Lab 1 Lexical Analysis

1 Goal

You are given:

1. A public repository of a incomplete project on Gitlab. The URL is http://210.45.114.146:880/staff/pl0compiler.
2. This Guide.

After this project, you should:

1. Learn to cooperate with Git
2. Complete a small lexical analyzer for PL/0 language with other members in your group.
3. Write a report for your participation and contributions.

2 Contents

2.1 PL/0 language tokens

Tokens in PL/0 are divided into 4 types: `identifiers`, `numbers`, `symbols` and `reserved words`.

2.1.1 Identifiers

Identifiers are strings which programmers define to represent variable and procedure names. In PL/0, there are no restrictions for the identifiers. However, arbitary names for variables and procedures may lead to unpredictable results. You are **REQUIRED** to ensure that identifiers, which are accepted by the lexical analysis module, contains **ONLY LITERAL LETTERS, NUMBERS AND UNDERLINE(_)**. Furthermore, numbers **SHOULD NOT** be the first character of an identifier.

> You can refer to C programming language if you do not know how to understand this.

2.1.2 Numbers

Numbers in PL/0 are **ALWAYS** integers ranging **from -999999999 to 999999999** (which can be easier to be represented in C). However, in lexical analyzer integers are recognized as unsigned integers ranging **from 0 to 999999999**. The sign `-` should be recognized as a symbol (which means "minus").

2.1.3 Symbols

Symbols accepted are: `+ - * / = != < <= > >= ( ) , ; . :=`.

`+ - * /` means 4 basic arithmetic operations. `=` here, which does not means assignment to variables (The correct choice is `:=`), together with `!= < <= > >=` are comparison operators. `( )` are parentheses. Other symbols are punctuations.

2.1.4 Reserved Words

Reserved words are here: `var`, `const`, `procedure`, `begin`, `end`, `if`, `then`, `do`, `while`, `call`, `odd`. **Case sensitive**.

2.1.5 Comments

Comments are dropped in lexical analysis stage. They are not regarded as tokens.

Line comments starts with double slash (`//`), on the right of which **in this line** any input source code are dropped. You should read from left to right for each line. Whenever `//` is detected, no more words need to be read.

Block comments starts with `/*` and ends with `*/`. Whenever `/*` is detected, all the words read are comments and should be dropped, until the next `*/` is read. An unterminated `/*` should be reported as an error, while an unexpected `*/` should be recognized as two tokens (`*` and `/`). For example, the following source code does not mean `more words` as a comment:

```
/* var x; */ more words */
```

In this case, the words in `more words` should not be dropped, and the last two characters `*/` should be recognized as token `*` and token `/` here.

In the code above, when the first `*/` is detected right after `var x;`, the comment ends. The second `*/` near the end of the line is invalid and should be reported. (More talk is here.)

2.1.6 Splitting

Tokens are recommended to be splitted by `' '`, `'\t'` and `'\n'`. However, this is not required. Splitting of tokens follows:

1. Symbols can always closely stand by other tokens (including other symbols). The longest match principle is used when multiple symbols is together.
   > `a+b` should be recognized as three tokens: identifier `a`, symbol `+` and identifier `b`.
2. Numbers are not accepted when next to identifiers and reserved words as token type "number" here.
   2.1. Numbers cannot appear before reserved words and identifiers without splitters. This should reported as an error.
   2.2. Numbers can appear in identifiers with literal letters (cannot be the first character in the identifier).
3. Identifiers and reserved keywords are regarded as words. Words **MUST** be splitted.

2.2 PL/0 Compiler Architecture

The PL/0 compiler follows the 4 phases of compilation: lexical analysis, syntax analysis, semantic analysis, code generation. The 4 parts are designed to be 4 modules to make up for the entire compiler. In this experiment, we are only concerned with its lexical analysis, thus we only focus on the lexical analyzer module.

