CS308 Compiler Principles Project 2 Report

Weichen Li
5120309662
Department of Computer Science and Engineering
Scholl of Electronic Information and
Electrical Engineering
Shanghai Jiaotong University
Email: eizo.lee@sjtu.edu.cn

***Abstract*—In project 1 we have finished some initial preparation of Lexer and Parser, using Lex and Yacc respectively. Now in project 2, our aim is to build a complete compiler for a language called *SMALLC*, based on our previous work, and generate intermediate code as well as target code, which will be tested by MIPS simulator SPIM.**

I. Introduction

C and some C-liked languages are the dominant programming languages. In this project, we are required to design and implement a simplified compiler, for a given programming language, namely *SMALLC* , which is a simplfied C-liked language containing only the core part of C language. After finishing this project,we can get a compiler, which can translate *SMALLC* source codes to MIPS assembly codes. These assembly codes can run on the SPIM simulator, or we can assemble it to machine code to run it on a real computer. In this project, Linux environment (e.g., Ubuntu or CentOS) is required.

To build a whole compiler may seem to be a huge, difficult, and complicated task (and actually it is).

To make this project easier, we divide the procedure into 5 parts. The first part is to write a lexical analyser to split the source codes into tokens. The second part is to write a parser for parse tree generation. In the third part we are required to check the parse tree to make sure that there are no semantic errors. Then we translate the parse tree into an intermediate representation (IR). The forth part, which is optional, requires us to optimize the IR. And the last part is to generate the target assembly codes.

II. Lexical Analyzer

In this section, we are going to write a lexical analyzer. The lexical analyser reads the source codes of *SMALLC* and separates them into tokens.

A. Tokens

There are several tokens in my grammar, see in TABLE.I

And there are 13 levels of operators, including binary operators and unary operators (details are in project introduction PDF).

TABLE I
Tokens

READ	read function
WRITE	write function
INT	integer(hex,oct,dec)
ID	identifier
SEMI	;
COMMA	,
BINARYOP	+,-,==,...
UNARYOP	!,++,...
TYPE	int
LP	(
RP	)
LB	[
RB	]
LC	{
RC	}
STRUCT	struct
RETURN	return
IF	if
ELSE	else
BERAK	break
CONT	continue
FOR	for

B. Flex

A key point in programming *.l* file is to return correct tokens, especially integers.

I seperate integers into three categories:

1) hexdigit (0x—0X)(digit—[a-fA-F])+
2) octdigit 0([0-7])+
3) decdigit (digit)+

So if one of them are detected by lex, the value of it will be checked. The code below presented how an octonary is returned with a decimal value. (*charToint(char x)* is a function returns corresponding integer value given a character x, for example, charToint(A) = 10)

```
{octdigit} { int k = 1; int sum = 0;
while (k < yyleng){
  sum += charToint(yytext[k++])*pow(8,yyleng-k);
```

```
  }
yylval.value = sum;
return INT;}
```

Lexical analyzer building is not hard at all compared with the next sections.

III. Symbol Table

At the very beginning I constructed the base of the compiler–Symbol Table.

A. Structure

I used a hash function to index in the Symbol Table, the main idea is to make every character of a name count, and calculate an overall value with shift operation and module operation. The value is returned as table index. With an index we can find an entry in the table.

There are 5 types of unit in Symbol Table:

- VAR_RECORD
- FUNC_RECORD
- ARR_RECORD
- STR_RECORD
- STR_BUILD_RECORD

VAR_RECORD means a variable, global or local, we need to record its scope, type, variable number(used to distinguish units from each other).

FUNC_RECORD means a function, only global, we need to record its scope, son scope(the scope its local variables are in) type, label number(used to distinguish when functions' names are the same, but parameters differ), and numbers of parameters.

ARR_RECORD means an array, global or local, we need to record its scope, type, variable number, dimension, numbers of elements in 1st dimension, numbers of elements in 2nd dimension.

STR_RECORD means a structure definition and STR_BUILD_RECORD, global or local, we need to record their scope, type, variable number, and member. Each member has a name and an index, I defined a struct called StructMemberRec to store them, and I linked members of STR_RECORD and STR_BUILD_RECORD together.

```
//a member in a struct
typedef struct StructMemberRec{
    string name;
    //0,4,8,12...
    int index;
    struct StructMemberRec *next;
} *StructMember;
```

So there are 5 kinds of insert operations: variable insert, array insert, function insert, struct definition insert and struct instance insert. Each of the insertion is achieved with a function respectively, we'll talk about these functions later.

The structure of the Symbol Table is in Fig.1

The table on the left end is a pointer array. Each pointer(called BucketList, indicated by dash line arrow) will point to NULL, or an unit called BucketListRec (defined as struct), which contains a string (the name) and a pointer(called LineList, indicated by full line arrow), LineList will point to NULL or an unit called LineListRec, which contains information and a LineList pointer named next.(See code below)

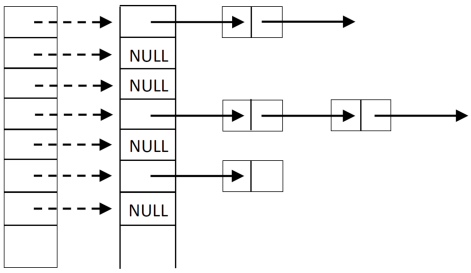

Fig. 1. Symbol Table Structure

So, units with the same name will be in one link in Symbol Table. And if a BucketList is not NULL, it must point to an instance of BucketListRec, whose LineList will point to at least 1 instance of LineListRec.

```
typedef struct LineListRec{
    int scope;
    int son_scope;
    recordType type;

    //for variable and array to distinguish
    //in intermediate code
    int VarNum;
    int LabNum;

    //for array
    int dimen;
    int members_1;
    int members_2;

    //for function
    int paraNum;

    //for struct
    StructMember members;

    struct LineListRec * next;
} * LineList;

typedef struct BucketListRec{
    string name;
    LineList lines;
} * BucketList;

//SIZE is 211
static BucketList hashTable[SIZE];
```

