Compilerbau
Vorlesung, Wintersemester 2005

Johannes Waldmann, HTWK Leipzig

2. Februar 2006

Überblick

Was ist ein Compiler?

- Informatik, Algorithmen
- für (abstrakte) Maschinen
- formuliert in Programmiersprache
- verschiedene Sprachen, brauchen Übersetzer
- Interpreter: sofort ausführen
- Compiler: erst übersetzen, dann ausführen

Beispiele: Interpreter: Shells (bash), Script-Sprachen (Perl), hugs
Beispiele: Compiler: gcc, javac, latex, dvips, ghc

Compiler zum Textsatz

Zielsprache: Seitenbeschreibungssprache, z. B. PostScript

```
42 42 scale 7 9 translate .07 setlinewidth .5 s
setgray}def 1 0 0 42 1 0 c 0 1 1{0 3 3 90 270 a
arcn 270 90 c -2 2 4{-6 moveto 0 12 rlineto}for
9 0 rlineto}for stroke 0 0 3 1 1 0 c 180 rotate
       # Eingabe:  compiler.tex
$ latex compiler.tex # erzeugt compiler.dvi
$ xdvi compiler.dvi
       # ansehen
$ dvips compiler.dvi # erzeugt compiler.ps
$ gv compiler.ps

$ latex2html compiler # erzeugt webseiten
```

Empfohlene Literatur/Links

- Webseite zur Vorlesung/Übung, mit Skript, Folien, Aufgaben: `http://www.imn.htwk-leipzig.de/~waldmann/edu/current/compiler/`
- klassisches Lehrbuch: Aho, Sethi, Ullman: Compilers. Principles, Techniques, and Tools. Addison-Wesley 1985.
- Andrew Appel und Jens Palsberg: Modern Compiler Implementation in Java, Cambridge Univ. Press 2002 `http://titles.cambridge.org/catalogue.asp?isbn=052182060X`
- Helmut Seidl: Abstrakte Maschinen (Teile 1 und 2): `http://www.informatik.uni-trier.de/PSI/`

`abstr_masch_ss01.html`

Organisation

- Vorlesung
  - donnerstags, 7:30–9:00, Li110
- Seminare
  - dienstags, 9:30–11:00, Z424
  - *oder* donnerstags, 9:30–11:00, Z424

  Einschreibung über `http://autotool.imn.htwk-leipzig.de/cgi-bin/Super.cgi`, bitte gleichmäßig verteilen . . . wer in kleinerer Gruppe ist, lernt mehr!

Leistungsnachweise

- Prüfungszulassung:
  - kleinere Aufgaben im Seminar
  - und Hausaufgaben (online — autotool)
- Prüfungs-Klausur

(Konkrete) Maschinen

Maschine:

- führt (Folge von) Befehlen aus
- jeder Befehl hat Wirkung: Änderung eines Zustands (der Maschine, der Außenwelt)

Beispiele:

- Geldautomat, Fahrkartenautomat etc.
- CD/DVD-Spieler
- Prozessor, Grafikkarte etc.

abstrakte Maschinen

nicht konkret (materiell), sondern gedacht (virtuell).
gleiches Denkmodell (Befehle ändern Zustand) auf selbst gewählter Abstraktions-Ebene.
Beispiele:

- SQL-Server, HTTP-Server
- PostScript-Interpreter (im Computer, im Drucker)
- die “C–Maschine”
- die “Java-(Bytecode-)Maschine”

Bedeutung abstrakter Maschinen

Definition und Benutzung abstrakter Maschinen
(d. h.: Syntax und Semantik ihrer Sprache)
ist wichtiges Mittel zur Strukturierung von Software.

- Standard-Sprachen und -Maschinen
- DSL (domain specific languages)

Sprachen und Übersetzer

jede Maschine hat ihre Sprache
Maschine liest Befehlsfolge (Programm, Satz der Sprache) und führt aus (*interpretiert*).
Programme für “rein virtuelle” Maschinen (z. B. C, Java) werden in andere Sprache (meist auf niederer Abstraktionsebene) (z. B. Executables, Byte-Code) übersetzt (*kompiliert*)

Arbeitsweise eines Übersetzers

- Eingabe: Quelltext (ist Folge von Zeichen)
- lexikalische Analyse (erzeugt Folge von Tokens)
- syntaktische Analyse (erzeugt Syntaxbaum)
- semantische Analyse (dekoriert Syntaxbaum):
- (bei Interpreter: Code-Ausführung)
- (bei Compiler:)
  - Optimierung (verändert Syntaxbaum)
  - Code-Ausgabe (erzeugt Zielprogramm-Text)

Vorteile eines Compilers

(gegenüber Interpreter)

- trennt Übersetzung und Ausführung
- möglichst großen Teil der Analyse-Arbeit bereits zur Übersetzungszeit ausführen, zur Laufzeit weglassen.
- Übersetzung und Ausführung auf getrennten Maschinen (spart Ressourcen)
- Cross-Compilation ermöglicht Portierungen

strenges Typsystem ⇒ viele Tests zur Compile-Zeit, keine zur Laufzeit ⇒ schnelle Programme!

Compilerbau heute?

- Lernen Sie Ihren Compiler kennen und verstehen und *vertrauen* Sie ihm!

  (er kann viel besser optimieren als Sie)
- *Jedes Programm* verarbeitet einen Eingabestrom, d. h. enthält Elemente eines Übersetzers
- Lernen Sie, DSL zu definieren und zu benutzen (anwendungsspezifische Sprachen)

DSL (domain specific languages)

- klassisch (interpretiert):

  eigene Syntax, Typsystem, Semantik
- eingebettet (und interpretiert) — APIs

  Syntax, Typ- und Modulsystem der Gastsprache, eigene Semantik
- XML-Syntax

Beispiel für DSL

Aufgabe:
welche (DS-)Sprachen bzw. Übersetzer-Programme sind beteiligt, wenn man ein `make` ausführt (bsp. um zu kompilieren oder Dokumentation herzustellen)

Typische Probleme beim Sprach-Entwurf

- viele Gedanken an Syntax

  Wadler's law: der Aufwand beim Entwurf einer Sprache verteilt sich *stark ansteigend* auf:
  4. Semantik, 3. Syntax, 2. Lexik, 1. Lexik von Kommentaren
- Typ- und Modul-System beim Entwurf „vergessen“ (Fortran, Basic, Perl, und wie sie alle heißen :-)
- deswegen “kludges on top of hacks” (z. B. Präprozessor (define/include), Templates?), um dann doch noch ähnliche Effekte zu erreichen

  Nutzung eines Präprozessors für eine Sprache zeigt immer (Design-)Fehler oder Lücken der Sprache selbst.

Wadlers „Gesetz“

(analog Murphys Gesetz)
Phil Wadler, 1996, `http://www.informatik.uni-kiel.de/~mh/curry/listarchive/0017.html`
Twice as much time is spent discussing syntax than semantics, twice as much time is spent discussing lexical syntax than syntax, and twice as much time is spent discussing syntax of comments than lexical syntax.

Compiler kennenlernen

ein simples C-Programm:

```
int f (int x, int y) {
  int i; int s = 0;
  for (i = 0; i < x; i++) {
    s = s + y;
  }
}
```

compilieren mit `gcc -S` ergibt (für Intel)

```
.file "compute.c"
.text
.globl f
.type f, @function
f:
```

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
movl $0, -8(%ebp)
movl $0, -4(%ebp)
.L2:
movl -4(%ebp), %eax
cmpl 8(%ebp), %eax
jl .L5
jmp .L3
.L5:
movl 12(%ebp), %eax
leal -8(%ebp), %edx
addl %eax, (%edx)
```

```
leal -4(%ebp), %eax
incl (%eax)
jmp .L2
.L3:
leave
ret
.size f, .-f
.section .note.GNU-stack,"",@progbits
.ident "GCC: (GNU) 3.3.3 (SuSE Linux)"
```

mit `gcc` für Sparc vergleichen,
mit `cc` vergleichen (Sun-compiler)

Kellermaschinen

PostScript:

```
42 42 scale 7 9 translate .07 setlinewidth .5 s
setgray}def 1 0 0 42 1 0 c 0 1 1{0 3 3 90 270 a
arcn 270 90 c -2 2 4{-6 moveto 0 12 rlineto}for
9 0 rlineto}for stroke 0 0 3 1 1 0 c 180 rotate
```

mit `gv` betrachten
einige Stellen des Programms ändern und Wirkung betrachten

Lexikalische Analyse

Daten-Repräsentation im Compiler

- Jede Compiler-Phase arbeitet auf geeigneter Repräsentation ihre Eingabedaten.
- Die semantischen Operationen benötigen das Programm als Baum

  (das ist auch die Form, die der Programmierer im Kopf hat).
- In den Knoten des Baums stehen Token,
- jedes Token hat einen Typ und einen Inhalt (eine Zeichenkette).

Token-Typen

Token-Typen sind üblicherweise

- reservierte Wörter (if, while, class, ...)
- Bezeichner (foo, bar, ...)
- Literale für ganze Zahlen, Gleitkommazahlen, Strings, Zeichen
- Trennzeichen (Komma, Semikolon)
- Klammern (runde: paren(these)s, eckige: brackets, geschweifte: braces) (jeweils auf und zu)
- Operatoren (`=`, `+`, `&&`, ...)

Reguläre Ausdrücke/Sprachen

Die Menge aller möglichen Werte einer Tokenklasse ist üblicherweise eine reguläre Sprache, und wird (extern) durch eine regulären Ausdruck beschrieben.

Zur Erinnerung: *Chomsky-Hierarchie*

- (Typ 0) aufzählbare Sprachen (beliebige Grammatiken, Turingmaschinen)
- (Typ 1) kontextsensitive Sprachen (monotone Grammatiken, linear beschränkte Automaten)
- (Typ 2) kontextfreie Sprachen (kontextfreie Grammatiken, Kellerautomaten)
- (Typ 3) reguläre Sprachen (rechtslineare Grammatiken, reguläre Ausdrücke, endliche Automaten)

Die Menge $E(\Sigma)$ der *regulären Ausdrücke* über einem Alphabet (Buchstabenmenge) Σ ist die kleinste Menge E, für die gilt:

- für jeden Buchstaben $x \in \Sigma : x \in E$

  (autotool: Ziffern oder Kleinbuchstaben)
- das leere Wort $\epsilon \in E$ (autotool: `Eps`)
- die leere Menge $\emptyset \in E$ (autotool: `Empty`)
- wenn $A, B \in E$, dann
  - (Verkettung) $A \cdot B \in E$ (autotool: `*` oder weglassen)
  - (Vereinigung) $A + B \in E$ (autotool: `+`)
  - (Stern, Hülle) $A^* \in E$ (autotool: `^*`)

Jeder solche Ausdruck beschreibt eine *reguläre Sprache*.

Beispiele/Aufgaben zu regulären Ausdrücken

Wir fixieren das Alphabet $\Sigma = \{a, b\}$.

- alle Wörter, die mit a beginnen und mit b enden: $a\Sigma^*b$.
- alle Wörter, die wenigstens drei a enthalten $\Sigma^*a\Sigma^*a\Sigma^*a\Sigma^*$
- alle Wörter mit gerade vielen a und beliebig vielen b?
- Alle Wörter, die ein aa oder ein bb enthalten: $\Sigma^*(aa \cup bb)\Sigma^*$
- (Wie lautet das Komplement dieser Sprache?)

Endliche Automaten

Intern stellt man reguläre Sprachen lieber effizienter dar:
Ein (nichtdeterministischer) endlicher Automat A ist ein Tupel (Q, S, F, T) mit

- endlicher Menge Q (Zustände)
- Menge $S \subseteq Q$ (Start-Zustände)
- Menge $F \subseteq Q$ (akzeptierende Zustände)
- Übergangs-Relation $T \subseteq (Q \times \Sigma \times Q)$

autotool:

