Lexical and Syntax Analysis

Hui Chen

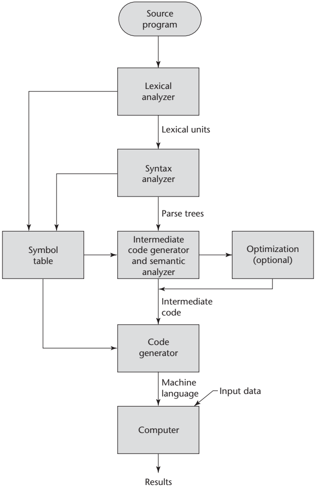
Computer Science
Virginia State University
Petersburg, Virginia

January 20, 2016

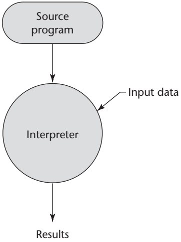
Acknowledgement

- Slides are prepared based on the textbook [Sebesta, 2012].

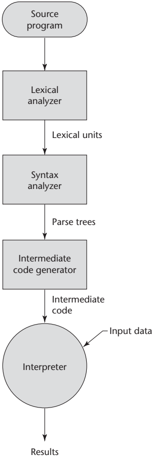
Language Implementation



(a) Compilation



(b) Pure Interpretation



(c) Hybrid Implementation

Syntax Analysis

- Consisting of two parts
  - Lexical analyzer (a finite automaton/finite state machine based on a regular grammar)
  - Syntax analyzer (a pushdown automaton based on a context-free grammar)

Lexical Analyzer

- Front-end for the parser
- Identifies *lexemes* and the tokens to which they belong
- Example: consider Java statement

```
index = 2 * count + 17;
```

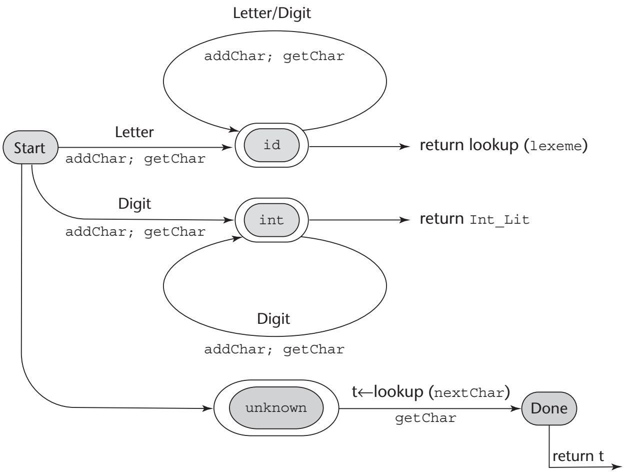
Lexeme	Token
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Building Lexical Analyzer

- Directly implementing the state diagram of a finite automaton from scratch
  - Design a state diagram that describes the tokens
  - write a program that implements the state diagram
- Implementing the state diagram of a finite automaton using a table-driven approach
  - Design a state diagram that describes the tokens
  - Hand-construct a table-driven implementation of the state diagram
- Implementing a finite automaton using a table-driven approach with a software tool
  - Write a formal description of the tokens
  - Use a software tool that constructs a table-driven lexical analyzer from formal description of tokens

An Example of Lexical Analyzer

- State Diagram



An Example of Lexical Analyzer

- Implementation: In Github

Obtaining Program from Github and Run Example on Linux System

```
$ git clone https://github.com/huichen-cs/sebesta.git
$ cd sebesta/lexer
$ make lexer
$ make test
```

The Example of Lexical Analyzer in Lex

- Implementing a finite automaton using a table-driven approach with a software tool
  - Write a formal description of the tokens
  - Use a software tool that constructs a table-driven lexical analyzer from formal description of tokens
  - Example software tool: Lex (C, Java, Python ...)

Run Example on Linux System

```
$ cd sebesta/lexer/lex
$ make test
```

Syntax Analysis

- Syntax analysis is also called *parsing*.
- Top-down parsing
- Tottom-up parsing
- Complexity of parsing

Goal of Parsing

- Determine whether an input program is syntactically correct, produce a diagnostic message and recover.
- Produce a complete parse tree, or at least trace the structure of the complete parse tree, for syntactically correct input for translation.

