Projects of Compiler Principles
CS215

Yaoming Zhu

5140219355

Date Performed: January 18, 2017
Instructor: Professor Li Jiang

1 Objective

In this project, it is required to design and implement a simplified compiler, for a given programming language, namely smallC, which can be regarded as a subset and the core part of C programming language. The complier should translate smallC source code to MIPS assembly codes.

1.1 Project Logic

Lexical Analyzer The *Flex* tool is used to help design a lexical analyzer generator.

Syntax Analyzer The *Bison*(A.K.A *Yacc*) tool is used to help design an LALR parser generator.

Abstract Syntax Tree After syntax analysis, the AST tree can be generated. It helps to test the output of former process.

Semantic Analyzer Possible semantic errors is checked in this part.

Intermediate Representation Code Translation The AST tree is translated into IR code in this part.

Optimization Some optimizations are applied to IR code in this part, i.e. dead code elimination.

Machine-code Generation The IR code is interpreted into MIPS code in this part.

Project Test Test the project with some smallC code and SPIM.

1.2 Project Environment

The project guidance says "In this project, Linux environment (Ubuntu) is required". Fortunately, with the help of *bash on ubuntu on windows*, this project can be implemented under Windows system.
Furthermore, *Win flex-bison* provided with all flex and bison function to the Windows platform, which means this project can be implemented even under Windows IDE! (And I actually finished most parts of the project with Visual Studio 2015). With IDE support, the debugging process becomes much more efficient and precise.
For setup details, please refer to
https://sourceforge.net/p/winflexbison/wiki/Visual%20Studio%20custom%20build%20rules/

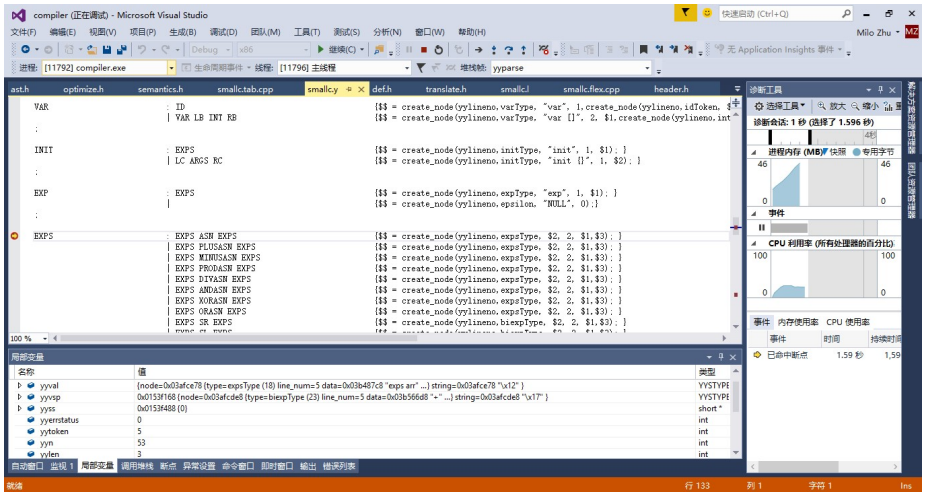

Figure 1: The Project on Visual Studio 2015

Besides the FTP-uploaded one, I have also uploaded a Visual Studio project, they only small difference they have is that, for the output short name of *Win flex-bison* is different form Ubuntu one, thus it leads to different head file reference. You can access Visual Studio version form the link in readMe file.

2 Lexical Analyzer

With the help of *Flex*, this part is fairly simple.
This part is done in *smallC.l*.

2.1 Keywords

Since the keywords are fixed, the lexical analyzer will return corresponding token and string value once it find the keywords. *yylval* is provided by *flex* to help to capture the string content of our token.

```
int             {yylval.string = strdup(yytext); return TYPE;}
struct            {yylval.string = strdup(yytext); return STRUCT;}
return            {yylval.string = strdup(yytext); return RETURN;}
if              {yylval.string = strdup(yytext); return IF;}
else             {yylval.string = strdup(yytext); return ELSE;}
break             {yylval.string = strdup(yytext); return BREAK;}
continue            {yylval.string = strdup(yytext); return CONT;}
for             {yylval.string = strdup(yytext); return FOR;}
read            {yylval.string = strdup(yytext); return READ;}
write             {yylval.string = strdup(yytext); return WRITE;}
```

Note. According to the guidance, the *'read'* and *'write'* are not the smallC keywords. However, they are used to test input and output function the compiler, thus they are regarded as keywords in this project.

