Syntax analysis

Module 3

Text book 2: Chapter 4 4.1, 4.2 4.3 4.4 4.5

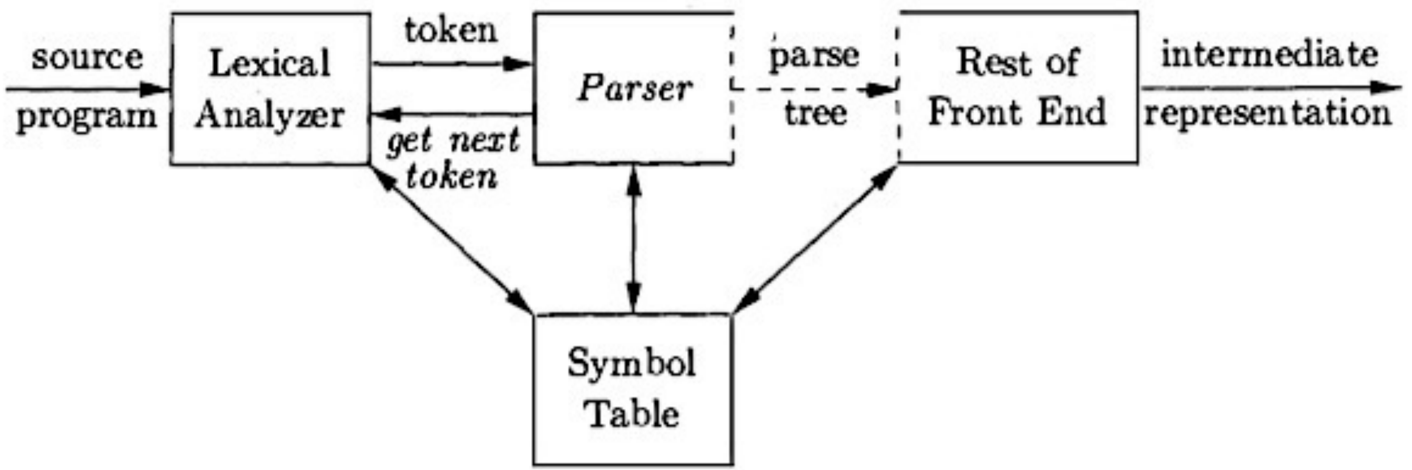
Content

- Syntax Analysis: Introduction,
- Context Free Grammars,
- Writing a grammar,
- Top Down Parsers,
- Bottom-Up Parsers

Introduction

- Every programming language has precise rules that prescribe the syntactic structure of program.
- The syntax of the programming language can be described by context free grammar.

The role of the parser



- Three general types of parsers for grammars
  - Universal
  - Top-down
  - Bottom-up
- The methods commonly used in parsers can be classified as top-down and bottom parsers.
- In either case input is scanned from left to right one at a time.
- Most efficient top-down and bottom methods work only for sub classes of grammars LL and LR grammars.

Syntax Error Handling

Common programming errors can occur at many different levels

- **Lexical errors**: misspellings of identifiers, keywords or operators
- **Syntactic errors**: misplaced semicolons, braces, extra braces.
- **Semantic errors**: type mismatches between operators and operands
- **Logical errors**: incorrect reasoning on the part of the programmer

Error recovery strategies

- Once an error is detected, how should the parser recover?.
- **Panic mode error recovery**
  - The parser discard the input symbols one at a time until one of the designated set of **synchronizing tokens** is found.
  - May not check some additional errors.
  - Guaranteed not to go into an infinite loop.

- **Phrase level recovery**
  - A parser may perform local correction on the remaining input, i,e. the parser may replace a prefix of the remaining input by some string that allows the parser to continue.
  - Replace a coma by semicolon, delete an extra semicolon, insert a missing semicolon.
  - Difficulty in coping with situations in which the actual error has occurred before the point of detection.

- **Error productions**
  - By anticipating common errors that might be encountered, we can augment the grammar for the language with productions that generate erroneous construct.
- **Global correction**
  - Algorithms for choosing a minimal sequences of changes to obtain a globally least- cost correction.

Context free grammar (CFG)

- A Context – free grammar has 4 components
  - A finite set of terminals
  - A finite set of non terminals
  - One nonterminal distinguished as start symbol.
  - A finite set of production rules in the following form

A->α Where A is a non-terminal, α is a string of terminals and non-terminals.

Each production consists of:

- A nonterminal called the head or left side of the production.
- The symbol -> Sometimes : : = has been used in place of the arrow.
- A body or right side consisting of zero or more terminals and non terminals.

