MiniC Compiler报告

Captain32

1 实习要求

本次实习需要实现一个MiniC编译器，共分为三个阶段：

- MiniC代码转换为Eeyore中间代码
- Eeyore中间代码转换为Tigger中间代码
- Tigger中间代码转换为RISC-V汇编代码

2 MiniC代码转换为Eeyore中间代码

2.1 词法分析

2.1.1 token定义

在 Parser.y 中与词法分析有关的token宏定义如下：

```
%union {
    int value;
    char* name;
    struct TreeNode* node;
};

//终结符
%token <name> TYPE MAIN IF ELSE WHILE RETURN ID
%token <value> NUM
%token <value> PRN_L PRN_R ARR_L ARR_R BRC_L BRC_R COMMA EOL

//非终结符
%type <node> Goal BeforeList MainFunc FuncDefn FuncDecl ParamDeclList
%type <node> FuncDeclStmtList VarDefn VarDecl Stmt StmtList Expr ParamList
%type <node> Type Integer Identifier

//运算符号，优先级从上到下递增
%right <name> ASSIGN
%left <name> OR
%left <name> AND
```

```
%left <name> ISEQUAL
%left <name> CMP
%left <name> ADDSUB
%left <name> MULDIV
%right <name> NOT
```

2.1.2 Lex代码

在 Parser.l 中，对于识别得到的不同token产生不同的操作，下面对每种处理举一个例子：

```
[\n]     { lineno++; } //换行，记录行号，用于决定符号活跃范围以及报错时显示
"int"    { yylval.name = strdup(yytext); return TYPE; } //关键字，利用yylval传字符串，返回为TYPE类型
 ......
[a-zA-Z_]([a-zA-Z0-9_])*    { yylval.name = strdup(yytext); return ID; } //标识符，利用yylval传字符串，
                                                                          //返回为ID类型
[0-9]+   { yylval.value = atoi(yytext); return NUM; } //数字，利用yylval传整数，返回为NUM类型
"="      { yylval.name = strdup(yytext); return ASSIGN; } //等号，利用yylval传字符串，返回为ASSIGN类型
 ......
"("      { yylval.value = lineno; return PRN_L; } //左括号，利用yylval传整数行号(决定函数所在行时使用)
 ......
"//"[^\n]* //注释，略过不处理
```

2.2 语法分析

2.2.1 树节点类型、定义及函数

为了区分树节点类型，有宏定义(部分)如下：

```
#define TN_INIT        0 //无类型
#define TN_GOAL        1 //GOAL类型
#define TN_FUNCDEFN    2 //函数定义类型
#define TN_FUNCDECL    3 //函数声明类型
#define TN_VARDEFN     4 //变量定义类型
#define TN_VARDECL     5 //变量声明类型(参数列表中)
#define TN_STMT_BLOCK  6 //语句块类型
#define TN_STMT_IF     7 //if语句类型
 ......
#define TN_EXPR_BIARITH 13 //二元算数运算表达式类型
 ......
#define TN_TYPE        20 //类型关键字类型
#define TN_INTEGER     21 //整数类型
#define TN_IDENTIFIER  22 //标识符类型
```

因为C语言中无C++中的类继承性质，所以语法树节点使用统一的树节点结构，因此TreeNode结构体中包含了需要的所有属性，定义如下：

```
struct TreeNode
{
    int idx; //编号
    int lineno; //所在行号
    int type; //节点类型
```

```
    int val; //值(对于数字常量)
    char* name; //节点名称
    int n_child; //节点子节点数(由类型决定)
    int child_idx; //本节点在父节点的数组中的索引
    struct TreeNode* parent; //父节点
    struct TreeNode* child[MAX_CN]; //子节点数组
    struct TreeNode* sibling_l; //左兄弟
    struct TreeNode* sibling_r; //右兄弟
};
```

关于树节点操作主要有两个函数，new_treenode 与 to_left 函数及作用如下：

```
//new一个树节点，lineno行号，type节点类型，name节点名称(若有)，n_child节点子节点数
struct TreeNode* new_treenode(int lineno,int type,char *name,int n_child)
{
    struct TreeNode* newnode=(struct TreeNode*)malloc(sizeof(struct TreeNode));
    newnode->idx=treenode_num++;
    newnode->lineno=lineno;
    newnode->type=type;
    newnode->val=-1;
    newnode->name=name;
    newnode->n_child=n_child;
    newnode->child_idx=-1;
    newnode->parent=NULL;
    for(int i=0;i<MAX_CN;i++)
        newnode->child[i]=NULL;
    newnode->sibling_l=NULL;
    newnode->sibling_r=NULL;
    return newnode;
}

//从curnode到最左兄弟，设置为parent的第child_idx个子节点
struct TreeNode* to_left(struct TreeNode* curnode,struct TreeNode* parent,int child_idx)
{
    while(curnode!=NULL){
        curnode->parent=parent;
        curnode->child_idx=child_idx;
        if(curnode->sibling_l==NULL) break;
        curnode=curnode->sibling_l;
    }
    return curnode;
}
```

2.2.2 符号表组织、表项定义及函数

符号表被组织为链表结构，所有出现的符号根据出现的先后顺序被串成一串，符号表项定义如下：

```
struct SymNode
{
    int idx; //编号
    int type; //符号类型，0为无类型，1为变量，2为数组，3为函数
```

```
    int born_line; //生效行
    int die_line ; //失效行，-1则代表永不失效
    char* name; //符号名
    char eeyore_type; //eeyore代码中的类型(T、t、f、p)
    int eeyore_idx; //eeyore代码对应的类型中的编号
    struct TreeNode *node; //对应语法树中的树节点
    struct SymNode *next; //前一个符号
    struct SymNode *prev; //后一个符号
};
```

