Parser

B11130038 Jia-Hong, Wang

Introduction

This is a compiler implement by lex and bison which can scan, parse and generate code from qv (similar to kotlin) code to c code. In qv language, there are some special expression in qv. In array addition, it will do Dimension-wise addition. In array multiplication, it will do inner product.

Context-Free Grammar (CFG)

```
1.  program ->
      function_definition
      | ε
2.  function_definition ->
      FUN IDENTIFIER '(' ')' '{' stmts '}'
      | FUN MAIN '(' ')' '{' stmts '}'
3.  stmts ->
      stmt stmts
      | ε
4.  stmt ->
      '{' stmts '}'
      | expr ';'
      | assign_stmt
      | print_stmt
      | RET expr ';'
      | NEWLINE
      | if_stmt
      | WHILE '(' expr ')' DO stmt
      | FOR '(' expr ';' expr ';' expr ')' stmt
5.  print_stmt ->
      PRINT '(' expr ')' ';'
      | PRINTLN '(' expr ')' ';'
      | PRINT '(' STRING ')' ';'
      | PRINTLN '(' STRING ')' ';'
6.  if_stmt ->
      IF '(' expr ')' stmt
      | IF '(' expr ')' stmt ELSE stmt
7.  assign_stmt ->
      VAR ID    ':' basic_type ';'
      | VAR ID ':' basic_type '=' expr ';'
      | VAR ID    ':' basic_type '[' expr ']' '=' '{' value_list '}' ';'
      | VAR ID    ':' basic_type '[' expr ']' ';'
      | VAL ID    ':' basic_type ';'
      | VAL ID ':' basic_type '=' expr ';'
      | VAL ID    ':' basic_type '[' expr ']' '=' '{' value_list '}' ';'
      | VAL ID    ':' basic_type '[' expr ']' ';'
      | ID '=' expr ';'
      | ID '=' '{' value_list '}' ';'
8.  basic_type ->
      INT
      | REAL
```

```
    | BOOL
    | CHAR
9.  value_list ->
    expr
    | value_list ',' expr
10. expr ->
    expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | expr EQJ expr
    | expr NE expr
    | expr LT expr
    | expr LE expr
    | expr GT expr
    | expr GE expr
    | '-' expr
    | '(' expr ')'
    | ID '[' expr ']'
    | ID '(' value_list ')'
    | value
11. value ->
    INTEGER
    | DOUBLE
    | CHARACTER
    | TRUE
    | FALSE
    | ID
```

main.h

This header file defines the essential data structures and classes used throughout the project. It includes various C++ libraries, declares an enumeration ‘Type’ for different data types, and defines classes like ‘TOTAL_TYPE’, ‘IntValue’, ‘RealValue’, and so forth, which are used to manage different types of data within the compiler.

11130038-scanner.l

This file is the Lex source file, which defines the rules for tokenizing the input source code. It specifies patterns for different tokens like identifiers, numbers, and keywords, and handles comments and strings. It also includes the header ‘main.h’ and communicates with the parser to send tokens.

11130038-parser.y

This is the Bison grammar file, where the syntax rules of the programming language are defined. It uses the tokens defined in the Lex file to form grammar rules that describe how different elements in the language are syntactically valid. It also includes functions for semantic analysis and constructs the output C code based on the input script. In the end, the output will be put in ‘output.c’.

The following will briefly describe what the functions do

1. yyerror: Outputs an error message indicating the type and location of the error encountered during parsing.
2. yylex: A tool generated by Flex, used to tokenize the input source code into symbols that Bison can parse.
3. yyparse: A function generated by Bison that controls the parsing process, interpreting input tokens according to the grammar rules defined in the .y file.
4. get_type_name: Returns a string representation of a Type enum value, which describes data types in the source language.
5. assign_initial: Initializes a variable with a specified type but without an initial value, creating an entry in the symbol table.
6. assign_expr: Initializes a variable with both a specified type and an initial value, updating the symbol table and generating corresponding C code.
7. create_symbol: Adds a new symbol to the symbol table, managing identifiers and their associated data types and values.
8. getValue: Converts and fetches the value of a variable from its current type to a specified target type, handling possible type conversions.
9. get_type_from_id: Retrieves the TOTAL_TYPE object associated with an identifier from the symbol table.
10. total_print: Generates C code for printing a variable's value to the console; handles both simple print and println (print with newline).
11. assign_value: Assigns a new value to an existing variable, updates the symbol table, and generates the corresponding assignment in C code.
12. create_int_array: Initializes an integer array with a defined size and optionally initializes elements if provided.
13. modify_int_array: Updates the values in an existing integer array variable with new elements, ensuring the array size remains consistent.
14. array_print: Generates code to define and initialize an integer array in C, or to simply declare it if no initial values are provided.
15. operate: Performs arithmetic operations (addition, subtraction, multiplication, division) on two expressions and handles type compatibility and promotion.
16. get_negative_value: Computes the negative of a given value, handling different data types and generating the corresponding unary operation in C code.

main.cpp

This source file contains the main function, which is the entry point of the program. It handles file input/output operations, setting up the parser, and printing the result of the parsing process. It reads from a source file and writes the translated C code to an output file.

MakeFile

This file automates the compilation process. It defines rules to build the executable by compiling and linking the Lex-generated scanner, the Bison-generated parser, and the main.cpp file. It also provides a clean command to remove all generated files to reset the project state.

