CS 541 — Spring 2014

Programming Assignment 2
CSX Scanner

Your next project step is to write a scanner module for the programming language **CSX** (**C**omputer **S**cience e**X**perimental). Use the *JFlex* scanner-generation tool (based on Lex). Future assignments will involve a CSX parser, type checker and code generator.

The CSX Scanner

Generate the CSX scanner, a member of class `Yylex`, using *JFlex*. Your main task is to create the file `csx.jflex`, the input to JFlex. The `jflex` file specifies the regular expression patterns for all the CSX tokens, as well as any special processing required by tokens.

When a valid CSX token is matched by member function `yylex()`, it returns an object that is an instance of class `java_cup.runtime.Symbol` (the class our parser expects to receive from the scanner). `Symbol` contains an integer field `sym` that identifies the token class just matched. Possible values of `sym` are identified in the class `sym`[1].

`Symbol` also contains a field `value`, which contains token information beyond the token's identity. For CSX, the `value` field references an instance of class `CSXToken` (or a subclass of `CSXToken`). `CSXToken` contains the line number and column number at which each token was found. This information is necessary to frame high-quality error messages. The line number on which a token appears is stored in `linenum`. The column number at which a token begins is stored in `colnum`. The column number counts tabs as one character, even though they expand into several blanks when viewed.

You must also store auxiliary information for identifiers, integer literals, character literals and string literals. For identifiers, class `CSXIdentifierToken`, a subclass of `CSXToken`, contains the identifier's name in field `identifierText`. For integer literals, class `CSXIntLitToken`, a subclass of `CSXToken`, contains the literal's numeric value in field `intValue`. For character literals, class `CSXCharLitToken`, a subclass of `CSXToken`, contains the literal's character value in field `charValue`. For

1 Java class names normally are capitalized. However, certain classes created by the tool Java CUP ignore this convention.

string literals, class `CSXStringLitToken`, a subclass of `CSXToken`, contains a field `stringText`, the full text of the string (with enclosing double quotes and internal escape sequences included as they appeared in the original string text that was scanned).

CSX Tokens

The CSX languages uses the following classes of tokens:

- The **reserved words** of the CSX language:

```
bool    break   char    class   const         continue
else    false   if      int     read          return
true    void    while   print
```

The `break` and `continue` reserved words are optional; compilers that include them receive extra credit.

- **Identifiers**. An identifier is a sequence of letters and digits starting with a letter, excluding reserved words.

  Id = (A | B | .|. Z | a | b | .z.) (A | B | .|. Z | a | b | .z. | 0 | 1 | .9.)* – Reserved

- **Integer Literals**. An integer literal is a sequence of digits, optionally preceded by a ~. A ~ denotes a negative value.

  IntegerLit = (~ | λ) (0 | 1 | .|. 9)+

- **String Literals**. A string literal is any sequence of printable ASCII characters, delimited by double quotes. A double quote within the text of a string must be escaped (as \", to avoid being misinterpreted as the end of the string). Tabs and newlines within a string must be escaped (`\n` is a newline and `\t` is a tab). Backslashes within a string must also be escaped (as `\\`). No other escaped characters are allowed. Strings may not cross line boundaries.

  StringLit = " (Not(" | \ | UnprintableChar) | \" | \n | \t | \\ )* "

- **Character Literals**. A character literal is any printable ASCII character, enclosed within single quotes. A single quote within a character literal must be escaped (as `\'`, to avoid being misinterpreted as the end of the literal). A tab or newline must be escaped ('`\n`' is a newline and '`\t`' is a tab). A backslash must also be escaped (as '`\\`'). No other escaped characters are allowed.

  CharLit = ' (Not(' | \ | UnprintableChar) | \' | \n | \t | \\ ) '

- **Other Tokens**. These are miscellaneous one- or two-character symbols representing operators and delimiters.

```
( ) [ ] = ; + - * / == != && || < > <= >= , ! { } :
```

- **End-of-File (EOF) Token**. The EOF token is automatically returned by `yylex()` when it reaches the end of file while scanning the first character of a token.

Comments and white space, as defined below, are not tokens because they are not returned by the scanner. Nevertheless, they must be matched (and skipped) when they are encountered.

