编译实习报告

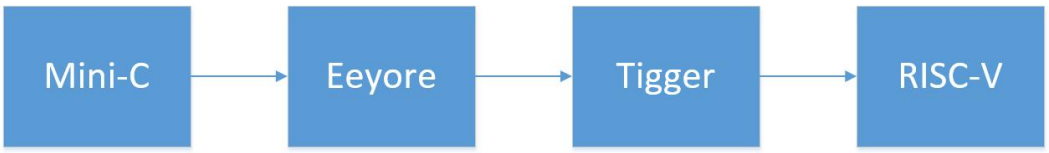
1600017705 苏睿

一、编译器概述

基本功能：

实现一个编译 Mini-C 源程序的编译器，编译结果为 RISC-V 汇编程序，能够运行在支持 RISC-V 的模拟器上。

整个编译器分为 3 个部分，首先将 Mini-C 源程序转化 Eeyore 程序，再将 Eeyore 转化为 Tigger 程序，最终将 Tigger 程序转化为 RISC-V 程序，如下图所示：



（Mini-C, Eeyore, Tigger, RISC-V 具体见说明文档）

二、编译器设计与实现

1、Mini-C -> Eeyore 部分

1）主要功能

完成对 Mini-C 的词法分析、语法分析，实现符号表与表达式翻译及错误处理，最终转化为三地址码 Eeyore。

2）概要

这部分原本应在大概第八周时完成，但由于我在前七周都没有怎么写，所以拖了比较久而且完成地比较仓促，匆匆看了一下 lex 和 yacc 的用法，先用 C 写了一部分后想换成 C++，结果编译报了错，考虑到时间问题就全程用 C 写完了，此阶段各种 malloc、char**简直太难了，之后时间充裕后找了找问题把编译器从 gcc 改成了 g++。

3）符号表

处理符号表采取的方式是，对于 Mini-C 中每个作用域（即每遇到一个 { ），就创建一个该作用域的新符号表，并用指针将原作用域的符号表与新符号表相连，而每当离开一个作用域（即每遇到一个 } ），就销毁当前作用域的符号表，并将当前作用域指针指向其上一个作用域的符号表。

符号表类 Symtab 和作用域类 Scope

```
typedef struct ORIGINID
{
    char* name;
    int index;
    int param; // 3 param, 1 not param
}ORIGINID;

typedef struct Symtab
{
    ORIGINID* ids;
    int size, maxsize;
```

```
}Symtab;


typedef struct Scope
{
    Symtab* st;
    struct Scope* pre;
    struct Scope* nxt;
}Scope;
```

符号表的查找算法 find_symbol，先从当前作用域的符号表开始寻找，若找到则返回结果，若没有就向上层作用域继续寻找。

```
Var find_symbol(Scope* p, char* name)
{
    Var ans; ans.type = 0; ans.index = -1; ans.flag = 0;


    for(int i = 0; i < p->st->size; ++i)
    {
        if(strcmp(p->st->ids[i].name, name) == 0)
        {
            ans.type = p->st->ids[i].param;
            ans.index = p->st->ids[i].index;
            return ans;
        }
    }
    if(p->pre)
        return find_symbol(p->pre, name);


    printf("\nerror! no vaiable named %s\n", name);
    exit(1);
}
```

4）表达式翻译和控制流翻译

表达式翻译使用树结构存储表达式，并从叶节点一步步向上进行。

```
typedef struct TreeNode
{
    int type;
    char* name;
    struct TreeNode* lchild;
    struct TreeNode* rchild;
    int val;
}TreeNode;
```

运算符的优先级，使用 yacc 中的%left 和%right 类型完成，分别表示左结合和右结合的运算符，运算符从前到后的顺序表示其优先级顺序。

```
%right ASSIGN
%left OR
```

```
%left AND
%left EQUN
%left CMP
%left ADDSUB
%left MULDIVMOD
%right NOT
```

