编译PJ实验报告

马成 20307130112

编译项目过程简介

词法分析

1. 使用LEX工具可以快速的进行词法分析，将代码中字符串转化为一个一个token序列给后续的句法分析阶段
2. LEX本质是一个自动机进行有优先级的最长匹配
3. 在LEX中可以定义若干状态，LEX可以在不同状态中选择不同的匹配规则，相对于直接使用C语言的分支语句会更加方便。我们利用这个机制处理正式代码和注释的区别。代码中支持了两种注释。

```
%start COMMENT1  COMMENT2
%%
<COMMENT1>"\n" {++line;pos=0;BEGIN INITIAL;}
<COMMENT1>. {pos+=yyleng;}

<COMMENT2>"*/" {pos+=yyleng;BEGIN INITIAL;}
<COMMENT2>"\n" {++line;pos=0;}
<COMMENT2>. {pos+=yyleng;}
...
```

4. 对于正式代码部分我们直接使用`<INITIAL>`状态即可，根据手册识别关键字、数字、符号标识符即可
5. 忽略一些空格等没有实际意义的东西

```
<INITIAL>"\n" {++line;pos=0;}
<INITIAL>" "|"\r"|"\t" {pos+=yyleng;}
```

6. 为了在后续类型检查的时候更加方便定位，在LEX部分就维护了代码的对应位置使用了`len`和`pos`来维护

语法分析并生成语法树

1. 语法分析的本质是一个上下文有关自动机，编写这个自动机有很多公用的代码。为了更方便的完成语法分析，这里使用了`yacc`作为工具辅助。
2. 定义yacc中的每一种token对应于C语言的实际类型，其中关键字原本可以定义为一个简单的int类型或者其他，但是我们希望在语法树中维护位置信息，所以将关键字都设置成了`A_Pos`类型，其他的token就按照他们自己的类型对应设置即可。

```
%union
{
    A_pos token; // 例：符号
    A_pos key; // 关键字
    A_type type;
    A_prog prog;
```

```
    A_mainMethod mainMethod;
    A_classDecl classDecl;
    A_classDeclList classDeclList;
    A_methodDecl methodDecl;
    A_methodDeclList methodDeclList;
    A_formal formal;
    A_formalList formalList;
    A_varDecl varDecl;
    A_varDeclList varDeclList;
    A_stmList stmList;
    A_stm stm;
    A_exp exp;
    A_expList expList;
}
```

3. 定义语法分析的终结符和非终结符

```
// 终结符
%token <token> OP_PLUS OP_MINUS OP_MULTIPLY OP_DIVTION OP_LESS OP_LE
OP_GREAT
    OP_GE OP_EQ OP_NEQ OP_OR OP_AND '(' ')' '=' ',' ';' '{' '}' '.' '!' '['
']'
%token <key> PUTINT PUTCH PUTARRAY GETINT GETCH GETARRAY MAIN INT PUBLIC
CLASS
    IF ELSE WHILE CONTINUE BREAK RETURN STARTTIME STOPTIME TTRUE FFALSE
LENGTH
    THIS NEW EXTENDS
%token <exp> ID NUM

// 非终结符的类
%type<type> TYPE
%type<prog> PROG
%type<mainMethod> MAINMETHOD
%type<classDecl> CLASSDECL
%type<classDeclList> CLASSDECLLIST
%type<methodDecl> METHODDECL
%type<methodDeclList> METHODDECLLIST
%type<formal> FORMALREST
%type<formalList> FORMALLIST FORMALRESTLIST
%type<varDecl> VARDECL
%type<varDeclList> VARDECLLIST
%type<stmList> STMLIST
%type<stm> STM
%type<exp> EXP NUMBER NUMBERREST EXPREST
%type<expList> EXPLIST EXPRESTLIST NUMBERLIST NUMBERRESTLIST
```

4. 语法上有一些歧义因此需要定义一些转化的优先级使得可以稳定的解释代码。本来是自己尝试按照自己的经验来设计运算的优先级，但是实际效果较差，`yacc`一直报错说还是有各种冲突。最后想到可以使用C语言的运算符结合性和优先级，果然就没有冲突了。

```
%right '='
%left OP_OR
%left OP_AND
%left OP_EQ OP_NEQ
%left OP_LE OP_LESS OP_GREAT OP_GE
%left OP_PLUS OP_MINUS
%left OP_MULTIPLY OP_DIVTION
%right '!' UMINUS
%left '[' ']' '(' ')'
%left '.'
%right ELSE
```

5. 编写匹配规则并按照规则生成语法树即可，最后结果就会存储在root中。在yacc的语法中就是当他检查到了一个`WHILE '(' EXP ')' STM`这么一串token之后yacc会将这一串token转化为一个`STM`并且会利用里面的内容进行转化`$$=A_WhileStm($1,$3,$5)`的含义就是将返回值设置为一个`A_WhileStm($1,$3,$5)`其中所有参数`$3`表示的是`WHILE '(' EXP ')' STM`中的第三个token也就是`EXP`的相关信息，`$1`表示的是`WHILE`这里使用了这个token的位置信息。

```
STM : WHILE '(' EXP ')' STM
    {
        $$=A_WhileStm($1,$3,$5);
    }
```

类型检查

1. 语法树中我们还保留了显示的数据类型，在语法中检查数据类型是较为方便。我们重点应该考虑几件事情
    - 声明变量的时候变量的类型是否合法
    - 声明变量的时候变量名有没有重复
    - 类名不能重复
    - 继承关系不能成环
    - 类内的成员变量不可以重名（包括从父类继承得到的成员变量），成员变量类型合法
    - 类内的成员函数不可以重名，如果有继承的成员函数要求函数签名一致
    - 在赋值的时候要求左值必须是可以赋值的，有空间位置的元素
    - 赋值的时候两边的类型要匹配，即如果是一般类型应该一样，如果是class那么右边必须是左边的父类
    - 使用`[]`的时候必须前面是一个数组
    - 取成员变量和函数的时候要考虑元素是不是一个类，如果是这个类里面有没有这个成员变量或成员函数
    - 返回值要和函数声明的返回值一致
2. 需要使用到的table和struct

```
S_table t;          // save local VAR and its type in it , key:name
value:type
S_table classtable; // save class type in it , key:className
Value:S_table{Var/method's name,TyAndInit}
                    // when save fuction use a Ty_record.
                    // in Ty_fildlist ,the first Ty in the list is the
returnType
S_table extends;    // save className and its treenode pointer key:calssName
value:tree
S_table classPos;   // save className and their struct
S_table classElementsOffset; // save every Var or function's offset
S_table tempFunctionName; // the temp_name of every fuction
int classElementsCnt=0;
```

```
typedef struct node{
    bool location;
    Ty_ty value;
}node; // use to return exp's location and type
typedef struct tree_* tree; // use when solve extend
typedef struct tree_{
    S_symbol name;
    S_symbol faname;
    bool vs;
    bool finish;
}tree_;
typedef struct TyAndInit_{
    Ty_ty ty;
    A_expList expList;
}TyAndInit_; // use in IR section, in typecheck we only use its type
typedef struct TyAndInit_* TyAndInit;
```

