Compilerbau
Vorlesung, Wintersemester 2003

Johannes Waldmann, HTWK Leipzig

28. Januar 2004

Überblick (8. 10. 03)

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
setgray}def 1 0 0 42 1 0 c 0 1 1{0 3 3 90 2
arcn 270 90 c -2 2 4{-6 moveto 0 12 rlineto
9 0 rlineto}for stroke 0 0 3 1 1 0 c 180 ro
```

Eingabe: compilerbau.tex

```
$ latex compilerbau.tex
```

erzeugt compilerbau.dvi

```
$ xdvi compilerbau.dvi

$ dvips compilerbau.dvi
```

erzeugt compilerbau.ps

```
$ gv compilerbau.ps
```

```
$ latex2html compilerbau
```

Empfohlene Literatur/Links

- Webseite zur Vorlesung/Übung, mit Skript, Folien, Aufgaben: `http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/`

- klassisches Lehrbuch: Aho, Sethi, Ullman: Compilers. Principles, Techniques, and Tools. Addison-Wesley 1985 (auch deutsch, aber in sprachlich und TEX-nisch schlechter Übersetzung)

- Helmut Seidl: Abstrakte Maschinen (Teile 1 und 2): `http://www.informatik.uni-trier.de/PSI/abstr_masch_ss01.html`

Organisation

- Vorlesung
  - ungerade, mittwochs, 13:45, Li318
  - gerade, montags, 15:30, G327
- Seminare: Einschreibung über ein Web-Interface, wird noch bekanntgegeben,
  - *für*
    - ungerade, freitags, 11:15, Z530
    - *und* gerade, freitags, 17:15, Z530
  - *oder*
    - ungerade, freitags, 19:00, Z530
    - *und* gerade, mittwochs, 11:15, Z530
- aller 2 Wochen kleinere Aufgaben (Web-Interface)

- Prüfungs-Klausur

Leistungsnachweise

- aller 2 Wochen kleinere Aufgaben (Web-Interface)
  `http://www.imn.htwk-leipzig.de/~waldmann/autotool.html`

  `problem: Computer, aufgabe: JVM, version:`

  (bzw. `version: EXP2, TIMES`)

- Prüfungs-Klausur

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

- 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 $\Rightarrow$ viele Tests zur Compile-Zeit, keine zur Laufzeit $\Rightarrow$ schnelle Programme!

Abstrakte Maschinen

Eine abstrakte Maschine

- Ausführungsmodell: ein (logischer) Prozeß
- Speichermodell:
  - Programm-Speicher C, mit Programm-Zeiger PC
  - statischer Daten-Speicher M (frei adressierbar)
  - Keller-Speicher S, mit Keller-Zeiger SP

Befehlssatz

- `B := C [PC]; PC := PC+1;` dann B ausführen
- Rechnen im Stack: wobei
  `push(x) = S[SP] := x; SP := SP+1;` und
  `pop(y) = y := S[SP]; SP := SP - 1;`
  - 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;`
- Sprünge: unbedingt oder bedingt (beide relativ)
  - Jump r: `PC := PC + r`

- Jumpz r: `pop (A); if 0 == A then Jump r;`

Im Zweifelsfall: RTFC (Read The F...n Code):
`http://theo1.informatik.uni-leipzig.de/cgi-bin/cvsweb/autotool/JVM/`

Aufgaben (1)

Vergleichen Sie unsere abstrakte Maschine mit der offiziellen JVM (Java Virtual Machine) (google).

automatisch korrigierte Aufgaben: Problem: Computer, Aufgabe: JVM.

Implementieren Sie Programme mit den Wirkungen $x_0 := 32$, $x_0 := 42$, benutzen Sie als Konstanten (in `Push i`) nur 0 und 1, und kein `Mul`.

Später: $x_0 :=$ Ihre Matrikelnummer.

EXP1) Implementieren Sie $x_0 := (x_1 + x_2) \cdot (3 + x_1 \cdot x_2) + 27$

EXP2) Implementieren Sie $x_0 := (x_1 + 2 \cdot x_2)^2$.

TIMES) (Später:) Implementieren Sie $x_0 := x_1 \cdot x_2$, ohne `Mul` zu benutzen!

Kompilation von Ausdrücken (13. 10. 03)

Übersetzung von Ausdrücken

Kontextfreie Grammatik, reguläre Baumsprache:

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

wir nehmen an, Baumstruktur ist bekannt (vollst. geklammert).

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

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

Übersetzung von Zuweisungen

Grammatik:

```
Statement -> Assign ; Assign  ->  Name := E
```

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

```
code ( n := e ) ==> code (e) ; Push (Adress
```

Anweisungsfolgen:

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

Felder

Vorsicht: erstmal Arrays ohne Bounds-Checking.

Identifizieren Array mit seiner Start-Adresse, und beginnen

Indizierung bei 0.

Erweitern Grammatiken

```
Element -> Name [ Exp ]
Exp     -> .. | Element
Assign  -> Element := Exp

address ( n [i] ) ==> Push (Addr. von n) ;
code ( n [i] ) ==> address (n [i]) ; load
code ( n [i] := e ) ==> code (e) ; address
```

Übersetzen von Verzweigungen

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

0 = falsch, sonst = wahr (wir haben noch kein Bool)

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

Übersetzen von Schleifen

```
Statement -> .. | While Exp Statement

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

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`

Java-Bytecode betrachten mit `javap`

```
emacs Foobar.java
javac Foobar.java
javap -c -v Foobar
```

Aufgabe 1

schreiben Sie Java-Programme entsprechend der Aufgaben EXP1, EXP2, TIMES.

```
class Check {
    static int exp1 (int x1, int x2) { ...
    public static int Main (String [] args)
}
```

Kompilieren Sie und vergleichen Sie den erzeugten Bytecode mit Ihrer Lösung für das autotool.

Lesen Sie die Dokumentation der dabei benutzten JVM-Befehle.

Wie werden Schleifen (while, for) übersetzt?

Aufgabe 2

Welcher Java-Quelltext gehört (wahrscheinlich) hierzu:

```
Method int example1(int, int)
   0 iconst_0                                8 istore
   1 istore_2                                9 iinc 1
   2 goto 12                                12 iload_
   5 iload_2                                13 ifgt 5
   6 iload_0                                16 iload_
   7 iadd                                   17 iretur
```

Schreiben Sie ein Java-Programm, dessen Kompilation möglichst genau obigen Text ergibt.

Aufgabe 3

Welcher Java-Quelltext gehört (wahrscheinlich) hierzu:

```
Method int example2(int, int)
   0 goto 21
   3 iload_0
   4 iload_1
   5 if_icmplt 15
   8 iload_0
   9 iload_1
  10 isub
  11 istore_0
  12 goto 21
```

```
  15 iloa
  16 isto
  17 iloa
  18 isto
  19 iloa
  20 isto
  21 iloa
  22 ifgt
  25 iloa
  26 iret
```

Ausblick (Registermaschinen)

Suchen/Installieren Sie einen Bytecode-Compiler (z. b. `gcj`, siehe `http://gcc.gnu.org/java/`) und kompilieren Sie ein Class-file zu Assembler-Code.

```
gcj --main=Loop Loop.class -S
```

Suchen Sie Ähnlichkeiten zum Original . . .

Unterprogramme (I) (22. 10. 03)

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
- Rückkehradresse
- (bei Funktionen) (Verweis auf) Platz für Resultat

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
```

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

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).

Tail-Rekursionen

Wenn die letzte Aktion eines Unterprogramms P der Aufru
eines Unterprogramms Q ist (eine End-Rekursion),

dann wird nach der Rückkehr aus Q der Frame von P nich
mehr gebraucht.

Deswegen diesen Frame bereits *vor* dem Ruf von Q
aufgeben, d. h. den Frame zum Q-Aufruf in den P-Frame
hineinschreiben!

Aufgabe: untersuche, ob `gcc` End-Rekursionen bei
Prozeduren und Funktionen auflösen kann, und welche
anderen Optimierung bei Prozedur-Aufrufen noch
stattfinden. Beachte Unterschiede zwischen `gcc -Ox` für
$x \in \{0, 1, 2, 3\}$.

Vergleiche mit der Dokumentation `gcc.gnu.org`.

Seminar 24. 10.

- Prozedur-Aufrufe, vergleiche `gcc -S` mit `gcc -S -O`, mit `gcc -S -O3`, bzw. mit Sun-Compiler `cc`
- Inlining, Constant Propragation, Loop Unrolling?
- Tail Recursion ($\rightarrow$ Sprung $\rightarrow$ Schleife)

Nested Functions