而控制流翻译采用的方式是，对于每个 Mini-C 语句，读取并识别后直接输出翻译的结果，此处利用了 yacc 的语法可以添加中间动作实现的。

如此例 if 语句，在 IfHead Statement ELSE Statement 语法的中间添加动作，以保证控制流没有问题。

```
IfHead Statement {
tagstack[top++] = gettag();
printf("goto l%d\n", tagstack[top - 1]);
printf("l%d:\n", tagstack[top - 2]);
} ELSE Statement  {
printf("l%d:\n", tagstack[top - 1]);
top -= 2;
}
```

5）错误处理

错误处理包含了对重名变量/函数的检测、对未定义变量/函数的检测、对函数声明、定义及调用时各错误的检测。

对重名变量的检测，在插入符号表是判断一下其是否已经在符号表中。

```
for(int i = 0; i < p->size; ++i)
    {
        if(strcmp(p->ids[i].name, name) == 0)
        {
            printf("\nerror! variable %s rename\n", name);
            exit(1);
        }
    }
```

对未定义变量/函数的检测，在搜索符号表时未找到该变量。

```
printf("\nerror! no vaiable named %s\n", name);
exit(1);
```

对函数声明、定义及调用时各错误的检测，1.声明要在定义之前；2.函数重复定义/声明；3.函数调用时参数数量错误。

```
if(def == 0 && func_ck[i].index == 1)
 {
     printf("\nerror! can't declare function %s after defined it\n", name);
     exit(1);
 }
 else if(def == func_ck[i].index)
 {
     if(def == 1)
         printf("\nerror! function %s is already defined\n", name);
```

```
            else
                printf("\nerror! function %s is already declared\n", name);
            exit(1);
        }
        else
        {
            if(num != func_ck[i].param)
            {
                printf("\nerror! function %s's paramters is not equal to its declarition\n", name)
;

                exit(1);
            }
            func_ck[i].index = 1;
            return;
        }
```

6）测试遇到的主要错误

运算符优先级问题，在上面已经说明过解决方法。

控制流的问题，由于我一开始就是对于每条语句单独翻译，所以在遇到 if 语句时要在输出翻译后再输出跳转的位置，但若出现 if 语句嵌套 if 语句时会出现问题（抱歉我真的记不得具体什么问题了...只记得当时一度打算全部删掉重写），后来发现 yacc 支持 midaction 即中间动作，就修修补补把他解决了。

Eeyore 不支持负数输入，如 a = -1，则 a 的值会变为二进制的 001111...111，至于为什么是两个 0 我也看不懂，最终会得到一个很大的正数而结果错误，遇到这种情况把他转化为 a = 1, a = -a 就好了（后来听助教说只需要在 - 和 1 之间加个空格就行）。

Eeyore 不支持 a = b + c[d]形式的指令，由于我每次遇到 c[d]形式的值时直接将其和变量等同看待，所以经常会输出类似上面的指令，后来尝试把他拆开成 e = c[d]，a = b + e，又发现当 c[d]是左值时会变成 e = c[d], e = 3 这种，于是先检查了 c[d]是左值还是右值再操作。

主要就是上述错误，使用所给的 15 个测试用例检查出来的，所给的全过了之后就 100 分了，没有构造自己的测试用例。

7）编码难点

这部分都是用 C 写的，没法使用 C++的 string、vector、new 等方便快捷的操作，充斥着 malloc、char**、typedef struct A{}A 等，令人头大。直接将 gcc 改成 g++后，还需要删除 yacc 文件头的#include “lex.yy.c”，并在编译命令中加上 lex.yy.c，就可以使用 g++编译了。

之前很少进行这种多文件的代码编写，所以使用头文件、外部变量的时候不太熟悉，总会编译错误。

主要是没有听老师讲的将所有语句全部构成树之后统一翻译，而是每识别一个语句直接翻译，从而遇到了控制流的种种问题，但是看起来挺直观的也不算坏。

2、Eeyore -> Tigger 部分

1）主要功能

完成对 Eeyore 的词法分析、语法分析，实现活性分析和寄存器分配，最终生

成 Tigger 代码。

2）概要

这部分本来应该在第十二周左右完成的，但是之前第第一部分拖了太久，这部分也就稍微拖了拖。但这次使用的是 C++，而且有了之前使用外部变量的经验，又学会了不再用注释来标识每个类型对应哪个数字，而是用 enum 枚举类型，感觉代码漂亮了许多。

