CS210 - Programming Languages
Homework #1 - Spring 2024

Due on or before Sunday, February 18 at 11:59:59 PM

Modern translation systems often use lexical analysis to divide an input into meaningful units. Once these meaningful units (or *lexemes* or *tokens*) have been derived, other components within the translation system are used to determine the relationships among the lexemes. Lexical analyzers (or *lexers*) are commonly used in compilers, interpreters, and other translation systems that you have often used. The act of lexical analysis is also known as *scanning*.

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 (cs-210.cs.uidaho.edu), so you should at least test your lexer in this same environment. You are expressly forbidden from using any standard template library (STL) facilities in your lexer. You may not use any lexer generators or other automated tools to complete this assignment.

A sample CCX program is shown in Figure 1 below. This program simply prints the string "Hello, world" to the screen and then prints the arguments to the program if any were provided. You have probably seen many programs like this before.

```
/*
 * Hello world with args.
 */
proc main(argc: integer; argv: string_vector_type)
{
    printf("Hello, world\n");
    {:
        argc := argc - 1;
        exit when (argc = 0);
        printf("arg @ %d: %s\n", argc, argv @ argc);
    }:
};
```

Figure 1: CCX Sample 1

The goal of lexical analysis is to break programs like the one in Figure 1 into lexemes that are eventually used by other components within the translation system to determine things like whether the program is "legal" and what the program does. Although the requirements and constraints imposed upon lexical analysis may vary considerably between different translation systems, the requirements for most lexers (and for this assignment) are very simple.

Your lexer shall open a file provided on the command-line, discover the lexemes found in the file, classify each lexeme, and print out each lexeme and its classification to the computer screen. Your lexer shall classify each lexeme found in a given source file into one of

8 categories. These categories are: `comment`, `string`, `keyword`, `character literal`, `numeric literal`, `operator`, `identifier`, and `UNK`. The details concerning each of these categories is specified later in this document, but for now let's look at the output for a CCX lexer that meets the requirements of this assignment.

The following is the output produced by a lexer when scanning the source code presented in Figure 1. 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.

```
/*
 * Hello world with args.
 */ (comment)
proc (keyword)
main (identifier)
( (operator)
argc (identifier)
: (operator)
integer (keyword)
; (operator)
argv (identifier)
: (operator)
string_vector_type (identifier)
) (operator)
{ (operator)
printf (identifier)
( (operator)
"Hello, world\n" (string)
) (operator)
; (operator)
{: (operator)
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)
} (operator)
; (operator)
```

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.

The Lexeme Categories

As mentioned, your lexer shall 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 bool case character constant else elsif end exit float func
  ```

`if ifc in integer is mutator natural null of or others out pkg positive proc ptr range 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 correct, you should never encounter an `UNK` lexeme. If, however, your lexer does encounter a lexeme that does not fit the requirements of any of the other categories, your lexer must print the offending lexeme, along with its category name in parenthesis, and immediately terminate.

Another CCX Sample

The following is another CCX sample source file. CCX provides direct support for modular programming, and as such makes use of the notion of module interfaces. This file encodes the opaque types and interface for a module to support linked lists.

```
pkg list ifc {
    /*
     * types
     */
    type list_type;
    type listel_type;
    subtype list_size_type is natural;

    /*
     * routines
     */
    mutator  append(l: in out list_type; x: in ptr);
    accessor data(e: listel_type) $ ptr;
    func find(l: list_type; x: ptr) $ listel_type;
    accessor head(l: list_type) $ listel_type;
    mutator  insert(l: in out list_type; e: in out listel_type; x: in ptr);
    accessor next(e: listel_type) $ listel_type;
    mutator  prepend(l: in out list_type; x: in ptr);
    accessor prev(e: listel_type) $ listel_type;
    mutator  remove(l: in out list_type; e: in out listel_type);
    function size(l: list_type) $ list_size_type;
};
```

Figure 2: CCX Sample 2

It is apparent from the sample in Figure 2 that CCX supports accessors, mutators, and functions as well as procedures. Your lexer need not be concerned with the differences between each of these.

The following is the output produced by a lexer when scanning the source code presented in Figure 2.

```
pkg (keyword)
list (identifier)
ifc (keyword)
{ (operator)
/*
     * types
     */ (comment)
type (keyword)
list_type (identifier)
; (operator)
```

```
type (keyword)
listel_type (identifier)
; (operator)
subtype (keyword)
list_size_type (identifier)
is (keyword)
natural (keyword)
; (operator)
/*
     * routines
     */ (comment)
mutator (keyword)
append (identifier)
( (operator)
l (identifier)
: (operator)
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
x (identifier)
: (operator)
in (keyword)
ptr (keyword)
) (operator)
; (operator)
accessor (keyword)
data (identifier)
( (operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
$ (operator)
ptr (keyword)
; (operator)
func (keyword)
find (identifier)
( (operator)
l (identifier)
: (operator)
list_type (identifier)
; (operator)
x (identifier)
: (operator)
ptr (keyword)
) (operator)
$ (operator)
```

