Lexical and Syntax Analysis

Lecture 07

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- Three different approaches to implementing programming languages are introduced
  - *Compilation*
  - *Pure interpretation*
  - *Hybrid implementation*

Introduction

- The ***compilation*** approach uses a program called a ***compiler***
  - Translates programs written in a high-level programming language into machine code

- Compilation is typically used to implement programming languages that are used for large applications
  - For example, languages such as C++ and COBOL

Introduction

- ***Pure interpretation*** systems perform no translation; rather, programs are interpreted in their original form by a software ***interpreter***
  - Pure interpretation is usually used for smaller systems in which execution efficiency is not critical
    - For example, scripts embedded in HTML documents, written in languages such as JavaScript

Introduction

- ***Hybrid implementation*** systems translate programs written in high-level languages into intermediate forms, which are interpreted
  - These systems are now more widely used than ever
  - Traditionally, hybrid systems have resulted in much slower program execution than compiler systems
  - However, in recent years the use of Just-in-Time (JIT) compilers has become widespread, particularly for Java programs and programs written for the Microsoft .NET system
  - A JIT compiler, which translates intermediate code to machine code, is used on methods at the time they are first called
  - In effect, a JIT compiler transforms a hybrid system to a delayed compiler system

Introduction

- All three of the implementation approaches just discussed use both ***lexical*** and ***syntax analyzers***
- Syntax analyzers, or parsers, are nearly always based on a formal description of the syntax of programs. The most commonly used syntax-description formalism is context-free grammars, or BNF
- Using BNF, as opposed to using some informal syntax description, has at least three compelling advantages
  - First, BNF descriptions of the syntax of programs are clear and concise, both for humans and for software systems that use them
  - Second, the BNF description can be used as the direct basis for the syntax analyzer
  - Third, implementations based on BNF are relatively easy to maintain because of their modularity

Introduction

- Nearly all compilers separate the task of analyzing syntax into two distinct parts
  - *Lexical analysis*
  - *Syntax analysis*

- The *lexical analyzer* deals with small-scale language constructs, such as names and numeric literals

- The *syntax analyzer* deals with the large-scale constructs, such as expressions, statements, and program units

Introduction

- There are three reasons why lexical analysis is separated from syntax analysis
  - ***Simplicity***
    - Techniques for lexical analysis are less complex than those required for syntax analysis, so the lexical-analysis process can be simpler if it is separate
    - Also, removing the low-level details of lexical analysis from the syntax analyzer makes the syntax analyzer both smaller and less complex

Introduction

- There are three reasons why lexical analysis is separated from syntax analysis
  - ***Efficiency***
    - Although it pays to optimize the lexical analyzer, because lexical analysis requires a significant portion of total compilation time, it is not fruitful to optimize the syntax analyzer.
    - Separation facilitates this selective optimization.

Introduction

- There are three reasons why lexical analysis is separated from syntax analysis
  - ***Portability***
    - Because the lexical analyzer reads input program files and often includes buffering of that input, it is somewhat platform dependent.
    - However, the syntax analyzer can be platform independent.
    - It is always good to isolate machine-dependent parts of any software system.

Lexical Analysis

- A ***lexical analyzer*** is essentially a ***pattern matcher***
- A *pattern matcher* attempts to find a substring of a given string of characters that matches a given *character pattern*
- Pattern matching is a traditional part of computing
- One of the earliest uses of pattern matching was with text editors, such as the ***ed*** line editor, which was introduced in an early version of UNIX
- Since then, pattern matching has found its way into some programming languages
  - For example, Perl and JavaScript
- It is also available through the standard class libraries of Java, C++, and C#

Lexical Analysis

- A lexical analyzer serves as the front end of a syntax analyzer
  - Technically, lexical analysis is a part of syntax analysis
- A lexical analyzer performs syntax analysis at the lowest level of program structure
  - An input program appears to a compiler as a single string of characters
  - The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure
    - These logical groupings are named ***lexemes***
    - The internal codes for categories of these groupings are named ***tokens***
- Lexemes are recognized by matching the input character string against character string patterns

Lexical Analysis

- Consider the following example of an assignment statement

```
result = oldsum - value / 100;
```

- Following are the tokens and lexemes of this statement

Token	*Lexeme*
IDENT	result
ASSIGN_OP	=
IDENT	oldsum
SUB_OP	-
IDENT	value
DIV_OP	/
INT_LIT	100
SEMICOLON	;

Lexical Analysis

- *Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens*
  - In the early days of compilers, lexical analyzers often processed an entire source program file and produced a file of tokens and lexemes
- Now, however, most lexical analyzers are subprograms that locate the next lexeme in the input, determine its associated token code, and return them to the caller, which is the syntax analyzer
  - Each call to the lexical analyzer returns a single lexeme and its token
  - The only view of the input program seen by the syntax analyzer is the output of the lexical analyzer, one token at a time

Lexical Analysis

- The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program

- Also, the lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of the compiler

- Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user

Lexical Analysis

- A state transition diagram, or just ***state diagram***, is a directed graph
  - The nodes of a state diagram are labeled with state names
  - The arcs are labeled with the input characters that cause the transitions among the states
    - An arc may also include actions the lexical analyzer must perform when the transition is taken
- State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called ***finite automata***
  - Finite automata can be designed to recognize members of a class of languages called regular languages

Lexical Analysis

- We now illustrate lexical-analyzer construction with a state diagram and the code that implements it
- The state diagram could simply include states and transitions for each and every token pattern
- However, that approach results in a very large and complex diagram, because every node in the state diagram would need a transition for every character in the character set of the language being analyzed
- We therefore consider ways to simplify it.

