Compiler Project Report

陈旭旸 5140309412

1. Introduction

In this project we are required to write a smallC compiler for scratch. I have to say that this project is of great difficulty. But I finally accomplished it, with **all semantic analysis implemented** and **6 optimization** added to my smallC compiler. We start from lexical analyzer, then syntax analyzer, intermediate-code generation, optimization and MIPS code generation. The project greatly improved my ability of coding and debugging, as we as my knowledge about compilers. I will clearly demonstrate my compiler in several stages.

2. Lexical Analyzer

1) Introduction

In this part we are required to write a lexical analyzer using flex to transform a source program into tokens. This part, from my perspective, is probably the simplest part during the whole project.

2) Definition of Tokens

We use the regular expressions given by the manual to derive tokens. Pay attention that the precedence of token is defined by their order in the file. So the regular expression of identifier should be placed at the end of the file. And for binary, unary and assign operators, I give unique token name to each operator, for example, "==" has the token name EQ, "&" has the token name BAND so that the following process can be simplified. My definition of tokens is in the following form:

Token	Meaning	Token	Meaning
Reserved word	As defined in manual	|	BOR
+	ADD	^	BXOR
-	SUB	&&	LAND
*	MUL	||	LOR
/	DIV	!	NOT
%	MOD	+=	ADDASSIGN
<<	SL	-=	SUBASSIGN
>>	SR	*=	MULASSIGN
<	LT	/=	DIVASSIGN
>	GT	&=	ANDASSIGN
<=	BGT	|=	ORASSIGN
>=	NLT	^=	XORASSIGN
==	EQ	++	PIN
!=	NEQ	--	PDE
&	BAND	~	BNOT

3) Solution for Hex and Dec Numbers

Actually in C, there is a function called strtol that can transform a hex, dec and oct string into oct integer number. So just need to define the token of number so that it can recognize hex and dec number and then we will get oct number. The regular expression for integer is as follow:

```
INTEGER (({DIGIT}+)|(0x|0X|0)[0-9a-fA-F]+)
```

3. Syntax Analyzer

1) Introduction

Syntax analyzer alone is quite simple. We just need to define the precedence of the tokens and their associativity and type the semantic rule into the bison file with some minor changes to the grammar

2) Definition of Precedence and Associativity

We can use the token we defined in the lexical analyzer and add %left or %right before them to indicate the associativity. And the larger their line number is, the higher their precedence is. The definition is in the format as follows:

```
%right ASSIGNOP ADDASSIGN SUBASS
%left LOR
%left LAND
%left BOR
%left BXOR
%left BAND
%left EQ NEQ
%left GT LT NGT NLT
%left SL SR
%left ADD SUB
%left MUL DIV MOD
%right LNOT PIN PDE UMIN BNOT
%left DOT LP RP LB RB
```

3) Providing Semantic Rules

To provide the semantic rules, we just need to type the rules given in the manual into bison file, as shown in follows:

```
extdefs : /*empty*/
    {

    }
    | extdef extdefs
    {

    }
;

extdef :
    TYPE extvars SEMI
    {

    }
    | stspec sextvars SEMI
    {

    }

    | TYPE func stmtblock
```

However, I did change some of the rules for the convenience for future work.

1. I convert the rules for definition of variables into left-recursive form. For example, the definition of structure variables in function in the manual is like:

   $$\begin{array}{lll} \text{SDECS} & \rightarrow & \text{ID COMMA SDECS} \\ & | & \text{ID} \end{array}$$

   But in my file, it is like:

   ```
   sdecs : sdecs COMMA ID
   ```

   This is used to solve the problem of L-attribute. I will talk about it in detail in the following section.
2. Since we only use up to 2 dimensional array in the project, there is no need to define the array in

recursive manner. We simply define it with 2 cases:
arrs->LB exp RB and arrs->LB exp RB LB exp RB.

3. If we add the empty non-terminals for back-patching, bison will discover the reduce/reduce conflict between stmt->IF LP exp RP m stmt and stmt->IF LP exp RP m stmt n ELSE m stmt when it encounters ELSE token. But if we arrange their definition order in bison correctly, bison will automatically choose the second rule by default. So there is no need to worry about the reduce/reduce conflict if we set the order right.
4. Since the symbol "-" will have higher precedence with it is recognized as SUB during the rule exps-> SUB exp, we need to raise its precedence. In bison we can do this by first define a terminal that has higher precedence that MIN, which is called UMIN and add %UMIN after the rule, as shown below:

```
| SUB exp %prec UMIN
```

5. For binary, unary and assign operation in manual, it is just simply BOP, UOP and ASSIGN. But since we have split these operations into several types, we need to give rule for each binary, unary and assign operation rules. For example, we need to define rule exps->exps ADD exps, exps->exps SUB exps, etc.

4. Intermediate-code Generation

1) Introduction

Here comes one of the most difficult parts of the project! In this section we are required to generate the intermediate code according to the parse tree and perform the semantic analysis at the same time. Basically I choose the three-address-code as the intermediate code and do the generation and sementic checking during the parsing in bison.

2) Choice of IR

I find that most of the compilers on the internet and Github use parse tree as the IR. It is quite simple to copy them on the internet and change some of its variables to claim that it is my project. But instead I choose to challenge myself by using the three-address-code from the text book and write the compiler from scratch. All the codes in this project is written by myself. It is really a tough time writing the compiler, but fortunately I did it at last. The generation of three-address-code is suitable for being embedded in bison file, which slightly lowers the difficult for code generation but it also makes optimization more difficult.

3) Symbol Table

To generate the intermediate-code we first need to define a symbol table which holds all the information for the variables, types and functions defined in the program. The definition of symbol table is in "type.h". A symbol table is defined as a linked list of symbol records, which is an entry of a symbol table. The definition of symbol record is shown below:

```
struct symrec //symbol record type
{
  char *name;  //name of the symbol
  int offset;  //memory offset of the symbol
  struct symrec *next; //its a linked list
  Type type;  //type of the symbol
  struct symrec *strField;  //the name space for this symbol, also a symbol table (only for function and structure)
  int tempCnt;  //the id of each temporal variable
  int ConstVal;  //the constant value if it is a constant or it holds constant
  int row;  //the maximum row number (only for array)
  int col;  //the maximum column number (only for array)
  int width;  //the width of this symbol (4 for int)
  int paranum;  //how many parameter the symbol has (only for function)
  int *arrayInit;  //initial value for array, stored in an array
  int isGlobal;  //indocate whether this is a global variable
};
```

The symbol record holds all the information of a variable. There are 3 symbol table pointers in my project. global_sym_table holds the symbol table of global variables. sym_table is the one that holds the current symbol table. const_table holds all the constant numbers. The reason why we need a symbol table that holds the current symbol table is because a function or a structure will have their own symbol table that holds all their local variables. We need to switch to their symbol table during the IR generation.

We use getsym function to get a symbol record from the current symbol according to the name of the symbol, putsym function to put a symbol record into the current symbol table and getsym_by_offset to get a symbol record by offset. We use a function called install to add a symbol record to the current symbol table after ensuring that the symbol is not defined and is valid.

For a function, its symbol record holds a symrec strField that points to the symbol table that has all its local variables. For a structure, its symbol record holds a symrec strField that points to the symbol table that has all its attributes.

