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:

$$
\begin{array}{llll}
S \to E\,\$ & E \to E + T & T \to T * F & F \to \mathrm{id} \\
 & E \to E - T & T \to T \,/\, F & F \to \mathrm{num} \\
 & E \to T & T \to F & F \to (\ E\ )
\end{array}
$$

```
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 $X \rightarrow \gamma$ 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.

LR-Syntaxanalyse

- LL(1)-Syntaxanalyse hat den Nachteil, dass allein aufgrund des nächsten Tokens die zu verwendende Produktion ermittelt werden muss.
- Bei LR(1)-Syntaxanalyse braucht diese Entscheidung erst gefällt werden, wenn die gesamte rechte Seite einer Produktion (plus ein Zeichen Vorausschau) gelesen wurde.
- Der Parser arbeitet mit einem Keller, dessen Inhalt aus Grammatiksymbolen besteht und (das ist die Invariante) stets den bisher gelesenen Symbolen entspricht (sich also zu denen reduzieren lässt mithilfe der Grammatikproduktionen).
- Enthält der Keller nur das Startsymbol und wurde die gesamte Eingabe eingelesen, so ist die Analyse zuende.

Aktionen des LR(1)-Parsers

Der LR(1)-Parser kann zu jedem Zeitpunkt eine der folgenden beiden Aktionen durchführen:

- Ein weiteres Eingabesymbol lesen und auf den Keller legen (*Shift*-Aktion)
- Eine Produktion auf die oberen Kellersymbole rückwärts anwenden, also den Kellerinhalt $\sigma\gamma$ durch σX ersetzen, falls eine Produktion $X \rightarrow \gamma$ vorhanden ist. (*Reduce*-Aktion)

Eine Grammatik ist per definitionem LR(1), wenn die Entscheidung, welche Aktion durchzuführen ist, allein aufgrund der bisher gelesenen Eingabe, sowie dem nächsten Symbol, getroffen werden kann.

Wann soll man mit $X \to \gamma$ reduzieren?

A: Wenn γ oben auf dem Keller liegt ($\gamma \in (\Sigma \cup V)^*$) und außerdem angesichts der bisher gelesenen Eingabe und des nächsten Symbols keine Sackgasse vorhersehbar ist:

$$L^{\text{Reduce by } X \to \gamma \text{ when } a} = \{\sigma\gamma \mid \exists w \in \Sigma^*. S \to_{\mathbf{rm}} \sigma X a w\}$$

Hier bezeichnet $\to_{\mathbf{rm}}$ die Rechtsableitbarkeit: die Existenz einer Ableitung, in der immer das am weitesten rechts stehende Nichtterminalsymbol ersetzt wird.

Wann soll man das nächste Symbol a einlesen?

A: Wenn es eine Möglichkeit gibt, angesichts des aktuellen Kellerinhalts später eine Produktion anzuwenden.

$$L^{\text{Shift } a} = \{\sigma\alpha \mid \exists w \in \Sigma^*.\exists \text{Produktion} X \to \alpha a \beta.S \to_{\mathbf{rm}} \sigma X w\}$$

Für festes nächstes Eingabesymbol müssen diese Shift- und Reduce-Mengen disjunkt sein, sonst kann eben die erforderliche Entscheidung nicht eindeutig getroffen werden und es liegt keine LR(1)-Grammatik vor.

Das Wunder der LR(1)-Syntaxanalyse

Die Mengen $L^{\text{Reduce by } X \to \gamma \text{ when } a}$ und $L^{\text{Shift } a}$ sind regulär.

Es existiert also ein endlicher Automat, der nach Lesen des Kellerinhaltes anhand des nächsten Eingabesymbols entscheiden kann, ob ein weiteres Symbol gelesen werden soll, oder die obersten Kellersymbole mit einer Produktion reduziert werden sollen und wenn ja mit welcher.