```
listel_type (identifier)
; (operator)
accessor (keyword)
head (identifier)
( (operator)
l (identifier)
: (operator)
list_type (identifier)
) (operator)
$ (operator)
listel_type (identifier)
; (operator)
mutator (keyword)
insert (identifier)
( (operator)
l (identifier)
: (operator)
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
e (identifier)
: (operator)
in (keyword)
out (keyword)
listel_type (identifier)
; (operator)
x (identifier)
: (operator)
in (keyword)
ptr (keyword)
) (operator)
; (operator)
accessor (keyword)
next (identifier)
( (operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
$ (operator)
listel_type (identifier)
; (operator)
mutator (keyword)
prepend (identifier)
( (operator)
l (identifier)
: (operator)
```

```
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
x (identifier)
: (operator)
in (keyword)
ptr (keyword)
) (operator)
; (operator)
accessor (keyword)
prev (identifier)
( (operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
$ (operator)
listel_type (identifier)
; (operator)
mutator (keyword)
remove (identifier)
( (operator)
l (identifier)
: (operator)
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
e (identifier)
: (operator)
in (keyword)
out (keyword)
listel_type (identifier)
) (operator)
; (operator)
function (identifier)
size (identifier)
( (operator)
l (identifier)
: (operator)
list_type (identifier)
) (operator)
$ (operator)
list_size_type (identifier)
; (operator)
} (operator)
; (operator)
```

Please carefully examine the output above. Note that even though the source in Figure 2 defines `list_type` as a type, the lexer classifies `list_type` as an identifier. This is correct. This means that your lexer need not be concerned about scalar, aggregate, or user-defined types that are encountered in source code. The list of keywords and operators provided earlier in this document will not change regardless of the *meaning* of the source code that your lexer is scanning at any given time. Since `list_type` is not a keyword and satisfies the rules for an identifier, it is classified as such.

In CCX the interface of a module and its implementation are two distinct entities. Each of these entities usually appears in a separate file. Files are usually named such that the interface for a module appears in a file named modulename.cci, and the implementation of the module appears in a file named modulename.ccx. In the sample code for the list interface shown in Figure 2 and the implementation shown above, the interface was placed in a file called `list.cci` and the implementation was placed in a file called `list.ccx`.

Grading

Your lexer will be built using `gcc` on the course server (cs-210.cs.uidaho.edu). Your lexer will be tested using the following input files:

```
hello_world.ccx
list.cci
list.ccx
complex.cci
complex.ccx
date.cci
date.ccx
natural.cci
natural.ccx
trie.cci
trie.ccx
```

Each of these source files and the result produced by a correct lexer when scanning the file is available on the course website. The output of your lexer will be compared with the correct result for each file. If your lexer does not compile using `gcc` on the course server for any reason, at least 50% of the total possible points on this assignment will be deducted from your score. You are required to do your own work for this assignment. Failure to comply will result in at least a score of 0 for this assignment and referral to the Dean of Students.

Your lexer must take its arguments from the **command-line**. If, for instance, the prompt on a given machine is `-bash-4.1$` your lexer will be executed as follows:

```
-bash-4.1$ ./hw1 hello_world.ccx
-bash-4.1$ ./hw1 list.cci
...
```

Your lexer is only required to process a single file passed as a command-line argument per program run. Your lexer will not be expected to handle two or more files on the command-line per program run. Your program may not ask the user for the name of a file to scan, or the

number of files to scan, or anything at all. Your lexer is not to interact with the user in any way other than via the command-line. If your lexer does not process command-line arguments for any reason, at least 25% of the total possible points on this assignment will be deducted from your score.

Your program will be evaluated for neatness and clarity as well as correctness. You must document your program using comments. You must use a consistent programming style which indicates that you are in control of your thoughts and the program which is being used to actualize them. Sloppy, ambiguous, convoluted, intentionally vague, undocumented, or insufficiently documented programs will be considered substandard and will be marked as such.

Hints

Read this assignment multiple times before you begin working on it. Points are deducted from the submissions of many students simply because they fail to fully understand the assignment before they turn it in. Don't be one of these students. Read and re-read this assignment, and refer to it while you are working on it.

Your lexer will only be tested against 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.

You would do well to think of your lexer as a state machine that operates on a character-by-character basis. The set of states in such a machine should be relatively small.

Build your lexer incrementally. For example, start off building a lexer that recognizes just keywords, and defaults to the `UNK` state. Then create 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.

Submitting Your Homework

Your homework must be submitted using the `cscheckin` program. A document detailing the use of `cscheckin` appears on the course website. You must turn in the source code for your lexer and a Makefile that will be used to build your lexer. Package the source code and your Makefile into a single archive using the `tar` program on the course server and check in the .tar file named "hw1.tar" (without quotes).