4) Quadruple

Quadruple is the three-address-code we use. The intermediate-code is defined as a linked list of quadruple. The definition of quadruple is shown below:

```
struct qnode  //quadruple
{
  char *Op;  //the operation of this quadruple
  symrec *arg1;  //the symbol record for the first argument
  symrec *arg2;  //the symbol record for the second argument
  symrec *dest;  //the symbol record for the destination argument
  struct qnode *target;  //the symbol record for the target quadruple (only for jump or branch)
  struct qnode *next;  //next quadruple, it is a linked list
  int id;  //the unique id of a quadruple
  int isBlock;  //indicate whether this quadruple is the head of a basic block
  int generateLabel;  //indicate whether we need to generate a MIPS code for label
  int need; //indicate whether this quadruple is needed for assign operation
};
```

Basically the structure of a quadruple is like this:

Op arg1, arg2, dest.

For array assign, the Op is “[]=”, and the quadruple means arg1[arg2] = dest.

For array read, the Op is “=[]”, and the quadruple means dest=arg1[arg2].

For jump or branch we may need target field called target to indicate the place of target IR.

For function call, we use arg1 to record the symbol record for function and dest to record the number of arguments.

For param, we use arg1 as the symbol record for the argument needed for the function.

For relation operation, take “>” for example, it means if arg1>arg2 then goto target.

Also we have defined quad_push function to add a quadruple in IR. Here we always generate a new

quadruple after each quad_push so that we can know the location of next instruction in advance. The location of next instruction is need for back-patching.
My quadruple has the following operators: +,-,*,/,%,<<,>>,~,&,|,^,[]=, =[], >, <, >=, <=, ==, !=jump, label, param, call.

5) Attributes for Terminals and Non-terminals

To generate the IR we need to convey and store values among terminals and non-terminals.
For non-terminals that are related to variables, we define STypeVal type:

```
struct //attributes for non-terminal that holds the information of variables
{
    Type type;
    int width;
    char *lexeme;  //the name of that variable
    symrec *structST;   //the unique symbol table for that variable (for function and structure)
} STypeVal;
```

Terminals and Non-terminals having that type are: TYPE, extvars, var, stspec, sdecs, sdefs, func
For non-terminals that are related to expressions, we define ExprTypetype:

```
struct //attribute for expression-related non-terminal
{
    symrec *address;  //symbol record that holds the value of this expression
    Type type;  //type of the result of this expression

    symrec *array;  //if the expression is to access an array,
                    //we need to store the symbol record for the array symbol

    symrec *offset; //if the expression is to access an array,
                    //we need to store the symbol record for the accessing offset
    list * truelist; //truelist for backpatching
    list * falselist; //falselist for backpatching
    list * nextlist; //nextlist for backpatching
    int isLeft; //whether this expression can be left value
    int isInt; //whether the expression value is integer
} ExprType;
```

Terminals and Non-terminals having that type are: exps, exp
For non-terminals that are related to arguments, we define ArgExpr:

```
struct //attribute for arguments-related non-terminal
{
    symrec* array[100]; //holds all the arguments
    int len;  //holds the length of argument list
    symrec *address;
} ArgExpr;
```

Terminals and Non-terminals having that type are: args, init
For non-terminals that are related to statements, we define ListExpr:

```
struct //attributes for statement
{
    list * nextlist; //nextlist for backpatching
    list * skiplist; //skiplist for backpatching
    list * breaklist; //breaklist for backpatching
} ListExpr;
```

Terminals and Non-terminals having that type are: n stmt stmts stmtblock
Also, for token INTEGER, it has intval attribute that holds the integer value, for ID, READ, WRITE, they have id attribute holding their lexeme. For non-terminal paras, it has an attribute intval that holds the number of arguments. For non-terminal m, it has an attribute instr that holds the current instruction.

6) IR Generation Embedded in Bison

Now comes the most complicated part of IR generation. We need to figure out all the action needed for generating IR for each semantic rule. To make my implementation clear and easy to understand, I will illustrate the operation for each semantic rule.

1. program -> extdefs

   We need to do all the initializations before everything starts, so we add the initialization before shifting extdefs:

```
program :
    {
        offset = 0;
        inloop = 0;
        ST_Stack_Init(&StStack); INT_Stack_Init(&OffsetStack);
        codeHead = malloc(sizeof(Quad)); codeHead->next = NULL;
        install("read", 0, Function, NULL, 0, 0, 0);
        install("write", 0, Function, NULL, 0, 0, 0);
    } extdefs
```

   Here we put read and write function into the symbol table since they are predefined function

2. extdefs ->//empty
   extdefs -> extdef extdefs

   Nothing need to be done, great!

3. extdef -> TYPE extvars SEMI
   extdef -> stspec sextvars SEMI

   Still nothing to be done.

4. extdef -> TYPE func stmtblock

   This rule is reduced when we have finished parsing a function. When we are parsing a function, we need to store the symbol table and set the current symbol table to a new one which stores the local variable of that function. The previous symbol table is stored in a stack. The previous offset is also stored in a stack. Since now we have finished parsing that function, we need to retrieve the symbol table and offset from the stack.

5. sextvars -> //empty

   Nothing to be done.

6. sextvars -> ID

   This rule is used to define a structure variable having name ID. Since we have changed all rules that are reduced to sextvars to left-recursive version, after investigating the rules for defining structure, we can make sure that during the reduction of this rule, the element at the top of the parsing stack always holds the information of the structure, including its strField, its width, etc. The element at the stack can be referred in bison by $0. So we can store all the information of this structure type into the reduced non-terminal sextvars. And we also have to install all the attribute of this ID variable into the symbol table. The procedure is a little bit

tedious, we need to frequently access the current symbol table and structure' s symbol table to install the symbol record. The variable we add has the name "ID.attribute name". And offset should also be modified accordingly.
The whole process is shown below:

```
{
    if (getsym($1) != 0)    //ID not defined
    {
        errors++;
        printf("Semantic Error!\n");
        printf("Variable %s Redeclaration!\n", $1);
    }
    //sextvars should also holds information for this structure
    $<STypeVal.type>$ = $<STypeVal.type>0;
    $<STypeVal.width>$ = $<STypeVal.width>0;
    $<STypeVal.lexeme>$ = $1;
    $<STypeVal.structST>$ = $<STypeVal.structST>0;
    //install ID
    sym_table = install($<STypeVal.lexeme>$, offset, Struct, $<STypeVal.structST>$, 0, 0, $<STypeVal.width>$);

    //install all of ID' s attributes
    symrec * ori = sym_table;
    symrec * str = $<STypeVal.structST>$;
    char * s1;
    char *s2, *s;
    for (int i = 0; i < $<STypeVal.width>$; i = i + 4)
    {
        sym_table = str;
        symrec *pt = getsym_by_offset(i);
        s2 = pt->name;
        s = malloc(sizeof(char) * (strlen($1) + strlen(s2) + 3));
        strcpy(s, $1);
        strcat(s, ".");
        strcat(s, s2);
        sym_table = ori;
        ori = install(s, offset + i, Integer, NULL, 0, 0, 4);
    }
    sym_table = ori;
    //move the offset ahead
    offset += $<STypeVal.width>$;
}
```

sextvars -> sextvars COMMA ID is almost the same, except that structure information can be referred from $1.

7. extvars -> var

   This rule is used to define a global variable. All the information of that variable is stored in the attributes of non-terminal var. If var has type Integer, we set its ConstVal field to zero. If var has type Array, we will set the initial value to zeros.

