

# 编译实习 MiniC 报告

罗昊 1700010686

2019 年 1 月 2 日

## 目录

<b>1</b>	<b>编译器概述</b>	<b>3</b>
1.1	额外支持的语法 . . . . .	3
1.2	错误报告 . . . . .	3
1.3	警告报告 . . . . .	4
<b>2</b>	<b>Eeyore</b>	<b>4</b>
2.1	简介 . . . . .	4
2.2	代码结构 . . . . .	4
2.3	BNF . . . . .	5
2.4	类结构 . . . . .	9
2.4.1	Symbol 类 . . . . .	9
2.4.2	SymbolTable 类 . . . . .	10
2.4.3	Node 类 . . . . .	11
2.4.4	Label 类 . . . . .	12
2.4.5	Gotolist 类 . . . . .	12
2.5	对于符号表管理的其他说明 . . . . .	13
2.6	对于回填法和逻辑运算的说明 . . . . .	13
2.7	对于逻辑表达式和算术表达式互相转化的说明 . . . . .	14
2.8	对于类型检查的说明 . . . . .	14
<b>3</b>	<b>Tigger</b>	<b>14</b>
3.1	代码结构 . . . . .	14
3.2	类结构 . . . . .	15

3.2.1	Expression 类 . . . . .	15
3.2.2	Func 类 . . . . .	16
3.3	程序流生成 . . . . .	17
3.4	return 语句检查 . . . . .	18
3.5	活性分析 . . . . .	18
3.6	死代码消除 . . . . .	18
3.7	常数传播 . . . . .	18
3.8	访存优化 . . . . .	19
3.9	寄存器分配 . . . . .	20
3.10	变量和符号说明 . . . . .	21
3.11	预着色节点的处理 . . . . .	21
3.12	合并传送节点的说明 . . . . .	21
3.13	形参的处理 . . . . .	21
3.14	调用函数 . . . . .	22
3.15	合并预着色点相关传送节点的例子 . . . . .	23
<b>4</b>	<b>Risc-v64/32</b>	<b>23</b>
<b>5</b>	<b>hcc</b>	<b>24</b>
<b>6</b>	<b>对工具软件的介绍</b>	<b>24</b>
6.1	GNU Bison . . . . .	24
6.2	flex . . . . .	25
<b>7</b>	<b>测试</b>	<b>25</b>
7.1	测试样例 1 – 快速排序 . . . . .	25
7.2	测试样例 2 – 局部数组 . . . . .	26
7.3	调试方法 . . . . .	27
7.3.1	getint 和 putint 和 putchar 函数的实现 . . . . .	27
7.3.2	Riscv-gcc 和 qemu-riscv64 的使用 . . . . .	27
7.4	发现课程所使用的工具中存在的问题 . . . . .	27
<b>8</b>	<b>实习总结</b>	<b>27</b>
8.1	收获与体会 . . . . .	27
8.2	对课程的建议 . . . . .	28

A hcc 脚本内容	29
B getint,putint,putchar 的实现	30
C hcc-run 测试脚本代码	31

## 1 编译器概述

hcc(Hao miniC Compiler) 是一个简单的 MiniC 编译器.

它由两部分组成: Eeyore 和 Tigger;

Eeyore 将 MiniC 代码转换为 Eeyore(一种三地址代码)

Tigger 将 Eeyore 代码转换为 Tigger 代码, Riscv64 asm 代码以及 Riscv32 asm 代码.

获取源码: <https://github.com/vangohao/hcc>

在 hcc 目录中运行 “make” 以编译

运行 “./hcc qsort.c” or “./hcc qsort” 将 qsort.c 编译为 qsort.s

添加 “-m32” 选项来生成 Riscv-32 代码 (默认生成 Riscv-64)

### 1.1 额外支持的语法

1. 支持空语句 ( ; )
2. 支持逻辑表达式与算术表达式互相自动转换.
3. 支持无返回值调用函数.
4. 支持调用函数时使用表达式作为参数.
5. 支持在程序体内声明函数.
6. 支持 C 风格多行注释和 C++ 风格单行注释.

### 1.2 错误报告

1. 报告语法错误和词法错误及其行号, 并给出该处正确的 token 类型提示.
2. 检查标识符使用, 如果使用了未定义的标识符会报错.
3. 检查标识符重复, 对于重复定义的报错, 函数名与变量名冲突的报错, 如果在 {} 程序块内使用与程序块外同名的变量, 则不会报错.

4. 检查函数参数表, 对于重复声明但参数表不一致, 或定义与声明参数表不一致, 或调用时所用的参数表与声明的类型不一致时报错.
5. 检查 `+`, `-`, `*`, `/`, `%` 运算符对于数组类型变量的不合法操作给出错误提示, 这些操作中除了 `(int[])+(int)`, `(int)+(int[])`, `(int[])-(int)`, 外涉及数组的运算都是不合法的.
6. 检查对数组变量的赋值, 无法将数值赋给数组变量.
7. 检查 `a[b]` 使用, 如果 `a` 不是数组变量, 会报错.

### 1.3 警告报告

1. 控制流到达函数结尾 (Control reaches end of function)

## 2 Eeyore

### 2.1 简介

1. 使用 flex, bison 和 C++ 构建, 将输入的 MiniC 代码转换为 Eeyore 三地址代码.
2. 使用 STL 的 map 模板制作符号表, 使用链表串联内外层程序块的符号表.
3. 使用回填法构建 eeyore 中的标号及 goto 语句.

### 2.2 代码结构

代码由 `symbol.cpp/h` `node.cpp/h` `gotolist.cpp/h` `fault.cpp/h` `main.cpp` `parser.y` `lexer.l` 构成

1. `symbol.cpp/h` 包含两个类的定义:
  - (1) Symbol 类, 即符号类, 每个 Symbol 对象对应一个符号 (包括函数, 常数, 原生变量和临时变量)
  - (2) SymbolTable 类, 即符号表.
2. `node.cpp/h` 包含两个类的定义:
  - (1) Node 类, 即语法树节点.
  - (2) Label 类, goto 语句后面的标识 (对于非 M 语句) 或标号 (对于 M 语句), 用于建立跳转关系