```
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 (tre
  void show (int
    { printf ("%d
  walk (& show, t
}

int count (tree *
  int counter = 0
  void one (int n
    { counter ++;
  walk (& one, t)
  return counter;
}
```

Das ist eine C-Erweiterung (implementiert in gcc).

Aufgabe: und in Java? (Helfen *innere Klassen*?)

Statische und Dynamische Ketten

Implementierung von nested functions:

damit Zugriff auf lokale Variablen in umgebenden Blöcken funktioniert, brauchen wir die *statische Aufrufkette* (= Verweise auf textlich umgebende Frames).

(Ein Zeiger auf) eine Funktion ist damit nicht nur eine Adresse, sondern auch ein (Zeiger auf die passende) statische Kette. So ein Paar heißt *closure*.

Unterprogramme (II) (27. 10. 03)

Wiederholung: Statische Ketten

```
int n1 ( int x1 ) {
  int n2 ( int x2 ) {
    int n3 ( int x3 ) {
      return x3 + x2 + x1; } // benuzten
    return n3 (n3 (0)); }    // verlängern
  return n2 (n2 (0)); }      // verlängern
```

Kette verlängern:

```
movl  %esp, %ebp
subl  $24, %esp
leal  8(%ebp), %eax
movl  %eax, -8(%ebp)
movl  %ecx, -4(%ebp)
movl  $0, (%esp)
movl  %ebp, %ecx
call  n3.1
```

statische Kette benutzen:

```
movl -4(%ecx), %edx
movl -8(%ecx), %eax
movl (%eax), %eax
addl 8(%ebp), %eax
movl -4(%edx), %edx
addl (%edx), %eax
```

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 ) {
    int f3 ( int x3 ) {
      return x3 + x2 +
    }
    return twice (f3, 0
  }
  return twice (f2, 0);
}
```

Für `twice(f0,*)` reicht einfacher Zeiger auf `f0`,
für `twice(f2,*)` braucht man closure von `f2` (mit Frame von `f1`).

Lösung in gcc: Trampolins

Entwurfs-Ziele:

- korrekte Lösung, aber kompatibel und ohne Overhead
- nicht aus *allen* Zeigern closures machen.
- nicht zwei Versionen von `twice`.

Falls Funktion g innerhalb von f definiert, und Closure (Zeiger auf) g benötigt, dann:

- erzeuge Programmstück (Trampolin), das FP von f in Register schreibt, und dann zu g springt.
- Am Anfang von g dieses Register in Static Link Field des aktuellen Frames schreiben.
- Zeiger auf Trampolin als Closure benutzen.

- nicht-lokale Zugriffe in g verfolgen statische Kette.

Trampolin-Beispiel

```
int f2 ( int x2 ) {
  int f3 ( int x3 ) { return x3 + x2 + x1;
  return twice (f3, 0);
}
```

```
movl %esp, %ebp
leal 8(%ebp), %eax
subl $56, %esp
leal -40(%ebp), %edx
movl %eax, -16(%ebp)
movl $f3.3+22, %eax
movl %ecx, -12(%ebp)
leal -8(%ebp), %ecx
subl %ecx, %eax
```

```
movl %eax, -34(%ebp
xorl %eax, %eax
movb $-71, -40(%ebp
movl %ecx, -39(%ebp
movb $-23, -35(%ebp
movl %eax, 4(%esp)
movl %edx, (%esp)
call twice
```

Funktionen als Werte

Funktionen als Werte von Argumenten. Beispiel in Pascal, in C.

Funktionen als Rückgabe-Werte. Beispiele? Warum nicht in Pascal?

Beachte: Wenn eine Funktion als Wert zurückgegeben werden soll, braucht man dazu auch ihre statische Kette. Die entsprechende dynamische Kette ist zu dem Zeitpunkt evtl. schon verschwunden!

Funktionen als Werte: Beispiel

In C geht es nicht (ausprobieren!)

```
typedef int fun (int);

fun * plus (int x) {
  int f (int y) {
    return x + y;
  }
  return & f;
}
```

```
int p (int x) {
  fun * g = plus
  fun * h = plus
  return g (h (0)
}
```

In Haskell (`www.haskell.org`) geht es:

```
plus :: Int
  -> ( Int -> Int )
plus x =
  let f :: Int -> Int
      f y = x + y
  in  f
```

```
p :: Int -> Int
p x =
  let f = plus 3
      g = plus 5
  in  f (g (0))
```

Funktionen als Werte

D. h. die Frames verhalten sich nicht LIFO und können nicht auf den Stack.

D. h. extra-Speicherverwaltung ist nötig (Garbage Collection).

Aufgabe: warum braucht man das nicht in Pascal?

In Standard-C: keine nested functions, statische Kette ist immer leer.

Aufgabe: suche in gcc-Beschreibung: was passiert, wenn Zeiger auf innere Funktion benutzt wird, wenn dynamische Kette schon weg ist?

Lokale Klassen in Java (Übung 29. 10.)

Beispiele

Versuchen Sie, die C-Konstruktioen von lokalen Unterprogrammen nach Java zu übersetzen. Benutzen Sie dazu lokale Klassen (inner classes). Beispiel:

```
class Nesting {
    int r (int x) {
        class D {
            int s (int y) { return x + y; }
        }
        D d = new D ();
        return d.s (x);
    }
}
```

Welche Files entstehen bei der Kompilation?

Fragen

Wie wird im erzeugten Code auf Variablen aus umgebenden Frames zugegriffen? Welche Zugriffe sind überhaupt erlaubt (siehe auch folgende Frage)?

Welche Unterschiede gibt es zwischen statischen und dynamischen lokalen Klassen?

Beantworten Sie das Quiz `http://developer.java.sun.com/developer/onlineTraining/new2java/supplements/quizzes/January03.html`

Lexikalische Analyse (5. 11.)

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.

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 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)
- Relation $T \subseteq (Q \times \Sigma) \times Q$

  bzw. Funktion $T : (Q \times \Sigma) \to 2^Q$

autotool:

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

Rechnungen und Sprachen von Automaten

Für $((p, c), q) \in T(A)$ schreiben wir auch $p \xrightarrow{c}_A q$.

Für ein Wort $w = c_1 c_2 \dots c_n$ 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 Sprache von A ist

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

Automaten mit Epsilon-Übergängen

Definition. Ein ϵ-Automat ist . . . 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 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

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

Optimierung: in B alle Zustände streichen, von denen in A *nur* ϵ-Pfeile ausgehen.

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 $s = s_X, f_X = s_Y, f_Y = f$.
  - $s \xrightarrow{X \cup Y} f$ 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. Der so erzeugt Automat A ist korrekt und $|Q(A)| \leq 2|X|$.

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 (wenigstens) eine der anderen Konstuktionen

inkorrekt.

Autotool-Aufgaben

Wir betrachten diese Sprachen:

- `Sub`. Alle Strings über Alphabet $\{a, b\}$, die keinen Teilstring aba enthalten.

- Kommentare (`Com`): beginnen mit ab, enden mit ba, dazwischen beliebige Zeichenfolge ohne ba. Alphabet ist $\{a, b, c, d\}$.

- String-Literale (`String`): beginnen mit q, enden mit q, dazwischen kein q, es sei denn, es ist escaped (bq) Alphabet ist $\{a, b, c, q\}$.

- (`Drei`): Die Anzahl der a in w ist durch 3 teilbar und alle b stehen links von allen c. Alphabet ist $\{a, b, c\}$.

Autotool-Aufgaben (II)

und dazu diese Aufgaben:

- Automaten-Synthese (`Synthese-S-{Sub,Com,String,Drei}`): Finden Sie jeweils einen endlichen Automaten, der die Sprache erzeugt.

- `Analyse-A-{Sub,Com,String,Drei}`: Finden Sie jeweils einen regulären Ausdruck, der die Sprache beschreibt.

`Synthese-S-Quiz`: finden Sie einen endlichen Automaten zu gegebenem regulären Ausdruck.

Lexikalische Analyse (10. 11.)

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 aktzeptierender 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.

Deteministische 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)
- $((p', c), q') \in T' \iff q' = \{q \mid \exists p \in p' : p \xrightarrow{c}_A q\}$
- $S' = \{S\}$
- $F' = \{q' \mid q' \in Q' \wedge q' \cap F \neq \emptyset\}$

Minimierung von det. Aut. (I)

Idee: Zustände zusammenlegen, die „das gleiche“ tun. Das „gleich“ muß man aber passend definieren: benutzen Folgen 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 $\leq k$ *beobachtbar* gleich:
$\forall w \in \Sigma^{\leq k} : w \in L(A, p) \leftrightarrow w \in L(A, q)$.

äquivalent ist induktive Definition:

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

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 $L(A') = L(A)$.

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 $1 \leq i \leq n$: 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

Übungen Reguläre Ausdrücke (12. + 14. 11.)

Determinieren, Minimieren

- `Determine-D-Quiz`:
  - gegeben: (nichtdeterministischer) Automat A
  - gesucht: deterministischer Automat A' mit $L(A) = L(A')$.