关于符号表操作主要有三个函数，new_symnode、get_symnode与set_death函数及作用如下：

```
//new一个新符号节点，放到链表尾，line行号，type符号类型，name符号名，tnode对应树节点
struct SymNode* new_symnode(int line,int type,char *name,struct TreeNode* tnode)
{
    struct SymNode* newnode=(struct SymNode*)malloc(sizeof(struct SymNode));
    newnode->idx=symbol_num++;
    newnode->type=type;
    newnode->born_line=line;
    newnode->die_line=-1;
    if(name==NULL) name=strdup("");
    newnode->name=strdup(name);
    newnode->eeyore_type='N';
    newnode->eeyore_idx=-1;
    newnode->node=tnode;
    newnode->next=NULL;
    newnode->prev=NULL;
    if(sym_head!=NULL){
        newnode->prev=sym_tail;
        sym_tail->next=newnode;
    }
    sym_tail=newnode;
    return newnode;
}

//获得符号表中在line行生效名为name的符号节点
struct SymNode* get_symnode(int line, char* name)
{
    struct SymNode* resnode=NULL;
    for(struct SymNode* tmpnode=sym_head->next;tmpnode!=NULL;tmpnode=tmpnode->next){
        if(strcmp(tmpnode->name,name)==0
           &&tmpnode->born_line<=line
           &&(tmpnode->die_line>=line||tmpnode->die_line==-1)){
            //需要出生于line前，死于line后，或者永远不死(-1)
            if(resnode==NULL||resnode->born_line<tmpnode->born_line) //取出生最晚的
                resnode=tmpnode;
        }
    }
    return resnode;
}
```

```
//将出生于[born_line,die_line]之间的符号，且die_line仍为-1(尚未初始化)的符号失效行设为die_line
void set_death(int born_line,int die_line)
{
    for(struct SymNode* tmpnode=sym_head->next;tmpnode!=NULL;tmpnode=tmpnode->next){
        if(tmpnode->born_line>=born_line&&tmpnode->born_line<=die_line&&tmpnode->die_line<0)
            tmpnode->die_line=die_line;
    }
}
```

2.2.3 Yacc语法规则

该部分为最冗长的部分，将简要挑选比较有代表性的几个语法规则进行说明。

2.2.3.1 语句序列的处理

对于语句序列块、参数列表都可以使用类似的方式进行处理，将所有语句从前至后串成一串，并将该串从前至后挂在父节点之下，示例如下：

```
Goal    : BeforeList MainFunc { //BeforeList为main之前语句形成的串的最右端
        struct TreeNode* tmp_node=$1;
        $$=new_treenode(lineno,TN_GOAL,NULL,CN_GOAL);
        tmp_node=to_left(tmp_node,$$,0);
        $$->child[0]=tmp_node;
        $$->child[1]=$2;
        $2->parent=$$;
        $2->child_idx=1;
        root=$$;
    }
    ;

BeforeList  : BeforeList VarDefn { //main函数之前的变量、函数声明、实现从前到后串成一串
        if($1!=NULL)
            $1->sibling_r=$2;
        $2->sibling_l=$1;
        $$=$2;
    }
    | BeforeList FuncDefn {
        if($1!=NULL)
            $1->sibling_r=$2;
        $2->sibling_l=$1;
        $$=$2;
    }
    | BeforeList FuncDecl {
        if($1!=NULL)
            $1->sibling_r=$2;
        $2->sibling_l=$1;
        $$=$2;
    }
    | { $$=NULL; }
    ;
```

2.2.3.2 函数定义的处理

对于有参数、无参数函数及main函数都可以使用类似的方式进行处理，示例如下：

```
FuncDefn    : Type Identifier PRN_L ParamDeclList PRN_R BRC_L FuncDeclStmtList BRC_R { //有参数函数
        struct TreeNode* tmp_node;
        $$=new_treenode($1->lineno,TN_FUNCDEFN,NULL,CN_FUNCDEFN);
        $$->child[0]=$1; $1->parent=$$; $1->child_idx=0;
        $$->child[1]=$2; $2->parent=$$; $2->child_idx=1;
        tmp_node=$4;
        tmp_node=to_left(tmp_node,$$,2);
        $$->child[2]=tmp_node;
        tmp_node=$7;
        tmp_node=to_left(tmp_node,$$,3);
        $$->child[3]=tmp_node;
        set_death($3,lineno);
        new_symnode($1->lineno,S_FUNC,$2->name,$$);
        if($7==NULL)
            new_errnode(WARN_NO_RETURN,$$,NULL,NULL);
        else{
            for(tmp_node=$7;tmp_node->sibling_r!=NULL;tmp_node=tmp_node->sibling_r);
            if(tmp_node->type!=TN_STMT_RETURN)
                new_errnode(WARN_NO_RETURN,$$,NULL,NULL);
        }
    }
```

2.2.3.3 变量定义的处理

变量、数组定义处理如下：

```
VarDefn : Type Identifier EOL { //变量定义
            $$=new_treenode(lineno,TN_VARDEFN,NULL,CN_VARDEFN);
            $$->child[0]=$1; $1->parent=$$; $1->child_idx=0;
            $$->child[1]=$2; $2->parent=$$; $2->child_idx=1;
            $$->child[2]=NULL;
            new_symnode($1->lineno,S_INT,$2->name,$$);
        }
        | Type Identifier ARR_L Integer ARR_R EOL { //数组定义
            $$=new_treenode(lineno,TN_VARDEFN,NULL,CN_VARDEFN);
            $$->child[0]=$1; $1->parent=$$; $1->child_idx=0;
            $$->child[1]=$2; $2->parent=$$; $2->child_idx=1;
            $$->child[2]=$4; $4->parent=$$; $4->child_idx=2;
            new_symnode($1->lineno,S_ARR,$2->name,$$);
        }
        ;
```

