

CS 215: COMPILER PROJECTS

Professor: Li Jiang

SMALLC COMPILER REPORT

Zhengtian Xu 5140309178

January 16, 2017

Contents

1	Introduction	3
1.1	Components of SmallC Compiler	3
1.2	Environment	3
1.3	File Decription	3
2	Abstract Syntax Tree	4
3	Lexical Analyzer	5
3.1	Read and Write	5
3.2	White Space	5
3.3	Newline	5
3.4	Identifier	5
3.5	Integer	5
4	Syntax Analyzer	6
4.1	Modification of Grammar	6
4.2	Precedence of IF and IF ELSE Statement	7
4.3	Error Message	7
4.4	Precedence of Operators	7
5	Semantic Analysis	8
5.1	Error Message	8
5.2	Symbol Table	8
5.3	Semantic Checking	9
5.3.1	Variables and functions have to declared before usage	9
5.3.2	Variables and functions should not be re-declared	9
5.3.3	Reserved words can not be used as identiers	10
5.3.4	Program must contain a function int main() to be the entrance	10
5.3.5	The number and type of variable(s) passed should match the denition of the function	10
5.3.6	Use [] operator to a non-array variable is not allowed	10
5.3.7	The . operator can only be used to a struct variable	10
5.3.8	break and continue can only be used in a loop	10
5.3.9	Right-value can not be assigned by any value to expression	10
5.3.10	The condition of if statement should be an expression with int type .	10
5.3.11	The condition of for should be an expression with int type or ϵ	11
5.3.12	Only expression with type int can be involved in arithmetic	11
5.3.13	Make sure the index of array can not be negative	11
6	Intermediate Representation	11
6.1	Quadruple	11
6.2	Design of Intermediate Code	12
6.3	Implementation	12
6.3.1	Expressions	12
6.3.2	Statements	12

7	Optimization	13
7.1	Dead Code Elimination	13
8	Machine-code Generation	14
8.1	Instruction Selection	14
8.2	Register Allocation	14
8.3	Read and Write	14
9	Extension	15
9.1	Timer	15
10	Conclusion	15

1 Introduction

SMALLC is a simplified C-like programming language containing only the core part of C language. This compiler can translate **SMALLC** source codes to **MIPS** assembly codes. These assembly codes can run on the **SPIM** simulator.

1.1 Components of SmallC Compiler

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analysis
- Intermediate Representation
- Optimization
- Machine-code Generation

1.2 Environment

This project is based on Ubuntu 16.04 LTS.

1.3 File Decription

File	Decription
header.h	A header which includes some necessary files in the <i>include</i> folder.
treeNode.h	The definition of abstract syntax tree in the <i>include</i> folder.
symbolTable.h	The definition of symbol table in the <i>include</i> folder.
quadruple.h	The definition of three-address code..
smallc.l	Lex program.
smallc.y	Yacc program.
semanticAnalysis.h	The semantic analyzer for the SmallC Compiler.
intermediateCode.h	Translate the quadruple into interCode.
optimization.h	Some optimizations for the compiler.
codeGeneration.h	Translate the interCode into MIPS code.
makefile	Makefile to compile the program.
InterCode	After make, this file stores the generated IR code.
MIPSCode.s	After make, this file stores the generated MIPS code.
scc	After make, this is a executable file.
README	Description of the files in the submission.
input.s	MIPS code for read and write.
5140309178-report.pdf	Report for SmallC Compiler.

Table 1: SmallC Compiler Files and Decription

2 Abstract Syntax Tree

In this project, I define the abstract syntax tree in *treeNode.h*. It is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. For the code following, Figure 1 shows the abstract syntax tree of the code.

```
1  int main(){
2      int a = 1;
3      return a;
4  }
```

