Lexical analysis

DEADLINE: March 5, 23:59.
IMPORTANT: You should hand in your solution on the Minerva Dropbox. Create an archive using the `make_tarball.sh` script, and make sure you send it to the teaching assistants.
For any questions, contact us at `compilers@lists.ugent.be`.

1 Introduction

The goal of this lab is to process source files in a C-like language, and generate a stream of tokens along with location information, or an error message if the syntax of the file is invalid. You will do so using the Flex lexical analyzer generator, an old but commonly-used piece of software that generates C code implementing a lexical analyzer.

The source code you have to parse is written in a simple C-like language, informally specified in Appendix A. Make sure your lexer detects each of these syntaxes, even if they are not covered by the example source code files as part of this assignment. If you create additional tests, be sure to put them in the `tests/` subfolder such that they are handed in together with the rest of your solution.

2 Set-up

For this and other labs, we will be using Ubuntu 18.10 which ships Flex version 2.6.4[1] as part of the `flex` and `libfl-dev` packages. The preferred way to set-up this environment is to use our Docker images, which come preinstalled with the necessary software to solve the lab.

After installing and initializing Docker for your operation system, you can spawn a container for this lab using the following command:

```
$ docker run --rm -it tbesard/compilers:pract1
root@0f01d2b521d6:/#
```

You are now in a `bash` shell within the container. The environment is interactive (i.e., you can enter commands from your terminal) by having passed the `-i` and `-t` flags. To exit the container, use the `exit` command or press ^D. Because of the `--rm` flag, the container is now also removed and does not consume and space on your computer.

Each container lives in a fully isolated environment, and does not have access to the so-called host system. To work with the files that are part of this assignment, you need to grant the container access to certain folders on your computer. You can do so using the `-v SRC:DST` option, mounting the path `SRC` on your host into the container at `DST`. Note that both paths need to be absolute; you can do so conveniently using, e.g., the `pwd` command:

```
$ docker run --rm -it -v $(pwd)/files:/files tbesard/compilers:pract1
root@453f44b434b0:/# ls /files
Makefile  lexer.hpp  lexer.l  main.cpp  make_tarball.sh  test
```

[1]You can find the documentation for this specific version of Flex at `https://users.elis.ugent.be/~tbesard/compilers/flex/`

You can now use a local editor to modify the assignment files, and compile them within the container. Note that you can always install the relevant pieces of software on your local system and not use the container at all. If you do so, always **make sure that your solution works in the container** before you hand in, because we will be using this infrastructure to grade.

3 Assignment

3.1 Basic lexing

The first part of the assignment consists of adding simple lexing rules to the `lexer.l` source file. Upon running `make` and executing the resulting `main` executable on a source file, the lexer will immediately error due to unrecognized symbols:

```
$ make && ./main test/dummy.c
Syntax error: unknown symbol
```

Now add the necessary rules to parse this file. The `Token` enum in `lexer.hpp` defines the tokens you need to emit. If certain values do not have corresponding tokens, as is the case with punctuation, you can emit its plain ASCII value. Whitespace can be ignored, and does not need any token. The `dummy.c` file should lex as follows:

```
$ make && ./main test/dummy.c
         1:1 → 1:1       42         INTEGER
         1:1 → 1:1        ;             ';'
```

3.2 Location information

The output as shown above lacks location information. By implementing the `update_location` function, which is called prior to the action of each matched rule, you can properly fill out the `begin` and `end` fields of the `Lexer` class. You can determine the necessary offsets by inspecting the current token string, accessible through the `YYText()` function of the lexer's base class.

After implementing this function, the output of lexing `dummy.c` should look as follows:

```
$ make && ./main test/dummy.c
         1:1 → 1:3       42         INTEGER
         1:4 → 1:5        ;             ';'
```

3.3 Error reporting

If the source file contains invalid syntax, the resulting error is not very useful:

```
$ make && ./main test/error.c
         1:1 → 1:5      here      IDENTIFIER
        1:6 → 1:11     comes      IDENTIFIER
       1:12 → 1:14        an      IDENTIFIER
       1:15 → 1:20     error      IDENTIFIER
Syntax error: unknown symbol
```

Using the location information you added before, implement the `error` function to generate more useful diagnostics. Modify the `update_location` function to maintain a buffer of source code, and access that buffer in order to display a line of context for the error:

```
$ make && ./main test/error.c
         1:1 → 1:5      here      IDENTIFIER
        1:6 → 1:11     comes      IDENTIFIER
       1:12 → 1:14        an      IDENTIFIER
       1:15 → 1:20     error      IDENTIFIER
Syntax error: unknown symbol at line 1 column 21
here comes an error $
                    ^
```

3.4 Context-sensitive rules

Certain types of syntax cannot be parsed with simple, singular rules. For example, block comments cannot easily be matched using regular expression (don't try to do so), and are better dealt with using context-sensitive rules. Start with defining a start condition, `BLOCK_COMMENT`, activating it accordingly, and providing a different set of rules for when this condition is active.

Given this condition, it is easy to add a rule matching the end of a file (`<<EOF>>`) and as such detect unterminated block comments. Try to figure out how to fix location information, as the rules within the `BLOCK_COMMENT` condition will probably invalidate this information:

```
$ make && ./main test/error_blockcomment.c
         1:1 → 1:8   runaway      IDENTIFIER
Syntax error: unterminated block comment at line 1 column 19
runaway /* comment
        ^^^^^^^^^^
```

A Language specification

A.1 Comments

Regular comments `// comment`

Block comments (only to be implemented in 2.4)

```
/*
block comment
*/
```

A.2 Source code

Terminated by semicolon (`;`), white-space is ignored.

A.3 Identifiers

Alphanumeric (including underscores), should start with a non-numeric character (eg. `foo` or `foo1`, but not `1foo`).

A.4 Literals

Integer `1` `123`

Floating point `1.2` `23.`

String everything between double quotes, e.g. `"foo"`, except for other quotes (unless escaped with a back-slash, e.g. `"foo\"bar"`).

A.5 Operators

```
=
==  !=
<   <=  >   >=
+   -   *   /   ^   %
```

A.6 Punctuation

```
,   ;
(   )   {   }   [   ]
```

A.7 Keywords

```
return
if  else  while   for
```