浙江大学

本科实验报告

课程名称：	编译原理
姓　　名：	刘皓、陶泓羽、朱缘明
学　　院：	计算机科学与技术学院
系：	计算机科学与技术、信息安全
专　　业：	计算机科学与技术、信息安全
学　　号：	3160104994、3170102625、3170102994
指导教师：	李莹

序言

项目功能

本项目实现 C 语言子集的编译器，实现的功能和语法参考《编译器原理及实践》附录中的 C minus。

运行程序可将 C 语言编译为三地址码，以及将三地址码翻译为 MIPS32 汇编代码，并在 SPIM 模拟器上运行，通过了所有测试。

可以在 ubuntu 下终端中执行 readme.txt 中的命令完成编译运行，并将 tests/test.c 文件转化。

测试环境

MacOS 10.15.4 下

flex 2.5.35

bison (GNU Bison) 3.6.2
GCC 4.2.1
Ubuntu 16.04 下

项目分工

刘皓：语义分析，中间代码生成，测试，文档编写

陶鸿羽：词法分析，语法分析，文档编写

朱缘明：语法树的可视化，目标代码生成，文档编写

第一章 词法分析

1.1 正规表达式

含义	正规表达式	Token
letter	[a-zA-Z]	
digit	[0-9]	
numbers	([0-9])+	
floats	([0-9]+\.[0-9]+)	
注释	"/*"[/]*([^*/][/]*|[*]|[^*/])*"*/" "//"[^\n]*	
identifier	{letter}({letter}|{digit})*	ID

Constant_int	{numbers}	CONST_INT
Constant_double	{floats}	CONST_DOUBLE
string	letter?\"(\\.|[^\\\"\n])*\"	STRING

表格 1 正则表达式

保留字：

else	"else"	ELSE
if	“if”	IF
int	“int”	INT
return	“return”	RETURN
void	“void”	VOID
while	“while”	WHILE
double	“double”	DOUBLE
true	“true”	TRUE
false	“false”	FALSE
break	“break”	BREAK
bool	“bool”	BOOL
char	“char”	CHAR
or	“or”	OR_OP
and	“and”	AND_OP

表格 2 保留字

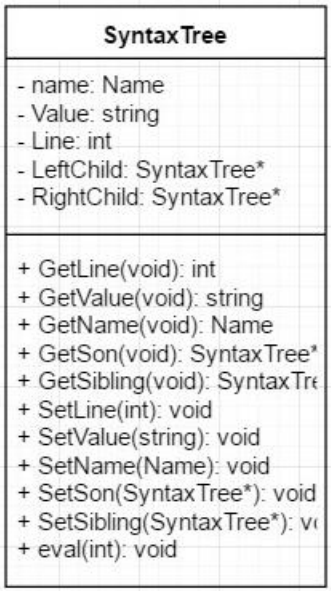
运算符：

||	"||"	OR_OP
&&	“&&”	AND_OP
--	“--”	DEC_OP
++	“++”	INC_OP
==	“==”	EQUAL
<=	“<=”	LESS_EQUAL
>=	“>=”	GREATER_EQUAL
!=	“!=”	NOT_EQUAL
+	“+”	ADD_OP
-	“-”	SUB_OP
*	“*”	MUL_OP
/	“/”	DIV_OP
<	“<”	LESS
>	“>”	GREATER
%	“%”	MOD_OP
;	“;”	SEMICOLON
,	“,”	COMMA
(	“(”	LEFT_PARE
)	“)”	RIGHT_PARE
[	“[”	LEFT_BRA
]	“]”	RIGHT_BRA
{	“{”	LEFT_BRACE

}	“}”	RIGHT_BRACE
=	“=”	ASSIGN

表格 3 运算符

1.2 数据结构



图表 1 SyntaxTree 类

SyntaxTree 有五个属性，分别是 name，Value，Line，LeftChild 和 RightChild。

其中，name 是枚举类 Name 类型的，表示该节点对应的终结符或者非终结符名称。

Value 中存放的是该节点的值，包括 int 的值，变量的名字等。由于数据类型丰富，因此统一用 string 存储，需要用的时候再转换成相应的类型。

Line 中存放的是该节点的非终结符或终结符对应的代码所在的行数，用于显示错误信息，同时也便于测试阶段的测试。

LeftChild 和 RightChild 分别放的是该节点的左儿子和右儿子。本结构采用左儿子右兄弟的结构存储，这样不用开一个 vector 存储所有的儿子信息。

五个 Get 函数和五个 Set 函数就是获取或者设置相应的属性，这是为了属性的安全。

最后一个 eval 函数用于语法树的输出。

1.3 原理以及实现

利用 lex，将输入的字符流按照前面设置的正规表达式，变为 token 流。
在分离解析出各个 token 之后，首先创建一个语法树的节点，将 token 信息（包括 name，Line）传递给构造函数。其中 Value 由 yytext 传递。这些在 lex 中创建的语法树节点，最后将成为该语法树的叶子节点。
如果 token 是注释或者是其他的空白字符，则无视它。
如果 token 是保留字，则创建相应的节点。

