# Lexical and Syntax Analysis – Chapter 4

## Introduction

- Language implementation systems (compilation, pure interpretation, and hybrid) must analyze source code.
- Syntax analyzers, or parsers, are almost always based on the formal description of the syntax of the source language – BNF
- Nearly all compilers separate the task of analyzing syntax into two distinct parts, lexical analysis and syntax analysis.
    - Lexical analysis deals with small-scale language constructs
        - names and literals
    - Syntax analysis deal with large-scale language constructs
        - expressions, statements, and program units

Reasons why lexical analysis is separated from syntax analysis:

- Simplicity
    - Lexical analysis can be simplified because its techniques are less complex than syntax analysis
    - The syntax analyzer can be smaller and cleaner by removing the low-level details of lexical analysis
- Efficiency – separate selective optimization
    - Lexical analysis should be optimized because is requires a significant portion of total compile time.
    - The syntax analyzer should not be optimized.
- Portability
    - The lexical analyzer is somewhat system dependent – input processing
    - The syntax analyzer is not system dependent

## Lexical Analysis

- The lexical analyzer is a pattern matcher for character strings.
- The lexical analyzer serves as the front end of the syntax analyzer.
- The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to structure.
    - Character groupings are called lexemes.
    - The internal codes are called tokens.

result = oldSum – value / 100;

| **Lexemes** | **Tokens** |
|---|---|
| result | IDENT |
| = | ASSIGN_OP |
| oldSum | IDENT |
| - | SUBTRACT_OP |
| value | IDENT |
| / | DIVISION_OP |
| 100 | INT_LIT |
| ; | SEMICOLON |

The lexical analyzer is usually a subprogram that called by the parser when it needs the next token.

Lexical analysis process also includes

- Skipping comments
- Inserting lexemes for user-defined names into the symbol table
- Detects syntactic errors in tokens – e.g., ill-formed floating-point literals

Three approaches to building a lexical analyzer:

- Write a formal description of the token patterns and use a software tool that constructs table-driven lexical analyzers based on the given description
- Design a state transition diagram that describes the token patterns of the language and write a program that implements the state 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 of a state diagram are labeled with state names. The arcs are labeled with the input characters that cause transitions. An arc may also include actions the lexical analyzers must perform.

State diagrams of the form used for lexical analyzers are representations of a class of mathematical machine called finite automata. Finite automata can be designed to recognize a class of languages called regular languages. Regular expressions and regular grammars are generative devices for regular languages.

The state diagram could include states and transitions for each and every token pattern, however, this would yield a very large and complex diagram. In many cases, transitions can be combined to simplify the state diagram:

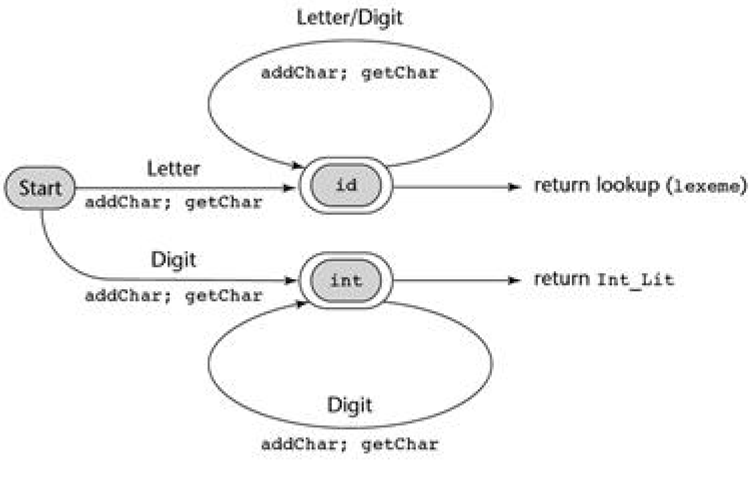
Lexical analyzer – recognizes names, reserved words, and integer literals

- Names
  - Starts with a letter followed by letters and digits – unlimited length
  - Lexical analyzer cares that it is a name, but not which specific name
    - Define a character class that represents all letters (upper and lower case) – requires 1 transition instead of 52
  - Names and reserved words have a similar pattern
    - Treat all words/names the same and use a reserved word lookup table instead of individual reserved word transitions
- Integer literals
  - Define a digit class that represents all digits – requires 1 transition instead of 10

Utility subprograms used to common tasks

- getChar
  - Reads the next character from input program and stores in variable nextChar
  - Determines character class and stores in variable charClass
- addChar
  - places nextChar in variable lexeme used for building the next lexeme
- lookup
  - Determines whether variable lexeme contains a name or a reserved word

State diagram describes the patterns of the supported tokens for this example.



```
int lex( ) {                            # simple lexical analyzer
    getChar( );
    switch( charClass ) {
        case LETTER:                    # identifiers and reserved words
            addChar( );
            getChar( );
            while ( charClass == LETTER || charClass == DIGIT ) {
                addChar( );
                getChar( );
            }
            return lookup( lexeme );    # 0 identifiers, otherwise token
            break;                      # code for reserved words
        case DIGIT:                     # integer literals
            addChar( );
            getChar( );
            while ( charClass == DIGIT ) {
                addChar( );
                getChar( );
            }
            return INT_LIT;
            break;
    }   /*  end switch  */
}   /*  end lex  */
```

## The Parsing Problem

The process of analyzing syntax that is referred to as syntax analysis is often called parsing.

Introduction to Parsing

Two goals of syntax analysis:

- Verify that the program is syntactically correct.
    - Produce meaningful feedback and recover when errors are found.
- Produce the parse tree, or at least a trace of the parse tree, for the program.

Two classes of parsers:

- Top-down – tree built from the root down to the leaves
- Bottom-up – tree built from the leaves upward towards the root

Discussion notation:

- Terminals – (a, b, …)
- Nonterminals – (A, B, …)
- Terminals or nonterminals – (W, X, Y, Z)
- Strings of terminals – (w, x, y, x)
- Mixed strings (terminal and/or nonterminals) – (α, β, δ, γ)

## Top-Down Parsers

- Traces or builds the parse tree in preorder – each node is visited before its branches – leftmost derivation.
- Given sentential form, xAα, the parser must choose the correct A-rule to get to the next sentential form in the leftmost derivation, using only the first token produced by A.
    - A-rules might be: A -> bB, A -> cBb, A -> a
- A recursive descent parser is a coded version of a syntax analyzer based directly on the BNF description of the language.
- LL algorithms – left-to-right scan of the input, leftmost derivation

## Bottom-Up Parsers

- Builds a parse tree by beginning at the leaves and progressing toward the root.
- Given a right sentential form, α, the parser must determine what substring of ά is the RHS of the rule in the grammar that must be reduced to its LHS to produce the previous sentential form in the rightmost derivation.
- A given right hand sentential form may include more than one RHS from the grammar of the language being parsed. The correct RHS is called the handle.
- LR algorithms – left-to-right scan of the input, rightmost derivation.

## The Complexity of Parsing

- Parsing algorithms that work for any unambiguous grammar are complex and inefficient – O($N^3$) where N is the input length.
- Faster algorithms have been found that work for only a subset of the set of all possible grammars – O(N) where N is the input length. Algorithms that work for only subsets of the set of all grammars are acceptable as long as the subset includes grammars that describe programming languages.

**Recursive-Descent Parsing**

- Consists of a collection of subprograms – many of which are recursive.
- Produces a parse tree in top-down (descending) order.
- EBNF is ideally suited for recursive-descent parsers.
- Each nonterminal in the grammar is represented by a subprogram.
    - Used to trace out the parse tree that can be rooted at that nonterminal.

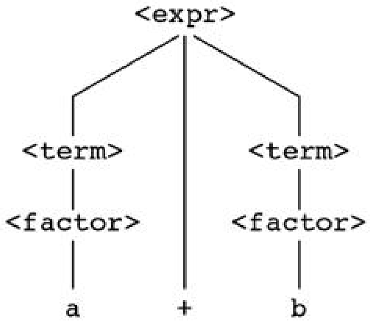
```
<expr>      ->  <term> {( + | - ) <term> }          include as part of the
<term>      ->  <factor> {( * | / ) <factor> }      subprogram documentation
<factor>    ->  <id> | ( <expr> )
```

```
void expr( ) {                           /*  nextToken always contains the next  */
    term( );                             /*  token to be processed – the lookahead  */
    while ( nextToken == PLUS_CODE || nextToken == MINUS_CODE ) {
        lex( );                          /*  lex places the next token to be  */
        term( );                         /*  processed in nextToken  */
    }
}

void term( ) {
    factor( );
    while ( nextToken == MULT_CODE || nextToken == DIV_CODE ) {
        lex( );
        factor( );
    }
}

void factor( ) {
    if ( nextToken == ID_CODE )                        /*  <id> RHS  */
        lex( );
    else if ( nextToken == LEFT_PAREN_CODE ) {         /*  ( <expr> ) RHS  */
        lex( );
        expr( );
        if ( nextToken == RIGHT_PAREN_CODE )
            lex( );
        else
            error( );
    } else                               /*  invalid factor token found  */
        error( );
}
```

Parse tree for a + b using the previously described grammar and recursive-descent parser.



Java if statement:
<ifstmt>      ->  if ( <boolexpr> ) <statement> [ else <statement> ]

```
void ifstmt( ) {
        if ( nextToken != IF_CODE )             /*  if is terminal – must be tested  */
                error( );
        else {
                lex( );
                if ( nextToken != LEFT_PAREN_CODE )
                        error( );
                else {
                        boolexpr( );
                        if ( nextToken != RIGHT_PAREN_CODE )
                                error( );
                        else {
                                statement( );
                                if ( nextToken == ELSE_CODE ) {    /*  optional  */
                                        lex( );                    /*  no else  */
                                        statement( );              /*  no error  */
                                }
                        }
                }
        }
}
```

These examples are intended to show that a recursive-descent parser can be easily written if an appropriate grammar is available for the language.

The LL Grammar Class

- Left recursion problem:
    - A -> A + B (direct left recursion)
        - A calls A, then A calls A, then A calls A, etc…
    - A -> B a A and B -> A b (indirect left recursion)
        - A calls B, then B calls A, then A calls B, etc…
    - Aho 1986 – algorithm to remove both of these problems.
- Pairwise disjointness:
    - Inability to choose the correct RHS based on the next token.
    - Def: FIRST(α) = {a | α =>* aβ} (If α =>* ε, ε is in FIRST(α))
        - =>* means 0 or more derivation steps

Pairwise Disjointness Test:
For each nonterminal, A, in the grammar that has more than one RHS, for each pair of rules, A -> $\alpha_i$ and A -> $\alpha_j$, it must be true that: FIRST($\alpha_i$) ∩ FIRST($\alpha_j$) = {}

| | | |
|---|---|---|
| A -> a \| bB \| cAb | FIRST sets {a}, {b}, {c} | Pass |
| A -> a \| aB | FIRST sets {a}, {a} | Fail |

In many cases a grammar that fails the pairwise disjointness test can be modified so that it will pass by using left factoring.

Replace
<variable> -> identifier | identifier [ <expression> ]
with
<variable> -> identifier <new>
<new> -> ε | [ <expression> ]   sometimes λ instead of ε
or
<variable> -> identifier [ [ <expression> ] ]   outer [] EBNF metasymbols meaning optional

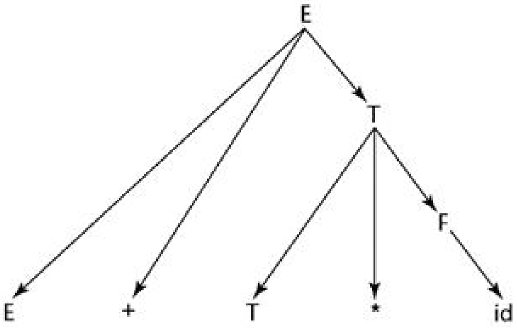
The Parsing Problem with Bottom-Up Parsers

The task of a bottom-up parser is to find the handle of any given right sentential form that can be generated by its associated grammar.

Def: $\beta$ is the **handle** of the right sentential form $\gamma = \alpha\beta w$ if and only if
S =>*rm $\alpha$Aw =>rm $\alpha\beta$w
=>rm specifies a rightmost derivation
=>*rm specifies 0 or more rightmost derivation steps

Def: $\beta$ is a **phase** of the right sentential form $\gamma$ if and only if
S =>* $\gamma = \alpha_1 A \alpha_2$ =>+ $\alpha_1 \beta \alpha_2$
=>+ specifies 1 or more derivation steps

Def: $\beta$ is a **simple phase** of the right sentential form $\gamma$ if and only if
S =>* $\gamma = \alpha_1 A \alpha_2$ => $\alpha_1 \beta \alpha_2$
=> specifies exactly 1 step



The parse tree leaves comprise the sentential form E + T * id. The parse tree consists of three internal nodes so there are 3 phases. Each internal node is the root of a subtree, whose leaves are a phase.

E (root node) generates the resulting sentential form E + T * id, which is a phase.
T (internal node) generates the leaves T * id, which is a phase.
F (internal node) generates id, which is a phase – also a simple phase.

Intuitions about handles:

- The handle of a right sentential form is its leftmost simple phase
- Given a parse tree, it is easy to find the handle
- Parsing can be thought of as handle pruning

Shift-Reduce Algorithms

- Bottom-up parsers are often called shift-reduce algorithms.
- The shift action moves the next input token onto the parser's stack.
- The reduce action replaces a RHS (handle) on top of the parser's stack by its corresponding LHS.

Pushdown automaton – PDA

- Simple mathematical machine that scans strings of symbols form left to right.
- So named because it uses a pushdown stack as its memory.
    - Recursive-descent parser uses the system runtime stack.
- Can be used as language recognizers.
    - Can determine if a string is a sentence in a language.
    - Can produce the information needed to construct a parse tree.

LR Parsers

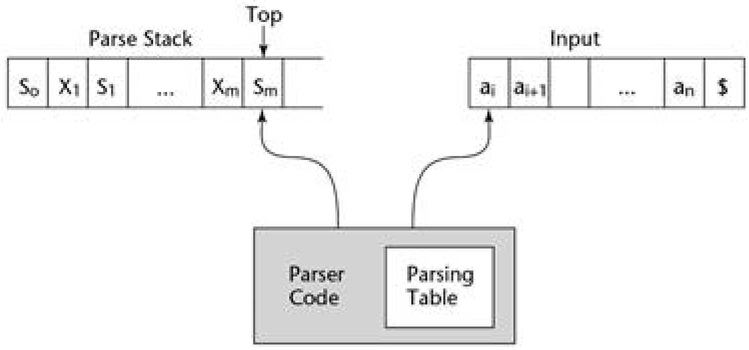
Use a relatively small program and a parsing table.

Advantages of:

- Will work for nearly all programming language grammars.
- Work on a larger class of grammars than other bottom-up algorithms but are as efficient as any other bottom-up parser.
- Can detect syntax errors as soon as possible.
- The LR class of grammars is a superset of the class parsable by LL parsers.

Disadvantages Of:

- It is difficult to produce by hand the parsing table for a given grammar.
    - Many available programs can produce parse tables from grammar input.



LR parser stack: S – state symbols; X – grammar symbols

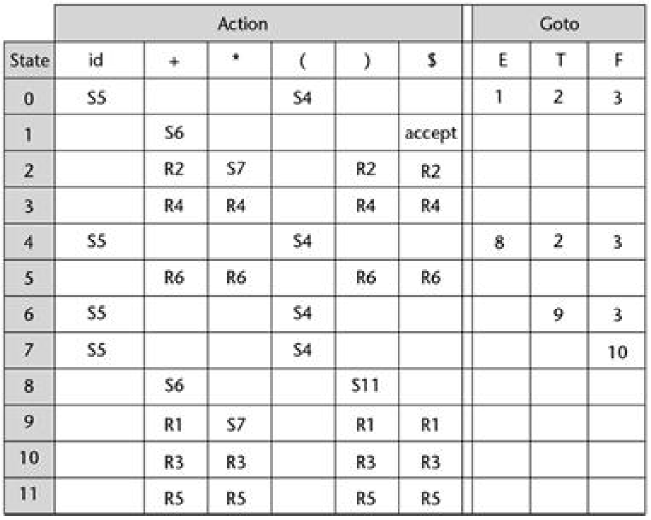
LR parser configuration – pair of strings (stack, input) – $(S_0X_1S_1\ldots X_mS_m,a_i\ldots a_n\$)$

- LR parser is based on the parsing table.
- LR parsing table has two parts:
    - The ACTION part specifies the action of the parser, given the parser state and the next token.
        - Rows are state symbols and columns are terminal symbols.
    - The GOTO part specifies which state to put on top of the parse stack after a reduction action is completed.
    - Rows are state symbols and columns are nonterminal symbols.
- Initial configuration of an LR parser is: $(S_0,a_i \ldots a_n\$)$
- Definitions of the parser actions:
    - If ACTION$[S_m, a_i]$ = Shift S, the next configuration is $(S_0X_1S_1 \ldots X_mS_m,a_i \ldots a_n\$)$
    - If ACTION$[S_m, a_i]$ = Reduce A -> β and S = GOTO$[S_{m-r}, A]$, where r = the length of β, the next configuration is $(S_0X_1S_1 \ldots X_{m-r}S_{m-r},a_i \ldots a_n\$)$
    - 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 and error-handling routine.
- Initially, the parse stack has the single symbol 0, state 0 of the parser.
- The input contains the input string with an end marker, $, at the right end.
- At each step, the parser actions are dictated by the top (rightmost) symbol of the parse stack and the next (leftmost) token input.
- The correct action is chosen from the corresponding cell of the ACTION table.
- The GOTO table is used after a reduction action.

Example:

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

R means reduce : R4 means reduce rule 4
S means shift : S6 means shift next input symbol onto stack
and push state 6 onto the stack
Empty entries in the Action table indicate syntax errors
Input string : id + id * id



| | 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 | | | |

| **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 |