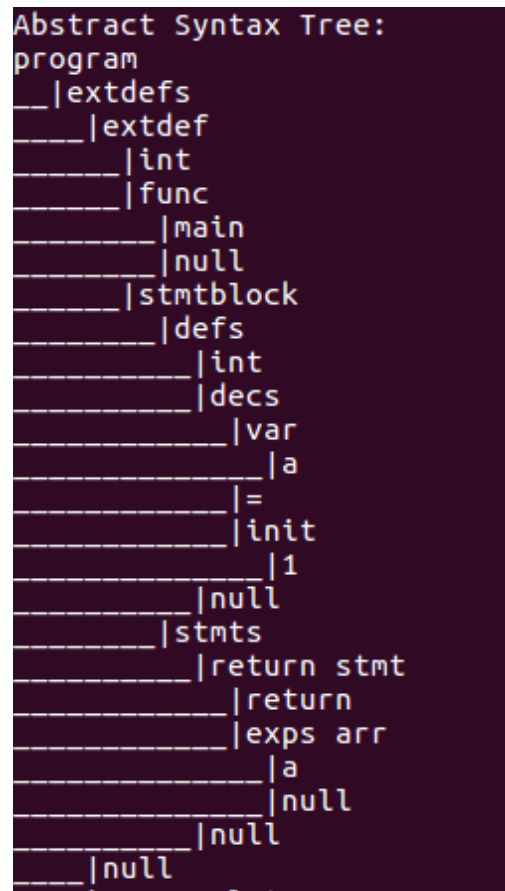


Figure 1: Abstract Syntax Tree for the Code

The way I define the abstract syntax tree is in the following code block.

```
1  typedef struct TreeNode{
2      char *data;
3      char *type;
4      int line, childNum;
5      struct TreeNode** children;
6  } TreeNode;
```

Tree node is defined as a struct, which contains its data, its type and line number. Since it is a tree struct, we should record its number of children and the link to its children. This

part will be used in syntax analysis and later operation.

3 Lexical Analyzer

A lexical analyser has been implemented in this part. The lexical analyser uses the tool which name is **Flex** to read the source codes and separates them into tokens. This part is implemented in *smallc.l*.

In the project decription, most tokens and operators are given. Thus, I will show some special tokens which are not given.

3.1 Read and Write

To solve the **Read** and **Write** token, it is defined as follow.

```
1  "read"                {yyval.string = strdup(yytext); return READ;}
2  "write"               {yyval.string = strdup(yytext); return WRITE;}
```

3.2 White Space

Write space should be ignored when in the process of lexical analyzer.

```
1  [ \t\v\f]+           {/*Need no actions*/}
```

3.3 Newline

When meeting a new line symbol, it will record the line number and add.

```
1  [\n]                 {yylineno = yylineno + 1;}
```

3.4 Identifier

For Identifier, some rules are constrained such as the first character in an identifier must be an alphabet or an underscore and can be followed only by any number alphabets, or digits or underscores. What's more, Commas or blank spaces are not allowed within an identifier.

```
1  [a-zA-Z_][a-zA-Z0-9_]* {yyval.string = strdup(yytext); return ID;}
```

3.5 Integer

Integer must involve values in different systems.

```
1  ([0-9]*|0[xX][0-9a-fA-F]+) {yyval.string = strdup(yytext); return INT;}
```

4 Syntax Analyzer

A syntax analyser has been implemented in this part. The lexical analyser uses the tool which name is **Yacc** to describe the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. This part is implemented in *smallc.y*.

4.1 Modification of Grammar

To avoid conflicts in the semantic analysis, I modify some semantic rules to prevent some empty expression in initialization and if statement.

The followings are several productions provided in the project description:

```

STMT      → EXP SEMI
           | STMTBLOCK
           | RETURN EXP SEMI
           | IF LP EXP RP STMT
           | IF LP EXP RP STMT ELSE STMT
           | FOR LP EXP SEMI EXP SEMI EXP RP STMT
           | CONT SEMI
           | BREAK SEMI

INIT       → EXP
           | LC ARGS RC
```

I change some **EXP** to **EXPS** to avoid some empty tokens in statement, such as *int a =* is forbidden, then I add two extra production for **READ** and **WRITE** tokens. The following is my productions modified from the above.

```

STMT      → EXP SEMI
           | STMTBLOCK
           | RETURN EXPS SEMI
           | IF LP EXPS RP STMT
           | IF LP EXPS RP STMT ELSE STMT
           | FOR LP EXP SEMI EXPS SEMI EXP RP STMT
           | CONT SEMI
           | BREAK SEMI
           | READ LP EXPS RP SEMI
           | WRITE LP EXPS RP SEMI

INIT       → EXPS
           | LC ARGS RC
```

4.2 Precedence of IF and IF ELSE Statement

The grammar given above may induce one or two reduce-reduce/shift-reduce conflicts, so I assign precedence of some expressions manually to eliminate these conflicts.