Test File

1. test1.qv

```
fun main () {
        var i: int;
        i = 10;
        print(i);
}
```

I. output.c

```
#include <stdio.h>
int main(){
        int i;
        i = 10;
printf("%d", i);
        return(0);
}
```

2. test2.qv

```
fun main () {
        var i: int = 10;
        var j: real = 3.1415;
        print(i);
        print(j);
}
```

II. output.c

```
#include <stdio.h>

int main(){
        int i = 10;
        double j = 3.141500;
        printf("%d", i);
        printf("%lf", j);
        return(0);
}
```

3. test3.qv

```
fun main () {
        var i: int = 10;
        var j: real = 3.1415;
        var k: int = i + j;
        var l: real = i + j;
        print(k);
        print(l);
}
```

III. output.c

```
#include <stdio.h>

int main(){
        int i = 10;
        double j = 3.141500;
        int k = i + j;
        double l = i + j;
        printf("%d", k);
        printf("%lf", l);
        return(0);
}
```

4. test4.qv

```
fun main () {
	var radius: real = 5;
	var pi: real = 3.1415;
	var area: real = radius * radius * pi;
	print(area);
}
```

IV. output.c

```
#include <stdio.h>

int main(){
	double radius = 5;
	double pi = 3.141500;
	double area = radius * radius * pi;
	printf("%lf", area);
	return(0);
}
```

5. test5.qv

```
fun main () {
	var i: real = 1.5;
	var j: real = 3.14;
	var k: real = 2.8;
	print(i + j * k);
	print("\n");
	print(i * (j + k));
	print("\n");
}
```

V. output.c

```
#include <stdio.h>

int main(){
	double i = 1.500000;
	double j = 3.140000;
	double k = 2.800000;
	printf("%lf", i + j * k);
	printf("%s", "\n");
	printf("%lf", i * (j + k));
	printf("%s", "\n");
	return(0);
}
```

6. test6.qv

```
fun main () {
	print(123.456);
	println(123.456);
}
```

VI. output.c

```
#include <stdio.h>

int main(){
	printf("%lf", 123.456000);
	printf("%lf\n", 123.456000);
	return(0);
}
```

7. test7.qv

```
fun main () {
        var vi1: real[5] = {5, 3, 4, 1, 2}; // vi1[0]~vi1[4]
        var vi2: real[5] = {2, -2, 4}; // Missing dimensions assumed 0s
        print( vi1 * vi2 ); // Inner product, output: "20"
        print( "\n" );
        print( vi1 + vi2 ); // Dimension-wise addition, output: "{7, 1, 8, 1, 2}"
        print( "\n" );
}
```

VII. output.c

```
#include <stdio.h>

int main(){
        double vi1[5] = { 5, 3, 4, 1, 2 };
        double vi2[5] = { 2, -2, 4 };
        int temp1 = 0;
        for(int i = 0; i < 5; i++){
                temp1 += vi1[i] * vi2[i]; //value is 20
        }
        printf("%d", temp1);
        printf("%s", "\n");
        int temp2[5];
        for(int i = 0; i < 5; i++){
                temp2[i] = vi1[i] + vi2[i]; //value is { 7, 1, 8, 1, 2}
        }
        for(int i=0; i < 5; i++){
                printf("%d ",temp2[i]);
        }
        printf("%s", "\n");
        return(0);
}
```

8. test8.qv

```
fun main () {
        var vi1: real[5] = {5, 3, 4, 1, 2}; // vi1[0]~vi1[4]
        var vi2: real[3] = {2, -2, 4}; // Missing dimensions assumed 0s
        print( vi1 * vi2 ); // yyerror("ERROR: mismatched dimensions")
    print( "\n" );
}
```

VIII. error

```
error. line 4: Mismatched Dimensions at yytext:())
```

9. test9.qv

```
fun main () {
        var i: int;
        var i: real[3] = {2, -2}; // yyerror("ERROR: duplicate declaration")
}
```

IX. error

```
error. line 3: Duplicate Declaration at yytext:(;)
```

10. test10.qv

```
fun main () {
        var vi: real[2] = {2, -2, 5}; // yyerror("ERROR: too many dimensions")
}
```

X. error

```
error. line 2: Too Many Dimensions at yytext:(;)
```

11. test11.qv

```
fun main() { // For simplicity, main()
always returns 0
   var i: int[3] = {1, 2, 3};
   print("\n");
}
```

XI. output.c

```
#include <stdio.h>

int main(){
        int i[3] = { 1, 2, 3 };
        printf("%s", "\n");
        return(0);
}
```

12. test12.qv

```
fun main () {
        var vi1: real[5] = {5, 3, 4, 1, 2};
        var vi2: real[5] = {2, -2, 4};
        print( (vi1 * vi2) + (vi1 * vi2) );
        print( "\n" );
        print( vi1 + vi2 +vi2 );
        print( "\n" );
}
```

XII. output.c

```
#include <stdio.h>

int main(){
        double vi1[5] = { 5, 3, 4, 1, 2 };
        double vi2[5] = { 2, -2, 4 };
        int temp1 = 0;
        for(int i = 0; i < 5; i++){
                temp1 += vi1[i] * vi2[i];
//value is 20
        }
        int temp2 = 0;
        for(int i = 0; i < 5; i++){
                temp2 += vi1[i] * vi2[i];
//value is 20
        }
        printf("%d", (temp1) + (temp2));
        printf("%s", "\n");
        int temp3[5];
        for(int i = 0; i < 5; i++){
                temp3[i] = vi1[i] + vi2[i];
//value is { 7, 1, 8, 1, 2}
        }
        int temp4[5];
        for(int i = 0; i < 5; i++){
                temp4[i] = temp3[i] + vi2[i];
//value is { 9, -1, 12, 1, 2}
        }
        for(int i=0; i < 5; i++){
                printf("%d ",temp4[i]);
        }
        printf("%s", "\n");
        return(0);
}
```

Conclusion and Experience

During more than 40 hours of design, I finally experience how to make a compiler. From scanner to parser and generator, the homeworks keep me on the power of the field of compiler. I only use tools to make a language, but experts doesn’t have any tools to translate a language, and it’s more reasonable and flexible. I admire them a lot. In conclusion, I appreciate teacher PC’s teaching. I find a large fields, such as Columbus discovering the new world.

