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. Your lexer will be compiled and tested using Code::Blocks Version 17.12 as configured on NIC lab workstations, so you should test your lexer in this environment. Your program must run from the DOS console (command line).

You are not allowed to use any standard template library (STL) containers (e.g. <vector>), algorithms (via <algorithm>), or other facilities in your lexer.

You may not use any lexer generators or other automated tools to complete this assignment.

All algorithms, structures, and techniques employed in your lexer must be of your own making.

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.

```
/*
* 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

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 Windows DOS command-line, discover the lexemes found in the file, classify each lexeme, and print out each lexeme and its classification to the output file named with the same name as the input file with ".lexer" as the last part of the name.

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)
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)
```

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 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 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.

```
module list interface
/*
* types
*/
type list_type;
type listel_type;
subtype list_size_type is natural;
/*
* routines
*/
mutator append(l: in out list_type; x: in ptr_type);
accessor data(e: listel_type) return ptr_type;
function find(l: list_type; x: ptr_type) return listel_type;
accessor head(l: list_type) return listel_type;
mutator insert(l: in out list_type; e: in out listel_type; x: in ptr_type);
accessor next(e: listel_type) return listel_type;
mutator prepend(l: in out list_type; x: in ptr_type);
accessor prev(e: listel_type) return listel_type;
mutator remove(l: in out list_type; e: in out listel_type);
function size(l: list_type) return list_size_type;
end list;
```

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.

```
module (keyword)
list (identifier)
interface (keyword)
```

```
/*
* 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_type (identifier)
) (operator)
; (operator)
accessor (keyword)
data (identifier)
( (operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
return (keyword)
ptr_type (identifier)
; (operator)
function (keyword)
find (identifier)
( (operator)
l (identifier)
: (operator)
list_type (identifier)
```

; (operator)
x (identifier)
: (operator)
ptr_type (identifier)
) (operator)
return (keyword)
listel_type (identifier)
; (operator)
accessor (keyword)
head (identifier)
((operator)
l (identifier)
: (operator)
list_type (identifier)
) (operator)
return (keyword)
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_type (identifier)
) (operator)
; (operator)
accessor (keyword)
next (identifier)
((operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
return (keyword)
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_type (identifier)
) (operator)
; (operator)
accessor (keyword)
prev (identifier)
((operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
return (keyword)
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 (keyword)
size (identifier)
((operator)
l (identifier)
: (operator)
list_type (identifier)

```
) (operator)
return (keyword)
list_size_type (identifier)
; (operator)
end (keyword)
list (identifier)
; (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.

The following CCX source code is the implementation of a module to support linked lists. This is the implementation of the module whose interface was shown in Figure 2 above.

```
module list is
type listel_type is
struct
data: ptr_type := null;
next: listel_type := null;
prev: listel_type := null;
end struct;
type list_type is
struct
head: listel_type := null;
tail: listel_type := null;
end struct;
mutator append(l: in out list_type; x: in ptr_type) is
tmp: listel_type;
begin
tmp := alloc(listel_type);
tmp.data := x;
tmp.next := null;
if (l.tail != null) then
tmp.prev := l.tail;
l.tail.next := tmp;
l.tail := tmp;
else
tmp.prev := null;
l.head := tmp;
l.tail := tmp;
end if;
end append;
```

```
accessor data(e: listel_type) return ptr_type is
begin
return e.data;
end data;
/*
* Old and nesty version (that works)
*
function find(l: list_type; x: ptr_type) return listel_type is
tmp: listel_type;
begin
tmp := l.head;
while (tmp != null) loop
if (tmp.data = x) then
exit;
end if;
tmp := tmp.next;
end loop;
return tmp;
end find;
*/
function find(l: list_type; x: ptr_type) return listel_type is
tmp: listel_type;
begin
tmp := l.head;
loop
exit when ((tmp = null) or (tmp.data = x));
tmp := tmp.next;
end loop;
return tmp;
end find;
accessor head(l: list_type) return listel_type is
begin
return l.head;
end head;
/*
* This routine performs a prefix insertion of an el with data x.
*/
mutator insert(
l: in out list_type; /* list to insert within */
e: in out listel_type; /* place to insert el before */
x: in ptr_type /* data of new el */
) is
tmp: listel_type;
begin
tmp := alloc(listel_type);
tmp.data := x;
tmp.next := e;
tmp.prev := e.prev;
```