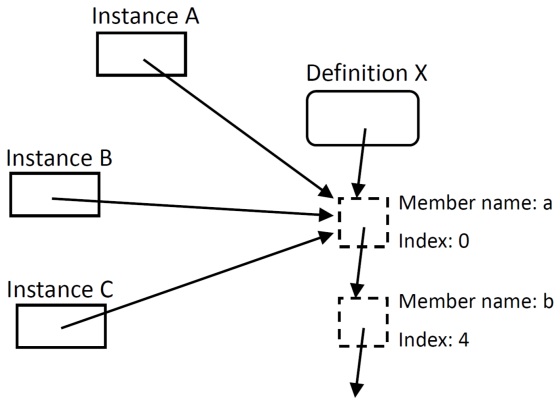

Fig. 2. Struct members link

B. Functions

Like mentioned above, there are 5 functions to operate Symbol Table.

1) *void var_insert(string name);*
2) *void func_insert(string name, int paraNum);*
3) *void arr_insert(string name, int dimen, int members_1, int members_2);*
4) *void str_def_insert(string name);*
5) *void str_build_insert(string name, StructMember members);*
6) *LineList st_lookup(string name, recordType type, int paraNum);*
7) *LineList st_check_scope(string name, recordType type, int paraNum);*

The 1st function will insert a variable with name *name*.

The 2nd funtion will insert a function with name *name*, and numbers of parameters *paraNum*.

The 3rd function will insert an array with name *name*, and dimension *dimen*, numbers of elements in 1st dimension *members_1*, numbers of elements in 2nd dimension *members_2*.

The 4th function will insert a struct definition with name *name*, the members of it will be added later when parser arrives at the struct body.

The 5th function will insert a struct instance with name *name*, and a members link *members*. Actually the link head is in a struct definition, the members pointer inside the struct instance just point to the link head. See in Fig.2 and the corresponding code is below, the full line arrow is the members pointer.

```
struct X{
    int a;
    int b;
};
struct X A;
struct X B;
struct X C;
```

The 6th function will look for the nearliest instance in scope with name *name*, type *type*, and numbers of parameters *paraNum*(If not a FUNC_RECORD type, set it 0). This funciont will return NULL if no such instance, or return the pointer if find one.

The 7th function will look for the instance in present scope with name *name*, type *type*, and numbers of parameters *paraNum*(If not a FUNC_RECORD type, set it 0).

When each Symbol Table unit is inserted, its scope will be recorded, the scope calculation will be introduced later.

IV. Runtime Setting

A. variable number

I set a global variable called VarCounter to count the units I inserted into Symbol Table.

VarCounter only increases using function newtemp().

B. label number

I set a global variable called LabCounter to count the labels I used in generating three address code.

LabCounter only increases using function newlabel().

C. scope setting

I used a global variable SCOPE to record the next available scope.

SCOPE only increases, unless parameter definition is end, it will decrease by 1. (to guarantee that parameters and local variables of the same function are in the same scope)

Instead of using a tree to trace the a dependency relationship of scopes, I used a 2-dimension array parent_table and a stack parent_stack.

For example, parent_table[1][3] == 1 means scope 3 is embedded in scope 1.

Every time the parser enter a new scope(can be a parameter definition, or a new statement block), the SCOPE will increase by 1, and pushed into parent_stack. When parser is leaving the scope, the parent_stack pops.

These work is done in non-terminals **M1**, **M2** and **M3**.

V. Grammar

In this section I solved the "IF ELSE" conflict by modifying the grammar. And with the process of semantic analysis carrying on, I inserted some non-terminals to perform attribute inheriting and intermediate code generation as well. So this section is combined with all the steps needed to generate the intermediate code (I chose to use three address code).

I didn't put the semantic actions here(details are in source code file smallc.y), but this grammar is vital to the entire project.

In addition, I modified some parts of the grammar.

For example, the production $INIT \rightarrow EXP$ is changed into $INIT \rightarrow EXPS$. Because *INIT* is following *ASSIGN*, it can't be empty.

And in production $DEFS \rightarrow STSPEC\ SDECS\ SEMI\ DEFS$ I changed *SDECS* into *SEXTVARS*, for that this production is used to define a type of struct and instantiate some instances, *SEXTVARS* has exactly the function we need, as long as we check whether it is empty when it is reduced.(*SDECS* will not be empty but *SEXTVARS* will)

