

# Exercise 1: Implementing a Lexer

## Introduction

In this assignment, you will construct a lexical analyzer using Flex, the fast lexical analyzer generator. Your lexer should be able to process a shading language known as RTSL (Ray Tracing Shading Language), used in high-quality image synthesis. A source file for RTSL, as quite common in computer graphics, is called a *shader*. As already presented in the lecture on lexical analysis, the lexical analyzer generator (FLEX) is a program that returns tokens and places the lexeme (value of the token) in a variable visible to the outside world. In this assignment, tokens will be printed to standard output. In the next assignment, we will use them for the implementation of an RTSL parser.

## Files

Inside the given tar archive you will find the following files:

- Three `*.rtsl` shader files.
- One expected output (`sphere.out`) of the lexical analyzer for the input file (`sphere.rtsl`).
- A paper (`rtsl.pdf`) describing the RTSL language.
- A `lexer.lex` stub and a `Makefile`.

## Getting Started

To get started, you should install Flex on your machine and also familiarize yourself with the topic of lexing.

- Flex repository: <https://github.com/westes/flex>
- Flex manual: <https://westes.github.io/flex/manual>
- Lecture 2 on Lexical Analysis
- [ALSU] 3.5 (the Dragon book)

Flex can be built from source, but is also available through various package managers (for instance, `sudo apt-get install flex` on linux, or `brew install flex` on MacOS, etc.).

When you have successfully set up your work environment, you should now be able to test-run the lexer which you are about to build. To do this, navigate to the directory of provided files, and run:

```
$ make
$ ./lexer sphere.rtsl
```

For now, the lexer will output a list of tokens that it could not match (since you have not implemented this yet). When you have completed your implementation, the output should match exactly what is shown in the file `sphere.out`.

## Introduction to Flex

A `.lex` file is organized into three sections separated by `%%`. Consider the following example:

```
/* Declarations */
%%
/* Rules */
break                return BREAK;
continue             return CONTINUE;
[a-zA-Z][a-zA-Z0-9]* return IDENTIFIER;
%%
/* User code */
```

Here, a few rules have been provided. Each rule is shown on a separate line, consisting of a regular expression to be matched, followed by code to be executed if the rule is matched.

Consider the input: `breakage`

The lexer could match this input in two different ways:

1. `BREAK` followed by `IDENTIFIER age`
2. `IDENTIFIER breakage`

It would be a reasonable assumption, that matching happens in order in which the rules are specified. However, this is a fallacy. Instead, the longest match wins and Flex will choose the second option, matching the entire input as `IDENTIFIER breakage`.

The order of the rules does matter if and only if two matches have the same length. For instance, the input `break` could be matched by both the first and the third rule, i.e., as the keyword `BREAK` or as an `IDENTIFIER break`. Since the matches are of equal length, Flex will apply the first matching rule.

The 'declarations' section can be used to declare regular expressions, which can then be used in the rules you specify. This may be helpful to keep your `.lex` file organized and easier to read when working with complex or compound regular expressions. This is shown in the following code box (see below). Note, that declarations may also make use of other declarations.

As you can see in the code box below, a rule can also be given a code block to execute. In this case, the box ensures that a successful match actually returns a tuple of type and lexeme, rather than just the type. As described above, we would like to return `IDENTIFIER breakage` rather than merely `IDENTIFIER`. However, the above rule did not actually accomplish that, and in order to do that we need to make use of `yylval`, to which we assign a copy of the matched text (`yytext`). Here, we assign to its `str` field as we are handling a string, but you may also wish to make use of other fields such as `bval`, `ival`, and `fval` when handling other data types, as indicated in the next section.

```

/* Declarations */
IDENT          [a-zA-Z][a-zA-Z0-9]*
%%

/* Rules */
break          return BREAK;
continue       return CONTINUE;
{IDENT}        { yylval.str = strdup(yytext); return IDENTIFIER; }
%%

/* User Code */

```

## Lexical Classes

RTLS is derived from GLSL, which is a language mainly based on C. Identifiers, numbers and symbols have the same specification as in C, but there is no char or string type, or pointer notation.

In the following, all token types that your lexer must recognize are listed, including a brief explanation. For a more precise definition of various syntactic elements (such as integer and floating point literals), please refer to the GLSL 4.40 specification (<https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf>) and the RTSL paper.

Like C, the language is case-sensitive (in particular, all keywords, types, etc. are case-sensitive). However, integer and floating point literals are *not* case-sensitive (e.g. `0xabc` is the same as `0XABC`).

**Tokens with semantic values:** The following token types have an associated semantic value, given in parentheses next to the token name. For booleans, integers and floats the semantic value contains the parsed value. For identifiers and similar, it contains the string of the identifier.