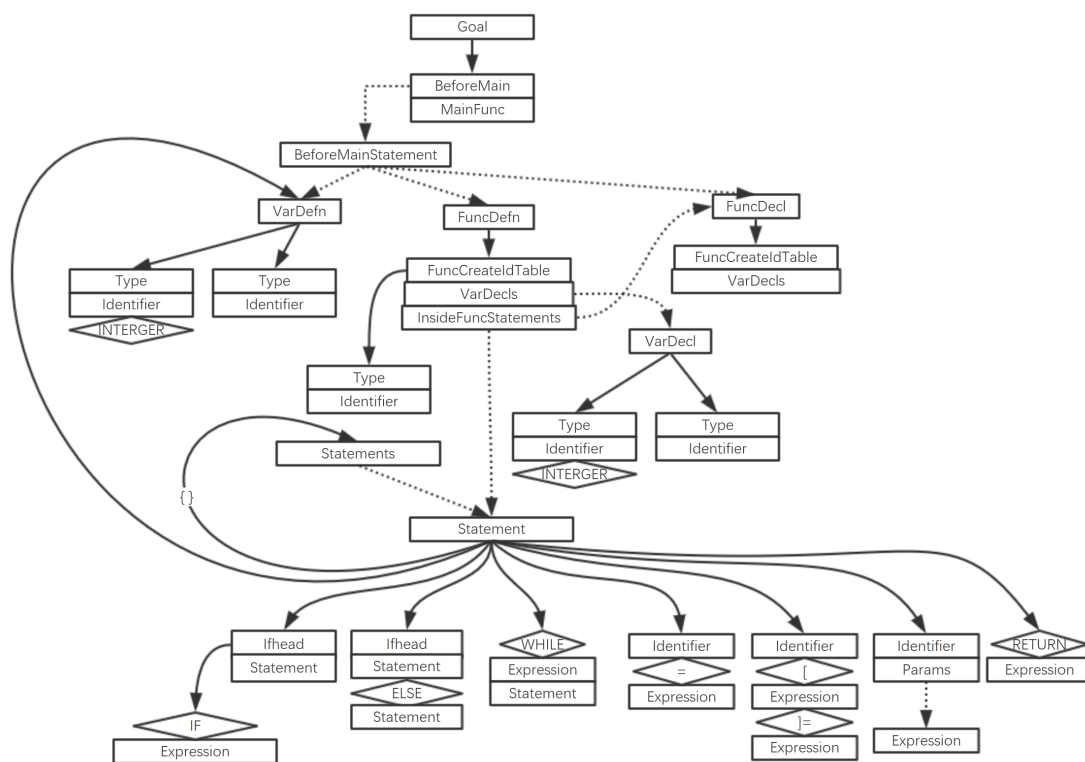
3. gotolist.cpp/h 包含 Gotolist 类的定义, gotolist 就是跳转表, 用于建立跳转关系.
4. fault.cpp/h 包含用于报错的函数.
5. main.cpp 主程序
6. parser.y 语法分析器, 进行语法制导翻译.
7. lexer.l 词法分析器, 进行词法分析并传递 token 及其基本属性给语法分析器

,

## 2.3 BNF

示意图, 省略了部分终结符号和 Expression 的图示.

$A \rightarrow B$  虚线表示  $A : A B$



下面给出 BNF(实际是删除了所有处理语句的 parser.y 代码)

```
1 %right '=='
2 %left LOR
```

```

3 %left LAND
4 %left EQUAL NOTEQUAL
5 %left '<' '>'
6 %left '+' '-'
7 %left '*' '/' '%'
8 %right NEGA '!'
9 %right '['
10
11 %%
12 Goal: BeforeMain MainFunc
13 ;
14 BeforeMain: BeforeMain BeforeMainStatement
15 | %empty
16 ;
17 BeforeMainStatement: VarDefn | FuncDefn | FuncDecl | ';'
18 ;
19 VarDefn: Type Identifier ';'
20 | Type Identifier '[' 'INTEGER' ']' ';'
21 ;
22 VarDecls: VarDecls ',' VarDecl
23 | VarDecl
24 | %empty
25 ;
26 VarDecl: Type Identifier
27 | Type Identifier '[' 'INTEGER' ']'
28 | Type Identifier '[' ']'
29 ;
30 FuncCreateIdTable: Type Identifier '('
31 ;
32 FuncDefn: FuncCreateIdTable VarDecls ')'
33 '{'
34 InsideFuncStatements M
35 '}'
36 ;
37 InsideFuncStatements: InsideFuncStatements M FuncDecl
38 | InsideFuncStatements M Statement
39 | FuncDecl
40 | Statement
41 ;
42 FuncDecl: FuncCreateIdTable VarDecls ')' ';'
43 ;
44 MainFunc: T_INT MAIN '(' ')'
45 '{' InsideFuncStatements M '}'
46 ;
47 Type: T_INT
48 ;
49 Statements: Statements M Statement
50 | Statement
51 ;
52 M: %empty
53 ;
54 N: %empty
55 }
56 ;

```

```

57 Ifhead: IF '(' Expression ')'
58 ;
59 Statement: '{' Statements '}'
60 | ';'
61 | Ifhead M Statement
62 | Ifhead M Statement N ELSE M Statement
63 | WHILE M '(' Expression ')' M Statement
64 | Identifier '=' Expression ';'
65 | Identifier '[' Expression ']' '=' Expression ';'
66 | Identifier '(' Params ')' ';'
67 | VarDefn
68 | RETURN Expression ';'
69 | error ';'
70 ;
71 Expression: Expression '+' Expression
72 | Expression '-' Expression
73 | Expression '*' Expression
74 | Expression '/' Expression
75 | Expression '%' Expression
76 | Expression '<' Expression
77 | Expression '>' Expression
78 | Expression EQUAL Expression
79 | Expression NOTEQUAL Expression
80 | Expression LAND M Expression
81 | Expression LOR M Expression
82 | Expression '[' Expression ']'
83 | INTEGER
84 | Identifier
85 | '!' Expression
86 | '-' Expression %prec NEGA
87 | Identifier '(' Params ')'
88 | '(' Expression ')'
89 ;
90 Params: Params ',' Expression
91 | Expression
92 | %empty
93 ;
94 Identifier: IDENTIFIER
95
96 %%

```

对以上 BNF 的说明:

- (1) 使用 bison 编译时会报一个 Shift/reduce 冲突, 这是因为 If 和 IfElse 有所冲突, 但由于 Bison 的规则是冲突时优先选择移入, 所以在保留此冲突的情况下编译的语法分析器仍然是正确的. 例如下面这段代码:

```

1  if (x < 5)  if (x<3)  y = 1; else y = 2;

```

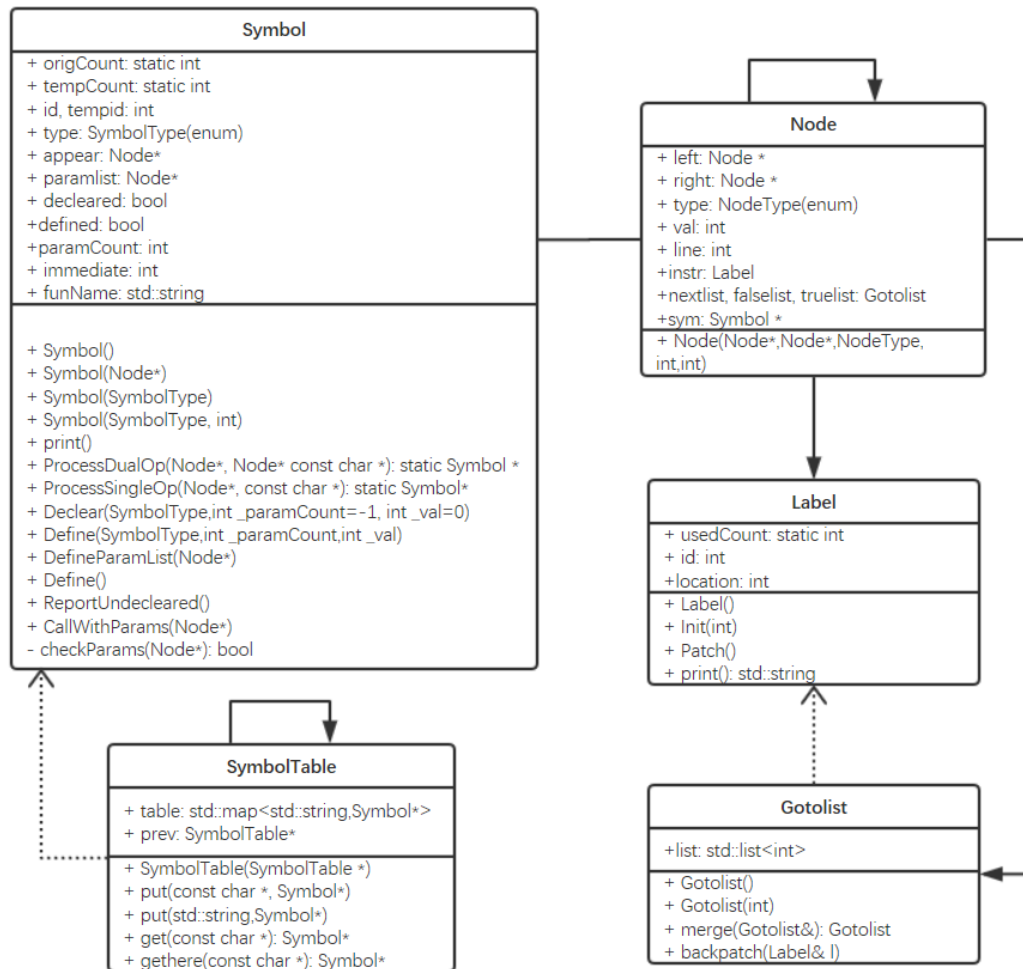
其中的 else 应该与第二个 if 构成 IfElse, 而第一个 if 是单独的 If, 没有 Else 与其配对.

