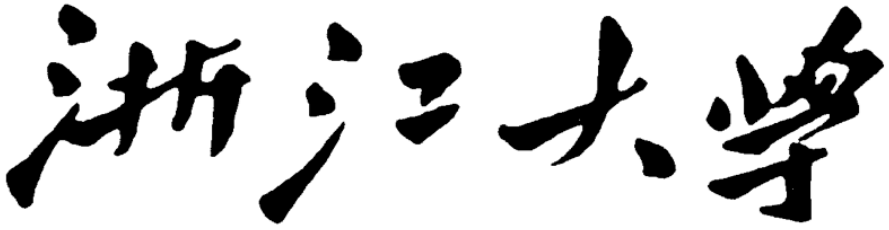


本科实验报告

课程名称：编译系统设计

姓　　名：吉辰阳 朱灏 王超奇 黄昭阳

学　　院：计算机科学与技术学院

系：计算机科学与技术系

专　　业：计算机科学与技术

2015 年　7　月　9　日

目录

1. Introduction

1.1 background

After we have spent a semester learning about those basic rules of compiler design, some advanced programming exercises would be needed. In this project, we designed a basic Pascal compiler with C++, which supports those syntaxes provided by our instructor.

In this report, we would firstly talk about our developing environment, then analyze each part of our program, including lexical analysis, syntax analysis, sematic analysis, code generation, as well as the error recovery and code optimization.

And finally, we would give some test cases to verify the correctness of our program.

1.2 develop environment

We use Mac OSX and Windows 10 to develop the program, using Flex to do the lexical analysis, and Bison (YACC) to do the syntax analysis.

Since we have learned about MIPS in Computer Organization and Computer Architecture, we use our most familiar assemble language: MIPS as our destination language.

There are many MIPS simulators, SPIM, MARS, to name a few. We here choose MARS as our MIPS simulator and run the generated MIPS code in the simulator. The result would be shown in the last part of our report.

2. Purpose and needs

To be more specific, in this experiment, we need to design and implement a compiler system for Pascal. We can learn about the process of construct a basic compiler, including lexical analysis, syntax analysis, sematic analysis, code generation, as well as the error recovery and code optimization.

To realize this, we need to implement some basic data structures learned in our previous course, for example, syntax tree and symbol table.

Then we would need to talk about the basic syntax rules of Pascal. Those rules would be depict in detail in the next few chapters, and we here just omit them.

The generated code should be MIPS, and the code should be run in the simulator, just as mentioned before.

To implement the system, we need to get an overall look of the system. Here I would like to talk about the processed, as shown below.

3. Module design

3.1 Lex

In this procedure, we use Lex to convert the words into tokens. The Lex input file is scanner.l. The general format of this Lex input file is shown below.

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

The first section called definitions contains any C doe that must be inserted external to any function, and names for regular expressions.

The second section called rules consists of a sequence of regular expressions followed by the C code that is to be executed when the corresponding regular expression is matched.

In this section we define the rules of different types of words like digit, letters, symbols and other values or keywords using in the pascal language. It is the most important part of the Lex input file.

The third section called auxiliary routines contains C code for any auxiliary routines that are called in the second section and not defined elsewhere. This section also includes a main program.

3.2 YACC

In this procedure, we use Yacc to generate a parser. The Yacc input file is parser.y. The general format of this Yacc input file is shown below.

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

The first section called definitions contains information about the token, data types and grammar rules that Yacc needs to build the parser. It also includes any C code that must go directly into the output file at its beginning.

The second section called rules contains grammar rules in a modified BNF form, together with actions in C codes that are to be executed whenever the associate d grammar rule is recognized.

In this section we check and modify a small part of the grammar of pascal language to make it clear, please refer to the file for details.

The third section called auxiliary routines contains procedure and functions = declarations that may not be available otherwise through #include files and that are needed to complete the parser and/or the compiler.

There is a tips for the storage system of a tree node that must be mentioned. Since different kinds of program instructions have different numbers of children, in the process of syntax tree construction, we optimize the storage system of a tree node. For the instructions that may have many children, for example, declaration-list, the left-most child and the sibling child of each are recorded, rather than storing all of them in their parent node.

3.3 Symbol table

There are mainly 3 data structure in the symbol table, since we need to store all these constants, all the variables and their names, as well as all the scopes of a variable.

First of all, we use “identifier” to store the variables, which includes the name, the line number, the value of the variable (there are a lot of many other fields in the

structure “identifier”, just as shown below).

```
struct identifier {
	std::string name;
	int lineno;
	Type type;
	simpleType stype;
	std::vector<std::string> *params;
	union {
		int i;
		float f;
		char c;
		std::string *s;
		SysCon syscon;
		identifier *idref;
		nameList *e;
		range *r;
		idrange *idr;
		array_typeDecl *array;
		record_typeDecl *record;
		program *prog;
		procDecl *proc;
		funcDecl *func;
	} value;
	int offset;
	int scope;
	bool constant;
	bool typeDefn;
	bool passByReference;
};
```

For those 3 stuff, we need to add some basic functions to support the operations on Symbol table. Such as insert a scope into the vector, look up for a scope of a identifier, add a constant into the set, and so on.

For the constants, we use a set to store the constants, which makes it easier to look up for the constant, and also avoid repetition of 2 constants.

```
set<identifier *> const
```

We can add and print the constants using the following functions:

```
void addConst(myPascal::identifier *id) {
		const_p.insert(id);
```

```
        }

void symbolTable::printConst() {
    for(set<identifier *>::iterator iter = const_p.begin(); iter!=const_p.end(); ++iter)
{
        printf("constant %s, ", (*iter)->name.c_str());
        switch((*iter)->stype) {
        case integer: printf("%d", (*iter)->value.i); break;
        case real: printf("%f", (*iter)->value.f); break;
        case char: printf("%c", (*iter)->value.c); break;
        case string: printf("%s", (*iter)->value.s->c_str()); break;
        }
        printf("\n");
    }
}
```

We use the vector to store the scope of a variable as well as the scope of its parent.

```
vector<struct scope>
```

We use the following function to do looking up on a scope, and print it out.

```
int symbolTable::lookUpScope(program *id) {
    for(int i=0; i<scope.size(); ++i)
        if(scope.at(i).choice==1&&
    scope.at(i).type.type1->child1->child1->name==id->child1->child1->name) {
            return i;
    }
    return -1;
}

void symbolTable::printScope() {
    for(int i=0; i<scope.size(); ++i) {
        printf("scope %d -> ", i);
        scope[i].print();
    }
}
```

For the correspondence of the name of a variable and its identifier, we use a map to store them.

```
map<string, vector<identifier *>>
```

We can do the add, delete, and find operation on this map. When we add a new identifier, we would insert an element in the map; and when an identifier is out of the

scope, we should do the delete operation on the map.