Lexical Analysis

- Suppose we need a lexical analyzer that recognizes only arithmetic expressions, including variable names and integer literals as operands
  - Assume that the variable names consist of strings of uppercase letters, lowercase letters, and digits but must begin with a letter
  - Names have no length limitation

Lexical Analysis

- The first thing to observe is that there are 52 different characters (any uppercase or lowercase letter) that can begin a name, which would require 52 transitions from the transition diagram's initial state
  - However, a lexical analyzer is interested only in determining that it is a name and is not concerned with which specific name it happens to be
  - Therefore, we define a character class named LETTER for all 52 letters and use a single transition on the first letter of any name

Lexical Analysis

- Another opportunity for simplifying the transition diagram is with the integer literal tokens
  - There are 10 different characters that could begin an integer literal lexeme
    - This would require 10 transitions from the start state of the state diagram
  - Because specific digits are not a concern of the lexical analyzer, we can build a much more compact state diagram if we define a character class named DIGIT for digits and use a single transition on any character in this character class to a state that collects integer literals
  - Because our names can include digits, the transition from the node following the first character of a name can use a single transition on LETTER or DIGIT to continue collecting the characters of a name.

Lexical Analysis

- Next, we define some utility subprograms for the common tasks inside the lexical analyzer
  - First, we need a subprogram, which we can name getChar, that has several duties
    - When called, getChar gets the next character of input from the input program and puts it in the global variable nextChar
    - getChar must also determine the character class of the input character and put it in the global variable charClass
    - The lexeme being built by the lexical analyzer, which could be implemented as a character string or an array, will be named lexeme

Lexical Analysis

- We implement the process of putting the character in nextChar into the string array lexeme in a subprogram named addChar
  - This subprogram must be explicitly called because programs include some characters that need not be put in lexeme, for example the white-space characters between lexemes
  - In a more realistic lexical analyzer, comments also would not be placed in lexeme

Lexical Analysis

- When the lexical analyzer is called, it is convenient if the next character of input is the first character of the next lexeme
- Because of this, a function named getNonBlank is used to skip white space every time the analyzer is called

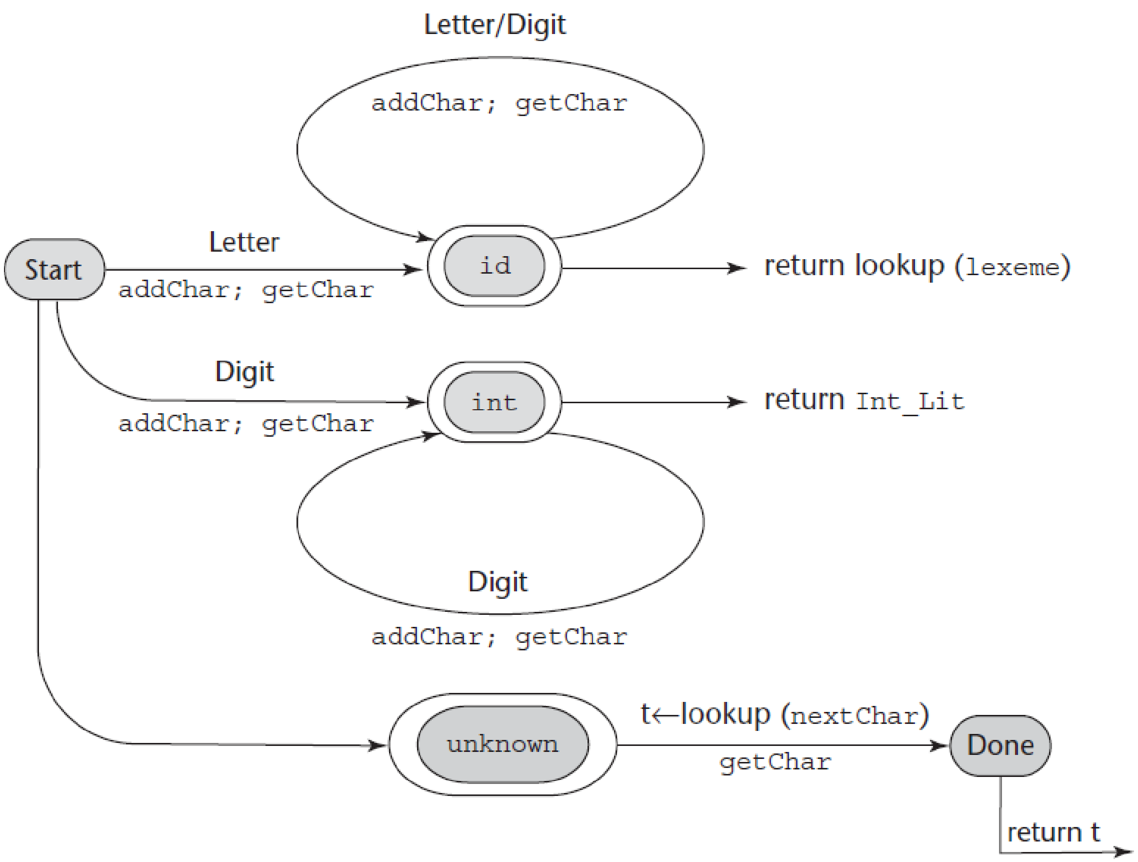
Lexical Analysis

- Finally, a subprogram named lookup is needed to compute the token code for the single-character tokens
  - In our example, these are parentheses and the arithmetic operators

- Token codes are numbers arbitrarily assigned to tokens by the compiler writer

Lexical Analysis

- The state diagram describes the patterns for our tokens



Lexical Analysis

- The following is a C implementation of a lexical analyzer specified in the state diagram, including a main driver function for testing purposes
  - **front.c**
- Consider the following expression:

```
(sum + 47) / total
```

- Following is the output of the lexical analyzer of front.c when used on this expression

```
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
```

MARSHALL