Homework 2: Lexical Analyzer for PL/0

See Webcourses and the syllabus for due dates.

1 Purpose

In this homework your team [Collaborate] will implement a lexical analyzer for the PL/0 language [UseConcepts] [Build], which is defined below.

2 Directions

1. (100 points) Implement and submit your code for the lexer and the output of our tests, as described in the rest of this document.

   For the implementation, your code must be written in 2017 ANSI standard C and must compile with gcc and run correctly on Eustis, when compiled with the `-std=c17` flag.[1] We recommend using the gcc flags `-std=c17 -Wall` and fixing all warnings before turning in this assignment.

You are *not* allowed to submit code generated by a lexical analyzer generator (such as `lext`) for this homework.

Similarly, since we want you to learn how to implement regular expression matching yourself, you are *not* allowed to submit code that uses a regular expression matching library (such as `regcomp` and `regexec`).

Note that we will randomly ask questions of students in the team to ensure that all team members understand their solution; there will be penalty of up to 10 points (deducted from all team members' scores for that assignment) if some team member does not understand some part of the solution to an assignment.

3 What to Turn In

Your team must submit on Webcourses a single zip file containing:

1. A plain text file named `sources.txt` that lists the names of all the `.c` files needed to compile your program, all on one line separated by spaces. For example, if you have files named `token.c`, `lexer.c`, `lexer_output.c`, `main.c`, and `utilities.c`, then your file `sources.txt` would look contain (only) the following line of text naming these files:

   ```
   token.c lexer.c lexer_output.c main.c utilities.c
   ```

   (The order of these names does not matter.)

   If there is only one file in your program, then put its name in your `sources.txt` file.

2. Each source file that is needed to compile your lexer with `gcc -std=c17` on Eustis, including all needed header files (if there are any).

3. The output of our tests, using the output formatting specified below. These are the `.myo` files created by the provided `Makefile`.

[1] See this course's resources page for information on how to access Eustis.

You can use the Unix command

```
make submission.zip
```

on Eustis to create a zip file that has all these files in it, after you have created your `sources.txt` file and run our tests (using the command `make check-outputs`) to create the `.myo` files.

We will take some points off for: code that does not work properly, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. Avoid duplicating code by using helping functions, or library functions. It is a good idea to check your code for these problems before submitting.

Don't hesitate to contact the staff if you are stuck at some point. Your code should compile properly; if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.

4 What to Read

You should read *Systems Software: Essential Concepts* (by Montagne) in which we recommend reading chapter 5 (pages 81-91).

5 Overview

In this assignment, you will implement a lexical analyzer for the PL/0 language, whose context-free grammar is defined in Figure 1 and whose lexical grammar is defined in Figure 2.

The following subsections specify the interface between the Unix operating system (as on Eustis) and the lexer as a program.

5.1 Inputs

The lexer is passed a single file name as its only command line argument; this file should be the name of a (readable) text file containing the program that the lexer should produce tokens for. Note that this program file is not necessarily legal according to the grammar for PL/0. (This homework is not about checking the context-free grammar of programs, it is only checking the lexical grammar.) For example, if the file name argument is `hw2-test1.pl0` (and both the lexer executable, `./lexer`, and the `hw2-test1.pl0` file are in the current directory), then the lexer should work on `hw2-test1.pl0` and send all its output to the file `hw2-test1.myo` by executing the following command in the Unix shell (e.g., at the command prompt on Eustis):

```
./lexer hw2-test1.pl0 >hw2-test1.myo 2>&1
```

The same thing can also be accomplished using the `make` command on Unix:

```
make hw2-test1.myo
```

5.2 Outputs

The lexer prints its normal output, as specified below, to standard output (`stdout`). However, all error messages (e.g., for unreadable files or illegal tokens) should be sent to standard error output (`stderr`). See subsection 7.4 for more details about error messages.

⟨program⟩ ::= ⟨block⟩ **.**

⟨block⟩ ::= ⟨const-decls⟩ ⟨var-decls⟩ ⟨proc-decls⟩ ⟨stmt⟩

⟨const-decls⟩ ::= **{**⟨const-decl⟩**}**
⟨const-decl⟩ ::= **const** ⟨const-def⟩ **{**⟨comma-const-def⟩**}** **;**
⟨const-def⟩ ::= ⟨ident⟩ = ⟨number⟩
⟨comma-const-def⟩ ::= **,** ⟨const-def⟩

⟨var-decls⟩ ::= **{**⟨var-decl⟩**}**
⟨var-decl⟩ ::= **var** ⟨idents⟩ **;**
⟨idents⟩ ::= ⟨ident⟩ **{**⟨comma-ident⟩**}**
⟨comma-ident⟩ ::= **,** ⟨ident⟩