第二章 语法分析

2.1 文法描述

progam -> declaration_list

declaration_list -> declaration_list declaration

declaration -> var_declaration | func_definition

var_declaration -> type_specifier SEMICOLON

| type_specifier init_declarator_list SEMICOLON

init_declarator_list -> init_declaretor

| init_declarator_list COMMA init_declarator

init_declarator -> declarator

| declarator ASSIGN expression

type_specifier -> INT | VOID | DOUBLE | CHAR | BOOL

func_definition -> type_specifier declarator compound_stmt

declarator -> ID

| declarator LEFT_PARE param_list RIGHT_PARE

| declarator LEFT_PARE RIGHT_PARE

| declarator LEFT_BRA RIGHT_BRA

param_list -> param_list COMMA param

| param

param -> tpre_specifier declarator

| type_specifer

compound_stmt -> LEFT_BRACE block_item_list RIGHT_BRACE

| LEFT_BRACE RIGHT_BRACE

block_item_list -> block_item_list block_item

| block_item

block_item -> var_declaration | statement

statement -> expression_stmt

| compound_stmt

| selection_stmt

| iteration_stmt

| jump_stmt

expression_stmt -> expression SEMICOLON

| SEMICOLON

selection_stmt -> IF LEFT_PARE expression RIGHT_PARE

| IF LEFT_PARE expression RIGHT_PARE statement ELSE statement

iteration_stmt -> WHILE LEFT_PARE expression RIGHT_PARE statement

jump_stmt -> RETURN expression SEMICOLON

| RETURN SEMICOLON

| BREAK SEMICOLON

expression -> postfix_expression ASSIGN expression

| logical_or_expression

postfix_expression -> primary_expression

| postfix_expression LEFT_PARE RIGHT_PARE

| postfix_expression LEFT_PARE arg_list RIGHT_PARE

primary_expression -> ID | TRUE | FALSE | CONST_INT | CONST_DOUBLE

| LEFT_PARE expression RIGHT_PARE

logical_or_expression -> logical_and_expression

| logical_or_expression OR_OP logical_and_expression

logical_and_expression -> equality_expression

| logical_and_expression AND_OP equality_expression

equality_expression -> simple_expression

| equality_expression EQUAL simple_expression

| equality_expression NOT_EQUAL simple_expression

simple_expression -> simple_expression relop additive_expression

| additive_expression

relop -> LESS_EQUAL | LESS | GREATER | GREATER_EQUAL

additive_expression -> additive_expression addop term

| term

addop -> ADD_OP | SUB_OP

term -> term mulop factor | factor

mulop -> MUL_OP | DIV_OP | MOD_OP

factor -> postfix_expression

| INC_OP factor

| DEC_OP factor

| unaryop factor

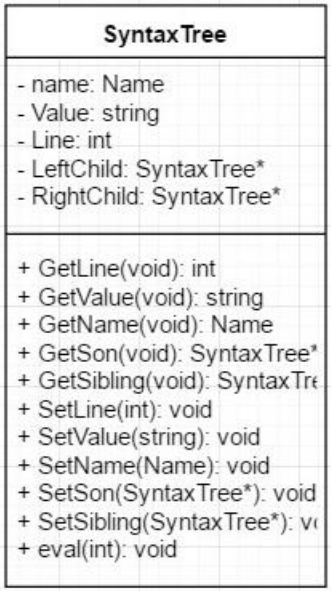
unaryop -> ADD_OP | SUB_OP

arg_list -> arg_list COMMA expression

| expression

2.2 数据结构

数据结构同词法分析，都是 SyntaxTree 这个类。不过词法分析负责的是叶子结点的创建，语法分析负责的是其余节点的创建。



图表 2 SyntaxTree 类

该部分的构造函数使用了 `cstdarg` 的可变参数。因为不同语法句需要的参数不同，我们想到了两种方法，一种是在 yacc 中首先创建参数的 vector，再传入构造函数；还有一个就是用可变参数。最终我们选择了后者。因为这样在 yacc 的部分不会过于冗长。

在传参数的时候，除了参数个数与子节点的指针，还有该非终结符在 Name 枚举类中对应的名字，用于区分不同的非终结符。

2.3 原理以及实现

利用 yacc 完成对语法的分析，得到 token 流的语法树。

所有的递归语法都是保证左递归，因为 yacc 使用的是 LALR(1)语法分析方法。

在完成当句语法的分析之后，会生成该非终结符的语法树节点。而它的子节点与子节点的兄弟则是语法右边的非终结符与终结符。

示例如下：

```
compound_stmt : LEFT_BRACE block_item_list RIGHT_BRACE{
            $$ = new SyntaxTree(T_COMPOUND_STMT,3,$1,$2,$3);
    }
    | LEFT_BRACE RIGHT_BRACE{
            $$ = new SyntaxTree(T_COMPOUND_STMT,2,$1,$2);
    }
    ;
```

第三章 语义分析

在语法分析阶段已经建立了抽象语法树，在语义分析阶段，我们对语法树进行深度优先遍历，然后同步进行语义分析与中间代码生成，过程中会对不符合定义的语法进行报错。

为了提高程序的模块性，便于多人合作，本阶段代码单独分离，而不在 lex 和 yacc 文件中。

3.1 输入文件假设

在这一阶段，我们对输入的 C 语言源代码文件作出如下假设，程序将能正确转换满足如下假设的源代码，并对不满足假设源代码的报错。

1. 没有超出第二章语法分析中定义的语法。
2. 不包含结构体和数组。
3. 不包含隐性类型转换。

4. 不包含指针和取地址。

5. 没有函数声明，只有函数定义。

3.2 中间语言定义

中间语言的形式同时参考了《编译器原理及实践》中的三地址码和本学期南京大学徐汇老师编译原理课程中给出的中间代码形式。
形式如下表所示。

语法	描述
LABEL x:	定义标签 x
FUNCTION f:	定义函数 f
x = y	赋值操作
x = y + z	加法操作
x = y - z	减法操作
x = y * z	除法操作
x = y / z	除法操作
x = y % z	取余操作
GOTO x	跳转到标签 x
IF x[relop] y GOTO z	如果 x 与 y 满足[relop]关系则跳转到标签 z
RETURN x	退出当前函数并返回 x 值
ARG x	传实参 x
x = CALL f	调用函数 f，并将返回值赋给 x
PARAM x	函数参数声明
READ x	从控制台读取 x 的值
WRITE x	向控制台打印 x 的值

