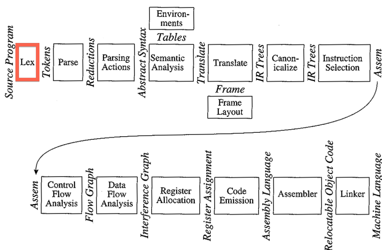
Lexikalische Analyse

Lexikalische Analyse



Lexikalische Analyse

- Erste Phase der Kompilierung
- Die String-Eingabe (Quelltext) wird in eine Folge von *Tokens* umgewandelt.
- Leerzeichen und Kommentare werden dabei ignoriert.
- Die Tokens entsprechen den *Terminalsymbolen* der Grammatik der Sprache, können jedoch auch mit Werten versehen sein.

Was sind Tokens?

Beispiele:	`if`	⟶	IF
	`!=`	⟶	NEQ
	`(`	⟶	LPAREN
	`)`	⟶	RPAREN
	`foo`	⟶	ID("foo")
	`73`	⟶	INT(73)
	`66.1`	⟶	REAL(66.1)
Keine Beispiele:	`/* bla */`		(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

(	$\to$	LPAREN
digit digit*	$\to$	INT$(\mathrm{ConvertToInt}(\mathrm{yytext}()))$
`print`	$\to$	PRINT
$\mathit{letter}(\mathit{letter} + \mathit{digit} + \{_\})^*$	$\to$	ID$(\mathrm{yytext}())$
...		

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

- Tokens werden durch reguläre Ausdrücke spezifiziert.
- Konkrete Eingaben werden mit einem endlichen Automaten erkannt; jeder akzeptierende Zustand stellt das jeweils erkannte Token dar.
- Automatische Lexergeneratoren wie lex, flex, JLex, JFlex, Ocamllex wandeln Spezifikation in Lexer-Programm (als Quelltext) um, das einen entsprechenden Automaten simuliert.

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`	$\to$	ID("print0") *nicht* PRINT INT(0)
`print`	$\to$	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 Spezifikations-Datei einen Lexer in Form einer Java-Klasse (bzw. C-Datei, ML-Modul) mit eine Methode/Funktion `nextToken()`.

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.

Die Spezifikations-Datei enthält Regeln der Form

$$\mathit{regexp} \;\longrightarrow\; \{\mathit{code}\}$$

wobei *code* Java-Code (oder 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.

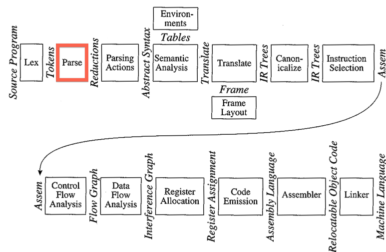
Zustände

In der Spezifikations-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 wie yybegin() der Zustand gewechselt werden.
- 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.
  - Ebenso führt " (Anführungszeichen) zu einem „Stringzustand“...

Syntaxanalyse

Syntaxanalyse



Syntaxanalyse

- Zweite Phase der Kompilierung
- Erkennen einer Folge von Tokens als eine Ableitung einer kontextfreien Grammatik
  - Überprüfung, ob Eingabe syntaktisch korrekt ist in Bezug auf die Programmiersprache
  - Umwandlung in einen Syntaxbaum (abstrakte Syntax, *abstract syntax tree*, AST).

Parsertechniken und -generatoren

- Laufzeit soll linear in der Eingabegröße sein:
  - Einschränkung auf spezielle Grammatiken (LL(1), LR(1), LALR(1)), für die effiziente Analysealgorithmen verfügbar sind.
- Diese Algorithmen (*Parser*) können automatisch aus einer formalen Grammatik erzeugt werden:
  - Parsergeneratoren: z.B. yacc, bison, ML-yacc, JavaCUP, JavaCC, ANTLR.
- Hier vorgestellt:
  - Parsertechniken LL(1), LR(1), LALR(1)
  - Parsergenerator JavaCUP

