Programming Project 2: Syntax Analysis

A subset of GLSL grammar

1. Tokens

The full list of tokens and grammar are highlighted in GLSL spec Chapter 10. The overall structure of a GLSL program is also shown at the beginning of Chapter 6.

a. TYPE_NAME is for user defined type. Not required for this assignment. So you can safely remove it from your token list.

b. Here is an example of tokens with corresponding lexemes:

```
"void"                  { return T_Void;        }
"int"                   { return T_Int;         }
"float"                 { return T_Float;       }
"bool"                  { return T_Bool;        }
"while"                 { return T_While;       }
"for"                   { return T_For;         }
"if"                    { return T_If;          }
"else"                  { return T_Else;        }
"return"                { return T_Return;      }
"break"                 { return T_Break;       }
"switch"                { return T_Switch;      }
"case"                  { return T_Case;        }
"default"               { return T_Default;     }
"do"                    { return T_Do;          }
"vec2"                  { return T_Vec2;        }
"vec3"                  { return T_Vec3;        }
"vec4"                  { return T_Vec4;        }
"mat2"                  { return T_Mat2;        }
"mat3"                  { return T_Mat3;        }
"mat4"                  { return T_Mat4;        }

"("                     { return T_LeftParen;   }
")"                     { return T_RightParen;  }
":"                     { return T_Colon;       }
";"                     { return T_Semicolon;   }
"{"                     { return T_LeftBrace;   }
"}"                     { return T_RightBrace;  }
"."                     { BEGIN(FIELDS); return T_Dot; }
"["                     { return T_LeftBracket; }
"]"                     { return T_RightBracket;}

"<="                    { return T_LessEqual;   }
">="                    { return T_GreaterEqual;}
"=="                    { return T_EqualOp;     }
"!="                    { return T_NotEqual;    }
"&&"                    { return T_AndOp;       }
"||"                    { return T_OrOp;        }
"++"                    { return T_Inc;         }
"--"                    { return T_Dec;         }
"+"                     { return T_Plus;        }
"-"                     { return T_Dash;        }
```

```
"*"                    { return T_Star;        }
"/"                    { return T_Slash;       }
"+="                   { return T_AddAssign;   }
"-="                   { return T_SubAssign;   }
"*="                   { return T_MulAssign;   }
"/="                   { return T_DivAssign;   }
"="                    { return T_Equal;       }
">"                    { return T_LeftAngle;   }
"<"                    { return T_RightAngle;  }
```

2. **Functions** (see Chapter 6 and 6.1 for details)

   GLS has a function prototype. The function definition can follow with actual implementation.

   ```
   int foo(); // prototype
   …
   int foo() {
     // actual implementation here
   }
   ```

   This shouldn't be treated as a function redeclaration error during semantic analysis.

   There is only one "**`main`**" function in the GLSL program. Functions don't allow a variable number of arguments (that is, no printf("format", x, …)-like functions).

   Function body can contain any regular C-style statements, such as "for", "while", "if-else" statements.

3. **Global variables** (Chapter 6 for grammar)

   GLSL can have global declarations, such as a global variable, uniform, input or output. In this assignment, "uniform" means its value doesn't change throughout this program (similar to "const" qualifier in C).

   We only test global declarations with basic types, e.g. int, float, vectors, or matrix. We don't require global declarations with array, or other composite types.

   A global variable can have an optional initializer, which must be a compile-time constant or expression.

   ```
   float a;            <<  ok
   float b = 1.0;      <<< ok
   …
   void main() { … }
   ```

NOTE:

a) In this assignment, each declaration only has a single variable or statement. We will NOT have any tests with a sequence of declarations or initializations. So you don't need to handle the cases like:
   ```
   int x, y;
   float a = 1.0, b = 2.0;
   ```
   But we will have cases like:
   ```
   int   x;
   float a = 1.0;
   float b = 2.0;
   ```

b) We will NOT have any tests with only type declaration (i.e. no identifier). For example
   ```
   int;
   float ;
   uniform float;
   ```

c)

4. **Vector and component selection** (Chapter 5.5)

For a vector type variable, we can select its component, via .xyzw (called swizzle). For example,
```
     vec4  myVar;
```
we can have
```
     myVar.x
```
which selects the first component. Similary, .y is the second, .z is the third, and .w is the fourth component.

In this assignment, we only allow swizzle to be a combination of "xyzw". It is an error to use any other combinations, such as "rgba".

If it is a vec3 type, you can only use up to "xyz", there is no fourth component (no .w allowed on vec3). Similarly, there are no .zw for vec2 type. Report an error if this is the case.

Using a swizzle to select components, we can form a new value. For example,
```
   void main() {
     vec4 myVar;
     vec2 a = myVar.xy;    // pick first/second components
     vec2 b = myVar.xz;    // pick first/third components
   }
```

5. **Operators** (Chapter 5)

We have several operators (highlighted) such as “++”, “--", “+”, “-“, “*”, “/”, and two-letter operators “+=”, “-=”, “*=”, and “/=”. We will not test other GLSL operators, (e.g. %=, &=, >>, or <<=). They won’t show in the test program, so you don’t need to worry about the error handling for them.

The full list of operators is highlighted in Chapter 5.1.

6. **Constructors** (Chapter 5.4.2)

We only need vector constructors with scalar values. For example,

```
vec2 a  = vec2(0.0);               <<< yes
vec3 b  = vec3(1.0);               <<< yes

vec3 c  = vec3(vec2(4.0, 5.0), 6.0);  <<< won't test
mat2 e  = mat2(vec2(1.0, 2.0),
               vec2(3.0, 4.0));       <<< won't test
```

The second part of statements above won’t show up in the test program, so no need to worry about error handling for them.

7. **Error recovery**

In this assignment, you only need to report the first error and stop parsing.

You may explore more advanced error recovery in this assignment for extra credit.

More error recovery will be done in the next assignment (semantic analysis).

8. **Types**

In ast_type.h, we have added new types from GLSL, such as vec2Type, mat3Type, etc. Then, in ast_type.cc, add initialization for these new types (you should pass the GLSL native type names as Type .ctor argument). For example,

```
Type Type::vec2Type = new Type("vec2");
Type Type::vec3Type = new Type("vec3");
Type Type::vec4Type = new Type("vec4");
Type Type::mat2Type = new Type("mat2");
Type Type::mat3Type = new Type("mat3");
Type Type::mat4Type = new Type("mat4");
```