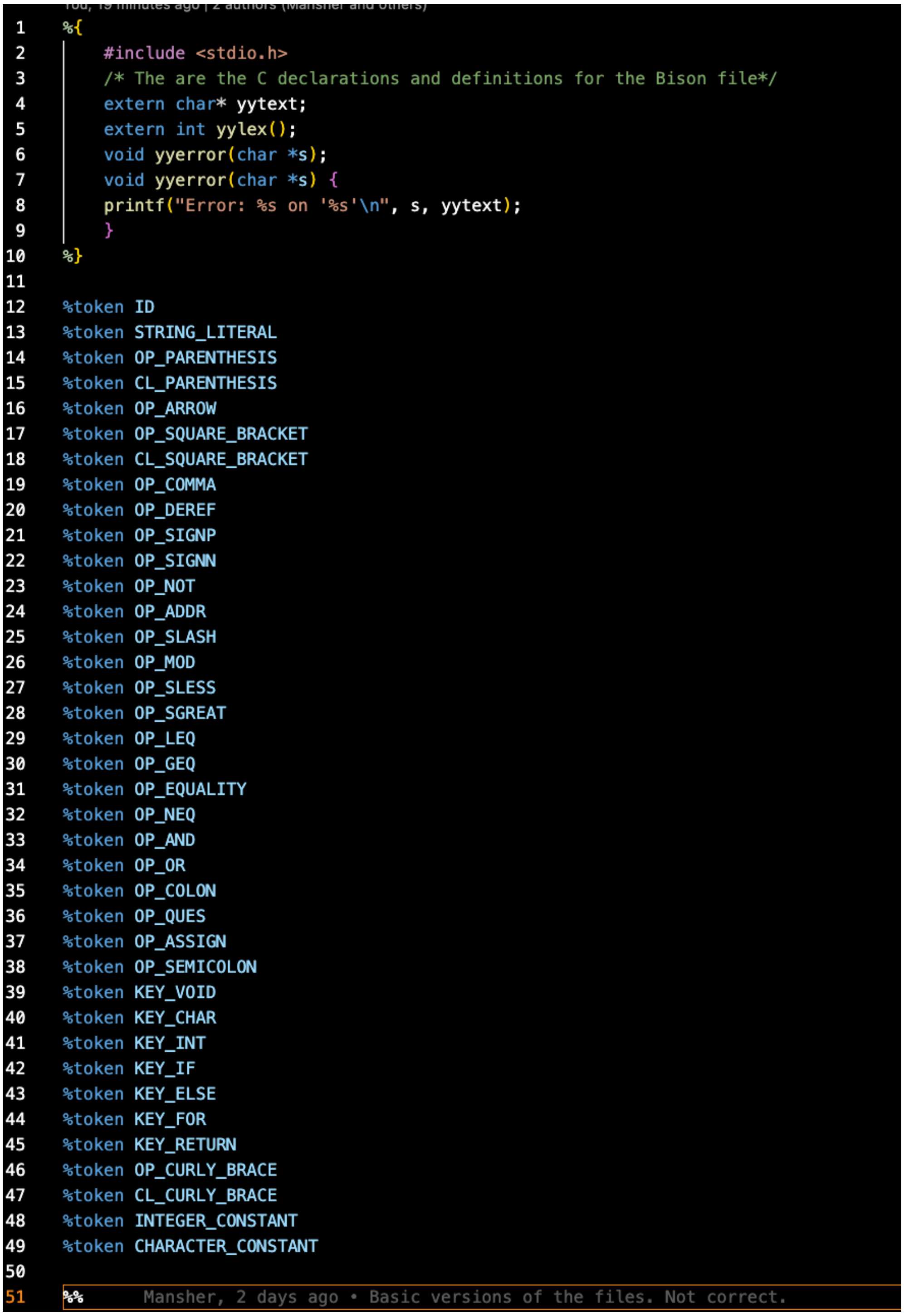
Assignment 3 - Parser for nanoC - CS-1319-1

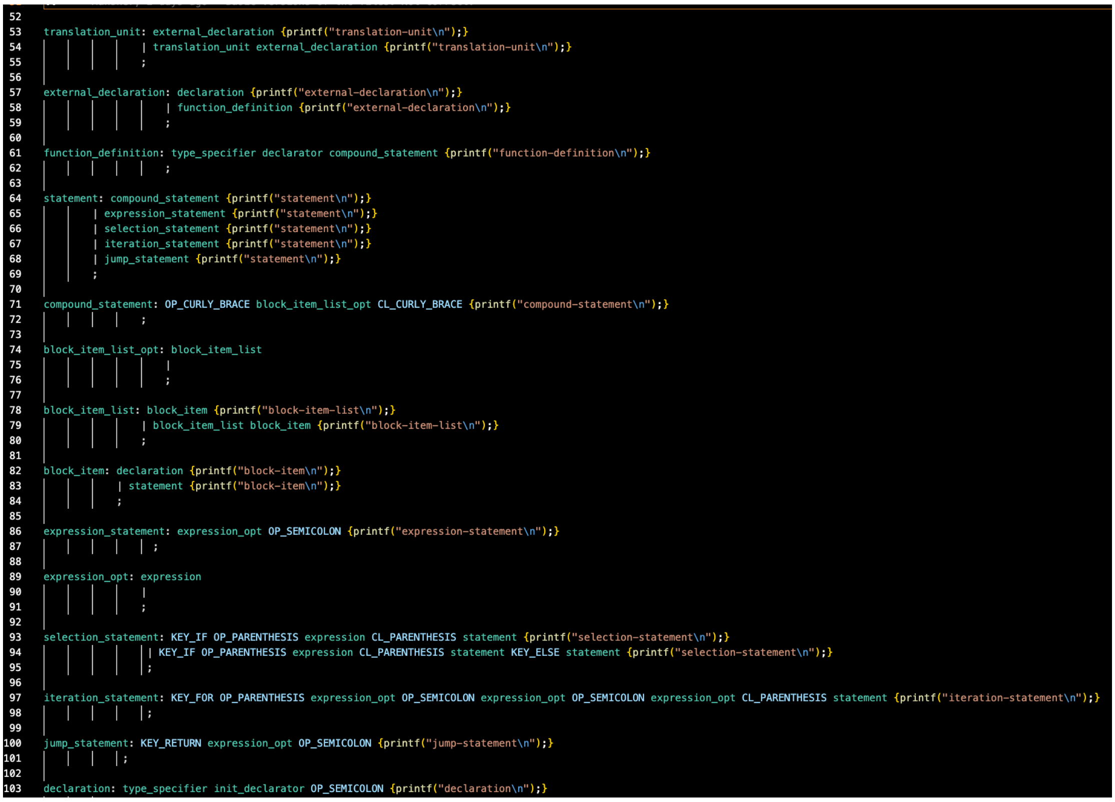
Team: partha
Members: Advithiya Jain, Mansher Singh
Date: 28|10|23

Parser (.y file):



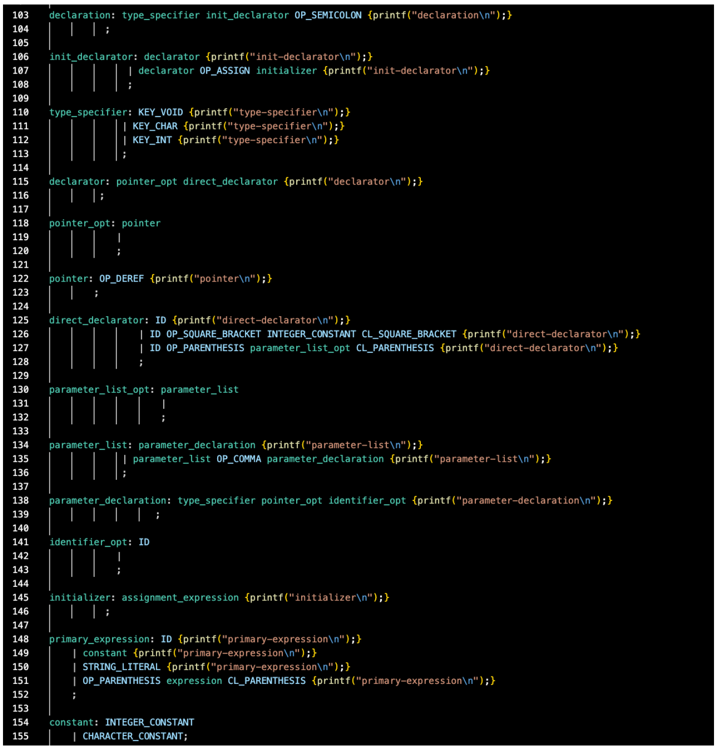
```
You, 19 minutes ago | 2 authors (Mansher and others)
%{
    #include <stdio.h>
    /* The are the C declarations and definitions for the Bison file*/
    extern char* yytext;
    extern int yylex();
    void yyerror(char *s);
    void yyerror(char *s) {
    printf("Error: %s on '%s'\n", s, yytext);
    }
%}

%token ID
%token STRING_LITERAL
%token OP_PARENTHESIS
%token CL_PARENTHESIS
%token OP_ARROW
%token OP_SQUARE_BRACKET
%token CL_SQUARE_BRACKET
%token OP_COMMA
%token OP_DEREF
%token OP_SIGNP
%token OP_SIGNN
%token OP_NOT
%token OP_ADDR
%token OP_SLASH
%token OP_MOD
%token OP_SLESS
%token OP_SGREAT
%token OP_LEQ
%token OP_GEQ
%token OP_EQUALITY
%token OP_NEQ
%token OP_AND
%token OP_OR
%token OP_COLON
%token OP_QUES
%token OP_ASSIGN
%token OP_SEMICOLON
%token KEY_VOID
%token KEY_CHAR
%token KEY_INT
%token KEY_IF
%token KEY_ELSE
%token KEY_FOR
%token KEY_RETURN
