

浙江大学

本科实验报告

课程名称: 编译原理

姓 名: 宋鼎 黄杨思博 赵宣栋

学 院: 计算机学院

系: 计算机科学与技术

专 业: 计算机科学与技术

学 号: 3150104414 3150104412 3150104910

指导教师: 李莹

2018 年 6 月 10 日



一、 引言	3
1.1 简介.....	3
1.2 编写目的.....	3
1.3 相关概念.....	3
1.4 开发环境.....	4
二、 实验需求.....	5
2.1 需求概述.....	5
2.2 要实现的编译器各个环节.....	5
三、 词法分析	6
3.1 LEX 表达式	6
3.2 正则表达式标记声明	7
3.3 部分符号匹配规则.....	8
3.4 具体实现	9
四、 语法分析	10
4.1 抽象语法树.....	10
4.2 定义 TYPE.....	11
4.3 部分 YACC 文法	11
4.4 部分处理函数	12
4.5 抽象语法树实现	14
五、 语义分析	18
5.1 模块概述	18
5.2 整体设计	19
5.3 TABLE 部分	19
5.4 TYPE 部分	20
5.5 SEMANT 部分	21
5.6 中间代码格式	22
5.7 数据结构描述	25
5.8 中间代码生成的实现	26
六、 符号表设计	27
6.1 符号表数据结构	28
6.2 符号表实现	29
七、 运行环境设计	32
7.1 数据结构描述	33
八、 转为目标代码.....	35
九、 测试	48
10.1 自定义数组测试	48
10.2 普通运算测试	54



10.3	循环测试	58
10.4	简单函数测试	61
10.5	函数递归	65
10.6	函数嵌套调用	69
10.7	错误测试	73
十、	总结	75
十一、	分工	76



一、引言

1.1 简介

近年来，计算机界的编程语言越来越多，从迄今已有 60 高龄的 Fortran，到近日苹果发布了 Swift。编程语言的学习是无止境的，然而，想要成为真正的计算机学者，学习编程语言背后的实现方式才是最为重要的。语言的语法是如何来的，想要解析语言语法，如何编写和生成语法树，这都是在设计和实现一门语言时应该考虑的。

根据维基百科的定义，编译器（compiler），是一种计算机程序，它会将用某种编程语言写成的源代码（原始语言），转换成另一种编程语言（目标语言）。它主要的目的是将便于人编写、阅读、维护的高级计算机语言所写作的源代码程序，翻译为计算机能解读、运行的低阶机器语言的程序，也就是可执行文件。编译器将原始程序（source program）作为输入，翻译产生使用目标语言（target language）的等价程序。源代码一般为高阶语言（High-level language），如 Pascal、C、C++、C#、Java 等，而目标语言则是汇编语言或目标机器的目标代码（Object code），有时也称作机器代码（Machine code）。

1.2 编写目的

设计并实现一个 Pascal 或 C 或其他语言的编译系统，掌握编译原理的各个环节：词法分析、语法分析、语义分析、代码生成，以及实现所需的数据结构：语法树、符号表等。通过这样的实验，提高学生协作编程的能力，加深对编译技术的理解，编译原理是一门综合各个学科知识的课程，编译系统设计让学生在实践中综合理解计算机学科知识。

1.3 相关概念

1.3.1 Pascal

Pascal 是一个有影响的面向对象和面向过程编程语言，由尼克劳斯·维尔特在 1968 年 9 月设计，在 1970 年发行，作为一个小型的和高效的语言，意图



鼓励使用结构化编程和数据结构进行良好的编程实践。

最早出现的结构化编程语言，具有丰富的数据类型和简洁灵活的操作语句，其主要特点有：严格的结构化形式；丰富完备的数据类型；运行效率高；查错能力强。

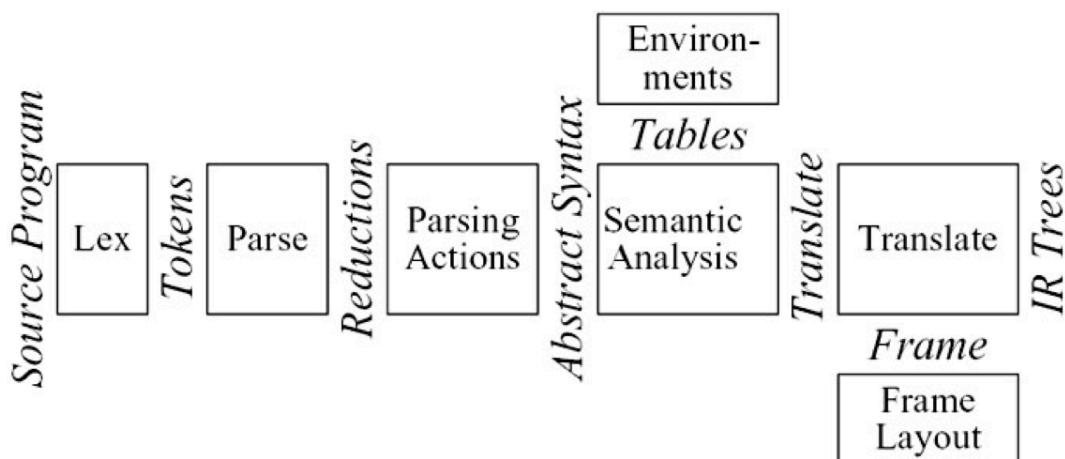
1.3.2 编译器

编译器是将一种语言翻译为另一种语言的计算机程序。编译器将源程序编写的程序作为输入，而产生用目标语言编写的等价程序。



1.4 开发环境

使用 MacOS 操作系统和 C 语言进行开发





我们总体参考了 Modern_Compiler_Implementation_in_C 的架构，完成了 Pascal 语言 mini 编译器，对 Pascal 语言程序进行词法分析，语法分析，生成抽象语法树后进行语义分析，对符号表进行管理，并在语义分析的同时翻译生成中间表示语言 IR trees，在栈帧中对变量地址进行分配。

二、实验需求

2.1 需求概述

2.1.1 语法定义

可检索获取 Pascal 或 C 或其他语言的语法定义。

2.1.2 生成代码

生成 MIPS 目标代码，代码在 MIPS 模拟器上运行，或者生成 TM 目标码，代码在 TM 虚拟机运行。也可以生成汇编代码，直接在 PC 上运行。

2.1.3 运行环境

Windows 系统环境或者 Linux 环境。

2.1.4 输入

输入为符合实验对应语言的语法规则程序示例。

2.1.5 输出

MIPS 指令代码或 TM 代码或者汇编代码。

2.2 要实现的编译器各个环节

编译器包含词法分析、语法分析、语义分析、代码生成、代码优化、运行环境等阶段和环节，是一个复杂的系统，本实验针对前四个阶段进行，不包括代码优化阶段。

2.2.1 词法分析

可以考虑用词法分析生成器 LEX 来生成。实验者提供 Pascal-的语法规则脚本文件。



2.2.2 语法分析

语法分析器考虑用语法分析器生成工具 YACC 来实现。实验者提供 Pascal-语法规则的 YACC 脚本文件。

2.2.3 语义分析

根据附录中的 Pascal-语法规则写出语义分析程序

2.2.4 代码生成

生成 MIPS 指令代码或者 TM 目标码或者汇编代码。

2.2.5 运行环境

基于堆栈的运行环境，支持局部函数。

2.2.6 符号表

用 hash table 实现符号表。给出 hash 函数设计及其实现。

三、词法分析

Lex 是一种生成扫描器的工具。扫描器是一种识别文本中的词汇模式的程序。这些词汇模式（或者常规表达式）在一种特殊的句子结构中定义。

一种匹配的常规表达式可能会包含相关的动作。这一动作可能还包括返回一个标记。当 Lex 接收到文件或文本形式的输入时，它试图将文本与常规表达式进行匹配。它一次读入一个输入字符，直到找到一个匹配的模式。如果能够找到一个匹配的模式，Lex 就执行相关动作（可能包括返回一个标记）。另一方面，如果没有可以匹配的常规表达式，将会停止进一步的处理，Lex 将显示一个错误消息。

3.1 Lex 表达式

常规表达式是一种使用元语言的模式描述。表达式由符号组成。符号一般是字符和数字，但是 Lex 中还有一些具有特殊含义的其他标记。下面两个表格定义了 Lex 中使用的一些标记并给出了几个典型的例子。

用 Lex 定义常规表达式

字符	含义



A-Z, 0-9, a-z	构成了部分模式的字符和数字。
.	匹配任意字符，除了 \n。
-	用来指定范围。例如：A-Z 指从 A 到 Z 之间的所有字符。
[]	一个字符集合。匹配括号内的 任意 字符。如果第一个字符是 ^ 那么它表示否定模式。例如: [abC] 匹配 a, b, 和 C 中的任何一个。
*	匹配 0 个或者多个上述的模式。
+	匹配 1 个或者多个上述模式。
?	匹配 0 个或 1 个上述模式。
\$	作为模式的最后一个字符匹配一行的结尾。
{}	指出一个模式可能出现的次数。 例如: A{1,3} 表示 A 可能出现 1 次或 3 次。
\	用来转义元字符。同样用来覆盖字符在此表中定义的特殊意义，只取字符的本意。
^	否定。
	表达式间的逻辑或。
"<一些符号>"	字符的字面含义。元字符具有。
/	向前匹配。如果在匹配的模版中的"/"后跟有后续表达式，只匹配模版中"/"前 面的部分。如：如果输入 A01，那么在模版 A0/1 中的 A0 是匹配的。
()	将一系列常规表达式分组。

对 Main 模块的接口，即主接口 int yyparse(); 把程序进行词法和语法分析，然后返回抽象树的根。

3.2 正则表达式标记声明

div	[dD][iI][vV]
mod	[mM][oO][dD]
integer	[0-9]+
char	[A-zA-Z\.\"]
string	\"char+\"
program	[pP][rR][oO][gG][rR][aA][mM]
const	[cC][oO][nN][sS][tT]



type	[tT][yY][pP][eE]
var	[vV][aA][rR]
array	[aA][rR][rR][aA][yY]
of	[oO][fF]
end	[eE][nN][dD]
function	[fF][uU][nN][cC][tT][iI][oO][nN]
procedure	[pP][rR][oO][cC][eE][dD][uU][rR][eE]
begin	[bB][eE][gG][iI][nN]
read	[rR][eE][aA][dD]
if	[iI][fF]
then	[tT][hH][eE][nN]
else	[eE][lL][sS][eE]
repeat	[rR][eE][pP][eE][aA][tT]
until	[uU][nN][tT][iI][lL]

3.3 部分符号匹配规则

把有效的符号对应的 Token 返回。

.."	{adjust(); return DOTDOT;}
"("	{adjust(); return LP;}
")"	{adjust(); return RP;}
"["	{adjust(); return LB;}
{adjust(); return RB;}	
"."	{adjust(); return DOT;}
";"	{adjust(); return COMMA; }
{adjust(); return COLON; }	
";"	{adjust(); return SEMI; }
"+"	{adjust(); return PLUS;}
"_"	{adjust(); return MINUS;}
"**"	{adjust(); return MUL;}
"/"	{adjust(); return REALDIV;}
{div}	{adjust(); return INTDIV;}
{mod}	{adjust(); return MOD; }
">="	{adjust(); return GE;}
">"	{adjust(); return GT;}
"<="	{adjust(); return LE;}
"<"	{adjust(); return LT;}
"=="	{adjust(); return EQUAL;}
"<>"	{adjust(); return UNEQUAL;}
"::"	{adjust(); return ASSIGN;}



部分关键字的匹配：

```
{program}           {adjust(); return PROGRAM;}  
{const}            {adjust(); return CONST;}  
{type}             {adjust(); return TYPE;}  
{var}              {adjust(); return VAR;}  
{array}            {adjust(); return ARRAY;}  
{of}               {adjust(); return OF;}  
{end}              {adjust(); return END;}  
{function}         {adjust(); return FUNCTION;}  
{procedure}        {adjust(); return PROCEDURE;}  
{begin}            {adjust(); return BEGIN_T;}  
{read}             {adjust(); return READ;}  
{if}               {adjust(); return IF;}  
{then}             {adjust(); return THEN;}  
{else}             {adjust(); return ELSE;}  
{repeat}           {adjust(); return REPEAT;}  
{until}            {adjust(); return UNTIL;}  
{while}            {adjust(); return WHILE;}  
{do}               {adjust(); return DO;}  
{for}              {adjust(); return FOR;}  
{to}               {adjust(); return TO;}  
{downto}           {adjust(); return DOWNTO;}  
{goto}             {adjust(); return GOTO;}  
{and}              {adjust(); return AND;}  
{or}               {adjust(); return OR;}  
{not}              {adjust(); return NOT;}  
[a-zA-Z][0-9a-zA-Z]* {adjust(); yylval.sval = yytext; return ID;}  
{integer}          {adjust(); int getNum; sscanf(yytext, "%d", &getNum); yylval.ival =  
getNum; return INTEGER;}  
{char}              {adjust(); yylval.eval = yytext[0]; return CHAR; }  
{string}            {adjust(); yylval.sval = yytext; return STRING; }
```

3.4 具体实现

Adjust 函数，每次读完一个 token 之后就把位置移动 yyleng 的长度，并记录下 error 的位置。

```
void adjust(void) {  
    EM_tokPos = charPos;  
    charPos += yyleng;
```



{}

四、 语法分析

根据维基百科的定义: yacc (Yet Another Compiler Compiler), 是 Unix/Linux 上一个用来生成编译器的编译器 (编译器代码生成器)。yacc 生成的编译器主要是用 C 语言写成的语法解析器 (Parser)，需要与词法解析器 Lex 一起使用，再把两部分产生出来的 C 程序一并编译。yacc 本来只在 Unix 系统上才有，但现时已普遍移植往 Windows 及其他平台。yacc 的输入是巴科斯范式 (BNF) 表达的语法规则以及语法规约的处理代码，yacc 输出的是基于表驱动的编译器，包含输入的语法规约的处理代码部分。yacc 采用 LALR (1) 语法分析方法，最初由 AT&T 的 Steven C. Johnson 为 Unix 操作系统开发，后来一些兼容的程序如 Berkeley Yacc, GNU bison, MKS yacc 和 Abraxas yacc 陆续出现。它们都在原先基础上做了少许改进或者增加，但是基本概念是相同的。

用 Yacc 来创建一个编译器包括四个步骤：

1. 通过在语法文件上运行 Yacc 生成一个解析器。
2. 说明语法：
 - a) 编写一个 .y 的语法文件（同时说明 C 在这里要进行的动作）。
 - b) 编写一个词法分析器来处理输入并将标记传递给解析器。这可以使用 Lex 来完成。
 - c) 编写一个函数，通过调用 yyparse() 来开始解析。
 - d) 编写错误处理例程（如 yyerror()）。
3. 编译 Yacc 生成的代码以及其他相关的源文件。
4. 将目标文件链接到适当的可执行解析器库。

4.1 抽象语法树

抽象树是源代码的抽象语法结构的树状表现形式，这里特指编程语言的源代码。树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套



括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于 if-condition-then 这样的条件跳转语句，可以使用带有两个分支的节点来表示。

对 Main 模块的接口，即主接口，int yyparse(); 把程序进行此法和语法分析，然后返回抽象树的根。

4.2 定义 type

```
%type <cval> direction
%type <exp> program routine sub_routine routine_body const_value stmt compound_stmt
non_label_stmt assign_stmt proc_stmt if_stmt else_clause repeat_stmt while_stmt for_stmt
goto_stmt expression expr term factor
%type <declist> routine_head label_part const_part const_expr_list type_part var_part
var_decl_list var_decl
%type <fundec> function_decl function_head procedure_decl procedure_head
%type <fundeclist> routine_part
%type <namety> type_definition
%type <nametylist> type_decl_list
%type <explist> stmt_list expression_list args_list
%type <sym> id sys_con sys_funct sys_proc sys_type
%type <ty> type_decl simple_type_decl array_type_decl
%type <fieldlist> name_list parameters para_type_list para_decl_list var_para_list val_para_list
%type <cval> stat;
```