```
NFA { alphabet = mkSet "ab"
    , states = mkSet [ 1, 2, 3]
    , starts = mkSet [ 2]
    , finals = mkSet [ 2]
    , trans = collect
      [ (1, 'a', 2)
      , (2, 'a', 1)
      , (2, 'b', 3)
      , (3, 'b', 2)
      ]
  }
```

Rechnungen und Sprachen von Automaten

Für $(p, c, q) \in T(A)$ schreiben wir auch $p \xrightarrow{c}_A q$.
Für ein Wort $w = c_1c_2 \dots c_n$ und Zustände $p_0, p_1, \dots, p_n$ mit

$$p_0 \xrightarrow{c_1}_A p_1 \xrightarrow{c_2}_A \dots \xrightarrow{c_n}_A p_n$$

schreiben wir $p_0 \xrightarrow{w}_A p_n$.
(es gibt in A einen mit w beschrifteten Pfad von p_0 nach p_n).
Die von A *akzeptierte* Sprache ist

$$L(A) = \{w \mid \exists p_0 \in S, p_n \in F : p_0 \xrightarrow{w}_A p_n\}$$

(die Menge aller Wörter w, für die es in A einen akzeptierenden Pfad von einem Start- zu einem akzeptierenden Zustand gibt)

Anwendung von Automaten in Compilern

Aufgabe: Zerlegung der Eingabe (Strom von *Zeichen*) in Strom von *Token*
Plan:

- definiere Tokenklassen (benutze reguläre Ausdrücke)
- übersetze Ausdrücke in nicht-deterministischen Automaten
- erzeuge dazu äquivalenten deterministischen minimalen Automaten
- simuliere dessen Rechnung auf der Eingabe

Automaten mit Epsilon-Übergängen

Def. Ein ϵ-Automat ... mit $T \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times Q)$.
Definition. $p \xrightarrow{c}_A q$ wie früher, und $p \xrightarrow{\epsilon}_A q$ für $(p, \epsilon, q) \in T$.

Satz. Zu jedem ϵ-Automaten A gibt es einen Automaten B (ohne ϵ-Kanten) mit $L(A) = L(B)$.

Beweis: benutzt ϵ-Hüllen: $H(q) =$ alle $r \in Q$, die von q durch Folgen von ϵ-Übergängen erreichbar sind:

$$H(q) = \{r \in Q \mid q \xrightarrow{\epsilon}{}^*_A r\}$$

Konstruktion: $B = (Q, H(S), A, T')$ mit

$$p \xrightarrow{c}_B r \iff \exists q \in Q : p \xrightarrow{c}_A q \xrightarrow{\epsilon}{}^*_A r$$

Automaten-Synthese

Satz: Zu jedem regulären Ausdruck X gibt es einen ϵ-Automaten A, so daß $L(X) = L(A)$.
Beweis (*Automaten-Synthese*) Wir konstruieren zu jedem X ein A mit:

- $|S(A)| = |F(A)| = 1$
- keine Pfeile führen nach $S(A)$
- von $S(A)$ führen genau ein Buchstaben- oder zwei ϵ-Pfeile weg
- keine Pfeile führen von $F(A)$ weg

Wir bezeichnen solche A mit $s \xrightarrow{X} f$.

Automaten-Synthese (II)

Konstruktion induktiv über den Aufbau von X:

- für $c \in \Sigma \cup \{\epsilon\}$: $p_0 \xrightarrow{c} p_1$
- für $s_X \xrightarrow{X} f_X$, $s_Y \xrightarrow{Y} f_Y e$:
  - $s \xrightarrow{X \cdot Y} f$ durch $s = s_X, f_X = s_Y, f_Y = f$.
  - $s \xrightarrow{X+Y} f$ durch $s \xrightarrow{\epsilon} s_X, s \xrightarrow{\epsilon} s_Y, f_X \xrightarrow{\epsilon} f, f_Y \xrightarrow{\epsilon} f$
  - $s \xrightarrow{X^*} f$ durch $s \xrightarrow{\epsilon} s_X, s \xrightarrow{\epsilon} f, f_X \xrightarrow{\epsilon} s_X, f_X \xrightarrow{\epsilon} f$.

Satz. Korrektheit: $L(A) = L(X)$. Größe: $|Q(A)| \le 2|X|$.

Aufgabe: Warum braucht man bei X^* die zwei neuen Zustände s, f und kann nicht $s = s_X$ oder $f = f_X$ setzen? Hinweise: (wenigstens) eine der Invarianten wird verletzt, und damit eine der anderen Konstuktionen inkorrekt.

Reduzierte Automaten

Ein Zustand q eines Automaten A heißt

- *erreichbar*, falls von q von einem Startzustand aus erreichbar ist: $\exists w \in \Sigma^*, s \in S(A) : s \xrightarrow{w} q$.
- *produktiv*, falls von q aus ein akzeptierender Zustand erreichbar ist: $\exists w \in \Sigma^*, f \in F(A) : q \xrightarrow{w} f$.
- *nützlich*, wenn er erreichbar *und* produktiv ist.

A heißt *reduziert*, wenn alle Zustände nützlich sind.
Satz: Zu jedem Automaten A gibt es einen reduzierten Automaten B mit $L(A) = L(B)$.
Beweis:
erst A auf erreichbare Zustände einschränken, ergibt A', dann A' auf produktive Zustände einschränken, ergibt B.

Deterministische Automaten

- A heißt *vollständig*, wenn es zu jedem (p, c) wenigstens ein q mit $p \xrightarrow{c}_A q$ gibt.
- A heißt *deterministisch*, falls
  - die Start-Menge $S(A)$ genau ein Element enthält und
  - die Relation $T(A)$ sogar eine partielle Funktion ist (d. h. zu jedem (p, c) gibt es höchstens ein q mit $p \xrightarrow{c}_A q$.

  Dann gibt es in A für jedes Wort w höchstens einen mit w beschrifteten Pfad vom Startzustand aus.

Satz: Zu jedem Automaten A gibt es einen deterministischen und vollständigen Automaten D mit $L(A) = L(D)$.

Potenzmengen-Konstruktion

- Eingabe: ein (nicht-det.) Automat $A = (Q, S, F, T)$
- Ausgabe: ein vollst. det. Automat A' mit $L(A') = L(A)$.

Idee: betrachten Mengen von erreichbaren Zuständen $A' = (Q', S', F', T')$ mit

- $Q' = 2^Q$ (Potenzmenge - daher der Name)
- $(p', c, q') \in T' \iff q' = \{q \mid \exists p \in p' : p \xrightarrow{c}_A q\}$
- $S' = \{S\}$
- $F' = \{q' \mid q' \in Q' \land q' \cap F \neq \emptyset\}$

Minimierung von det. Aut. (I)

Idee: Zustände zusammenlegen, die „das gleiche" tun.
Das „gleich" muß man aber passend definieren:
benutze Folge von Äquivalenz-Relationen $\sim_0, \sim_1, \ldots$ auf Q

$p \sim_k q \iff$ Zustände p und q verhalten sich für alle Eingaben der Länge $\le k$ *beobachtbar* gleich:

$$\forall w \in \Sigma^{\le k} : w \in L(A, p) \leftrightarrow w \in L(A, q).$$

äquivalent ist induktive Definition:

- $(p \sim_0 q) :\iff (p \in F \leftrightarrow q \in F)$
- $(p \sim_{k+1} q) :\iff (p \sim_k q) \land \forall c \in \Sigma : T(p, c) \sim_k T(q, c)$.

Minimierung von det. Aut. (II)

Nach Definition ist jeder Relation eine Verfeinerung der Vorgänger: $\sim_0 \supseteq \sim_1 \supseteq \ldots$. Da die Trägermenge Q endlich ist, kann man nur endlich oft verfeinern, und es gibt ein k mit $\sim_k = \sim_{k+1} = \ldots$. Wir setzen $\sim := \sim_k$.
Konstruiere $A' = (Q', S', F', T')$ mit

- $Q' = Q / \sim$ (Äquivalenzklassen)
- $S' = [s]_\sim$ (die Äq.-Klasse des Startzustands)
- $F' = \{[f]_\sim \mid f \in F\}$ (Äq.-Kl. v. akzt. Zust.)
- für alle $(p, c, q) \in T : ([p]_\sim, c, [q]_\sim) \in T'$.

Satz: Wenn A vollständig und deterministisch, dann ist A' ein kleinster vollst. det. Aut mit $L(A') = L(A)$.

Nicht reguläre Sprachen

gibt es reguläre Ausdrücke/endliche Automaten für diese Sprachen?

- Palindrome $P = \{w \mid w \in \{a, b\}^*, w = \mathrm{reverse}(w)\}$
- $E_2 = \{w \mid w \in \{a, b\}^*, |w|_a = |w|_b\}$
- $E_3 = \{w \mid w \in \{a, b, c\}^*, |w|_a = |w|_b = |w|_c\}$
- $K =$ korrekt geklammerte Ausdrücke ($a =$ auf, $b =$ zu)

NEIN! Beweis (Beispiel):
Falls es einen endlichen Automaten mit q Zuständen gibt, der E_2 akzeptiert, dann …

Endliche Automaten als Scanner

Während ein Automat nur akzeptiert (oder ablehnt), soll ein Scanner die Eingabe in Tokens zerteilen.
Gegeben ist zu jedem Tokentyp T_k ein Ausdruck X_k, der genau die Token-Werte zu T_k beschreibt.
Der Eingabestring w soll so in Wörter w_{k_i} zerlegt werden, daß

- $w = w_{k_1} w_{k_1} \ldots w_{k_n}$
- für alle $1 \le i \le n$: w_{k_i} ist *longest match*:
  - $w_{k_i} \in L(X_{k_i})$
  - jedes Anfangsstück von $w_{k_i} \ldots w_{k_n}$, das echt länger als w_{k_i} ist, gehört zu keinem der X_k.

Automaten als Scanner (II)

Man konstruiert aus den X_i Automaten A_i und vereinigt diese, markierte jedoch vorher ihre Endzustände (jeweils mit i). Dann deterministisch und minimal machen.
Beim Lesen der Eingabe zwei Zeiger mitführen: auf Beginn des aktuellen matches, und letzten durchlaufenen akzeptierenden Zustand.
Falls Rechnung nicht fortsetzbar, dann bisher besten match ausgeben, Zeiger entsprechend anpassen, und wiederholen.
Beachte: evtl. muß man ziemlich weit vorausschauen:
Tokens $X_1 = ab, X_2 = ab^*c, X_3 = b$, Eingabe $abcabbbbbbac$.

Komprimierte Automatentabellen

Für det. Aut. braucht man Tabelle (partielle Funktion) $T : (Q \times \Sigma) \hookrightarrow Q$. Die ist aber riesengroß, und die meisten Einträge sind leer. Sie wird deswegen komprimiert gespeichert. Benutze Felder
`next, default, base, check`.
Idee: es gibt viele ähnlichen Zustände:
Zustand p verhält sich wie q, außer bei Zeichen c:
`default[base[p]] = q; check[base[p]+c] = p;`
Übergang $T(p, c) =$ `lookup(p,c)` mit

```
lookup (p, c) {      int a = base[p] + c;
    if  ( p == check[a] ) { return next[a]; }
    else { return lookup (default [p],c); }    }
```

Scanner mit Flex (I)

Das Programm `flex` erzeugt aus einer Scanner-Beschreibung einen Scanner (ein C-Programm).
Wie beschrieben wird aus regulären Ausdrücken X_i ein (markierter) deterministischer Automaten A bestimmt.
Beim Feststellen eines matches kann eine Aktion ausgeführt werden (default: String ausgeben).
Bei mehreren gleichlangen matches wird der (im Quelltext) erste genommen.
Damit der Scanner niemals hängt, gibt es einen Default-Tokentyp, der (zuletzt) jedes einzelne Zeichen matcht (und ausgibt).

Scanner mit Flex (II)

```
DIGIT   [0-9]
%%
DIGIT+  { fprintf (stdout, "%s", yytext); }
" "+    { fprintf (stdout, " "); }
"\n"    { fprintf (stdout, "\n"); }
%%
int yywrap () { return 1; }
int main ( int argc, char ** argv ) { yylex ();
```

Aufruf mit `flex -t simple.l > simple.c`. Optionen:

- `-T` (Table) zeigt Automatentabellen
- `-d` (debug),
- `-f` (fast) ohne Tabellen-Kompression

Übung Scanner-Generator flex

Übung zu flex

- flex-Homepage:
  `http://sourceforge.net/projects/lex/`,
  Dokumentation:
  `http://www.gnu.org/software/flex/manual/`
- ist `flex` installiert? Falls nicht, dann (bash-Syntax)

  `export PATH=$PATH:/home/waldmann/built/bin`
- Beispiele für Übung:
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws03/compilerbau/programme/scanner/`

flex benutzen

Ein Scanner, der Folgen von `ab` erkennen (und zählen) soll (in ein File `scanner.l`)

```
%{
#include <string.h>
#include <stdio.h>
%}

%%

"ab"+  { fprintf (stdout, "%d ",
                  strlen (yytext)); }
.        /* ignore */
```

```
%%

int yywrap () {
        return 1;
}

int main ( int argc, char ** argv ) {
  yylex ();
  fprintf (stderr, "\n");
}
```

Make-Regeln für flex

im Makefile:

```
CC = gcc
LEX = flex

%.c : %.l
        $(LEX) -t $< > $@
```

(die Regel ist tatsächlich schon als Default-Regel eingebaut)

Den flex-Scanner analysieren

Scanner herstellen mit `gmake scanner`
Testen mit `echo 'abaabbababba' | ./scanner`
Konstruieren Sie von Hand einen endlichen Automaten für den Scanner.
Vergleichen Sie mit dem von flex konstruierten.
Benutzen Sie passende debug-Optionen.

Scanner für Java

Schreiben Sie einen flex-Scanner für (eine Teilmenge von) Java.
Wählen Sie dazu passende Tokenklassen und jeweils einen regulären Ausdruck.
Ergänzen Sie `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws03/compilerbau/programme/scanner/java.l`
und testen Sie auf einigen Java-Quelltexten

Syntaktische Analyse (21. 12.)

Wort-Ersetzungs-Systeme

Berechnungs-Modell (Markov-Algorithmen)

- Zustand (Speicherinhalt): Zeichenfolge (Wort)
- Schritt: Ersetzung eines Teilwortes

Regelmenge $R \subseteq \Sigma^* \times \Sigma^*$
Regel-Anwendung:
$u \rightarrow_R v \iff \exists x, z \in \Sigma^*, (l, r) \in R : u = x \cdot l \cdot z \wedge x \cdot r \cdot z = v.$
Beispiel: Bubble-Sort: $\{ba \rightarrow ab, ca \rightarrow ac, cb \rightarrow bc\}$
Beispiel: Potenzieren: $ab \rightarrow bba$
Aufgaben: gibt es unendlich lange Rechnungen für:
$R_1 = \{1000 \rightarrow 0001110\}, R_2 = \{aabb \rightarrow bbbaaa\}$?

Grammatiken

Grammatik G besteht aus:

- Terminal-Alphabet Σ
  (üblich: Kleibuchst., Ziffern)
- Variablen-Alphabet V
  (üblich: Großbuchstaben)
- Startsymbol $S \in V$
- Regelmenge
  $R \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$

```
Grammatik
  { terminale
      = mkSet "abc"
  , variablen
      = mkSet "SA"
  , start = 'S'
  , regeln = mkSet
      [ ("S", "abc")
      , ("ab", "aabbA")
      , ("Ab", "bA")
      , ("Ac", "cc")
      ]
  }
```

von G erzeugte Sprache: $L(G) = \{w \mid S \rightarrow^* w \wedge w \in \Sigma^*\}$.

Eingeschränkte Grammatiken

Für allgemeine Grammatiken ist $w \in L(G)$ (*Wortproblem*) gar nicht entscheidbar.
Regelmenge einschränken $\rightarrow$

- Vorteil: leichter (automatisch) zu behandeln
- Nachteil: weniger ausdrucksstark

Die Chomsky-Hierarchie

- Typ-0 (beliebige Regeln)
  (Wortproblem nicht entscheidbar)
- Typ-1 (keine verkürzenden Regeln): $\forall (l, r) \in R : |l| \leq |r|$.
  ($\Rightarrow$ Wortproblem entscheidbar, jedoch aufwendig)
- Typ-2 (nur kontextfreie Regeln): $\forall (l, r) \in R : l \in V$
  ($\Rightarrow$ Wortproblem in Polynomialzeit)
- Typ-3 (nur rechtslineare Regeln):
  $\forall (l, r) \in R : l \in V \wedge r \in \Sigma V \cup \Sigma$.
  ($\Rightarrow$ Wortproblem in Linearzeit)

Kontextfreie Sprachen

Def (Wdhlg): G ist kontextfrei (Typ-2), falls $\forall (l, r) \in R(G) : l \in V$.
geeignet zur Beschreibung von Sprachen mit hierarchischer Struktur.