3.4 Sematic analysis

This part is about sematic analysis, which is used to analysis the correctness of the input code.

If the part to be checked is consisted of several child parts, such as a routine, a program, we do the sematic check for all its children.

For example, a routine is consisted of 2 parts: HEAD and BODY. The sematic check for a routine is shown below:

```
bool routine::semanticCheck(int scope) {
        bool temp1 = child1->semanticCheck(scope);
        bool temp2 = child2->semanticCheck(scope);
        return temp1&&temp2;
    }
```

As for the routine head, since we can learn from the Pascal syntax that the routine head is consisted of 5 parts: const part, type part, label part, routine part, and var part. So the definition of the sematic check of routine head is shown below:

```
    bool routine_head::semanticCheck(int scope) {
        bool temp1 = child1->semanticCheck(scope);
        bool temp2 = child2->semanticCheck(scope);
        bool temp3 = child3->semanticCheck(scope);
        bool temp4 = child4->semanticCheck(scope);
        bool temp5 = child5->semanticCheck(scope);
        return temp1&&temp2&&temp3&&temp4&&temp5;
    }
```

For the const expression, we need to do the following function to do sematic check, including a check for name collision:

```
bool const_expr::semanticCheck(int scope) {
        child2->semanticCheck(child1, scope);
        bool temp1 = gtable.insertID(child1, scope);
        if(!temp1)
            reportNameCollision(child1, scope);
        gtable.addConst(child1);
        bool temp2 = next==NULL?true:next->semanticCheck(scope);
        return temp1&&temp2;
    }
```

And for the variable declaration, which is consisted of 2 parts (type declaration,

name lists), we need also do the sematic check for the 2 children.

```
bool var_decl::semanticCheck(int scope) {
        child2->semanticCheck(child1, scope, 2);//2 stands for variable definition
        bool temp1 = child1->semanticCheck(scope);
        bool temp2 = next==NULL?true:next->semanticCheck(scope);
        return temp1&&temp2;
    }
```

Also, for the procedure, the sematic check is shown below:

```
bool procedure_decl::semanticCheck(int scope) {
        int temp = gtable.lookUpScope(this);
        bool temp1 = child2->semanticCheck(temp);
        bool temp2 = next==NULL?true:next->semanticCheck(scope);
    return temp1&&temp2;
    }

    bool procedure_head::semanticCheck(procedure_decl *root, int scope) {
        child1->scope = scope;
        child1->type = pprocedure;
        child1->value.proc = root;
        child1->constant = true;
        child1->typedefinition = false;
        child1->paras = NULL;
        child1->offset = 8;
        if(!gtable.insertID(child1, scope)) {
            reportNameCollision(child1, scope);
            return false;
        }
        int temp =    gtable.insertScope(root, scope);
        return child2==NULL?true:child2->semanticCheck(child1, temp);
    }
```

For IF, ELSE, WHILE, FOR statements, the sematic check is almost the same as above.

```
bool if_stmt::semanticCheck(int scope) {
        bool temp1 = child1->semanticCheck(scope);
        bool temp2 = child2==NULL?true:child2->semanticCheck(scope);
        bool temp3 = child3==NULL?true:child3->semanticCheck(scope);
        return temp1&&temp2&&temp3;
    }
```

```
    bool else_clause::semanticCheck(int scope) {
        return choice1==0?true:child1->semanticCheck(scope);
    }
bool while_stmt::semanticCheck(int scope) {
        bool temp1 = child1==NULL?true:child1->semanticCheck(scope);
        bool temp2 = child2==NULL?true:child2->semanticCheck(scope);
        return temp1&&temp2;
    }
bool for_stmt::semanticCheck(int scope) {
        bool temp1 = gtable.lookUp(child1, scope)!=NULL;
        bool temp2 = child2==NULL?true:child2->semanticCheck(scope);
        bool temp3 = child4==NULL?true:child4->semanticCheck(scope);
        bool temp4 = child5==NULL?true:child5->semanticCheck(scope);
        if(!temp1)
            reportNameMissing(child1, scope);
        if(!temp2||!temp3) return false;
        temp2 = child2==NULL?true:

    (child2->type==ssimple_type&&(child2->value.stype==iinteger||child2->value.
stype==cchar));
        temp3 = child4==NULL?true:

    (child4->type==ssimple_type&&(child4->value.stype==iinteger||child4->value.
stype==cchar));
        return temp1&&temp2&&temp3&&temp4&&typeEqual(child2, child4);
    }
```

Apart from these functions, we have implemented the other functions for sematic check which are much similar to the generators shown in above, for simplicity, these part of codes and analysis are omitted here. For the detailed implementations you can refer to the source codes in the appendix file. In the following part, we will list the remaining functions.