%token OP_CURLY_BRACE
%token CL_CURLY_BRACE
%token INTEGER_CONSTANT
%token CHARACTER_CONSTANT

%%      Mansher, 2 days ago • Basic versions of the files. Not correct.
```

We first declared all the terminals that would be used in the phrase grammar of nanoC. They have to be declared in new lines otherwise an error gets thrown where the file is unable to read the tokens that we have specified. These are the same tokens that the lexer will be identifying while going through the code and using this parser's header file to recognize everything.

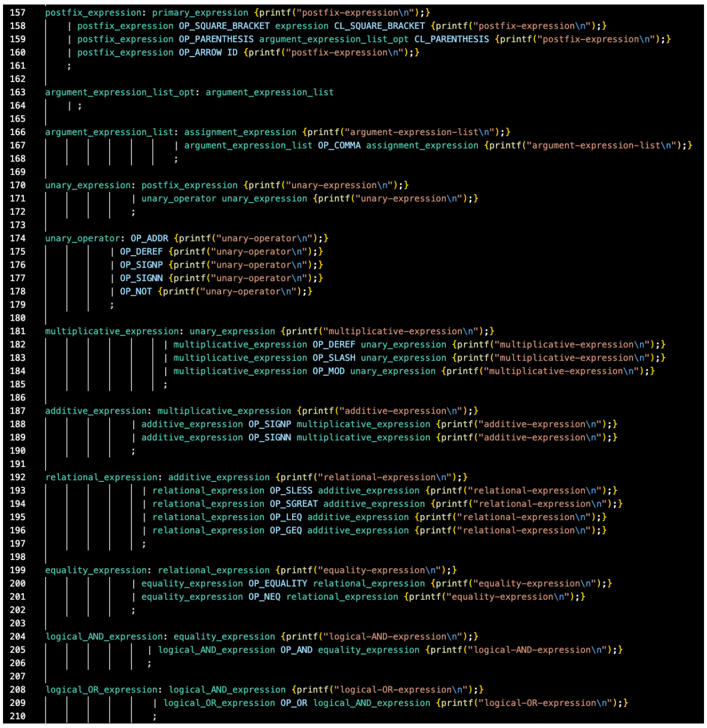


```

translation_unit: external_declaration {printf("translation-unit\n");}
                | translation_unit external_declaration {printf("translation-unit\n");}
                ;

external_declaration: declaration {printf("external-declaration\n");}
                    | function_definition {printf("external-declaration\n");}
                    ;

function_definition: type_specifier declarator compound_statement {printf("function-definition\n");}
                    ;

statement: compound_statement {printf("statement\n");}
          | expression_statement {printf("statement\n");}
