CSE302: Compiler Design

Instructor: Dr. Liang Cheng
Department of Computer Science and Engineering
P.C. Rossin College of Engineering & Applied Science
Lehigh University

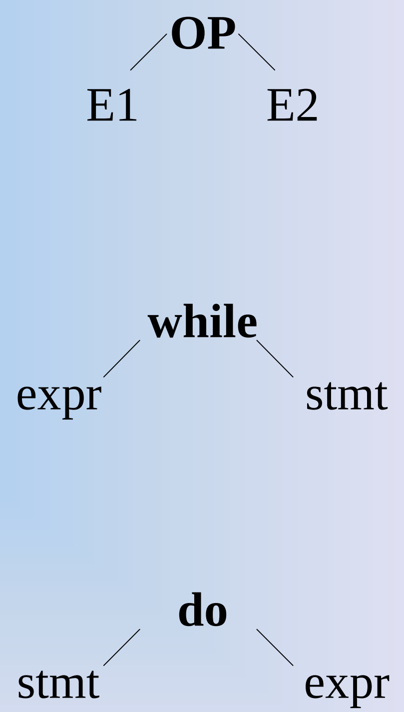
February 01, 2007

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
    - E1 **op** E2
  - Statements
    - **while** (expr) stmt
    - **do** stmt **while** expr
  - Use a translation scheme
    - Semantic rules or semantic actions
- Three-address-code representation



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

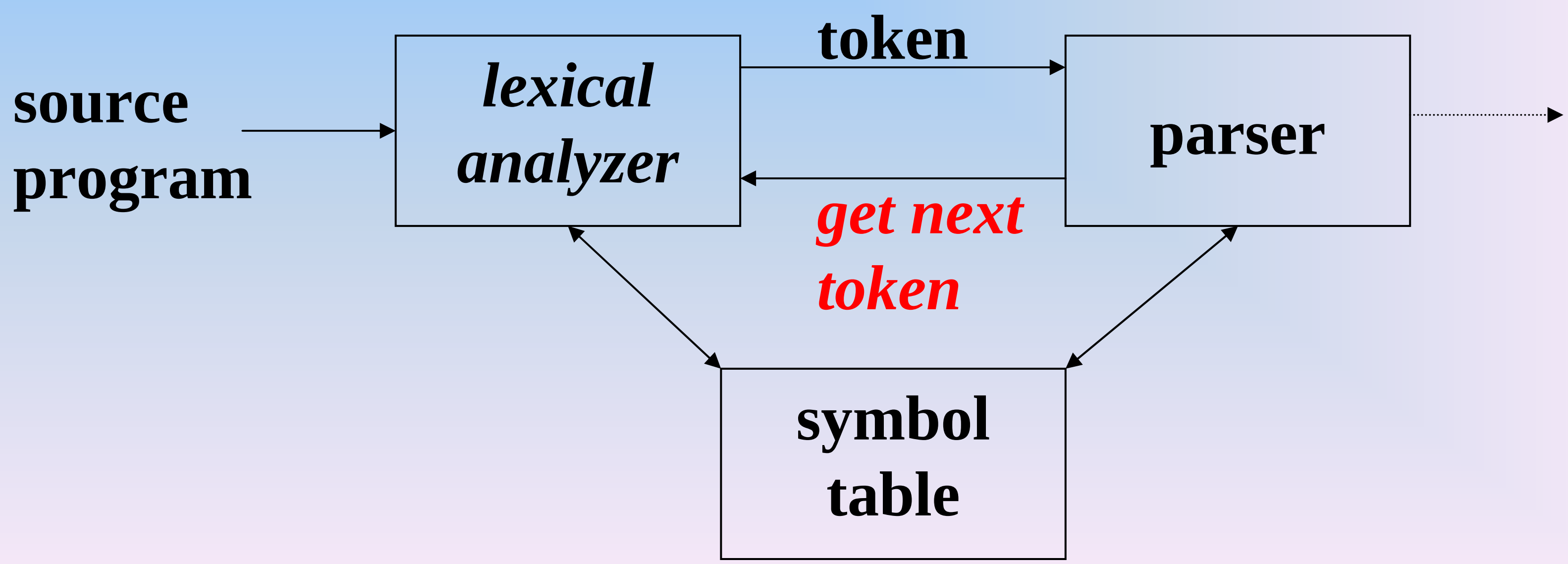
Outline

- 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



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:

    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 Σ
  - $L(\boldsymbol{a}) = \{a\}$
  - $L(\varepsilon) = \{\varepsilon\}$
  - $L(\emptyset) = \{\}$
- Regular expression operations
  - Choice among alternatives: $L(\boldsymbol{r}|\boldsymbol{s}) = L(\boldsymbol{r}) \cup L(\boldsymbol{s})$
  - Concatenation: $L(\boldsymbol{rs}) = L(\boldsymbol{r})L(\boldsymbol{s})$
  - Repetition (zero or more times): $L(\boldsymbol{r}^*) = L(\boldsymbol{r})^*$
  - A regular expression for a sequence of one or more numeric digits
    - (0|1|...|9)(0|1|...|9)*
    - *digit digit** where *digit* = 0|1|...|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

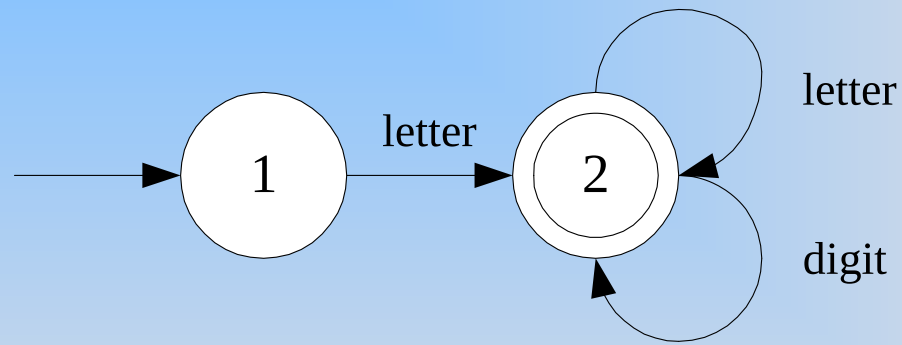
- An identifier starts with a letter, followed by one or more letters or one or more digits

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

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*];

    }

	Characters in the alphabet *c*
States *s*	States representing transitions T(*s*,*c*)

Outline

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