A. New Grammar

The new grammar is below, as you can see, there are many new non-terminals. I will introduce their functions in later sections.

```
1 PROGRAM: EXTDEFS

2 EXTDEFS: EXTDEF EXTDEFS
3 | %empty

4 EXTDEF: TYPE EXTVARS SEMI
5 | STSPEC SEXTVARS SEMI
6 | TYPE FUNC STMTBLOCK

7 SEXTVARS: ID
8 | ID COMMA SEXTVARS_COMMA SEXTVARS
9 | %empty

10 EXTVARS: VAR EXTVARS_VAR_ALONE
11 | VAR EXTVARS_VAR_ALONE ASSIGN INIT
EXTVARS_VAR_INIT
12 | VAR EXTVARS_VAR_ALONE COMMA EXTVARS
13 | VAR EXTVARS_VAR_ALONE ASSIGN INIT
EXTVARS_VAR_INIT COMMA EXTVARS
14 | %empty

15 STSPEC: STRUCT ID STRUCT_DEFINE LC
STSPEC_SDEFS SDEFS RC
16 | STRUCT STRUCT_EMPTY LC STSPEC_SDEFS
SDEFS RC
17 | STRUCT ID

18 FUNC: ID FUNC_ID LP M1 PARAS M3 RP

19 PARAS: TYPE ID PARAS_ID COMMA PARAS
20 | TYPE ID PARAS_ID
21 | %empty

22 STMTBLOCK: LC M1 DEFS STMTS RC M2

23 STMTS: STMT STMT_M1 STMTS
24 | %empty

25 STMT: IF LP EXPS EXPS_N2 RP STMT_M
STMT1 STMT_N ELSE STMT_M STMT STMT_N
26 | IF LP EXPS EXPS_N2 RP STMT_M STMT
STMT_N
27 | EXPS SEMI
28 | STMTBLOCK
29 | RETURN EXPS SEMI
30 | FOR LP EXP SEMI STMT_M EXP EXPS_N2
STMT_N_1 SEMI STMT_M EXP STMT_N_1 RP
STMT_M STMT STMT_N
31 | CONT SEMI
32 | BREAK SEMI
33 | WRITE LP EXPS RP SEMI
34 | READ LP EXPS RP SEMI

35 STMT1: IF LP EXPS EXPS_N2 RP STMT_M
STMT1 STMT_N ELSE STMT_M STMT1 STMT_N
36 | EXPS SEMI
37 | STMTBLOCK
38 | RETURN EXPS SEMI
39 | FOR LP EXP SEMI STMT_M EXP EXPS_N2
STMT_N_1 SEMI STMT_M EXP STMT_N_1
RP STMT_M STMT1 STMT_N
40 | CONT SEMI
41 | BREAK SEMI
42 | WRITE LP EXPS RP SEMI
43 | READ LP EXPS RP SEMI

44 DEFS: TYPE DECS SEMI DEFS
45 | STSPEC SEXTVARS CHECK_SEXTVARS
SEMI DEFS
46 | %empty

47 SDEFS: TYPE SDEFS_SDECS SDECS SEMI
SDEFS_SEMI SDEFS
48 | %empty

49 SDECS: ID SDECS_ID COMMA SDECS_COMMA
    SDECS
50 | ID SDECS_ID

51 DECS: VAR DECS_VAR_ALONE
52 | VAR DECS_VAR_ALONE COMMA DECS
53 | VAR DECS_VAR_ALONE ASSIGN INIT
DECS_VAR_INIT COMMA DECS
54 | VAR DECS_VAR_ALONE ASSIGN INIT
DECS_VAR_INIT

55 VAR: ID
56 | VAR LB INT RB

57 INIT: EXPS
58 | INIT_ARGS LC ARGS RC

59 EXP: EXPS
60 | %empty

61 EXPS: MINUS EXPS
62 | LOGICNOT EXPS
63 | PREINCRE EXPS
64 | EXPS PREINCRE
65 | PREDEC EXPS
66 | EXPS PREDEC
67 | BITNOT EXPS
68 | EXPS PRODUCT EXPS
69 | EXPS DIVISION EXPS
70 | EXPS MODULUS EXPS
71 | EXPS PLUS EXPS
72 | EXPS MINUS EXPS
```

```
73 | EXPS SHIFTLEFT EXPS
74 | EXPS SHIFTRIGHT EXPS
75 | EXPS GREATERT EXPS
76 | EXPS LESST EXPS
77 | EXPS NOTLESST EXPS
78 | EXPS NOTGREATERT EXPS
79 | EXPS EQUAL EXPS
80 | EXPS NOTEQUAL EXPS
81 | EXPS BITAND EXPS
82 | EXPS BITXOR EXPS
83 | EXPS BITOR EXPS
84 | EXPS LOGICAND EXPS_N STMT_M EXPS
85 | EXPS LOGICOR EXPS_N STMT_M EXPS
86 | EXPS ASSIGN EXPS
87 | EXPS PLUSASSIGN EXPS
88 | EXPS MINUSASSIGN EXPS
89 | EXPS PRODUCTASSIGN EXPS
90 | EXPS DIVISIONASSIGN EXPS
91 | EXPS ANDASSIGN EXPS
92 | EXPS NORASSIGN EXPS
93 | EXPS ORASSIGN EXPS
94 | EXPS SLASSIGN EXPS
95 | EXPS SRASSIGN EXPS
96 | LP EXPS RP
97 | ID EXPS_ARGS LP ARGS RP
98 | ID ARRS
99 | ID DOT ID
100| INT

101 ARRS: LB EXPS RB ARRS
102 | %empty

103 ARGS: EXP ARGS_COMMA COMMA ARGS
104 | EXP ARGS_COMMA

105 EXTVARS_VAR_ALONE: %empty

106 EXTVARS_VAR_INIT: %empty

107 INIT_ARGS: %empty

108 ARGS_COMMA: %empty

109 DECS_VAR_ALONE: %empty

110 DECS_VAR_INIT: %empty

111 SEXTVARS_COMMA: %empty

112 STRUCT_DEFINE: %empty

113 STSPEC_SDEFS: %empty

114 STRUCT_EMPTY: %empty

115 SDEFS_SEMI: %empty

116 SDEFS_SDECS: %empty

117 SDECS_ID: %empty

118 SDECS_COMMA: %empty

119 CHECK_SEXTVARS: %empty

120 FUNC_ID: %empty

121 PARAS_ID: %empty

122 EXPS_ARGS: %empty

123 STMT_M: %empty

124 STMT_M1: %empty

125 STMT_N: %empty

126 STMT_N_1: %empty

127 EXPS_N: %empty

128 EXPS_N2: %empty

129 M1: %empty

130 M2: %empty

131 M3: %empty
```

VI. Semantic Analysis & Intermediate Code Generation

While the parser is operating, there will be numbers of shift operation and reduce operation.

