Prof. Jingke Li (FAB120-06, lij@pdx.edu), Mon 8:15-10:05 @FAB 171, Lab: Wed 8:15-10:05 @FAB 88-10

Homework Assignment 1: Lexer

(Due 1/28/15 @ 11:59pm)

In this assignment, you are going to implement a lexer for a small Java-like language, called "miniJava." This assignment builds on top of Lab 2. If you have not attended the lab for any reason, you should go through its materials first. The assignment carries a total of 100 points.

Preparation

Download the zip file `"hw1.zip"` from D2L. After unzipping, you should see a `hw1` directory with the following items:

- `hw1.pdf` — this document
- `mjTokens.pdf` — miniJava's token specification
- `mjTokenConstants.java` — token code definition for the lexer
- `mjLexer0.java` — a starting version of the lexer
- `Makefile` — for compiling the lexer (*usage:* `make lexer`)
- `run` — a script for running the tests (*usage:* `./run tst/*.in`)
- `tst/` — a subdirectory containing a set of test inputs

The Lexer Program

Your main task is to implement a lexer in Java to recognize all the tokens specified in the file `mjTokens.pdf`.

Name your lexer program `mjLexer.java`. You are free to organize the program in any way you see fit, but it needs to satisfy the following requirements:

1. Use the provided token code definition in `mjTokenConstants.java`. An easy way to do this is to use the following top-level class declaration for your program:

```
public class mjLexer implements mjTokenConstants {
  ...
}
```

2. Use the following `Token` class to represent tokens:

```
  static class Token {
    int code;            // token code
    String lexeme;       // lexeme string
    int line;            // line number of token's first char
    int column;          // column number of token's first char
    ...
  }
```

3. Define a `nextToken()` method; have it retun a new token in the form of a `Token` object each time it is called:

   ```
       public static Token nextToken() throws Exception {
         ...
       }
   ```

   Note: The above code snippets can be found in the starter program `mjLexer0.java`.

4. Display tokens in the following format:

   - One token per line.
   - Each line starts with a pair `(linNum,colNum)`, where the two numbers are the starting line and column numbers of the token.
   - Then,
     . for an `ID` token, display `ID(lexeme);`

     . for an `INTLIT` token, display `INTLIT(literal's value);`
     Note: Regardless of the literal's base, its value is to be displayed, *e.g.* for input `23`, the display should be `INTLIT(23)`, while for input `023`, the display should be `INTLIT(19)`.

     . for an `DBLLIT` token, display `DBLLIT(literal's value);`

     . for an `STRLIT` token, display `STRLIT(lexeme);`

     . for any other token, display just `lexeme`.
   - After all tokens are displayed, show one last line: `Total: tknCnt tokens`.

   Exact spacing among items within a line is not required.

   Here is an example:

   Input:

   ```
   int i = 23 + 023 - 0x10;
   "Hello World!"
   The end.
   ```

   Output:

   ```
   (1,1)   int
   (1,5)   ID(i)
   (1,7)   =
   (1,9)   INTLIT(23)
   (1,12)  +
   (1,14)  INTLIT(19)
   (1,18)  -
   (1,20)  INTLIT(16)
   (1,24)  ;
   (2,1)   STRLIT("Hello World!")
   (3,1)   ID(The)
   (3,5)   ID(end)
   (3,8)   .
   Total: 13 tokens
   ```

Hints Java has a set of utilities for converting strings to different types of numerical values:

```
Integer.parseInt(String s);
Integer.parseInt(String s, int base);
Double.parseDouble(String s);
```

You may use them to convert `INTLIT` and `DBLLIT` lexemes to their corresponding numerical values.

General Advice

One general advice for writing the lexer program is to take an incremental approach. Start small and work on one category of tokens at a time. Test your program before moving on to the next category. For integer literals, you may want to further subdivide, *i.e.* work on decimals first, then octals, then hexadecimals.

The Test Inputs

A set of test inputs, `test*.in`, are provided in the `tst` subdirectory. They cover the following token cases:

- operators and delimiters (`test01.in`)
- reservered words (`test02.in`)
- identifiers (`test03.in`)
- string literals (`test04.in`)
- decimal integer literals (`test05.in`)
- octal integer literals (`test06.in`)
- hexadecimal integer literals (`test07.in`)
- floating-point literals (`test08.in`)
- single-line comment (`test00.in`)
- multi-line comment (`test09.in`)

For these inputs, your lexer should generate outputs that match those in the `.ref` files. Note that these inputs do not provide a complete coverage of all possible token cases. Your program will be graded on additional tests. You are encouraged to create more tests of your own to test your program.

Your second task is to develop six new test inputs for testing the following six types of lexical errors:

- illegal characters
- integer overflow
- ill-formed octal integers
- ill-formed hexadecimal integers
- unterminated strings
- unterminated multi-line comments

For each of these inputs, your lexer should display a message indicating the error type, and the line and column numbers where the error occurred. Name your new test files `error01.in` — `error06.in`.

Hints For detecting integer overflow and ill-formations, you may use the same Java conversion utility routines mentioned before. Those routines will throw an exception if the string represents an overflown or ill-formed integer. You may catch those exceptions and repackage them as our own lexical errors.

Grading Metric

- Correctly recognize tokens (60 points)
- Correctly detect lexical errors (20 points)
- Follow program requirements described earlier (10 points)
- Develop six valid error-testing inputs (10 points)

Minimum Requirement The minimum requirement for passing this assignment is that your lexer program compiles on the CS Linux system without error and correctly recognizes at least category of tokens.

Submission

Submit a single zip file, `hw1sol.zip`, containing your lexer program `mjLexer.java` and the six new error tests `error0[1-6].in` through the "Dropbox" on the D2L class website. You don't need to encode your name in the zip file name; D2L automatically does that. *But don't forget to include your name in your program files.*