- `Equiv-E-Quiz`:
  - gegeben: deterministischer Automat A
  - gesucht: ein Liste $[l_0, l_1, ..l_n]$, wobei l_k die Liste aller Tripel $(p, q, c) \in Q \times Q \times \Sigma$ ist, für die $p \sim_k q$, aber $T(p, c) \not\sim_k T(q, c)$. Die letzte Unter-Liste l_n soll leer (= []) sein.

Reguläre Ausdrücke in egrep

Siehe `man egrep`, `man regexp`.

Beispiel: `egrep '^[^ ]*$' foo.text`

Aufgaben:

- Wie werden die aus der Vorlesung (bzw. autotool) bekannten Konstruktoren für reguläre Ausdrück hingeschrieben?

- Was bedeuten
  - `[a-zA-Z0-1]`
  - `[^0-9]`

- finden Sie in einem File alle Zeilen, die wenigstens einen Kleinbuchstaben enthalten.

- enthält die manpage von egrep eine Zeile, die mit einem Kleinbuchstaben beginnt? (`man egrep | egrep '...'`)

- Matchen Sie alle Zeilen, die insgesamt wenigstens 3 Ziffern enthalten.

Reguäre Ausdrücke in sed

(sed = stream editor)

Aufgaben:

- Suchen und ersetzen, Beispiel:
  `echo foo | sed -e 's/o/O/g'` Was bewirkt das abschließende g?

- Schreiben Sie einen sed-Aufruf, der alle Zeilen-Kommentare (ab `//`) löscht. Hinweis: benutzen Sie nicht `/`, sondern z. b. + als sed-Trennzeichen.

- Bezugnahme auf frühere Matches. Was bedeutet das?
  `sed -e 's/\([a-z]*\)/\1\1/g'`

Reguläre Ausdrücke in der Bash

(bash = “bourne again” sh)

Bei Filenamen: `ls foo/*x*/*.{tex,ps}`

PS: Ansonsten nützlich:

- Zeilen-Editor (probieren Sie Cursor-rechts, Cursor-links, C-f, C-b, C-a, C-e, C-w, C-k C-y)
- Command-History (Cursor-hoch, Cursor-runter, C-p, C-n)
- inkrementelle Suche in Command-History (C-r und dann Suchstring)
- Prompt-Symbol (diese Zeile in .bashrc eintragen:)

```
export PS1="\u@\h:\w\$ "
```

Reguläre Ausdrücke in Emacs

`M-x query-replace-regexp` (Syntax wie in sed)

PS: Sonstiger Spaß mit Emacs:

- `C-x r k` (kill rectangle), `C-x r y` (yank rectangle)
- `M-x hanoi`
- `M-x gomoku` (5 in einer Reihe)
- `M-x dunnet` (Abenteuer! - Nicht trivial!)

Übung Scanner-Generator flex (21. 11.)

Übung zu flex (21. 11. 03)

Finden Sie `flex`? Falls nicht, dann (bash-Syntax)

```
export PATH=$PATH:/usr/local/share/bin
export MANPATH=$MANPATH:/usr/local/share/ma
```

flex benutzen

Ein Scanner, der Folgen von ab erkennen (und zählen) sol
(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 'abaababbbababba' | ./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/ws03/compilerbau/programme/scanner/java.l`

Hinweis: dieser Scanner wird später nochmal verwendet

Syntaktische Analyse (24. 11.)

Grammatiken

Wort-Ersetzungs-Systeme

Berechnungs-Modell (Markov-Algorithmen)

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

Regelmenge $R \subseteq \Sigma^* \times \Sigma^*$

Regel-Anwendung:

$u \to_R v \iff \exists x, z \in \Sigma^*, (l, r) \in R : u = x \cdot l \cdot z \wedge x \cdot r \cdot z = v$

Beispiel: Bubble-Sort: $\{ba \to ab, ca \to ac, cb \to bc\}$

Beispiel: Potenzieren: $ab \to bba$

Aufgaben: gibt es unendlich lange Rechnungen für:
$R_1 = \{1000 \to 0001110\}, R_2 = \{aabb \to bbbaaa\}$?

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"
  , nichtterminale
      = mkSet "SA"
  , startsymbol = 'S'
  , regeln = mkSet
      [ ("S", "abc")
      , ("ab", "aabb
      , ("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)

Theorie der Formalen Sprachen

untersucht (Hierarchien von) Sprachfamilien:

- äquivalente Definitionen für Familien (Grammatiken, Automaten, Ausdrücke)
- (echte) Inklusionen (ist jede Typ-2-Sprache eine Typ-1-Sprache? gibt es Typ-2-Sprachen, die keine Typ-3-Sprache sind?)
- Abschluß der Familien bzgl. Operationen (ist der Durchschnitt von zwei Typ-3-Sprachen wieder Typ-3? Desgl. für Typ-2?) (Falls ja, wie konstruiert man ihn effektiv?)
- Entscheidbarkeit/Komplexität (z. b. des Wort-Problems)

Theorie (II)

$i \geq j \Rightarrow$ Jede Typ-i-Sprache ist eine Typ-j-Sprache.

$i > j$: es gibt L in Typ-$j \setminus$ Typ-i:

$\{a^n b^n \mid n \geq 0\} \in$ Typ-$2 \setminus$ Typ-3.

$\{a^n b^n c^n \mid n \geq 0\} \in$ Typ-$1 \setminus$ Typ-2.

Wenn $A, B \in \text{Typ-}3$, dann $A \cup B \in \text{Typ-}3$ (trivial) und $A \cap B \in \text{Typ-}3$ (Aufgabe — benutze Automaten).

Wenn $A, B \in \text{Typ-}2$, dann $A \cup B \in \text{Typ-}2$ (trivial — Aufgabe).

Es gibt $A, B \in \text{Typ-}2$ mit $A \cap B \notin \text{Typ-}2$.

Typ-3-Grammatiken und Reguläre Sprachen

Def (Wdhlg): L heißt regulär $\iff$ existiert regulärer Ausdruck X mit $L = L(X)$.

Satz (Wdhlg): L regulär $\iff$ existiert endlicher Automat A mit $L = L(A)$. (Beweis: Analyse/Synthese)

Satz: L regulär $\iff$ existiert Typ-3-Grammatik G mit $L = L(G)$. Beweis (trivial):

- Zustände von $A =$ Variablen von G,
- Startzustand von $A =$ Startsymbol von G
- jede Regel $p \xrightarrow{a} q$ in $A =$ Regel $p \to aq$ in G

Sternhöhe

Def: die Sternhöhe $\mathrm{SH}(X)$ eines regulären Ausdrucks X ist die maximale Schachteltiefe der Sterne.

Bsp: $\mathrm{SH}((a^*b)^*) = 2$.

Def: die Sternhöhe $\mathrm{SH}(L)$ einer regulären Sprache L ist $\min\{\mathrm{SH}(X) \mid L(X) = L\}$.

Bsp: $\mathrm{SH}((a^*b)^*) = 1$, denn $(a^*b)^* = \epsilon \cup \Sigma^*b$

Erweiterte Sternhöhe

Def: in *erweiterten* regulären Ausdrücken ist als zusätzlicher Operator *Komplement* bzgl. Σ^* und *Mengendifferenz* gestattet.

Def: erweiterte Sternhöhe $\mathrm{ESH}(X)$ eines Ausdrucks wie oben als maximale Schachteltiefe, erweiterte Sternhöhe $\mathrm{ESH}(L)$ einer Sprache wie oben als kleinste $\mathrm{ESH}(X)$ mit $L(X) = L$.

Bsp: $\mathrm{ESH}((ab)^*) = 0$, denn $(ab)^* = \epsilon \cup a\Sigma^*b \setminus \Sigma^*(aa \cup bb)\Sigma^*$

beachte: $\mathrm{ESH}(\Sigma^*) = 0$ wegen Σ^* = Komplement von $\emptyset$.

Vermutung: für jede reguläre Sprache L gilt $\mathrm{ESH}(L) \leq 1$.