```
Anweisung -> Bezeichner = Ausdruck
    | if Ausdruck then Anweisung else Anweisung
Ausdruck -> Bezeichner | Literal
    | Ausdruck Operator Ausdruck
```

Bsp: korrekt geklammerte Ausdrücke:
$G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSbS, S \rightarrow \epsilon\})$.
Bsp: Palindrome:
$G = (\{a, b\}, \{S\}, S, \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \epsilon)$.
Bsp: alle Wörter w über $\Sigma = \{a, b\}$ mit $|w|_a = |w|_b$

Ableitungsbäume für CF-Sprachen

Def: ein geordneter Baum T mit Markierung $m : T \rightarrow \Sigma \cup \{\epsilon\} \cup V$ ist Ableitungsbaum für eine CF-Grammatik G, wenn:

- für jeden inneren Knoten k von T gilt $m(k) \in V$
- für jedes Blatt b von T gilt $m(b) \in \Sigma \cup \{\epsilon\}$
- für die Wurzel w von T gilt $m(w) = S(G)$ (Startsymbol)
- für jeden inneren Knoten k von T mit Kindern $k_1, k_2, \ldots, k_n$ gilt $(m(k), m(k_1)m(k_2)\ldots m(k_n)) \in R(G)$ (d. h. jedes $m(k_i) \in V \cup \Sigma$)
- für jeden inneren Knoten k von T mit einzigem Kind $k_1 = \epsilon$ gilt $(m(k), \epsilon) \in R(G)$.

Ableitungsbäume (II)

Def: der *Rand* eines geordneten, markierten Baumes (T, m) ist die Folge aller Blatt-Markierungen (von links nach rechts).
Beachte: die Blatt-Markierungen sind $\in \{\epsilon\} \cup \Sigma$, d. h. Terminalwörter der Länge 0 oder 1.
Für Blätter: $\mathrm{rand}(b) = m(b)$, für innere Knoten: $\mathrm{rand}(k) = \mathrm{rand}(k_1)\,\mathrm{rand}(k_2)\ldots\mathrm{rand}(k_n)$
Satz: $w \in L(G) \iff$ existiert Ableitungsbaum (T, m) für G mit $\mathrm{rand}(T, m) = w$.

Eindeutigkeit

Def: G heißt *eindeutig*, falls $\forall w \in L(G)$ *genau ein* Ableitungsbaum (T, m) existiert.
Bsp: ist $\{S \to aSb|SS|\epsilon\}$ eindeutig?
(beachte: mehrere Ableitungen $S \to_R^* w$ sind erlaubt, und wg. Kontextfreiheit auch gar nicht zu vermeiden.)

Normalformen von CFG

Erreichbare und produktive Variablen

Für eine CFG G heißt die Variable A

- erreichbar, falls $\exists u, v \in (\Sigma \cup V)^* : S \to_R^* uAv$
- produktiv, falls $\exists w \in \Sigma^* : A \to_R^* w$

Aufgabe: wie kann man entscheiden, ob diese Eigenschaften zutreffen (ohne alle Ableitungen aufzuzählen)?
Vergleiche mit gleichen Begriffen für Zustände von endlichen Automaten.

Reduzierte Grammatiken

Def: Die Grammatik G heißt *reduziert*,
wenn alle Variablen erreichbar und produktiv sind.
Satz: zu jeder CFG G gibt es eine reduzierte CGF G' mit $L(G) = L(G')$.
Beweis: lösche in G zuerst alle nicht produktiven Variablen, dann alle (im Rest) nicht erreichbaren Variablen
(jeweils mit allen Regeln, in denen sie vorkommen)
Aufgabe: warum gerade diese Reihenfolge?

Nullierbare Variablen

Def: Eine Variable A heißt *nullierbar*, falls $A \to_R^* \epsilon$.
Die Menge der nullierbaren Variablen von G
ist die kleinste Menge $N \subseteq V$ mit:

- wenn $(A \to \epsilon) \in R$, dann $A \in N$
- wenn $(A \to x) \in R$ und $x \in N^*$, dann $A \in N$

Bemerkung: der erste Fall ist tatsächlich im zweiten enthalten.
Def: eine Regel $A \to r$ heißt nullierbar, falls $r \to_R^* \epsilon$.

Epsilon-freie Grammatiken

Def: eine CFG G heißt epsilon-frei, falls $\forall (A \to r) \in R : r \neq \epsilon$.
Bemerkung: wenn G epsilon-frei, dann $\epsilon \notin L(G)$.
Satz: Für jede CFG G existiert eine epsilon-freie CGF G' mit $L(G') = L(G) \setminus \{\epsilon\}$.
Beweis: Wende auf alle rechten Regelseiten die Substitution
$A \to$ (wenn A nullierbar, dann $\{A, \epsilon\}$, sonst $\{A\}$)
an, und lösche dann alle Epsilon-Regeln.
Aufgabe: Konstruiere G' für G mit $R = \{S \to \epsilon \mid aSb \mid SS\}$.

Kettenregeln

Eine Regel $(l \to r)$ mit $|r| = 1$ heißt *Kettenregel*.
Der Ketten-Abschluß von G ist die kleinste Menge R' mit

- $R \subseteq R'$
- falls $(A \to B) \in R$ und $(B \to r) \in R'$, dann $(A \to r) \in R'$.

Aufgaben: Wieviele Regeln kann man maximal hinzufügen?
Satz: Zu jeder CFG G existiert eine CFG G' ohne Kettenregeln mit $L(G) = L(G')$.
Beweis: $G' = (\Sigma, V, S, R'$ ohne Kettenregeln).
Aufgabe: falls R epsilon-frei, dann auch R'.

Kreise und kreisfreie Grammatiken

Def: eine *Kreis-Ableitung* ist von der Form $A \to_R^+ A$ für eine Variable A.
Satz: für jede CFG G gibt es eine kreisfreie CFG G' mit $L(G) = L(G')$.
Idee: G epsilon-frei: G_1, kettenfrei: G_2, ist kreisfrei.
Aufgaben:

- Anwenden auf $S \to \epsilon \mid SS \mid aSb$
- Behauptung beweisen, dabei beachten:
- wie wird $\epsilon \in L(G)$ behandelt?
- Wie kann man entscheiden, ob gegebenes G kreisfrei ist (ohne alle Ableitungen aufzuzählen)?

Chomsky-Normalform

Def: CFG G ist in Chomsky-Normal-Form, falls $\forall(l \to r) \in R : r \in \Sigma \cup V^2$.
Satz: Zu jeder CFG G gibt es eine CFG G' in Chomsky-NF mit $L(G) \setminus \{\epsilon\} = L(G')$.
Beweis: benutze Hilfsvariablen.
Aufgabe: Wer ist Noam Chomsky? (google)

Greibach-Normalform

Def: CFG G ist in Greibach-Normal-Form, falls $\forall(l \to r) \in R : r \in \Sigma V^*$.
Satz: Zu jeder CFG G gibt es eine CFG G' in Greibach-NF mit $L(G) \setminus \{\epsilon\} = L(G')$.
Beweis ist schwer. Aber einzelne Beispiele gehen (mühsam) von Hand.
Aufgabe: finde Greibach-Nf von:

- $S \to TS \mid b, T \to ST \mid a$
- $S \to b \mid TT, T \to a \mid SS$
- der Vorname von Greibach?

Aufgaben zu Grammatiken

Sprachen (und ihre Komplemente `Com*`):

- `Pali`: $\{w \mid w = \text{reverse}(w)\}$
- `Gleich`: $\{w \mid d(w) = 0\}$, wobei $d(x) := |w|_a - |w|_b$
- `Dyck`-Sprache (korrekt geklammerte Wörter): $\{w \mid d(w) = 0 \land \forall u \sqsubseteq w : d(u) \geq 0\}$,

Aufgaben: `G`: finde CF-Grammatik, `Ein`: eindeutige CF-Grammatik

- empfohlen: `Ein-Pali, G-ComPali, Ein-Dyck, G-ComGleich`
- Highscores: `Ein-(Com)Gleich, {G,Ein}-Com{Pali,Dyck}`

Dangling else

In vielen Programmiersprachen ist definiert:

```
Anweisung -> ...
    |  if Ausdruck then Anweisung
    |  if Ausdruck then Anweisung else Anweisung
```

Modell: $\{S \to e|tS|tSeS\}$.
Diese Regelmenge führt zu einer mehrdeutigen Grammatik.
Aufgabe: finden Sie eine eindeutige Grammatik mit den „richtigen“ Ableitungsbäumen.

Arithmetische Ausdrücke

Arithmetische Ausdrücke kann man so definieren:

```
Ausdruck -> Zahl
    | Ausdruck + Ausdruck
    | Ausdruck - Ausdruck
    | Ausdruck * Ausdruck
    | Ausdruck / Ausdruck
```

Das ist mehrdeutig.
Aufgabe: machen Sie das so eindeutig, daß die aus der Grundschule bekannten Ableitungsbäume entstehen.

Implementierungen von Parsern

Überblick

Art der Implementierung:

- von oben (Startsymbol der Grammatik) nach unten
- von unten (Terminale der Grammatik) nach oben

Art der Herstellung:

- manuell: Parser programmieren
- automatisch: aus Grammatik generieren

Rekursiver Abstieg

Einfachste Methode, einen Parser von Hand zu schreiben: zu jeder Variablen A eine Prozedur, die ein Wort aus $L(G, A)$ liest (d. h. den entsprechenden Teil der Eingabe verbraucht) und den Syntaxbaum zurückgibt

```
parse_Anweisung =   case lookahead () of
      "while" -> parse_Ausdruck ; parse_Block
      "if"    -> parse_Ausdruck ; parse_Block
      default -> parse_Zuweisung
parse_Zuweisung = n <- parse_Name ; parse "="
      x <- parse_Ausdruck ; parse ";"
parse_Block = parse "{" ; parse_Folge ; parse "
parse_Folge = leer
        oder  parse_Anweisung ; parse_Folge
```

Rekursiver Abstieg/gnat/Übung

benutzt (z. B.) in `gnat` (Gnu Ada Translator, Teil von `gcc`), `http://www.gnat.com`, `ftp://cs.nyu.edu/pub/gnat`

Aufgaben:

- Schreiben Sie ein C- oder Java-Programm, das das “dangling-else”-Problem illustriert. Komplieren Sie.
- Wie ist das “dangling else”-Problem in Ada gelöst? Suchen Sie in der Sprach-Definition: `http://www.adahome.com/rm95/`
- Vergleichen Sie mit dem Parser-Quelltext: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws03/compilerbau/programme/gnat-3.15p-src/src/ada/`

Rekursiver Abstieg (III)

Vorteile:

- gibt Struktur gut wieder,
- aussagefähige Fehlermeldungen möglich

Nachteile/Einschränkungen:

- Look-Ahead,
- Links-Rekursionen

Links-Faktorisierung

Durch Hilfsvariablen Entscheidungen verschieben, Beispiel if/then — if/then/else

Links-Rekursion

```
Ausdruck = Name oder Literal
   oder Zeichen "(" ; Ausdruck ; Zeichen ")"
   oder Ausdruck Operator Ausdruck
```

Def: eine Variable A in CFG G heißt links-rekursiv, falls $\exists w \in (V \cup \Sigma)^* : A \to_R^+ Aw$.

Aufgabe: Wie kann man entscheiden, ob gegebenes G links-rekursiv ist?

Links-Rekursion (II)

Satz: Zu jeder CFG G gibt es CFG G' ohne Links-Rekursion mit $L(G) = L(G')$.

Beweis (Idee): ersetze $\{A \to Aw, A \to r\}$ durch $\{A \to rB, B \to \epsilon, B \to wB\}$.

Aufgaben (evtl. autotool): entferne Links-Rekursionen aus

- $S \to Aa \mid b, A \to Ac\,Sd \mid \epsilon$.
- $S \to TS \mid b, T \to ST \mid a$
- $S \to b \mid TT, T \to a \mid SS$

Keller-Automaten

Keller-Automaten

im Prinzip wie endlicher Automat, aber als Arbeitsspeicher nicht nur Zustand, sondern auch Keller (Band).

