CS 536 — Fall 2018

Programming Assignment 2
CSX Scanner

Due: Tuesday, October 16, 2018

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

The CSX Scanner

The CSX scanner, a member of class `Yylex`, will be generated using *JLex*. Your main task will be to create the file `csx.jlex`, the input to JLex. `csx.jlex` 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 will return an object that is an instance of class `java_cup.runtime.Symbol` (this is 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` that contains additional token information (beyond the token's identity). For CSX, the `value` field will reference an instance of class `CSXToken` (or a subclass of `CSXToken`). `CSXToken` will contain 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 began is stored in `colnum`. The column number should count tabs as one character, even though, when viewed, they expand into several blanks.

You will also have to store auxiliary information for identifiers, integer literals, integer bit strings, character literals and string literals. For identifiers, class `CSXIdentifierToken`, a subclass of `CSXToken`, will contain the identifier's name in field `identifierText`. For integer literals, class `CSXIntLitToken`, a subclass of `CSXToken`, will contain the literal's numeric value in field `intValue`. For bit strings, class `CSXBitStringToken`, a subclass of `CSXToken`, will contain the bit string's numeric value in field `intValue` and string form (as scanned) in `bitString`. For character literals, class `CSXCharLitToken`, a subclass of `CSXToken`, will contain the literal's character value in field `charValue`. For string literals, class `CSXStringLitToken`, a subclass of `CSXToken`, will contain a field `stringText` that is 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).

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

CSX Tokens

CSX contains the following classes of tokens:

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

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

- **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)$^{+}$

- **Integer Bit String**. An integer bit string is a sequence of bits, followed by a B or b. Bit strings can be no longer than 32 bits.

  BitString = (0 | 1)$^{+}$(B | b)

- **String Literals**. A string literal is any sequence of printable characters, delimited by double quotes. A double quote within the text of a string must be escaped (to avoid being misinterpreted as the end of the string). Tabs and newlines within a string are escaped as usual (e.g., `\n` is a newline and `\t` is a tab). Backslashes within a string must also be escaped (as `\\`). Strings may not cross line boundaries.

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

- **Character Literals**. A character literal is any printable character, enclosed within single quotes. A single quote within a character literal must be escaped (to avoid being misinterpreted as the end of the literal). A tab or newline must be escaped (e.g., '`\n`' is a newline and '`\t`' is a tab). A backslash must also be escaped (as '`\\`').

  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 EOF is reached while scanning the first character of a token.

- **A Single Line Comment**. As in C++ and Java, this 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)*

- **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**. This separates tokens; otherwise it is ignored.

  WhiteSpace = (Blank | Tab | Eol) +

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

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

Considerations/Requirements

- Because reserved words "look like" identifiers, you must be careful not to misscan 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.
- 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 negative values. Also, you must detect and report overflow. You should do this 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. Bit strings shorter than 32 bits are assumed to have 0 bits added on the left. The only bit strings that represent negative (two's complement) values are those that are 32 bits long and have a leading (leftmost) 1 bit. For bit strings longer than 32 bits, issue a warning and remove bits from the left end until a length of 32 is reached.

  One easy way to convert a string representing an integer to an `int`, with overflowing checking, is to convert it first to a `double`, then compare the value against `MAX_VALUE` and `MIN_VALUE`, then convert the `double` to an `int` if it is "in range."
- An on-line reference manual for JLex may be found in the "Useful Programming Tools" section of the class homepage.
- Although JLex's regular expression syntax is designed to be very similar to that of Lex, it is not identical. Read the Jlex manual carefully. Significant differences include:
  - Escaped characters within quoted strings *are not* recognized. Hence `"\n"` *is not* a new line character.
  - 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 `]`).

- To get you started, a partial solution to this assignment is available as an Eclipse archive file: http://www.cs.wisc.edu/~fischer/cs536.f18/course/proj2/startup/eclipse). Also available is a folder that contains Java source files, the *JLex* scanner generator and a `build.xml` build file.

What to hand in

Create a folder (directory) and name it using your first and last name (e.g., `CharlesFischer`). Copy into this folder a `README` file, a `build.xml` file (if you changed what we provide), and all source files necessary to build an executable version of your program (`.java` source files and a `csx.jlex` file). Do not hand in any `.class` files. Name the class that contains your `main` `P2.java`. Electronically submit your folder to the Project 2 tab on Canvas. Partners should submit only *one* solution. The other partner should submit only a `README` file identifying the partnership.

When your program begins execution it should print out your full name and student ID number. You should also print the name of the file being scanned. 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 and bit strings include the token's numeric value, and 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 `test.csx` is:

```
class T {
// hello, this is
// a test
const
   cnst
"hello"
^
10;
101B
```

You should produce:

```
Scanning file test.csx:
 1:1 class
 1:7 Identifier (T)
 1:9 {
 4:1 const
 5:4 Identifier (cnst)
 6:1 String literal ("hello")
 7:1 **ERROR: invalid token(^)
 8:1 Integer literal (10)
 8:3 ;
 9:1 bit string(5)
```

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. Appropriate error messages should also be printed if an invalid token is scanned. Since you are testing only your scanner, the input file you use need not be a valid CSX program.