Praktikum Compilerbau

Wintersemester 2005/06

Martin Hofmann, Andreas Abel, Hans-Wolfgang Loidl

Einführung

- Organisatorisches
- Aufgaben und Aufbau eines Compilers
- Überblick über das Praktikum
- Interpretation geradliniger Programme

Organisatorisches

- Das P richtet sich nach dem Buch *Modern Compiler Implementation in Java* von Andrew W Appel, CUP, 2005, 2. Aufl.
- Es wird ein Compiler für eine Teilmenge von Java: MiniJava entwickelt.
- Jede Woche wird ein Kapitel durchgenommen; ca. 30% VL und 70% Programmierung im Beisein der Dozenten.
- Die beaufsichtigte Programmierzeit wird i.A. nicht ausreichen; Sie müssen noch ca. 4h/Woche für selbstständiges Programmieren veranschlagen.
- Die Programmieraufgaben werden in Gruppen à zwei Teiln. bearbeitet (Extreme Programming).
- Scheinvergabe aufgrund erfolgreicher Abnahme des Programmierprojekts durch die Dozenten. Die Abnahme wird mündliche Fragen zum in der VL vermittelten Stoff enthalten.

Aufgaben eines Compilers

- Übersetzt Quellcode (in Form von ASCII Dateien) in Maschinensprache (“Assembler”)
- Lexikalische und Syntaxanalyse
- Semantische Analyse (Typüberprüfung und Sichtbarkeitsbereiche)
- Übersetzung in Zwischencode: keine lokalen Variablen, Sprunganweisungen und Funktionsaufrufe als einzige Kontrollstruktur
- Erzeugung von Maschineninstruktionen (architekturabhängig)
- Registerzuweisung
- Ausgabe in Binärformat

Zu verschiedenen Zeitpunkten können Optimierungen vorgenommen werden.

Geradlinige Programme

- Bestehen aus Zuweisungen, arithmetischen Ausdrücken, mehrstelligen Druckanweisungen.
- Beispiel:

  ```
  a := 5+3; b := (print(a, a-1), 10*a); print(b)
  ```

  Ausgabe:

  ```
  8 7
  80
  ```

  BNF Grammatik:

$$
\begin{array}{lll}
\mathit{Stm} & ::= & \mathit{Stm}\,\texttt{;}\,\mathit{Stm} \mid \mathit{ident}\,\texttt{:=}\,\mathit{Exp} \mid \texttt{print(}\,\mathit{ExpList}\,\texttt{)} \\
\mathit{Exp} & ::= & \mathit{ident} \mid \mathit{num} \mid \texttt{(}\,\mathit{Stm}\,\texttt{,}\,\mathit{Exp}\,\texttt{)} \mid \ldots \\
\mathit{ExpList} & ::= & \mathit{Exp} \mid \mathit{Exp}\,\texttt{,}\,\mathit{ExpList}
\end{array}
$$

Abstrakte Syntax in Java

```
abstract class Stm {}

class CompoundStm extends Stm {
   Stm stm1, stm2;
   CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}
}

class AssignStm extends Stm {
   String id; Exp exp;
   AssignStm(String i, Exp e) {id=i; exp=e;}
}

class PrintStm extends Stm {
   ExpList exps;
   PrintStm(ExpList e) {exps=e;}
}
```

Abstrakte Syntax in Java

```
abstract class Exp {}

class IdExp extends Exp {
   String id;
   IdExp(String i) {id=i;}
}

class NumExp extends Exp {
   int num;
   NumExp(int n) {num=n;}
}

class OpExp extends Exp {
   Exp left, right; int oper;
   final static int Plus=1,Minus=2,Times=3,Div=4;
   OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}
}
```

Abstrakte Syntax in Java

```
class EseqExp extends Exp {
   Stm stm; Exp exp;
   EseqExp(Stm s, Exp e) {stm=s; exp=e;}
}

abstract class ExpList {}

class PairExpList extends ExpList {
   Exp head; ExpList tail;
   public PairExpList(Exp h, ExpList t) {head=h; tail=t;}
}

class LastExpList extends ExpList {
   Exp head;
   public LastExpList(Exp h) {head=h;}
}
```

Beispiel”programm”

```
new CompoundStm(new AssignStm("a",new OpExp(new NumExp(5), OpExp.P
                                             new NumExp(3))),
 new CompoundStm(new AssignStm("b",
     new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
                          new LastExpList(new OpExp(new IdExp("a"), O
                                                    new NumExp(1)))))
                 new OpExp(new NumExp(10), OpExp.Times, new IdExp("a")
  new PrintStm(new LastExpList(new IdExp("b")))));
}
```

Programmieraufgabe für heute

- Implementieren einer Klasse `Table`, die Zuordnungen Bezeichner $\mapsto$ Werte, d.h. Umgebungen, modelliert.