Complete code

1. main.h

```
#ifndef MAIN_H
#define MAIN_H

#define judge if(isError) YYABORT

#include <iostream>
#include <cstring>
#include <ctype.h>
#include <fstream>
#include <variant>

using namespace std;

typedef enum {
        INT_TYPE,
        REAL_TYPE,
        CHAR_TYPE,
        BOOL_TYPE,
        STRING_TYPE,
        INT_ARRAY_TYPE
} Type;


class TOTAL_TYPE{
public:
        Type type;
        string name;
        TOTAL_TYPE(Type t) : type(t) {}
};

class IntValue : public TOTAL_TYPE{
public:
        int value;
        IntValue(int v) : TOTAL_TYPE(Type::INT_TYPE), value(v) {}
};

class RealValue : public TOTAL_TYPE{
public:
        double value;
        RealValue(double v) : TOTAL_TYPE(Type::REAL_TYPE), value(v) {}
};

class CharValue : public TOTAL_TYPE{
public:
        char value;
        CharValue(char v) : TOTAL_TYPE(Type::CHAR_TYPE), value(v) {}
};

class IntArray : public TOTAL_TYPE{
public:
        const int SIZE;
        int* data;
        IntArray(int s) : TOTAL_TYPE(Type::INT_ARRAY_TYPE), SIZE(s)
```

```
    {
        data = NULL;
    }
    IntArray(int s, int* d) : TOTAL_TYPE(Type::INT_ARRAY_TYPE), SIZE(s)
    {
        changeData(d);
    }
    void changeData(int* d)
    {
        if(data == NULL)
            data = (int*)malloc(SIZE * sizeof(int));

        data = d;
    }
    ~IntArray() { free(data); }
};

class Symbol{
public:
    string name;
    TOTAL_TYPE* data;
    bool valid;
    Symbol(string n, TOTAL_TYPE* d, bool v) : name(n), data(d), valid(v) {}
    Symbol(Type t, string n) : name(n), data(NULL) {}
    ~Symbol() { delete(data); }
};

struct SymbolNode{
    Symbol *symbol;
    SymbolNode* next;
};

struct IntElement{
    int size;
    int* elements;
};

extern SymbolNode *symbolTable;

#endif
```