2.2 Numbers and Identifiers

The token for numbers and identifiers are a little more tricky, regular expression is needed.

```
digitnon0 [1-9]
hexdigit [0-9A-Fa-f]
letter [A-Za-z]
digit  ({digitnon0}|0)

identifier ((_|{letter})(_|{letter}|{digit})*)
dec_int  (({digitnon0})+({digit})*)
oct_int  (0({digit})*)
hex_int  (0(x|X)({digit})*)
%%
....// keywords
{dec_int}           {yylval.string = strdup(yytext); return INT; }
{oct_int}           {yylval.string = strdup(yytext); return INT; }
{hex_int}           {yylval.string = strdup(yytext); return INT; }
{identifier}          {yylval.string = strdup(yytext); return ID;}
```

2.3 Blank Characters and Comments

Like most C-like languages, smallC will just ignore the blank characters, so do the lexical analyzer. The only exception is the newline characters: the lexical analyzer will use it to get current line number. A variable *yylineno* is introduced to keep record of current line number.

```
extern int yylineno;

%%
[\t\r\f]            {;}
```

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

As for comments, two different cases shall be treated, the in-line comment(//) and the out-of-line comment(/* */). The former one will end once the newline character is encountered, the later one will end only if "*/" is encountered.

```

void commentInLine();
void commentOutLine();

%%
"//"      {commentInLine();}
"/*"      {commentOutLine();}
%%

void commentInLine(void){
 char c;
 while((c = input())!='\n');
 yylineno++;
}

void commentOutLine(void){
 char c1, c2;
 c2 = input();
 do{
  c1 = c2;
  c2 = input();
  if(c1 == '\n'){
   yylineno++;
  }
 }while(!(c1 == '*' && c2 == '/'));
}
```

2.4 Test

Unless otherwise specified, the following smallC code will be used as test code.

```
int main()
{
 int x;
 read(x);
 x = x + 10;
 write(x);
 return 0;
}
```

test.c

After *flex* the .l file and compile, we can get the result:
The lexical analyzer works well.

3 Syntax Analyzer

The design of syntax analyzer is simple but tough.

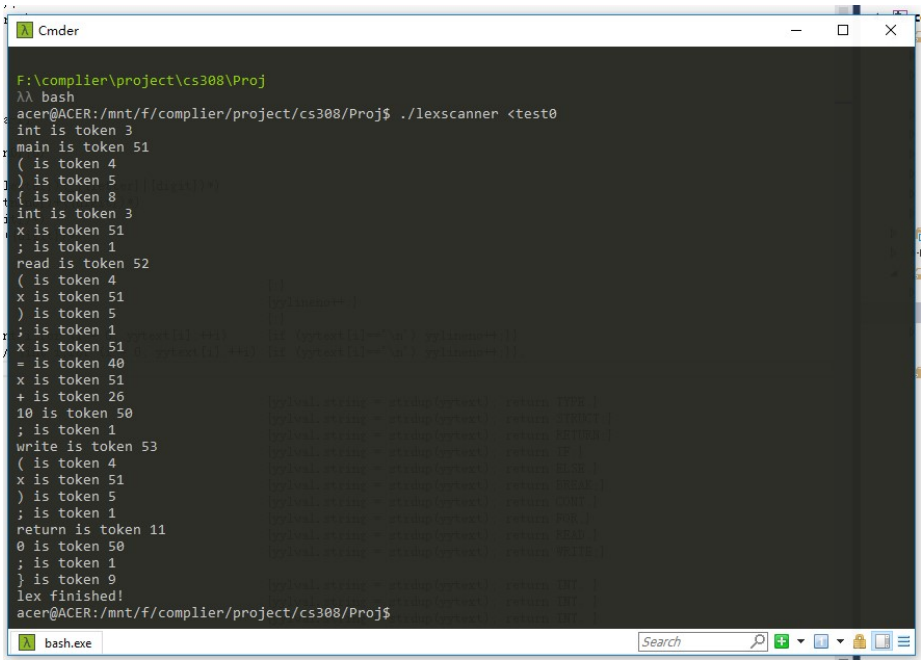

Figure 2: The Output of Lexical Analyzer

This part is done in *smallC.y*. *Bison* is used in this part.
First and foremost, a data structure is required to represent abstract syntax tree(AST). Thus, a struct is constructed as AST node, which shall have following functions:

a. Hold the AST node type

b. Hold the pointers pointing to its sub AST-node.

