Programming Languages

Lexical Analysis

Introduction

When implementing a programming language, the first step is reading in the source code of a program written in that language. Typically, the source code is stored as a file of characters. To read in a source code file, one groups the important individual characters into tokens and discards the unimportant characters. For example, consider the Python program:

```
print 'Hello World!'
```

There are two tokens in this program, `print` and `'Hello World!'`. The unimportant characters are the space that follows the token `print` and the newline that follows the token `'Hello World'`. Note that the space within the token `'Hello World!'` *is* important, so the subsystem for reading in source code must be smart enough to distinguish between important and unimportant spaces, among other things. This subsystem is called *lexical analysis*.

Lexical Analysis

Lexical analysis is the process of identifying tokens in a file of characters. The analyzer is also tasked with categorizing those tokens as to the kind of thing they are. This *kind of thing* is known as a *type*. You may think of a token as a word and a type as a part of speech. For example, the English sentence:

```
I like yellow peaches.
```

is composed of five tokens, `I` (pronoun), `like` (verb), `yellow` (adjective), `peaches` (noun), and `.` (punctuation). The tokens have been annotated with their types (parts of speech). Note that the spaces and the newline, while important for separating the characters of adjacent tokens, are not saved when the tokens are extracted from the sentence.

The same process that you use in identifying tokens in English prose is used for identifying token - type pairs in source code. A token - type pair is known as a *lexeme* lexical analysis is the process of converting characters into lexemes. lexical analyzer maps a stream of characters onto a stream of lexemes.

A lexeme data structure

At a minimum, a lexeme is an object that holds at least two pieces of information, the type of the token and, in some cases, the actual token itself. As will be seen further on, a lexeme representing a semicolon would generally only use the type field, since the actual word `";"` can easily be reconstructed from the type information. A data structure for holding the parts of a lexeme might look like:

```
class lexeme implements types
    {
    String type;
    String word;
    }
```

With respect to the lexemes found in a programming language, it may be efficient to store some of the words as other than strings. For example, numbers may best be stored as native numbers rather than as strings. To reflect this idea, we add a field corresponding to each of the primitive types in the language we are to implement:

```
class lexeme extends types
    {
    String type;
    String string;
    int integer;
    double real;
    }
```

Since memory is relatively cheap, we will not worry about the fact that for any given lexeme, two of the three value fields will remain empty .

As an example of lexical analysis with respect to computer languages, consider some simple expressions in Java:

```
int a = 2;
double zeta;

if (a != 0)
    a = 64.0 / a;
zeta = log(a*a,2);

System.out.println("the value of zeta is " + zeta);
```

The parts of speech in this language are *keywords*, *variables*, *operators*, *numbers*, and *punctuation* (in order of appearance). Now consider a program for taking this source code and producing the lexical stream. The function that reads a portion of the input and produces the next lexeme has historically been called *lex*.

The *lex* function

How does the *lex* function decide what are the tokens and what type a particular token has? That depends on the language being implemented. For example, an *integer* in the language might be described by this simple rule:

> *an integer is either a minus sign followed by one or more digits or just a digit followed by any number of digits*

A *variable*, on the other hand, has a different rule:

> *a variable is token whose first character is a letter and whose subsequent characters, if any, are letters or digits*

Often, in a language, a keyword fits the variable rule but it is not, of course, a variable. Therefore, rules for identifying keywords must be checked before checking the variable rule. For example, the keyword *if* has the following rule:

> *the* `if` *keyword is an* `'i'` *followed by an* `'f'`

Putting these ideas together, it becomes rather simple to write the lex function:

```
function lex()
    {
    char ch;

   skipWhiteSpace();  //why do we skip whitespace?

    ch = Input.read();

    if (Input.failed) return new lexeme(ENDofINPUT);

    switch(ch)
        {
        // single character tokens

        case '(': return new lexeme(OPAREN);
        case ')': return new lexeme(CPAREN);
        case ',': return new lexeme(COMMA);
        case '+': return new lexeme(PLUS); //what about ++ and += ?
        case '*': return new lexemeTIMES);
        case '-': return new lexeme(MINUS);
        case '/': return new lexeme(DIVIDES);
        case '<': return new lexeme(LESSTHAN);
        case '>': return new lexeme(GREATERTHAN);
        case '=': return new lexeme(ASSIGN);
        case ';': return new lexeme(SEMICOLON);

        //add any other cases here

        default:
            // multi-character tokens (only numbers,
            // variables/keywords, and strings)
            if (Character . isDigit(ch))
                {
                Input.pushback(ch);
                return lexNumber();
                }
            else if (Character . isLetter(ch))
                {
                Input.pushback(ch);
                return lexVariableOrKeyword();
                }
            else if (ch == '\"')
                {
                return lexString();
                }
            else
                return new Lexeme(UNKNOWN, ch);
        }
    }
```

Specific lexing functions

In this (partial) implementation of *lex*, you can see a number of helper functions: *read* and *pushback*, *skipWhitespace*, *lexVariableOrKeyword*, *lexNumber*, and *lexString*.

