# *Compiler*

## *Step By Step*

上海交通大学计算机系

编译原理课程设计


钱风 @ 上海交通大学

计算机科学与工程系

E-mail: shinyflute@gmail.com



2005 年 8 月第一版

最后更新 2005 年 9 月 8 日

# *Compiler*

## *Step By Step*

Course Design in Compiler Theory

**Feng Qian @ SJTU**

Department of Computer Science and Engineering

E-mail: shinyflute@gmail.com

First Edition: Aug 2005

Last Update: Sep 8, 2005

# 目录

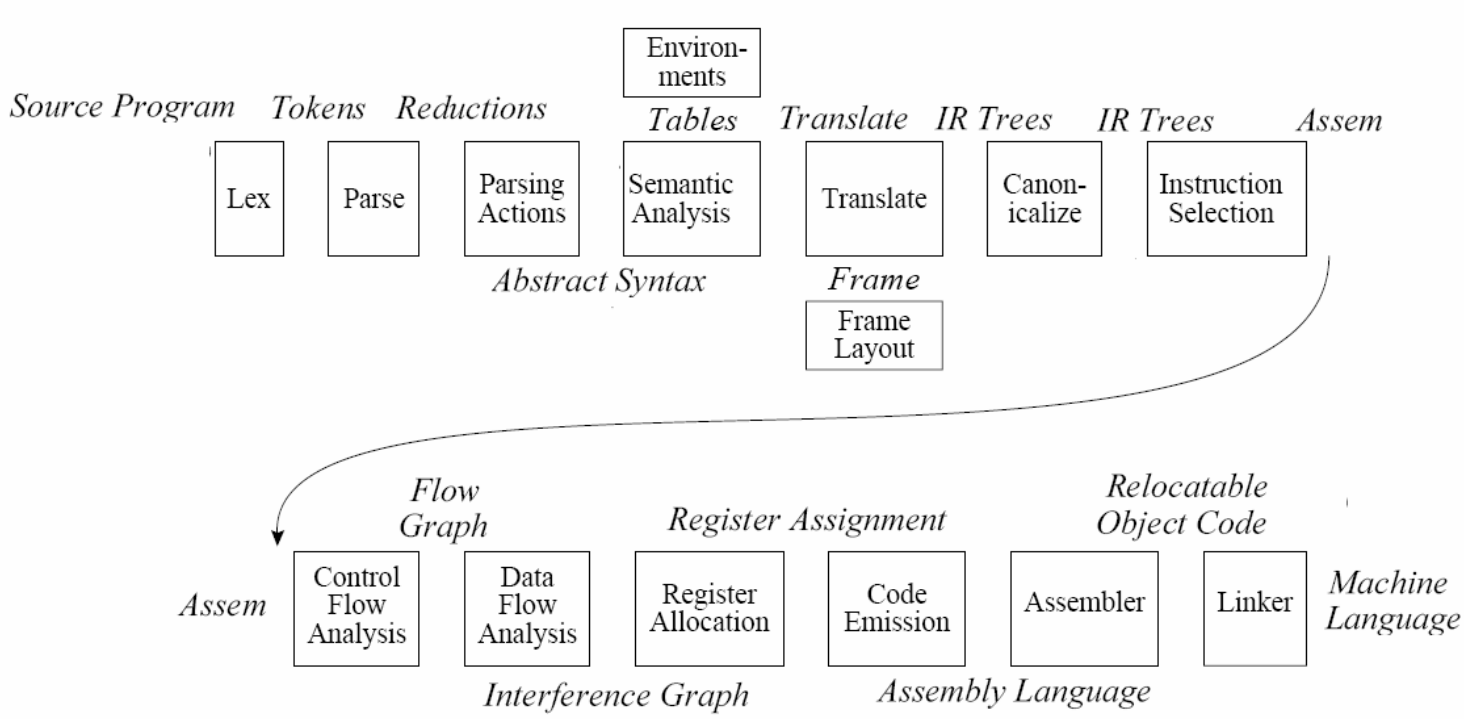
# 第 0 部分 项目概览

## 0.1 项目目标

制作 Tiger 语言编译器

## 0.2 主要步骤

1. 词法分析 (Lexical Analysis) :把 Tiger 源程序分割成符号(单词),要求制作自动机文件, 词法分析程序由工具 JFlex 生成
2. 语法分析 (Syntax Parsing) :识别程序的语法结构,检查语法错误,要求制作文法文件, 语法分析程序由工具 Java Cup 生成
3. 抽象语法树 (Abstract Syntax Tree) :根据语法结构生成抽象语法树
4. 语义分析 (Semantic Analysis) :进行变量和类型检查等
5. 活动记录 (Activation Record) :与函数调用相关的活动记录
6. 中间代码生成 (Intermediate Code) :生成中间表示树 (IR Trees),它与程序和机器语言无关
7. 规范化 (Canonicalize) :优化表达式、条件分支等,只需复制代码
8. 指令选择 (Instruction Selection): 生成基本的 MIPS 汇编指令
9. 活性分析与寄存器分配 (Liveness Analysis and Register Allocation)
10. 使之成为整体: 生成完整的编译器程序




## 0.3 Tiger 语言语法概要

类型:

| | |
|---|---|
| 整数 | **int** |
| 字符串 | **string** |
| 数组 | **array of** int |

| | |
|---|---|
| 记录 | {x: int, y: string} |
| Name 类型 | 预先代替未知的类型.例如:<br>type typeA={a:typeB}<br>此时 typeB 是未知的类型,先用 Name 表示 |

声明:

| | |
|---|---|
| 函数 | **function** a (p1:int, p2:string) = ( … ) |
| 类型 | **type** mytype = {x: int, y: string} |
| 变量 | **var** row := mytype [10] of 0 |

表达式:

| | |
|---|---|
| 空表达式 | **nil** |
| 整数表达式 | 123 |
| 字符串表达式 | “abc” |
| 列表表达式 | 123 ; “abc” |
| if 表达式 | **if** … **then** … **else** … |
| let 表达式 | **let** … **in** … **end** |
| 关系表达式 | a>b |
| 变量表达式 | mystr |
| 记录表达式 | {x: int, y: string} |
| for 表达式 | **for** i:=1 to 10 **do** … |
| break 表达式 | **break** |
| 数组表达式 | mytype [10] of 0 |
| 测试相等表达式 | a=b 或 a<>b |
| 赋值表达式 | a:=5 |
| 计算表达式 | a+3 |
| 函数调用表达式 | func (1) |
| while 表达式 | **while** (a>0) **do** … |

Tiger 语言没有语句的概念,程序是由表达式组成的

# 第 1 部分 词法分析

## 1.1 Tiger.flex 清单 (蓝色部分为注释)

```
package Parse;
import ErrorMsg.ErrorMsg;

%%
%% 符号将词法文件分成 3 部分, 第一部分用户代码;第二部分选项和声明;第三部分自动机规则

%implements Lexer   实现的接口名
%function nextToken 扫描函数的名称,默认为 yylex
%type java_cup.runtime.Symbol 每个扫描元素的返回类型
%char 打开这个开关,yychar 变量将统计符号数(从 0 开始)

接下来定义一些用户函数,如出错报告等
%{
StringBuffer string = new StringBuffer();
int count;
private void newline() {
  errorMsg.newline(yychar);
}

private void err(int pos, String s) {
  errorMsg.error(pos,s);
}

private void err(String s) {
  err(yychar,s);
}

private java_cup.runtime.Symbol tok(int kind, Object value) {
    return new java_cup.runtime.Symbol(kind, yychar, yychar+yylength(),
value);
}

private ErrorMsg errorMsg;

Yylex(java.io.InputStream s, ErrorMsg e) {
  this(s);
  errorMsg=e;
}
```

```
%}

扫描完 Tiger 源程序后,状态不可能为注释或字符串中间,如果遇到下面的情况,报错
%eofval{
    {if (yystate()==COMMENT) err("Comment symbol don't match!");
     if (yystate()==STRING) err("String presentation error!");
     if (yystate()==STRING1) err("String presentation error!");
     return tok(sym.EOF, null);
       }
%eofval}

定义换行符、标识符、空白符等
JFlex 支持下面的符号
a|b 或;a b 连接;a* 0 次或多次重复;a+ 1 次或多次重复; a? 可选记号;a! 不能出现该记号;~a
仅匹配出现该记号第一次;a{n} 连续出现 n 次;a{n,m} 至少出现 n 次,之多出现 m 次;(a) 就是 a
LineTerminator = \n|\r|\r\n|\n\r
Identifier = [a-zA-Z][:jletterdigit:]*
DecIntegerLiteral = [0-9]+
WhiteSpace = \n|\r|\r\n|\t|\f|\n\r

定义三种状态:字符串中,注释中和\...\中
%state STRING
%state COMMENT
%state STRING1

%%

YYINITIAL 表示初始状态
<YYINITIAL> \"  {string.setLength(0);yybegin(STRING);}
<YYINITIAL> "/*" {count=1;yybegin(COMMENT);} 转入注释状态

<YYINITIAL> "*/" {err("Comment symbol don't match!");}
<YYINITIAL> " " {}
<YYINITIAL> {WhiteSpace} {}
<YYINITIAL> "," {return tok(sym.COMMA, null);}
<YYINITIAL> "/" {return tok(sym.DIVIDE,null);}
<YYINITIAL> ":" {return tok(sym.COLON,null);}
<YYINITIAL> "else" {return tok(sym.ELSE,null);}
<YYINITIAL> "do" {return tok(sym.DO,null);}
<YYINITIAL> "nil" {return tok(sym.NIL,null);}
<YYINITIAL> "|" {return tok(sym.OR,null);}
<YYINITIAL> ">" {return tok(sym.GT,null);}
<YYINITIAL> ">=" {return tok(sym.GE,null);}
```

```
<YYINITIAL> "<" {return tok(sym.LT,null);}
<YYINITIAL> "<=" {return tok(sym.LE,null);}
<YYINITIAL> "of" {return tok(sym.OF,null);}
<YYINITIAL> "-" {return tok(sym.MINUS,null);}
<YYINITIAL> "array" {return tok(sym.ARRAY,null);}
<YYINITIAL> "type" {return tok(sym.TYPE,null);}
<YYINITIAL> "for" {return tok(sym.FOR,null);}
<YYINITIAL> "to" {return tok(sym.TO,null);}
<YYINITIAL> "in" {return tok(sym.IN,null);}
<YYINITIAL> "end" {return tok(sym.END,null);}
<YYINITIAL> ":=" {return tok(sym.ASSIGN,null);}
<YYINITIAL> "." {return tok(sym.DOT,null);}
<YYINITIAL> "if" {return tok(sym.IF,null);}
<YYINITIAL> ";" {return tok(sym.SEMICOLON,null);}
<YYINITIAL> "while" {return tok(sym.WHILE,null);}
<YYINITIAL> "var" {return tok(sym.VAR,null);}
<YYINITIAL> "(" {return tok(sym.LPAREN,null);}
<YYINITIAL> ")" {return tok(sym.RPAREN,null);}
<YYINITIAL> "[" {return tok(sym.LBRACK,null);}
<YYINITIAL> "]" {return tok(sym.RBRACK,null);}
<YYINITIAL> "{" {return tok(sym.LBRACE,null);}
<YYINITIAL> "}" {return tok(sym.RBRACE,null);}
<YYINITIAL> "let" {return tok(sym.LET,null);}
<YYINITIAL> "+" {return tok(sym.PLUS,null);}
<YYINITIAL> "then" {return tok(sym.THEN,null);}
<YYINITIAL> "function" {return tok(sym.FUNCTION,null);}
<YYINITIAL> "=" {return tok(sym.EQ,null);}
<YYINITIAL> "break" {return tok(sym.BREAK,null);}
<YYINITIAL> "&" {return tok(sym.AND,null);}
<YYINITIAL> "*" {return tok(sym.TIMES,null);}
<YYINITIAL> "<>" {return tok(sym.NEQ,null);}
<YYINITIAL> {Identifier} {return tok(sym.ID,yytext());}
<YYINITIAL> {DecIntegerLiteral}
{return tok(sym.NUM,new Integer(yytext()));}
<YYINITIAL> [^] {err("Illegal character < "+yytext()+" >!");}

<STRING> {
    \"      {yybegin(YYINITIAL);return tok(sym.STR,string.toString());}
    要特殊处理:ASCII 码转义字符,\t,\n,\",\\,\
    \\[0-9][0-9][0-9]
        { int tmp=Integer.parseInt(yytext().substring(1, 4));
        if(tmp>255) err("exceed \\ddd"); else string.append((char)tmp);}
    [^\n\t\"\\]+    {string.append(yytext());}
    \\t {string.append('\t');}
```

```
    \\n     {string.append('\n');}
    \\\"    {string.append('\"');}
    \\\\    {string.append('\\');}
    {LineTerminator} {err("String presentation error!");}
    \\      {yybegin(STRING1);}
}

<STRING1> {
   {WhiteSpace} {} 在\...\中间空白字符忽略,这样利于字符串换行
   " " {}
    \\ {yybegin(STRING);}
    \" {err("\\dont match");}
    [^] {string.append(yytext());}
}

使用计数器匹配注释
<COMMENT> {
    "/*"    {count++;}
    "*/"    {count--;if (count==0) {yybegin(YYINITIAL);}}
    [^]     {}
}
```

### 1.2 JFlex 的使用

参数都使用默认的,运行 bin 目录下的 jflex.bat,在 GUI 界面选择上述输入文件和输出文件夹,则会生成词法分析程序 yylex.java

### 1.3 在本阶段中遇到的主要问题

四种状态的转换

注释的处理

# 第 2 部分 语法分析

# 第 3 部分 抽象语法树

### 2.1 Symbol 包

Symbol 包中封装了符号 (Symbol.Symbol) 和符号表 (Symbol.Table)

其中常用函数:

Symbol.symbol (String) 静态函数,如果字符串表示的符号存在,则返回原来的符号,否则新建一个与字符串对应的符号

Table.put (Symbol, Object) 将 “符号－对象”序对放入当前符号表中

Table.get (Symbol) 返回当前符号表中符号所对应的对象

Table.BeginScope() 在当前符号表中新建一个子符号表

Table.EndScope() 退回到上一级符号表

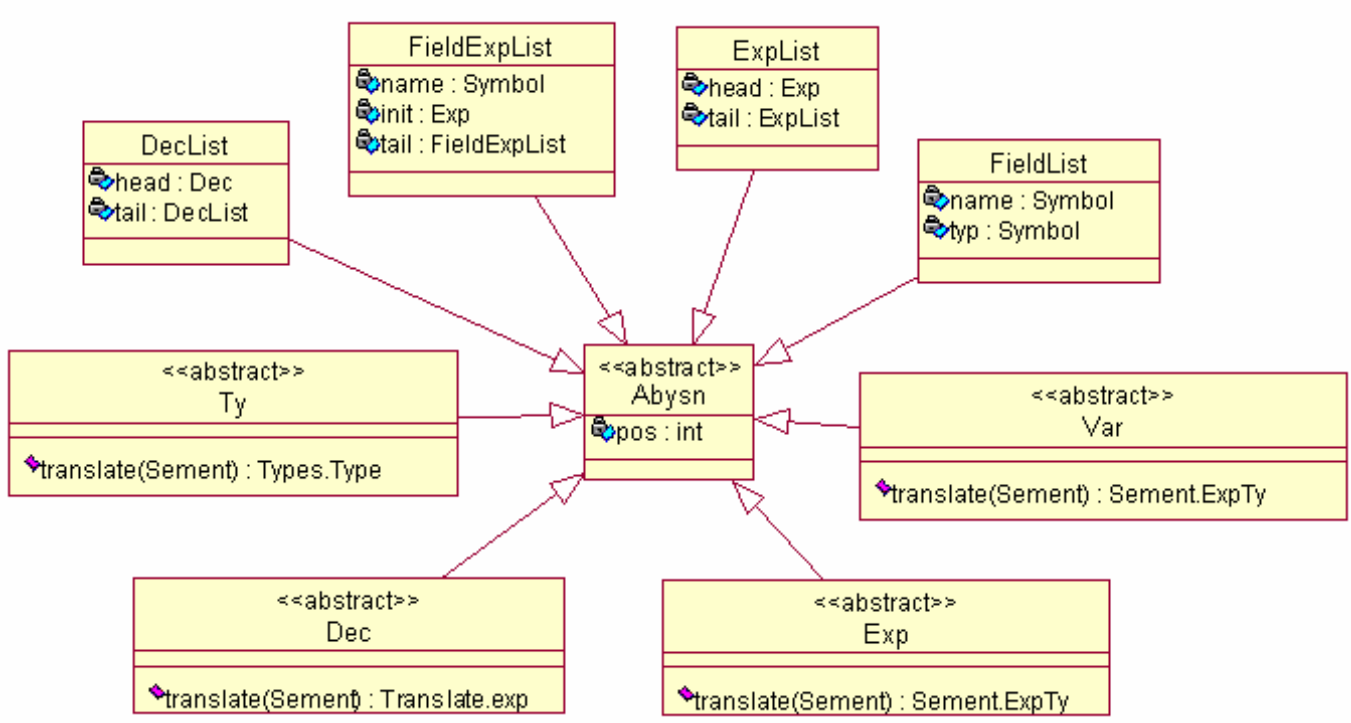
例如,一个常见的用法:

```
Object myobject;
sym = Symbol.symbol("My Symbol");
vEnv.put(sym, myobject);
```

BeginScope 与 EndScope 用于变量等的作用域控制

### 2.2 Absyn 包

Absyn 包中包含了抽象语法树的结点:

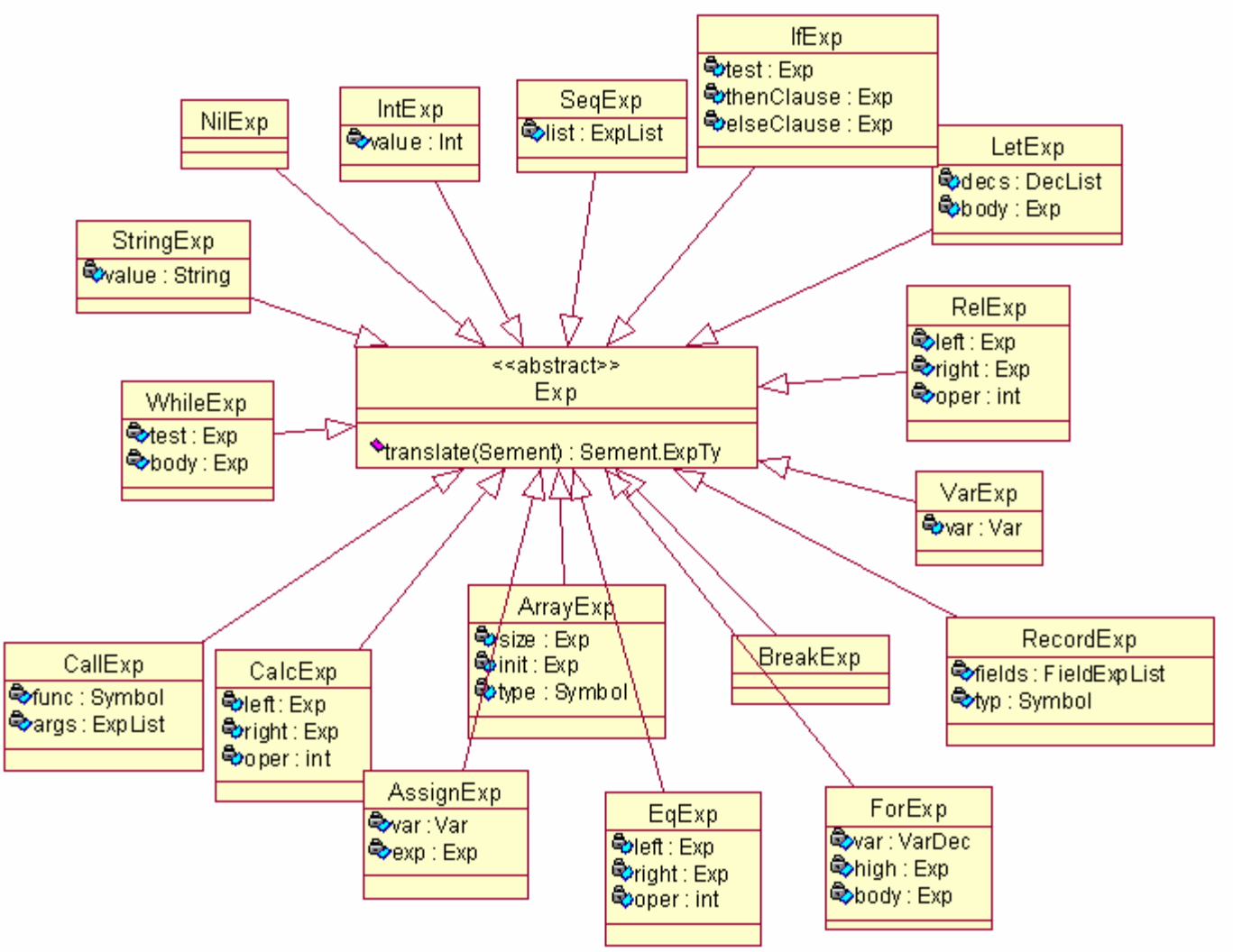


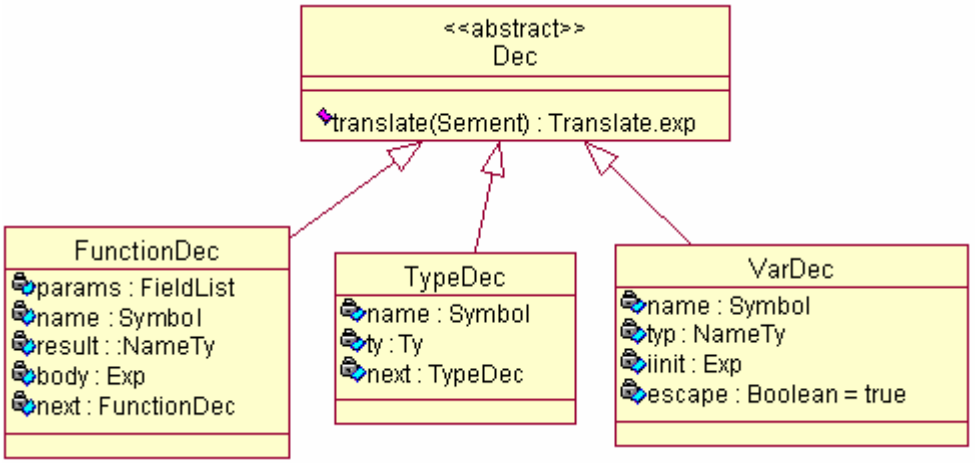

Dec: 声明 (函数、类型、变量声明)

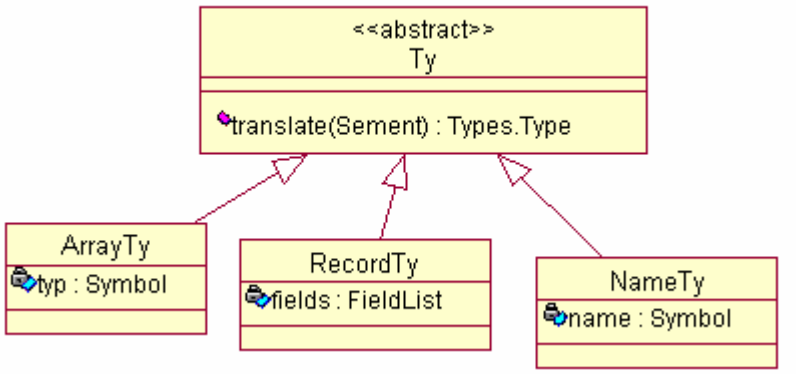
Exp: 表达式 (多种)
Var: 变量 (简单变量、域变量、下标变量)
Ty: 类型 (数组、记录、Name)
Lists:

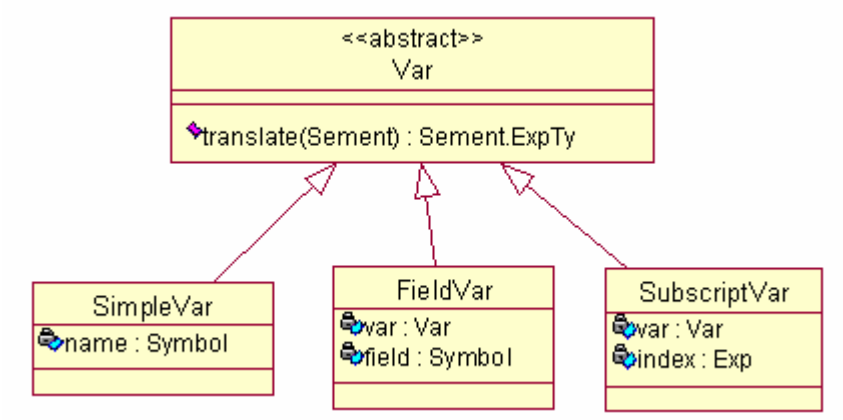
DecList: 声明列表,用于 Tiger 语言的声明块结构
FieldExpList: 用逗号分割的表达式列表,用于函数调用.例如:func(a,b,c)
ExpList: 用分号分割的表达式列表,例如 a=3;"abc";b=a+1.常用于 let...in...end 表达式中
FieldList: 域列表,用在域变量中.例如{name="abc", age=10}中的 list