```
data Konfiguration x y z =
     Konfiguration { eingabe :: [ x ]
                   , zustand :: z
                   , keller  :: [ y ]
                   }
```

Ein Arbeitsschritt ist abhängig vom obersten Kellersymbol y und Zustand z und besteht aus:

- Zeichen x lesen oder ϵ lesen
- neuen Zustand annehmen und
- oberstes Kellersymbol ersetzen

Keller-Automaten (II)

```
data NPDA x y z =
     NPDA { eingabealphabet  :: Set x
          , kelleralphabet   :: Set y
          , zustandsmenge    :: Set z
          , startzustand     :: z
          , startsymbol      :: y
          , akzeptiert       :: Modus z
          , tafel            ::
            Relation (Maybe x, z, y) (z, [y])
          }
```

Übergangsrelation $(w, z, k) \to_A (w', z', u'k')$, falls

- $w = xw'$ für $x \in \Sigma$ und $k = yk'$ und $(z', u') \in T(x, z, y)$
- *oder* $w = w'$ und $k = yk'$ und $(z', u') \in T(\epsilon, z, y)$

Beispiel autotool/Kellerautomaten

Aufgaben

`Acceptor-NPDA(det)-{OGleich,Gleich,Pali,Dyck}`

Beispiel:

```
NPDA { eingabealphabet = mkSet "ab"
     , kelleralphabet = mkSet "XA"
     , zustandsmenge = mkSet [ 0, 1, 2]
     , startzustand = 0 , startsymbol = 'X'
     , akzeptiert = Leerer_Keller
     -- ODER: , akzeptiert = Zustand ( mkSet [
     , tafel = collect
             [ ( Just 'a', 0 , 'X', 0, "AX")
             , ( Just 'a', 0 , 'A', 0, "AA")
             , ( Just 'b', 0 , 'A', 2, ""  )
             , ( Just 'b', 2 , 'A', 2, ""  )
             ]
     }
```

Sprachen von Keller-Automaten

- Die durch *leeren Keller* akzeptierte Sprache:

  $$L_K(A) = \{w \mid \exists z : (w, z_0, [y_0]) \to^* (\epsilon, z, \epsilon)\}$$

- Die durch *Endzustandsmenge* F akzeptierte Sprache:

  $$L_F(A) = \{w \mid \exists z \in F, k \in Y^* : (w, z_0, [y_0]) \to^* (\epsilon, z, k)\}$$

Beachte in beiden Fällen: ϵ-Übergänge am Wortende sind noch möglich.

Keller-Automaten-Sprachen und CFG

Satz: Für alle Sprachen L gilt: $\exists$ CFG G mit $L(G) = L \iff \exists$ Kellerautomat A mit $L(A) = L$.

Beweis ($\Rightarrow$)

- nur ein Zustand z_0
- Variablenmenge $=$ Kelleralphabet
- Startsymbol $=$ Startsymbol (im Keller)
- Regel $B \to w =$
  ϵ-Übergang $(w, z_0, Bk') \to (w, z_0, wk')$
- jedes Terminalzeichen $x \in \Sigma$:
  Übergang $(xw', z_0, xk') \to (w', z_0, k')$.

Rechts/Links-Ableitungen

Def: eine Ableitung $w_0 \to_G w_1 \to \ldots$ heißt *Rechts- (bzw. Links-)Ableitung*, falls in jedem Schritt die am weitesten rechts (bzw. links) stehende Variable ersetzt wird.

Beispiel: $G = (\{a, b\}, \{S\}, S, \{S \to b, S \to aSS\})$

Linksableitung: $S \to aSS \to aaSSS \to aabSS \to aabbS \to aabbaSS \to aabbabS \to aabbabb$,

Rechtsableitung: $S \to aSS \to aSaSS \to aSaSb \to aSabb \to aaSSabb \to aaSbabb \to aabbabb$.

Zu jedem Ableitungsbaum gehören genau eine Rechts- und eine Links-Ableitung.

(D. h.: Grammatik G eindeutig
$\iff$ jedes $w \in l(G)$ besitzt genau eine Rechts-Ableitung
$\iff$ jedes $w \in l(G)$ besitzt genau eine Links-Ableitung.)

Rechts/Links-Ableitungen und Parser

Ein Top-Down-Parser sucht von links eine Links-Ableitung, ein Bottom-Up-Parser sucht von links eine (umgekehrte) Rechts-Ableitung

Beispiel: $G = (\{a, b\}, \{S\}, S, \{S \to b, S \to aSS\})$

Rechtsableitung: $S \to aSS \to aSaSS \to aSaSb \to aSabb \to aaSSabb \to aaSbabb \to aabbabb$.

Kellerautomat (shift/reduce), Zustand: (Keller, Eingabe)

$(\epsilon, aabbabb) \to_S (a, abbabb) \to_S (aa, bbabb) \to_S (aab, babb) \to_R (aaS, babb) \to_S (aaSb, abb) \to_R (aaSS, abb) \to_R (aS, abb) \to_S (aSa, bb) \to_S (aSab, b) \to_R (aSaS, b) \to_S (aSaSb, \epsilon) \to_R (aSaSS, \epsilon) \to_R (aSS, \epsilon) \to (S, \epsilon)$

Shift und Reduce

LR-Parser: deterministischer endlicher Automat mit Keller, konstruiert *von links* eine *Rechts*-Ableitung.

Kellerinhalt: $X_1 X_2 \ldots X_m$ (jedes X_i ist Terminal oder Variable)

Konfiguration: (Kellerinhalt, Rest der Eingabe $a_i a_{i+1} \ldots a_n$) repräsentiert (rechts-)abgeleitete Satzform

$$X_1 X_2 \ldots X_m \; a_i a_{i+1} \ldots a_n$$

Parser-Aktionen sind:

- shift: zu $(\ldots X_m a_i, a_{i+1} \ldots)$
- reduce: falls Regel $(A \to X_{m-r+1} \ldots X_m) \in G$:
  zu $(\ldots X_{m-r} A, a_i a_{i+1} \ldots)$, wobei $r = |w|$

Top-Down/Bottom-Up und Eindeutigkeit

Für effizientes Parsen möchte man kein Backtracking, also *Eindeutigkeit* (der Auswahl der anzuwendenden Regel).
Das ist bei Top-Down-Parsern eine starke Einschränkung, aber bei Bottom-Up-Parsern nicht so gravierend:
diese können Entscheidungen „in die Zukunft“verschieben, indem Zwischenergebnisse auf dem Stack gespeichert werden.

Vorlesung 24. 11.

Dangling Else

Grammatik mit diesen Regeln

`S -> if ( E ) S , S -> if ( E ) S else S`

ist mehrdeutig, Beispiel:

`if ( true ) if ( false ) foo (); else bar ();`

- Problem besteht in Algol, C, Pascal
- Lösung in Ada: `if .. end if`
- Lösung in Java siehe Grammatik
  `http://java.sun.com/docs/books/jls/`

  zur Java-Grammatik vergleiche auch:
  `http://www.ergnosis.com/java-spec-report/java-language/jls-18.html`

Operatoren

Grammatik mit Regeln

`E -> Konstante | Variable | E Op E`

`Op -> + | - | * | /`

ist mehrdeutig, Beispiel `1+2*3`

Lösungsmöglichkeiten:

- Zusatzvereinbarungen (Operator-Präzedenzen)
- eindeutige Grammatik (mit mehr Variablen)

Aufgabe:

- beachte Assoziativität
- neuer Operator `^` mit Assoziativität `2^3^2=2^(3^2)` (warum?)

Syntaxbäume

- konkreter Syntaxbaum (CST) = Ableitungsbaum
- abstrakter Syntaxbaum (AST)

  enthält nur die wesentlichen Knoten des CST

  (unwesentliche entstehen z. B. durch Grammatik-Transformationen)

Syntaktische Analyse mit SableCC

SableCC

(Etienne Gagnon, ab 1998) `http://sablecc.org/`

- Eingabe: Token-Definitionen (reg. Ausdr.), (kontextfr.) Grammatik
- Ausgabe: Java-Klassen für
  - Lexer (komprimierter minimaler det. Automat)
  - Parser (deterministischer Bottom-up-Kellerautomat)
  - (konkreten und abstrakten) Syntaxbaum
  - Visitor-Adapter (tree walker)

Eine SableCC-Eingabe

(vereinfacht)

```
Tokens   whitespace = (' ' | '\t' | 10 | 13)+;
   number = ['0' .. '9']+;
   plus = 'plus';
   lpar = '('; rpar = ')'; comma = ',';
Ignored Tokens  whitespace;
Productions
   expression = number
      | plus lpar
        expression comma expression
        rpar;
```

Annotierte Grammatiken

```
Productions
   expression = { atomic } number
      | { compound } plus lpar [left]:expression
           comma [right]:expression rpar;
```

SableCC generiert Klassen

```
abstract class Node; -- einmal
-- für jede Regel:
abstract class PExpression extends Node;
-- für jede Alternative einer Regel:
final class ACompoundExpression extends PExpres
   -- für jede Variable in rechter Regelseite:
   PExpression getLeft ();
}
```

Durchlaufen von Syntaxbäumen

```
class Eval extends DepthFirstAdapter {
  public void outACompoundExpression
        (ACompoundExpression node) {
    System.out.println (node);
    Integer l =
       (Integer) getOut (node.getLeft());
    Integer r =
       (Integer) getOut (node.getRight());
    setOut (node, l + r);
  }
  public void outAAtomicExpression
        (AAtomicExpression node) { .. }
}
```

Aufruf eines SableCC-Parsers

```
class Interpreter {
  public static void main (String [] argv) {
    PushbackReader r =
        new PushbackReader
           (new InputStreamReader (System.in));
    Parser p = new Parser (new Lexer (r));
    Start tree = p.parse ();
    AnalysisAdapter eval = new Eval ();
    tree.apply (eval);
  }
}
```

Attribut-Grammatiken

= kontextfreie Grammatik + Regeln zur Berechnung von Attributen von Knoten im Syntaxbaum

- berechnete (synthetisierte) Attribute:

  Wert des Att. im Knoten kann aus Wert der Att. der Kinder bestimmt werden

  komplette Berechnung für alle Knoten im Baum von unten nach oben (bottom-up, depth-first)

- ererbte (inhärierte) Attribute

  Wert des Att. im Knoten kann aus Wert der Att. im Vorgänger bestimmt werden

  Berechnung von oben nach unten (top-down)

Durch Kombination (mehrere Durchläufe) können auch andere Abhängigkeiten behandelt werden.

CST zu AST

AST-Typ deklarieren (wie Grammatik)

```
Abstract Syntax Tree
    exp = { plus }   [left]:exp [right]:exp
       | { times }   [left]:exp [right]:exp
       | { number } number  ;
```

und Übersetzungen für jeder Regel

```
Productions
  expression { -> exp } =  sum { -> sum.exp } ;
  sum { -> exp }
     = { simple } product { -> product.exp }
     | { complex } sum plus product
  { -> New exp.plus (sum.exp, product.exp) } ;
```

links Typ, rechts Kopie oder Konstruktion (new)

Das ist Attributgrammatik (jeder Knoten des CST bekommt als Wert eine Knoten des AST)

Übung SableCC

- SableCC in `/home/waldmann/built/sablecc-3.1`
- Beispiele in `http://www.imn.htwk-leipzig.de/~waldmann/edu/current/compiler/programme/rechner/`
- Quelltexte generieren

  `sablecc rechner.grammar`

  Welche Dateien wurden erzeugt? Wo stehen der endliche Automat, der Kellerautomat?

```
javac Interpreter.java # kompilieren
echo "1 + 3 + 5" | java Interpreter # testen
```

Aufgaben: erweitern:

- `Integer` durch `BigInteger` ersetzen
- Subtraktion: `4 - 2 + 1`
- geklammerte Ausdrücke: `1 + (2 + 3)`
- Potenzen: `2^3^2`
- Funktion Fakultät `fac(6)`

Aufgaben:

- lokale Konstanten (Werte deklarieren, Werte benutzen): `let { x = 3 + 5 ; y = 2 * x } in x + y`
- Zuordnung Name → Wert durch `Map<String,Integer>` aus package `java.util`

- was fehlt noch zu Programmiersprache?

Quiz-Aufgabe

(hat nichts ursächlich mit Compilerbau zu tun)

- 50 Gefangene, 1 Raum mit Schalter und Lampe
- Wärter führt Gefangene einzeln in den Raum in beliebiger Reihenfolge und jeden beliebig oft
- gesucht ist Verfahren, nach dem wenigstens einer der Gefangenen feststellen kann, daß jeder wenigstens einmal im Raum war
- Verfahren kann vorher festgelegt werden, aber während es läuft, gibt es keinen Informationsaustausch (außer über Lampe an/aus)

zitiert nach Bulletin EATCS

Semantik

Interpretation

Interpretation = Wert eines Ausdrucks (Programms) (sofort) ausrechnen.
Oft geht das durch Auswertung von unten (Blätter des AST) nach oben (Wurzel)...
d. h. der Wert ist ein berechnetes (sythetisiertes) Attribut.

```
class Eval extends DepthFirstAdapter {
  public void outAPlusExp (APlusExp node) {
    Integer l = (Integer) getOut (node.getLeft(
    Integer r = (Integer) getOut (node.getRight
    Integer s = l + r;
    setOut (node, s);
```

```
} }
```

Funktionen

Benutzen konkrete Syntax

```
expression
  = number
  | identifier lpar arguments rpar ;
arguments
  =  /* leer */  | expression cexp* ;
cexp = comma exp ;
```

abstrakte Syntax

```
exp = { apply } identifier exp*
    | { number } number         ;
```

und Tabelle von vordefinierten Funktionen.

Randbemerkung: Anonyme Klassen in Java

Eine (vordefinierte) Funktion hat Namens- und Typinformation (später) und Auswertungsvorschrift das ist eigentlich ein Unterprogramm, muß aber als Methode in einem Objekt versteckt werden:

```
class Function {
    ...
    private Code code;
}
interface Code {
    Object evaluate (List<Object> argv);
}
```

man braucht eine Klasse, die Code implementiert, aber von dieser nur ein *einziges* Objekt. Deswegen Klasse *lokal* und *anonym*:

```
static Function add() {
  return new Function(
    ... ,
    new Code() {
      public Object evaluate(List<Object> argv)
          return iGet(argv,0).add(iGet(argv,1));
      }
    });
}
static BigInteger iGet (List<Object> argv, int
  return (BigInteger) argv.get(i);
```

```
}
```

Typen

- außer arithmetischen Ausdrücken auch Boolesche: `3+4 == 1+6` bzw. `eq(add(3,4),add(1,6))`
- Wert-Typ muß Zahlen und Wahrheitswerte umfassen.
- jede Funktion (add, equal) hat *Typ*: Anzahl und Typen der Argumente, Typ des Resultats

```
Bool equal (Number x, Number y);
Number add (Number x, Number y);
```

Typ-Prüfung als abstrakte Interpretation

Jedem Teilausdruck wird *vor der Auswertung* sein Typ zugeordnet. Das ist auch ein synthetisches Attribut!