          | selection_statement {printf("statement\n");}
          | iteration_statement {printf("statement\n");}
          | jump_statement {printf("statement\n");}
          ;

compound_statement: OP_CURLY_BRACE block_item_list_opt CL_CURLY_BRACE {printf("compound-statement\n");}
                ;

block_item_list_opt: block_item_list
                    |
                    ;

block_item_list: block_item {printf("block-item-list\n");}
                | block_item_list block_item {printf("block-item-list\n");}
                ;

block_item: declaration {printf("block-item\n");}
            | statement {printf("block-item\n");}
            ;

expression_statement: expression_opt OP_SEMICOLON {printf("expression-statement\n");}
                    | ;

expression_opt: expression
                |
                ;

selection_statement: KEY_IF OP_PARENTHESIS expression CL_PARENTHESIS statement {printf("selection-statement\n");}
                    || KEY_IF OP_PARENTHESIS expression CL_PARENTHESIS statement KEY_ELSE statement {printf("selection-statement\n");}
                    |;

iteration_statement: KEY_FOR OP_PARENTHESIS expression_opt OP_SEMICOLON expression_opt OP_SEMICOLON expression_opt CL_PARENTHESIS statement {printf("iteration-statement\n");}
                    |;

jump_statement: KEY_RETURN expression_opt OP_SEMICOLON {printf("jump-statement\n");}
            |;

declaration: type_specifier init_declarator OP_SEMICOLON {printf("declaration\n");}
```

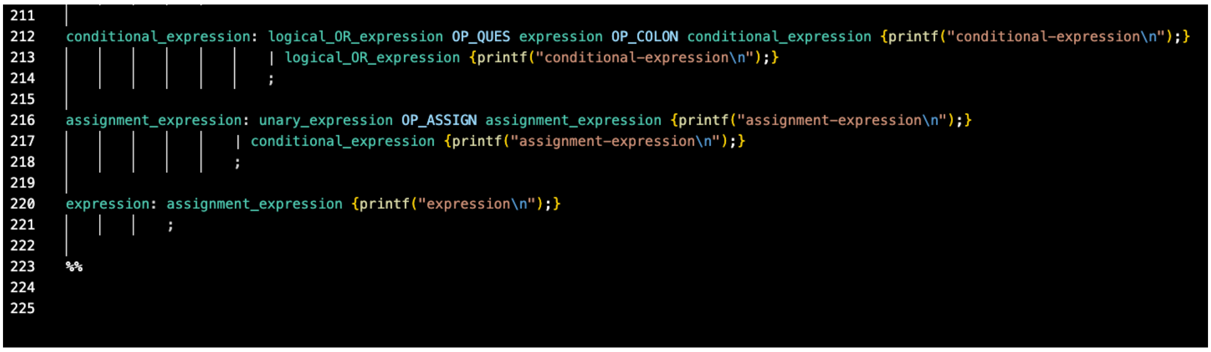
At first glance, the grammar is almost identical to the grammar specifications from the assignment file. However, we had to make changes to the order of the sections. For example, we had to specify that the translation unit (set of rules under that section of the assignment) had to be placed at the beginning of the rules specifications. This is because the bison file is read from top to bottom and understands the rules and states as such. Following any other order results in an error where some set (sometimes a very large set) of rules become classified as "useless rules". This also happens for "useless terminals". This occurs because the file is unable to reach a specific state when following the order of the grammar.



