

Compiler Principles Project Report

5140309258

Tanyi Chen

Contents

1.	Introduction	3
2.	Execution.....	3
2.1.	Environment	3
2.2.	Instruction	3
3.	Lexical Analyzer	3
3.1.	Tokens	4
3.2.	Implementation.....	5
4.	Syntax Analyzer.....	5
4.1.	Grammar	5
4.2.	Grammar Refinements	7
4.2.1.	Add to grammar for IO.....	7
4.2.2.	Eliminate reduce-reduce/shift-reduce conflicts	7
4.2.3.	Change some error grammars to be correct	7
4.2.4.	Refine the priority of operators.....	8
4.2.5.	Left Recursion.....	8
4.3.	Syntax Tree	8
5.	Intermediate-code Generation.....	8
5.1.	Semantic Analysis	8
5.1.1.	Variables and functions should be declared before usage	9
5.1.2.	Variables and functions should not be re-declared.....	9
5.1.3.	Reserved words can't be used as identifiers	9
5.1.4.	Program must contain a function main.....	9
5.1.5.	The number and type of variable(s) passed should match the definition of the function 9	
5.1.6.	Use [] operator to a non-array variable is not allowed and the "." operator can only be used to a structure variable	9
5.1.7.	Break and continue can only be used in a for-loop	10
5.1.8.	Right-value can't be assigned by any value or expression	10
5.1.9.	The condition of if statement should be <i>int</i> type, the condition of for statement should be <i>int</i> or ϵ and only expression with type <i>int</i> can be involved in arithmetic 10	
5.1.10.	Read function should use as the assignment	10
5.1.11.	Array access should be as same dimensions as when it declared	10
5.2.	Intermediate Representation	10
5.2.1.	All labels	10
6.	Code Optimization.....	11
7.	Machine-code Generation	11
7.1.	Register Allocation.....	11
7.2.	Input and Output.....	11
8.	Test.....	12
9.	Summary	13

1. Introduction

In this project, we are required to implement a small-c compiler to translate small-c source codes to MIPS assembly codes. These assembly codes can run on the SPIM simulator or be assembled into machine code and run on a real computer. In my testing, I use the SPIM simulator to test my output assembly codes.

Small-c is a C-like language containing a subset of the C programming language. Using **flex**, **bison** and **g++** to generate the executable file “scc”, we can get the final form of the compiler, which can easily generate the MIPS codes from small-c codes.

Flex is used for Lexical Analyzer, **Bison** is used for Syntax Analyzer and **g++** is used for make code become executable file. And another tool **SPIM** is used for running the MIPS code to check the accuracy of the compiler.

2. Execution

2.1. Environment

Ubuntu 14.04, Flex 2.5.35, Bison 3.0.2, g++ 6.2.0, SPIM 8.0

2.2. Instruction

`./scc [“input file” “output MIPS code file” “output IR code file” “output parser tree file”]`

All parameter is optional, which can satisfy with the require in the project instruction. (If you don't use input file, you need to write code in the shell, the same as the output MIPS code file.) And other information will output in the stderr.

Or you can put your input file with suffix cpp into any folder in the folder testInput and type:

`make run`

This instruction will automatically compile all cpp files and output MIPS code in folder MipsCode, IR code in folder InterCode and parser tree in folder ParserTree with corresponding name.

3. Lexical Analyzer

The lexical analyzer is implemented by using flex. It will simply read the source codes and separate them into tokens.

3.1. Tokens

We separate the input characters into reserved words, operators, integers and identifiers which are as follow:

INT	⇒	/* integer ¹ */
ID	⇒	/* identifier ² */
SEMI	⇒	;
COMMA	⇒	,
BINARYOP	⇒	/* binary operators ³ */
UNARYOP	⇒	/* unary operators ⁴ */
TYPE	⇒	int
LP	⇒	(
RP	⇒)
LB	⇒	[
RB	⇒]
LC	⇒	{
RC	⇒	}
STRUCT	⇒	struct
RETURN	⇒	return
IF	⇒	if
ELSE	⇒	else
BREAK	⇒	break
CONT	⇒	continue
FOR	⇒	for

In addition, I use two more reserved words: read and write, which are used to get input and output for the program.