所以, 对于  $x == 4$  运行结果应该是  $y = 2$ ;

- (2) FuncCreateIdTabel 非终结符是 FuncDefn 和 FuncDecl 的公共部分, 由于程序需要在这部分之后首先为函数创建符号, 所以需要加上控制代码, 为了避免冲突, 所以将这部分单独拿出来加上控制代码; 类似的, Ifhead 非终结符对 if 的条件表达式先做处理, 为了避免冲突, 所以将这部分单独拿出来加上控制代码.
- (3) 对 M 和 N 的说明: M 和 N 是用于流程控制的占位符, 其中 M 将成为语句标号. 本 MiniC 转 Eeyore 编译器使用回填法<sup>[1]</sup>处理 while 语句,if 语句以及逻辑运算符涉及的条件跳转. 具体作用后面会讲.



## 2.4 类结构



下面简要介绍一下以上几个类的组成和作用

### 2.4.1 Symbol 类

Symbol 类是符号类.

符号的类型定义在 enum SymbolType 中:

```

1 enum SymbolType //符号类型
2 {
3     Int=0,        //整形变量
4     IntPtr=1,     //整形数组
5     FunPtr=2,     //函数
6     Immediate=3,  //直接数

```

```

7   Logic=4,      //逻辑表达式
8   Error=5,
9 };

```

符号类主要功能是处理表达式运算和函数调用

- (1) 两个静态方法 ProcessDualOp 和 ProcessSingleOp 传入两个 Node\* 类型节点, 返回一个 Symbol 指针, 指向运算所得的临时变量, eeyore 在翻译每个运算时都会创建一个临时变量, 将运算结果赋给临时变量.
- (2) Declclear 方法配合 decleared 属性, 由于 parser 第一次遇到一个令牌 IDENTIFIER 并不是在 VarDefn 非终结符中, 而是在 identifier 终结符中, 因此在此时就会将新遇到的符号插入当前符号表 top 中, 而在 VarDefn 中, 调用 Declclear 方法将 decleared 属性设为 true. 在 Expression 中遇到相应符号时, 需要检查符号的 decleared 属性, 必须是 true, 否则会报错. 对于函数, Declclear 方法会存入参数表.
- (3) Define 方法, 适用于函数, 作用是声明函数已定义, 若此函数已被声明, 则还需检查此处传入的参数表是否与声明时使用的参数表相匹配, 若不匹配则会报错. 但函数声明和定义都不是调用的必要条件, 没有声明的函数也可以直接调用, 但已经声明或定义的函数在调用时会通过 checkParams 方法检查参数表是否匹配.
- (4) CallWithParams 方法, parser 在遇到函数调用语句时调用, 作用是按照函数的声明或定义状态检查参数表并给出相应的报错.
- (5) checkParams 方法, 这是个 private 方法, 作用是检查传入的参数表是否与 paramList 相匹配. 参数表的类型是 Node\*, 也就是 VarDecl 对应的语法树节点. 检查的方法是遍历链表, 检查节点的 sym 属性的 type 属性是否对应或可以隐式转换, 不匹配的给出错误提示.

## 2.4.2 SymbolTable 类

SymbolTable 类是符号表, 有两个属性, 分别是 std::map<std::string, Symbol\*> 类型的 table 表和 SymbolTable\* 类型的 prev, 指向上一个符号表, 所以 SymbolTable 实际是一个链式栈.

下面介绍 SymbolTable 支持的主要方法

- (1) put 方法, 参数为一个字符串和一个 Symbol 指针, 作用是将该符号插入符号表.
- (2) get 方法, 参数是一个字符串, 返回值是一个 Symbol 指针, 将首先在当前符号表中查找该符号, 如果没有, 则调用 prev 即上一级符号表的该方法, 如果整个符号表中都找不到该符号, 则返回 null

- (3) gethere 方法, 在当前符号表中查找该符号, 如果没有就返回 null, 不会递归查找上一级符号表.

对于符号表管理的其他说明

- (1) 程序开始时有创建一个全局符号表 top;
- (2) 语法分析器处理到函数开头 int func( 以及其他程序块开头时会创建一个新的符号表 (其 prev 属性指向上一级符号表), 将旧的符号表保存到栈中;
- (3) 函数结束 (对于声明, 以 “);” 作为结束, 对于函数定义, 以 “}” 作为结束) 或程序块结束时, 将栈顶的符号表弹出, 覆盖当前符号表.
- (4) 此外, 注意函数参数中的符号与函数体中的符号处于同一个符号表中

### 2.4.3 Node 类

Node 类是语法树的节点类, 实质是一个结构体, 没有实质性的方法.

Node 节点的种类用 enum NodeType 类型定义如下:

```
1  enum NodeType //语法树节点所对应语句类型
2  {
3  ExprArith=0,          //数值表达式
4  ExprLogic=1,          //逻辑表达式
5  Assign=2,             //赋值语句
6  If=5,                 //If语句
7  IfElse=6,            //IfElse语句
8  While=7,              //While语句
9  Return=9,             //Return语句
10 Vardfn=10,            //变量定义语句
11 Fundcl=11,            //函数声明语句
12 Fundfn=12,            //函数定义语句
13 MainFun=13,           //main函数
14 ArrayAssign=14,       //数组赋值语句
15 Symbol1 = 15,         //未定类型的临时符号
16 Params=16,            //参数表(调用时)
17 Funcall=18,           //函数调用语句
18 VarDcl = 20,          //参数表(定义或声明时)
19 Empty = 21,           //空类型
20 Stmts = 22,           //程序体
21 };
```

Node 类包含的属性如下:

- (1) left, right Node 指针, 指向左子节点和右子节点
- (2) type, NodeType 类型, 节点类型.

- (3) val int 类型, 用于储存属性值, 例如函数节点的参数个数, Params 节点的当前参数个数等, 此外, 在 identifier 中, 该属性还指示该符号是在最顶层符号表中找到的还是上级符号表中找到的, 以便后续必要时进行覆盖定义. 该属性为综合属性.
- (4) line int 类型, 储存该节点在源文件中的代码行号
- (5) instr Label 类型, goto 语句后面的标识 (对于非 M 语句) 或标号 (对于 M 语句), 用于跳转
- (6) nextlist, truelist, falselist, 是 Gotolist 类型指针, 实际是跳转表, 分别储存跳转到当前语句结尾、当前语句为真位置、当前语句为假位置的 goto 指令, 用于回填. 该属性为综合属性.
- (7) sym 是 Symbol 指针类型, 保存当前节点对应的符号.

