Lexical and Syntax Analysis

In Text: Chapter 4

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

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)
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 building 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 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 (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)

Implementation Pseudo-code

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;
            break;
    }
    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

• 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
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>LL</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow E + T</td>
</tr>
<tr>
<td>$E' \rightarrow + TE'</td>
<td>\epsilon$</td>
</tr>
<tr>
<td>$T \rightarrow F T'$</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$T' \rightarrow * F T'</td>
<td>\epsilon$</td>
</tr>
</tbody>
</table>

Two Categories of Parsers

- LL(1) Parsers
  - L: scanning the input from left to right
  - L: producing a leftmost derivation
  - 1: 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
  - 1: the same as above

Two Categories of Parsers

- LL(1) parsers (predicative parsers)
  - Top down
    - Build the parse tree from the root
    - Find a leftmost derivation for an input string
- LR(1) parsers (shift-reduce parsers)
  - Bottom up
    - Build the parse tree from leaves
    - 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, 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.