3. 类型检查的时候先要浏览一遍所有的类和成员函数并维护每一个成员的父类关系，将成员变量和成员函数存储以便后续的使用。这一步就是给编译器一个全局的视角，让他知道整个函数中定义了什么类，即什么样的类型是合法的，了解一下继承关系。在这个位置可以检查的错误就只有类名重复定义了。

```
void fillTable(A_classDeclList list){
    if (!list) return ;
    A_classDecl x=list->head;
    S_enter(classPos,S_Symbol(x->id),x);
    if (S_look(classtable,S_Symbol(x->id))){
        printError(x->pos,"this class name has been used");
    }
    S_table table=S_empty();
    typeCheckVarDeclList(x->vdl,table,1);
    filltableMethodList(x->mdl,table,x->id);
    S_enter(classtable,S_Symbol(x->id),table);
    if (x->parentID){
        tree now=checked_malloc(sizeof(*now));
        now->faname=S_Symbol(x->parentID);
        now->name=S_Symbol(x->id);
        now->vs=0;
        now->finish=0;
```

```
        S_enter(extends,S_Symbol(x->id),now);
    }
    fillTable(list->tail);
}
```

4. 处理继承关系，将父类的成员变量复制给子类，将父类的成员函数也传递给子类。同时检查一下是否有继承关系环。成员变量的类型是不是合法有没有重复。成员函数的声明是不是合法，如果有重写成员函数的函数标签是否一致。

```
void solveExtendsList(A_classDeclList list){
    if (!list) return ;
    solveExtends(list->head);
    solveExtendsList(list->tail);
}
void solveExtends(A_classDecl x){
    if (x->parentID==NULL){
        checkExistVarList(x->vdl);
        checkExistMethodList(x->mdl);
        return ;
    }
    tree now=S_look(extends,S_Symbol(x->id));
    assert(now);
    if (now->finish) return ;
    if (now->vs){
        printError(x->pos,"the extends relationship has circle");
    }
    now->vs=1;
    checkExistVarList(x->vdl); // 检查成员变量
    checkExistMethodList(x->mdl); // 检查成员函数
    A_classDecl fa=S_look(classPos,now->faname);
    if (fa==NULL){
        printError(x->pos,"the extends class does not defined");
    }
    // assert(fa);
    solveExtends(fa); // 递归调用
    extendsTable(x,S_look(classtable,now->name),S_look(classtable,now-
>faname));
    // 将父类的所有元素也复制到子类中并做一定的处理
    now->finish=1;
}
```

5. 对于主函数和每一个类中的成员函数的内容进行检查
    1. 纪录函数的返回值类型`returnType`在返回的时候需要检查
    2. 纪录函数所在的类，在使用`this`的时候需要做检查，注意main函数不可以使用this
    3. 检查每一个函数中声明的所有参数定义，检查他们的类型是否被定义以及他们的名字是否重复。检查完成后将他们存储到table中
    4. 检查每一个`stm`和`exp`，对于每一个`exp`需要返回一个`node`来表示`exp`的返回属性，如果发现错误就给出，感觉没有什么特别需要注意的地方。
6. 在typecheck阶段也为后续的IR操作做了很多准备
    1. 为IR树构建了对于class类引入的offset表

```
// classVar
if (isClassVar&&!S_look(classElementsOffset,S_Symbol(x->v))){
    int *now=checked_malloc(sizeof(int));
    *now=classElementsCnt++;
    S_enter(classElementsOffset,S_Symbol(x->v),now);
}
// class method
if (!S_look(classElementsOffset,S_Symbol(x->id))){
    int *cnt=checked_malloc(sizeof(int));
    *cnt=classElementsCnt++;
    S_enter(classElementsOffset,S_Symbol(x->id),cnt);
}
```

2. 对每一个函数都给出了他们函数名到Temp_namedlabel的映射

```
S_enter(tempFunctionName,S_Symbol(fuctionName),Temp_namedlabel(fuctionNa
me));
```

3. 前面提到的`TyAndInit`这类中`explist`只有在IR树中对class做初始化的时候有效，他会用来初始化所有的成员变量

```
if (isClassVar){
    TyAndInit ans=checked_malloc(sizeof(*ans));
    ans->ty=Ty_Int();
    ans->expList=x->elist;
    S_enter(table,S_Symbol(x->v),ans);
}
```

4. `classTable`这个table在IR树中也有用处

生成IR树

1. 相对于语法树，IR树的抽象程度更高，不会再考虑具体变量名只是用一个temp来表示，同时if和while这样的语句会被进一步解析成为跳转语句。
2. 准备的table和结构

```
S_table temp_table; // 用于纪录变量和temp之间的对应关系
extern S_table tempFunctionName;
extern S_table classElementsOffset;
extern int classElementsCnt;
extern S_table classtable;
struct whilenode_{ // 用于处理循环结构
    Temp_label testLable;
    Temp_label endLable;
    Temp_label loopLable;
    struct whilenode_* nxt;
};
typedef struct whilenode_* whilenode;
whilenode head; // 本身构成一个栈
```

3. 在实现的时候发现比较符号的赋值较为复杂，比如`a=8>7`因此我将这样的赋值变成了一个if语句

```
if (8>7) a=1;
else a=0;
```