Example

- Stmt -> **if** expr **then** stmt **else** stmt
- **Terminals** –if, then, else
- **Non – terminals-** expr and stmt
- **Start symbol-** Stmt

- Grammar that defines simple arithmetic expressions

expression -> expression **+** expression

expression -> expression **-** expression

expression -> term

term -> term ***** factor

term -> term **/** factor

term -> factor

factor -> (expression)

factor -> **id**

Notational conventions

- Terminals
  - a, b, c..
  - +, *, …
  - (,), ,
  - 0, 1, ..9
  - Strings : **id** , **if** (token name)
- Nonterminals
- A, B,C …
- S
- *expr, stmt …*

- The grammar to define arithmetic expressions

E->E+T | E-T | T

T-> T*F | T/F |F

F-> (E) | id

Derivations

- It is the process of obtaining string of terminal symbols by sequence of replacements of non terminal symbols.
- E->E+E | id
- String of terminal symbols id+id
- Leftmost derivation
- Rightmost derivation
- Parse tree – graphical representation of derivation

Ambiguous grammar

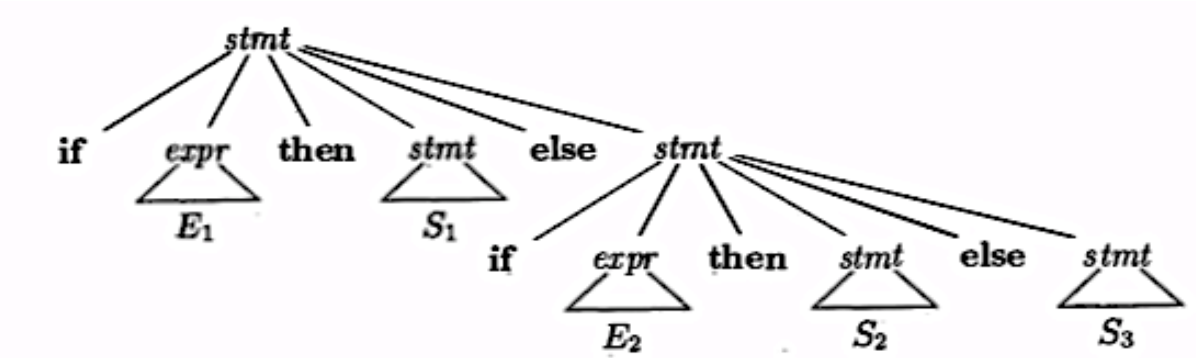
- A grammar producing more than one parse tree for some sentence is called as an **ambiguous grammar**.
- E->E+E | E*E | id
- Show that the following grammar is ambiguous for the sentence **id+id*id**
- Use disambiguating rules that throw away undesirable parse trees, leaving only one tree for each sentence.

- Ambiguous grammar can be rewritten to eliminate the ambiguity.

Eliminate ambiguity from Dangling else grammar

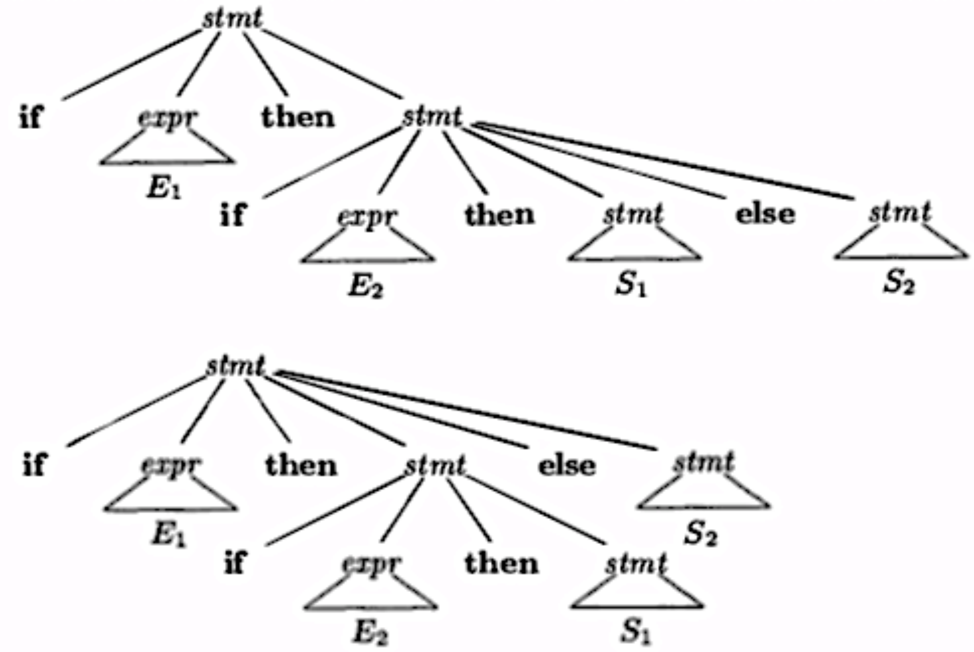
- *stmt*->**if** *expr* **then** *stmt* | **if** *expr* **then** *stmt* **else** *stmt* | **other**

If E1 **then** s1 **else if** E2 **then** S2 **else** S3



- The grammar is ambiguous

if *E1* **then if** *E2* **then** *S1* **else** *S2*



- Stmt -> matched_stmt | open_stmt
  matched_stmt -> **if** expr **then** matched_stmt **else** matched_stmt
  | **other**

  open_stmt -> **if** expr **then** stmt
  | **if** expr **then** matched_stmt **else** open_stmt

