Lexical and Syntax Analysis

In Text: Chapter 4
N. Meng, F. Poursardar
Lexical and Syntactic Analysis

- Two steps to discover the syntactic structure of a program
  - Lexical analysis (Scanner): to read the input characters and output a sequence of tokens
  - Syntactic analysis (Parser): to read the tokens and output a parse tree and report syntax errors if any
Compilation Process
Interaction between lexical analysis and syntactic analysis
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 is always portable.
Scanner

- Pattern matcher for character strings
  - If a character sequence matches a pattern, it is identified as a token

- Responsibilities
  - Tokenize source, report lexical errors if any, remove comments and whitespace, save text of interesting tokens, save source locations, (optional) expand macros and implement preprocessor functions
Tokenizing Source

- Given a program, identify all lexemes and their categories (tokens)
Lexeme, Token, & Pattern

- **Lexeme**
  - A sequence of characters in the source program with the lowest level of syntactic meanings
    - E.g., sum, +, -

- **Token**
  - A category of lexemes
  - A lexeme is an instance of token
  - The basic building blocks of programs
## Token Examples

<table>
<thead>
<tr>
<th>Token</th>
<th>Informal Description</th>
<th>Sample Lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>All keywords defined in the language</td>
<td>if else</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;, &gt;, &lt;=, &gt;=, ==, !=</td>
<td>&lt;=, !=</td>
</tr>
<tr>
<td>id</td>
<td>Letter followed by letters and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric constant</td>
<td>3.14159, 0, 6</td>
</tr>
<tr>
<td>literal</td>
<td>Anything surrounded by “”, but exclude “”</td>
<td>“core dumped”</td>
</tr>
</tbody>
</table>
Another Token Example

Consider the following example of an assignment statement:

\[
\text{result} = \text{oldsum} - \text{value} / 100;
\]

• Following are the tokens and lexemes of this statement:

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
<td>result</td>
</tr>
<tr>
<td>ASSIGN_OP</td>
<td>=</td>
</tr>
<tr>
<td>IDENT</td>
<td>oldsum</td>
</tr>
<tr>
<td>SUB_OP</td>
<td>-</td>
</tr>
<tr>
<td>IDENT</td>
<td>value</td>
</tr>
<tr>
<td>DIV_OP</td>
<td>/</td>
</tr>
<tr>
<td>INT_LIT</td>
<td>100</td>
</tr>
<tr>
<td>SEMICOLON</td>
<td>;</td>
</tr>
</tbody>
</table>
Lexeme, Token, & Pattern

- Pattern
  - A description of the form that the lexemes of a token may take
  - Specified with regular expressions
Motivating Example

- Token set:
  - assign -> :=
  - plus -> +
  - minus -> -
  - times -> *
  - div -> /
  - lparen -> (  
  - rparen -> )
  - id -> letter(letter|digit)*
  - number -> digit digit*|digit*.(digit|digit.)digit*
Motivating Example

• What are the lexemes in the string “a_var:=b*3” ?
• What are the corresponding tokens ?
• How do you identify the tokens?
Lexical Analysis

• Three approaches to build a lexical analyzer:
  – Write a formal description of the tokens and use a software tool that constructs a table-driven lexical analyzer from 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
State Diagram

- A **state transition diagram**, or just 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 the transitions among the states.
- An arc may also include actions the lexical analyzer must perform when the transition is taken.
State Diagram

- State diagrams of the form used for lexical analyzers are *representations of* a class of mathematical machines called **finite automata**.
- Finite automata can be designed to recognize members of a class of languages called **regular languages**.
- Regular grammars are generative devices for regular languages.
- The tokens of a programming language are a regular language, and a *lexical analyzer is a finite automaton*.
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!

• Reason? Because every node in the state diagram would need a transition for every character in the character set of the language being analyzed.

• Solution: Consider ways to simplify
State Diagram Design - Example

- Design a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands.
- Assume that the variable names consist of strings of uppercase letters, lowercase letters, and digits but must begin with a letter.
- Names have no length limitation.