- **A Single Line Comment**. As in C++ and Java, this style of comment begins with a pair of slashes and ends at the end of the current line. Its body can include any character other than an end-of-line.

  LineComment = // Not(Eol)* Eol

- **A Multi-Line Comment**. This comment begins with the pair ## and ends with the pair ##. Its body can include any character sequence other than two consecutive #'s.

  BlockComment = ## ((# | λ) Not(#))* ##

- **White Space**. White space separates tokens; otherwise it is ignored.

  WhiteSpace = (Blank | Tab | Eol) +

Any character that cannot be scanned as part of a valid token, comment or white space is invalid and should generate an error message.

Considerations/Requirements

• Because reserved words look like identifiers, you must be careful not to miss-scan them as identifiers. You should include distinct token definitions for each reserved word *before* your definition of identifiers.

• Upper- and lower-case letters are equivalent in reserved words and in identifiers. When you print a reserved word or an identifier, you may either print its original case or a conversion to standard case (such as lower case).

• Print character and string literals as they are input, that is, with the escaped characters shown as \n, \\, or whatever, and with the surrounding quotes. However, you should also store the effective values of character and string literals, in which escaped characters are replaced by their meaning, and surround quotes are removed.

• You should not assume any limit on the length of identifiers.

• You should not assume any limit on the length of input lines that are scanned.

• You may use Java API classes to convert strings representing integer literals to their corresponding integer values. Be careful though; in Java a minus sign, –, and not ~ represents a negative value. You must detect and report overflow in a system-independent fashion, perhaps using the constants `MIN_VALUE` and `MAX_VALUE` in

class `Integer`. Do not halt on overflow; print an error message and return `MAX_VALUE` or `MIN_VALUE` as the "value" of the literal.

An online reference manual for JFlex may be found in the "Useful Programming Tools" section of the class homepage.

• Although JFlex's regular expression syntax is designed to be very similar to that of Lex, it is not identical. Read the JFlex manual carefully.

◦ A blank *should not* be used within a character class (i.e., `[` and `]`). You may use `\040` (which is the character code for a blank).

◦ A doublequote *is* meaningful within a character class (i.e., `[` and `]`).

• As was the case in project 1, `javac` requires an environment variable `CLASSPATH` to define the directories to be searched to find "`.class`" files stored in libraries. JFlex and Java Cup (in the next assignment) use `CLASSPATH` to tell Java where to find the classes that they use. Once again, the `Makefile` we supply places all "`.class`" files in subdirectory `classes`.

• Skeleton files and a `makefile` are in the directory `~raphael/-courses/cs541/public/proj2/startup`.; they are also available through the class homepage. Do not assume that the Java is up to coding standard; use a style checker to improve the code.

What to hand in

Submit your project electronically by mailing it to raphael@cs.uky.edu. Please run `make clean` first to remove all `class` files. Hand in: (1) the `csx.jflex` file you create, (2) any other classes you create, (3) the test data you use to test your scanner, (4) the outputs produced using your test data, (4) a `README` file, (5) a `Makefile` , and (6) all source files necessary to build an executable version of your program (`.java` files and a `csx.jflex` file). Name the class that contains your `main()`routine `P2.java`.

Your scanner test program should act like the test program illustrated below, reading a stream of characters from the command line file and printing out the tokens matched to the standard output, one per line in the following format:

```
line:column token
```

For identifiers, include the text of the identifier; for integer literals, include the literal's numeric value; for character and string literals, include the literal's full text (with enclosing quotes and escape sequences). Use the following format

```
line:column Token (value)
```

For example, if the contents of command line file is:

```
class T {
// hello, this is
```

```
   //  a test
const
cnst
"hello\n"
^
 ~10;
```

You should produce:

```
1:1 class
1:7 Identifier (T)
1:9 {
4:1 const
5:4 Identifier (cnst)
6:1 String literal ("hello\n")
7:1 Invalid token (^)
8:1 Integer literal (-10)
8:4 ;
```

Your program should try to follow this format to ease grading. A significant fraction of your grade will be based on the **quality of your test data**. Please exercise your program in every possible way. Your program should print appropriate error messages if it scans an invalid token . You may handle strings that attempt to cross a line boundary either by refusing to accept the initial double-quote, which will lead to a cascade of error messages, or by explicitly treating such attempts by returning an error token that contains all the input up to the line boundary.