2. 11130038-scanner.l

```
%{
    #include "main.h"
    #include "11130038-parser.tab.h"

    extern int lineno;
    char stringBuffer[1024];
    void yyerror(const char *s);
%}

%x CHARSTART
%x CHARESCAPE
%x MULTIPLECOMMENT
%x SINGLECOMMENT
%x STRINGSTATE
%x STRINGESCAPE
```

```
%%
"main"                          { return MAIN; }
"var"                           { return VAR; }
"val"                           { return VAL; }
"bool"                          { return BOOL; }
"char"                          { return CHAR; }
"int"                           { return INT; }
"real"                          { return REAL; }
"true"                          { return TRUE; }
"false"                         { return FALSE; }
"class"                         { return CLASS; }
"if"                            { return IF; }
"else"                          { return ELSE; }
"for"                           { return FOR; }
"while"                         { return WHILE; }
"do"                            { return DO; }
"switch"                        { return SWITCH; }
"case"                          { return CASE; }
"fun"                           { return FUN; }
"ret"                           { return RET; }
"println"                       { return PRINTLN; }
"print"                         { return PRINT; }

[a-zA-Z_][a-zA-Z0-9_]*    { yylval.stringType = strdup(yytext); return ID; }
[0-9]+                          { yylval.intNum = atoi(yytext); return INTEGER; }
[0-9]+"."[0-9]+                 { yylval.realNum = atof(yytext); return DOUBLE; }

"=="                            { return EQJ; }
"!="                            { return NE; }
">"                             { return GT; }
">="                            { return GE; }
"<"                             { return LT; }
"<="                            { return LE; }

[\(\)\[\]\{\};:.,+\-*/=]                { return yytext[0];}



[\n\r]+                         { lineno++; return NEWLINE;}
[\t ]                           { ; }
"\\"                            {yyerror("invalid escape character");yyterminate();}

\'                              {BEGIN(CHARSTART);}
<CHARSTART>[\'\"\n]         {yyerror("invalid character");yyterminate();}
<CHARSTART><<EOF>>            {yyerror("missing terminating '
character");yyterminate();}
<CHARSTART>\\\'             {yyerror("missing terminating '
character");yyterminate();}
<CHARSTART>\\\'\'           {yylval.charType='\'';BEGIN(INITIAL);return
CHARACTER;}
<CHARSTART>\\                 {BEGIN(CHARESCAPE);}
<CHARSTART>.\'                {yylval.charType=*yytext;BEGIN(INITIAL);return
CHARACTER;}
<CHARESCAPE>(\\|\'|\"|\?)\' { yylval.charType=yytext[0]; BEGIN(INITIAL);return
CHARACTER;}
<CHARESCAPE>t\'               {yylval.charType=9;BEGIN(INITIAL);return
CHARACTER;}
<CHARESCAPE>n\'                {yylval.charType=10;BEGIN(INITIAL);return
```

```
CHARACTER;}
<CHARESCAPE><<EOF>>          {yyerror("invalid escape
character");yyterminate();}
<CHARESCAPE>.                {yyerror("invalid escape character");yyterminate();}

\"                           { BEGIN(STRINGSTATE); stringBuffer[0] = '\0'; }
<STRINGSTATE>\"                 { char* temp = strdup(stringBuffer);
                                  if (!temp) {
                                        yyerror("Memory allocation failed");
                                        yyterminate();
                                  }
                             yylval.stringType = temp;
                             BEGIN(INITIAL);
                             return STRING; }
<STRINGSTATE>\\                 { BEGIN(STRINGESCAPE); }
<STRINGSTATE>[^\\\n\"]+    { size_t len = strlen(stringBuffer);
                                  size_t max_append = sizeof(stringBuffer) - len - 1;
// -1 to leave space for null terminator
                                  strncat(stringBuffer, yytext, max_append); }
<STRINGSTATE>\n                 { yyerror("missing terminating \" character");
yyterminate(); }
<STRINGESCAPE>n                  { strcat(stringBuffer, "\\n");
BEGIN(STRINGSTATE); }
<STRINGESCAPE>t                 { strcat(stringBuffer, "\\t");
BEGIN(STRINGSTATE); }
<STRINGESCAPE>\"                { strcat(stringBuffer, "\"");
BEGIN(STRINGSTATE); }
<STRINGESCAPE>\\                { strcat(stringBuffer, "\\");
BEGIN(STRINGSTATE); }
<STRINGESCAPE>\'                { strcat(stringBuffer, "\'");
BEGIN(STRINGSTATE); }
<STRINGESCAPE>\?                { strcat(stringBuffer, "\?");
BEGIN(STRINGSTATE); }
<STRINGESCAPE>.                 { yyerror("invalid escape character");
yyterminate(); }
<STRINGESCAPE><<EOF>>        { yyerror("EOF in string constant");
yyterminate(); }


"//"                         { BEGIN SINGLECOMMENT; }
<SINGLECOMMENT>[^\n]*                { ; }
<SINGLECOMMENT>\n                    { lineno++;BEGIN 0; return NEWLINE; }

"/*"                         { BEGIN(MULTIPLECOMMENT); }
<MULTIPLECOMMENT>"*/"                  { BEGIN(INITIAL); }
<MULTIPLECOMMENT>.                     { ; }
<MULTIPLECOMMENT>\n                    { lineno++; }
<MULTIPLECOMMENT><<EOF>>                 { yyerror("Unclosed comment at end
of file."); yyterminate(); }

.                            {yyerror("scanner error");yyterminate();}
%%

int yywrap(void) {
       return 1;
}

void yyerror(const char *s) {
	printf("error. line %d: %s at yytext:(%s)\n", lineno, s, yytext);
}
```