Thus, it is designed as:

```
typedef struct TreeNode {
    nodeTypeEnum type;
    int lineNum;
    char* data;
    int size, capacity;
    struct TreeNode** children;
} TreeNode;
```

size and *capacity* hold values for number of this kind of AST node and max number of AST can have respectively. *lineNum* is the corresponding line number of the AST node. *data* store some informations. For terminals(i.e. identifier), it contains string content; for non-terminals, it store the derivation case, for most non-terminals have more than one production. *type* is an enum type to hold the AST node type, which is defined as

```
typedef enum{
 programType,
 extdefsType,
 extdefType,
 //......other non-terminals
 unaryopType,
 idToken,
 biexpType,
 keywords,
 typeToken,
 intToken,
 epsilon
} nodeTypeEnum;
```

Besides the non-terminals given by the guidance, here some other AST types are introduced:

a. Terminal types: keywords, typeToken, intToken, idToken,epsilon

b. Auxiliaries: unaryopType, biexpType

"Auxiliaries" is special cases of *EXPS* non-terminal. *unaryopType* is for unary operators, while *biexpType* is for all binary expressions other than those concerned with assignment.
Since the sub AST varies from AST node types, a variable arguments functions is implemented to create AST node in the process.

```
TreeNode* createNode(int lineno,nodeTypeEnum type, char* data, int
    cnt, ...) {
    TreeNode* ptr = (TreeNode*)malloc(sizeof(TreeNode));
    ptr->lineNum = lineno;
    ptr->type = type;
    ptr->data = strdup(data);
    ptr->size = ptr->capacity = cnt;
    ptr->children = (TreeNode**)malloc(sizeof(TreeNode*) * cnt);
    va_list ap;
    va_start(ap, cnt);
    int i;
    for (i = 0; i < cnt; i++) {
        ptr->children[i] = va_arg(ap, TreeNode*);
    }
    va_end(ap);
    return ptr;
}
```

Then, the grammars of smallC is translated into *bison* file. Taken *EXTDEF* as instance:

```
EXTDEF      : TYPE EXTVARS SEMI        {$$ = createNode(yylineno,
    extdefType, "extdef1", 2, createNode(yylineno,typeToken, $1, 0)
    ,$2); }
    | STSPEC SEXTVARS SEMI       {$$ = createNode(yylineno,
    extdefType, "extdef2", 2, $1,$2); }
    | TYPE FUNC STMTBLOCK        {$$ = createNode(yylineno,
    extdefType, "extdef func", 3, createNode(yylineno,typeToken, $1
    , 0),$2,$3); }
```

```
;
```

As the guidance indicated, grammar given may induce one or two reduce-reduce/shift-reduce conflicts. Specifically, *"IF LP EXP RP STMT* should have lower precedence than *IF LP EXP RP STMT ELSE STMT*. A trick of *bison* is applied here:

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%%
STMT:   //......
        | IF LP EXPS RP STMT %prec IFX     {$$ = createNode(yylineno,
    stmtType, "if stmt", 2, $3,$5); }
        | IF LP EXPS RP STMT ELSE STMT %prec ELSE {$$ = createNode(
    yylineno,stmtType, "if stmt", 3, $3,$5,$7);}
 ;
```

One more modification is applied to grammar. As *read* and *write* is regarded as keywords in the project, the production of *STMT* now is

```
STMT        : EXP SEMI              {$$ = ...; }
        | STMTBLOCK             {$$ = ...; }
        | RETURN EXPS SEMI          {$$ = ...; }
        | IF LP EXPS RP STMT %prec IFX     {$$ = ...; }
        | IF LP EXPS RP STMT ELSE STMT %prec ELSE {$$ = ...;}
        | FOR LP EXP SEMI EXP SEMI EXP RP STMT  {$$ = ...; }
        | CONT SEMI             {$$ = ...; }
        | BREAK SEMI            {$$ = ...; }
        | READ LP EXPS RP SEMI        {$$ = ...;}
        | WRITE LP EXPS RP SEMI        {$$ = ...;}
```

4 Abstract Syntax Tree

This part is done in *ast.h* and *def.h*.
With *samllC.y* finished, the program now can output the AST. The struct *TreeNode* has already implemented the tree, thus the generation of AST is quite simple with techniques learned form data structure classes.