4.3 部分 yacc 文法

```
const_part : CONST const_expr_list {$$ = $2;}
| {$$ = NULL;}

const_expr_list : id EQUAL const_value SEMI const_expr_list {$$ = A_DecList(A_ConstDec(EM_tokPos, $1, $3), $5);}
| id EQUAL const_value SEMI {$$ = A_DecList(A_ConstDec(EM_tokPos, $1, $3), NULL);}

const_value : INTEGER {$$ = A_IntExp(EM_tokPos, $1);}
| CHAR {$$ = A_CharExp(EM_tokPos, $1);}
| STRING {$$ = A_StringExp(EM_tokPos, $1);}

type_part : TYPE type_decl_list {$$ = A_DecList(A_TypeDec(EM_tokPos, $2), NULL);}
| {$$ = NULL;}

type_decl_list : type_definition type_decl_list {$$ = A_NametyList($1, $2);}
| type_definition {$$ = A_NametyList($1, NULL);}

type_definition : id EQUAL type_decl SEMI {$$ = A_Namety($1, $3);}

type_decl : simple_type_decl {$$ = $1;}
```



```
| array_type_decl { $$ = $1; }

simple_type_decl : sys_type { $$ = A_NameTy(EM_tokPos, $1); }
| id { $$ = A_NameTy(EM_tokPos, $1); }
| LP name_list RP { $$ = A_EnumType(EM_tokPos, $2); }
| const_value DOTDOT const_value { $$ = A_RangeTy(EM_tokPos, $1, $3); }
| MINUS const_value DOTDOT const_value { A_RangeTy(EM_tokPos,
A_OpExp(EM_tokPos, A_minusOp, A_IntExp(EM_tokPos, 0), $2), $4); }
| MINUS const_value DOTDOT MINUS const_value { A_RangeTy(EM_tokPos,
A_OpExp(EM_tokPos, A_minusOp, A_IntExp(EM_tokPos, 0), $2), A_OpExp(EM_tokPos, A_minusOp,
A_IntExp(EM_tokPos, 0), $5)); }
| id DOTDOT id { A_RangeTy(EM_tokPos, A_VarExp(EM_tokPos,
A_SimpleVar(EM_tokPos, $1)), A_VarExp(EM_tokPos, A_SimpleVar(EM_tokPos, $3))); }

array_type_decl : ARRAY LB simple_type_decl RB OF type_decl { $$ = A_ArrayTy(EM_tokPos, $3,
$6->u.name); }

name_list : name_list COMMA id { $$ = A_linkFieldList($1, A_FieldList(A_Field(EM_tokPos, $3, NULL),
NULL)); }
| id { $$ = A_FieldList(A_Field(EM_tokPos, $1, NULL), NULL); }

var_part : VAR var_decl_list { $$ = $2; }
| { $$ = NULL; }

var_decl_list : var_decl var_decl_list { $$ = A_linkDecList($1, $2); }
| var_decl { $$ = $1; }

var_decl : name_list COLON type_decl SEMI { $$ = A_setDecListType(A_unDecList($1), $3); }
```

4.4 部分处理函数

```
A_fieldList A_linkFieldList(A_fieldList front, A_fieldList tail) { // link two fieldlists
    if (!front)
        return tail;
    A_fieldList pos = front;
    for (; pos->tail; pos = pos->tail);
    pos->tail = tail;

    return front;
}

A_fieldList A_setFieldListType(A_fieldList fieldList, A_ty ty) {
    A_fieldList front = fieldList;
    for (; fieldList; fieldList = fieldList->tail) {
        fieldList->head->_type = ty->u.name;
    }
}
```



```
        return front;
    }

A_decList A_linkDecList(A_decList front, A_decList tail) {
    if (!front)
        return tail;
    A_decList pos = front;
    for (;pos->tail;pos = pos->tail);
    pos->tail = tail;

    return front;
}

A_decList A_setDecListType(A_decList decList, A_ty ty) {
    A_decList front = decList;
    for (;decList;decList = decList->tail) {
        decList->head->u.var._type = ty->u.name;
    }

    return front;
}

A_decList A_unDecList(A_fieldList fieldList) {
    if (!fieldList)
        return NULL;

    A_field field = fieldList->head;
    A_decList decList = A_DecList(A_VarDec(field->pos, field->name, field->_type, NULL),
NULL);
    A_decList front = decList;

    for(fieldList= fieldList->tail;fieldList;fieldList = fieldList->tail, decList = decList->tail) {
        field = fieldList->head;
        decList->tail = A_DecList(A_VarDec(field->pos, field->name, field->_type, NULL),
NULL);
    }

    return front;
}
```



4.5 抽象语法树实现

Union 数据结构:

```
%union {  
    int pos;  
    int ival;  
    double rval;  
    char eval;  
    string sval;  
    A_var var;  
    A_exp exp;  
    S_symbol sym;  
    A_dec dec;  
    A_decList declist;  
    A_expList explist;  
    A_ty ty;  
    A_fieldList fieldlist;  
    A_fundec fundec;  
    A_fundecList fundeclist;  
    A_namety namety;  
    A_nametyList nametylist;  
}
```

部分节点的函数

```
A_var A_SimpleVar(A_pos pos, S_symbol sym);  
A_var A_FieldVar(A_pos pos, A_var var, S_symbol sym);  
A_var A_SubscriptVar(A_pos pos, A_var var, A_exp exp);  
  
A_exp A_VarExp(A_pos pos, A_var var);  
A_exp A_NilExp(A_pos pos);  
A_exp A_IntExp(A_pos pos, int i);  
A_exp A_CharExp(A_pos pos, char c);  
A_exp A_BoolExp(A_pos pos, bool b);  
A_exp A_StringExp(A_pos pos, string s);  
A_exp A_ArrayExp(A_pos pos, S_symbol typ, A_exp size, A_exp init);  
  
A_exp A_CallExp(A_pos pos, S_symbol func, A_expList args);  
A_exp A_OpExp(A_pos pos, A_oper oper, A_exp left, A_exp right);  
A_exp A_SeqExp(A_pos pos, A_expList seq);  
A_exp A_AssignExp(A_pos pos, A_var var, A_exp exp);
```



```
A_exp A_IfExp(A_pos pos, A_exp test, A_exp then, A_exp else);
A_exp A_WhileExp(A_pos pos, A_exp test, A_exp body);
A_exp A_RepeatExp(A_pos pos, A_exp body, A_exp test);
A_exp A_GotoExp(A_pos pos, A_exp label);
A_exp A_ForExp(A_pos pos, S_symbol var, A_exp lo, A_exp hi, A_exp body);
A_exp A_BreakExp(A_pos pos);
A_exp A_LetExp(A_pos pos, A_decList decs, A_exp body);
A_expList A_ExpList(A_exp head, A_expList tail);

A_dec A_ConstDec(A_pos pos, S_symbol constt, A_exp init);
A_dec A_VarDec(A_pos pos, S_symbol var, S_symbol typ, A_exp init);
A_dec A_TypeDec(A_pos pos, A_nametyList type);
A_dec A_FunctionDec(A_pos pos, A_fundecList function);
A_decList A_DecList(A_dec head, A_decList tail);

A_ty A_NameTy(A_pos pos, S_symbol name);
A_ty A_ArrayTy(A_pos pos, A_ty range, S_symbol element);
A_ty A_RangeTy(A_pos pos, A_exp lo, A_exp hi);
A_ty A_EnumType(A_pos pos, A_fieldList valueList);

A_field A_Field(A_pos pos, S_symbol name, S_symbol typ);
A_fieldList A_FieldList(A_field head, A_fieldList tail);

A_fundec A_Fundec(A_pos pos, S_symbol name, A_fieldList params, S_symbol result,
A_exp body);
A_fundecList A_FundecList(A_fundec head, A_fundecList tail);

A_namety A_Namety(S_symbol name, A_ty ty);
A_nametyList A_NametyList(A_namety head, A_nametyList tail);

A_exp linkIf(A_exp if1, A_exp if2);
```

部分打印函数的实现

打印抽象语法树，便于之后的直观查看。

```
void pr_exp(FILE *out, A_exp v, int d) {
    pindent(out, d);
    if (!v) {
        fprintf(out, "null");
        return;
    }
```



```
switch (v->kind) {
    case A_varExp:
        fprintf(out, "varExp(\n");
        pr_var(out, v->u._var, d + 1);
        fprintf(out, "%s", ")");
        break;
    case A_nilExp:
        fprintf(out, "nilExp()");
        break;
    case A_intExp:
        fprintf(out, "intExp(%d)", v->u._ival);
        break;
    case A_stringExp:
        chstr(v->u._sval);
        fprintf(out, "stringExp(%s)", res);
        break;
    case A_callExp:
        fprintf(out, "callExp(%s\n", S_name(v->u._call.func));
        pr_expList(out, v->u._call.args, d + 1);
        fprintf(out, ")");
        break;
    case A_opExp:
        fprintf(out, "opExp(\n");
        pindent(out, d + 1);
        pr_oper(out, v->u._op.oper);
        fprintf(out, ",\n");
        pr_exp(out, v->u._op.left, d + 1);
        fprintf(out, ",\n");
        pr_exp(out, v->u._op.right, d + 1);
        fprintf(out, ")");
        break;
    case A_seqExp:
        fprintf(out, "seqExp(\n");
        pr_expList(out, v->u._seq, d + 1);
        fprintf(out, ")");
        break;
    case A_assignExp:
        fprintf(out, "assignExp(\n");
        pr_var(out, v->u._assign.var, d + 1);
        fprintf(out, ",\n");
```



```
pr_exp(out, v->u._assign.exp, d + 1);
fprintf(out, ")");
break;

case A_ifExp:
    fprintf(out, "iffExp(\n");
    pindent(out, d + 1);
    fprintf(out, "if:\n");
    pr_exp(out, v->u._if.test, d + 2);
    fprintf(out, ",\n");
    pindent(out, d + 1);
    fprintf(out, "then:\n");
    pr_exp(out, v->u._if.then, d + 2);
    if (v->u._if._else) { /* else is optional */
        fprintf(out, "\n");
        pindent(out, d + 1);
        fprintf(out, "else:\n");
        pr_exp(out, v->u._if._else, d + 2);
    }
    fprintf(out, ")");
    break;

case A_whileExp:
    fprintf(out, "whileExp(\n");
    pr_exp(out, v->u._while.test, d + 1);
    fprintf(out, ",\n");
    pr_exp(out, v->u._while.body, d + 1);
    fprintf(out, ")");
    break;

case A_forExp:
    fprintf(out, "forExp(%s,\n", S_name(v->u._for.var));
    pr_exp(out, v->u._for.low, d + 1);
    fprintf(out, ",\n");
    pr_exp(out, v->u._for.high, d + 1);
    fprintf(out, "%s\n", ",");
    pr_exp(out, v->u._for.body, d + 1);
    fprintf(out, ",\n");
    pindent(out, d + 1);
    fprintf(out, "%s", v->u._for.escape ? "TRUE)" : "FALSE)");
    break;

case A_repeatExp:
    fprintf(out, "repeatExp( \n");
```



```
pindent(out, d + 1);
fprintf(out, "do:\n");
pr_exp(out, v->u._repeat.body, d + 2);
fprintf(out, "\n");
pindent(out, d + 1);
fprintf(out, "until:\n");
pr_exp(out, v->u._repeat.test, d + 2);
break;

case A_breakExp:
    fprintf(out, "breakExp()");
    break;

case A_letExp:
    fprintf(out, "letExp(\n");
    pr_decList(out, v->u._let.decs, d + 1);
    fprintf(out, ",\n");
    pr_exp(out, v->u._let.body, d + 1);
    fprintf(out, ")");
    break;

case A_arrayExp:
    fprintf(out, "arrayExp(%s,\n", S_name(v->u._array._type));
    pr_exp(out, v->u._array.size, d + 1);
    fprintf(out, ",\n");
    pr_exp(out, v->u._array.init, d + 1);
    fprintf(out, ")");
    break;

default:
    printf("%d\n", v->kind);
    assert(0);
}
```

五、语义分析

5.1 模块概述

语义分析是编译过程的一个逻辑阶段，语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，进行类型审查。语义分析是审查源程序有



无语义错误，为代码生成阶段收集类型信息。

语义分析阶段，需要实现的功能有：

- 将变量的定义与其各个阶段的使用联系起来
- 检查每一个表达式是否有正确的类型
- 将抽象语法树转变为中间代码表示。

5.2 整体设计

语义分析模块分为环境子模块和类型检查子模块两个模块，其中环境子模块存储变量申明，类型检查子模块调用环境子模块查询变量从而检查变量使用，如果说有的话，产生变量未申明或者类型不匹配等错误信息。下面分别介绍两个子模块的设计思路：

环境子模块：环境子模块采用了命令式风格的符号表，也就是破坏-更新式的表。在这种方式下，环境不断地更新，但是可以撤销更新从而回到之前的状态。于是，存在着单一的全局环境和撤销栈。撤销栈中记录了符号被压入到符号表的次序。每当添加一个符号到环境的同时，也将该符号加入到撤销栈中；在作用域的结束点，属于该作用域的符号将从撤销栈中弹出，并且它们的最近一次绑定也会从环境中被删除。

环境子模块包含了两个环境，分别是类型环境和值环境。类型环境记录的是类型标识符在语法上下文中表示的类型，值环境记录的是变量标识符在语法上下文中表示的类型。每当遇到类型、变量的声明时，类型检查器就会扩大这两个环境；在表达式处理期间(类型检查、中间代码生成)遇到的每一个标识符都需要查阅这两个环境。

语义分析包含了4个语法树上的递归函数。包括变量转换函数、表达式转换函数、申明转换函数和类型转换函数。

5.3 Table部分

模块 table 实现了通用的指针散列表，这个散列表即为我们的符号表，我们在此实现了包括新建空表、插入新表项、查找表项、删除表项、删除全表的功能。table.h 中的具体接口如下：



```
typedef struct TAB_table_ *TAB_table;

TAB_table TAB_empty(void);

void TAB_enter(TAB_table t, void *key, void *value);

void *TAB_look(TAB_table t, void *key);

void *TAB_pop(TAB_table t);

void *TAB_front(TAB_table t);

void TAB_dump(TAB_table t, void (*show)(void *key, void *value));
```

5.4 Type部分

Types 模块表示了 Pascal 支持的各种类型的结构。Pascal 中的基本类型是 integer 和 int; 每一种类型或者是基本类型，或者是由其他类型（基本类型、数组）构造出来的类型。本项目中，暂时没有实现 Pascal 中的记录类型。

除此之外，types 模块还实现了类型检查函数 Ty_IsMatch 和自定义类型解析函数 Ty_UnwrapNamedTy。

Types.h 中的接口如下：

```
typedef struct Ty_ty_ *Ty_ty;
typedef struct Ty_tyList_ *Ty_tyList;

struct Ty_ty_
{
    enum
    {
        Ty_nil, Ty_int, Ty_string, Ty_array,
        Ty_name, Ty_void
    } ty_type;
    union
    {
        Ty_ty array;
        struct {
            S_symbol sym;
```



```
Ty_ty ty;
} name;
} ty_storage;
};

struct Ty_tyList_
{
    Ty_ty head;
    Ty_tyList tail;
};

Ty_ty Ty_Nil();
Ty_ty Ty_Int();
Ty_ty Ty_String();
Ty_ty Ty_Void();

Ty_ty Ty_Array(Ty_ty ty);
Ty_ty Ty_Name(S_symbol sym, Ty_ty ty);

Ty_tyList Ty_TyList(Ty_ty head, Ty_tyList tail);

/* ----- Semant Utils ----- */
Ty_ty Ty_UnwrapNamedTy(Ty_ty);
bool Ty_IsMatch(Ty_ty, Ty_ty);
```