3. 11130038-scanner.l

```
%{
#include "main.h"

SymbolNode *symbolTable=NULL;
string code;
bool isError=false;
int arrayOperation=0;
int lineno=1;
extern ofstream outFile;

void yyerror(const char *s);
extern int yylex();
extern int yyparse();
std::string get_type_name(Type type);
void assign_initial(string ID,Type type);
void assign_expr(string ID,Type type,TOTAL_TYPE* expr);
void create_symbol(Symbol* token);
std::variant<int, double, char> get_value(TOTAL_TYPE* value, Type targetType);
TOTAL_TYPE* get_type_from_id(const char* name);
void total_print(TOTAL_TYPE* value,bool isPrint);
void assign_value(const char* name, TOTAL_TYPE* value);
void create_int_array(const char* name, int def_size, IntElement items);
void modify_int_array(const char* name, IntElement items);
void array_print(string ID,Type type,TOTAL_TYPE* expr);
void array_print(string ID,Type type,TOTAL_TYPE* expr,IntElement elements);
TOTAL_TYPE* operate(TOTAL_TYPE* value1, TOTAL_TYPE* value2, char
symbol);
TOTAL_TYPE* get_negative_value(TOTAL_TYPE* value);
%}


%union {
	int intNum;
	char charType;
	char* stringType;
	double realNum;
	bool boolean;
	TOTAL_TYPE* allValue;
	IntElement intElement;
	Type types;
};

%type <allValue> stmt
%type <allValue> assign_stmt
%type <allValue> expr
%type <allValue> value
%type <intElement> value_list
%type <types> basic_type;

%token <intNum> INTEGER
%token <realNum>DOUBLE
%token <stringType> ID
%token <stringType> STRING
%token <charType> CHARACTER
```

```
%token <types>BOOL
%token <types>CHAR
%token <types>INT
%token <types>REAL

%token FUN
%token RET
%token MAIN
%token VAR VAL
%token TRUE FALSE
%token CLASS DOT
%token IF ELSE FOR WHILE DO
%token SWITCH CASE

%token PRINT PRINTLN

%token EQJ NE GT GE LT LE


%token NEWLINE

%start program

%left GT GE LT LE
%left EQJ NE
%left '+' '-'
%left '*' '/'
%right UMINUS

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%%

program:
	{ code+="#include <stdio.h>\n\n"; }
	function_definition program { printf("%s\n",code.c_str()); outFile<<code; }
	|
	;

function_definition:
	FUN ID '(' { code+="("; }    ')' { code+=")"; } '{' { code+="{"; } stmts '}'
{ code+="}"; }
	| FUN MAIN { code+="int main()"; } '('    ')' { code+="{"; } '{' stmts '}'
{ code+="\treturn(0);\n}"; }
	| NEWLINE { code+="\n"; }
	;

stmts:
	stmt stmts
	|
	;

stmt:
	'{' stmts '}'                          { judge;}
	| expr ';'                              { $$=$1;judge;}
	| assign_stmt                             { $$=$1;judge;}
	| print_stmt                           { judge; }
```

```
        | RET expr ';'                          { judge; }
        | NEWLINE { code+="\n"; }       { judge; }
        | if_stmt                               { judge; }
        | WHILE '(' expr ')' DO stmt{ judge; }
        | FOR '(' expr ';' expr ';' expr ')' stmt       { judge; }
        ;

print_stmt:
        PRINT '(' expr ')' ';'          { total_print($3,true);    }
        | PRINTLN '('    expr ')' ';' { total_print($3,false); }
        | PRINT '(' STRING ')' ';'    { std::string temp($3);code+="\tprintf(\"%s\",
\""+temp+"\");"; }
        | PRINTLN '(' STRING ')' ';'{ std::string temp($3);code+="\tprintf(\"%s\n\",
\""+temp+"\");"; }
        ;

if_stmt:
        IF '(' expr ')' stmt %prec LOWER_THAN_ELSE
        | IF '(' expr ')' stmt ELSE stmt
        ;

assign_stmt:
        VAR ID    ':' basic_type ';'       { assign_initial($2,$4); free($2); }
        | VAR ID ':' basic_type '=' expr ';'    { assign_expr($2,$4,$6); free($2); delete $6; }
        | VAR ID    ':' basic_type '[' expr ']' '=' '{' value_list '}' ';'
{ create_int_array($2, ((IntValue*)$6)->value, $10); array_print($2,$4,$6,$10);
free($2); }
        | VAR ID    ':' basic_type '[' expr ']' ';'
{ create_symbol(new Symbol(strdup($2), new IntArray(((IntValue*)$6)->value),
false));array_print($2,$4,$6); free($2);}
        | VAL ID    ':' basic_type ';'      { assign_initial($2,$4); free($2); }
        | VAL ID ':' basic_type '=' expr ';'    { assign_expr($2,$4,$6); free($2); delete $6; }
        | VAL ID    ':' basic_type '[' expr ']' '=' '{' value_list '}' ';'
{ create_int_array($2, ((IntValue*)$6)->value, $10); array_print($2,$4,$6,$10);
free($2); }
        | VAL ID    ':' basic_type '[' expr ']' ';'
{ create_symbol(new Symbol(strdup($2), new IntArray(((IntValue*)$6)->value),
false)); array_print($2,$4,$6); free($2); }
        | ID '=' expr ';'                           { assign_value($1, $3); delete $3; }
        | ID '=' '{' value_list '}' ';'        { modify_int_array($1, $4);    }
        ;
basic_type:
        INT                                     { $$ = INT_TYPE; }
        | REAL                                  { $$ = REAL_TYPE; }
        | BOOL                                  { $$ = BOOL_TYPE; }
        | CHAR                                  { $$ = CHAR_TYPE; }
        ;

value_list:
        expr                                { IntElement items; items.size = 1; items.elements
= (int*)malloc(sizeof(int)); items.elements[0] = ((IntValue*)$1)->value; $$=items; }
        | value_list ',' expr       { $$.size = $1.size + 1; $$.elements =
(int*)realloc($1.elements, $$.size * sizeof(int)); $$.elements[$$.size - 1] =
((IntValue*)$3)->value; }
        ;

expr:
        expr '+' expr                   { $$ = operate($1, $3, '+'); if($1-
>type!=INT_ARRAY_TYPE) $$->name= $1->name+" + "+$3->name; }
        | expr '-' expr                 { $$ = operate($1, $3, '-'); $$->name= $1->name+" -
"+$3->name;}
        | expr '*' expr                 { $$ = operate($1, $3, '*'); if($1-
>type!=INT_ARRAY_TYPE) $$->name= $1->name+" * "+$3->name; }
        | expr '/' expr                 { $$ = operate($1, $3, '/'); $$->name= $1->name+" /
"+$3->name;}
        | expr EQJ expr                 {}
        | expr NE expr                  {}
        | expr LT expr                  {}
        | expr LE expr                  {}
        | expr GT expr                  {}
        | expr GE expr                  {}
        | '-' expr %prec UMINUS { $$ = get_negative_value($2); }
        | '(' expr ')'                  { $$ = $2; $$->name= "("+$2->name+")"; }
        | ID '[' expr ']'               {}
        | ID '(' value_list ')' {}
        | value                         { $$=$1; }
        ;

value:
        INTEGER                 { $$ = new IntValue($1); $$->name=to_string($1); }
        | DOUBLE                { $$ = new RealValue($1); $$->name=to_string($1); }
        | CHARACTER             { $$ = new CharValue($1); $$->name=to_string($1); }
        | TRUE                  { $$ = new IntValue(1); $$->name=to_string(1); }
        | FALSE                 { $$ = new IntValue(0); $$->name=to_string(0); }
        | ID                    { $$ = get_type_from_id($1); $$->name=$1; }
        ;
%%

void assign_initial(std::string ID,Type type)
{
        switch(type)
        {
                case Type::INT_TYPE:
                        create_symbol(new Symbol(ID, new IntValue(0), false));
                        code=code+"\tint "+ID+";";
                        break;
                case Type::REAL_TYPE:
                        create_symbol(new Symbol(ID, new RealValue(0.0), false));
                        code=code+"\tdouble "+ID+";";
                        break;
                case Type::CHAR_TYPE:
                        create_symbol(new Symbol(ID, new CharValue(0), false));
                        code=code+"\tchar "+ID+";";
                        break;
