

## Syntax Analysis (Parser)

41

- Determine syntax (structure) of a program:
  - context-free syntax analysis.
- Construct an intermediate representation.
  - syntax tree
  - symbol table
- Error handling:
  - report error messages with location in source
  - perform error recovery
    - » try to parse as much as possible of the code
    - » try to avoid error cascades
  - maybe also error correction

Compiler Design, WS 2005/2006

## Specification and Recognition of Languages

42

1. Specification of programming language:
  - **Context-free grammar (CFG).**
  - A CFG generates a language:  
The CFG  $\underline{G}$  generates the language  $\underline{L(G)}$ .
2. Recognition of programming language:
  - **Nondeterministic pushdown automata (NPA).**
  - An NPA accepts a language:  
The NPA  $\underline{M}$  accepts the language  $\underline{L(M)}$ .
3. Requirement:  $L(G) = L(M)$

Compiler Design, WS 2005/2006

## Kontextfreie Grammatik

43

**Def.:** Eine kontextfreie Grammatik ist ein Quadrupel  $G=(N,\Sigma,P,S)$  mit

- $N$  ... Menge der Nichtterminalsymbole.
- $\Sigma$  ... Menge der Terminalsymbole.  
Es gilt:  $N \cap \Sigma = \emptyset$  (leere Menge).  
Mit  $V = N \cup \Sigma$  bezeichnet man das Gesamtalphabet.
- $P$  ... Menge von Produktionsregeln von der Form  $\alpha \rightarrow \beta$   
wobei  $\alpha \in N$  und  $\beta \in V^*$ .
- $S \in N$ : Startvariable.

**Notation:**

- $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$  kann zusammengefaßt werden  
zu  $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$ .

Compiler Design, WS 2005/2006

## Ableitung

44

- Als nächstes wollen wir definieren, welche Sprachen eine Grammatik erzeugt. Dazu benötigen wir die Ableitungsrelation  $\Rightarrow$ , die mit Hilfe der Relation  $\rightarrow$  erklärt wird.

- **Def.:** Ableitungsrelation  $\Rightarrow$  über  $V^*$ . Sei  $G=(N,\Sigma,P,S)$ .

$$\alpha \Rightarrow \beta \text{ gdw } \alpha = \alpha_1 A \alpha_3 \text{ und } \beta = \alpha_1 \alpha_2 \alpha_3$$

mit  $A \in N$  und  $A \rightarrow \alpha_2 \in P$ .

Man sagt  $A$  leitet  $\alpha_2$  direkt ab, oder  $\alpha_2$  kann aus  $A$  direkt abgeleitet werden.

- **Def.:** Ausgehend von  $\Rightarrow$  ist  $\overset{*}{\Rightarrow}$  wie folgt definiert:

$$\alpha \overset{*}{\Rightarrow} \beta \text{ gdw } \alpha_1, \alpha_2, \dots, \alpha_n \in V^* \text{ existiert, sodass}$$

$$\alpha = \alpha_1, \beta = \alpha_n \text{ und } \alpha_1 \Rightarrow \alpha_2, \alpha_2 \Rightarrow \alpha_3, \dots, \alpha_{n-1} \Rightarrow \alpha_n \in P.$$

Compiler Design, WS 2005/2006

## Linksableitungen und Rechtsableitungen

45

- Wenn bei jedem Ableitungsschritt immer die am weitesten links stehende Variable ersetzt wird, sprechen wir von einer **Linksableitung**. Analog wird bei einer **Rechtsableitung** immer die am weitesten rechts stehende Variable ersetzt, wie die folgende Definition besagt.

**Def.:** Sei  $G = (N, \Sigma, P, S)$  und gelte  $\alpha \Rightarrow \beta$  mit  $\alpha = \alpha_1 A \alpha_3$ ,  $\beta = \alpha_1 \alpha_2 \alpha_3$ , und  $A \rightarrow \alpha_2 \in P$ .

1.  $\alpha \Rightarrow \beta$  heißt ein **Linksableitungsschritt**, in Zeichen  $\alpha \xrightarrow{L} \beta$ , gdw  $\alpha_1 \in \Sigma^*$ .
2.  $\alpha \Rightarrow \beta$  heißt ein **Rechtsableitungsschritt**, in Zeichen  $\alpha \xrightarrow{R} \beta$ , gdw  $\alpha_3 \in \Sigma^*$ .
3. Eine Ableitung heißt **Linksableitung**, wenn jeder Schritt ein Linksableitungsschritt ist.
4. Eine Ableitung heißt **Rechtsableitung**, wenn jeder Schritt ein Rechtsableitungsschritt ist.

Compiler Design, WS 2005/2006

## Eindeutigkeit

46

**Def.:** Eine kontextfreie Grammatik  $G$  heißt **eindeutig**, falls es zu jedem Satz  $w \in L(G)$  genau eine Linksableitung gibt bzw. genau eine Rechtsableitung gibt. Sonst heißt  $G$  **mehrdeutig**.