```
void printAst(TreeNode* ptr, int depth) {
    int i;
    int n = (depth - 1) * 2;

    if (depth > 0) {
        for (i = 0; i < n; i++) {
            putchar(buffer[i]);
        }
        putchar('|');
        puts("");
    }

    for (i = 0; i < n; i++) {
        putchar(buffer[i]);
    }
```

```
    if (depth > 0) {
        putchar('|');
        putchar('_');
    }
    n = depth * 2;
    buffer[n] = '|';
    buffer[n + 1] = ' ';
 if (ptr == NULL) {
  printf("NULL\n");
  return;
 }
    printf("%s\n", ptr->data);
    // terminal customized configuration
    if (ptr->size > 0) {
        printf("%s\n", ptr->data);
    } else {
        printf("%s\n", ptr->data);

        for (i = 0; i < n; i++) {
            putchar(buffer[i]);
        }
        puts("");

    }
    for (i = 0; i < ptr->size; i++) {
        if (i == ptr->size - 1) {
            buffer[n] = ' ';
        }
        printAst(ptr->children[i], depth + 1);
    }
}
```

The AST of test.c is stored at file *AST*. It turned out that the syntax analyzer works well.

5 semantic analysis

This part is done in *semantic.h*.
In this part, potential semantic errors after generating a parse tree is checked. Besides, the symbol table is generated. Some basic evaluation (i.e. get values of int) is also done in this part.

5.1 Symbol Table

In this part, a struct is designed to implement the symbol table.

```
struct ptr_cmp {
 bool operator()(const char* s1, const char* s2) const {
  return strcmp(s1,s2) < 0;
 }
};
struct SymbolTable {
 map <char*, char*,ptr_cmp> table;
 map <char*, vector<char *>, ptr_cmp > struct_table;
```

```
 map <char*, vector<char *>, ptr_cmp > struct_id_table;
 map < char*, int , ptr_cmp > struct_name_width_table;
        map <char*, int , ptr_cmp> width_table;
 map <char*, vector<int >, ptr_cmp> array_size_table;
 int parent_index;
} symTable[MAX_LEVEL][MAX_DEPTH];
```

Here STL data structure *map* is used for string-key inquiry.
As the member variable name indicates, they are used to map between:

member	string	key
table	local/global variable names	variable's type
struct_table	struct names	all member variable names of the struct
struct_id_table	struct names	member names certain type of the struct
struct _name_width_table	struct names	the number of members of the struct
width_table	int/array names	space it comsumes
array_size_table	array names	vhow many 4 bytes of the width each dimension

Table 1: Symbol Table Content

symTable[MAX_LEVEL][MAX_DEPTH]: Here the symbol table is implemented as 2D array. The row index indicates the level of certain table. The smallC code has code blocks(" "), which requires a new symbol table. when the program is running, the symbol table of each code blocks has its own scope. Thus it is necessary to keep the level on track.
Besides, member variable *parent_index* is used to record the upper level's index. *parent_index* will be 1 for global symbol table.

5.2 Semantic Checking

For each AST node type, a corresponding check function is implemented.Normally it will get the AST node type of its sub-AST and call corresponding check function to check its sub-tree one by one. i.e.