8. extvars -> var ASSIGNOP init

   This rule is used to define and assign an initial value to a global variable. For global value, since it is defined in .data segment in MIPS, we can not use a three-address-code to set its value, so we have to **record its value in symbol table**. Here we use the ConstVal field to hold integer variable' s initial value and use arrayInit to hold the array' s initial value.
   Detailed implementation is shown below:

```
| var ASSIGNOP init
{
    if (($<STypeVal.type>1 == Integer) && ($<ArgExpr.len>3 == 0))
    {
        //ConstVal is used to hold initial value
        getsym($<STypeVal.lexeme>1)->ConstVal = $<ExprType.address>3->ConstVal;
    }
    else if (($<STypeVal.type>1 == Array) && ($<ArgExpr.len>3 > 0)) //set initial value
    {
        symrec *ptr = getsym($<STypeVal.lexeme>1), *pt, *zero;
        zero = putconst(0);
        int length = ptr->row * ptr->col;
        int base = ptr->offset;
        ptr->arrayInit = (int *) malloc ((length+1) * sizeof(int));
        for (int i = 0; i < length; i++)
        {
            //pt = putconst(4*i);
            if (i < $<ArgExpr.len>3)
            {
                //init has attributes holding initial value
                ptr->arrayInit[i] = $<ArgExpr.array>3[i]->ConstVal;
                //quad_push("[]=", ptr, pt, $<ArgExpr.array>3[i], NULL);
            }
            else ptr->arrayInit[i] = 0;
            //quad_push("[]=", ptr, pt, zero, NULL);
        }
    }
}
```

extvars -> extvars COMMA var and extvars -> extvars COMMA var ASSIGNOP init are almost the same as previous two.

9. var -> ID

   This rule is used to define an integer variable called ID. So we just set all the variable information to var' s attribute and install ID into the current symbol table.

10. var -> ID LB INT RB

    This rule is used to define a one-dimensional array variable called ID. So we just set all the variable information to var' s attribute and install ID into the current symbol table. Pay attention that the width of this array need to be INT multiplied 4 because an integer takes up 4 bytes. The detailed implementation is shown below:

```
| ID LB INT RB
{
    //set information
    $<STypeVal.type>$ = Array;
    $<STypeVal.width>$ = $3 * 4;
    $<STypeVal.lexeme>$ = $1;
    //install ID
    sym_table = install($1, offset, Array, NULL, 1, $3, $<STypeVal.width>$);
    offset += $<STypeVal.width>$;
}
```

11. var -> ID LB INT RB LB INT RB

    Almost the same as 10. This shows the convenience to split non-terminal arr into two rules.

12. stspec -> STRUCT ID LC sdefs RC

This rule is used to define a structure type. During the reduction of sdefs, all of the attributes of the structure will be installed in the unique symbol table. So we need to store the current symbol table and create a new one before sdefs and restore it and set the unique symbol table to stspec' s attribute. Also, ID should be installed to the restored symbol table and all the information of this structure type should be sotred to stspec' s attribute.
The rule should be changed to:
stspec -> STRUCT ID {store sym_table, store offset}LC sdefs RC{restore sym_table, restore offset, install ID}
The detailed implementation is shown below:

```
stspec :
    STRUCT ID LC
    {
        ST_Stack_Push(&StStack, sym_table);//store symbol table
        sym_table = (symrec *)0;
        INT_Stack_Push(&OffsetStack, offset);//store offset
        offset = 0;
    }
    sdefs RC
    {
        //set attributes of stspec
        $<STypeVal.structST>$ = sym_table;
        $<STypeVal.type>$ = Struct;
        $<STypeVal.width>$ = offset;
        //restore environment
        sym_table = ST_Stack_Pop(&StStack);
        offset = INT_Stack_Pop(&OffsetStack);
        sym_table = install($2, 0, StructID, $<STypeVal.structST>$, 0, 0, $<STypeVal.width>$);
    }
```

stspec -> STRUCT LC sdefs RC is almost the same, except that we don't need to install the name of the structure type.

13. sdefs -> //empty
    sdefs -> TYPE sdecs SEMI sdefs

    These rules is used to define the attribute of structure type. So we only need to record the offset. Notice that the install of variable has been done during the reduction to sdecs.

14. sdecs -> sdecs COMMA ID

    This rule is used to define attribute called ID to the unique symbol table of a structure. What we need to do is almost the same as in rule sextvars -> sextvars COMMA ID. So we can simply copy the code from rule sextvars -> sextvars COMMA ID.
    sdecs -> ID can also use the code for sextvars -> ID

15. decs -> var ASSIGNOP init

    This rule is used to assign initial value to a variable. So we first need to get the symbol record according to the lexeme attribute of var. Then if var is integer type, we generate a "=" quadruple with the source operand the address attribute of init and with the destination operand the symbol record of var. If var is array type, we will use array attribute of init as the array that holds the initial value and len attribute as the length of the array so that we can use a loop to generate quadruple for var.symbol_record[i*4]=init.array[i]. The detail for attributes for init will be covered later. The detailed implementation is as shown below:

```
| var ASSIGNOP init
{
    symrec *pt = getsym($<STypeVal.lexeme>1);
    if (pt == 0)  //if var not defined, error
    {
        errors++;
        fprintf(stderr, "Semantic Error!\n");
        fprintf(stderr, "Variable %s not declared!\n", $<STypeVal.lexeme>1);
    }
    if (($<STypeVal.type>1 == Integer) && ($<ArgExpr.len>3 == 0)) //integer, derect assign
    {

        quad_push("=", $<ExprType.address>3, NULL, getsym($<STypeVal.lexeme>1), NULL);
    }
    else if (($<STypeVal.type>1 == Array) && ($<ArgExpr.len>3 > 0))  //array, use a loop to assign
    {
        symrec *ptr = getsym($<STypeVal.lexeme>1), *pt, *zero;//get the symbol record for array
        zero = putconst(0);
        int length = ptr->row * ptr->col;
        int base = ptr->offset;
        for (int i = 0; i < length; i++)
        {
            pt = putconst(4*i); //1 integer is 4 bytes
            if (i < $<ArgExpr.len>3)
            {
                quad_push("[]=", ptr, pt, $<ArgExpr.array>3[i], NULL);
            }
            else quad_push("[]=", ptr, pt, zero, NULL);
        }
    }
}
```

decs -> decs COMMA var ASSIGNOP init is almost the same as previous one.

16. exp -> exps

    We simple assign all the attributes of exps to exp. Attributes for exps will be covered later.

17. init -> exp

    This rule is to set the value of an expression as initial value of an integer variable. For exp, we have an attribute address that records the symbol record holding its value. So we need to the address attribute of init with that symbol record. And set the len attribute to zero so that other rules can ensure init is for initialization of integer variable.

18. init -> LC args RC

    This rule is used to set a series of numbers as the initial value of an array. For args, we have attribute array to hold all these numbers and attribute len to hold the length of the array. So we simply copy those value to the corresponding attribute of init.

19. exps -> READ LP ID RP

    This rule is used to call read function with argument having the name ID. So we just produce one param quadruple and one call quadruple. Detailed implementation is shown below:

```
exps :
    READ LP ID RP
    {
        symrec *pt = getsym($3);
        if (pt == 0)
        {//whether ID is defined
            errors++;
            fprintf(stderr, "Semantic Error!\n");
            fprintf(stderr, "Variable %s not declared!\n", $3);
        }
        else if ((pt->type != Integer) && (pt->type != Constant) && (pt->type != Temp))
        {
            errors++;
            fprintf(stderr, "Function argument type not fit.\n");
        }
        else //load argument and call function
        {
            quad_push("param", getsym($3), NULL, NULL, NULL);
            symrec * pt = putconst(1);
            quad_push("call", getsym($1), pt, getsym($3), NULL);
        }
        $<ExprType.isLeft>$ = 0;
        $<ExprType.isInt>$ = 0;
    }
```

The exps -> WRITE LP ID RP is the same as the previous one.

