Kapitel 8

Codeerzeugung

KIT Karlsruhe Institute of Technology

Kapitel 8: Codeerzeugung

1 Einbettung

2 Einführung in x86-Assembler

3 Registerverbrauch bei Ausdrücken

4 Befehlsauswahl

5 Befehlsauswahl mit Termersetzung
- Beispiel: Termersetzung
- Baumautomaten, TES
- BUPM, BURS, BEG
- Beispiel: BEG

Die Synthesephase

Aufgabe:

attributierter Strukturbaum $\rightarrow$ ausführbarer Maschinencode

Problem:

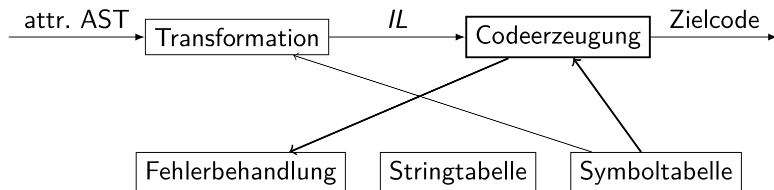
Außer bei Codeerzeugung für die abstrakte Quellsprachenmaschine *QM*, eine Kellermaschine, sind alle Aufgaben „guter“ Codeerzeugung NP-vollständig; Qualität also nur näherungsweise erreichbar

KIT Karlsruhe Institute of Technology

Zerlegung der Synthesephase

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

Eingliederung in den Compiler



Aufgaben der Codeerzeugung

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** (?)

Wiederholung: 2 Klassen von Zwischensprachen

1. Code für **Kellermaschine mit Heap**, z.B. Pascal-P, ..., JVM, CLR (.net)
   - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
   - Datentypen und Operationen auf Daten entsprechen weitgehend der *QM*, zusätzlich Umfang und Ausrichtung im Speicher berücksichtigen
2. Code für **RISC-Maschine mit unbeschränkter Registerzahl** und (stückweise) linearem Speicher
   - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
   - Datentypen entsprechen Zielmaschine einschließlich Umfang und Ausrichtung im Speicher
   - Operationen entsprechen Zielmaschine (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

Wiederholung: 3 Unterklassen

Im Fall „Code für RISC-Maschine mit unbeschränkter Registerzahl" drei Darstellungsformen:

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

KIT Karlsruhe Institute of Technology

Wiederholung: Programmstruktur der *IL*

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

Kapitel 8: Codeerzeugung

1 Einbettung

2 Einführung in x86-Assembler

3 Registerverbrauch bei Ausdrücken

4 Befehlsauswahl

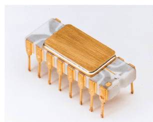
5 Befehlsauswahl mit Termersetzung
- Beispiel: Termersetzung
- Baumautomaten, TES
- BUPM, BURS, BEG
- Beispiel: BEG

Einführung

Die folgenden Folien geben einen Überblick über die weit verbreitete x86-Architektur die in PCs benutzt wird. Die Einführung ist aus der Sicht des Compilerbauers:

- Schwerpunkt liegt auf Instruktionssatz und Performance
- Wenig Beachtung der Hardware-Ebene und Peripherie

Geschichte



Intel 4004

Intel 8086

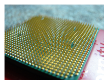
1971 **4004 Prozessor**
4-Bit Datenbus; 8-Bit Befehlssatz

1978 **8086/8088 Prozessor**
16-Bit Prozessoren, 1-MByte Adressraum; 8088 besitzt 16-Bit Datenbus.

1982 **80286 Prozessor „286er“**
MMU bringt Speicherschutz („Protected Mode“); 24 Bit Segmente ⇒ 16 MByte Arbeitsspeicher.

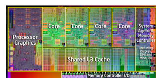
1985 **80386 Prozessor „386er“**
32-Bit Adressbus ⇒ 4 GBytes Arbeitsspeicher; Virtuelle Speicherverwaltung (Paging); 32-Bit Befehlssatz (IA-32); Virtual-8086 Mode

Geschichte



AMD Athlon 64

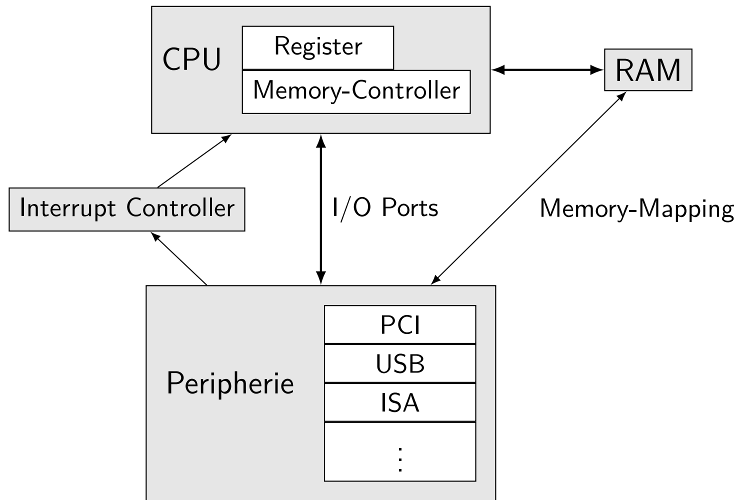
1989	**80486 Prozessor „486er“** Level1-Cache für Instruktionen; Integrierte Gleitkommaeinheit (x87)
1993	**Pentium Prozessor, K5, K6** Schnellerer Virtual 8086 Mode; MMX-Befehlssatz (SIMD – single instruction multiple data)
1995	**P6-Prozessoren, Athlon** SSE-Befehlserweiterung mit 128-Bit Registern
ab 2000	**Pentium 4, Pentium M, Core, Opteron, Phenom** SSE2 - SSE5. 64-Bit Modus („AMD64“, „Intel 64“)
2008	**Core-i-Familie, Bulldozer** AVX Instruktionssatz



Intel Core i7

Verbindung zur Außenwelt

Programmiermodell:



Register

- 8 General Purpose Register: A, B, C, D, SI, DI, BP, SP
  - in 32-Bit Form: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
  - in 16 Bit Form: AX, BX, CX, DX, SI, DI, BP, SP
  - in 8 Bit Form: AH/AL, BH/BL, CH/CL, DH/DL
- Flagsregister (Vergleichsergebnisse, Überläufe, . . .)
- Segmentregister: CS, DS, SS, ES, FS, GS
- Instruction Pointer (EIP)
- x87 Register: ST0-ST7 als Stack organisiert, x87 Status- und Control-Words
- weitere Register (Control, Debug Registers, Performance Counters, ...)