When a shift operation is performed, we can inherit some attributes from the previous production to the next production.

When a reduce operation is performed, we can synthesize some attributes to the target production we are reducing to.

In the next subsections I will introduce the method I used in non-terminals to generate intermediate code.

I built a 2-dimension array called code to store three address code in the memory, when it is optimized, it will be written into the file 'InterCode'.

A. Inherited attributes & Synthesized attributes

To get attributes inherited, we can just simply use command like this: \$0.place, \$-2.value. the number after \$ indicate the token whose attribues is wanted by us. That token is former than the present token we are handling.

The code below shows all the attributes I used.

I use bold font to indicate non-terminals.

- *var_type*: indicate whether **VAR** returns a variable or an array.

- *arrs_type*: tell **ARRS** whether **ID** is a variable or an array. This attribute will be passed on with **ARRS**.
- *exp_type*: indicate whether **EXP** is empty or not.
- *exps_type*: used in statement, to indicate whether **EXPS** is a boolean expression or not.
- *extvars_type*: indicate whether **EXTVARS** is empty or not.
- *paras_type*: indicate whether **PARAS** is empty or not.
- *args_type*: indicate whether parser is handling a source code which is calling a function. Inherited by **ARGS**.
- *args_count*: count the number of parameters of the function the source code is calling. Inherited by **ARGS**.
- *has_return*: check whether there is a return in **STMT**.
- *name*: store the name of **ID**.
- *isLeft*: indicate whether **EXPS** is left value or not.
- *value*: store the value of **INT**.
- *place_1*: store the number of temporary variable which stores the offset of the 1st dimension. used in **ARRS**.
- *place_2*: store the number of temporary which stores the offset of the 2nd dimension. used in **ARRS**.
- *isleft_1*: indicate whether the temporary which stores the offset of the 1st dimension is left value or not. used in **ARRS**.
- *isleft_2*: indicate whether the temporary which stores the offset of the 2nd dimension is left value or not. used in **ARRS**.
- *members_1*: indicate the element of the 1st dimension of an array. used in **VAR**.
- *members_2*: indicate the element of the 2nd dimension of an array. used in **VAR**.
- *dimen*: indicate the dimension of an array. used in **VAR**.
- *arraytop*: indicate the base of an array, used in **ARGS** to initiate the array.
- *arrayoffset*: indicate the offset of the current element of an array, used in **ARGS** to initiate the array.
- *stspec_type*: indicate whether the **STSPEC** is defining a definition or creating an instance of struct.
- *sextvars_type*: indicate whether **SEXTVARS** is empty or not.
- *members*: link the members of a new struct definition, used in **SDEFS**.
- *place*: store the temporary which is storing the data we need.
- *commacount*: count the commas.
- *truelist*: used in backpatching.
- *falselist*: used in backpatching.
- *nextlist*: used in backpatching.
- *breaklist*: used in backpatching.
- *continuelist*: used in backpatching.
- *instr*: the line number of an intermediate code instruction.
- *stmts_type*: indicate whether **STMTS** is empty or not.
- *for_break*: indicate whether **STMT** containing **BREAK** or not.
- *for_continue*: indicate whether **STMT** containing **CONTINUE** or not.

```
struct Type
{
//1:var 2:array
int var_type;

//1:var 2:array
int arrs_type;

//0:none 1:EXPS
int exp_type;

//0:not bool exps  1: bool exps
//2:EXP is empty
//used in for statement
int exps_type;

//0:none 1:not-none
int extvars_type;

//0:none 1:not-none
int paras_type;

//0:not func 1:func
int args_type;

//count the args when call func
int args_count;

//check_whether return is in STMT
int has_return;

//for IDs
string name;

//for variables and integer
int isLeft;
int value;

//for arrays
int place_1;
int place_2;
int isleft_1;
int isleft_2;
int members_1;
int members_2;
int dimen;
int arraytop;
int arrayoffset;

//for struct
//1: define new 2:build new
int stspec_type;
int sextvars_type;
StructMember members;

//run-time
```

```
//for right value,tx stores value
//for left value, tx stores address
int place;
//count the comma
int commacount;

//flow of control
BackPatchList truelist;
BackPatchList falselist;
BackPatchList nextlist;
BackPatchList breaklist;
BackPatchList continuelist;

//the line number of an instruction
int instr;

//0:none 1:not-none
int stmts_type;

//0:not 1:break
int for_break;

//0:not 1:continue
int for_continue;

};
#define YYSTYPE Type
```

B. Global definition

There are 4 kinds of global definition: variable, array, struct and function.

1) Variable & Array definition: **EXTVARS_VAR_ALONE** can get the name and type from **VAR** and insert a variable or an array into Symbol Table respectively.

When a variable or an array is defined, a three address code like ***STA v0 4*** will be generated. This code means a global variable or array with variable number 0 is defined, and it occupies 4 bytes.

2) Struct definition: **STRUCT_DEFINE** will insert a new struct definition into Symbol Table, and then return the attribute members. (**STRUCT_EMPTY** will create a new member link and return it)

Then in **SDEFS** we can pass the link of members and define new members of the struct using **SDEFS_SDECS** and **SDEFS_SEMI**.

In **SEXTVARS** we will define new struct instances.

When a struct instance is defined, a three address code like ***STA v5 20*** will be generated. This code means a global struct instance with variable number 5 is defined, and it occupies 20 bytes.