20. exps -> exps ADD exps

Now it' s for binary operations. I will use add operation as an example. For other binary operation, we only need to change the Op field of the quadruple according to the operator. We first generate a temp variable to hold the result of the add result. The if both the two expression is constant, we can directly compute the result of the add operation. Otherwise we will have to generate a quadruple to compute the value. Under both circumstance, the result will bw stored in the address attribute of the reduced element. The detailed implementation is shown below:

```
| exps ADD exps
{
    tempCnt ++;
    symrec *tp;
    tp = puttemp("t", 0, Temp, NULL, tempCnt, 0, 0, 0);//generate the temporal variable
    $<ExprType.address>$ = tp;
    //if both are constant, we can calculate the value
    if (($<ExprType.type>1 == Constant) && ($<ExprType.type>3 == Constant))
    {
        $<ExprType.type>$ = Constant;
        $<ExprType.address>$ = putconst($<ExprType.address>1->ConstVal + $<ExprType.address>3->ConstVal);
    } else //otherwise generate the quadruple to compute the result
    quad_push("+", $<ExprType.address>1, $<ExprType.address>3, $<ExprType.address>$, NULL);
    $<ExprType.isLeft>$ = 0;
    $<ExprType.isInt>$ = 1;

}
```

21. exps -> exps ASSIGNOP exps

This rule is used to assign a value between expressions. So if the left expression has type integer, we will generate a quadruple with Op "=". If the left expression is an array element, we will generate a quadruple with Op "[]=". The symbol record for the array and offset are stored in offsets of left exps. The detailed implementation is shown below:

```
| exps ASSIGNOP exps
{
    if ($<ExprType.isLeft>1 == 0)//left value check
    {
        errors++;
        fprintf(stderr, "Not left value.\n");
    }
    if ($<ExprType.type>1 == Array)
    {
        symrec *zero = putconst(0);
        //assign value to an array
        quad_push("[]=", $<ExprType.array>1, $<ExprType.offset>1, $<ExprType.address>3, NULL);
    }
    else quad_push("=", $<ExprType.address>3, NULL, $<ExprType.address>1, NULL);//assign integer value
    $<ExprType.address>$ = $<ExprType.address>1;
    $<ExprType.isLeft>$ = 0;
    $<ExprType.isInt>$ = 1;
}
```

22. exps -> exps ADDASSIGNOP exps

    This rule is to do the add and the assign at the same time. If the left expression has type array, we will first generate a temporal variable t to store the value of the add operation. Then we generate a “=[]” quadruple to store the value of left expression into t. We also generate “+” quadruple to compute result of adding the left expression and the right expression. Finally we generate “[]=” to store back the result of result. If the left expression has type integer, we only generate one “+’ quadruple to add the left and right expression and store the result to the address attribute of the left expression. Detailed implementation is shown below:

```
| exps ADDASSIGN exps
{
    if ($<ExprType.isLeft>1 == 0)// left value check
    {
        errors++;
        fprintf(stderr, "Not left value.\n");
    }
    if ($<ExprType.type>1 == Array)
    {
        symrec *zero = putconst(0);
        tempCnt ++;
        $<ExprType.address>$ = puttemp("t", 0, Temp, NULL, tempCnt, 0, 0, 0);
        //t get the value of left expression
        quad_push("=[]", $<ExprType.array>1, $<ExprType.offset>1, $<ExprType.address>$, NULL);
        //do the addition
        quad_push("+", $<ExprType.address>$, $<ExprType.address>3, $<ExprType.address>$, NULL);
        //store t to the left expression
        quad_push("[]=", $<ExprType.array>1, $<ExprType.offset>1, $<ExprType.address>$, NULL);
    }
    else
    {
        tempCnt ++;
        $<ExprType.address>$ = puttemp("t", 0, Temp, NULL, tempCnt, 0, 0, 0);
        //add left and right, store to left
        quad_push("+", $<ExprType.address>1, $<ExprType.address>3, $<ExprType.address>1, NULL);
    }
    $<ExprType.isLeft>$ = 0;
    $<ExprType.isInt>$ = 1;
}
```

    Other assign operation is almost the same as ADDASSIGN, except that the “+” operation need to be modified.

23. exps -> exps LAND exps, exps LOR exps, and other Boolean expression for control flow

    The code for those expression is the same as taught in the Dragon Book. So I will not talk too much on this topic. The Dragon Book solution is shown below:

1) $B \rightarrow B_1 \ \texttt{||} \ M \ B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \rightarrow B_1 \ \&\& \ M \ B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

3) $B \rightarrow \ ! \ B_1$ { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \rightarrow (\ B_1 \ )$ { $B.truelist = B_1.truelist$;
$B.falselist \ = \ B_1.falselist$; }

5) $B \rightarrow E_1 \ \textbf{rel} \ E_2$ { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$emit('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto \_}')$;
$emit('\texttt{goto \_}')$; }

6) $B \rightarrow \textbf{true}$ { $B.truelist = makelist(nextinstr)$;
$emit('\texttt{goto \_}')$; }

7) $B \rightarrow \textbf{false}$ { $B.falselist = makelist(nextinstr)$;
$emit('\texttt{goto \_}')$; }

8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

24. exps -> SUB exp

We first get the address for exp, then use a temporal variable and generate a quadruple to compute the negative value of the exp. If exp has type Constant, we simply use the negative value of the constant as the value. The detailed

```
| SUB exp %prec UMIN
{
    symrec *zero = putconst(0);
    tempCnt ++;
    symrec *tp;
    //if exp is constant, we can just get the negative value of the constant as result
    if ($<ExprType.address>2->type == Constant)
    {
        $<ExprType.address>2 = putconst($<ExprType.address>2->ConstVal * -1);
        $<ExprType.address>$ = $<ExprType.address>2;
    }
    else//other wise generate a quadruple to compute the resule
    {
        tp = puttemp("t", 0, Temp, NULL, tempCnt, 0, 0, 0);
        $<ExprType.address>$ = tp;
        quad_push("-", zero, $<ExprType.address>2, $<ExprType.address>$, NULL);
    }
    $<ExprType.isLeft>$ = 0;
    $<ExprType.isInt>$ = 1;
}
```

exps -> BNOT exp is almost the same as the previous one.

25. exps -> ID LP args RP

This rule is used to call a function with arguments stored in args. So we first generate a series of “params” quadruple to load all variables stored in args.array. Then we generate a temporal variable to store the result of the function call. Finally we generate “call” quadruple with the dest field as the temporal variable.

26. exps -> ID arrs

    This rule is used to set the attribute of exps the a variable or array element. If arrs.address = NULL, which means it is an integer variable, we simply set the exp.address=ID' s symbol record. Otherwise it is an array element and we will use a temporal variable to store the value of this element by generating "=[]" quadruple.

27. ID DOT ID

    This rule is used to access the attribute of a structure. Since we have store all the attribute of a structure in the symbol table, we only need to get the corresponding name of the attribute in symbol table and get it. The name of the attribute is in the form "ID.ID". After we find the symbol record, we pass it to the address of the reduced element.

28. INT

    We get the value of an integer, store it in the constant table, and pass the symbol record to the reduced element.

29. arrs -> LB exp RB

    We use a temporal variable t to hold the memory offset of an array element. We generate a quadruple to represent t =exp.address *4 because an integer takes 4 bytes. And pass t to arrs.address.
    arrs -> LB exp RB LB exp RB is almost the same as the previous one, except the offset is exp1.address * col + exp2.address.

30. arrs -> args COMMA exp

    We simple add exp.address to args.array and increase its length so that args will hold all the arguments.

