Compiler Design - WS 2015/16, TU Berlin - Lecturer: Dr. Biagio Cosenza

Exercise 2: Implementing a Parser

The second assignment consists on 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, however this time the lexer should not just print the tokens on `stdout`, but it should just return them as tokens for Bison (see the examples to understand how).

As now we will do syntax analysis, we will be able to solve some ambiguities we had in the previous assignment (e.g., is “`inline`” an identifier or a keyword?). Also, you will have to do some changes on the `*.lex` file you have submitted for the assignment 1.

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

RTSL Grammar

RTSL grammar specifications have been presented in the previous assignment.

To start your implementation, I suggest to have a look at ANSI C lexical and grammar specification

- 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.

While RTSL grammar is mainly similar to C specification, there are few extra features to handle:

- a translation unit corresponds to a single input shader file (a `.rtsl` file), therefore the parser should not care about implementing `yywrap()` logic (a simple, empty implementation is fine);
- a translation unit is the full processed code, and it should have a shader definition (e.g., `class Example : rt_Material`) and zero or more interface methods, depending on the shader type (e.g., a 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 may implement one of the function in the interface specification for that shader type, but implementing a wrong one is considered a (semantic) error (e.g., a `primitive` cannot implement `illuminance()`, as it is designed for `light`);
- a declaration may have a qualifier, e.g. `public vec3 position`.

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

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

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, namely the head of (some) production rules. Those include generic statement, if/else, function definition and shader definition (with the shader type). Use as reference the output file provided in the assignment `tar.gz` file in order to understand what to print.

On `stderr`, you should print only information about error in the parsing. You should print

- Nothing, if the file is correctly parsed
- A simple generic "`Syntax error`" message, for general parsing error
- A specific error message for few specific semantic errors, discussed in Test Set 3

Don't forget that we are building the parse tree bottom-up: do not assume a node is visited before its children.

We are using a semi-automated evaluation approach, so it is imperative that you match the provided `*.out` and `*.err` for each `*.rtsl`. In particular, we will use the following command to compiler your parser (be sure to do the same):

```
> flex -ostudent.lex.c ./student.lex
> bison -vd ./student.y -ostudent.yy.c
> gcc -o parser.out student.lex.c student.yy.c
```

and the output files are generated with

```
> ./parser.out test0.rtsl > test0.out 2> test0.err
```

for each input `rtsl` file; note that this requires something like: `yyin = fopen(argv[1],’r’)`

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 tokens, as provided in the reference output. Some of them address particular parsing problem, such as dangling else, public variable definitions, while/for syntax, and other possible ambiguity that need to be solved with clever grammar definition.

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 specifications. For instance, your parser should understand that `vec3` is a type that can be used in an definition like:

```
public vec3 center;
```

as well as am 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 implement 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 have both the same problem: a wrong function interface is called 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 function for a camera shader (check Table 1).

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

Test 8 is a bonus test code which ask to check whether a `rtState` is read/written from wrong shader type.

Debugging

When the Bison parser detects syntax errors, it invokes the `yyerror` function. Bison offers a more advanced way to implement error reporting, for instance by using

```
%define parse.error message
```

however, this message sometime contains incorrect information, that can be fixed be enabling LAV. You can find more information on the Bison manual, section 4.7
https://www.gnu.org/software/bison/manual/html_node/Error-Reporting.html

Hints

Work incrementally: starting from `test0.rtsl`, gradually add new symbols to your grammar so that it will gradually be able to parser more complicate 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 to write new ones.

Be sure to do not have 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, does not necessary require the explicit usage of a symbol table. You are free to use it or not as soon as your output matches the expected one.

You can call Bison with flag –d to create header file containing token numbers (used in Flex). Since Bison creates token numbers, it should be called before Flex.

You can also call Bison with –v flag to dump an extra output file showing resulting LR state machine, as well as shift-reduce and reduce-reduce conflicts.

You can propagate information bottom-up by using attribute types, or use global variables or a global symbol table. Do not always assume that "what it comes first, is parsed before", as this depends essentially on the way nonterminal 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 have to be: `lastname1_lastname2.{lex|y}`, and so forth for all participants, for example: `maradona_klinsmann.lex maradona_klinsmann.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-lastname1-lastname2" then tar/zip the whole directory with the following file name:
  `lastname1_lastname2.tar.gz`
- The submission deadline is on 11:59pm of the due date

Links

- Lecture 3 (Syntax Analysis) and 4 (Semantic Analysis)
- Bison http://www.gnu.org/software/bison
- Flex http://flex.sourceforge.net