Project #1: Lexical Analysis

Introduction to Compilers (Fall 2024)

TA: Sanghyeon Lee (sanghyeon@snu.ac.kr)

Released: September 24, 2024
Due: October 9, 2024

Contents

1 Introduction

1.1 Lexical analysis

Lexical analysis is the process of converting text into meaningful lexical tokens (lexemes) belonging to categories defined by a lexical analyzer (also known as lexer) program. **The lexer gets a program as an input and outputs a sequence of predefined tokens** after analyzing the program. For example, consider the following C program as an input to the lexer.

```
int foo() { return 0; }
```

The lexer tokenizes this program into a sequence of meaningful tokens. In addition, some actions can be performed when each token is recognized by the lexer.

The following strings are the tokens recognized by the lexer.

```
"int", "foo", '(', ')', '{', "return", '0', ';', '}'
```

In this project, we will build a lexer for our toy language C+-, referred to as subC (sub-C). It is a variant of C with some added or removed features.

1.2 Building Lexers

Lexer generators can automatically build a lexer with just a few lines of tokenization rules. Given the availability of numerous lexer generators, we don't have to build a lexer from scratch. `Flex` (Fast Lexical Analyzer Generator), a reimplementation of POSIX `lex`, is a widely used lexer generator. It is also compatible with POSIX `yacc`, a parser generator, which will be used later in this course. Throughout this course, we will utilize Flex to generate the lexer for subC.

2 Setting up the Project Environment

We strongly recommend using Docker to set up your project environment. However, you can work on the project on any platform, as long as the project compiles and runs correctly.

For students who are not willing to use the provided docker image, we provide a list of packages required for the project.

- Flex
- GNU Bison
- GCC (GNU Compiler Collection)
- GNU Make
- Git

2.1 Install Docker

Please follow the official instructions for your operating system (Linux, Windows, MacOS) to install Docker on your machine.

For linux users to manage the docker as a non-root user, add a user to the docker group after finishing the installation. Type the following command and re-login to the shell.

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

2.2 Create a project container

Pull the project image from docker hub.

```
docker pull namulee/flex-bison
```

Go to the directory you want to work on the project, and run a new container from the image. The directory will be bind mounted [1] to the container after executing the following command.

```
cd {YOUR_PROJECT_DIRECTORY}
docker run  -itd \
            --volume=./:/root \
            --workdir=/root \
            --name 2024-compilers \
            namulee/flex-bison
```

You can check whether the container is running or not using `docker ps -a`.

```
$ docker ps -a
CONTAINER ID   IMAGE                 COMMAND       CREATED          STATUS
    PORTS     NAMES
0db5efd033ce   namulee/flex-bison    "/bin/bash"   4 seconds ago    Up 4 seconds
               2024-compilers
```

If the container is not running, start the container using `docker start`.

```
docker start 2024-compilers
```

2.3 Attach to the container

Once the container is running, you can attach to the container's shell using the `docker exec` command.

```
docker exec -it 2024-compilers bash
```

Microsoft provides Dev Containers extension for VS Code, which allows you to run VS Code inside a Docker container. It is recommended to use this extension for your convenience. Please follow the official guide to get started.

2.4 Clone the project repository

Clone the repository by executing the following command while attached to the container.

```
git clone https://github.com/namu-lee/2024-compilers.git
```

2.5 How to build and test

Navigate to the `src` directory, and build the project using `make` command.

```
cd ./src
make all
```

[1] When you use a bind mount, a file or directory on the host machine is mounted into a container.

Once the build is completed, you can test it with the following command. If you do not specify the `[input_file]`, `stdin` will be used as the input stream.

```
./subc [input_file]
```

3 Lexical Structure of subC

This section demonstrates the structure of our toy language, subC.

3.1 Definitions

Following definitions of tokens are written in POSIX ERE (Extended Regular Expressions)

Name of token	Definition
letter	`[A-Za-z_]`
digit	`[0-9]`
whitespace	`[ \t]` (space and tab)
integer-constant	`[1-9]{digit}*|0`
float-constant	`{digit}+\.{digit}*([eE][-+]?{digit}+)?` (‘\.’ means ‘.’)

Table 1: Definitions of Tokens

3.2 Operators

`( ) [ ] { } -> . , .. ! ++ -- * / % + - < <= > >= == != & && || ; =`

3.3 Comments

`/* */`

3.4 Keywords

`break char continue else float for if int return struct void while NULL`

4 Goal of the Project

Build a lexer using Flex that recognizes all the subC tokens from the input code and prints them to `stdout` each time the token is recognized.

You need to implement your own symbol table using a hash table. You are free to implement any type of hash table, as long as it does not interfere with the execution of the program.

Complete the code in `subc.l` and `hash.c`. You may add or modify any source files as needed, but please make sure that the project can be built using `make all` command.