2.2.3.4 stmt语句的处理

以if-else、赋值语句为例，其他与之类似，如下：

```
Stmt    : IF PRN_L Expr PRN_R Stmt ELSE Stmt { //if-else语句
            $$=new_treenode(lineno,TN_STMT_IF,NULL,CN_STMT_IF);
            $$->child[0]=$3; $3->parent=$$; $3->child_idx=0;
            $$->child[1]=$5; $5->parent=$$; $5->child_idx=1;
            $$->child[2]=$7; $7->parent=$$; $7->child_idx=2;
            set_death($2,$7->lineno-1);
            set_death($7->lineno,lineno);
            if($3->type!=TN_EXPR_BILOGIC
              &&!($3->type==TN_EXPR_UNI&&strcmp($3->name,"!") == 0)
              &&$3->type!=TN_EXPR_IDENTIFIER
              &&$3->type!=TN_EXPR_ARR
              &&$3->type!=TN_EXPR_CALL){
                new_errnode(ERR_WRONG_EXPR,$3,NULL,NULL);
                print_ew();
            }
        }
        | Identifier ASSIGN Expr EOL { //变量赋值语句
            $$=new_treenode(lineno,TN_STMT_VARASSN,NULL,CN_STMT_VARASSN);
            $$->child[0]=$1; $1->parent=$$; $1->child_idx=0;
            $$->child[1]=$3; $3->parent=$$; $3->child_idx=1;
            if($3->type==TN_EXPR_BILOGIC||($3->type==TN_EXPR_UNI&&strcmp($3->name,"!")==0))
                new_errnode(WARN_MIXED_EXPR,$3,NULL,NULL);
            if(find_var(S_INT,$1)==0)
                print_ew();
        }
```

2.2.3.5 表达式的处理

以二元、一元运算、函数调用为例，其他与之类似，如下：

```
Expr    : Expr ADDSUB Expr { //expr + expr或expr - expr
        $$=new_treenode(lineno,TN_EXPR_BIARITH,strdup($2),CN_EXPR_BIARITH);
        $$->child[0]=$1; $1->parent=$$; $1->child_idx=0;
        $$->child[1]=$3; $3->parent=$$; $3->child_idx=1;
        if($1->type==TN_EXPR_BILOGIC||($1->type==TN_EXPR_UNI&&strcmp($1->name,"!")==0)
         ||$3->type==TN_EXPR_BILOGIC||($3->type==TN_EXPR_UNI&&strcmp($3->name,"!")==0))
            new_errnode(WARN_MIXED_EXPR,$$,NULL,NULL);
    }
    | ADDSUB Expr { //取相反数, - expr
        $$=new_treenode(lineno,TN_EXPR_UNI,strdup($1),CN_EXPR_UNI);
        $$->child[0]=$2; $2->parent=$$; $2->child_idx=0;
        if($2->type==TN_EXPR_BILOGIC||($2->type==TN_EXPR_UNI&&strcmp($2->name,"!")==0))
            new_errnode(WARN_MIXED_EXPR,$$,NULL,NULL);
    }
    | Identifier PRN_L ParamList PRN_R { //有参数函数调用
        struct TreeNode *tmp_node;
        $$=new_treenode(lineno,TN_EXPR_CALL,NULL,CN_EXPR_CALL);
        $$->child[0]=$1; $1->parent=$$; $1->child_idx=0;
        tmp_node=$3;
        tmp_node=to_left(tmp_node,$$,1);
        $$->child[1] = tmp_node;
    }
    | PRN_L Expr PRN_R { $$=$2; } //(expr)括号最高优先级
```

2.3 转换为Eeyore中间代码

代码稍长且各类型节点处理不统一，详见 Convert.c ，思路为遍历语法树，根据每种节点的类型不同依据Eeyore的语法规则进行输出转换，如下：

- MiniC中函数调用Expr节点对应Eeyore中的一系列参数Expr节点和一个调用Expr节点
- MiniC中其他Expr节点对应Eeyore中Expr节点，并产生一个新的临时变量
- MiniC中条件跳转Stmt节点对应Eeyore中的一个条件跳转Expr节点和一个跳转Expr节点，并产生两个新的跳转标签
- MiniC中循环Stmt节点对应Eeyore中的一个条件跳转Expr节点和一个跳转Expr节点，以及循环体后的一个跳转Expr节点，并产生三个新的跳转标签
- MiniC中Stmt节点对应Eeyore中赋值Expr节点，使用Stmt中Expr节点产生的临时变量
- MiniC中的VarDefn和FuncDefn节点分别对应Eeyore中的VarDefn和FuncDefn节点
- 其余MiniC中的节点均有Eeyore中节点的直接简单对应

2.4 错误处理

2.4.1 无法识别的输入

在lex代码的最后，对于不属于任何token的输入，进行报错：

```
1 . {
2     char msg[50] = "unrecognized input ";
3     strcat(msg, yytext);
4     yyerror(msg);
```

```
}
```