```
enum Type { Number, Bool }
class Typecheck extends DepthFirstAdapter {
  public void outANumberExp(ANumberExp node) {
    setOut (node, Type.Number);
  }
  public void outAApplyExp(AApplyExp node) {
    String name = node.getIdentifier().getText(
    Function fun = defined.get(name);
    // vergleiche Argumenttypen
    setOut(node, fun.getResult());
  }
}
```

Aufgabe zu Interpretation/Typprüfung

- Autotool: typeQ

Zu Beispiel aus Vorlesung, Quelltexte:
`http://www.imn.htwk-leipzig.de/~waldmann/edu/current/compiler/programme/types/`

- Funktionen hinzufügen (in welcher Datei wird geändert? Nur eine!)
  - `mul` (Multiplikation, zweistellig)
  - `quad` (Quadrieren, einstellig)
  - `not` (Negation, einstellig)
  - `and` (Konjunktion, zweistellig)
- boolesche Konstanten (`True, False`) hinzufügen

  (beachte große Anfangsbuchstaben)

  (in welchen Dateien wird geändert?)
- Verzweigungen hinzufügen: `if (Ex1, Ex2, Ex3)`

  Type von Ex1 ist Bool, Typ von Ex2 und Ex3 ist Number, das ist auch Resultattyp

Bemerkung zu Auswertungsstrategien

- sollte man `if` als dreistellige Funktion schreiben?
- beachte `if (i>=0) then a[i] else 42`
- Antwort: bei unserer Auswertungsstrategie (alles von unten nach oben) lieber nicht.
- es gibt andere Strategien, die dafür besser passen: *lazy evaluation* wertet von oben nach unten nur die *benötigten* Teilausdrücke aus.
- damit kann man auch sehr schön mit unendlichen Datenstrukturen rechnen (solange man sich nur endliche Teile davon anschaut).

Lazy Evaluation (Beispiel)

Sieb des Eratosthenes: berechnet die unendliche Liste aller Primzahlen, durch wiederholtes Filtern aus der unendlichen Liste `[2..]`

```
primes :: [ Integer ]
primes = sieve [ 2 .. ]
sieve :: [ Integer ] -> [ Integer ]
sieve (x : ys) =
  x : sieve
      ( filter ( \ y -> 0 < y `mod` x ) ys )
```

lazy evaluation ist softwaretechnisch sinnvoll, denn damit kann man *Erzeugung* von *Verarbeitung* von Daten trennen. Vgl. XML-Parser, die Teildokumente erst bei Bedarf lesen/verarbeiten.

Typen

Typen

Ein Typ ist eine Menge von Werten, Beispiele:

- `int` kleine ganze Zahlen, `boolean` Wahrheitswerte
- `char` Zeichen, `String` Zeichenkette

erst durch Angabe eines *Typs* kann einem *Bitmuster* im Hauptspeicher ein *Sinn* zugeordnet werden

wenn sich die Zuordnung (Speicherstelle → Typ)
beim Lauf des Programms (dynamisch) nicht ändert,
dann kann sie vollständig zur Übersetzungszeit (statisch) bestimmt werden

Nutzen der statischen Typprüfung

- höhere und frühere Sicherheit:

  Programmierfehler bereits zur Übersetzungs(Entwicklungs)-Zeit erkennen und nicht erst (eventuell) zur Laufzeit

- Geschwindigkeit:

  wenn Typkorrektheit zur Übersetzungszeit feststeht, dann kann man zur Laufzeit alle Typprüfungen weglassen (das spart Platz und Zeit).

wenn man Zahlen und Wahrheitswerte verwechselt

. . . und Anweisungen als Ausdrücke gestattet
(Übung: in C probieren, in Java probieren)

```
int x = foobar ();
if (x = 0) { .. };
```

(deswegen bei Vergleich mit Konstanten *immer* die Konstante zuerst schreiben!)

Trick Question

Was ist hier falsch? (Algol68)

```
int sum = 3;
sum = sum + 4;
print (sum);
```

ist kompilierbar und gibt 0 aus. Warum?

Algol68: „eine deutliche Verbesserung gegenüber vielen ihrer Nachfolger“(E. W. Dijkstra)

wenn man nur einen Typ (int) kennt

```
hanoi (a, b, c) {
    if (0 < c) {
        hanoi (a, 6 - a - b, c - 1);
        print (a, b, c);
        hanoi (6 - a - b, b, c - 1);
    }
}
```

vgl. *code smell*: *primitive obsession*

Passend gewählte Datenstruktur und „dummes“ Programm ist viel viel besser als andersherum!

Typ-Prüfung bei Funktions-Aufrufen

- (einstellig)

  wenn $f :: A_1 \to B$ und $X_1 :: A_1$,

  dann $f(X_1) :: B$

- (zweistellig)

  wenn $f :: A_1 \times A_2 \to B$ und $X_1 :: A_1$ und $X_2 :: A_2$,

  dann $f(X_1, X_2) :: B$

- (entspr. für andere Stelligkeiten)

Analogie zur Aussagenlogik:
wenn $A_1 \to B$ und A_1 wahr ist, *dann* ist B wahr.
(Curry-Howard-Isomorphie, Calculus of Constructions)

Statische Typen sind streng

jede Funktion hat genau einen Resultattyp. Also nicht:

```
?? foo (int x) {
   if (12 < x) { return "umpf" ; }
   else        { return 2 * x  ; }
}
void bar () { int y = foo (9); }
```

Es geht aber doch: in Sprachen mit . . .

- ausschließlich dynamischer Typprüfung
- genauerer statischer Typprüfung (dependent types),

Methoden-Aufrufe

nicht-statische Methode

```
class C { Foo m (Bar b); }

C c; ... c.m (y) ...
```

für Typ-Prüfung behandeln normales Unterprogramm (statische Methode) mit zusätzlichem Argument:

```
static Foo m (C this, Bar b);

C c; ... m (c, y) ...
```

Überladen von Bezeichnern

(nicht verwechseln mit *Überschreiben*!)
Beispiel: `eq(true, true),eq(3,add(1,2))`
`eq` ist ein Name für zwei verschiedene Funktionen (mit verschiedenen Typen).
Sprechweise: der Name ist *überladen*, es gibt *ad-hoc-Polymorphie*.
Wie muß dazu die Tabelle der (vordefinierten) Funktionen aussehen?
In Java ist Überladung nur anhand der Argument-Typen, aber nicht des Resultat-Typs gestattet—warum?

Überladen/Überschreiben

zur Erinnerung:

- Überladen: Methoden in gleicher Klasse mit unterschiedlichen Typen
- Überschreiben: Methoden in unterschiedlichen Klassen (abgeleitete Klasse und Basisklasse/Schnittstelle) mit übereinstimmenden Typen

Generische Polymorphie

- bis jetzt nur *einfache* Typen (Number, Bool).
- nützlich sind auch *zusammengesetzte* Typen,
- diese entstehen durch Anwendung von Typkonstruktoren auf Typ-Argument(e),
- Beispiel: Liste von Bool, Abbildung (Map) von Number nach Number.
- dann gibt es *generisch polymorphe* Funktionen, Beispiel: Länge einer Liste (mit beliebigem Elementtyp).

  vgl. Java:
  ```
  interface Collection<T> { int size (); }
  ```

Typprüfung für generische Funktionen

wie bisher: Argumenttypen müssen passen.
aber jetzt: nicht einfacher Vergleich mit festem Typ, sondern Anpassung an Typschema, das Typvariablen enthält.
bei Anpassung werden Typvariablen *gebunden*. mit dieser Bindung (Zuordung Typvariable $\rightarrow$ Typ) kann Resultattyp berechnet werden.

```
interface List<T> { T get (int i); }

List<List<Integer>> l;
??? l.get(2);

   Bindung T => List<Integer>
```

Aufgabe zu generischen Typen

Vervollständigen Sie die Deklarationen, so daß das Programm typkorrekt wird:

```
interface F<A,B> {

        ...    m     ... ;
        H<B> q ( H<A> x );
}
interface G<A> { H<A> s(); }

interface    ...     { A r(); }

static String check (F<H<Integer>,   ...   > y)
        return y.q(y.m().s()).r();
}
```

Eingeschränkte generische Polymorphie

die Werte der Typvariablen (= Argumente für Typschablonen) müssen gewisse Bedingungen erfüllen (gewisse Schnittstellen implementieren):

```
interface Comparable<T> {
  public int compareTo(T o);
}
interface SortedSet<T extends Comparable<T>>;
```

Haskell:

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) =
    let (lo, hi) = partition (< x) xs
    in  qsort lo ++ [x] ++ qsort hi
```

Funktionen als Argumente und Resultate

als Argumente:

```
even      :: Int -> Bool
partition :: ( a -> Bool ) -> [a] -> ([a], [a])

partition even [ 1,2,3,4,5 ] = ( [2,4], [1,3,5]
```

partiell angewendete Funktionen (liefern Funktionen als Resultat)

```
partition :: ( a -> Bool ) -> [a] -> ( [a], [a]
partition even :: [a] -> ( [a], [a] )
partition even [ 1 .. 5 ] :: ( [Int], [Int] )
```

Änderungen von Typen

- Konversion (Änderung der Repräsentation = des Bitmusters)
  - von Ganzzahl (int) zu gebrochener Zahl (double)
  - Auto(un)boxing in Java
  - automatisches `toString` in Java
- *keine* Konversion (Bitmuster bleibt gleich)
  - immer möglich vom Speziellen zum Allgemeinen
    `Set<String> s = new HashSet<String>();`
  - sonst mit sog. *cast*: der Programmierer überstimmt den Compiler

Übungen zu versteckten Typänderungen

- Zahlbereiche:

```
class Cast {
    static double ceck (int i) {
         return i + 2.71828;
    }
}
```

Kompilieren, class-file disassemblieren

```
javac Cast.java ; javap -c Cast
```

welche Typumwandlungen werden sichtbar?

- was passiert bei Änderung zu `static int check ...?`
- Entsprechendes mit einem C-Compiler (`gcc -S`)
- toString in Java: Erklären Sie Unterschiede zwischen

```
System.out.println ("a" + 3 + 4       );
System.out.println (      3 + 4 + "b");
```

Schlagen Sie Erklärungen in Java-Sprachdefinition nach (java.sun.com)

Warum gibt es zu schwache Typsysteme?

(z. B. char = boolean = int in C, unsichere Collections im Vorzeit-Java) … wenn die Nachteile doch bekannt sind: Fehler treten erst zur Laufzeit auf (gefährlich) oder müssen abgefangen werden (aufwendig).

- „in der Maschine ist sowieso alles int“
- Unkenntnis der Designer (hoffentlich nicht)
- Designer hat Angst vor Unkenntnis der Programmierer
- Designer will Programmierer nicht einschränken…

  das deutet aber auf Designfehler an anderen Stellen hin:

Typsystem wird umgangen (oder weggelassen), weil man *Polymorphie* möchte, aber nicht korrekt hinschreiben kann.

Diskussion

- Wir haben bis jetzt Ausdrücke, Typen, Werte.
- Was fehlt noch zu einer Programmiersprache?

Kompilation für Keller-Maschinen

Befehlssatz

Betrachten einfaches (Keller-)Maschinenmodell, ähnlich zu offizieller JVM.
ein Rechenschritt ist: Befehl B aus Programmspeicher C holen und ausführen:
`B := C [PC]; execute(B); PC := PC+1;`

- Rechnen im Stack: wobei
  `push(x) ==> S[SP] := x; SP := SP+1;` und
  `pop(y) ==> SP := SP-1; y := S[SP];`
  - Push i: `push(i);` und Drop: `pop();`
  - Add (Sub, Mul) `pop(B); pop(A); push (A+B);`

- Speicherzugriffe:
  - Load: `pop(A); push(M[A]);`
  - Store: `pop(A); pop(B); M[A] := B;`
    d. h. Benutzung so:
    `push (Wert); push (Adresse); Store`
- Sprünge: unbedingt oder bedingt (beide relativ - warum?)
  - Jump r: `PC := PC + r`
    ändert Stack nicht
  - Jumpz r: `pop (A); if 0 == A then Jump r;`
    verbraucht top of stack

Im Zweifelsfall: RTFC (Read The F...ing Code):
`http://141.57.11.163/cgi-bin/cvsweb/tool/src/JVM/#dirlist`

PostScript

```
/Times-Roman findfont   % Get the basic font
20 scalefont            % Scale the font to 20 points
setfont                 % Make it the current font

newpath                 % Start a new path
72 72 moveto            % Lower left corner of text at (
(Hello, world!) show    % Typeset "Hello, world!"

showpage
```

Beispiel aus
`http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html`
Dort auch kompletter Befehlssatz. (Aufgabe: Berechne Liste der ersten 100 Primzahlen.)

Forth

`http://www.forth.org/`, *Gordon Charlton*: Introduction to Forth using StackFlow

`100 9 5 */ 32 + . [cr] 212 ok`

Stack-Operationen:

- SWAP $(ab \to ba)$
- ROT $(abc \to bca)$
- DUP $(a \to aa)$
- OVER $(ab \to aba)$
- DROP $(a \to)$

Euklid:

`BEGIN DUP WHILE TUCK MOD REPEAT DROP ;`

`WHILE` testet top of stack. Was macht `TUCK`?

Die Java-VM

Definiert hardware-unabhängige Plattform zur Ausführung von Java-Programmen.
Spezifikation: `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

```
public static int example1(int, int);
  Code:
   0:   iconst_0
   1:   istore_2
   2:   iload_1
   3:   ifle    16
   6:   iload_2
   7:   iload_0
   8:   iadd
```

```
   9:   istore_2
   10:  iinc    1, -1
   13:  goto    2
   16:  iload_2
   17:  ireturn
```

benutzt *operand stack* zum Rechnen und lokalen Speicher (Tload/Tstore), Anweisungen sind getypt (iload, . . .)