function name
`program::semanticCheck(int scope)`
`bool program_head::semanticCheck(program *root, int scope)`
`bool routine::semanticCheck(int scope)`
`bool sub_routine::semanticCheck(int scope)`

bool routine_head::semanticCheck(int scope)
bool const_part::semanticCheck(int scope)
bool const_expr::semanticCheck(int scope)
bool const_value::semanticCheck(identifier *id, int scope)
bool type_part::semanticCheck(int scope)
bool type_definition::semanticCheck(int scope)
bool type_decl::semanticCheck(identifier *id, int scope, int type)
bool simple_type_decl::semanticCheck(identifier *id, int scope, int type)
bool type_decl::semanticCheck(name_list *ids, int scope, int type)
bool array_type_decl::semanticCheck(identifier *id, int scope, int type)
bool range::semanticCheck()
bool name_list::semanticCheck(int scope)
bool var_part::semanticCheck(int scope)
bool var_decl::semanticCheck(int scope)
bool routine_part::semanticCheck(int scope)
bool function_decl::nameCheck(int scope)
void function_decl::reversePosition(int scope)
bool function_decl::semanticCheck(int scope)
bool function_head::semanticCheck(function_decl *root, int scope)
bool procedure_decl::semanticCheck(int scope)
bool procedure_head::semanticCheck(procedure_decl *root, int scope)
bool parameters::semanticCheck(identifier *root, int scope)
bool para_type_list::semanticCheck(identifier *root, int scope)
bool routine_body::semanticCheck(int scope)
bool assign_stmt::semanticCheck(int scope)
bool proc_stmt::semanticCheck(int scope)
bool if_stmt::semanticCheck(int scope)
bool else_clause::semanticCheck(int scope)
bool repeat_stmt::semanticCheck(int scope)
bool while_stmt::semanticCheck(int scope)
bool for_stmt::semanticCheck(int scope)
bool case_stmt::semanticCheck(int scope)
bool case_expr::semanticCheck(int scope)
bool expression::semanticCheck(int scope)

bool expr::semanticCheck(int scope)
bool term::semanticCheck(int scope)
bool factor::semanticCheck(int scope)
bool stmt::semanticCheck(int scope)

3.5 Code generation

This part is about code generation. The mid-level code input to this module will be processed to generate the assembly codes. Below is the details about our implementations.

3.5.1 program head generator

The first part is to generate the program head.

```
string progHead::assembleGenerator()
{
	string res;
	char temp[30];
	sprintf(temp, "%d", currentScope);
	string label = child1->name + string(temp);
	res += label + ":\n";
	res += "sw $ra, 0($fp)\n";
	return res;
}
```

This module is used to generate the program head, temp is used to store the scope number. And child->name is the program name(Usually at the beginning of a fragment of PASCAL source code.). And we should store the return address in $ra, which is used when RETURN occurred.

3.5.2 routine generator

```
string routine::assembleGenerator()
{
	string res;
	res += child2->assembleGenerator();
	res += "addi $sp, $sp, 100\n";
	res += "addi $v0, $zero, 17\n";
	res += "syscall\n";
	res += child1->assembleGenerator();
	return res;
}
```

In this part, we are going to generate the assembly code part. The first step is to recursively call the child-part's generator function, and then we should clear the stack, so we append "addi $sp, $sp, 100" to the result. After clear the stack, we should exit the routine, so we juest add 17 to $v0. Finally, we have generated this node's codes, So, we recursively call the right child.

3.5.3 routine head generator

```
string routHead::assembleGenerator() {
	string res;
	res += child1->assembleGenerator();
	res += child2->assembleGenerator();
	res += child3->assembleGenerator();
	res += child4->assembleGenerator();
	res += child5->assembleGenerator();
	return res;
}
```

This is to generate the routine head part's assembly codes. This can be seen as the root node of this part. And we can recursively call the child and combine the code generated by the child nodes together.

3.5.4 function declaration

```
string funcDecl::assembleGenerator() {
        string res;
        int originalScope = currentScope;
        res += child1->assembleGenerator();
        currentScope = gtable.lookUpScope(this);
        res += child2->assembleGenerator();
        currentScope = originalScope;
        return res;
}
```

This is to generate the function declaration part. Function declaration part has two child, the first child is for the function calling preparation. Such as we should generate the assembly codes for parameter pass in and result return.

3.5.5 for-loop

```
string forStmt::assembleGenerator() {
        string res;
        char temp[30];

        sbCounter++;
        string forAddr1 = "for_start_" + to_string(sbCounter);
        string forAddr2 = "for_end_" + to_string(sbCounter);
....
....
(Too many lines…)
....
....
        res += "sw $v0, " + offset + "($t0)\n";
        res += "addi $sp, $sp, 8\n";
        return res;
}
```

Here we are dealing with the for-loop part, what we should generate are: condition judgement, stack operations and jump. Before we enter the for-loop we should judge the condition, if not satisfied, we should just skip, or we should enter the for-loop. And after each iteration, we should execute the third part of for-loop (maybe make increment to indicator variables), and then judge the conditions.

Apart from these generators, we have implemented the other generators which are much similar to the generators shown in above, for simplicity, these part of codes and analysis are omitted here. For the detailed implementations you can refer to the source codes in the appendix file. In the following part of Code generation section, we will list the remaining generators.

Generator name	Remarks
string lbP::assembleGenerator()	For generating labels.
string consP::assembleGenerator()	For const part.
string consExp::assembleGenerator()	For const expression. i.e. const a:=1;
string sysCon::assembleGenerator()	For system const value part.
string typeP::assembleGenerator()	For type part.
string typeDef::assembleGenerator()	For type define part.
string typeDecl::assembleGenerator()	For type declaration part.
string sysType::assembleGenerator()	For system preserved part.
string range::assembleGenerator()	For range type.
string simple_typeDecl::assembleGenerator()	For type simple type declaration. i.e. integer.
string idrange::assembleGenerator()	For id range.
string array_typeDecl::assembleGenerator()	For array type declaration.
string record_typeDecl::assembleGenerator()	For record type declaration.
string fieldlDecl::assembleGenerator()	For field declaration.
string nameList::assembleGenerator()	For name list.
string varP::assembleGenerator()	For variable part. i.e. var a
string varDecl::assembleGenerator()	For variable declaration part.
string routP::assembleGenerator()	For routine part.
string funcDecl::assembleGenerator()	For function declaration.
string funcHead::assembleGenerator()	For function head.
string procDecl::assembleGenerator()	For procedure declaration.
string procHead::assembleGenerator()	For procedure head.
string parameters::assembleGenerator()	For parameters. i.e. passed into function or procedure.
string paraTypeList::assembleGenerator()	For parameter type list part.
string subRout::assembleGenerator()	For sub-routine part.
string routBody::assembleGenerator()	For routine part.
string sysProc::assembleGenerator(simpleType choice1)	For system procedure.
string sysFunc::assembleGenerator()	For system function. Such as print_str, print_int and so on.
string compStmt::assembleGenerator()	For computation statement.