31. func -> ID LP paras RP

    This rule is used to define a function. We have a global variable jumpmain to record the start address of main function because we must generate jump main as the first quadruple. For the function definition routine, we need to store the symbol table and offset, then add parameters to the new symbol table and generate "label" quadruple to indicate the function. The detailed implementation is shown below:

```
func :
    ID LP
    {
        if (jumpmain == NULL)//record the palce of main function
        {
            //jump main should be the first quadruple
            jumpmain = quad_push("jump", NULL, NULL, NULL, NULL);
            jumpmain->target = 0;
        }
        sym_table = install($1, 0, Function, NULL, 0, 0, 0);
        symrec * tempt = sym_table;
        //store symbol table and offset
        ST_Stack_Push(&StStack, sym_table);
        sym_table = 0;
        //generate label to represent the function
        Quad * pt = quad_push("label", NULL, NULL, tempt, NULL);
        if (strcmp($<id>1, "main") == 0)//if the current  function is main, update jump main
        {
            jumpmain->target = pt;
            tempt->isCalled = 1;
        }
        INT_Stack_Push(&OffsetStack, offset);
        offset = 0;

    }
    paras RP
    {
        symrec *pt = getsym($1);
        //record parameter number
        pt->paranum = $4;
        //pass attributes to func
        $<STypeVal.structST>$ = sym_table;
        $<STypeVal.type>$ = Function;
        $<STypeVal.width>$ = offset;
        $<STypeVal.lexeme>$ = $1;
    }
```

32. stmtblock and stmts

    For statements I just follow the instructions on Dragon Book to implement the back-patching and record nextlist. But Dragon Book does not tell us how to implement break and continue, so we have to think about it by ourselves. I find a feasible solution which only need to add two new list: skiplist and breaklist to the original attributes. The Dragon Book solution is shown below:

1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

3) $S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$
{ $backpatch(S_1.nextlist,\ M_1.instr)$;
$backpatch(B.truelist,\ M_2.instr)$;
$S.nextlist\ =\ B.falselist$;
$emit('\texttt{goto}'\ M_1.instr)$; }

4) $S \rightarrow \{\ L\ \}$ { $S.nextlist\ =\ L.nextlist$; }

5) $S \rightarrow A\ ;$ { $S.nextlist\ =\ \textbf{null}$; }

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$emit('\texttt{goto}\ \_')$; }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr)$;
$L.nextlist\ =\ S.nextlist$; }

9) $L \rightarrow S$ { $L.nextlist\ =\ S.nextlist$; }

The break and continue solution is shown below:

(1) S → **repeat** M_1 L **while** M_2 E; { backpatch(L.nextlist,M_2.quad);
backpatch(L.skiplist,M_2.quad);
backpatch(E.truelist.M_1.quad);
S.nextlist = merge(E.falselist,L.breaklist); S.breaklist = nil; }
(2) S → **continue;** { S.skiplist = newList(nextAddr()); emit('goto __'); S.breaklist = nil; S.nextlist = nil;}
(3) S → **break;** { S.breaklist = newList(nextAddr()); emit('goto __'); S.nextlist = nil; S.skiplist = nil; }
(4) L_1 → S ; M_2 L_2 { backpatch(S.nextlist, M_2.quad);
L_1.nextlist = L_2.nextlist;
L_1.breaklist = merge(S.breaklist,L_2.breaklist);
L_1.skiplist = merge(S.skiplist,L_2.skiplist); }
(5) L → S { L.breaklist = S.breaklist; L.nextlist = S.nextlist; L.skiplist = S.skiplist; }
(6) M_1 → ε { M_1.quad = nextAddr; }
(7) M_2 → ε { M_2.quad = nextAddr; }

combine those two, we can get the solution for break and continue in out project.

7) Semantic check

I have implemented **all the semantic analysis** mentioned in the manual. I will briefly introduce the method.

1. Variables and functions should be declared before usage

   I modify the install function so that before we install a variable into the symbol table, we first check whether the variable having the same name is already in the symbol table. If it is, then we print semantic error.

   e.g.

```
int main()
{
    x = 1;
    return 0;
}
```

result

```
Semantic Error!
Variable x not declared!
```

2. Reserved words can not be used as identifiers. Reserved words can be found in TOKENs
   I modify the install function so that before we install a variable into the symbol table, we will check whether the symbol name is valid. If not, then we print semantic error.

   e.g.

```
int main()
{
    int for = 1;
    read(read);
    return 0;
}
```

   result

```
syntax error
Parse Completed
1 Errors, please check!
```

3. Reserved words can not be used as identifiers. Reserved words can be found in TOKENs
   We just need to check whether jumpmain' s target is not NULL

   e.g.

```
int foo()
{
    int for = 1;
    read(read);
    return 0;
}
```

   result

```
syntax error
Parse Completed
Main function not defined.
2 Errors, please check!
```

4. The number and type of variable(s) passed should match the definition of the function.
   In rule exps -> ID LP args RP, we just need to check whether args.len = function ID' s argument length. And we also need to check whether each elements in args.array has type integer.

   e.g.

```
int foo(int x, int y)
{
    return 0;
}
int main()
{
    int k;
    int a[10];
    k = foo(1);
    k = foo(a,1);
    return 0;
}
```

   result

```
Function argument number not fit.
Semantic Error!
Variable a is not Integer!
Function argument type not fit.
Parse Completed
3 Errors, please check!
```

5. Use [] operator to a non-array variable is not allowed.
   In rule exps -> ID arrs, we need to check whether ID has type Array if arrs.address is not empty, which means its an array access.
   e.g.

```
int main()
{
    int k;
    k[1] = 1;
    return 0;
}
```

   result

```
Variable k is not an array.
Parse Completed
1 Errors, please check!
```

6. The . operator can only be used to a struct variable.
   In rule exps -> ID DOT ID, we need to check whether ID has type structure.
   e.g.

```
int main()
{
    int k, x;
    k.x = 1;
    return 0;
}
```

   result

```
Varible k is not struct type.
Semantic Error!
Variable k.x not declared!
Parse Completed
2 Errors, please check!
```

7. break and continue can only be used in a for-loop.
   In the stmt -> FOR LP exp SEMI m exp SEMI m exp n RP m stmt, we modify it into
   Stmt -> FOR LP exp SEMI m exp SEMI m exp n RP m {inloop += 1;} stmt {inloop -= 1;}.
   So that inloop will always count the level of loops of the current state. So when we encounter a continue or break, we need to check whether inloop is >0.
   e.g.

```
int main()
{
    int i;
    break;
    continue;
}
```

   result

```
Break not in loop.
Continue not in loop.
Parse Completed
2 Errors, please check!
```

8. Right-value can not be assigned by any value or expression

   We use attribute isLeft to indicate whether an expression can be left value or not. The only time an expression can be a left value is when exps -> ID arrs or exps -> ID DOT ID. We need to check whether left expression is left value when reducing rules about assign.

   e.g.

```
int main()
{
    int x, y;
    y + 1 = x;
    return 0;
}
```

   result

```
Not left value.
Parse Completed
1 Errors, please check!
```

9. The condition of if statement should be an expression with int type
10. The condition of for should be an expression with int type or ε
11. Only expression with type int can be involved in arithmetic

    We can combine those three requirements together by only allowing integer type during the reducing of expression. And the only time an expression can not be an integer is when exps -> ID arrs. So we need to check whether ID arrs has type Integer when arrs.address is NULL. But the check has already been done in the previous requiremtns. So we' re alredy done here.

    e.g.

```
int main()
{
    int x[10], y;
    if (x) return 1;

    for (y = 0; x; ++y)
    {
        break;
    }

    y = x + 1;
    return 0;
}
```

    result

```
Semantic Error!
Variable x is not Integer!
Semantic Error!
Variable x is not Integer!
Semantic Error!
Variable x is not Integer!
Parse Completed
3 Errors, please check!
```

All of the error test file is in error_test folder with name of error_N.sc. N indicates it is the error file for the Nth requirements.

