CS 536 — Spring 1998

Programming Assignment 2
CSX Scanner

Due: Tuesday, February 24, 1998

Not accepted after Tuesday, March 3, 1998

Your next project step will be to write a scanner module for the programming language CSX (Computer Science eXperimental). You will use *Flex* or *JLex*, scanner generation tools based on Lex. You will utilize the `Registration` class written in Programming Assignment 1. Future assignments will involve a CSX parser, type checker and code generator.

The CSX Scanner (C++ version)

The CSX scanner, `yylex`(), will be generated using Flex. Your main task will be to create the file `csx.lex`, the input to Flex. `csx.lex` 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, `yylex`() will return the token's code. For one character tokens, the token code is the character itself. Thus the token code for `'('` is 40, because that's the ASCII character code for `'('`. For multi-character tokens (identifiers, string literals, `"=="`, etc.), the appropriate token code, in the form of a named constant, is defined in `y.tab.h`.

As well as returning token codes, your Flex-generated scanner will have to store the line number and column number at which each token is found. (This information is necessary to frame high-quality error messages.) Line and column information will be stored in the variable `yylval`, which is a union. The line number on which a token appears will be stored in `yylval.t.linenum`. The column number at which a token begins will be stored in `yylval.t.colnum`.

You will also have to store auxiliary information for identifiers, integer literals, character literals and string literals. For identifiers, `yylval.t.idval` will contain the identifier's serial number (obtained by registering it using `registerStr`()). For integer literals, `yylval.t.intval` will contain the numeric value of the literal. For character literals, `yylval.t.charval` will contain the character value of the literal. For string literals, `yylval.t.strval` will point to the text of the string (with enclosing double quotes removed and internal escape sequences replaced with the values they denote). `yylval.t.fullstr` will point to 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).

The CSX Scanner (Java version)

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 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`.

You will also have to store auxiliary information for identifiers, integer literals, character literals and string literals. For identifiers, class `CSXIdentifierToken`, a subclass of `CSXToken`, will contain the identifier's serial number in field `serialNum`. `CSXIdentifierToken` will also contain a static field `reg`, of class `Registration`, that can be used to obtain serial numbers for identifiers using `registerStr()`. For integer literals, class `CSXIntLitToken`, a subclass of `CSXToken`, will contain the literal's numeric value in field `intValue`. 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 `stringValue` that is the value of the string (with enclosing double quotes removed and internal escape sequences replaced with the values they denote). Field `fullStringValue` will contain 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

CSX contains the following classes of tokens

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

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

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

- **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 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.

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 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**. This 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 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 C++ library routines or Java API classes to convert strings representing integer literals to their corresponding integer values. Be careful though; in C++ and 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 `MAXINT` and `MININT` defined in `csx.lex` (or `MIN_VALUE` and `MAX_VALUE` in class `Integer)`. Do not halt on overflow; print an error message and return `MAXINT` or `MININT` as the "value" of the literal.

  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 `MAXINT` and `MININT,` then convert the `double` to an `int` if it is "in range."
- On-line reference manuals for Lex, Flex and 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 and Flex, 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 `]`).
- `java` and `javac` use an environment variable `CLASSPATH` to define the directories to be searched to find ".class" files stored in libraries. To use JLex and Java Cup (in the next assignment) you'll need to add the following line to your `.cshrc.local` file in your home directory (enter this as one line):

```
setenv CLASSPATH .:/p/course/cs536-fischer/public/java:/p/course/
cs536-fischer/public/java/cup
```

- Skeleton files and a makefile are in the directories `~cs536-1/public/proj2/c++` and `~cs536-1/public/proj2/java`; they are also available through the class homepage.

What to hand in

Hand in listings of: (1) the `csx.lex` or `csx.jlex` file you create, (2) any auxiliary subroutines or classes you create, (3) the test data you use to test your scanner, and (4) the outputs produced using the test data. We've created a directory for you using your login in `~cs536-1/public/proj2/handin`. Copy into your handin directory a `README` file (this is optional), a `Makefile` (this is required), and all source files necessary to build an executable version of your program (`.cc` or `.java` files and a `csx.lex` or `csx.jlex` file). Do not hand in executables or `.class` files. For C++ projects, name the executable your `Makefile` creates `proj2`; for Java name the class that contains your `main` `proj2.java`.

When your program begins execution it should print out your full name and student ID number. 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 serial numbers assigned to the identifier, for integer literals include the literal'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 command line file is:

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

You should produce:

```
1:1 class
1:7 Identifier (0)
1:9 {
4:1 const
5:4 Identifier (1)
6:1 String literal ("hello")
7:1 **ERROR: invalid token (^)
8:1 Integer literal (10)
8:3 ;
```

Your program doesn't have to exactly follow this format, but should provide an easy to follow listing of scanned tokens. 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.