Register Erweiterungen

- MMX Register: MM0-MM7 – 64bit als:
  - 8x8, 4x16, 2x32 oder 1x64 bit integer
- SSE Register: XMM0-XMM7 – 128 Bit als:
  - 4x32 oder 2x64 bit float
  - 16x8, 8x16, 4x32 oder 2x64 bit integer
- SEE Register (AVX): 256 Bit

AMD64/Intel 64:

- 64-Bit: RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8-R18
- RIP, RFlags
- SSE XMM0-XMM15

Flagsregister

0		2		4		6	7	8	9	10	11	12-15	16	17	18	19	20	21	22-31
CF		PF		AF		ZF	SF	TF	IF	DF	OF		RF	VM	AC	VIF	VIP	ID	

Arithmetische Flags:

Bit	Name	Beschreibung
CF	Carry	Carry oder Borrow nach höchstwertigstem Bit
ZF	Zero	Ergebnis ist 0
SF	Sign	höchstwertigstes Bit im Ergebnis ist gesetzt (negativer Wert)
OF	Overflow	Carry oder Borrow nach zweithöchsten Bit (für vorzeichenbehaftete Zahlen)

Vergleiche

Nach CMP oder SUB Befehl:

	Unsigned		Signed	
Vergleich	Name	Bits	Name	Bits
$<$	B	CF	L	genau SF oder OF
$\leq$	BE	CF oder ZF	LE	ZF oder genau SF or OF
$=$	E	ZF	E	ZF
$\neq$	NE	$\neg$ZF	NE	$\neg$ZF
$>$	A	$\neg$CF und $\neg$ZF	G	$\neg$ZF und weder SF noch OF
$\geq$	AE	ZF und $\neg$CF	GE	ZF oder weder SF noch OF

CISC vs. RISC

x86 ist eine CISC (complex instruction set computing) Architektur.

- Reichhaltiger Befehlssatz, insbesondere viele verschiedene Adressierungsmodi für Daten.
- Programme sind kompakt.
- 2-Address-Code. Ziel einer Operation muss gleich einem der Quelloperanden sein: Befehle müssen Form $A = A + B$ haben.
- Prozessoren übersetzen CISC-Befehle intern in Microcode mit RISC Eigenschaften (um Pipelining zu ermöglichen).

Adressierungsmodi

Die meisten Befehle erlauben verschiedene Varianten um ihre Operanden zu erhalten:

- Konstante Werte (immediates)
- Register
- „Address-Mode“: Wert aus Speicher laden

Mögliche Adressberechnungen

$$\text{addr} = \text{Const} + \text{Base} + \text{Index} * \text{Scale}$$

- **Const** – 8-, 16- oder 32-Bit Konstante, die im Befehl kodiert wird.
- **Base** – beliebiges Registers
- **Index** – beliebiges Registers außer ESP
- **Scale** – 1, 2, 4 oder 8
- Komponenten sind optional, mindestens Const oder Base muss gegeben sein.

Assembler Syntax (AT&T)

Befehle

- Register: `%eax`, `%esp`, ...
- Konstanten: `$5`, `$0x32`, `symbol`, ...
- Address-Mode: `Const`, `Const(Base)`, `Const(Base,Index,Scale)`
- Befehle bekommen ein Suffix um ihre Breite zu signalisieren: `b`, `w`, `l`, `q` für 8-, 16-, 32- oder 64-bit Breite Operationen.
- Bei mehreren Operanden wird erst der Quelloperand, dann der Zieloperand angegeben. `addl $4, %eax`

Beispiele

```
xorl %eax, %eax
subl $4, %esp
movl array+20(%eax,%ecx,4), %eax
incl (%esp)
```

Assembler Syntax (AT&T)

Assembler Direktiven

- Label: `name:` – Namensvergabe für Programmstellen
- Export/Import: `.globl name` – Linker löst Namen auf.
- Daten/Code-Segment: `.data`, `.text`
- Datenwerte: `.byte`, `.word`, `.long`

Beispiel

Globale Variable **int** var = 42;

```
.data
.globl var
var:
.long 42
```

Grundlegende Befehle

`mov`	Daten kopieren
`add`	Addition
`sub`	Subtraktion
`neg`	Negation
`inc`	Addition mit 1
`dec`	Subtraktion mit 1
`imul`	Multiplikation
`mul`	(unsigned) Multiplikation, Ergebnis in `EAX:EDX`
`imul`	mit einem Operand wie `mul` aber signed statt unsigned
`div`	Division. Dividend stets in `EAX:EDX`, Divisor wählbar
`and`	Bitweises Und
`or`	Bitweises Oder
`xor`	Bitweises exklusives Oder
`not`	Bitweises invertieren
`shl`	Linksshift
`shr`	Rechtsshift
`sar`	(signed) Rechtsshift

Grundlegende Befehle

`jmp`	unbedingter Sprung
`cmp`	Werte vergleichen
`jCC`	bedingter Sprung
`setCC`	Register abhängig von Testergebnis setzen
`call`	Unterfunktion aufrufen
`ret`	Aus Funktion zurückkehren
`push`	Wert auf den Stack legen und ESP vermindern
`pop`	Wert vom Stack legen und ESP erhöhen
`int`	„interrupt"-Routine aufrufen (nötig für Systemaufrufe)
`lea`	Führt Adressrechnung durch, schreibt Ergebnis in Register „3-Adressmode Addition".

Funktionsaufrufe

Register:

- Stackzeiger (ESP) zeigt auf Ende des Stacks
- Rahmenzeiger (engl. Basepointer; EBP) zeigt auf Beginn des Activation Records

Befehle:

- call: Legt Rücksprungadresse auf den Stack, springt zu Ziel.

Typische Aufrufkonventionen für x86 („C“):

- Parameter auf den Stack von rechts nach links auf den Stack
- Ergebnisrückgabe in EAX/ST0
- Aufrufer räumt Parameter vom Stack.