Absyn.Print 输出抽象语法树






### 2.3 grm.cup 清单

```
package Parse;
import Absyn.*;

代码部分放在 {: :}中
action code {: static Symbol.Symbol sym(String s) {
            return Symbol.Symbol.symbol(s);
          }
       :};
```

```
parser code  {:
  public Exp parseResult;
  Lexer lexer;

  public void syntax_error(java_cup.runtime.Symbol current) {
   report_error("Syntax error (" + current.sym + ")", current);
  }

  ErrorMsg.ErrorMsg errorMsg;

  public void report_error(String message,
                java_cup.runtime.Symbol info) {
      errorMsg.error(info.left, message);
  }

  public Grm(Lexer l, ErrorMsg.ErrorMsg err) {
    this();
    errorMsg=err;
    lexer=l;
  }
:};

scan with {: return lexer.nextToken(); :};

terminal String ID, STR; 定义字符串类型的终结符
terminal Integer NUM; 定义整数类型的终结符
定义其它终结符
terminal COMMA, COLON, SEMICOLON, LPAREN, RPAREN,
    LBRACK, RBRACK, LBRACE, RBRACE, DOT, PLUS, MINUS,
    TIMES, DIVIDE, EQ, NEQ, LT, LE, GT, GE, AND, OR,
    ASSIGN, ARRAY, IF, THEN, ELSE, WHILE, FOR, TO, DO,
    LET, IN, END, OF, BREAK, NIL, FUNCTION, VAR, TYPE,STRING,INT,COMMENT,UMINUS;

非终结符,前面是抽象语法树结点的类型,在 Absyn 包中,后面是非终结符名称
non terminal Exp expr,program;
non terminal ExpList exprlist,exprseq;
non terminal Dec declaration;
non terminal DecList declarationlist;
non terminal VarDec variabledeclaration;
non terminal TypeDec typedeclaration,typedeclist;
non terminal FunctionDec functiondeclaration,functiondeclist;
non terminal Ty type;
non terminal Var lvalue;
non terminal FieldExpList fieldlist;
```

```
non terminal FieldList typefields;

定义优先级,从低到高
precedence right FUNCTION, TYPE;
precedence right OF;
precedence right DO, ELSE, THEN;
precedence nonassoc ASSIGN;
precedence left OR;
precedence left AND;
precedence nonassoc EQ , NEQ , LT , LE , GT , GE;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;
precedence left LPAREN;

文法起始符
start with program;

前半部分是文法规则,花括号内的部分是产生抽象语法树结点
program::= expr:e {:parser.parseResult=(Exp)e;:}
    ;

表达式文法
expr::=
    优先处理有二义性的负号,-a 处理成(0-a)
    MINUS:m expr:e
{:RESULT=new OpExp(mleft,new IntExp(e.pos,0),OpExp.MINUS,e);:} %prec UMINUS

    RESULT 为结果,xleft 为符号 x 左侧的位置,类似地 xright 为符号 x 右侧的位置
    |NUM:i {:RESULT=new IntExp(ileft, i.intValue());:}
    |STR:s {:RESULT=new StringExp(sleft,s);:}
    |NIL:n {:RESULT=new NilExp(nleft);:}

    变量(左值)
    |lvalue:e1{:RESULT=new VarExp(e1.pos,e1);:}

    算术运算
    |expr:e1 PLUS expr:e2{:RESULT=new OpExp(e1.pos,e1,OpExp.PLUS,e2);:}
    |expr:e1 TIMES expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.MUL,e2);:}
    |expr:e1 DIVIDE expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.DIV,e2);:}
    |expr:e1 MINUS expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.MINUS,e2);:}
    逻辑运算,处理为 if-then 表达式,真为 1,假为 0
    |expr:e1 OR expr:e2
        {:RESULT=new IfExp(e1.pos,e1,new IntExp(e1.pos,1),e2);:}
```

```
    |expr:e1 AND expr:e2
        {:RESULT=new IfExp(e1.pos,e1,e2,new IntExp(e1.pos,0));:}

    比较运算
    |expr:e1 EQ expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.EQ,e2);:}
    |expr:e1 LT expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.LT,e2);:}
    |expr:e1 LE expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.LE,e2);:}
    |expr:e1 GT expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.GT,e2);:}
    |expr:e1 GE expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.GE,e2);:}
    |expr:e1 NEQ expr:e2 {:RESULT=new OpExp(e1.pos,e1,OpExp.NE,e2);:}

    赋值运算
    |lvalue:l ASSIGN expr:e {:RESULT=new AssignExp(lleft,l,e);:}

    函数调用,参数可以为空.例如 func(a,3)或 func()
    |ID:i LPAREN exprlist:elist RPAREN
        {:RESULT=new CallExp(ileft,Symbol.Symbol.symbol(i),elist);:}
    |ID:i LPAREN RPAREN
        {:RESULT=new CallExp(ileft,Symbol.Symbol.symbol(i),null);:}

    用括号括起来的表达式列表,可以为空.例如()或(a=3;b=4)
    |LPAREN:l exprseq:eseq RPAREN
        {:RESULT=new SeqExp(lleft,eseq);:}
    |LPAREN:l RPAREN {:RESULT=new SeqExp(lleft,null);:}

    域表达式,例如 rectype {name="Nobody", age=1000},可以为空
    |ID:i LBRACE fieldlist:flist RBRACE
        {:RESULT=new RecordExp(ileft,Symbol.Symbol.symbol(i),flist);:}
    |ID:i LBRACE RBRACE
        {:RESULT=new RecordExp(ileft,Symbol.Symbol.symbol(i),null);:}

    数组表达式,例如 intArray [5] of 0
    |ID:i LBRACK expr:e1 RBRACK OF expr:e2
        {:RESULT=new ArrayExp(ileft,Symbol.Symbol.symbol(i),e1,e2);:}

    if,while,for 表达式
    如果某些成分可选,要分情况写,如 if 和 let,因为这里没有空匹配
    |IF:i expr:e1 THEN expr:e2 {:RESULT=new IfExp(ileft,e1,e2);:}
    |IF:i expr:e1 THEN expr:e2 ELSE expr:e3 {:RESULT=new IfExp(ileft,e1,e2,e3);:}
    |WHILE:w expr:e1 DO expr:e2 {:RESULT=new WhileExp(wleft,e1,e2);:}
    |FOR:f ID:i ASSIGN expr:e1 TO expr:e2 DO expr:e3
{:RESULT=new ForExp(fleft,new VarDec(ileft,Symbol.Symbol.symbol(i),
    new NameTy(ileft,Symbol.Symbol.symbol("int")),e1),e2,e3);:}
    |BREAK:b {:RESULT=new BreakExp(bleft);:}
```

```
    let 表达式
    |LET:l declarationlist:dlist IN END
{: RESULT = new LetExp(lleft, dlist, null); :}
    |LET:l declarationlist:dlist IN exprseq:eseq END
{: RESULT = new LetExp(lleft, dlist,new SeqExp(eseqleft,eseq)); :}
    ; 一个文法项目完后用分号分割

左值(变量)文法
lvalue::=
lvalue:l LBRACK expr:e RBRACK 带下标的变量,如 a.b[5]
{:RESULT=new SubscriptVar(lleft,l,e);:}
    |lvalue:l DOT ID:i 带域的变量,如 a.b
{:RESULT=new FieldVar(lleft,l,Symbol.Symbol.symbol(i));:}
    |ID:i {:RESULT=new SimpleVar(ileft,Symbol.Symbol.symbol(i));:} 简单变量
    |ID:i LBRACK expr:e RBRACK
{:RESULT=new 带下标的变量,如 a[5]
SubscriptVar(ileft,new SimpleVar(ileft,Symbol.Symbol.symbol(i)),e);:}
    ;
之所以需要两种带下标的变量,因为有可能出现规约冲突
如 a.b[5]可能会被规约为 a.(b[5])

用逗号分割的表达式列表,用于函数调用.例如:func(a,b,c)
exprlist::=
    expr:e {:RESULT=new ExpList(e,null);:}
    |expr:e COMMA exprlist:elist {:RESULT=new ExpList(e,elist);:}
    ;

用分号分割的表达式列表,例如 a=3;"abc";b=a+1.常用于 let...in...end 表达式中
exprseq::=
    expr:e {:RESULT=new ExpList(e,null); :} 递归边界
    |expr:e SEMICOLON exprseq:eseq {:RESULT=new ExpList(e,eseq); :}
    ;

域列表,用在域变量中.例如{name="abc", age=10}中的 list
fieldlist::=
    ID:i EQ expr:e
{:RESULT=new FieldExpList(ileft,Symbol.Symbol.symbol(i),e,null);:}
    |ID:i EQ expr:e COMMA fieldlist:flist
{:RESULT=new FieldExpList(ileft,Symbol.Symbol.symbol(i),e,flist);:}
    ;

声明列表,是由各种声明连接起来的块
之所以要这么写是由于 Tiger 的一个特殊语法:
```

若干个 type 构成一个块,若干个 function 构成一个块,它们中间不能重名.而下一个块的同名可以覆盖以前的块中的声明或函数

```
declarationlist::=
    declaration:dec {:RESULT=new DecList(dec,null);:}
    |declaration:dec declarationlist:declist
{:RESULT=new DecList(dec,declist);:}
    ;
```

一个声明可以是三种类型:类型声明列表,变量声明或函数声明列表

```
declaration::=
    typedeclist:tdec {:RESULT=tdec;:}
    |variabledeclaration:vdec {:RESULT=vdec;:}
    |functiondeclist:fdec {:RESULT=fdec;:}
    ;
```

类型声明列表,是由多个类型声明构成的块

```
typedeclist::=
    typedeclaration:t {:RESULT=t;:}
    |typedeclaration:t typedeclist:l
{:RESULT=new TypeDec(t.pos,t.name,t.ty,l);:}
    ;
```

类型声明,将已知类型定义成新类型

```
typedeclaration ::=
    TYPE:ty ID:i EQ type:t
{:RESULT=new TypeDec(tyleft,Symbol.Symbol.symbol(i),t,null); :}
    ;
```

类型:包括未知类型,域类型,空类型,数组类型

```
type::=
    ID:i {:RESULT=new NameTy(ileft,Symbol.Symbol.symbol(i));:}
    |LBRACE:l typefields:f RBRACE {:RESULT=new RecordTy(lleft,f);:}
    |LBRACE:l RBRACE {:RESULT=new RecordTy(lleft,null);:}
    |ARRAY:a OF ID:i {:RESULT=new ArrayTy(aleft,Symbol.Symbol.symbol(i));:}
    ;
```

typefileds, 用于域类型或函数的形式参数

```
typefields::=
    ID:i COLON ID:j
{:RESULT=new
FieldList(ileft,Symbol.Symbol.symbol(i),Symbol.Symbol.symbol(j),null);:}
    |ID:i COLON ID:j COMMA typefields:tf
{:RESULT=new
FieldList(ileft,Symbol.Symbol.symbol(i),Symbol.Symbol.symbol(j),tf);:}
```

```
    ;
```

变量声明
类型是可选的,可以显示说明也可以不显示
例如:var a:int=5 或 var a=5

```
variabledeclaration::=
    VAR:v ID:i ASSIGN expr:e
{:RESULT=new VarDec(vleft,Symbol.Symbol.symbol(i),null,e);:}
    |VAR:v ID:i COLON ID:j ASSIGN expr:e
{:RESULT=new VarDec(vleft,Symbol.Symbol.symbol(i),
new NameTy(ileft,Symbol.Symbol.symbol(j)),e);:}
    ;
```

函数声明列表,是由多个函数声明组成的块

```
functiondeclist::=
    functiondeclaration:f {:RESULT=f;:}
    |functiondeclaration:f functiondeclist :l
{:RESULT =new FunctionDec(f.pos, f.name, f.params ,f.result,f.body,l );:}
    ;
```

函数声明,分四种:
无参数无返回值
有参数无返回值
无参数有返回值
有参数有返回值

```
functiondeclaration::=
    FUNCTION:f ID:i LPAREN RPAREN EQ expr:e
{:RESULT=new FunctionDec(fleft,Symbol.Symbol.symbol(i),null,null,e,null);:}
    |FUNCTION:f ID:i LPAREN typefields:flist RPAREN EQ expr:e
{:RESULT=new  FunctionDec(fleft,Symbol.Symbol.symbol(i),  flist,  null,  e,
null); :}
    |FUNCTION:f ID:i LPAREN RPAREN COLON ID:j EQ expr:e
{:RESULT=new FunctionDec(fleft,Symbol.Symbol.symbol(i), null, new NameTy(jleft,
Symbol.Symbol.symbol(j)), e, null); :}
    |FUNCTION:f ID:i LPAREN typefields:flist RPAREN COLON ID:j EQ expr:e
{:RESULT=new FunctionDec(fleft,Symbol.Symbol.symbol(i),flist,new NameTy(jleft,
Symbol.Symbol.symbol(j)),e,null);:}
    ;
```

### 2.4 javacup 的使用

javacup 目录提示符下输入:`java java_cup.Main -expect 2 <grm.cup`

其中参数 –expect 2 表示至多 2 个冲突(规约－移进或规约－规约是允许的),在这种情况下,javacup 将使用移进代替规约,遇到规约－规约冲突时采用高优先级.

运行后将生成两个 java 文件:parser.java 和 sym.java. parser 为分析器,sym 为常数表

## 2.5 词法分析器与文法分析器的连接

Parse 包中封装了词法分析与语法分析，其中 Parse.Parse 类担任对外接口（门面类）

下面是 Parse.java 中的主要代码:

```
//Yylex 执行词法分析
//Grm 执行语法分析
Grm parser = new Grm(new Yylex(inp, errorMsg), errorMsg);
parser.parse();
//返回抽象语法树的根
absyn = parser.parseResult;
```

Grm.java 中拥有一个 Yylex 的引用，通过构造函数传入:

```
//Grm.java 构造函数
public Grm(Lexer l, ErrorMsg.ErrorMsg err) {
this();
errorMsg=err;
lexer=l;    //传入的词法分析器
}
```

当文法分析需要下一个符号时,调用 scan 函数，它调用词法分析器使它返回下一个符号:

```
public java_cup.runtime.Symbol scan() throws java.lang.Exception
{   return lexer.nextToken();    }
```

为了让词法和文法分析的符号保持一致，它们共用 java_cup.runtime.Symbol 作为符号系统. Yylex 的 tok 函数负责把 Yylex 的符号转换成公共符号

```
private java_cup.runtime.Symbol tok(int kind, Object value) {
    return new java_cup.runtime.Symbol
        (kind, yychar, yychar+yylength(), value);
}
```

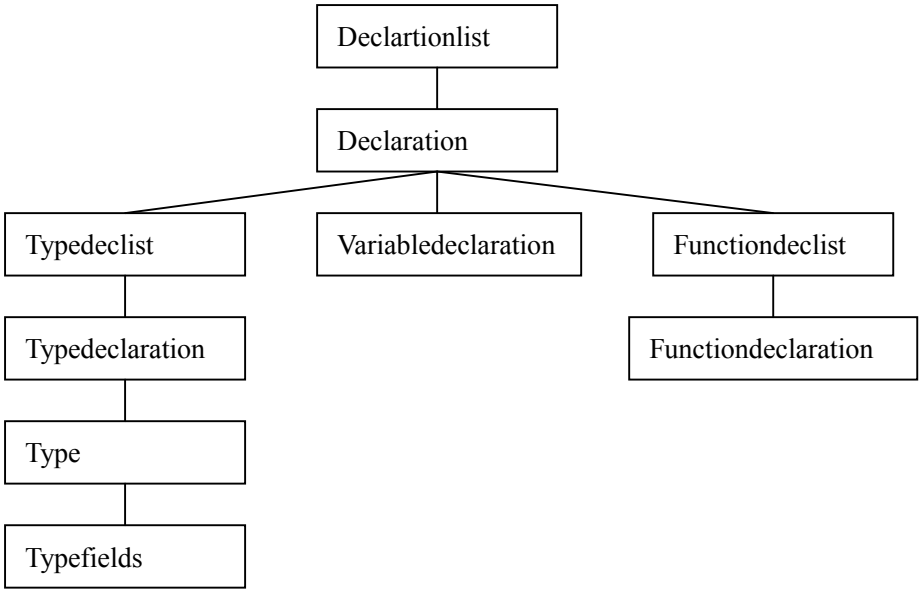
## 2.6 在本过程中主要遇到的问题

负号的优先处理

左值文法问题

声明列表块的问题

三种声明的关系比较复杂:


Declartionlist
Declaration
Typedeclist
Variabledeclaration
Functiondeclist
Typedeclaration
Functiondeclaration
Type
Typefields

