University of Central Florida
Department Computer Science
COP3402: Systems Software
Fall 2022
Homework #2 (Lexical Analyzer)

Assignment Overview:

In this assignment, you will implement a lexical analyzer for the tiny language PL/0. Lexical analyzers take an input program, identify the tokens, and check for lexical errors. Tokens are the basic units of the programming language. PL/0 has four types of tokens: keywords, identifiers, numbers, and symbols. Lexical errors are mistakes at the token level, PL/0 has five: identifier length, number length, invalid identifiers, invalid symbols, and invalid identifier names.

You will implement the function

*lexeme *lex_analyze(int list_flag, char *input);*

within the project file **lex.c**.

The char array, **input**, holds the entirety of the input file. Your program will return the tokens as an array of type lexeme. The lexeme struct is defined in compiler.h, and a copy is included here:

```
typedef struct lexeme {
     token_type type;
     char identifier_name[12];
     int number_value;
     int error_type;
} lexeme;
```

Every token has a type. The token_type typedef is also in compiler.h and included in Appendix B. It is an enum definition. Each token type accepted in PL/0 is listed in the enum. An enum is a C structure that allows the definition of multiple integer constants. It's especially useful for enumeration: if you define the first entry, each successive entry will have a value one greater than the last.

Identifier tokens use the field identifier_name and number tokens use the field number_value. They are the only types to use these fields.

When an error is encountered in the input file, it is added to lexeme array before the lexical analyzer moves past it. This way all the errors can be printed where they occur in the

program. If your lexical analyzer finds any error in the program, it should return NULL instead of the lexeme array. This lets the compile driver know to stop instead of proceeding to the parser. Error lexemes should have their type set to -1.

The input can contain whitespace including space characters and control characters (‘\n’, ‘\t’, ‘\r’, etc). Your program should ignore these. Control characters can be identified with the library function iscntrl() from ctype.h.

The input may also contain comments. Comments begin with ‘?’ and end at the next newline character (‘\n’) or EOF (‘\0’). Comments should be ignored.

Implementing the Lexical Analyzer:

Variables:

Generally speaking you will need two indexing variables: one for the input program and one for the lexeme array. The lexeme array should be ARRAY_SIZE. You will also need an error flag so your program knows whether to return the error free lexeme array or NULL. Additionally you’ll need a buffer char array and an indexer for it. You can have separate buffers for different token types or use the same one for all of them. The maximum buffer size should be the MAX_IDENT_LENGTH (11) + 1.

Functions:

You may implement as many functions as you desire in addition to lex_analyze(). We provide a printing function (*void print_lexeme_list(lexeme *list, int list_end)*) which you should call after you finish processing the input if the list_flag argument is equal to 1. We also provide the function *int keyword_check(char buffer[])* which will return the token_type value for the buffer. If the buffer is “main” or “null” (which are invalid identifier values), it returns -1. If the buffer is not a keyword, “main”, or “null”, it will return the identifier token_type.

The keyword_check function uses strcmp() from string.h, so it is necessary to mark the end of your buffer before you call it. This is a really common mistake students make: forgetting to mark the end of your buffer. ‘Marking the end of your buffer’ means adding the end of string character (‘\0’). This tells all the string.h functions where the end of the string is. It is necessary for them to work. Ex. if you buffer has three letters/numbers, then buffer[3] = ‘\0’.

Helpful Library Functions:

The ctype.h header has many helpful functions for processing characters. All of these functions take a single char as argument and will return 1 for true and 0 for false.

`isalpha()` - returns whether the char is an uppercase or lowercase letter
`isdigit()` - returns whether the char is a digit
`isalnum()` - returns whether the char is an uppercase or lowercase letter or a digit
`isspace()` - returns whether the char is a space or a whitespace control character

Overall Structure:

Your program should work char by char, while you haven't reached the end of the input array:

```
while (input[index] != '\0')
```

You can then use a if-else-if structure to determine which token type category the current token falls under:

- Identifier or keyword - these start with a letter (uppercase or lowercase)
- Number - starts with a digit
- Comment - starts with a '?'
- Whitespace
- Symbol - this will be your final else

Identifiers and keywords should be processed char by char.