Beispiel

(1)E →E + T (2)E→T

(3)T →T * F (4)T→F

(5)F →id (6)F→(E)

	0	1	2	3	4	5	6	7	8	9	10	11
0												
1	E											
2	T				T							
3	F				F		F					
4	(				(		(	(				
5	id				id		id	id				
6		+							+			
7			*							*		
8					E							
9							T					
10								F				
11									)			
	S	S	R2*	R4	S	R5	S	S	S	R1*	R3	R6

S = Shift. Rx = Reduce mit (x). Rx* = Shift, falls nächstes Symbol *, sonst Reduce mit (x).

Optimisierung: LR(1)-Tabelle

- Annotiere Kellereinträge mit erreichtem Zustand (in der Praxis lässt man die ursprünglichen Kellersymbole ganz weg und arbeitet mit Automatenzuständen als Kelleralphabet).
- Konstruiere Tafel, deren Zeilen mit Zuständen und deren Spalten mit Grammatiksymbolen indiziert sind. Die Einträge enthalten eine der folgenden vier *Aktionen*:

Shift(n**)**	"Shift" und gehe in Zustand n;
Goto(n**)**	Gehe in Zustand n;
Reduce(k**)**	"Reduce" mit Regel k;
Accept	Akzeptiere.

  Leere Einträge bedeuten Syntaxfehler.

Der LR(1)-Algorithmus

- Ermittle Aktion aus der Tabelle anhand des obersten Kellerzustands und des nächsten Symbols.
- Ist die Aktion...

  Shift(n): Lies ein Zeichen weiter; lege Zustand n auf den Keller.

  Reduce(k):
  - Entferne soviele Symbole vom Keller, wie die rechte Seite von Produktion k lang ist,
  - Sei X die linke Seite der Produktion k:
  - Finde in Tabelle unter dem nunmehr oben liegenden Zustand und X eine Aktion "**Goto(n)**";
  - Lege n auf den Keller.

  Accept: Ende der Analyse, akzeptiere die Eingabe.

  Error: Ende der Analyse, erzeuge Fehlermeldung.

Beispiel

(1)E →E + T (2)E→T (3)T →T * F (4)T→F (5)F →id (6)F→(E) (7)S→E$

	0	1	2	3	4	5	6	7	8	9	10	11
0												
1	E											
2	T				T							
3	F				F		F					
4	(				(		(	(				
5	id				id		id	id				
6		+							+			
7			*							*		
8					E							
9							T					
10								F				
11									)			
	S	S	R2*	R4	S	R5	S	S	S	R1*	R3	R6

	id	(	)	+	*	$	F	T	E
0	s5	s4					g3	g2	g1
1				s6		a			
2	r2	r2	r2	r2	s7	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5	s4					g3	g2	g8
5	r5	r5	r5	r5	r5	r5			
6	s5	s4					g3	g9	
7	s5	s4					g10		
8			s11	s6					
9	r1	r1	r1	r1	s7	r1			
10	r3	r3	r3	r3	r3	r3			
11	r6	r6	r6	r6	r6	r6			

Konstruktion des Automaten (und der Tabelle)

Der Automat wird zunächst nichtdeterministisch konzipiert.

Die Zustände haben die Form $(X \to \alpha.\beta, a)$ wobei $X \to \alpha\beta$ eine Produktion sein muss und a ein Terminalsymbol ist.

Solch ein Zustand heißt “LR(1)-Item”.

Die Sprache, die zum Erreichen des Items $(X \to \alpha.\beta, a)$ führen soll, ist:

$$L(X \to \alpha.\beta, a) = \{\gamma\alpha \mid \exists w \in \Sigma^*.S \to_{\mathbf{rm}} \gamma X a w\}$$

also gilt insbesondere:

$$L^{\text{Reduce by } X \to \gamma \text{ when } a} = L(X \to \gamma., a)$$
$$L^{\text{Shift } a} = \bigcup_{X \to \alpha a \beta \text{Produktion}} L(X \to \alpha a \beta., a)$$

Jetzt muss man nur noch die Transitionen so bestimmen, dass tatsächlich diese Sprachen “erkannt” werden:

Transitionen des Automaten

Erinnerung:

$$L(X \to \alpha.\beta, a) = \{\gamma\alpha \mid \exists w \in \Sigma^*.S \to_{\mathbf{rm}} \gamma X a w\}$$

