CS 321 Homework 2 – due 4:30pm, Wednesday, October 27, 2010

Lexical Analysis

Write a lexical analyzer for the full **fab** language. The lexical structure is described in Section 2 of the **fab** *Programming Language Reference Manual*.

Your lexical analyzer must be implemented as a class of the following form:

```
class Yylex {
  Yylex(java.io.InputStream s) { ... };
  public Symbol yylex() throws java.io.IOException,
                               ParseError { ... };
}
```

The lexical analyzer will consume source text from the specified `InputStream` and produce tokens, with their associated line numbers, and attributes where appropriate. Each call to `yylex()` returns one token; tokens are returned as instances of the `Symbol` class. If any portion of the input text cannot be converted to a legal token, `yylex()` should throw an instance of `ParseError` with an appropriate informative error message.

Definitions of `Symbol` and `ParseError` are available from the course web page. `Symbol` includes an enumeration of the possible token codes, and provides a function to convert them to printable names. Token names are as follows:

- `ID` for identifiers.
- `INTEGER` for integers.
- `REAL` for reals.
- `STRING` for strings.
- Keywords token names are simply the keywords themselves, but converted to upper-case.
- Operator and delimiter token names are according to the following table:

AT	@	LT	<	LPAREN	(
ARROW	->	LEQ	<=	RPAREN	)
ASGN	:=	GT	>	LSQBRA	[
PLUS	+	GEQ	>=	RSQBRA	]
MINUS	-	EQ	=	LCUBRA	{
TIMES	*	NEQ	<>	RCUBRA	}
SLASH	/	COLON	:		
		SEMI	;		
		COMMA	,		
		DOT	.		

The following token types have an associated attribute object: `ID` (a `String` containing the identifier), `INTEGER` (an `Integer` containing the integer value), `REAL` (a `String` containing the lexeme matching the real pattern), and `STRING` (a `String` containing the string, without its enclosing double-quotes (`""`)).

A suitable driver for the analyzer, called `LexerDriver`, is also provided on the course web page. It has the following definition:

```
class LexerDriver {
  public static void main(String argv[])
                          throws java.io.IOException {
    Yylex yy = new Yylex(System.in);
    Symbol t;
    try {
        while ((t = yy.yylex()).sym != Symbol.EOF)
            System.out.println(t);
    } catch (ParseError e) {
        System.err.println (e.getMessage());
    }
  }
}
```

This driver reads a **fab** program from standard input, and prints the resulting token stream, one token per line, on standard output. Any errors are sent to standard error. For example, the input stream

```
write (4, "= 2+2 =", [* newline here! *]
                  4.00);
```

should produce the output

```
1:      WRITE
1:      LPAREN
1:      INTEGER 4
1:      COMMA
1:      STRING  "= 2+2 ="
1:      COMMA
2:      REAL    4.00
2:      RPAREN
2:      SEMI
```

A working implementation of `Yylex` is in on the web page in file `Yylex.class`. Your program should generate the same output as this one, except that errors (sent to standard error) may be different in format (though not in substance). Note that checking for lexical errors is a very important part of the assignment; failing to implement error checks correctly can have a large impact on your score.

Implementation and Assignment Submission

All you need to implement is the `Yylex` class, which must operate correctly in conjunction with the other provided classes.

You are strongly encouraged (though not required) to use the lexical analyzer generator tool `JFlex`. You can download a copy of `JFlex`, including documentation, from the course web page. The User's Manual describes how to install and use `JFlex`.

If you choose to use `JFlex`, you should submit a single file `fab.lex` containing your `JFlex` specification. (Remember, if you need to define any additional auxiliary classes, you can put them at the top of your `JFlex` file.) When fed to `JFlex`, your specification must produce a file `Yylex.java` that defines a class `Yylex` with constructor and `yylex` function as described above; you may need to use some of the `yylex` `%` options to make this happen properly.

If you choose not to use `JFlex`, you must write your lexer by hand; you may *not* use a different lexer-generator tool. If you choose this option, submit a single file `Yylex.java` defining class `Yylex` in the usual way.

In either case, your file should be submitted as a plain text attachment to a mail message sent to `cs321-03@cecs.pdx.edu`. Your code must work correctly with the provided `LexerDriver`, `Symbol`, and `ParseError` classes; you may *not* modify these classes, and you should not submit any code for them. We will process your submission by creating a fresh directory, copying in the provided `.java` files and saving your attachment. If you submit a `.lex` file, we will pass it through `JFlex` to obtain a file `Yylex.java`. We will then execute

```
javac Yylex.java Symbol.java ParseError.java LexerDriver.java
```

To test the resulting program on a **fab** file `foo.fab`, we should be able to type

```
java LexerDriver < foo.fab
```

Note that we will be using automated mechanisms to read, compile, and test your programs, so adherence to this naming and mailing policy is important! You may lose points if you fail to submit your program in the correct way.