3.2. Implementation

The lexical analyzer is easy to implement by using regular expressions with flex tool. The return value and the token type which I defined to be char* would be defined in the next part, syntax analyzer, which can combine lexical with syntax together to make the front-end of compiler.

Meanwhile, since the flex separate tokens with first match when at same length, we need to put identifier after the reserved words.

4. Syntax Analyzer

The syntax analyzer is implemented by using bison or another word, yacc. It uses tokens generated by flex and grammar to build a syntax tree, which can be used for back-end code generation.

4.1. Grammar

The grammar that syntax analyzer uses is as follow:

PROGRAM	→	EXTDEFS
EXTDEFS	→	EXTDEF EXTDEFS
		ε
EXTDEF	→	TYPE EXTVARS SEMI
		STSPEC SEXTVARS SEMI
		TYPE FUNC STMTBLOCK
SEXTVARS	→	ID
		ID COMMA SEXTVARS
		ε
EXTVARS	→	VAR
		VAR ASSIGN INIT
		VAR COMMA EXTVARS
		VAR ASSIGN INIT COMMA EXTVARS
		ε
STSPEC	→	STRUCT ID LC SDEFS RC
		STRUCT LC SDEFS RC
		STRUCT ID

FUNC	→	ID LP PARAS RP
PARAS	→	TYPE ID COMMA PARAS
		TYPE ID
		ϵ
STMTBLOCK	→	LC DEFS STMTS RC
STMTS	→	STMT STMTS
		ϵ
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
DEFS	→	TYPE DECS SEMI DEFS
		STSPEC SDECS SEMI DEFS
		ϵ
SDEFS	→	TYPE SDECS SEMI SDEFS
		ϵ
SDECS	→	ID COMMA SDECS
		ID
DECS	→	VAR
		VAR COMMA DECS
		VAR ASSIGN INIT COMMA DECS
		VAR ASSIGN INIT
VAR	→	ID
		VAR LB INT RB
INIT	→	EXP
		LC ARGS RC
EXP	→	EXPS
		ϵ
EXPS	→	EXPS BINARYOP EXPS
		UNARYOP EXPS
		LP EXPS RP
		ID LP ARGS RP
		ID ARRS
		ID DOT ID
		INT
ARRS	→	LB EXP RB ARRS
		ϵ
ARGS	→	EXP COMMA ARGS
		EXP

4.2. Grammar Refinements

The grammar showed above has some of bugs, which needs refining.

4.2.1. Add to grammar for IO

Add “READ LP EXPS RP SEMI” and “WRITE LP EXPS RP SEMI” in STMT, for we need read and write statement to do input/output.

4.2.2. Eliminate reduce-reduce/shift-reduce conflicts

There exists a conflict in the implement of “IF LP EXP RP STMT” and “IF LP EXP RP STMT ELSE STMT”, and the former one should have a lower precedence than the latter one. Therefore, we use the priority setting in bison to eliminate this conflict.

```
%nonassoc IF_NO_ELSE
%nonassoc ELSE
```

```
| IF LP EXPS RP STMT %prec IF_NO_ELSE
| IF LP EXPS RP STMT ELSE STMT
```

4.2.3. Change some error grammars to be correct

For the grammar given, there is some error which doesn't meet the standard of C as well as small-c, such as:

4.2.3.1. “IF LP EXP RP STMT” should be “IF LP EXPS RP STMT” since the bool expression can't be empty. The same as “INIT: EXP”, “ARRS: LB EXP RB ARRS”, “ARGS: EXP COMMA ARGS” and etc.

4.2.3.2. For the grammar like:

$$\begin{array}{lcl} \text{SEXTVARS} & \rightarrow & \text{ID} \\ & | & \text{ID COMMA SEXTVARS} \\ & | & \epsilon \end{array}$$

This external variable declare can become “ID COMMA” or “ID COMMA ID COMMA” and so on. With the SEMI behind, it could become wrong grammar in C. Therefore, it should be changed into:

```
SEXTVAR: SEXTVARS
        | /* empty */
        ;
SEXTVARS: ID
        | SEXTVARS COMMA ID
        ;
```

The same as EXTVARS, PARAS and ARGS.