- $(X \to \alpha.s\beta, a) \longrightarrow^{s} (X \to \alpha s.\beta, a)$, $s \in \Sigma \cup V$,
- $(X \to \alpha.Y\beta, a) \longrightarrow^{\epsilon} (Y \to .\gamma, b)$, falls $Y \to \gamma$ und $b \in \text{FIRST}(\beta a)$

Startzustände: $(S \to .\gamma\$, ?)$, wobei $S \to \gamma\$$ die einzige Produktion ist, die S involviert.

Man zeigt durch Induktion, dass der so definierte Automat tatsächlich die gewünschten Sprachen erkennt.

Aktionen

Nunmehr determinisiert man den Automaten und erhält so Mengen von LR(1)-Items als Zustände.

Enthält ein Zustand das Item $(X \to \gamma., a)$ und ist das nächste Symbol a, so reduziert man mit $X \to \gamma$.

Enthält ein Zustand das Item $(X \to \alpha.a\beta, c)$ und ist das nächste Symbol a, so wird ge-Shiftet.

Gibt es mehrere Möglichkeiten, so liegt ein Shift/Reduce, beziehungsweise ein Reduce/Reduce-Konflikt vor und die Grammatik ist nicht LR(1).

LALR(1) und SLR

LR(1)-Tabellen sind recht groß (mehrere tausend Zustände für typische Programmiersprache).

LALR(1) ist eine heuristische Optimierung, bei der Zustände, die sich nur durch die Vorausschau-Symbole unterscheiden, identifiziert werden. Eine Grammatik heißt LALR(1), wenn nach diesem Prozess keine Konflikte entstehen.

Bei SLR wird auf Vorausschau-Symbole in den Items verzichtet, stattdessen verwendet man FOLLOW-Mengen, um Konflikte aufzulösen.

Zusammenfassung LR-Syntaxanalyse

- LR-Parser haben einen Keller von Grammatiksymbolen, der der bisher gelesenen Eingabe entspricht.
- Sie legen entweder das nächste Eingabesymbol auf den Keller (“Shift”) oder wenden auf die obersten Kellersymbole eine Produktion rückwärts an (“Reduce”).
- Die Shift/Reduce-Entscheidung richtet sich nach der bereits gelesenen Eingabe und dem nächsten Symbol. Sie kann durch einen endlichen Automaten vorgenommen werden, der auf dem Keller arbeitet.
- Um Platz zu sparen, werden nur die Automatenzustände auf dem Keller gehalten und jeweils aktualisiert.
- LR-Parser sind allgemeiner und auch effizienter als LL-Parser.
- LALR(1) und SLR sind heuristische Optimierungen von LR(1)

Parsergeneratoren

- Parsergenerator: Grammatikspezifikation → Parser (als Quellcode).
- JavaCUP ist ein LALR(1)-Parsergenerator, der Java-Code generiert.
- Grammatikspezifikationen sind in die folgenden Abschnitte gegliedert:

  Benutzerdeklarationen (z.B. `package` statements, Hilfsfunktionen)

  %%

  Parserdeklarationen (z.B. Mengen von Terminal- / Nichtterminalsymbolen)

  %%

  Produktionen

- Die Produktionen haben die folgende Form (exemplarisch):

  `exp : exp PLUS exp` (*Semantische Aktion*)

  Die “semantische Aktion” wird ausgeführt, wenn die entsprechende Regel “feuert”.

Beispielgrammatik

Stm→*Stm*;*Stm*	(CompoundStm)		
Stm→`id :=`*Exp*	(AssignStm)	*ExpList*→*Exp* `,` *ExpList*	(PairExpList)
Stm→`print (`*ExpList*`)`	(PrintStm)	*ExpList*→*Exp*	(LastExpList)
Exp→`id`	(IdExp)	*BinOp*→`+`	(Plus)
Exp→`num`	(NumExp)	*BinOp*→`-`	(Minus)
Exp→*Exp BinOp Exp*	(OpExp)	*BinOp*→`*`	(Times)
Exp→`(`*Stm* `,` *Exp*`)`	(EseqExp)	*BinOp*→`/`	(Div)

Implementierung in CUP