```
if (e.prev != null) then
e.prev.next := tmp;
end if;
e.prev := tmp;
if (e = l.head) then
l.head := tmp;
end if;
end insert;
accessor next(e: listel_type) return listel_type is
begin
return e.next;
end next;
mutator prepend(l: in out list_type; x: in data_type) is
tmp: listel_type;
begin
tmp := alloc(listel_type);
tmp.prev := null;
tmp.data := x;
if (l.head != null) then
tmp.next := l.head;
l.head.prev := tmp;
l.head := tmp;
else
tmp.next := null;
l.head := tmp;
l.tail := tmp;
end if;
end prepend;
accessor prev(e: listel_type) return listel_type is
begin
return e.prev;
end next;
mutator remove(l: in out list_type; e: in out listel_type) is
begin
if (e.prev != null) then
e.prev.next := e.next;
end if;
if (e.next != null) then
e.next.prev := e.prev;
end if;
if (e = l.head) then
l.head := e.next;
end if;
if (e = l.tail) then
l.tail := e.prev;
end if;
dealloc(e);
e := null;
```

```
end remove;
function size(l: list_type) return list_size_type is
tmp: listel_type;
n: list_size_type;
begin
n := 0;
tmp := l.head;
while (tmp != null) loop
tmp := tmp.next;
n := n + 1;
end loop;
return n;
end size;
accessor tail(l: list_type) return listel_type is
begin
return l.tail;
end head;
end list;
```

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`.

The following is the output produced by a lexer when scanning the source code shown above.

```
module (keyword)
list (identifier)
is (keyword)
type (keyword)
listel_type (identifier)
is (keyword)
struct (keyword)
data (identifier)
: (operator)
ptr_type (identifier)
:= (operator)
null (keyword)
; (operator)
next (identifier)
: (operator)
listel_type (identifier)
:= (operator)
null (keyword)
; (operator)
```

prev (identifier)
: (operator)
listel_type (identifier)
:= (operator)
null (keyword)
; (operator)
end (keyword)
struct (keyword)
; (operator)
type (keyword)
list_type (identifier)
is (keyword)
struct (keyword)
head (identifier)
: (operator)
listel_type (identifier)
:= (operator)
null (keyword)
; (operator)
tail (identifier)
: (operator)
listel_type (identifier)
:= (operator)
null (keyword)
; (operator)
end (keyword)
struct (keyword)
; (operator)
mutator (keyword)
append (identifier)
((operator)
l (identifier)
: (operator)
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
x (identifier)
: (operator)
in (keyword)
ptr_type (identifier)
) (operator)
is (keyword)
tmp (identifier)
: (operator)
listel_type (identifier)
; (operator)
begin (keyword)

```
tmp (identifier)
:= (operator)
alloc (identifier)
( (operator)
listel_type (identifier)
) (operator)
; (operator)
tmp (identifier)
. (operator)
data (identifier)
:= (operator)
x (identifier)
; (operator)
tmp (identifier)
. (operator)
next (identifier)
:= (operator)
null (keyword)
; (operator)
if (keyword)
( (operator)
l (identifier)
. (operator)
tail (identifier)
!= (operator)
null (keyword)
) (operator)
then (keyword)
tmp (identifier)
. (operator)
prev (identifier)
:= (operator)
l (identifier)
. (operator)
tail (identifier)
; (operator)
l (identifier)
. (operator)
tail (identifier)
. (operator)
next (identifier)
:= (operator)
tmp (identifier)
; (operator)
l (identifier)
. (operator)
tail (identifier)
:= (operator)
```

```
tmp (identifier)
; (operator)
else (keyword)
tmp (identifier)
. (operator)
prev (identifier)
:= (operator)
null (keyword)
; (operator)
l (identifier)
. (operator)
head (identifier)
:= (operator)
tmp (identifier)
; (operator)
l (identifier)
. (operator)
tail (identifier)
:= (operator)
tmp (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
end (keyword)
append (identifier)
; (operator)
accessor (keyword)
data (identifier)
( (operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
return (keyword)
ptr_type (identifier)
is (keyword)
begin (keyword)
return (keyword)
e (identifier)
. (operator)
data (identifier)
; (operator)
end (keyword)
data (identifier)
; (operator)
/*
* Old and nesty version (that works)
```