5.5 Semant部分

Semant 模块执行了对于抽象语法的语义分析以及类型检查。在翻译抽象语法时，会对变量、声明和表达式都加以翻译，并返回对应的中间语法类型（用 expty 包裹）。此外，SEM_argmatch 函数实现了对于函数调用时的参数传递类型的检查。

Semant.h 中的接口如下：

```
static expty expTy(Tr_exp e, Ty_ty t);

static expty transVar(S_table, S_table, A_var);

static expty transExp(S_table, S_table, A_exp);
```



```
static Tr_exp transDec(S_table, S_table, A_dec);

static expty transExp(S_table venv, S_table tenv, A_exp a);

static expty transVar(S_table venv, S_table tenv, A_var v);

static Tr_exp transDec(S_table venv, S_table tenv, A_dec d);

static inline bool SEM_argmatch(S_table venv, S_table tenv, A_expList exp_list, Ty_tyList arg_list,
A_exp fun);
```

5.6 中间代码格式

我们将中间代码树存储为 json 文件，即保存每个树节点的名字及左右子节点。一个 IRTree.json 文件如下所示：

```
{
  "name": "ESEQ",
  "children": [
    {
      "name": "EXP",
      "children": [
        {
          "name": "ESEQ",
          "children": [
            {
              "name": "MOVE",
              "children": [
                {
                  "name": "MEM",
                  "children": [
                    {
                      "name": "BINOP",
                      "children": [
                        {"name": "PLUS"},
                        {
                          "name": "MEM",
                          "children": [
                            {
                              "name": "BINOP",
                              "children": [
                                {
                                  "name": "BINOP"
                                }
                              ]
                            }
                          ]
                        }
                      ]
                    }
                  ]
                }
              ]
            }
          ]
        }
      ]
    }
  ]
}
```



```
"children": [
    {"name": "PLUS"},
    {"name": "TEMP(t4)"},  

    {"name": "CONST(0)"}
]
}
]
},
{
    {"name": "CONST(-4)"}
]
}
]
},
{
    {"name": "CONST(0)"}
]
},
{
    {"name": "CONST(0)"}
]
}
{
]
},
{
{
    "name": "ESEQ",
    "children": [
        {
            "name": "MOVE",
            "children": [
                {
                    "name": "MEM",
                    "children": [
                        {
                            "name": "BINOP",
                            "children": [
                                {"name": "PLUS"},  

                                {
                                    "name": "MEM",
                                    "children": [
                                        {
                                            "name": "BINOP",
                                            "children": [
                                                {
                                                    "name": "PLUS"
                                                }
                                            ]
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
```



```
"children": [
    {"name": "PLUS"},

    {"name": "TEMP(t4)"},

    {"name": "CONST(0)"}

],


}

]

},


{"name": "CONST(-4)"}

]

}

]

},


{"name": "CONST(1)"}

]

},


{"name": "CONST(0)"}

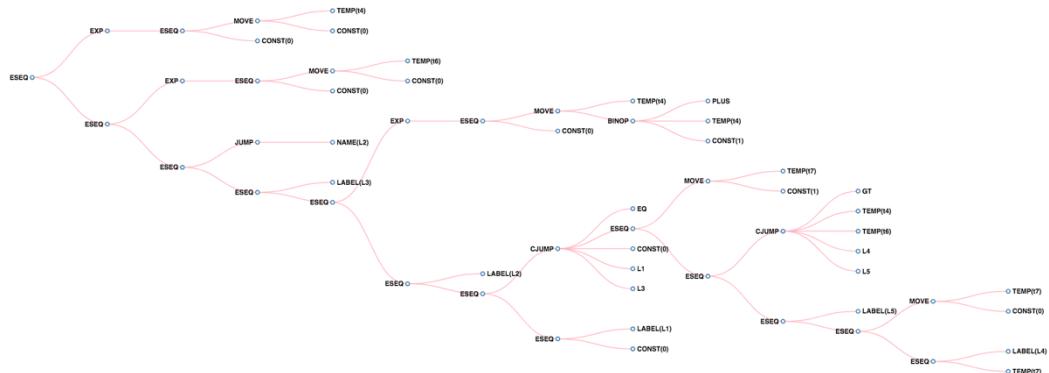
]

}

]

}
```

上述 json 文件可可视化为下图结果：



而 IR tree 中每一种操作符的含义如下：

表达式	含义
CONST i	整型常数 i



NAME n	符号常数 n,包括字符串, 函数名, Label
TEMP t	临时变量 t(寄存器此时也用临时变量表示)
BINOP o e1 e2	对操作数 e1, e2 施加二元操作符 o 表示的操作
MEM e	读取(存储)开始于地址 e 的 wordSize 个字节的内容
ESEQ s e	先执行语句 s,再以语句 e 为表达式的返回 值
MOVE TEMP t e	计算 e 并将结果送入临时单元 t
MOVE MEM e1 e2	计算 e1,得到地址 b,然后计算 e2,并将结果 存储在从 a 开始的 wordSize 个字节的存储 单元中
EXP e	计算 e 但忽略结果
JUMP e labs	将控制转移到地址 e
CJUMP o e1 e2 t f	依次计算 e1, e2, 生成 a,b,然后用关系操作符 o 比较 a 和 b。如果结果为 true,则跳转 到 t;否则跳转到 f。
SEQ s1 s2	执行完语句 s1 后再执行语句 s2
LABEL n	定义名字 n 的常数值为当前机器代码的地 址

5.7 数据结构描述

Tree.h 中定义了中间语法树所支持的所有语句类型, 主要分为无返回值的 Tr_Exp 和有返回值的 Tr_stm。两种语句对应的类型分别定义在各自的 type 中。



```
struct T_exp_
{
    enum {T_BINOP, T_MEM, T_TEMP, T_ESEQ, T_NAME, T_CONST, T_CALL} exp_type;
    union
    {
        struct {T_binOp op; T_exp left, right;} BINOP;
        T_exp MEM;
        Temp_temp TEMP;
        struct {T_stm stm; T_exp exp;} ESEQ;
        Temp_label NAME;
        int CONST;
        struct {T_exp fun; T_expList args;} CALL;
    } exp_value;
};

struct T_stm_
{
    enum {T_SEQ, T_LABEL, T_JUMP, T_CJUMP, T_MOVE, T_EXP} stm_type;
    union
    {
        struct {T_stm left, right;} SEQ;
        Temp_label LABEL;
        struct {T_exp exp; Temp_labelList jumps;} JUMP;
        struct {
            T_relOp op;
            T_exp left, right;
            Temp_label dest_true, dest_false;
        } CJUMP;
        struct {T_exp dst, src;} MOVE;
        T_exp EXP;
    } stm_value;
};
```

5.8 中间代码生成的实现

有了如上模块的辅助支持，我们现在就可以从抽象语法树生成中间代码了，translate 模块完成的就是这一功能。对于每一个读入的抽象语法语句，此模块会完成语义分析，即进行绑定与语法检查，完成后，则会将该语句翻译成对应



的 Tr 语句。

Translate.h 中的接口如下：

```
Tr_exp Tr_arithExp(A_oper, Tr_exp, Tr_exp);
Tr_exp Tr_simpleVar(Tr_access, Tr_level);
Tr_exp Tr_fieldVar(Tr_exp, int);
Tr_exp Tr_subscriptVar(Tr_exp, Tr_exp);
Tr_exp Tr_stringExp(char *);
Tr_exp Tr_intExp(int);
Tr_exp Tr_noExp();
Tr_exp Tr_callExp(Temp_label label, Tr_level, Tr_level, Tr_expList *);
Tr_exp Tr_nilExp();
Tr_exp Tr_arrayExp(Tr_exp, Tr_exp);
Tr_exp Tr_seqExp(Tr_expList);
Tr_exp Tr_doneExp();
Tr_exp Tr_forExp(Tr_exp, Tr_exp, Tr_exp, Tr_exp, Tr_exp);
Tr_exp Tr_whileExp(Tr_exp, Tr_exp, Tr_exp);
Tr_exp Tr_assignExp(Tr_exp, Tr_exp);
Tr_exp Tr_breakExp(Tr_exp);
Tr_exp Tr_eqExp(A_oper, Tr_exp, Tr_exp);
Tr_exp Tr_eqStringExp(A_oper, Tr_exp, Tr_exp);
Tr_exp Tr_eqRef(A_oper, Tr_exp, Tr_exp);
Tr_exp Tr_relExp(A_oper, Tr_exp, Tr_exp);
Tr_exp Tr_ifExp(Tr_exp, Tr_exp, Tr_exp);
```

此外，还有以下接口完成不同类型 Tr 语句间的转换。

```
T_exp unEx(Tr_exp trexp);
Tr_exp Tr_Ex(T_exp texp);
T_stm unNx(Tr_exp trexp);
Tr_exp Tr_Nx(T_stm tstm);
Cx unCx(Tr_exp trexp);
```

六、 符号表设计

符号表是一种用于语言翻译器（例如编译器和解释器）中的数据结构。在符号表中，程序源代码中的每个标识符都和它的声明或使用信息绑定在一起，比如其数据类型、作用域以及内存地址。

符号表在编译程序工作的过程中需要不断收集、记录和使用源程序中一些



语法规则的类型和特征等相关信息。这些信息一般以表格形式存储于系统中。如常数表、变量名表、数组名表、过程名表、标号表等等，统称为符号表。对于符号表组织、构造和管理方法的好坏会直接影响编译系统的运行效率。

6.1 符号表数据结构

符号表是由符号构成的，关于符号表中的符号，我们以类型 S_symbol_ 将其定义在 Symbol.h 中。

```
typedef struct S_symbol_ *S_symbol;
struct S_symbol_ {
    char* name;
    S_symbol next;
};
```

可以看到，每一个符号表中的元素包含了一个 char *类型的字符串名称和一个指向同一 sym_table 下一环的指针 next。

全局的 sym_table 用于打印和管理所有符号元素，对应的每一个表象中，symbol 以指针相连接。

S_table 则是由 S_symbol 类型的符号组成的符号表，用于在 Pascal 语言中出现 begin-end，即出现新环境时使用，定义在 symbol.h 中。

```
typedef struct TAB_table * S_table;
```

其符号表的表实现是由 TAB_table 完成的，这个类型定义在 table.c 中。

```
struct TAB_table_ {
    TAB_bucket bucket_list[TABSIZE];
    TAB_stack aux_stack;
};
```

与其相关的两个类型 TAB_bucket 和 TAB_stack 定义如下。

```
typedef struct TAB_bucket_ *TAB_bucket;
struct TAB_bucket_
{
    void *key;
    void *value;
    TAB_bucket next;
};
static TAB_bucket Bucket(void *, void *, TAB_bucket);
typedef struct TAB_stack_ *TAB_stack;
```



```
struct TAB_stack_
{
    void * symbol;
    TAB_stack next;
};
```

它们分别以 hash 表和 stack 的形式存储了每一个符号。

这样的表，在使用的过程中，对于每一组环境，我们维持了两个符号表，`type_env` 与 `value_env`，即类型和值的符号表。

6.2 符号表实现

每产生一个新的符号。就需要建立一个新的 `S_Symbol` 类型。每一个符号的本质其实就是一个字串，因此构造函数非常简单。

```
S_symbol S_Symbol(char *name)
{
    int index = hash(name) % SYMTAB_SIZE;
    S_symbol sym = sym_table[index];

    for (S_symbol p = sym; p; p = p->next)
    {
        if (!strcmp(p->name, name))
        {
            return p;
        }
    }

    S_symbol new_sym = (S_symbol)malloc(sizeof(*new_sym));
    new_sym->name = (char *)malloc(sizeof(strlen(name)) + 1);
    strcpy(new_sym->name, name);
    new_sym->next = sym;
    sym_table[index] = new_sym;
    return new_sym;
}
```

`symbol` 的生成在进行语法分析时就已经全部结束。如分析 `a := b` 时，`a` 和 `b` 就作为 `Symbol` 被存入抽象语法树中了。其调用大致如下。

```
{(yyval.sym) = S_Symbol((yyvsp)[(1)-(1)].sval);}
```

在构建 `type_env` 与 `value_env` 的符号表的过程，其实就是从 `begin` 起建立新的过程、插入符号，再从 `end` 后结束、删除符号的过程。如在 `function` 中有



层层嵌套的 repeat/while/for，其中有 begin-end 时，同名的变量就可能导致冲突。而符号表的存在很好地将这个问题解决了。因为允许自定义类型的存在类型名称也可能产生类似问题，因此才需要 type_env 与 value_env 两个表。对于这两个表，其处理方式大致如下：

```
S_beginScope(venv);
S_beginScope(tenv);
...
S_endScope(venv);
S_endScope(tenv);
```

其中，beginScope 调用了 S_enter，即在表中插入一个<mark>标记，而 endScope 则是将栈中的元素一直弹出直到<mark>标记为止。

```
static struct S_symbol_ marksym = {"<mark>", 0};
void S_beginScope(S_table t)
{
    S_enter(t, &marksym, NULL);
}
void S_endScope(S_table t)
{
    S_symbol s;
    while (TAB_front(t) != &marksym)
    {
        TAB_pop(t);
    }
    TAB_pop(t);
}
```

插入符号进入符号表，使用的是 S_enter 函数，这个函数则通过 Tab_table_ 实现。即每一次插入会在 bucket 中插入元素，也在对应的堆栈中压栈。

```
void TAB_enter(TAB_table t, void *key, void *value)
{
    assert(t && key);
    unsigned int idx = ((uintptr_t)key) % TABSIZE;
    t->bucket_list[idx] = Bucket(key, value, t->bucket_list[idx]);
    t->aux_stack = TAB_stack_push(t->aux_stack, key);
}
```

同理，将符号元素弹出调用了 TAB_Pop，不仅要删除 bucket 中的元素，也需要在栈中进行弹栈。



```
void *TAB_pop(TAB_table t)
{
    void *symbol = TAB_stack_front(t->aux_stack);
    unsigned int idx = ((uintptr_t)symbol) % TABSIZE;

    TAB_bucket b = t->bucket_list[idx];
    t->bucket_list[idx] = b->next;
    TAB_stack_pop(t->aux_stack);
    free(b);

    return symbol;
}
```

之所以需要这样复杂的设计，其实是栈和哈希表的应用范围不同导致的。哈希表适合搜索，而栈适合层层嵌套的作用域之间符号的管理。因此 S_Symbol 类型作为表和栈中的元素，被指针指引处理起来，因此也更加方便。

下面来看一下运用符号表进行类型检验的方式。

```
Ty_ty var_ty = S_look(tenv, d->u.var._type);
if (!var_ty)
{
    EM_IError(d->pos, "Undefined type: %s", S_name(d->u.var._type));
}
```

这是在 semant.c 中进行语义分析时所做的变量类型搜索。可以看到其借助的是一个非常简单的 S_look() 函数，即在 tenv 中搜索是否有该类型。如果搜索不到，则汇报错误。如果需要检查类型是否匹配，如在 assignment 中，则需要继续用 Ty_IsMatch() 对比两个类型。其中前者是被赋值变量在符号表里搜索出的符号对应类型，后者则是表达式的类型。运用 Ty_IsMatch() 进行匹配，从而达到类型检查的作用。

```
expty assignvar_expty = transVar(venv, tenv, a->u._assign.var);
expty assignexp_expty = transExp(venv, tenv, a->u._assign.exp);
if (!Ty_IsMatch(assignvar_expty.ty, assignexp_expty.ty)) {
    EM_IError(a->pos, "Semantic Error: Type of Left Unmatched Type of Right for Assign");
}
```



