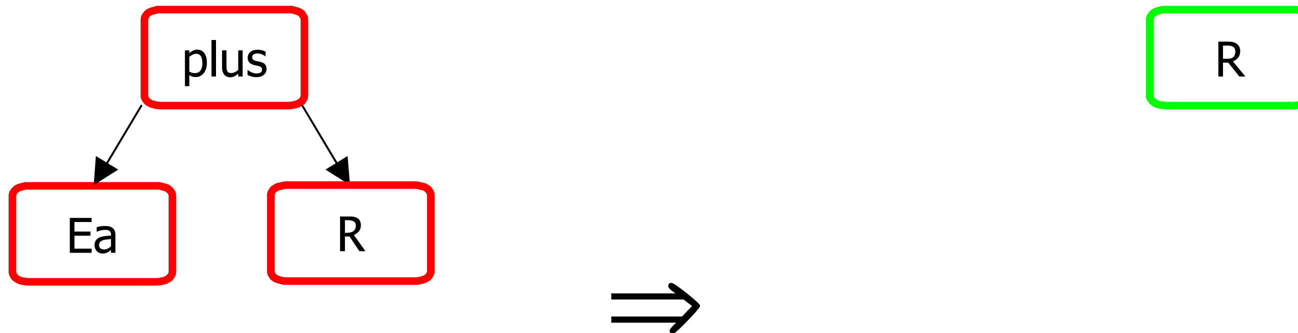
9.4.3 Beispiel – Regel 7 & 9



9.4.3 Beispiel – Regel 1 & 6



9.4.3 Beispiel – Regel 3



9.4.3 BEG –Spezifikation

Zwischensprachdefinition

- Nichtterminale
- Operatoren

Maschinenbeschreibung

- Register
- Nichtterminale

Überdeckungsregeln

Term	`RULE`	`...`
Kosten	`COST`	`...`
zu generierender Code	`EMIT`	`...`
direkt auszuwertender Code	`EVAL`	`...`
Ort des Resultats	`TARGET`	`...`

9.4.3 BEG – Zwischensprache

```
INTERMEDIATE_REPRESENTATION

NONTERMINALS
   BArg;

OPERATORS
BBase                                    -> BArg;
BConst ( value: long )                   -> BArg;
BCntent                  BArg            -> BArg;
BPlus                    BArg + BArg     -> BArg;
BSet                     Barg * BArg;
```

9.4.3 BEG – Maschinensprache

```
MACHINE_DESCRIPTION

REGISTERS
   (* Ganzzahl-Register 32bit *)
   eax, ebx, ecx, edx,
   (* Basepointer *)
   ebp, ... ;
NONTERMINALS
   (* general purpose registers *)
   reg        REGISTERS < eax..esp >;
   (* value *)
   immediate    ADRMODE
                   COND_ATTRIBUTES ( imm : tImmediate );
   (* base, offset *)
   address    ADRMODE ( ma : tMemAddress );
```

9.4.3 BEG – Abdeckungen

```
RULE immediate -> reg;
CONDITION { s.imm.value == 0 }
COST 1;
EMIT   {.      xorl {r reg}, {r reg} }
RULE immediate -> reg;
COST 2;
EMIT   {.      movl \${i immediate.imm}, {r reg} }
RULE reg -> address
COST 0;
EMIT   {       address.ma.base = reg;
               address.ma.offset = 0; }
RULE immediate -> address;
COST 0;
EMIT   {       address.ma.base = 0;
               address.ma.offset = immediate.imm.value; }
RULE address -> reg;
COST 2;
EMIT   {.      leal {a address.ma}, {r reg} }
```

9.4.3 BEG – Abdeckungen

```
RULE Bconst -> immediate;
COST 0;
EVAL    {       immediate. imm. value = BConst. value; }
RULE Bcontent address  -> reg;
COST 4;
EMIT    {.      movl {a address.ma}, {r reg} }
RULE Bplus address.a address.b -> address.c;
CONDITION {a.ma.base == 0 || b.ma.base == 0}
COST 0
EMIT    {       c.ma.base = a.ma.base ? a.ma.base : b.ma.base;
                c.ma.offset = a.ma.offset + b.ma.offset; }
RULE Bplus reg.a reg.b -> reg;
COST 2;
TARGET b;
EMIT    {.      addl {r a}, {r b} }
RULE Bbase -> reg<epb>;
COST 0;
```