#### 2.4.4 Label 类

Label 类是 goto 语句后面的标识 (对于非 M 语句) 或标号 (对于 M 语句). 属性如下:

- (1) int 类型 id 是标号的编号 (对于 M 语句), 对于其他语句则为-1
- (2) static int 类型 usedCount 是 Label 的计数, 用于编号.
- (3) int 类型 location 是该 goto 语句或标号在生成代码序列中的编号, 用于回填

下面是方法

- (1) Init(int) 方法, 初始化当前 Label, 传入的参数为 location
- (2) Patch() 方法, 只有标号会调用, 作用是在生成的代码序列中补充显示该标号.

#### 2.4.5 Gotolist 类

属性只有一个, 是 std::list<int>, 储存的是 goto 语句在生成代码序列中的编号.  
方法如下:

1. Gotolist(int) 初始化, 构造函数, 使 list 属性初始化为一个值的列表
2. merge(Gotolist&) 返回值是 Gotolist, 用于合并两个 Gotolist
3. backpatch(Label\* l) 这个函数非常重要, 用于回填, 作用是对于 list 中的每个元素 i, 对于其每个元素调用全局函数 Output::patch(int i, Label & l); 对于非空的 list, 一并调用 l 的 Patch() 函数显示标号.

## 2.5 对于符号表管理的其他说明

- (1) 程序开始时有创建一个全局符号表 `top`;
- (2) 语法分析器处理到函数开头 `int func(` 以及其他程序块开头时会创建一个新的符号表 (其 `prev` 属性指向上一级符号表), 将旧的符号表保存到栈中;
- (3) 函数结束 (对于声明, 以 “);” 作为结束, 对于函数定义, 以 “}” 作为结束) 或程序块结束时, 将栈顶的符号表弹出, 覆盖当前符号表.
- (4) 此外, 注意函数参数中的符号与函数体中的符号处于同一个符号表中

## 2.6 对于回填法和逻辑运算的说明

回填法用于解决语法制导翻译过程中生成的 `goto` 语句跳转到的编号不确定的问题.<sup>1</sup>

下面分几个部分说明:

- (1) M 占位符的作用:

M 占位符是标号位置, 翻译到 M 占位符时会在该位置初始化一个标号.

- (2) N 占位符的作用:

N 占位符是 `goto` 语句, 翻译到 N 占位符时会将该 Node 的 `instr` 属性初始化为一个 `goto` 语句, 也就是说, 该处必有跳转. 这只用于含有 `else` 的 `if` 语句, 因为执行完 `true` 的代码块以后必须跳过后面的 `else` 代码块;

- (3) Gotolist 的作用, 每个逻辑表达式都有他的 `nextlist`, `truelist` 和 `falselist`(实质是综合属性); 分别对应于一个标号 (也即 M 占位符), 内容是这三个标号对应的 `goto` 语句的集合 (在生成代码序列中的编号)
- (4) 对于比较运算符, 对其生成 `eyore` 中的 `if` 语句, 并初始化两个 `goto` 语句对应 `true` 和 `false`, 初始化 `truelist` 和 `falselist` 包含分别包含对应的 `goto` 语句, 对于比较运算符表达式, `nextlist` 为空.
- (5) 对于逻辑运算符 (包括与, 或, 非), 对应的调用 `merge` 函数由子节点的三个 `list` 计算出父节点的三个 `list`. 很显然, 这一步过程同时能够完成所谓的短路现象.
- (6) 对于回填过程整个 `If, IfElse` 或 `While` 程序块处理完后即可进行调用三个 `gotolist` 的 `backpatch` 函数进行回填操作, 该函数调用全局函数 `Output::patch(int i, Label & l)`; 这个全局

---

<sup>1</sup>参考了《编译原理 (第 2 版)》中第六章 6.7 节

函数找到生成代码序列中的第  $i$  项 (必然以 `goto` 结尾), 在其后添加上跳转到的标号 1, 即完成了标号回填的过程.

## 2.7 对于逻辑表达式和算术表达式互相转化的说明

本编译器将比较表达式以及或与非表达式视为逻辑表达式, 五则运算表达式视为算术表达式, 支持逻辑表达式和算术表达式的自动转化.

- (1) 逻辑表达式转算术表达式: 生成一个临时变量  $x$  表示结果, 将逻辑表达式的 `truelist` 指向运行  $x=1$ , 然后跳过后面一句  $x=0$ ; 将逻辑表达式的 `falselist` 指向运行  $x=0$ .
- (2) 算术表达式转逻辑表达式: 设算数表达式对应的符号为  $x$ , 则将该语句翻译为和逻辑比较表达式  $x==0$  相同的翻译结果. 即为其生成 `if-goto-goto` 语句, 并将 `truelist` 初始化为第一个 `goto` 对应的序号, 将 `falselist` 初始化为第二个 `goto` 对应的序号.

## 2.8 对于类型检查的说明

1. 对于指针和整数隐式转换: 检查 `+, -, *, /, %` 运算符对于数组类型变量的不合法操作给出错误提示, 这些操作中除了 `(int[]) + (int)`, `(int) + (int[])`, `(int[]) - (int)`, 外涉及数组的运算都是不合法的.
2. 对于函数调用的参数表, 不支持指针类型隐式转换为整数类型, 会按照每个参数依次比对. 比对的方法为: 链表遍历法: 函数的 `Symbol` 中存有该函数的参数表, 形式为语法树子树 `Node*`, 调用函数时, 将调用时的节点为根的子树传入, 这两棵子树实际都是链表 (每个节点只有一个孩子), 遍历链表比较他们的 `sym` 属性类型, 全部一致方可调用. 此外, 参数不仅是变量, 可以是任何类型匹配的表达式 (包括常数)
3. 不允许对 `int[]` 类型变量赋值

# 3 Tigger

## 3.1 代码结构

代码由 `analyz.cpp/h` `lexer.l` `tigger.y` `tigger.cpp/h` `main.cpp` 构成  
`analyz.cpp`: 包含三个类的定义:

1. `Expression` 类, 每个 `Expression` 对象对应一条 Tigger/RiscV 指令;
2. `Func` 类, 每个 `Func` 对象对应一个函数;

3. Analyz 类, 单例类, 负责整体处理工作以及函数间优化工作.

analyz.h: 包含上述三个类的声明.

lexer.l: 词法分析器, 负责词法分析, 其中对于变量, 区分全局变量, 局部变量和形参, 使用 Analyz 中的 vcount 属性为变量分配 id, 并包含在 yylval 中.

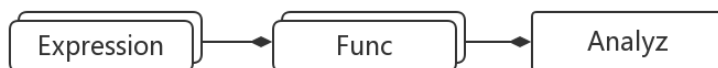
tigger.y: 语法分析器, 负责语法分析, 负责将 Eeyore 语句转换为 Expression 对象, 以及负责 Func 类对象创建, Func 类对象负责 tigger 的主要函数体内工作.

tigger.cpp: 包含一些公共函数.

tigger.h: 公共头文件.

main.cpp: 主程序函数和少量全局变量.