表格 4 中间语言语法

3.3 类与数据结构定义

在翻译过程中，为了保存必要的信息，比如运算式的值、当前作用域中的变量名、已经定义的函数名和参数类型、当前生成的中间代码等，我们定义了一系列的类与数据结构。

函数表

记录了各种函数的名称以及对应信息。

```
map<string, funcInfo> funcTable;    //函数定义表
```

Scope

记录了一个作用域（{ }包围的代码）中的变量信息和是否是在函数中，以及发生 break 时跳转的 label。

```
struct Scope {
public:
    bool isfunc = false; //记录是否是函数作用域
    funcNode func;    //如果是函数，记录函数名
    map<string, struct varNode> varMap;       //变量的 map
    string breakLabelname;
    bool canBreak = false;
    Scope() {

    }
    Scope(string fname, Types ftype) {
        func = funcNode(fname, ftype);
        isfunc = true;
    }
};
```

函数和变量结点

储存了变量的类型和名称信息以及函数的类型和参数信息。

```
//变量节点
struct varInfo {
    string name;
    Types type;
```

```
    int num = -1;
        string condition;
};

//函数节点
struct funInfo {
    string name;              //函数名
    Types rtype;              //函数返回类型
    vector<varNode> paralist;    //记录形参列表
};
```

3.4 中间代码管理

包含了添加中间代码，根据变量和函数结点生成代码和输出所有代码的功能。

```
// 中间代码生成和优化
class InterRepresent {
public:
    InterRepresent();

    void outputIR(bool tofile = true, string filename = "intercode.txt");

    void addIR(string IR);
    string newTemp();
    string newLabel();

    string genBinaryOpIR(string temp, string op, varNode var1, varNode var2); // temp = var0 + var1
    string genAssignIR(varNode var1, varNode var2);
    string genParamIR(varNode param); // 函数定义参数 PARAM var0
    string genReturnIR(varNode var); // RETURN var0
    string genArgIR(varNode var); // 传递实参
    string genConditonIR(); // IF GOTO

    void optimize();

    vector<string> codes;
    int tempNo = 0;
    int varNo = 0;
```

```
    int labelNo = 0;
    int arrayNo = 0;
};
```

3.5 翻译模式

翻译过程在类 Translator 中定义了很多函数，对应每种 token 的翻译，这些函数相互调用，共同完成中间代码的生成，最终结果存储在 interCode 的 codes 中。

```
    void translate_SyntaxTree(SyntaxTree *node); // 从语法树的法 root 开始遍历并生成代码相当于从 Program 开始
    void translate_var_declaration(SyntaxTree *node);
    void translate_init_declarator_list(SyntaxTree *node, string type);
    void translate_init_declarator(SyntaxTree *node, string type);

    // 下面的排列是按照优先级
    varNode translate_expression(SyntaxTree *node);
    varNode translate_logical_or_expression(SyntaxTree *node); // ||
    varNode translate_logical_and_expression(SyntaxTree *node); // &&
    varNode translate_equality_expression(SyntaxTree *node); // == !=
    varNode translate_relational_expression(SyntaxTree *node); // > < >= <=
    varNode translate_additive_expression(SyntaxTree *node); // + -
    varNode translate_multiplicative_expression(SyntaxTree *node); // * /
    varNode translate_unary_expression(SyntaxTree *node); // - +
    varNode translate_postfix_expression(SyntaxTree *node); // 包含函数调用
    void translate_arg_list(SyntaxTree *node, string funcName); // 调用函数的参数
    varNode translate_primary_expression(SyntaxTree *node); // ID、字面量、(expression)

    void translate_function_definition(SyntaxTree *node);
    void translate_parameter_list(SyntaxTree *st, string funcName, Types type);
    void translate_parameter_declaration(SyntaxTree *node, string funcName);

    void translate_statement(SyntaxTree *node);
    void translate_compound_stmt(SyntaxTree *node);
    void translate_selection_stmt(SyntaxTree* node);
    void translate_iteration_stmt(SyntaxTree *node); // 只支持 while
    void translate_jump_statement(SyntaxTree* node);
```

初始化

添加 WRITE 和 READ 到函数表，然后开始对整个语法树进行翻译。

整个程序

```
// program -> declaration_list
// declaration_list -> declaration_list declaration
// declaration -> var_declaration | func_definition
void Translator::translate_SyntaxTree(SyntaxTree *node);
```

从根节点开始进行深度优先遍历，如果遇到变量声明、函数定义或者语句，则调用相应翻译函数。

变量声明

```
// var_declaration -> type_specifier; | type_specifier init_declarator
void Translator::translate_var_declaration(SyntaxTree *node);
```

如果没有 init_declarator，则忽略语句。
否则调用 translate_init_declarator_list(init_declarator_list, type);

变量声明符列表

```
// init_declarator_list -> init_declarator | init_declarator_list ,
init_declarator
void Translator::translate_init_declarator_list(SyntaxTree *node, string type);
```

递归调用自身，直到获得 init_declarator 调用 void Translator::translate_init_declarator(SyntaxTree *node, string type)再返回，然后处理“，”后的 init_declarator。

函数定义

```
// function_definition -> type_specifier declarator compound_stmt
void Translator::translate_function_definition(SyntaxTree *node);
```

分别处理返回值类型、函数声明符。

处理函数声明符的时候直接获取函数名，调用 translate_parameter_list(declarator->GetSon()->GetSibling()->GetSibling(), funcName, funcType);处理参数列表。

然后生成对应的函数作用域，储存起来。

生成相应的中间代码 FUNCTION funcName :

然后处理函数声明符中的参数，如果没有直接跳过，否则调用

```
translate_parameter_list(declarator->GetSon()->GetSibling()->GetSibling(),
funcName, funcType);
```

最后函数表中记录函数信息，弹出当前 scope。

参数列表

```
// parameter_list -> parameter_list , parameter_declaration |
parameter_declaration
void Translator::translate_parameter_list(SyntaxTree *node, string funcName, Types type);
```

递归调用自身或者调用

```
translate_parameter_declaration(node->GetSon(), funcName);
```

参数声明

```
// parameter_declaration -> type_specifier declarator
void Translator::translate_parameter_declaration(SyntaxTree *node, string funcName);
```