4.2.4. Refine the priority of operators

The binary operators and the unary operators have priority defined in the lexical analyzer. In order to implement it, which eliminate another conflict, we define their priority the same as the IF-ELSE statement and split the “BINARYOP” into these specific ones.

```
%right ASSIGN ADD_AS SUB_AS MUL_AS DIV_AS MOD_A
%left LOG_OR
%left LOG_AND
%left BIT_OR
%left BIT_XOR
%left BIT_AND
%left EQ NE
%left GT GE LT LE
%left LEFT_SHIFT RIGHT_SHIFT
%left PLUS MINUS
%left PRODUCT DIV MOD
%right UNARY
%left DOT LP LB
```

4.2.5. Left Recursion

Since the bison is more efficient when using left recursion grammar, and the grammar showed above is nearly all right recursion. Therefore, I change most of them into left recursion grammar if possible.

4.3. Syntax Tree

In order to build a syntax tree, I define the tree node class and an insert function in another header file. Then, use these in the parser procedure. When a statement is reduced by the parser grammar, it would simply insert the node into syntax tree. Finally, when parsing complete, we will get the whole syntax tree with the root of the starting symbol PROGRAM.

5. Intermediate-code Generation

In this part, we will generate the intermediate code by using the parser tree, and meanwhile, do the semantic analysis to check the program is correct.

5.1. Semantic Analysis

After syntax analysis, we can build a syntax tree with the whole program. It matches our grammar to be a program. But not all program that matches the grammar is the correct program, we need to do semantic analysis and syntactic checking to examine potential semantic errors.

In this part, I derive most of the tree node class with the class name of parser type from the basic tree node class. I use a virtual function “codeGen”, in which different implementation has been used according to its class name. This “codeGen” function will simply do semantic check as well as intermediate code generation.

5.1.1. Variables and functions should be declared before usage

I use a static function table to save all functions that have been declared, and symbol table and structure table within the PROGRAM class and the STMTBLOCK class since variable could only be declared in these two classes and the variable inside one STMTBLOCK can have the same name as that outside this STMTBLOCK. Then, when a variable is used, I will check its type, and according this type, I find it on function table or on symbol table and get if it has been declared or not.

5.1.2. Variables and functions should not be re-declared

Like 5.1.1, if a function or a variable is declared, I will check the function table or symbol table according to its type. If there exist the same name, then a re-declared error will arise.

5.1.3. Reserved words can't be used as identifiers

This has been solved in lexical analysis part in its implement for identifiers come later than reserved words.

5.1.4. Program must contain a function main

After the generation, I will check the function table if there exist the function name main and with the parameter number 0.

5.1.5. The number and type of variable(s) passed should match the definition of the function

When execute the function by “EXPS: ID (ARG)”, we generate the ARG and calculate its number, and then compare it with the parameter number defined in the function table.

5.1.6. Use [] operator to a non-array variable is not allowed and the “.” operator can only be used to a structure variable

These two semantic check is used the same as the 5.1.1, for checking the variable type before use. And arise error message if necessary.

5.1.7. Break and continue can only be used in a for-loop

When meeting a for-loop statement, we add the for-loop calculation and save its jump label. Then, when we meet a break or a continue statement, we check if the for-loop calculation is 0. If it is not 0, we set jump into the corresponding label. When exiting a for-loop, minus the for-loop calculation.

5.1.8. Right-value can't be assigned by any value or expression

When meeting any assignment operator, we check whether the left expression is left-value or not. Since the left-value expression is only "ID", "ID.ID" and "ID ARRS", it can easy to be checked.

5.1.9. The condition of if statement should be *int* type, the condition of for statement should be *int* or ε and only expression with type *int* can be involved in arithmetic

Since the whole program is just use *int* type for calculation, it doesn't matter for this rule with *int* type. And the ε of for statement has been solved when we do parser and meet null expression.

