Compiler Design - WS 2023/24, TU Berlin

Exercise 2: Implementing a Parser

The second assignment consists of the implementation of a syntax analyzer for RTSL.

To create your parser, you will use the Bison parser generator. Your parser will use the tokens produced by the lexer you implemented in the previous assignment.

Your parser must correctly accept all the input codes which are valid RTSL, but should not accept codes with syntax errors. Additionally, a number of semantic checks have to be implemented.

RTSL Grammar

There is no formal specification of the RTSL language. However, it is based on GLSL, which in turn is very similar to C. The paper provided in the previous assignment describes the additional features and differences between RTSL and GLSL.

A grammar specification of GLSL can be found in chapter 9 of https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf, and a Bison-compatible version of this grammar is provided with the assignment files. We suggest that you base your parser implementation on this grammar. The `glsl_grammar.txt` file contains some instructions on what you need to do in order to use this grammar.

Alternatively, because RTSL (or at least the parts we are testing) is very similar to C, you can also start from an ANSI C parser. An example implementation is available at:

- http://www.quut.com/c/ANSI-C-grammar-l-2011.html
- http://www.quut.com/c/ANSI-C-grammar-y.html
  (note that this grammar has two shift/reduce conflicts)

Be aware that many ANSI C features are not implemented in RTLS. For instance, there are no external declarations, pointers, atomic and alignment specifiers. Your goal is to implement a parser able to understand all the features used in the provided test sets.

The two main differences between RTSL and GLSL are:

- Each file starts with a shader definition (e.g., `class Example : rt_Material`) and zero or more interface methods, depending on the shader type (e.g., an `rt_Light` can only have `constructor()` and `illumination()`). This information is available in Table 1 from the RTLS reference paper (and copied later in this document). Note that a shader does not have to implement all of the functions in the interface specification for that shader type, but implementing a wrong one is considered a (semantic) error (e.g., a `rt_Primitive` cannot implement `illumination()`, as it is designed for `rt_Light`).
- A declaration may have a qualifier, e.g. `public vec3 position`.

Camera	Primitive	Texture	Material	Light
vec3 RayOrigin vec3 RayDirection vec3 InverseRayDirection float Epsilon float HitDistance vec2 ScreenCoord vec2 LensCoord float du float dv float TimeSeed	vec3 RayOrigin vec3 RayDirection vec3 InverseRayDirection float Epsilon float HitDistance vec3 BoundMin vec3 BoundMax vec3 GeometricNormal vec3 dPdu vec3 dPdv vec3 ShadingNormal vec2 TextureUV vec3 TextureUVW vec2 dsdu vec2 dsdv float PDF float TimeSeed	vec2 TextureUV vec3 TextureUVW color TextureColor float FloatTextureValue float du float dv float dsdu float dtdu float dsdv float dtdv vec3 dPdu vec3 dPdv float TimeSeed	vec3 RayOrigin vec3 RayDirection vec3 InverseRayDirection vec3 HitPoint vec3 dPdu vec3 dPdv vec3 LightDirection float LightDistance color LightColor color EmissionColor vec2 BSDFSeed float TimeSeed float PDF color SampleColor color BSDFValue float du float dv	vec3 HitPoint vec3 GeometricNormal vec3 ShadingNormal vec3 LightDirection float TimeSeed
void constructor() void generateRay()	void constructor() void intersect() void computeBounds() void computeNormal() void computeTextureCoordinates() void computeDerivatives() void generateSample() void samplePDF()	void constructor() void lookup()	void constructor() void shade() void BSDF() void sampleBSDF() void evaluatePDF() void emission()	void constructor() void illumination()

Table 1: RTSL state variables and interface methods. In code, all state variables are prefixed with rt_

Unfortunately, Table 1 contains a number of errors:

- `Primitive` also supports `vec3 HitPoint`.
- `Material` also supports `vec3 ShadingNormal`.
- `Material` also supports `float HitDistance`.

The stub `parser.y` file provided with the assignment also contains this table in more convenient form.

Output

As expected output, you will have to produce two different outputs for each input shader (`.rtsl`).

On `stdout`, you should print some information of the ongoing parsing, for example when you encounter a shader or function definition or certain statements. Use as reference the output files provided in the assignment `tar.gz` file in order to understand what to print.
Important: Semantic actions are only executed once the entire production has been matched. As such, your output may appear in a different (partially reversed) order than the corresponding structures in the input files. This is normal and expected.