4. 在处理or或者and等的需要短路性质，但是我不希望一个exp的转化被外部干扰，因此转换了形式，在if和while中也是同理的。

```
c=(a||b);
// =>
if (a){
    d=1;
}else{
    if (b){
        d=1;
    }else{
        d=0;
    }
}
c=d;
```

5. 我们希望数组的-1位置存储一下数组的长度，举一个例子我们的操作如下：

```
int []a={1,2}
//=>
b=malloc(4*3);
b[0]=3;
b[1]=1;
b[2]=2;
a=b+4;
```

```
T_exp arrayInit(A_expList list){
    Temp_temp temp=Temp_newtemp();
    int cnt=0;
    T_stm assignStm=NULL;
    for (;list;list=list->tail){
        A_exp x=list->head;
        T_stm
nowstm=T_Move(T_Mem(T_Binop(T_plus,T_Temp(temp),T_Const((cnt+1)*OFFSETSTEP))
),ast2treepExp(x));
        if (assignStm==NULL) assignStm=nowstm;
        else assignStm=T_Seq(assignStm,nowstm);
        cnt++;
    }
    if (assignStm)
assignStm=T_Seq(T_Move(T_Mem(T_Temp(temp)),T_Const(cnt)),assignStm);
    else assignStm=T_Move(T_Mem(T_Temp(temp)),T_Const(cnt));
    return
T_Eseq(T_Seq(T_Move(T_Temp(temp),T_ExtCall("malloc",T_ExpList(T_Const((cnt+1
)*OFFSETSTEP),NULL))),assignStm),T_Binop(T_plus,T_Temp(temp),T_Const(OFFSETS
TEP)));
}
```

6. 在IR中，一个类也只是一个普通的地址。因此在一开始给一个赋值的时候就要在相应位置给类的对应地址中存储成员变量或者成员函数。注意为了方便起见我们给所有的类都开同样大小的空间，成员相对应的偏移量只由成员的名字确定。成员名字和偏移量的关系在类型检查的时候存储子在了`classElementsOffset`中，成员函数对应的temp_name存储在了`tempFunctionName`，同时成员变量的初始化信息也存储在了`classtable`中，这个表的具体定义在类型检查中有详细说明

```
T_stm classInit(S_table table,Temp_temp temp){
    S_symbol key=table->top;
    if (!key) return NULL;
    TyAndInit ty=S_look(table,key);
    int offset=*(int*)S_look(classElementsOffset,key);
    T_stm now=NULL;
    if (ty->ty->kind==Ty_record){
        string functionName=
            getFuctionName(S_name(ty->ty->u.record->head-
>name),S_name(key));

 now=T_Move(T_Mem(T_Binop(T_plus,T_Temp(temp),T_Const(offset*OFFSETSTEP))),
           T_Name(S_look(tempFunctionName,S_Symbol(functionName))));
    }else if (ty->expList){
        if (ty->ty->kind==Ty_int){

 now=T_Move(T_Mem(T_Binop(T_plus,T_Temp(temp),T_Const(offset*OFFSETSTEP))),
                       ast2treepExp(ty->expList->head));
        }else{

 now=T_Move(T_Mem(T_Binop(T_plus,T_Temp(temp),T_Const(offset*OFFSETSTEP))),
                       arrayInit(ty->expList));
        }
    }else{
        now=T_Move(T_Mem(T_Binop(T_plus,T_Temp(temp),
                                 T_Const(offset*OFFSETSTEP))),T_Const(0));
    }
    TAB_pop(table);
    T_stm nxt=classInit(table,temp);
    S_enter(table,key,ty);
    if (now&&nxt) return T_Seq(now,nxt);
    else if (now) return now;
    else if (nxt) return nxt;
    return NULL;
}
```

7. 每一次使用一个类中的成员变量或者是成员函数的时候只需要查找偏移量表找到偏移量读出后使用即可，typecheck保证了对应偏移量中一定是有值的，否则在typecheck中就爆出类中没有改成员变量或函数。
8. 在class赋值的时候就考虑了多态问题，所以在调用的时候只要按照offset正常调用即可
9. 在处理`while`上，由于可能会进行迭代的循环，因此我维护了一个栈来保证`break`、`continue`等关键字可以跳转到正确的位置上。

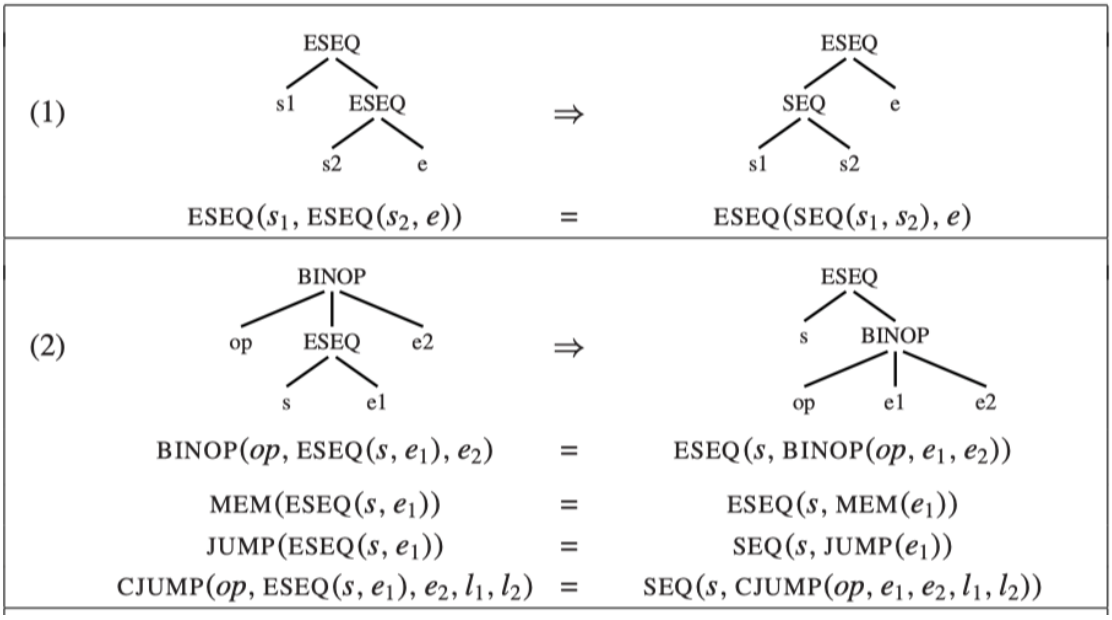
```
case A_whileStm:{
    whilenode now=newnode();
    T_stm ifstm=T_Cjump(T_ne,ast2treepExp(x->u.if_stat.e),
                        T_Const(0),now->loopLable,now->endLable);
```