3) Function definition: **FUNC_ID** will start a new parameter link to link parameters together, and when the production *FUNC → ID FUNC_ID LP M1 PARAS M3 RP* is returned, a three address code like:

FUNCTION label0
PARAM v3
PARAM v4
PARAM v5

will be generated.

PARAS_ID is used to define new parameters.

C. Local definition

In **DEFS** we can difine new local variables, arrays and structs.

This procedure is pretty like **EXTDEF**.

For example, when a local array is defined, a three address code like ***DEC v5 20*** will be generated. This code means a local array instance with variable number 5 is defined, and it occupies 20 bytes.

D. Definition initiation

In **EXTVARS** and **DECS** there are initiations indicated by **INIT**.

In **INIT**, we need to inherit attributes from previous non-terminals, such as links, arraytop, arrayoffset, etc. **INIT_ARGS** is used to pass links when an array is initiated.

By acquiring these attributes, we can store corresponding data into correct memory location.

During the initiation process, three address code will be generated as well.

E. Expressions

When an expression is reduced, corresponding three address code will be generated. In **EXPS**, there will be left value and right values, boolean expression and non-boolean expressions, so we need to generate the IR very carefully.

There are pointer sign '*', which indicates the present temporary is storing an address.

The address aquisition sign '&' will get the address from the temporary.

The immediate number sign '#' indicates an immediate number.

For example, ****t3 := &t5 + #4*** means that get the address of t5, add it and an immediate number 4, store the result to the address that t3 stores.

1) Binary operations: There are binary operations like +, -, % , &, etc.

We need to judge left values and right values, because each of them will generate different three address code.

Take plus operation for example.

```
EXPS: EXPS PLUS EXPS {
$$.place = newtemp();
if ($1.isLeft == 1){
if ($3.isLeft == 1){
sprintf(code[nextinstr++],"t%d :=
*t%d + *t%d",$$.place,$1.place,$3.place);
}
else{
sprintf(code[nextinstr++],"t%d :=
```

```
*t%d + t%d",$$.place,$1.place,$3.place);
}
}
else{
if ($3.isLeft == 1){
sprintf(code[nextinstr++],"t%d :=
t%d + *t%d",$$.place,$1.place,$3.place);
}
else{
sprintf(code[nextinstr++],"t%d :=
t%d + t%d",$$.place,$1.place,$3.place);
}
}
$$.isLeft = 0;

$$.exps_type = 0;
}
```

2) Unary operations: Unary operations are relatively less than binary operations.

Take unary minus for example.

```
EXPS: MINUS EXPS %prec PRODUCT {
$$.place = newtemp();

if ($2.isLeft == 1){
sprintf(code[nextinstr++],"t%d :=
 - *t%d",$$.place,$2.place);
}
else{
sprintf(code[nextinstr++],"t%d :=
 - t%d",$$.place,$2.place);
}
$$.isLeft = 0;
$$.exps_type = 0;
}
```

3) Boolean operations: In boolean expressions, we will need to return truelist, falselist, and nextlist. Their functions and meanings are exactly the same as what is taught in the lecture, so I will go pass the introduction and just give the code here.

Take greater operation for example.

```
EXPS: EXPS GREATERT EXPS{
int next = nextinstr;

$$.truelist = makelist(next);
$$.falselist = makelist(next+1);

if ($1.isLeft == 1){
if ($3.isLeft == 1){
sprintf(code[next],"IF *t%d > *t%d
  GOTO label_",$1.place,$3.place);
}
else{
sprintf(code[next],"IF *t%d > t%d
  GOTO label_",$1.place,$3.place);
}
}
else{
if ($3.isLeft == 1){
sprintf(code[next],"IF t%d > *t%d
  GOTO label_",$1.place,$3.place);
}
else{
sprintf(code[next],"IF t%d > t%d
  GOTO label_",$1.place,$3.place);
}
}
sprintf(code[next+1],"GOTO label_");

nextinstr+=2;
$$.exps_type = 1;
}
```

4) Function calls: When an expression is calling a function, the three address code generated will be like:

ARG t2
ARG *t5
ARG t1
t6 := CALL label3

These code means that we have arguments t2, *t5 and t1, now we are calling a function named label3 with 3 parameters, the return value of the called function will be stored in t6.

```
EXPS: ID EXPS_ARGS LP ARGS RP {
//count the paraNum of ARGS here

int args_count;
if ($4.exp_type == 1)
args_count = $4.commacount+1;
else
args_count = 0;


GlobalLineList = st_lookup($1.name,
      FUNC_RECORD,args_count);

if (GlobalLineList == NULL){
fprintf(stderr,"Undefined function: %s
 [line %d]\n",$1.name.c_str(),linecount+1);
exit(1);
}


if (GlobalLineList->LabNum == main_label){
fprintf(stderr,"Illegal call of main
     function: [line %d]\n",linecount+1);
exit(1);

}

$$.place = newtemp();
```

```
if ($4.exp_type == 1){

args_unit temp = args_link;

while(temp->next != NULL){
temp = temp->next;
}


while(args_count != 0){
if (temp->isLeft == 1){
sprintf(code[nextinstr++],"ARG *t%d",
                  temp->place);
}
else{
  sprintf(code[nextinstr++],"ARG t%d",
                  temp->place);
}
temp = temp->prev;
delete_args_unit();
args_count--;
}
}

sprintf(code[nextinstr++],"t%d :=
   CALL label%d",$$.place,
   GlobalLineList->LabNum);

$$.isLeft = 0;

$$.exps_type = 0;
}
```

5) Variable & Array reference: When an expression is refering to a variable or an array, we use some inherited attributes introduced above to help to get the correct memory location.

The source code of this part is too long, so for details please goto the source file.