活性分析使用书上给的方法，计算 use 和 def，再通过其计算 in 和 out，从而完成活性分析。寄存器分配采用图分配的方式。

3）代码结构

整个翻译过程以函数为单位，把每个函数中的所有语句串成树结构，再对整个函数进行活性分析和寄存器分配，流程如下：

```
func_varmap(node->lchild);


 init_usedef(node->lchild);


 cal_inout(node->lchild);


 ConflictGraph cfgraph(var_num);
 dfs_addedge(node->lchild, cfgraph);
 cfgraph.allocate_reg();


 translate(node);
```

func_varmap 将变量名统一映射为数字并保存在 map 中，init_usedef 根据变量数量初始化并计算每个块的 use 和 def，cal_inout 使用上述计算出的 use 和 def 计算每个块的 in 和 out，之后创建冲突图对象，用 in 和 out 的值来给冲突图加边 dfs_addedge，再通过寄存器分配算法 allocate_reg 分配寄存器，分配完寄存器后进行翻译的工作 translate。

4）活性分析

活性分析过程将每行语句视为一个单独的语句块，各自保存 use、def 和 in、out，树结点结构如下：

```
class TreeNode
{
public:
    TreeType type;
    string name;
    int value;
    TreeNode *lchild;
    TreeNode *rchild;
    TreeNode *nxt;


    vector<bool> use;
    vector<bool> def;    //resize
    vector<bool> in;
    vector<bool> out;
```

```
    TreeNode();
    TreeNode(TreeNode *lc, TreeNode *rc, TreeNode *n, TreeType t);
    void set_namevalue(string n, int v);

    void dfs_show();
    void delete_node();
    bool set_use(TreeNode *node);
    bool set_def(TreeNode *node);
    bool set_in();                      // in = (out - def) + use
    bool set_out(TreeNode *node);       // out = + nxt_in    node => jump pos
};
```

use 和 def 的计算根据各语句的具体类型来判断，语句中定义的则记为 def，语句中使用到的则记为 use。

in 和 out 的计算根据式

```
// in = (out - def) + use
// out = + nxt_in    node => jump pos
```

由于每条语句为一个语句块，则即使遇见跳转语句，也至多为 1 个，所以 set_out 时单独把跳转的目标节点放进参数中，经过多次迭代后直到不动点则完成 in 和 out 的计算。

```
void cal_inout(TreeNode *node)
{
    while(1)
    {
        bool flag = 0;
        TreeNode *p = node;
        while(p != NULL)
        {
            TreeNode *n = NULL;
            if(p->type == GOTO_TYPE || p->type == IF_TYPE)
                n = label_pos[p->value];
            flag |= p->set_out(n);
            flag |= p->set_in();
            p = p->nxt;
        }

        if(!flag) break;
    }
}
```

5）寄存器分配

使用冲突图进行寄存器分配，冲突图结构如下：

```
class ConflictGraph
```

```
{
public:
    vector<vector<bool> > g;
    vector<int> sum;
    int vars;

    ConflictGraph();
    ConflictGraph(int size);
    void add_edge(int i, int j);
    void allocate_reg();
};
```

根据上述活性分析的结构，通过遍历每个语句块中 in 给冲突图加边，in 中同时存在的两个变量就是相互冲突的。

```
void dfs_addedge(TreeNode *node, ConflictGraph &cfg)
{
    if(!node) return;
    for(int i = 0; i < var_num; ++i)
        if(node->in[i])
        {
            for(int j = i + 1; j < var_num; ++j)
                if(node->in[j])
                    cfg.add_edge(i, j);
        }
    dfs_addedge(node->nxt, cfg);
}
```