# 第 4 部分 语义分析

## 4.1 入口 (Entry)

入口指明了一个非数据类型标识符 (变量和函数) 的种类.数据类型标识符由 Types 包描述

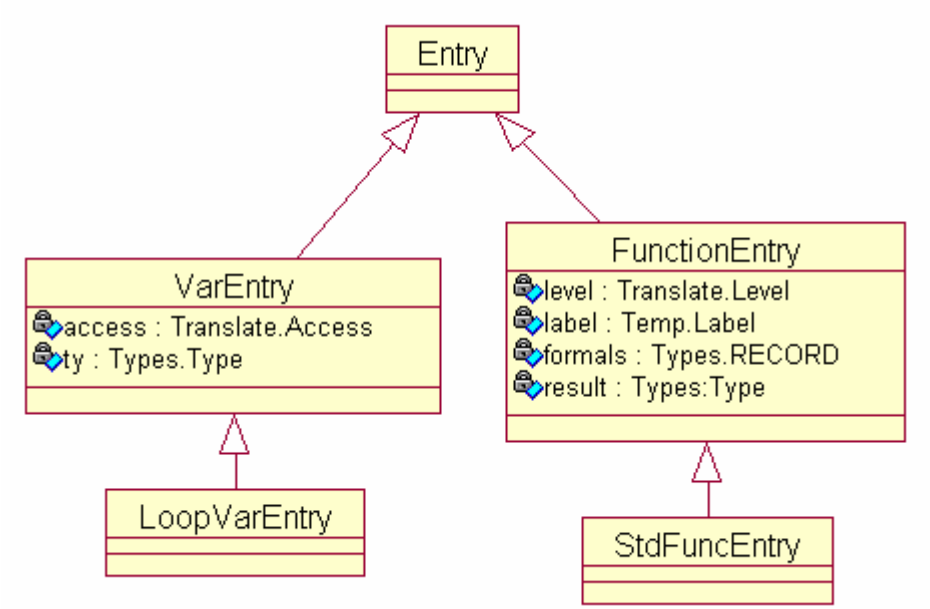
有下面 4 种入口,它们派生自 Semant.Entry:

VarEntry: 普通变量

StdFuncEntry: 库函数

LoopVarEntry:循环变量

FuncEntry: 普通函数




## 4.2 tEnv 和 vEnv

它们是两个符号表 (Symbol.Table), 关于符号表见 2.1

tEnv 记录了当前的类型名称,是符号－类型表, 注意其中会有类型绑定机制: 如果其中的类型是用 Types.NAME 表示的,则实际类型存储在 NAME.binding 里面 (见 4.3)

vEnv 记录了当前的变量、函数等信息.是符号－入口表.在其中的每个项目可能是 4.1 中的 4 种入口

在初始化 tEnv 时 (函数 initTEnv),要添加两种基本类型 int 和 string

在初始化 vEnv 时 (函数 initVEnv),要先把库函数 (如 print, flush 等) 作为 StdFuncEntry 添加进去

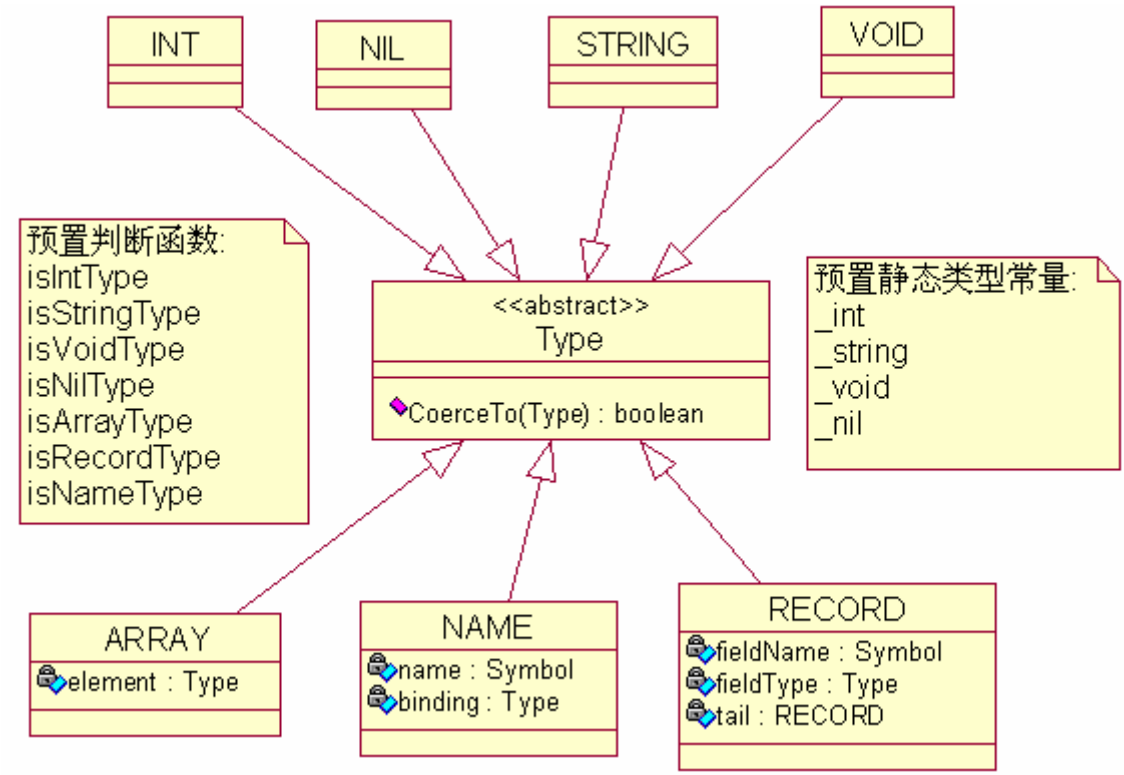
```
class Env {
    Table vEnv = null;      //nEnv
    Table tEnv = null;      //tEnv
    ErrorMsg errorMsg = null;
    Level root = null;      //传入 main 函数的层，添加库函数时使用

    Env(ErrorMsg errorMsg, Level root) {
```

```
        this.errorMsg = errorMsg;
        this.root = root;
        initTEnv();
        initVEnv();
    }

    //初始化 tEnv
    void initTEnv() {
        tEnv = new Table();
        //基本类型 int
        tEnv.put(Symbol.symbol("int"), Type._int);
        //基本类型 string
        tEnv.put(Symbol.symbol("string"), Type._string);
    }

    //初始化 vEnv
    public void initVEnv() {
        vEnv = new Table();

        Symbol sym = null;      //函数名称
        RECORD formals = null;  //参数表
        Type result = null;     //返回值类型
        Level level = null;     //层

        //添加函数: print(s: string)
        sym = Symbol.symbol("print");
        formals = new RECORD(Symbol.symbol("s"), Type._string, null);
        result = Type._void;
        level = new Level(root, sym, new BoolList(true, null));
        vEnv.put(sym, new StdFuncEntry(level, new Label(sym), formals, result));

        //其它函数类似
        //...
    }
}
```

### 4.3 Type 包

Type 包中封装了类型信息,类名用大写表示以示区分

注意和 Absyn 包中的类型信息区别,后者是根据程序字面翻译而来的,没有经过语义检查.




Type 包中类的 coerceTo 函数描述了关于类型的强制转换部分的信息:
除了 nil 类型可以赋值给 record 类型外, 其它只能转换到本身.
且不能将 nil 赋值给 nil

这里比较特殊的是 NAME 类型:它用来表示一种未知的类型.而真正的类型需要用 binding 方法绑定到 NAME 类型中.需要得到实际类型,应用 NAME.actual 方法.此时,其它类型(如 int 等)为了兼容,也提供了默认的 actual 方法 (返回本身).

之所以采用这种机制是为了防止处理类型的循环定义,这在 Tiger 语言中是不允许的,例如:

```
type a=c
type b=a
type c=d
type d=a
```

注意这与递归定义不同,递归定义是允许的,例如:

```
type tree ={key: int, children: treelist}
type treelist = {hd: tree, tl: treelist}
```

NAME 类型中用 isLoop 来检测是否出现循环定义
NAME 类型的代码如下:

```
public class NAME extends Type {
    public Symbol.Symbol name; //类型名称
    private Type binding;   //绑定的实际类型
    public NAME(Symbol.Symbol n) {name = n;}

    //检测循环定义
```

```
    public boolean isLoop() {
        Type b = binding;
        boolean any;
        binding = null; //先把 binding 去掉并临时保存
        if (b == null) any = true; //如果下一次出现了以上状态则出现循环引用
        else if (b instanceof NAME) //如果还是 NAME 类型,递归继续
            any = ((NAME) b).isLoop();
        else any = false; //否则递归结束,没有循环定义
        binding = b; //测试完毕,恢复
        return any;
    }

    public Type actual() { //返回实际类型
        return binding.actual(); //其它类型的 actual 函数返回真正的类型
    }

    public boolean coerceTo(Type t) {
        return this.actual().coerceTo(t);
    }

    //绑定实际类型
    public void bind(Type t) {binding = t;}
}
```

某个普通类型 (如 int) 的代码则简单许多:

```
public class INT extends Type {
    public INT() {}
    public boolean coerceTo(Type t) {
        return (t.actual() instanceof INT);
    }
}
```

### 4.4 语义分析的一般步骤

语义分析的一般步骤为:

当检查某个语法结点时, 需要递归地检查结点的 (包括以后的中间代码翻译) 每个子语法成分, 确认所有子语法成分的正确且翻译完毕后, 调用 Translate 对整个表达式进行翻译.

对表达式的检查称为语义分析, 但从上面的步骤可以看出, 它和中间代码的翻译是联系在一起的.

例如下面是检查和翻译 while 表达式的一段代码

```
    public ExpTy transExp(WhileExp e) {
        ExpTy test_ty = e.test.translate(this); //检查并翻译测试部分
        //检查测试部分
```

```
        if (!Type.isIntType(test_ty.ty))
            error(e.test.pos, "ERROR : Bad test condition in while expression!");
        Temp.Label done = new Temp.Label();
        translator.newLoop();
        //检查并翻译循环体部分
        ExpTy body_ty = e.body.translate(this);
        translator.exitLoop();
        if (!Type.isVoidType(body_ty.ty))
            error(e.body.pos, "ERROR : While body should not return a value!");
        //调用 Translate 部分, 翻译整个 while 表达式
        return  new  ExpTy(translator.transWhileExp(test_ty.exp,  body_ty.exp,
    done), Type._void);
    }
```

下面列出类型检查的输入和输出:

| 输入类型 | 输出类型 |
|---|---|
| Absyn.Exp, Absyn.Var | Semant.ExpTy |
| Absyn.Ty | Types.Type |
| Absyn.FieldList | Types.RECORD |
| Absyn.Dec | Translate.Exp |
| Absyn.FunctionDec (翻译返回值类型) | Types.Type |

Absyn 包中的表达式被重新翻译成带类型的表达式 ExpTy;Absyn 中的类型被翻译成类型类 Type 的子类(在 Type 包中);声明将被翻译成 Translate.Exp

Semant.ExpTy 的结构如下:

```
import Translate.Exp;
import Types.Type;
public class ExpTy {
    Exp exp; //语义表达式
    Type ty; //语义类型
    ExpTy(Exp exp, Type ty) {
        this.exp = exp;
        this.ty = ty;
    }
}
```

可以看出它把 Translate.Exp 和 Types.Type 封装起来了

### 4.5 具体表达式/声明/变量的检查

该任务在 Semant.Semant 中完成, 这里着重讨论检查部分

下面是表达式检查,返回带类型的表达式类 ExpTy

*a 下面的表达式无需检查和其它特别翻译*

整数表达式 IntExp

字符串表达式 StringExp

nil 表达式 NilExp
变量表达式 VarExp

*b 检查算术表达式 CalcExp:*
要求左右均为 int 类型

*c 测试相等不等表达式 EqExp:*
左右中任一个不能为 void 类型
左右不能全为 nil
可以一个为 nil 一个为 record 类型
其它情况下必须左右类型完全一致

*d 关系运算符 RelExp:*
左右两边必须全为 int 或 string

*e 赋值运算符 AssignExp:t1:=t2*
如果 t1 是简单变量并且在 vEnv 中查得它是 LoopVarEntry,则报错: 不能给循环变量赋值
如果 t2 是 void 类型则出错
如果 t2 不能强制转换成 t1 则出错
注意:赋值表达式无返回值

*f 函数调用运算符 CallExp*
如果在 vEnv 里查不到函数名或者它不是 FuncEntry (包括 StdFuncEntry) 则报错:函数未定义
然后逐个检查形参和实参是否匹配(用 AssignExp 的方法检查),遇到不匹配则报错.
在遍历形参链表的时候,可能遇到链表空或有剩余的情况,此时分别报告实参过多或不足的错误

*g 记录表达式 RecordExp*
先在 tEnv 中查找类型是否存在,若否或非记录类型报告未知记录类型错误
然后逐个检查记录表达式和记录类型域的名字是否相同
然后逐个检查记录表达式和记录类型域的类型是否匹配 (用 AssignExp 方法检查)
在遍历记录类型链表的时候,可能遇到链表空或有剩余的情况,此时分别报告域过多或不足的错误

*h 数组表达式 ArrayExp*
先在 tEnv 中查找类型是否存在,若否或非数组类型报告未知记录类型错误
再检查数组范围是否为整数,若否报错
再用 AssignExp 的方法检查 tEnv 中的数组类型和实际类型是否匹配,若否报告类型匹配错误

*i if 表达式 IfExp*
如果测试条件的表达式不返回整数,报告测试条件错误(Tiger 中非 0 为真,0 为假)

如果缺少 else 子句,且 then 子句有返回值,报错

如果不缺少 else 子句,检查 then 和 else 的返回值是否匹配(采用 AssignExp 的方法,只是都返回 nil 被认为是合法的)

*j while 表达式 WhileExp*

如果测试条件不是整数,报告测试条件错误

如果循环体有返回值,则报错(while 循环体不能有返回值)

注意这里不需要使 Begin/EndScope

注意进/出循环使要调用 newLoop 和 exitLoop (详见 break 表达式)

*k for 表达式 ForExp*

如果初始值和终止值不是整数类型,则报错

用 BeginScope 进入 vEnv 新的符号表 (循环内部用)

为帧分配循环变量的 Access

把循环变量添加到 vEnv 的 LoopVarEntry 项目中

用 EndScope 退出 vEnv 的符号表

注意进/出循环使要调用 newLoop 和 exitLoop (详见 break 表达式)