Wiederholung: Kontextfreie Grammatiken

- Kontextfreie Grammatik: $G = (\Sigma, V, P, S)$
  - $a, b, c, \ldots \in \Sigma$: Terminalsymbole (Eingabesymbole, Tokens)
  - $X, Y, Z, \ldots \in V$: Nichtterminalsymbole (Variablen)
  - P: Produktionen (Regeln) der Form $X \to \gamma$
  - $S \in V$: Startsymbol der Grammatik
- Symbolfolgen:
  - $\alpha, \beta, \gamma \in (\Sigma \cup V)^*$: Folge von Terminal- und Nichtterminalsymbolen
  - $u, v, w \in \Sigma^*$: Wörter (Folge von Nichtterminalsymbolen)
  - ϵ: leeres Wort
- Ableitungen:
  - Ableitung $\alpha X \beta \Rightarrow \alpha\gamma\beta$ falls $X \to \gamma \in P$
  - Links-/Rechtsableitung: $wX\beta \Rightarrow_{\mathbf{lm}} w\gamma\beta$ bzw. $\alpha X w \Rightarrow_{\mathbf{rm}} \alpha\gamma w$
  - Sprache der Grammatik: $L(G) = \{w \mid S \Rightarrow^* w\}$
  - Ableitungsbaum: Symbole aus γ als Unterknoten von X

LL(1)-Syntaxanalyse

Beispiel-Grammatik:

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

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(); break;
        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:

$S \to E\,\$$

$E \to E + T$, $E \to E - T$, $E \to T$

$T \to T * F$, $T \to T\ /\ F$, $T \to F$

$F \to \mathrm{id}$, $F \to \mathrm{num}$, $F \to (\ E\ )$

```
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)-Syntaxanalyse

- Eine Grammatik heißt LL(1), wenn ein Parse-Algorithmus basierend auf dem Prinzip des rekursiven Abstiegs für sie existiert.
- Der Parser muss anhand des nächsten zu lesenden Tokens (und der erwarteten linken Seite) in der Lage sein zu entscheiden, welche Produktion zu wählen ist.
- Die zu wählende Produktion kann zur Optimierung in einer Parser-Tabelle abgespeichert werden.

LL(1)-Parser – Beispiel

Grammatik:
(1) S→F
(2) S→(S+F)
(3) F→a

	(	)	a	+	$
S	2		1		
F			3		

Input: (a+a)

Stack	Input	Aktion
S $	(a + a) $	apply (2) S→(S+F)
(S + F) $	(a + a) $	match '('
S + F) $	a + a) $	apply (1) S→F
F + F) $	a + a) $	apply (3) F→a
a + F) $	a + a) $	match 'a'
+ F) $	+ a) $	match '+'
F) $	a) $	apply (3) F→a
a) $	a) $	apply (3) F→a
) $	) $	match ')'
$	$	match '$' = accept

LL(1)-Syntaxanalyse

- Die Eingabe wird von links nach rechts verarbeitet, dabei wird eine Linksableitung vollzogen, und die Regelauswahl wird anhand des ersten Symbols der verbleibenden Eingabe und des obersten Stacksymbols entschieden.
  - LL(1): **l**eft-to-right parsing, **l**eft-most derivation, 1 token lookahead
- Verarbeitung beginnt beim Startsymbol (Top-down-Ansatz).
- Welche Produktion soll gewählt werden, bzw. wie wird die Tabelle erzeugt?
  - Ansatz: Wenn das nächste Eingabesymbol a ist und X auf Stack liegt (also erwartet wird), kommt diejenige Produktion für X infrage, die zu einer Ableitung mit a an der ersten Stelle führt.

Die First- und Follow-Mengen

- $\mathsf{FIRST}(\gamma)$ ist die Menge aller Terminalsymbole, die als Anfänge von aus γ abgeleiteten Wörtern auftreten:
  $\mathsf{FIRST}(\gamma) = \{a \mid \exists w.\gamma \Rightarrow^* aw\}$
- $\mathsf{FOLLOW}(X)$ ist die Menge der Terminalsymbole, die unmittelbar auf X folgen können:
  $\mathsf{FOLLOW}(X) = \{a \mid \exists \alpha, \beta. S \Rightarrow^* \alpha X a \beta\}$.
- $\mathrm{nullable}(\gamma)$ bedeutet, dass das leere Wort aus γ abgeleitet werden kann:
  $\mathrm{nullable}(\gamma) \iff \gamma \Rightarrow^* \epsilon$.

Berechnung der First- und Follow-Mengen

Iterative Berechnung mithilfe der folgenden Regeln:

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

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

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

Konstruktion des Parsers

Soll die Eingabe gegen X geparst werden (d.h. wird X erwartet) und ist das nächste Token a, so kommt die Produktion $X \to \gamma$ infrage, wenn

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

Die infrage kommenden Produktionen werden in die LL-Tabelle in Zeile X und Spalte a geschrieben.

Kommen aufgrund dieser Regeln mehrere Produktionen infrage, so ist die Grammatik nicht LL(1). Ansonsten spezifiziert die Tabelle den LL(1)-Parser, der die Grammatik erkennt.

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 (z.B. bei ANTLR und JavaCC).

Wenn die Grammatik nicht LL(k) ist

Wenn eine Grammatik nicht LL(k) ist, gibt es dennoch Möglichkeiten, LL-Parser einzusetzen.

- Produktionsauswahl:
  - Kommen mehrere Produktionen infrage, kann man sich entweder auf eine festlegen, oder alle durchprobieren.
  - Damit ist der Parser aber nicht mehr unbedingt „vollständig“ in Bezug auf die Grammatik, d.h. es werden evtl. nicht alle gültigen Ableitungen erkannt.
  - Außerdem muss darauf geachtet werden, dass keine unendliche Rekursion stattfindet (bei linksrekursiven Grammatiken).
- Umformung der Grammatik:
  - Die Grammatik kann u.U. in eine LL(k)-Grammatik umgeformt werden, die die gleiche Sprache beschreibt.
  - Beispiel: Elimination von Linksrekursion.
  - Die Ableitungsbäume ändern sich daduch natürlich.

Zusammenfassung LL(1)-Syntaxanalyse

- **Top-down parsing:** Das nächste Eingabesymbol und die erwartete linke Seite entscheiden, welche Produktion anzuwenden ist.
- Eine **LL(1)-Grammatik** liegt vor, wenn diese Entscheidung in eindeutiger Weise möglich ist. Die Entscheidung lässt sich dann mithilfe der First- und Follow-Mengen automatisieren.
- Der große **Vorteil** des LL(1)-Parsing ist die leichte Implementierbarkeit: auch ohne Parsergenerator kann ein rekursiver LL(1)-Parser leicht von Hand geschrieben werden, denn die Tabelle kann relativ einfach berechnet werden.
- Der **Nachteil** ist, dass die Grammatik vieler Programmiersprachen (auch MiniJava) nicht LL(k) ist.

Parsergenerator JavaCUP

- Parsergenerator: Grammatikspezifikation → Parser-Quellcode
- JavaCUP generiert LALR(1)-Parser als Java-Code (LALR-/LR-Parser werden in der nächsten Woche vorgestellt)
- Grammatikspezifikationen sind in die folgenden Abschnitte gegliedert:
  Benutzerdeklarationen (z.B. package statements, Hilfsfunktionen)
  Parserdeklarationen (z.B. Mengen von Grammatiksymbolen)
  Produktionen
- Beispiel für eine Produktion:
  `exp ::= exp PLUS exp` {: *Semantische Aktion* :}
  Die „semantische Aktion“ (Java-Code) wird ausgeführt, wenn die entsprechende Regel „feuert“. Üblicherweise wird dabei die AST-Repräsentation zusammengesetzt.

Beispielgrammatik

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

Implementierung in CUP

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

parser code  {: <Hilfsfunktionen für den Parser> :}

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 stm;

precedence left SEMI;
precedence nonassoc ASSIGN;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```


Implementierung in CUP