On `stderr`, you should print only information about errors during parsing. You should print:

- Nothing, if the file is correctly parsed.
- A syntax error message for generic parsing errors (this is already implemented in yyerror).
- A specific error message for a few specific semantic errors, discussed in Test Set 3.

We are using a semi-automated evaluation approach, so it is imperative that you exactly match the provided `*.out` and `*.err` for each `*.rtsl`. You can generate two output files for stdout and stderr using:

```
> ./parser test0.rtsl > test0.my_out 2> test0.my_err
```

You can then compare the output with the expected output using `diff`.

Your parser should have at most 1 shift/reduce conflict (dangling else) and no reduce/reduce conflicts.

Provided files

The tar.gz archive for this assignment includes:

- A `parser.y` file. This is the file you need to work on.
- A `lexer.lex` file. You should replace this file with your lexer implementation from the previous assignment. If you did everything correctly, this should be a drop-in replacement, which does not require further changes.
- A `Makefile`. Run `make` to run `flex` and `bison` and generate the `./parser` binary.
- A `glsl_grammar.txt` file. This contains a Bison-compatible GLSL grammar based on the grammar in the GLSL specification. The top of the file includes additional instructions, if you want to base your parser on this grammar.
- A number of test files (`*.rtsl`, `*.out` and `*.err`), which are described in the following.

Test Set 1 - RTSL Test Examples 0 to 5

The first six test examples will test simple, syntactically correct codes that your parser should accept. They will produce no error message, if your parser correctly implements the RTSL grammar, and a list of syntactic elements, as provided in the reference output. Some of them address particular parsing problems, such as dangling else, public variable definitions, while/for syntax, and other possible ambiguities that need to be solved with clever grammar definitions.

Test Set 2 - RTSL Test Examples {dieletric_material|sphere|pinhole_camera}.rtsl

The second set of codes has more complicated RTSL codes and requires solving possible ambiguity in the language specification. For instance, your parser should understand that `vec3` is a type that can be used in a definition like:

```
public vec3 center;
```

as well as an expression as follows:

```
rt_BoundMin = center - vec3(radius);
```

Test Set 3 - RTSL Test Examples 6 to 8

In the last test set, you should implement a basic mechanism to support a few semantic checks. All the codes belonging to this set are syntactically correct but present some semantic error that must be checked.

Test 6 and 7 both have the same problem: a wrong function interface is defined in a shader type that does not support it. E.g., test 6 has the `shade` function, which is not one of the supported interface functions for a camera shader (check Table 1).

While implementing the semantic checks for this test set, don't forget to also try it with the other, semantically correct codes in test sets 1 and 2, where your semantic check is not supposed to print an error.

Similarly, Test 8 tests whether a state variable is read/written from the wrong shader type. You can use the data tables provided in the parser.y file to implement these semantic checks.

Hints

Work incrementally: starting from `test0.rtsl`, gradually add new symbols to your grammar so that it will gradually be able to parse more complicated examples. Carefully check any new shift/reduce or reduce/reduce conflicts as soon as you introduce new production rules, and try to fix them before you write new ones. The parser.output file is useful to diagnose conflicts.

Avoid use of ε-productions like A → ε, which may lead to conflicts. Where possible, prefer left recursion instead of right recursion (see Section 3.3.3 of the Bison manual).

The way this assignment is formulated, it does not necessary require the explicit usage of a symbol table. You are free to use it or not as long as your output matches the expected one.

You can propagate information bottom-up by using semantic values, or use global variables or a global symbol table. Do not always assume that "what comes first, is parsed first", as this depends essentially on the way non-terminals are expressed and grouped in the grammar specification.

Submission

- Use the ISIS website
- You should only submit your input file to bison and your input file to flex
- File names should stay as `lexer.lex` and `parser.y`.
- First line of both lex and bison files should include first name, surname and student id, for each group participant, e.g.
  `/* Diego Maradona 10, Juergen Klinsmann 18 */`
- Put all your submission files in a directory called `parser_group12`, (substitute the number of your group) then tar the whole directory with the following file name:
  `parser_group12.tar.gz`

Links

- Lecture 3 (Syntax Analysis) and 4 (Semantic Analysis)
- Bison http://www.gnu.org/software/bison
- Flex http://westes.github.io/flex/manual/