  Umgebungen werden funktional implementiert, z.B. als Listen oder und können nicht imperativ verändert werden.

- Bereitstellen und Implementieren einer Methode

  ```
  Table interp(Table t)
  ```

  in der Klasse `Stm`.

  Der Aufruf `Table tneu = s.interp(t)` soll das Programm `s` in der Umgebung `t` auswerten und die daraus resultierende neue Umgebung in `tneu` abspeichern.

  Hierzu deklarieren Sie diese Methode in `Stm` als abstrakt und implementieren sie dann jeweils in den konkreten Unterklassen. Sie benötigen entsprechende Hilfsmethoden in der Klasse `Exp`.

- Viel Erfolg.

Lexikalische Analyse

- Was ist lexikalissche Analyse
- Durchführung mit endlichen Automaten
- Bedienung von Werkzeugen: JLex, JFlex.
- Praktikum: Lexer für MiniJava mit JLex.

Was ist lexikalische Analyse?

- Erste Phase der Kompilierung
- Entfernt Leerzeichen, Einrückungen und Kommentare
- Die Eingabe wird in eine Folge von *Token* umgewandelt.
- Die *Tokens* spielen die Rolle von Terminalsymbolen für die Grammatik der Sprache.

Was sind Tokens?

Beispiele:

`foo` (ID), `73` (INT), `66.1` (REAL), `if` (IF), `!=` (NEQ),
`(` (LPAREN), `)` (RPAREN)

Keine Beispiele:

`/* huh? */` (Kommentar),
`#define NUM 5` (Präprozessordirektive),
`NUM` (Makro)

Beispiel für die lexikalische Analyse

```
void match0(char *s) /* find a zero */
{if (!strncmp (s, "0.0", 3))
    return 0.;
}
```

VOID ID(`match0`) LPAREN CHAR STAR
ID(`s`) RPAREN LBRACE IF LPAREN
BANG ID(`strncmp`) LPAREN ID(`s`)
COMMA STRING(`0.0`) COMMA INT(`3`)
RPAREN RPAREN RETURN REAL(`0.0`)
SEMI RBRACE EOF

Reguläre Ausdrücke und NEA/DEA

`(`	$\rightarrow$	LPAREN
$digit\,digit^*$	$\rightarrow$	INT(ConvertToInt(“verarbeiteter Text”))
`print`	$\rightarrow$	PRINT
$letter(letter + digit + \{\_\})^*$	$\rightarrow$	ID(“verarbeiteter Text”)
...		

wobei $digit = \{\texttt{0}, \ldots, \texttt{9}\}$ und $letter = \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}\}$.

Motto: Tokens werden durch reguläre Ausdrücke spezifiziert; Verarbeitung konkreter Eingaben mit DEA. Automatische Lexergeneratoren wie (lex, flex, JLex, JFlex, Ocamllex) wandeln Spezifikation in Programm um.

Auflösung von Mehrdeutigkeiten

I.a. gibt es mehrere Möglichkeiten eine Folge in Tokens zu zerlegen.

Lexergeneratoren verwenden die folgenden zwei Regeln:

Längste Übereinstimmung: Das längste Präfix, das zu irgendeinem der regulären Ausdrücke passt, wird das nächste Token.

Regelpriorität: Wenn das nicht hilft, kommt die weiter oben stehende Regel zum Zug. Die Reihenfolge der Regeln spielt also eine Rolle.

`print0` $\rightarrow$ ID(`print0`) *nicht* PRINT INT(`0`)

`print` $\rightarrow$ PRINT *nicht* ID(`print`)

Implementierung

Man baut mit Teilmengenkonstruktion einen Automaten, dessen Endzustände den einzelnen Regeln entsprechen. “Regelpriorität” entscheidet Konflikte bei der Regelzuordnung.

Zur Implementierung von “längste Übereinstimmung” merkt man sich die Eingabeposition, bei der das letzte Mal ein Endzustand erreicht wurde und puffert jeweils die darauffolgenden Zeichen, bis wieder ein Endzustand erreicht wird. Kommt man dagegen in eine Sackgasse, so bestimmt der letzte erreichte Endzustand die Regel und man arbeitet zunächst mit den gepufferten Zeichen weiter.

Lexergeneratoren

Ein Lexergenerator erzeugt aus einer `.lex`-Datei einen Lexer in Form einer Java-Klasse (bzw. C, ML Modul), das eine Methode (Funktion) `nextToken()` enthält.

Jeder Aufruf von `nextToken()` liefert das “Ergebnis” zurück, welches zum nächsten verarbeiteten Teilwortes der Eingabe gehört.

