Programming Project 2: Lexical Analyzer

Due Date: Tuesday, October 18, 2005, Noon

Overview

Create a lexical analyzer for the PCAT language. The lexical structure is described in Section 2 of the PCAT Programming Language Reference Manual. Your program should be written in Java.

The lexer will consume source text and produce tokens (with attributes where appropriate), one token per call to the **getToken()** method. It will flag lexical errors by printing error messages on the standard error output (stderr). Any portion of the input text that cannot be converted to a legal token stream should trigger an appropriate informative error message to standard error. The lexer should then proceed, ultimately returning a valid token to its caller (e.g., a parser).

Your goal is to create a file called **Lexer.java** containing a class called **Lexer**. This class should have a method

```
int getToken ()
```

This class must have at least the following fields:

```
int lineNumber;     // The current line number
int iValue;         // Attributes associated with the
double rValue;      //   most recent token
String sValue;      //
```

The **Lexer** class should have a constructor:

```
Lexer (Reader rdr)
```

I will provide the following files:

```
Main.java
Token.java
StringTable.java
FatalError.java
LogicError.java
```

Your class will be compiled with these files to produce an executable. The **Main.java** file contains a **main** method which will be used to test your lexer. The **main** method will create a **Lexer** object, invoke **getToken** repeatedly, and print the tokens that the lexer returns on the standard output.

Files

The following useful files can be found via the class web page or ftp'ed from:

```
~harry/public_html/compilers/p2
```

Main.java
Token.java
StringTable.java
FatalError.java
LogicError.java
The framework in which your code will be used. Do not alter these classes!!! Your program will be tested using these files.

makefile
If you are familiar with the Unix "make" command, you can use this file to compile your code. If you are working in a different environment, you may not want this file.

tst
This is a subdirectory containing a number of test files. The tests are named, with names like **test1**, **test2**, **test3**, ... For each test, there are 3 files in the directory, such as:

```
test1.pcat
test1.out.bak
test1.err.bak
```

The *XXX***.pcat** file is a sample input file for your program. The *XXX***.out.bak** file is the desired output to **stdout**. The *XXX***.err.bak**. is the desired output to **stderr**.

go
This is a shell script that will run your program on a test file and print the input file, **stdout**, and **stderr**. For example, you can type something like

```
go test1
```

When your program is not working correctly, you can use this to see exactly what output your program is producing.

run
This is shell script that will run your program on a test file, compare the results to the desired results, and print only the differences. For example, you can type something like

```
run test1
```

If your program is working correctly on this test, you'll see no output. Otherwise, you'll see the differences.

runAll
This is shell script that will run your program on all the tests in the **tst** directory. It calls **run** once for each test. To test your program, type

```
runAll
```

If your program is working correctly, you'll see only a list of the tests that were run.

Main.jar
This is a "black box" solution. This file contains a working **Lexer** class. You can execute it with a command such as:

```
java -classpath Main.jar Main tst/test1.pcat
```

This is the code that was used to produce the desired output files in the **tst** directory.

You must not modify any of these files. Look through these files and make sure you understand them.

Token Definitions

The PCAT Language Reference Manual contains a detailed specification of the tokens.

Take a look a the file **Token.java**. It contains a list of all the token types:

```
final static int
    AND           = 0,
    ARRAY         = 1,
    BEGIN         = 2,
    ...
    ID            = 29,
    INTEGER       = 30,
    REAL          = 31,
    STRING        = 32,
    ...
    PLUS          = 33,   // +
    MINUS         = 34,   // -
    ...
    RBRACE        = 49,   // }
    ASSIGN        = 50,   // :=
    LEQ           = 51,   // <=
    ...
    EOF           = 54;
```

Your **getToken** method must return an integer indicating the type of token just scanned. Your code must return one of these codes. For example, after scanning a REAL token, you might execute:

```
return Token.REAL;
```

For tokens with an associated value, your lexer must also store an associated attribute value before returning. Toward this end, your lexer should include 3 fields:

```
int iValue;
double rValue;
String sValue;
```

In addition, your lexer should include a field which you will set to the line number of the last token returned:

```
int lineNumber;
```

If an INTEGER token is scanned, your code should set the **iValue** field to the integer value of the lexeme. If a REAL token is scanned, your code should set **rValue**. If an ID or a STRING is scanned, your code should set **sValue**.

The code using your lexer will access these fields. For example, the parser might contain code like this:

```
Lexer lex = new Lexer (...);
...
nextTok = lex.getToken ();
...
if (nextTok == Token.REAL) {
    d = lex.rValue;
} else {
    System.out.println ("Syntax Error on line " + lex.lineNumber);
...
```

For ID tokens, there is an additional constraint on the **sValue** attribute: If the same identifier appears in several places in the source input, the lexer must return a pointer to the same String object each time. Later, during semantic analysis, we will often need to compare two IDs to see if they are the same ID. At that time, we want to be able to simply compare pointers without having to look at each of the individual characters in the Strings.