2.4.2 使用未声明/定义的变量

处理语法规则阶段，每当使用变量，需使用 find_var 函数检查符号表中在对应行是否有该变量，没有则须报错， find_var 函数如下：

```
int find_var(int type,struct TreeNode* node) //查找node对应的符号节点，并检查类型是否与type一致
{
    struct SymNode* sym=get_symnode(node->lineno,node->name);
    if(sym==NULL){
        new_errnode(ERR_UNDEFINED_VAR,node,NULL,NULL);
        return 0;
    }
    else if(sym->type!=type){
        new_errnode(ERR_WRONG_ASSN,node,NULL,NULL);
        return 0;
    }
    else return 1;
}
```

2.4.3 错误的函数调用

在语法规则处理完之后，使用 find_wrong_call 函数遍历语法树对于函数节点调用进行函数调用 find_func 函数进行正确性检查，检查未定义函数错误、参数类型错误、参数数量不一致错误，两个函数实现如下：

```
int find_func(struct TreeNode* tnode) //tnode代表了一个函数调用节点，检查正确性
{
    struct SymNode* sym=get_symnode(tnode->child[0]->lineno,tnode->child[0]->name);
    struct TreeNode *formal,*actual;
    if(sym==NULL){ //函数名不在符号表
        new_errnode(ERR_UNDEFINED_FUNC,tnode->child[0],NULL,NULL);
        return 0;
    }
    else if(sym->type!=S_FUNC){ //函数名在符号表但并不是函数
        new_errnode(ERR_WRONG_CALL,tnode->child[0],NULL,NULL);
        return 0;
    }
    for(formal=sym->node->child[2],actual=tnode->child[1];
        formal!=NULL&&actual!=NULL;
        formal=formal->sibling_r,actual=actual->sibling_r){ //对于参数序列一一比对
        if(actual->type==TN_INTEGER){ //调用给的参数是数字常量
            if(formal->child[2]==NULL)
                continue; //函数定义给的参数不是数组(即int型变量)，允许数字常量调用
            else{
                new_errnode(ERR_WRONG_PARAM,tnode,NULL,NULL);
                return 0;
            }
        }
```

```
24          struct SymNode* tmp=get_symnode(actual->lineno,actual->name);
25          if(tmp==NULL){ //调用参数(变量or数组)不在符号表
26              new_errnode(ERR_WRONG_PARAM,tnode,sym,NULL);
27              return 0;
28          }
29          if((formal->child[2]==NULL&&tmp->type!=S_INT)
30           ||(formal->child[2]!=NULL&&tmp->type!=S_ARR)){ //需要变量对应变量，数组对应数组
31              new_errnode(ERR_WRONG_PARAM,tnode,sym,NULL);
32              return 0;
33          }
34      }
35      if(formal!=NULL||actual!=NULL){ //调用多给参数或少给参数
36          new_errnode(ERR_WRONG_PARAM,tnode,sym,NULL);
37          return 0;
38      }
39      return 1;
40  }
41
42  void find_wrong_call(struct TreeNode* tnode) //递归查找函数调用错误
43  {
44      if(tnode==NULL)
45          return;
46      if(tnode->type==TN_EXPR_CALL&&find_func(tnode)==0)
47          print_ew();
48      find_wrong_call(tnode->sibling_r);
49      for(int i=0;i<tnode->n_child;i++)
50          find_wrong_call(tnode->child[i]);
51  }
```

2.5 Bonus项

因为在78号样例程序修改前便修改代码通过，应助教要求写下方法。在修改代码之前，78号样例一直是RE，后来通过不断上交测试，将问题锁定在if语句与while语句的判断表达式上，MiniC中判断表达式有两种形式：

- x>y||(a+b)!=c这样的逻辑表达式
- x或f(x)或a[x]这样的单个变量或函数

一开始只考虑了第一种形式，报错处理遇到第二种形式便会导致测试中的RE，于是对于判断表达式添加了第二种形式的三种情况的考虑，在Yacc语法规则与转换代码都进行了相应修改，便通过了78号样例。

2.6 测试结果

Check report for lab "eeyore":

You have passed 100 / 100 test cases for lab-eeyore. Congratulations!

This report was automatically generated by Minic Checker at [Wed Oct 30 22:29:32 CST 2019]

3 Eeyore中间代码转换为Tigger中间代码

3.1 数据结构

由于词法分析、语法分析与第2节相似，便不再赘述，经过词法、语法分析，将会构建语法树和全局变量表，并为每个函数建立自己的变量表且以此计算函数栈大小，树节点与符号表项的定义如下：

```
typedef struct treeNode
{
    struct treeNode * Child[3]; //子节点
    struct treeNode * sibling; //兄弟节点
    struct Systable * Table;    //指向该节点对应的符号表的指针
    struct treeNode * Belong;   //Belong记录该节点是在哪个FuncDefn节点之下的,
                                //以便快速定位到该节点所属于的函数
    struct treeNode * Pre;      //活性分析是从后往前扫的,Pre记录当前语句的前一条语句
    struct treeNode * End;      //End记录一个函数节点里最后一条语句,
                                //以便快速从最后一条语句往前做活性分析
    struct treeNode * Next[2];  //活性分析时需要知道每条语句的下一条可能执行语句.对于
                                //控制语句可能不是物理上的下一条语句,需要进行记录
    NodeKind nodekind;  //节点类型
    int expkind=0;  //表达式类型
    int op=0;       //操作符的类型
    char *name;     //记录变量名
    int size=0;     //如果是数组,记录数组的大小
    int val=0;      //如果是变量,记录变量的值
    int paramnum=0;    //对于函数定义节点,需要记录其有多少个参数
    int stacksize=0;   //对于函数定义节点,需要记录该函数的栈空间的大小
    //对于每条语句对应的节点,需要记录该语句中哪些变量是define的,哪些是use,哪些是活跃的
    bitset<bitsetsize> use;
    bitset<bitsetsize> define;
    bitset<bitsetsize> live;
}TreeNode;

struct Systable
{
   char *id;//变量的名字
    int location=0; //记录临时变量在栈中的位置
    int size=0; //如果是数组,记录数组的大小
    int isglobal=0; //是否是全局的变量
    int paramnum=0;  //记录当前变量是该函数的第几个参数,如果不是参数,设为0.
    int reg=0;  //记录哪个寄存器包含了该变量
};
```