*l break 表达式 BreakExp*

设一个堆栈,进入循环推入一个 Label, 退出循环弹出一个 Label,如果堆栈为空时出现 break 语句则报告错误,关于栈的操作由以下函数完成:

translate.loopExit (堆栈)

void translate.newLoop (入栈)

void translate.exitLoop (出栈)

boolean translate.isInLoop (测试是否在堆栈中)

newLoop 和 exitLoop 在翻译 for 和 while 语句时成对使用

这样做的优点是在以后翻译成 IR 树时可以利用这个堆栈

另一种更简单的做法是:采用一个计数器,初始化为 0.进入循环加 1,退出循环减 1.如果在计数器为 0 时出现 break 语句则报告错误

*m let 表达式 LetExp*

无需错误检查,只需按如下步骤进行:

vEnv 和 tEnv 分别用 BeginScope 进入新的符号表

翻译定义部分 (let…in 之间), 用 translate.combine2stm 将 IR 树结点连接

翻译体部分 (in…end 之间)

vEnv 和 tEnv 用 EndScope 返回到原符号表

如果体部分为空或者没有返回值,用 translate.combine2stm 将定义和体部分连接

如果体部分不空或有返回值,用 translate.combine2exp 将定义和体部分连接,把体部分的返回值和类型作为最终的返回值和类型

*n 顺序表达式 SeqExp*

无需错误检查,因为它是若干个已经检查过的表达式的链

分别将每个子表达式进行翻译,用 translator.combine2stm 函数将它们的 IR 树结点连

接起来

在翻译最后一个表达式时:

如果它的类型是 VOID,则仍用 combine2stm 将它们的 IR 树结点连接起来,并返回 VOID 作为返回类型

否则,用 translate.combine2exp 将它们的 IR 树结点连接并返回最后一个表达式的值和类型作为返回值和返回类型

o *简单变量 SimpleVar*

先检查 vEnv, 若没找到变量名或类型不是 VarEntry 则报告变量未定义

p *下标变量 SubscriptVar*

如果它除去下标部分后的类型不是数组类型,则报错

若下标部分不是 int 类型报错

q *域变量 FieldVar*

若除去域部分后不是记录类型,则报错

然后逐个查找记录的域,如果没有一个匹配当前域变量的域,则报错

下面是类型的检查返回类型 Type 类,可能是 ARRAY, RECORD, NAME 等

r *未知类型 NameTy*

检查 tEnv,若没有发现类型,则报告未知类型错误

s *数组类型 ArrayTy*

检查 tEnv,若没有发现类型,则报告未知类型错误,否则返回转换后的 ARRAY 类型

t 记录类型 RecordTy

检查该记录类型每个域的类型在 tEnv 中是否存在,若否,则报告未知类型错误

若全部正确,则最后返回转换后的 RECORD 类型

下面是声明的检查,在这个阶段可以不需要返回值

u 变量声明 VarDec

如果有显式的类型声明,按 AssignExp 的方法检查是否类型匹配

若无显式的类型声明且初始值为 nil,则报错,因为只有记录类型可以用 nil 初始化

若无初始值,报错,因为 Tiger 语言所有的变量声明必须有初始化

如果没有以上错误,则为变量在帧上分配空间把变量作为 VarEntry 添加到 vEnv 中

变量声明不采用块机制,而是在任何地方都直接覆盖

v 类型声明 TypeDec

1 Tiger 语言的类型声明采用块机制(见第二部分),所以要先在一个声明块中检查是否有重复的声明(而在不同的块中可以有重复的声明的,新的将冲掉旧的),若有,报告重定义错误(具体实现时可采用 HashSet 类)

2 接下来翻译出实际的 Type 类型 (Typedec.ty.translate 方法) ,再从 tEnv 中查找出

NAME 类型,将实际的 Type 类型绑定到 NAME 类型

3 检测循环定义问题 (见 4.3)

4 如果以上均正确,添加类型到 tEnv 中

w 函数声明 FunctionDec

函数声明同样采用块机制. 对于函数声明块中的每个声明,做以下检查 1~5:

1 同类型声明类似,先检查声明块中是否有重复的声明

2 还要检查是否与标准函数冲突,可以通过扫描 vEnv 中的 StdFuncEntry 实现,若有报错

3 然后检查参数列表,与记录类型 RecordTy 的检查完全相同,得到 RECORD 类型的形参列表

4 接着检查函数返回值,如果没有返回值则设置成 void

5 无误后,为函数创建新层,将函数作为 FuncEntry 添加到 vEnv 中

然后再重新扫描一遍声明,如果某函数不是标准函数,则做以下操作 1~5:

1 用 BeginScope 进入 vEnv 的一张子表(函数内部用),并转移到新层

2 将函数参数存入子表

3 翻译函数体, 并用 ProcEntryExit 给函数体加上关于函数调用的指令

4 检查函数体的返回类型是否和声明部分匹配,若否则报错

5 用 EndScope 退出子表,转移到当前层

### 4.6 本部分中与中间代码生成有关的问题

一般而言,在语义检查完毕后,只要简单调用 translator 包中的相应函数即可.例如翻译整数:

```
    public ExpTy transExp(IntExp e) {
        return new ExpTy(translator.transIntExp(e.value), Type._int);
    }
```

下面给出 Semant 与 Translator 的调用关系表

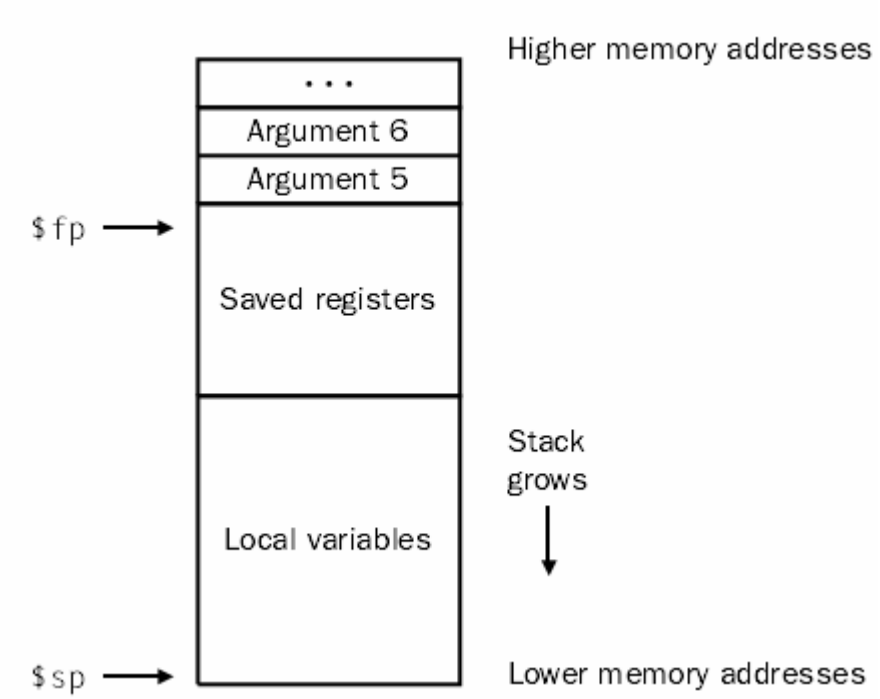
| Semant 的翻译对象 | Translator 的处理函数 |
|---|---|
| IntExp | transIntExp |
| StringExp | transStringExp |
| NilExp | transNilExp |
| VarExp | transVarExp |
| CalcExp | transCalcExp |
| EqExp | 字符串比较:transStringRelExp<br>其它情况: transOtherRelExp |
| RelExp | 字符串比较:transStringRelExp<br>其它情况: transOtherRelExp |
| AssignExp | transAssignExp |
| CallExp | 若库函数:transExtCallExp<br>普通函数:transCallExp |
| SeqExp | 无返回值:combine2Stm<br>有返回值:combine2Exp |

| | |
|---|---|
| RecordExp | transRecordExp |
| ArrayExp | transArrayExp |
| IfExp | transIfExp |
| WhileExp | transWhileExp |
| ForExp | transForExp |
| BreakExp | transBreakExp |
| LetExp | 无返回值:combine2Stm<br>有返回值:combine2Exp |
| SimpleVar | transSimpleVar |
| SubscriptVar | transSubscriptVar |
| FieldVar | transFieldVar |
| 类型定义:NameTy, ArrayTy, RecordTy | 无动作,仅修改 tEnv |
| VarDec | transSimpleVar, transAssignExp<br>变量需要识别和赋初始值 |
| 类型声明:TypeDec | 无动作 |
| FunctionDec (检查返回值) | 无动作 |
| FunctionDec (检查声明) | procEntryExit |

有时无动作调用 translator.transNoOp, 生成一条空指令

# 第 5 部分 活动记录

## 5.1 Frame 的结构和函数调用步骤




函数调用分三步进行,调用者称为 Caller,被调用者称为 Callee

第一步,在 Caller 调用之前:

1 传递参数,前三个参数放入寄存器$a0-$a3,后面的参数按顺序推入堆栈

2 保存 Caller-Saved 寄存器 ($t0-$t9) 和参数寄存器 ($a0-$a3), Callee 可以直接使用 Caller-Saved 寄存器,所以 Caller 要把它们先保存

3 执行 jal 指令,它跳转到 Callee 的第一条指令并把返回地址存在$ra 中

第二步,在 Callee 获得控制权时

1 分配帧空间:将$sp 减去帧空间 (如 32byes)

2 保存 Callee-Saved 寄存器 ($s0-$s7), 帧寄存器 $fp 和返回地址寄存器 $ra, 因为 Caller 期望这些寄存器在返回后是不变的. $fp 是一定要保存的, 如果 Callee 还要发起一个函数调用,那么$ra 也要保存, 如果函数中用到$s0-$s7 的话, 它们也要保存

3 令$fp=$sp-帧空间+4 bytes (如 $fp=$sp-28 bytes)

第三步,Callee 返回到 Caller 之前

1 如果函数有返回值,把它放到$v0 中

2 恢复所有 Callee-Saved 寄存器

3 将$sp 加上相应的帧空间 (如 32bytes)

4 跳转回返回地址(存在 $ra 中)

## 5.2 抽象的 Frame

在 Frame.java 中定义了抽象的 Frame (帧) 结构,它存放了一个函数的参数、返回地址、

寄存器、帧指针、栈指针、返回值等重要信息

```
public abstract class Frame implements TempMap {
    //建立新帧(名称、参数逃逸信息)
    public abstract Frame newFrame(Label name, BoolList formals);
    public Label name;  //名称
    public AccessList formals = null; //本地变量(局部量、参数)列表
    public abstract Access allocLocal(boolean escape); //分配新本地变量(是否逃逸)
    public abstract Expr externalCall(String func, ExpList args); //外部函数
    public abstract Temp FP(); //帧指针
    public abstract Temp SP(); //栈指针
    public abstract Temp RA(); //返回地址
    public abstract Temp RV(); //返回值
    public abstract TempList registers(); //寄存器列表
    public abstract Stm procEntryExit1(Stm body); //添加额外函数调用指令,见 5.4
    public abstract InstrList procEntryExit2(InstrList body); //同上
    public abstract InstrList procEntryExit3(InstrList body); //同上
    public abstract String string(Label label, String value);
    public abstract InstrList codegen(Stm s); //生成 MIPS 指令用
    public abstract int wordSize(); //返回一个字长(定义为 4bytes)
}
```

其中,Frame.Access 用于描述那些存放在帧中或是寄存器中的形式参数和局部变量,它也是抽象数据类型, 要得到 Access 所描述的变量 (IR 树结点), 可以使用下面两个函数中的一个:

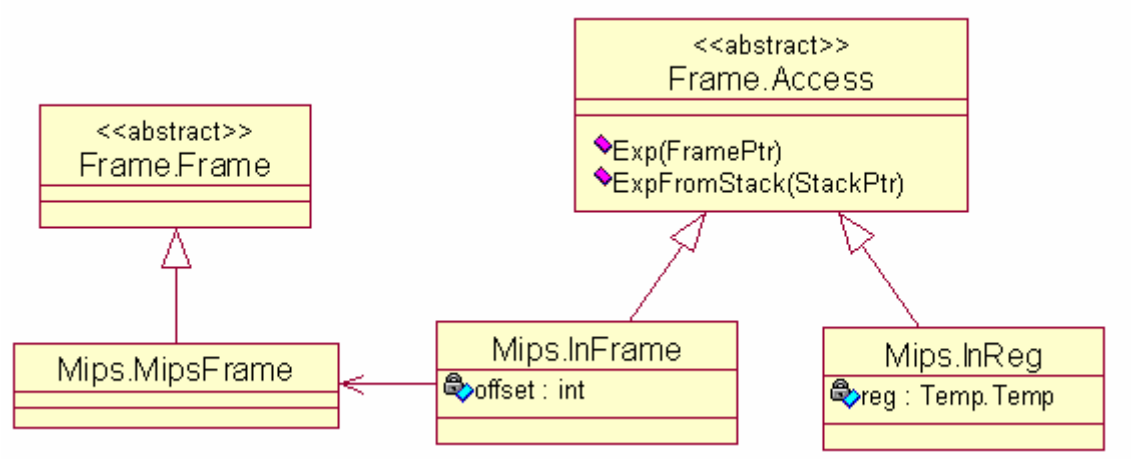
```
public abstract class Access {
    public abstract Expr exp(Expr framePtr); //以 fp 为起始地址返回变量
    public abstract Expr expFromStack(Expr stackPtr); //以 sp 为起始地址返回变量
}
```

AccessList 用于把 Access 连成链表.