七、 运行环境设计

运行环境这个模块主要负责在语义分析生成 IR tree 的过程中，对基于堆栈的运行环境的搭建。

举个例子， 将简单变量用读取该简单变量在栈帧中所存放的位置来代替，即： `MEM(BINOP(PLUS, TEMP fp, CONST k))`。同理数组变量。这就说明在 `translate` 阶段以需要对变量在栈帧中的位置进行分配。所以 `translate` 阶段还包括 `frame` 的内容。(这里的 `fp` 是一个特殊的 `temp`, `t 100`)。

具体负责的主要事宜如下：

- 在定义一个函数的时候，需要为函数新建一个栈帧。
- 在定义一个变量的时候，在当前栈帧中为其分配位置。
- 在引用一个变量的时候，如果该变量在当前栈帧中，返回读取该变量所在地址的内容作为表达式；如果该变量不在当前栈帧中，则返回读取追随静态链后该变量在定义它的栈帧中所在地址的内容作为表达式。

在链表 `fraglist` 中记录定义的字符串和函数。存储一些特殊的变量，在 `translate` 阶段代表以后一些特殊功能寄存器，比如 `fp`, `return value` 等。

运行环境具体相关，主要可以分为几个模块：

1. 对外接口
2. 该模块主要是被 `translate.h` 调用。

```
/*返回 fraglist*/
F_fragList Tr_getResult()
/*生成变量引用的 IR tree 表达式*/
T_exp F_Exp(F_access access, T_exp framePtr)
/*生成静态链*/
static Tr_exp Tr_StaticLink(Tr_level now, Tr_level def);
/*新建一个栈帧*/
F_frame F_newFrame(Temp_label name, F_boolList formals)
/*定义变量时在当前栈帧中为变量分配地址*/
Tr_access Tr_allocLocal(Tr_level l, bool escape)
/*生成外部函数引用的 IR tree 表达式*/
T_exp F_externalCall(char *str, T_expList args)
```



7.1 数据结构描述

Struct F_frame_ 和 struct F_access_

Struct F_frame_ 是栈帧的数据结构。其中 formals 是所有在该 level 中存储在栈帧中或寄存器中的偏移或寄存器值(FRM_access_)的链表。

```
struct F_frame_
{
    Temp_label name;
    F_accessList formals;
    int lc;
};

struct F_access_
{
    enum {inFrame, inReg} acc_type;
    union
    {
        int offset;
        Temp_temp reg;
    } acc_value;
};
```

struct F_frag_

F_fragList_ 是存储在该 level 中定义的 string 或 procedure 的信息 (struct F_frag_) 。

```
struct F_frag_
{
    enum {F_stringFrag, F_procFrag} frag_type;
    union
    {
        struct {
            Temp_label label;
            char *str;
        } string_frag;
        struct {
            T_stm body;
            F_frame frame;
        } proc_frag;
    };
};
```



```
    } frag_value;  
};
```

具体实现

7.1.1 定义变量

调用过程: Tr_allocLocal -> F_allocLocal -> Inframe

```
F_access F_allocLocal(F_frame f, bool escape)  
{  
    f->lc++;  
    if(escape)  
    {  
        return InFrame(-1 * F_WORD_SIZE * f->lc);  
    }  
    return InReg(Temp_newTemp());  
}  
  
static F_access InFrame(int offs)  
{  
    F_access a = (F_access)malloc(sizeof(*a));  
    a->acc_type = inFrame;  
    a->acc_value.offset = offs;  
    return a;  
}
```

可以看到,在 F_allocLocal 中,调用 InFrame 返回了一个类型为 F_access 的变量,里面存放着变量是否存放在栈帧内以及如果存放在栈帧内存放的位置,之后 a->acc_value.offset = offs 则是栈指针后移,在栈帧中已经为变量分配了位置

7.1.2 引用变量

调用过程

Tr_callExp -> FRM_staticLink

```
static Tr_exp Tr_StaticLink(Tr_level now, Tr_level def)  
{  
    Tr_level tmp = now;  
    T_exp addr = T_Temp(F_FramePointer());  
  
    while(tmp && (tmp != def->parent))  
    {
```



```
F_access sl = F_formals(tmp->frame)->head;
addr = F_Exp(sl, addr);
tmp = tmp->parent;
}
return Tr_Ex(addr);
}
```

在本程序中， caller 的地址是放在 callee 栈帧的偏移为 0 的位置中，所以只要生成一个 IR tree 表达式，一直读取栈帧偏移为 0 处地址的内容（即 caller 的地址），将 caller 作为新栈帧，再判断一下如果该栈帧是变量定义的栈帧，则停止，返回该栈帧首地址加上该变量的偏移量。否则继续追随静态链。

八、 转为目标代码

本模块的任务是将中间语法树转换为目标代码，即 MIPS 汇编代码。经过之前从抽象语法树向中间语法树的转换，树根与树上的每个结点存储在 T_stmList 类型的对象中。这个数据结构具体如下：

```
// statement 类型
typedef struct T_stm_ *T_stm;
// Expression 类型
typedef struct T_exp_ *T_exp;
//Statement list
typedef struct T_stmList_ *T_stmList;
struct T_stm_
{
    enum {T_SEQ, T_LABEL, T_JUMP, T_CJUMP, T_MOVE, T_EXP} stm_type;
    union
    {
        struct {T_stm left, right;} SEQ;
        Temp_label LABEL;
        struct {T_exp exp; Temp_labelList jumps;} JUMP;
        struct {
            T_relOp op;
            T_exp left, right;
            Temp_label dest_true, dest_false;
        };
    };
};
```



```
    } CJUMP;
    struct {T_exp dst, src;} MOVE;
    T_exp EXP;
    } stm_value;
};

struct T_stmList_
{
    T_stm head;
    T_stmList tail;
};
```

可以看到，这个从 `stmList` 为根，得到整棵 IR 树。对于含有函数与过程的程序，每一个函数/过程的 IR 树被存放在 `fragment.js` 中，与 `main` 部分区分开。除此之外，每次进行目标代码转换时还需要对应栈帧信息。这是一个 `F_Frame` 类型的对象，具体的定义如下：

```
struct F_frame_
{
    Temp_label name; //函数名
    F_accessList formals; //参数在哪里
    int lc;
};

typedef struct F_access_ * F_access; //访问方式（寄存器 or 栈帧）
typedef struct F_accessList_ * F_accessList;
struct F_accessList_
{
    F_access head;
    F_accessList tail;
};
struct F_access_
{
    enum {inFrame, inReg} acc_type;
    union
    {
        int offset;
        Temp_temp reg;
    } acc_value;
};
```

可以看到，对于每一个函数/过程，对应的 `frame` 中存放了每一个变量的存储位置（包括在寄存器和在栈帧中）。



通过已知的数据结构，我们从 IR 树的树叶开始生成 MIPS 汇编代码。

```
assem_instr_list code_generate(F_frame f, T_stmList s)
{
    auxi_frame = f;
    for (T_stmList stm_list_it = s; stm_list_it; stm_list_it = stm_list_it->tail) {
        munch_stm(stm_list_it->head);
    }
    assem_instr_list assem_instructions = final_instruction_list;
    final_instruction_list = last = NULL;
    return assem_instructions;
}
```

每一条语句，我们使用 `munch_stm()` 函数生成汇编代码，再将指令序列 `final_instruction_list` 返回，供输出到.s 文件中。生成汇编代码的具体细节如下：

```
// 解析 statement
static void munch_stm(T_stm s)
{
    // 汇编指令字符串
    string assemStrP;
    char assemString[100];
    char temp_assemString[100];
    switch (s->stm_type) {
        // MOVE 指令
        case T_MOVE:
        {
            T_exp dst = s->stm_value.MOVE.dst, src = s->stm_value.MOVE.src;
            // 判断类型
            bool condition_dst_type_mem = (dst->exp_type == T_MEM);
            bool condition_dst_type_temp = (dst->exp_type == T_TEMP);
            if (condition_dst_type_mem) {
                T_exp temp_dst_expvalue = dst->exp_value.MEM;
                // 是否是加法二元运算
                bool temp_condi_1 = (temp_dst_expvalue->exp_type == T_BINOP);
                bool temp_condi_2 = (temp_dst_expvalue->exp_value.BINOP.op == T_plus);
                bool temp_condi_mem_const = (temp_dst_expvalue->exp_type == T_CONST);
                bool temp_src_mem = (src->exp_type == T_MEM);
                if (temp_condi_1 && temp_condi_2) {
                    bool temp_condi_binop_right_const = (temp_dst_expvalue->exp_value.BINOP.right->exp_type
                    == T_CONST);
                    // MOVE (MEM(BINOP(+, e1, CONST)), e2)
                }
            }
        }
    }
}
```



```
if (temp_condi_binop_right_const) {
    T_exp e1 = dst->exp_value.MEM->exp_value.BINOP.left;
    T_exp e2 = src;
    sprintf(temp_assemString, "sw $@s1, %d($@s0)",
dst->exp_value.MEM->exp_value.BINOP.right->exp_value.CONST);
    strcpy(assemString, temp_assemString);
    assemStrP = String(assemString);
    Temp_tempList temp_temp_list_1 = Temp_TempList(munch_exp(e2), NULL);
    Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(e1), temp_temp_list_1);
    mk_ins_list(assem_move(assemStrP, NULL, temp_temp_list_2));
}
bool temp_condi_binop_left_const = (temp_dst_expvalue->exp_value.BINOP.left->exp_type ==
T_CONST);
// MOVE (MEM(BINOP(+, CONST, e1)), e2)
if (temp_condi_binop_left_const) {
    T_exp e1 = temp_dst_expvalue->exp_value.BINOP.right;
    T_exp e2 = src;
    sprintf(temp_assemString, "sw $@s1, %d($@s0",
dst->exp_value.MEM->exp_value.BINOP.left->exp_value.CONST);
    strcpy(assemString, temp_assemString);
    assemStrP = String(assemString);
    Temp_tempList temp_temp_list_1 = Temp_TempList(munch_exp(e2), NULL);
    Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(e1), temp_temp_list_1);
    mk_ins_list(assem_move(assemStrP, NULL, temp_temp_list_2));
}
// MOVE(MEM(CONST), e)
else if (temp_condi_mem_const) {
    sprintf(temp_assemString, "sw $@s0, %d($0)", dst->exp_value.MEM->exp_value.CONST);
    strcpy(assemString, temp_assemString);
    assemStrP = String(assemString);
    mk_ins_list(assem_move(assemStrP, NULL, Temp_TempList(munch_exp(src), NULL)));
}
// MOVE(MEM(e), MEM(e))
else if (temp_src_mem) {
    Temp_tempList temp_temp_list_1 = Temp_TempList(munch_exp(src->exp_value.MEM),
NULL);
    Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(dst->exp_value.MEM),
temp_temp_list_1);
    mk_ins_list(assem_move("sw $@s1, 0($@s0)", NULL, temp_temp_list_2));
}
// MOVE(MEM(e), e)
```



```
        else {
            Temp_tempList temp_temp_list_1 = Temp_TempList(munch_exp(src), NULL);
            Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(dst->exp_value.MEM),
temp_temp_list_1);
            mk_ins_list(assem_move(String("sw $@s1, 0($@s0)"), NULL, temp_temp_list_2));
        }
    }

    // MOVE(TEMP(e), e)

    else if(condition_dst_type_temp) {
        Temp_tempList temp_temp_list_1 = Temp_TempList(munch_exp(dst), NULL);
        Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(src), NULL);
        mk_ins_list(assem_move(String("add $@d0, $@s0, $0"), temp_temp_list_1, temp_temp_list_2));
    }
    else {
        // 报错
        printf("The type of the dst->exp is not valid!\n");
        assert(0);
    }
    break;
}

case T_SEQ:
{
    //依次计算两个 statement
    munch_stm(s->stm_value.SEQ.left);
    munch_stm(s->stm_value.SEQ.right); break;
}

case T_LABEL:
{
    sprintf(temp_assemString, "%s", Temp_labelString(s->stm_value.LABEL));
    strcpy(assemString, temp_assemString);
    assemStrP = String(assemString);
    assem_instr temp_assem_label = assem_label(assemStrP, s->stm_value.LABEL);
    mk_ins_list(temp_assem_label);
    break;
}

case T_JUMP:
{
    // 计算JUMP exp
    Temp_temp r = munch_exp(s->stm_value.JUMP.exp);
    Temp_tempList temp_temp_list = Temp_TempList(r, NULL);
    assem_instr temp_assem_oper = assem_oper(String("j @d0"), temp_temp_list, NULL,
assem_Tar(s->stm_value.JUMP.jumps));
}
```



```
        mk_ins_list(temp_assem_oper);
        break;
    }

    case T_CJUMP:
    {
        // 提取条件表达式各部分
        char * cmp;
        Temp_temp left = munch_exp(s->stm_value.CJUMP.left), right = munch_exp(s->stm_value.CJUMP.right);
        Temp_temp store_jump = Temp_newTemp();
        // ==判断 beq 即可
        if(s->stm_value.CJUMP.op == T_eq){
            mk_ins_list(
                assem_oper(String("beq $@s0, $@s1, @j0"), NULL, Temp_TempList(left,
                    Temp_TempList(right, NULL)),
                assem_Tar(Temp_LabelList(s->stm_value.CJUMP.dest_true, NULL))));

        }
        // !=判断 bne 即可
        else if(s->stm_value.CJUMP.op == T_ne){
            mk_ins_list(
                assem_oper(String("bne $@s0, $@s1, @j0"), NULL, Temp_TempList(left,
                    Temp_TempList(right, NULL)),
                assem_Tar(Temp_LabelList(s->stm_value.CJUMP.dest_true, NULL))));

        }
        // <判断, sub+bltz(使用$zero)
        else if(s->stm_value.CJUMP.op == T_lt){
            mk_ins_list(assem_oper(String("sub $@s0, $@s1, $@s2"), NULL,
                Temp_TempList(store_jump, Temp_TempList(left,
                    Temp_TempList(right, NULL))),
                NULL));
            mk_ins_list(assem_oper(String("bltz $@s0, @j0"), NULL, Temp_TempList(store_jump, NULL),
                assem_Tar(Temp_LabelList(s->stm_value.CJUMP.dest_true, NULL))));

        }
        // >判断, sub+bgtz
        else if(s->stm_value.CJUMP.op == T_gt){
            mk_ins_list(assem_oper(String("sub $@s0, $@s1, $@s2"), NULL,
                Temp_TempList(store_jump, Temp_TempList(left,
                    Temp_TempList(right, NULL))),
                NULL));
            mk_ins_list(assem_oper(String("bgtz $@s0, @j0"), NULL, Temp_TempList(store_jump, NULL),
                assem_Tar(Temp_LabelList(s->stm_value.CJUMP.dest_true, NULL))));

        }
        // <=判断, sub+blez
    }
```



```
else if(s->stm_value.CJUMP.op == T_le){
    mk_ins_list(assem_oper(String("sub $@s0, $@s1, $@s2"), NULL,
                           Temp_TempList(store_jump, Temp_TempList(left,
                           Temp_TempList(right, NULL))),
                           NULL));
    mk_ins_list(assem_oper(String("blez $@s0, @j0"), NULL, Temp_TempList(store_jump, NULL),
                           assem_Tar(Temp_LabelList(s->stm_value.CJUMP.dest_true, NULL))));
}
// >= 判断, sub+bgez
else if(s->stm_value.CJUMP.op == T_ge){
    string temp_string = String("sub $@s0, $@s1, $@s2");
    Temp_tempList temp_temp_list = Temp_TempList(store_jump, Temp_TempList(left,
                           Temp_TempList(right, NULL)));
    mk_ins_list(assem_oper(temp_string, NULL, temp_temp_list, NULL));
    temp_temp_list = Temp_TempList(store_jump, NULL);
    mk_ins_list(assem_oper(String("bgez $@s0, @j0"), NULL, temp_temp_list,
                           assem_Tar(Temp_LabelList(s->stm_value.CJUMP.dest_true, NULL))));
}
break;
}
case T_EXP: munch_exp(s->stm_value.EXP); break;
default: assert(0);
}
}
```