```
package straight;
import java_cup.runtime.*;

action code  {: <here go helper functions for the actions> :}
parser code  {: <here go helper functions for the parse> :}
scan with    {: return lexer.nextToken(); :};

terminal String ID; terminal Integer INT;
terminal COMMA, SEMI, LPAREN, RPAREN, PLUS, MINUS,
         TIMES, DIVIDE, ASSIGN, PRINT;

non terminal Exp exp;
non terminal ExpList explist;
non terminal Stm prog;

precedence left SEMI;
precedence nonassoc ASSIGN;
```

Implementierung in CUP

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;

start with prog;

prog ::= stm {: :}

exp ::= exp TIMES exp {: :}
       |exp DIVIDE exp {: :}
       |exp PLUS exp {: :}
       |exp MINUS exp {: :}
       |INT {: :}| ID {: :}
       |LPAREN stm COMMA exp RPAREN {: :}
       |LPAREN exp RPAREN {: :}

explist ::= exp {: :}
           |exp COMMA explist {: :}
```

Implementierung in CUP

```
stm ::= stm SEMI stm {: :}
        |PRINT LPAREN explist RPAREN {: :}
        |ID ASSIGN exp {: :}
```

Präzedenzdirektiven

Die obige Grammatik ist mehrdeutig, also insbesondere nicht LR(1).

Präzedenzdirektiven werden hier zur Auflösung der Konflikte verwendet.

Die Direktive

```
%right SEMI COMMA
%left PLUS MINUS
%left TIMES DIV
```

besagt, dass `TIMES, DIV` stärker als`PLUS, MINUS` binden, welche wiederum stärker als `SEMI, COMMA`. `PLUS, MINUS, TIMES, DIV` assoziieren nach links; `SEMI, COMMA` nach rechts.

Also wird $s1; s2; s3, x + y + z * w$ zu $(s1; (s2; s3)), ((x + y) + (z * w))$.

Unäres Minus

Direktiven beziehen sich auf Tokens. Will man sie auf Produktionen beziehen, so führt man ein künstliches Token ein. Beispiel: durch

```
exp: MINUS exp %prec UMINUS
```

erbt diese Produktion die Präzedenz von `UMINUS`. So kann man $-e_1 + e_2$ als $(-e_1) + e_2$ statt $-(e_1 + e_2)$ lesen.

Konflikte

Die verbleibenden Konflikte werden als Shift/Reduce, bzw. Reduce/Reduce-Konflikte gemeldet.

Diese muss man sehr ernst nehmen; in den meisten Fällen deuten sie auf Fehler in der Grammatik hin.

Ein Parser wird auch bei Konflikten generiert; er wird im Zweifel immer Shiften und ansonsten weiter oben geschriebenen Regeln Priorität einräumen (wie beim Lexer).

Ein(ziger) Fall, bei dem solch ein Konflikt sinnvoll ist: “dangling else”.