The following shows the original IF and IF ELSE rules:

STMT \rightarrow IF LP EXPS RP STMT

STMT \rightarrow IF LP EXPS RP STMT ELSE STMT

Obviously, the second production has a higher precedence than the first one. `%prec` is used to show the precedence. `%prec` appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal. It causes the precedence of the grammar rule to become that of the following token.

```
1  %nonassoc IFX
2  %nonassoc ELSE
```

And the production become:

STMT \rightarrow IF LP EXPS RP STMT `%prec` IFX

STMT \rightarrow IF LP EXPS RP STMT ELSE STMT `%prec` ELSE

4.3 Error Message

The error message is done together with generating the parse tree. Once an error occurs during the procedure, the parse process will shutdown and report the just-found mistake:

```
1  void yyerror(const char *msg)
2  {
3      fflush(stdout);
4      fprintf(stderr, "Error: %s at line %d\n", msg,yylineno);
5      fprintf(stderr, "Parser does not expect '%s'\n",yytext);
6  }
```

4.4 Precedence of Operators

According the project description, I set the precedence to every operator. Left and right also indicates whether this operator is a left value or a right value.

```
1  %right SHLASSIGN SHRASSIGN ORASSIGN XORASSIGN ANDASSIGN DIVASSIGN MULASSIGN
    MINUSASSIGN PULSASSIGN ASSIGN
2  %left LOGICALOR
3  %left LOGICALAND
4  %left BITOR
```



```

5  %left BITXOR
6  %left BITAND
7  %left NEQ EQ
8  %left GT GE LT LE
9  %left SHL SHR
10 %left PLUS MINUS
11 %left MUL DIV MOD
12 %right LOGICALNOT PREINC PREDEC BITNOT UMINUS
13 %left LP RP LB RB DOT

```

5 Semantic Analysis

A semantic analyser has been implemented in this part. In this part, a symbol table is defined to store the information of variable, struct and function. This part is implemented in *semanticAnalysis.h* and *symbolTable.h*.

5.1 Error Message

In this part, I also has a *error* function to report the line number and error message in semantic checking. If some errors happen, it will call this function and *exit(1)*.

```

1  void error(int line, const char* s1, const char* s2){
2      fprintf(stderr, "At line %d: ", line);
3      if (s1 == NULL) fprintf(stderr, "%s\n", s2);
4      else if (s2 == NULL) fprintf(stderr, "%s\n", s1);
5      else fprintf(stderr, "%s %s\n", s1, s2);
6      exit(1);
7  }

```

5.2 Symbol Table

Symbol table is the data structure to hold information about the variables of the source programs. The symbol table records the type declarators, variables and the information of parameters and return type of functions. The following code is the symbol table of this SmallC Compiler, which I use *map* as the technique for mapping because its efficiency. Redefining comparson is necessary for *char** in *map*.

```

1  struct ptr_cmp {
2      bool operator()(const char* s1, const char* s2) const {
3          return strcmp(s1,s2) < 0;
4      }
5  };
6  struct SymbolTable {
7      map <const char*, const char*,ptr_cmp> table;
8      map <char*, vector<char *>, ptr_cmp > struct_object;
9      map <char*, vector<char *>, ptr_cmp > struct_class;
10     map <char*, int, ptr_cmp > struct_variable_num;

```

```

11     map <char*, int, ptr_cmp> variable_num;
12     map <char*, vector<int>, ptr_cmp> array;
13     int parent_index;
14 } symbolTable[MAX_NUM][MAX_NUM];