```
void check_program(TreeNode* p) {
 if (p == NULL) return;
 int i;
 for (i = 0; i < p->size; ++i) {
  check(p->children[i]);
 }
}
```

Specifically, the following things need to be checked: (since the amount of code in this part is really large, only design logic instead of code is presented here)

5.2.1 Variables and functions should be declared before usage

For variables, every time an identifier node is encountered, the compiler will check it in the symbol table on current level and its upper levels.

For functions, the complier only allows global functions, which is checked on level 0 symbol table. *Overloaded function* is allowed in the compiler, thus not only the function name but its arguments will be checked in this part.

5.2.2 Variables and functions should not be redeclared

Similarly to the previous one, whenever a new variable is declared, it will be checked in the current-level symbol table to find out whether it has already existed.

5.2.3 Reserved words can not be used as identifiers

Whenever an identifier node is encountered, its name(stored in the *data*) will be compared to keywords.

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

A global variable *bool_main* is initialized to be false. Once *main* is detected in *FUNC* node, it will be true,otherwise the semantic checking won't pass.

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

This part is already done as a by-product of part 5.2.1

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

With *width_table* from symbol table, the compiler can get access to the space an array or variable consumes. From it the compiler can tell whether it is an array or a int, then [] operation to int will be rejected.
Note. The declaration like a[1] is not allowed by the compiler, since it cannot distinguish it with ordinary int.

5.2.7 The . operator can only be used to a struct variable

With *struct_table* from symbol table, the complier can check whether a variable has been declared as struct. Then with *struct_id_table*, the identifier after . is checked whether it is a member of the struct.

5.2.8 *break* and *continue* can only be used in a *for*-loop

A global bool variable is declared to tell whether the current *STMT* node is inside a loop.

5.2.9 Right-value can not be assigned by any value or expression

This part will be implemented in the intermediate representation section.

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

The logic is similar to part 5.2.6, *width_table* is checked to tell the type of expression.

5.2.11 The condition of for should be an expression with int type or epsilon

5.2.12 Only expression with type int can be involved in arithmetic

These parts are similar to 5.2.10

5.2.13 Other details

a. Size of array cant be negative or 0

b. Members of a struct and argument of a function cant have same names

6 Intermediate Representation Code Translation

This part is done in *translate.h*.
Here comes the most tough, complex and exhausting part. After passing semantic analysis, the smallC code can be regarded as a correct one and the AST node will be translated into intermediate representation(IR).

6.1 Intermediate Representation

Here a MIPS-like three-address code is selected as IR. The different between IR and MIPS is IR has infinite registers so that every variable can be hold in the registers.
Like before, a struct is designed to implement IR instructions.

```
// for MIPS generation
struct Address{
    RegType type; //type of the argument
    string name; //For labels and function calls
    int value; // register value
    int real;// physical register in MIPS
    int needload; // whether load op is necessary
    int needclear;//  whether this op should be cleared
};

struct Quadruple{
    string op; // name of operation
    int active; //whether it's active
    int flag; //whether its arguments should be modified
    Address arguments[3]; //the three arguments
};
```

RegType in struct Address is an enum type, which represent the type of corresponding address. It can be:

a. ADDRESS_LABEL: The address is jump label number

b. ADDRESS_CONSTANT: The address is constant number

c. ADDRESS_TEMP: It represents a register

d. ADDRESS_NAME: It represents a name

opname	Destination	Operand 1	Operand 2	Description
or	a	b	c	$a = b\|c$
xor	a	b	c	a = bĉ
and	a	b	c	a = b&c
sll	a	b	c	$a = b << c$
srl	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			$a = -b$
lnot	a	b		$a =!b$
not	a	b		$a = \ b$
beqz		a	label	if a==0 goto label
bnez		a	label	if a!=0 goto label
bgez		a	label	if a¿=0 goto label
bgtz		a	label	if a¿0 goto label
blez		a	label	if a¡=0 goto label
bltz		a	label	if a¡0 goto label
li	a	b(constant)		a = b
lw	a	b	c(constant)	a = mem[b+c]
sw	[memory]	a,b,c(c is constant)		mem[b+c] = a
move	a	b	c	a = b
label		l		l is int, set label
goto		l		l is int, unconditional jump
func		s		s is string, set function
call		s		s is string, call function

Table 2: Three Address Code Logic

Some vectors are also used to help the translation process:

```
vector <Quadruple> IR, GIR, MIR;
vector <int> RegisterState, RegisterOffset,LabelCont, LabelBreak,
    vs_reg;
vector <string> vs_id;
```

a. RegisterState: The register can either hold value or address. RegisterState is used to tell its current state

b. RegisterOffset: contains the info of offsets of the register.

c. LabelBreak and LabelCont: indicates the label for break and continue operation.

d. vs_reg and vs_id: associates identifiers with the registers.

6.2 Translate

6.2.1 EXPS translate

For *EXPS* AST ndoe, the complier will classify it into following cases:

Arithmetic A new register will be allocated to store the result of the arithmetic process

Logical The following IR is used to assign the bool value to register:

```
li [register],1
beqz/bnez/bgez/bgtz/blez/bltz judgement label
li [register],0
label 1
```

Unary For ordinary ones, a new register will be allocated.

For "++" and "–", not only a new register is allocated but the one participating in the operation will be modified as well.

Assignment First, the operations like "*=" will be decomposed into arithmetic one and pure assignment.

Then for pure assignment, the compiler will just do register modifications.

Array and Struct Offset will be applied to visit the member.

6.2.2 FUNC translate

The arguments of functions are stored on the stack.
Every time a function(except main()) call is encountered, the $ra will be stored in the stack. The stack pointer will be restalled after function call.

6.2.3 STMT translate

if *goto* label in IR can be used here. That is if the judgement of if is false, the complier will use jump to else statement(let's call it s2); otherwise it will first execute the statement right after judgement(let's call it s1). Then a *goto* is placed right after s1 to jump out of the block, which will not execute s2 in the case.

for *goto* label in IR can be used here, too. The basic logic is almost the same except there are one more jump label to jump back in order to form a loop.

read and write

The *read()* is to assign the value of register $v0 into the register associated with variable, while *write()* assign value to register $a0.
It is implemented with *scanf_one* and *printf_one*