**Satz:** Sei  $G = (N, \Sigma, P, S)$  eine kontextfreie Grammatik und  $w \in \Sigma^*$ . Falls  $S \xRightarrow{*} w$ , dann auch  $S \xrightarrow{*L} w$  bzw.  $S \xrightarrow{*R} w$

D.h. jedes Wort das ableitbar ist, ist auch als Linksableitung bzw. als Rechtsableitung ableitbar.

Compiler Design, WS 2005/2006

## Erzeugte Sprache

47

Sei  $G=(N,\Sigma,P,S)$ .

**Def.:** Satzform, Satz.

Ein Wort  $\alpha \in V^*$  wird Satzform genannt, falls gilt  $S \Rightarrow^* \alpha$ .  
Eine Satzform  $\alpha$  mit  $\alpha \in \Sigma^*$  heißt Satz von  $G$ .

**Def.:**  $L(G)$ .

Mit  $L(G)$  bezeichnen wir die Sprache, die durch die Grammatik  $G$  erzeugt wird.  $L(G)$  ist folgende Wortmenge:

$$L(G) = \{w \mid w \in \Sigma^* \text{ und } S \Rightarrow^* w\}.$$

- Eine formale Sprache heißt kontextfrei (regulär, etc.) gdw es eine kontextfreie (reguläre, etc.) Grammatik gibt, die diese Sprache erzeugt. Zwei Grammatiken  $G_1$  und  $G_2$  heißen äquivalent, wenn  $L(G_1)=L(G_2)$ . Zu jeder Grammatik gibt es unendlich viele äquivalente Grammatiken.

Compiler Design, WS 2005/2006

## Parse Tree

48

Parse tree over a grammar  $G$  has the following properties:

1. Root node is labeled with start symbol  $S$ .
2. Each leaf node is labeled with a terminal symbol or with  $\epsilon$ .
3. Each nonleaf node is labeled with a nonterminal.
4. If a node labeled with nonterminal  $X$  has  $n$  successor nodes  $X_1, X_2, \dots, X_n$  (from left to right), then grammar  $G$  contains the following production:

$$X \rightarrow X_1 \mid X_2 \mid \dots \mid X_n$$

- A left-to-right traversal of the leaf nodes yields sentence  $w \in L(G)$ .

Compiler Design, WS 2005/2006

## Beispiel: Ableitung und Ableitungsbaum

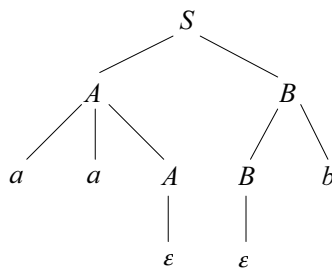
49

-  $G = (\{S, A, B\}, \{a, b\}, P, S)$  mit  
 $P = \{ S \rightarrow AB, A \rightarrow aaA \mid \varepsilon, B \rightarrow Bb \mid \varepsilon \}$ .

- Ableitung:

$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaABb \Rightarrow aaBb \Rightarrow aab$

- Ableitungsbaum:



Compiler Design, WS 2005/2006

## Ambiguity: Dangling Else Problem (1)

50

ISO C Standard:

$stmt \xrightarrow{1} \text{if} ( exp ) stmt$

$stmt \xrightarrow{2} \text{if} ( exp ) stmt \text{ else } stmt$

Example:

`if (0) if (1) stmt1 else stmt2`

Two leftmost derivations, i.e. grammar is ambiguous:

1.  $stmt \xrightarrow{2} \text{if} ( exp ) stmt \text{ else } stmt$   
 $\xrightarrow{*} \text{if} (0) \text{if} (exp) stmt \text{ else } stmt$

2.  $stmt \xrightarrow{1} \text{if} ( exp ) stmt$   
 $\xrightarrow{*} \text{if} (0) \text{if} (exp) stmt \text{ else } stmt$

Program:

```
1) if (0)
    if (1)
        stmt1
    else
        stmt2
```

```
2) if (0)
    if (1)
        stmt1
    else
        stmt2
```

Compiler Design, WS 2005/2006

## Ambiguity: Dangling Else Problem (2)

51

### Ambiguous grammar:

- incomplete specification of the syntax of a language.
- causes problems for parsing.

### Approaches to deal with ambiguous grammars:

1. Change grammar to remove ambiguity.  
Grammar reflects syntactic structure of the language at the price of possibly more complex grammar rules.
2. Specify **disambiguating rule**.  
Ambiguity is corrected without changing (and possibly complicating) grammar at the price that syntactic structure of the language is no longer given by the grammar alone.