```

- **map <const char*, const char*, ptr_cmp> table;**
This mapping table maps the name of variable to its type and the type is either int or struct.
- **map <char*, vector<char * >, ptr_cmp > struct_object;**
This mapping table maps the struct object name to a vector which contains the name of variables defined in its namespace.
- **map <char*, vector<char * >, ptr_cmp > struct_class;**
This mapping table maps the struct class name to a vector which contains the name of variables defined in its namespace.
- **map <char*, int, ptr_cmp > struct_variable_num;**
This mapping table maps the struct class name to the number of variables defined in its namespace.
- **map <char*, int, ptr_cmp> variable_num;**
This mapping table maps the name of variable to its number.
- **map <char*, vector<int>, ptr_cmp> array;**
This mapping table maps the name of array to its size.
- **int parent_index;**
It records the location of the upper namespace.

5.3 Semantic Checking

5.3.1 Variables and functions have to declared before usage

For variables, I have defined a function which name is *_id()* to check whether the name of variables are in symbol table. The function will go through its own namespace, if it fails to find, it will find its parent namespace and repeat. Or it will call the function *error()*.

For functions, a function table is defined to store the information of function. What's more, **overloaded function** should be taken into consideration. A table for overloaded function is used in this part. For overloaded function, I change the name of overloaded function to make sure its normal execution. Thus, we will check the number of arguments and make the calling functions' name consistent with the previously modified name.

5.3.2 Variables and functions should not be re-declared

Like the operation above, I use function which name is *_id()* to check whether the name of variables are in symbol table. For function name, a function map also map the function. So we just go through the function table and find whether the function is re-declared.

5.3.3 Reserved words can not be used as identifiers

In this part, I defined a function *checkReservedWord()* to detect whether the name of identifiers is a reserved word. If it is a reserved word, the compiler will call *error()* function and terminate.

5.3.4 Program must contain a function `int main()` to be the entrance

In this part, if the process of checking meets the name of function is **main**, the bool flag *have_main* becomes true. If at the end the flag is false, just call the error function.

5.3.5 The number and type of variable(s) passed should match the denition of the function

In this part, *func_cnt_table* is used to record the number of parameters of a function. It's worth mentioning that there may be cases that the parameters of a function are also function calls, so whenever there is a function call, we have to store the number of parameters temporarily and restores afterwards.

5.3.6 Use `[]` operator to a non-array variable is not allowed

When the syntax analyzer meets an ID followed by `[]`, it will regard it as an array type expression and pass it to semantic analyzer. The map table *variable_num* is defined to check the bytes the variable occupy. So when we meet the `[]` operator, we check whether the variable is a struct, if not, we get the bytes the variable occupy, if the bytes is one \times four, then it's not an array, error.

5.3.7 The `.` operator can only be used to a struct variable

First we check whether the ID is a struct variable by going through the *struct_object* table, then we check whether the variable followed by `.` operator is a variable defined in the struct object.

5.3.8 `break` and `continue` can only be used in a loop

When we meet `break` and `continue`, we will check the bool flag *for_loop*, if the flag value is false, we call the *error()* function.

5.3.9 Right-value can not be assigned by any value to expression

Define a function which name is *left_value*. It can check whether the EXPS is a left value. When setting the type of the node, check whether it is a left value at the same time. When doing assignment to a node, check whether it is a left value. Exspecially, **read()** is a left value.

5.3.10 The condition of if statement should be an expression with int type

We will check whether the EXPS is a struct type or a int type, go throuth the symbol table to check the type of the EXPS. If no, then condition of if statement isn't an int expression. Report an error.

5.3.11 The condition of for should be an expression with int type or ϵ

We will check whether the EXPS is a struct type or a int type, go through the symbol table to check the type of the EXPS. For ϵ , the semantic analyzer will check its condition expression as a part of the semantic check. If no, then condition of if statement isn't an int expression. Then we will call the *error()* function.

5.3.12 Only expression with type int can be involved in arithmetic

We will check whether the EXPS is a int type, go through the symbol table to check the type of the EXPS. If no, then condition of if statement isn't an int expression. Report an error.

5.3.13 Make sure the index of array can not be negative

In *_var()* function, I make sure that the index of array can not be negative.

