Chapter 3

Lexical and Syntax Analysis

Chapter 4 Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing
- Bottom-Up Parsing

4.1 Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach (Compilation, Pure interpretation and JIT)
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Using BNF to Describe Syntax

- Provides a clear and concise syntax description – contex free, can be used by human and softwares
- The parser (syntax analyzer) can be based directly on the BNF
- Parsers based on BNF are easy to maintain - modularity

Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* - separation allows optimization of the lexical analyzer
- *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

4.2 Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a "front-end" for the parser
- Identifies substrings of the source program that belong together - *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
  - `sum` is a lexeme; its token may be `IDENT`

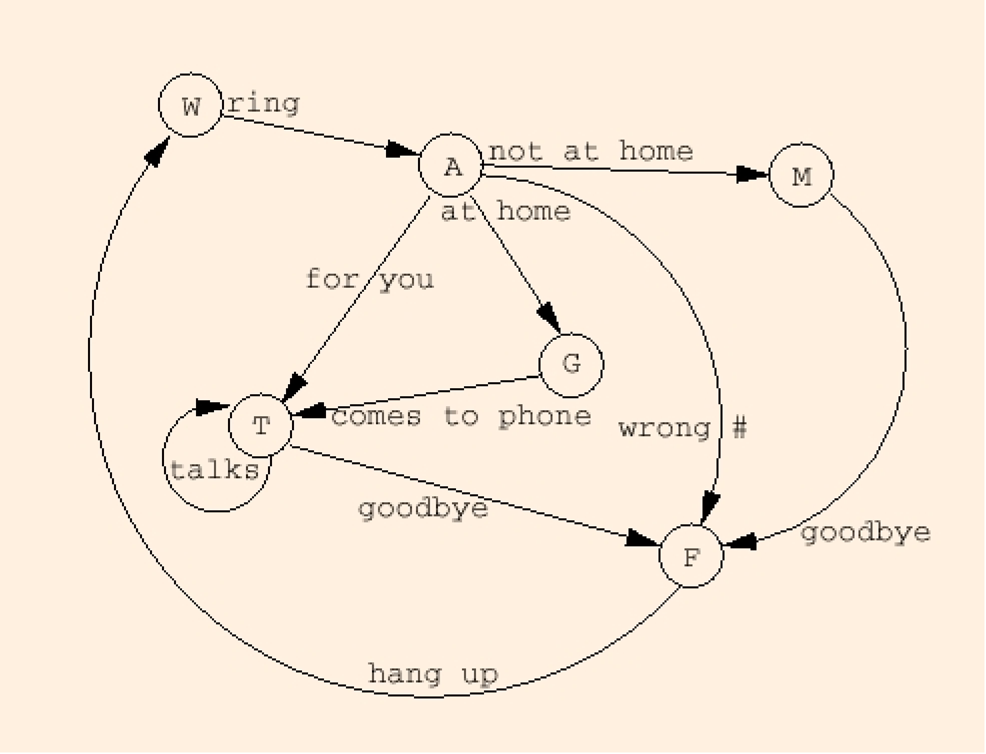
Lexical Analysis (continued)

- The lexical analyzer
  - Extracts lexemes from given input string and produce corresponding tokens
  - Syntax analyzer will only see the output of lexical analyzer (one lexeme at a time)
  - Skips comments and blanks
  - Inserts lexemes for user-defined names into symbol table – used by compiler
  - Detects syntactical errors in tokens

Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram

Example of a State Diagram



State Diagram Design

- A naïve state diagram would have a transition from every state on every character in the source language - such a diagram would be very large!