                case Type::BOOL_TYPE:
                        create_symbol(new Symbol(ID, new IntValue(0), false));
                        code=code+"\tint "+ID+";";
                        break;
                default:
                        yyerror("invalid operation with negative value");
                        isError=true;
                        return;
        }
}

void assign_expr(std::string ID,Type type,TOTAL_TYPE* expr)
```

```
{
    switch(type)
    {
        case Type::INT_TYPE:
        {
            int intValue = std::get<int>(get_value(expr, INT_TYPE));
            create_symbol(new Symbol(ID, new IntValue(intValue), true));
            code += "\tint " + ID + " = " + expr->name + ";";
            break;
        }
        case Type::REAL_TYPE:
        {
            double value = std::get<double>(get_value(expr,REAL_TYPE));
            create_symbol(new Symbol(ID, new RealValue(value), true));
            code=code+"\tdouble "+ID+" = "+expr->name+";";
            break;
        }

        case Type::CHAR_TYPE:
        {
            char value = std::get<char>(get_value(expr,CHAR_TYPE));
            create_symbol(new Symbol(ID, new CharValue(value), true));
            code=code+"\tchar "+ID+" = "+expr->name+";";
            break;
        }
        case Type::BOOL_TYPE:
        {
            int value = std::get<int>(get_value(expr,INT_TYPE));
            create_symbol(new Symbol(ID, new IntValue(value), true));
            code=code+"\tint "+ID+" = "+expr->name+";";
            break;
        }
        default:
            yyerror("invalid operation with negative value");
            isError=true;
            return;
    }
}

string get_type_name(Type type){
    switch(type){
        case INT_TYPE:
            return "int";
        case REAL_TYPE:
            return "double";
        case CHAR_TYPE:
            return "char";
        case STRING_TYPE:
            return "string";
        case INT_ARRAY_TYPE:
            return "int";
        default:
            return "unknown";
    }
}

TOTAL_TYPE* get_negative_value(TOTAL_TYPE* v)
{
    TOTAL_TYPE* result;
    switch(v->type)
    {
        case Type::INT_TYPE:
            result = new IntValue(- ((IntValue*)v)->value);
            break;
        case Type::REAL_TYPE:
            result = new RealValue(- ((RealValue*)v)->value);
            break;
        case Type::CHAR_TYPE:
            result = new CharValue(- ((CharValue*)v)->value);
            break;
        default:
            yyerror("invalid operation with negative value");
            isError=true;
    }
    return result;
}
std::variant<int, double, char> get_value(TOTAL_TYPE* value, Type targetType)
{
    switch(targetType){
        case Type::INT_TYPE:
            switch(value->type)
            {
                case Type::INT_TYPE:
                    return ((IntValue*)value)->value;
                case Type::REAL_TYPE:
                    return (int)((RealValue*)value)->value;
                case Type::CHAR_TYPE:
                    return (int)(((IntValue*)value)->value);
                default:
                    yyerror("invalid operation with int value");
                    isError=true;
                    return -1;
            }
            break;
        case Type::REAL_TYPE:
            switch(value->type)
            {
                case Type::INT_TYPE:
                    return (double)(((IntValue*)value)->value);
                case Type::REAL_TYPE:
                    return ((RealValue*)value)->value;
                case Type::CHAR_TYPE:
                    return (double)(((IntValue*)value)->value);
                default:
                    yyerror("invalid operation with real value");
                    isError=true;
                    return -1.0;
            }
            break;
        case Type::CHAR_TYPE:
            switch(value->type)
            {
                case Type::INT_TYPE:
                    return (char)((IntValue*)value)->value;
                case Type::REAL_TYPE:
                    return (char)((RealValue*)value)->value;
                case Type::CHAR_TYPE:
                    return (char)(((IntValue*)value)->value);
                default:
                    yyerror("invalid operation with char value");
                    isError=true;
                    return ' ';
            }
            break;
        default:
            yyerror("invalid operation with char value");
            isError=true;
            return ' ';
    }
}

void total_print(TOTAL_TYPE* value,bool isPrint)
{
    if(isPrint){
        switch(value->type)
        {
            case Type::INT_TYPE:
                code+="\tprintf(\"%d\", "+value->name+");";
                break;
            case Type::REAL_TYPE:
                code+="\tprintf(\"%lf\", "+value->name+");";
                break;
            case Type::CHAR_TYPE:
                code+="\tprintf(\"%c\", "+value->name+");";
                break;
            case Type::INT_ARRAY_TYPE:
                code+="\tfor(int i=0; i < "+to_string(((IntArray*)value)-
>SIZE)+"; i++){\n";
                code+="\t\tprintf(\"%d \","+value->name+"[i]);\n\t}";
                break;
            default:
                yyerror("printing invalid value");
                isError=true;
        }
    }
    else
    {
        switch(value->type)
        {
            case Type::INT_TYPE:
                code+="\tprintf(\"%d\\n\", "+value->name+");";
                break;
            case Type::REAL_TYPE:
                code+="\tprintf(\"%lf\\n\", "+value->name+");";
                break;
            case Type::CHAR_TYPE:
                code+="\tprintf(\"%c\\n\", "+value->name+");";
                break;
            case Type::INT_ARRAY_TYPE:
                code+="\tfor(int i=0; i < "+to_string(((IntArray*)value)-
>SIZE)+"; i++){\n";
                code+="\t\tprintf(\"%d\\n\","+value->name+"[i]);\n\t}";
                break;
            default:
                yyerror("Cannot print the value");
                isError=true;
                return;
        }
    }
}

Symbol* findSymbol(const char* name)
{
    SymbolNode* nowSymbolNode = symbolTable;
    while(nowSymbolNode != NULL && strcmp(nowSymbolNode->symbol->name.c_str(), name) != 0)
    {
        nowSymbolNode = nowSymbolNode->next;
    }
    if(nowSymbolNode == NULL)
    {
        yyerror("Cannot find symbol");
        isError=true;
    }
    else if(strcmp(nowSymbolNode->symbol->name.c_str(), name) == 0)
    {
        return nowSymbolNode->symbol;
    }
    yyerror("unknown error");
    isError=true;
    return NULL;
}

TOTAL_TYPE* get_type_from_id(const char* name)
{
    Symbol* symbol = findSymbol(name);
    return symbol->data;
}

void assign_value(const char* name, TOTAL_TYPE* value)
{
    Symbol* symbol = findSymbol(name);

    switch(symbol->data->type)
    {
    case Type::INT_TYPE:
    {
        int int_value = std::get<int>(get_value(value,INT_TYPE));
        ((IntValue*)symbol->data)->value = int_value;
        code=code+"\t"+name+" = "+to_string(int_value)+";";
        break;
    }
    case Type::REAL_TYPE:
    {
        double real_value = std::get<double>(get_value(value,REAL_TYPE));
        ((RealValue*)symbol->data)->value = real_value;
        code=code+"\t"+name+" = "+to_string(real_value)+";";
        break;
    }
    case Type::CHAR_TYPE:
    {
        char char_value = std::get<char>(get_value(value,CHAR_TYPE));