可以看到，IRT 中的 STM 类型被分别处理（CJUMP LABEL SEQUENCE MOVE）。其中，调用的 assem.c 中的函数用于将对应的寄存器填入指令中，其大体形式如下（assem 为指令，desitination 与 source 为寄存器，jump 为对应 JUMP 指令的目标地址标签）：

```
assem_instr assem_oper(string assem, Temp_tempList destination, Temp_tempList
source, assem_tar jump)
{
    assem_instr p = (assem_instr) malloc (sizeof *p);
    p->type = type_oper;
    p->value.OPER.assem = assem;
    p->value.OPER.dst = destination;
    p->value.OPER.src = source;
    p->value.OPER.jumps = jump;
    return p;
}
```



同 STM, EXP 被类似的手法翻译成汇编代码。此处篇幅有限，我们摘取部分代码进行分析。可以看到 Expression 与 Statement 的最大区别之处在于其有一个返回值，这个返回值被保存在通过 Temp_newTemp()分配的空间中。许多表达式计算在树结构上层层嵌套，这里通过先序遍历的递归手法将内容翻译出来。

```
static Temp_temp munch_exp(T_exp e)
{
    // 从临时变量中选取一个空间
    Temp_temp r = Temp_newTemp();
    // MIPS 汇编指令码
    char assemString[100];
    char temp_assemString[100];
    string assemStrP;
    switch (e->exp_type) {
        // BINOP 二元操作符
        case T_BINOP:
            {
                T_binOp e_op = e->exp_value.BINOP.op;
                // 操作符左右变量
                char * op = NULL;
                T_exp left = e->exp_value.BINOP.left;
                T_exp right = e->exp_value.BINOP.right;
                // + - * / 类型
                if(e_op == T_plus)
                    op = "add";
                ...
                // 若为表达式 OP 常量
                if (left->exp_type == T_CONST) {
                    // 汇编指令生成
                    sprintf(temp_assemString, "%si $@d0, $@d0, %x", op, left->exp_value.CONST);
                    strcpy(assemString, temp_assemString);
                    assemStrP = String(assemString);
                    // 递归解析
                    Temp_tempList temp_temp_list = Temp_TempList(r = munch_exp(right), NULL);
                    assem_instr temp_assem_ins = assem_oper(assemStrP, temp_temp_list, NULL, NULL);
                    mk_ins_list(temp_assem_ins);
                }
                // 常量 OP 表达式
                else if (e->exp_value.BINOP.right->exp_type == T_CONST) {
                    ...
                }
            }
    }
}
```



```
        }

        // 表达式 OP 表达式

        else {

            sprintf(temp_assemString, "%bs $@d0, $@s0, $@d0", op);
            strcpy(assemString, temp_assemString);
            assemStrP = String(assemString);

            // 解析右表达式

            Temp_tempList temp_temp_list_1 = Temp_TempList(r = munch_exp(right), NULL);

            // 解析左表达式

            Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(left), NULL);

            // 分配寄存器

            assem_instr temp_assem_ins = assem_oper(assemStrP, temp_temp_list_1, temp_temp_list_2, NULL);
            mk_ins_list(temp_assem_ins);

        }

        return r;
    }

    // MEM 取地址

    case T_MEM:

    {
        // 地址 EXP

        T_exp mem = e->exp_value.MEM;

        bool condi_1 = (mem->exp_type == T_BINOP);
        bool condi_2 = (mem->exp_value.BINOP.op == T_plus);
        bool condi_mem_const = (mem->exp_type == T_CONST);
        // 二元加法表达式

        if (condi_1 && condi_2) {

            T_exp left = mem->exp_value.BINOP.left, right = mem->exp_value.BINOP.right;
            bool condition_left_const = (left->exp_type == T_CONST);
            bool condition_right_const = (right->exp_type == T_CONST);

            // MEM(BINOP(+, CONST, exp))

            if (condition_left_const) {

                sprintf(temp_assemString, "lw $@d0, %d($@s0)", left->exp_value.CONST);
                strcpy(assemString, temp_assemString);
                assemStrP = String(assemString);

                // 计算 Expression

                Temp_tempList temp_temp_list = Temp_TempList(munch_exp(right), NULL);
                assem_instr temp_assem_instr = assem_move(assemStrP, Temp_TempList(r, NULL),
                temp_temp_list);
                mk_ins_list(temp_assem_instr);

            }

            // MEM(BINOP(+, exp, CONST))

        }
```



```
else if (condition_right_const) {
    sprintf(temp_assemString, "lw $@d0, %d($@s0)", right->exp_value.CONST);
    strcpy(assemString, temp_assemString);
    assemStrP = String(assemString);
    // CONST -> 新的临时变量表
    Temp_tempList temp_temp_list_1 = Temp_TempList(r, NULL);
    // EXP -> 临时变量表
    Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(left), NULL);
    assem_instr temp_assem_instr = assem_move(assemStrP, temp_temp_list_1, temp_temp_list_2);
    mk_ins_list(temp_assem_instr);
}

else {
    // 出错
    printf("Both the left and right are not const!\n");
}

// MEM(CONST)
else if (condi_mem_const) {
    ...
}

// MEM(exp)
else {
    string temp_str = String("lw $@d0, 0($@s0)");
    Temp_tempList temp_temp_list_1 = Temp_TempList(r, NULL);
    Temp_tempList temp_temp_list_2 = Temp_TempList(munch_exp(mem->exp_value.MEM), NULL);
    assem_instr temp_assem_instr = assem_move(temp_str, temp_temp_list_1, temp_temp_list_2);
    mk_ins_list(temp_assem_instr);
}

return r;
}

case T_TEMP: {
    Temp_temp temp_return_temp = e->exp_value.TEMP;
    return temp_return_temp;
}

// ESEQ
case T_ESEQ: {
    // 计算 statement
    munch_stm(e->exp_value.ESEQ.stm);
    // 返回 expression 值
    return munch_exp(e->exp_value.ESEQ.exp);
}

case T_NAME:
{
```



```
        ...
    }

    // 常量

case T_CONST:
{
    // 常量存入寄存器中
    sprintf(temp_assemString, "addi $@d0, $0, %d", e->exp_value.CONST);
    strcpy(assemString, temp_assemString);
    assemStrP = String(assemString);
    assem_instr temp_assem_instr = assem_move(assemStrP, Temp_TempList(r, NULL), NULL);
    mk_ins_list(temp_assem_instr);
    return r;
}

// CALL 调用函数

case T_CALL:
{
    // 对应函数结果在 r 中
    r = munch_exp(e->exp_value.CALL.fun);

    Temp_temp fp_temp = Temp_newTemp();
    frame_pointer = fp_temp;

    // 保存帧指针
    string temp_instr = String("lw $@d0, 0($1)");
    Temp_tempList temp_temp_list = Temp_TempList(fp_temp, NULL);
    assem_instr temp_assem_instr = assem_move(temp_instr, temp_temp_list, NULL);
    mk_ins_list(temp_assem_instr);

    // 调用函数 传参
    temp_t_call_instr = String("call @s0");
    temp_t_call_list = Temp_TempList(r, munch_args(0, e->exp_value.CALL.args));
    temp_t_call_assem_ins = assem_oper(temp_t_call_instr, F_calldefs(), temp_t_call_list, NULL);
    mk_ins_list(temp_t_call_assem_ins);

    fp_temp = Temp_newTemp();
    temp_t_call_assem_ins = assem_move(String("lw $@d0, 0($1)"), Temp_TempList(fp_temp, NULL),
    NULL);
    mk_ins_list(temp_t_call_assem_ins);

    char assem_temp[100];
    char temp_assem_temp[100];
    sprintf(temp_assem_temp, "addi $@d0, $@d0, %x", arg_count * 4);
    strcpy(assem_temp, temp_assem_temp);
    mk_ins_list(assem_oper(String(assem_temp), Temp_TempList(fp_temp, NULL), NULL, NULL));
}
```



```
sprintf(temp_assem_temp, "sw $@d0, 0($1)");
strcpy(assem_temp, temp_assem_temp);
mk_ins_list(assem_move(String(assem_temp), Temp_TempList(fp_temp, NULL), NULL));

return r;
}

default: {
    printf("The type of exp is not valid!\n");
    assert(0);
}
}
```

这里美中不足的是，我们并没有对寄存器的分配进行过多的优化，只是秉着有多少寄存器分配多少空间、尽可能先分配寄存器的原则来生成新寄存器。仅仅保证了已有寄存器不和具有特定功能的寄存器产生冲突，没有考虑在使用效率方面的优化。

在确定了寄存器之后，我们使用 `format()` 函数将完整的指令生成，即将之前用 @ 占位的寄存器用真实的分配的寄存器替换。

```
static void format(char *result, string assem, Temp_tempList destination, Temp_tempList source, assem_tar
jumps, Temp_map m)
{
    int i = 0;
    char * deliver_p = NULL;
    for(deliver_p = assem; deliver_p && *deliver_p != '\0'; deliver_p++) {
        // 替换@
        bool condition_begin_flag = (*deliver_p == '@');
        if(condition_begin_flag) {
            char replace_flag = *(++deliver_p);
            // source
            bool condition_follow_s = (replace_flag == 's');
            if(condition_follow_s){
                int n = atoi(++deliver_p);
                string s = Temp_look(m, nthTemp(source,n));
                strcpy(result+i, s);
                i += strlen(s);
            }
            // destination
            bool condition_follow_d = (replace_flag == 'd');
            if(condition_follow_d){
                int n = atoi(++deliver_p);
            }
        }
    }
}
```



```
string s = Temp_look(m, nthTemp(destination,n));
strcpy(result+i, s);
i += strlen(s);
}

// If the char is j, means the jump
bool condition_follow_j = (replace_flag == 'j');
if(condition_follow_j){
    assert(jumps);
    int n = atoi(++deliver_p);
    string s = Temp_labelString(nthLabel(jumps->labels,n));
    strcpy(result+i, s);
    i += strlen(s);
}

bool condition_follow_at = (replace_flag == '@');
if(condition_follow_at){
    result[i] = '@';
    i++;
}

bool invalid_cond = !(condition_follow_s || condition_follow_d || condition_follow_j ||
condition_follow_at);
if(invalid_cond){
    // 报错
    printf("The flag follows '@' is not valid!\n");
    assert(0);
}

else {
    result[i] = *deliver_p;
    i++;
}

result[i] = '\0';
}
```

值得注意的是，在涉及到有关函数调用和 goto 语句时，需要将 Label 与具体位置对应起来，这里便需要在处理 LABEL 时将对应指令和标签绑定，并且在输出时对应输出。



九、 测试

我们的测试集包含测试用例，对于词法分析，语法分析，语义错误以及可以正确生成中间语法树的测试用例。

10.1 自定义数组测试

我们实现了自定义类型实现，下面的测试样例是自定义数组：

```
program arraytest;
type
    a = array[1..10] of integer;
var
    i:a;
begin
    i[1]:=1;
    i[2]:=10;
    i[3]:=100;
end.
```

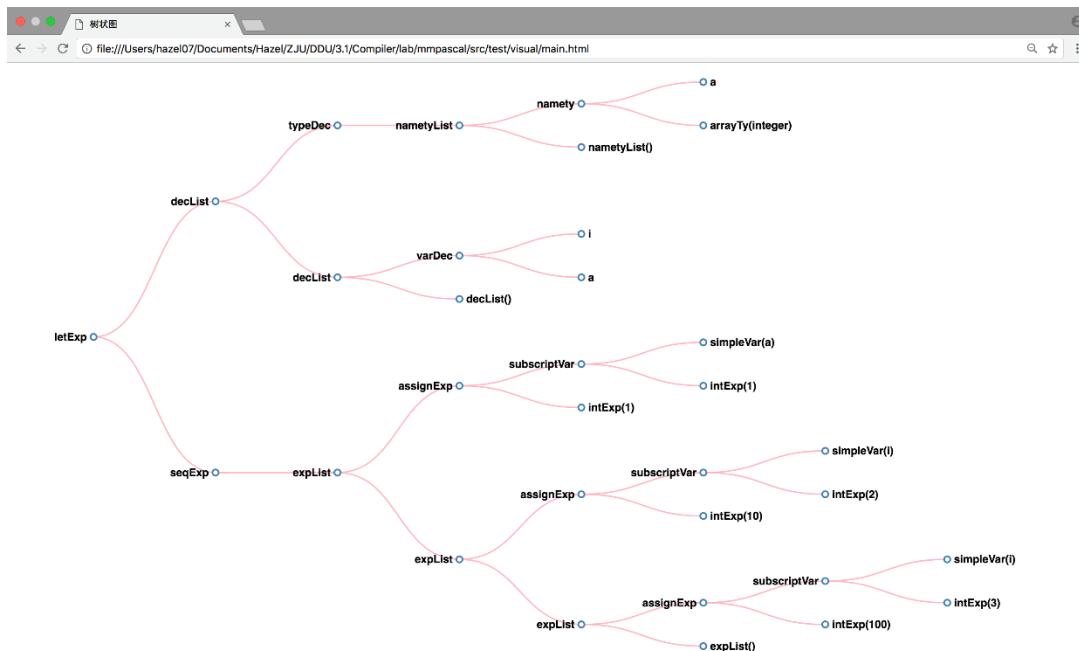
```
###AST done.###
typeDecKey: 1606428720  Key: 1606428560      varDecKey: 1606428560  Key: 1606428944
###AST2IRT done.###
No fragment in the program
Key: 1606430832      Key: 1606430832      Key: 1606430832      Key: 1606430832
Successful Write to "IRTree.json"
Tr_ex, change it to stm
Key: 1606437040      Key: 1606430832      Key: 1606437040
Successful Write to "../test/result.s"
```

抽象语法树：

```
|letExp(
|  |decList(
|  |  |typeDec(
|  |  |  |nametyList(
|  |  |  |  |namety(a,
|  |  |  |  |  |arrayTy(integer)),
|  |  |  |  |  |nametyList())),
|  |  |decList(
|  |  |  |varDec(i,
|  |  |  |  |a,
|  |  |  |  |null,
|  |  |  |  |FALSE),
|  |  |  |  |decList())),
|  |seqExp(
|  |  |expList(
```



```
| | | | assignExp(  
| | | | | subscriptVar(  
| | | | | | simpleVar(i),  
| | | | | | intExp(1)),  
| | | | | | intExp(1)),  
| | | | | expList(  
| | | | | | assignExp(  
| | | | | | | subscriptVar(  
| | | | | | | | simpleVar(i),  
| | | | | | | | intExp(2)),  
| | | | | | | | intExp(10)),  
| | | | | | | expList(  
| | | | | | | | assignExp(  
| | | | | | | | | subscriptVar(  
| | | | | | | | | | simpleVar(i),  
| | | | | | | | | | intExp(3)),  
| | | | | | | | | | intExp(100)),  
| | | | | | | | | expList()))))
```