Kontextfreie Grammatiken

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 Anwei
Ausdruck -> Bezeichner | Literal
     | Ausdruck Operator Ausdruck
```

Bsp: korrekt geklammerte Ausdrücke:
$G = (\{a, b\}, \{S\}, S, \{S \to aSbS, S \to \epsilon)$.

Bsp: Palindrome:
$G = (\{a, b\}, \{S\}, S, \{S \to aSa, S \to bSb, S \to \epsilon)$.

Bsp: alle Wörter w über $\Sigma = \{a, b\}$ mit $|w|_a = |w|_b$

. . . und ähnlich Aufgaben, siehe demnächst autotool.

Ableitungsbäume für CF-Sprachen

Def: ein geordneter Baum T mit Markierung $m : T \to \Sigma \cup \{\epsilon\} \cup V$ 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.)

Eindeutigkeit (II)

(äquiv. Definition ohne Bäume)

Def: ein Schritt (Regel-Anwendung) $u = xlz \to xrz = v$ heißt *Links-Schritt*, geschrieben $u \to_L v$, falls $x \in \Sigma^*$

(d. h. l ist die am weitesten links stehende Variable).

Eine *Links-Ableitung* ist eine Folge von Links-Schritten.

Satz: G ist eindeutig $\iff$ jedes w besitzt genau eine Links-Ableitung.

Beweis: betrachte Pre-Order-Durchquerung des Abl.-Baums.

Dangling else

In vielen Programmiersprachen ist definiert:

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

Modell: $\{S \rightarrow e|tS|tSeS\}$.

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.

Seminar 26./28. 11.

Aufgaben zu Grammatiken

Sprachen (und ihre Komplemente `Com*`):

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

  $\{w \mid d(w) = 0 \wedge \forall u \sqsubseteq w : d(u) \geq 0\}$,

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}`

Syntakt. Analyse (4. 12.)

Normalformen von CFG

Erreichbare und produktive Variablen

Für eine CFG G heißt die Variable A

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

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 $A \rightarrow_R^* \epsilon$.

Die Menge der nullierbaren Variablen von G
ist die kleinste Menge $N \subseteq V$ mit:

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

Bemerkung: der erste Fall ist tatsächlich im zweiten enthalten.

Def: eine Regel $A \rightarrow r$ heißt nullierbar, falls $r \rightarrow_R^* \epsilon$.

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)?

Top-Down-Parser (von Hand)

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_Blo
      "if"    -> parse_Ausdruck ; parse_Blo
      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/gn`
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/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, Beispie
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 $\exists w \in (V \cup \Sigma)^* : A \rightarrow_R^+ Aw$.

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

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

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?

Programmier-Übung (5. 12.)

Programmier-Übung Top-Down-Parser

Arithmetische Ausdrücke auswerten.

Syntax: LISP. `(* (+ 3 8) (+ 2 31))`

Quelltexte hier:
`http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/parser/`

Aufgaben:

- ändere Parser für diese Syntax:
  `m (p (3,8), p(2, 31))`

  (aus `+` wird `p`, aus `*` wird `m`)

- erweitere das auf p und m mit beliebig vielen Argumenten

Operator-Präzedenz-Parser

Operator-Präzedenz-Parser

```
Ausdruck -> Literal | ( Ausdruck )
    | Ausdruck + Ausdruck
    | Ausdruck * Ausdruck | ...
```

Eingabe: $A_1 o_1 A_2 o_2 A_3 o_3 A_4$

können nicht sofort ausrechnen, Benutzen Keller für Zwischenresultate.

Entscheide zw. Push (warten) und Pop (rechnen) anhand der Präzedenz der Operatoren (müssen mit gekellert werden).

Operator-Grammatiken

Def: eine CFG heißt *Operator-Grammatik*, falls für jede Regel $(l \to r) \in R$ gilt:

- $r \neq \epsilon$
- r enthält keine direkt benachbarten Variablen

Wir benutzen *Prioritäts-Relationen* $<$ auf Terminalen.

Für Wort $cw\$$ (mit Links- und Rechtsmarkierung):

Füge Relationen zwischen Terminalen ein (übebrspringe dabei Variablen): z. B. $c < a_1 < a_2 > \ldots a_n > \$$

Suche am weitesten links stehendes $>$, dann das nächste davon links stehende $<$, wende Regel auf Teilwort $< \ldots >$ an.

Implementierung mit Stack

Konstruiere Ableitungsbaum zu geg. Operator-Grammatik mit Präzedenz

Interface Stack: `push(), pop(), top`, Interface Eingabe (Scanner): `next()`

```
push (’c’); -- linke Rand-Markierung
Token n = next ();
while ((’c’ != top) || (’$’ != n)) {
  if ( top < n ) {
    push (n); n = next ();
  } else {
    while ( top > n ) { pop (); }
  }
}
```

Bemerkungen

stark vereinfacht. Es fehlen:

- Syntaxbaum bauen/semantisch Aktion ausführen
- Assoziativität (Links, Rechts, keine)
- Fehlerbehandlung

Aufgaben (Praktikum):

- Welche Präzedenz entspricht der üblichen Arithmetik (Operatoren $+, -, *, /, \uparrow$)?
- Welche Präzedenz bekommen Klammern?
- (Kombination mit Parser mit rekursivem Abstieg, für Funktions-Aufrufe in Ausdrücken?)

Übung (10./12. 12. 03) Op-Präzedenz-Parser

Dateien hier:
`http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/parser/`
kompilieren mit: `gmake`, erzeugt Echse: `operator`
ausführen: `echo "1 + 2 * 3 + 4" | ./operator`

- semantische Aktionen (Rechnungen) hinzufügen.

  Dazu `stack.[ch]` erweitern (für Zwischenergebnisse).

  (Neuer Datenbereich `int valstack []`,
  aber alter Stack-Pointer.)

- Klammern, Bsp `3 - 4 * (5 + 6)`

- Fehlererkennung, Bsp `3 + 5 * / 6`

- Assoziativität, Bsp `10 - 8 - 2`, `2 ^ 3 ^ 2`

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            ::
        FiniteMap (Maybe x, z, y) (Set (z,
          }
```

Übergangsrelation $(w, z, k) \rightarrow_A (w', z', u'k')$, 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 $(z', u') \in T(\epsilon, z, y)$

Sprachen von Keller-Automaten

Die durch leeren Keller akzeptierte Sprache:

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

Die durch Endzustandsmenge F akzeptierte Sprache:

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

(Beachte in beiden Fällen: ϵ-Übergänge 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 $B \to w =$ Übergang $(w, z_0, Bk') \to (w, z_0, wk')$
- jedes Terminalzeichen $x \in \Sigma$: Übergang
  $(xw', z_0, xk') \to (w', z_0, k')$.

Keller-Automaten als Parser

Determinismus!

Übung (10./12. 12. 03)
autotool/Kellerautomaten

Aufgaben

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

bitte heute diese Adresse benutzen:

`http://theo1.informatik.uni-leipzig.de/~joe/cgi-bin/Face.cgi` matrikel: 318, wort: guest

Beispiel:

```
NPDA { eingabealphabet = mkSet "ab"
     , kelleralphabet = mkSet "XA"
     , zustandsmenge = mkSet [ 0, 1, 2]
     , startzustand = 0 , startsymbol = 'X'
     , akzeptiert = Leerer_Keller
     -- ODER: , akzeptiert = Zustand ( mkSe
```

```
    , tafel = listToFM
            [ ( ( Just 'a', 0 , 'X'), mkSe
            , ( ( Just 'a', 0 , 'A'), mkSe
            , ( ( Just 'b', 0 , 'A'), mkSe
            , ( ( Just 'b', 2 , 'A'), mkSe
            ]
    }
```

Top-Down/Bottom-Up (Wiederholung)

Top-Down/Bottom-Up

Parsen durch rekursiver Abstieg ist top-down-Methode, erzeugt Links-Ableitung.

Operator-Präzedenz-Parsen ist bottom-up-Methode, erzeugt Rechts-Ableitung. (Beispiel!)

Beide Methoden lesen die Eingabe *von links*!

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.

Shift und Reduce

LR-Parser: deterministischer endlicher Automat mit Keller.

Kellerinhalt: $s_0 X_1 s_1 X_2 \ldots X_m s_m$

(abwechselnd Zustand und Zeichen)

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

repräsentiert (rechts-)abgeleitete Satzform

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

action(s_m, a_i) kann sein:

- shift s: zu $(\ldots X_m s_m a_i s, a_{i+1} \ldots)$
- reduce $A \to b$: zu $(\ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots)$,

  wobei $|b| = r$ und $s =$ jump(s_m, a_i).

Synt. Analyse mit Tools

Vorlesung

Parser-Generator bison

typischen Kombination:

- lexikalische Analyse (Scanner) mit lex (flex)
- syntaktische Analyse (Parser) mit yacc (bison)

`parser.y` enthält erweiterte Grammatik (Regeln mit semantischen Aktionen).