3.2 活性分析

为了化简，不进行基本块划分，每条语句均被划分为单独的基本块，基本块按跳转连接起来，构成数据流图，对每个函数构建其数据流图。

3.2.1 构建后继结点

语法树之中每个Stmt节点对应一个流图节点，线性扫描的时候需要知道每个节点的后继节点是谁，使用 BuildNextLink 函数来完成，如下：

```
void BuildNextLink()//获得每个节点的后继节点,供后续线性扫描使用
{
    TreeNode* t=root;
    while(t!=NULL){
        if(t->nodekind!=FuncK){ //现在在主线，只有函数定义与全局变量定义，将全局变量定义忽略
            t=t->sibling;
            continue;
        }
        TreeNode* tmp1=t->Child[0];
        while(tmp1!=NULL){
            if(tmp1->nodekind==ExpK && tmp1->expkind==6){ //if ... goto ...
                TreeNode*tmp2=t->Child[0];
                while(tmp2!=NULL){ //寻找目标label
                    if(tmp2->nodekind==ExpK && tmp2->expkind == 8
                       && strcmp(tmp1->name,tmp2->name)==0){
                        tmp1->Next[0]=tmp1->sibling;
                        tmp1->Next[1]=tmp2;
                        break;
                    }
                    tmp2=tmp2->sibling;
                }
            }
            else  if(tmp1->nodekind==ExpK && tmp1->expkind==7){ //goto ...
                TreeNode*tmp2=t->Child[0];
                while(tmp2!=NULL){
                    if(tmp2->nodekind==ExpK && tmp2->expkind == 8
                       && strcmp(tmp1->name,tmp2->name)==0){
                        tmp1->Next[0]=tmp2;
                        break;
                    }
                    tmp2=tmp2->sibling;
                }
            }
            else{ //其他语句后继节点就是下一个节点
                tmp1->Next[0]=tmp1->sibling;
            }
            tmp1=tmp1->sibling;
        }
        t=t->sibling;
    }
}
```

3.2.2 计算live集合

活性分析的数据流算法之中，需要对每个基本块和所有基本块中的语句，根据def和use集合，计算live集合。整个算法由后向前迭代，直至收敛到达不动点(即流图中任何节点和任何语句live集合都不变)，计算公式如下：

$$live(stmt) = (\cup_{s \in next(stmt)} live(s) / def(stmt)) \cup use(stmt) \tag{1}$$

使用 LiveAnalysis 函数进行def、use集合初始化以及计算live集合，实现(部分)如下：

```
1  void LiveAnalysis()//活性分析
2  {
3      TreeNode* t=root;//先遍历语法树,将每条语句的use和def初始化
4      while(t!=NULL){
5          if(t->nodekind!=FuncK){ //跳过非函数节点
6              t=t->sibling;
7              continue;
8          }
9          TreeNode* tmp=t->Child[0];
10         t=t->sibling;
11         while(tmp!=NULL)
12         {
13             if(tmp->nodekind==VarDefnK){ //跳过变量定义语句
14                 tmp=tmp->sibling;
15                 continue;
16             }
17             if(tmp->expkind==1) //l = r op2 r
18             {
19                 tmp->define[Getnum(tmp,tmp->Child[0]->name)]=1;
20                 if(tmp->Child[1]->op==0) //id
21                     tmp->use[Getnum(tmp,tmp->Child[1]->name)]=1;
22                 if(tmp->Child[2]->op==0)
23                     tmp->use[Getnum(tmp,tmp->Child[2]->name)]=1;
24             }
25              ......
26             tmp=tmp->sibling;
27         }
28     }
29
30     t=root;
31     while(t!=NULL){ //遍历这棵语法树,以函数为单位进行活性分析.
32         if(t->nodekind!=FuncK){
33             t=t->sibling;
34             continue;
35         }
36         bool flag =true; //记录是否达到不动点
37         while(flag){
38             TreeNode*tmp=t->End;
39             bool Change=false;
40             while(tmp!=NULL){ //不断循环重复活性分析直到达到不动点
41                 if(tmp->Next[0]==NULL) //无后继节点
42                     tmp->live=tmp->use;
43                 else{
```

```
                    bitset<bitsetsize> tmpbs;
                    if(tmp->nodekind==ExpK && tmp->expkind==6) //if ... goto ...
                        tmpbs = tmp->Next[0]->live|tmp->Next[1]->live;
                    else
                        tmpbs = tmp->Next[0]->live;
                    tmpbs = tmpbs & (~tmp->define);
                    tmpbs = tmpbs | tmp->use;
                    if(tmpbs!=tmp->live)
                        Change=true;
                    tmp->live=tmpbs;
                }
                tmp=tmp->Pre;
            }
            if(!Change)
                flag=false;
        }
        t=t->sibling;
    }

}
```

