Compiler Design - WS 2016/17, TU Berlin - Lecturer: Dr. Biagio Cosenza

Exercise 1: Implementing a Lexer

The first assignment is the construction of a lexical analyzer for a shading language known as RTSL (Ray Tracing Shading Language), used in the context of high-quality image synthesis. You should use the lexical analyzer generator Flex. A source file for RTLS, as quite common in computer graphics, is called *shader.*

Inside the given tar archive you will find

- 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 specification. You must read the whole sections 3, all the listed examples and table 1, **before** starting the assignment

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 should be printed to standard output according to the following specifications.

The lexical analyzer should recognize identifiers, variable qualifiers, built-in functions and other keywords defined by the RTSL language.

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`).

The following tokens need to be recognized:

`BOOL true/false`: Booleans true and false.

`INT num`: 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 num`: 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 name`: 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 name`: Types include:

- Default types: `int float bool void`
- Vector types: `vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4`
- Built-in types: `rt_Primitive rt_Camera rt_Material rt_Texture rt_Light`

`QUALIFIER name`: Variable qualifiers (`attribute, uniform, varying, const`) and class scope variable modifiers (`const, public, private, scratch`).

`STATE name`: 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 type, as a state variable.)

`KEYWORD name`: Keywords include:

- Inherited from C: `break case const continue default do double else enum extern for goto if sizeof static struct switch typedef union unsigned while`
- RTSL keywords: `illuminance ambient`
- Built-in functions: `dominantAxis dot hit inside inverse luminance max min normalize perpendicularTo pow rand reflect sqrt trace` (This is only a small subset of built-in GLSL/RTSL functions, but you do not need to recognize more.)

`SWIZZLE name`: Internal components of a type can be accessed using the dot operator, e.g. `pos.x`. Here the `.x` part should be lexed as `SWIZZLE x`. Note that the `SWIZZLE` argument does not contain the leading dot.

Operators: The operators `+ * - / = == != < <= > >= , : ; ( ) [ ] { } && || ++ --` should be recognized using the following tokens: `PLUS, MUL, MINUS, DIV, ASSIGN, EQUAL, NOT_EQUAL, LT, LE, GT, GE, COMMA, COLON, SEMICOLON, LPARENTHESIS, RPARENTHESIS, LBRACKET, RBRACKET, LBRACE, RBRACE, AND, OR, INC, DEC`.

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.

Be sure that your lexer understands all keywords present in the three *shader* files.

Example

The following example illustrates the expected output for a number of tokens, including the swizzle operator. The code

```
vec2 pos;
pos.x = 0;
```

should return the following tokens:

```
TYPE vec2
IDENTIFIER pos
SEMICOLON
IDENTIFIER pos
SWIZZLE x
ASSIGN
INT 0
SEMICOLON
```

For a more detailed example output, see the `sphere.out` file.

Error Reporting

You must keep track of the current line number while you are scanning to report errors. For unrecognized symbols an error message should be provided.

For example:

```
ERROR(23): Unrecognized symbol "#"
```

where 23 is the line number of the error. Do not abort lexing on first error.

Keeping track of line numbers needs to account for the fact that comments can span multiple lines. Line numbers start with line=1.

Output

We are using a semi-automated evaluation approach, so it is imperative that you match the provided output exactly. All output should go to the standard output (not a file).

We compare your output with our expected output using the command:

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

for all the provided rtls files and, optionally, with additional ones not too much different from the others.

Additional remarks

To complete this exercise, you should be able to write a lexer, similar to the ones provided as examples in the previous lectures. Your goal is to write a simple program that repeatedly invokes the lexer and prints out the token (i.e token name and lexeme if needed) that the lexer returns (one token per line). You can make the development and testing part easier by using an automated script or Makefile that compiles your file(s) and creates the output.

You should test your lexer on the test program and compare with the reference output.

Submission

- 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 for group
- The submission deadline is 11:59pm of the due date

Links

- FLEX http://flex.sourceforge.net
- Lecture 2 on Lexical Analysis
- [ALSU] 3.5 (the Dragon book)

Be sure that your project works on the machines available at the TEL building.