Typüberprüfung

- Datenstruktur Symboltabelle: funktional und imperativ
- Typüberprüfung in mehreren Durchgängen
- Erzeugung von Zwischencode (erst nächstes Mal)

Symboltabellen

- Eine Symboltabelle bildet Bezeichner ab auf “semantische Werte”.
  - Semantischer Wert eines Programms: Klassennamen mit ihren semantischen Werten.
  - Semantischer Wert einer Klasse: Elternklasse, Feldnamen mit ihren semantischen Werten, Methodennamen mit ihren semantischen Werten.
  - Semantischer Wert eines Feldes: Sein Typ
  - Semantischer Wert einer Methode: Vektor der Parameter mit ihren Typen, Ergebnistyp, lokale Variablen mit ihren Typen.
- Das Wörtchen “mit” wird hier immer durch eine Abbildung (= `Map`), konkret entweder HashMap oder TreeMap realisiert.
- Später werden die semantischen Werte noch um zusätzliche Komponenten erweitert, etwa (relative) Speicheradresse oder Größe (in Byte).

Die Typprüfung

Die Typüberprüfung selbst besteht nun aus einem Satz rekursiver Methoden, welche in Gegenwart einer Symboltabelle ein Programm / Klasse / Methode / Variablendeklaration / . . . auf Typkorrektheit prüfen (unter Bezugnahme auf die Symboltabelle) und als Seiteneffekt diese ergänzen.

Da Klassen und Methoden sich (in MiniJava) gegenseitig rekursiv verwenden dürfen bietet es sich an, die Typüberprüfung in zwei bis drei Phasen zu gliedern:

- In der ersten Phase werden nur die definierten Klassen eingetragen (mit Platzhalter als semantischem Wert). Fehler können auftreten, wenn eine Klasse mehrfach deklariert wird.
- In der zweiten Phase werden die Methoden mit ihrem Aufrufmuster (Parameterliste und Rückgabetyp) eingetragen. Fehler können auftreten, wenn ein undefinierter Typ benutzt wird, oder Methoden- oder Parameternamen doppelt deklariert werden. Ausnahme: Overloading, allerdingsn icht in MiniJava.

Die Typprüfung

- In der dritten Phase werden alle Rümpfe auf Typkorrektheit geprüft und die lokalen Variablen eingetragen. Fehlermöglichkeiten: falsche Typen bei Methodenaufruf, Verwendung undeklarierter lokaler Variablen, etc. pp.

Die erste und zweite Phase können zusammengefasst werden. Man muss dann in der nächsten Phase nochmal prüfen, ob alle verwendeten Typen tatsächlich deklariert wurden.

Geschachtelte Blockstruktur

In MiniJava kann eine lokale Variable nicht von einer anderen lokalen Variablen überschattet werden.

In anderen Sprachen wie C, Pascal, ML können innerhalb eines Rumpfes beliebig geschachtelte Blöcke mit neuen lokalen Variablen auftreten. Beispiel:

```
int main(void){
  int a = 9;
  {float a = 10.0;
    printf("%f",a);
  }
  printf("%d",a);
}
```

Hier muss man einen Stack von lokalen Abbildungen benutzen, der während der Abarbeitung des Programmtextes wächst und schrumpft. Details in Appels Buch.

Typprüfung eines Methodenaufrufs

```
public Type visit(Call n) {
  if (! (n.e.accept(this) instanceof IdentifierType)){
    // Fehler
  }
  String mname = n.i.toString();
  String cname = ((IdentifierType) n.e.accept(this)).s;
  Method calledMethod =
            TypeCheckVisitor.symbolTable.getMethod(mname,cname);
  for ( int i = 0; i < n.el.size(); i++ ) {
      Type t1 =null;
      Type t2 =null;
      if (calledMethod.getParamAt(i)!=null)
          t1 = calledMethod.getParamAt(i).type();
      t2 = n.el.elementAt(i).accept(this);
      if (!TypeCheckVisitor.symbolTable.compareTypes(t1,t2)){
         // Fehler
      }
  }
  return TypeCheckVisitor.symbolTable.getMethodType(mname,cname);
}
```