解析参数类型和名称，生成对应的 varInfo，然后添加到函数信息的参数表、函数作用域的变量表中。

最后生成对应的中间代码 ARG v_agr

```
  interCode.addIR(interCode.genParamIR(var));
```

语句

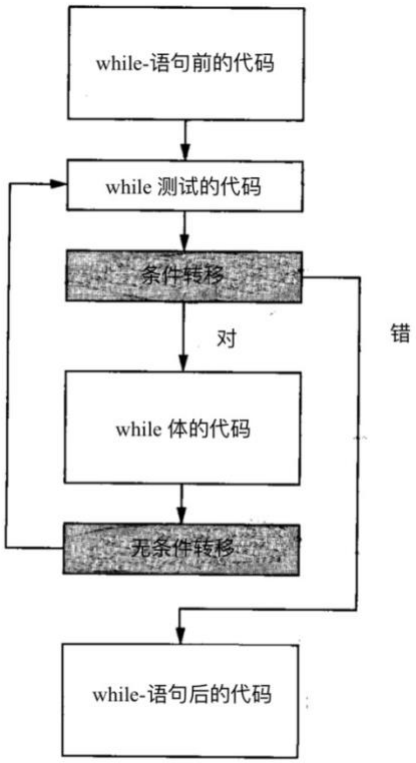
```
// statement -> compound_stmt | expression_stmt | selection_stmt |
// iteration_statement | jump_statement
void Translator::translate_statement(SyntaxTree *node);
```

编译器支持的语句分为五种，根据语句的类型分别调用相应的函数。

循环语句

循环语句只支持 while，足够表达所有的循环类型。

while 语句的典型结构如下图所示。



图表 3 while 结构

```
// iteration_stmt -> WHILE ( expression ) statement
void Translator::translate_iteration_stmt(SyntaxTree *node);
```

根据模版生成中间代码，注意进入 S 对应的语句时压入一个新的 scope，S 对应的 code 由解析 expression 的函数生成。

while（E） S

对应的模版为

LABEL Label1:

<code to evaluatue E to t1>

if t1 GOTO Label2

GOTO Label3

Label2

<code for S>

goto L1

Label Label3:

其中对于非 bool 类型的条件 E（比如 a = a + 1），将表达式值转换为 E != #0。

分支语句

分支语句只支持 if 和 if else，不支持 else if，可以通过嵌套来表达与 else if 相同的语义。

```
// selection_stmt : IF ( expression ) statement
// | IF ( expression ) statement ELSE statement
void Translator::translate_selection_stmt(SyntaxTree *node);
```

根据模版生成中间代码，注意进入 if 和 else 对应的语句时压入一个新的 scope，S 对应的 code 由解析 expression 的函数生成。

if(E) S

对应的模版为

IF E GOTO Label1

GOTO Label2

Label1:

<code for S>

Label2

if(E) S1 else S2

对应的模版为

IF E GOTO Label1

GOTO Label2

Label1:

<code for S1>

GOTO Label3

Label2:

<code for S2>

Label3:

复合语句

复合语句由多条语句构成，翻译方法与整个程序相同，是一种类似于套娃的关系。

```
// statement -> compound_stmt | expression_stmt | selection_stmt |
// iteration_statement | jump_statement
// statement 和 compound_stmt 相互套娃
void Translator::translate_compound_stmt(SyntaxTree *node) {
    translate_SyntaxTree(node);
}
```

跳转语句

```
// jump_stmt -> BREAK ; | RETURN ; | RETURN expression ;
void Translator::translate_jump_statement(SyntaxTree *node);
```

编译器支持的跳转语句包括 BREAK 和 RETURN。

对于 BREAK，寻找上一个可以 Break 的 Label 号码，然后添加中间代码 GOTO Label，如果不能 Break 则报错。

对于 RETURN，检查是否有返回值，如果没有返回值，检查函数返回类型是否为 void，如果不为 void 则报错，检查通过则添加中间代码 RETURN。

如果有返回值，检查函数返回类型，如果与函数定义不一致则报错，检查通过则添加中间代码 RETURN e_val。

表达式

表达式具有优先级，在本程序中只考虑了八层优先级，如下表所示。

优先级	运算符	结合性	描述
1	( )	左结合	括号或函数调用
2	-	右结合	取负
	+		取正
3	*	左结合	乘
	/		除
	+		加
	-		减
5	<		小于
	<=		小于或等于
	>		大于
	>=		大于或等于
	==		等于
	!=		不等于
6	&&		逻辑与
7	||		逻辑或
8	=	右结合	赋值

表格 5 表达式优先级

所以在处理时与语法保持一致，用到了 8 层套娃来处理，每一次都返回上一次生成的 varInfo 信息，使用这些函数翻译，这些函数层层调用其他函数。

```
varInfo translate_expression(SyntaxTree *node);
varInfo translate_logical_or_expression(SyntaxTree *node); // ||
varInfo translate_logical_and_expression(SyntaxTree *node); // &&
varInfo translate_equality_expression(SyntaxTree *node); // == !=
varInfo translate_relational_expression(SyntaxTree *node); // > < >= <=
varInfo translate_additive_expression(SyntaxTree *node); // + -
varInfo translate_multiplicative_expression(SyntaxTree *node); // * /
varInfo translate_unary_expression(SyntaxTree *node); // - +
varInfo translate_postfix_expression(SyntaxTree *node); // 包含函数调用
void translate_arg_list(SyntaxTree *node, string funcName); // 调用函数的参数
varInfo translate_primary_expression(SyntaxTree *node); // ID、字面量、(expression)
```

函数调用

```
// postfix_expression -> primary_expression | postfix_expression ( ) |
postfix_expression ( argument_expression_list )
// primary_expression -> IDENTIFIER | TRUE | FALSE | CONST_INT | CONST_DOUBLE
varInfo Translator::translate_postfix_expression(SyntaxTree *node);
```

