4. LEXICAL AND SYNTAX ANALYSIS

Introduction

- Chapter 1 described three approaches to implementing programming languages: compilation, pure interpretation, and hybrid implementation. All three use both a lexical analyzer and a syntax analyzer.

- The job of a syntax analyzer is to check the syntax of a program and create a parse tree from it.

- Syntax analyzers, or parsers, are nearly always based on a formal description of the syntax of programs, usually in form of a context-free grammar or BNF.

- Advantages of using BNF:

  BNF descriptions are clear and concise, both for humans and software systems.
  Syntax analyzers can be generated directly from BNF.
  Implementations based on BNF are easy to maintain.

- Nearly all compilers separate the task of analyzing syntax into two distinct parts:

  The lexical analyzer deals with small-scale language constructs, such as names and numeric literals.
  The syntax analyzer deals with large-scale constructs, such as expressions, statements, and program units.

- Reasons for the separation:

  Simplicity—Removing the details of lexical analysis from the syntax analyzer makes it smaller and less complex.
  Efficiency—It beomes easier to optimize the lexical analyzer.
  Portability—The lexical analyzer reads source files, so it may be platform-dependent.

Lexical Analysis

- A lexical analyzer collects input characters into groups (**lexemes**) and assigns an internal code (a **token**) to each group.

- Lexemes are recognized by matching the input against patterns.

- Tokens are usually coded as integer values, but for the sake of readability, they are often referenced through named constants.

- An example assignment statement:

```
result = oldsum - value / 100;
```

  Tokens and lexemes of this statement:

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

- In early compilers, lexical analyzers often processed an entire program file and produced a file of tokens and lexemes. Now, most lexical analyzers are subprograms that return the next lexeme and its associated token code when called.

- Other tasks performed by a lexical analyzer:

  Skipping comments and white space between lexemes.
  Inserting lexemes for user-defined names into the symbol table.
  Detecting syntactic errors in tokens, such as ill-formed floating-point literals.

Lexical Analysis (Continued)

- Approaches to building a lexical analyzer:

  Write a formal description of the token patterns of the language and use a software tool such as `lex` to automatically generate a lexical analyzer.
  Design a state transition diagram that describes the token patterns of the language and write a program that implements the diagram.
  Design a state transition diagram that describes the token patterns of the language and hand-construct a table-driven implementation of the state diagram.

- A state transition diagram, or **state diagram,** is a directed graph.

  The nodes are labeled with state names.
  The arcs are labeled with input characters.
  An arc may also include actions to be done when the transition is taken.

Lexical Analysis (Continued)

- Consider the problem of building a lexical analyzer that recognizes lexemes that appear in arithmetic expressions, including variable names and integer literals.

  Names consist of uppercase letters, lowercase letters, and digits, but must begin with a letter.
  Names have no length limitations.

- To simplify the transition diagram, we can treat all letters the same way, having one transition from a state instead of 52 transitions. In the lexical analyzer, LETTER will represent the class of all 52 letters.

- Integer literals offer another opportunity to simplify the transition diagram. Instead of having 10 transitions from a state, it is better to group digits into a single character class (named DIGIT) and have one transition.

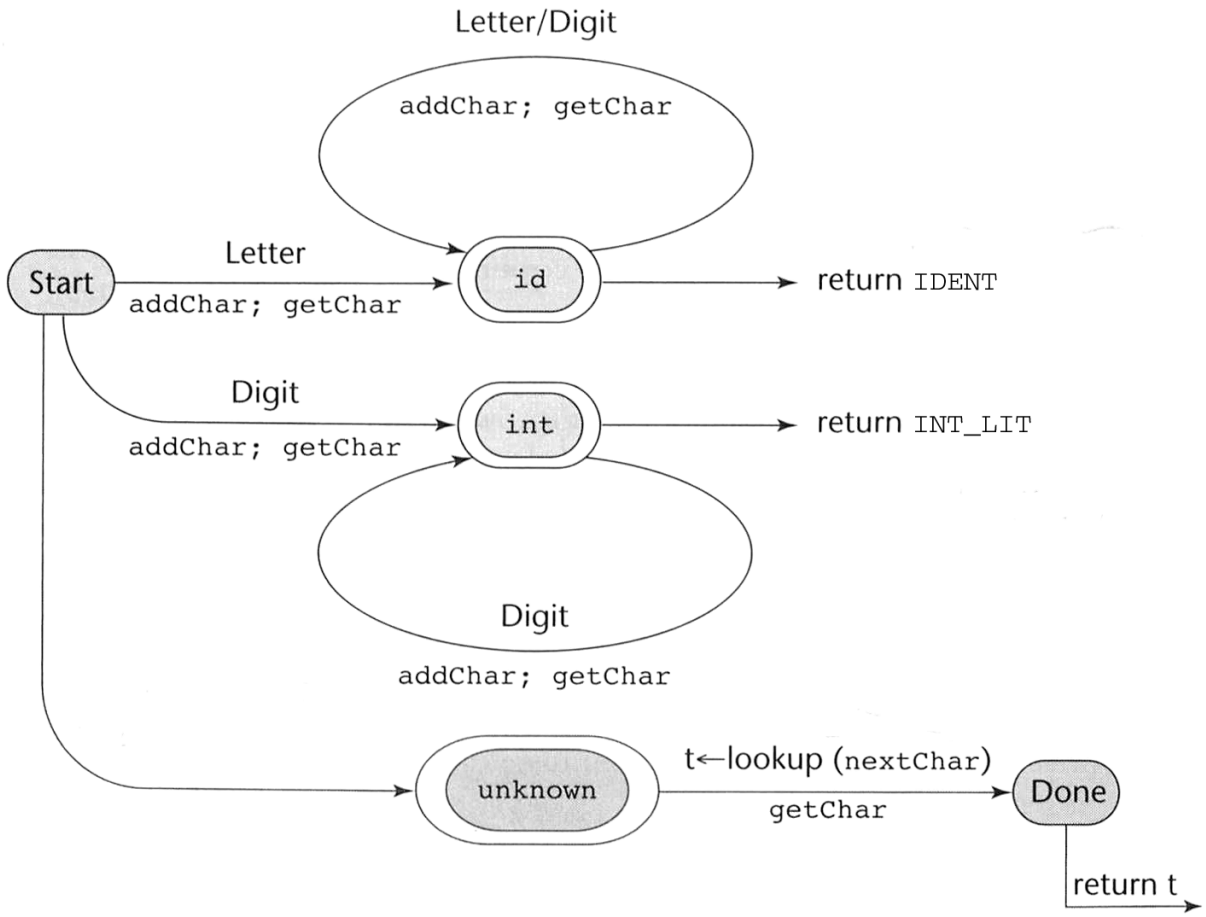
- The lexical analyzer will use a string (or character array) named `lexeme` to store a lexeme as it is being read.

- Utility subprograms needed by the lexical analyzer:

  `getChar`—Gets the next input character and puts it in a global variable named `nextChar`. Also determines the character class of the input character and puts it in the global variable `charClass`.
  `addChar`—Adds the character in `nextChar` to the end of `lexeme`.
  `getNonBlank`—Skips white space.
  `lookup`—Computes the token code for single-character tokens (parentheses and arithmetic operators).

Lexical Analysis (Continued)

- A state diagram that recognizes names, integer literals, parentheses, and arithmetic operators:



The diagram includes the actions required on each transition.

Lexical Analysis (Continued)

- C code for a lexical analyzer that implements this state diagram:

```
/* front.c - a lexical analyzer system for simple
             arithmetic expressions */

#include <stdio.h>
#include <ctype.h>

/* Global declarations */
/* Variables */
int charClass;
char lexeme[100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp;

/* Function declarations */
int lookup(char ch);
void addChar(void);
void getChar(void);
void getNonBlank(void);
int lex(void);

/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
```

```
/*****************************************************/
/* main driver */
int main(void) {
/* Open the input data file and process its contents */
  if ((in_fp = fopen("front.in", "r")) == NULL)
    printf("ERROR - cannot open front.in \n");
  else {
    getChar();
    do {
      lex();
    } while (nextToken != EOF);
  }
  return 0;
}
/*****************************************************/
/* lookup - a function to look up operators and
            parentheses and return the token */
int lookup(char ch) {
  switch (ch) {
    case '(':
      addChar();
      nextToken = LEFT_PAREN;
      break;
    case ')':
      addChar();
      nextToken = RIGHT_PAREN;
      break;
    case '+':
      addChar();
      nextToken = ADD_OP;
      break;
    case '-':
      addChar();
      nextToken = SUB_OP;
      break;
    case '*':
      addChar();
      nextToken = MULT_OP;
      break;
    case '/':
      addChar();
      nextToken = DIV_OP;
      break;
    default:
      addChar();
      nextToken = EOF;
      break;
  }
  return nextToken;
}
```

```
/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar(void) {
  if (lexLen <= 98) {
    lexeme[lexLen++] = nextChar;
    lexeme[lexLen] = '\0';
  }
  else
    printf("Error - lexeme is too long \n");
}

/*****************************************************/
/* getChar - a function to get the next character of
             input and determine its character class */
void getChar(void) {
  if ((nextChar = getc(in_fp)) != EOF) {
    if (isalpha(nextChar))
      charClass = LETTER;
    else if (isdigit(nextChar))
      charClass = DIGIT;
    else
      charClass = UNKNOWN;
  }
  else
    charClass = EOF;
}

/*****************************************************/
/* getNonBlank - a function to call getChar until it
                 returns a non-whitespace character */
void getNonBlank(void) {
  while (isspace(nextChar))
    getChar();
}
```

```
/*****************************************************/
/* lex - a simple lexical analyzer for arithmetic
         expressions */
int lex(void) {
  lexLen = 0;
  getNonBlank();
  switch (charClass) {

/* Identifiers */
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT) {
        addChar();
        getChar();
      }
      nextToken = IDENT;
      break;

/* Integer literals */
    case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      nextToken = INT_LIT;
      break;

/* Parentheses and operators */
    case UNKNOWN:
      lookup(nextChar);
      getChar();
      break;

/* EOF */
    case EOF:
      nextToken = EOF;
      lexeme[0] = 'E';
      lexeme[1] = 'O';
      lexeme[2] = 'F';
      lexeme[3] = '\0';
      break;
  }  /* End of switch */

  printf("Next token is: %d, Next lexeme is %s\n",
         nextToken, lexeme);
  return nextToken;
}  /* End of function lex */
```

Lexical Analysis (Continued)

- Sample input for `front.c`:

```
(sum + 47) / total
```

- Output of `front.c`:

```
Next token is: 25, Next lexeme is (
Next token is: 11, Next lexeme is sum
Next token is: 21, Next lexeme is +
Next token is: 10, Next lexeme is 47
Next token is: 26, Next lexeme is )
Next token is: 24, Next lexeme is /
Next token is: 11, Next lexeme is total
Next token is: -1, Next lexeme is EOF
```

Introduction to Parsing

- Syntax analysis is often referred to as **parsing.**

- Responsibilities of a syntax analyzer, or parser:

  Determine whether the input program is syntactically correct.
  Produce a parse tree. In some cases, the parse tree is only implicitly constructed.

- When an error is found, a parser must produce a diagnostic message and recover. Recovery is required so that the compiler finds as many errors as possible.

- Parsers are categorized according to the direction in which they build parse trees:

  Top-down parsers build the tree from the root downward to the leaves.
  Bottom-up parsers build the tree from the leaves upward to the root.

- Notational conventions for grammar symbols and strings:

  Terminal symbols—Lowercase letters at the beginning of the alphabet (a, b, …)
  Nonterminal symbols—Uppercase letters at the beginning of the alphabet (A, B, …)
  Terminals or nonterminals—Uppercase letters at the end of the alphabet (W, X, Y, Z)
  Strings of terminals—Lowercase letters at the end of the alphabet (w, x, y, z)
  Mixed strings (terminals and/or nonterminals)—Lowercase Greek letters (α, β, γ, δ)

Top-Down Parsers

- A top-down parser traces or builds the parse tree in preorder: each node is visited before its branches are followed.

- The actions taken by a top-down parser correspond to a leftmost derivation.

- Given a sentential form $xA\alpha$ that is part of a leftmost derivation, a top-down parser's task is to find the next sentential form in that leftmost derivation.

  Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its left-hand side (LHS).
  If the A-rules are $A \rightarrow bB$, $A \rightarrow cBb$, and $A \rightarrow a$, the next sentential form could be $xbB\alpha$, $xcBb\alpha$, or $xa\alpha$.
  The most commonly used top-down parsing algorithms choose an A-rule based on the token that would be the first generated by A.

- The most common top-down parsing algorithms are closely related.

  A **recursive-descent parser** is coded directly from the BNF description of the syntax of a language.
  An alternative is to use a parsing table rather than code.

- Both are **LL algorithms,** and both are equally powerful. The first L in LL specifies a left-to-right scan of the input; the second L specifies that a leftmost derivation is generated.

Bottom-Up Parsers

- A bottom-up parser constructs a parse tree by beginning at the leaves and progressing toward the root. This parse order corresponds to the reverse of a rightmost derivation.

- Given a right sentential form α, a bottom-up parser must determine what substring of α is the right-hand side (RHS) of the rule that must be reduced to its LHS to produce the previous right sentential form.

- A given right sentential form may include more than one RHS from the grammar. The correct RHS to reduce is called the **handle.**

- Consider the following grammar and derivation:

  $S \rightarrow aAc$
  $A \rightarrow aA \mid b$

  S => aAc => aaAc => aabc

  A bottom-up parser can easily find the first handle, b, because it is the only RHS in the sentence aabc. After replacing b by the corresponding LHS, A, the parser is left with the sentential form aaAc. Finding the next handle will be more difficult because both aAc and aA are potential handles.

- A bottom-up parser finds the handle of a given right sentential form by examining the symbols on one or both sides of a possible handle.

- The most common bottom-up parsing algorithms are in the LR family. The L specifies a left-to-right scan and the R specifies that a rightmost derivation is generated.

The Complexity of Parsing

- Parsing algorithms that work for any grammar are inefficient. The worst-case complexity of common parsing algorithms is $O(n^3)$, making them impractical for use in compilers.

- Faster algorithms work for only a subset of all possible grammars. These algorithms are acceptable as long as they can parse grammars that describe programming languages.

- Parsing algorithms used in commercial compilers have complexity $O(n)$.

The Recursive-Descent Parsing Process

- A recursive-descent parser consists of a collection of subprograms, many of which are recursive; it produces a parse tree in top-down order.

- A recursive-descent parser has one subprogram for each nonterminal in the grammar.

- EBNF is ideally suited for recursive-descent parsers.

- An EBNF description of simple arithmetic expressions:

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

- These rules can be used to construct a recursive-descent function named `expr` that parses arithmetic expressions.

- The lexical analyzer is assumed to be a function named `lex`. It reads a lexeme and puts its token code in the global variable `nextToken`. Token codes are defined as named constants.

The Recursive-Descent Parsing Process (Continued)

- Writing a recursive-descent subprogram for a rule with a single RHS is relatively simple.

  For each terminal symbol in the RHS, that terminal symbol is compared with `nextToken`. If they do not match, it is a syntax error. If they match, the lexical analyzer is called to get to the next input token.
  For each nonterminal, the parsing subprogram for that nonterminal is called.

- A recursive-descent subprogram for <expr>, written in C:

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
  */
void expr(void) {
  printf("Enter <expr>\n");

/* Parse the first term */
  term();

/* As long as the next token is + or -, get
   the next token and parse the next term */
  while (nextToken == ADD_OP || nextToken == SUB_OP) {
    lex();
    term();
  }
  printf("Exit <expr>\n");
}
```

- Each recursive-descent subprogram, including `expr`, leaves the next input token in `nextToken`.

- `expr` does not include any code for syntax error detection or recovery, because there are no detectable errors associated with the rule for <expr>.

The Recursive-Descent Parsing Process (Continued)

- The subprogram for <term> is similar to that for <expr>:

```
/* term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>}
   */
void term(void) {
  printf("Enter <term>\n");

/* Parse the first factor */
  factor();

/* As long as the next token is * or /, get the
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
  printf("Exit <term>\n");
}
```

The Recursive-Descent Parsing Process (Continued)

- A recursive-descent parsing subprogram for a nonterminal whose rule has more than one RHS must examine the value of `nextToken` to determine which RHS is to be parsed.

- The recursive-descent subprogram for <factor> must choose between two RHSs:

```
/* factor
   Parses strings in the language generated by the rule:
     <factor> -> id | int_constant | ( <expr> )
   */
void factor(void) {
  printf("Enter <factor>\n");

/* Determine which RHS */
  if (nextToken == IDENT || nextToken == INT_LIT)

/* Get the next token */
    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) {
      lex();
      expr();
      if (nextToken == RIGHT_PAREN)
        lex();
      else
        error();
    }

/* It was not an id, an integer literal, or a left
   parenthesis */
    else
      error();
  }

  printf("Exit <factor>\n");
}
```

- The `error` function is called when a syntax error is detected. A real parser would produce a diagnostic message and attempt to recover from the error.

The Recursive-Descent Parsing Process (Continued)

- Trace of the parse of `(sum + 47) / total`:

```
Next token is: 25, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10, Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24, Next lexeme is /
Exit <factor>
Next token is: 11, Next lexeme is total
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

The Recursive-Descent Parsing Process (Continued)

- An EBNF description of the Java `if` statement:

  <ifstmt> → **`if`** (<boolexpr>) <statement> [**`else`** <statement>]

- The recursive-descent subprogram for <ifstmt>:

```
/* ifstmt
   Parses strings in the language generated by the rule:
   <ifstmt> -> if (<boolexpr>) <statement>
               [else <statement>]
   */
void ifstmt(void) {
  if (nextToken != IF_CODE)
    error();
  else {
    lex();
    if (nextToken != LEFT_PAREN)
      error();
    else {
      lex();
      boolexpr();
      if (nextToken != RIGHT_PAREN)
        error();
      else {
        lex();
        statement();
        if (nextToken == ELSE_CODE) {
          lex();
          statement();
        }
      }
    }
  }
}
```

The LL Grammar Class

- Recursive-descent and other LL parsers can be used only with grammars that meet certain restrictions.

- Left recursion causes a catastrophic problem for LL parsers.

- Calling the recursive-descent parsing subprogram for the following rule would cause infinite recursion:

  $A \rightarrow A + B$

- The left recursion in the rule $A \rightarrow A + B$ is called **direct left recursion,** because it occurs in one rule.

- An algorithm for eliminating direct left recursion from a grammar:

  For each nonterminal A,

  1. Group the A-rules as $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
     where none of the β's begins with A.

  2. Replace the original A-rules with
     $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$
     $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

- The symbol ε represents the empty string. A rule that has ε as its RHS is called an *erasure rule,* because using it in a derivation effectively erases its LHS from the sentential form.

The LL Grammar Class (Continued)

- Left recursion can easily be eliminated from the following grammar:

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

  For the E-rules, we have $\alpha_1 = + T$ and $\beta_1 = T$, so we replace the E-rules with

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

  For the T-rules, we have $\alpha_1 = * F$ and $\beta_1 = F$, so we replace the T-rules with

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

  The F-rules remain the same.

- The grammar with left recursion removed:

  $E \rightarrow T E'$
  $E' \rightarrow + T E' \mid \varepsilon$
  $T \rightarrow F T'$
  $T' \rightarrow * F T' \mid \varepsilon$
  $F \rightarrow (E) \mid id$

- Indirect left recursion poses the same problem as direct left recursion:

  $A \rightarrow B\ a\ A$
  $B \rightarrow A\ b$

- Algorithms exist that remove indirect left recursion from a grammar.

The LL Grammar Class (Continued)

- Left recursion is not the only grammar trait that disallows top-down parsing. A top-down parser must always be able to choose the correct RHS on the basis of the next token of input.

- The **pairwise disjointness test** is used to test a non-left-recursive grammar to determine whether it can be parsed in a top-down fashion. This test requires computing FIRST sets, where

  FIRST(α) = {a | α =>* aβ}

  The symbol =>* indicates a derivation of zero or more steps. If α =>* ε, then ε is also in FIRST(α), where ε is the empty string.

- There are algorithms to compute FIRST for any mixed string. In simple cases, FIRST can usually be computed by inspecting the grammar.

- The pairwise disjointness test:

  For each nonterminal A that has more than one RHS, and for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing$.

- An example:

  $A \rightarrow a\ B \mid b\ A\ b \mid B\ b$
  $B \rightarrow c\ B \mid d$

  The FIRST sets for the RHSs of the A-rules are {a}, {b}, and {c, d}. These rules pass the pairwise disjointness test.

- A second example:

  $A \rightarrow a\ B \mid B\ A\ b$
  $B \rightarrow a\ B \mid b$

  The FIRST sets for the RHSs of the A-rules are {a} and {a, b}. These rules fail the pairwise disjointness test.

The LL Grammar Class (Continued)

- In many cases, a grammar that fails the pairwise disjointness test can be modified so that it will pass the test.

- The following rules do not pass the pairwise disjointness test:

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

  (The square brackets are terminals, not metasymbols.) This problem can solved by **left factoring.**

- Using left factoring, these rules would be replaced by the following rules:

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

- Using EBNF can also help. The original rules for <variable> can be replaced by the following EBNF rules:

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

  The outer brackets are metasymbols, and the inner brackets are terminals.

- Formal algorithms for left factoring exist.

- Left factoring cannot solve all pairwise disjointness problems. In some cases, rules must be rewritten in other ways to eliminate the problem.

The Parsing Problem for Bottom-Up Parsers

- The following grammar for arithmetic expressions will be used to illustrate bottom-up parsing:

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

  This grammar is left recursive, which is acceptable to bottom-up parsers.

- Grammars for bottom-up parsers are normally written using ordinary BNF, not EBNF.

- A rightmost derivation using this grammar:

  E => <u>E + T</u>
  => E + <u>T * F</u>
  => E + T * <u>id</u>
  => E + <u>F</u> * id
  => E + <u>id</u> * id
  => <u>T</u> + id * id
  => <u>F</u> + id * id
  => <u>id</u> + id * id

  The underlined part of each sentential form shows the RHS of the rule that was applied at that step.

- A bottom-up parser produces the reverse of a rightmost derivation by starting with the last sentential form (the input sentence) and working back to the start symbol.

- At each step, the parser's task is to find the RHS in the current sentential form that must be rewritten to get the previous sentential form.

The Parsing Problem for Bottom-Up Parsers (Continued)

- A right sentential form may include more than one RHS.

  The right sentential form E + T * id includes three RHSs, E + T, T, and id.

- The task of a bottom-up parser is to find the unique handle of a given right sentential form.

- *Definition:* β is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if $S =>^*_{rm} \alpha A w =>_{rm} \alpha\beta w$.

  $=>_{rm}$ specifies a rightmost derivation step, and $=>^*_{rm}$ specifies zero or more rightmost derivation steps.

- Two other concepts are related to the idea of the handle.

- *Definition:* β is a **phrase** of the right sentential form $\gamma = \alpha_1\beta\alpha_2$ if and only if $S =>^* \alpha_1 A \alpha_2 =>^+ \alpha_1\beta\alpha_2$.

  $=>^+$ means one or more derivation steps.

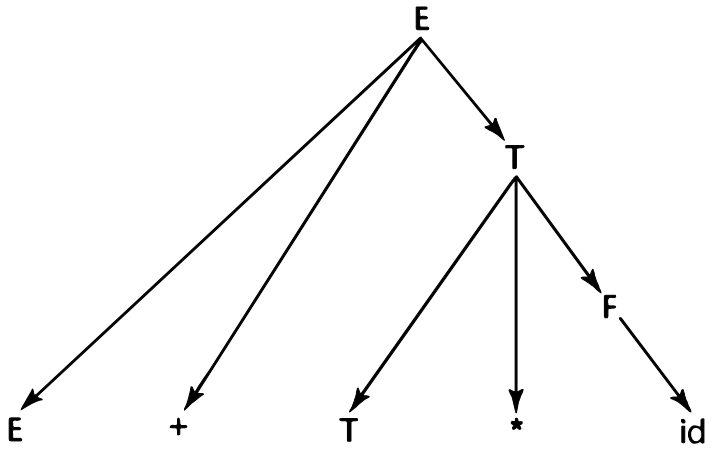
- *Definition:* β is a **simple phrase** of the right sentential form $\gamma = \alpha_1\beta\alpha_2$ if and only if $S =>^* \alpha_1 A \alpha_2 => \alpha_1\beta\alpha_2$.

The Parsing Problem for Bottom-Up Parsers (Continued)

- A *phrase* is a string consisting of all of the leaves of the partial parse tree that is rooted at one particular internal node of the whole parse tree.

- A *simple phrase* is a phrase that is derived from a nonterminal in a single step.

- An example parse tree:



  The leaves of the parse tree represent the sentential form E + T * id. Because there are three internal nodes, there are three phrases: E + T * id, T * id, and id.

- The simple phrases are a subset of the phrases. In this example, the only simple phrase is id.

- The handle of a right sentential form is the leftmost simple phrase.

- Once the handle has been found, it can be pruned from the parse tree and the process repeated. Continuing to the root of the parse tree, the entire rightmost derivation can be constructed.

Shift-Reduce Algorithms

- Bottom-up parsers are often called **shift-reduce algorithms,** because shift and reduce are their two fundamental actions.

  The shift action moves the next input token onto the parser's stack.
  A reduce action replaces a RHS (the handle) on top of the parser's stack by its corresponding LHS.

LR Parsers

- Most bottom-up parsing algorithms belong to the LR family. LR parsers use a relatively small amount of code and a parsing table.

- The original LR algorithm was designed by Donald Knuth, who published it in 1965. His **canonical LR** algorithm was not widely used because producing the parsing table required large amounts of computer time and memory.

- Later variations on the table construction process were more popular. These variations require much less time and memory to produce the parsing table but work for smaller classes of grammars.

- Advantages of LR parsers:

  They can be built for all programming languages.
  They can detect syntax errors as soon as possible in a left-to-right scan.
  The LR class of grammars is a proper superset of the class parsable by LL parsers.

- It is difficult to produce an LR parsing table by hand. However, there are many programs available that take a grammar as input and produce the parsing table.

LR Parsers (Continued)

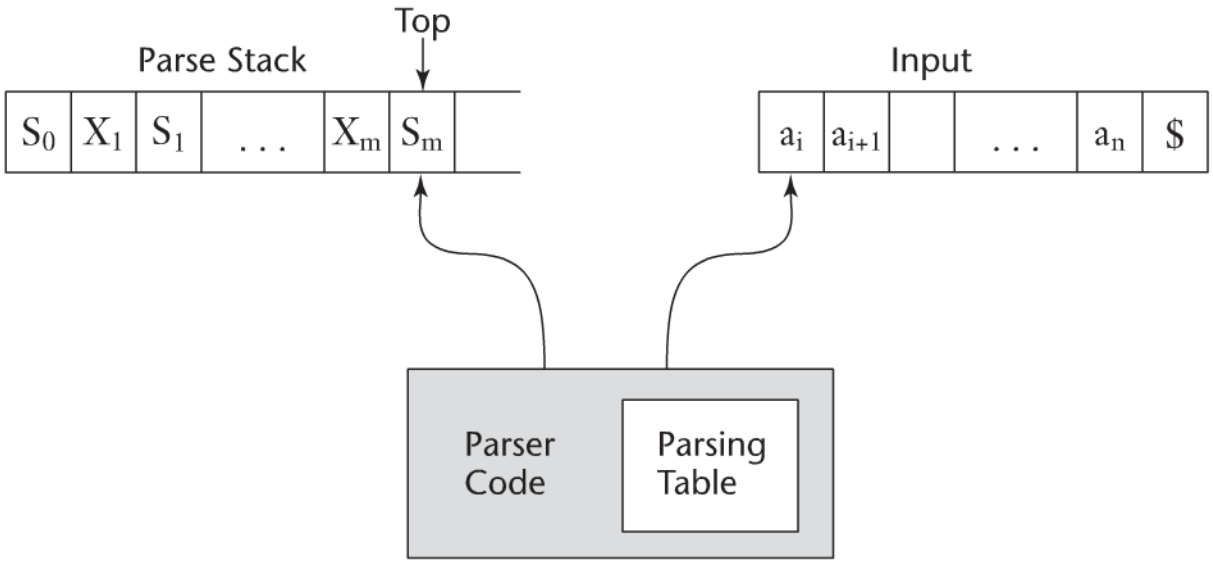
- Older parsing algorithms would find the handle by looking both to the left and to the right of the substring that was suspected of being the handle.

- Knuth's insight was that it was only necessary to look to the left of the suspected handle (by examining the stack) to determine whether it was the handle.

- Even better, the parser can avoid examining the entire stack if it keeps a summary of the stack contents in a "state" symbol on top of the stack.

- In general, each grammar symbol on the stack will be followed by a state symbol (often written as a subscripted uppercase S).

- The structure of an LR parser:



LR Parsers (Continued)

- The contents of the parse stack for an LR parser has the following form, where the Ss are state symbols and the Xs are grammar symbols:

  $S_0X_1S_1X_2S_2\ldots X_mS_m$ (top)

- An LR parser configuration is a pair of strings representing the stack and the remaining input:

  $(S_0X_lS_lX_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$

  The dollar sign is an end-of-input marker.

- The LR parsing process is based on the parsing table, which has two parts, ACTION and GOTO.

- The ACTION part has state symbols as its row labels and terminal symbols as its column labels.

- The parse table specifies what the parser should do, based on the state symbol on top of the parse stack and the next input symbol.

- The two primary actions are *shift* (shift the next input symbol onto the stack) and *reduce* (replace the handle on top of the stack by the LHS of the matching rule).

- Two other actions are possible: *accept* (parsing is complete) and *error* (a syntax error has been detected).

- The values in the GOTO part of the table indicate which state symbol should be pushed onto the parse stack after a reduction has been completed.

  The row is determined by the state symbol on top of the parse stack after the handle and its associated state symbols have been removed.
  The column is determined by the LHS of the rule used in the reduction.

LR Parsers (Continued)

- Initial configuration of an LR parser:

  $(S_0, a_1 \ldots a_n\$)$

- Informal definition of parser actions:

  Shift: The next input symbol is pushed onto the stack, along with the state symbol specified in the ACTION table.

  Reduce: First, the handle is removed from the stack. For every grammar symbol on the stack there is a state symbol, so the number of symbols removed is twice the number of symbols in the handle. Next, the LHS of the rule is pushed onto the stack. Finally, the GOTO table is used to determine which state must be pushed onto the stack.

  Accept: The parse is complete and no errors were found.

  Error: The parser calls an error-handling routine.

- All LR parsers use this parsing algorithm, although they may construct the parsing table in different ways.

- The following grammar will be used to illustrate LR parsing:

  1. E → E + T
  2. E → T
  3. T → T * F
  4. T → F
  5. F → (E)
  6. F → id

LR Parsers (Continued)

- The LR parsing table for this grammar:

	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			

R4 means reduce using rule 4; S6 means shift the next input symbol onto the stack and push state S_6. Empty positions in the ACTION table indicate syntax errors.

- A trace of the parse of id + id * id using the LR parsing algorithm:

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