`bison -d` erzeugt daraus:

- `parser.tab.c` (Parser)
- `parser.tab.h` (Token-Definitionen, für `scanner.l`)

Beispiel:
`http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/taschenrechner/`

Flex-Bison-Beispiel

parser.y:

```
%token NUM
%left '-' '+'
%%
input : | input line ;
line  : exp '\n'
  { printf ("%d\n", $1); }
exp   : NUM  { $$ = $1; }
   | exp '+' exp
      { $$ = $1 + $3; }
   | exp '-' exp
      { $$ = $1 - $3; }
   ;
%%
```

scanner.l:

```
%{
#include "parser.tab.h
%}
%%
[0-9]+ {
  yylval = atoi (yytex
  return NUM; }
[-+*/^()] {
  return yytext[0]; }
```

interpreter.c:

```
int main (int argc, char ** argv) { yyparse (); }
```

Yacc-Grammatiken

Definitionen %% Regeln %% Hilfs-Funktionen

Definitionen: Namen von Token (auch Präzedenz, Assoziativität)

Regeln: Variable `:` (Variable)* Aktion `| ... ;`

zu jeder Variablen gibt es semantischen Wert,

Bezeichnung $$ (links), `$1, $2, ..` (rechts).

Fehlerbehandlung

Fehler: keine action(s_m, a_i) anwendbar.

Ausgabe:

```
%%
int yyerror (char * s) {
        fprintf (stderr, "%s\n", s);
        return 0;
}
```

Reparatur: Regeln mit eingebautem Token `error`

```
 stmnts:  /* empty string */
               | stmnts '\n'
               | stmnts exp '\n'
               | stmnts error '\n'
```

Symbole und -Tabellen

```
typedef double (*func_t) (double);
typedef struct {
  char *name;  /* name of symbol */
  int type;    /* type: either VAR or FNCT
  union
  { double var;         /* value of a VAR   *
    func_t fnctptr;   /* value of a FNCT  *
  } value;
  struct symrec *next;    /* link field */
} symrec;
extern symrec * sym_table;
```

Scanner muß neue Symbole eintragen und alte wiederfinden (getsym/putsym).

Bessere Implementierung durch Hashing.

Unions in semantischen Werten

```
%union {
double      val;  /* Zahlen */
symrec  *tptr;    /* Symboltabellen-Eintrag
}
%token <val>  NUM
%token <tptr> VAR FNCT
%type  <val>  exp
%%
exp: NUM                { $$ = $1; }
 | VAR                { $$ = $1->value.var; }
 | VAR '=' exp { $$ = $3; $1->value.var = $
 | FNCT '(' exp ')'
          { $$ = (*($1->value.fnctptr))($3);
 | exp '+' exp { $$ = $1 + $3; }
```

Übung

Übung zu Bison

- `gcc` benutzt bison-Grammatik, siehe `http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/gcc-3.3.2/gcc/`
- Taschenrechner-Dateien kopieren von `http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/taschenrechner/`, `gmake`, testen: `./interpreter`, dann zeilenweise Eingabe von Ausdrücken, z. B. `1 + 2 * 3 - 4`

Bison-Übung (II)

- Beispiel-Parser untersuchen:
  - Kellerautomaten betrachten: `bison -v parser.y` → `parser.output`
  - Lauf der Automaten betrachten: in `interpreter.c`: `yydebug = 1;`
- Beispiel-Parser erweitern:
  - Operator `%` (Rest bei Division)
  - einstellige Funktion `quad` (Quadrieren)
    - neues Token `QUAD` in `parser.y`,
    - neue Zeile in `scanner.l`,
    - neue Regel in `parser.y`

Autotool-Lösungen

Autotool-Lösungen

A-Com: `ab (a + b^* (c + d))^* b^* ba`

A-Drei:

```
(      a b^*  a b^* a + b)^*
(Eps + a b^* (a b^* + c^* a ) c^* a)
(                    a c^* a   c^* a  + c)^*
```

A-String: `q (b Sigma + (a+c))^* q`

A-Sub: `b^* (a + bb b^*)^* b^*`

Havannah

Gewinne durch

- *einen Kreis* (mit Inhalt)
- oder Verbindung *zweier* Eckpunkte
- oder *dreier* Seiten.

`http://141.57.11.163/havannah/different-applet/`
(gegen (dummes) Programm: Port 1968 eintragen)

Symboltabellen/Hashing (5. 1.)

Die (sogenannte) Symboltabelle

(eigentlich ist es die Tabelle für *Bezeichner*)

(Wdhlg.) Tokenklassen:

- Syntaxzeichen (Klammern usw.),
- Literale (für Zahlen, Strings, usw.),
- Schlüsselwörter (if, then, usw.)
- *Bezeichner* (für Variablen, Unterprogramme, Typen, Module usw.)

(Was ist mit Operatoren?)

Inhalt der Symboltabelle

wichtige Attribute der Bezeicher aufbewahren:

- Variable: Typ
- Unterprogramm: Argument- und Ergebnistyp
- Typ: Supertypen (OO) usw.
- (alles:) Quelltextposition

Symboltabellen als Abbildungen

Abstrakter Datentyp *Map* (Abbildung)

- von Menge A (hier: String)
- nach Menge B (hier: Bezeichner-Information)

```
interface Map(A, B) {
    void clear ();
    void put (A key, B value);
    B get (A key);
    void remove (A key);
}
```

(anderer Name: *Wörterbuch* (Dictionary) — warum ist das inzwischen “deprecated”?)

Abbildungen in Java

`Map` ist ein *Typkonstruktor*

(= Typ höherer Ordnung, wenn A und B Typen sind, dann ist `Map(A,B)` ein Typ),

aber man darf das in Java nicht hinschreiben. Deswegen (`http://java.sun.com/j2se/1.4.2/docs/api/java/util/Map.html`):

```
interface Map {
    void clear ();
    void put (Object key, Object value);
    Object get (Object key);
    void remove (Object key);
}
```

und man muß fleißig casten:

```
Entry e = (Entry) table.get ("foo");
```

Das Fehlen von Typen höherer Ordnung

. . . ist eine schwerwiegende Lücke in vielen gängigen Sprachen (die besonders beim Umgang mit Collections, Containern usw. schmerzt).

Mehr oder wenig zaghafte Versuche, um diese Lücke herumzuturnen, heißen dann Casts, Templates (C++, Java 1.5), Design Patterns usw.

und erfordern dann eigene Vorlesungen, Lehrbücher, Hilfsprogramme (sog. CASE-Tools), Zaubersprüche und Gurus . . .

(. . . und schaffen auf diese Weise immerhin Umsatz und Arbeitsplätze).

Abbildungen: mögliche Implementierungen

- ungeordnete Liste/Array
- alphabetisch geordnete Liste/Array
- unbalancierter Suchbaum
- balancierter Suchbaum (z. B. AVL, 2-3, Rot-Schwarz)

Aufgabe (Wdhlg 1. Semester, hoffe ich): diskutiere (für alle 6 Varianten) die Laufzeiten für die o. g. Operationen.

Beispiel: ungeordnete Liste:

- `put` ist nur eine Operation, also $\Theta(1)$, konstant
- aber `get` muß alle Einträge betrachten, also $\Theta(n)$, linear

Hashing

benutze schnelle Hashfunktion $h : \Sigma^* \to \{0 \ldots m-1\}$

Idee: Speichere x in $t[h(x)]$.

Parameter (Tabellengröße, Hashfunktion) geeignet wählen
dann “praktisch konstante” Laufzeit für alle Operationen.

Hash-Kollision: $x \neq y$ und $h(x) = h(y)$.

x ist schon in $t[h(x)]$, wo soll y hin?

Kollisionen behandeln:

- außerhalb der Tabelle: $t[i] =$ *Liste aller* x mit $h(x) = i$.

  Nachteil: Extraplatz für Listenzeiger

- innerhalb der Tabelle:
  - speichere y in $t[h(y) + 1]$ oder$t[h(y) + 2]$ oder ...
    Nachteil: Tabelle „verklebt“
    (belegte Blöcke erzeugen weitere Kollisionen)
  - doppeltes Hashing: benutze h_1, h_2 und benutze Indizes
    $h_1(y), h_1(y) + h_2(y), h_1(y) + 2h_2(y), \ldots$.
    Vorteil: kein Verkleben (Schrittweiten sind verschieden!)

Aufgabe: Pseudocode für Einfügen und Suchen bei doppeltem Hashing. *was ist mit Löschen?*

Re-Hashing

Problem:

- Tabelle zu klein $\rightarrow$ zu voll $\rightarrow$ viele Kollisionen $\rightarrow$ langsam.

- Tabelle zu groß: Platz verschenkt.

Lösung: falls Tabelle gewissen Füllstand erreicht, dann zu neuer, größerer Tabelle wechseln ($=$ re-Hashing).

am einfachsten: Tabellengröße ist immer Potenz von 2; dann: vergrößern $=$ verdoppeln.

Beim re-Hashing müssen alle Einträge betrachtet werden, das findet aber nur selten statt, so daß die amortisierte Laufzeit trotzdem konstant ist (Aufgabe: nachrechnen).

Bastelstunde: Hashing (7./9. 1.