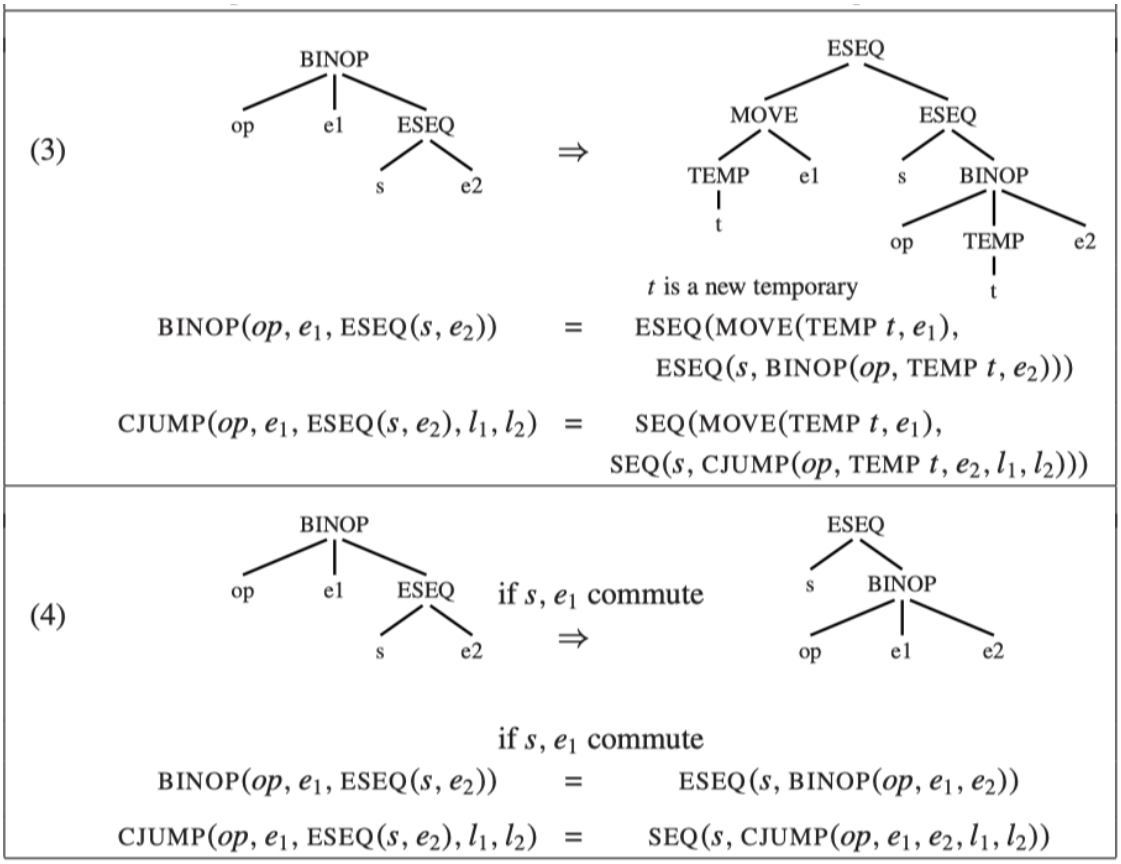
```
    T_stm loopstm=ast2treepStm(x->u.while_stat.s);
    if (loopstm)
        ans=T_Seq(T_Label(now->testLable),
                  T_Seq(ifstm,T_Seq(T_Label(now->loopLable),
                                    T_Seq(loopstm,T_Seq(T_Jump(now-
>testLable),
                                                        T_Label(now-
>endLable))))));
    else
        ans=T_Seq(T_Label(now->testLable),
                  T_Seq(ifstm,T_Seq(T_Label(now->loopLable),
                                    T_Seq(T_Jump(now->testLable),
                                          T_Label(now->endLable)))));
    popnode();
    break;
    case A_continue:{
        whilenode now=head->nxt;
        if (!now) printError(x->pos,"can not use continue out of while");
        ans=T_Jump(now->testLable);
        break;
    }
    case A_break:{
        whilenode now=head->nxt;
        if (!now) printError(x->pos,"can not use break out of while");
        ans=T_Jump(now->endLable);
    }
```

10. 经过了这一个操作之后，语法树变成了IR树，抽象程度变高，消去了数据类型的概念，同时命令减少IR树可以更加简单的翻译成汇编语言。

IR树线性化、成块以及特殊处理

1. 生成IR树的时候提到为了翻译的方便和灵活经常使用使用eseq和seq这种树结构作为返回值，但是实际的汇编代码是一个线性的结构。因此我们需要解构这些命令，将他们做等价转换。最后我们就会得到一个没有eseq和seq命令的IR树，这样的树是一个类似线性的结构，非常有利于后续的解析。





2. 成块的过程就是将IR树遍历后如果发现有跳转命令就进行一个分块，这样保证每一个块中所有的命令是顺序执行的，同时每一个块的第一个命令都是一个label标记，最后一个命令都是一个跳转指令。这样会更有益于后续的翻译分析。
3. 最后可能还要做一些特殊处理，比较典型的就是比如我们本来应该默认当函数中所有语句都执行完毕的时候应该会有一个return语句，但是很多时候这种问题不方便检查，并且在一些语言中也不会有强制的要求，但是在汇编等一些中必须要显示的返回语句，所以需要再最后加一个统一的返回语句。由于在IR树中已经没有了数据类型的概念，这里只要`return -1`即可