Übersetzung von Ausdrücken

Ausdruck gegeben als abstrakter Syntaxbaum

`Exp -> Zahl | Name | Operator Exp Exp`

Für Ausdruck a erzeuge Code, der insgesamt `Push (Wert a)` entspricht.

```
code (Zahl i)
    ==> Push i
code (Name n)
    ==> Push (Adresse von n) ; Load
code (Op e1 e2)
    ==> code (e1) ; code (e2) ; Op
```

Übersetzung von Anweisungen

- einfache Anweisungen:
  Zuweisungen, Prozedur-Aufrufe
- zusammengesetzte Anweisungen:
  Folge (Block), Verzweigung, Schleifen

Für Anweisung: erzeuge Code,
der Stack insgesamt *nicht* ändert.

Folgen:

```
code ( s1 ; s2 ; .. )
    => code (s1) ; code (s2) ; ...
```

Übersetzung von Zuweisungen

Grammatik:

```
Statement -> Assignment | ... ;
Assignment  ->  Name := Exp ;
```

Übersetzung:

```
code ( n := e )
  ==> code (e) ; Push (Adresse von n) ; Store
```

Was passiert, wenn der Name komplizierter ist? Beispiel:

`a [i+1] := a [i];`

Adressen und Werte

`a [i+1] := a [i];`

- Ausdruck links von `:=` muß *Adresse* liefern
- Ausdruck rechts von `:=` muß *Wert* liefern

Bezeichnung in Sprache C: lvalue/rvalue
Übersetzung von rvalues durch Funktion `code`
Übersetzung von lvalues:

```
lvalue (Name n) ==> Push ( Adresse von n )
lvalue (Zahl z) ==>  verboten !
lvalue ( Array a [ Exp ] ) ==>
    code ( Exp ) ; // als rvalue!
    Push ( Adresse von a ) ; Add
```

Bereichsprüfungen?

Übersetzen von Verzweigungen

`Statement -> .. | If Exp Statement Statement`

```
code (If e y n) ==>
    code (e)
    Jumpz nein
    code (y)
    Jump ende
nein: code (n)
ende:
```

JVM benutzt intern int 0/1 statt boolean false/true,
das kann aber von außen niemand ausnutzen,
da vorher die Java-Typprüfung stattfindet.

Übersetzen von Schleifen

`Statement -> .. | While Exp Statement`

```
code (While e b) ==>
test: code (e)
    Jumpz ende
    code (b)
    Jump test
ende:
```

Stacktiefe

Wenn man auf vorher beschriebene Weise Ausdrücke und Anweisungen kompiliert, dann kann die während der Ausführung maximal nötige Stacktiefe bereits zur Compilezeit bestimmt werden.
Bemerkung: gilt nicht für Unterprogramm-Aufrufe. in JVM gibt es bei jedem solchen Aufruf einen neuen Stack(-Frame).

Übungsaufgaben

- autotool/JVM (Beispiel)

```
Konstruieren Sie eine Maschine, welche die
Funktion \ [ x1, x2 ] -> x1 + x2  berechnet!
    [ Push 1, Load, Push 2, Load, Add
    , Push 0, Store, Halt ]
```

- wirkliche JVM:

  welcher Quelltext gehört zu `example1`?

- welcher Quelltext gehört zu

```
static int guess(int, int);
  Code:                    8:   iload_1
```

```
   0:   iload_1          9:   istore_0
   1:   ifle    15       10:  iload_2
   4:   iload_0          11:  istore_1
   5:   iload_1          12:  goto    0
   6:   irem             15:  iload_0
   7:   istore_2         16:  ireturn
```

- Wie wird `a[i] = 5` übersetzt?
- Wie wird `a [++i] = a [++i]` übersetzt, wo ist das in JLS festgelegt?

Unterprogramme

Definition

Unterprogramm ist:

- benannter Block
- mit (evtl.) Ein- und Ausgabe-Schnittstellen

übliche Unterscheidung (entspr. Schlüsselwörtern in Pascal):

- Funktion:

  liefert Wert, Funktionsaufruf ist Ausdruck

- Prozedur:

  liefert keinen Wert, Prozeduraufruf ist Anweisung

Beispiele für Unterprogramme (Funktionen)

$$\begin{aligned} f(x) &= \textbf{if } x > 100 \textbf{ then } x - 10 \textbf{ else } f\big(f(x+11)\big) \\ t(x,y,z) &= \textbf{if } x \le y \textbf{ then } y \\ &\quad \textbf{else } t\big(t(x-1,y,z),\, t(y-1,z,x),\, t(z-1,x,y)\big) \end{aligned}$$

Aufgaben: $f(7), t(30, 20, 10)$
Beobachtung: es ist gar nicht klar,...

- (denotationale Semantik) ... ob solche Gleichungen überhaupt eine oder genau eine Funktion als Lösung haben.
- (operationale Semantik) ... ob und nach welcher Zeit man durch mutiges Ausrechnen Funktionswerte findet.

Parameter-Übergabe (Semantik)

Datenaustausch zw. Aufrufer (caller) und Aufgerufenem (callee): über globalen Speicher

```
#include <errno.h>
extern int errno;
```

oder über Parameter.
Datentransport (entspr. Schüsselwörtern in Ada)

- in: (Argumente) vom Aufrufer zum Aufgerufenen
- out: (Resultate) vom Aufgerufenen zum Aufrufer
- in out: in beide Richtungen

Parameter-Übergabe (Implementierungen)

- pass-by-value (Wert)
- copy in/copy out (Wert)
- pass-by-reference (Verweis)
- pass-by-name (textuelle Substitution)

  selten ... Algol68, CPP-Macros ... Vorsicht!

Parameterübergabe

häufig benutzte Implementierungen:

- Pascal: by-value (default) oder by-reference (VAR)
- C: by-value (Verweise ggf. selbst herstellen)
- C++ unterscheidet zwischen Zeigern (*, wie in C) und Referenzen (&, verweisen immer auf die gleiche Stelle, werden automatisch dereferenziert)
- Java: primitive Typen *und* Referenz-Typen (= Verweise auf Objekte) by-value

Call-by-name

In Algol 68 (und CPP und ähnlich) geht im Prinzip sowas:

```
double scalar
        (int i, int n, double x, double y)
{ double sum = 0;
  for (i = 0; i < n; i++) { sum += x * y; }
  return sum;
}
```

(das ist allein völlig unverständlich — aber die Absicht ist:)

```
int n;
double [n] [n] a; double [n] [n] b;
int i;
int s = scalar (i, n, a[0,i], b[i,0]);
```

Aufgaben zu Parameter-Modi

Durch welchen Aufruf kann man diese beiden Unterprogramme semantisch voneinander unterscheiden:

Funktion (C++): (call by reference)

```
void swap (int & x, int & y)
   { int h = x; x = y; y = h; }
```

Makro (C): (call by name)

```
#define swap(x, y) \
   { int h = x; x = y; y = h; }
```

Wie kann man jedes der beiden von copy-in/copy-out unterscheiden?

Frames

Frame (Rahmen) ist Speicherbereich zur Verwaltung eines Unterprogramm-Aufrufs. Bestandteile sind:

- Platz für Parameter (Argumente)
- Platz für lokale Variablen
- (bei JVM) Operand Stack für lokale Rechnungen

  beachte: Größe steht zur Compile-Zeit fest
- Rückkehradresse
- (bei Funktionen) (Verweis auf) Platz für Resultat

Frames (II)

Zu jedem Zeitpunkt gibt es einen *aktuellen* Frame.
Zum Unterprogramm-Aufruf legt der Caller einen neuen Frame an und schreibt die Werte der Parameter hinein.
Wenn das fertig ist, wechselt die Kontrolle zum Callee.
Wenn der fertig ist, wieder zurück zum vorigen Frame.

Unterprogramme bei x86

Unterprogramm-Deklaration:

```
int simple (int x) {
 return x + 5;
}
```

```
simple: pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%edx
        addl $5,%edx
        movl %edx,%eax
        jmp .L5
        .p2align 4,,7
.L5:    leave
        ret
```

Unterprogramm-Aufruf:

```
    r = simple (4);
```

```
addl $-12,%esp
pushl $4
call simple
addl $16,%esp
movl %eax,%eax
movl %eax,-4(%ebp)
```

Frame-Optimierungen

Literatur: Apppel: Modern Compiler Implementation, Chapt. 6
Die Frames decken den allgemeinsten Fall ab (rekursive Unterprogramme mit beliebig vielen Parametern).
Viel häufiger sind jedoch Spezialfälle: keine Rekursion, und nur geringe Schachtel-Tiefe; deswegen lohnt es sich, diese zu optimieren.

Register

Die Übergabe in Frames (im Hauptspeicher) ist langsam.
Besser sind Register (in der CPU). Frames gibt es viele, aber Register nur einmal. Vor Überschreiben Werte retten!
Wer macht das? Der Caller (vorausschauend) oder der Callee (wenn er es braucht).
Dazu gibt es evtl. Konventionen (z. B. auf MIPS sollen Register 16–23 callee-save sein)

Register-Windows

der Registersatz ist tatsächlich ziemlich groß, damit spart man sich (für eine Weile) das Arbeiten auf dem Stack.
Man benutzt Register g0 . . . g7 (global), i0 . . . i7 (input), l0 . . . l7 (local), o0 . . . o7 (output).
Sparc hat 128 Register, eingeteilt in 8 Blöcke zu je 8 in-registern und 8 local-registern. Die out-register sind die in-register des nächsten Blocks! (D. h. dort schreibt man Argumente für Unterprogramme hin, und holt sich auch das Ergebnis.)
Erst wenn die Schachteltiefe größer als 8 Aufrufe wird, muß in den Speicher geschrieben werden.

Unterprogramme auf Sparc

Unterprogramm-Deklaration:

```
int simple (int x) {
 return x + 5;
}
```

```
simple: !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        st      %i0, [%fp+68]
        ld      [%fp+68], %o1
        add     %o1, 5, %o0
        mov     %o0, %i0
        ret
        restore
```

Unterprogramm-Aufruf:

```
    r = simple (4);
```

```
mov     4, %o0
call    simple, 0
st      %o0, [%fp-20]
```

Aufgaben: wo liegen die beteiligten Register, was passiert genau bei `save/ret/restore`?

Seminar Unterprogramme

- warum kann man für Java bereits zur Compile-Zeit ausrechnen, wie groß der operand stack *einer Methode* zur Laufzeit höchstens sein wird?
- Frames und register windows auf Sparc: was passiert wirklich bei save/restore/ret? (Google nach Dokumentation!)
- welche Transformationen nehmen Gnu-CC bzw. Sun-CC (mit `-O`) bei (evtl. geschachtelten) Unterprogramm-Aufrufen vor? (An selbst gewählten Beispielen ausprobieren, Resultate (Assembler) interpretieren.)

End-Aufrufe (Beispiel)

Bei End-Aufruf von Q durch P:

- den Frame zum Q-Aufruf in den P-Frame hineinschreiben
- dann zum Anfang von Q *springen* (*jump*, nicht *call*)

```
void p () { int a; int b; ... ; q (); }

void q () { int c; int d; ... ; return; }
```

Aufgabe: Was ist zu beachten, wenn die Unterprogramme Parameter und Resultate haben?

Transformationen für Tail-Calls

die linke Funktion ist nicht tail-rekursiv, aber die rechte.

```
int sum (int x) {
  if (x > 0) {
    return
      sum (x-1) + x*x;
  } else {
    return 0;
  }
}
```

```
int sum_trick (int x) {
  return sum_helper (x, 0);
}

int sum_helper
     (int x, int accu) {
  if (x > 0) {
    return sum_helper
      (x-1, accu + x*x);
  } else {
    return accu;
  }
}
```

Aufgabe: werden *genau* die gleichen Rechnungen ausgeführt? (Nein.)

Gesamt-Analyse

Wenn der Compiler den kompletten Programmtext sieht: Unterprogramm-Aufrufe können aufgelöst werden (inlining).
Aufgabe: ausprobieren mit `gcc`.
Für verbleibende Aufrufe ist interprozedurale Register-Allokation möglich. Dann müssen Werte von Argumenten nicht umkopiert werden.
Beachte: Konflikt mit modularer Programmierung (Quelltext der Module muß vorhanden sein).

End-Aufrufe (tail calls)

Wenn die letzte Aktion eines Unterprogramms P der Aufruf eines Unterprogramms Q ist (eine End-Ruf, *tail call*), dann wird während der Rechnung von Q der Frame von P nicht mehr gebraucht.
Deswegen sollte man diesen Frame bereits *vor* dem Ruf von Q freigeben.

End-Rekursionen (tail recursion)

ein Spezialfall von End-Aufrufen sind End-Rekursionen.
Durch Auflösung von End-Rekursionen entstehen Schleifen.

```
void tabulate (int x) {
  if (x > 0) {
    printf ("%9d, %9d\n",
            x, x*x*x);
    tabulate (x-1);
  }
}
```

```
void tabulate (int x) {
  start: if (x > 0) {
    printf ("%9d, %9d\n",
            x, x*x*x);
    x = x-1;
    goto start;
  }
}
```

Aufgabe: als while-Schleife?

Geschachtelte Unterprogramme

Blockstruktur und Orthogonalität

```
void f (int x) { int y = x * x; return y - 1; }
```

- Unterprogramm: Schnittstelle + Implementierung (Block)
- Block: Deklarationen + Anweisungen
- Deklaration: von Variablen, Typen?, Unterprogrammen?
- Werte (Argumente, Resultate von Unterprogrammen)?

Sprachelemente sollten unabhängig voneinander (orthogonal zueinander) sein ⇒ beliebig kombinierbar ⇒ unterstützt modularen Entwurf

Einschränkungen der Orthogonalität

In vielen Programmiersprachen sind einige Elemente nicht orthogonal zueinander. Nenne Beispiele!
Warum ist das so?

- aus Unkenntnis (nachlässiger Sprach-Entwurf)
- absichtlich:
  - zum „Schutz“ des Programmierers (vor sich selbst)
  - für leichtere Implementierung

Versuche, die o. g. Beispiele zuzuordnen.