Programmieraufgaben

n

Hashing in gcc:
`http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/gcc-3.3.2/gcc/`

Ein Scanner soll alle Bezeichner (der Form `Buchstabe (Buchstabe + Ziffer)^*`) des Eingabestroms in die Symboltabelle eintragen (und alle anderen Zeichen ignorieren), und schließlich alle Bezeichner ausgeben, die öfter als 5 mal vorkamen. (Die genaue Anzahl ist mit auszugeben.)

- Quellen kopieren (`http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/hash/`), kompilieren

(`gmake`), ausprobieren (`cat README ./scanner`)

- Betrachten Sie den Verlauf des Hashing in einem Debugger

```
$ gdb scanner
(gdb) break ht_lookup
(gdb) run < README
```

weitere gdb-Kommandos:

- n: next (nächste Zeile, Unterprogramme nicht anzeigen)
- s: step (nächste Zeile, in Unterprogramme verzweigen)
- c: continue (bis zum nächsten breakpoint)
- p: print (Ausdruck auswerten, z. b. `print hash2`, in Ausdrücken sind auch Funktionsaufrufe möglich)

Falls gdb nicht funkioniert:

```
$ export PATH=/home/waldmann/built/bin:$PA
$ export LD_LIBRARY_PATH=/home/waldmann/bu
$ export MANPATH=/home/waldmann/built/man:
```

(das ist bash-Syntax. für (t)csh `setenv` benutzen)

- Wie lautet der Hashwert von “AB” (von Hand ausrechnen, mit Debugger prüfen und Ergebnis interpretieren)

```
$ gdb scanner
(gdb) break main
(gdb) run
(gdb) p calc_hash ("AB", 2)
```

- Warum wird im `ht_identifier` auch der Hash-Wert gespeichert? (Wann wird dieser Wert gelesen?)

- Laut `hashtable.[ch]` enthalten die `hashnodes` (im wesentlichen) nur einen String.

  Wo stehen die Zusatzinformationen (gcc: Knoten des Syntaxbaums, unser Scanner: Anzahl der Vorkommen)?

  Warum muß man in C so programmieren (welches Ausdrucksmittel fehlt)?

- Wann (bei welchem Füllstand) wird die Hashtabelle vergrößert? Ändern Sie die enstprechenden Parameter, so daß mehr gefüllt wird, und beobachten Sie den Effekt (auf mittlere Suchlängen).

- Geben Sie drei Strings mit gleichem Hash-Wert an.

Fügen Sie diese der Reihe nach in die Hashtabelle ein. Welche Kollisionen gibt es?

- Vereinfachen Sie die zweite Hashfunktion (`hash2` in `hashtable.c`, Vorsicht, kommt mehrfach vor) zu `hash2 = 1;`, vergleichen Sie die Resultate, erklären Sie.

- Ist es sinnvoll, daß `hash2` aus `hash` berechnet wird? Schlagen Sie eine bessere Variante vor und testen Sie diese.

Typsysteme (5. 1.)

Typ-Prüfungen

Typ: eine Menge von Daten und zugeörigen Operationen.

Typ-Prüfung: Test, ob gegebenes Datum zu gewünschter Menge gehört.

- Typ-Deklarationen sind hervorragende Dokumenationen (warum?)

Typfehler sind Vorboten von Laufzeitfehlern.

Statische Typ-Prüfung: jedes Objekt, jeder Bezeichner hat genau einen Typ. Dieser ist bereits zur Kompile-Zeit bekannt.

Vorteile:

- frühestmögliche Fehlererkennung
- keine Laufzeit-Typtests/information nötig

Arten von Typen

Einfache Typen:

- Aufzählungstypen (Boolean, Char, Int, selbst definierte)
- andere Zahltypen (Float, Double)

Zusammengesetze Typen:

- Produkte,
- Summen,
- Exponentiale.

zusammengesetzte Typen: Produkte

- unbenannt: Tupel `type T = ( T1, T2 )`

  entspricht $T = T_1 \times T_2$

- benannt: Records, Structs, Objekte,
  `...data T = C { f1 :: T1, f2 :: T2 }`

  ist isomorph zu $T = T_1 \times T_2$

  Konstruktor: $C :: (T_1 \times T_2) \to T$

  Destruktoren: $f_i :: T \to T_i$

zusammengesetzte Typen: Summen

- unbenannt (z. B. `union` in C) $T = T_1 \cup T_2$ (gefährlich)
- benannt (z. B. `record .. case .. of ..` in Pascal)

  (Aufzählungstypen sind Spezialfall!)

  ```
  data T = C1 { ... } | C2 { ... }
  ```

  ermöglicht Fallunterscheidungen:

  ```
  case (x :: T) of C1 {} -> ... ; C2 {} -> ...
  ```

  (enthält if/then/else als Spezialfall!)

Aufgabe: warum gibt es keine Summen in Java? Was soll stattdessen benutzt werden?

zusammengesetzte Typen: Exponentiale

```
type T = T1 -> T2
```

entspricht $T = T_2^{T_1} = \{f \mid f : T_1 \to T_2\}$.

Wesentliche Regel:

wenn $f :: T_1 \to T_2$ und $x :: T_1$, dann $f(x) :: T_2$.

(vergleiche Aussagenlogik: aus $p_1 \to p_2$ und p_1 folgt p_2.)

In vielen Sprachen gibt es Exponential-Typen nur versteckt in Funktionsdeklarationen, d. h. statt

```
(T1 -> T2) f; // Deklaration,
f = \ x -> { return 2 * x + 7; } // Zuweisu
```

schreibt man

```
T2 f (T1 x) { return 2 * x + 7; }
```

Ganz ganz einfaches Typprüfen

Wenn die Sprache nur wenige Typen hat (Beispiel: int und boolean), dann kann man die Typprüfung in die Grammatik verlegen:

statt `Ausdruck -> ...` benutze

```
Int_Ausdruck -> ...
Bool_Ausdruck -> ...

Verzweigung -> if Bool_Ausdruck then Anweis
```

Array-Typen

Arrays sind auch (Implementierungen von) Funktionen!

`char [] s;` bedeutet:

s ist partielle Funktion von `int` nach `char`.

In den meisten Sprachen können Arrays auch

- als Argumente und Resultate von Funktionen
- als Elemente von Arrays

auftreten. Das erfordert spezielle Behandlung beim Typprüfen.

Zeiger-Typen

in einigen Sprachen gibt es den Typ-Konstruktor „Zeiger auf“

(in C und Pascal. In Java nicht, bzw. Objekt-Zeiger sind implizit.)

- Programierung: Zeiger erlauben Sharing von Speicherbereichen

- Implementierung: Zeiger brauchen (viel) weniger Platz als das Objekt, auf das sie verweisen

Mengentheoretisch: Zeigertyp `^t` ist benannte Vereinigung von `null` („leerer Zeiger“) und `t`.

Typ-Ausdrücke, Abstrakte Interpretation

Typen werden im Compiler durch Typ-Ausdrücke (Bäume) repräsentiert. (Beispiele.)

Bei der Kompilation von Ausdrücken der Quellsprache wird für jeden Teilausdruck sein Typ(-Ausdruck) berechnet.

Das ist eine Art von *abstrakter Interpretation*.

- konkrete Interpretation (auf Daten): berechne *Wert*.

```
4 + strlen ("foo") -> 4 + 3 -> 7
```

- abstrakte Interpretation (auf Typen): berechne *Typ*

```
4 + strlen ("foo") -> int + strlen (char *
     -> int + int -> int
```

Typ-Regeln

- Typen von Literalen sind bekannt
- Typen für Bezeichner ((lokale) Variablen, formale Parameter) sind vom Programmierer deklariert
- für zusammengesetzte Ausdrücke der Form $A = f(A_1, \ldots, A_n)$
  - die Typen der Argument-Ausdrücke sind schon (rekursiv) berechnet: $A_1 :: T_1, \ldots, A_n :: T_n$
  - der Typ der Funktion ist bekannt: $f :: (S_1 \times \ldots \times S_n) \rightarrow S$
  - die Argument-Typen müssen passen: $T_1 = S_1, \ldots$
  - dann ist Typ des Ausdrucks $A :: S$.

  entspr. für Operator-Ausdr. sowie Array- und Zeiger-Zugriffe.

Vergleich von Typ-Ausdrücken

oft gibt es Namen für Typen:

```
typedef int * T1;
T1 x1;
typedef int * T2;
T2 x2;
```

sind jetzt T_1 und T_2 gleich?