3.3 寄存器分配与释放

3.3.1 寄存器分配函数

使用 allocate 函数为变量分配寄存器时，若变量为函数参数，则根据其所在第几个参数分配a0-a7寄存器；否则分配t0-t6、s0-s7寄存器，s8-s10寄存器作为溢出保留(被分配了这3个寄存器的变量须在使用后立即释放，保证3个寄存器时时可分配)，s11不能被分配，用作保留寄存器，比如生成代码时用作保留中间结果，函数实现如下：

```
//分配寄存器
int allocate(TreeNode*t,int index)
{
    if(t->Belong->Table[index].paramnum>0) //如果是参数，根据所在第几个参数确定a0-a7
    {
        int tmp=19+t->Belong->Table[index].paramnum;
        t->Belong->Table[index].reg=tmp;
        Regused[tmp]=index;
        return tmp;
    }
    for(int i=13;i<=19;++i) //t0-t6，被调用者保存
    {
        if(!Regused[i])
        {
            Regused[i]=index;
            t->Belong->Table[index].reg=i;
            return i;
        }
    }
    for(int i=1;i<=11;++i) //s0-s7，调用者保存,s8、s9、s10用作溢出保留，s11保存结果
    {
        if(!Regused[i])
```

```
        {
            Regused[i]=index;
            t->Belong->Table[index].reg=i;
            return i;
        }
    }
    return 0;
}
```

3.3.2 寄存器释放函数

使用 DeleteUseless 函数时，若释放的变量占用的寄存器是s8-s11寄存器(保留寄存器)，则立马释放，否则使用线性扫描算法，检查之后是否该变量是否失活，若以后用不到该变量，则释放寄存器，否则不释放，函数实现如下：

```
//如果某个变量之后都不活跃,那么释放它所占用的寄存器
void DeleteUseless(TreeNode*t,int index)
{
    if(t->sibling==NULL)return;
    if(t->Belong->Table[index].reg>=9&&t->Belong->Table[index].reg<=11){ //保留寄存器
        Regused[t->Belong->Table[index].reg]=0;
        t->Belong->Table[index].reg=0;
        return;
    }
    int  flag=0;
    while(t->sibling!=NULL) //线性扫描之后变量活性
    {
        if(t->sibling->live[index]!=0)
        {
            flag=1;
            break;
        }
        t=t->sibling;
    }
    if(flag==0)
    {
        Regused[t->Belong->Table[index].reg]=0;
        t->Belong->Table[index].reg=0;
    }
}
```

3.4 转换为Tigger中间代码

3.4.1 整体框架

Translate 函数遍历语法树，对于全局变量定义节点生成相应Tigger代码，对于函数定义节点，生成函数定义头，之后开始逐句转换，每句转换前需尽可能多的为live集中的变量分配寄存器，随后根据不同的语句类型跳转到对应的函数进行处理，代码(部分)如下：

```
//针对不同的语句,进行翻译
void Translate()
{
    TreeNode*t=root;
    while(t!=NULL){
        if(t->nodekind!=FuncK){  //全局变量定义
            GenGlobalVarDefn(t);
            t=t->sibling;
            continue;
        }
        TreeNode *tmp=t->Child[0];
        printf("%s [%d] [%d]\n",t->name,t->paramnum,t->stacksize); //函数定义头
        memset(Regused,0,sizeof(Regused)); //将寄存器映射表清空
        for(int i=1;i<=t->size;++i){ //先为参数变量分配寄存器
            if(t->Table[i].paramnum>0)
                allocate(tmp,i);
        }
        while(tmp!=NULL){ //进入函数，逐句翻译
            if(tmp->nodekind==VarDefnK){
                tmp=tmp->sibling;
                continue;
            }
            for(int i=1;i<tmp->Belong->size;++i){ //为尽可能多的在live集中的变量分配寄存器
                if(tmp->live[i]==1&&tmp->Belong->Table[i].reg==0){
                    int tmpreg=allocate(tmp,i);
                    if(tmpreg>=9&&tmpreg<=11) break;
                    if(tmp->Belong->Table[i].isglobal==1){
                        if(tmp->Belong->Table[i].size>0)
                            printf("loadaddr v%d %s\n",i-1,Reg[tmpreg]);
                        else
                            printf("load v%d %s\n",i-1,Reg[tmpreg]);
                    }
                    else{
                        if(tmp->Belong->Table[i].size>0)
                            printf("loadaddr %d %s\n",tmp->Belong->Table[i].location,Reg[tmpreg]);
                        else
                            printf("load %d %s\n",tmp->Belong->Table[i].location,Reg[tmpreg]);
                    }
                }
            }
            DeleteUseless(tmp,Regused[9]);
            DeleteUseless(tmp,Regused[10]);
            DeleteUseless(tmp,Regused[11]);
            if(tmp->expkind==1) //根据语句类型跳转至对应函数进行处理
                GenVarEqRvOpRv(tmp);
             ......
            tmp=tmp->sibling;
        }
        printf("end %s\n",t->name); //函数尾
        t=t->sibling;
    }
}
```