1. Add chars to your buffer while they are letters or digits and you have space in your buffer (up to MAX_IDENT_LENGTH).
2. After you stop adding chars to your buffer, if the next char is a letter or digit (meaning you stopped because you hit MAX_IDENT_LENGTH), you've found ERR_IDENT_LENGTH. You should move past any remaining letters and digits before moving on to the next token.
3. Call the keyword_check function to get the type for your token.
    A. If keyword_check returned identifier, copy the buffer to the identifer_name field
    B. If keyword_check returned -1, you've found ERR_INVALID_IDENT_NAME

Numbers should also be processed char by char.

1. Add chars to your buffer while they are digits and you have space in your buffer (up to MAX_NUM_LENGTH).
2. After you stop adding chars to your buffer, if the next char is a digit (meaning you stopped because you hit MAX_NUM_LENGTH), you've found ERR_NUM_LENGTH. You should move past any remaining digits or letters before moving on to the next token.

3. If the next char is a letter (meaning you stopped because you didn't have more digits), you've found ERR_INVALID_IDENT. You should move past any remaining digits or letters before moving on to the next token.
4. Finally add the number to the lexeme array.

Comments begin with '?'. Move past characters until you reach a '\n' or the end of the input ('\0'). Whitespace characters should be moved past as well.

You can process symbols with an if-else-if or switch structure. You should not use string.h functions for symbols. There are two types of symbols: single character symbols and double character symbols. Single character symbols are straightforward: add them to the lexeme array and move on. The single character symbols in PL/0 are: **. - ; { } + * / ( )** The remaining symbols are double character symbols: **:= == < <= > >= !=** The symbols **:= == !=** must have an equal sign immediately following the first character. A **: =** or **!** by itself is an ERR_INVALID_SYMBOL. For **<** and **>**, if they are followed by an **=**, they are **<=** or **>=** respectively, but if they are by themselves, they are still valid tokens. Any other symbol is ERR_INVALID_SYMBOL.

A Note on Errors:

Input files may not be grammatically valid PL/0 code. The lexical analyzer only concerns itself with lexical errors. Questions often come up about the priority of error identification. This is best illustrated with some examples.

Input: `abcdefghijk123456`

Since the token starts with a letter, you process letters and digits until you reach the 12th consecutive letter or digit ('1'). At this point you know you have ERR_IDENT_LENGTH. Move past the characters until you reach one that's not a letter or digit.

Input: `123456abcdef`

Since the token starts with a number, you process digits until you reach the 6th consecutive digit ('6'). At this point you know you have ERR_NUMBER_LENGTH. Move past the remaining characters until you reach one that's not a letter or a digit. The number length error takes priority over the invalid identifier error and identifier length errors because it occurs first.

Input: `12345abcdefgh`

Since the token starts with a number, you process digits until you reach a character that's not a digit ('a'). Since the next character is a letter, you have ERR_INVALID_IDENT. Move past the remaining characters until you reach one that's not a letter or a digit. The invalid identifier error takes priority over the identifier length error because it occurs first.

Input: `const12`

This identifier may contain a keyword, but since there isn't a whitespace or symbol between the 't' and the '1', this is tokenized as an identifier.

Testing Your Program:

As explained in the project overview of HW1, your assignment is one part of a multi-file C program. We understand that this is the first time many of you are working with a multi-file C project. To test your program, you need to upload all of the project files to Eustis3:

- driver.c - the driver program; no change from HW1
- compiler.h - the header file; no change from HW1
- vm.o and parser.o - compiled versions of our implementation of vm.c and parser.c. parser.o is the same as from HW1. Since this assignment is about lex.c, we don't want you to worry about your previous work in vm.c. We will use vm.o for grading.
- lex.c - where you'll write your code
- Any input and output files you want to use.