## 3.2 类结构



### 3.2.1 Expression 类

```
1 class Expression
2 {
3 public:
4     ExprType type;           //类型
5     bool isMove;             //是否为传递指令
6     bool dead;               //是否为死代码
7     bool visited;            //是否被访问过(用于活性分析)
8     vector<int> left;         //左值变量
9     vector<int> right;        //右值变量
10    vector<int> imm;           //直接数
11    vector<int> use;           //使用的变量集合
12    vector<int> def;           //定义的变量集合
13    set<int> in;               //入口活跃集合
14    set<int> out;              //出口活跃集合
15    list<Expression*> nexts;    //下一条指令集合
16    list<Expression*> prevs;    //上一条指令集合
17    map<int,int> constant;      //常数变量表,用于常数传播
18    string funtocall;           //仅用于 call 语句
19    //构造函数,用于创建 Expression 并自动将指针加入当前所在函数的 exprs 表
20    Expression(ExprType _type,vector<int> _left,
21               vector<int> _right,vector<int> _imm,
22               string _funtocall="",bool push=true);
23 };
```

### 3.2.2 Func 类

```
1  class Func
2  {
3  public:
4      Func(int _paramCount, string _name); //构造函数
5      void Processor(); //主函数
6      int insert(int s, int v); //向栈空间添加变量, s为大小,
7                                //v为变量id, 返回值为栈上的编号
8      void ReturnFunc(int v, int t); //处理Return语句, v为变量id或常数值,
9                                //t为选项(0表示常数, 1表示变量)
10     void CallParam(int v, int t); //处理Param语句, v和t的含义同上
11     void CallFunc(int v, string f); //处理call语句, v为存返回值变量, f为函数名称
12     int getParamVar(int r); //获取形参对应的局部变量编号
13     friend class Expression; //将Expression声明为友元
14 private:
15     int paramCount; //参数数量
16     int paramToCallWithCount; //调用参数计数器
17     int frameSize; //栈空间大小(不含保存调用者保存寄存器的临时空间)
18     int frameMaxSize; //栈空间大小
19     unordered_map<int, int> frameSaveTable; //记录被调用者保存寄存器的保存位置
20     unordered_map<int, int> frameArrayTable; //从栈上数组变量id映射到栈上位置
21     string name; //函数名称
22     vector<int> offset; //栈上数据位置
23     vector<int> size; //栈上数据大小
24     vector<int> paramTable; //形参对应局部变量表
25     list<Expression*> exprs; //语句表
26     vector<int> spilledVariableFrameMap; //由变量id映射到上面的offset和size数组的下标
27
28
29     //Color Algorithm图染色算法
30     static int colorNumber; //颜色数
31     list<int> initial; //待初始化的节点表
32     list<int> simplifyWorklist; //低度数传递无关节点表
33     list<int> freezeWorklist; //低度数传递相关节点表
34     list<int> spillWorklist; //高度数节点表
35     list<int> spilledNodes; //溢出节点
36     list<int> coalescedNodes; //已合并节点
37     list<int> coloredNodes; //已染色节点
38     list<int> selectStack; //栈
39     list<Expression*> coalescedMoves; //已合并传递指令
40     list<Expression*> constrainedMoves; //已约束传递指令
41     list<Expression*> frozenMoves; //已冻结传递指令
42     list<Expression*> worklistMoves; //待合并的传递指令
43     list<Expression*> activeMoves; //活跃的传递指令
44     vector<vector<int>> adjMatrix; //邻接矩阵
45     vector<list<int>> adjList; //邻接表
46     vector<int> degrees; //度
47     vector<int> alias; //别名
48     vector<int> color; //颜色
49     vector<NodeStatus> status; //顶点状态
50     vector<list<Expression*>> useList; //使用该变量的指令表
51     vector<list<Expression*>> defList; //定义该变量的指令表
```



```

52 vector<list<Expression*>> moveList; //与该变量有关的传递指令表
53
54
55 void ColorAlgorithmMain();           //图染色主函数
56 void AddEdge(int x,int y);           //建图添加边
57 void livelyAnalyz();                 //活性分析
58 void InitColorAlgorithm();           //初始化
59 void DecrementDegree(int m);          //顶点度数减1
60 bool MoveRelated(int n);              //n是否是传递相关的
61 list<Expression*>& NodeMoves(int n); //与n相关的传递指令(未冻结的)
62 list<int>& Adjacent(int n);           //n的邻点集
63 void Simplify();                     //简化
64 void Coalesce();                     //合并
65 int GetAlias(int x);                 //获取别名(由合并产生)
66 void AddWorklist(int u);             //加入工作表
67 bool TestPrecoloredCombine(int u/*precolored*/,int v); //测试预着色节点相关传递指令是否可以合并
68 bool TestConservative(int u,int v); //测试传递指令是否可以保守合并
69 void Combine(int u,int v);           //合并
70 void EnableMoves(int m);             //将m相关的传递指令设为待合并的
71 void FreezeMoves(int u);            //将u相关的传递指令冻结
72 void FreezeAction();                 //冻结
73 void SelectSpill();                  //选择高度数节点溢出
74 void AssignColors();                 //分配颜色
75 void RewriteProgram();               //重写程序,对于有真实溢出的情况
76 void InsertExprForWrite(Expression* e,int v); //插入栈内存写入
77 void InsertExprForRead(Expression* e,int v);  //插入栈内存读取
78 int GenTempVariable();               //获取一个新临时变量的id
79
80 //程序流处理及优化
81 void InitializeVectorSpace();         //初始化
82 void InitFunEnv();                   //函数入口形参处理
83 int insert();                         //添加一个int变量到栈中
84 void frameFree();                    //释放栈的最后一个空间
85 void SaveReg();                      //call语句出保存调用者保存的寄存器
86 void OptimizeFlow();                 //优化程序流(常数传播)
87 void OptimizeDead();                 //死代码消除
88 void OptimizeLoadStore();            //优化
89 void genFlow();                      //生成程序流
90
91 //生成代码
92 string opstring(int op);              //获得op对应的运算符
93 string opinstruct(int op);           //获得op对应的RiscV指令
94 void OutputArithRIMul(int reg1,int reg2,unsigned imm); //处理ArithRI指令的RiscV输出(将乘以2的幂改为左移)
95 void GenCode();                      //生成tiger代码
96 void GenRiscv64();                   //生成RiscV64代码
97 void GenRiscv32();                   //生成Riscv32代码
98 void checkReturn();                  //检查函数退出前是否有返回指令
99 };

```

### 3.3 程序流生成

Parser 分析到函数体结尾 end func 时会生成一个 Empty 类型语句表示函数体的结束.

程序流分析过程对 Expression 对象的 nexts 属性和 prevs 属性赋值, 使其分别指向 e 的所有前驱和所有后继.

除 Goto, If 类, Return, Empty 语句外, 所有语句在  $\text{vector}\langle\text{Expression}^*\rangle$  exprs 中的后继都是其程序流中的后继 (称为默认后继).

对于 Goto 语句, 其后继为 Goto 所指向标号对应的 Label 语句;

对于 If 类语句, 其后继为默认后继以及所指向标号对应的 Label 语句;

对于 Return 语句, 其后继为表示函数体结束的 Empty 语句.

### 3.4 return 语句检查

如果函数结尾处的 Empty 语句有不是 return 语句的前驱, 则说明程序控制流到了函数末尾, 会给出警告;

并会自动在函数结尾处添加一条 `return 0;`

### 3.5 活性分析

以语句为基本块进行自下而上的活性分析. 使用队列进行宽度优先搜索, 递推式为:

$$e.in = e.use \cup (e.out - e.def)$$

$$e.out = \bigcup_{x \in e.nexts} (x.in)$$

如果进行上述计算后 e.in 和 e.out 有所改变, 则将 e 的前驱加入队列.

迭代直到队列为空.

### 3.6 死代码消除

此处处理的死代码分为两种:

(1)  $e.def \cap (\bigcup_{x \in e.nexts} x.in) = \Phi$  即该语句定义的变量在这条语句的所有后继中都不是入口活跃的.