```

```
			((CharValue*)symbol->data)->value = char_value;
			code=code+"\t"+name+" = "+to_string(char_value)+";";
			break;
		}
		default:
			yyerror("Cannot assign number value");
			isError=true;
		}
}

void modify_int_array(const char* name, IntElement items)
{
		Symbol* symbol = findSymbol(name);
		if(symbol->data->type != Type::INT_ARRAY_TYPE)
		{
			yyerror("Cannot assign different type value to a vector");
			isError=true;
		}
		int def_size = ((IntArray*)(symbol->data))->SIZE;

		if(items.size > def_size)
		{
			yyerror("Too Many Dimensions");
			isError=true;
		}
		else if(items.size < def_size)
		{
			items.elements = (int*)realloc(items.elements, def_size * sizeof(int));
			for(int i = items.size; i < def_size; i++)
			{
				items.elements[i] = 0;
			}
		}
		((IntArray*)(symbol->data))->changeData(items.elements);
}

void create_int_array(const char* name, int def_size, IntElement items)
{
		if(items.size > def_size)
		{
			yyerror("Too Many Dimensions");
			isError=true;
		}
		else if(items.size < def_size)
		{
			items.elements = (int*)realloc(items.elements, def_size * sizeof(int));
			for(int i = items.size; i < def_size; i++)
			{
				items.elements[i] = 0;
			}
		}
		create_symbol(new Symbol(strdup(name), new IntArray(def_size,
items.elements), true));
}

void array_print(string IDs,Type type,TOTAL_TYPE* expr,IntElement elements)
{
		code=code + "\t" +get_type_name(type)+" "+ (string) IDs+ "[" + expr->name    +
"] = { "+ to_string(elements.elements[0]);
        for(int i=1;i<elements.size;i++)
        {
                code = code + ", " + to_string(elements.elements[i]);
        }
        code+=" };";
}
void array_print(string IDs,Type type,TOTAL_TYPE* expr)
{
        code=code +get_type_name(type)+" "+ (string) IDs+ "[" + expr->name    + "];";
}

void create_symbol(Symbol* symbol)
{
        SymbolNode* nowSymbolNode = symbolTable;
        SymbolNode* prevSymbolNode = NULL;
        while(nowSymbolNode != NULL && strcmp(nowSymbolNode->symbol-
>name.c_str(), symbol->name.c_str()) != 0)
        {
                prevSymbolNode = nowSymbolNode;
                nowSymbolNode = nowSymbolNode->next;
        }
        if(nowSymbolNode == NULL)
        {
                SymbolNode* newSymbolNode = new SymbolNode;
                newSymbolNode->symbol = symbol;
                newSymbolNode->next = NULL;
                if(prevSymbolNode != NULL)
                {
                        prevSymbolNode->next = newSymbolNode;
                }
                else
                {
                        symbolTable = newSymbolNode;
                }
        }
        else if(strcmp(nowSymbolNode->symbol->name.c_str(), symbol->name.c_str())
== 0)
        {
                yyerror("Duplicate Declaration");
                isError=true;
        }
}



TOTAL_TYPE* operate(TOTAL_TYPE* value1, TOTAL_TYPE* value2, char
symbol)
{
        TOTAL_TYPE* result = nullptr;

        if(value1->type == Type::INT_ARRAY_TYPE || value2->type ==
Type::INT_ARRAY_TYPE)
        {
                int value1_size = ((IntArray*)value1)->SIZE;
                int value2_size = ((IntArray*)value2)->SIZE;
                int dimension = value1_size ;
        if(value1_size!=value2_size)
        {
            yyerror("Mismatched Dimensions");
            isError=true;
        }

        int* data1 = ((IntArray*)value1)->data;
        int* data2 = ((IntArray*)value2)->data;
        string finalValue="{ ";
        if(symbol == '+')
        {
            arrayOperation+=1;
            result = new IntArray(dimension, (int*)malloc(dimension *
sizeof(int)));
            result->name="temp"+to_string(arrayOperation);
            for(int count = 0; count < dimension; count++)
            {
                ((IntArray*)result)->data[count] = data1[count] + data2[count];
                if(count!=0) finalValue+=", "+ to_string(((IntArray*)result)-
>data[count]);
                else finalValue+= to_string(((IntArray*)result)->data[count]);
            }
            finalValue+="}";

            code+="\tint "+result->name+"["+to_string(dimension)+"];\n";
            code+="\tfor(int i = 0; i < "+to_string(dimension)+"; i++){\n";
            code+="\t\t"+result->name+"[i] = ";
            code+=value1->name + "[i] + " + value2->name + "[i]; //value is
"+finalValue+"\n\t}\n";
        }
        else if(symbol == '*')
        {
            arrayOperation+=1;
            result = new IntValue(0);
            result->name="temp"+to_string(arrayOperation);
            for(int i = 0; i < dimension; i++)
            {
                ((IntValue*)result)->value += data1[i] * data2[i];
            }

            code+="\tint "+result->name+" = 0;\n";
            code+="\tfor(int i = 0; i < "+to_string(dimension)+"; i++){\n";
            code+="\t\t"+result->name+" += ";
            code+=value1->name + "[i] * " + value2->name + "[i]; //value is
"+to_string(((IntValue*)result)->value)+"\n\t}\n";
        }
    }
    else if(value1->type == Type::REAL_TYPE || value2->type ==
Type::REAL_TYPE)
    {
        result = new RealValue(0.0);
        double value1_value = std::get<double>(get_value(value1,REAL_TYPE));
        double value2_value = std::get<double>(get_value(value2,REAL_TYPE));
        switch(symbol)
        {
            case '+':
                ((RealValue*)result)->value = value1_value + value2_value;
                break;
            case '-':
                ((RealValue*)result)->value = value1_value - value2_value;
                break;
            case '*':
                ((RealValue*)result)->value = value1_value * value2_value;
                break;
            case '/':
                ((RealValue*)result)->value = value1_value / value2_value;
                break;
        }
    }
    else if(value1->type == Type::INT_TYPE || value2->type == Type::INT_TYPE)
    {
        result = new IntValue(0);
        int value1_value = std::get<int>(get_value(value1,INT_TYPE));
        int value2_value = std::get<int>(get_value(value2,INT_TYPE));
        switch(symbol)
        {
            case '+':
                ((IntValue*)result)->value = value1_value + value2_value;
                break;
            case '-':
                ((IntValue*)result)->value = value1_value - value2_value;
                break;
            case '*':
                ((IntValue*)result)->value = value1_value * value2_value;
                break;
            case '/':
                ((IntValue*)result)->value = value1_value / value2_value;
                break;
        }
    }
    else if(value1->type == Type::CHAR_TYPE && value2->type ==
Type::CHAR_TYPE)
    {
        result = new CharValue(0);
        char value1_value = ((CharValue*)value1)->value;
        char value2_value = ((CharValue*)value2)->value;
        switch(symbol)
        {
            case '+':
                ((CharValue*)result)->value = value1_value + value2_value;
                break;
            case '-':
                ((CharValue*)result)->value = value1_value - value2_value;
                break;
            case '*':
                ((CharValue*)result)->value = value1_value * value2_value;
                break;
            case '/':
                ((CharValue*)result)->value = value1_value / value2_value;
                break;
        }
    }
    else
    {
        yyerror("Unknown operation");
        isError=true;
    }
    return result;
}
```