`string stmt::assembleGenerator()`	For statement.
`string nlbStmt::assembleGenerator()`	For NLB statement.
`string assignStmt::assembleGenerator()`	For assignment statement.
`string procStmt::assembleGenerator()`	For procedure statement.
`string expression::assembleGenerator()`	For expression.
`string repeatStmt::assembleGenerator()`	For repeat statement.
`string whileStmt::assembleGenerator()`	For while statement.
`string forStmt::assembleGenerator()`	For for statement.
`string factor::assembleGenerator()`	For factor statement.
`std::string ifStmt::assembleGenerator()`	For if statement.
`std::string elsePart::assembleGenerator()`	For else part of if statement.
`std::string gotoStmt::assembleGenerator()`	For goto statement.
`std::string caseStmt::assembleGenerator()`	For case statement.
`std::string caseExp::assembleGenerator()`	For case expression. Much similar to the expression in if-clause.
`std::string expr::assembleGenerator()`	For the expression.
`std::string term::assembleGenerator()`	For the term expression.

3.6 Code optimization

Dead code elimination

In compiler theory, dead code elimination is a compiler optimization to remove code which doesn't affect the program results. Removing such code has two benefits: 1. It shrinks program size, an important consideration is some contexts, and it allows the running program to avoid executing irrelevant operations, which reduces its running time. Dead code includes code that can never be executed, and code that only affects dead variables.

Below is a sample of dead in PASCAL:

```
function foo()
var
	a,b,c:integer;
begin
	a:=24;
	b:=25;
	c:=a<<2;
	exit(c);
	b:=24;
	exit(0);
end;
```

There are some dead codes in the source code shown in the table. The first part is that we make an assignment to the dead variable b. And we have two returns, so, the codes after the first return will never be executed, that is, this part is also dead codes.

That means codes shown in the above table is equivalent to:

```
function foo()
var
	a,b,c:integer;
begin
	a:=24;
	c:=a<<2;
	exit(c);
end;
```

However, it's not easy to do optimization on the PASCAL source code directly. Alternatively, we do optimization after the generation of the mid-level codes.

Integer divide optimization

This is also very important in complier optimizations, since most of programmers are tend to use division instead of bit-wise operations which is much faster. A good compiler can always fix these issues. The code we present in the following table is not in an efficient way:

```
function foo()
var
		a:integer;
begin
		a:=a*2;
```

```
        exit(a);
end
```

So, after optimized, the code will be like:

```
function foo()
var
        a:integer;
begin
        a:=a<<1;
        exit(a);
end
```

If optimization

The conditional expression of an IF statement is known to be true in the then-clause and false in the else-clause, and this information can be used to simplify nested IF and other statements. In addition, two adjacent IF statements with the same conditional expressions can be combined into one IF statement.

```
function foo()
begin
if (exp)
begin
    foobar(1);
    if(exp)
begin
     foobar(2);
end
     foobar(3);
     exit(0);
end
end
```

The code in the above table should be optimized, since the IF inside the first IF is meaningless. If the condition 'EXP' is satisfied, then the codes in the IF will definitely be executed, or it won't.

So, below is the code after optimization:

```
function foo()
begin
if (exp)
begin
    foobar(1);
     foobar(2);
    foobar(3);
     exit(0);
end
end
```

3.7 Error recovery

Case1: type mismatch

```
program testErr1;
var a:integer;
b:real;
c:real;
begin
a:=1;
b:=a;
a:=2
c:=a;
end.
```

```
parse complete
error at LINE:: 7 , between real and integer -> type  mismatch!!!.
error at LINE:: 9 , between real and integer -> type  mismatch!!!.
semantic error.
```

Case2: Not found

```
program testErr2;
var a:integer;
begin
a:=1;
b:=a;
c:=a;
end.
```

```
parse complete
error at LINE:: 5 , b is not defined!!!.
error at LINE:: 6 , c is not defined!!!.
semantic error.
```

Case3: Missing ‘;’ or end

```
program testErr3;
var a:integer
begin
a:=1
a:=2;
end.
```

```
syntax error occurred at line 3 : Possibly missing ';'
syntax error occurred at line 5 : Possibly missing ';'
parse complete
semantic check complete.
.macro print_int (%x)
        li $v0, 1
        add $a0, $zero, %x
        syscall
.end_macro

.macro print_char (%x)
        li $v0, 11
        add $a0, $zero, %x
        syscall
.end_macro

.macro print_str (%str)
        .data
                myLabel: .asciiz %str

        .text
        li $v0, 4
        la $a0, myLabel
        syscall
```

We can see from the syntax error information, this is handled in the YACC phase.

Case4: redefinition

```
program testErr4;
var
    a:integer;
    a:real;
    b:real;
    b:integer;
begin
a:=1
```

a:=2; end.

```
parse complete
Find name conflict!!!, id = a, scope = 1; prev_id = a, prev_scope = 1!!!
error at LINE:: 3 , a was already defined at LINE:: 2.!!!
Find name conflict!!!, id = b, scope = 1; prev_id = b, prev_scope = 1!!!
error at LINE:: 5 , b was already defined at LINE:: 4.!!!
semantic error.
```

4. Test cases and results

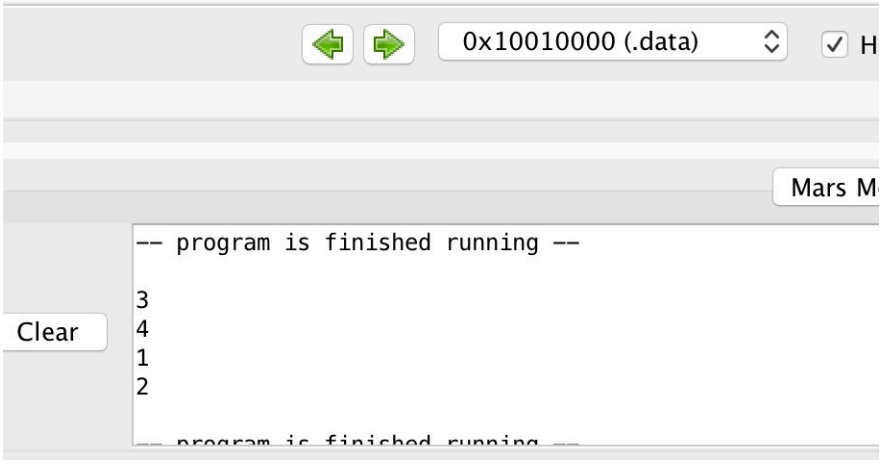
Note: As there too many lines of assembly codes, so, for simplicity, we just put up the head and tail parts of assembly codes. For the full version of test codes(in PASCAL or the corresponding assembly codes can be found in the appendix file.)