中间语法树

```
{  
  "name": "ESEQ",  
  "children": [  
    {  
      "name": "EXP",  
      "children": [  
        {"name": "CONST(0)"}  
      ]  
    }  
  ]  
}
```



```
]
},
{
"name": "ESEQ",
"children": [
{
"name": "EXP",
"children": [
{
"name": "ESEQ",
"children": [
{
"name": "MOVE",
"children": [
{"name": "TEMP(t4)"}, {"name": "CONST(0)"}
]
},
{
"name": "CONST(0)"
}
]
}
],
{
"name": "ESEQ",
"children": [
{
"name": "EXP",
"children": [
{
"name": "ESEQ",
"children": [
{
"name": "MOVE",
"children": [
{
"name": "MEM",
"children": [
{
"name": "BINOP",
"children": [
{"name": "PLUS"},
```



```
{"name": "TEMP(t4)",  
 {  
 "name": "BINOP",  
 "children": [  
 {"name": "TIMES"},  
 {"name": "CONST(1)"},  
 {"name": "CONST(4)"}  
 ]  
 }  
 ]  
 }  
 ]  
 },  
 {"name": "CONST(1)"  
 ]  
 },  
 {"name": "CONST(0)"  
 ]  
 }  
 ]  
 },  
 {  
 "name": "ESEQ",  
 "children": [  
 {  
 "name": "EXP",  
 "children": [  
 {"name": "ESEQ",  
 "children": [  
 {"name": "MOVE",  
 "children": [  
 {  
 "name": "MEM",  
 "children": [  
 {"name": "BINOP",  
 "children": [  
 {"name": "PLUS"},  
 {"name": "TEMP(t4)"},  
 {
```



```
"name": "BINOP",
  "children": [
    {"name": "TIMES"},
    {"name": "CONST(2)"},  

    {"name": "CONST(4)"}
  ]
}
]
}
]
},
{
  {"name": "CONST(10)"}
]
},
{
  {"name": "CONST(0)"}
]
}
]
}
],
{
  "name": "ESEQ",
  "children": [
    {
      "name": "MOVE",
      "children": [
        {
          "name": "MEM",
          "children": [
            {
              "name": "BINOP",
              "children": [
                {"name": "PLUS"},  

                {"name": "TEMP(t4)"},  

                {
                  "name": "BINOP",
                  "children": [
                    {"name": "TIMES"},  

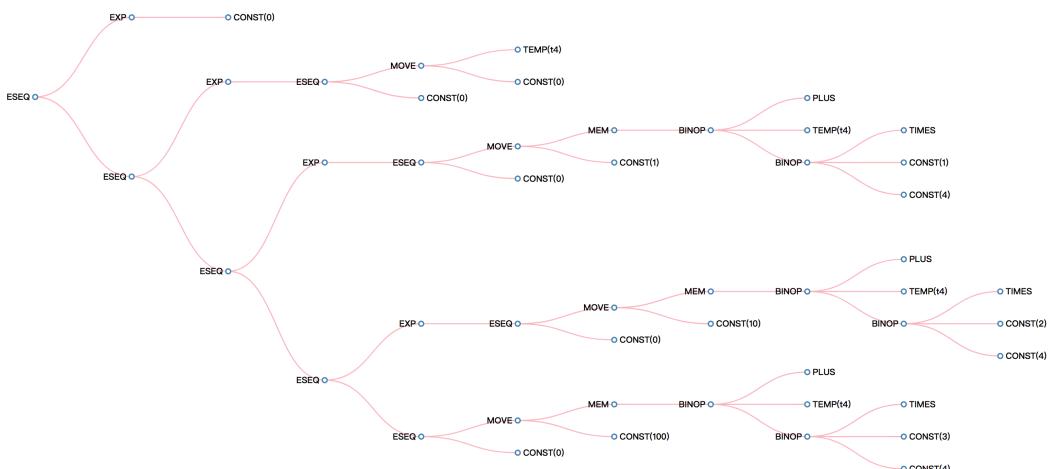
                    {"name": "CONST(3)"},  

                    {"name": "CONST(4)"}
                  ]
                }
              ]
            ]
          ]
        ]
      ]
    }
  ]
}
```



```
        }
    ]
},
{"name": "CONST(100)"}
]
},
{"name": "CONST(0)"}
]
}
]
}
]
}
]
}
]
}
}
]
```

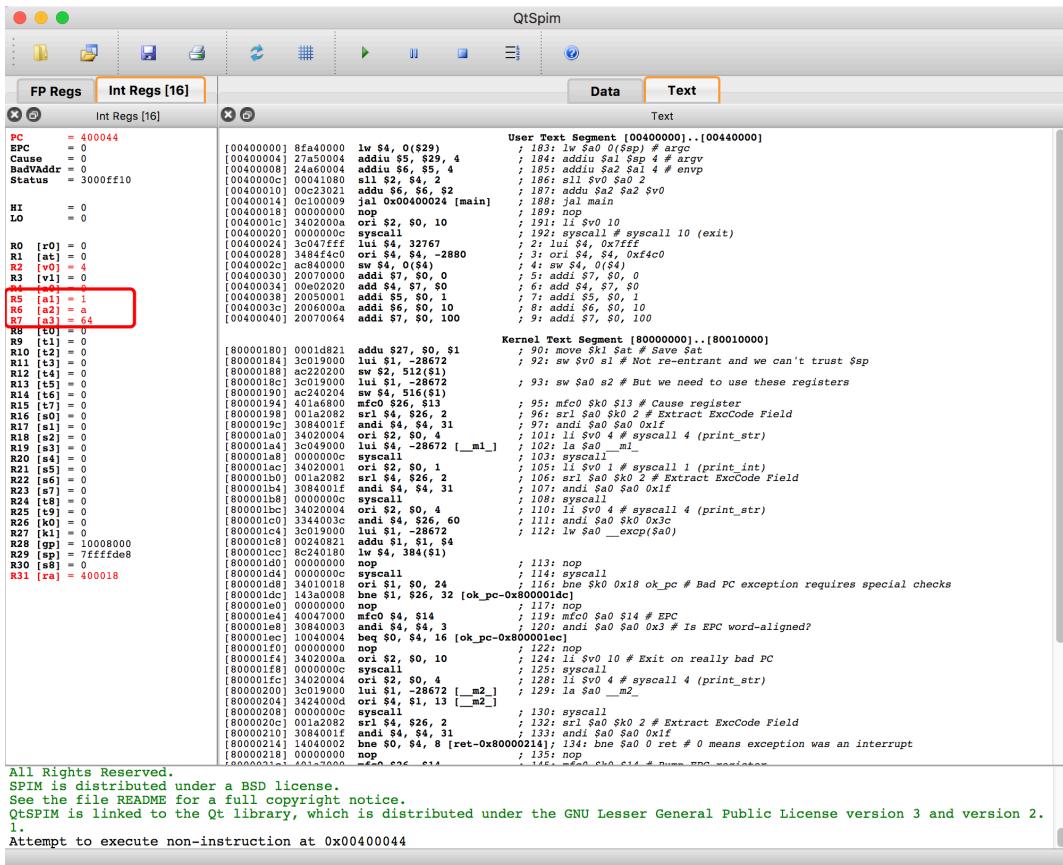
中间语法树可视化



汇编代码执行情况:

```
main:  
lui $4, 0x7fff  
ori $4, $4, 0xf4c  
sw $4, 0($4)  
addi $7, $0, 0  
add $4, $7, $0  
addi $5, $0, 1  
addi $6, $0, 10  
addi $7, $0, 100
```

用模拟器执行汇编代码。



可以看到寄存器的值被置位。

10.2 普通运算测试：

program a;

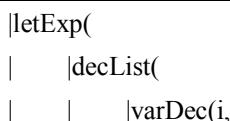
```

var
i,j,z,k,m,n:integer;

begin
i:=2;
j:=-3;
z:=i*j;
k:=i div j;
m:=i + j;
n:=i - j;
end.

```

产生的抽象语法树：

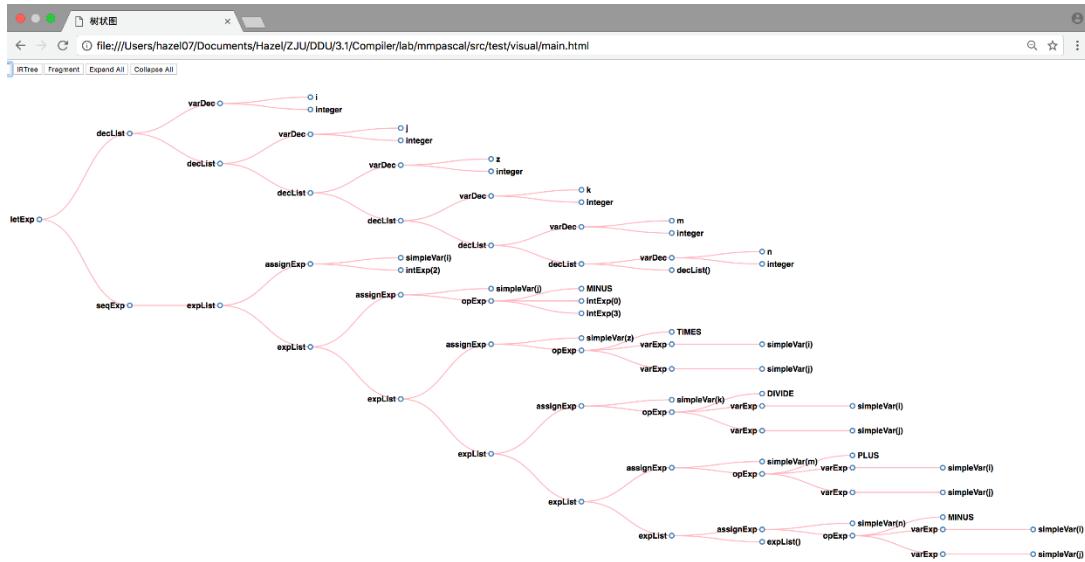




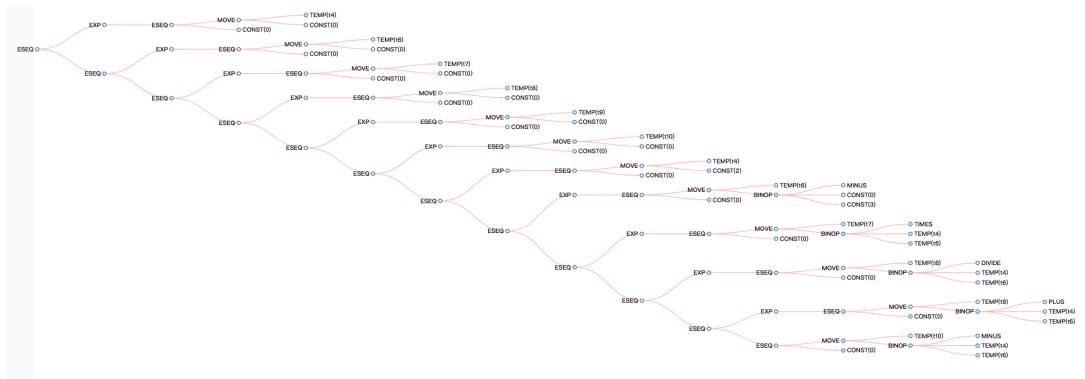
integer,
null,
FALSE),
decList(
varDec(j,
integer,
null,
FALSE),
decList(
varDec(z,
integer,
null,
FALSE),
decList(
varDec(k,
integer,
null,
FALSE),
decList(
varDec(m,
integer,
null,
FALSE),
decList(
varDec(n,
integer,
null,
FALSE),
decList()))))),
seqExp(
expList(
assignExp(
simpleVar(i),
intExp(2)),
expList(
assignExp(
simpleVar(j),
opExp(
MINUS,
intExp(0),
intExp(3))),
expList(



assignExp(
simpleVar(z),
opExp(
TIMES,
varExp(
simpleVar(i)),
varExp(
simpleVar(j))),
expList(
assignExp(
simpleVar(k),
opExp(
DIVIDE,
varExp(
simpleVar(i)),
varExp(
simpleVar(j))),
expList(
assignExp(
simpleVar(m),
opExp(
PLUS,
varExp(
simpleVar(i)),
varExp(
simpleVar(j))),
expList(
assignExp(
simpleVar(n),
opExp(
MINUS,
varExp(
simpleVar(i)),
varExp(
simpleVar(j))),
expList()))))))))



中间代码可视化



生成的汇编代码：

```
main:  
lui    $4, 0x7fff  
ori    $4, $4, 0xf4c  
sw    $4, 0($4)  
addi   $12, $0, 0  
add    $4, $12, $0  
addi   $14, $0, 0  
add    $6, $14, $0  
addi   $16, $0, 0  
add    $7, $16, $0  
addi   $18, $0, 0  
add    $8, $18, $0  
addi   $20, $0, 0  
add    $9, $20, $0
```



```
addi $22, $0, 0
add $10, $22, $0
addi $24, $0, 2
add $4, $24, $0
addi $27, $0, 3
subi $27, $27, 0
add $6, $27, $0
mul $6, $4, $6
add $7, $6, $0
div $6, $4, $6
add $8, $6, $0
add $6, $4, $6
add $9, $6, $0
sub $6, $4, $6
add $10, $6, $0
```

10.3 循环测试

针对 Pascal 的循环结构，如 repeat 循环，for 循环，while-do 循环进行测试。

Pascal 源代码：

```
program repeatUntilLoop;
var
  a: integer;
  S: integer;
begin
  a := 10;
  repeat
    a := a + 1
  until a = 20;

  For a := 1 to 100 do
  Begin
    S := S+a;
  End;

  while a > 0 do
  Begin
    S := 1;
    a := a-1;
```

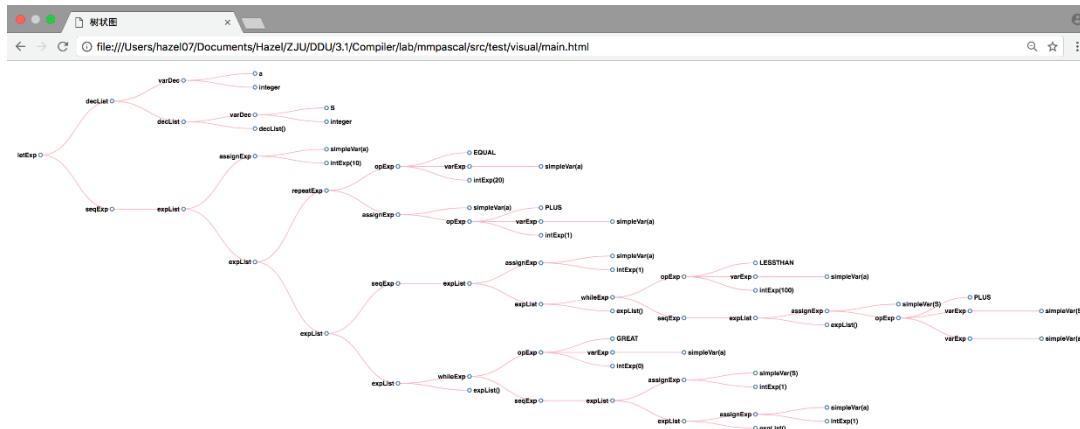


```
End;  
end.
```

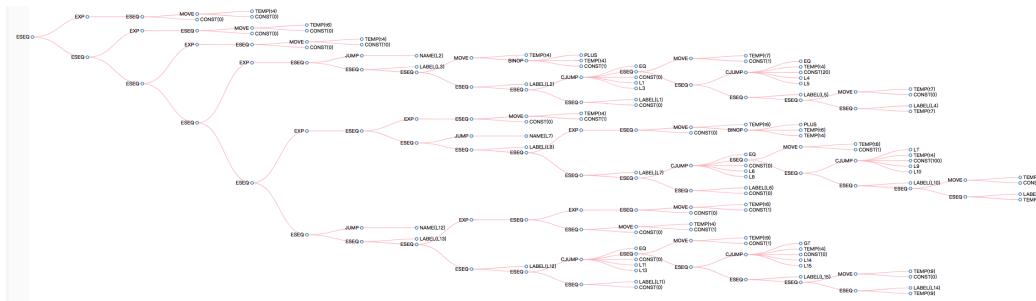
程序执行情况

```
/Users/zhaoxuandong/CLionProjects/mmpascal/src/cmake-build-debug/src ..//test/test.pas  
program repeatUntilLoop;  
var  
    a: integer;  
    S: integer;  
begin  
    a := 10;  
    repeat  
        a := a + 1  
    until a = 20;  
  
    For a := 1 to 100 do  
    Begin  
        S := S+a;  
    End;|  
  
    while a > 0 do  
    Begin  
        S := 1;  
        a := a-1;  
    End;  
end.  
###AST done.###  
varDeckKey: -1346361392 varDeckKey: -1346361392 Key: -1346361472 Key: -1346361472 Key: -1346361472  
###AST2IRT done.###  
No fragment in the program  
Key: -1345322384 Key: -1345321616 Key: -1345322384 Key: -1345322384 Key: -1345322384  
Successful Write to "IRTree.json"  
Tr_ex, change it to stm  
Key: -1344264928 Key: -1344261584 Key: -1345322384 Key: -1344264928 Key: -1344264496  
Successful Write to ".../test/result.s"
```