5. Optimization

1) Introduction

After generating the intermediate-code, we are free to do all kind of optimization to the 3-address-code. In particular, I have implemented **6 optimizations** to the three-address-code.

2) Arithmetic Identities

In real machines, move always cost less time than add/sub, and add/sub always cost less time than mul/div. We can use some properties in maths to transform mul/div/add/sub to move, as shown below:

- $x = a + 0$ or $x = 0 + a \Rightarrow x = a$
- $x = a - 0 \Rightarrow x = a$
- $x = 0 - a \Rightarrow x = -a$
- $x = a - a \Rightarrow x = 0$
- $x = a * 1$ or $x = 1 * a \Rightarrow x = a$
- $x = a * 0$ or $x = 0 * a \Rightarrow x = 0$
- $x = a * -1$ or $x = -1 * a \Rightarrow x = -a$
- $x = a / 1 \Rightarrow x = a$
- $x = a / -1 \Rightarrow x = -a$
- $x = 0 / a \Rightarrow x = 0$
- $x = a / a \Rightarrow x = 1$

The implementation is simple, we just check whether the Arithmetic Identities is satisfied when we add a new quadruple and transform it into move quadruple, part of the implementation is shown below:

```
//OPTIMIZATION: Arithmetic Identities and Constant folding
if (strcmp(Op, "+") == 0)
{

    if ((arg1->type == Constant) && (arg1->ConstVal == 0))  // x=0+a -> x=a
    {
        Op[0] = '=';
        arg1 = arg2;
        arg2 = NULL;
    }

    else if ((arg2->type == Constant) && (arg2->ConstVal == 0))  // x=a+0 -> x=a
    {
        Op[0] = '=';
        arg2 = NULL;
    }
    else if ((arg1->type == Constant) && (arg2->type == Constant))  //x=a+b -> x =c
    {
        Op[0] = '=';
        arg1 = putconst(arg1->ConstVal + arg2->ConstVal);
    }

}
```

e.g.

```
int main()
{
    int x;
    x = x / 1;
    return 0;
}
```

3-address-code code:

```
1:  jump NULL NULL NULL 2
2:  label NULL NULL main 0
3:  = x NULL _t1 0
4:  = _t1 NULL x 0
5:  return 0 NULL NULL 0
```

We can see that the divide operation is optimized out

3) Reduce in Strength

There is another way we can transform a mul to add, which is shown below:

- $x = 2 * a$ or $x = a * 2 \Rightarrow x = a + a$

The implementation is pretty simple:

```
else if (strcmp(Op, "*") == 0)
{
    if ((arg1->type == Constant) && (arg1->ConstVal == 1))  // x=1*a -> x=a
    {
        Op[0] = '=';
        arg1 = arg2;
        arg2 = NULL;
    }
    else if ((arg2->type == Constant) && (arg2->ConstVal == 1))  // x=a*1 -> x=a
    {
        Op[0] = '=';
        arg2 = NULL;
    }
    else if ((arg1->type == Constant) && (arg1->ConstVal == 2))  // x=2*a -> x=a+a
    {
        Op[0] = '+';
        arg1 = arg2;
    }
    else if ((arg2->type == Constant) && (arg2->ConstVal == 2))  // x=a*2 -> x=a+a
    {
        Op[0] = '+';
        arg2 = arg1;
    }
    else if ((arg1->type == Constant) && (arg2->type == Constant))  //x=a*b -> x=c
    {
        Op[0] = '=';
        arg1 = putconst(arg1->ConstVal * arg2->ConstVal);
    }
```

e.g.

```
int main()
{
    int x;
    x = 2 * x;
    return 0;
}
```

result

```
1:  jump NULL NULL NULL 2
2:  label NULL NULL main 0
3:  + x x _t1 0
4:  = _t1 NULL x 0
5:  return 0 NULL NULL 0
```

We can see that the multiply is reduced to add operation.

4) Constant folding

If there exists a quadruple that its 2 operands are all constant, then instead of producing a quadruple that calculate the result of operating those two numbers, we can get the result at compile time and generate a move quadruple to set the value of x, the example is shown below:

- *x = 1 + 2 => x = 3*
- *x = 3 - 1 => x = 2*
- *x = 3 * 2 => x = 6*
- *x = 5 / 2 => x = 2*
- *x = 5 % 2 => x = 1*

To perform the constant folding, we still need to check when we are adding a quadruple into the IR. If the quadruple' s operation is +,-,*,/,%, then we will check whether its two operands have type Constant, if so, we change the operator to "=", and set the arg1 to the constant result of the +,-,*,/,% operator. Part of the implementation is shown below:

```
else if (strcmp(Op, "%") == 0)
{
    if ((arg2->type == Constant) &&  (arg2->ConstVal == 0))//check error
    {
        errors++;
        printf("Error: Divided by zero.\n");
    }
    else if ((arg1->type == Constant) && (arg2->type == Constant))  //x=a%b -> x=c
    {
        Op[0] = '=';
        arg1 = putconst(arg1->ConstVal % arg2->ConstVal);
    }
}
```

In addition, we can check the divided by zero error at the same time.

e.g.

```
int main()
{
    int x = (1 + 2 + 3 * 5 + 5) / 5;
    return 0;
}
```

result

```
1:   jump NULL NULL NULL 2
2:   label NULL NULL main 0
3:   = 4 5 _t5 0
4:   = _t5 NULL x 0
5:   return 0 NULL NULL 0
```

This is really an effective optimization, which greatly reduce the number of quadruples. What' s more, this optimization can be much powerful if we combine it with constant propagation and assign elimination.

5) Constant propagation

Constant propagation is an optimization method that allow us to propagate the value of a constant stored in a register to the following quadruples. For example, consider the following quadruples:

```
a = 2
b = c * a
a = 5
a = a + b
c = a + 2
```

It can be optimized into:

```
a = 2
b = c * 2
a = 5
a = 5 + b
c = a + 2
```

Constant propagation should be carried out in a basic block.

In each basic block we scan the code in sequence, if we find the a quadruple satisfying one of the following properties, we begin constant propagation:

1. It is a "=" quadruple and arg1 is Constant type
2. It is a binary quadruple and arg1 and arg2 are both constant, the we modify the Op of this quadruple into "=" and set arg1 as the constant result of the operation.

To propagate, we scan trough all the quadruples after the current one in the block, if we find a quadruple with arg1 or arg2 equal to the current quadruple' s dest then we can replace arg1 or arg2 with the constant result of the current quadruple.

e.g.

We use the same code as above:

```
int main()
{
    int a = 2;
    int c;
    int b = a * c;
    a = 5;
    a = a + b;
    c = a + 2;
    return 0;
}
```

result

```
1:  jump NULL NULL NULL 2
2:  label NULL NULL main 0
4:  * 2 c _t1 0
5:  = _t1 NULL b 0
7:  + 5 b _t2 0
8:  = _t2 NULL a 0
9:  + a 2 _t3 0
10:  = _t3 NULL c 0
11:  return 0 NULL NULL 0
```

We can see that the constant value is propagated into next quadruples.

6) Assign Elimination

Having propagated constants, we are ready to remove the unused assign of constant number to registers. For example, in the example of 5), if a' s value is propagated, then the initialization of a is not needed, so we can remove the assign of a. Still, the elimination of assign should be in a basic block. Basically the logic of assign elimination is as follows:

We scan the code in sequence, if we find a "=" quadruple, we call it current. Then we begin assign elimination.