- How many transitions for initial state?
- How can we simplify it?
Example (continued)

• There are 52 different characters (any uppercase or lowercase letter) that can begin a name, which would require 52 transitions from the transition diagram’s initial state.
• However, a lexical analyzer is interested only in determining that it is a name and is not concerned with which specific name it happens to be.
• Therefore, we define a character class named LETTER for all 52 letters and use a single transition on the first letter of any name.
Another opportunity for simplifying the transition diagram is with the integer literal tokens. There are 10 different characters that could begin an integer literal lexeme. This would require 10 transitions from the start state of the state diagram. Define a character class named DIGIT for digits and use a single transition on any character in this character class to a state that collects integer literals.
Lexical Analysis (continued)

- 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 (continued)

• 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 (continued)

- 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
  - `lookup` - determines whether the string in lexeme is a reserved word (returns a code)
/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();

/* 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
static TOKEN nextToken;

static CHAR_CLASS charClass;

int lex() {
    switch (charClass) {
    case LETTER:
        // add nextChar to lexeme
        addChar();
        // get the next character and determine its class
        getChar();
        while (charClass == LETTER || charClass == DIGIT) {
            addChar();
            getChar();
        }
        nextToken = ID;
        break;
    }
case DIGIT:
    addChar();
    getChar();
    while (charClass == DIGIT) {
        addChar();
        getChar();
    }
    nextToken = INT_LIT;
    break;

... case EOF:
    nextToken = EOF;
    lexeme[0] = 'E';
    lexeme[1] = 'O';
    lexeme[2] = 'F';
    lexeme[3] = 0;
}
printf("Next token is: %d, Next lexeme is %s\n",
    nextToken, lexeme);
    return nextToken;
}  /* End of function lex */
Lexical Analyzer

Implementation:
→ front.c (pp. 166–170)

Following is the output of the lexical analyzer of front.c when used on (sum + 47) / total

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
The Parsing Problem

- Given an input program, the goals of the parser:
  - 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
The Parsing Problem (continued)

- 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)
Two Classes of Grammars

- Left-to-right, Leftmost derivation (LL)
- Left-to-right, Rightmost derivation (LR)
- We can build parsers for these grammars that run in linear time
## Grammar Comparison

<table>
<thead>
<tr>
<th></th>
<th>LL</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>→ TE’</td>
<td>E → E + T</td>
</tr>
<tr>
<td>E’</td>
<td>→ + TE’</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>→ FT’</td>
<td>F → id</td>
</tr>
<tr>
<td>T’</td>
<td>→ * FT’</td>
<td>ε</td>
</tr>
<tr>
<td>F</td>
<td>→ id</td>
<td></td>
</tr>
</tbody>
</table>
Two Categories of Parsers

• LL(1) Parsers
  – L: scanning the input from left to right
  – L: producing a leftmost derivation
  – L: using one input symbol of lookahead at each step to make parsing action decisions

• LR(1) Parsers
  – L: scanning the input from left to right
  – R: producing a rightmost derivation in reverse
  – L: the same as above
Two Categories of Parsers

• **LL(1) parsers (predicative parsers)**
  – **Top down**
    o Build the parse tree from the root
    o Find a left most derivation for an input string

• **LR(1) parsers (shift-reduce parsers)**
  – **Bottom up**
    o Build the parse tree from leaves
    o Reducing a string to the start symbol of a grammar
Top-down Parsers

• Given a sentential form, $xA\alpha$, 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
Bottom-up parsers

- Given a right sentential form, \( \alpha \), determine what substring of \( \alpha \) 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.
Recursive Descent Parsing

• Parsing is the process of tracing or **constructing a parse tree** for a given input string
• Parsers usually do not analyze lexemes; that is done by a lexical analyzer, which is called by the parser
• A **recursive descent parser** traces out a parse tree in top-down order; it is a **top-down parser**
• Each nonterminal has an **associated subprogram**; the subprogram parses all sentential forms that the nonterminal can generate
• The recursive descent parsing subprograms are **built directly from the grammar rules**
• Recursive descent parsers, like other top-down parsers, cannot be built from **left-recursive grammars**
Recursive Descent Example

Example: For the grammar:

\[ \text{<term>} \rightarrow \text{<factor>} \{ (\ast \mid /) \text{<factor>} \} \]

Simple recursive descent parsing subprogram:

```c
void term() {
    factor(); /* parse the first factor*/
    while (next_token == ast_code ||
           next_token == slash_code) {
        lexical(); /* get next token */
        factor(); /* parse the next factor */
    }
}
```
Top-down (left) and bottom-up parsing (right) of the input string A, B, C;
Grammar appears at lower left.