3.4.2 具体语句翻译示例

3.4.2.1 二元运算语句

GenVarEqRvOpRv 实现(部分)如下：

```
void GenVarEqRvOpRv(TreeNode*t)
{
    int id0=Getnum(t,t->Child[0]->name);
    int reg0=t->Belong->Table[id0].reg;
    if(t->Belong->Table[id0].reg==0) //为目标变量分配寄存器(若无)
        reg0=allocate(t,id0);
    if(t->Child[1]->op==0 && t->Child[2]->op==0){ //两操作数都为变量
        int id1=Getnum(t,t->Child[1]->name);
        int reg1=t->Belong->Table[id1].reg;
        int id2=Getnum(t,t->Child[2]->name);
        int reg2=t->Belong->Table[id2].reg;
        if(t->Belong->Table[id1].reg==0){ //为源变量1分配寄存器(若无)
            reg1=allocate(t,id1);
            if(t->Belong->Table[id1].isglobal==1)
                printf("load v%d %s\n",id1-1,Reg[reg1]);
            else
                printf("load %d %s\n",t->Belong->Table[id1].location,Reg[reg1]);
        }
        if(t->Belong->Table[id2].reg==0){ //为源变量2分配寄存器(若无)
            reg2=allocate(t,id2);
            if(t->Belong->Table[id2].isglobal==1)
                printf("load v%d %s\n",id2-1,Reg[reg2]);
            else
                printf("load %d %s\n",t->Belong->Table[id2].location,Reg[reg2]);
        }
        printf("%s = %s %s %s\n",Reg[reg0],Reg[reg1],OP[t->op],Reg[reg2]); //输出运算语句

        DeleteUseless(t,id0);
        DeleteUseless(t,id1);
        DeleteUseless(t,id2);
    }
     ......
    if(t->Belong->Table[id0].isglobal==1){ //全局变量及时写回内存
        printf("loadaddr v%d s11\n",id0-1);
        printf("s11[0] = %s\n", Reg[reg0]);
    }
    else if(reg0>=9&&reg0<=11){ //使用了保留寄存器，需及时写回栈中
        printf("loadaddr %d s11\n",t->Belong->Table[id0].location);
        printf("s11[0] = %s\n", Reg[reg0]);
    }
}
```

3.4.2.2 准备参数语句

GenParamRv 中有很多易错点，需要将本函数的参数先存入栈中、传参是变量还是数组要区分、本函数的参数变量作为参数需从栈中取出(之前已存入栈中)，这三个点十分关键，并且发现最后一点使我通过了最后一个样例，实现如下：

```
void GenParamRv(TreeNode*t)
{
    if(paramcount==0){ //paramcount为全局变量，表示当前已准备好几个参数
        for(int i=20;i<=27;++i) //将本函数的a0-a7寄存器内容存入栈中
            if(Regused[i])
                printf("store %s %d\n",Reg[i],t->Belong->Table[Regused[i]].location);
    }
    if(t->Child[0]->op==0){ //参数为变量
        int id0=Getnum(t,t->Child[0]->name);
        int reg0=t->Belong->Table[id0].reg;
        if(t->Belong->Table[id0].reg==0){
            reg0=allocate(t,id0); //分配寄存器(若无)
            if(t->Belong->Table[id0].isglobal==1){ //全局变量
                if(t->Belong->Table[id0].size==0)
                    printf("load v%d %s\n",id0-1,Reg[reg0]);
                else //数组!
                    printf("loadaddr v%d %s\n",id0-1,Reg[reg0]);
            }
            else{ //局部变量
                if(t->Belong->Table[id0].size==0)
                    printf("load %d %s\n",t->Belong->Table[id0].location,Reg[reg0]);
                else //数组!
                    printf("loadaddr %d %s\n",t->Belong->Table[id0].location,Reg[reg0]);
            }
        }
        if(reg0>=20&&reg0<=27) //是本函数的参数变量(在刚开始准备参数时已经被存入栈中)，需从栈中获取
            printf("load %d %s\n",t->Belong->Table[Regused[reg0]].location,Reg[20+paramcount]);
        else
            printf("%s = %s\n",Reg[20+paramcount],Reg[reg0]);
        DeleteUseless(t,id0);
    }
    else{ //参数为整数
        printf("%s = %d\n",Reg[20+paramcount],t->Child[0]->val);
    }
    paramcount++; //更新已准备的参数数量
}
```

3.4.2.3 调用语句

GenVarEqCall中也有一些关键点，比如在发出call指令前需要将本函数的局部变量先存入栈中、在获得结果后从栈中恢复寄存器状态时不能将新得到的结果寄存器中的内容覆盖，实现(部分)如下：

```
void GenVarEqCall(TreeNode*t)
{
    paramcount=0; //清零，方便下次准备参数时使用
    for(int i=1;i<=t->Belong->size;++i){ //将本函数的局部变量存入栈中
        if(t->Belong->Table[i].reg==0)continue;
        if(t->Belong->Table[i].paramnum>0)continue; //参数变量准备参数时已经存过
        if(t->Belong->Table[i].isglobal==1) continue;
        if(t->Belong->Table[i].size>0) continue; //切记不存数组，否则将数组地址存入了数组第一个元素
```

```
9           printf("store %s %d\n",Reg[t->Belong->Table[i].reg],t->Belong->Table[i].location);
10      }
11      int id0=Getnum(t,t->Child[0]->name);
12      int reg0=t->Belong->Table[id0].reg;
13      if(t->Belong->Table[id0].reg==0) //分配寄存器(若无)
14          reg0=allocate(t,id0);
15      printf("call %s\n",t->name);
16      printf("%s = a0\n",Reg[reg0]);
17      DeleteUseless(t,id0);
18       ..... //目标变量为全局变量或使用保留寄存器的变量时，需存回
19      for(int i=1;i<=t->Belong->size;++i){ //从栈中恢复寄存器
20          if(t->Belong->Table[i].reg==0)continue;
21          if(t->Belong->Table[i].reg==reg0) continue; //注意！不能将刚获得的结果覆盖，这里应跳过
22          if(t->Belong->Table[i].isglobal==1){
23              if(t->Belong->Table[i].size>0)
24                  printf("loadaddr v%d %s\n",i-1,Reg[t->Belong->Table[i].reg]);
25              else
26                  printf("load v%d %s\n",i-1,Reg[t->Belong->Table[i].reg]);
27          }
28          else{
29              if(t->Belong->Table[i].size>0)
30                  printf("loadaddr %d %s\n",t->Belong->Table[i].location,Reg[t->Belong->Table[i].reg]);
31              else
32                  printf("load %d %s\n",t->Belong->Table[i].location,Reg[t->Belong->Table[i].reg]);
33          }
34      }
35  }
```

3.5 测试结果

Check report for lab "tigger":

You have passed 100 / 100 test cases for lab-tigger. Congratulations!