Weitere Konventionen z.B. „Pascal“, „Fastcall“

Beispiel – printf aufrufen

```
    .data
.STR0:
    .string "Hello\n"

    .text
.globl main
main:
    # Rahmenzeiger sichern und neu setzen
    pushl %ebp
    movl %esp, %ebp

    # Argument auf den Stack legen
    pushl $.STR0
    # Funktion aufrufen
    call printf
    # Argument vom Stack entfernen
    addl $4, %esp

    # Ergebnis für "main" setzen
    movl $0, %eax
    # Alten Rahmenzeiger wiederherstellen und zurückkehren
    movl %ebp, %esp
    popl %ebp
    ret
```

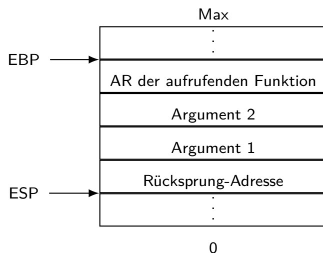
Beispiel – Funktion die 2 Zahlen addiert

```
    .text
.globl add
add:
    # Rahmenzeiger sichern und neu setzen
    pushl %ebp
    movl %esp, %ebp

    # Argumente laden
    movl 8(%ebp), %eax
    movl 12(%ebp), %edx

    # Addieren (%eax = %eax + %edx)
    addl %edx, %eax

    # Alten Rahmenzeiger wiederherstellen und zurückkehren
    movl %ebp, %esp
    popl %ebp
    ret
```

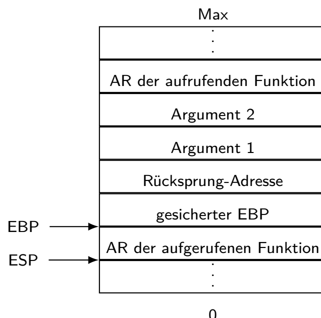
Funktionsprolog

pushl %ebp
movl %esp, %ebp
subl $XX, %esp # XX bytes für activation record allozieren

vorher:



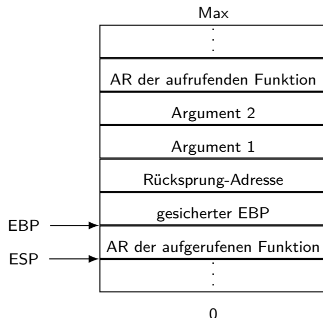
nachher:



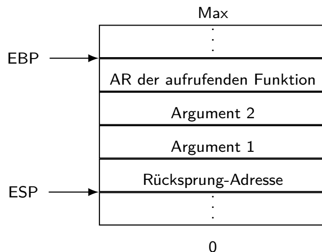
Epilog

```
movl %ebp, %esp
popl %ebp
ret
```

vorher:



nachher:



Fibonacci-Funktion

```
.globl fib
    .p2align 4,,15
fib:
# Argument laden
    movl 4(%esp), %edx
# Fälle n==0 und n==1 behandeln
    cmpl $1, %edx
    ja .continue
    movl %edx, %eax
    ret
# Callee−saves sichern
.continue:
    pushl %ebp
    pushl %edi
```

Fibonacci Funktion (Fortsetzung)

```
# fib(n−1) aufrufen
    movl %edx, %ebp
    decl %edx
    pushl %edx
    call fib
    movl %eax, %edi
# fib(n−2) aufrufen
    leal −2(%ebp), %edx
    pushl %edx
    call fib
# fib−Argumente vom Stack nehmen
    addl $8, %esp
# Ergebnis berechnen
    addl %edi, %eax
# Callee−saves wiederherstellen
    popl %edi
    popl %ebp
    ret
```

Aufrufe ohne Rahmenzeiger

Falls Größe des Activation Records statisch bekannt:

- Adressierung relativ zum Stackpointer möglich
- Basepointer als normales Register für Berechnungen
- Schwierigkeiten Stackframes im Debugger zu erkennen (mit modernen Debugformaten allerdings möglich)
- Funktioniert nicht bei dynamischen Arrays/alloca.

Optimieren für x86

Eine Auswahl:

- Registerallokation(!)
- Ausnutzen von Adressierungsmodi
- Alignment von Funktionen, Schleifen (auf Cachezeilen)
- Auf modernen CPUs simple Befehle oft besser (`and`, `test` statt `bt`)
- Zugriffe mit kleinen Bitbreiten vermeiden

SSE (Streaming SIMD Extensions)

Eingeführt mit Pentium III (1999). Zusätzliche Befehle für Multimedia nach dem SIMD (Single Instruction Multiple Data) Prinzip. Bei angepasstem Code oft deutliche Geschwindigkeitssteigerungen.

- 8 zusätzliche 128-Bit Register (für jeweils 4 float/int oder 2 double Werte)
- Neue Befehle (Arithmetic, Comparison, Logical, Shuffle, Conversion, ...)

Nachteile

- Programmierung nur manuell oder mit speziellen Bibliotheken.
- Code läuft nur auf modernen CPUs.

Skalarprodukt in ANSI C

```
float scalar_product(float *xs, float *ys, int k) {
    float result = 0.0;

    for (int i = 0; i < k; ++i)
        result += xs[i] * ys[i];

    return result;
}
```

Skalarprodukt SSE (gcc mit Builtins)

```
float scalar_product_sse(float *xs, float *ys, int k) {
    /* Datentyp für SSE Werte */
    typedef float v4sf __attribute__((vector_size(16)));
    /* Immer 4 Werte auf einmal berechnen */
    v4sf result = {0, 0, 0, 0};
    assert(k % 4 == 0);
    for (int i = 0; i < k; i += 4) {
        /* Werte in SSE Register laden, multiplizieren, addieren */
        v4sf X = __builtin_ia32_loadups(&xs[i]);
        v4sf Y = __builtin_ia32_loadups(&ys[i]);
        v4sf mul = __builtin_ia32_mulps(X, Y);
        result = __builtin_ia32_addps(result, mul);
    }
    /* Werte zurück in normale Variable, Addieren */
    float temp[4]; __builtin_ia32_storeups(temp, result);
    return temp[0] + temp[1] + temp[2] + temp[3];
}
```

⇒ etwa doppelt so schnell als nicht-SSE Version auf Core 2 Duo.