Categories of Parser

- Top down
  - Produce the parse tree, beginning at the root
  - Order is that of a leftmost derivation
  - Traces or builds the parse tree in preorder
- Bottom up
  - Produce the parse tree, beginning at the leaves
  - Order is that of the reverse of a rightmost derivation
  - Useful parsers look only one token ahead in the input

Top-Down Parser

- 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, where x is a string of terminal symbols, α is a mixed string of terminals and nonterminals, and A is a nonterminal.
- The most common top-down parsing algorithms:
  - Recursive descent: a coded implementation
  - LL parsers: a table driven implementation

Top-Down Parser: Example

- Given $xA\alpha$ and A-rules,

$$A \rightarrow bB$$
$$A \rightarrow cBb$$
$$A \rightarrow a$$

which one of the three rules to choose to get the next sentential form, which could be xbB, $xcBb$, or xa.

Bottom-Up Parser

- Given a right sentential form, α, a mixed string of terminals and nonterminals, 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

Bottom-Up Parser: Example

- Consider the following grammar,

$$S \to aAc$$
$$A \to aA|b$$

and derivation:

$$S \Rightarrow aAc \Rightarrow aaAc \Rightarrow aabc$$

where S is a start nonterminal symbol; A is a nonterminal; a, b, and c are nonterminals.

- A bottom-up parser of this sentence, $aabc$, starts with the sentence and must find the handle (i.e., the correct RHS to reduce) in it.

Complexity of Parsing

- The time complexity of parsers that work for any unambiguous grammar are of $O(n^3)$ where n is the length of the input.
- Compilers use parsers that only work for a subset of all unambiguous grammars and do it in linear time, i.e., $O(n)$

Implementation of Parsers

- Top-down: Recursivee descent parsers
- Top-down: LL parsers
- Bottom-up: LR parsers

Recursive Descent Parsers

- A subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because the extensions in EBNF minimizes the number of nonterminals

Recursive Descent Parsers: Example

- Consider the following EBNF description of simple arithmetic expressions:

$$
\begin{aligned}
<\text{expr}> &\rightarrow <\text{term}> \{(+|-)<\text{term}> \} \\
<\text{term}> &\rightarrow <\text{factor}> \{(*|/)<\text{factor}> \} \\
<\text{factor}> &\rightarrow \text{ id} \mid \text{int\_constant} \mid (<\text{expr}>)
\end{aligned}
$$

Recursive Descent Parsers: Example

- Assume we have a lexical analyzer named `lex` that puts the next token code in `nextToken`
- *When a nonterminal has only one RHS*, the coding process:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an *error*
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive Descent Parsers: Example

- For the first rule,

$$<\text{expr}> \;\rightarrow <\text{term}> \{(+|-)<\text{term}> \}$$

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

Recursive Descent Parsers: Example

- For the second rule,

$$<\text{term}> \rightarrow <\text{factor}> \{(*|/)<\text{factor}> \}$$

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

Recursive Descent Parsers: Example

- A nonterminal that has *more than one RHS*, it requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive Descent Parsers: Example

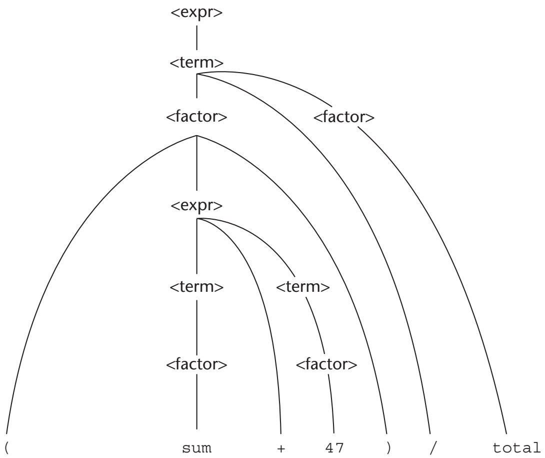
- For the third rule,

$$<\text{factor}> \rightarrow \text{id} \mid \text{int\_constant} \mid (<\text{expr}>)$$

```
void factor() {
    printf("Enter␣<factor>\n");
    /* Determine which RHS */
    if (nextToken == IDENT || nextToken == INT_LIT) {
        lex(); /* Get the next token */
    } else {
        /* If the RHS is (<expr>), call lex to pass over the
        left parenthesis, call expr, and check for the right
        parenthesis */
        if (nextToken == LEFT_PAREN) {
            lex(); expr();
            if (nextToken == RIGHT_PAREN) lex(); else error();
        } /* End of if (nextToken == ... */
        /* It was not an id, an integer literal, or a left parent
        else { error(); }
    } /* End of else */
    printf("Exit␣<factor>\n");;
} /* End of function factor */
```

Recursive Descent Parsers: Example

- The resulting parse tree



References I

Sebesta, R. W. (2012).
Concepts of Programming Languages.
Pearson, 10th edition.