llvm的tile

1. tile的详情

```
%L = BOP i64 OP1, OP2
    MOVE(Temp,Binop(binop,e/temp/const,e/temp/const))
    MOVE(Temp,e/temp/Const)
    Binop(binop,e/temp/const,e/temp/const)
    Temp
    Const


%L = ptrtoint ptr OP to i64
    T_Name(Temp_namedlabel(s))


%L = BOP i64 OP1, OP2
%L = inttoptr i64 OP to ptr
store T OP1, ptr OP2
    MOVE(MEM(Binop(binop,e/temp/const,e/temp/const)),e/temp/const)
```

```
%L = inttoptr i64 OP to ptr
Store T OP2 ptr OP2
    MOVE(MEM(e/temp/const),e/temp/const)


%L = BOP i64 OP1, OP2
%L = inttoptr i64 OP to ptr
%L = load T, ptr OP
    MOVE(Temp,MEM(Binop(binop,e/temp/const,e/temp/const)))
    MEM(Binop(binop,e/temp/const,e/temp/const))

%L = inttoptr i64 OP to ptr
%L = load T, ptr OP
    MOVE(Temp,MEM(e/temp/const))
    MEM(e/temp/const)


%L = icmp CND i64 OP1, OP2
br i1 <cond>, label <iftrue>, label <iffalse>
    CJUMP(relop,e/temp/const,e/temp/const,l1,l2)


br label <dest>
    JUMP(l1)


call void OP1(T2 OP2, ... ,TN OPN)
    ExtCall("starttime",NULL)
    ExtCall("stoptime",NULL)
    ExtCall("putint",e/temp/const)
    ExtCall("putch",e/temp/const)

%L = inttoptr i64 OP to ptr
call void OP1(T2 OP2, ... ,TN OPN)
    ExtCall("putarray",e/temp/const,e/temp/const)


%L = call T OP1(T2 OP2, ..., Tn OPN)
    ExtCall("getint",NULL)
    ExtCall("getch",NULL)
    MOVE(temp,ExtCall("getint",NULL))
    MOVE(temp,ExtCall("getch",NULL))



%L = inttoptr i64 OP to ptr
%L = call T OP1(T2 OP2, ..., Tn OPN)
    ExtCall("getarray",e/temp/const)
    Call(s,e,e/temp/const,...)
    MOVE(temp,ExtCall("getarray",e/temp/const))
    MOVE(temp,Call(s,e,e/temp/const,...))


%L = call T OP1(T2 OP2, ..., Tn OPN)
%L = ptrtoint ptr OP to i64
```

```
    ExtCall("malloc",ExtCall(e/temp/const,NULL))
    MOVE(temp,ExtCall("malloc",ExtCall(e/temp/const,NULL)))


ret void and ret i64 OP
    Return(e/temp/Const)
```

2. `AS_instrList treep2llvmExp(T_exp x,Temp_temp* rettemp,bool canMiss)`在这个函数的设计上做了一点小工作，rettemp如果是不是NULL的话我就认为这个exp外面可能是一个赋值操作，他已经有了一个意向的temp来存储值，否则我认为这是一个中间步骤会生成一个临时的temp并给`*rettemp`，canmiss表示的是这个exp是否可以忽略，只在exp是temp的时候有效

```
Temp_temp ret;
if (*rettemp) ret=*rettemp;
else ret=Temp_newtemp();
```

```
case T_TEMP:{
    if(canMiss&&*rettemp==NULL) ret=x->u.TEMP;
    else ans->head=OI("%`d0 = add i64 %`s0, 0", T(ret), T(x->u.TEMP), NULL);
    break;
}
```

3. 感觉代码上没什么要说的，就是根据tile的简单翻译

arm的tile

1. tile的详情

```
binop destReg , srcReg , op2
    MOVE(Temp,Binop(binop,e/temp,e/temp/const))
    Binop(binop,e/temp,e/temp/const)

mov destReg , op2
    MOVE(Temp,e/temp/Const)
    Temp
    Const


str  label, [desreg]
    MOVE(MEM(e),T_Name)


str srcReg , [locationReg]
    MOVE(MEM(e),e)

ldr destReg , [locationReg]
    MOVE(Temp,MEM(e))
    MEM(e)

cmp Reg1, op2
(beq,bne,...) iffalse
    CJUMP(relop,e,e,l1,l2)
```

```
b label
    JUMP(l1)

push{Op1} (多于4个的参数)
mov t_i Op1 (编号小于4的参数)
blx label
    call(str,e,args)

push{Op1} (多于4个的参数)
mov t_i Op1 (编号小于4的参数)
bl reg
    ExtCall(str,args)

mov r0 Op1
mov t13, t11
pop {t11}
bx returl_temp
ret void and ret i64 OP
    Return(e)
```

2. 除了换了一下tile的方式其他部分和llvm几乎一样没什么要说的
3. 有一个地方需要注意的是在调用函数的时候由于`lr,r0~r4`会发生改变，即在函数调用的过程中这些寄存器会发生改变，所以需要再函数调用的时候的将这些寄存器标注为使用以及定义，这样才能在后续的寄存器分配中得到正确的结果。

产生块分析，活跃度分析等

1. 在tile的过程中我们会给出每一个命令对应的des变量，src变量以及跳转变量(如果没有跳转到一个label就默认是跳转到下一条指令)
2. 这里构建一个图结构，这若干个分析都需要用到这个图结构
3. 经过块分析之后我们就得到了每一个块的相关信息，知道了一个块的前驱后序等。这里的块和在做IR树的分块的含义一样。
4. 将多个块进行合并，可以得到一个完成汇编语言，这个时候做每一个语句的前驱后序，那么如果不是跳转语句后继就是下一句，否则就是跳转到的位置(有可能是多个)
5. 对每一个变量做活跃性分析，考虑每一个语句中使用了什么变量，定义了什么变量。随后综合考虑一个语句需要从外部接受什么样的变量，需要向下传递哪一些变量。
   $in[n] = use[n] \cup (out[n] - def[n])$，$out[n] = \bigcup_{s \in succ[n]} in[s]$使用这两个式子不断迭代知道稳定为止
6. 经过活跃度分析我们知道了每一个语句中需要保留的变量，这样我们可以得到变量之间的冲突关系，每一个语句中def[n]和out[n]是会产生冲突的