- strukturelle Gleichheit: ja
- Namesgleichheit: nein

strukturelle Gleichheit erlaubt (ein wenig) Polymorphie

Namensgleichheit erlaubt stärkere Kontrolle (bessere Fehlererkennung)

Rekursive Datentypen

```
struct T {
    int contents;
    struct T * next;
}
```

entspricht Gleichung

$T = \texttt{int} \times (\texttt{null} \cup T)$

vergleiche diese Gleichungen für Funktionen:

```
int f (int x) {
 if ( x < 100 )
  return x - 10;
 else
  return
   f (f (x + 11));}
```

```
int t (int x, int y, int
  if (x <= y) { return y;
  else return
    t ( t ( x-1, y, z)
      , t ( y-1, z, x)
      , t ( z-1, x, y) );
```

Gleichungen für Zahlen:

```
int y = y + 4;
```

```
double x = 2 + 3 * x / 4;
```

eine (rekursive) Gleichung (für Typen, Funktionen, Zahlen) kann keine, eine oder mehrere Lösungen haben.

Repräsentation rekursiver Typen

```
struct T {
    int contents;
    struct T * next;
}
```

$$T = \texttt{int} \times (\texttt{null} \cup T)$$

kann im Kompiler so dargestellt werden:

- strukturell: (Rückwärts-)Zeiger im Typ-Ausdruck

  (Vorsicht: Typvergleich muß trotzdem immer terminieren)

- benannt: (eigener) Typname im Typ-Ausdruck

Sprache C benutzt (u. a. deswegen)

- Namensgleichheit für structs,

- und strukturelle Gleichheit sonst.

Typprüfung bei Zuweisungen

Die übliche Notation `a := b;` ist irreführend, denn in Zuweisung $l := r$ bezeichnet:

- r einen Ausdruck mit Typ T
- l eine *Adresse* (einen Zeiger) auf T

es ist üblich, auch l als Ausdruck vom Typ T zu schreiben.

Bei Ausführung der Zuweisung (im kompilierten Code) wird von l

- nicht der *Wert* (rvalue),
- sondern der *Adress-Wert* (lvalue) bestimmt.

Nicht jeder Ausdruck $l :: T$ besitzt einen Adress-Wert.

Implizite Typumwandlungen (Erweiterungen)

Für einen (Teil-Ausdruck) A wurde Typ T berechnet,
aber Typ $S \neq T$ ist verlangt:

kann gestattet werden, falls $T \subseteq S$,

oder falls es eine (durch Sprachdefinition festgelegte)
Abbildung (Einbettung) $T \to S$ gibt
(Code dafür wird dann vom Kompiler eingebaut)

C: int $\to$ long, int $\to$ float, char $\leftrightarrow$ int

Java: von Object $\to$ String

ist Fehlerquelle (versteckter Code)

Aufgabe (Wdhlg): an welchen Stellen können solche Umwandlungen vorkommen? (D. h.: wann wird Gleichheit von Typen geprüft?)

Explizite Typumwandlungen (Zuschnitte)

C, Java: casts, coercions, z. B. von Typ Object nach eigener Klasse

durch `T x; S y = (S)x;` behauptet der Programmierer Wertebereich von $x \subseteq S$,
und der Kompiler muß es glauben.

ist Fehlerquelle (Behauptung wird nicht geprüft).

Typischer Fall: Collections in Java.

nur notwendig, wenn das Typsystem der Sprache nicht in der Lage ist, das Wissen/die Absicht der Programmierers auszudrücken.

(D. h. Sprache ist für Programmierung ungeeignet. :-)

Statische Polymorphie (Überladung)

Polymorphie (wörtlich): Viel-Förmigkeit

(in Programmiersprachen): der gleiche Bezeichner steht für verschiedene Dinge.

statisches Überladen:

es gibt $f_1 :: T_1$ und $f_2 :: T_2$, aber beide f_i heißen nur f. Bei jeder Benutzung von f ist zu entscheiden, welches f_i gemeint ist.

in Java ist Überladen von Bezeichnern $f_1, f_2, \ldots$ nur erlaubt, wenn alle f_i Unterprogramme sind, die sich in den Argumentlisten (Länge und Typen) unterscheiden.

Damit wird erreicht, daß der Kompiler alle Typen bottom-up (ohne Backtracking) bestimmen kann.

Typkonstruktoren

Ein Typkonstruktor (z. B. „Liste von“) bildet zu einem Argumenttyp einen Ergebnistyp (z. B. `String = List Char`)

in Haskell (generisch mit Typ-Parameter `elt`)

```
data List elt
  = Nil
  | Cons { head :: elt, tail :: List elt }
```

vgl. in C (nicht generisch):

```
typedef struct list
  { elt head; struct list * tail; }
```

Der Listen-Konstruktor

Tatsächliche Notation in Haskell benutzt eckige Klammern:

- Typname: nicht `List elt`, sondern `[ elt ]`
- Konstruktoren:
  - nicht `Nil`, sondern `[]`,
  - nicht `Cons {head = x, tail = ys}`, sondern `x : ys`
- Literale: `[ "foo", "bar" ]`, `[ 1 .. 100 ]`

Welche Typen haben die Funktionen head, tail, Cons (:) ?

welchen Type hat die Funktion

`f x = head ( head x )` ?

Polymorphe (generische) Programme

Ein Unterprogramm ist *generisch*, wenn es (bei gleichem Code) auf verschiedenen Argumenttypen operieren kann

```
length :: [ a ] -> Int
length l = case l of
    []      -> 0
    x : xs -> 1 + length xs

append :: [a] -> [a] -> [a]
append l r = case l of
    []      -> r
    x : xs -> x : append xs r
```

Typprüfung von polymorphen Ausdrücken

Die Typ-Ausdrücke können Variablen enthalten.

Beim Vergleich von Typen muß der tatsächliche Typ T auf den geforderten Typ S *passen*:

d. h. es muß möglich sein, die Typ-Variablen aus S so durch Typ-Ausdrücke zu ersetzen, daß T entsteht.

```
length [ ["foo", "bar" ], [ "frob" ], [] ]
  [ ["foo", "bar" ], [ "frob" ], [] ]
    :: [[String]]
  length
    :: [a] -> Int
Ersetzung mit  a = [String]
```

Generische Programmierung

polymorphe (Container-Typen) $+$ polymorphe Funktionen höherer Ordnung $\Rightarrow$ sehr praktisch.

```
map :: (a -> b) -> [a] -> [b]
map f l = case l of
    []     -> []
    x : xs -> f x : map f xs
```

Aufgabe: welchen Typ muß f haben, damit das korrekt ist:

```
x :: [ Int ]
x = map f "foobar"
```

Welchen Typ hat

```
g = map map
```

Übung Typsysteme 16. 1.

Übung 16. 1.

Sie sollen jetzt nicht Funktionales Programmieren lernen, sondern nur advanced Typchecking üben, indem sie (Fehler-)Meldungen von Hugs studieren und verstehen.

Sprache Haskell `http://www.haskell.org/`,
Interpreter Hugs `http://www.haskell.org/hugs/`

```
$ export PATH=/home/waldmann/built/bin:$PAT
$ hugs +tT
Prelude> 1
Prelude> [1 .. 100]
Prelude> :t length
```

Zeilen-Editor verwenden (Kursor hoch/runter, Control-A,E,P,N usw., wie bash)

hugs verlassen mit Control-D

Beispiele

Fakultät von 100:

```
product [ 1 .. 100 ]
```

als Zeichenkette:

```
show $ product [ 1 .. 100 ]
```

Auf wieviele 0 endet die Fakultät von 100?

```
length $ takeWhile (== '0') $ reverse $ sho
```

Listen und Tupel und Funktionen

Stellen Sie den Typ von takeWhile fest (:t) und erläutern Sie.

Rufen Sie die Funktion dropWhile auf (d. h. geben Sie ihr passende Argumente).

Unbenannte Kreuzprodukte werden als Tupel geschrieben

```
(7, True, "foo")
```

Probieren Sie die Funktionen fst und snd aus.

Rufen Sie die Funktion zip auf. Desgl. für zipWith

concat

Weitere Typ-Rätsel

Finden Sie passende Argumente für `map map`

rufen Sie curry und uncurry auf.

Prüfen Sie von Hand Typ und Wert für

```
d succ 0  where d f x = f (f x)
((d d) succ) 0  where d f x = f (f x)
```

Ergänzungen

Quelltexte der Standardfunktionen:

`/home/waldmann/built/lib/hugs/lib/Prelude.h`

Sprach-Definition:

`http://haskell.org/onlinereport/`

Programm-Beispiele: Autotool-Quellen:

`http://theo1.informatik.uni-leipzig.de/cgi-bin/cvsweb/autotool/`

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)

Informationen im Syntaxbaum

Parser benutzt *kontextfreie* Grammatik

jedes Programm besitzt auch *kontextabhängige* Eigenschaften

$\rightarrow$ müssen durch Durchlaufen des gesamten Syntaxbaumes berechnet werden.

Beispiele für solche Informationen:

- Sichtbarkeiten (public, private, usw.)
- Typen
- Datenfluß-Eigenschaften von Variablen

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?