```
declaration: type_specifier init_declarator OP_SEMICOLON {printf("declaration\n");}
           ;

init_declarator: declarator {printf("init-declarator\n");}
               | declarator OP_ASSIGN initializer {printf("init-declarator\n");}
               ;

type_specifier: KEY_VOID {printf("type-specifier\n");}
              | KEY_CHAR {printf("type-specifier\n");}
              | KEY_INT {printf("type-specifier\n");}
              ;

declarator: pointer_opt direct_declarator {printf("declarator\n");}
          ;

pointer_opt: pointer
           |
           ;

pointer: OP_DEREF {printf("pointer\n");}
       ;

direct_declarator: ID {printf("direct-declarator\n");}
                 | ID OP_SQUARE_BRACKET INTEGER_CONSTANT CL_SQUARE_BRACKET {printf("direct-declarator\n");}
                 | ID OP_PARENTHESIS parameter_list_opt CL_PARENTHESIS {printf("direct-declarator\n");}
                 ;

parameter_list_opt: parameter_list
                  |
                  ;

parameter_list: parameter_declaration {printf("parameter-list\n");}
              | parameter_list OP_COMMA parameter_declaration {printf("parameter-list\n");}
              ;

parameter_declaration: type_specifier pointer_opt identifier_opt {printf("parameter-declaration\n");}
                     ;

identifier_opt: ID
              |
              ;

initializer: assignment_expression {printf("initializer\n");}
           ;

primary_expression: ID {printf("primary-expression\n");}
    | constant {printf("primary-expression\n");}
    | STRING_LITERAL {printf("primary-expression\n");}
    | OP_PARENTHESIS expression CL_PARENTHESIS {printf("primary-expression\n");}
    ;

constant: INTEGER_CONSTANT
    | CHARACTER_CONSTANT;
```

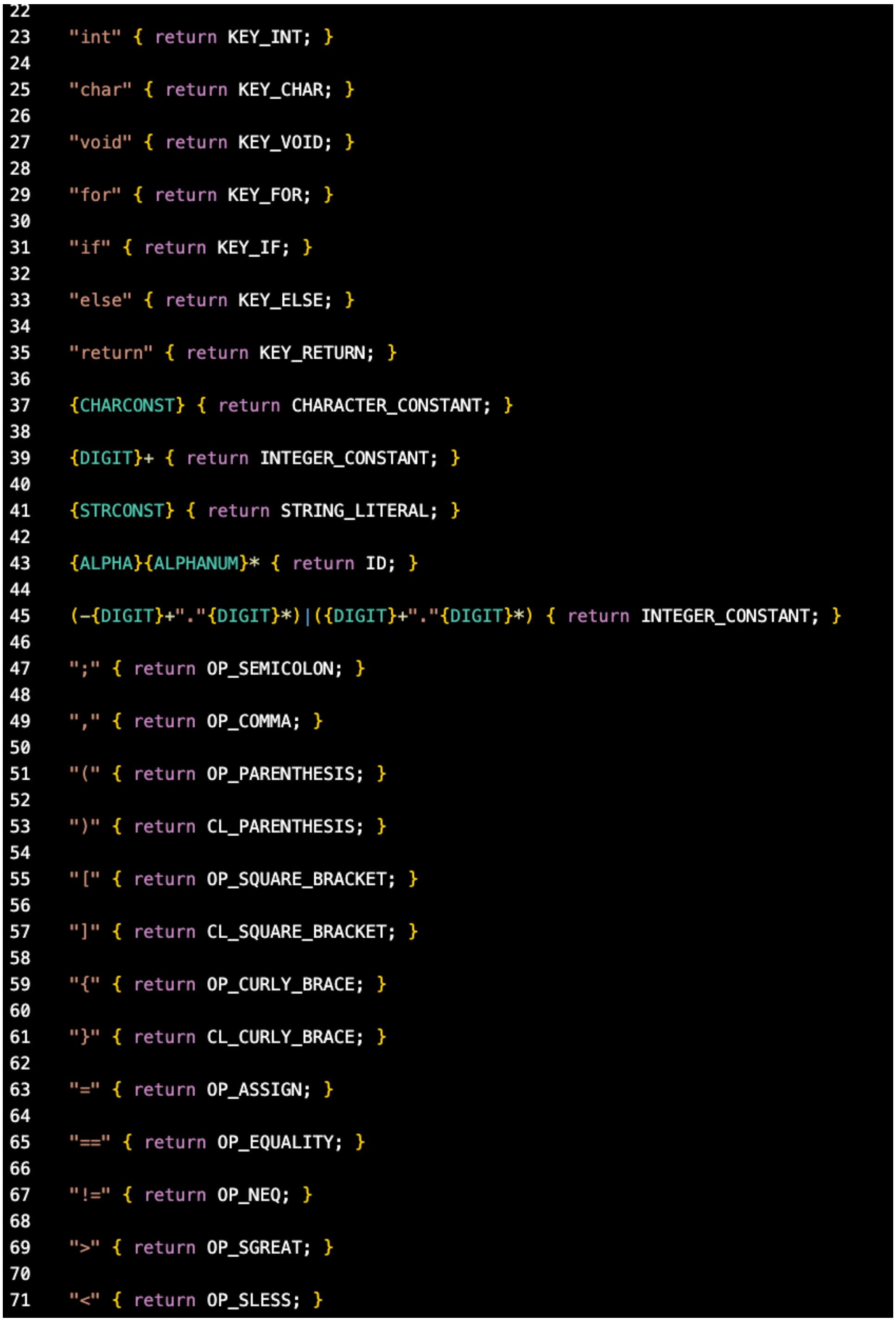


```
postfix_expression: primary_expression {printf("postfix-expression\n");}
    | postfix_expression OP_SQUARE_BRACKET expression CL_SQUARE_BRACKET {printf("postfix-expression\n");}
    | postfix_expression OP_PARENTHESIS argument_expression_list_opt CL_PARENTHESIS {printf("postfix-expression\n");}
    | postfix_expression OP_ARROW ID {printf("postfix-expression\n");}
    ;

argument_expression_list_opt: argument_expression_list
    | ;