Compiler Design, WS 2005/2006

## Ambiguity: Dangling Else Problem (3)

52

### Approach 1:

```
stmt → matched_stmt | unmatched_stmt
matched_stmt → if ( exp ) matched_stmt else matched_stmt
               | other
unmatched_stmt → if ( exp ) stmt
                | if ( exp ) matched_stmt else unmatched_stmt
```

### Approach 2:

cf. ISO C Standard:

An else is associated with the lexically immediately preceding else-less if that is in the same block (but not in an enclosed block).

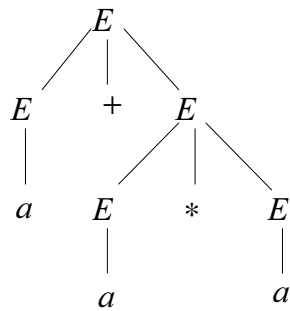
**Comment:** Usually Approach 2 taken, since **most closely nested rule** for dangling else problem can be easily realized by parsing methods (however parser generators will report a conflict).

Compiler Design, WS 2005/2006

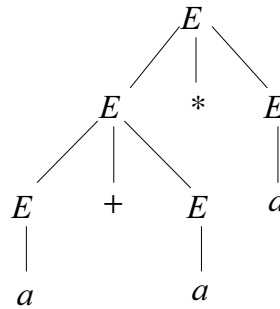
## Ambiguity: Expressions (1)

53

- $G = (\{E\}, \{+, *, (, ), a\}, P, E)$  with  
 $P = \{E \rightarrow E + E \mid E * E \mid (E) \mid a\}$
- Two parse trees for  $a+a*a$ :



$a=2$ : result=6



result=8

Compiler Design, WS 2005/2006

## Ambiguity: Expressions (2) - Precedence and Associativity

54

Usually: precedence and associativity expressed by grammar.

### Precedence cascade:

- group operators into groups of equal precedence.
- write a rule for each precedence level.
- operators with lower precedence "higher" in the syntax tree (closer to the root) and will be evaluated later.

Compiler Design, WS 2005/2006

## Ambiguity: Expressions (3)

55

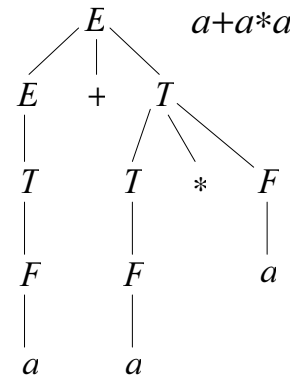
- To eliminate ambiguity and to reflect precedence, the grammar is rewritten in the following way:

$G = (\{E, T, F\}, \{+, *, (, ), a\}, P, E)$  mit

$$P = \{E \rightarrow E + T, E \rightarrow T, T \rightarrow T * F, T \rightarrow F, F \rightarrow (E), F \rightarrow a\}$$

Now derivation of  $a+a*a$  results in the correct derivation tree.

$$\begin{aligned} E &\Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \\ &\Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \\ &\Rightarrow a + a * a \end{aligned}$$



Compiler Design, WS 2005/2006

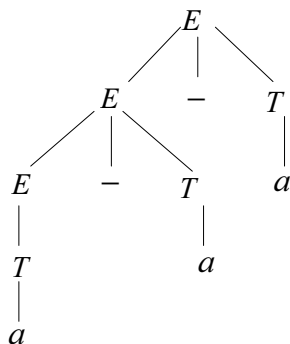
## Ambiguity: Expressions (4) - Associativity

56

1. Left recursive rule results in left associative operator.
2. Right recursive rule results in right associative operator.

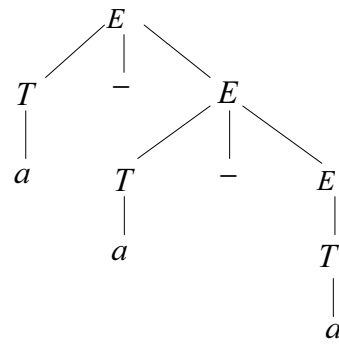
**Example:**

$$P = \{E \rightarrow E - T \mid T, T \rightarrow a\}$$



left - associative :  $a = 2, result = -2$

$$P = \{E \rightarrow T - E \mid T, T \rightarrow a\}$$



right - associative :  $a = 2, result = 2$

Compiler Design, WS 2005/2006

## Notations

57

### 1. BNF (Backus-Naur Form), CFG rules of the form:

$$\text{exp} \rightarrow \text{exp} + \text{exp} \mid \text{exp} * \text{exp} \mid ( \text{exp} ) \\ \mid \text{number}$$

