Compilerbau Vorlesung Wintersemester 2007

Johannes Waldmann, HTWK Leipzig

24. Januar 2008

Literatur

Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman:
Compilers: Principles, Techniques, and Tools (2nd edition)
Addison-Wesley, 2007, ISBN-13: 9780321493453
`http://dragonbook.stanford.edu/`

Inhalt

- Motivation, Hintergründe
- ein kleiner Beispiel-Compiler
- lexikalische Analyse
- syntaktische Analyse
- syntaxgesteuerte Übersetzung
- Zwischencode-Erzeugung
- Laufzeitumgebungen
- Zielcode-Erzeugung

Sprachverarbeitung

- mit Compiler:
  - Quellprogramm → Compiler → Zielprogramm
  - Eingaben → Zielprogramm → Ausgaben
- mit Interpreter:
  - Quellprogramm, Eingaben → Interpreter → Ausgaben
- Mischform:
  - Quellprogramm → Compiler → Zwischenprogramm
  - Zwischenprogramm, Eingaben → virtuelle Maschine → Ausgaben

Compiler und andere Werkzeuge

- Quellprogramm
- Präprozessor → modifiziertes Quellprogramm
- Compiler → Assemblerprogramm
- Assembler → verschieblicher Maschinencode
- Linker, Bibliotheken → ausführbares Maschinenprogramm

Phasen eines Compilers

- Zeichenstrom
- lexikalische Analyse → Tokenstrom
- syntaktische Analyse → Syntaxbaum
- semantische Analyse → annotierter Syntaxbaum
- Zwischencode-Erzeugung → Zwischencode
- maschinenunabhängige Optimierungen → Zwischencode
- Zielcode-Erzeugung → Zielcode
- maschinenabhängige Optimierungen → Zielcode

Methoden und Modelle

- lexikalische Analyse: reguläre Ausdrücke, endliche Automaten
- syntaktische Analyse: kontextfreie Grammatiken, Kellerautomaten
- semantische Analyse: Attributgrammatiken
- Code-Erzeugung: bei Registerzuordnung: Graphenfärbung

Anwendungen von Techniken des Compilerbaus

- Implementierung höherer Programmiersprachen
- architekturspezifische Optimierungen (Parallelisierung, Speicherhierarchien)
- Entwurf neuer Architekturen (RISC, spezielle Hardware)
- Programm-Übersetzungen (Binär-Übersetzer, Hardwaresynthese, Datenbankanfragesprachen)
- Software-Werkzeuge

Überblick

ein einfaches Compiler-Frontend
Quellen:

- Kapitel 2 aus Drachenbuch
- Code: `http://dragonbook.stanford.edu/dragon-front-source.tar`

Absicht:

- sehen, wie es geht (alles „von Hand programmiert“)
- Übungen: diesen Compiler verstehen und erweitern
- später: Nutzen von Werkzeugen und Theorie erkennen

Beispiel

Eingabe (≈ Java):

```
{ int i;
  float prod;
  float [20] a;
  float [20] b;
  prod = 0;
  i = 1;
  do {
    prod = prod
      + a[i]*b[i];
    i = i+1;
  } while (i <= 20);
}
```

Ausgabe (Drei-Adress-Code):

```
L1: prod = 0
L3: i = 1
L4: t1 = i * 8
    t2 = a [ t1 ]
    t3 = i * 8
    t4 = b [ t3 ]
    t5 = t2 * t4
    prod = prod + t5
L6: i = i + 1
L5: if i <= 20 goto L4
L2:
```

Aufbau

- lexikalische Analyse
- syntaktische Analyse
- Zwischencode-Erzeugung

alle Teile benutzen: Symboltabelle(n)

Gliederung

- Syntax-Definition (Grammatiken)
- top-down parsing
- lexikalische Analyse
- Symboltabellen, Bereiche (scopes)
- Zwischencode-Erzeugung

Syntax-Definition

Wiederholung aus PPS:
(kontextfreie) Grammatik, Ableitung, Ableitungsbaum, Eindeutigkeit
typische Grammatiken für

- Anweisungen
- Ausdrücke

Top-Down parsing

für jede Variable V aus der Grammatik:
schreibe eine Prozedur P_V, die ein Wort liest, das aus V erzeugt werden kann,
und dabei den Eingabestrom „verbraucht“(d. h. voranschreitet).