Literatur

- Ausführliche Dokumentation:
  `http://www.intel.com/products/processor/manuals/`
- Knappe Übersicht:
  `http://www.posix.nl/linuxassembly/nasmdochtml/nasmdoca.html`
- Gut organisierte Sammlung von Dokumenten zu x86:
  `http://www.sandpile.org`
- Aufrufkonventionen und Optimierungstechniken:
  `http://www.agner.org/optimize/`

Kapitel 8: Codeerzeugung

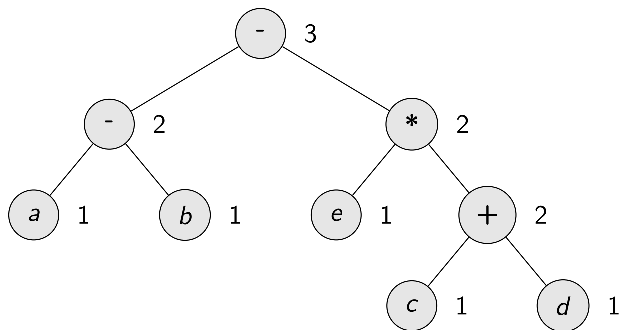
1. Einbettung
2. Einführung in x86-Assembler
3. **Registerverbrauch bei Ausdrücken**
4. Befehlsauswahl
5. Befehlsauswahl mit Termersetzung
   - Beispiel: Termersetzung
   - Baumautomaten, TES
   - BUPM, BURS, BEG
   - Beispiel: BEG

Registerverbrauch bei Ausdrücken

Problematik:

- Um eine Operation zu berechnen, berechne zunächst ihre Operanden.
- Wert eines Operanden wird in Register gespeichert; Für weitere Operanden steht ein Register weniger zur Verfügung.
- $\Rightarrow$ Optimaler Code berechnet Operanden mit kleinstem Registerverbrauch zuletzt.

Wiederholung: Baum mit Ershov-Zahlen



Ausdruck: $(a - b) - e * (c + d)$

Wiederholung: Ershov-Zahlen

Ershov-Zahlen geben die Zahl der Register an, die zur Auswertung eines Ausdrucks benötigt werden.
Markieren eines Ausdrucksbaums:

1. Kennzeichne alle Blätter mit 1.
2. Bei 2 Kindern:
   - gleiche Kennzeichnung der Kinder: übernimm Kennzeichnung plus 1
   - sonst: nimm größte Kennzeichnung der Kinder
3. Allgemein: Für absteigend sortierte Markierungen der Kinder $M_1, \ldots, M_n$:

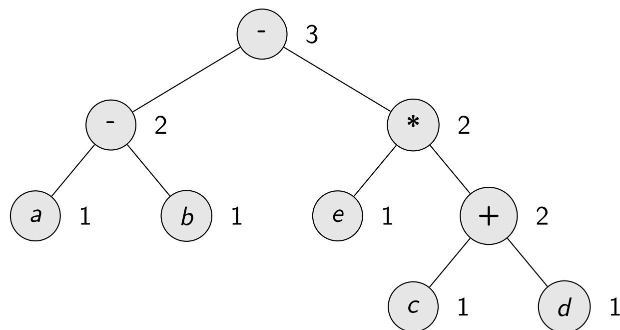
$$\max(M_1, M_2 + 1, \ldots, M_n + (n - 1))$$

Codeerzeugung

Mit rekursivem Algorithmus. Register werden relativ zu momentaner Basis b verwendet $(R_b, R_{b+1}, \ldots)$. Beginne an der Wurzel mit Basis $b = 0$. Pro Knoten:

1. Sortiere Kinder nach absteigender Ershov-Zahl (Registerverbrauch).
2. Erzeuge Code für Kinderknoten $K_0, K_1, \ldots, K_{n-1}$ mit Basis $b, b+1, \ldots, b+(n-1)$
3. Erzeuge Operation: OP $R_b, R_b, R_{b+1}, \ldots, R_{b+n-1}$.

Anwendung




```
LD R0, d
LD R1, c
ADD R0, R1, R0
LD R1, e
MUL R0, R1, R0
LD R1, b
LD R2, a
SUB R1, R2, R1
SUB R0, R1, R0
```

Auslagern

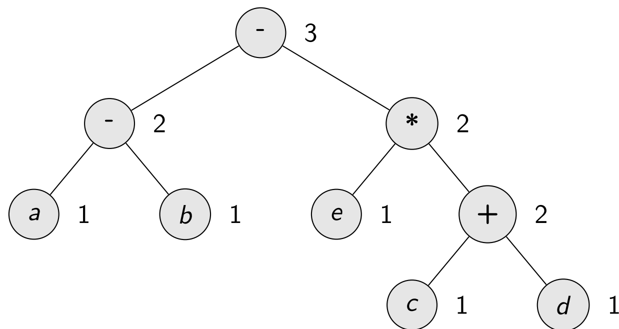
Modifikation bei beschränkter Registerzahl k:

1. Sortiere Kinder nach absteigender Ershov-Zahl (Registerverbrauch).
2. $b_0 \leftarrow b$.
3. Für jeden Kinderknoten $K_i \in \{K_0, \ldots, K_{n-1}\}$.
   a. Erzeuge Code für K_i mit Basis b.
   b. Falls Markierung des nächsten Kindes plus weitere Operanden $M_{i+1} + n - i$ größer als k: Erzeuge Auslagerungsbefehl ST t_x, R_b (t_x ist eine neue Speicherstelle)
   c. $b \leftarrow b + 1$.
4. Lade ausgelagerte Operanden in Register $R_b, R_{b+1}, \ldots$: LD R_b, t_x.
5. Erzeuge Operation: OP $R_{b_0}, R_{b_0}, R_{b_0+1}, \ldots, R_{b_0+n-1}$.

Ershov-Pseudocode für $n = 2$ und $r \geq 2$

```
ershov(t,b) { // t AST mit Ershovzahl M_t, b Zielregister
  if (t ist Blatt für Variable x) {
    println("LD r"+b+", "+x);
  } else {
    s1 = größerer Unterbaum von t;
    s2 = kleinerer Unterbaum von t;
    ershov(s1, b);
    if (M_t ≤ r) { // kein spill nötig
      ershov(s2, b+1);
    } else { // spill
      v = neue temporäre Variable;
      println("ST "+v+", r"+b);
      ershov(s2, b);
      // b+1 ist frei, da M_t = M_{s_i} + 1 für mindestens einen Unterbaum s_i
      println("LD r"+(b+1)+", "+v);
    }
    println(op(t)+" r"+b+", r"+b+", r"+(b+1));
    // bzw. falls op(t) s1=rechter Unterbaum:
    println(op(t)+" r"+b+", r"+(b+1)+", r"+b);
} }
```

Anwendung bei 2 Registern




```
LD R0, d
LD R1, c
ADD R0, R1, R0
LD R1, e
MUL R0, R1, R0
ST v, R0
LD R1, b
LD R0, a
SUB R0, R0, R1
LD R1, v
SUB R0, R0, R1
```

Verallgemeinerung

Kombiniere Operation mit Laden:

LD R1, b
LD R2, c
MUL R1, R2
LD R2, a
ADD R1, R2
	$\Rightarrow$	LD R1, b
		MUL R1, c
		ADD R1, a

Wann ist Postfixform optimal?