6 Intermediate Representation

Intermediate Representation is independent of the details of both source language and target language. I used quadruple as the intermediate representation in this SmallC Compiler. These functions are defined in *quadruple.h*. This will produce the intermediate code, and this part is in *intermediateCode.h*.

6.1 Quadruple

The following codes is the definition of quadruple.

```
1  struct Address{
2      RegType type;
3      string name;
4      int value;
5      int real;
6      int needload;
7      int needclear;
8  };
9
10 struct Quadruple{
11     string op;
12     int active;
13     int flag;
14     Address parameters[3];
15 };
```

- **Quadruple** *op* means the name of operation, *active* means whether it's active, *flag* means whether its arguments should be revised and the *Adress parameters* means the parameters of quadruple.

- **Address** *type* means the type of the parameter, *name* is used for ‘label’ or ‘call’ process, *value* indicates the value of register, *real* indicates the read register in MIPS, *needload* and *needclear* are flags.

6.2 Design of Intermediate Code

IR	Decription	IR	Decription
or a b c	$a = b \mid c$	xor a b c	$a = b \mid c$
sll a b c	$a = b \ll c$	srl a b c	$a = b \gg c$
and a b c	$a = b \& c$	add a b c	$a = b + c$
sub a b c	$a = b - c$	mul a b c	$a = b * c$
div a b c	$a = b / c$	rem a b c	$a = b \% c$
neg a b	$a = b$	lnot a b	$a = !b$
not a b	$a = b$	beqz a b	if $a == 0$ goto label b
bnez a b	if $a != 0$ goto label b	bgez a b	if $a \geq 0$ goto label b
bgtz a b	if $a > 0$ goto label b	blez a b	if $a \leq 0$ goto label b
bltz a b	if $a < 0$ goto label b	li a b	$a = b$
lw a b c	$a = *(b+c)$	sw a b c	$*(b+c) = a$
move a b	$a = b$	label a	a is an integer
goto a	a is an integer	func s	s is a string
call s	s is a string	ret	function return

Table 2: IR and description

6.3 Implementation

6.3.1 Expressions

A function whose name is *interCode_exps()* is created to translate expressions.

- **Binary Operator** For binary operator, we first create the register for two expressions and it will get the value and store into these register, then we check which operator is used and execute.
- **Unary Operator** For unary operator, we first use a register to store the original value, then we check which operator is used and execute and store the new value into the original register.
- **Assignment Expressions** For assignment expressions, a function is designed to catch the value and store it. Some operator like ‘+ =’ is just the combination of two operators.

6.3.2 Statements

- **If** If statement is straightforward, if the expression in if statement is true, it will enter the true state of if, then skip the else part. Or it will just skip the true part and execute

else part. That is we should create two labels for thses two parts, and a *goto* label for this part. After translate the code of the non-terminals, set the labels.

```

1   Quadruple t1 = create_quadruple_label();
2   Quadruple t2 = create_quadruple_label();
3   Quadruple t3 = create_quadruple_goto(get_label_number(t2));
4   interCode_assignment_1(t->children[0], 0, get_label_number(t1));
5   interCode_stmt(t->children[1]);
6   IR_push_back(t3);
7   IR_push_back(t1);
8   if (t->childNum == 3) {
9       interCode_stmt(t->children[2]);
10  }
11  IR_push_back(t2);

```

- **For** For statement is always a loop part, we should check the condition. If false, just go to the label defined in this part. Or we enter the loop. In the loop, we make sure that if we meet *break* and *continue*, we create a label for this.
- **Continue and Break** Create a *goto* label.
- **Return** Return statement should first create a new register to store the value of statement. Then create return quadruple.

7 Optimization

This part is in the *optimization.h*.