Falls es zu dieser Variablen mehrere Regeln gibt,
betrachte nächstes Zeichen (Token), um zu entscheiden.

Beispiel: der GNU-Ada-Parser ist auf diese Weise von Hand geschrieben.

Linksrekursion

Für prädiktive Parser (ein Zeichen Vorschau) wird das schwer:

```
exp -> exp + term | ...
```

Def: eine Grammatik $G = (\Sigma, V, S, R)$ heißt linksrekursiv, wenn $\exists v \in V, w \in (\Sigma \cup V)^* : v \to_R^+ v \cdot w$.
Satz: zu jeder CFG G gibt es eine nicht linksrekursive CFG G' mit $L(G') = L(G)$.
einfachster Fall: Grammatik mit Regeln $\{A \to Ab, A \to c\}$

Linksrekursion (II)

Def: CFG $G = (\Sigma, V, S, R)$ ist in Greibach-Normalform, falls für jede Regel $(l \to r) \in R$ gilt: $r \in \Sigma V^*$.
möglich ist auch: $\ldots r \in \Sigma(V \cup \Sigma)^*$
Satz: zu jeder CFG G gibt es eine CFG G' in Greibach-Normalform mit $L(G) \setminus \epsilon = L(G')$.
Aufgaben (evtl. autotool): Greibach-Normalform von

- $(\{a, b\}, \{S, T\}, S, \{S \to TS \mid b, T \to ST \mid a\})$
- $(\{a, b\}, \{S, T\}, S, \{S \to TT \mid b, T \to SS \mid a\})$

Lexikalische Analyse

Wiederholung aus PPS:

- Token, Tokenklassen
- Beschreibung durch reguläre Ausdrücke
- Realisierung durch endliche Automaten
- in einfachen Fällen: zu Fuß programmieren

Symboltabellen

ordnen jedem Bezeichner zu:

- Namen
- Position (der Definition) im Quelltext
- Typ
- Position im Speicher

durch Blockstruktur:

- (Gültigkeits/Sichtbarkeits)bereiche (scopes)
- für jeden Bereich eine Symboltabelle

Semantik

- zu jedem Knoten des Syntaxbaumes ein *Attribut* zuordnen: das Zwischencode-Programm für den Teilbaum, der in diesem Knoten beginnt.
- ist *synthetisiertes* Attribut: (Wert ergibt sich aus Knoten selbst und Attributen der Kinder)

L/R-Values

Zuweisung: `Ausdruck := Ausdruck;` (?)

- links vom Zuweisungsoperator müssen Ausdrücke anders übersetzt werden als rechts davon.
- rechts wird ein Programm generiert, das einen *Wert* erzeugt, links eine *Adresse*

Anwendungen/Diskussion (Übung)

- Array-Zugriffe
- Pre/Post-Inc/Decrement-Operatoren

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 (`=`, `+`, `&&`, ...)