⟨proc-decls⟩ ::= **{**⟨proc-decl⟩**}**
⟨proc-decl⟩ ::= **procedure** ⟨ident⟩ **;** ⟨block⟩ **;**

⟨stmt⟩ ::= ⟨ident⟩ **:=** ⟨expr⟩
| **call** ⟨ident⟩
| **begin** ⟨stmt⟩ **{**⟨semi-stmt⟩**}** **end**
| **if** ⟨condition⟩ **then** ⟨stmt⟩ **else** ⟨stmt⟩
| **while** ⟨condition⟩ **do** ⟨stmt⟩
| **read** ⟨ident⟩
| **write** ⟨expr⟩
| **skip**
⟨semi-stmt⟩ ::= **;** ⟨stmt⟩
⟨empty⟩ ::=

⟨condition⟩ ::= **odd** ⟨expr⟩
| ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨expr⟩ ::= ⟨term⟩ **{**⟨add-sub-term⟩**}**
⟨add-sub-term⟩ ::= ⟨add-sub⟩ ⟨term⟩
⟨add-sub⟩ ::= ⟨plus⟩ | ⟨minus⟩
⟨term⟩ ::= ⟨factor⟩ **{**⟨mult-div-factor⟩**}**
⟨mult-div-factor⟩ ::= ⟨mult-div⟩ ⟨factor⟩
⟨mult-div⟩ ::= ⟨mult⟩ | ⟨div⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨sign⟩ ⟨number⟩ | **(** ⟨expr⟩ **)**
⟨sign⟩ ::= ⟨plus⟩ | ⟨minus⟩ | ⟨empty⟩

Figure 1: Context-free grammar of PL/0. The grammar uses a `terminal font` for terminal symbols, and a **`bold terminal font`** for reserved words. As in EBNF, curly brackets **{**x**}** means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Also note that an **`else`** clause is required in each if-statement.

⟨ident⟩ ::= ⟨letter⟩ **{**⟨letter-or-digit⟩**}**
⟨letter⟩ ::= `a` | `b` | `...` | `y` | `z` | `A` | `B` | `...` | `Y` | `Z`
⟨number⟩ ::= ⟨digit⟩ **{**⟨digit⟩**}**
⟨digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨letter-or-digit⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨rel-op⟩ ::= `=` | `<>` | `<` | `<=` | `>` | `>=`
⟨plus⟩ ::= `+`
⟨minus⟩ ::= `-`
⟨mult⟩ ::= `*`
⟨div⟩ ::= `/`

⟨ignored⟩ ::= ⟨blank⟩ | ⟨tab⟩ | ⟨vt⟩ | ⟨formfeed⟩ | ⟨eol⟩ | ⟨comment⟩
⟨blank⟩ ::= "A space character (ASCII 32)"
⟨tab⟩ ::= "A horizontal tab character (ASCII 9)"
⟨vt⟩ ::= "A vertical tab character (ASCII 11)"
⟨formfeed⟩ ::= "A formfeed character (ASCII 12)"
⟨newline⟩ ::= "A newline character (ASCII 10)"
⟨cr⟩ ::= "A carriage return character (ASCII 13)"
⟨eol⟩ ::= ⟨newline⟩ | ⟨cr⟩ ⟨newline⟩
⟨comment⟩ ::= ⟨pound-sign⟩ **{**⟨non-nl⟩**}** ⟨newline⟩
⟨pound-sign⟩ ::= `#`
⟨non-nl⟩ ::= "Any character except a newline"

Figure 2: Lexical grammar of PL/0. The grammar uses a `terminal font` for terminal symbols. Note that all ASCII letters are included in the production for ⟨letter⟩. Again, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Some character classes are described in English, these are described in a Roman font between double quotation marks (" and "). Note that all characters matched by the nonterminal ⟨ignored⟩ are ignored by the lexer (except that each instance of ⟨eol⟩ ends a line).

5.3 Exit Code

When the lexer finishes without any errors, it should exit with a zero error code (which indicates success on Unix). However, when the lexer encounters an error it should terminate with a non-zero exit code (which indicates failure on Unix).

5.4 Tokens

The tokens that the lexer returns are defined in the header file `token.h` (see Figure 4). The correspondence between the token types and strings those represent is shown in Figure 3.