argument_expression_list: assignment_expression {printf("argument-expression-list\n");}
                        | argument_expression_list OP_COMMA assignment_expression {printf("argument-expression-list\n");}
                        ;

unary_expression: postfix_expression {printf("unary-expression\n");}
                | unary_operator unary_expression {printf("unary-expression\n");}
                ;

unary_operator: OP_ADDR {printf("unary-operator\n");}
              | OP_DEREF {printf("unary-operator\n");}
              | OP_SIGNP {printf("unary-operator\n");}
              | OP_SIGNN {printf("unary-operator\n");}
              | OP_NOT {printf("unary-operator\n");}
              ;

multiplicative_expression: unary_expression {printf("multiplicative-expression\n");}
                         | multiplicative_expression OP_DEREF unary_expression {printf("multiplicative-expression\n");}
                         | multiplicative_expression OP_SLASH unary_expression {printf("multiplicative-expression\n");}
                         | multiplicative_expression OP_MOD unary_expression {printf("multiplicative-expression\n");}
                         ;

additive_expression: multiplicative_expression {printf("additive-expression\n");}
                   | additive_expression OP_SIGNP multiplicative_expression {printf("additive-expression\n");}
                   | additive_expression OP_SIGNN multiplicative_expression {printf("additive-expression\n");}
                   ;

relational_expression: additive_expression {printf("relational-expression\n");}
                     | relational_expression OP_SLESS additive_expression {printf("relational-expression\n");}
                     | relational_expression OP_SGREAT additive_expression {printf("relational-expression\n");}
                     | relational_expression OP_LEQ additive_expression {printf("relational-expression\n");}
                     | relational_expression OP_GEQ additive_expression {printf("relational-expression\n");}
                     ;

equality_expression: relational_expression {printf("equality-expression\n");}
                   | equality_expression OP_EQUALITY relational_expression {printf("equality-expression\n");}
                   | equality_expression OP_NEQ relational_expression {printf("equality-expression\n");}