Scanner mit Flex

```
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

Reguläre Ausdrücke/Sprachen

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

Die folgenden Aussagen sind äquivalent:

- L wird von einem regulären Ausdruck erzeugt.
- L wird von einer rechtslinearen Grammatik erzeugt. (Chomsky-Typ 3)
- L wird von einem endlichen Automaten akzeptiert.
- L wird von einem endlichen deterministischen Automaten akzeptiert.

Reguläre Ausdrücke

...über einem Alphabet Σ ist die kleinste Menge E mit:

- atomare Ausdrücke:
  - 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`)
- zusamenngesetzte Ausdrücke: 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: `^*`)

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

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 $w = c_1 c_2 \dots c_n$ 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 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

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:

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

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 $s \stackrel{X}{\rightarrow} f$.

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. Der so erzeugt Automat A ist korrekt. $|Q(A)| \leq 2|X|$.

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 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.

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:
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 $\leq k$ *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 $L(A') = L(A)$.

Nicht reguläre Sprachen

gibt es reguläre Ausdrücke/endliche Automaten für diese Sprachen?

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

Nein.

Die Nerode-Kongruenz (I)

Für jede Sprache $L \subseteq \Sigma^*$ definieren wir eine Äquivalenzrelation auf Σ^* durch

$$u \sim_L v :\iff \forall w \in \Sigma^* : (uw \in L \iff vw \in L)$$

Beispiele: $\Sigma = \{a, b\}, L_1 = a^* b^*, L_2 = \{a^n b^n \mid n \geq 0\}$.
Welche der Wörter sind jeweilskongurent:

$$\epsilon, a, b, ab, ba, a^4, a^4 b^4?$$

Wieviele Kongruenzklassen gibt es?

Die Nerode-Kongruenz (II)

Satz: Für jede Sprache $L \subseteq \Sigma^*$:
L ist regulär $\iff$ $\Sigma^*/\sim_L$ ist endlich ($\sim_L$ besitzt endlich viele Äquivalenzklassen).
Beweis: die Äquivalenzklassen von $\sim_L$ sind die Zustände eines minimalen deterministischen vollständigen Automaten für L.

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

- $w = w_{k_1} w_{k_1} \dots w_{k_n}$
- für alle $1 \leq i \leq n$: w_{k_i} ist *longest match*:
  - $w_{k_i} \in L(X_{k_i})$
  - jedes Anfangsstück von $w_{k_i} \dots w_{k_n}$, 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 akz. Zustände (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); }    }
```

Aufgabenstellung

- Eingabe:
  - ein Wort (Muster) m
  - ein Wort (Ziel) w
- Ausgabe: die kleinste Zahl i,
  für die m ein Präfix von $w_i w_{i+1} \dots$ ist,
  oder die Feststellung, daß es kein solches i gibt.

Variante: eine endliche Menge von Mustern $\{m_1, \dots, m_k\}$

Triviale Lösung

```
outer:
  for (int i = 0; i < w.length(); i++) {
    for (int j = 0; i < m.length (); j++) {
      if (m[j] != w[i+j]) continue outer;
    }
    return i;
  }
```

- triviale Laufzeit: $O(|w| \cdot |m|)$
- läßt sich $O(|w| + |m|)$ erreichen?
- …unterbieten?

Knuth-Morris-Pratt

Idee: benutze den minimalen deterministischen Automaten für $\Sigma^* m \Sigma^*$.

- nichtdeterministischer Automat mit $|m| + 1$ Zuständen
- Potenzmengenkonstruktion
- vereinfache die Zustandsbezeichnungen
- beschreibe Zustandsübergänge durch *failure function*
- Laufzeit?
- Beste/schlechteste Fälle?

KMP-Failure-Function

$$f \quad : \quad \{1 \dots |m|\} \to \{0 \dots |m| - 1\}$$
$$k \mapsto \max\{i : 0 \leq i < k \wedge m[1 \dots i] = m[k - i + 1 \dots k]\}$$

Beispiel:

m	a	a	b	a	b	a	a	b
k	1	2	3	4	5	6	7	8
f	0	1	0	1	0	1	2	3

Rekonstruktion, Beispiel: ($\Sigma = \{a, b\}$)

m							a					
k	1	2	3	4	5	6	7	8	9	10	11	12
f			1				4		0	1		1

Boyer-Moore

```
int i = 0;
while (true) {
  int j = m.length ();
  while (true) {
    if (m[j] != w[i+j]) break;
    if (j == 1) return i;
    j--;
  }
  i += offset;
}
```

offset ist Maximum von:

- bad-character heuristics
- good-suffix heuristics

Bad-Character-Heuristik

$$\begin{aligned} \lambda \quad : \quad & \Sigma \to \{0 \ldots |m|\} \\ & c \mapsto \max\left(\{0\} \cup \{i : 1 \leq i < k \wedge m[i] = c\}\right) \end{aligned}$$

Beispiel (für $m = abcaac$)

Σ	a	b	c	d
λ	5	2	6	0

Anwendung: $\texttt{offset} = j - \lambda[w[i + j]]$

Good-Suffix-Heuristik

$u \sim v :\iff$ u ist Suffix von v oder v ist Suffix von u
(Vorsicht: ist keine Äquivalenzrelation)

$$\begin{aligned} \gamma' \quad : \quad & \{1 \dots |m|\} \to \{1 \dots |m|\} \\ & j \mapsto \max \left\{ k : \begin{array}{rl} & 0 \leq k < |m| \\ \wedge & m[j+1 \dots |m|] \sim m[1 \dots k] \end{array} \right\} \end{aligned}$$

Beispiel:

m	a	b	a	b	b	a	b	c	a	b
j	1	2	3	4	5	6	7	8	9	10
γ'	2	2	2	2	2	2	2	7	7	9

Anwendung: $\texttt{offset} = |m| - \gamma'[j]$

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
       , transitionen     ::
     FiniteMap (Maybe x, z, y) (Set (z, [y]))
       }

data Modus z =
  Leerer_Keller | Zustand ( Set z )
```

Beispiel Kellerautomat

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

Sprachen von Keller-Automaten

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 $(z', u') \in T(\epsilon, z, y)$

akzeptierte Sprachen:

- 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 ($\Rightarrow$)

Grammatik	$\Rightarrow$ Automat
	nur ein Zustand z_0
	Akzeptanz durch leeren Keller
Variablen $\cup$ Terminale	$=$ Kelleralphabet
Startsymbol	$=$ Startsymbol (im Keller)
Regel $X \to w$	Regel $(\epsilon, z_0, X, z_0, w)$
jedes $x \in \Sigma$	Regel $(x, z_0, x, z_0, \epsilon)$.

Invariante (während der Rechnung):
(verbrauchter Teil der Eingabe $\circ$ Kellerinhalt)
ist aus Startsymbol ableitbar.

$LL(k)$-Grammatiken

Für $k \in \mathbb{N}$:
$LL(k)$ ist die Menge aller CFG G, bei denen im angegebenen Automaten durch Vorausschau um k Zeichen die auszuführende Regel eindeutig bestimmt ist.

Definition:
Für alle Paare von Linksableitungen

$$S \to_L^* uTv \to_L uwv \to_L^* ux$$
$$S \to_L^* uTv \to_L uw'v \to_L^* ux',$$

bei denen x und x' bis zur Länge k übereinstimmen,
gilt $w = w'$.

LR-Parsing

Invariante:
verbrauchte Eingabe ist aus Spiegelbild des Kellers ableitbar
Aktionen im Keller:

- *shift*: Eingabezeichen $\rightarrow$ push
- (eventuell vorher)
  reduce (mit Regel $T \rightarrow w$):
  Anfangsstück w des Kellers durch T ersetzen

LR-Items

- benutze größeres Kelleralphabet, damit man bereits am top-of-stack erkennt, welche Regeln anwendbar sind.
- ein solches Kellerzeichen ist eine Menge von Items,
- ein Item hat die Form $(T \to u \cdot v)$
  mit der Bedeutung: u haben wir schon gesehen; wenn noch v kommt, dann können wir alles durch T ersetzen.
- Items mit möglichen Übergängen bilden endlichen Automaten.
  (groß $\Rightarrow$ Werkzeuge)

$LR(k)$-Grammatiken

Für $k \in \mathbb{N}$:
$LR(k)$ ist die Menge aller CFG G, bei denen im angegebenen Automaten durch Vorausschau um k Zeichen die auszuführende Regel (reduce) eindeutig bestimmt ist.

Definition:
Für alle Paare von Rechtsableitungen

$$S \to_R^* uTv \to_R uwv$$
$$S \to_R^* u'T'v' \to_R uwv',$$

bei denen v und v' bis zur Länge k übereinstimmen,
gilt $u = u', T = T', v = v'$

Operator-Präzedenz-Parser

Ziel: wertet arithmetische Ausdrücke mit Zahlen und Operatoren $+, -, *, /$ aus.

Realisierung: benutzt zwei Stacks:

- Wert-Stack
- Operator-Stack

Arbeitsweise:

- Zahl: push auf Wert-Stack
- Operator:
  - vergleiche mit top-of-opstack
  - ggf. reduce (Rechnen auf Wert-Stack)
  - push auf opstack

Die Quelltexte zur Übung:

`http://dfa.imn.htwk-leipzig.de/cgi-bin/cvsweb/cb07/src/opp/?cvsroot=pub`

Kann auch über Eclipse/CVS importiert werden:

```
connection type: pserver
user: anonymous
host: dfa.imn.htwk-leipzig.de
```

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.
- Beide Methoden lesen die Eingabe *von links*!
- Beide Methoden benutzen *Vorschau-Zeichen* (üblich: eines)

Top-Down/Bottom-Up und Eindeutigkeit

- Für effizientes Parsen möchte man kein Backtracking, also *Eindeutigkeit* (der Auswahl der anzuwendenden Regel, abhängig von den Vorschau-Zeichen)
- Das ist bei Top-Down(LL)-Parsern eine starke Einschränkung, aber bei Bottom-Up(LR)-Parsern nicht so gravierend:
- diese können Entscheidungen „in die Zukunft“verschieben, indem Zwischenergebnisse auf dem Stack gespeichert werden.
- für Konstruktion von LR-Parsern benötigt man Werkzeuge, das das benutzte Kelleralphabet sehr groß ist.

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 (yytext);
  return NUM; }
[-+*/^()] {
  return yytext[0]; }
```

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 { $$ = $3; $1->value.var = $3;}
 | FNCT '(' exp ')'
          { $$ = (*($1->value.fnctptr))($3); }
 | exp '+' exp { $$ = $1 + $3; }
```

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`

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 PExpression
   -- 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

- Quelle: `http://sablecc.org/`
- im Linux-Pool installiert (sablecc in `/home/waldmann/built/bin`)
- Beispiele in `http://www.imn.htwk-leipzig.de/~waldmann/edu/ws05/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
```

SableCC-Aufgaben

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?

Einleitung

sablecc ist eine DSL zur Beschreibung/Erzeugung von Parsern.
ist *aufgesetzt* (auf Java):

- eigene konkrete Syntax
- benötigt Parser
- benötigt Interpreter/Compiler

Eingebettete DSL

Bsp: Parser als Java-Objekte (elementare und Kombinatoren) können von Gastsprache übernehmen:

- konkrete Syntax
- Modulsystem
- Abstraktionen (Unterprogramme)
- Bibliotheken (Datenstrukturen)
- Ausführungsumgebung (Interpreter/Compiler)

Beispiel: Java-Parsec

Original: Daan Leijen (2000) für Haskell
`http://legacy.cs.uu.nl/daan/parsec.html`
`http://www.haskell.org/ghc/docs/latest/html/libraries/parsec/Text-ParserCombinators-Parsec.html`
Hier: Nachbau für Java (Machbarkeitststudie)
Vgl. Atze Dijkstra, Doaitse S. Swierstra: Lazy Functional Parser Combinators in Java (2001), `http://citeseer.ist.psu.edu/dijkstra01lazy.html`