LEXICAL ANALYSIS (CONT.)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent - use a digit class

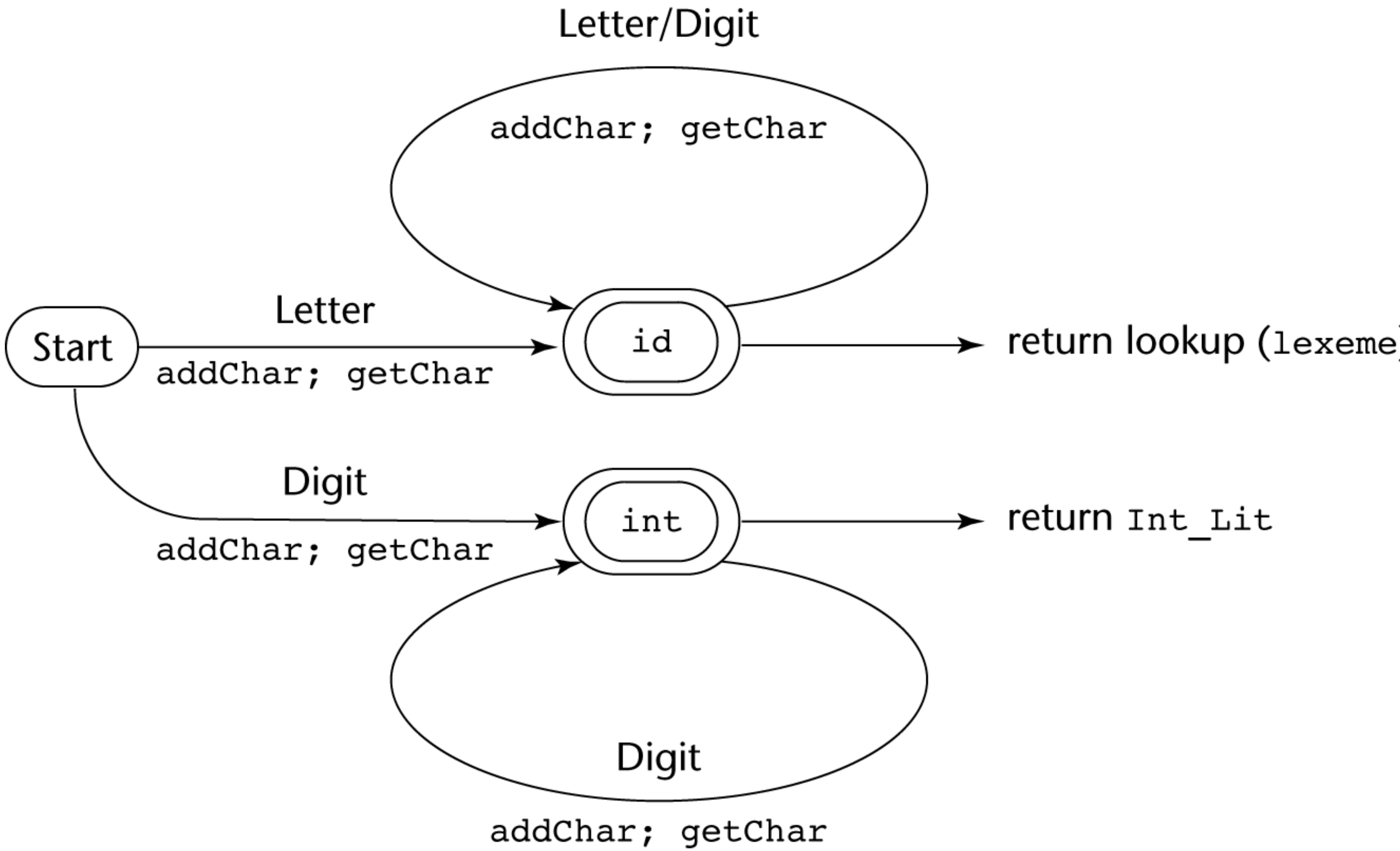
Lexical Analysis (cont.)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (cont.)

- Convenient utility subprograms:
  - **`getChar`** - gets the next character of input, puts it in **`nextChar`**, determines its class and puts the class in **`charClass`**
  - **`addChar`** - puts the character from **`nextChar`** into the place the lexeme is being accumulated, **`lexeme`**
  - lookup - determines whether the string in **`lexeme`** is a reserved word (returns a code)

State Diagram



Implementation (assume initialization):

```
int lex() {
  getChar();
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT)
      {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;

   case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }  /* End of switch */
}  /* End of function lex */
```

Parsing

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message, and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

PARSING (CONT.)