Gegeben eine Maschine mit n uniformen Registern R_i und Befehlen der Form:

- $R_i :=$ Speicherplatz,
- Speicherplatz $:= R_i$,
- $R_i := op(v_j, \ldots, v_k)$, v_h Register oder Speicherplatz.

Programm $P_1 S_1 P_2 \ldots P_{s-1} S_{s-1} P_s$ in Normalform, wobei 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.

Starkes Normalformtheorem

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, Gleitkomma- und Ganzzahloperationen erfüllt außer Ganzzahlmultiplikation und -division.

Registerklassifikation

Befehlsauswahl ist abhängig von Eigenschaften (Attributen) des Zwischencodes und der Zielbefehle.

- Klassifikation der Register:
  - allgemeine, Gleitkomma-, Adressregister
  - reservierte Register für Rücksprungadresse usw.
  - Doppelregister nur für gerade/ungerade Paare, z.B. (R2, R3)
  - Welche Operanden dürfen/müssen in welche Register?
- Nutzung der Adressierungspfade statt expliziter Berechnung

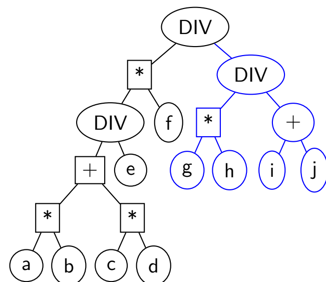
Gerade/Ungerade Register

Ad-hoc Vorgehen! Siehe auch Kapitel Registerzuteilung.

```
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;
    mul: ungerade;
    div: gerade
  end;
  ausdruck[3].wunsch := case operator.operator of
    plus, minus:
      if ausdruck[1].wunsch=paar then gerade
      else ausdruck[1].wunsch;
    mul: ungerade;
    else beliebig
  end;
```

Merke: Attribut nicht bindend, aber dann zusätzliche Kosten

Optimale Reihenfolge kann springen



Rund: einfach langes Resultat
Rechteckig: doppelt langes Resultat

```
LD R0, a        (R0,R1):= a*b
MUL R0, b
LD R2, c        (R2,R3):= c*d
MUL R2, d
ADD R1, R3      (R0,R1):= (R0,R1)+(R2,R3)
ADC R2          addiere Übertrag
ADD R0, R2
DIV R0, e       R0:= (R0,R1) DIV e
LD R2, g        (R2,R3):= g*h
MUL R2, h
LD R1, i        R1:= i+j
ADD R1, j
DIV R2, R1      R2:= (R2,R3) DIV R1
MUL R0, f       (R0,R1):= R0*f
DIV R0, R2      R0:= (R0,R1) DIV R2
```

Geschlossene Reihenfolge (Ershov): 1 Register mehr

Kapitel 8: Codeerzeugung

1 Einbettung

2 Einführung in x86-Assembler

3 Registerverbrauch bei Ausdrücken

4 Befehlsauswahl

5 Befehlsauswahl mit Termersetzung
- Beispiel: Termersetzung
- Baumautomaten, TES
- BUPM, BURS, BEG
- Beispiel: BEG

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.

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

Bewertung:

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

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	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	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				x		x		x	x			x			x		
lreg(l,desire)				x				x				x				x				x				x				x				x
gen(A l r)		x	x	x										x	x	x		x	x	x										x	x	x
gen(AR l r)	x												x				x												x			
gen(S l r)						x	x	x		x	x	x										x	x	x		x	x	x				
gen(SR l r)					x				x												x				x							
gen(LCR l l)							x			x			x	x	x	x	x	x	x	x		x					x					
free(r)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	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	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Tabelle: Vollständige Entscheidungstabelle

Auswertung: Bedingungen als Index

Kapitel 8: Codeerzeugung

1 Einbettung

2 Einführung in x86-Assembler

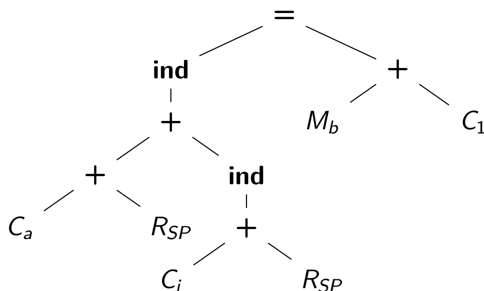
3 Registerverbrauch bei Ausdrücken

4 Befehlsauswahl

5 Befehlsauswahl mit Termersetzung
- Beispiel: Termersetzung
- Baumautomaten, TES
- BUPM, BURS, BEG
- Beispiel: BEG

Zwischencodebäume

Weitere mögliche Form der Zwischencodes (Geeignet für Bottom-Up Pattern Matching)



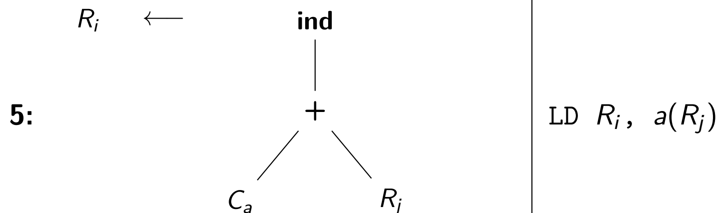
Zwischencodebaum für a[i] = b + 1

Unterschied zu AST: Explizite Adressierung / Dereferenzierung

Baumersetzungsregeln (1/3)

	Regel	Code
1:	$R_i \longleftarrow C_a$	`LD` R_i `,` `#a`
2:	$R_i \longleftarrow M_x$	`LD` R_i `,` x
3:	$M \longleftarrow =(M_x, R_i)$	`ST` x `,` R_i
4:	$M \longleftarrow =(\textbf{ind}(R_i), R_j)$	`ST *` R_i `,` R_j