Lokale Unterprogramme: Beispiel

Das geht in Standard-C nicht (aber `gcc` kann es doch):

```
int f ( int x ) {
  int g ( int y ) {
    int h ( int z ) {
      return x + y + z;
    }
    return h (h (h (0)));
  }
  return g (g (0));
}
```

Wie verläuft die Rechnung `f(3)`?

Lokale Unterprogramme: Implementierung

Für jeden Aufruf eines Unterprogramms gibt es einen Frame.
Werte von lokalen Variablen aus umgebenden Blöcken müssen aus entsprechenden Frames gelesen werden.
Wie findet man diese? (Es ist *nicht* unbedingt der direkt vor dem aktuellen Frame liegende.)
Lösung: *statische Kette*.
Jeder Frame (eines lokalen Unterprogramms) enthält einen Verweis auf den Frame des textuell direkt umschließenden Unterprogramms.

Statische Ketten: Beispiel

	statische Kette benutzen	Kette verlängern
`int f ( int x )`	`h.37:`	`g.33:`
`{`	`pushl %ebp`	`pushl %ebp`
` int g ( int y )`	`movl %esp,%ebp`	`movl %esp,%ebp`
` {`	`subl $24,%esp`	`subl $24,%esp`
`  int h ( int z )`	`movl %ecx,-4(%ebp)`	`movl %ecx,-4(%ebp)`
`  {`	`movl -4(%ecx),%eax`	`leal 8(%ebp),%eax`
`   return`	`movl -4(%eax),%edx`	`movl %eax,-8(%ebp)`
`     x + y + z;`	`movl -8(%ecx),%eax`	`addl $-12,%esp`
`  }`	`movl (%eax),%eax`	`addl $-12,%esp`
`  return`	`addl (%edx),%eax`	`addl $-12,%esp`
`    h (h (h (0)));`	`addl 8(%ebp),%eax`	`pushl $0`
` }`	`leave`	`movl %ebp,%ecx`
` return g (g (0));`	`ret`	`call h.37`
`}`		

Statische und Dynamische Ketten

- statische Kette:

  Verweis auf Frame des *textuell umgebenden* Unterprogramms
- dynamische Kette:

  Verweis auf Frame des *aufrufenden* Unterprogramms

im einfachsten Fall (Standard-C)

- dynamischer Vorgänger = der vorige Frame (auf dem Stack)
- statischer Vorgänger: gibt es nicht (keine lokalen Funktionen)

Seminar 11. 11.: Statische Ketten

- Wo erzeugt GCC für Sparc statische Links? Zeigen Sie an einem selbst gewählten Testfall das Verlängern und das Benutzen einer statischen Kette.
- Schreiben Sie einen Testfall, bei dem der statische Link (möglicherweise) sehr viel weiter zeigt als der dynamische Link.

  Verifizieren Sie (durch Betrachten des Assembler-Files) das korrekte Erzeugen und Benutzen des statischen Links (vgl. vorige Teilaufgabe).

Beispiel Statische/Dynamische Kette

Wie verläuft der Aufruf $f(3)$?

```
int f (int x) {
  int g (int y) { return x + y; }
  int h (int x) { return g (g (x)); }
  return h (2);
}
```

Unterprogramme als Daten

Orthogonalität (II)

oft gibt es diese Unterscheidung:

- Daten: Zahlen, Zeichen, Arrays/Records von Daten
- Funktionen: Abbildung von (Tupel von) Daten nach Daten

ist mathematisch nicht gerechtfertigt und führt zu umständlichen Entwürfen.
Richtig ist: *Funktionen sind Daten*.
wg. Orthogonalität: Funktionen können auch *Argumente* und *Resultate* von Funktionen sein. (Wenn man das nicht hat, muß man es simulieren ⇒ sog. objektorientierte Programmierung: Klassen sind Records von Funktionen)

Funktionen als Argumente

```
void walk
(void action (int),
  tree * t) {
  void help (tree * t) {
    if (NULL != t) {
      action (t -> node);
      help (t -> left);
      help (t -> right);
    }
  }
  help (t);
}
```

```
void display (tree * t)
  void show (int n)
    { printf ("%d ", n);
  walk (& show, t);
}
int count (tree * t) {
  int counter = 0;
  void one (int n)
    { counter ++; }
  walk (& one, t);
  return counter;
}
```

Aufgabe: geht das in C? in Pascal? in Java?

Zeiger auf Nested Functions

```
int twice
  ( int fun (int)
  , int x )
{
  return
    fun (fun (x));
}
```

```
int f0 ( int x0 ) {
  return x0 * x0;
}

int f1 ( int x1 ) {
  int f2 ( int x2 ) {
    return x1 + x2;
  }
  return twice (f2, 5);
}
```

Für `twice(f0,4)` reicht Zeiger auf den Code von `f0`,
für `twice(f2,5)` *nicht* —

Closures

für `twice(f2,5)` *nicht* —
weil auf eine Variable (`x_1`) zugegriffen wird, die in einem umgebenden Block deklariert ist.
deren Wert steht in einem Frame, dieser ist zu finden!

Lösung:
Für Verweis auf Funktion benutze :
Closure = Paar aus

- (wie üblich:) Zeiger auf Code der Funktion
- (zusätzlich:) Zeiger auf Frame des passenden (umgebenden) Funktionsaufrufes

Funktionen als Resultate?

(Motivation/Wiederholung:) Funktionen sind Daten, können also auch Argumente *und Resultate* von Funktionen sein.
— Aber Vorsicht:

```
int f0 () {
  int (*f1 (int x1))(int)
  {
    int f2 (int x2) {
      return x1 * x2;
    }
    return f2;
  }
```

```
  int (*g3) (int) = f1 (3);
  int (*g4) (int) = f1 (4);
  return g3 (g4 (1));
}
```

`f1(3), f1(4)` sind wie beschrieben *closures*

zur Syntax siehe auch Lars Händel:
`http://www.function-pointer.org/`

Funktionen als Resultate (II)

Bei Verwendung von

Closure = (Code-Zeiger, Frame-Zeiger)

muß zum Zeitpunkt der Benutzung des Codes der verwiesene Frame noch existieren (im Stack).

Das ist so für Funktionen *als Argumente* (Frames stehen noch im Stack),
aber *nicht* für Funktionen als Resultate.

wenn das doch benötigt wird, dann dürfen Frames nicht im Stack stehen, sondern müssen gesondert verwaltet werden (im Heap, mit Garbage collection).

Funktionales Programmieren in Java?

„Rechnen mit Funktionen" ist oft die softwaretechnisch vernünftige Lösung. In Sprachen, die das nicht direkt unterstützen, kann man das evtl. simulieren.
Bsp: Java hat keine Zeiger auf Funktionen, deswegen:

- Collections-Framework, Sortiermethode für Listen, mit nutzerdefinierter Vergleichsfunktion:

  Nutzer übergibt Comparator-Objekt,
  die Vergleichsfunktion ist dessen (einzige) Methode.
- Ereignisbehandlung in Applets:

  Nutzer übergibt ActionListener-Objekt,
  das Unterprogramm (das bei Click auszuführen ist), ist dessen (einzige) Methode.

Beispiel: Collections, Komparatoren

```
interface Comparator<E> {
    int compare (E x, E y);
}
class LengthLex
    implements Comparator<String>
{
  int compare (String x, String y) {
    // erst nach Länge, dann lexikografisch
  }
}
OrderedSet<String> s =
   new TreeSet<String> (new LengthLex ());
```

vgl. VL Objektorientiert Konzepte (Sommersemester)

Beispiel: Ereignisbehandlung

```
public class Counter { ...
  Button inc = new Button ("inc");
  public void init () { ...
    add (inc);
    inc.addActionListener (
      // anonyme innere Klasse:
      new ActionListener () {
        public void actionPerformed
            (ActionEvent a) {
          status++; update ();
      } }
    );
} }
```

Funktionales Programmieren in Java (II)

Funktionen gibt es nur als Methoden von Objekten/Klassen. Was folgt daraus für die hier diskutieren Probleme und Lösungen:

- wo sind die Frames?
- wo ist die statische Kette?
- wie sehen Closures aus?
- welche Speicherverwaltung ist nötig?

(= Testfragen zur Wiederholung des Kapitels „Unterprogramme“)

Innere Klassen

- nicht statische innere Klasse

```
class A {
  class B { .. }
}
```

B gehört zu einem A-Objekt, B darf keine statischen Komponenten haben.

- statische innere Klasse

```
class A {
  static class B { .. }
}
```

Lokale Klassen

allgemein:

```
void h (int z) { class C { ... } }
```

Wie wird das implementiert (wo ist das Problem)?

```
interface I { void p (int x); }
void h (int z) {
  final int a = 4 + z;
  class C implements I {
    void p (int x) {
       System.out.println (a + x);
  } }
  return new C ();
}
void check () { I b = h (5); b.p (3); }
```

Java funktional?

Motto: *a good programmer can write LISP in any language*, hier: *LISP* = Haskell,

```
$ ghci
Prelude> let  square x = x*x
Prelude> map ( succ . square ) [1,2,3]
[2,5,10]
```

und *any* = Java:

```
interface Function<From,To> {
  To apply (From arg);
}
public static void main(String[] args) {
  List<Integer> xs =
    Arrays.asList(new Integer[] { 1, 2, 3 });
```

```
  System.out.println(xs);
  List<Integer> ys =
    Functions.map(
       Functions.combine(square,succ), xs);
  System.out.println(ys);
}
```

Schreiben Sie die fehlenden Deklarationen (succ, square, map, combine). Hinweis:

```
static final Function<Integer, Integer> succ =
  new Function<Integer, Integer>() {
    public Integer apply(Integer x) {
      return x + 1;
    }
  };
```

Welcher Bytecode wird erzeugt für

```
static Function<Integer, Integer>
  multiplyBy (final Integer y) {
    return new Function<Integer,Integer>() {
      public Integer apply (Integer x) {
        return x*y;
      }
    };
}
```

Code-Generierung und -Optimierung

Compiler-Phasen

- Front-End (abhängig von Quellsprache):
  - Eingabe ist (Menge von) Quelltexten
  - lexikalische Analyse (Scanner) erzeugt Liste von Tokens
  - syntaktische Analyse (Parser) erzeugt Syntaxbaum
  - semantische Analyse (Typprüfung, Kontrollfluß, Registerwahl) erzeugt Zwischencode
- Back-End (Abhängig von Zielsprache/Maschine):
  - Zwischencode-Optimierer
  - Code-Generator erzeugt Programm der Zielsprache
  - (Assembler, Linker, Lader)

Zwischencode-Generierung

Aufgabe:

- Eingabe: annotierter Syntaxbaum
- Ausgabe: Zwischencode-Programm (= Liste von Befehlen)

Arbeitsschritte (für Registermaschinen):

- common subexpression elimination (CSE)
- Behandlung von Konstanten
- Register-Zuweisungen
- Linearisieren

Common Subexpression Elimination — CSE

- Idee: gleichwertige (Teil)ausdrücke (auch aus verschiedenen Ausdrücken) nur einmal auswerten.
- Implementierung: Sharing von Knoten im Syntaxbaum
- Vorsicht: Ausdrücke müssen wirklich völlig gleichwertig sein, einschließlich aller Nebenwirkungen.
- Auch Pointer/Arrays gesondert behandeln.

Beispiele: $f(x) + f(x)$; $f(x) + g(y)$ und $g(y) + f(x)$; $a * (b * c)$ und $(a * b) * c$; `.. a [4] .. a [4] ..`
Aufgabe: untersuchen, wie weit `gcc` CSE durchführt. Bis zum Seminar Testprogramme ausdenken!

Constant Propagation

- konstante Teil-Ausdrücke kennzeichnen
- und so früh wie möglich auswerten

  z. B. *vor* der Schleife statt in der Schleife)
- aber nicht zu früh!

  z. B. `A` nicht vor einer Verzweigung
  `if ( .. ) { x = A; }`

Constant Folding, Strength Reduction

strength reduction:
"starke" Operationen ersetzen,
z. B. `x * 17` durch `x << 4 + x`

constant folding:
Operationen ganz vermeiden:
konstante Ausdrücke zur Compile-Zeit bestimmen
z. B. `c + ('A' - 'a')`

Aufgabe: wieweit macht `gcc` das? Tests ausdenken!
evtl. autotool zu strength reduction (Additionsketten)

Daten-Fluß-Analyse

bestimmt für jeden Code-Block:

- gelesene Variablen
- geschriebene Variablen

ermöglicht Beantwortung der Fragen:

- ist Variable x hier initialisiert?
- wann wird Variable y zum letzten mal benutzt?
- ändert sich Wert des Ausdrucks A?

Datenfluß (II)

Problem: zur exakten Beantwortung müßte man Code ausführen. (Bsp: Verzweigungen, Schleifen)

```
while ( .. ) {
  int x = 3;   int y;
  if ( .. ) { x = 2 * y; } // ??
  else      { y = 2 * x; }
}
```

Ausweg: Approximation (sichere Vereinfachung) durch *abstrakte Interpretation*, die Mengen der initialisierten/benutzten/geänderten Variablen je Block berechnet (d. h. als Attribut in Syntaxbaum schreibt)
z. B. bei Verzweigungen beide Wege „gleichzeitig" nehmen
weiteres Beisp. f. abst. Interpretation: Typprüfung

Linearisieren

zusammengesetzte (arithmetische) Ausdrücke übersetzen:

- für Stack-Maschinen (bereits behandelt, siehe JVM)
- für Register-Maschinen: Linearisieren, d. h.

  in einzelne Anweisungen mit neuen Variablen (für jeden Teilbaum eine):

  aus `x = a * a + 4 * b * c` wird:

```
h1 = a*a;
h2 = 4*b;
h3 = h2 * c;
x = h1 + h3
```

Registervergabe

benötigen Speicher für

- lokale Variablen und
- Werte von Teilausdrücken (wg. Linearisierung)

am liebsten in Registern (ist schneller als Hauptspeicher)
es gibt aber nur begrenzt viele Register.