The **main** method and one of the tests ("test5") are set up to check that, for a particular identifier, the **sValue** will always point to the same String object.

To assist you with this, I have provided the **StringTable.java** file. That code can be used to initialize, query, and update a "string" table. It can also be used to facilitate a quick lookup to see if some sequence of letters happens to be one of the reserved keywords and, if so, which token type to use.

STRING tokens should not be entered into the String Table since it is more trouble than it is worth. All string constants in a program will generally be unique and we won't need to compare them later anyway.

REAL tokens may be scanned character-by-character to create a string. This string can then be converted to a double using something like:

```
... = Double.parseDouble (str);
```

If the string specifies a value that is too large for a double, then the result of this conversion will be "positive infinity", which will be returned from **parseDouble**. Your lexer should print an error message and set **rValue** to 0.0.

Handling Errors

When a lexical error occurs, your lexer should print a message on **stderr** and keep going until a legal token can be returned. The **Main** class contains a static variable:

```
static int errorCount = 0;
```

which you should increment every time a lexical error occurs.

Error messages that your program should print out are shown by example:

```
Error on line 1: End-of-file encountered within a comment
Error on line 2: End-of-line encountered within a string
Error on line 3: End-of-file encountered within a string
Error on line 4: Illegal character in source ignored
Error on line 5: Illegal character in string ignored
Error on line 6: Integer out of range (0..2147483647)
Error on line 7: Maximum identifier length (255) exceeded
Error on line 8: Maximum string length (255) exceeded
Error on line 9: Real number is too large
```

I am supplying two classes called **FatalError** and **LogicError**. These will be used throughout the compiler.

FatalError is similar to **CompileTimeError** in project 1. The idea is that if something really bad goes wrong during compilation, the code will throw a **FatalError**, which is caught in the **main** method. If a **FatalError** is ever thrown, the compiler will print an error message and terminate immediately. In your lexer, it is possible that an **IOException** will occur whenever you try to read from the input. If an **IOException** occurs, a reasonable thing is to throw a **FatalError**. Here is how your **getToken** might look:

```
int getToken ()
   throws FatalError
{
   ...
   try {
         ... lots of code, including calls to "read" a character ...
   } catch (IOException exn) {
         throw new FatalError ("I/O error: " + exn.getMessage() + " at line " + lineNumber);
   }
}
```

Your code should throw **LogicError** whenever some program logic error or inconsistency is detected. When **main** catches **LogicError**, it will print the message and also print information about what methods are executing at the time of the error. This might be helpful in debugging. For example, your code might contain code like this:

```
if (x == 1) {
   ...
} else if (x == 2) {
   ...
} else if (x == 3) {
   ...
} else {
   throw new LogicError ("Unexpected value for x");
}
```

(My own lexer code doesn't happen to use **LogicError** and you don't need to use it in your lexer if it doesn't make sense. I am only supplying **LogicError** now since it will be useful in later projects.)

Requirements

Your code should deal with its input in exactly the same way as my code; this applies both to handling correct PCAT source programs as well as to printing error messages for incorrect PCAT source programs. Your programs will be tested mechanically using test data generated by my code. If there is any question about the exact functionality required...

(1) Use my code (the "black box" **.jar** file) on test files of your own creation, to see how it performs.

(2) *Please* let me know so I can modify my documents and alert other students. If my test data fails to catch some bug in your program or you feel the testing could be improved, please let me know so I can add to my test files for next year's course.

Program Submission

Place your code in a single file named "Lexer.java" and email it to:

```
cs321-01@cs.pdx.edu
```

Your file should be a *plain text* attachment; the contents of the email itself will be ignored. The subject line of the email must say:

```
Proj 2 - Xxxxxx Yyyyyy
```

where "Xxxxxx Yyyyyy" is your name. Please use your full name as it appears in the PSU registration system. *Don't forget your name!*

Do not submit your program twice without prior approval. If a second submission has been approved, use a subject line such as:

```
Proj 2 - Xxxxxx Yyyyyyy - Second submission
```

Your program is due at noon. Please email yourself a copy of the program and keep it for a while. Also, please keep a copy of your file on Sirius as you submitted it; do not modify this file. If any issues arise, we can also look at the timestamp on this file.

Do Not Work Together

Do not work together on any programming project in this class. Do not look at anyone else's code. Do not allow anyone to use your code. Every line of code you submit must be your own work (except of course the code we distribute to the class). You may discuss Java and the assignments with other students, but only in general terms. You may not look at someone else's code or share Java code.

You are free to look at the lexer code I distributed for the previous project and use portions of that wherever convenient.

Questions / Comments

Email questions to:

`harry@cs.pdx.edu` ← *Questions for instructor*

Please include “cs321” in the subject line.