7.1 Dead Code Elimination

Some case link repetitive assignment should be eliminated if the register is not used in the future. For thses rundundant instruction, we should not allocate the registers for them. We check the value of the instructions and reassign it to the register. Then release the register. The following is code for the check for rundundant instructions. This can divided into two parts, one is the assignment dead code elimination and the other is the assignment after read value. Both these two parts are using the folloeing codes to check the rundundant instruction.

```

1   if (y.parameters[1].type == _TMP && y.parameters[1].value == value) {
2       y.parameters[1].value = x.parameters[1].value;
3   }
4   if (y.parameters[2].type == _TMP && y.parameters[2].value == value) {
5       y.parameters[2].value = x.parameters[1].value;
6   }
7   if (y.parameters[0].type == _TMP && y.parameters[0].value == value) {
8       active_flag = 0;
9       break;
10  }

```

8 Machine-code Generation

8.1 Instruction Selection

For instruction selection, I do the following parts.

- When faced with a node, push the corresponding instructions into the vector according to its type.
- For the constant zero, use the zero register instead.
- Translate the constant in the arithmetic expression into the intermediate code.

8.2 Register Allocation

Linear scan algorithm is used for register allocation, this part is in *codeGeneration.h*. This algorithm is not based on graph coloring, but allocates registers to variables in a single linear-time scan of the variables' live ranges. This algorithm results in code that is almost as efficient as that obtained using more complex and time-consuming register allocators based on graph coloring.

8.3 Read and Write

For Read operation. First create a new register to store the value of read, if the read operator is not in main function, then we should store and restore the \$ra register. The following is about the code for read.

```
1      int reg = new_register();
2      interCode_exps(t->children[0], reg);
3      assert(get_register_state(reg) == REGISTER_STATE_ADDRESS);
4      int v0 = -3;
5      if (!main_flag) {
6          IR_push_back(create_quadruple_sw_offset(retad_pointer, stack_pointer,
7              -2000));
8      }
9      IR_push_back(create_quadruple_call("read"));
10     if (!main_flag) {
11         IR_push_back(create_quadruple_lw_offset(retad_pointer, stack_pointer,
12             -2000));
13     }
14     IR_push_back(create_quadruple_sw(v0, reg));
```

For write function, almost the same as the read function.

```
1      int reg = new_register();
2      interCode_exps(t->children[0], reg);
3      catch_value_self(reg);
4      IR_push_back(create_quadruple_move(-2, reg));
5      if (!main_flag) {
6          IR_push_back(create_quadruple_sw_offset(retad_pointer, stack_pointer,
7              -2000));
```

```

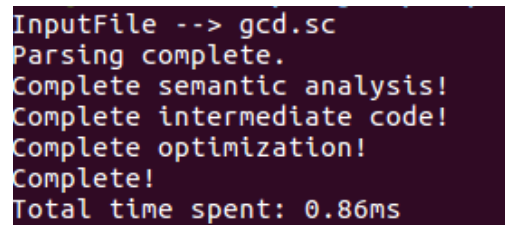
7      }
8      IR_push_back(create_quadruple_call("write"));
9      if (!main_flag) {
10         IR_push_back(create_quadruple_lw_offset(retad_pointer, stack_pointer,
11            -2000));
12     }

```

9 Extension

9.1 Timer

In this compiler, I add a timer to calculate the total time of the process. Figure 2 shows the total time of the process.



```

InputFile --> gcd.sc
Parsing complete.
Complete semantic analysis!
Complete intermediate code!
Complete optimization!
Complete!
Total time spent: 0.86ms

```

Figure 2: Timer

10 Conclusion

This is a really challenging project because it needs a lot of coding and time to debug. I think my coding ability and engineering skills surely improved. Thanks for TA's project description because it helps me think seriously how to implement a compiler. In this project, I also learnt a lot through searching online for some knowledge about compiler and some novel algorithms. Finally, thanks to TA and my classmates for helping me finish this hard project.

References

- [1] Shanghai Jiao Tong University, Lex introduction, <http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/LexIntroduction.pdf>
- [2] Shanghai Jiao Tong University, Project description, <http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/project.pdf>
- [3] Shanghai Jiao Tong University, Yacc introduction, <http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/YaccIntroduction.pdf>
- [4] Shanghai Jiao Tong University, Set up Environment, <http://www.cs.sjtu.edu.cn/~jiangli/teaching/CS308/projects/SetupEnvironment.pdf>
- [5] The LEX & YACC Page, dinosaur.compilertools.net