For assign elimination, we search all the quadruples after it, the search will end if we reach the end of a block or we reach a quadruple whose dest is current' s dest. During the search if we find that there is a quadruple that needs to read the value of current' dest, then current cannot be eliminated. Otherwise current will be eliminated.

e.g

```
int main()
{
    int a = 2;
    int b = 3;
    int c = 9;
    int x = 9;
    int y = 0;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = 1000;
    x = c + b * a;
    y = c/b*x - 1000;
    write(y);
    return 0;
}
```

This code contains a lot of assigns that can be eliminated, and the value of y and x can be determined at compiler time without the need to compute in MIPS code.

Result

```
1:  jump NULL NULL NULL 2
2:  label NULL NULL main 0
3:  = 2 NULL a 0
4:  = 3 NULL b 0
5:  = 9 NULL c 0
22:  = 15 NULL x 0
26:  = -955 NULL y 0
27:  return 0 NULL NULL 0
```

We can see that all the values for variables are determined in compile time and useless assignments are eliminated. The IR is even shorter that the high-level program, which is really amazing!

7) Unreachable Function Elimination

Another minor optimization is the elimination of unreachable functions. We just need to scan over the program and record all the functions we have called during the program. For those functions which are not called in the program, we simply do not generate the code for them.

e.g.

```
int foo(int x)
{
    int a = 2;
    a = 1;
    a = 1;
    a = 1;
    a = 1;
    a = 1;
    a = 1;
    a = 1;
    a = 1;
    a = 1;
}
int main()
{
    int x;
    return x;
}
```

result

```
1:  jump NULL NULL NULL 13
2:  label NULL NULL foo 0
12:  = 1 NULL a 0
13:  label NULL NULL main 0
14:  return x NULL NULL 0
```

We can see that foo is not called in the function so we do not generate its code, only leave its label.

All the optimization example file is included in /opt_test directory and is named by optim_N.sc, meaning the example for the Nth optimization.

6. Code Generation

1) Introduction

The last part of the project is still very difficult. In this part we are faced with three problems: register allocation, stack allocation and instruction selection and. I will talk about them separately.

2) Register Allocation

For register allocation I use an algorithm different from the Dragon Book. The algorithm is as follows:

```
1  for each operation z = x op y
2    rx = Ensure(x)
3    ry = Ensure(y)
4    if (x is not needed after the current operation)
5      Free(rx)
6    if (y is not needed after the current operation)
7      Free(ry)
8    rz = Allocate(z)
9  emit MIPS32 code for rz = rx op ry
```

Here Free(rx) means spill all the variables stored in the register into the memory.
Ensure and Allocate are defined as follow:

```
Ensure(x):
  if (x is already in register r)
    result = r
  else
    result = Allocate(x)
    emit MIPS32 code [lw result, x]
  return result

Allocate(x):
  if (there exists a register r that currently has not been assigned to
      any variable)
    result = r
  else
    result = the register that contains a value whose next use is farthest
        in the future
    spill result
  return result
```

Having decided the algorithm, we then need to design the register. The register is defined as follows:

```
struct Register
{
    int id; //the unique id for each register
    RType type; //register type, e.g. S, T, V, GP, ST, ...
    ST_Stack reg_descriptor; //record the variable stored in the register
} ;
```

For register allocation, we define hasEmptyMipsReg() to determine if there is a S or T register that holds no variable.

The design of Ensure is as follow:

```
//check whether there is a register containning x, if no, allocate one for x
int Ensure(Quad *bs, symrec *x)
{
    int result = -1;
    int i;
    for (i = 0; i < 32; i++)
    {//check whether there is a register containning x
        if (ST_Stack_has(&(MipsReg[i].reg_descriptor), x) == 1)
        {
            result = i;
            return result;
        }
    }
    if (result == -1)//if no, allocate one for x
    {
        result = Allocate(bs, x);
        if (x->type == Constant)
        {
            printf("li %s, %d\n", RegName(result), x->ConstVal);
            fprintf(outfp, "li %s, %d\n", RegName(result), x->ConstVal);
        }
        else if (x->isGlobal == 1) //Global variable shoule be accessed using name
        {
            printf("la $a2, _glb_%s\n", x->name);
            fprintf(outfp, "la $a2, _glb_%s\n", x->name);
            printf("lw %s, 0($a2)\n", RegName(result));
            fprintf(outfp, "lw %s, 0($a2)\n", RegName(result));
        }
        else //local variable shoule be accessed using offset
        {
            printf("lw %s, -%d($fp)\n", RegName(result), x->offset+4);
            fprintf(outfp, "lw %s, -%d($fp)\n", RegName(result), x->offset+4);
        }

    }
    return result;
}
```

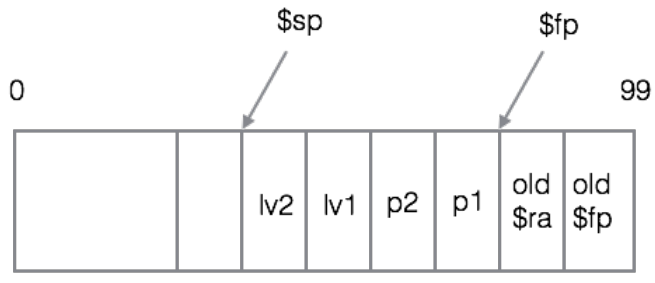
Pay attention that we use different routine for storing global variable and local variable.

The design of Allocate is as follow:

```
//allocate a register to store variable x
int Allocate(Quad *bs, symrec *x)
{
    int result = -1;
    int i, j;
    for (i = 0; i < 32; i++)
    {
        //have empty register or a register already has x
        if (((ST_Stack_isEmpty(&(MipsReg[i].reg_descriptor)))|| (ST_Stack_has(&(MipsReg[i].reg_descriptor), x)))
            && ((MipsReg[i].type == S) || (MipsReg[i].type == T)))
        {
            result = i;
            if (ST_Stack_isEmpty(&(MipsReg[i].reg_descriptor)))
                ST_Stack_Push(&(MipsReg[result].reg_descriptor), x);
            return result;
        }
    }
    int max = bs->id;
    if (result == -1) //if not, spill one register
    {
        for (i = 0; i < 32; i++)
        {
            //find the register that contains the variable used last in the block
            if ((MipsReg[i].type == S) || (MipsReg[i].type == T))
            {
                for (j = 0; j < MipsReg[i].reg_descriptor.size; j++)
                {
                    int temp = get_last_used_id_in_block(bs->next, MipsReg[i].reg_descriptor.data[j]);
                    if (temp >= max)
                    {
                        max = temp;
                        result = i;
                    }
                }
            }
        }
        //TODO spill result
        Free(result);
        ST_Stack_Push(&(MipsReg[result].reg_descriptor), x);
    }
    return result;
}
```

3) Stack Allocation

In MIPS, all the local variable is stored in the stack. So we need to find a way to allocate the stack properly. In my project, I use register $sp as the stack top pointer and $fp as the stack base pointer, just like the %esp and %ebp in X86. We store variables with offset x (recorded in symbol table) in place –x($fp), which makes it easy to access variables. When we encounter a function call, we first push the current $fp in stack, then push $ra (return address) in stack. Then we further push all the function parameters into stack and set $fp to the address where the first parameters begins at. So at any time, our stack is as follows:



p means parameters
lv means local variables

4) Instruction Selection

In this part we need to decide the corresponding MIPS code for each quadruples. I will only demonstrate some representative quadruples.