4.1 Test for basic operation(+,-,*,/,&,|….)

program basic;
var x,y:integer;
a,b: integer
begin
x:=1;
y:=2;
a:=x+y;
b:=y*y;
writeln(a);
writeln(b);
a:=y-x;
b:=b/y;
writeln(a);
writeln(b);
end.
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	basic0:
	sw $ra, 0($fp)
	addi $v0, $zero, 1
	add $t0, $zero, $fp
	addi $t0, $t0, 8
	sw $v0, 0($t0)
	addi $v0, $zero, 2
	add $t0, $zero, $fp
	addi $t0, $t0, 12
	sw $v0, 0($t0)
	add $t0, $zero, $fp
	….(Too many lines, omitted here)
	
	add $t0, $zero, $v0
	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	add $t0, $zero, $fp
	lw $v0, 20($t0)
	add $t0, $zero, $v0

	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test **the basic operations**. Below is the result of executing the assembly codes.

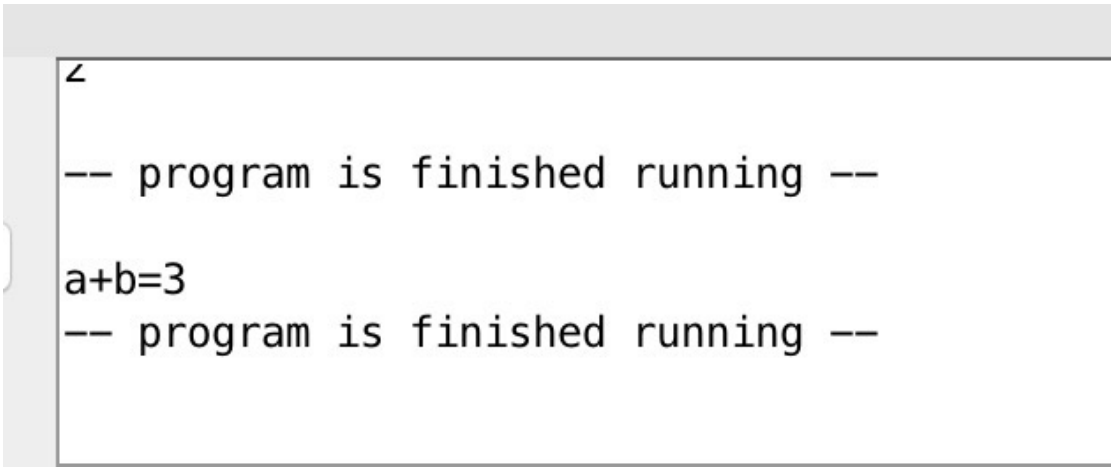


4.2 test for IF

program if2;
var a,b:integer;
begin
a:=1;
b:=2;
if a+b>3 then
write("a+b>3")
else if a+b<3 then
write("a+b<3")
else
write("a+b=3");
end.
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	if20:
	sw $ra, 0($fp)
	addi $v0, $zero, 1
	add $t0, $zero, $fp
	addi $t0, $t0, 8
	sw $v0, 0($t0)
	addi $v0, $zero, 2
	add $t0, $zero, $fp
	…
	….
	beq $v0, $zero, else_2
	print_str("a+b<3")
	j endif_2
	else_2:
	print_str("a+b=3")

	endif_2:
	endif_1:
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test the **IF**. Below is the result of executing the assembly codes.

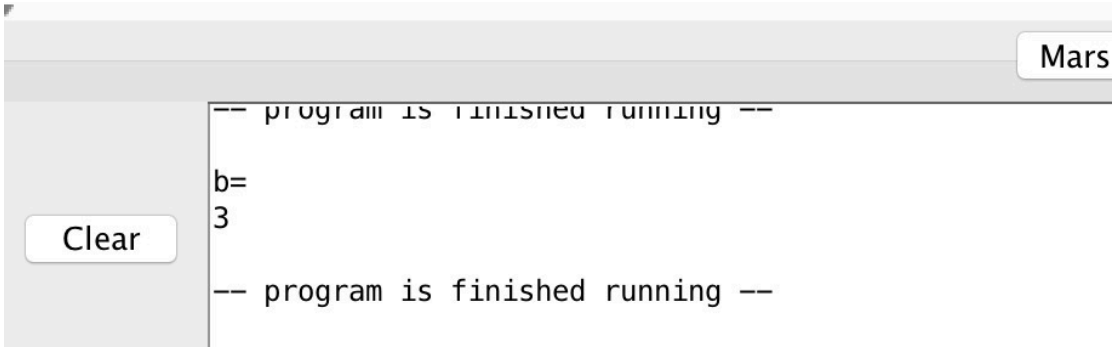


4.3 test for case

program Testcase;
var a, b:integer;
begin
a:=3;
case (a) of
1:b:=1;
2:b:=2;
3:b:=3;
4:b:=4;
end;
writeln("b=");
writeln(b);
end.
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	testcase0:
	sw $ra, 0($fp)
	addi $v0, $zero, 3
	add $t0, $zero, $fp
	addi $t0, $t0, 8
	sw $v0, 0($t0)
	add $t0, $zero, $fp
	lw $v0, 8($t0)
	addi $t1, $zero, 1
	bne $v0, $t1 nextcaseaddr_2
	addi $v0, $zero, 1
	add $t0, $zero, $fp
	addi $t0, $t0, 12
	…

	…
	lw $v0, 12($t0)
	add $t0, $zero, $v0
	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test the **CASE**. Below is the result of executing the assembly codes.