5 Implementation Details

5.1 Identifiers and Keywords

Keywords are reserved tokens for special purposes by the language.
Identifiers are programmer defined names for entities.

Identifiers and Keywords are recognized by the same pattern.

- Only uppercase and lowercase English letters, numbers, and underscores(_) can be used.
- Predefined keywords cannot be overridden by identifiers.
- A lexeme must start with an English letter or an underscore. Lexemes starting with a number are not permitted.

Then, how can they be distinguished between each other?

When the lexer recognizes a lexeme matches this pattern, it searches the symbol table. Each symbol table entry has a reference count to keep track of how many times the lexeme has been referenced. If an entry already exists, check whether the type of the lexeme is Identifier or Keyword. If it is, increment the reference count of the entry. Conversely, if an entry for the lexeme does not exist in the symbol table, create a new entry with its token type set to Identifier and insert it into the symbol table.

Therefore, to enable the lexer to distinguish Identifiers from Keywords, all Keywords must be pre-inserted into the symbol table before the lexer gets the input.

We recommend implementing the interface to the symbol table using the `enter()` function in `hash.c`.

5.2 Nested comments

[TL;DR] *Use the **start condition** to implement nested comments.*

Your lexer must be able to recognize nested comments such as, "`/* I /* am a */ comment */`", which are against the C standard.

Nested comments cannot be recognized with regular expressions, so we are going to use the start condition in Flex. The start condition allows us to easily distinguish between normal mode and comment mode.

Lexer starts in normal mode. If the lexer recognizes '`/*`' while reading the input, it switches to comment mode. In comment mode, the lexer performs the following three actions:

1. When it recognizes another '`/*`', comment depth count is incremented.
2. After recognizing '`*/`', it decrements the comment depth count. The lexer returns to normal mode when the count reaches 0.
3. All other characters are ignored in the comment mode.

In order to make the lexer begin with normal mode, place the `main()` function in the procedure section of Flex, and set start mode as normal within the `main()`. (Call for `yylex()` should be in there too)

5.3 Dotdot (..) operator and float constant

[TL;DR] *Use the **lookahead operator** to implement the `..` operator.*

In subC, there is a unique dotdot (`..`) operator, which encodes a range of integers. The range starts from the number before the operator, and ends at the number after the operator.

The definition of float constant in subC slightly differs from that in standard C. It is illegal to write a float constant like `.325`. The lexer must recognize the `..` operator while distinguishing it from the float constant. For instance, 1..2 must be tokenized as 'integer', '..', 'integer' rather than 'float', 'float'. You can use the lookahead operator(\) to deal with this ambiguity.

5.4 Output format

Please follow the format described below when printing the result.

```
{Token type}\t{Lexeme}\t{Reference count}
```

- Print **Token type**, **Lexeme** and **Reference count** to `stdout`, each on a new line, in that order.
  - **Token type** is the type of a token.
    e.g. Keyword(KEY), Operator(OP), Identifier(ID), Float-constant(F), Integer-constant(INT)
  - **Lexeme** is the recognized string itself.
    e.g. 'int', '=', '123' ...
  - **Reference count** is the number of recognitions of a token in a program.
- Print Reference count only for identifiers or keywords.
- Each element **MUST** be separated by a **SINGLE TAB ('\t')**.
- **Disregarding these rules will result in a 5% penalty to your project score.**

I/O Example

Input code: `input.c`

```
/************************
  /* nested comments*/
************************/
struct _point {
    float x, y, z;
    int color;
} point[20];

struct _line {
    struct _point *p[2];
    int color;
    float meter = 0.5;
} line[20];

1..50
```

Output:

```
$ ./subc input.c
KEY    struct     1
ID     _point     1
OP     {
KEY    float      1
ID     x          1
OP     ,
ID     y          1
OP     ,
ID     z          1
OP     ;
KEY    int        1
ID     color      1
OP     ;
OP     }
ID     point      1
OP     [
INT    20
OP     ]
OP     ;
KEY    struct     2
ID     _line      1
OP     {
KEY    struct     3
ID     _point     2
OP     *
ID     p          1
OP     [
INT    2
OP     ]
OP     ;
KEY    int        2
ID     color      2
OP     ;
KEY    float      2
ID     meter      1
OP     =
F      0.5
OP     ;
OP     }
ID     line       1
OP     [
INT    20
OP     ]
OP     ;
```

```
INT    1
OP     ..
INT    50
```

6 Submission

- Run `make clean`.
- Compress all the source files with `zip`, including the Makefile.
- Rename the compressed file to `project1_<student number>.zip`.
- Submit it on eTL.
- **Disregarding the file naming rule will result in a 5% penalty to your project score.**

Appendix

Flex Manual

CS143 - flex in A Nutshell

Yash (VS Code extension for Flex/Bison syntax highlighting)