```
__printf_one:
li $v0, 1
syscall
jr $ra
__scanf_one:
li $v0,5
syscall
jr $ra
```

7 Optimization

Subexpression elimination and dead code elimination is done in this part.

7.1 Subexpression elimination

This part is done in *optimize.h*
Subexpression elimination are like change:

```
move t2,t1
move t3,t2
```

into

```
move t3,t1
```

and

```
li t1,1
move t2,t1
```

into

```
move t2,1
```

Here t2 must not be reassigned afterwards. A flag is used to indicate whether the register is reassigned later.

7.2 Instruction Selection

This part is done in *interprete.h*
The following operation is done to reinforcement optimization:

a. Check labels and delete those whose destination immediately following.

b. For all arithmetic expressions, interpret into intermediate instructions.

c. Check for successive lws and sws. Delete the sw where the the former instruction is lw and their target are the same.

d. use shift operations when multiplying 2 or 4.

8 Machine Code Generation

This part is done in *interprete.h* and *optimize.h*
After optimization, the compiler will interpret the IR into MIPS code. The major difference between IR and MIPS is the number of registers. Thus the compiler shall implement register allocation.
Here the compiler will scan through $11 to $25 to find out the free register. If no free one, it will scan $8 to $10. If no register is available currently, *lw* and *sw* will be applied to store variables into memory.

8.1 Project Test

The complier is already finished, the test is necessary.
The test.c is translated into

```
.globl main

.data
temp: .space 44

.text
__printf_one:
li $v0, 1
syscall
jr $ra
__scanf_one:
li $v0,5
syscall
jr $ra

main:
la $a3, temp
add $11, $29, -40
move $13, $11
move $11, $13
jal __scanf_one
sw $2, 0($11)
move $11, $13
```

```
move $12, $13
lw $12, 0($12)
add $12, $12, 10
sw $12, 0($11)
move $11, $13
lw $11, 0($11)
move $4, $11
jal __printf_one
li $11, 0
add $12, $29, 4
sw $11, 0($12)
j __program_end
j __program_end
__program_end:
li $v0, 10
syscall
```

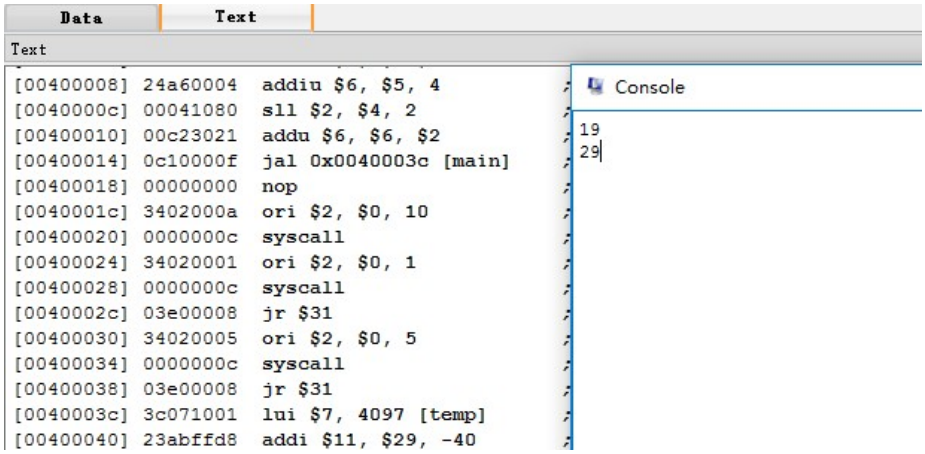

Figure 3: Test on SPIM

It runs on QtSPIM precisely. The compiler works well.

9 Acknowledgement

This project is large and tough. After finishing it, I learned a lot about compliers ,C language and MIPS.
I would like to express my sincere thanks to instructor Prof. Jiang and teaching assistant. The class and project taught me a lot. Thank you for your contribution!