接下来就是寄存器分配，方法是任选一个冲突图中度小于寄存器数量的结点放入栈中，删去该节点并重复进行该操作，若是所有结点度均大于等于寄存器数量则任选一个放入栈中，直至全部放到栈中为止。

```
 stack<int> p;
vector<bool> check(vars);

while(p.size() < vars)
{
    bool flag = 0;
    for(int i = 0; i < vars; ++i)
    {
        if(!check[i] && sum[i] < REGNUM)
        {
            p.push(i);
            check[i] = 1;
            flag = 1;
            for(int j = 0; j < vars; ++j)
                if(g[i][j]) sum[j] -= 1;
        }
```

```
        }

        if(!flag)
        {
            for(int i = 0; i < vars; ++i)
                if(!check[i])
                {
                    p.push(i);
                    check[i] = 1;
                    for(int j = 0; j < vars; ++j)
                        if(g[i][j]) sum[j] -= 1;
                    break;
                }
        }
    }
```

每次从栈顶提取结点分配寄存器，检查其相连顶点的寄存器情况，若无寄存器可以分配则分配内存位置。

```
    while(!p.empty())
        {
            int t = p.top();
            p.pop();

            for(int i = 0; i < reg; ++i)
            {
                bool flag = 1;
                for(int j = 0; j < vars; ++j)
                    if(g[t][j] && var_store[j].type == 1 && var_store[j].index == i)
                    {
                        flag = 0;
                        break;
                    }

                if(!flag) continue;
                else
                {
                    var_store[t].type = 1;
                    var_store[t].index = i;
                    break;
                }
            }

            if(var_store[t].type == -1)
            {
                if(reg < REGNUM)
```

```
            {
                var_store[t].type = 1;
                var_store[t].index = reg++;
            }
            else
            {
                var_store[t].type = 2;
                var_store[t].index = mem++;
            }
        }
    }
```

6）翻译过程

考虑到 a0-a7 为参数寄存器，若是调用函数时有 8 个参数就需要全部腾出来，因此直接不分配该 8 个寄存器。又由于一些操作需要用到额外的寄存器，所以把 s9, s10, s11 保留下来使用。因此分配的寄存器共有 16 个，为 t0-t6, s0-s8，将他们统一编号。

```
0-6 : t0-t6
7-18 : s0-s11
19-26 : a0-a7
```

栈内存的分配，对函数栈内存进行统一管理，分为 5 个部分：

1. 未分配寄存器的变量分配的内存区域；
2. s 寄存器保存区域；s 寄存器为被调用者保存寄存器，要在函数最开头保存到栈中，并在返回时恢复。（寄存器分配时考虑到这个问题，所以就把 t 寄存器先分配，s 寄存器后分配，这样若函数不调用其他函数，则 t 寄存器不需要占用空间）
3. 参数保存区域；函数参数最初存在 a0-a7 中，但若其未使用而调用了其他函数，就没地方去了，于是在函数最开头将参数们存在此区域防止找不到。
4. t 寄存器保存区域；若函数不调用其他函数，此区域不需要。
5. 局部数组区域；局部创建的数组保存在此。

```
/*****************************
    var memory area
*****************************
    s reg area
*****************************
    param save area
*****************************
    t reg area
*****************************
    local arr area
*****************************/
```

用 I, II, III, IV, V 记录各区域顶部的位置。

翻译时对于函数参数和全局变量需要额外处理，使用 first_use 记录他们是否使用过，若是第一次使用，则需要将他们从原来的位置转移到分配的寄存器的位置。

剩余过程逐句翻译即可。

7）测试中遇到的主要错误

对于数组的处理，一开始没想明白局部数组应该存到哪，甚至以为直接用寄存器方括号取值就行，一直 segmentation fault，后来才明白局部数组要存到栈中自行管理，于是在原本的四个区域上加了一个局部数组区域。