所谓逃逸 (Escape) 是指某个 Access 是放在帧 (内存) 中还是寄存器中.若为 true 放在帧中,false 则放在寄存器中,AccessList 的逃逸信息用 Util.BoolList 来描述

### 5.3 用于 Mips 机的 Access - InFrame 和 InReg

Access 在 Mips 中有两种类型: Mips.Inframe 描述了存放在帧中的变量, Mips.InReg 描述了存放在寄存器中的变量 – 尽管它们存放的地址不一样, 但都必须在 Frame 中有引用 (作为基类 Access).




```
public class InFrame extends Access {
    private MipsFrame frame;    //帧
    private int offset;         //偏移量
    public InFrame(MipsFrame frame, int offset) {
        this.frame = frame;
        this.offset = offset;
    }
    //以 fp 为起始地址返回变量的 IR 树结点
    public Expr exp(Expr framePtr) {
        return new MEM(new BINOP(BINOP.PLUS, framePtr, new CONST(offset)));
    }

    //以 sp 为起始地址返回变量的 IR 树结点
    //fp=sp+帧帧空间+4bytes
    public Expr expFromStack(Expr stackPtr) {
        return new MEM(new BINOP(BINOP.PLUS, stackPtr, new CONST(offset
                - frame.allocDown - frame.wordSize())));
    }
}

public class InReg extends Access {
    private Temp reg;
    public InReg() {reg = new Temp();}
    //寄存器中无所谓 fp 和 sp 了, 不管以何种方式,总是返回那个寄存器
    public Expr exp(Expr framePtr) {return new TEMP(reg);}
    public Expr expFromStack(Expr stackPtr) {return new TEMP(reg);}
}
```

### 5.4 用于 Mips 机的 Frame – Mips.MisFrame

Mips.MipsFrame 具体描述了用于 Mips 机的帧信息,派生自 Frame.Frame,有如下任务:

*a 初始化寄存器:*

$fp – 帧指针
$sp – 栈指针
$ra – 返回地址
$v0 – 返回值
$zero – 保留存 0
$ax – 4 个参数寄存器
$tx – 10 个 Caller-Save 寄存器
$ax – 8 个 Callee-Save 寄存器
在 Mips.MisFrame 中的静态初始化部分中描述

Caller-Save 和 Callee-Save 的区别:它们是在硬件手册中人为设定的规范,而不是通过硬件实现的.MIPS 计算机中,16～23 号寄存器被保留用作 Callee-Save 寄存器.Callee-Saved 寄存器常用来保存生存期长的变量(如全局量);Caller-Saved 寄存器常用来保存生存期短的变量(如即时计算结果).例如,如果函数 f 知道某个变量 x 在函数调用以后将不再需要时,它可以把 x 放入 caller-save 寄存器,并且在调用函数 g 时不必对变量 x 进行保存.相反如果函数 f 中有一个局部变量 i,该变量在几次函数调用中都将被用到,那么可以把 i 放入 callee-save 寄存器 ri 中,并且在 f 开始时将 ri 保存起来,在 f 返回时将 ri 取回.这样为局部变量和临时变量合理选择 caller-save 或 callee-save 将减少存储和取回操作的执行次数.

*b 新建帧*

为帧分配 Access, 并生成相应的汇编指令

```
    public Frame.Frame newFrame(Label name, BoolList formals) {
        MipsFrame ret = new MipsFrame();
        ret.name = name; //名称
        TempList argReg = argRegs; //参数表
        for (BoolList f = formals; f != null; f = f.tail, argReg = argReg.tail)
{
            Access a = ret.allocLocal(f.head); //为每个参数分配 Access
            //注意区分 Frame.formals 和本地的 formals, 前者是 AccessList 类型
            ret.formals = new Frame.AccessList(a, ret.formals);
            if (argReg != null)//产生保存参数的汇编指令:把参数放入 frame 的 Access 中
                ret.saveArgs.add(new MOVE(a.exp(new TEMP(fp)),
                new TEMP(argReg.head)));
        }
        return ret;
    }
```

*c 分配 Access*

```
    public Access allocLocal(boolean escape) {
        if (escape) { //逃逸,在帧中分配
            Access ret = new InFrame(this, allocDown);
            //注意! allocDown 使用负数表示
            allocDown -= wordsize; //增加分配的帧空间
            return ret;
```

```
        } else //否则分配给寄存器
            return new InReg();
    }
```

*d procEntryExit1*

Callee 经过 procEntryExit1 处理后增加了如下指令

保存原 fp → 计算新 fp → 保存 ra → 保存 Callee-save 寄存器 →

保存参数 → (函数体原指令) → 恢复 Callee-save 寄存器 → 恢复返回地址 →

恢复 fp

```
    public Stm procEntryExit1(Stm body) {
        //1 在 body 前面加上保存参数的汇编指令
        for (int i = 0; i < saveArgs.size(); ++i)
            body = new SEQ((MOVE) saveArgs.get(i), body);
        //2 在 body 前面加上保存 Callee-save 寄存器的指令
        Access fpAcc = allocLocal(true);
        Access raAcc = allocLocal(true);
        Access[] calleeAcc = new Access[numOfCalleeSaves];
        TempList calleeTemp = calleeSaves;
        for (int i = 0;i < numOfCalleeSaves;++i,calleeTemp = calleeTemp.tail) {
            calleeAcc[i] = allocLocal(true);
            body = new SEQ(new MOVE(calleeAcc[i].exp(new TEMP(fp)), new TEMP(
                    calleeTemp.head)), body);
        }
        //3 在 body 前面加上保存返回地址 $ra 的指令
        body = new SEQ(new MOVE(raAcc.exp(new TEMP(fp)), new TEMP(ra)), body);
        //4 令$fp=$sp-帧空间+4 bytes
        body = new SEQ(new MOVE(new TEMP(fp), new BINOP(BINOP.PLUS,
                new TEMP(sp), new CONST(-allocDown - wordsize))), body);
        //5 在 body 前保存 fp
        body = new SEQ(
                new MOVE(fpAcc.expFromStack(new TEMP(sp)), new TEMP(fp)), body);
        //6 在 body 后恢复 callee
        calleeTemp = calleeSaves;
        for (int i = 0; i < numOfCalleeSaves; ++i, calleeTemp = calleeTemp.tail)
            body = new SEQ(body, new MOVE(new TEMP(calleeTemp.head),
                    calleeAcc[i].exp(new TEMP(fp))));
        //body 后恢复返回地址
        body = new SEQ(body, new MOVE(new TEMP(ra), raAcc.exp(new TEMP(fp))));
        //body 后恢复 fp
        body = new SEQ(body, new MOVE(new TEMP(fp), fpAcc.expFromStack(new
            TEMP(sp))));
        return body;
    }
```

此外, Translate.java 中的 procEntryExit 为函数体加上返回值的信息, 并把 IR 树结点加入段中

```
    //Translate.java
    public void procEntryExit(Level level, Exp body, boolean returnValue) {
        Stm b = null;
        if (returnValue)
            //有返回值
            b = new MOVE(new TEMP(level.frame.RV()), body.unEx());
        else
            //无返回值,按 Nx 处理
            b = body.unNx();
        b = level.frame.procEntryExit1(b);
        addFrag(new ProcFrag(b, level.frame)); //加入函数段
    }
```

*e ProcEntryExit2*

函数经 procEntryExit2 处理后保持不变(增加一条空指令)

```
    public InstrList procEntryExit2(InstrList body) {
        return append(body, new InstrList(new OPER("", null, new TempList(zero,
                new TempList(sp, new TempList(ra, calleeSaves)))), null));
    }
```

*f ProcEntryExit 3*

Callee 经过 procEntryExit3 处理后:

设置函数体标号→分配帧空间→(函数体原指令)→

将$sp 加上相应的帧空间→跳转到返回地址

```
    public InstrList procEntryExit3(InstrList body) {
        //分配帧空间:将$sp 减去帧空间 (如 32byes)
        body = new InstrList(new OPER("subu $sp, $sp, " + (-allocDown),
                new TempList(sp, null), new TempList(sp, null)), body);
        //设置函数体标号
        body = new InstrList(new OPER(name.toString() + ":", null, null), body);
        //跳转到返回地址
        InstrList epilogue = new InstrList(new OPER("jr $ra", null,
                new TempList(ra, null)), null);
        //将$sp 加上相应的帧空间 (如 32bytes)
        epilogue = new InstrList(new OPER("addu $sp, $sp, " + (-allocDown),
                new TempList(sp, null), new TempList(sp, null)), epilogue);
        body = append(body, epilogue);
        return body;
    }
```

*g 产生字符串的数据段汇编代码*

```
    public String string(Label label, String value) {
```

```
        String ret = label.toString() + ":\n";
        ret = ret + ".word " + value.length() + "\n";
        ret = ret + ".asciiz \"" + value + "\"";
        return ret;
    }
```

关于这些代码的格式, 参见 8.6

### 5.5 层 (Level)

层是用来描述静态连接用的数据结构.它被封装在 Translate.Level 中:
层有三个重要属性:

```
    public Level parent;    //直接上级层
    Frame.Frame framel;     //帧
    AccessList formals;     //参数(第一个为静态连接,后面为其它参数)
```

层的主要方法为 staticLink,它返回静态链接

```
    public Access staticLink() {return formals.head;}
```

注意 parent 和 staticLink()的区别, 前者是 Level 对象, 后者是地址
函数调用是总是把第一个参数作为返回地址, 一般放在$a0 寄存器中

在层的构造函数中还负责为层创建新帧
并把帧中的 Access 拷贝给层

```
    public Level(Level parent, Symbol name, BoolList fmls) {
        this.parent = parent;
        this.frame = parent.frame
                .newFrame(new Label(name), new BoolList(true, fmls));
        for (Frame.AccessList f = frame.formals; f != null; f = f.next)
            this.formals = new AccessList(new Access(this, f.head),
            this.formals);
    }
```

在翻译变量和函数调用时需要用到静态链接,见第 6 部分
层总是和 Frame 绑定在一起的.层描述了函数的静态信息,而帧描述函数的动态信息.

### 5.6 ProcEntryExit 总结

四个 ProcEntryExit 的执行顺序为:
ProcEntryExit (translate.java) 中调用 ProcEntryExit1
emitProc 中 (Main.java) 中调用 ProcEntryExit2
这样最终生成的关于函数体的汇编代码为:

设置函数体标号
计算 $SP, 分配帧空间
保存原 $FP

计算新 $FP ($fp=$sp-帧空间+4 bytes)
保存 $RA
保存 Callee-Saved 寄存器
保存参数 $A0~$A3
**[函数体]**
将函数体返回值写入 $RV
恢复 Callee-Saved 寄存器
恢复 $RA
恢复 $FP
恢复 $SP, 将$SP 加上相应的帧空间
跳转到返回地址

其中绿色部分由 ProcEntryExit3 完成,蓝色部分由 ProcEntryExit 完成,其余由 ProcEntryExit2 完成

# 第 6 部分 中间代码生成

## 6.1 中间代码

中间表示 (IR) 是一种抽象的机器语言,它无需太多地考虑机器特性的细节就可以对目标机的操作进行表达.这里用中间树 (IR Tree) 来表示.

中间树的结点:(定义在 Tree 包中)

Tree.exp 和 Tree.stm 的区别在于有无返回值

从 Tree.Exp 中派生,表示有返回值的表达式:

CONST (i) 整数常量 i

NAME (n) 字符常量 n

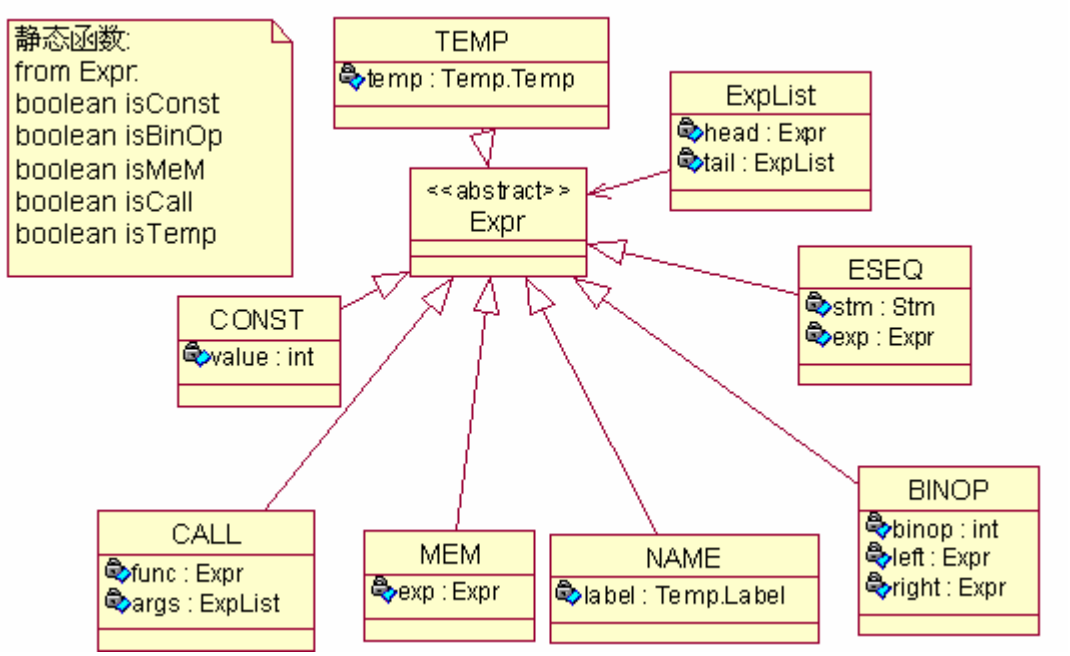
TEMP (t) 临时变量,是实现机中理想的寄存器

BINOP (o, t1, t2) 用运算符 o 对 t1 和 t2 进行运算,包括算术、逻辑运算等

MEM (e) 表示地址 e 的存储器中 wordsize 字节的内容

CALL (f, l) 函数调用,函数为 f,参数列表为 l

ESEQ (s, e) 先计算 stm s,再根据 stm s 计算 exp e,得出结果




从 Tree.Stm 中派生,表示无返回值的表达式:

MOVE (d, s) 将源 s 移入目标 d

其中的两种常见情况为:

MOVE (TEMP (t), e) 计算 e 并把它放入临时单元 t 中

MOVE (MEM (e1), e2) 计算 e1,得到地址 a,再计算 e2,放入地址 a

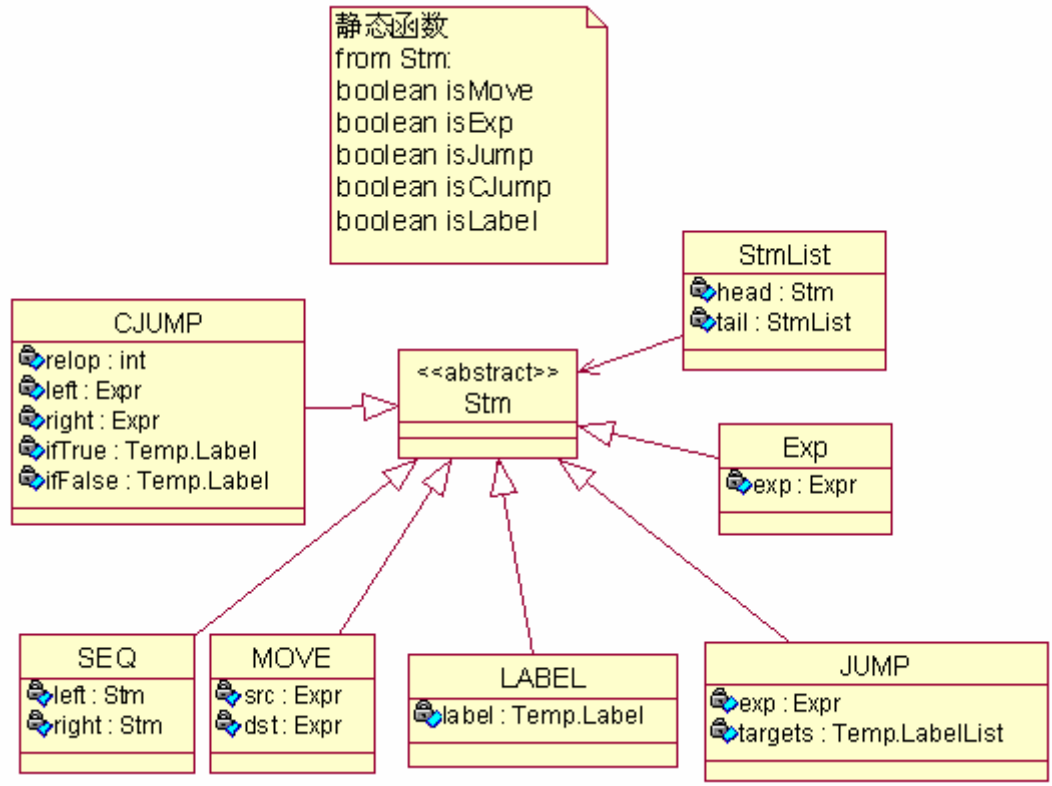
EXP (e) 计算 e, 释放结果

JUMP (e, labs) 无条件跳转到 labs (只使用链表中的第一个 label, e 可以忽略掉)

CJUMP (o, e1, e2, t, f) 对 e1, e2 求值,再用 o 运算.结果为真跳到 t,为假跳到 f.比较关系定义在 CJUMP.java 中,如 CJUMP.EQ, CJUMP.NE, CJUMP.LT

SEQ (s1, s2) 将 stm s2 放在 stm s1 后面

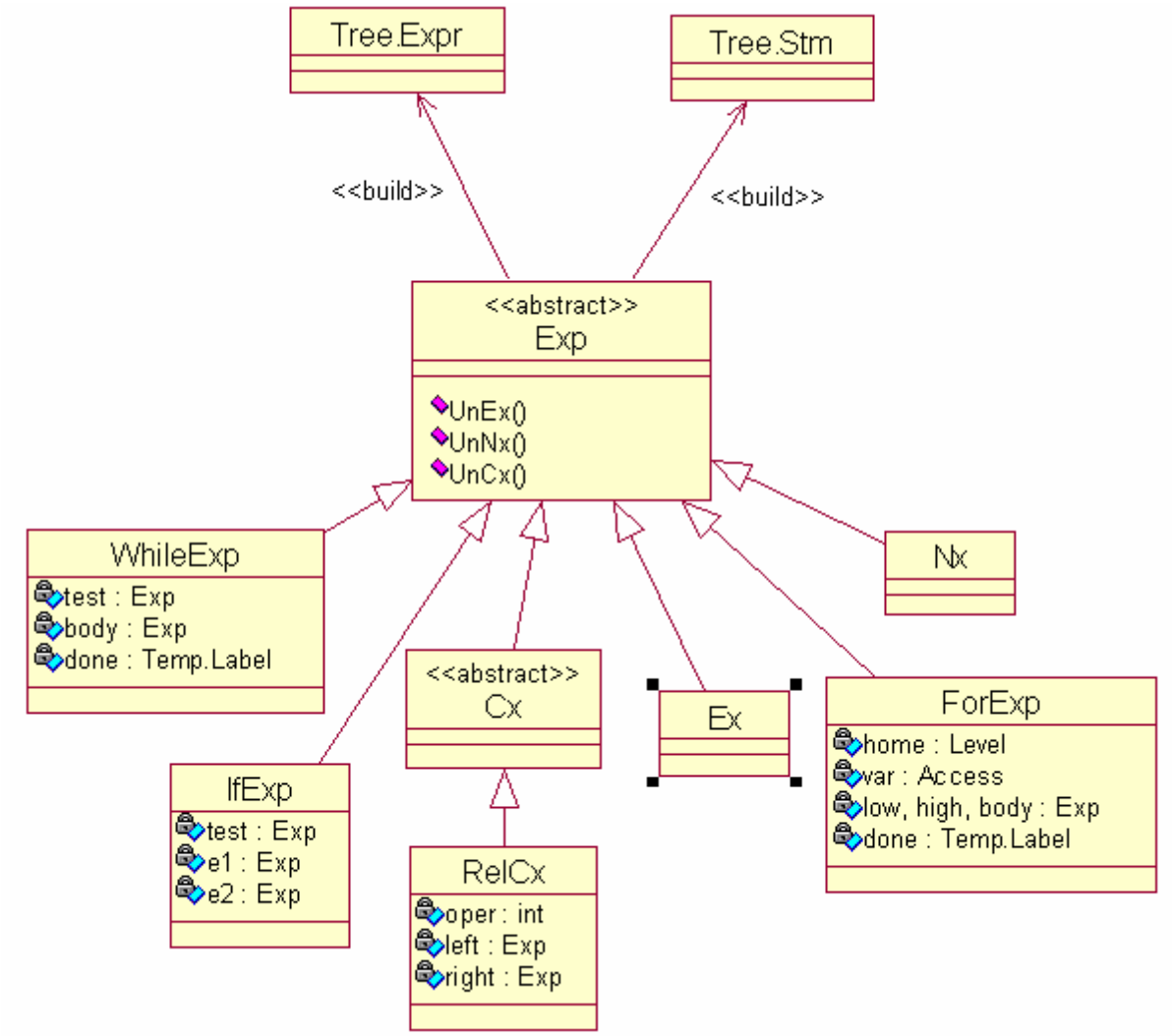
LABEL (n)　　定义标号 n 作为当前机器码地址




另外在 Tree 包中还有 ExpList 和 StmList,表示有(无)返回值表达式的链表

注意区分 Exp 和 Expr,前者是 Stm 的一个子类表示无返回值的表达式,后者是有返回值的抽象类

### 6.2 Translate 包中的表达式




这些类可以认为是产生 IR 树的"宏观构造者", 或是"代理建造者"一 translate.java 不直接生成树的具体结点,而是先生成这些"代理类",由它们通过方法 Ex, Cx 或 Nx 来生成具体的结点.

### 6.3 Ex, Cx 和 Nx

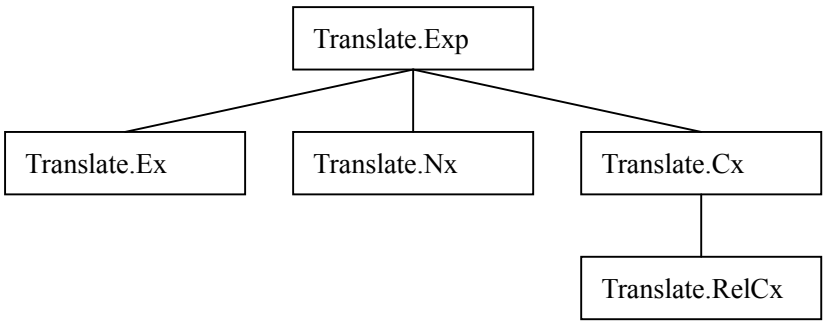
对一个宏观表达式(Translate.Exp)又三种翻译方法:Ex,Cx 和 Nx

Ex 为有返回值的表达式

Nx 为无返回值的表达式

Cx 表示一个判断条件,有真假两个出口

它们可以分别用下面的抽象类表示:



在 Translate.Exp 是抽象类,有三个函数需要子类重载:UnEx, UnNx 和 UnCx 分别表示转换到 Ex, Nx 和 Cx (RelCx 是 Cx 的具体化). 它们的返回值分别为 Tree.Expr, Tree.Stm, Tree.Stm

*a Translate.Exp 清单*

```
public abstract class Exp {
    abstract Tree.Expr unEx();
    abstract Stm unNx();
    abstract Stm unCx(Label t, Label f);
}
```

*b Translate.Ex 清单*

```
public class Ex extends Exp {
    Tree.Expr exp = null;
    Ex(Tree.Expr exp) {this.exp = exp;}
    Tree.Expr unEx() {  return exp;} //Tree.Expr 本身就是有返回值表达式
    Stm unNx() { return new EXP(exp);} //Tree.EXP 无返回值表达式
    Stm unCx(Label t, Label f) {
        //若表达式非 0 转到 t,否则转到 f
        //if (exp!=0) goto T else goto F
        return new CJUMP(CJUMP.NE, exp, new CONST(0), t, f);
    }
}
```

*c Translate.Cx 清单*

```
public abstract class Cx extends Exp {
    //Cx→Ex 比较复杂:
    //r=1 (r 为返回值)
    //if (exp!=0) goto T else goto F (这句的具体处理由子类完成)
    //LABEL f:
    //r=0
    //LABEL t:
    //return r
    Tree.Expr unEx() {
        Temp r = new Temp();
        Label t = new Label();
        Label f = new Label();
        return new ESEQ(
                new SEQ(new MOVE(new TEMP(r), new CONST(1)),
                        new SEQ(unCx(t, f),
                        new SEQ(new LABEL(f),
                        new SEQ(new MOVE(new TEMP(r), new Tree.CONST(0)),
                        new LABEL(t))))),
                new TEMP(r));
```

```
    }
    Stm unNx() {return new EXP(unEx());} //留给子类具体处理
    abstract Stm unCx(Label t, Label f); //留给子类具体处理
}
```

*d Translate.Nx 清单*

```
public class Nx extends Exp {
    Stm stm;
    Nx(Stm stm) {   this.stm = stm; }
    Tree.Expr unEx() {  return null;} //无法操作
    Stm unNx() {return stm;} //返回 stm 本身
    Stm unCx(Label t, Label f) { return null;} //无法操作
}
```

*e Translate.RelCx 清单*

```
public class RelCx extends Cx {
    int oper = 0;
    Exp left = null;
    Exp right = null;
    public RelCx(int oper, Exp left, Exp right) {
        switch (oper) {
        case EqExp.EQ: //=
            this.oper = CJUMP.EQ;
            break;
        case EqExp.NE: ...//!=
        case ... //省略
        }
        this.left = left;
        this.right = right;
    }

    //先把左右都翻译成由返回值的 Ex
    //再用 CJUMP 处理跳转
    public Stm unCx(Label t, Label f) {
        return new CJUMP(oper, left.unEx(), right.unEx(), t, f);
    }
}
```

*f Translate.IfExp 清单*

```
public class IfExp extends Exp {
    private Exp test; //测试条件
    private Exp e1; //then 子句
    private Exp e2; //else 子句
```

```
    IfExp(Exp test, Exp e1, Exp e2) {
        this.test = test;
        this.e1 = e1;
        this.e2 = e2;
    }

    //if-else 这样翻译成有返回值的表达式:
    //if (test) goto T else goto F
    //LABEL T: r=e1
    //goto JOIN
    //LABEL F: r=e2
    //LABEL JOIN: return r
    Expr unEx() {
        Temp.Temp r = new Temp.Temp();
        Temp.Label join = new Temp.Label();
        Temp.Label t = new Temp.Label();
        Temp.Label f = new Temp.Label();
        return new ESEQ(new SEQ(test.unCx(t, f),
                        new SEQ(new LABEL(t),
                        new SEQ(new MOVE(new TEMP(r), e1.unEx()),
                        new SEQ(new JUMP(join),
                        new SEQ(new LABEL(f),
                        new SEQ(new MOVE(new TEMP(r), e2.unEx()),
                                new LABEL(join)))))))),
                        new TEMP(r));
    }

    //这样翻译成无返回值的 if-then
    //如果没有 else 子句
    //  if (test) goto T else goto JOIN
    //  LABEL T: e1
    //  LABEL JOIN:
    //如果有 else 子句
    //  if (test) goto T else goto F
    //  LABEL T: e1
    //  goto JOIN
    //  LABEL F: e2
    //  LABEL JOIN:
    Stm unNx() {
        Temp.Label join = new Temp.Label();
        Temp.Label t = new Temp.Label();
        Temp.Label f = new Temp.Label();
        if (e2 == null)
            return  new SEQ(test.unCx(t, join),
```

```
                    new SEQ(new LABEL(t),
                    new SEQ(e1.unNx(),
                            new LABEL(join))));
        else
            return  new SEQ(test.unCx(t, f),
                    new SEQ(new LABEL(t),
                    new SEQ(e1.unNx(),
                    new SEQ(new JUMP(join),
                    new SEQ(new LABEL(f),
                    new SEQ(e2.unNx(),
                            new LABEL(join)))))));
    }

    //直接变成 Cx:
    //if test!=0 goto T else goto F
    Stm unCx(Label t, Label f) {
        return new CJUMP(CJUMP.NE, unEx(), CONST._zero, t, f);
    }
}
```

*g Translate.WhileExp 清单*

```
public class WhileExp extends Exp {
    Exp test = null;    //测试条件
    Exp body = null;    //循环体
    Label done = null;  //完成出口

    WhileExp(Exp test, Exp body, Label done) {
        this.test = test;
        this.body = body;
        this.done = done;
    }

    //whlie 没有返回值
    Expr unEx() {
        System.err.println("WhileExp.unEx()");
        return null;
    }

    //LABEL BEGIN:
    //if (test) goto T else goto DONE
    //LABEL T:
    //body
    //goto BEGIN
    //LABEL DONE:
```

```
    Stm unNx() {
        Label begin = new Label();
        Label t = new Label();
        return  new SEQ(new LABEL(begin),
                new SEQ(test.unCx(t, done),
                new SEQ(new LABEL(t),
                new SEQ(body.unNx(),
                new SEQ(new JUMP(begin),
                        new LABEL(done))))));
    }

    //while 只有一个出口
    Stm unCx(Label t, Label f) {
        System.err.println("WhileExp.unCx()");
        return null;
    }
}
```