```
start with 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 {: :}
           ;

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


Präzedenzdirektiven

Die obige Grammatik ist mehrdeutig. Präzedenzdirektiven werden hier zur Auflösung der Konflikte verwendet.

Die Direktive

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

besagt, dass `TIMES, DIVIDE` stärker als `PLUS, MINUS` binden, welche wiederum stärker als `SEMI`. Alle Operatoren assoziieren nach links.

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

Ihre heutige Aufgabe (Teil 1)

Generieren und Testen des Straightline-Parsers:

- ZIP-Datei mit Lexer- und Parserspezifikation von Vorlesungsseite herunterladen
- Lexer und Parser mittels JFlex und JavaCUP erzeugen
  - Tools sind am CIP-Pool installiert
  - ZIP-Datei enthält Makefile, das diese Tools aufruft
- erzeugten Parser mit vorgefertigtem Testprogramm testen
  - Testprogramm auf Beispieldatei `example.sl` ausführen
  - optional: Testprogramm mit Eval-Visitor aus letzter Woche zu StraightLine-Interpreter ausbauen

Ihre heutige Aufgabe (Teil 2)

Schreiben eines MiniJava-Lexers und -Parsers:

- MiniJava-Grammatik ist auf Vorlesungsseite verlinkt
- StraightLine-Spezifikationsdateien abändern
- Parser soll zunächst nur akzeptierend sein
  - keine semantischen Aktionen, keine AST-Klassenhierarchie