To compile, use the command "`gcc driver.c lex.c parser.o vm.o`"

To execute, use the command "`./a.out input.txt -l`"

- Replace input.txt with the name of whatever input file you're using
- "`-l`" is a tag for the driver so your program knows to print the lexical analyzer output. This sets list_flag to true. It is a lowercase L.

To compare your output to correct output, use the command "`./a.out input.txt -l > output.txt`" to generate your output. Then compare your output to correct output with the command "`diff -w -B your_output.txt correct_output.txt`" This will print out any differences between the two files. If the command doesn't print anything, then the two files are exactly the same (the desired outcome).

Making Your Own Test Cases:

We're providing you with two test cases to use when developing your program, but we will be using different test cases to grade, so you may want to write your own test cases. Input files are written in PL/0, which is fairly simple. We've included the grammar in Appendix E. To get the correct output for your test cases, we're providing the "magic" file. "magic" is a compiled version of our implementation of the project. It works like a.out. You must be on Eustis3 to run it. You may get an error with permissions, use the command "`chmod +x magic`" if this happens. To get the lexical analyzer output for you input program, use the command "`./magic input.txt -l`". If your input program was didn't have any lexical errors, but does have grammar errors, the project will print them. To write the output to a file, add "`> output.txt`" to the end of the command.

Administrative Guidelines:

1. The lexical analyzer must be written in C and must run on Eustis3. If it runs on your PC but not on Eustis3, for us it does not run. If you need help setting up your computer to access Eustis3, reach out to a TA or Dr. Aedo.
2. Do not change the token_type values or lexeme struct.
3. Do not try to implement parser.c or resubmit vm.c
4. Do not add a main function to lex.c
5. Include comments in your program.
6. If you submit a program from another semester or section or from the internet, this is considered plagiarism. We regard this as cheating. At a minimum, you will receive a zero on this assignment.
7. Submit to Webcourses:
    a) The source code of your lexical analyzer which should be named "lex.c"
    b) Student names should be written in the header comment of the source code file and in the comments of the submission.

Rubric:

15	Compiles
10	Produces lines of meaningful output before segfaulting or looping infinitely
5	Well commented source code
5	Program processes at least one error type correctly Continues processing after the error Returns NULL to the driver
5	ERR_IDENTIFIER_LENGTH
5	ERR_INVALID_IDENT_NAME
5	ERR_NUMBER_LENGTH
5	ERR_INVALID_IDENTIFIER
5	ERR_INVALID_SYMBOL
5	Identifiers
5	Keywords
5	Numbers
5	Comments
10	Whitespace
5	Single character symbols
5	Double character symbols

Appendix A: Example Program

Input Program ("megatest.txt"***):***

```
.const==xyz3:=71;var!=procedure<call<=begin>end>=if+then-
while*do/read(write)def{}return
? now i'll do the errors, notice how the invalid symbols are
ignored when they're in the comment %
main
null
mainnull
123456
123a
abcedef123456
123456abcdef
# ? final comment
```

Lexical Analyzer Output ("./a.out megatest.txt -l"***)/(***"megatest_out.txt"***):***

```
Lexeme List:
lexeme          token type
          .     17
      const     3
         ==     23
       xyz3     1
         :=     18
         71     2
          ;     20
        var     4
         !=     24
  procedure     5
          <     25
       call     6
         <=     26
      begin     7
          >     27
        end     8
         >=     28
         if     9
          +     29
       then     10
          -     19
      while     11
          *     30
         do     12
```

```
          /    31
       read    13
          (    32
      write    14
          )    33
        def    16
          {    21
          }    22
     return    15
Lexical Analyzer Error: identifiers cannot be named 'null' or
'main'
Lexical Analyzer Error: identifiers cannot be named 'null' or
'main'
   mainnull    1
Lexical Analyzer Error: maximum number length is 5
Lexical Analyzer Error: identifiers cannot begin with digits
Lexical Analyzer Error: maximum identifier length is 11
Lexical Analyzer Error: maximum number length is 5
Lexical Analyzer Error: invalid symol
```

Please note we are aware of the typo in the invalid symbol error message. We're leaving it since correcting it would require updating all of the precompiled files and it's a very minor issue. Do not correct it in your implementation.