SSA

1. 静态单复制是llvm可以被工具接受的必要结果，同时llvm也可以让我们十分方便的对代码进行优化，是一个十分重要的技术。
2. 准备工作，这里面table的定义含义和书本几乎一样，table里面的key和value详细写明了。需要注意的就是`notDOfBlocks`是一个反向的表，如果两个lable之间没有支配关系则被存储。这是由于迭代关系是一个与关系，我认为使用这种反向表示会更加便于编码。

```
S_table label2Block;
S_table notDOfBlocks; // key=block.label value=S_table(block.label,bool)
S_table DFOfBlocks; // key=block.label value=S_table(block.label,bool)
S_table stackOfParams; //key=Temp_temp value=Temp_tempList
S_table originOfBlock; // key=block.label value=S_table(Temp_temp,bool)
S_table inOfBlock; // key=block.label value=S_table(Temp_temp,bool)
S_table defsites; // key=Temp_temp value=S_table(Label,bool)
S_table phiBlock; // key=S_label value=S_table(Temp_temp,bool)
S_table visBlock; // key=S_label value=bool(isvis)
```

3. 不断迭代得到块之间的支配关系

```
void getDOfBlocks(G_nodeList list){
    Temp_label headLable=((AS_block)G_nodeInfo(list->head))->label;
    S_enter(label2Block,headLable,list->head);
    for (G_nodeList x=list->tail;x;x=x->tail){
        Temp_label nowLable=((AS_block)G_nodeInfo(x->head))->label;
        S_enter(label2Block,nowLable,x->head);
        enter_table(notDOfBlocks,headLable,nowLable,FALSE);
    }
    bool change=TRUE;
    while(change){
        change=FALSE;
        for (G_nodeList x=list->tail;x;x=x->tail){
            Temp_label nowLable=((AS_block)G_nodeInfo(x->head))->label;
            for (G_nodeList y=G_pred(x->head);y;y=y->tail){
                change|=mergeTable(nowLable,((AS_block)G_nodeInfo(y->head))-
>label);
            }
        }
    }
}
```

4. 找到支配边界，直接用定义寻找即可

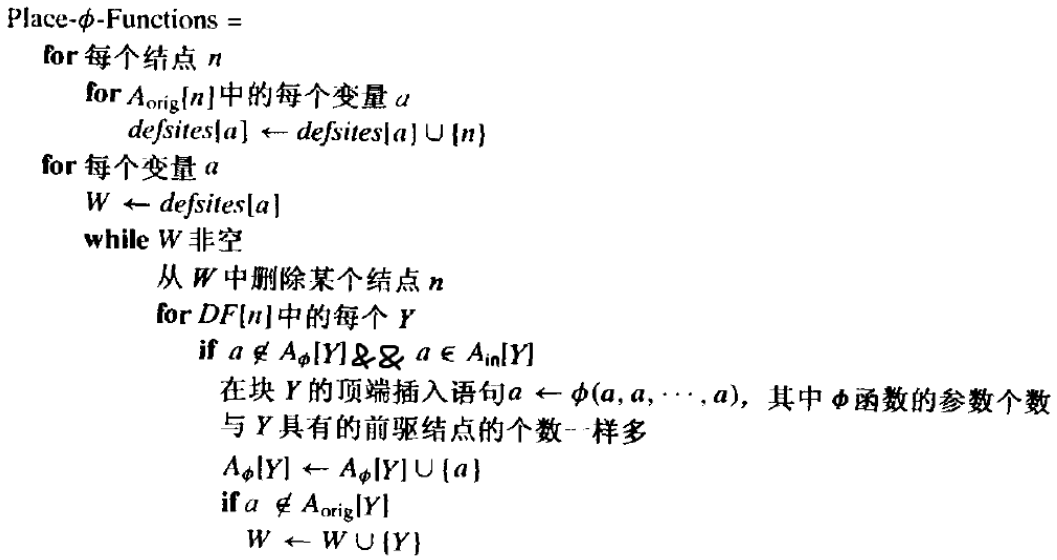
```
bool isDF(G_node x,G_node w){
    Temp_label xLabel=((AS_block)G_nodeInfo(x))->label;
    Temp_label wLabel=((AS_block)G_nodeInfo(w))->label;
    for (G_nodeList predList=G_pred(w);predList;predList=predList->tail){
        G_node p=predList->head;
        Temp_label pLabel=((AS_block)G_nodeInfo(p))->label;
        if (isDomin(xLabel,pLabel)&&(!
(isDomin(xLabel,wLabel)&&wLabel!=xLabel)))
            return TRUE;
    }
    return FALSE;
}
```

5. 得到每一个块的需要的in和def的信息

```
void initOriginAndIn(G_nodeList ig){
    Temp_label nowLabel=NULL;
    for (G_nodeList ttt=ig;ttt;ttt=ttt->tail){
```

```
        G_node x=ttt->head;
        AS_instr nowinstr=((AS_instr)G_nodeInfo(x));
        if (nowinstr->kind==I_LABEL){
            nowLabel=nowinstr->u.LABEL.label;
            for (Temp_tempList tempList=FG_In(x);tempList;tempList=tempList-
>tail){
                enter_table(inOfBlock,nowLabel,tempList->head,FALSE);
            }
        }else{
            for (Temp_tempList
tempList=FG_def(x);tempList;tempList=tempList->tail){
                enter_table(originOfBlock,nowLabel,tempList->head,FALSE);
            }
        }
    }
}
```

6. 添加phi信息（书本中的伪代码有一点问题，这幅图中已经做出了修改）



7. 改名，包括更改内部变量的名字和更改phi函数内的名字（书本中的伪代码使用了DF树，但是我的方法中没有构建，所以我对这个算法做出了一些改变，这幅图中已经做出了修改）

```
初始化:
    for 每一个变量a
        Count[a] ← 0
        Stack[a] ← empty
        将0压入Stack[a]
Rename(n) =
    for 基本块n中的每一个语句S
        if S不是φ函数
          for S中某个变量x的每一个使用
              i ← top(Stack[x])
              在S中用x_i替换x的每一个使用
        for S中某个变量a的每个定值
            Count[a] ← Count[a] + 1
            i ← Count[a]
            将i压入Stack[a]
            在S中用a_i替换a的定值
    for基本块n的每一个后继Y,
        设n是Y的第j个前驱
        for Y中的每一个φ函数
            设该φ函数的第j个操作数是a
            i ← top(Stack[a])
            用a_i替换第j个操作数
    for n的每一个后继X
        Rename(X)
    for 原来的S中的某个变量a的每一个定值
        从Stack[a]中弹出栈顶元素
```