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

Ausweg: Approximation durch abstrakte Interpretation

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)

Linearisieren

zusammengesetzte Ausdrücke übersetzen

in einzelne Anweisungen mit neuen Variablen:

$x = a * a + 4 * b * c \rightarrow$

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

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 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 geschrieben, 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 $x =$ Knoten mit minimalem Grad,
  - färbe $G \setminus \{x\}$
  - gib x die kleinste Farbe, die nicht in Nachbarn $G(x)$ vorkommt.

Aufgabe: finde Graphen G, für die man nach dieser Heuristik keine optimale Färbung erhält.

Falls dabei mehr Farben als Maschinenregister, dann lege die seltensten Registerfarben in Hauptspeicher.

(Es gibt bessere, aber kompliziertere Methoden.)

Peephole-Optimierung, Instruction Selection

Zwischencode-Liste übersetzen in Zielcode-Liste.

kurze Blöcke von aufeinanderfolgenden Anweisungen optimieren (peephole) 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.

Seminar: Codegenerierung

Transformationen

Transformationen (Aufgabe)

Betrachte Funktionen aus File
`http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/code/matrix.c`

Kompilieren mit `gcc -S` und ohne/mit `-O6`, jeweils Assembler-Output studieren (matrix.s)

Register

Seminar: Registervergabe

http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/code/register.c

Datenfluß-Analyse für:

```
int fun (int a, int b) {
int c ;
int d ;
d = b + a ;
d = a + d ;
b = b + b ;
b = d + a ;
c = a - d ;
b = a - b ;
b = d + b ;
d = d + b ;
```

```
b = b - a ;
a = d + b ;
c = c + b ;
c = c + a ;
b = c - d ;
b = c - b ;
b = b + a ;
return b ;
}
```

Register-Inferenz-Graph bestimmen und färben.

(Hinweis: Programmtext gewürfelt mit diesem Haskell-Skript: `http://www.imn.htwk-leipzig.de/~waldmann/ws03/compilerbau/programme/code/Gen.hs`. So:

```
$ hugs +tT Gen.hs
```

```
Main> make 4 15 >>= putStr
```

mit größeren Parametern würfeln, gleiche Aufgabe nochmal.

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 $\rightarrow$ 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: $Q_I Z$:

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

Beispiel: $S = C_{Sparc}$Sparc. — Wie entsteht C_{Intel}Intel?

Schreiben $P = C_C$Intel.

Compiliere P mit S (auf Sparc), es entsteht $Q = C_{Sparc}$Intel

Compiliere P mit Q (auf Sparc), es entsteht C_{Intel}Intel.

Bootstrapping zur Selbst-Optimierung

gegeben:

- ein „guter“ (optimierender) $G = \mathsf{S_S M}$,
- ein „schlechter“ $A = \mathsf{S_{M'} M'}$.

  (läuft lange und erzeugt langsamen Code)

gesucht: $\mathsf{S_M M}$

Compiliere G mit A, ergibt $B = \mathsf{S_{M'} M}$
(erzeugt effizienten Code, ist aber selbst nicht effizient).

Compiliere G mit B, ergibt $C = \mathsf{S_M M}$
(erzeugt effizienten Code *und* ist selbst effizient).

auf diese Weise auch Implementierung von Sprach-Erweiterungen.

Getrennte Kompilation

reale Compiler haben als Eingabe gar nicht „das Quellprogramm“.

größere Software besteht aus Modulen, die getrennt entwickelt und implementiert werden.

eventuell gibt es gar kein Hauptprogramm (z. B. bei Bibliotheken).

modulweise Kompilation:

- Vorteil: modulares Entwickeln möglich, Zeitgewinn durch weniger (Re-)Kompilation
- Nachteil: Beschreibung und Verwaltung von Modul-Abhängigkeiten nötig

Getrennte Kompilation (II)

schwierig, falls Sprache kein Modulkonzept hat (C, Pascal)

$\rightarrow$ Aufgaben an Tools delegieren (Make(file), Präprozessor, Linker, Loader).

die wissen aber nicht mehr viel über die Sprache (z. B. Typen, Initialisierungs-Reihenfolgen)

schon besser sind Sprachen mit Modulen (Modula, Ada)

aber immer noch *schwierig*, falls (wieder einmal) das Objekt/Klassen-Konzept dafür herhalten muß (C++, Java)

Modul-Schnittstellen

Modul besteht aus Schnittstelle und Implementierung

Ideal: zur Kompilation eines Moduls soll Kenntnis der Schnittstellen der benutzen Module ausreichen.

(d. h. keine Re-Kompilation, falls sich nur andere Implementierung ändert.)

Design-Entscheidungen:

- (beim Sprach-Entwurf): sind Schnittstelle und Implementierung getrennt? (C: Ja (Header-Dateien), Ada: Ja, Java: Nein)

- (bei Compiler-Entwurf): werden Schnittstellen (vor-)kompiliert? (C: Nein, Ada: Ja, Java: Nein)

  vgl: gcc/gxx: pre-compiled headers

Geschichte des Compilerbaus

- erste maschinenunabhängige höhere Programmiersprache (mit arithmetischen Ausdrücken): Fortran $\approx$ 1955,

  Keller-Prinzip (siehe Vortrag von F. L. Bauer)

- formale Syntax für Programmiersprachen (durch CF-Grammatiken): John Backus, Peter Naur, für Algol $\approx$ 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)

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).

Mathematische Methoden

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

Datenstrukturen

- Bäume: für Quellprogramm, für Typ-Ausdrücke, für Zwischencode
- Listen: für Zielprogramm
- Hash-Tabelle: für Symboltabelle

Autotool-Highscore-Auswertung

320	29892	Andre Heinicke
263	29860	Andreas Möhlenbrock
196	27328	Felix Franck
121	...	

Preise (teilweise) gesponsort von:
`http://www.capitospiele.de`

Test-Fragen

Testfragen

Welche Bedingung (bzg. der Stack-Höhe) erfüllt der Code für eine (virtuelle) Stack-Maschine, der bei Kompilation a) eines Ausdrucks, b) einer Anweisung entsteht?

Wozu sind bei der Implementierung von Unterprogramm-Aufrufen im Allgemeinen sowohl dynamische als auch statische Kette nötig? Wieso benötigt man für C-Programme keine statische Kette?

Für welchen Verwendung von Unterprogrammen ist es nicht ausreichend, die dynamische Kette durch zusätzliche Verweise zwischen Stack-Frames zu implementieren?

Gegeben sind Wörter $w_1, \ldots$ und reguläre Ausdrücke $X_1, l, dots$. Finden Sie alle Paare (i, j), für die $w_i \in L(X_j)$.

$w : \epsilon, a, b, abba, bababa$

$X : (a^*b)^*, (ba)^*, ba^*, (ab + ba)^*, (a^*b^*)^*$

Warum sind nicht-deterministische Automaten während der Herstellung eines (flex-)Scanners natürlich und nötig, aber beim tatsächliche Einsatz des Scanners unerwünscht?

Auf welche (zwei) Arten überträgt ein flex-Scanner zur Laufzeit Informationen an einen bison-Parser?

Welche Information benötigt der Scanner vom Parser zur Compile-Zeit?

Erklären Sie am Beispiel eines Integer-Tokens.

Welche Tokenklassen gibt es in den üblichen Programmiersprachen?

Wie implementieren Sie in einem flex-Scanner das Ignorieren von Kommentaren zwischen `//` und Zeilenende?

Geben Sie für die Grammatik $G = (\{a, b\}, \{S\}, s, \{S \to \epsilon, S \to aSbS, S \to bSaS\}$ zwei

verschiedene Links-Ableitungen (mit dazugehörigen Ableitungsbäumen) für das Wort $abbbaa$.

Worin besteht das Problem des „hängenden else“(dangling else)? Geben Sie ein Beispiel.

Nennen Sie die Arbeitsschritte (shift/reduce) und den jeweiligen Stack-Inhalt eines Operator-Präzedenz-Parsers für die Eingabe $3 - 2 * (4 - 1) + 5$ bei den üblichen Vorrang-Regeln.

Wieso sind deterministische Bottom-Up-Parser weiter (d. h. für mehr Sprachen) einsetzbar als deterministische Top-Down-Parser?

Wieso können Top-Down-Parser verständlichere Fehlermeldungen liefern als Bottom-Up-Parser?

Welche Typen müssen die Funktionen f, g haben, damit das folgende korrekt ist:

```
length ( f ( map g [ 1 .. 10 ] ) )
```

```
wobei  map :: (a -> b) -> ([a] -> [b])
```

Welche Möglichkeiten der Optimierung sehen Sie (ggf. unter welchen zusätzlichen Voraussetzungen)?

```
int f (int n, int a) {
  int s = 0;
  for (int i=0; i<n; i++) {
     s += g(a) * h(s);
  }
  return s;
}
```

Welche mathematischen Modelle werden im Compilerbau benutzt?