函数调用的翻译在这个函数中实现，如果监测到 postfix_expression 转换到 postfix_expression ( ) 或者 postfix_expression (argument_expression_list)就开始。

对于 READ 和 WRITE 函数中间代码特殊处理，比如 READ v0 和 WRITE v1。

对于其他函数调用通过

```
translate_arg_list(argument_expression_list, funcName);
```

生成压入函数参数的中间代码，比如

ARG v_arg1

ARG v_arg2

然后生成调用函数的中间代码，过程中会检查参数信息与之前的记录是否一致。

v_res = CALL add

优化考虑

在中间代码中去除多余的中间临时变量。

t1 = v0 == t0

IF v0 == t0 GOTO label0

优化为

IF v0 == t0 GOTO label0

第四章 目标代码生成

寄存器分配

将 a0,a1,a2,a3 四个参数寄存器特定分配给函数调用时的参数，如 PARAM var0,即可将 a0 寄存器分配给 var0。

将 s[0:9]和 t[0:9]分配给剩余的变量如 temp，在变量确定释放后，需要将寄存器释放，以空出足够的空间给予新的变量。

变量记录

针对以(temp\d*)|(var\d*)描述的块，将其视作是变量，利用 python 的 re 库可以按序记录变量，从而实现寄存器分配

语句分析

中间代码生成的语句与对应的目标代码语句可以形成下表

中间代码	目标代码
LABEL label0:	label0:
GOTO label0	j label0
temp0 = #1	li $t0, 1
temp0 = temp1	move $t0, $t1
temp0 = temp1 + temp2	add $t0, $t1, $t2
temp0 = temp1 +(-) #1	addi $t0, $t1, (-)1
Temp0 = temp1 - temp2	Sub $t0, $t1, $t2

Temp 0 = temp1 * temp2	Mul $t0, $t1, $t2
Temp0 = temp1 / temp2	Div $t2, $t1 Mflo $t0
ARG temp*	Move $t*, $a* Move $a*, $(temp*)
FUNCTION * :	*:
RETURN *;	Move $v0, %s Jr $ra

READ temp*	addi $sp,$sp,-4
	sw $ra,0($sp)
	jal read
	lw $ra,0($sp)
	addi $sp,$sp,4

Write temp*	addi $sp,$sp,-4
	sw $ra,0($sp)
	jal write
	lw $ra,0($sp)
	addi $sp,$sp,4

IF temp0 == temp1 GOTO label0	beq $t0, $t1, label0

Temp0 = CALL read	addi $sp,$sp,-8
	sw $t0,0($sp)
	sw $ra,4($sp)

	jal read
	lw $a0,0($sp)
	lw $ra,4($sp)
	addi $sp,$sp,8
	move %(temp0),$v0

参数分析

在多参时，如果调用函数，需要根据 ARG 语句分别分配参数，压入栈后再调用函数，调用函数时要根据参数个数将 PARAM 后的变量分别分配 a 类寄存器，要注意参数的顺序。

算法分析

实现本次目标代码生成有几大前提

1 根据变量记录，在后续代码段没有出现的变量将不继续占用寄存器，变量数目不会大于 s 类和 t 类寄存器总和。

2 每次调用函数，参数都只选取 a 类寄存器，并且每次函数调用完毕都会释放 a 类寄存器。

3 除了调用函数参数时会使用 a 类寄存器，其余情况下 a 类寄存器不会变化。

4 在调用参数(ARG var)时，会自动将参数压入栈。

整体的算法很清晰，在寄存器分配方面，给遍历到的每一个变量分配寄存器，每有一个变量最后一次出现即解除其与寄存器的对应关系；在参数调用方面，按顺序压入栈并读取，根据参数多少调整栈的偏移；在语句翻译方面，注意记录参数的个数，由于逐行翻译，要记录参数的个数来进行跨行的两个语句之间的信息交流。

在上述前提和构想的基础上，可以发现程序必然存在如下 bug：

1 类似 while(i++)之后再没有出现 i 的 code 段，根据前提 1 可以发现 i 所对应的寄存器会在第一次循环就释放，之后循环就会出错，这个 bug 由于逻辑方面的漏洞暂时无法规避，所以在代码输入方面要新加规范，循环的参数必然要在循环体内部的最后出现。

2 多参（大于 4），由于本身的设计只考虑了 4 个 a 类寄存器来存储参数，多于六个参数的函数会导致多余的参数产生 bug。

3 在调用参数时，不能在第二个参数上对第一个参数进行变动，因为逻辑上在第二个参数读取的时候，第一个参数已经入栈，此时更改第一个参数，虽然中间代码可以得出正确结果，但是目标代码的逻辑会导致 bug。

除了这三个逻辑上的 bug 外，还有一些设计上的缺漏，如本身没有考虑数组。

不过本目标代码可以实现一些基本的 c 程的编译。

第五章 测试案例

功能测试

每个语句成分的测试案例，至少两个复杂语句组合后的测试案例。

第一个测试例子在文档中完整展示，其他请见文件夹中的 test_result。

测试复杂与常量计算

第一个测试

原代码：

```
/* This is a test file for mini C compiler*/
int fact(int n){
   int temp;
   if(n==1)
      return n;
   else{
      temp=(n*fact(n-1));
      return temp;
   }
}


int main()
{
   int result,times;
   result = 1 + 2*3;
   write(result);
   times=read();
   while(times = times - 1){
```

```
    int m = read();
    if( m > 1) {
        result=fact(m);
    }
    else {
        result = 1;
    }
    write(result);
  }
  return 0;
}
```

中间代码：

FUNCTION fact :

PARAM v0

t0 = #1

t1 = v0 == t0

IF v0 == t0 GOTO label0

GOTO label1

LABEL label0 :