Elimination of left recursion

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for (each i from 1 to n) {
    for (each j from 1 to $i-1$) {
        replace each production of the form $A_i \to A_j\gamma$ by the
            productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
            $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

Figure 4.11: Algorithm to eliminate left recursion from a grammar

Left factoring

Algorithm 4.21: Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\to \alpha A' \mid \gamma \\ A' &\to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. $\square$

Top Down Parsers

4.4.1 Recursive-Descent Parsing

```
void A() {
      Choose an A-production, A → X_1 X_2 ··· X_k;
      for ( i = 1 to k ) {
            if ( X_i is a nonterminal )
                  call procedure X_i();
            else if ( X_i equals the current input symbol a )
                  advance the input to the next symbol;
            else /* an error has occurred */;
      }
}
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

FIRST AND FOLLOW

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) $= \{X\}$.

2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.

3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

LL(1) Grammars

Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

Non recursive Predictive parsing

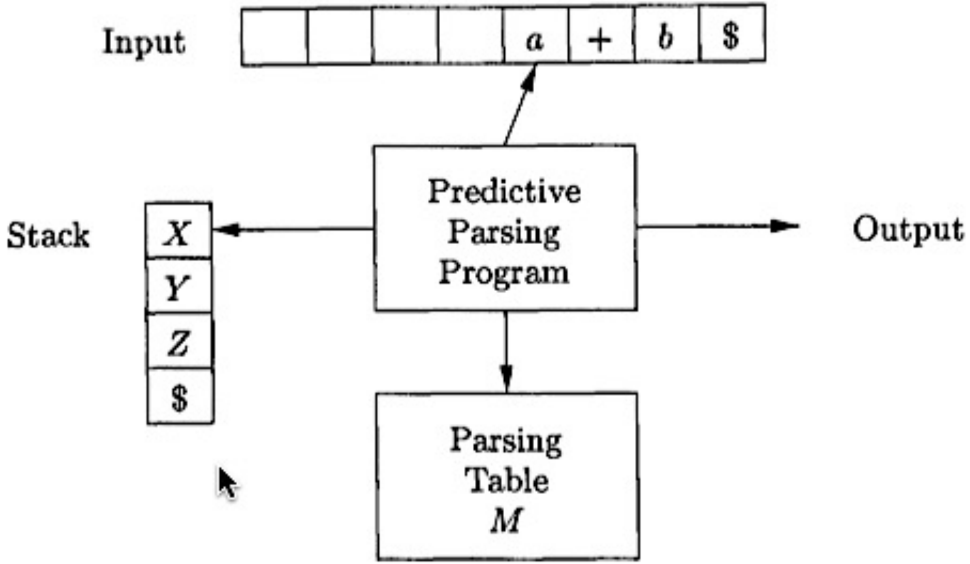


Figure 4.19: Model of a table-driven predictive parser

Predictive parsing algorithm

Algorithm 4.34: Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ··· Yk ) {
                output the production X → Y1Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

Error recovery in predictive parsing

- An error is detected during predictive parsing when
  - The terminal on the top of the stack does not match the next input symbol.
  - The nonterminal A is on the top of the stack, a is the next input symbol and M[A,a] is error(entry is empty)

Panic mode error recovery

- Use FIRST and FOLLOW symbols as synchronizing tokens.
- If the parser looks up entry M[A,a] and finds that it is blank, then the input symbol a is skipped.
- If the entry is “synch” then the nonterminal on top of the stack is popped in an attempt to resume parsing.
- If a token on top of the stack does not match the input symbol, the pop the token from the stack

Panic mode error recovery

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	\$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$	synch	synch

Panic mode error recovery

STACK	INPUT	REMARK
$E\,\$$	$)\,\mathbf{id} * + \mathbf{id}\,\$$	error, skip)
$E\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	**id** is in FIRST(E)
$TE'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\,\$$	
$T'E'\,\$$	$* + \mathbf{id}\,\$$	
$* FT'E'\,\$$	$* + \mathbf{id}\,\$$	
$FT'E'\,\$$	$+ \mathbf{id}\,\$$	error, $M[F, +] =$ synch
$T'E'\,\$$	$+ \mathbf{id}\,\$$	F has been popped
$E'\,\$$	$+ \mathbf{id}\,\$$	
$+ TE'\,\$$	$+ \mathbf{id}\,\$$	
$TE'\,\$$	$\mathbf{id}\,\$$	
$FT'E'\,\$$	$\mathbf{id}\,\$$	
$\mathbf{id}\,T'E'\,\$$	$\mathbf{id}\,\$$	
$T'E'\,\$$	$\$$	
$E'\,\$$	$\$$	
$\$$	$\$$	

Phrase level error recovery

- Implemented by filling in the blank entries in the predictive parsing table with pointers to error routines.
- Routines may change, insert ,or delete symbols on the input and issue appropriate error messages.
- Also pop from the stack