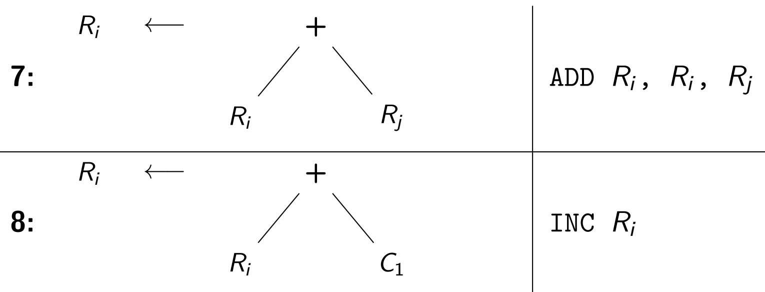
Baumersetzungsregeln (2/3)



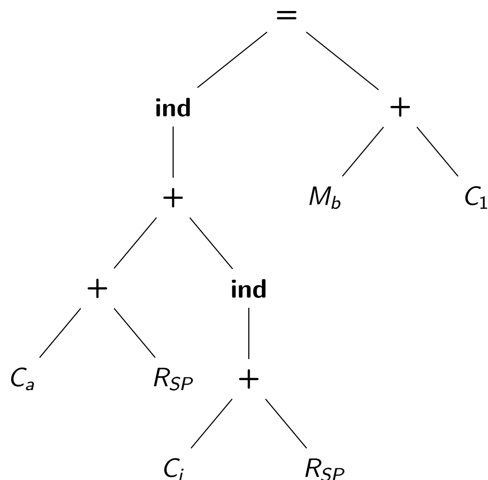
6: $R_i \longleftarrow$ + (R_i, ind (+ (C_a, R_j)))

ADD R_i, R_i, $a(R_j)$

Baumersetzungsregeln (3/3)

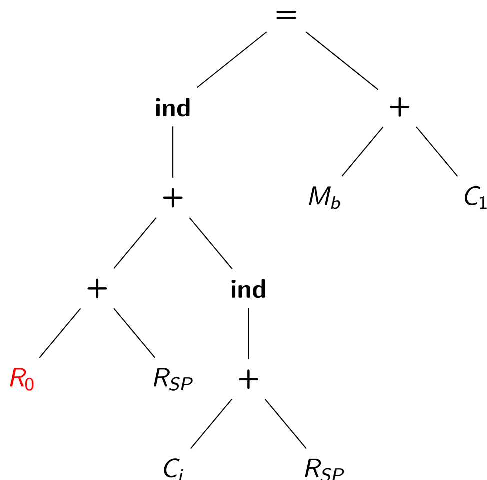


Beispiel Baumübersetzungsverfahren



Anwendung von Regel **1** möglich

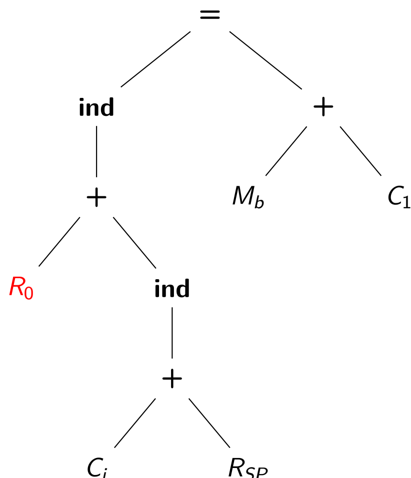
Beispiel Baumübersetzungsverfahren



`LD` R_0`, #a`

Anwendung von Regel **7** möglich

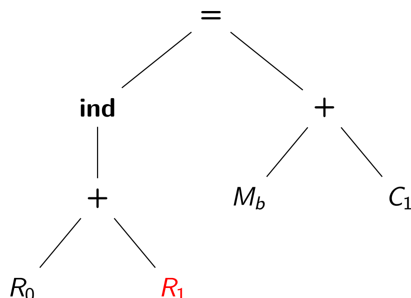
Beispiel Baumübersetzungsverfahren



LD R_0, #a
ADD R_0, R_0, R_{SP}

Anwendung von Regel **5** möglich

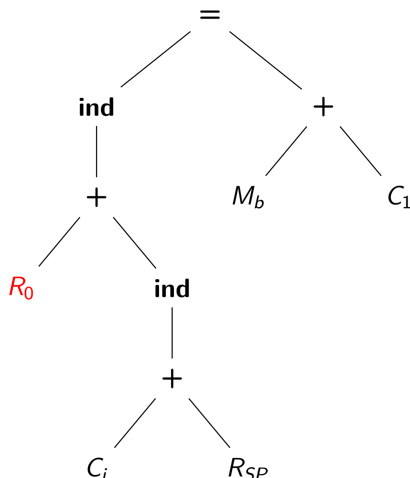
Beispiel Baumübersetzungsverfahren



LD R_0, #a
ADD R_0, R_0, R_{SP}
LD R_1, $i(R_{SP})$

Achtung: Es gibt eine Alternative

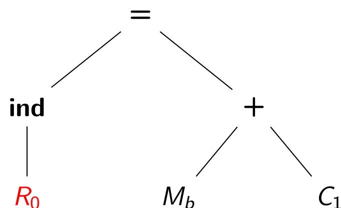
Beispiel Baumübersetzungsverfahren



Anwendung von Regel **6** möglich

```
LD R0, #a
ADD R0, R0, RSP
```

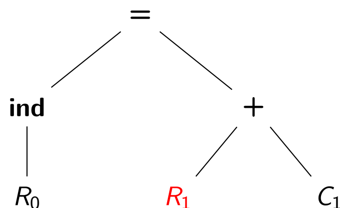
Beispiel Baumübersetzungsverfahren



LD R_0, #a
ADD R_0, R_0, R_{SP}
ADD R_0, R_0, $i(R_{SP})$

Anwendung von Regel **2** möglich

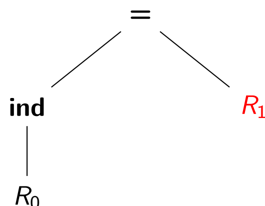
Beispiel Baumübersetzungsverfahren



LD R_0, #a
ADD R_0, R_0, R_{SP}
ADD R_0, R_0, $i(R_{SP})$
LD R_1, b

Anwendung von Regel **8** möglich

Beispiel Baumübersetzungsverfahren



LD R_0, #a
ADD R_0, R_0, R_{SP}
ADD R_0, R_0, $i(R_{SP})$
LD R_1, b
INC R_1

Anwendung von Regel **4** möglich

Beispiel Baumübersetzungsverfahren

M

```
LD R0, #a
ADD R0, R0, RSP
ADD R0, R0, i(RSP)
LD R1, b
INC R1
ST *R0, R1
```

Baumreduktion ist abgeschlossen

Befehlsauswahl als Termersetzung

Voraussetzung: Fasse einen Grundblock als Folge von (Ausdrucks-) Bäumen (Termen) auf. Knoten sind die Tupel

ST a t', LD a, τ 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'.