- Notations used …
  - Lowercase letters at the beginning of the alphabet (a, b, …) for terminal symbols.
  - Uppercase letters at the beginning of the alphabet (A, B, …) for nonterminals symbols.
  - Uppercase letters at the end of the alphabet (W, X, Y, Z) for terminals or nonterminals.
  - Lowercase letters at the end of the alphabet (w, x, y, z) for strings of terminal.
  - Lowercase Greek letters (α, β, γ, δ) for mixed strings (terminals and/or nonterminals)

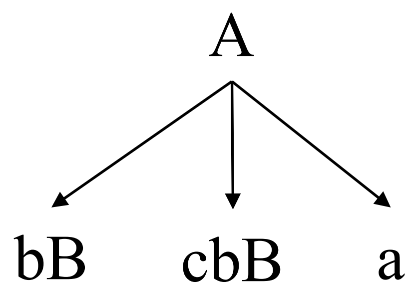
UNIVERSITI TENAGA NASIONAL

PARSING (CONT.)

- Two categories of parsers
  - *Top down* - produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* - produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Parsers look only one token ahead in the input

Parsing (cont.)

- Top-down Parsers
  - Given a sentential form, xAα , the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A



- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation

The Parsing Problem (cont.)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the LR family
  - Eg.

$S \rightarrow aAc$

$A \rightarrow aA \mid b$

S => aAc => aaAc => aabc

The Parsing Problem (cont.)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$, where n is the length of the input)

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing (cont.)

- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | ( <expr> )
```

Recursive-Descent Parsing (cont.)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing (cont.)

```
void expr() {
   term();
  while (nextToken == PLUS_CODE || nextToken == MINUS_CODE){
    lex();
    term();
  }
}
```

$<$expr$>$ → $<$term$>$ {(+ | -) $<$term$>$}
$<$term$>$ → $<$factor$>$ {(* | /) $<$factor$>$}
$<$factor$>$ → id | ($<$expr$>$)

```
void  term(){
   factor();
         while (nextToken == AST_CODE || nextToken ==
   SLACH_CODE){
         lex();
         factor();
   }
}

void factor() {
   if (nextToken) == ID_CODE)
      lex();

   else if (nextToken == LEFT_PAREN_CODE) {
     lex();
      expr();
      if (nextToken == RIGHT_PAREN_CODE)
         lex();
      else
         error();
    }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
 }
```

Recursive-Descent Parsing (cont.)

```
/* Function expr
   Parses strings in the language generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

  /* Parse the first term */
  term();
  /* As long as the next token is + or -, call lex to get the
  next token, and parse the next term */

  while (nextToken == PLUS_CODE || nextToken == MINUS_CODE){
    lex();
    term();
  }
}
```

- This particular routine does not detect errors
- Convention: Every parsing routine leaves the next token in `nextToken`

Recursive-Descent Parsing (cont.)

```
/* Function term
Parses string in the language generated by the rule: <term>
  -> <factor> {(*|/)<factor>}*/

void  term(){
  /*parse the first factor */
  factor();
  /*As long as the next token is * or /, call lex to get the
  next token, and parse the next facrtor*/
  while (nextToken == AST_CODE || nextToken == SLACH_CODE){
       lex();
       factor();
  }
}
```

Recursive-Descent Parsing (cont.)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing (cont.)

```
/* Function factor
   Parses strings in the language generated by the rule:
   <factor> -> id  |  (<expr>)   */

 void factor() { /* Determine which RHS */

  if (nextToken) == ID_CODE) /* For the RHS id, just call lex */
    lex();