寄存器分配

1. 如果你需要的目标函数是assem的话那么实际上你只有有限的寄存器可以使用，你需要将无限的temp分配给有限的寄存器中进行操作。
2. 根据变量的冲突关系来建立一个栈，这个栈是一个维护了将来染色的顺序，一个边数小于k的节点会优先加入栈，加入栈的节点所有的边后会断开(在这个步骤认为断开，实际可以通过技巧不更改真实的数据机构)。最后可能会加入一些边数大于等于k的节点，这些节点是有可能需要做splile的节点。这里为了更好的时间复杂度，使用了类似拓扑排序的栈纪录了所有可以度数小于k的节点

```
void simplify(G_nodeList ig){
    bool finish=0;
    while (!finish){
        while (canInStack){
            Temp_temp x=canInStack->head;
            G_node xnode=(G_node)TAB_look(tempToNode,x);
            canInStack=canInStack->tail;
            colorstack=Temp_TempList(x,colorstack);
            G_nodeList adj=G_succ(xnode);
            for (;adj;adj=adj->tail){
```

```
                Temp_temp nowTemp=(Temp_temp)G_nodeInfo(adj->head);
                if (TAB_look(inCanInStack,nowTemp)) continue;
                int *num=TAB_look(fanout,nowTemp);
                assert(num);
                *num--;
                if (*num<canUserRegister){
                    canInStack=Temp_TempList(nowTemp,canInStack);
                    TAB_enter(inCanInStack,nowTemp,emptyPoint);
                }
            }
        }
        finish=1;
        for (G_nodeList list=ig;list;list=list->tail){
            Temp_temp nowTemp=(Temp_temp)G_nodeInfo(list->head);
            if (!TAB_look(inCanInStack,nowTemp)){
                finish=0;
                canInStack=Temp_TempList(nowTemp,canInStack);
                TAB_enter(inCanInStack,nowTemp,emptyPoint);
                break;
            }
        }
    }
}
```

3. 根据栈的顺序对节点k染色，无法染色的节点只能splile到栈中，用一个哈希表纪录对应节点会splle到的位置方便后续的操作。

```
void selectColor(){
    while (colorstack){
        Temp_temp xTemp=colorstack->head;
        G_node xnode=TAB_look(tempToNode,xTemp);
        colorstack=colorstack->tail;
        int vis=0;
        G_nodeList adj=G_succ(xnode);
        colorNode nowcNode=checked_malloc(sizeof(*nowcNode));
        for (;adj;adj=adj->tail){
            Temp_temp nowTemp=(Temp_temp)G_nodeInfo(adj->head);
            colorNode cNode=TAB_look(colorTable,nowTemp);
            if (cNode==NULL) continue;
            if (cNode->isSpill) continue;
            vis|=1<<cNode->renum;
        }
        nowcNode->isSpill=1;
        for (int i=0;i<canUserRegister;i++){
            if ((vis&(1<<sx[i]))==0){
                nowcNode->isSpill=0;
                nowcNode->renum=sx[i];
                mxrenum=mxrenum>i?mxrenum:i;
                break;
            }
        }
        if (nowcNode->isSpill){
            nowcNode->renum=spillOffset;
            spillOffset+=OFFSETSTEP;
        }
```

```
            TAB_enter(colorTable,xTemp,nowcNode);
        }
    }
```

4. 对所有的temp进行rename，不需要splile的节点直接更改，需要的节点如果是des就在赋值后马上存入栈，否则就从栈中读出后马上使用。我们为了方便留下了r8-r10的寄存器专门给splile的节点使用。

```
AS_instrList renameColor(AS_instrList il){
    AS_instrList pre=NULL;
    for (AS_instrList list=il;list;list=list->tail){
        AS_instr x=list->head;
        if (x->kind==I_LABEL){
            pre=list;
            continue;
        }
        Temp_tempList desList=x->u.OPER.dst,srcList=x->u.OPER.src;
        int ptrnum=8;
        for (;srcList;srcList=srcList->tail){
            colorNode nowcNode=(colorNode)TAB_look(colorTable,srcList-
>head);
            if (nowcNode->isSpill){
                sprintf(des,"ldr `d0 , [`s0 , #-%d]",nowcNode-
>renum+mxrenum*4+8);
                AS_instr y=OI(String(des),T(r(ptrnum)),T(fp),NULL);
                if (pre==NULL){
                    pre=il=AS_InstrList(y,list);
                    pre->tail=list;
                }else{
                    pre->tail=AS_InstrList(y,list);
                    pre=pre->tail;
                }
                srcList->head=r(ptrnum);
                ptrnum++;
            }else{
                srcList->head=r(nowcNode->renum);
            }
        }
        for (;desList;desList=desList->tail){
            colorNode nowcNode=(colorNode)TAB_look(colorTable,desList-
>head);
            if (nowcNode->isSpill){
                sprintf(des,"str `s0 , [`s1 , #-%d]",nowcNode-
>renum+mxrenum*4+8);
                AS_instr y=OI(String(des),NULL,TL(r(ptrnum),T(fp)),NULL);
                list->tail=AS_InstrList(y,list->tail);
                list=list->tail;
                desList->head=r(ptrnum);
                ptrnum++;
            }else{
                desList->head=r(nowcNode->renum);
            }
        }
        pre=list;
    }
```

```
    return il;
}
```