- John Backus, Peter Naur, used for Algol60

### 2. EBNF (Extended BNF)

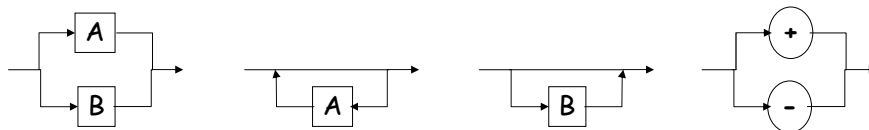
- repetition: curly brackets  $\{ \dots \}$

$$A \rightarrow aA \mid b = A \rightarrow \{a\}b$$

- optional constructs: square brackets  $[ \dots ]$

$$\text{if-stmt} \rightarrow \text{if} ( \text{exp} ) \text{stmt} [ \text{else} \text{stmt} ]$$

### 3. Syntax Diagrams



Compiler Design, WS 2005/2006

## Top-Down Parsing

58

- Top-down parsing:
  - parse tree constructed from root to leaves.
  - find leftmost derivation for an input string.
- Top-down parsing algorithms:
  - recursive-descent parsing.
  - LL(1) parsing.
- Common parsing approach:
  - given current input symbol  $a$  and nonterminal  $A$  in the parse tree, select one production of  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  to derive a string beginning with  $a$ .
- Common problems:
  - left recursion.
  - common prefix for alternatives.

Compiler Design, WS 2005/2006

## First and Follow Sets

59

**Definition:** First set.

Let  $\alpha$  be any string of grammar symbols, then:

$$\text{First}(\alpha) = \{a \mid a \in \Sigma \text{ and } \alpha \Rightarrow^* a\beta\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}.$$

**Definition:** Follow set.

Let  $A$  be a nonterminal and  $\$$  the input endmarker, then:

$$\text{Follow}(A) = \{a \mid a \in \Sigma \text{ and } S \Rightarrow^* \alpha A a \beta\} \cup \{\$ \mid S \Rightarrow^* \alpha A\}.$$

Note: Sets defined for lookahead  $k=1$ .

Compiler Design, WS 2005/2006

## Computation First Sets

60

**Algorithm:** Compute  $\text{First}(X)$  for grammar symbols  $X$ .

Apply the following 3 rules until the First sets do not change any more:

1. If  $X$  is a terminal, then  $\text{First}(X) = \{X\}$ .
2. If  $X \rightarrow \varepsilon$  is a production, then add  $\varepsilon$  to  $\text{First}(X)$ .
3. If  $X$  is nonterminal with  $X \rightarrow Y_1 Y_2 \dots Y_i \dots Y_k$ , then:
  - add  $a$  to  $\text{First}(X)$  if
    - $a \in \text{First}(Y_i)$  and
    - $\varepsilon \in \text{First}(Y_1) \wedge \dots \wedge \varepsilon \in \text{First}(Y_{i-1})$
  - add  $\varepsilon$  to  $\text{First}(X)$  if  $\varepsilon \in \text{First}(Y_1) \wedge \dots \wedge \varepsilon \in \text{First}(Y_k)$

Compiler Design, WS 2005/2006

## Computation Follow Sets

61

**Algorithm:** Compute Follow(A) for nonterminals A.

Apply the following 3 rules until the Follow sets do not change any more:

1. If S is the start symbol \$ the input endmarker, then add \$ to Follow(S).
2. If there is a production  $B \rightarrow \alpha A \gamma$ , then add  $\text{First}(\gamma) - \{\epsilon\}$  to Follow(A).
3. If there is a production  $B \rightarrow \alpha A$  or a production  $B \rightarrow \alpha A \gamma$  with  $\epsilon \in \text{First}(\gamma)$ , then add Follow(B) to Follow(A).

Compiler Design, WS 2005/2006

## Left Recursion Removal

62

1. Immediate left recursion:  
Productions of the following form with none of the  $\beta_1, \dots, \beta_m$  starting with A

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

are rewritten in the following way

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \epsilon$$

2. General left recursion:  
E.g.  $S \rightarrow Aa \mid b$ ,  $A \rightarrow Ac \mid Sd \mid \epsilon$ .  
Above approach does not work, algorithm to eliminate general left recursions exists.

**Note:** Left recursion has been used to make operations left associative!

Compiler Design, WS 2005/2006

## Left Factoring

63

- Productions of the following form where  $\gamma$  represents all alternatives that do not start with  $\alpha$

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

are rewritten in the following way

$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

Compiler Design, WS 2005/2006

## Recursive-Descent Parsing

64

Idea of recursive-descent parsing simple:

Consider production  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ .

- production viewed as definition of a procedure  $A$ .
- right-hand side  $\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  specifies structure of procedure body.
- terminal on rhs:
  - » match terminal with input and advance the input.
  - » if terminal does not match: syntax error.
- nonterminal on rhs:
  - » call of corresponding procedure.