(2) 不可达语句, 即在活性分析时未处理到的语句.

对于第 (1) 情形, 未定义变量的语句和控制流跳转类语句为例外排除.

### 3.7 常数传播

每个 Expression 语句对象都有一个 constant 属性, 他是一个 map 表, 从变量 id 映射到该处 (出口处) 的变量常数值. 此表只会储存此处为常数值的变量.

按顺序遍历所有语句, 对于每个语句, 计算一个  $mp$ (map 表)

$$mp = \bigcap_{x \in e.prevs} x.constant$$

即  $mp$  为  $e$  的所有前驱的  $constant$  表中项的交集, 也即对于所有前驱语句都具有相同常数值  
的变量.

对于非跳转类或访存类语句, 如果  $x \in mp \forall x \in e.use$  成立, 也即所有被  $x$  使用的变量在此  
处均为常数, 则可说明此语句定义的变量也为常数. 调用 `calcarith` 函数计算此算术运算. 并将  
该语句的类型改为 `MoveRI`(即将直接数赋值给变量的语句)

对于上面公式不成立的语句, 以及跳转类和访存类语句, 令

$$mp = mp - e.def$$

即此处被定义的变量标记为不是常数.

最后令  $e.constant = mp$  记录此语句处的常量表

### 3.8 访存优化

死代码消除和常数传播的优化是在寄存器分配之前就完成的, 而访存优化主要是优化对局  
部变量的访存指令. 考虑这两种情况: (1) 当寄存器不够用时, 图染色算法将局部变量溢出至栈  
帧, 然后 `RewriteProgram` 过程会为每次对该局部变量的读取增加 `FrameLoad` 指令, 对修改增  
加 `FrameStore` 指令.(2) 连续的 `call` 语句时, 调用者保存的寄存器会被连续存取但却不会改变.

以上两种情况可能产生一些冗余的访存指令.

此处使用两种规则优化:

(1) 设本条语句 `FrameLoad n x`. 若当前寄存器  $x$  在上一句 `FrameLoad n x` 后未改变, 且  
上条语句和本条语句之间没有跳转类语句, 标号或者数组写入语句, 则当前 `FrameLoad` 语句可  
被优化.

(2) 设本条语句 `FrameStore n x`. 若当前寄存器  $x$  在上一句 `FrameLoad n x` 后未改变, 且  
上条语句和本条语句之间没有跳转类语句, 标号或者数组写入语句, 则当前 `FrameStore` 语句和  
之前的 `FrameLoad` 语句都可被优化.

举例:

(1)

```
1 //此处设a保存在栈帧中
2 c = a + b;
3 d = a + c;
4 e = c + d;
```

优化前的 `tigger` 代码:

```

1  load  0 a0
2  a1 = a0 + a1
3  load  0 a0
4  a1 = a0 + a1
5  a1 = a1 + a1

```

优化后的 tigger 代码

```

1  load  0 a0
2  a1 = a0 + a1
3  a1 = a0 + a1
4  a1 = a1 + a1

```

(2)

```

1  b = a + 2;
2  d = func(1);
3  d = func(2);
4  return b;

```

优化前的 tigger 代码:

```

1  a1 = a0 + 2
2  a0 = 1
3  store a1 0
4  call func
5  load 0 a1
6  a0 = 2
7  store a1 0
8  call func
9  load 0 a1
10 a0 = a1
11 return

```

优化后的 tigger 代码

```

1  a1 = a0 + 2
2  a0 = 1
3  store a1 0
4  call func
5  a0 = 2
6  call func
7  load 0 a1
8  a0 = a1
9  return

```

### 3.9 寄存器分配

使用图着色算法进行寄存器分配.<sup>2</sup>

利用活性分析的结果构造冲突图, 兼用邻接表和邻接矩阵表示冲突图.

图染色有四种操作:

---

<sup>2</sup>寄存器分配的算法参考了《现代编译原理-C 语言描述 (修订版)》第 11 章

1. 简化: 对于度数小于  $k$  的传送无关的点, 将其从图中删除并放入栈中.
2. 合并: 对于一条传送指令, 对其关联的两个点在保守规则下合并.
3. 冻结: 冻结一条传送指令, 将他当作非传送指令.
4. 溢出: 将图中度数最大的点放入栈中, 并从图中删除.

操作按 1.2.3.4 的优先级进行, 直到图中不再有点.

将栈中元素依次弹出, 并为其分配颜色, 如果无颜色可用, 则将该顶点真实溢出.

若有真实溢出发生, 则重写程序, 为溢出的变量分配栈空间, 加入 load 语句和 store 语句来读取和写回, 并创建临时变量进行运算.

### 3.10 变量和符号说明

Tigger 不再创建符号表, 因为其中所有变量名称的作用域都是全局, 所以直接由词法分析器为其分配一个 id(>27).

全局变量和栈数组不分配寄存器, 直接存到内存中.

### 3.11 预着色节点的处理

Tigger 有 28 个寄存器, 给其确定 id 为 0-27, 以作为预着色节点, 在活性分析中与其他变量处于同等地位. 在图着色时也可与与其相关的传送节点合并. 但两个不同色的预着色节点无法合并.

### 3.12 合并传送节点的说明

1. 对于关联两个普通节点的传送指令, 合并这两个节点的条件是:  
如果节点  $a$  和  $b$  合并产生的节点  $ab$  的高度数 ( $\geq K$ ) 的邻节点的个数小于  $K$ , 则认为  $a$  和  $b$  的合并是保守的.
2. 对于一个普通节点  $a$  和一个预着色节点  $b$  的合并, 合并条件是:  
对于  $a$  的每一个邻接点  $t$ , 或者  $t$  与  $b$  已有冲突, 或者  $t$  是低度数 ( $< K$ ) 节点.

### 3.13 形参的处理

函数参数及返回值需要预着色, 函数体的开头会创建临时变量, 使用传送指令将预着色节点传送到临时变量作为形参, 以避免形参长期占用  $a$  开头寄存器.

### 3.14 调用函数

该部分 tigger.y 代码:

```
1 Expression:
2 | ...
3 | PARAM Symbol {AnalyzInstance.currentFunc().CallParam($2,1);}
4 | PARAM INTEGER {AnalyzInstance.currentFunc().CallParam($2,0);}
5 | Symbol '=' CALL FUNCTION {AnalyzInstance.currentFunc().CallFunc($1,$4);}
6 | ...
```

该部分 analyz.cpp 代码:

```
1 void Func::CallParam(int v,int t)
2 {
3     if(t == 1)
4     {
5         new Expression(MoveRR, {(int)(a0)+paramToCallWithCount}, {v}, {});
6     }
7     else
8     {
9         new Expression(MoveRI, {(int)(a0)+paramToCallWithCount}, {}, {v});
10    }
11    paramToCallWithCount++;
12 }
13 void Func::CallFunc(int v,string f)
14 {
15     vector<int> paramvec;
16     paramvec.push_back((int)(a0));
17     for(int i = 1; i<paramToCallWithCount;i++)
18     {
19         paramvec.push_back((int)(a0) + i);
20     }
21     new Expression(Call, {(int)(a0)}, paramvec, {}, f);
22     new Expression(MoveRR, {v}, {(int)(a0)}, {});
23     paramToCallWithCount = 0;
24 }
```

Func 类设置了一个计数器计数当前传入参数的个数. 计数器初始化为 0, 每扫描并处理一个 Param 语句, 将计数器值加一.

对于 'Param x' 语句, 将其处理为 'a0 = x' (对应 Expression 的类型为 MoveRR), 其中 0 可能为 0-7, 是传参计数器的当前值.