```
*
function find(l: list_type; x: ptr_type) return listel_type is
tmp: listel_type;
begin
tmp := l.head;
while (tmp != null) loop
if (tmp.data = x) then
exit;
end if;
tmp := tmp.next;
end loop;
return tmp;
end find;
*/ (comment)
function (keyword)
find (identifier)
( (operator)
l (identifier)
: (operator)
list_type (identifier)
; (operator)
x (identifier)
: (operator)
ptr_type (identifier)
) (operator)
return (keyword)
listel_type (identifier)
is (keyword)
tmp (identifier)
: (operator)
listel_type (identifier)
; (operator)
begin (keyword)
tmp (identifier)
:= (operator)
l (identifier)
. (operator)
head (identifier)
; (operator)
loop (keyword)
exit (keyword)
when (keyword)
( (operator)
( (operator)
tmp (identifier)
= (operator)
null (keyword)
) (operator)
```

or (keyword)
((operator)
tmp (identifier)
. (operator)
data (identifier)
= (operator)
x (identifier)
) (operator)
) (operator)
; (operator)
tmp (identifier)
:= (operator)
tmp (identifier)
. (operator)
next (identifier)
; (operator)
end (keyword)
loop (keyword)
; (operator)
return (keyword)
tmp (identifier)
; (operator)
end (keyword)
find (identifier)
; (operator)
accessor (keyword)
head (identifier)
((operator)
l (identifier)
: (operator)
list_type (identifier)
) (operator)
return (keyword)
listel_type (identifier)
is (keyword)
begin (keyword)
return (keyword)
l (identifier)
. (operator)
head (identifier)
; (operator)
end (keyword)
head (identifier)
; (operator)
/*
* This routine performs a prefix insertion of an el with data x.
*/ (comment)
mutator (keyword)

insert (identifier)
((operator)
l (identifier)
: (operator)
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
/* list to insert within */ (comment)
e (identifier)
: (operator)
in (keyword)
out (keyword)
listel_type (identifier)
; (operator)
/* place to insert el before */ (comment)
x (identifier)
: (operator)
in (keyword)
ptr_type (identifier)
/* data of new el */ (comment)
) (operator)
is (keyword)
tmp (identifier)
: (operator)
listel_type (identifier)
; (operator)
begin (keyword)
tmp (identifier)
:= (operator)
alloc (identifier)
((operator)
listel_type (identifier)
) (operator)
; (operator)
tmp (identifier)
. (operator)
data (identifier)
:= (operator)
x (identifier)
; (operator)
tmp (identifier)
. (operator)
next (identifier)
:= (operator)
e (identifier)
; (operator)
tmp (identifier)

```
. (operator)
prev (identifier)
:= (operator)
e (identifier)
. (operator)
prev (identifier)
; (operator)
if (keyword)
( (operator)
e (identifier)
. (operator)
prev (identifier)
!= (operator)
null (keyword)
) (operator)
then (keyword)
e (identifier)
. (operator)
prev (identifier)
. (operator)
next (identifier)
:= (operator)
tmp (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
e (identifier)
. (operator)
prev (identifier)
:= (operator)
tmp (identifier)
; (operator)
if (keyword)
( (operator)
e (identifier)
= (operator)
l (identifier)
. (operator)
head (identifier)
) (operator)
then (keyword)
l (identifier)
. (operator)
head (identifier)
:= (operator)
tmp (identifier)
; (operator)
```

```
end (keyword)
if (keyword)
; (operator)
end (keyword)
insert (identifier)
; (operator)
accessor (keyword)
next (identifier)
( (operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
return (keyword)
listel_type (identifier)
is (keyword)
begin (keyword)
return (keyword)
e (identifier)
. (operator)
next (identifier)
; (operator)
end (keyword)
next (identifier)
; (operator)
mutator (keyword)
prepend (identifier)
( (operator)
l (identifier)
: (operator)
in (keyword)
out (keyword)
list_type (identifier)
; (operator)
x (identifier)
: (operator)
in (keyword)
data_type (identifier)
) (operator)
is (keyword)
tmp (identifier)
: (operator)
listel_type (identifier)
; (operator)
begin (keyword)
tmp (identifier)
:= (operator)
alloc (identifier)
```

