ICOM 4036
Programming Languages
Lexical and Syntax Analysis

• Lexical Analysis
• The Parsing Problem
• Recursive-Descent Parsing
• Bottom-Up Parsing

This lecture covers review questions 1-13
Dr. Amirhossein Chinei, ECE, UPRM Spring 2010
***Some slides are adapted from the Sebesta’s textbook

Syntax Analysis

• The syntax analysis portion of a language processor nearly always consists of two parts:
  – A low-level part called a lexical analyzer (mathematically, a finite automaton based on a regular grammar)
  – A high-level part called a syntax analyzer, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Introduction

• Language implementation systems must analyze source code, regardless of the specific implementation approach
• Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

Advantages of Using BNF to Describe Syntax

• Provides a clear and concise syntax description
• The parser can be based directly on the BNF
• Parsers based on BNF are easy to maintain
Why 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 always is portable

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a “front-end” for the parser
- Identifies substrings of the source program that belong together - *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
  - E.g. `sum` is a lexeme; its token may be `IDENT`

Lexical Analysis (cont’d)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- **Three approaches to building a lexical analyzer:**
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given 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!
- 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 (cont’d)

- 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 (cont’d)

- 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, `lexeme`
  - `lookup` - determines whether the string in `lexeme` is a reserved word (returns a code)

Simple Lexical Analyzer

```c
/* lex - a simple lexical analyzer */

int lex() {
  char nextChar;
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT) {
        addChar();
        getChar();
      }
      return ID_LIT;
    break;
    } /* End of switch */
    case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
    break;
  } /* End of switch */
    } /* End of function lex */
```
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 (cont'd)

- Two categories of parsers
  - **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

The Parsing Problem (cont'd)

- **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

The Parsing Problem (cont'd)

- **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