Enum value	Enum name	The token's text
0	periodsym	"`.`"
1	constsym	"**`const`**"
2	semisym	"`;`"
3	commasym	"`,`"
4	varsym	"**`var`**"
5	procsym	"**`procedure`**"
6	becomessym	"`:=`"
7	callsym	"**`call`**"
8	beginsym	"**`begin`**"
9	endsym	"**`end`**"
10	ifsym	"**`if`**"
11	thensym	"**`then`**"
12	elsesym	"**`else`**"
13	whilesym	"**`while`**"
14	dosym	"**`do`**"
15	readsym	"**`read`**"
16	writesym	"**`write`**"
17	skipsym	"**`skip`**"
18	oddsym	"**`odd`**"
19	lparensym	"`(`"
20	rparensym	"`)`"
21	identsym	(⟨ident⟩, e.g., "`temp1`")
22	numbersym	(⟨number⟩, e.g., "`3402`")
23	eqsym	"="
24	neqsym	"<>"
25	lessym	"<"
26	leqsym	"<="
27	gtrsym	">"
28	geqsym	">="
29	plussym	"+"
30	minussym	"−"
31	multsym	"⋆"
32	divsym	"/"
33	eofsym	(end-of-file)

Figure 3: Token types and the strings of characters that they represent (the quotation marks (" and ") are not part of these strings of characters. See the `token.h` file (Figure 4) for details. The reserved words are written in a **`bold terminal font`**.

```
/* $Id: token.h,v 1.7 2023/02/01 17:14:05 leavens Exp leavens $ */
#ifndef _TOKEN_H
#define _TOKEN_H

#define MAX_IDENT_LENGTH 255

// types of tokens
typedef enum {
    periodsym, constsym, semisym, commasym,
    varsym, procsym, becomessym, callsym, beginsym, endsym,
    ifsym, thensym, elsesym, whilesym, dosym,
    readsym, writesym, skipsym,
    oddsym, lparensym, rparensym,
    identsym, numbersym,
    eqsym, neqsym, lessym, leqsym, gtrsym, geqsym,
    plussym, minussym, multsym, divsym,
    eofsym
} token_type;

// information about each token
typedef struct token {
    token_type typ;
    const char *filename;
    unsigned int line;
    unsigned int column;
    char *text; // non-NULL, if applicable
    short int value; // when typ==numbersym, its value
} token;

// Return the name of the token_type enum
// corresponding to the given token_type value
extern const char *ttyp2str(token_type ttyp);

#endif
```

Figure 4: The `token.h` file that defines the token types and struct for tokens.

6 Lexer API

Tokens returned by the lexer must be elements of the type `token` defined in the `token.h` file provided in the `hw2-tests.zip` file in the course homeworks directory.

The lexer you are to implement will have a stream-like API, supporting the following functions, as declared in the provided file `lexer.h`:

```
/* $Id: lexer.h,v 1.2 2023/01/31 06:45:02 leavens Exp $ */
#ifndef _LEXER_H
#define _LEXER_H
#include <stdbool.h>
#include "token.h"

// Requires: fname != NULL
// Requires: fname is the name of a readable file
// Initialize the lexer and start it reading
// from the given file name
extern void lexer_open(const char *fname);

// Close the file the lexer is working on
// and make this lexer be done
extern void lexer_close();

// Is the lexer's token stream finished
// (either at EOF or not open)?
extern bool lexer_done();

// Requires: !lexer_done()
// Return the next token in the input file,
// advancing in the input
extern token lexer_next();

// Requires: !lexer_done()
// Return the name of the current file
extern const char *lexer_filename();

// Requires: !lexer_done()
// Return the line number of the next token
extern unsigned int lexer_line();

// Requires: !lexer_done()
// Return the column number of the next token
extern unsigned int lexer_column();
#endif
```

A few notes about these functions to add to the comments in the `lexer.h` file (above):

- For `lexer_open`, if the file named cannot be read (e.g., is not readable), then an informative error message is issued to `stderr` (see subsection 7.4) and the program exits with a non-zero error code.
- `lexer_done()`, the lexer is "done," so this function returns true, when the lexer's file has been closed, encountered an error, or is finished reading (has returned the end-of-file token).
- `lexer_next()`, "advancing in the input" means that the characters in the file corresponding to the token returned (and any ignored whitespace characters and comments before it) have been consumed by the lexer and will not be read again.

7 Output

The output consists of a listing of the tokens read from the input file, as printed by the code provided in `lexer_output.c`, which is provided (along with a header file `lexer_output.h`) in the `hw2-tests.zip`

file in the course homeworks directory.