**BOOL (bval):** Booleans `true` and `false`.

**INT (ival):** Integer literal. As in C, decimal (`123`), octal (`0777`) and hexadecimal (`0xABC`) are supported. Integer literals may have an optional `u` suffix. Integer literals do not contain a sign.

**FLOAT (fval):** Floating point literal. All of `3.14`, `.14` and `3.` are allowed. Additionally, there may be a trailing exponent part, e.g. `1.1e10`. In this case the leading fraction is not required to contain a dot, i.e. `1E-20` is also a float. Float literals may have an optional `f` or `lf` suffix.

**IDENTIFIER (str):** An identifier is a sequence of alphabetic characters, digits and underscores. The first character must not be a digit. Of course, the keywords, types etc. listed below should not be recognized as identifiers.

**TYPE (str):** The following types are supported.

- Simple types: `void bool int uint float double`

- Vector types: `vecN dvecN bvecN ivecN`, where  $N=2,3,4$
- Matrix types: `matN matNxM dmatN dmatNxM`, where  $N=2,3,4$  and  $M=2,3,4$
- RTSL types: `color`
- (We omit sampler types)

**STATE** (`str`): RTSL supports a number of interface methods and state variables, as listed in the paper in Table 1. Note that RTSL state variables have an `rt_` prefix. Interface methods should be recognized as ordinary identifiers, but state variables use a separate **STATE** token. (You do not need to explicitly list all state variables. You can consider any identifier starting with `rt_`, that is not a keyword, as a state variable.)

**Keywords:** For keywords, the uppercase form of the keyword is used as the token name. For example, the keyword `break` yields the token **BREAK**. The following keywords are recognized:

- C keywords: `break continue do for while switch case default if else return struct`
- GLSL keywords: `attribute const uniform varying buffer shared coherent volatile restrict readonly writeonly layout centroid flat smooth noperspective patch sample subroutine in out inout invariant precise discard lowp mediump highp precision`
- RTSL keywords: `class illuminance ambient public private scratch`
- RTSL interface types: `rt_Primitive rt_Camera rt_Material rt_Texture rt_Light`
- (We omit reserved keywords)

**Single-char operators:** The single character symbols `( ) [ ] { } . , ; + - ~ ! * / % < > & ^ | ? : =` should return their own ASCII character code as the token.

**Operators:** The operators `<< >> ++ -- <= >= == != && || ^^ *= /= += %= <<= >>= &= ^= |= -=` should return the tokens **LEFT\_OP** **RIGHT\_OP** **INC\_OP** **DEC\_OP** **LE\_OP** **GE\_OP** **EQ\_OP** **NE\_OP** **AND\_OP** **OR\_OP** **XOR\_OP** **MUL\_ASSIGN** **DIV\_ASSIGN** **ADD\_ASSIGN** **MOD\_ASSIGN** **LEFT\_ASSIGN** **RIGHT\_ASSIGN** **AND\_ASSIGN** **XOR\_ASSIGN** **OR\_ASSIGN** **SUB\_ASSIGN** respectively.

**Whitespace:** Includes spaces, tabs, newline and carriage return, vertical tab and form-feed. Do *not* emit a token for whitespace.

**Comments:** Consist of text enclosed in `/*` and `*/`, or any text following the `//` symbol until the end of the line. Do *not* emit a token for comments. You do not need to implement support for C-like newline escaping.

**Errors:** If a character cannot be recognized as any of the other token classes, an **ERROR** token with the character as the semantic `str` value should be yielded.

When handling whitespace and comments, take care to also update the `line_number` variable, as appropriate.

## Output

The provided lexer stub already contains code to print the tokens. You only need to implement the lexer definition, the output code should not be changed.

Make sure that you provide exactly the same output for the provided sphere.rtsl file and verify that the other RTSL files also produce correct results. We also suggest to create some additional test-cases for various edge-cases. For example, the provided test files do not fully exercise all the possible integer and floating-point notations. We will be testing your lexer on more inputs than the ones provided.

To compare your output with our expected output, you can use the following command:

```
> diff -b our_output your_output
```

## Submission

- Before submitting, ensure that your program produces the correct output for the above specification, including for all provided shader files
- Use the ISIS website
- Only submit the `lex` file
- File name has to be: `lastname1_lastname2_lastname3.lex`, e.g.:  
`maradona_klinsmann.lex`
- First line of the `lex` file has to include first name, surname and student id, for each group participant, e.g.  
`/* Diego Maradona 10, Juergen Klinsmann 18 */`
- Up to **three** people per group
- ~~Be sure that your project works on the machines available at the TEL building.~~