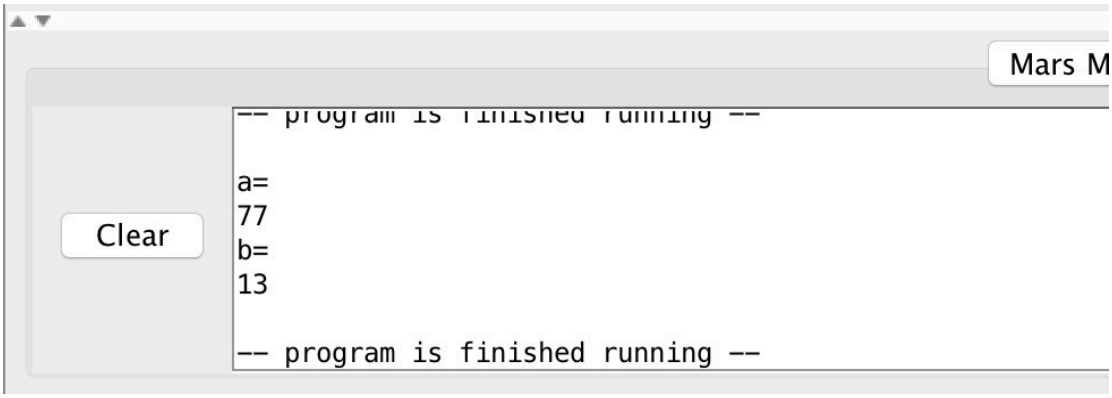


4.4 test for FOR

program testFor;
var a,b:integer;
c: integer;
begin
b:=2;
for c := 0 to 10 do
begin
a := a+b;
b := b+1;
end
writeln("a=")
writeln(a);
writeln("b=")
writeln(b)
end.
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	testfor0:
	sw $ra, 0($fp)
	addi $v0, $zero, 2
	add $t0, $zero, $fp
	addi $t0, $t0, 12
	sw $v0, 0($t0)
	add $t0, $zero, $fp
	addi $sp, $sp, -4
	sw $t0, 0($sp)
	addi $v0, $zero, 10
	addi $v0, $v0, 1
	addi $sp, $sp, -4
	…

	…
	addi $t0, $zero, 10
	print_char($t0)
	add $t0, $zero, $fp
	lw $v0, 12($t0)
	add $t0, $zero, $v0
	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test the **FOR**. Below is the result of executing the assembly codes.

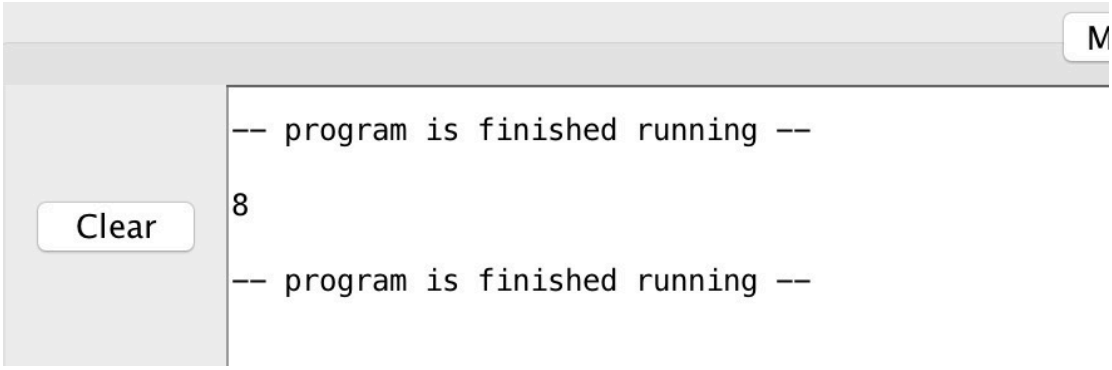


4.5 test for WHILE

program testWhile;
var a,b:integer;
begin
a:=1;
b:=0;
while b < 3 do
begin
a := a*2;
b := b+1;
end
writeln(a);
end.
	slt $v0, $v1, $v0
	addi $sp, $sp, 4
	beq $v0, $zero, while_end_1
	add $t0, $zero, $fp
	lw $v0, 8($t0)
	addi $sp, $sp, -4
	sw $v0, 0($sp)
	addi $v0, $zero, 2
	lw $t1, 0($sp)
	mult $t1, $v0

	mflo $v0
	addi $sp, $sp, 4
	add $t0, $zero, $fp
	addi $t0, $t0, 8
	sw $v0, 0($t0)
	....
	....
	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test the **WHILE**. Below is the result of executing the assembly codes.

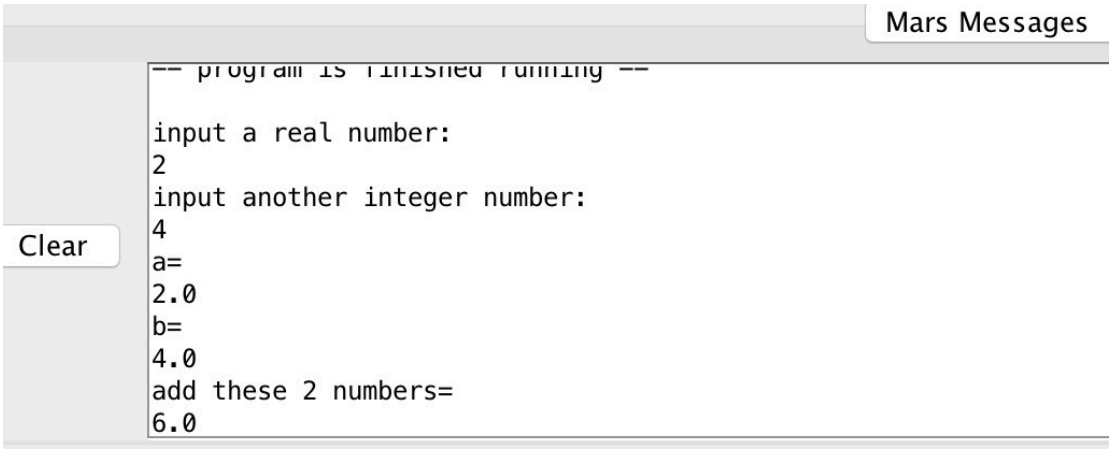


4.6 test for INPUT

program testInput;
var a, b, c:real;
begin
writeln("input a real number:");
read(a);
writeln("input another integer number:");
read(b);
c:=a+b;
writeln("a=");
writeln(a);
writeln("b=");
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	testinput0:
	sw $ra, 0($fp)
	print_str("input a real number:")
	addi $t0, $zero, 10
	print_char($t0)
	add $t0, $zero, $fp
	addi $t0, $t0, 8
	li $v0, 6