Beispiel: Java-Parsec

```
interface Parser<T> {
  Result<T> parse (PushbackReader in) throws IOExce
}

class Arithmetic {
  final static Parser<Integer> product =
    Combine.transform(
      Combine.sepBy(Atom.expect('*'), Basic.natural
        Combine.fold(1,
          new Function<Pair<Integer, Integer>, Inte
            public Integer compute(Pair<Integer, In
              return x.getFirst() * x.getSecond();
```

Motivation

Unterprogramme sind wichtiges Mittel zur Abstraktion
das möchte man überall einsetzen
also sind auch lokale Unterprogramme wünschenswert
(Konzepte *Block* und *Unterprogramm* sollen orthogonal sein)
Dann entsteht Frage: Wie greifen lokale Unterprogramme auf nichtlokale Variablen zu?

Frames, Ketten

Während ein Unterprogramm rechnet, stehen seine lokalen Daten in einem Aktivationsverbund (Frame), jeder Frame hat zwei Vorgänger:

- dynamischer V. (Frame des aufrufenden UP) (benutzt zum Rückkehren)
- statischer V. (Frame des textuell umgebenden UP) (benutzt zum Zugriff auf “fremde” lokale Variablen)

Beispiel: zeichen Frames und statische/dynamische Links für `a(3,4)` bei

```
int a (int x, int y) {
  int b (int z) { return z > 0 ? 1 + b (z-1) : x; }
  return b (y);
}
```

Übung: Assemblercode verstehen (`gcc -S`)

Unterprogramme als Argumente

```
int d ( int g(int x) ) { return g(g(1)); }

int p (int x) {
  int f (int y) { return x + y ; }
  return d (f);
}
```

Betrachte Aufruf $p(3)$.
Das innere Unterprogramm f muß auf den p-Frame zugreifen, um den Wert von x zu finden.
Dieser Frame lebt.
Wenn Unterprogramme nur “nach innen” als Argumente übergeben werden, können die Frames auf einem Stack stehen.
Übung: Assemblercode verstehen

Unterprogramme als Resultate

```
int x1 = 3;

int (*s (int foo)) (int x2) {
  int f (int y) { return x1 + y; }
  return &f;
}

int main (int argc, char ** argv) {
  int (*p) (int) = s(4);
  printf ("%d\n", (*p)(3));
}
```

In `f` ersetze `x1` durch `x2`.
Assemblercode erklären.

Lokale Klassen

- static nested class:

  `class C { static class D { .. } .. }`

  dient lediglich zur Gruppierung
- nested inner class:

  `class C { class D { .. } .. }`

  jedes D-Objekt hat einen Verweis auf ein C-Objekt ($\approx$ statische Kette) (bezeichnet durch `C.this`)
- local inner class:

  `class C { void m () { class D { .. } .. } }`

  Zugriff auf lokale Variablen in *m* nur, wenn diese final sind. Warum?

Unterprogramme/Zusammenfassung

in prozeduralen Sprachen:

- alle UP global: dynmische Kette reicht
- lokale UP: benötigt auch statische Kette
- lokale UP as Daten: benötigt Closures
  $=$ (Code, statischer Link)
- UP als Argumente: Closures auf Stack
- UP als Resultate: Closures im Heap

vgl. `http://www.function-pointer.org/`
in objektorientierten Sprachen: keine lokalen UP, aber lokale (inner, nested) Klassen.