4. main.cpp

```
#include "main.h"
#include "11130038-parser.tab.h"

extern int yyparse(void);
extern FILE* yyin;
ofstream outFile;
extern string code;
int main()
{
        int mode;

        string sFile;
        printf("Please input the path of the file: ");
        cin >> sFile;
        FILE* fp = fopen(sFile.c_str(), "r");
        if (fp == NULL) {
                printf("Cannot open %s\n", sFile.c_str());
        }
        else {
                yyin = fp;
        }

        outFile.open("output.c");

        if (yyparse() == 0) {
                printf("Parsing successful!\n");
        }
        else {
                printf("Parsing failed!\n");
        }

        return 0;

}
```

5. Makefile

```
LEX=flex
YACC=bison
CC=g++
OBJECT=main
YACCNAME=11130038-parser
LEXNAME=11130038-scanner

$(OBJECT): lex.yy.o ${YACCNAME}.tab.o main.o
                $(CC) main.o lex.yy.o ${YACCNAME}.tab.o -o $(OBJECT)
lex.yy.o: lex.yy.c ${YACCNAME}.tab.h main.h
                $(CC) -c lex.yy.c
${YACCNAME}.tab.o: ${YACCNAME}.tab.c main.h
                $(CC) -c ${YACCNAME}.tab.c
${YACCNAME}.tab.c ${YACCNAME}.tab.h: ${YACCNAME}.y
                $(YACC) -d ${YACCNAME}.y
lex.yy.c: ${LEXNAME}.l
                $(LEX) ${LEXNAME}.l
main.o: main.cpp
```

```
		$(CC) -c main.cpp
clean:
		@del -f $(OBJECT) *.o lex.yy.c ${YACCNAME}.tab.h
${YACCNAME}.tab.c main.exe
```