writeln(b);
writeln("add these 2 numbers=");
writeln(c);
end.
	syscall
	swc1 $f0, 0($t0)
	print_str("input another integer number:")
	addi $t0, $zero, 10
	print_char($t0)
	…
	…
	print_str("add these 2 numbers=")
	addi $t0, $zero, 10
	print_char($t0)
	add $t0, $zero, $fp
	lwc1 $f0, 16($t0)
	add.s $f12, $f31, $f0

This program is used to test the **INPUT**. Below is the result of executing the assembly codes.

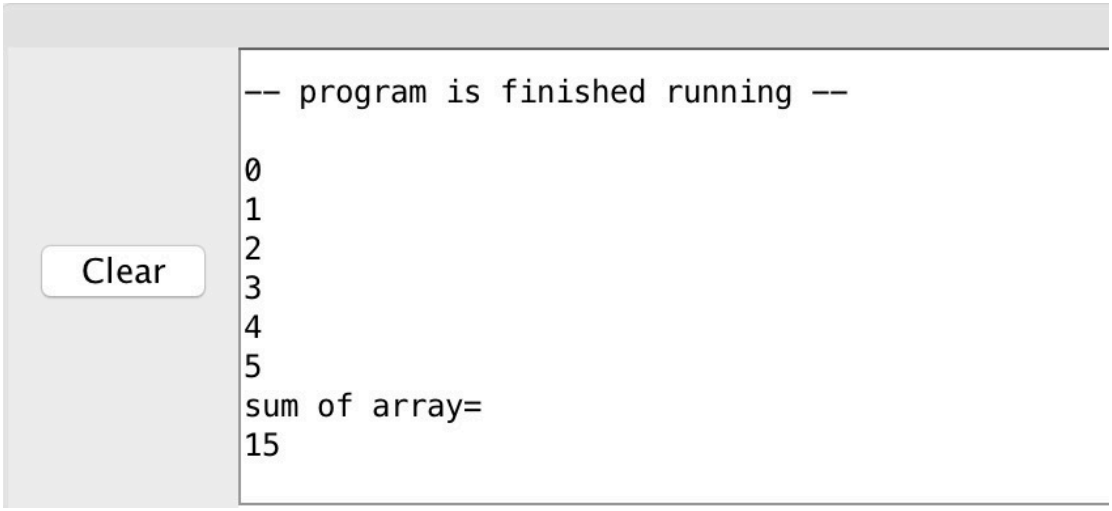


4.7 test for Array

program testArray;
var a: array [0..5] of integer; b: integer;
i: integer;
begin
b:=0;
for i := 0 to 5 do a[i] := i;
for i := 0 to 5 do b := b + a[i];
for i := 0 to 5 do writeln(a[i]);
writeln("sum of array=");
writeln(b);
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	addi $sp, $sp, -24
	sw $sp, 8($fp)
	testarray0:
	sw $ra, 0($fp)
	addi $v0, $zero, 0
	add $t0, $zero, $fp
	addi $t0, $t0, 12

end.	….
	add $t0, $zero, $fp
	lw $v0, 12($t0)
	add $t0, $zero, $v0
	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test the **ARRAY**. Below is the result of executing the assembly codes.

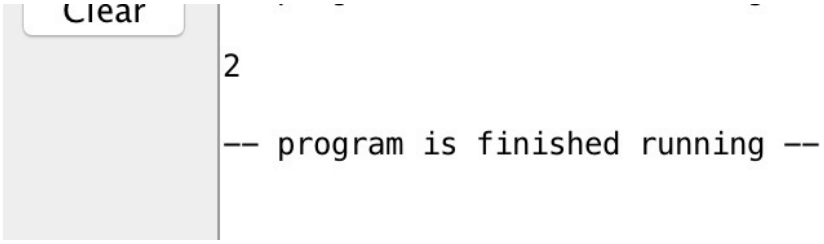


4.8 test for GOTO

program TestGoto;
var x: integer;
begin
goto 1;
x := 1;
1: x := 2;
writeln(x);
end.
	addi $sp, $sp, -100
	add $fp, $zero, $sp
	testgoto0:
	sw $ra, 0($fp)
	j Label_1
	addi $v0, $zero, 1
	add $t0, $zero, $fp
	addi $t0, $t0, 8

	sw $v0, 0($t0)
	Label_1:
	addi $v0, $zero, 2
	add $t0, $zero, $fp
	addi $t0, $t0, 8
	sw $v0, 0($t0)
	add $t0, $zero, $fp
	lw $v0, 8($t0)
	add $t0, $zero, $v0
	print_int($t0)
	addi $t0, $zero, 10
	print_char($t0)
	addi $sp, $sp, 100
	addi $v0, $zero, 17
	syscall

This program is used to test the **GOTO**. Below is the result of executing the assembly codes.