Zwischencode-Generierung für "unendlich" viele symbolische Register, dann Abbildung auf Maschinenregister und (bei Bedarf) Hauptspeicher (register spilling).

Register-Interferenz-Graph

(für einen basic block)

- Knoten: die symbolischen Register $r_1, r_2, \ldots$
- Kanten: $r_i \leftrightarrow r_k$, falls r_i und r_k gleichzeitig lebendig sind.

  (lebendig: wurde initialisiert und wird noch gebraucht)

finde Knotenfärbung (d. i. Zuordnung c: symbolisches Register $\rightarrow$ Maschinenregister) mit möglichst wenig Farben (Maschinenregistern), für die $\forall (x,y) \in E(G) : c(x) \neq c(y)$.
Ist algorithmisch lösbares, aber schweres Problem (NP-vollständig)

Register-Graphen-Färbung (Heuristik)

Heuristik für Färbung von G:

- wenn $|G| = 1$, dann nimm erste Farbe
- wenn $|G| > 1$, dann
  - wähle $x =$ irgendein Knoten mit minimalem Grad,
  - färbe $G \setminus \{x\}$
  - gib x die kleinste Farbe, die nicht in Nachbarn $G(x)$ vorkommt.

Aufgabe: finde Graphen G und zulässige Reihenfolge der Knoten, für die man so keine optimale Färbung erhält.
Falls dabei mehr Farben als Maschinenregister, dann lege die seltensten Registerfarben in Hauptspeicher.
(Es gibt bessere, aber kompliziertere Methoden.)

Seminar: Registervergabe

Datenfluß-Analyse für:

```
int fun (int a, int b) {
  int c; int d; int e; int f;
  c = a + b;
  d = a + c;
  e = d + a;
  b = c + d;
  e = a + b;
  b = d + b;
  f = c + e;
  return b ;
}
```

Register-Interferenz-Graph bestimmen und färben.

Danach . . . mit Ausgabe von `gcc -S -O` vergleichen, Unterschiede erklären. Besseren Testfall konstruieren.

Peephole-Optimierung, Instruction Selection

- Zwischencode-Liste übersetzen in Zielcode-Liste.
- kurze Blöcke von aufeinanderfolgenden Anweisungen optimieren (peephole — Blick durchs Schlüsselloch)
- und dann passenden Maschinenbefehl auswählen.
- durch Mustervergleich (pattern matching), dabei Kosten berechnen und optimieren

`gcc`: Zwischencode ist maschinenunabhängige RTL (Register Transfer Language),
damit ist nur Instruction Selection maschinenabhängig
$\rightarrow$ leichter portierbar.

Einzelheiten zu gcc

- Home: `http://gcc.gnu.org/`

  Kopie der Sourcen hier:
  `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws03/compilerbau/programme/gcc-3.3.2/`
- Beschreibung von Prozessor-Befehlen (RTL-Patterns) z. B.: `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws03/compilerbau/programme/gcc-3.3.2/gcc/config/sparc/sparc.md`
- Java-Code compilieren:
  `/usr/local/bin/gcj -S [-O]` für

```
class Check {
```

```
    static int fun (int x, int y) {
        return x + x - y - y;
    }
}
```

(vergleiche mit `javac, javap -c`)

Ergänzungen, Zusammenfassung

Compiler oder Interpreter?

- Compiler *und* Interpreter:

  lexikalische $\rightarrow$ syntaktische ($\rightarrow$ semantische) Analyse $\rightarrow$ Zwischencode
- Compiler:

  $\rightarrow$ Optimierer $\rightarrow$ Zielprogramm
- Interpreter:

  $\rightarrow$ sofortige Ausführung des Zwischencodes

Compiler *und* Interpreter

schließlich wird doch interpretiert: (CPU ist Interpreter für die Maschinensprache).
statt konkreter Maschine: abstrakte Maschine benutzen.
Compiler erzeugt dann abstrakten Maschinencode,
Laufzeitumgebung muß Interpreter für abstrakte Maschine enthalten (Beispiele: Java, Pascal)

- Vorteil: ist modular → flexibel (in Quellsprache, in Zielsprache/Maschine)
- Nachteil: Ineffizienz durch Interpretation.
- Abhilfe: Compilierung des abstrakten Codes
  - vor Ausführung (z. B. `gcj`)
  - während Ausführung, an *hot spots*

Cross-Compilation

Drei Parameter (Sprachen) für einen Compiler: $\mathrm{Q}_{\mathrm{I}}\mathrm{Z}$:

- Quellsprache Q
- Implementierungssprache (/Maschine) I
- Zielsprache (/Maschine) Z

Beispiel: $S = \mathrm{C}_{\mathrm{Sparc}}\mathrm{Sparc}$. — Wie entsteht $\mathrm{C}_{\mathrm{Intel}}\mathrm{Intel}$?

Schreiben $P = \mathrm{C}_{\mathrm{C}}\mathrm{Intel}$.
Compiliere P mit S (auf Sparc), es entsteht $Q = \mathrm{C}_{\mathrm{Sparc}}\mathrm{Intel}$.
Compiliere P mit Q (auf Sparc), es entsteht $\mathrm{C}_{\mathrm{Intel}}\mathrm{Intel}$.

Bootstrapping zur Selbst-Optimierung

gegeben:

- ein „guter" (optimierender) $G = \mathrm{S}_{\mathrm{S}}\mathrm{M}$,
- ein „schlechter" $A = \mathrm{S}_{\mathrm{P}}\mathrm{P}$.

  (läuft lange und erzeugt langsamen Code)

gesucht: $\mathrm{S}_{\mathrm{M}}\mathrm{M}$
Compiliere G mit A, ergibt $B = \mathrm{S}_{\mathrm{P}}\mathrm{M}$
(erzeugt effizienten Code, ist aber selbst nicht effizient).
Compiliere G mit B, ergibt $C = \mathrm{S}_{\mathrm{M}}\mathrm{M}$
(erzeugt effizienten Code *und* ist selbst effizient).
auf diese Weise auch Implementierung von Sprach-Erweiterungen.

Super-Compilation

(Massalin 1987, Granlund und Kenner 1992, zitiert in Grune et al: Modern Compiler Design, 4.2.8)

- für arithmetische/logische Geradeaus-Programme
- teste *alle* Folgen von Maschinenbefehlen der Länge 1, 2, ... durch Ausführen (die meisten Fehler werden schnell entdeckt)
- Beweise die Korrektheit der Folgen, die Tests überstanden haben.

Geschichte des Compilerbaus

- erste maschinenunabhängige höhere Programmiersprache (mit arithmetischen Ausdrücken): Fortran ≈ 1955,

  Keller-Prinzip (siehe Vortrag von F. L. Bauer)
- formale Syntax für Programmiersprachen (durch CF-Grammatiken): John Backus, Peter Naur, für Algol ≈ 1960
- Zwischencode-Erzeugung und -Transformation durch Attribut-Grammatiken: Donald Knuth, 1967

Details wie Symboltabellen, Fehlerbehandlung, Optimierungen ab 1970 im wesentlichen bekannt.

Geschichte des Compilerbaus (II)

seitdem neuere Entwicklungen zu

- Modulsysteme,
- Typsysteme,
- abstrakte Maschinen
- standardisierter Syntax (XML)

Mathematische Methoden

- reguläre Ausdrücke, endliche Automaten: lexikalische Analyse;
- kontextfreie Grammatiken, Kellerautomaten: syntaktische Analyse;
- Bäume, Pattern Matching: Typ-Prüfung, Code-Generierung
- Graphen, Färbungen: Datenabhängikeiten, Registervergabe

Compilerbau: Zweck der Lehrveranstaltung

Bestandteile und Arbeitsweise eines Compilers verstehen, dabei zugrundeliegende Modelle (Grammatiken, Automaten, Bäume, Graphen) und Verfahren (aus dem Grundstudium) wiedererkennen,
Fähigkeiten und Fertigkeiten im Umgang mit Compilern (gcc, javac) und Werkzeugen (make, flex, bison, javap, gdb) erwerben,
bei eigenen Projekten anwenden können:
Jedes Programm, das einen Eingabetext liest und eine Aktion ausführt, enthält Elemente eines Übersetzers (Compilers/Interpreters).

Domainspezifische Sprachen

Allgemeine Programmiersprachen gibt es schon genug, und es werden nur wenige verschiedene gebraucht (entspr. der Programmier-Paradigmen).

Es gibt aber viele *anwendungsspezifische* Sprachen. (Konfiguration für Webserver, für Spamfilter; Datenbankabfragen, Gerätesteuerungen, DVD-Menüs…)

Entwurfsfragen sind: Semantik? (Daten? Operationen? Ablaufsteuerung?) Abstraktionen? (Unterprogramme?) Module? Syntax? Ausführung?

- alles selbst entwerfen (ineffizient)
- *oft besser:* Sprache in Gastsprache einbetten (nur Semantik ist domainspezifisch)

Domainspezifische Sprachen

- eingebettet:

  oft als Bibliothek (von ADTs: (abstrakte) Datentypen und Operationen dafür)

  Bsp: `java.awt.*` u. v. a. m.

- wenn die Gastsprache kompiliert wird, kann man die darin eingebettete Sprache nicht interpretieren

  Ausweg: Entwurfsmuster *Interpreter*

Intercal (1972)

Datentypen: 16- und 32-bit-Zahlen, Operationen:

- binär:
  - interleave (mingle): $11\$6 = 158$
  - select: $179 \sim 201 = 9$
- unär (Argument x):
  - and, or, xor jeweils zwischen x und rotate(x)
- Zahleneingabe englisch (ONE, TWO, …), Ausgabe römisch

Intercal (Ablaufsteuerung)

- PLEASE DO [NOT] Anweisung

  (sonst Fehlermeldung: Programmer is insufficiently polite)

- GIVE UP (exit)
- COME FROM (invers zu GOTO)
- PLEASE IGNORE .1

  (Zuweisungen an Variable Nr. 1 werden ignoriert, bis …)

  PLEASE REMEMBER .1

- PLEASE ABSTAIN FROM IGNORING … PLEASE REINSTATE

Intercal

- D. Alexander Garrett: Abstraction and Modularity in Intercal

  …one of the most important aspects of abstraction is *information hiding* …, which sometimes goes by its other name, *job security* …

- Eric S. Raymond: Intercal Resource page

  `http://www.catb.org/~esr/intercal/`

Brainfuck — Beispiel

(quine410.b)

```
->++>+++>+>+>++>>+>+>+++>>+>+>++>+++>+++>+>>
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+>+>++>>>+++>
>>>>+++>+>>>>>>>>>>>>>>>>>>>>>>++>>>>>>>>++>
+++>+++>+>>+++>>>+++>+>+++>+>++>+++>>>+>+>+>
+>++>+++>+>+>>+++>>>>>>>+>+>>>+>+>++>+++>+++
>+>>+++>+++>+>+++>+>++>+++>++>>+>+>++>+++>+>
+>>+++>>>+++>+>>>++>+++>+++>+>>+++>>>+++>+>+
++>+>>+++>>+++>>+[[>>+[>]+>+[<]<-]>>[>]<+<++
+[<]<<+]>>+[>]+++[++++++++++>++[-<++++++++++
++++++>]<.<-<]
```

Brainfuck

Arbeitsspeicher und *ein* Zeiger, Operationen:

- `>` Zeiger nach rechts, `++p`
- `<` Zeiger nach links, `--p`
- `+` Inhalt größer, `++*p`
- `-` Inhalt kleiner, `--*p`
- `.` Inhalt ausgeben, `putchar(*p)`
- `,` Inhalt lesen, `*p=getchar()`
- `[` Schleifenanfang, `while(*p){`
- `]` Schleifenende, `}`

Brainfuck

- kein Witz: diese Sprache ist *Turing-vollständig* (kann jede andere Programmiersprache simulieren, Halteproblem ist unentscheidbar usw.)
- Brain Raiter:
  `http://www.muppetlabs.com/~breadbox/bf/`,
- Panu Kalliokoski, Helsinki:
  `http://esoteric.sange.fi/brainfuck/`
- mit Beispielprogrammern, Interpretern, Compilern, Debuggern, IDEs

Autotool-Highscore-Auswertung

```
•  57 :     30888  Christian Bulz
   56 :     32591  Marcel Matzat
   55 :     32626  Sven Pietsch
```

- Regulärer Ausdruck für $\{w \mid 3 \text{ teilt } |w|_a \wedge 2 \text{ teilt } |w|_b\}$:

```
(a (((bb)^* a (bb)^* b + (bb)^* b a (bb)^*)
 a  (a ((bb)^* a (bb)^* + (bb)^* b a (bb)^* b) a)^*
  (a ((bb)^* a (bb)^* b + (bb)^* b a (bb)^*) a + b)
  +  ((bb)^* a (bb)^* + (bb)^* b a (bb)^* b) a)
 + b ((a ((bb)^* a (bb)^* + (bb)^* b a (bb)^* b) a)^*
  (a ((bb)^* a (bb)^* b + (bb)^* b a (bb)^*) a + b)
  + b))^*
```

mit Größe 200

Ausblick: Erweiterte Sternhöhe

- ein *erweiterter regulärer Ausdruck* besteht aus:
  - $\emptyset$, ϵ, Buchstabe
  - Summe (Vereinigung), Produkt (Verkettung), Stern
  - (das ist die Erweiterung): Komplement bzgl Σ^*
- Die ESH (erweiterte Sternhöhe) eines e. r. Ausdrucks X ist die maximale Schachteltiefe der Sterne.
- Die ESH eine Sprache L ist die kleinste ESH eines e. r. Ausdrucks X mit $L(X) = L$.
- ESH von Σ^* ist 0, denn Σ^* ist Komplement von $\emptyset$, also wird kein Stern benutzt.
- Beispiel: $(ab)^* = a\Sigma^*b \setminus \Sigma^*(aa + bb)\Sigma^*$, also ESH 0.

seit vielen Jahren ungeklärt: *gibt es eine reguläre Sprache mit erweiterter Sternhöhe* > 1*?*