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)

Einfacher Matrix-Zugriff

```
#include <stdio.h>
#define N 100

typedef int matrix [N][N];

// zum betrachten der index-rechnungen
int access (matrix a, int i) {
  int x = a[3][i];
  int y = a[i][5];
  return x + y;
}
```

Compilieren mit `gcc -S` ergibt:

```
diag:
        !#PROLOGUE# 0
        save    %sp, -120, %sp
        !#PROLOGUE# 1
        st      %i0, [%fp+68]    -- &a
```

Constant folding

Zugriff auf `a[3][i]`.

```
ld      [%fp+72], %o0      -- i
mov     %o0, %o1           -- i
sll     %o1, 2, %o0        -- 4*i
ld      [%fp+68], %o1      -- &a
add     %o0, %o1, %o0      -- &a + 4*i
ld      [%o0+1200], %o1    -- mem [ &a + 4 *
st      %o1, [%fp-20]      -- x
```

strength reduction

Zugriff auf `a[i][5]`.

```
ld      [%fp+72], %o0      -- i
mov     %o0, %o2           -- i
sll     %o2, 1, %o1        -- 2*i
add     %o1, %o0, %o1      -- 3*i
sll     %o1, 3, %o2        -- 24*i
add     %o2, %o0, %o2      -- 25*i
sll     %o2, 4, %o0        -- 25*16*i = 4 * N
ld      [%fp+68], %o1      -- &a
add     %o0, %o1, %o0      -- &a + 4*N*i
ld      [%o0+20], %o1      -- mem [&a + 4*N*i
st      %o1, [%fp-24]      -- y
```

Rückgabe

Ergebnis ausrechnen und zurückgeben:

```
        ld      [%fp-20], %o0
        ld      [%fp-24], %o1
        add     %o0, %o1, %o0
        mov     %o0, %i0
        b       .LL2
         nop
.LL2:
        ret
        restore
```

Stack-Frames

Compiliere jetzt mit `gcc -S -O`:

```
diag:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        sll     %o1, 2, %g2        -- 4*i
        add     %g2, %o0, %g2      -- &a + 4*i
        ld      [%g2+1200], %g3 -- mem [&a + 4*i +
        sll     %o1, 1, %g2        -- 2*i
        add     %g2, %o1, %g2      -- 3*i
        sll     %g2, 3, %g2        -- 24*i
        add     %g2, %o1, %g2      -- 25*i
        sll     %g2, 4, %g2        -- 16*25i = 4*N*i
        add     %g2, %o0, %g2      -- &a + 4*N*i
        ld      [%g2+20], %o0      -- mem [ &a + 4*N*i
        retl
        add     %g3, %o0, %o0
```

wir rechnen im Stack-Frame des Callers.
benutzen globale register.

Schleifen, Code-Verschiebung

```
// c := a + b
void add (matrix c, matrix a, matrix b) {
  int i; int j;
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      c [i][j] = a[i][j] + b[i][j];
    }
  }
}
```

Kompilieren mit `gcc -S -O`

```
add:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        mov     %i0, %o7            -- &c
        mov     0, %g4              -- i
        mov     0, %i3              -- j
```

Arithmetische Umformungen

Gleiche Funktion `add` jetzt mit `gcc -S -O6`, betrachte innere Schleife:

```
.LL11:
        ld      [%i3+%i5], %g2
        add     %i4, 1, %i4
        ld      [%i3+%i0], %g3
        cmp     %i4, 99
        add     %g2, %g3, %g2
        st      %g2, [%i3+%g1]
        ble     .LL11
        add     %i3, 4, %i3         -- Index weiterzä
```

Mehr Schleifen

```
// c := a * b
void times (matrix c, matrix a, matrix b) {
  int i; int j; int k;
  for (i=0; i<N; i++) {
    for (k=0; k<N; k++) {
      c[i][k] = 0.0;
      for (j=0; j<N; j++) {
        c[i][k] = c[i][k] + a[i][j] * b[j][k];
      }
    }
  }
}
```

mit `gcc -S -O6`:

```
times:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
```

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`)

Grundsätzliches

- ein Compiler verarbeitet Programme
- alle nicht trivialen semantischen Eigenschaften von Programmen (einer Turing-vollständigen Programmiersprache) sind unentscheidbar (z. B.: wird eine Anweisung jemals ausgefÃ$\frac{1}{4}$hrt, wird eine Speicherstelle mehr als einmal geschrieben usw.)