5.1.10. Read function should use as the assignment

Actually, the read function should also use for left-value expression like the assignment.

5.1.11. Array access should be as same dimensions as when it declared

When access the array, we will check its argument numbers, confirming it has the same argument as when it declared.

5.2. Intermediate Representation

As for intermediate code, I use the three-statement sequence to represent it. And also, I use some of my own label to representation for what is not the expression due to it is easy to recognition for translating into lower-level language.

5.2.1. All labels

global: mean the global definition.

define: mean the function definition start

return: mean the return of the function

label: mean the jump label

call: mean the call of the function

read: mean the call of the read function

write: mean the call of the write function

paras: mean the save of the function parameters

[]=: mean the array value to be left-value

[]: mean the array value to be right-value

#: mean this use the value in memory

@: mean this use the result of the previous one, with the value later mean the relative distance

Others is the same as expression or the same as the MIPS.

6. Code Optimization

Since we maintain the table, we can easily remove the unused function declaration. These function code will be simply dropped as well as structure type definition.

7. Machine-code Generation

In this part, we will do the translation from the intermediate code we generated in the last part into the format MIPS code, and we can run the MIPS code in the SPIM simulator.

For every intermediate code, we can choose a most suitable instruction in MIPS to translate it, but some of the instruction have not common with others, like it has “addi” but no “subi”, “muli” and etc. Therefore, we give up some of the instruction in order to make the translation more formative and easy to read.

The intermediate code is easy to translate because we generate it with purpose.

7.1. Register Allocation

We use a set to save the available registers. When we need to allocate register, we will get a register from it. Meanwhile, we recorded the effect of every instruction. If the instruction will affect a later one, we will save its result register for later use, or otherwise we will free it into available one.

7.2. Input and Output

The input and output function is defined in the parser tree. We use the “syscall” instruction in MIPS to do input/output, and also we use a global newline character for every write function to make line feed after an integer has been outputted.

8. Test

After implementation, we use lots of the program in the “testInput” folder to test the correctness of the compiler. Diverse programs are used to test all of the condition of our compiler and confirm it has hardly bugs. Here is all the program purpose:

io: test the basic function of input/output

op: test all of the basic operators use write function

struct: test all of the definition of structure (with or without name, with or without id and etc.)

ifelse: test the if else condition to check if its priority is correct as well as for-loop

fib: test the array variable and for-loop by using the calculation of Fibonacci number

gcd: test the function call and the recursion of function by calculating greatest common divisor