Appendix B:

Declaration of Token Types (from compiler.h):

```
typedef enum token_type {
        identifier = 1, number, keyword_const, keyword_var, keyword_procedure,
        keyword_call, keyword_begin, keyword_end, keyword_if, keyword_then,
        keyword_while, keyword_do, keyword_read, keyword_write, keyword_return,
        keyword_def, period, assignment_symbol, minus, semicolon,
        left_curly_brace, right_curly_brace, equal_to, not_equal_to, less_than,
        less_than_or_equal_to, greater_than, greater_than_or_equal_to, plus, times,
        division, left_parenthesis, right_parenthesis
} token_type;
```

Token Types:

identifier	1	identifier
number	2	number
keyword_const	3	“const”
keyword_var	4	“var”
keyword_procedure	5	“procedure"
keyword_call	6	“call”
keyword_begin	7	“begin”
keyword_end	8	“end”
keyword_if	9	“if”
keyword_then	10	“then”
keyword_while	11	“while"
keyword_do	12	“do”
keyword_read	13	“read"
keyword_write	14	“write”
keyword_return	15	“return"
keyword_def	16	“def"
period	17	.
assignment_symbol	18	:=
minus	19	-
semicolon	20	;
left_curly_brace	21	{
right_curly_brace	22	}
equal_to	23	==
not_equal_to	24	!=
less_than	25	<
less_than_or_equal_to	26	<=

greater_than	27	>
greater_than_or_equal_to	28	>=
plus	29	+
times	30	*
division	31	/
left_parenthesis	32	(
right_parenthesis	33	)

Support function for keywords (from lex.c skeleton):

```
int keyword_check(char buffer[])
{
     if (strcmp(buffer, "const") == 0)
          return keyword_const;
     else if (strcmp(buffer, "var") == 0)
          return keyword_var;
     else if (strcmp(buffer, "procedure") == 0)
          return keyword_procedure;
     else if (strcmp(buffer, "call") == 0)
          return keyword_call;
     else if (strcmp(buffer, "begin") == 0)
          return keyword_begin;
     else if (strcmp(buffer, "end") == 0)
          return keyword_end;
     else if (strcmp(buffer, "if") == 0)
          return keyword_if;
     else if (strcmp(buffer, "then") == 0)
          return keyword_then;
     else if (strcmp(buffer, "while") == 0)
          return keyword_while;
     else if (strcmp(buffer, "do") == 0)
          return keyword_do;
     else if (strcmp(buffer, "read") == 0)
          return keyword_read;
     else if (strcmp(buffer, "write") == 0)
          return keyword_write;
     else if (strcmp(buffer, "def") == 0)
          return keyword_def;
     else if (strcmp(buffer, "return") == 0)
          return keyword_return;
     else if (strcmp(buffer, "main") == 0)
          return -1;
     else if (strcmp(buffer, "null") == 0)
```

```
          return -1;
     else
          return identifier;
}
```