2.2.1 Project files

The project manages files by CMAKE (Minimum version is 2.8). Files are arranged as below:

```
(root directory)
-- + pl0compiler
   ├ -- + common
   │    ├ --   CMakeLists.txt  (for cmake in subdirectory)
   │    ├ --   common.c  (implementation of functions in common.h)
   │    └ --   common.h  (initial structure for the whole compiler, declaration for global variables)
   ├ -- + examples  (different examples for PL/0 program)
   ├ -- + lex
   │    ├ --   CMakeLists.txt  (for cmake in subdirectory)
   │    ├ --   pl0_lex.c  (implementation of functions in pl0_lex.h)
   │    └ --   pl0_lex.h  (lexical analyzer)
   ├ --   .gitignore
   ├ --   CMakeLists.txt  (for cmake)
   ├ --   README.md
   ├ --   do_cmake.sh  (shell script for automatically do cmake)
   └ --   test.c  (for testing)
```

2.2.2 Lexical analyzer workflow

First we dive into `pl0_lex.h` to see the structure of the analyzer.

```
typedef struct _tPL0Lex {
    /* Parent return pointer */
    struct _tPL0Compiler * compiler;

    /* For output */
    PL0TokenType last_token_type;
    char last_id[MAX_ID_LEN + 1];
    int last_num;

    /** --------------------------
     * TODO: Your variables here
     */
} PL0Lex;
```

The first pointer `compiler` is used to link back to the compiler which contains the lexical analyzer. It does not matter much in this project.

The next 3 variables are very important, which play the key roles in token streams for the syntax analyzer. As you should know, the interface exposed by the lexical analyzer to the syntax analyzer is `get_token()`. The syntax analyzer invokes `get_token()` function and then fetches a token from the lexical analyzer. Each token is either an identifier, a number, a symbol or a reserved keyword. That is, a token should have its *TYPE*. Then how do we represent a token?

The `last_token_type` variable holds the type for the last token read. We defined 32 token types here. **Each symbol and reserved keyword has its own type here**.

The `last_id` variable contains the string for the last identifier read. The maximum length for an identifier is 10 bytes. **It is valid ONLY IF the last token read is an identifier**.

The `last_num` variable is similar to `last_id`. We use a C-style integer to represent the last number read. **Remember also it is valid ONLY IF the last last token read is a number**.

> As we mentioned above, the valid range for a number is 0 to 999999999. That is, the length ofa number is **no more than 9**.

Now when the function `get_token` is invoked, the three variables should be set **(This is for you to implement)**. The syntax analyzer recognizes each token by first distinguishing the `last_token_type` variable.

If this variable is set to `TOKEN_IDENTIFIER`, then the token is an identifier. The `last_id` variable contains the name for the identifier.

If this variable is set to `TOKEN_NUMBER`, then the token is a number, whose value is set in `last_num`.

In other cases, a token is a symbol or a reserved keyword. If a keyword is recognized, the `last_token_type` is set to a value larger than `TOKEN_RESWORDS`. The syntax analyzer is able to tell if is in this case by simply comparing the two values. Vice versa for `TOKEN_SYMBOL`.

> Variables in enumeration of C is automatically arranged by the ascending order of natural numbers (0, 1, 2, 3, ...). As a result, values of enumeration items can be compared.

Last but not least, if the source code contains lexical errors, the lexical analyzer set the `last_token_type` to `TOKEN_NULL` when the error is read.

2.2.3 Implementation of lexical module

The function for `get_token()` is `PL0Lex_get_token`, which accepts 1 parameter (the pointer of a lexical analyzer). The return value is boolean, `TRUE` for successfully read, `FALSE` for failures (In most cases, it means the end of the source code file, otherwise it should be I/O errors).

> This function should return `TRUE` EVEN IF an invalid token is read. When this happens, the invoker knows the problem when it finds the `last_token_type` is set to `TOKEN_NULL`.
> In later experiments, we will handle errors. Now when you find errors in the source code file, just print it out in the standard output.

This function is **for you to implement**. Tips: remember to **drop the comments** (Refer to section 2.1.5 for details).

