Outline

- Recap
  - Intermediate code generation (Section 2.8)
  - Symbol tables (Section 2.7)
- Lexical analysis (Chapter 3)
- Summary and homework
Two Kinds of Intermediate Representation

- Syntax tree representation
  - Expressions
    - \( E_1 \ op \ E_2 \)
  - Statements
    - \textbf{while} (expr) stmt
    - \textbf{do} stmt \textbf{while} expr
  - Use a translation scheme
    - Semantic rules or \textit{semantic actions}

- Three-address-code representation

Instructor: Dr. Liang Cheng
CSE302: Compiler Design
02/01/07
Symbol Tables

- Hold info of source program constructs
  - Collected during analysis
  - Used for synthesis
- Support multiple declarations of the same identifier within a program
  - A separate symbol table for each scope
    - A program block
    - A class
Outline

- Recap
- Lexical analysis in a nutshell
  - Overview
  - Regular expressions
  - Finite automata
  - Implementation of a scanner
- Summary and homework
Overview

- Pattern matching between
  - Input: characters in a source file
  - Output: tokens

based on theories of regular expressions and finite automata

source program → **lexical analyzer** → **parser**

- get next token
- token

symbol table
Tokens and Lexemes

- A **lexeme** is the lowest level syntactic unit of a language described by a lexical specification.
- A **token** is a category/abstraction of lexemes.
Tokens

- Defined as an enumerated type
  - in C:
    
    ```
    typedef enum {
      IF, THEN, ELSE, EQ, GE, LE, NE, NUM, ID, ...
    } TokenType;
    ```
  - in Java:
    
    ```java
    Appendix A: Tag.java
    ```

- Fall into several categories
  - Reserved words
    - The lexeme or string value of the token IF is `if`
  - Special symbols
    - The lexeme or string value of the token EQ is `==`
  - Identifiers
    - Represent multiple lexemes
  - Literals or constants
Overview

- The scanner is operated under the control of the parser
  - In Parser.java: `move() {look=lex.scan();};`
  - In Lexer.java: `public Token scan() {...}`
Outline

- Recap
- Lexical analysis in a nutshell
  - Overview
  - Regular expressions
  - Finite automata
  - Implementation of a scanner
- Summary and homework
Regular Expressions

- Represent patterns of strings of characters
- The set of strings generated by a regular expression $r$ is as $L(r)$
Basic Regular Expressions

- Single characters from the alphabet
  - The set of legal symbols $\Sigma$
  - $L(a) = \{a\}$
  - $L(\epsilon) = \{\epsilon\}$
  - $L(\emptyset) = \{\}$

- Regular expression operations
  - Choice among alternatives: $L(r|s) = L(r) \cup L(s)$
  - Concatenation: $L(rs) = L(r)L(s)$
  - Repetition (zero or more times): $L(r^*) = L(r)^*$
  - A regular expression for a sequence of one or more numeric digits
    - $(0|1|\ldots|9)(0|1|\ldots|9)^*$
    - $digit\ digit^*$ where $digit = 0|1|\ldots|9$
Extensions to Regular Expressions

- One or more repetitions
  - \( r^+ \): digit+ where digit = 0|1|…|9

- A range of characters in the alphabet
  - a|b|c: [abc]
  - a|b|…|z: [a-z]
  - 0|1|…|9: [0-9]

- Any character in the alphabet, any character not in a given set …
Regular Expressions for Identifiers

- An identifier starts with a letter, followed by one or more letters or one or more digits
Outline

- Recap
- Lexical analysis in a nutshell
  - Overview
  - Regular expressions
  - Finite automata
  - Implementation of a scanner
- Summary and homework
A Finite Automaton for Identifiers

- There is an algorithm that constructs a finite automaton below for the regular expression of identifiers, e.g. Thompson’s construction.

- States in the pattern recognition process:
  - State 1: start state
  - State 2: the state after a single letter has been matched
    - Accepting states drawn in double-line border

Instructor: Dr. Liang Cheng
Outline

- Recap
- Lexical analysis in a nutshell
  - Overview
  - Regular expressions
  - Finite automata
  - Implementation of a scanner
- Summary and homework
Implementation of Finite Automata and Demo

- A transition table based approach
  - $s = 1$;
  
  ```
  while( $s!=$acceptState and $s!=$errorState) {
    $c =$ next input character;
    $s =$ $T[s,c];$
  }
  ```

<table>
<thead>
<tr>
<th>Characters in the alphabet $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>States representing transitions $T(s,c)$</td>
</tr>
<tr>
<td>States</td>
</tr>
<tr>
<td>$s$</td>
</tr>
</tbody>
</table>
Outline

- Recap
- Lexical analysis (Chapter 3)
- Summary and homework