翻译时对于 first_use 的处理遇到了问题，原因是若这个程序循环中第一次使用了某个函数参数，那么函数参数每次循环都会从原本的位置读取到分配的寄存器的位置，而赋值时未考虑到这种情况，直接对分配的寄存器赋值，于是每次循环中该函数参数最初的值始终不变，会陷入无限循环。修改方式就比较粗糙了，每次给函数参数赋值时，不光修改分配的寄存器，还修改栈中的值。

程序若是对一个变量只赋值不使用的时候遇到了问题，例如：

```
a = 3
b = 4
a = a + 1
```

在这种情况下，由于 b 在之后不会再使用，所以 a 和 b 有可能分配到同一个寄存器中，而翻译时未考虑这个问题，导致 b 的赋值把 a 寄存器中的值覆盖掉。解决方法是赋值前查看一下该语句 out 中该变量是否活跃，若不活跃则不再赋值。

全局变量和函数参数的处理上还发生了很多小问题，多加了些 ifelse 就完成了...

8）编码难点

活性分析和寄存器分配的原理需要重新复习一下编译原理的课本，实现起来用 C++方便多了。在细节处理上问题还是比较多的，不过大体思路比较合理，每完成一部分就输出调试一下看看和想象的是否一致，总体比较顺利。

3、Tigger -> RISC-V 部分

1）主要功能

将 Tigger 代码翻译成 RISC-V 汇编。

2）概要设计

没有什么操作，只需要对着文档一句一句翻译就好了。

文档中没有的 a && b，转化成 !(!a||!b)再翻译。

3）测试中遇到的主要错误

文档问题 goto label 那行应为 j .label，其余所有 label 都加了. ，只有这一行没加，经尝试把所有 . 都去掉也没问题。

main 函数和 function 不太一样，最开始的.size f, .-f 好像有些问题，删去就可以了。

使用 RISC-V 模拟器会提示找不到 getint, getchar, putint, putchar 函数，需要自己写一个程序链接一下。

Tigger 模拟器的问题，我在写 Eeyore -> Tigger 时发现 98 测试样例里好多 putint，每个 putint 我都需要存一遍 t0-t6，再恢复 t0-t6，显得特别丑...于是我就检查了一下 getint, getchar, putint, putchar，遇到这四个函数不再保存 t0-t6，然后就变得比较好看，提交了是 100 分也没问题。**然而！！！**最后一次作业就很有问题，调用 getint, getchar, putint, putchar 的时候会改变 t0-t6 的值，导致结果错误，所以还得保存 t0-t6，这错在 Eeyore -> Tigger 里面，我那天晚上找到 3 点...找了好久好久...所以还是**希望能够把 Tigger 模拟器调用 getint, getchar, putint, putchar 时不改变调**

用者保存寄存器的问题修复一下。

三、实习总结

1、收获与体会

1）主要的收获是什么

学会了 lex, yacc 等工具的使用，熟悉了 C++多文件一起编译的方法，对编译器也有了一个更加具体的认识，写完很有成就感。

2）学习过程的难点是什么

最主要的困难是与拖延症作斗争，由于没人督促又大四了，所以前 7 周根本就没有动工，各作业拖啊拖，终于赶上了；每次 debug 找不到问题会很缺乏耐心，一生气就明天再写，然后又拖啊拖。所以我以后做事情要摆脱拖延，沉下心来冷静一些。

2、对课程的建议

= =多督促一下童鞋们。

整体上还是很有趣的，我把各种模拟器什么的问题都加粗了，希望能够完善一下。

ps：记得老师说过要把最终的结果合成为一个可执行文件，我最初尝试用 C 语言的 system 方法通过命令行调用产生三个阶段的可执行文件 eeyore, tigger, riscv64，生成最终的 riscv64C 程序，但是服务器为防止同学获得测试样例，会把最终的可执行文件单独拷贝到其他目录中运行，这样 riscv64C 报找不到 eeyore, tigger, riscv64 的错，所以除了手动把程序拼在一起我也想不到什么好主意，最终还是交的 3 个单独的 eeyore, tigger, riscv64。