```
( (operator)
listel_type (identifier)
) (operator)
; (operator)
tmp (identifier)
. (operator)
prev (identifier)
:= (operator)
null (keyword)
; (operator)
tmp (identifier)
. (operator)
data (identifier)
:= (operator)
x (identifier)
; (operator)
if (keyword)
( (operator)
l (identifier)
. (operator)
head (identifier)
!= (operator)
null (keyword)
) (operator)
then (keyword)
tmp (identifier)
. (operator)
next (identifier)
:= (operator)
l (identifier)
. (operator)
head (identifier)
; (operator)
l (identifier)
. (operator)
head (identifier)
. (operator)
prev (identifier)
:= (operator)
tmp (identifier)
; (operator)
l (identifier)
. (operator)
head (identifier)
:= (operator)
tmp (identifier)
; (operator)
else (keyword)
```

tmp (identifier)
. (operator)
next (identifier)
:= (operator)
null (keyword)
; (operator)
l (identifier)
. (operator)
head (identifier)
:= (operator)
tmp (identifier)
; (operator)
l (identifier)
. (operator)
tail (identifier)
:= (operator)
tmp (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
end (keyword)
prepend (identifier)
; (operator)
accessor (keyword)
prev (identifier)
((operator)
e (identifier)
: (operator)
listel_type (identifier)
) (operator)
return (keyword)
listel_type (identifier)
is (keyword)
begin (keyword)
return (keyword)
e (identifier)
. (operator)
prev (identifier)
; (operator)
end (keyword)
next (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)
is (keyword)
begin (keyword)
if (keyword)
( (operator)
e (identifier)
. (operator)
prev (identifier)
!= (operator)
null (keyword)
) (operator)
then (keyword)
e (identifier)
. (operator)
prev (identifier)
. (operator)
next (identifier)
:= (operator)
e (identifier)
. (operator)
next (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
if (keyword)
( (operator)
e (identifier)
. (operator)
next (identifier)
!= (operator)
null (keyword)
) (operator)
then (keyword)
e (identifier)
. (operator)
next (identifier)
. (operator)
prev (identifier)
```

```
:= (operator)
e (identifier)
. (operator)
prev (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
if (keyword)
( (operator)
e (identifier)
= (operator)
l (identifier)
. (operator)
head (identifier)
) (operator)
then (keyword)
l (identifier)
. (operator)
head (identifier)
:= (operator)
e (identifier)
. (operator)
next (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
if (keyword)
( (operator)
e (identifier)
= (operator)
l (identifier)
. (operator)
tail (identifier)
) (operator)
then (keyword)
l (identifier)
. (operator)
tail (identifier)
:= (operator)
e (identifier)
. (operator)
prev (identifier)
; (operator)
end (keyword)
if (keyword)
; (operator)
```

dealloc (identifier)
((operator)
e (identifier)
) (operator)
; (operator)
e (identifier)
:= (operator)
null (keyword)
; (operator)
end (keyword)
remove (identifier)
; (operator)
function (keyword)
size (identifier)
((operator)
l (identifier)
: (operator)
list_type (identifier)
) (operator)
return (keyword)
list_size_type (identifier)
is (keyword)
tmp (identifier)
: (operator)
listel_type (identifier)
; (operator)
n (identifier)
: (operator)
list_size_type (identifier)
; (operator)
begin (keyword)
n (identifier)
:= (operator)
0 (numeric literal)
; (operator)
tmp (identifier)
:= (operator)
l (identifier)
. (operator)
head (identifier)
; (operator)
while (keyword)
((operator)
tmp (identifier)
!= (operator)
null (keyword)
) (operator)
loop (keyword)

```
tmp (identifier)
:= (operator)
tmp (identifier)
. (operator)
next (identifier)
; (operator)
n (identifier)
:= (operator)
n (identifier)
+ (operator)
1 (numeric literal)
; (operator)
end (keyword)
loop (keyword)
; (operator)
return (keyword)
n (identifier)
; (operator)
end (keyword)
size (identifier)
; (operator)
accessor (keyword)
tail (identifier)
( (operator)
l (identifier)
: (operator)
list_type (identifier)
) (operator)
return (keyword)
listel_type (identifier)
is (keyword)
begin (keyword)
return (keyword)
l (identifier)
. (operator)
tail (identifier)
; (operator)
end (keyword)
head (identifier)
; (operator)
end (keyword)
list (identifier)
; (operator)
```

Grading

Your lexer will be built using Code::Blocks Version 16.01 as installed on NIC lab workstations. Your lexer will be tested against 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
widget.cci
widget.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 produces correct results for each of the files listed above, you will earn 100% of the total possible points on this assignment.

If your lexer does not compile using Code::Blocks Version 16.01 on an NIC standard lab workstation, 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 a score of 0 for this homework.

Your lexer must be named CS210_MidtermF19.exe and take its arguments from the command-line. Your lexer will be executed against all files shown above as follows:

```
CS210_MidtermF19 hello_world.ccx
CS210_MidtermF19 list.cci
```

More Requirements
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 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
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 bullet proof, 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. If you take this approach, your lexer should end up containing a very small number of states.

Submitting Your Midterm
Your homework must be submitted using the Canvas website for the course. You must turn in only the source code for your lexer.

If your lexer is contained within a single source code file, you should name your file "CS210_MidtermF19.cpp" (without quotes).

If your lexer is composed of multiple source code files, you should name the file containing the `main` function "CS210_MidtermF19.cpp" and header and external .cpp files should be named according to the classes they contain, for example. Package multiple source code files into a single archive using a compression program that creates a .zip file and name it "CS210_MidtermF19.zip" (without quotes).