Furthermore, location should be detected for each token. A token can be described by its **line number** and **the index of its first and last letter** in the line. For example, in the PL/0 program below:

```
var a;
begin
    a := -1;
end.
```

The first token is `var`. Then `last_token_type` should be set to `TOKEN_VAR`. Now we need more variables for the location of the token (line number is 1 and location in the line is 0 to 2). **THIS IS FOR YOU TO IMPLEMENT**. Add variables to lexical analyzer structure and make it!

2.2.4 Constants list

You are offered two lists of constants (declared in `pl0_lex.h` and defined in `pl0_lex.c`), which contains the strings of reserved words and symbols. What's more, the lists follows the same order for the token types in the type enumeration. The index of the list is useful when a token is detected as a reserved word or a symbol and should set the `last_token_type` variable.

2.2.5 Testing part

A snippet is pre-defined to help you test your implementation of lexical analyzer (written as `test.c`. Dive in and modify as you like.

The default program does not accept invalid tokens (when the `last_token_type` is set to `TOKEN_NULL`. Change it if necessary.

The location of each token is not shown by default. **You need to implement this depending on your data structure of your lexical analyzer**. The preferred style of output is `(line number):(first letter index)-(last letter index)`. For the given program in section 2.2.3, the output should be:

```
Reserved word:    var        1:0-2
Identifier:        a        1:4-4
Symbol:            ;        1:5-5
Reserved word:    begin    2:0-4
...
```

> Alignments is optional but recommended.

2.3 Cooperating with Git

Now it is the first time for all the members to cooperate in your team.

2.3.1 LEADER: Creating your group's fork

Only leaders of all the groups need to do this part.

First open the public project URL (http://210.45.114.146:880/staff/pl0compiler) in your browser. **Make sure you are logged in** before doing anything.

Now click `Fork` button on the main panel. Then you are asked to select a namespace. Choose your group and wait. When the progress is finished, it will appear on the screen.

2.3.2 Branching

It is not recommended to directly commit on the *master* branch. Use your own branches to develop.

Once the project is forked into your group's namespace, all members of your team will find it under the "Your projects" section. First you should clone the remote repository onto your host with `git clone`. The cloning URL can be found on the web pages on the server.

Once you have cloned to your computer, do not make changes to the master branch of the source code directly. Use your own branches with `git branch` and `git checkout` to shift to your branch to work. This helps when multiple people works in parallel.

2.3.3 Committing

NEVER regard commits as saving of your progress. A commit should be checkpoints of your work, that is, a commit should be a snapshot of output of a working stage. For example, part of work is done, or is decided to be delayed.

When a commit is decided to happen, make sure the codes are clear and graceful. Maybe there are BUGS or the codes are incomplete, it's OK. Do not leave the codes in a mess when committing, even if you decide to stop working. In the future, a rollback may happen, and others (or even yourself) will be confused by messy source code.

2.3.4 Merging

When the work on your branch is done, you need to merge your branch onto the *master* branch.

If your role is *maintainer* or higher in the group, you are able to directly push to the *master* branch. In this case, merge your branch to the local *master* branch, then push to the remote repository.

If your role is *developer* in your group, then you cannot push to *master* branch, since the branch is **protected** by default. You have to start a merge request on the web pages.

3 Instructions for lab 1

1. **(LEADER)** Create your group's fork repository of the public repository.
2. Each group need to read the guide, especially section 2.2.3. The output should meet the requirement in section 2.2.5.
3. Each member in the group shares part of work and complete.
4. Finally the project contains the full implementation, with no errors or bugs, which gives a valid output of any input source code file of PL/0.
5. Create a directory in the repository with name `docs`, then create a subdirectory `lab-1`. **Everyone** writes his own report for this project **with a markdown file** describing his own work and problems. Each markdown file should be named with the student ID of each member (e.g.

`PB16XXXXXX.md`).

> Each problem should be described as follows: what it is and how it happens, the possible solutions, how you work to make it done, and what you have learnt.

6. **(LEADER)** Remember to describe how your team cooperate in the project in your own report.