This report was automatically generated by Minic Checker at [Thu Nov 21 23:59:40 CST 2019]

4 Tigger中间代码转换为RISC-V汇编代码

4.1 翻译规则

由Tigger代码转换为RISC-V汇编代码已经十分简单，只需根据语句类型逐句翻译即可，代码已无需列出，以下给出翻译规则：

Tigger Statement	RISC-V Instructions
VAR = VAL	.global VAR .section .sdata .align 2 .type VAR, @object .size VAR, 4

	VAR: .word VAL .comm ARR, SIZE*4, 4

FUNC[INT1][INT2]
(SIZE=INT2/4+1)*16
	.text
	.align 2
	.type FUNC, @function
	FUNC:
	addi sp, sp, SIZE
	sw ra, SIZE(sp)

end FUNC	.size FUNC, .-FUNC
REG = REG1 + REG2	add REG, REG1, REG2
REG = REG1 - REG2	sub REG, REG1, REG2
REG = REG1 * REG2	mul REG, REG1, REG2
REG = REG1 / REG2	div REG, REG1, REG2
REG = REG1 % REG2	rem REG, REG1, REG2
REG = REG1 && REG2	and REG, REG1, REG2 snez REG, REG
REG = REG1 || REG2	or REG, REG1, REG2 snez REG, REG
REG = REG1 == REG2	xor REG, REG1, REG2 seqz REG, REG
REG = REG1 != REG2	xor REG, REG1, REG2 snez REG, REG
REG = REG1 < REG2	slt REG, REG1, REG2
REG = REG1 > REG2	slt REG, REG2, REG1
REG = REG1 + IMM	addi REG, REG1, IMM
REG = REG1 < IMM	slti REG, REG1, IMM
REG = - REG1	sub REG, x0, REG1
REG = ! REG1	seqz REG, REG1
REG = REG1	mv REG, REG1
REG = IMM	li REG, IMM
REG[IMM] = REG1	sw REG1, IMM(REG)
REG = REG1[IMM]	lw REG, IMM(REG1)
if REG1 < REG2 goto LABEL	blt REG1, REG2, .LABEL
if REG1 > REG2 goto LABEL	bgt REG1, REG2, .LABEL
if REG1 <= REG2 goto LABEL	ble REG1, REG2, .LABEL
if REG1 >= REG2 goto LABEL	ble REG2, REG1, .LABEL
if REG1 == REG2 goto LABEL	beq REG1, REG2, .LABEL
if REG1 != REG2 goto LABEL	bne REG1, REG2, .LABEL
goto LABEL	j .LABEL
LABEL:	.LABEL:
call FUNC	call FUNC

store REG IMM	sw REG, IMM(sp)
load IMM REG	lw REG, IMM(sp)
load VAR REG	lui REG, %hi(VAR) lw REG, %lo(VAR)(REG)
loadaddr IMM REG	add REG, sp, IMM
loadaddr VAR REG	lui REG, %hi(VAR) addi REG, %lo(VAR)(REG)

return	lw ra, SIZE-4(sp)
	addi sp, sp, SIZE
	jr ra

4.2 Bonus项

应助教要求写下该项，我在上交代码测试时，发现有两个样例编译时RE，但是在电脑上可以编译成功，便使用了工具链汇编器运行我的输出文件，发现RE，经多次尝试发现原因在于，RISC-V立即数指令只有12位，所以进行与立即数相关的操作时，大于2048便会出现错误，于是，我对于代码中跟立即数相关的地方都进行了特判处理，在第二阶段中的保留寄存器s11也发挥到了作用，最终通过样例，具体处理示例如下：

```
if(stk<2048){
    fprintf(yyout,"\taddi sp, sp, %d\n",-stk);
    fprintf(yyout,"\tsw ra, %d(sp)\n",stk-4);
}
else{ //大于2048，利用s11进行中转，转化为寄存器运算
    fprintf(yyout,"\tli s11, %d\n",stk);
    fprintf(yyout,"\tsub sp, sp, s11\n");
    fprintf(yyout,"\tadd s11, sp, s11\n");
    fprintf(yyout,"\tsw ra, %d(s11)\n",-4);
}
```

4.3 测试结果

Check report for lab "riscv64":

You have passed 100 / 100 test cases for lab-riscv. Congratulations!

This report was automatically generated by Minic Checker at [Thu Dec 19 00:34:37 CST 2019]

5 总结

“纸上得来终觉浅，绝知此事要躬行”，本次实习中我从前端到后端实现了一个简易的编译器，更加深入具体地了解到了编译器的各个组件，以及开发过程所使用的工具，并且看到自己的编译器可以成功运行，使我获得了极大的满足感。在实现编译器的过程中，我认为万事开头难，一开始不知从何下手，想的太多反而束缚了手脚，直到开始慢慢尝试使用并熟悉工具软件，才逐渐走入正轨，所以遇到问题不要害怕，先试一试再说。