The *read* function reads a single character from the input stream, while the *pushback* function puts a previously read character back on the input stream to be read again. If your implementation language does not support pushback, you can easily implement your own pushback system with two global variables:

```
Char PushbackCharater;
Boolean CharacterHasBeenPushed = FALSE;

function myRead()
    {
    if (CharacterHasBeenPushed)
        {
        CharacterHasBeenPushed = FALSE;
        return PushbackCharacter;
        }
    else
        return Input.read();
    }

function myPushback(ch)
    {
    if (CharacterHasBeenPushed) Fatal("too many pushbacks");
    CharacterHasBeenPushed = TRUE;
    PushbackCharacter = ch;
    }
```

This implementation only allows one character of pushback; to push back multiple characters, use a stack instead. Most lexical analyzers need only one character of pushback; the *skipWhitespace* function illustrates its use:

```
function skipWhiteSpace()
    {
    var chl;
    while (isWhiteSpace(ch))
        ch = Input.read();

    // the character that got us out of the loop was NOT whitespace
    // so we need to push it back so it can be read again.

    Input.pushback(ch);
    }
```

Most modern programming languages are either completely or mostly *free format.* Free format means that, in most cases, you can place as many spaces, tabs, and newlines as you wish between the tokens making up the source code of a program. Collectively, spaces, tabs, and newlines are known as *whitespace.* Source code comments are another kind of whitespace; comments are usually dealt with inside the whitespace while loop. We use the function *skipWhiteSpace* to get past the arbitrary amounts of whitespace, so that the next character read from the input stream is the start of the next token in the source code.

The function *lexVariableOrKeyword* is charged with reading the complete variable (or keyword) token and then distinguishing between the two:

```
function lexVariableOrKeyword()
    {
    var ch;
    String token = "";

    ch = Input.read();
    while (isLetter(ch) || isDigit(ch))
        {
        token = token + ch; //grow the token string
        ch = Input.read();
        }

    //push back the character that got us out of the loop
    //it may be some kind of punctuation

    Input.pushback(ch);

    //token holds either a variable or a keyword, so figure it out

    if (stringEquals(token,"if")) return new Lexeme(IF);
    else if (stringEquals(token,"else")) return new Lexeme(ELSE);
    else if (stringEquals(token,"while")) return new Lexeme(WHILE);
    ... //more keyword testing here
    else //must be a variable!
        return new Lexeme(VARIABLE,token);
    }
```

The implementations of *lexNumber* and *lexString* are similar to *lexVariable*.

The tokens `ENDofINPUT`, `PLUS`, `MINUS`, `WHILE` etc. are String constants, defined in the class/module *types*, and are used to fill out the type component of a lexeme. Note that when input is exhausted, *lex* repeatedly returns the `ENDofINPUT` lexeme.

Lexically scanning a file

When given the input:

```
int alpha = 10;

while (alpha > 0)
      {
      System.out.println("alpha is " + alpha);
      alpha = alpha - 1;
      }
```

repeated calls to *lex* might produce the following stream of lexemes:

```
INTEGER_TYPE
VARIABLE alpha
ASSIGN
INTEGER 10
SEMICOLON
WHILE
OPEN_PAREN
VARIABLE alpha
GREATER_THAN
INTEGER 0
CLOSE_PAREN
OPEN_BRACE
VARIABLE System
```

```
DOT
VARIABLE out
DOT
VARIABLE println
OPEN_PAREN
STRING "alpha is "
PLUS
VARIABLE alpha
CLOSE_PAREN
SEMICOLON
VARIABLE alpha
ASSIGN
VARIABLE alpha
MINUS
INTEGER 1
SEMICOLON
CLOSE_BRACE
ENDofINPUT
```

A program which repeatedly calls *lex* and displays the resulting lexemes is called a *scanner*:

```
function scanner(filename)
    {
    var token;
    var i = new lexer(fileName);

    token = i.lex();
    while (token.type != ENDofINPUT)
        {
        Lexeme.display(token);
        token = i.lex();
        }
    }
```

Lexical analysis is the very first phase of implementing a programming language. The next phase, called *syntactic analysis* or *recognition*, ensures that the order of the lexemes makes logical sense.