Print function, use just before returning to the driver, only if list_flag is true (1):

```
void print_lexeme_list(lexeme *list, int list_end)
{
     int i;
     printf("Lexeme List: \n");
     printf("lexeme\t\ttoken type\n");
     for (i = 0; i < list_end; i++)
     {
          // not an error
          if (list[i].type != -1)
          {
               switch (list[i].type)
               {
                    case identifier :
                         printf("%11s\t%d\n",
list[i].identifier_name, identifier);
                         break;
                    case number :
                         printf("%11d\t%d\n",
list[i].number_value, number);
                         break;
                    case keyword_const :
                         printf("%11s\t%d\n", "const",
keyword_const);
                         break;
                    case keyword_var :
                         printf("%11s\t%d\n", "var",
keyword_var);
                         break;
                    case keyword_procedure :
                         printf("%11s\t%d\n", "procedure",
keyword_procedure);
                         break;
                    case keyword_call :
                         printf("%11s\t%d\n", "call",
keyword_call);
                         break;
                    case keyword_begin :
```

```
                    printf("%11s\t%d\n", "begin",
keyword_begin);
                    break;
                case keyword_end :
                    printf("%11s\t%d\n", "end",
keyword_end);
                    break;
                case keyword_if :
                    printf("%11s\t%d\n", "if",
keyword_if);
                    break;
                case keyword_then :
                    printf("%11s\t%d\n", "then",
keyword_then);
                    break;
                case keyword_while :
                    printf("%11s\t%d\n", "while",
keyword_while);
                    break;
                case keyword_do :
                    printf("%11s\t%d\n", "do",
keyword_do);
                    break;
                case keyword_read :
                    printf("%11s\t%d\n", "read",
keyword_read);
                    break;
                case keyword_write :
                    printf("%11s\t%d\n", "write",
keyword_write);
                    break;
                case keyword_def :
                    printf("%11s\t%d\n", "def",
keyword_def);
                    break;
                case keyword_return :
                    printf("%11s\t%d\n", "return",
keyword_return);
                    break;
                case period :
                    printf("%11s\t%d\n", ".", period);
                    break;
                case assignment_symbol :
```

```
                         printf("%11s\t%d\n", ":=",
assignment_symbol);
                         break;
                    case minus :
                         printf("%11s\t%d\n", "-", minus);
                         break;
                    case semicolon :
                         printf("%11s\t%d\n", ";",
semicolon);
                         break;
                    case left_curly_brace :
                         printf("%11s\t%d\n", "{",
left_curly_brace);
                         break;
                    case right_curly_brace :
                         printf("%11s\t%d\n", "}",
right_curly_brace);
                         break;
                    case equal_to :
                         printf("%11s\t%d\n", "==",
equal_to);
                         break;
                    case not_equal_to :
                         printf("%11s\t%d\n", "!=",
not_equal_to);
                         break;
                    case less_than :
                         printf("%11s\t%d\n", "<",
less_than);
                         break;
                    case less_than_or_equal_to :
                         printf("%11s\t%d\n", "<=",
less_than_or_equal_to);
                         break;
                    case greater_than :
                         printf("%11s\t%d\n", ">",
greater_than);
                         break;
                    case greater_than_or_equal_to :
                         printf("%11s\t%d\n", ">=",
greater_than_or_equal_to);
                         break;
                    case plus :
```

```
                        printf("%11s\t%d\n", "+", plus);
                        break;
                    case times :
                        printf("%11s\t%d\n", "*", times);
                        break;
                    case division :
                        printf("%11s\t%d\n", "/", division);
                        break;
                    case left_parenthesis :
                        printf("%11s\t%d\n", "(",
left_parenthesis);
                        break;
                    case right_parenthesis :
                        printf("%11s\t%d\n", ")",
right_parenthesis);
                        break;
                    default :
                        printf("Implementation Error:
unrecognized token type\n");
                        break;
               }
          }
          // errors
          else
          {
               switch (list[i].error_type)
               {
                    case ERR_IDENT_LENGTH :
                        printf("Lexical Analyzer Error:
maximum identifier length is 11\n");
                        break;
                    case ERR_NUM_LENGTH :
                        printf("Lexical Analyzer Error:
maximum number length is 5\n");
                        break;
                    case ERR_INVALID_IDENT :
                        printf("Lexical Analyzer Error:
identifiers cannot begin with digits\n");
                        break;
                    case ERR_INVALID_SYMBOL :
                        printf("Lexical Analyzer Error:
invalid symol\n");
                        break;
```

```
                    case ERR_INVALID_IDENT_NAME :
                         printf("Lexical Analyzer Error:
identifiers cannot be named 'null' or 'main'\n");
                         break;
                    default :
                         printf("Implementation Error:
unrecognized error type\n");
                         break;
               }
          }
     }
     printf("\n");
}
```