CS 536 — Spring 1995
Announcement 4

Programming Assignment #2

CSX Scanner

Due: Monday, March 6, 1995
Not accepted after class on March 20, 1995

Your next project step will be to write a scanner module for the programming language CSX (Computer Science eXperimental). You will utilize the symbol table module written in Programming Assignment 1. To gain experience with scanner generators, you will also use Lex to create a skeletal CSX scanner. Future assignments will involve a CSX parser, type-checker and code generator.

The CSX Scanner

The CSX scanner should be implemented as a subprogram (written in C or C++) that takes a text file as input, and returns a struct with three fields, tokenkind, tokenval, and linenumber. Tokenkind is an enumeration type, with values defined for each token class (there are 39 of them). The tokenval part of the record is relevant only for identifiers, integer literals and string literals. For integer literal tokens the value returned should be an integer, not a character string. This means that the scanner must translate sequences of digits to integers. For identifiers, the value returned should be a serial number returned by the registerStr routine. For string literals, the value returned should be a pointer to the text of the string. Linenumber is the number of the source line in which the token appeared. (It will be used by later compiler phases for error messages). The scanner should also write messages (to the terminal or to an error file) reporting the presence of illegal tokens or illegal characters in the input.

CSX Tokens

- Reserved = (main | const | proc | int | bool | true | false | if | then | else | while | loop | end | get | print | skip | ref | return)

  These are the reserved words of the CSX language.

- Id = (A | B | $\cdots$ | Z | a | b | $\cdots$ z) (A | B | $\cdots$ | Z | a | b | $\cdots$ z | 0 | 1 | $\cdots$ 9)* − Reserved

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

- IntegerLit = (˜ | λ) (0 | 1 | $\cdots$ | 9)$^+$

  An integer literal is a sequence of digits, optionally preceded by a ˜. A ˜ denotes a negative value.

- StringLit = " (Not(" | UnprintableChars) | "")* "

  A string literal is any sequence of printable characters, delimited by double quotes. A double quote within the text of a string must be repeated (to avoid being misinterpreted as the end of the string). Strings may not cross line boundaries.
- OtherTokens = (′(′ | ′)′ | = | ; | + | – | ∗ | / | == | != | && | ′||′ | < | > | <= | >= | ′,′)

  These are miscellaneous one- or two-character symbols representing operators and delimiters.
- A special EOF token to indicate that the entire source file has been read.

Comments and WhiteSpace, as defined below, are not tokens because they are not returned by the scanner. Nevertheless they must be scanned (and deleted) when they are encountered.

- Comment1 = // Not(Eol)* Eol

  A single line comment, as in C++, 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.
- Comment2 = { Not(})* }

  A multi-line comment is delimited by curly braces; it can include any character other than a right (closing) curly brace. These comments can occur anywhere other than in the middle of a token (*i.e.* the beginning of a comment ends the current token); they **may** cross line boundaries.
- WhiteSpace = (Blank | Tab | Eol)+

  White space separates tokens; otherwise it is ignored.

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

- When the scanner recognizes what looks like an identifier, it must check to see if the token is actually a reserved word. You may include CSX reserved words directly in your scanner code (*e.g.* use a series of *if-then-else* statements to test whether an identifier is a reserved word), or you may store reserved words in a table (perhaps using the symbol table module implemented for program #1) and do lookups on scanned identifiers. Remember, the scanner must return a *different* token for each of the reserved words.
- Upper- and lower-case letters are equivalent in reserved words and in identifiers. To simplify things you may convert all letters to a uniform case while scanning (or let the registerStr routine handle the problem).
- You should not assume any limit on the length of identifiers other than that possibly imposed by your symbol table module.
- You should not assume any limit on the length of input lines that are scanned. (I.e., you should *not* try to store an entire source line in a line buffer).
- You may use C library routines (like strtol or atoi) to convert strings representing integer literals to their corresponding integer values. Be careful though; in C – 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

scanner.h. Do not halt on overflow; print an error message and return MAXINT or MININT as the “value” part of the returned struct.

- Skeleton files and a makefile will be in the directories ˜cs536-1/public/proj2/c and ˜cs536-1/public/proj2/c++.

Generating a CSX scanner using Lex

To get a bit of experience using a scanner generator, a lex definition of CSX tokens appears in ˜cs536-1/public/proj2/c/csx.lex and ˜cs536-1/public/proj2/c++/csx.lex. This is a complete lex specification for the tokens of CSX except for the regular expressions defining the tokens. These have been filled in with a sequence of XXXXXXXXX markers. Replace them with the correct regular expression definitions. (The token assignment at the right of each line says what token the line should match.) Run

```
make lexscanner
```

This will run lex to produce the file lex.yy.c, and then run gcc to produce the executable program lexscanner. Lexscanner will scan a file (from standard input) and list out the tokens it matches, indented one tab stop. (Characters it cannot scan at all will be echoed, without indentation). Run

```
make lextest
```

to build lexscanner and then test it using test.csx.

The lex specification provided is actually a valid token specification. It can be processed by lex, and then compiled by gcc. The only problem is that all tokens are defined to be XXXXXXXXXXXXXX — a most uninteresting definition. A good way to start this project step is to replace XXXXXXXXXXXXXX markers with the correct regular expressions for a few simple tokens. Once these are working add the more complex ones. There is a good example of a lex token specification on p. 67 of *Crafting a Compiler with C.* Remember that characters like parentheses, plus, star, etc that are used to form regular expressions may need to be quoted when they appear in token definitions. Thus you should use (e.g.)

```
"("          {token = LparenTok; return;}
```

instead of

```
(            {token = LparenTok; return;}
```

What to hand in

Hand in listings of: (1) the source files for your scanner module and main program, (2) the csx.lex file you create, (3) the test data you use to test your scanner modules (both C/C++ and lex), and (4) the outputs produced using the test data. You should also place an executable copy of your scanner test program (the “hand coded” one; not the Lex scanner) in the directory ˜cs536-1/public/proj2/handin. The name of the program should be your login. When your program begins execution it should print out your full name and student ID number. Your scanner test program should act like “lexscanner” which accepts a stream of characters from standard input and lists out the tokens matched to the standard output. The following is an example :

```
input :

      MAIN
      { hello, this is
        a test }
      const
           cnst
      "hello"
      ˆ &
      10;

output :

      main
      const
      Identifier: cnst
         serial number : 1
      String_lit: "hello"
      **ERROR, line 7, invalid token ˆ
      **ERROR, line 7, invalid token &
      Int_lit: 10
      ;
```

Your program doesn't have to exactly follow this format. 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.

Try to make your test data sophisticated enough so that the TA doesn't have to run your program with *his* test cases.