抽象语法树表示:



中间语法树可视化:



汇编代码:

main:

```
lui $4, 0xffff  
ori $4, $4, 0xf4c0  
sw $4, 0($4)  
addi $11, $0, 0  
add $4, $11, $0  
addi $13, $0, 0  
add $6, $13, $0  
addi $15, $0, 10  
add $4, $15, $0
```

j L2

L3:

```
addi $4, $4, 1  
add $4, $4, $0
```

L2:

```
addi $21, $0, 1  
add $7, $21, $0  
addi $23, $0, 20  
beq $4, $23, L4
```

L5:

```
addi $26, $0, 0  
add $7, $26, $0
```

L4:

```
addi $28, $0, 0  
beq $7, $28, L1
```

L1:

```
addi $31, $0, 1  
add $4, $31, $0
```

j L7

L8:

```
add $4, $6, $4  
add $6, $4, $0
```

L7:

```
addi $38, $0, 1
```



```
add $8, $38, $0
addi $40, $0, 100
sub $41, $4, $40
bltz $41, L9
L10:
addi $43, $0, 0
add $8, $43, $0
L9:
addi $45, $0, 0
beq $8, $45, L6
L6:
j L12
L13:
addi $49, $0, 1
add $6, $49, $0
addi $51, $0, 1
add $4, $51, $0
L12:
addi $53, $0, 1
add $9, $53, $0
addi $55, $0, 0
sub $56, $4, $55
bgtz $56, L14
L15:
addi $58, $0, 0
add $9, $58, $0
L14:
addi $60, $0, 0
beq $9, $60, L11
L11:
```

10.4 简单函数测试

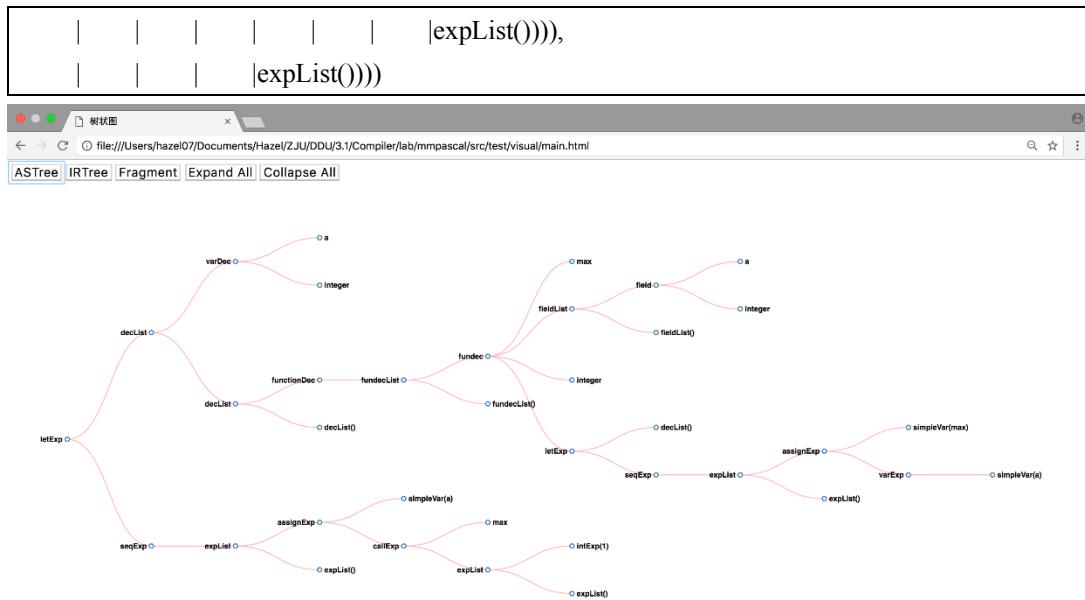
```
program test;
var
    a: integer;
Function max(a:integer): integer;
    Begin
        max := a;
    End;
begin
    a := max(1);
end.
```



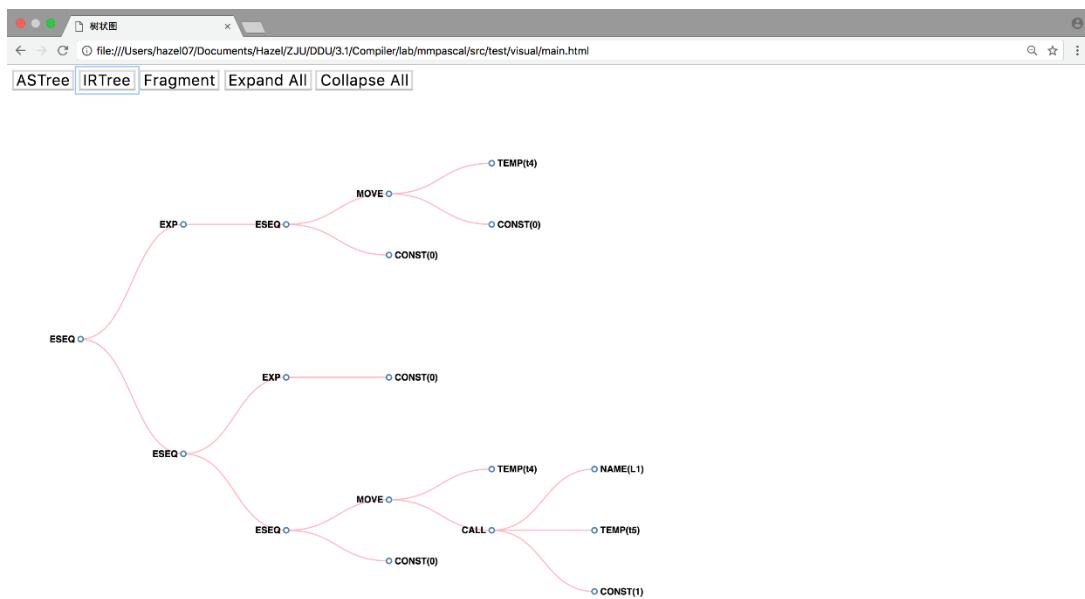
```
###AST done.###
varDecKey: 1850736784    functionDecKey: 1850736784  Key: 1850736784      Key: 1850739840
###AST2IRT done.###
Key: 1850739584      Key: 1850738880
Successful Write to "fragment.json"
Key: 1850738640      Key: 1850738640      Key: 1850738880
Successful Write to "IRTree.json"
Key: 1851786416      Key: 1850738880      Key: 1851786272      Key: 1851786416      Key: 1850739840
Key: 1851788240      Key: 1850738640      Key: 1851788240      Key: 1851788944      Key: 1851788944
Successful Write to "../test/result.s"
```

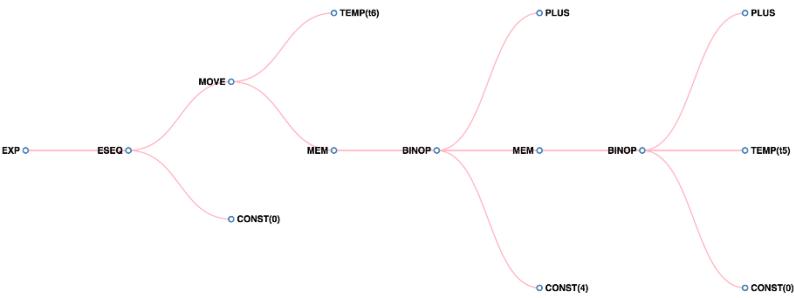
抽象语法树：

```
|letExp(
|  |decList(
|  |  |varDec(a,
|  |  |  |integer,
|  |  |  |null,
|  |  |  |FALSE),
|  |  |decList(
|  |  |  |functionDec(
|  |  |  |  |fundeclist(
|  |  |  |  |  |  |fieldList(
|  |  |  |  |  |  |  |field(a,
|  |  |  |  |  |  |  |  |integer,
|  |  |  |  |  |  |  |  |TRUE),
|  |  |  |  |  |  |  |  |fieldList()),
|  |  |  |  |  |  |  |  |integer,
|  |  |  |  |  |  |  |letExp(
|  |  |  |  |  |  |  |  |  |decList(),
|  |  |  |  |  |  |  |  |  |seqExp(
|  |  |  |  |  |  |  |  |  |  |expList(
|  |  |  |  |  |  |  |  |  |  |  |  |assignExp(
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |simpleVar(max),
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |varExp(
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |simpleVar(a))),
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |expList()))),
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |fundeclist())),
|  |  |  |  |  |  |  |  |  |  |  |  |  |decList()),
|  |  |  |  |  |  |  |  |  |  |  |  |  |seqExp(
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |expList(
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |assignExp(
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |simpleVar(a)),
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |callExp(max,
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |expList(
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |intExp(1),
```



中间语法树可视化





汇编语言结果

```
BEGIN L1
lw $9, 0($5)
lw $8, 4($9)
add $6, $8, $0
jr $ra
END L1

main:
lui $4, 0x7fff
ori $4, $4, 0xf4c0
sw $4, 0($4)
addi $12, $0, 0
add $4, $12, $0
lw $16, 0($1)
addi $17, $0, 1
sw $17, 4($16)
sw $5, 0($16)
subi $16, $16, 4
sw $16, 0($1)
addi $ra, 4($sp)
call L1
lw $25, 0($1)
addi $25, $25, 4
sw $25, 0($1)
add $4, $L1, $0
```



10.5 函数递归

我们实现了函数递归的测试：

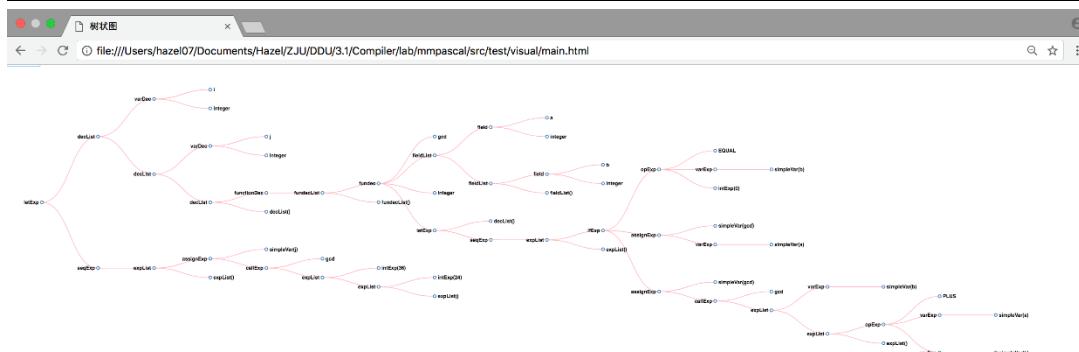
```
program a;

var
i,j:integer;
function gcd(a:integer;b:integer):integer;
begin
  if b=0 then
    gcd:=a
  else
    gcd:=gcd(b,a mod b);
end;

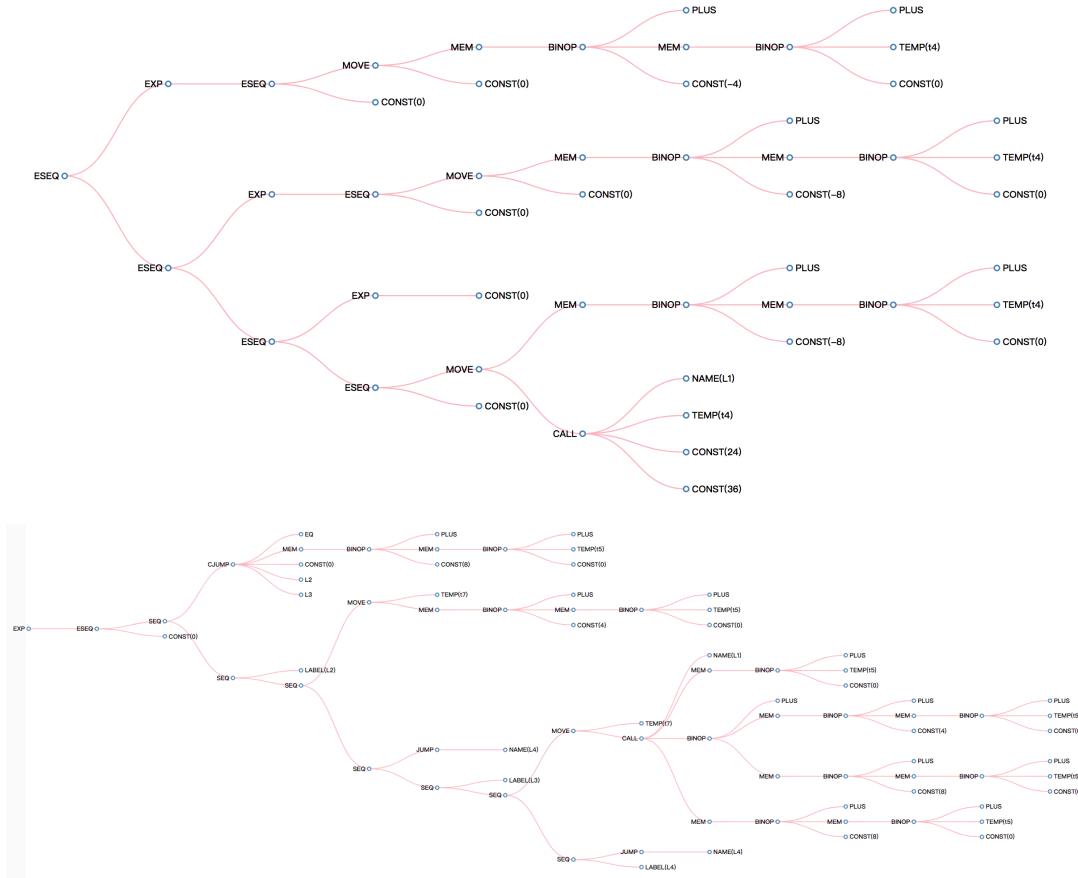
begin
j:=gcd(36,24);
end.
```

抽象语法树

```
|letExp(
|  |decList(
|  |  |varDec(i,
|  |  |  |integer,
|  |  |  |null,
|  |  |  |TRUE),
|  |  |decList(
|  |  |  |varDec(j,
|  |  |  |  |integer,
|  |  |  |  |null,
|  |  |  |  |TRUE),
|  |  |  |decList(
|  |  |  |  |functionDec(
|  |  |  |  |  |fundeclist(
|  |  |  |  |  |  |fundecl(gcd,
|  |  |  |  |  |  |  |fieldList(
|  |  |  |  |  |  |  |  |field(a,
|  |  |  |  |  |  |  |  |  |integer,
|  |  |  |  |  |  |  |  |  |TRUE),
|  |  |  |  |  |  |  |  |fieldList(
|  |  |  |  |  |  |  |  |  |field(b,
```



中间语法树：



汇编代码:

```
main:  
lui $4, 0x7fff  
ori $4, $4, 0xf4c0  
sw $4, 0($4)  
addi $11, $0, 0  
add $4, $11, $0  
addi $13, $0, 0  
add $6, $13, $0  
addi $15, $0, 10  
add $4, $15, $0  
j L2  
L3:  
addi $4, $4, 1  
add $4, $4, $0  
L2:  
addi $21, $0, 1  
add $7, $21, $0  
addi $23, $0, 20  
beq $4, $23, L4  
L5:
```



```
addi $26, $0, 0
add $7, $26, $0
L4:
addi $28, $0, 0
beq $7, $28, L1
L1:
addi $31, $0, 1
add $4, $31, $0
j L7
L8:
add $4, $6, $4
add $6, $4, $0
L7:
addi $38, $0, 1
add $8, $38, $0
addi $40, $0, 100
sub $41, $4, $40
bltz $41, L9
L10:
addi $43, $0, 0
add $8, $43, $0
L9:
addi $45, $0, 0
beq $8, $45, L6
L6:
j L12
L13:
addi $49, $0, 1
add $6, $49, $0
subi $4, $4, 1
add $4, $4, $0
L12:
addi $54, $0, 1
add $9, $54, $0
addi $56, $0, 0
sub $57, $4, $56
bgtz $57, L14
L15:
addi $59, $0, 0
add $9, $59, $0
L14:
addi $61, $0, 0
```



```
beq $9, $61, L11
```

```
L11:
```

