Build parser using Bison parser generator

=> Accepts language generated by mini GLSL

Implement true parser functionality of the compiler

make man in starter

No semantic Analysis / AST required for now

Complete implementation of Scanner.l, dummy parser.y file

CFG for Mini GLSL

```
program → Scope
  scope → '{' declaration statements '}'
  declarations → declarations declaration
               → ϵ
  statements → Statements statement
             → ϵ

  declaration → type <identifier> ';'
              → type <identifier> '=' expression ';'
              → 'const' type <identifier> '=' expression ';'
              → ϵ

  Statement → Variable = ' expression ';'
            → 'if' '(' expression ')' statements else_statement
            → 'while' '(' expression ')' statement
            → Scope
            → ';'

  else_statement → 'else' statement
                 → ϵ
```

```
type → 'int' | 'ivec2' | 'ivec3' | 'ivec4'
     → 'bool' | 'bvec2' | 'bvec3' | 'bvec4'
     → 'float' | 'vec2' | 'vec3' | 'vec4'
expressions → constructor
            → function
            → <integer literal>
            → floating point literal
            → Variable
            → unary_op_expression
            → expression binary_op expression
            → 'true' | 'false'
            → '(' expression ')'

variable → <identifier>
         → <identifier> '[' <integer literal> ']'
unary_op → '!' | '-'
binary_op → '&&' | '||' | '==' | '!=' | '<' | '<=' | '>' | '>=' | '+' | '-' | '*' | / | '^' |
Constructor → type '(' arguments ')'
function → function_name '(' arguments_opt ')'
function_name → 'dp3' | 'lit' | 'rsq'
arguments_opt → arguments | ϵ
arguments → arguments ',' expression | expression
```

Precedence

0	[] () ()		L to R
1	-	!	
2	^		R to L
3	*	/	
4	+	-	
5	== != < <= > >=		
6	&&	AND	
7	||	OR	

Integers -2^{21}, 2^{21} base 10 → 0|[1–10][0–10]*

Floats .[0–10]+

Identifiers [a-z A-Z _][a-z A-Z 0-9]*

Parser is only responsible for syntax only

Implement trace scanner functionality -ln switch

Build lexical analyzer Mini GLSO

make man in starter dir

Compiler 467.c

Makefile

scanner.l - skeleton flex scanner

Parser.y - skeleton bison parser

Compiler 467.man - man page / manual

globalvars.C - global vars

common.h - global definitions

Store appropriate info for each token

identified in a tracing file, errors in error file

when ln on, global var traceScanner is TRUE

Send traces to global traceFile FILE*

Use yTRACE(x) to output the trace

output tokens in same order that they appear in the program

Report error on illegal input

=> yERROR(x) to report

check for out of bounds integers & identifiers that exceed the allowed length

Define tokens in parser.y file

Bison only takes the tokens and defines them in a header file - parser.tab.h

Associate info with tokens to store important information

yyval union

Bison defines yyval to be %union type from parser.y file

Use yyval in scanner.l to store the value of each token

Parser - ask Scanner for new tokens

Add new tokens defined to grammar in

parser.y

What is Lex / Flex

Takes .l file => generates a lexical analyzer

implemented in C   lex.yy.c

Lex uses reg expressions

header

%%

body

%%

helper functions

Header Section

place header file

```
%{
    #include "helper.h"
%}
```

place definitions

Syntax <name> <definitions>

DIGIT   [0-9]

Body Section

Lexical rules placed here

<rule> <action>

[0-9]+

{DIGIT}+ "." {DIGIT}*

Helper Section

C functions here

Lex func & Variables

char * yytext - current matched text

int yyleng - length of matched text

int yylex() - scanner function

char input() - get next char in input stream

int yyterminate() - terminate scanner - return 0

Scanner - breaks input into tokens

parser - checks grammar & builds structure from those tokens

Scanner.l

comment handler /*

catch all error → { yyERROR(...) }

yyTRACE

Add tokens def in parser.y

Write regex patterns in scanner.l to recognize each token

Return correct token, call yyTRACE

Regex - [0-9]+ for INT_C

Flex matches pattern in order- it error pattern comes first, it Catch everything

Parser.y => Token declarations %token INT_C - read char, recognize pattern, return token types

Scanner.l => pattern + action {DIGIT} + {... functions} → receive tokens from Scanner & check grammar

C style comments works for both

Flex rules

Flex matches longest possible string

when lengths are equal, first rule wins

Keywords come BEFORE identifiers ★

Float comes BEFORE int checks ★

Key words
Bool
Type keywords
operators
Float
Integers
Identifiers
Single char tokens
white space
New line

Token # - Bison assigns to each token

when Bison processes parser.y - it generates parser.tab.h

Defines each token as an integer constant

Single char tokens use their ASCII values input & output FILE defined

yyparse() → yylex()

getopts → parse command line arguments

fileOpen

SourceDump

Scanner.l

include "parser.tab.h"

yyin = inputFile

flex input → yyinput

yTRACE

yERROR → yyterminate

yyline=1 initially

%option noyywrap

%% pattern {action} %%

"string" { program }

IF defined in parser.tab.h

. matches any single char

=> catch all

Define tokens in parser.y

Assign each token an integer => Scanner returns these values

Grammar rules

parser.out => LR parser state machine

Bison → Parser.y => Parser.c

parser.tab.h

Scanner.l => Scanner.c

.c → .o

→ compiler program