Compiler haben es schwer

- Compiler muß raten/approximieren
- oder Programmierer muß mithelfen (z. B. Typen deklarieren)

Schwere Aufgaben für Compiler/Werkzeuge (Bsp 1)

lexikalische/syntaktische Analyse (Generierung von Werkzeugen aus Beschreibungen)

$$L(A) = L(B)$$

- für reguläre Ausdrücke A, B: entscheidbar
- für kontextfreie Grammatiken A, B: nicht entscheidbar

Schwere Aufgaben für Compiler (Bsp 2)

Äquivalenz von algebraischen Ausdrücken (Polynomen)

- ist unentscheidbar (Hilberts 10. Problem)
- Anwendung: Optimierung von Zählschleifen

vgl. (In-)Äquivalenz regulärer Ausdrücke, kleinster äquivalenter regulärer Ausdruck

Schwere Aufgaben für Compiler (mehr Bsp)

- (Maschinen)Befehls-Auswahl
  $\longrightarrow$ Rucksack-Problem, NP-vollst.
- Registervergabe
  $\longrightarrow$ Graphenfärbung, NP-vollst.
- Typprüfung/Inferenz
  mit generischen Typen: wenigstens Exp-Time

Registervergabe

- Werte von lokalen Variablen in Blöcken und von (Teil-)Ausdrücken sollten, wenn möglich, in Prozessor-Registern stehen, (Zugriff ist sehr viel schneller aus auf Hauptspeicher).
- gleichzeitig benötigte Werte müssen in verschiedenen Registern stehen (definiert Graph)
- Registerzahl der CPU ist begrenzt (Anzahl der Farben)

Problem COL$= \{(G, k)$: Graph G besitzt konfliktfreie Färbung mit k Farben $\}$.
…3COL ist NP-vollständig (Reduktion von 3SAT)

3COL – Hausaufgabe

finde einen Baustein H mit den Eigenschaften:

- von H führen genau 4 Kanten nach außen
- wenn die dadurch bestimmten 4 Nachbarknoten von H alle identisch gefärbt sind, läßt sich das nicht zu einer 3-Färbung von H fortsetzen.
- wenn die 4 Nachbarknoten insgesamt genau zwei Farben benutzen, dann läßt sich das zu einer 3-Färbung von G fortsetzen.

Färbung (Heuristik)

Farben $\{1, 2, \ldots, k\}$

- nächster freier Knoten erhält kleinste freie Farbe
- Knotenreihenfolge wählen z. B. nach Grad (im Restgraphen)

wie gut ist diese Heuristik?

Typprüfung/Inferenz

Begriffe:

- Prüfung: Programmierer deklariert, Compiler prüft
- Inferenz: Compiler ergänzt Deklaration (C#: var, Haskell: let)

Aufgabe: vlg. in Java: Konstruktor/Fabrikmethode
Komplexität:

- einfache Typen (System F1): polynomiell (Wort- bzw. Baumautomaten)
- generische Typen (Variablen über Typen, System F2): exponentiell
- höhere Variablen (System F): unentscheidbar

Siehe Beispiele Java, C#, Haskell

Typprüfung/Inferenz (Beispiel)

```
next :: Integer -> Integer
next x = x + 1

twice :: (a -> a) -> (a -> a)
twice f x = f (f x)

main = print $ twice twice twice twice next 0
```

Fragen:

- Welcher Wert wird ausgegeben?
- Welchen Typ hat das linke `twice`?
- Wie sieht entsprechendes Programm in Java aus? (Hinweis: verwende Funktionsobjekte)

  ```
  interface F<A,B> { B apply(A x); }
  ```
- Wie in C#?
- Welche für Java 7 angekündigten Neuerungen könnten helfen?