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 | Operator Ausdruck | Konstante. Operator ::= + | - | * | divmod | ...
- 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.

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 Compilern 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

Exkurs: Baumsprachen, Baumautomaten

Gegeben sei ein Alphabet Σ von Terminalen f mit Stelligkeit $s(f) = k$, $k \geq 0$. Die Menge $B(\Sigma)$ der Bäume über Σ ist induktiv definiert durch

- $a \in B(\Sigma)$, wenn $a \in \Sigma$ und $s(a) = 0$, d.h. $a \in \Sigma_0$
- wenn $b_1, ..., b_k \in B(\Sigma)$ und $f \in \Sigma$, $s(f) = k$ dann $f(b_1, ..., b_k) \in B(\Sigma)$

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

- N ist eine endliche Menge von Nichtterminalen
- $Z \in N$ ist das Zielsymbol
- P ist eine Menge von Produktionen $X \to w, w \in B(\Sigma \cup N), X \in N$
- Der Typ $t(p) = (X_1, ..., X_k)$ einer Produktion $p : X \to w$ 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.

Baumautomaten

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

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

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.

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

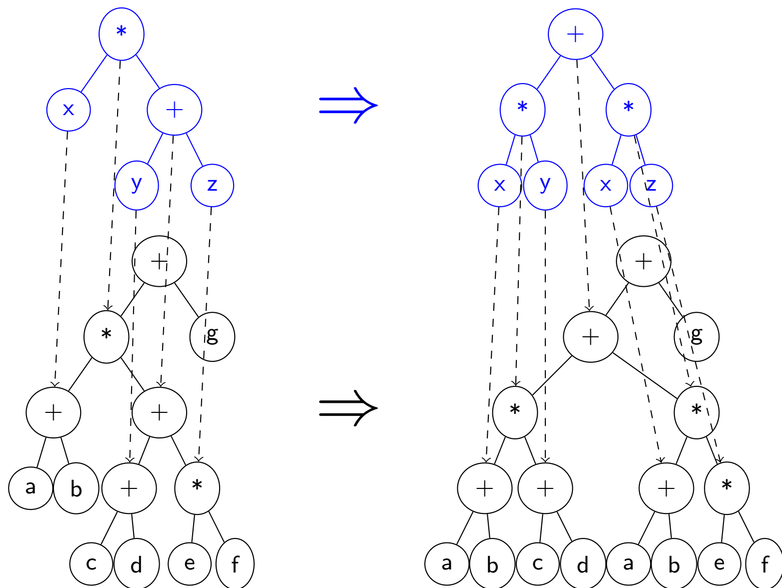
Problem: Termersetzung (mit Variablen) für einen kompletten Baum nicht effizient berechenbar: Ersetzung des Termersetzungssystems (TES) durch ein Grundtermersetzungssystem (GTES, enthält keine Variable), für das es effiziente Verfahren gibt.

Termersetzungssystem TES

- T sei Σ-Termalgebra mit Variablen V und Axiomen Q
- TES: Menge von Termersetzungsregeln $l \to r$, $l, r \in T$ für Termalgebra T
  - l, r können Variable enthalten
  - alle Variablen in l müssen auch in r vorkommen
- $l \to r$ beschreibt Ersetzung eines Unterterms t' von Term t durch s', falls Substitution s existiert mit $t' = ls$ und $s' = rs$.
- $t \Rightarrow s$, 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; $t \to s$ sagt nicht, welche Regel an welcher Stelle benutzt wurde.

Beispiel: Distributivgesetz mittels Termersetzung



Ableitung mit festem Ziel Z

Gegeben:

TES, Zielsymbol Z, Regeln $l \rightarrow r$, Term $t \in T$

Gesucht:

Ableitung $t \Rightarrow^* Z$

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

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 $L(\text{GTES}, Z) \subseteq L(\text{TES}, Z)$.
Ableitung $\Rightarrow^* Z$ ist effizient berechenbar für Grundtermersetzungssysteme.

Gesucht:
Instanz GTES von TES mit $L(\text{GTES}, Z) = L(\text{TES}, Z)$

Termersetzung $\rightarrow$ Grundtermersetzung

Konstruktion eines GTES aus TES:

Prinzip: ersetze Regel $l \rightarrow r$ durch (potentiell unendlich viele) Regeln $l\sigma \rightarrow r\sigma$ für alle benötigten (!) Substitutionen σ
Variable stellen Operanden (Unterbäume) dar, daher praktisch bei Befehlsauswahl nur endlich viele σ, die die Register, Konstanten, Speicherplätze, ... für Operanden substitutieren

Test auf Vollständigkeit $L(\text{GTES}) = L(\text{TES})$ effizient möglich

Konstruktion eines GTES mit $\text{L(GTES)} = \text{L(TES)}$ unentscheidbar, aber berechenbar:
Es gibt Algorithmen, die ein vollständiges GTES aus TES erzeugen, falls es existiert (sonst unendliche Laufzeit).

Ableitung $t \Rightarrow^* Z$ 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 $S \rightarrow K(L, R)$ 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.

Kosten

- Zu jeder Ersetzungsregel gehört Kostenangabe
- Metriken: Laufzeit, Speicher, Energie, ...
- Vorsicht: Cache- und Pipelineeffekte

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

Bottom-up Rewrite System – BURS

Graham und Pelegrini-Llopart ('88)

- Termersetzungssystem statt Grundtermersetzungssystem,
- Kleinere Spezifikation möglich,
- Findet theoretisch alle Abdeckungen
  - exponentiell viele
  - Grenzen für Implementierung
- Anschließende Suche nach globalem Kostenoptimum (NP-hart)
- Angenähert durch Kosten
  - Karlsruhe: A*-Suche in CGGG (Boesler '98)

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 handhabbar

Back-End-Generator – BEG-2

Emmelmann ('94)

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

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.