Kapitel 9: Codeerzeugung

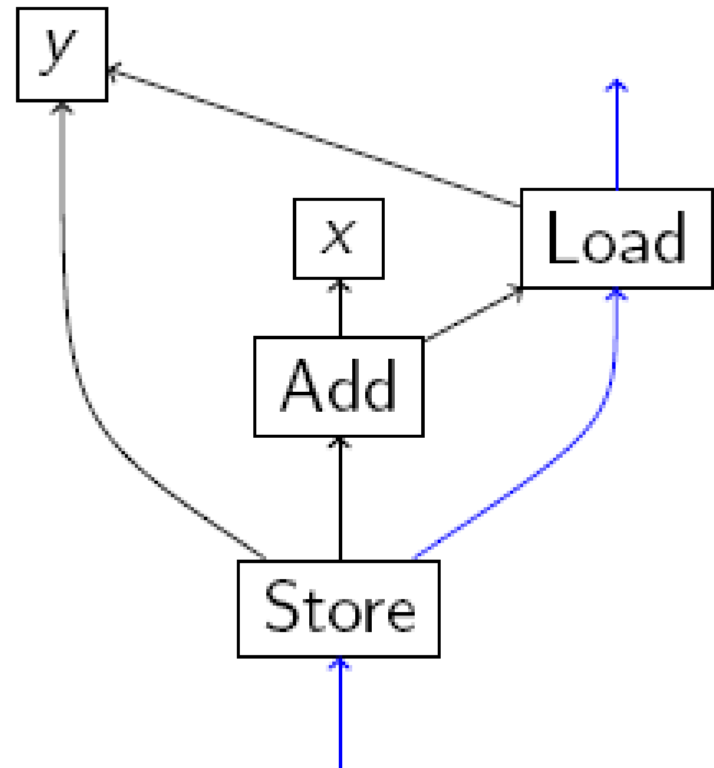
0. Einbettung
1. Grundlegendes
2. Optimierungen
   - Kurzauswertung
   - Algebraische Vereinfachungen
   - Registerverbrauch bei Ausdrücken
3. Befehlsauswahl
4. Befehlsauswahl mit Termersetzung
   - 4.1 Baumautomaten, TES
   - 4.2 BUPM, BURS, BEG
   - 4.3 Beispiel: BEG
5. Die letzten 10%

9.5 Die letzten 10 %

- Datenabhängigkeiten in Grundblöcken liefern (wegen Optimierungen) oft DAGs, keine Bäume.
  - Daher: Termersetzungsverfahren nur bedingt anwendbar
  - Es existieren Erweiterungen von TES für DAGs
    - CGGG von Boesler, IPD
    - PBQP-Verfahren von Eckstein, König und Scholz, TU Wien
    - BURS Code-Generator des Java Hotspot Compilers (SUN)
  - Oder: Brachial-Methode: Aufbrechen der DAGs in Bäume
- Generative Verfahren steuern Mustersuche bei, Nebenbedingungen sind gesondert zu berücksichtigen

9.5 Befehlsauswahl – Nebenbedingungen

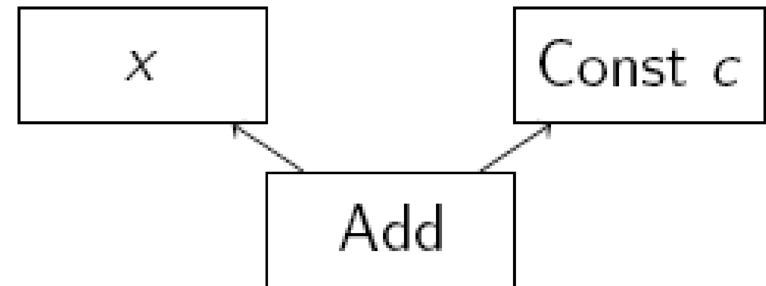
- Kann für ia32-Prozessoren in einem Befehl implementiert werden
- Allerdings darf das Load von keinem anderen Befehl verwendet werden
- Mit generativen Verfahren derzeit nicht möglich
- Ersetzung muss von Hand implementiert werden



Load/Modify/Store-Befehle bei ia32

9.5 Befehlsauswahl – Nebenbedingungen

- Um Konstante in den Befehl”hereinzuziehen“ müssen sie bestimmte Eigenschaften erfüllen:
  c = k ror (2k + 1),
  wobei k eine 8-bit Zahl ist
  (ror: rotate right).
- Laden von beliebigen Konstanten mit generativen Verfahren derzeit nicht möglich
- Ersetzung muss von Hand implementiert werden



Immediates bei ARM

9.5 Konventionen der Laufzeitumgebung

- wissenschaftlich uninteressant
- in der Praxis essentiell
- weitere Parametrisierung der Codeerzeugung durch Laufzeitumgebung
- bei den meisten Sprachen/Betriebssystemen nicht standardisiert, deswegen auch oft übersetzerabhängig

Hier nicht weiter behandelt

9.5 Spezialoptimierungen

Viele Prozessoren haben Eigenschaften, die bei der Befehlsauswahl nur schwer nutzbar sind.

- Spezielle Registerbänke
- Vektor-Einheiten
- Bedingte Befehle (Predication)
- usw.

9.5 Spezialoptimierungen

- Der PowerPC besitzt acht 4-bit Register um Vergleichsergebnisse zu speichern.
- Es gibt Rechenoperationen für diese Register (and, or, xor, usw.)
- Sie können direkt von Sprungoperationen verwendet werden.
- Ideal, um boolesche Terme (Bedingungen) zu berechnen und die Ergebnisse als Sprungbedingung zu nutzen.
- Das spart Allzweck-Register
- Allerdings sind die Bedingungsregister nicht auslagerbar.
  - Man kann also niemals an mehr als 8 Bedingungen gleichzeitig arbeiten
  - und was tut man bei einem Prozeduraufruf?
    Antwort: danach sind alle Bedingungsregister undefiniert !?!

9.5 Befehlsauswahl – Fazit

- Übersetzer muss eine Menge an Zusatzaufgaben erledigen, die über die eigentliche Code-Erzeugung hinausgehen
- Generische Verfahren
  - ermöglichen bequeme Spezifikation der Muster
  - Eigenheiten der Zielmaschinen nur ungenügend darstellbar
  - Die „letzten 10%“ lassen sich nur schwer erledigen
- Daher oft Hybrid-Verfahren (softwaretechnisch ”bedenklich“)
- Ausprogrammieren von Hand ist zwar aufwändig, aber nicht unbedingt von Nachteil