Normalerweise besteht dieses “Ergebnis” aus dem Namen des Tokens und seinem Wert; es kann aber auch irgendetwas anderes sein.

Die `.lex` Datei enthält Regeln der Form

$$regexp \rightarrow \{code\}$$

wobei *code* Java (C, ML)-code ist, der das “Ergebnis” berechnet. Dieses Codefragment kann sich auf den (durch *regexp*) verarbeiteten Text durch spezielle Funktionen und Variablen beziehen, z.B.: `yytext()` und `yypos`. Siehe Beispiel + Doku.

Zustände

- In der `.lex` Datei können auch Zustände angegeben werden:
- Regeln können mit Zuständen beschriftet werden; sie dürfen dann nur in diesem Zustand angewandt werden.
- Im *code* Abschnitt kann durch spezielle Befehle der Zustand gewechselt werden, z.B. `yybegin()`.
- Man verwendet das um geschachtelte Kommentare und Stringliterale zu verarbeiten:
  - `/*` führt in einen “Kommentarzustand”.
  - Kommt `/*` im Kommentarzustand vor, so inkrementiere einen Tiefenzähler.
  - `*/` dekrementiert den Tiefenzähler oder führt wieder in den “Normalzustand” zurück.
  - Erzeuge Fehlermeldung, wenn `/*` oder `*/` unerwartet vorkommen.
  - Ebenso führt `"` (Anführungszeichen) zu einem “Stringzustand”…

Ihre Aufgabe

Schreiben eines Lexers (mit JLex/JFlex) für MiniJava.

- Fehlerausgabe mit Zeile/Spalte (vorgefertigte Klasse auf Homepage).
- Geschachtelte Kommentare, sowie Kommentare bis zum Zeilenende (`//`).
- Dateiende (`EOF`) korrekt behandeln.

Syntaxanalyse

- Zweite Phase der Kompilierung
- Konversion einer Folge von Tokens in einen Syntaxbaum (abstrakte Syntax, AST (*abstract syntax tree*) anhand einer Grammatik.
- Laufzeit linear in der Eingabegröße: Einschränkung auf spezielle Grammatiken (LL(1), LR(1), LALR(1)) für die effiziente Analysealgorithmen verfügbar sind.
- Parsergeneratoren (yacc, bison, ML-yacc, JavaCUP, JavaCC, ANTLR) erzeugen Parser (Syntaxanalysatoren) aus formaler Grammatik. JavaCC, ANTLR verwenden LL(1), die anderen LALR(1).

LL(1)-Grammatiken

Betrachte folgende Grammatik:

1. $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
2. $S \rightarrow \text{begin } S\ L$
3. $S \rightarrow \text{print } E$
4. $L \rightarrow \text{end}$
5. $L \rightarrow ;\ S\ L$
6. $E \rightarrow \text{num} = \text{num}$

Ein Parser für diese Grammatik kann mit der Methode des rekursiven Abstiegs (*recursive descent*) gewonnen werden:

Für jedes Nichtterminalsymbol gibt es eine Funktion, die gegen dieses analysiert:

In C

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();eat(ELSE); S(); b
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();}}
void L(void) {switch(tok) {
        case END:  eat(END); break;
        case SEMI: eat(SEMI); S(); L(); break;
        default:   error();}}
void E(void) { eat(NUM); eat(EQ); eat(NUM); }
```

Manchmal funktioniert das nicht:

$$S \to E\,\$ \qquad E \to E + T \qquad T \to T * F \qquad F \to \mathrm{id}$$
$$E \to E - T \qquad T \to T \,/\, F \qquad F \to \mathrm{num}$$
$$E \to T \qquad T \to F \qquad F \to (\ E\ )$$

