CS210 - Programming Languages
Lab Assignment #2 - Fall 2020

The purpose of this assignment is to build on your assignment #1 by adding classifications to the lexemes you found earlier.

For this assignment you are to build a lexer that will successfully scan through a set of programs expressed in the CCX programming language. You have never used CCX, and that is just fine: scanning through CCX programs won't require intimate knowledge of CCX.

Your lexer shall be written in the C programming language. You may not use C++. Your lexer will be compiled and tested using `gcc` on the course server, so you should at least test your lexer in this same environment.

The sample CCX program you used in assignment #1 is repeated below. You can use this same file to test your new program.

```
/*
 * Hello world with args.
 */
procedure main(argc: integer; argv: string_vector_type) is
begin
    printf("Hello, world\n");
    loop
        argc := argc - 1;
        exit when (argc = 0);
        printf("arg[%d]: %s\n", argc, argv[argc]);
    end loop;
end main;
```

Figure 1: CCX Sample 1

As before, your lexer should open a file provided on the command-line and discover the lexemes found in the file. Additionally, your program should classify each lexeme, and print out each lexeme and its classification to the computer screen. Your lexer should classify each lexeme found in the source file into one of 8 categories. These categories are: `comment`, `string`, `keyword`, `character literal`, `numeric literal`, `operator`, `identifier`, and `UNK`.

The following is the output produced by a lexer when scanning the example source code. Each lexeme and its classification is printed on a separate line. A single space appears between each lexeme and its classification, and the classification appears in parentheses.

Please examine this output closely. Each lexeme must be printed on a separate line, and a single space must appear between the lexeme and its classification. The lexeme itself must start in the first column on a given line. The classification of a lexeme must appear in parentheses. No "blank" or "empty" lines can appear in the output unless they are part of a multi-line comment. Each lexeme must be printed precisely as it appears in the source file. Do not bracket the lexeme in quotes or any other characters.

```
/*
 * Hello world with args.
 */ (comment)
procedure (keyword)
main (identifier)
( (operator)
argc (identifier)
: (operator)
integer (keyword)
; (operator)
argv (identifier)
: (operator)
string_vector_type (identifier)
) (operator)
is (keyword)
begin (keyword)
printf (identifier)
( (operator)
"Hello, world\n" (string)
) (operator)
; (operator)
loop (keyword)
argc (identifier)
:= (operator)
argc (identifier)
- (operator)
1 (numeric literal)
; (operator)
exit (keyword)
when (keyword)
( (operator)
argc (identifier)
= (operator)
0 (numeric literal)
) (operator)
; (operator)
printf (identifier)
( (operator)
"arg[%d]: %s\n" (string)
, (operator)
argc (identifier)
, (operator)
argv (identifier)
[ (operator)
argc (identifier)
] (operator)
) (operator)
; (operator)
end (keyword)
loop (keyword)
; (operator)
end (keyword)
main (identifier)
; (operator)
```

The Lexeme Categories

As mentioned, your lexer should classify each lexeme encountered into one of 8 categories. The details of each category follow.

- `comment`
  Comments in CCX begin with `/*` and end with `*/` (C-style comments). Comments can span multiple lines. Everything encountered between (and including) the `/*` and `*/` delimiters is considered part of the comment lexeme.

- `identifier`
  Identifiers are used in programs to name entities such as variables. Every programming language has its own rules as to what constitutes a legal identifier. In CCX an identifier can be composed of letters, digits, and underscores, but must start with a letter. You may assume that your lexer will never encounter an identifier that is more than 256 characters long.

- `string`
  Strings in CCX are literals delimited by double-quotes "like this". The double-quotes are part of the lexeme. When you print a lexeme that has been classified as a string, you **must** print the double-quotes. You may assume that your lexer will never encounter a string that is more than 256 characters long.

- `keyword`
  CCX contains many keywords. Keywords are sometimes called *reserved words*. Keywords (like all of CCX) are case-sensitive, and may not be used as identifiers in legal programs. It is not the job of the lexer to determine whether a keyword is misused; the lexer simply classifies a particular lexeme as being a keyword. The following are the list of CCX keywords that your lexer must recognize:

  ```
  accessor and array begin bool case character constant else elsif end exit function
  if in integer interface is loop module mutator natural null of or others out positive
  procedure range return struct subtype then type when while
  ```

- `character literal`
  Character literals in CCX are literals in single-quotes like this: 'x'. CCX allows character escape sequences in character literals, such as '\020' but your lexer need not support this.

- `operator`
  CCX contains many operators. Some operators consist of a single character, whereas others contain multiple characters. The following is a list of the operators that your lexer must recognize. Each operator is enclosed in double-quotes for the purpose of disambiguation, but these double-quotes are **not** part of the operator:

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

- `numeric literal`
  CCX allows numeric literals in multiple forms. Your lexer will recognize a simplified subset of CCX numeric literals. Each numeric literal encountered by your lexer will start with a decimal digit and will contain only the following:

  - decimal digits (0 through 9)
  - hexadecimal digits (A through F and/or a through f)
  - the special characters '_', '.', and '#'.

  any other character encountered will denote that the numeric literal has ended and a new lexeme has begun.

- `UNK`
  This special category is set aside for lexemes that your lexer cannot classify, and is intended to assist you in building and debugging your lexer. This category is composed of all lexemes that do not fit in any of the other specified categories. Your lexer will only be tested against legal CCX programs, so if the logic in your lexer is correcti and complete, you should never encounter the `UNK` lexeme. If, however, your lexer does encounter a lexeme that does not fit the requirements of any of the other categories, your lexer should print the offending lexeme, along with its category name in parenthesis, and immediately terminate. If you build your lexer incrementally (as suggested in the Hints section below) you will find that the `UNK` category is quite useful.

Grading

Your lexer will be built using `gcc` on the course server. Your lexer will be tested on the course server against one of the example files provided to you in the tar file provided with the assignment.

The output of your lexer will be compared with the correct result for each file. Your program may not ask the user for the name of a file to scan, or the number of files to scan, or anything of this sort. If your lexer does not process command-line arguments for any reason, at least 20% of the total possible points on this assignment will be deducted from your score. Your program must not rely upon external files (other than the input sorce code) in order to run. You must not encode keywords, operators, etc. in separate files.

You are encouraged to use a commenting style that best fits your workflow, but you should at least provide comments that describe the role and inputs/outputs of each function that you create.

Hints

Your lexer will only be tested against one of the source files listed above. Each of these source files is legal CCX. Your lexer is not expected to be bulletproof, so don't spend time trying to handle the rather large set of all legal and illegal CCX programs.

Build your lexer incrementally. For example, one strategy is to start off with all lexemes being classified as `UNK`, then add classification categories one at a time until your lexer is complete, and doesn't show any `UNK` messages. Perhaps the first category you tackle should be `keywords`. You could also test this step by creating a file containing just the CCX keywords, and test your lexer. Once it has been tested, add the ability to recognize CCX operators. Then create a file containing just keywords and operators and test your lexer. Continue in this fashion until your lexer is complete.

Submit your final program source code using the `checkin` program. If it consists of multiple source files, use `tar` to places the individual files into a single archive (`.tar`) file.