由 Eeyore 的构成, 在 Param 语句和对应的 Call 语句之间不会有其他会改变参数寄存器值的语句, 而 Call 语句的 right 属性会被设置成 a0,a1,...,at, 其中 t 为参数的个数-1;

因此活性分析的结果会阻止 Param 语句和 Call 语句之间的其他无关变量被分配为 a0-at 寄存器.

### 3.15 合并预着色点相关传送节点的例子

MiniC 代码

```
1 int v0;
2 v0 = getint();
3 int v1;
4 v1 = func(v0);
5 int v2;
6 v2 = putint(v1+1);
7 return v2;
```

tigger 代码

```
1 call getint
2 call func
3 a0 = a0 + 1
4 call putint
5 return
```

riscv64 代码

```
1 main:
2     add    sp,sp,-16
3     sd     ra,8(sp)
4     call   getint
5     call   func
6     addiw  a0,a0,1
7     call   putint
8     ld     ra,8(sp)
9     addi   sp,sp,16
10    jr     ra
11    .size  main, .-main
```

变量大量合并, 只用了 a0, 没有一条 move 指令.

## 4 Risc-v64/32

本编译器生成 Risc-v64/32 的代码写在 tigger 部分之中, 未单独编写.

Tigger 中设置了全局变量 target 指示需要生成的代码类型.

Func 类中的函数 GenCode(), GenRiscv64(), GenRiscv32() 三个函数分别生成对应函数的 Tigger, Riscv64 和 Riscv32 代码, Analyz 类中的 GenGlobalTigger() 和 GenGlobal() 函数分别生成 Tigger 和 Riscv32/64 的全局变量信息.

tigger 程序运行时接受一个参数, 如提供 -m64 参数,Riscv-64 代码, 如提供 -m32 参数, 则输出 Riscv-32 代码. 如未提供参数或提供了其他参数, 则默认生成 Tigger 代码.

由于 Riscv 不支持寄存器乘以直接数的指令, 因此在 Tigger 中规定了寄存器乘以直接数的指令中的直接数必须为 2 的幂以便转换为左移指令. 在 Riscv 代码输出过程中,Func 类有 OutputArithRIMulI(int reg1,int reg2,int op) 函数来处理这种指令, 将其变为左移指令输出.

此外, 依照 MiniC-Checker 平台的要求, tigger 文件夹中提供了三个利用管道实现的脚本 tiggerC, riscv64C 和 riscv32C, 他们接受 miniC 代码, 输出对应代码.

## 5 hcc

整个编译器已整合为 hcc

从 <https://github.com/vangohao/hcc> 获取源码

在 repo 根目录运行 “make”

运行 “./hcc qsort.c” or “./hcc qsort” 来将 qsort.c 编译为 RISCv64 汇编代码 qsort.s

运行 “./hcc -m32 qsort.c” or “./hcc -m32 qsort” 来将 qsort.c 编译为 RISCv32 汇编代码 qsort.s

hcc 是一个脚本<sup>3</sup>, 实际是调用 eeyore 程序和 tigger 程序实现, 在 eeyore 有错误时会提前终止程序.

## 6 对工具软件的介绍

### 6.1 GNU Bison

GNU bison 是一个自由软件, 用于自动生成语法分析器程序, 实际上可用于所有常见的操作系统。Bison 把 LALR 形式的上下文无关文法描述转换为可做语法分析的 C 或 C++ 程序。在新近版本中, Bison 增加了对 GLR 语法分析算法的支持。GNU bison 基本兼容 Yacc, 并做了一些改进。它一般与 flex 一起使用。<sup>4</sup>

本编译器编写过程中使用的 GNU Bison 版本为 3.0.4

Bison 对于程序员以统一的形式化方法生成语法分析器提供了帮助, 减轻了负担, 同时提供了错误处理机制, 以及内置的语法错误报错功能。

但也存在些许不足 (可能是我没用好), 例如两条前缀相同的文法在前缀中具有相同子代码块时, 也会被视为冲突, 必须修改文法; 此外, 文法中的代码块不是在同一个函数体中执行的, 因此为了在代码块间共享语法树节点信息以外的信息, 不可避免地需要添加全局变量或更大作用域的变量。

---

<sup>3</sup>hcc 脚本内容详见附录

<sup>4</sup>摘自维基百科 [https://zh.wikipedia.org/zh-hans/GNU\\_bison](https://zh.wikipedia.org/zh-hans/GNU_bison)



## 6.2 flex

flex（快速词法分析产生器，英语：fast lexical analyzer generator）是一种词法分析程序。它是 lex 的开放源代码版本，以 BSD 许可证发布。通常与 GNU bison 一同运作，但是它本身不是 GNU 计划的一部分。<sup>5</sup>

本编译器编写过程中使用的 flex 版本为 2.6.4

flex 使用拓展的正则表达式分析词法，非常方便易懂，通过 yylval 变量作为接口，可以向 parser 传递 token 属性。

## 7 测试

### 7.1 测试样例 1 – 快速排序

MiniC 代码:

```
1  int a[20];
2  int qsort(int l,int r)
3  {
4      int i;int j;int x;
5      i=l;j=r;
6      x=a[(l+r) / 2];
7      while(i<j)
8      {
9          while (a[i]<x) i= i+1;
10         while (x<a[j]) j=j-1;
11         if(i<j || i == j)
12         {
13             int y;y=a[i];a[i]=a[j];a[j]=y;
14             i = i +1;j=j-1;
15         }
16     }
17     if (l<j)    qsort(l,j);
18     if (i<r)    qsort(i,r);
19     // return 0;
20 }
21 int main()
22 {
23     int i;
24     i= 0;
25     while(i < 5)
26     {
27         a[i] = getint();
28         i = i + 1;
29     }
30     qsort(0,4);
31     i = 0;
```

<sup>5</sup>摘自维基百科 <https://zh.wikipedia.org/wiki/Flex> 词法分析器

```

32     while(i < 5)
33     {
34         putint(a[i]);
35         putchar(10);
36         i = i + 1;
37     }
38     return 0;
39 }

```

这是一个相当综合的程序, 结合了函数调用和递归, 全局数组读取和写入等操作.

编写这个样例时一开始忘记 qsort 过程的 return 0, 导致运行出错.

于是在程序流分析时检查是否程序流结尾处是否都是 return 语句, 如果有不是的, 给出警告, 并自动添加一条 return 0; 语句.

## 7.2 测试样例 2 – 局部数组

```

1  int main()
2  {
3      int a[20];
4      int i;
5      i=0;
6      while(i<20)
7      {
8          a[i] = i;
9          i = i + 1;
10     }
11     i = 0;
12     while(i<20)
13     {
14         putint(a[i]);
15         i = i + 1;
16     }
17     return 0;
18 }

```

此程序涉及对栈上数组的随机访问读取和写入操作, 事实上, MiniCChecker 服务器中的测试样例没有测试这样的访问. 我在 Accepted 以后才发现没有处理这个问题, 结论是要用 loadaddr 和加法和数组访问语句结合来解决这个问题:

如果 a 是栈上数组, 设 a 存在栈上第 k 个位置, i 所在寄存器为 a1, 则  $a[i] = i$  需要翻译为:

```

1  a2 = a1 * 4
2  loadaddr 0 a0
3  a0 = a0 + a2
4  a0 [0] = a1

```

putint(a[i]) 需要翻译为:

```
1 a2 = a1 * 4
2 loadaddr 0 a0
3 a0 = a0 + a2
4 a0 = a0 [0]
5 store a1 20
6 call putint
7 load 20 a1
```

## 7.3 调试方法

除上面两个之外还使用了一些测试样例, 有一些 (例如传送节点合并, 访存优化的样例) 已经在前面部分给出了, 另外的一些样例代表性不强, 或者代码太长, 就不展示了.

