University of Toronto
CSC467F Compilers and Interpreters, Fall 2018

MiniGLSL Specification

MiniGLSL language description

As part of this course you will implement a compiler for the MiniGLSL language. MiniGLSL is a simplified version of GLSL that was designed for this course. Before reading this document,it is highly recommended that you have some general knowledge of shaders and GLSL âĂŞ going over the introduction slides at http://dsrg.utoronto.ca/csc467/shader.pdf should be enough.

GLSL

GLSL (OpenGL Shading Language), also known as GLslang, is a high level shading language based on the C programming language. It was created by the OpenGL ARB to give developers more direct control of the graphics pipeline without having to use assembly language. Further knowledge of GLSL is not really required for this lab, but if you are curious please refer to: http://www.lighthouse3d.com/opengl/glsl.

MiniGLSL

MiniGLSL, as well as GLSL has a very similar syntax to the C programming language. Below is a description of everything you need to know about MiniGLSL in order to successfully complete this lab. Note that there are a lot of small rules and corner cases, so if you are not sure just ask!

Main Function

Typically GLSL programs, just like C programs have a main() function. However, to simplify parsing, MiniGLSL emits the main() function âĂŞ the code is enclosed inside a scope instead. For example:

```
{
    int num = 5;
    num = 10;
}
```

Data Types

MiniGLSL suporst the following data types:

- bool, bvec2, bvec3, bvec4
- int, ivec2, ivec3, ivec4
- float, vec2, vec3, vec4

Where vec{2,3,4} are vectors of 2,3 or 4 floats, ivec{2,3,4} are vectors of 2,3 or 4 integers, and bvec{2,3,4} are vectors of 2,3 or 4 booleans.

Structures are not supported. Arrays(aside from vectors) are not supported.

Variables

Declaration is similar to C, but it is not allowed to declare 2 variables in 1 declaration. For example:

- `int num; // Ok`
- `vec3 myvec; //Ok`
- `int num1,num2; //Wrong`

Variables can be initialized during declaration, if the variable is not initialized the default value is set to 0(for vectors all components are set to 0). For example:

- `int num = 1; // Ok`
- `int num; // Ok, same like int num = 0;`

Operators do not associate. For example:

- `int a = b =3; // Wrong`

Vectors are initialized using constructors, the constructor expects arguments of the same type like the vector (if it is a vector of integers, the arguments must be all integers). For example:

- `vec2 myVec = vec2(1.0,5.0); // Ok`
- `vec2 myVec = vec2(1,5); // Wrong`
- `bvec3 myBVec = bvec3(true,false,true); // Ok`

Vectors can be accessed similarly to arrays. For example:

```
ivec4 myVec = ivec4(0,1,2,3);
int num = myVec[2];
```

Type casting is not allowed. For example:

- `int num = 5.0; // Wrong, 5.0 is float, not int`
- `int num = (int) 5.0; // Wrong`
- `int num = int (5.0); // Wrong`

Math Operations

MiniGLSL defines most of the same math operations that are supported in C .(The list of operations is specified in the grammar). However, due to the introduction of some new types, some rules have to be clarified. If one of the operands is a vector than the other operand must be either:

1. A vector of the same type and size
2. A scalar of the same type

In both cases the return type is the same as the vector type. For example:

```
vec4   myVec1;
vec4   myVec2;
vec3 myVec3;
vec4 tempVec = myVec1 * myVec2; // Ok
tempVec = myVec1 * Vec3; // Wrong
tempVec = myVec1 * 3.2; // Ok
```

Qualifiers

This part can get a little tricky, so pay a close attention.
GLSL supports the following variable qualifiers:

- const - The declaration is of a compile time constant. Same like C.
- attribute - Global variables that may change per vertex, that are passed from the OpenGL application to the shaders. For the shader this is a read-only variable.
- uniform - Global variables that may change per primitive (they dont change for the scene), that are passed from the OpenGL application to the shaders. . For the shader this is a read-only variable.
- varying - used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.

It is too complicated to implement the attribute, uniform and varying types. Doing that would require to design a linker, and an interface to openGL. Therefore MiniGLSL implements *only* the const qualifier. It is implemented in a similar way to C, except that consts must be assigned a value that is static in compile time. For example:

```
const int num = 1; // Ok
const int num = 1 + 1; // Wrong
```

However, we do need a way to communicate between OpenGL and our shader. So we are going to predefine some variables in our compiler to allow communication. These variables are usually declared in GLSL using one of the three mentioned above qualifiers (attribute, uniform,varying), and are logically linked to underlying hardware registers.

Because of that we have to treat these predefined variables according to the rules of their qualifier. We are going to simplify it and divide our predefined variables into 3 groups of custom type qualifiers:

- Attributes: Read only, cannot be assigned to a constant.
- Uniforms: Read Only, can be assigned to a constant.
- Result: Write only. Cannot be assigned to in an 'if' or 'else' statement(think about the reason for that when doing lab4).