1) label quadruple

   We should print “label: target” MIPS code. And if the label target is a function, we should arrange offset for temporal variables for the function’ s symbol table

2) []= quadruple

   The accessing routine for array is different between global array and local array. Since global is defined in .data segment, we need to locate the address using the name of the array. For local array we need to use the offset and the $gp stack. Stack allocation will be covered later. The program logic of []= quadruple is as follows, suppose []= means arg1[arg2] = dest:

if arg1 is global variable and dest has type Const	allocate register Ry for arg2
	emit:
	la $a0, global_variable_name
	add $a0, $a0, Ry
	li $a1, dest->ConstValue
	sw $a1, 0($a1)

if arg1 is global variable and dest is not Const	allocate register Ry for arg2
	allocate register Rz for dest
	emit:
	la $a0, global_variable_name
	add $a0, $a0, Ry
	sw Rz, 0($a0)

if arg1 is not global and dest is const	allocate register Ry for arg2
	emit:
	addi $a0, $fp, arg1->offset
	add $a0, $a0, Ry
	li $a1, dest->ConstValue
	sw $a1, -4($a0)

if arg1 is not global and dest is not const	allocate register Ry for arg2
	emit:
	addi $a0, $fp, arg1->offset
	add $a0, $a0, Ry
	sw Rz, -4($a0)

=[] quadruple is almost the same as []= quadruple, except we need to replace sw with lw.

3) param quadruple
   We push all the variable into the stack and set $fp to the beginning place of the first parameter when we finally call a function after a series of param quadruples.

4) call quadruple
   For call quadruple, we need to emit “jal arg1->name” and move the value in $v0 to arg1->dest. We regulate that the return value of a function is stored in $v0. Before jal, we need to spill all the variables stored in the register into memory.

5) branch quadruple
   For branch, we simple load the two operands, arg1 and arg2 into register and emit MIPS code for branch. The constant value can be loaded into $a0 and $a1 in my project. In my project, I do not use $a0~$a3 to hold the argument. Instead, I use them to load constant value.

6) arithmetic quadruple
   For arithmetic quadruple, we simply use the register allocation algorithm mentioned in the register allocation section.

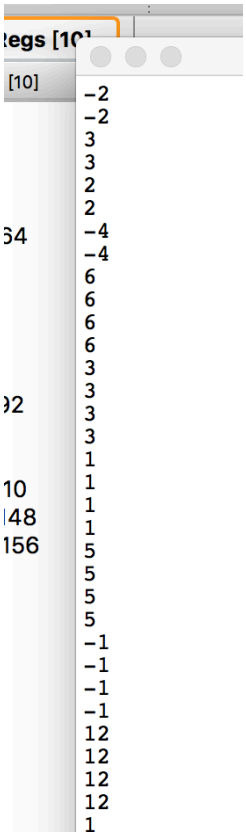
7. Tests

To ensure the correctness of the compiler, I set up multiple tests. And I will show the result of each test.

1) arith_test
   In this test we test all the arithmetic calculation defined in smallC.
   Part of the program is shown below:

```
int main()
{
    int x = 1, y = 2, z = 3;

    x = -y;
    write(x);
    x = -2;
    write(x);

    x = ++y;
    write(x);
    write(y);

    x = --y;
    write(x);
    write(y);

    x = ~3;
    write(x);
    x = ~z;
    write(x);

    x = 2 * 3;
    write(x);
    x = y * 3;
    write(x);
    x = 2 * z;
    write(x);
    x = y * z;
    write(x);
```

Part of the result:



2) test_struct

This program test the correctness of structure

```
struct stu {int x; int y;} z;
int main()
{
    struct {int x; int y;} g;
    struct stu x;
    z.x = 1;
    x.y = 2;
    g.y = 3;
    write(z.x);
    write(x.y);
    write(g.y);
    return 0;
}
```

Result

```
1
2
3
```

3) Hex_oct_test

This program is used to test whether the program can read hex number and oct number.

```
int main()
{
    int x = 0xfff;
    int y = 010;
    x = x << 1;
    write(x);
    write(y);
    return(0);
}
```

The result is:

```
8190
8
```

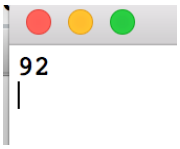
4) eight_queen

Yes, my smallC compiler can compile and run the **eight queen problem** smoothly!

The program is as follows:

```
int chess[8][8]={0};
int a[8],b[15],c[15];
int sum=0;
int PutQueen(int n)
{
  int col,i,j;
  for(col=0;col<8;++col)
  {
    if((a[col] == 1)&& (b[n+col] == 1) &&(c[n-col+7] == 1))
    {
      chess[n][col]=1;
      a[col]=0;
      b[n+col]=0;
      c[n-col+7]=0;
      if(n==7)
      {
        ++sum;
      }
      else
        PutQueen(n+1);
      chess[n][col]=0;
      b[n+col]=1;
      c[n-col+7]=1;
      a[col]=1;
    }
  }
  return 0;
}
int main()
{
  int i;
  for(i=0;i<8;++i)
    a[i]=1;
  for(i=0;i<15;++i)
  {
    b[i]=1;
    c[i]=1;
  }
  PutQueen(0);
  write(sum);
  return 0;
}
```

The result is:



which is the correct answer for the eight queen problem.

5) fib

My smallC compiler can also run the recursive Fibonacci algorithm

The program is as follows:

```
int fib(int n)
{
    if (n <= 2) return 1;
    else return (fib(n-1)+fib(n-2));
}
int jj(int x)
{
    return 0;
}
int main()
{
    int x;
    read(x);
    write(fib(x));
    return 0;
}
```

The result is (I enter 20)

```
Enter an integer:20
6765
```

6) gcd

The program of finding common divisor can also be carried out in smallC

The program is as follows:

```
int gcd(int a, int b)
{
    int i;
    if (b==0)
        i=a;
    else
        i=gcd(b,a%b);
    return i;
}
int main()
{
    int m,n;
    read(m);
    read(n);
    write(gcd(m,n));
    return 0;
}
```

The result is:

```
Enter an integer:192
Enter an integer:256
64
```

7) Hanoi

My smallC can also solve the problem of Hanoi Tower! The program is as follow:

```
int mov(int n,int a, int b, int c)
{
    if(n==1)
    {
        write(a);
        write(0);
        write(c);
    }
    else
    {
        mov(n-1,a,c,b);
        write(a);
        write(0);
        write(c);
        mov(n-1,b,a,c);
    }
    return 0;
}

int main()
{
    int n;
    read(n);
    mov(n,1, 2, 3);
    return 0;
}
```

The result is:

```
Enter an integer:3
1
0
3
1
0
2
3
0
2
1
0
3
2
0
1
2
0
3
1
0
3
```

Here 1 0 3 means move from 1 to 3.

8) Loop Test

This program is used to test whether the loop, break and continue is right.

The program is as follows:

```
int main()
{
    int i;
    int j;
    int t;
    for (i = 0; i < 10; ++i)
    {
        if (i > 5) break;
    }
    write(i);
    for (i = 0; i < 10; ++i)
    {
        t = t + 1;
        for (j = 0; i < 10; ++j)
            if (j > 5) break;
        continue;
        t = 0;
    }
    write(t);
    return 0;
}
```

The result is:

```
6
10
```

9) Sort

In this program we perform a selection sort to a global array. The program is as follows:

```
int a[10] = {-1,134,124,64,908,1324,5243,2,0,-341};
int sort(int len)
{
    int i=0;
    int j;
    int t;
    for(i=0;i<len;++i)
    {
        for(j=0;j<len-i-1;++j)
        {
            if(a[j]>a[j+1])
            {
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
        }
    }
    return 0;
}
int main()
{
    int i=0;
    sort(10);
    for(i=0;i<10;++i)
    {
        write(a[i]);
    }
    return 0;
}
```

The result is:

```
-341
-1
0
2
64
124
134
908
1324
5243
```