RETURN v0

GOTO label2

LABEL label1 :

t2 = #1

t3 = v0 - t2

ARG t3

t4 = CALL fact

```
t5 = v0 * t4

v1 = t5

RETURN v1

LABEL label2 :

FUNCTION main :

t6 = #1

t7 = #2

t8 = #3

t9 = t7 * t8

t10 = t6 + t9

v2 = t10

WRITE v2

READ t11

v3 = t11

LABEL label3 :

t12 = #1

t13 = v3 - t12

v3 = t13

t14 = #0

IF v3 != t14 GOTO label4

GOTO label5

LABEL label4 :
```

```
READ t15

v4 = t15

t16 = #1

t17 = v4 > t16

IF v4 > t16 GOTO label6

GOTO label7

LABEL label6 :

ARG v4

t18 = CALL fact

v2 = t18

GOTO label8

LABEL label7 :

t19 = #1

v2 = t19

LABEL label8 :

WRITE v2

GOTO label3

LABEL label5 :

t20 = #0

RETURN t20
```

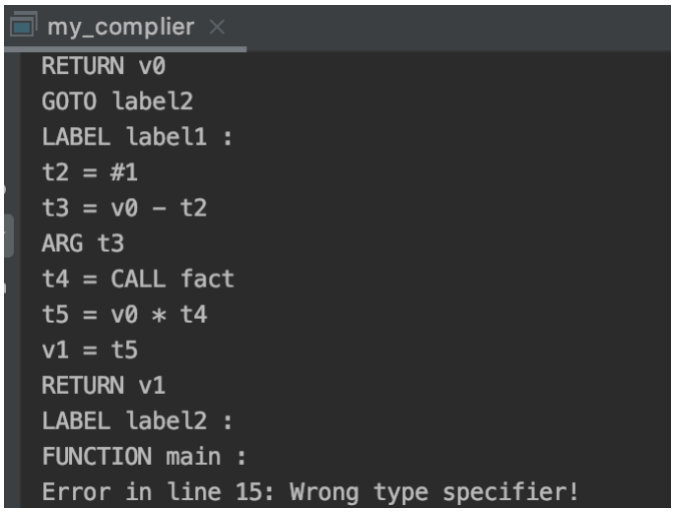
目标代码（MIPS）：

模拟器运行结果：

报错测试

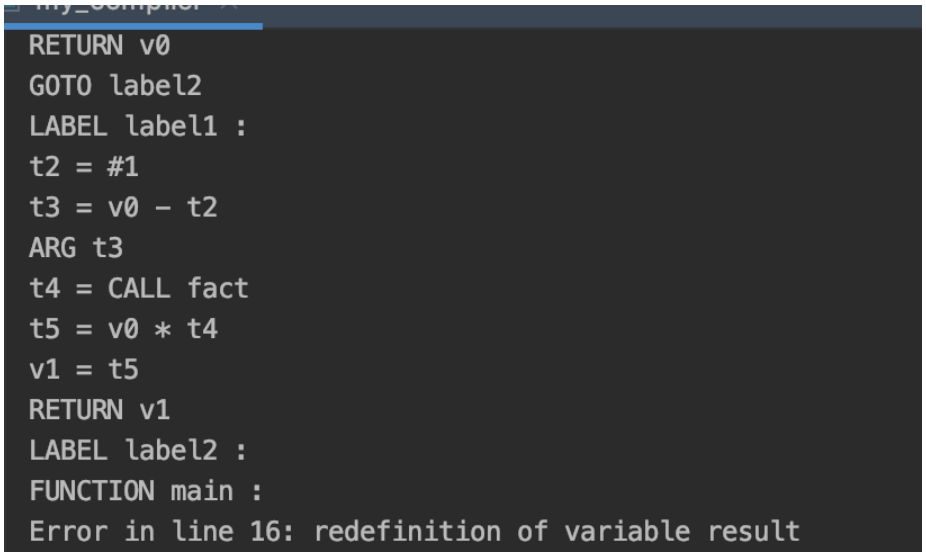
测试类型错误

```
void result,times;
```



测试重复定义错误

```
int result,times;
int result = 1 + 2*3;
```



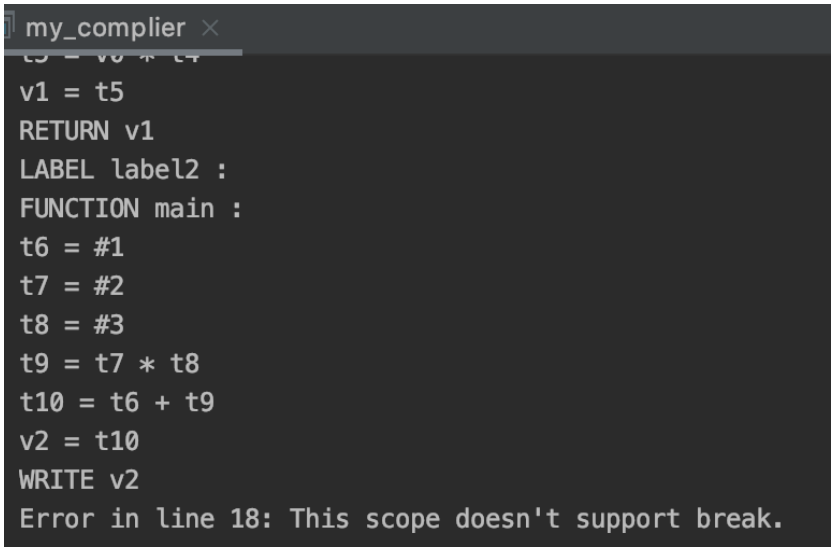
测试函数参数错误

```
        result=fact(m,m);
```

```
my_complier
IF v3 != t14 GOTO label4
GOTO label5
LABEL label4 :
READ t15
v4 = t15
t16 = #1
t17 = v4 > t16
IF v4 > t16 GOTO label6
GOTO label7
LABEL label6 :
ARG v4
ARG v4
Error in line 22: Wrong argument!
```