Compiler Design, WS 2005/2006

## Recursive-Descent Parsing: Example

65

Production:  $factor \rightarrow ( exp ) \mid number$

Code:

```
procedure factor()
begin
  case token of
    '(' : match('(');
          exp();
          match(')');
    'number' :
          match('number');
    default:
          error();
  end case;
end factor
```

```
procedure match (expectedToken)
begin
  if (token==expectedToken) then
    getToken();
  else
    error();
  end if;
end match
```

Compiler Design, WS 2005/2006

## LL(1) Parsing

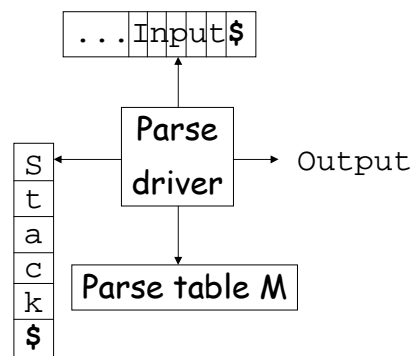
66

LL(1):

- input scanned from Left to right.
- Leftmost derivation.
- 1-token lookahead.

**Table-driven approach:**

- Parse table contains parsing actions.
- Parse driver fetches actions from parse table.
- Use of explicit stack.
- \$ as end symbol for stack and input.



Compiler Design, WS 2005/2006

## Example:

67

1. Production:  $S \rightarrow (S)S \mid \epsilon$
2. Parsing actions:

	Parsing stack	Input	Action
1	\$ S	()\$	$S \rightarrow (S)S$
2	\$ S)S(	()\$	match
3	\$ S)S	)\$	$S \rightarrow \epsilon$
4	\$ S)	)\$	match
5	\$ S	\$	$S \rightarrow \epsilon$
6	\$	\$	accept

Compiler Design, WS 2005/2006

## Parse Driver

68

- Let  $A$  be top of stack,  $x$  next input token, and  $M$  the parse table.

```
push start symbol S onto top of stack; // initialization
while (A ≠ $ and x ≠ $) do
  if (A terminal and A = x) then
    pop parsing stack; // match
    advance input;
  else
    if (A nonterminal and  $A \rightarrow B_1 B_2 \dots B_n \in M[A, x]$ ) then
      pop A from stack; // replace A by rhs
      push  $B_n, \dots, B_2, B_1$  onto stack;
    else
      error;
end while;
if (A = $ and x = $) then
  accept;
else
  error;
```

Compiler Design, WS 2005/2006

## Construction of Parse Table M

69

- Input: Grammar G.
- Output: Parse Table  $M[A,x]$  with A...nonterminal, x...terminal  $\cup$  \$, and  $M[A,x] = \{ \text{production rule or error} \}$ .
- Algorithm:

For each production  $A \rightarrow \alpha$  of grammar do:

1. for each terminal  $a \in \text{First}(\alpha)$  :  
add  $A \rightarrow \alpha$  to  $M[A, a]$ .
2. if ( $\epsilon \in \text{First}(\alpha)$ ) then:
  - a) for each terminal  $b \in \text{Follow}(A)$  :  
add  $A \rightarrow \alpha$  to  $M[A, b]$ .
  - b) if ( $\$ \in \text{Follow}(A)$ ) then:  
add  $A \rightarrow \alpha$  to  $M[A, \$]$ .

Each undefined entry of M: set to error.

Compiler Design, WS 2005/2006

## LL(1)-Grammar

70

**Def.:** A grammar is an LL(1) grammar, if the associated LL(1) parse table has at most one production in each table entry.

Remark: LL(1) grammar cannot be ambiguous.

**LL(k) grammars:**

- extending lookahead to k symbols.
- uncommon in practice.
- a non-LL(1) grammar is likely to be also non-LL(k) for any k.

Compiler Design, WS 2005/2006

## Example if-statement

71

### Grammar for if-statement

- 1:  $stmt \rightarrow if-stmt$
- 2:  $stmt \rightarrow other$
- 3:  $if-stmt \rightarrow if ( exp ) stmt else-part$
- 4:  $else-part \rightarrow else stmt$
- 5:  $else-part \rightarrow \epsilon$
- 6:  $exp \rightarrow 0$
- 7:  $exp \rightarrow 1$

### First and Follow sets:

- 1:  $First(if-stmt) = \{if\}$ , 2:  $First(other) = \{other\}$ ,  
3:  $First(if\dots) = \{if\}$ , 4:  $First(else\dots) = \{else\}$ ,  
5:  $First(\epsilon) = \{\epsilon\}$ , 6:  $First(0) = \{0\}$ , 7:  $First(1) = \{1\}$ ,  
 $Follow(else-part) = \{\$, else\}$ .

Compiler Design, WS 2005/2006

## Example if-statement (cont'd)

72

- Parse table  $M[A,x]$ : (blanks are error entries)

$M[A,x]$	if	other	else	0	1	\$
<i>stmt</i>	1	2				
<i>if-stmt</i>	3					
<i>else-part</i>			4,5			5
<i>exp</i>				6	7	

Compiler Design, WS 2005/2006

## Example if-statement (cont' d)

73

### Disambiguating rule:

Prefer the rule that generates the current lookahead token.