 /* If the RHS is (<expr>) – call lex to pass over the left
  parenthesis, call expr, and check for the right parenthesis */

  else if (nextToken == LEFT_PAREN_CODE) {
    lex();
     expr();
     if (nextToken == RIGHT_PAREN_CODE)
       lex();
     else
       error();
   }  /* End of else if (nextToken == ...  */

  else error(); /* Neither RHS matches */
 }
```

Recursive-Descent Parsing (cont.)

- The LL Grammar Class
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove left recursion

Direct

$A \rightarrow A + B$............

Indirect

$A \rightarrow B\ a\ A$

$B \rightarrow A\ b$

Recursive-Descent Parsing (cont.)

- The LL Grammar Class
  - The Left Recursion Problem
    - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
      - A grammar can be modified to remove left recursion

$A \rightarrow A\alpha \mid \beta$ left recursion rule

Sol:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$ ε is the null value

Recursive-Descent Parsing (cont.)

- The LL Grammar Class
  - The Left Recursion Problem

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

ε is the null value

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E)\ \ id$

RECURSIVE-DESCENT PARSING (CONT.)

- The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness
  - The inability to determine the correct RHS on the basis of one token of lookahead
  - Def: FIRST(α) = $\{a \mid \alpha \Rightarrow^* a\beta\}$

    (If $\alpha \Rightarrow^* \varepsilon$, ε is in FIRST(α))

Recursive-Descent Parsing (cont.)

- Pairwise Disjointness Test:
  - For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that

    $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$
- Examples:

  $A \rightarrow aA \mid bB$ - pass the test

  $A \rightarrow aA \mid aB$ - fail the test

  SOL:

  $A \rightarrow aA'$

  $A' \rightarrow A \mid B$

  $A \rightarrow aA \mid a$ - fail the test

  SOL:

  $A \rightarrow aA'$

  $A' \rightarrow A \mid \varepsilon$ ε is the null value

Recursive-Descent Parsing (cont.)

- Left factoring can resolve the problem

  Replace

  <variable> → identifier | identifier [<expression>]

  with

  <variable> → identifier <new>

  <new> → ε | [<expression>]

  or

  <variable> → identifier [[<expression>]]

  (the outer brackets are metasymbols of EBNF)

Recursive-Descent Parsing (cont.)

- Left factoring can resolve the problem

$s \rightarrow a \mid ab \mid abc \mid abcd$

<u>Sol:</u>

$s \rightarrow as'$

$s' \rightarrow \varepsilon \mid bs''$ $\quad\quad$ ε is the null value

$s'' \rightarrow \varepsilon \mid cs'''$