To reemphasize, the custom qualifiers are used only for internal compiler representation, they are not part of the MiniGLSL language. For example:

```
Attribute int myNum; // ILLEGAL!
```

Here is the list of our predefined variables, with the custom qualifiers in brackets:

```
vec4 gl_FragColor // result
bool gl_FragDepth // result
vec4 gl_FragCoord // result
vec4 gl_TexCoord // attribute
vec4 gl_Color // attribute
vec4 gl_Secondary // attribute
vec4 gl_FogFragCoord // attribute
vec4 gl_Light_Half // uniform
vec4 gl_Light_Ambient // uniform
vec4 gl_Material_Shininess // uniform
vec4 env1 // uniform
vec4 env2 // uniform
vec4 env3 // uniform
```

As far as you are concerned, all you have to do for this lab is:

- Predefine these variables in the compiler (I suggest you do not do it in the scanner, instead do it in Lab3 and Lab4).
- In the semantics check(Lab3) make sure that all the qualifier rules are met.
- In the code generator(Lab4) use the appropriate register type(more about that in Lab 4 handout).

A few examples of how to use predefined variables:

```
vec4 myVec = gl_Color; // Ok
vec4 myVec = gl_FragColor ; // Wrong, cannot read a result
   variable.
gl_FragColor = myVec; // Ok
gl_Light_Half = myVec;// Wrong, cannot write to a uniform
   variable.
const vec4 myVec = gl_Light_Half; // Ok
const vec4 myVec = gl_Color; // Wrong, cannot assign an
   attribute to a const.
if(...){
    gl_FragColor = myVec; // Wrong,  cannot assign to a result
                          // variable inside an if statement
}
```

Input and Output

MiniSLSL is a shading language, therefore it operates on a set of input and output registers. The input registers are logically linked to by the Attribute and Uniform variables. The output registers are logically linked to by the Result registers. A meaningful program (one that outputs something) should write to one of the output registers(assign to one of the result variables).

Functions

MiniGLSL does not allow creating new functions. However 3 types of predefined functions are supported :

- lit:

```
vec4 lit(vec4);
```

- dp3:

```
float dp3(vec4,vec4);
float dp3(vec3,vec3);
int dp3(ivec4,ivec4);
int dp3(ivec3,ivec3);
```

- rsq:

```
float rsq(float);
float rsq(int);
```

Loops

While loops are allowed in the language grammar. They follow the exact same syntax like C 'while' loops. However, we have no way of supporting 'while' loops in our assembly language. Therefore, you do not have to worry about them in parts 3 and 4 of the lab. However, you do have to implement them in the Lab 1 scanner and Lab 2 parser.

Language Grammar

The project source language is case sensitive. Tokens may be separated by blanks, comments, or line boundaries. An identifier or keyword must be separated from a following identifier, keyword, or integer; in all other cases, tokens need not be separated. As the example indicates, quotation marks appearing inside text are denoted by pairs of quotation marks. Comments can be continued across a line boundary, but no other token can. Each identifier(except the predefined variables) must be declared before it is used. The precedence and associativity of operators is shown in Table 1.

Precedence	Operator	Associativity
0	!	Unary
1	^	Right-associative
2	* /	Left-associative
3	+ -	Left-associative
4	== != < <= > >=	Non-associative
5	&&	Left-associative
6	|||	Left-associative

Table 1: MiniGLSL operator associativity

Examples of the tokens are listed in Table 2.

Type	Example
Identifier	AM A1 A_B
Integer Literal	0 32767
Float Literal	0.0 17.45
Comment	/* comments are bracketed by * and / */

Table 2: MiniGLSL Token Example

Below is the full description on the MiniGLSL Program Grammar. Notes:

- Terminal symbols are enclosed in single quote marks (' ').
- Alternatives within each rule are separated by commas.
- The construct (thing1 thing2 ...) indicates a grouping of things.
- The construct thing? means zero or one occurrence of thing.
- The construct thing* means zero or more occurrences of thing.
- Statements enclosed between /* */ are comments.

```
program:
    scope
statement:
    /* empty */ ';',
    variable '=' expression ';' ,
    'if' '(' expression ')' statement ( 'else' statement )? ,
    'while' '(' expression ')' statement ,
    scope
scope:
    '{'declaration* statement* '}'
expression:
    constructor,
    function,
```

```
    integer,
    float,
    '-' expression,
    expression '+' expression,
    expression '-' expression,
    expression '*' expression,
    expression '/' expression,
    expression '^' expression,
    'true',
    'false',
    '!' expression,
    expression '&&' expression,
    expression '||' expression,
    expression '==' expression,
    expression '!=' expression,
    expression '<' expression,
    expression '<=' expression,
    expression '>' expression,
    expression '>=' expression,
    '(' expression ')',
    variable
declaration:
    type identifier ';',
    type identifier '=' expression ';',
    'const' type identifier '=' expression ';',
    declaration declaration
type:
    'int',
    'bool',
    'float',
    'vec2',
    'vec3',
    'vec4',
    'bvec2',
    'bvec3',
    'bvec4',
    'ivec2',
    'ivec3',
    'ivec4'
constructor:
    type'(' arguments ')',
    funcName'('arguments_opt')'
funcName:
    'dp3',
    'lit',
    'rsq'
arguments:
    expression,
    arguments ',' arguments
arguments_opt:
    arguments*,
variable:
    variablename ,
    arrayname '['<integer literal>']'
```

```
variablename: identifier
arrayname: identifier
```