- in our example:  $M[\textit{else-part}, \textit{else}] = \{ 4, 5 \} \rightarrow \{ 4 \}$   
(i.e. delete  $\textit{else-part} \rightarrow \epsilon$ ).
- parsing now unambiguous - as if it were LL(1) grammar.
- Modification realizes **most closely nested rule!**

Compiler Design, WS 2005/2006

## Top-down Parsing and Syntax Tree Construction

74

- Produce syntax tree rather than parse tree.
- **Problem:** syntax tree obscured by left factoring and left recursion removal.
- Recursive descent parsing:
  - syntax tree construction relatively easy.
- LL(1) parsers:
  - syntax tree construction more difficult.
  - extra value stack required.
- Bottom-up parsers:
  - syntax tree construction relatively easy.
  - one reason to be preferred as table-driven stack-based parsing method.

Compiler Design, WS 2005/2006

## Program Errors and Parsers

75

- Minimal requirement:
  - Parsers must report errors.
- Parsers shall produce meaningful error messages and attempt to determine location as closely as possible.
- Error recovery:
  - should be supported (common in practice).
- Error correction:
  - rare feature in actual compilers.

Compiler Design, WS 2005/2006

## Error Recovery During Parsing

76

- Issues in case of an error:
  - try to parse as much as possible of the code.
  - try to avoid error cascades.
- Strategy **Panic Mode** (better: *don't panic mode*):
  - In case of error: parsing resumed at **synchronizing tokens** (tokens are skipped until synchronizing tokens are recognized).
  - Specification of synchronizing tokens:
    - » Follow sets are used.
    - » First sets also important to reduce skipping (i.e. resume at major constructs like statements).

Compiler Design, WS 2005/2006

## Bottom-Up Parsing

77

- Bottom-up parsing:
  - parse tree constructed from leaves to root.
  - find rightmost derivation for an input string.
- Bottom-up parsing algorithms:
  - SLR(1) / LALR(1) / LR(1) - parsing.
- Common parsing approach:
  - "shift-reduce parsing": "reduce" an input string to the start symbol of the grammar.
- Bottom-up methods more powerful than top-down methods (e.g. no problem with left recursion).

**However:** bottom-up methods are more complex too.

Compiler Design, WS 2005/2006

## LR Parser

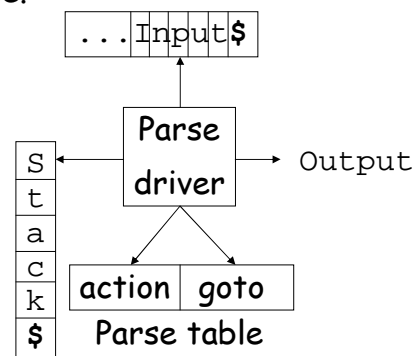
78

### LR(1):

- input scanned from Left to right.
- Rightmost derivation in reverse.
- 1-token lookahead.

### Table-driven approach:

- Parse table contains actions/goto.
- Parse driver fetches instructions from parse table.
- Use of explicit stack.
- \$ as end symbol for stack and input.



Compiler Design, WS 2005/2006

## Example: Intuitively

79

Two main actions of bottom-up parser:

1. **Shift** terminal from input to stack.
2. **Reduce** string  $\alpha$  at top of stack to nonterminal  $A$  given  $A \rightarrow \alpha$ .

**Grammar:**  $E' \rightarrow E$ , **Input:**  $n+n$   
 $E \rightarrow E+n \mid n$  **Derivation:**  $E' \Rightarrow E \Rightarrow E+n \Rightarrow n+n$

	Parsing stack	Input	Action
1	\$	$n+n$ \$	shift
2	\$ n	$+n$ \$	reduce $E \rightarrow n$
3	\$ E	$+n$ \$	shift
4	\$ E+	$n$ \$	shift
5	\$ E+n	\$	reduce $E \rightarrow E+n$
6	\$ E	\$	reduce $E' \rightarrow E$
7	\$ E'	\$	accept

Compiler Design, WS 2005/2006

## LR Parser

80

- **Configuration** of an LR parser is given by the pair  $\langle \text{stack contents, unexpanded input} \rangle$ .
- A configuration  $\langle s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$ \rangle$  represents the right sentential form  $X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$  with symbols  $X_1 X_2 \dots X_m$  on the parser stack being called viable prefix.
- **Contents of stack:**  $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$  with  $s_0 \dots$  bottom of stack,  $s_i \dots$  states,  $X_i \dots$  grammar symbols.