6) Struct member reference: This is pretty like array reference. Code is below.

```
EXPS: ID DOT ID {
$$.isLeft = 1;
$$.place = newtemp();

$$.exps_type = 0;

GlobalLineList = st_lookup($1.name,
             STR_BUILD_RECORD,0);

if (GlobalLineList == NULL){
fprintf(stderr,"No such struct: %s
   [line %d]\n",$1.name.c_str(),
   linecount+1);
exit(1);
}

sprintf(code[nextinstr++],"t%d
   := &v%d",$$.place,
   GlobalLineList->VarNum);

StructMember temp =
    GlobalLineList->members;

int index = 0;
while(temp != NULL){
if (temp->name == $3.name){
  index = temp->index;
  break;
}
  temp = temp->next;
}

if (temp == NULL){
  fprintf(stderr,"No such struct
   member: %s [line %d]\n",
   $3.name.c_str(),linecount+1);
  exit(1);
}

if (index > 0){
  sprintf(code[nextinstr++],"t%d :=
  t%d + #%d",$$.place,$$.place,index);
}

}
```

7) Integer aquisition: When we aquire an integer, we need to check its range first, then we can store it in temporary. Code is below.

```
EXPS: INT  {
if ($1.value >= -2147483648
     && $1.value < 0){
  fprintf(stderr,"Range Exceeded:
      [line %d]\n",linecount+1);
exit(1);
}
else{

  $$.isLeft = 0;

  $$.exps_type = 0;
  $$.place = newtemp();
  sprintf(code[nextinstr++],"t%d :=
        #%d",$$.place,$1.value);
}
}
```

F. Statements

STMT will only be in a function body.

And a special point of this non-terminal is that it can recuisively reduce from **STMTBLOCK**. So when it is reduced, synthesized attributes like nextlist should be handled carefully and properly through **STMT**, **STMTS** and **STMTBLOCK**. For these details please see the source file.

1) If else statememt & If statement: Here I followed the introductions on the course PPT, and added some new feature in the production body.

EXPS_N2 will check the previous **EXPS** whether it has a truelist(falselist), if not **EXPS_N2** will produce one for it.

STMT_N has similar function.

The function of **STMT_M** is to create a new label and backpatch when the whole production reduce.

```
STMT: IF LP EXPS EXPS_N2 RP STMT_M STMT1
    STMT_N ELSE STMT_M STMT STMT_N {

if ($3.exps_type == 1){
  backpatch($3.truelist,$6.instr);
  backpatch($3.falselist,$10.instr);
}
else{
  backpatch($4.truelist,$6.instr);
  backpatch($4.falselist,$10.instr);

}

BackPatchList temp = merge($7.nextlist,
                     $8.nextlist);
temp = merge(temp,$11.nextlist);
$$.nextlist = merge(temp,$12.nextlist);

$$.breaklist = merge($7.breaklist,
                 $11.breaklist);
$$.continuelist = merge($7.continuelist,
                     $11.continuelist);

$$.has_return = $7.has_return |
               $11.has_return;

}

EXPS: IF LP EXPS EXPS_N2 RP STMT_M
                 STMT STMT_N {

BackPatchList temp = merge($7.nextlist,
                      $8.nextlist);


if ($3.exps_type == 1){
  backpatch($3.truelist,$6.instr);
  $$.nextlist = merge($3.falselist,temp);
}
else{
  backpatch($4.truelist,$6.instr);
  $$.nextlist = merge($4.falselist,temp);
}


$$.breaklist = $7.breaklist;
$$.continuelist = $7.continuelist;

$$.has_return = $7.has_return;

}

EXPS_N2:{

if ($0.exps_type == 0){
int next = nextinstr;
$$.truelist = makelist(next+1);
$$.falselist = makelist(next);

if ($0.isLeft == 1){
  sprintf(code[next],"IF *t%d == #0
           GOTO label_",$0.place);
}
else{
  sprintf(code[next],"IF t%d == #0
          GOTO label_",$0.place);
}
sprintf(code[next+1],"GOTO label_");

nextinstr+=2;
}

}
;
```

2) Experssion statement: We will only need to set relevant attributes and return here.

```
STMT: EXPS SEMI {

$$.nextlist = NULL;
$$.breaklist = NULL;
$$.continuelist = NULL;

$$.has_return = 0;

}
```

3) Statement block: Similar to expression statement.

```
STMT: STMTBLOCK {

$$.nextlist = $1.nextlist;

$$.breaklist = $1.breaklist;
$$.continuelist = $1.continuelist;

$$.has_return = $1.has_return;
}
```

4) Return statememt: Clean the lists and set has_return to 1.

```
STMT: RETURN EXPS SEMI {

if ($2.isLeft == 1){
  sprintf(code[nextinstr++],
     "RETURN *t%d",$2.place);
}
else{
  sprintf(code[nextinstr++],
      "RETURN t%d",$2.place);
}

$$.nextlist = NULL;

$$.breaklist = NULL;
$$.continuelist = NULL;

$$.has_return = 1;

}
```

5) For loop statement: Building the for loop semantic actions is very much like if else statement, except that we need to include **BREAK** and **CONT** here.

I linked relevant list to proper location and backpatched it.