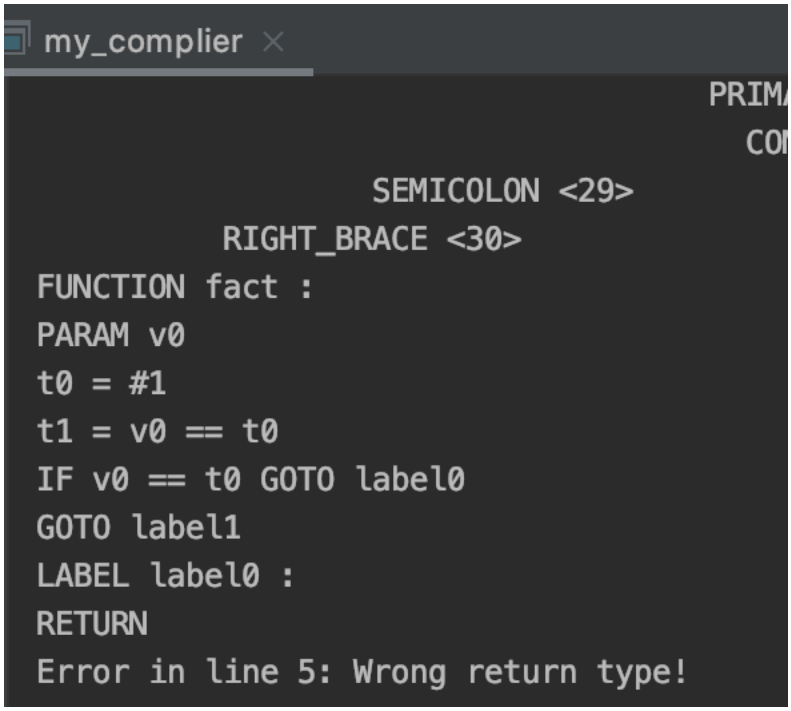
测试 break 错误

```
int main()
{
   int result,times;
   result = 1 + 2*3;
   write(result);
   break;
   times=read();
   while(times = times - 1){
      int m = read();
      if( m > 1) {
         result=fact(m);
      }
      else {
         result = 1;
      }
      write(result);
   }
   return 0;
}
```



测试 return 错误

```
int fact(int n){
   int temp;
   if(n==1)
      return;
   else{
      temp=(n*fact(n-1));
      return;
   }
}
```



下面是对于生成 mips 汇编代码和虚拟机运行的测试

测试多参

```
seeker@seeker:~/project/project(Ubuntu)$ spim
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
(spim) load "result-mulparam.asm"
(spim) run
Enter your integar:10
Enter your integar:5
Enter your integar:4
Enter your integar:2
58
(spim)
```

```
seeker@seeker:~/project/project(Ubuntu)$ cat resu
lt-mulparam.asm

.data
_prompt: .asciiz "Enter your integar:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0,4
    la $a0,_prompt
    syscall
    li $v0,5
    syscall
    jr $ra

write:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
fact:
        mul $s0,$a0,$a1
        mul $s7,$a2,$a3
        add $t8,$s0,$s7
        move $v0,$t8
        jr $ra
main:
        addi $sp,$sp,-4
        sw $ra,0($sp)
        jal read
        lw $ra,0($sp)
        addi $sp,$sp,4
        move $s0,$v0

        move $s7,$s0
        addi $sp,$sp,-4
        sw $ra,0($sp)
        jal read
        lw $ra,0($sp)
        addi $sp,$sp,4
        move $s0,$v0

        move $t8,$s0
```

seeker@seeker: ~/project/project(Ubuntu) 49x27

```
seeker@seeker:~/project/project(Ubuntu)$ cat test
2.c
int fact(int m, int n, int a, int b){
        return m*n+a*b;
}


int main()
{
    int result;
    int m = read();
    int n = read();
    int a = read();
    int b = read();
    result=fact(m, n, a, b);
    write(result);
    return 0;
}
```