                   ;

logical_AND_expression: equality_expression {printf("logical-AND-expression\n");}
                      | logical_AND_expression OP_AND equality_expression {printf("logical-AND-expression\n");}
                      ;

logical_OR_expression: logical_AND_expression {printf("logical-OR-expression\n");}
                      | logical_OR_expression OP_OR logical_AND_expression {printf("logical-OR-expression\n");}
                      ;
```



```

conditional_expression: logical_OR_expression OP_QUES expression OP_COLON conditional_expression {printf("conditional-expression\n");}
                      | logical_OR_expression {printf("conditional-expression\n");}
                      ;

assignment_expression: unary_expression OP_ASSIGN assignment_expression {printf("assignment-expression\n");}
                     | conditional_expression {printf("assignment-expression\n");}
                     ;

expression: assignment_expression {printf("expression\n");}
          ;

%%
```

Lexer (.l file):

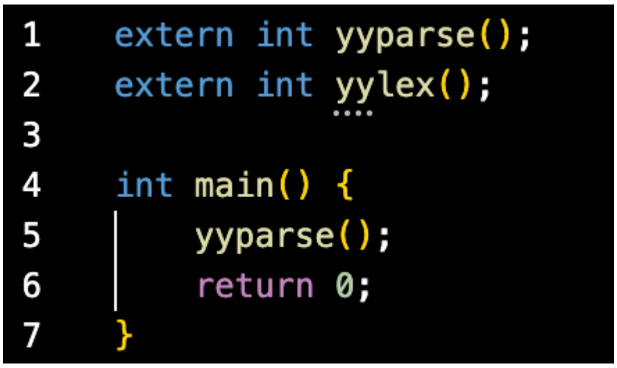


```

"int" { return KEY_INT; }

"char" { return KEY_CHAR; }

"void" { return KEY_VOID; }

"for" { return KEY_FOR; }

"if" { return KEY_IF; }

"else" { return KEY_ELSE; }

"return" { return KEY_RETURN; }

{CHARCONST} { return CHARACTER_CONSTANT; }

{DIGIT}+ { return INTEGER_CONSTANT; }

{STRCONST} { return STRING_LITERAL; }

{ALPHA}{ALPHANUM}* { return ID; }

(-{DIGIT}+"."{DIGIT}*)|({DIGIT}+"."{DIGIT}*) { return INTEGER_CONSTANT; }

";" { return OP_SEMICOLON; }

"," { return OP_COMMA; }

"(" { return OP_PARENTHESIS; }

")" { return CL_PARENTHESIS; }

"[" { return OP_SQUARE_BRACKET; }

"]" { return CL_SQUARE_BRACKET; }

"{" { return OP_CURLY_BRACE; }

"}" { return CL_CURLY_BRACE; }

"=" { return OP_ASSIGN; }

"==" { return OP_EQUALITY; }

"!=" { return OP_NEQ; }

">" { return OP_SGREAT; }

"<" { return OP_SLESS; }
```

We had to almost entirely change out lexer from the previous assignment, by making it return the correct terminal that the bison .y file generates in the y.tab.h file (6_A3.tab.h in this case). We had to make these changes so that the parser would get the correct terminals from the lexer.