transpose: test the 2-dimensional array and its initialization by transposing

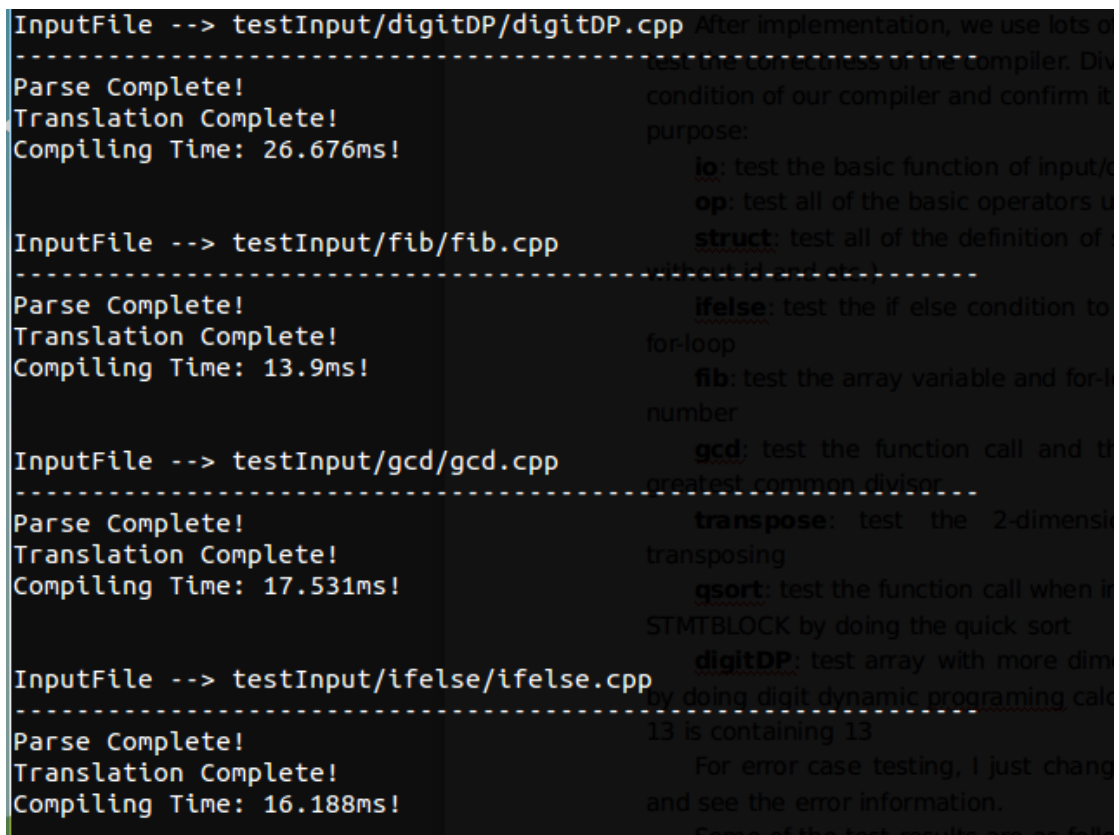
qsort: test the function call when initializing and the variable declaration in STMTBLOCK by doing the quick sort

digitDP: test array with more dimension and test more complex program by doing digit dynamic programming calculating the numbers of what divided by 13 is containing 13

For error case testing, I just change the program above to make it wrong and see the error information.

Some of the test results are as follow:

The compiling:



```
InputFile --> testInput/digitDP/digitDP.cpp
-----
Parse Complete!
Translation Complete!
Compiling Time: 26.676ms!

InputFile --> testInput/fib/fib.cpp
-----
Parse Complete!
Translation Complete!
Compiling Time: 13.9ms!

InputFile --> testInput/gcd/gcd.cpp
-----
Parse Complete!
Translation Complete!
Compiling Time: 17.531ms!

InputFile --> testInput/ifelse/ifelse.cpp
-----
Parse Complete!
Translation Complete!
Compiling Time: 16.188ms!
```

Compiling with error:

```
c-tan-one@Tanyi:~/study/00_compilerPrinciple/project$ ./scc testInput/gcd/gcd.cpp MipsCode/gcd/gcd.s
InputFile --> testInput/gcd/gcd.cpp
-----
Parse Complete!
Error: Semantic error at line 11
Expected rules: EXPS: ID ( ARG )
call function gcd with 1 paras error
Exit

c-tan-one@Tanyi:~/study/00_compilerPrinciple/project$ ./scc testInput/digitDP/digitDP.cpp MipsCode/digitDP/digitDP.s
InputFile --> testInput/digitDP/digitDP.cpp
-----
Parse Complete!
Error: Semantic error at line 25
Expected rules: EXPS: ID ARRS
error in get value of array
Exit
```

Example run of quick sort in SPIM:

```
c-tan-one@Tanyi:~/study/00_compilerPrinciple/project$ spim -f MipsCode/qsort/qsort.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
8
9
6
3
5
8
4
2
1
1
2
3
4
5
6
8
9
```

9. Summary

In this project, I just use **Flex**, **Bison**, **g++** and **SPIM** to create a compiler and test it. It is a little easy to do lexical analysis and syntax analysis since they have specific tools to use. And more difficult in the back-end coding. No matter how, I just completed the whole compiler and test it successfully.

With doing the whole project, I think my understanding of the compiler principle has improved. I know more about how to create a compiler and how the compiler works. It practices me much more than other projects and makes my coding skills and debugging skills better and better.

Thanks to teacher and TAs who have helped me more or less.