10.6 函数嵌套调用

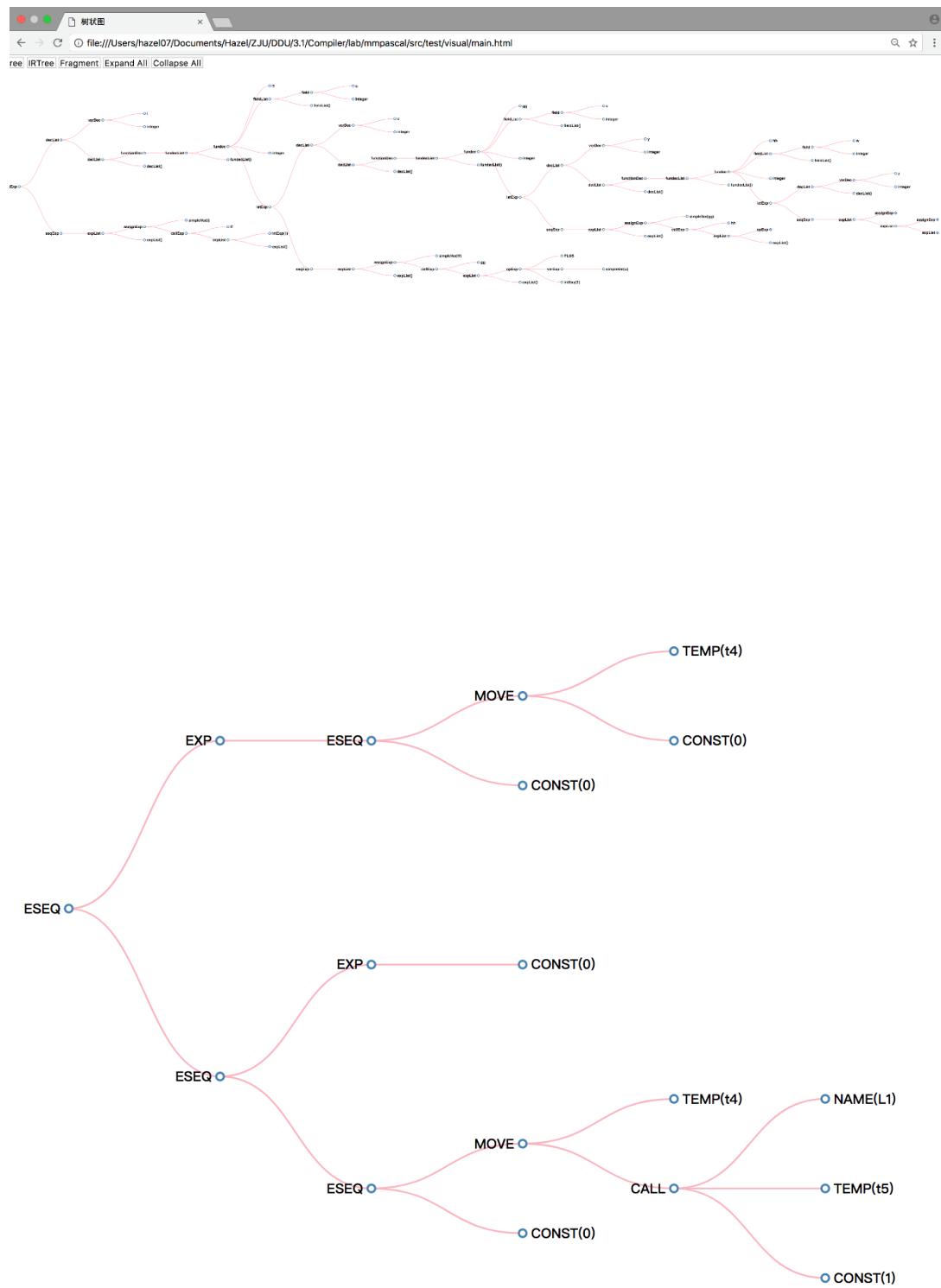
```
program a;

var
i:integer;
function ff(u:integer):integer;
var x:integer;
    function gg(v:integer):integer;
        var y:integer;
            function hh(w:integer):integer;
                var z:integer;
                begin
                    z:=10;
                    hh:=w;
                    x:=11;
                    y:=12;
                end;
                begin
                    gg:=hh(v+1);
                end;
begin
    ff:=gg(u+1);
end;

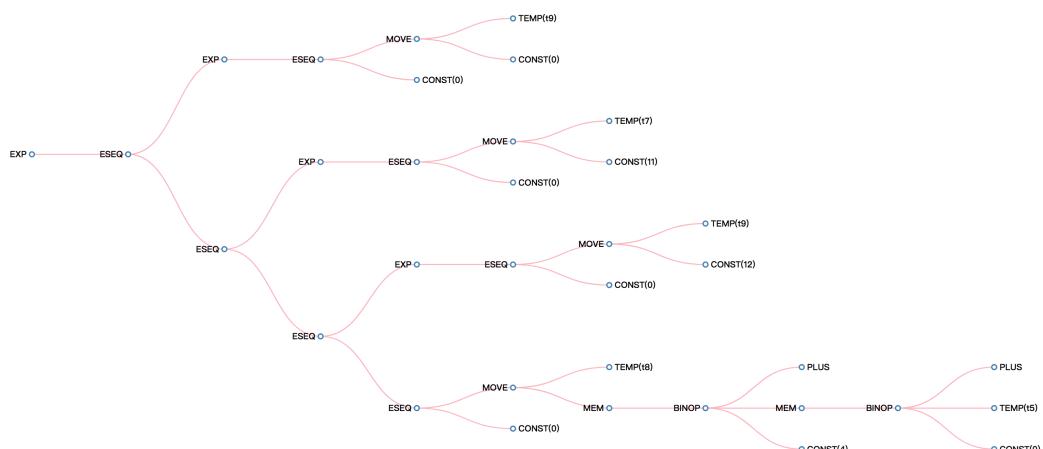
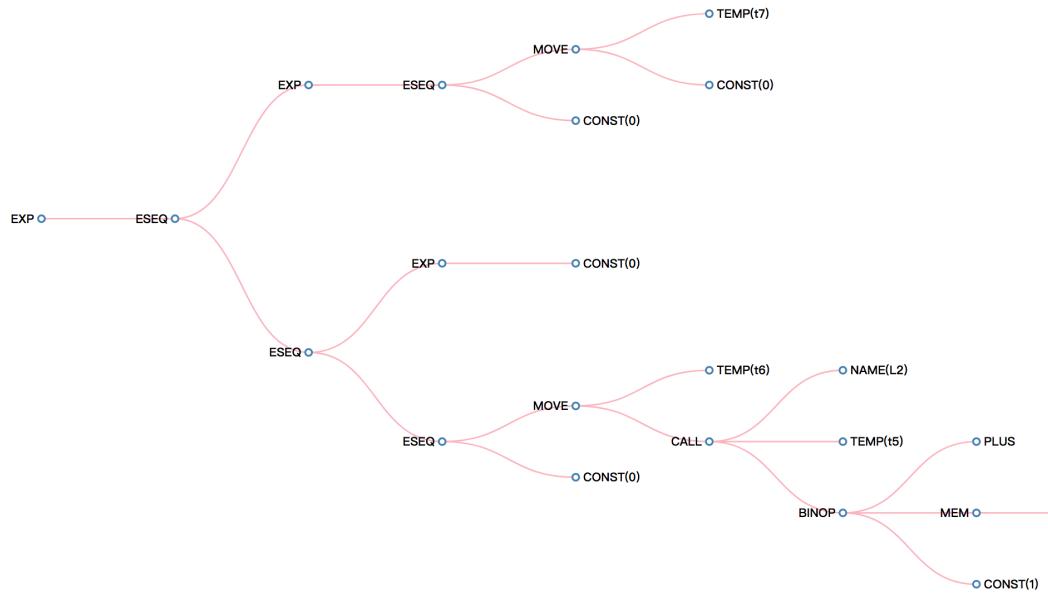
begin
    i:=ff(1);
end.
```

```
###AST done.###
varDeckKey: -380632912  functionDeckKey: -380632912  Key: -380632912      Key: -380627008      varDe
###AST2IRT done.###
Key: -380626480      Key: -380627264      Key: -380627968      Key: -380627968      Key: -380624880
Successful Write to "fragment.json"
Key: -380628208      Key: -380628208      Key: -380627968
Successful Write to "IRTree.json"
Key: -381669504      Key: -381666880      Key: -381666880      Key: -380626480      Key: -381669504
Key: -381666544      Key: -381654304      Key: -380628208      Key: -381666544      Key: -381654304
Successful Write to "../test/result.s"
```

抽象语法树表示：



Fragment 表示:



汇编结果

```
BEGIN L1
addi $11, $0, 0
add $7, $11, $0
lw $15, 0($1)
lw $18, 0($5)
lw $17, 4($18)
addi $17, $17, 1
sw $17, 4($15)
sw $5, 0($15)
subi $15, $15, 4
sw $15, 0($1)
addi $ra, 4($sp)
call L2
```



```
lw $27, 0($1)
addi $27, $27, 4
sw $27, 0($1)
add $6, $L2, $0
jr $ra
END L1
```

```
BEGIN L2
addi $29, $0, 0
add $9, $29, $0
addi $31, $0, 11
add $7, $31, $0
addi $33, $0, 12
add $9, $33, $0
lw $36, 0($5)
lw $35, 4($36)
add $8, $35, $0
jr $ra
END L2
```

```
main:
lui $4, 0x7fff
ori $4, $4, 0xf4c0
sw $4, 0($4)
addi $39, $0, 0
add $4, $39, $0
lw $43, 0($1)
addi $44, $0, 1
sw $44, 4($43)
sw $5, 0($43)
subi $43, $43, 4
sw $43, 0($1)
addi $ra, 4($sp)
call L1
lw $46, 0($1)
addi $46, $46, 4
sw $46, 0($1)
add $4, $L1, $0
```



10.7 错误测试

10.7.1 类型未定义

The screenshot shows the CLion IDE interface with the project structure on the left and the code editor on the right. The code editor displays a Pascal program named `test.pas`. A red box highlights the variable `S` in the declaration section:

```
program Loop;
var
    a: integer;
begin
    a := 10;
    repeat
        a := a + 1
    until a = 20;

    For a := 1 to 100 do
    Begin
        S := S+a;
    End;

    while a > 0 do
    Begin
        S := 1;
    End;
end.
```

The run configuration at the bottom of the editor is set to `/Users/zhaoxuandong/CLionProjects/mmpascal/src/cmake-build-debug/src ..test/test.pas`. A tooltip "Var Undefined: S" appears over the highlighted variable. The status bar at the bottom right shows the time as 9:17 and encoding as UTF-8.

10.7.2 函数未定义

The screenshot shows the CLion IDE interface with the project structure on the left and the code editor on the right. The code editor displays a Pascal program named `test.pas`. A red box highlights the function call `hh` in the declaration section:

```
program a;
var
    i:integer;
    function ff(u:integer):integer;
    var x:integer;
        function gg(v:integer):integer;
        var y:integer;
            begin
                gg:=hh(v+1);
            end;
        begin
            ff:=gg(u+1);
        end;
begin
    i:=ff(1);
end.
```

The run configuration at the bottom of the editor is set to `/Users/zhaoxuandong/CLionProjects/mmpascal/src/cmake-build-debug/src ..test/test.pas`. A tooltip "Semantic Error: Undefined Function hh" appears over the highlighted function name. The status bar at the bottom right shows the time as 13:10 and encoding as UTF-8.



10.7.3 变量使用错误

The screenshot shows the Clion IDE interface with the project structure on the left and the code editor on the right. The code editor contains the following Pascal-like code:

```
program arraytest;
type
  a = array[1..10] of integer;
var
  i:a;
  b:integer;
begin
  b[1]:=1;
  i[2]:=10;
  i[3]:=100;
end.
```

A red box highlights the assignment statement `b[1]:=1;`. In the run tab below, the output shows:

```
Invalid Subscript type
```

The code is then shown again with the error resolved:

```
type
  a = array[1..10] of integer;56  integer
var
  i:a;
  b:integer;
begin
  b[1]:=1;
  i[2]:=10;
  i[3]:=100;
end.
##AST done.##
typeDeckKey: -1329584080  Key: -1329584240  varDeckKey: -1329584240  varDeckKey: -1329584080  Key: -1329583664
Process finished with exit code 255
```

At the bottom, it says "Process finished with exit code 255".

10.7.4 类型不匹配

The screenshot shows the Clion IDE interface with the project structure on the left and the code editor on the right. The code editor contains the same code as the previous screenshot, but with a different error highlighted:

```
program arraytest;
type
  a = array[1..10] of integer;
var
  i:a;
  b:integer;
begin
  i:=10;
  i[3]:=100;
end.
```

A red box highlights the assignment statement `i[3]:=100;`. In the run tab below, the output shows:

```
Semantic Error: Type of Left Unmatched Type of Right for Assign
```

The code is then shown again with the error resolved:

```
type
  a = array[1..10] of integer;56  integer
var
  i:a;
  b:integer;
begin
  i:=10;
  i[3]:=100;
end.
##AST done.##
typeDeckKey: -54515664  Key: -54515824  varDeckKey: -54515824  varDeckKey: -54515664  Key: -54515440
Process finished with exit code 255
```

At the bottom, it says "Build finished in 139 ms (moments ago)".



10.7.5 函数参数错误

The screenshot shows the CLion IDE interface with a Pascal file named 'test.pas' open. The code defines a function gcd with two parameters, a and b, both of type integer. A red box highlights the line 'else gcd:=gcd(b);'. Below the code editor, the run configuration is set to 'src' and the command is 'program a;'. A red box also highlights the error message 'Semantic Error: Params Unmatched' in the terminal window. The terminal also displays the build log and some key values.

```
program a;
var
  i,j:integer;
function gcd(a:integer;b:integer):integer;
begin
  if b=0 then
    gcd:=a
  else
    gcd:=gcd(b);
end;
begin
  j:=gcd(36,24);
end.
```

Run: src
Program a;
Semantic Error: Params Unmatched

Var
i,j:integer;
Function gcd(a:integer;b:integer):integer;
Begin
 If b=0 Then
 Gcd:=a
 Else
 Gcd:=Gcd(b);
End;
Begin
 J:=Gcd(36,24);
End.
##AST done.##
VarDeckKey: 1463822384 VarDeckKey: 1463822384 FunctionDeckKey: 1463822384 Key: 1463822384 Key: 1463822384 Key: 1463826912 Key: 146382270

十、 总结

经过验证，本编译系统实现的功能如下：

- 实现基本类型的常量定义
- 实现所有基本类型，以及数组，自定义结构体的类型定义
- 实现算数表达式和逻辑表达式
- 实现变量赋值语句
- 实现 If-then ,If-then-else 条件语句
- 实现 while, for, repeat 循环语句
- 实现带基本类型参数的函数和过程
- 实现函数及过程的嵌套定义
- 实现子函数或子过程访问父函数或父过程变量
- 实现函数及过程的嵌套调用



十一、 分工

宋鼎：运行环境设计、转为目标代码

黄杨思博：语义分析、符号表设计

赵宣栋：词法分析、语法分析