STMT_N_1 is used to create new list for **EXP** under certain circumstances.

```
STMT: FOR LP EXP SEMI STMT_M EXP
    EXPS_N2 STMT_N_1 SEMI STMT_M
    EXP STMT_N_1 RP STMT_M STMT1 STMT_N{

BackPatchList temp = NULL;

if ($6.exps_type == 0){
temp = merge($7.truelist,$8.nextlist);
}
else{
temp = merge($6.truelist,$8.nextlist);
}

backpatch(temp,$14.instr);

temp = merge($15.nextlist,$16.nextlist);
backpatch(temp,$10.instr);

if ($15.breaklist != NULL){
if ($6.exps_type == 0){
  $$.nextlist = merge($7.falselist,
                      $15.breaklist);
}
else{
  $$.nextlist = merge($6.falselist,
                    $15.breaklist);
}
}
else {
if ($6.exps_type == 0){
  $$.nextlist = $7.falselist;
}
else{
  $$.nextlist = $6.falselist;
}

}

if ($15.continuelist != NULL){
  temp = merge($15.continuelist,$12.nextlist);
  backpatch(temp,$5.instr);
}
else{
  backpatch($12.nextlist,$5.instr);
}

$$.breaklist = NULL;
$$.continuelist = NULL;

$$.has_return = $15.has_return;

}

STMT_N_1 :{

if ($0.truelist == NULL &&
     $0.falselist == NULL){
  $$.nextlist = makelist(nextinstr);
  sprintf(code[nextinstr++],"GOTO label_");
}
else
  $$.nextlist = NULL;
}
;
```

6) Continue & Break statement: In these two statements, the actions are similar.

```
STMT: CONT SEMI {

$$.continuelist = makelist(nextinstr);
$$.breaklist = NULL;

$$.nextlist = NULL;

$$.has_return = 0;

sprintf(code[nextinstr++],
       "GOTO label_");

}
STMT: BREAK SEMI {
```

```
$$.breaklist = makelist(nextinstr);
$$.continuelist = NULL;

$$.nextlist = NULL;

$$.has_return = 0;

sprintf(code[nextinstr++],"GOTO label_");

}
```

7) Write & Read statement: In these two statements, the actions are similar too, we will produce three address code like ***READ t3*** and ***WRITE *t5***.

```
STMT: WRITE LP EXPS RP SEMI {
if ($3.isLeft == 1){
sprintf(code[nextinstr++],
    "WRITE *t%d",$3.place);
}
else{
sprintf(code[nextinstr++],
     "WRITE t%d",$3.place);
}

$$.nextlist = NULL;
$$.breaklist = NULL;
$$.continuelist = NULL;

$$.has_return = 0;
}

STMT: READ LP EXPS RP SEMI {
if ($3.isLeft == 1){
sprintf(code[nextinstr++],
     "READ *t%d",$3.place);
}
else{
fprintf(stderr,"Wrong usage of
  read(): [Line %d]\n",linecount+1);
exit(-1);
}

$$.nextlist = NULL;
$$.breaklist = NULL;
$$.continuelist = NULL;

$$.has_return = 0;
}
```

VII. Code Optimization

This is an optional part. I deleted some redundant three address code like ***GOTO label3*** following ***RETURN *t7***.

I also deleted some redundant three address code like ***GOTO label4*** following ***GOTO label1***.

And I added a quit instruction at the end of the three address code:

```
fprintf(MIPSCode," li $v0,10\n");
fprintf(MIPSCode," syscall\n");
```

VIII. Machine-Code Generatoin

This is the final step in our project.

From previous steps we have already get the intermediate code, now we are going to generate the MIPS code. In this process we will need use Symbol Table too.

The generated intermediate code is linear, which means that the instructions are in the same order of the source file. But this will lead to a problem that if we want to initiate some global variable where can we put the initiation MIPS code? My idea is to move them to the beginning of the main function.

This contains two steps: find the delcaration of global variables(arrays, structs), and move those codes to the main function body.

In addition, We need to consider the allocation of registers. There are several ways to dynamically allocate registers during the translation, I chose the simplist one: to store every global variable and local variable into static data segmentation.

When a data is needed, we will read data from the static data segmentation, also, when a data is changed, we store it back. So in my MIPS code there are plenty of lw and sw instructions.

A. Static data delaration & Global initiation replace

This is the first step in MIPS code generation.

The translator will go throungh the entire intermediate code array, if a ***STA*** is found, such a MIPS code may be generated: ***v3 .word 0***, which means there is an address v3 indicating one byte in the static data segmentation, and the value of that byte is 0.

Then the translator will go through the intermediate code and link every instruction it meets until it meets ***FUNCTION*** or another ***STA***. The linked instructions will be inserted into main function later.

By the time the translator is seeking for ***DEC*** and ***PARAM***, every time the translator finds one, it will generate MIPS code to store the variable to the static data segmentation. PS: ***PARAM*** will be changed into ***XX***.

Then we can perform the replacement of the global initiations,

B. Code translation

Now we need only translate the intermediate code into MIPS code one instruction by one instruction. The method is relatively easy here.

1) FUNCTION instruction: When translator encounters ***FUNCTION***, actions are below.

strtok(char *, char *) is a function to seperate strings into substrings.

atoi(char *) is a function to translate char to int.

maintemp is a string whose content is the label of main function.

```
temp = strtok(code[now]," ");
temp = strtok(NULL," ");
```

```
char *label = temp + 5;

func_label = atoi(label);
stack_counter = 0;

if (!strcmp(temp,maintemp))
fprintf(MIPSCode,"main:\n");
else
fprintf(MIPSCode,"%s:\n",temp);
```

2) READ instruction: In READ instruction, there will only be left values.

```
temp = strtok(NULL," ");

//only '*tx'

fprintf(MIPSCode," li $v0,5\n");
fprintf(MIPSCode," syscall\n");

fprintf(MIPSCode," la $t0,%s\n",temp+1);
fprintf(MIPSCode," lw $t1,0($t0)\n");

fprintf(MIPSCode," sw $v0,0($t1)\n");
```

3) Write instruction: In WRITE instruction, we need to decide whether the temporary is left value or not.

```
temp = strtok(code[now]," ");
temp = strtok(NULL," ");

switch (temp[0]){
case '*':{
fprintf(MIPSCode," la $t0,%s\n",temp+1);
fprintf(MIPSCode," lw $t1,0($t0)\n");
fprintf(MIPSCode," lw $a0,0($t1)\n");

break;}
default:{
fprintf(MIPSCode," la $t0,%s\n",temp);
fprintf(MIPSCode," lw $a0,0($t0)\n");
```

4) IF GOTO instruction: In this instruction, the translator will first decide which operator is between two operands.