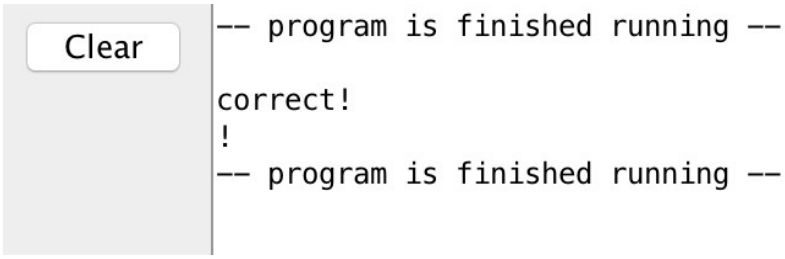


4.9 test for FUNCTION and PROCEDURE

program testFunProc;
var a, b: integer;
c: char;ss: string;
procedure proc(a: integer);
begin
case (a) of
1:writeln("correct!");
2:writeln("not correct!!");
end;
end;
 	
function func(a: integer; b:
	jal func1
	lw $fp, -4($fp)
	addi $sp, $sp, 4
	addi $sp, $sp, 16
	add $t0, $zero, $fp
	addi $t0, $t0, 16
	sw $v0, 0($t0)
	add $t0, $zero, $fp
	lw $v0, 16($t0)
	add $t0, $zero, $v0
	…

integer):char;
begin
b:=a+b;
case (b) of
2:func := '?';
3:func := '!';
end;
end;
 	
begin
a := 1;
b := 2;
proc(a);
c := func(a, b);
write(c);
end.
	…
	bne $v0, $t1 nextcaseaddr_5
	print_str("correct!")
	addi $t0, $zero, 10
	print_char($t0)
	j endcaseaddr_4
	nextcaseaddr_5:
	addi $t1, $zero, 2
	bne $v0, $t1 nextcaseaddr_6
	print_str("not correct!!")
	addi $t0, $zero, 10
	print_char($t0)
	j endcaseaddr_4
	nextcaseaddr_6:
	endcaseaddr_4:
	lw $ra, 0($fp)
	jr $ra

This program is used to test the . Below is the result of executing the assembly codes.

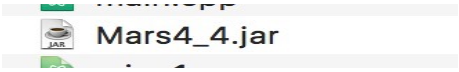


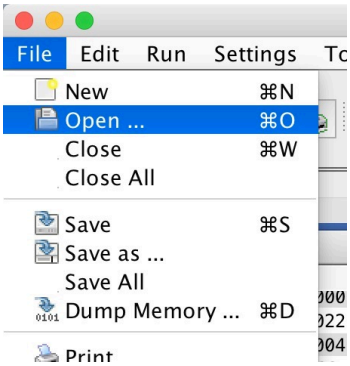
5. How to use the program

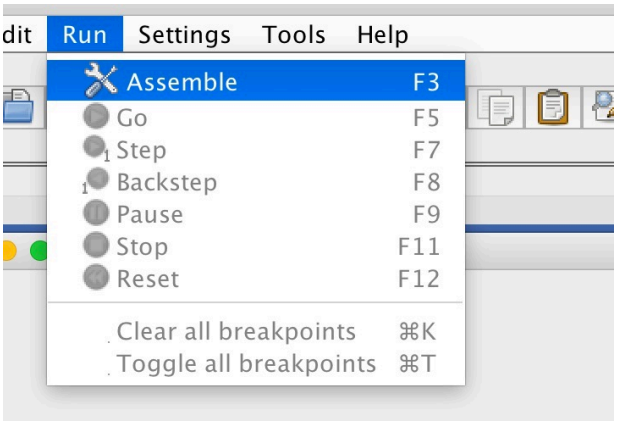
Step1: make

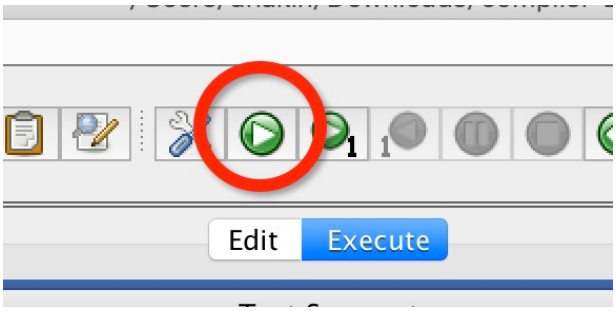
Step2: ./simplePascal < YOUR_PASCAL_CODE_FILE.pas

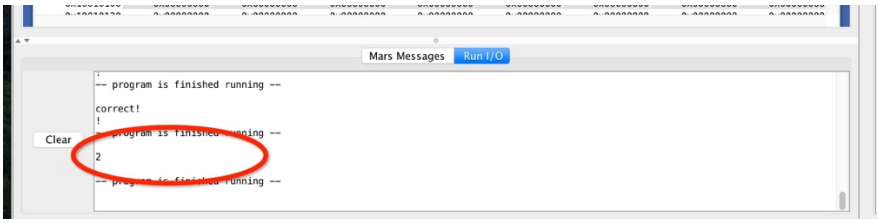
How to use Mars? See below:

1. Run 

2. 

3. 

4. 

5. 

6. Conclusion

We use the knowledge learnt in the theory course in practice during this project, and we gain much new experience and have better comprehension.

We have learnt the structure of the compiler in the theory course, but do have a specify comprehension until we finish the project. Nowadays the technic of compiler is well-developed, so we have many tools including both front-end and back-end, such

like Lex and Yacc, to support our design the project. In the procedures of lexical analysis, grammar analysis, semantic analysis and code generator, a piece of advanced language can be turned into assembly language, which is hard for people to understand, but can be executed in the computer. This procedure seems easy, but many details need to be take care of, so we really spend our time in design such program.

First, the process of the compiling need to conform to the language grammar. The designers must totally obey the rules made by the standard to construct the compiler. We can use Lex and Yacc to analysis now, which give us strong support. In these procedures, we must refer to the grammar carefully, so that the codes are more like to be free from bugs. The logic of compiler is very complex, every little mistake will have huge influence on the final results, and of course it is really hard to fix it.

Second, the process of the compiling need to check statements. In the semantic analysis, mistakes are always found, which should be recorded immediately and return to the users. For unreasonable codes, we should modify and optimize them to increase the efficiency of the program according to the grammar, under the premise that the result will not be changed.

When generating the codes, we did not use the traditional ways that generate the intermediate codes first, and then turn them into assembly codes. In the consideration of MIPS have the same structure of the intermediate codes, we directly analyze the syntax tree into MIPS and output. This solution is not likely to made mistakes, also more easy to implement. We believe it will also accelerate the program.

Compiler is an elegant and sophisticated tool. Our ability of writing codes cooperatively was fully developed in this course project, which might be the hardest project in our university life.

7. Division of labor:

Since all of the members of our group are participated in each module. And when some problems show up, we discuss them as a whole, so it's hard and unfair to give a specific division of labor. Finally, after our thorough discussion, we made an agreement that the contributions are all the same for each member.☺