```
void S(void) {  E(); eat(EOF);  }
void E(void) {switch(tok) {
        case ?: E(); eat(PLUS); T(); break;
        case ?: E(); eat(MINUS); T(); break;
        case ?: T(); break;
        default:   error(); }}
void T(void) {switch(tok) {
        case ?: T(); eat(TIMES); F(); break;
        case ?: T(); eat(DIV); F(); break;
        case ?: F(); break;
        default:   error();  }}
```

LL(1)-Parsing

- Eine Grammatik $G = (\Sigma, V, P, S)$ heißt LL(1), wenn ein Parse-Algorithmus basierend auf dem Prinzip des rekursiven Abstiegs für sie existiert.
- Nicht jede Grammatik ist LL(1), etwa die aus dem zweiten Beispiel ist es nicht. Problem: der Parser muss schon anhand des ersten Tokens (und der erwarteten linken Seite) in der Lage sein, zu entscheiden, welche Produktion zu wählen ist.

  Manchmal kann man eine Grammatik in die LL(1)-Form bringen. Man kann auch mehr als ein Zeichen weit vorausschauen: LL(k).
- Wie erzeugt man einen LL(1)-Parser automatisch aus der Grammatik?

Die First- und Follow-Mengen

Seien X, Y, Z Nichtterminalsymbole, α, β, γ Folgen von Terminal- und Nichtterminalsymbolen.

- $\mathrm{nullable}(\gamma)$ bedeute, dass $\gamma \rightarrow^* \epsilon$.
- FIRST(γ) ist die Menge aller Terminalsymbole, die als Anfänge von aus γ abgeleiteten Wörtern auftreten.
  (FIRST$(\gamma) = \{a \mid \exists w.\gamma \rightarrow^* aw\}$.
- FOLLOW(X) ist die Menge der Terminalsymbole, die unmittelbar auf X folgen können. (FOLLOW$(X) = \{a \mid \exists \alpha, \beta.S \rightarrow^* \alpha X a \beta\}$).

Berechnung der First- und Follow-Mengen

Die First- (für rechte Seiten von Produktionen) und Follow-Mengen, sowie das Nullable-Prädikat, lassen sich mithilfe der folgenden Regeln iterativ berechnen:

- $\text{FIRST}(\epsilon) = \emptyset, \text{FIRST}(a\gamma) = \{a\}, \text{FIRST}(X\gamma) =$
  if $\text{nullable}(X)$ **then** $\text{FIRST}(X) \cup \text{FIRST}(\gamma)$ **else** $\text{FIRST}(X)$.
- $\text{nullable}(\epsilon) = \mathit{true}$, $\text{nullable}(a\gamma) =$ *false*,
  $\text{nullable}(X\gamma) = \text{nullable}(X)) \wedge \text{nullable}(\gamma)$.

Für jede Produktion $X \rightarrow \gamma$ gilt:

- Wenn $\text{nullable}(\gamma)$, dann auch $\text{nullable}(X)$.
- $\text{FIRST}(\gamma) \subseteq \text{FIRST}(X)$.
- Wenn $\gamma = \alpha Y \beta$ und $\text{nullable}(\beta)$, dann
  $\text{FOLLOW}(Y) \subseteq \text{FOLLOW}(X)$.
- Wenn $\gamma = \alpha Y \beta Z \delta$ und $\text{nullable}(\beta)$, dann
  $\text{FIRST}(Z) \subseteq \text{FOLLOW}(Y)$.

Konstruktion des Parsers

Man tabuliert durch sukzessive Anwendung der Regeln die First- und Follow-Mengen, sowie das Nullable-Prädikat für alle Nichtterminalsymbole. Erweiterung auf Phrasen erfolgt “on-the-fly”.

Soll die Eingabe gegen X geparst werden (also in der Funktion X) und ist das nächste Token a, so kommt die Produktion $X \to \gamma$ infrage, wenn

- $a \in \text{FIRST}(\gamma)$ oder
- $\text{nullable}(\gamma)$ und $a \in \text{FOLLOW}(X)$.

Kommen aufgrund dieser Regeln mehrere Produktionen infrage, so ist die Grammatik nicht LL(1). Ansonsten kann anhand der Regeln der Parser konstruiert werden.

Von LL(1) zu LL(k)

Der LL(1)-Parser entscheidet aufgrund des nächsten Tokens und der erwarteten linken Seite, welche Produktion zu wählen ist. Bei LL(k) bezieht man in diese Entscheidung die k nächsten Token mit ein.

Dementsprechend bestehen die First- und Follow-Mengen aus Wörtern der Länge k und werden dadurch recht groß.

Durch Einschränkung der k-weiten Vorausschau auf bestimmte benutzerspezifizierte Stellen, lässt sich der Aufwand beherrschbar halten. Das passiert bei den Parsergeneratoren ANTLR und JavaCC.

Zusammenfassung LL(1)-Syntaxanalyse

- Das nächste Eingabesymbol und die erwartete linke Seite entscheiden, welche Produktion (für das nächste Eingabesymbol und die folgenden) zu wählen ist.
- Ist diese Entscheidung in eindeutiger Weise möglich, so liegt eine LL(1) Grammatik vor und die Entscheidung lässt sich mithilfe der First- und Follow-Mengen automatisieren.
- Der große Vorteil des LL(1)-Parsing ist die leichte Implementierbarkeit: ist kein Parsergenerator verfügbar (etwa bei Skriptsprachen), so kann ein rekursiver LL(1)-Parser leicht von Hand geschrieben werden. Beachte, die First- und Follow-Mengen müssen ja nur einmal pro Grammatik berechnet werden, was wiederum von Hand geschehen kann.