7.1 A Simple Example

Consider the following input in the file `hw2-test0.pl0`, (note that the suffix is lowercase 'P', lowercase 'L', and the numeral zero, i.e., '0') which is included in the `hw2-tests.zip` file in the course homeworks directory.

```
var x;
x := 3
```

This produces the output found in the following file (`hw2-test0.out`):

```
Tokens from file hw2-test0.pl0
Number Name       Line Column Text/Value
4      varsym     1    1      "var"
21     identsym   1    5      "x"
2      semisym    1    6      ";"
21     identsym   2    1      "x"
6      becomessym 2    3      ":="
22     numbersym  2    6      3
33     eofsym     3    1
```

7.2 Provided Driver

We provide a driver, found in `lexer_output.c` to run tests with the proper output formatting. After your program opens the given file in the lexer, by calling `lexer_open`, your program should call `lexer_output` to run the lexer. The `lexer_output` function asks for each token (in the lexer's file), until the lexer is done, and it also prints the tokens in a standard format.

More extensive tests are found in the files named `hw2-test*.pl0`, where `*` is replaced by a number (or letter). The expected output of each test is found in a file named the same as the test input but with the suffix `.out`. For example, the expected output of `hw2-test3.pl0` is in the file `hw2-test3.out`.

You can check your own lexer by running the tests using the Unix command on Eustis:

```
make check-outputs
```

Running the above command will generate files with the suffix `.myo`; for example your output from test `hw2-test3.pl0` will be put into `hw2-test3.myo`.

7.3 Do Not Change the Provided Files

You must not change any of the provided `.h` or `.c` files. You must use the provided files in your program.

7.4 Errors that Must be Detected

Your code must detect the following errors:

1. An ⟨ident⟩ has more than `MAX_IDENT_LENGTH` characters, where `MAX_IDENT_LENGTH` is defined in the provided `token.h` file.

2. A number's value is too large to be contained in a C short int (use `SHRT_MIN` and `SHRT_MAX` from the standard header file `limits.h` to determine if the number's value is outside the range that the implementation of C allows).

3. A character in the input is not one of the characters permitted by the lexical grammar (i.e., the character cannot be part of a token and is not one of the recognized whitespace characters). However, note that any character is allowed inside a comment.

4. The input ends during a comment (i.e., a comment was started but not finished when an end-of-file was seen reading from the input file).

Error messages sent to `stderr` should start with a file name, a colon, a space and the line and column numbers, followed by a space. Use the provided function `lexical_error` (found in `utilities.c`) to produce such error messages. See the header file `utilities.h` below for the interface:

```
/* $Id: utilities.h,v 1.2 2023/01/31 08:32:17 leavens Exp $ */
#ifndef _UTILITIES_H
#define _UTILITIES_H

// Format a string error message and print it using perror (for an OS error)
// then exit with a failure code, so a call to this does not return.
extern void bail_with_error(const char *fmt, ...);

// Print a lexical error message to stderr
// starting with the filename, a colon, the line number, a comma
// the column number, a colon, and then the message.
// Output goes to stderr and then an exit with a failure code,
// so a call to this function does not return.
extern void lexical_error(const char *filename, unsigned int line,
                          unsigned int column, const char *fmt, ...);
#endif
```

The `fmt` argument passed to `lexical_error` points to a string that uses the same conventions for formatting as the standard `printf` function, so the arguments that follow `fmt` should be arguments appropriate to be printed according to the formats called for in the `fmt` string.

There are examples of programs with lexical errors in the files named `hw2-errtest*.pl0`, where `*` is replaced by a number (or letter). The expected output of each test is found in a file named the same as the test input but with the suffix `.out`. For example, the expected output of `hw2-errtest3.pl0` is in the file `hw2-test3.out`.

7.5 Checking Your Work

You can check your own lexer by running the tests using the Unix command on Eustis, which uses the `Makefile` from the `hw2-tests.zip` file in the course homeworks directory.

```
make check-outputs
```

Running the above command will generate files with the suffix `.myo`; for example your output from test `hw2-errtest3.pl0` will be put into `hw2-errtest3.myo`.

A Hints

Create a transition diagram for a finite state automaton (DFA) to recognize each lexeme on the source program and once accepted generate the token, otherwise emit an error message.

You can use the `ungetc` function from the C standard library to push a character back on the input.

You may find the macros and functions in the standard library's header file `ctype.h` useful.

We are providing several files for this homework, all of which are in the `hw2-tests.zip` file in the course homeworks directory:

- `token.h` and `token.c`, which have declarations of types and a function related to tokens and their types.
- `lexer_output.h` and `lexer_output.c` which provide declarations and definitions for formatting the output and driving the analysis.
- `utilities.h` and `utilities.c`, which provide for printing of error messages to `stderr`.
- `lexer.h`, which declares the types of the functions you are to implement.
- `Makefile`, which can compile your program (based on the files named in your `sources.txt` file), and can check the output of tests against our tests.