Compiler Design, WS 2005/2006

## LR Parsing Technique

81

- Parsing table: **action** function and **goto** function.
- An action table entry can be:
  1.  $s_i$  for **shift**  $i$  where  $i$  is a state.
  2.  $r_i$  for **reduce** by production  $i$ :  $A \rightarrow \beta$ .
  3. **accept**, and
  4. **error**.
- Action table is indexed by  $\text{action}[s_m, a_i]$  where  $s_m$  is the topmost state on the stack and  $a_i$  is the next input symbol.
- Parser driver the same for all LR parsers (SLR(1), LALR(1), LR(1)): only the parsing table changes.

Compiler Design, WS 2005/2006

## LR Parsing Technique (cont'd)

82

Suppose parser configuration

$$\langle s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$ \rangle.$$

Then the four actions are given as follows:

1. if "action[ $s_m, a_i$ ]=shift  $s$ ", the new configuration is  $\langle s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$ \rangle$
2. if "action[ $s_m, a_i$ ]=reduce  $A \rightarrow \beta$ ", the new configuration is  $\langle s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} \underline{A} s, a_i a_{i+1} \dots a_n \$ \rangle$   
where:  $r$  = length of  $\beta$ , and  
 $s$  = goto[ $s_{m-r}, A$ ].

Note:  $2xr$  symbols are popped off the stack ( $r$  state symbols and  $r$  grammar symbols).

3. if "action[ $s_m, a_i$ ]=accept", parsing is completed.
4. if "action[ $s_m, a_i$ ]=error", parser has discovered an error and calls an error recovery routine.

Compiler Design, WS 2005/2006

## LR Parse Driver

83

```
parser has initial state  $s_0$  on its stack and ip points to
the first symbol of  $w\$$ ;
repeat forever
  let  $s$  be the state on top of the stack and
  a the symbol pointed to by ip;
  if (action[s,a]=shift  $s'$ ) {
    push a then  $s'$  on top of the stack;
    advance ip to the next input symbol; }
  else
    if (action[s,a]=reduce  $A \rightarrow \beta$ ) {
      pop  $2*|\beta|$  symbols off the stack;
      let  $s'$  be the state now on top of the stack;
      push  $A$  and then goto[s', A] on top of the stack; }
    else
      if (action[s,a]=accept)
        return;
      else
        error;
end
```

Compiler Design, WS 2005/2006

## Construction of Parsing Tables

84

Three methods of varying power and efficiency:

1. SLR(1) parsing ... Simple LR(1) parsing
2. LALR(1) parsing ... LookAhead LR(1) parsing
3. LR(1) parsing

**Def.:** A grammar is an SLR(1)/LALR(1)/LR(1) grammar, if the associated SLR(1)/LALR(1)/LR(1) parse table has no multiply defined entries.

For grammars the following holds:  $SLR(1) \subset LALR(1) \subset LR(1)$ .

Compiler Design, WS 2005/2006

## LR(0) items

85

**Def.:** An LR(0) item of a grammar is a production with a distinguished position in its right-hand side denoted by a period "." (which is a metasymbol and no token).

**Example:** Production  $A \rightarrow aB$  gives the LR(0) items

- $A \rightarrow .aB$ ,  $A \rightarrow a.B$ ,  $A \rightarrow aB.$

**Intuition:** Item  $A \rightarrow \alpha . \beta$  denotes that in the parsing process

- we have just seen on the input a string derivable from  $\alpha$
- we hope next to see a string derivable from  $\beta$

**Idea:** LR(0) items are used as states of a DFA that maintains information about the progress of parsing.

Compiler Design, WS 2005/2006

## LR(0) items: closure operation and goto

86

**Def.:** If  $I$  is a set of items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of items constructed by the two rules:

1. Initially, every item in  $I$  is added to  $\text{closure}(I)$ .
2. If  $A \rightarrow \alpha . B \beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production, then add  $B \rightarrow . \gamma$  to  $I$  (if it is not already there).  
Apply this rule until no new items can be added to  $\text{closure}(I)$ .

**Def.:**  $\text{goto}(I, B)$  with  $I$  denoting a set of items and  $B$  a grammar symbol is defined as follows: If  $A \rightarrow \alpha . B \beta$  is in  $I$ , then  $\text{goto}(I, B)$  is defined to be the closure of the set of all items  $A \rightarrow \alpha B . \beta$ ;

Compiler Design, WS 2005/2006

## Viable Prefix DFA for LR(0) Items

87

Given grammar  $G=(N,\Sigma,P,S)$  with start symbol  $S$ .