$s''' \rightarrow \varepsilon \mid d$

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form (*handle*) to reduce to get the previous right-sentential form in the derivation
- Example grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id \qquad (1)$$

E.g. derived sentence: id + id * id

Bottom-up Parsing (cont.)

- Intuition about handles:
  - Def: β is the *handle* of the right sentential form

    $\gamma = \alpha\beta w$ if and only if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

  - Def: β is a *phrase* of the right sentential form

    γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$

  - Def: β is a *simple phrase* of the right sentential form γ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$

Bottom-up Parsing (cont.)

- Intuition about handles:
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning

Bottom-up Parsing (cont.)

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (cont.)

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

Bottom-up Parsing (cont.)

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, and stored in the parse stack

Bottom-up Parsing (cont.)

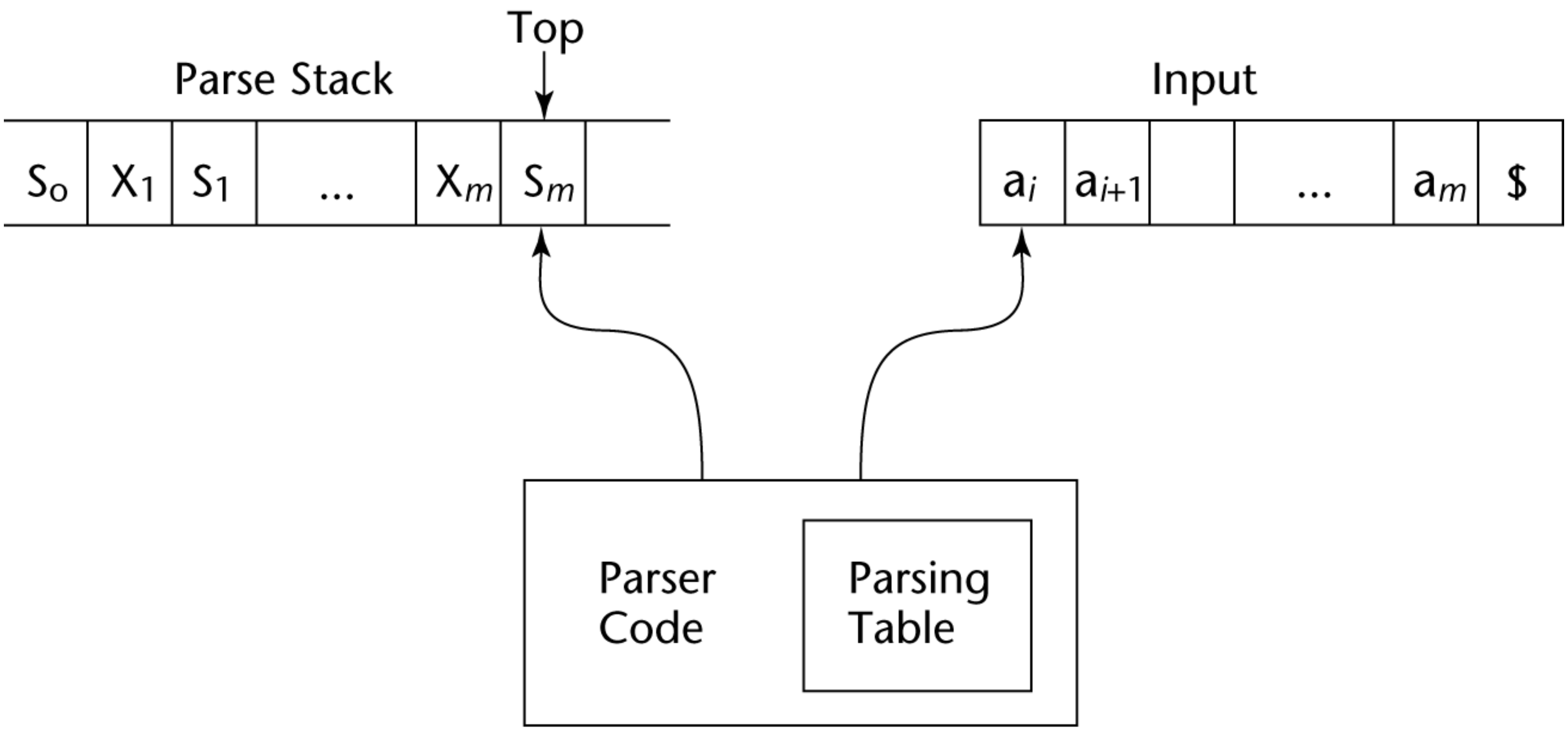
- An LR configuration stores the state of an LR parser

$(S_0X_1S_1X_2S_2\ldots X_mS_m,\ a_ia_i{+}1\ldots a_n\$)$

BOTTOM-UP PARSING (CONT.)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Structure of An LR Parser



Bottom-up Parsing (cont.)

- Initial configuration: $(S_0, a_1 \ldots a_n\$)$
- Parser actions:
  - If ACTION$[S_m, a_i]$ = Shift S, the next configuration is:
    $(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$

  - If ACTION$[S_m, a_i]$ = Reduce $A \rightarrow \beta$ and S = GOTO$[S_{m-r}, A]$, where r = the length of β, the next configuration is

    $(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$

Bottom-up Parsing (cont.)

- Parser actions (continued):
  - If ACTION[S_m, a_i] = Accept, the parse is complete and no errors were found.
  - If ACTION[S_m, a_i] = Error, the parser calls an error-handling routine.

LR Parsing Table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (cont.)

- Grammar (1) rewritten and numbered for easy referencing in a parsing table.

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

UNIVERSITI TENAGA NASIONAL

Bottom-up Parsing (cont.)

Stack	Input	Action
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

Bottom-up Parsing (cont.)

- A parser table can be generated from a given grammar with a tool, e.g., **`yacc`**

SUMMARY

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach