4. Lexical and Syntax Analyzer

In this chapter:
- An introduction to lexical analysis,
- Two primary approaches to parsing and complexity
- Recursive-descent technique for LL parsers
- Bottom-up parsing and LR parsing algorithm

4.1 Introduction

Three different approaches to implementing languages are
1. compilation (C++, COBOL, ADA)
2. pure interpretation (JavaScript, UNIX shell scripts, Lisp)
3. hybrid implementation (Java, Perl)

Syntax analyzers: S_A (parsers) are nearly always based on formal description of the syntax of programs.

The most commonly used syntax description formalism is CFG or BNF

Advantages of using BNF
1. syntax description is clear and concise for both humans and for software systems using them.
2. BNF syntax description can be used as the basis for syntax analyzer.
3. implementations based on BNF are relatively easy to maintain because of modularity.

Almost all compilers separate the task of analyzing syntax into two distinct parts
First: the lexical analyzer deals with small-scale language constructs as names and numeric literals
Second: the syntax analyzer deals with large scale constructs, expressions, statements, and program units.

Why lexical analysis is separated from the syntax analysis
- Simplicity- lexical analysis is less complex so it is much simpler if it is separated from the syntax analyzer.
- Efficiency- it makes easier to optimize lexical analyzer and syntax analyzer.
- Portability- At some point lexical analyzer is platform dependent (remember it reads the input stream) However, the S_A can be platform independent.

It is always good idea to separate machine dependent part from the software.

4.2 Lexical Analysis

Lexical analyzer is essentially a pattern matcher. The earliest uses of P_M was with text editors (Unix ed line editor, Perl, or JavaScript).
A L_A serves as front end of S_A. Technically L_A is S_A at the lowest level of program structures.
The L_A collect characters (from the input stream) into logical groups and assigns internal codes
to (often referenced by named construct for the sake of readability) the groupings according to their
structure.

These groupings are called lexemes. The internal codes are called tokens.

**Example:**

```
sum = aldsum – value/100;
```

<table>
<thead>
<tr>
<th>token</th>
<th>lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
<td>sum</td>
</tr>
<tr>
<td>ASSIN_OP</td>
<td>=</td>
</tr>
<tr>
<td>IDENT</td>
<td>aldsum</td>
</tr>
<tr>
<td>SUBT_OP</td>
<td>-</td>
</tr>
<tr>
<td>IDENT</td>
<td>value</td>
</tr>
<tr>
<td>DIVIS_OP</td>
<td>/</td>
</tr>
<tr>
<td>INT_LIT</td>
<td>100</td>
</tr>
<tr>
<td>SEMICOLON</td>
<td>;</td>
</tr>
</tbody>
</table>

L_As extracts lexemes from a given input and produce the corresponding tokens.

However, now days most L_A are subprograms that produces next lexeme and its associated
token code from the input and return them to the caller(S_A). So the only view of the input
program seen by S_A is the output of the L_A, one lexeme at a time.

The L_A also skips comments, and blanks outside lexemes and inserts lexemes for user-defined
names into symbol table. Finally, the L_As detect syntactic errors in tokens, such as ill-formed
floating-point literals, and report such error to the users.

There are three basic approaches to building L_A:

1. Write a formal description of the token pattern of the language using a descriptive
   language related to regular expressions and use a software (special program) to
   automatically generate L_A. (UNIX lex program)
2. Design a state transition diagram that describes the token pattern of the language and write
   a program that implements the diagram.
3. 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, is graph like the syntax graph introduced in chapter 3.

The nodes are labeled with state names.

The arcs are labeled with the input characters that causes transitions. An arc may also include
an actions the L_A must do when the transition is taken. This is nothing but so called finite
automata (mathematical) machines. FAM as you remember can be designed to recognize a class
of languages called regular languages. Regular expressions and regular grammars are generative
devices for regular languages.

*The tokens of a programming language are regular language.*
Example: L_A construction with the state diagram and the code that implements it.
The state diagram could include states and transitions for every token pattern. (it could be very
large of course). Let assume we need a L_A that only recognizes:

- **program names**, (strings that starts with letter followed by letter or digits no length restriction)
- **reserve wards** (same as names)
- **integer literals**.

First: observe that it is possible to build a state diagram to recognize every single reserve ward of
language but that would result a huge state diagram.

It is much faster and easier to have the L_A recognize the names and the reserve wards with the
same pattern and use lookup table of reserve wards to determine which names are reserved wards.

Second: one can introduce two classes of characters to simplify the state diagram: Letter
(representing 52 characters) and Digit (for 10 digits)

Next: we can define several utility subprograms for the common tasks inside the L_A

1. **getChar**:: gets the next character from the input and puts it in the global variable
   **nextChar** it also determines the character class of input and put it in the global variable
   **charClass**.

The lexemes being build by L_A, which could be character string or character array we name
**lexeme**.

The subprogram **addChar** implements the process of putting **nextChar** into the **lexeme**.

Finally, we need a subprogram named **lookup** to determine whether the current content of **lexeme**
is reserve ward, or name. This subprogram returns 0 (if **lexeme** is not a reserve ward) or token
code otherwise (the codes are nonzero numbers assigned to tokens be compiler writer)

Here is the desired state transition diagram

![State Diagram](image-url)
The following C function is an example of implementation of a state diagram described:

```
// lex - simple lexical analyzer
int lex()
{
    get Char();
    switch(charClass) {  // parse identifiers and reserve words
        case LETTER:  // parse identifiers and reserve words
            addChar();
            getChar();
            while(charClass = = LETTER || charClass = = DIGIT){
                addChar();
                getChar();
            }
            return lookup.lexeme  // Parse integer literals
        break;
    }
    getChar();
    break;           // Parse integer literals

    case DIGIT:        // Parse integer literals
        addChar();
        getChar();
        while( charClass = = DIGIT){
            addChar();
            getChar();
        }
        return INT_LET
    break;
}
```

### 4.3 The Parsing Problem

The part of the process of analyzing syntax that is referred to as syntax analysis $S_A$ is often called **parser**. Next we will discuss the general parsing problems and two main categories of parsing algorithms: Top-down, Bottom-up, and also complexity of the parsing process.

**Introduction to Parsing:**
Parsers, -construct parse tree for given program. In most cases information required to build parse tree is generated. There are two distinct goals of syntax analysis.

First: check whether or not an input program is synthetically correct. (in case of error it must produce a diagnostic message and recovery).

Second: produce either a complete parse tree or at least trace the structure of the complete parse tree.

In either case, the result is used as the basis for translation.

The parsers are categorized according to the direction in which they build parse tree.

Top-down- tree is build from the root downward to the leaves;

Bottom-up- tree is build from the leaves upward to the root. All commonly used parsing algorithms operate under the same constraint that they never look ahead more than one token into the input program. It results in elegant and efficient parsing.

Some notational conventions:

Terminals: lowercase letters (a, b, c, …)

Nonterminals: uppercase letters at the beginning of the alphabet (A, B, C, …)

Terminals or noterminals: uppercase letters at the end of alphabet (W, X, Y, Z)

String of terminals: Lowercase letters at the end of alphabet. (w, x, y, z)

Mixed strings (term or nonterm): Lowercase Greek letters (α, β, δ, γ)

For the programming languages, terminal symbols are the small-scale syntactic constructs of language including the punctuation symbols and numeric literals. Nonterminals (<expr>, <function_def>, e.g.) are connotative names or abbreviations surrounded by ‘< ’.

Top-down Parser
In terms of derivation, the top-down parser can be describe as:

Given a sentential form that is part of l-m-derivation the parsers task is to find the next sentential form in the l–m- derivation.

Example :

If the current sentential form is xAα and A-rules are A → bB, A → cBb, and A → a

Then next sentential form could be only xbBα, xcBbα, xaα

Under the constraints of one token ahead, a top-down parser must choose the correct RHS on the basis of the next token of the input program. The most common top-down parsing algorithms are closely related.

A recursive descent parser is a coded version of a syntax analyzer based directly on BNF description of the language. Rather then code one can use also a parsing table to implement the BNF rules. Thus are called LL algorithms and they are equally powerful.

Bottom-up Parser
Constructs the parse tree by the beginning at the leaves and progressing toward the root. It produces the reverse of rightmost derivation. Thus are called LR algorithms.

The Complexity of Parsing
Parsing algorithm that works for any unambiguous grammar are complex and inefficient (O(n^3)) But all the algorithms used for the syntax analyzers of compilers have complexity of O(n). Thus algorithms usually work for only subset of rules describing the language.
4.4 Recursive-Descent Parsing Process

A recursive-descent parser (RDP) consists of a collection of subprograms (mostly recursive) that produces a parse tree in top-down (descending) order.

EBNF is ideally suited for RDP.

A RDP has a subprogram for each nonterminal in the language grammar. The subprogram associated with particular nonterminal is as follows:

For given input string, it traces out the parse tree that can be rooted at that nonterminal and whose leaves match the input string.

**Example:**

Consider the following EBNF description of simple arithmetic expressions:

\[
\begin{align*}
<\text{expr}> & \rightarrow <\text{term}> \{( + | - ) <\text{term}> \} \\
<\text{term}> & \rightarrow <\text{factor}> \{(* | / ) <\text{factor}> \} \\
<\text{factor}> & \rightarrow \text{id} | ( <\text{expr}> )
\end{align*}
\]

Let remind that L_A is a function `lex()` that gets next lexeme and puts its code in the global variable `nextToken`.

A recursive-descent subprogram for the rule with single RHJS is relatively easy:

For each terminal symbol in RHS, that is compared with `nextToken` if no match then syntax error if they match, L_A is called to get next input token.

For each nonterminal the parsing subprogram for that nonterminal is called.

**Rule** \[<\text{expr}> \rightarrow <\text{term}> \{( + | - ) <\text{term}> \}\]

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

RDP subprograms are written with the convention that each one leaves the next token of input in `nextToken`. 
A RDP subprogram with more than one RHS begins with the code to determine which RHS is to be parsed. Here is the program for our `<factor>` nonterminal

```c
void factor() {              // first determine which RHS to parse
    if (nextToken == ID_CODE)
        lex();
    else if (nextToken == LEFT_PR_CODE)
        lex();
        expr();
        if (nextToken == RIGHT_PR_CODE)
            lex();
        else
            error();
    else
        error();
}
```

**Rule**   

```c
<factor> -> id | (<expr>)
```

The LL grammar class

Left recursion causes problem for LL parser

**Example**   

```
A -> A + B
```

A RDP subprogram for A first calls itself immediately and so forth. The same problem poses for this case as well

```
A -> BaA
B -> Ab
```

This is a problem for all top-down recursive descent parsers (fortunately not for bottom up parsing algorithms). However there is an algorithm to modify a given grammar rules to remove both direct and indirect left recursions.

Another grammar trait that disallows top-down parsing is whether the parser can always choose the correct right side on the basis of the next token of input. This is relatively easy test for non-left recursive grammar (pairwise disjointness test).

The test requires to compute set $\varepsilon$

$$\text{FIRST}(\alpha \varepsilon) = \{ a \mid \alpha \Rightarrow a\beta\}$$

An algorithm to compute $\text{FIRST}$ for any mixed string $\alpha$ can be found in Aho (1986). But you can compute the $\text{FIRST}$ by inspecting grammar rules
For each pair of rules $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$ it must be true that

$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$$

Example

$$A \rightarrow aB \mid bAb \mid c$$

Example

$$A \rightarrow aB \mid aAb$$

In many cases a grammar that fails to pass $\text{FIRST}$ test can be modified so that it will pass the test.

Example

$$\langle\text{var}\rangle \rightarrow \text{ident} \mid \text{ident} \langle\text{expr}\rangle$$

this rule does not pass the test (both start with ident terminal). However, the problem can be settled by applying so called left factoring process.

$$\langle\text{var}\rangle \rightarrow \text{ident} \langle\text{new}\rangle$$
$$\langle\text{new}\rangle \rightarrow \epsilon \mid \langle\text{expr}\rangle$$