5. 注意保存在寄存器分配中使用了哪一些寄存器在开始后的时候需要保存下来

```
AS_instrList prolog=I(OI("push {`s0,`s1}",NULL,TL(fp,T(lr)),NULL));
prolog=AS_splice(prolog,I(OI("add `d0, `s0, #4",T(fp),T(sp),NULL)));
for (int i=4;i<=mxrenum;i++){
    prolog=AS_splice(prolog,I(OI("push {`s0}",NULL,T(r(i)),NULL)));
}
for (int i=8;i<=10;i++){
    prolog=AS_splice(prolog,I(OI("push {`s0}",NULL,T(r(i)),NULL)));
}
sprintf(des,"sub `d0, `s0, #%d",spillOffset);
prolog=AS_splice(prolog,I(OI(String(des),T(sp),T(sp),NULL)));
sprintf(des,"add `d0, `s0, #%d",spillOffset);
AS_instrList epilog=I(OI(String(des),T(sp),T(sp),NULL));
for (int i=10;i>=8;i--){
    epilog=AS_splice(epilog,I(OI("pop {`d0}",T(r(i)),NULL,NULL)));
}
for (int i=mxrenum;i>=4;i--){
    epilog=AS_splice(epilog,I(OI("pop {`d0}",T(r(i)),NULL,NULL)));
}
epilog=AS_splice(epilog,I(OI("pop {`d0,`d1}",TL(fp,T(lr)),NULL,NULL)));
epilog=AS_splice(epilog,I(OI("bx `s0",NULL,T(lr),NULL)));
```

6. 由此就得到了真实可以使用的assem代码。

对Example.fmj的分析（仅分析重点部分）

原码

1. 原码

```
public int main() {
    int[] a={0};
    int i=0;
    int l;
    class c1 o1;
    class c1 o2;
    a=new int[getint()];
    l=length(a);
    o1=new c1();
    o2=new c2();
    while (i< ({i=i+1;} l))
        if ( (i/2)*2 == i )
            a[i-1]=o1.m1(i);
        else
            a[i-1]=o2.m1(i);
    putarray(l, a);
    return l;
}
public class c1 {
    int i1=2;
```

```
    public int m1(int x) {
      return this.i1;
    }
}
public class c2 extends c1 {
    int i2=3;
    public int m1(int x) {
        return this.i2+x;
    }
}
```

词法分析、语法分析、类型检查

1. 感觉这一块没什么要说的。总之经过了这三步之后得到了一颗语法树，并且对这颗语法树初步分析了继承关系每一个类的成员变量和成员函数以及他们所在的偏移量等关系。由于这个代码通过了类型检查所以才能进行到下一步。

生成IR树（为了方便这里就会展示线性化后的代码）

1. 数组的定义`int[] a={0};`

```
Move t108, malloc 8
Move Mem(t108), 1
Move Mem(Binop(T_plus, t108, 4)), 0
Move t107, Binop(T_plus, t108, 4)
```

2. class的定义和多态的使用，针对成员变量和成员函数存储的具体位置是在typecheck的时候确定的。可以看到下面无论是什么样的类都留出了12个空间。对于c1类，4位置存储的是c1$m1而对于c2类4位置存储的是c2$m1这里体现了多态的性质。同时可以看到c2类的0号位置也存储了一个`i1=2`体现了继承的相关性质

```
Move t118, malloc 12
Move Mem(Binop(T_plus, t118, 4)), c1$m1
Move Mem(Binop(T_plus, t118, 0)), 2
Move t111, t118
Move t117, malloc 12
Move Mem(Binop(T_plus, t117, 0)), 2
Move Mem(Binop(T_plus, t117, 4)), c2$m1
Move Mem(Binop(T_plus, t117, 8)), 3
Move t112, t117
```

IR树线性化、成块以及特殊处理

1. `int[] a={0};`

```
T_Move(
T_Temp(TC(String("t107"))),
T_Eseq(
  T_Seq(
  T_Move(
    T_Temp(TC(String("t108"))),
    T_ExtCall(String("malloc"),
    T_ExpList(T_Const(8), NULL))),
```

```
  T_Seq(
    T_Move(
    T_Mem(
      T_Temp(TC(String("t108")))),
    T_Const(1)),
    T_Move(
    T_Mem(
      T_Binop(T_plus,
      T_Temp(TC(String("t108"))),
      T_Const(4))),
    T_Const(0)))),
  T_Binop(T_plus,
  T_Temp(TC(String("t108"))),
  T_Const(4))))
```

以上是修改前的样子，线性化将eseq中的seq内容慢慢提取到前面的位置，并最后做一个赋值

```
Move t108, malloc 8
Move Mem(t108), 1
Move Mem(Binop(T_plus, t108, 4)), 0
Move t107, Binop(T_plus, t108, 4)
```

2. `i< ({i=i+1;} l)`，由于代码较为复杂，这里我口述一下过程，这一部分就会变成如下

```
j=i;
i=i+1;
j<l
```

3. 做了一个分块，这个每一个label为一个快
4. 最后加了一个块

```
L20:
  Return -1
```

tile、产生块分析，活跃度分析等、SSA和寄存器分配

1. 这里面的东西比较的程式化，都是根据已有的伪代码进行编程实现没有太多创新思维的部分，并且逻辑比较繁杂可以直接查看第一部分理解。

运行项目

1. 在linux环境下输入make a.out 可以得到转换程序
2. make runarm 可以测试所有测试代码，将语法树输出到.ast文件中，将IR树输出到.ir中，将进行线性化的IR树输出到.stm中，将流等分析信息输出到.liv中，将转换出来的assem代码到.s文件,并使用模拟器运行
3. make runarm 可以测试所有测试代码，将语法树输出到.ast文件中，将IR树输出到.ir中，将进行线性化的IR树输出到.stm中，将流等分析信息输出到.liv中，将转换出来的llvm代码到.ll文件,并使用模拟器运行
4. make run 先后运行runarm和runllvm
5. 在定义的时候没有初始化的参数全都定义为0，包括所有类型

6. 对于没有定义的数组和Class类型相当于指针是NULL,如果不进行初始化就使用的话就会在运行的时候产生段错误，我认为这是正常现象
7. 几乎所有的测试都可以直接跑过，除了下述的测试中需要有一些输入
8. `mctest04` 重点用于测试libsys，一开始是对getarrrray的测试不要输入超过10个数字。随后会输出前10个质数。后面是一个默比乌斯反言的模板题，你需要先输入一个T表示测试的组数，最后依次输入a,b,c,d。程序会给出满足x∈[c,a],y∈[d,b]且gcd(x,y)=1的(x,y)组数
9. `example` 需要给出一个输入来确定数组的规模
10. `fibonacci` 需要给出一个数字n，输出1~n的斐波那契数列数值，如果n大于47则会输出-1
11. `ttest06` 和 `ttest05` 的输入在测试中用处不大，随便输入一个数字即可