- Construct *augmented grammar*  $G'$  by adding  $S' \rightarrow S$  with  $S' \notin (N \cup \Sigma)$ .
- Algorithm for constructing the collection  $C$  of LR(0) items representing the set of states of the viable prefix DFA for  $G'$ :

```
begin
  C := { closure([S'→.S]) };
  repeat
    for (each set of items I in C and
         each grammar symbol X) do
      if (goto(I,X) ≠ ∅ and goto(I,X) ∉ C) then
        add goto(I,X) to C;
      end if
    end for
  until no new set of items can be added to C;
end
```

Compiler Design, WS 2005/2006

## Construction SLR(1) Parse Table

88

**Input:** An augmented grammar  $G'$ .

**Output:** SLR(1) parse table functions action and goto for  $G'$ .

**Algorithm:**

1. Construct  $C=\{ I_0, I_1, \dots, I_n \}$  the set of states of the viable prefix DFA for  $G'$ .
2. Parsing actions for state  $i$  are determined as follows:
  - a) if  $(A \rightarrow \alpha . a \beta \in I_i, a \text{ a terminal, and } goto(I_i, a)=I_j)$ :  
set  $action[i, a] := \text{"shift } j"$ ;
  - b) if  $(A \rightarrow \alpha . \in I_i, A \neq S')$ :  
for each  $a \in Follow(A)$  do  
set  $action[i, a] := \text{"reduce } A \rightarrow \alpha"$  ;  
end for
  - c) if  $(S' \rightarrow S . \in I_i)$ :  
set  $action[i, \$] := \text{"accept"}$ ;

Compiler Design, WS 2005/2006

## Construction SLR(1) Parse Table (cont' d)

89

Algorithm: (cont' d)

3. goto transition for state  $i$  and all nonterminals  $A$ :  
if ( $goto(I_i, A)=I_j$ ) then  $goto[i,A]=j$ ;
4. All entries not defined by above steps are set to "error".
5. Initial state of parser: constructed from set of items containing  $S' \rightarrow . S (I_0)$

Compiler Design, WS 2005/2006

## Exercise SLR(1)-Parsing

90

Consider the grammar:

$$G = (\{S, I\}, \{e, i, o\}, P, S) \text{ with}$$
$$P = \{S \rightarrow I \mid o,$$
$$I \rightarrow i S \mid i S e S\}.$$

Note that the grammar corresponds to a language with an if-statement, if we assume the following abbreviations:  
 $S$  ... *statement*,  $I$  ... *if-statement*,  $o$  ... *other (non-if) statements*,  $i$  ... *keyword if*,  $e$  ... *keyword else*.

- a) Construct the LR(0) items for this grammar.
- b) Construct the SLR(1) parse table (action and goto).
- c) Is the grammar an SLR(1) grammar? If not, what can be done to enable parsing?
- d) Show the parsing stack and the actions of an SLR(1) parser for the input string: **ii~~o~~ee~~o~~**.

Compiler Design, WS 2005/2006

## LR(1)-Parsing ( also: canonical LR(1) )

91

- **Idea:** SLR(1)-parsing extended by lookahead token.
- Based on **LR(1) items**:  
 $[A \rightarrow \alpha . \beta , a ]$   
where  $A \rightarrow \alpha . \beta$  is an LR(0) item and  $a \in \text{Follow}(A)$ , i.e.  $a$  is the lookahead token.
- Method for constructing the collection of sets of LR(1) items (viable prefix DFA for LR(1) items) is essentially the same as for LR(0) items.
  - Operations *closure* and *goto*,
  - algorithm for creating collection  $C$ , and
  - construction of LR(1) parse table  
must be adopted appropriately.
- Parser driver is not changed.

Compiler Design, WS 2005/2006

## LALR(1)-Parsing

92

- **Observation:** Many of the states in  $C$  differ only by the lookahead token, the lookahead token is important only for reductions.
- **Idea:** Merge states with LR(1) items which differ only in the lookahead token.
- **Table size:** SLR- and LALR-tables have same number of states.
- Merger can produce a **reduce/reduce conflict**, but not a shift/reduce conflict if the original grammar was an LR(1) grammar.
- LALR(1) parse table can be created without full construction of sets of LR(1) items, i.e. can be done **more efficient** than the pictorial merger approach.

Compiler Design, WS 2005/2006

## SLR(1) / LALR(1) / LR(1) - Parsing

93

- The number of states of a parse table for a programming language is typically
  - for SLR and LALR in the range of several hundred states,
  - for LR in the range of several thousand states.
- Generally LR(1)-parsing considered too expensive in practice.
- Widespread: LALR(1)-parsing (used by Yacc / Bison) .
- Bottom-up parsing: left-recursion more efficient than right-recursion.

Compiler Design, WS 2005/2006

## Error Recovery in LR Parsing

94

Panic-Mode Error Recovery:

- Parser determines by consulting parse table that string derivable from  $A$  contains an error.
  - Processed part of string on stack - remainder of the string on the input.
    1. Scan down the stack to a state  $s_i$  with  $\text{goto}[s_i, A]=s_j$ .
    2. Discard input tokens until a token in  $\text{Follow}(A)$  is found.
    3. Push state  $s_j$  and resume normal parsing.
- Effect: Parser pretends to have found an instance of  $A$ .

Other error recovery strategies possible:

- Yacc: error productions.

Compiler Design, WS 2005/2006