Main (.c file):

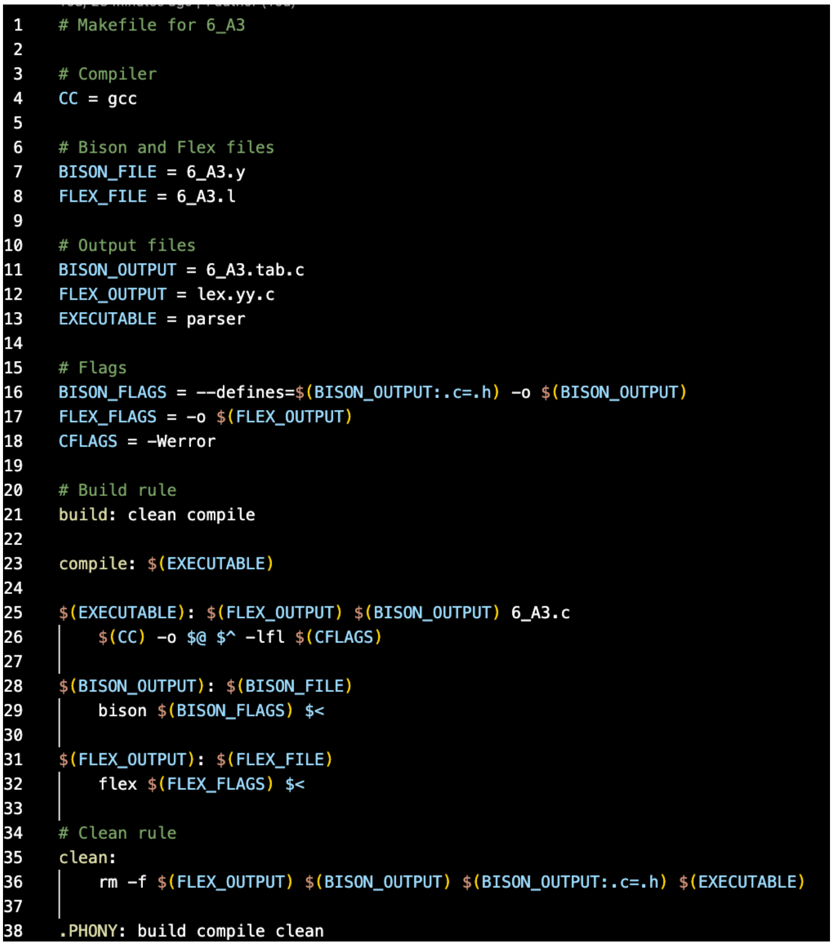
The main file is very simple and just calls yyparse().



```c
extern int yyparse();
extern int yylex();

int main() {
    yyparse();
    return 0;
}
```

Makefile

The Makefile has 3 total rules: compile build and clean.
The compile rule compiles the parser executable.
The clean rule cleans up the lex.yy.c, 6_A3.tab.h, 6_A3.tab.c, and parser files.
The build rule cleans, and then recompiles the executable forcibly every time it is called using ‘make build’



```makefile
# Makefile for 6_A3

# Compiler
CC = gcc

# Bison and Flex files
BISON_FILE = 6_A3.y
FLEX_FILE = 6_A3.l

# Output files
BISON_OUTPUT = 6_A3.tab.c
FLEX_OUTPUT = lex.yy.c
EXECUTABLE = parser

# Flags
BISON_FLAGS = --defines=$(BISON_OUTPUT:.c=.h) -o $(BISON_OUTPUT)
FLEX_FLAGS = -o $(FLEX_OUTPUT)
CFLAGS = -Werror

# Build rule
build: clean compile

compile: $(EXECUTABLE)

$(EXECUTABLE): $(FLEX_OUTPUT) $(BISON_OUTPUT) 6_A3.c
	$(CC) -o $@ $^ -lfl $(CFLAGS)

$(BISON_OUTPUT): $(BISON_FILE)
	bison $(BISON_FLAGS) $<

$(FLEX_OUTPUT): $(FLEX_FILE)
	flex $(FLEX_FLAGS) $<

# Clean rule
clean:
	rm -f $(FLEX_OUTPUT) $(BISON_OUTPUT) $(BISON_OUTPUT:.c=.h) $(EXECUTABLE)

.PHONY: build compile clean
```