### 7.3.1 getint 和 putint 和 putchar 函数的实现

标准库有 putchar 函数, 但为了统一起见, 我自己使用 System I/O 库函数实现了这三个函数, 以便调试时链接, 具体代码见附录.

### 7.3.2 Riscv-gcc 和 qemu-riscv64 的使用

由于没有自己编写汇编器和链接器, 对生成的 Risc-v64 代码使用 Riscv-gcc 提供的汇编器和链接器, 对于需要使用 getint 等函数的样例程序, 链接上我自己实现的三个函数.

为便于一键调试, 我编写了 hcc-run 调试脚本, 一键进行 eeyore 编译, tigger 编译, riscv64 编译, riscv-gcc 链接, qemu-riscv64 模拟运行全过程, 并打印出程序返回值便于调试, 具体脚本内容参见附件.

## 7.4 发现课程所使用的工具中存在的问题

1. eeyore 模拟器在输入代码各 token 之间没有空格时会发生错误。

# 8 实习总结

## 8.1 收获与体会

我因为时间冲突没报编译技术课程, 除了刘老师在课上着重讲解的 lex/yacc 使用、活性和寄存器分配、基本优化方法以外的其他基础的编译原理相关的知识, 看了好长时间龙书才基本搞懂。编写 eeyore 的时候主要努力搞懂 NFA,DFA, 以及带 L 属性的 SDD 语法分析过程等等, 编写 tigger 的时候主要搞懂了图染色算法, 另外, 阅读了 LLVM2.4 中的线性扫描寄存器分配的代码, 虽然最后写了图染色算法, 但还是收到很大启发。此外, 了解了基本的机器

无关优化技术。我进行了基于活性分析的死代码消除和常数传播优化，另外，使用 pipehole 方法优化减少访存指令。显然，还有无穷无尽的优化的可能。

学习过程中的难点主要是无处不在的 Bug，特别是在写 tigger 的时候，因为 Func 类使用了大量 list 和 Expression 指针，导致多次出现段错误，通过 gdb 仔细查找和尝试，才解决了问题。

此外，在使用 Qemu 模拟器调试的过程中遇到了一些问题，请教了曹德福助教，感谢他的耐心帮助。

另外，大神同学的分享也给了我一些启发，例如活性分析我本来不是用队列实现的，代码冗长，改为队列实现后代码缩短了，效率也提升了。

## 8.2 对课程的建议

感谢编译实习课程对我学习编译技术和代码能力的巨大帮助。对于 MiniCChecker 服务器评测系统有一些改进的建议。

1. 建议将 FTP 上传方式改为网页提交，这样可以做到及时批改，现在使用的系统每次要间隔 5 分钟才能批改，在 debug 的过程中，可能运气不好相邻两次批改就间隔 10 分钟，无形中增加了 debug 消耗的时间，此外，我还出现过 2 次 tar 包上传到一半的时候服务器开始评测的情况，发给我的反馈是解压错误，改为网页上传或许可以解决这个问题。
2. 建议增加更全面的测试样例，当前的测试样例不能检测前面给出的栈数组随机访问功能。
3. 祝老师、助教和同学们新年快乐！

(附录见下一页)

## A hcc 脚本内容

```
1  #!/bin/bash
2
3  Help(){
4      echo "Please input a source file name."
5      echo "For example:"
6      echo "./hcc qsort.c"
7      echo "or"
8      echo "./hcc qsort"
9      echo "or"
10     echo "./hcc -m32 qsort.c"
11 }
12
13 if [[ $# -gt 0 ]] && [[ $1 != "--help" ]]
14 then
15     if [ $1 == "-m32" ]
16     then
17         TC="-m32"
18         if [[ $# -gt 1 ]]
19         then
20             F=${2%.c}
21         else
22             Help
23             exit
24         fi
25     else
26         if [[ $# -gt 1 ]] && [[ $2 == "-m32" ]]
27         then
28             TC="-m32"
29         else
30             TC="-m64"
31         fi
32         F=${1%.c}
33     fi
34     if [ -f "$F.c" ]
35     then
36         ./tigger/eeyore < "$F.c" > "$F.ee"
37         EC=$?
38         if [ ${EC} -eq 0 ]
39         then
40             ./tigger/tigger ${TC} < "$F.ee" > "$F.s"
41             EC=$?
42             if [ ${EC} -eq 0 ]
43             then
44                 rm -f "$F.ee"
45                 if [[ ${TC} == "-m32" ]]
46                 then
47                     echo "RISC-V32 code has sucessfully written to $F.s"
48                 else
49                     echo "RISC-V64 code has sucessfully written to $F.s"
50                 fi

```

```

51         else
52             rm -f "$F.ee" "$F.s"
53             echo "tigger return exit code ${EC}: hcc stoped."
54         fi
55
56     else
57         rm -f "$F.ee"
58         echo "eeyore return exit code ${EC}: hcc stoped."
59     fi
60 else
61     echo "$F.c not exists."
62 fi
63 else
64     Help
65 fi

```

## B getint,putint,putchar 的实现

```

1  #include<unistd.h>
2  int getchar()
3  {
4      char ch=0;
5      read(0,&ch,1);
6      return ch;
7  }
8  int putint(int a)
9  {
10     char buf[40];
11     int i = 0;
12     int negative = 0;
13     if(a<0)
14     {
15         negative =1;
16         a = -a;
17     }
18     do
19     {
20         buf[35-i] = a%10 + '0';
21         a/=10;
22         i++;
23     }
24     while(a);
25     if(negative) {buf[35-i] = '-'; i++;}
26     write(1,buf+(36-i),i);
27     if(negative) a= -a;
28     return a;
29 }
30 int getint()
31 {
32     int ch; int negative = 0;

```

```

33     int result=0;
34     while((ch=getchar()) != -1 &&
35           (ch == ' ' || ch == '\n' || ch=='\t'));
36     if(ch == -1 ) return 0;
37     do
38     {
39         if(ch == '-' && negative ==0 ) negative = 1;
40         else if(ch == '-') return 0;
41         else
42         {
43             result = result * 10 + ch - '0';
44             if(negative == 0) negative = 2;
45         }
46         ch = getchar();
47     }
48     while(ch<='9' && ch >='0');
49     if(negative == 1) return -result;
50     else return result;
51 }

```

## C hcc-run 测试脚本代码

```

1  #!/bin/bash
2  RISCVC=#RISCV 工具包安装路径
3  if [ $# -gt 0 ]
4  then
5      ./eeyore < $1.c > $1.ee
6      echo ".eeyore < $1.c > $1.ee"
7      ./tigger < $1.ee > $1.tg
8      echo ".tigger < $1.ee > $2.tg"
9      ./tigger -m64 < $1.ee > $1.s
10     echo ".tigger -m64 < $2.ee > $2.s"
11     $RISCVC/bin/riscv64-unknown-linux-gnu-gcc $1.s getint_putint.s
12     echo "$RISCVC/bin/riscv64-unknown-linux-gnu-gcc $1.s getint_putint.s"
13     echo "$RISCVC/bin/qemu-riscv64 -L $RISCVC/sysroot a.out"
14     $RISCVC/bin/qemu-riscv64 -L $RISCVC/sysroot a.out
15     echo Return Value:$?
16 else
17     echo "no parameter"
18 fi

```