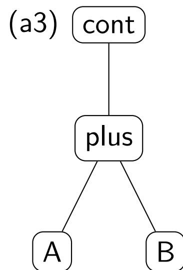
Beispiel – Resultierendes TES

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

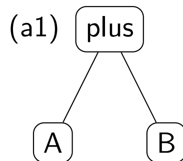
(1)	add (R,Ea)	$\rightarrow$ R
(2)	mov (Ea)	$\rightarrow$ R
(3)	bb	$\rightarrow$ R
(4)	R	$\rightarrow$ Ea
(5)	c	$\rightarrow$ Ea
(6)	di (R,c)	$\rightarrow$ Ea
(a1)	plus (A,B)	$\rightarrow$ add (A,B)
(a2)	A	$\rightarrow$ mov (A)
(a3)	cont (plus(A,B))	$\rightarrow$ di (A,B)
(a4)	plus (A,B)	$\rightarrow$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

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

Zwischensprachterme und Zielprogramm:




```
di A, B
```




```
add A, B
```

Beispiel – Resultierendes GTES

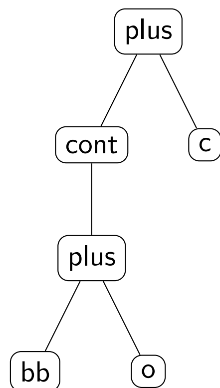
Anmerkung: Offenbar werden alle Variablen mit den Ressourcen der Maschinenbeschreibung instantiiert.
Dieses GTES ist vollständig, aber nicht optimal bzgl. der Kosten.

(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)

Beispiel – Resultierender Baumautomat mit Kostenbewertung

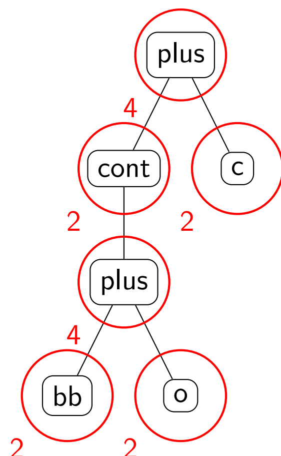
Nr	Regel	Kosten	Aktion
(1)	Ea $\rightarrow$ R	2	R_1 := mov (Ea_1)
(2)	plus (R,Ea) $\rightarrow$ R	4	R_1^2 := add (R_1^1,Ea_1)
(3)	plus (Ea,R) $\rightarrow$ R	4	R_1^2 := add (R_1^1,Ea_1)
(4)	bb () $\rightarrow$ R	0	R_1 := bb ()
(5)	R $\rightarrow$ Ea	0	Ea_1 := R_1
(6)	cont (P) $\rightarrow$ Ea	0	Ea_1 := 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

Beispiel – Zwischensprachterm für $a + c$



a ist eine lokale Variable im Activation Record bb, wobei $o := \text{offset}(bb, a)$
c ist eine Konstante

Beispiel – „dumme“ Überdeckung und Zielprogramm

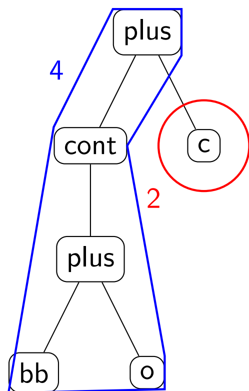


Maschinencode durch Macroexpansion

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

Gesamtkosten: 16

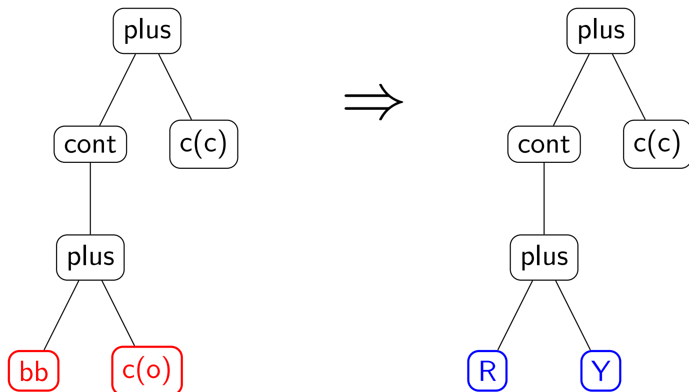
Beispiel – Effizientere Überdeckung und Programm



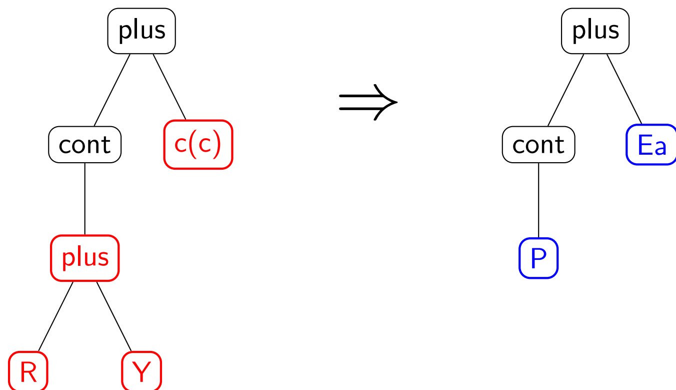

```
R1 := mov c
R1 := add R1, di(bb, o)
```

Gesamtkosten: 6

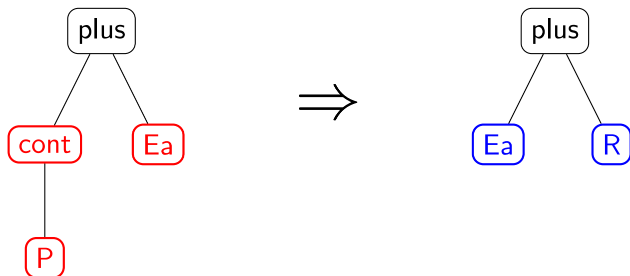
Beispiel – Regel 4 & 8



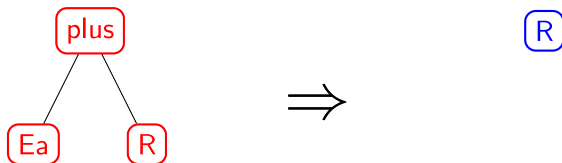
Beispiel - Regel 7 & 9



Beispiel - Regel 1 & 6



Beispiel - Regel 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 ...`

BEG – Zwischensprache

Z

```
INTERMEDIATE_REPRESENTATION

NONTERMINALS
   BArg;

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

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 );
```

BEG – Abdeckungen

Z

```
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} }
```

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;
```