Lab 2: Parser

Description You are required to build a parser (using the Bison parser generator) that accepts the language generated by the MiniGLSL grammar (see below). Your parser must implement the "trace parser" functionality (`-Tp` command-line flag) of the compiler (see the man page for details; `make man` in the starter directory). No semantic analysis or AST construction need be performed. If you do semantic analysis or AST construction, then be sure to disable it in your submission.

Starter Files You will be provided with a complete implementation of scanner.l, as well as a dummy parser.y files. You are free to use your implementation of scanner.l from Lab 1. Note: not all lexical rules needed for Lab 1 are needed for Lab 2, as such, some modification of your scanner from Lab 1 will be necessary.

Language Grammar Below is an (ambiguous) context-free grammar generating the MiniGLSL language:

$$
\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{scope} \\
\mathit{scope} & \rightarrow & '\{' \ \mathit{declarations}\ \mathit{statements}\ '\}' \\
\mathit{declarations} & \rightarrow & \mathit{declarations}\ \mathit{declaration} \\
 & \rightarrow & \epsilon \\
\mathit{statements} & \rightarrow & \mathit{statements}\ \mathit{statement} \\
 & \rightarrow & \epsilon \\
\mathit{declaration} & \rightarrow & \mathit{type}\ \langle identifier \rangle\ ';' \\
 & \rightarrow & \mathit{type}\ \langle identifier \rangle\ '='\ \ \mathit{expression}\ ';' \\
 & \rightarrow & 'const'\ \mathit{type}\ \langle identifier \rangle\ '='\ \ \mathit{expression}\ ';' \\
 & \rightarrow & \epsilon \\
\mathit{statement} & \rightarrow & \mathit{variable} =' \ \ \mathit{expression}\ ';' \\
 & \rightarrow & 'if'\ '('\ \mathit{expression}\ ')'\ \mathit{statement}\ \mathit{else\_statement} \\
 & \rightarrow & 'while'\ '('\ \mathit{expression}\ ')'\ \mathit{statement} \\
 & \rightarrow & \mathit{scope}
\end{array}
$$

$$
\begin{array}{rcl}
 & \rightarrow & ';' \\
\mathit{else\_statement} & \rightarrow & 'else'\ \mathit{statement} \\
 & \rightarrow & \epsilon \\
\mathit{type} & \rightarrow & 'int' \mid 'ivec2' \mid 'ivec3' \mid 'ivec4' \\
 & \rightarrow & 'bool' \mid 'bvec2' \mid 'bvec3' \mid 'bvec4' \\
 & \rightarrow & 'float' \mid 'vec2' \mid 'vec3' \mid 'vec4' \\
\mathit{expression} & \rightarrow & \mathit{constructor} \\
 & \rightarrow & \mathit{function} \\
 & \rightarrow & \langle \mathit{integer\ literal} \rangle \\
 & \rightarrow & \langle \mathit{floating\ point\ literal} \rangle \\
 & \rightarrow & \mathit{variable} \\
 & \rightarrow & \mathit{unary\_op}\ \mathit{expression} \\
 & \rightarrow & \mathit{expression}\ \mathit{binary\_op}\ \mathit{expression} \\
 & \rightarrow & 'true' \mid 'false' \\
 & \rightarrow & '('\ \mathit{expression}\ ')' \\
\mathit{variable} & \rightarrow & \langle \mathit{identifier} \rangle \\
 & \rightarrow & \langle \mathit{identifier} \rangle\ '['\ \langle \mathit{integer\ literal} \rangle\ ']' \\
\mathit{unary\_op} & \rightarrow & '!' \mid '-' \\
\mathit{binary\_op} & \rightarrow & '\&\&' \mid '||' \mid '==' \mid '!=' \mid '<' \mid '<=' \\
 & \rightarrow & '>' \mid '>=' \mid '+' \mid '-' \mid '*' \mid '/' \mid '\hat{}' \\
\mathit{constructor} & \rightarrow & \mathit{type}\ '('\ \mathit{arguments}\ ')' \\
\mathit{function} & \rightarrow & \mathit{function\_name}\ '('\ \mathit{arguments\_opt}\ ')' \\
\mathit{function\_name} & \rightarrow & 'dp3' \mid 'lit' \mid 'rsq' \\
\mathit{arguments\_opt} & \rightarrow & \mathit{arguments} \mid \epsilon \\
\mathit{arguments} & \rightarrow & \mathit{arguments}\ ','\ \mathit{expression} \mid \mathit{expression}
\end{array}
$$

Operator Precedence Below is an operator precedence table for MiniGLSL.

Precedence	Operators		Associativity
0 (highest)	[ ]	vector subscript	left-to-right
	( )	function call	
	( )	constructor call	
1	-	arithmetic negate	
	!	logical negate	
2	^	exponentiation	right-to-left
3	*	multiply	
	/	divide	
4	+	add	left-to-right

Precedence	Operators		Associativity
	-	subtract	
5	==	equal to	
	!=	not equal to	
	<	less than	
	<=	less than or equal to	
	>	greater than	
	>=	greater than or equal to	
6	&&	Boolean AND	
7 (lowest)	||	Boolean OR	

Left-to-right associativity implies that $1+2+3$ is interpreted as $(1+2)+3$.
Right-to-left associativity implies that 1^2^3 is interpreted as 1^(2^3).
High precedence implies that $-1-2$ is interpreted as $(-1)-2$.

Miscellaneous Language Details

Integers Valid integers fall in the open range $(-2^{21},\ 2^{21})$. Integers are base 10, and are generated with the following regular expression: $0|[1-10][0-10]^*$.

Floats Valid floating point numbers are exactly those represented by the C `float` type. Floats are generated with either of the following regular expressions: $(0|[1-10][0-10]^*).[0-10]^*$ or $.[0-10]^+$, where '.' is a decimal point.

Identifiers Valid identifiers must be no more than 32 characters in length and are generated by the following regular expression: $[a-zA-Z\_][a-zA-Z0-9\_]^*$.

Documentation You must document your code (excessive documentation is not necessary), and provide a written report detailing the breakdown of work, challenges faced, overall approach taken, how your group verified the parser, and (optional) anything interesting that your group would like to highlight.

Testing It is suggested that you test your parser using tests that are expected to pass and fail. Submitting tests in a subdirectory will be appreciated.

Hint Don't try to make your parser too smart. *The parser is responsible for syntax only.* That is, the parser can accept semantically invalid programs, so long as they are syntactically correct. The implication here is that the parser accepts a "bigger" language than MiniGLSL, i.e. every string generated by the above grammar (not all of which can be compiled).

For example, the expression '`1+true`' is semantically invalid, but syntactically correct.