测试递归

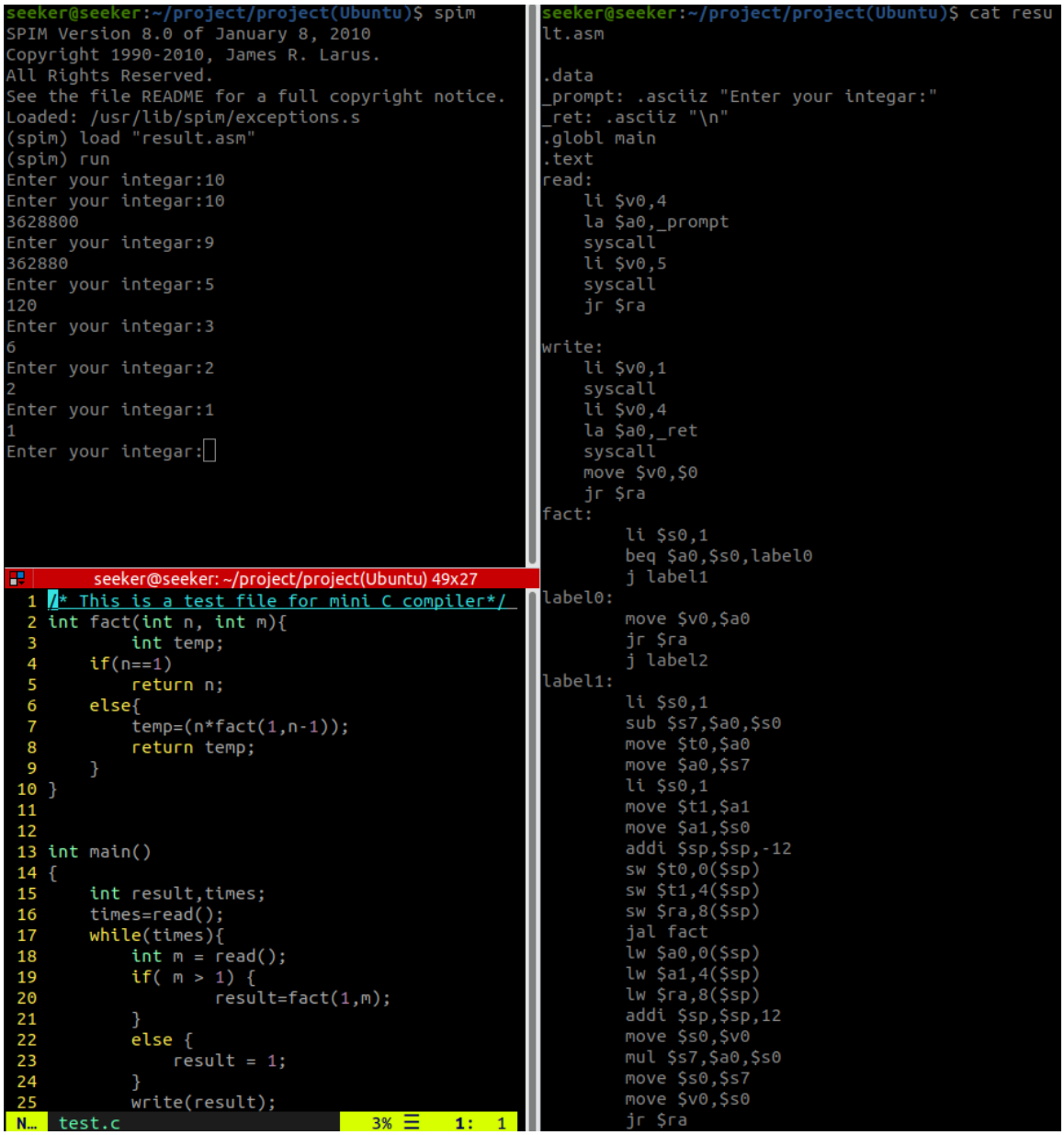
```
seeker@seeker:~/project/project(Ubuntu)$ spim
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
(spim) load "result.asm"
(spim) run
Enter your integar:10
Enter your integar:10
3628800
Enter your integar:
```

```
seeker@seeker:~/project/project(Ubuntu)$ cat resu
lt.asm

.data
_prompt: .asciiz "Enter your integar:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0,4
    la $a0,_prompt
    syscall
    li $v0,5
    syscall
    jr $ra

write:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
fact:
        li $s0,1
        beq $a0,$s0,label0
        j label1
label0:
        move $v0,$a0
        jr $ra
        j label2
label1:
        li $s0,1
        sub $s7,$a0,$s0
        move $t0,$a0
        move $a0,$s7
        addi $sp,$sp,-8
        sw $t0,0($sp)
        sw $ra,4($sp)
        jal fact
        lw $a0,0($sp)
        lw $ra,4($sp)
        addi $sp,$sp,8
        move $s0,$v0
        mul $s7,$a0,$s0
        move $s0,$s7
        move $v0,$s0
        jr $ra
label2:
main:
        addi $sp,$sp,-4
        sw $ra,0($sp)
        jal read
        lw $ra,0($sp)
```

seeker@seeker: ~/project/project(Ubuntu) 49x27

```
seeker@seeker:~/project/project(Ubuntu)$ cat test
1.c
/* This is a test file for mini C compiler*/
int fact(int n){
        int temp;
    if(n==1)
        return n;
    else{
        temp=(n*fact(n-1));
        return temp;
    }
}


int main()
{
    int result,times;
    times=read();
    while(times){
        int m = read();
        if( m > 1) {
                result=fact(m);
        }
        else {
            result = 1;
        }
        write(result);
```

测试多参递归



```
seeker@seeker:~/project/project(Ubuntu)$ spim
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
(spim) load "result.asm"
(spim) run
Enter your integar:10
Enter your integar:10
3628800
Enter your integar:9
362880
Enter your integar:5
120
Enter your integar:3
6
Enter your integar:2
2
Enter your integar:1
1
Enter your integar:
```

```
seeker@seeker:~/project/project(Ubuntu)$ cat resu
lt.asm

.data
_prompt: .asciiz "Enter your integar:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0,4
    la $a0,_prompt
    syscall
    li $v0,5
    syscall
    jr $ra

write:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
fact:
        li $s0,1
        beq $a0,$s0,label0
        j label1
label0:
        move $v0,$a0
        jr $ra
        j label2
label1:
        li $s0,1
        sub $s7,$a0,$s0
        move $t0,$a0
        move $a0,$s7
        li $s0,1
        move $t1,$a1
        move $a1,$s0
        addi $sp,$sp,-12
        sw $t0,0($sp)
        sw $t1,4($sp)
        sw $ra,8($sp)
        jal fact
        lw $a0,0($sp)
        lw $a1,4($sp)
        lw $ra,8($sp)
        addi $sp,$sp,12
        move $s0,$v0
        mul $s7,$a0,$s0
        move $s0,$s7
        move $v0,$s0
        jr $ra
```

```
seeker@seeker: ~/project/project(Ubuntu) 49x27
 1 /* This is a test file for mini C compiler*/
 2 int fact(int n, int m){
 3         int temp;
 4     if(n==1)
 5         return n;
 6     else{
 7         temp=(n*fact(1,n-1));
 8         return temp;
 9     }
10 }
11
12
13 int main()
14 {
15     int result,times;
16     times=read();
17     while(times){
18         int m = read();
19         if( m > 1) {
20                 result=fact(1,m);
21         }
22         else {
23             result = 1;
24         }
25         write(result);
N… test.c                    3% ☰    1:  1
```

参考资料

编译器原理及实践

南京大学许畅老师编译原理课程
https://cs.nju.edu.cn/changxu/2_compiler/index.html
清华大学编译原理

https://github.com/decaf-lang/decaf-2019-project/blob/master/PA5.md