Then generate corresponding MIPS code depending on whether it is left value or not.

Here is a example when operator is == , 1st operand is right value, 2nd operand is left value:

```
if(op1[0] != '*' && op2[0] == '*'){
fprintf(MIPSCode," la $t0,%s\n",op1);
fprintf(MIPSCode," lw $t1,0($t0)\n");

fprintf(MIPSCode," la $t3,%s\n",op2+1);
fprintf(MIPSCode," lw $t4,0($t3)\n");
fprintf(MIPSCode," lw $t5,0($t4)\n");

fprintf(MIPSCode," beq $t1,$t5,%s\n",label);
}
```

5) GOTO instruction:

```
temp = strtok(code[now]," ");
temp = strtok(NULL," ");

fprintf(MIPSCode," j %s\n",temp);
```

6) LABEL instruction:

```
temp = strtok(code[now]," ");
temp = strtok(NULL," ");

fprintf(MIPSCode,"%s:\n",temp);
```

7) CALL instruction: When translator encounters a CALL instruction, it will store all the local variables into stack using function stack_in(), then store the return address in stack, next call ***jal*** instruction.

When the callee returns, the return address will be loaded, as well as all the local vriables.(use stack_out())

```
if(!strcmp(tokenlist[2],"CALL")){
// tx = CALL labelx
if(ra_stored == 0){

stack_counter = stack_in(func_label);

fprintf(MIPSCode," addi $sp,$sp,-4\n");
fprintf(MIPSCode," sw $ra,0($sp)\n");
}

fprintf(MIPSCode," jal %s\n",
                    tokenlist[3]);

fprintf(MIPSCode," lw $ra,0($sp)\n");
fprintf(MIPSCode," addi $sp,$sp,4\n");

stack_out(stack_counter,func_label);

fprintf(MIPSCode," la $t0,%s\n",
                     tokenlist[0]);
fprintf(MIPSCode," sw $v0,0($t0)\n");

ra_stored = 0;
}
```

8) XX instruction: XX is used to load datas into parameters from the stack.

```
temp = strtok(code[now]," ");
temp = strtok(NULL," ");

fprintf(MIPSCode," la $t0 %s\n",temp);
fprintf(MIPSCode," lw $t1,0($sp)\n");
fprintf(MIPSCode," sw $t1,0($t0)\n");

fprintf(MIPSCode," addi $sp,$sp,4\n");
```

9) ARG instruction: When the first ARG is encountered, translator need to store all the local variables into stack using stack_in() function.

Also the data might be a left value, translator should judge which kind of code to generate.

```
temp = strtok(code[now]," ");
temp = strtok(NULL," ");

if (ra_stored == 0){

stack_counter = stack_in(func_label);

fprintf(MIPSCode," addi $sp,$sp,-4\n");
fprintf(MIPSCode," sw $ra,0($sp)\n");

ra_stored = 1;
}

fprintf(MIPSCode," addi $sp,$sp,-4\n");

if (temp[0] == '*'){
fprintf(MIPSCode," la $t0 %s\n",temp+1);
fprintf(MIPSCode," lw $t1,0($t0)\n");
fprintf(MIPSCode," lw $t2,0($t1)\n");
fprintf(MIPSCode," sw $t2,0($sp)\n");
}
else{
fprintf(MIPSCode," la $t0 %s\n",temp);
fprintf(MIPSCode," lw $t2,0($t0)\n");
fprintf(MIPSCode," sw $t2,0($sp)\n");
}
```

10) RETURN instruction: Firstly the translator will decide whether the returned data is a left value, then it will generate ***jr $ra*** to jump to the return address.

```
temp = strtok(NULL," ");

switch (temp[0]){
case '*':{
fprintf(MIPSCode," la $t0,%s\n",temp+1);
fprintf(MIPSCode," lw $t1,0($t0)\n");

fprintf(MIPSCode," lw $v0,0($t1)\n");
fprintf(MIPSCode," jr $ra\n");

break;}
default:{
fprintf(MIPSCode," la $t0,%s\n",temp);
fprintf(MIPSCode," lw $v0,0($t0)\n");
fprintf(MIPSCode," jr $ra\n");
break;}

}
```

11) Unary & Binary operation & Assign instruction: In this step the translator will check whether operands are left value or not, then generate corresponding MIPS code. For details please see the source file.

IX. Compiler Evaluation

When the MIPS file is finally generated, I can't wait to test it using SPIM simulator.

I tested it on the source code given by TA, and some other programs as well, they all performed well!

But when I tested it on eight queen puzzle, I found that with the input number increasing, the time system needed to get the result can be dramatically longer.

However, I am still pretty satisfied to see the completion of the project.

References

[1] Anthony A. Aaby, *Compiler Construction using Flex and Bison*, 2004

[2] Pablo Nogueira Iglesias, *Notes about lex and yacc*, 1999.

[3] Tsan-sheng Hsu, *Intermediate Code Generation*.

[4] Jia Chen, Chang Xu *Compiler Principle Experiment 3: Intermediate Code Generatoin*.

[5] Jia Chen, Chang Xu *Compiler Principle Experiment 4: Machine Code Generatoin*.