*h Translate.ForExp 清单*

```
public class ForExp extends Exp {
    Level home;     //层
    Access var;     //循环变量
    Exp low, high;  //初试值、终止值
    Exp body;       //循环体
    Label done;     //完成后出口

    ForExp(Level home, Access var, Exp low, Exp high, Exp body, Label done) {
        this.home = home;
        this.var = var;
        this.low = low;
        this.high = high;
        this.body = body;
        this.done = done;
    }
    //for 不能有返回值
    Expr unEx() {
        System.err.println("ForExp.unEx() should not be called.");
        return null;
    }

    //for 只有一个出口
    Stm unCx(Label t, Label f) {
        System.err.println("ForExp.unEx() should not be called.");
        return null;
```

```
    }

    /*
        循环变量和循环上限被分配在帧空间中
        MOVE VAR, LOW
        MOVE LIMIT, HIGH
        if (VAR<=LIMIT) goto BEGIN else goto DONE
        LABEL BEGIN:
        body
        if (VAR<LIMIT) goto GOON else goto DONE
        LABEL GOON:
        VAR=VAR+1
        GOTO BEGIN:
        LABEL DONE:
    */
    Stm unNx() {
        Access limit = home.allocLocal(true);
        Label begin = new Label();
        Label goon = new Label();

        return  new SEQ(new MOVE(var.access.exp(new TEMP(home.frame.FP())),
low.unEx()),    new SEQ(new MOVE(limit.access.exp(new TEMP(home.frame.FP())),
high.unEx()),   new SEQ(new CJUMP(CJUMP.LE, var.access.exp(new
TEMP(home.frame.FP())),limit.access.exp(new TEMP(home.frame.FP())), begin,
done),  new SEQ(new LABEL(begin),   new SEQ(body.unNx(), new
 SEQ(new CJUMP(CJUMP.LT, var.access.exp(new TEMP(home.frame.FP())),
limit.access.exp(new TEMP(home.frame.FP())), goon, done),  new
 SEQ(new LABEL(goon),   new SEQ(new MOVE(   var.access.exp(new
TEMP(home.frame.FP())), new BINOP(BINOP.PLUS, var.access.exp(new
TEMP(home.frame.FP())), CONST._one)),  new SEQ(new JUMP(begin),
new LABEL(done)))))))));
    }
}
```

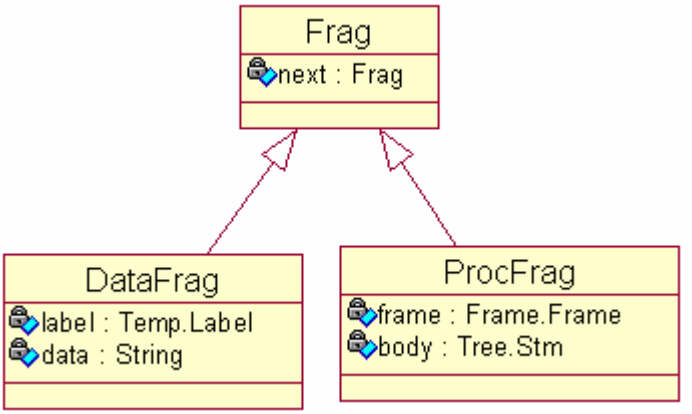
### 6.4 段 (Frag)

段可能是一个字符串或一个函数块,被封装成 Frag 类,段可以自成一链

字符串为 DataFrag, 其中有标号和字符数据.处理字符串时,一般操作的是标号 (引用)

函数块为 ProcFrag, 其中有函数体 (Tree.Stm) 和其 Frame

在翻译过程中,段总是不断增大的,段链表被私有地维护在 translate 类中,它提供了两个接口:AddFrag 和 GetResult 用于添加段和取回整个段链表




```
public class Frag { //Frag.java
    public Frag next = null;
}
public class DataFrag extends Frag { //DataFrag.java
    Label label = null;
    public String data = null;
}
//Translate.java
private Frag frags = null;
public Frag getResult() {return frags;}
public void addFrag(Frag frag) {
    frag.next = frags;frags = frag;
}
```

Translate 中的 transStringExp 在翻译字符串时调用 addFrag 把字符串加入数据段列表中
Translate 中的 procEntryExit 在翻译函数时把整个函数的 IR 树结点加入函数段中
具体细节见 6.6

## 6.5 翻译成 IR 树的一般过程

这个过程的输入是一棵抽象语法树,输出是拥有 6.1 中结点的 IR 树
1 抽象语法树 →
2 Semant 检查 (第四部分) →
3 Translate 翻译出宏观树型→
4 通过 Translate.Ex, Cx 或 Nx 生成具体 IR 树结点

## 6.6 具体翻译过程

*a 整型常数 value*

**CONST (value)** (粗体表示翻译的 IR 树结果,下同)
代码: Exp transIntExp(int value)

```
return new Ex(new CONST(value));
```

*b 字符串常数 value*

**NAME (label)**

代码: Exp transStringExp(String value)

```
Label l = new Label(); //新建一个标号
addFrag(new DataFrag(l, frame.string(l, value)));
//将字符串添加到段 (DataFrag) 中去,段存放在 translate 的私有段列表中
return new Ex(new NAME(l)); //返回 NAME, 以标号作为标示
```

*c nil 变量*

**用常数 0 代替 EX(CONST(0))**

代码:Exp transNilExp()

```
return new Ex(CONST(0));
```

*d 变量表达式 translate.Exp*

已经是 translate.Exp 了,无需转换,直接返回 **Exp**

代码:Exp transVarExp(Exp ex)

*e 运算表达式 left oper right (加减乘除)*

**EX(BINOP (binop, left, right))**

代码: Exp transCalcExp(int oper, Exp left, Exp right)

```
return new Ex(new BINOP(binOp, left.unEx(), right.unEx()));
```

其中 binOp 中为常数 BINOP.PLUS, BINOP.MINUS, BINOP.MUL, BINOP.DIV 等

*f 字符串运算 left oper right*

**RELCX(OPER, EX (COMP) , EX (0))**

其中 COMP 是调用外部函数 stringCompare 得出的比较结果

代码: Exp transStringRelExp

```
Expr comp = home.frame.externalCall("_stringCompare", new ExpList(left.unEx(),
            new ExpList(right.unEx(), null)));
return new RelCx(oper, new Ex(comp), new Ex(CONST._zero));
```

*g 其它关系运算 left oper right*

**RELCX (OPER, LEFT, RIGHT)**

代码: Exp transOtherRelExp(int oper, Exp left, Exp right)

```
return new RelCx(oper, left, right);
```

*h 赋值运算 lvalue=ex*

**NX(MOVE (lvalue, ex))**

代码: Exp transAssignExp(Exp lvalue, Exp ex)

```
return new Nx(new MOVE(lvalue.unEx(), ex.unEx()));
```

*i TransCallExp*

EX(CALL(NAME(func_name), args))

代码: Exp transCallExp(Level home, Level dest, Label name, ArrayList argValue)

注意:静态链接被隐式地作为第一个参数

```
//抽取参数
```

```
        ExpList args = null;
        for (int i = argValue.size() - 1; i >= 0; --i)
            args = new ExpList(((Exp) argValue.get(i)).unEx(), args);
        Level l = home;
        Expr slnk = new TEMP(l.frame.FP()); //静态链接
        //找到 Callee 直接上层的静态连接
        while (dest.parent != l) {
            slnk = l.staticLink().access.exp(slnk);
            l = l.parent;
        }
        //将静态连接作为第一个参数
        args = new ExpList(slnk, args);
        return new Ex(new CALL(new NAME(name), args));
```

*j 库函数调用 TransExtCallExp*

**EX ( EXTERNAL_CALL )**

处理完参数后直接调用

代码: Exp transExtCallExp(Level home, Label name, ArrayList argValue)

```
        return new Ex(home.frame.externalCall("_" + name, args));
```

*k stm 的连接 e1, e2*

**NX(SEQ(NX(E1), NX(E2)))**

注意处理空的情况

代码: Exp combine2Stm(Exp e1, Exp e2)

```
        if (e1 == null)
            return new Nx(e2.unNx());
        else if (e2 == null)
            return new Nx(e1.unNx());
        else
            return new Nx(new SEQ(e1.unNx(), e2.unNx()));
```

*l exp 的连接 e1, e2*

**EX(ESEQ(NX(E1), EX(E2)))**

注意处理空的情况,且返回值为后面的,故 e1 为 Nx, e2 为 Ex

代码: Exp combine2Exp(Exp e1, Exp e2)

```
        if (e1 == null)
            return new Ex(e2.unEx());
        else
            return new Ex(new ESEQ(e1.unNx(), e2.unEx()));
```

这里正是 ESEQ 的用处,两个表达式连接的返回值是后一个表达式,前面可以作为无返回值的 stm

*m transRecordExp*

**EX(ESEQ(SEQ(MOVE(TEMP(addr),alloc),init),TEMP(addr)))**

alloc 为记录首地址,由外部函数 allocRecord 分配

init 为一串把记录内容送到帧空间中的 MOVE 指令(先用 BINOP.PLUS 计算出偏移地址)

返回记录的首地址 addr

```
    public Exp transRecordExp(Level home, ArrayList field) {
        Temp addr = new Temp();
        //调用外部函数 _allocRecord 为记录在 frame 上分配空间,
        // 并得存储空间首地址
        //_allocRecord 执行如下的类 C 代码,注意它只负责分配空间
        //初始化操作需要我们来完成
        //# int *allocRecord(int size)
        //# {int i;
        //#  int *p, *a;
        //#  p = a = (int *)malloc(size);
        //#  for(i=0;i<size;i+=sizeof(int)) *p++ = 0;
        //#  return a;
        //# }
        //注意如果记录为空,也要用 1 个 word,否则每个域为一个 word,按顺序存放
        Expr alloc = home.frame.externalCall("_allocRecord", new ExpList(
                new CONST((field.size() == 0 ? 1 : field.size()) *
                wordSize),null));
        Stm init = transNoOp().unNx(); //初始化指令
        for (int i = field.size() - 1; i >= 0; --i) {
            //为记录中每个域生成 MOVE 指令,将值复制到帧中的相应区域
            Expr offset = new BINOP(BINOP.PLUS, new TEMP(addr),
                new CONST(i * wordSize));
            Expr v = ((Exp) field.get(i)).unEx();
            init = new SEQ(new MOVE(new MEM(offset), v), init);
        }
        //返回记录的首地址
        return new Ex(new ESEQ(new SEQ(new MOVE(new TEMP(addr), alloc), init),
                new TEMP(addr)));
    }
```

*n transArrayExp*

**EX(alloc)**

alloc 为数组首地址,由外部函数 initArray 分配

```
    public Exp transArrayExp(Level home, Exp init, Exp size) {
        //调用外部函数 initArray 为数组在 frame 上分配存储空间,并得到
        //存储空间首地址
        //initArray 执行如下的类 C 代码,需要提供数组大小与初始值
        //# int *initArray(int size, int init)
        //# {int i;
        //#  int *a = (int *)malloc(size*sizeof(int));
```

```
        //#  for(i=0;i<size;i++) a[i]=init;
        //#  return a;
        //# }
        Expr alloc = home.frame.externalCall("_initArray", new ExpList(size
                .unEx(), new ExpList(init.unEx(), null)));
        return new Ex(alloc);
    }
```

*o if, while, for*

被封装在 translate.IfExp 中,分别翻译成:

**IfExp, WhileExp, ForExp**

注意:while 和 for 只能翻译成 Nx 而不能是 Ex 或 Cx

代码: Exp transIfExp(Exp test, Exp e1, Exp e2)

Exp transWhileExp(Exp test, Exp body, Label done)

Exp transForExp(Level home, Access var, Exp low,Exp high,Exp body,Label done)

```
        return new IfExp(test, e1, e2);
        return new WhileExp(test, body, done);
        return new ForExp(home, var, low, high, body, done);
```

*p break*

**NX (JUMP (LABEL))**

LABEL 是堆栈 LoopExit 的栈顶,在第 4 部分语义分析的对 break 的描述中有介绍

代码:Exp transBreakExp()

```
        return new Nx(new JUMP((Label) loopExit.peek()))
```

*q 简单变量 (存储在 Frame.Access 中)*

**EX(MEM (BINOP (PLUS, TEMP(FP), CONST(OFFSET))))**

它由 Frame.Access 的 Exp 方法完成:

Fp: 当前帧指针寄存器

Offset:地址偏移量

代码: Exp transSimpleVar(Access access, Level home)

首先要找到变量所在的帧: 沿着静态链接不断上溯, 直到变量的层与当前层相同

```
        Expr res = new TEMP(home.frame.FP());
        Level l = home;
        while (l != access.home) {
            res = l.staticLink().access.exp(res);
            l = l.parent;
        }
```

然后由 Frame.Access 的 Exp 方法返回由 fp 的偏移地址作为变量地址

```
        return new MEM(new BINOP(BINOP.PLUS, framePtr, new CONST(offset)));
```

*r 下标变量 var[idx]*

**EX(MEM(BINOP(BINOP.PLUS, EX(VAR),**

**BINOP(MUL,EX(IDX),CONST(WORDSIZE)))**

代码: Exp transSubscriptVar(Exp var, Exp idx)

```
Expr arr_addr = var.unEx(); //arr_addr 数组首地址
//arr_off 偏移量, 等于下标乘以字长
Expr arr_off = new BINOP(BINOP.MUL, idx.unEx(), new CONST(wordSize));
return new Ex(new MEM(new BINOP(BINOP.PLUS, arr_addr, arr_off)));
```

*s 域变量 var[num]*

**EX(MEM(BINOP(BINOP.PLUS, EX(VAR) , CONST(NUM*WORDSIZE)))**

代码: Exp transFieldVar(Exp var, int num)

```
Expr rec_addr = var.unEx(); .//记录首地址
//偏移量 (每个记录项目占一个 wordsize)
Expr rec_off = new CONST(num * wordSize);
return new Ex(new MEM(new BINOP(BINOP.PLUS, rec_addr, rec_off)));
```

*t 翻译函数体:*

由 procEntryExit 函数实现,它为函数体加上返回值的信息, 并整个函数的 IR 树结点加入段中

这个函数十分重要,细节见 5.4 和 5.6 节

# 第 7 部分 规范化

从上一部分的 IR 树出发,规范化过程将帮助完成下面的工作:

a IR 树被写成一个没有 SEQ 和 ESEQ 结点的规范树表

b 根据该表划分基本块,每个基本块中不包含内部跳转和标号

c 基本块被顺序放置,所有的 CJUMP 都跟有 false 标号

不需要知道任何细节,这部分只需将提供的 Canon 文件夹复制,对于以上 a,b,c 三步,只需在最后装配时调用下面的接口就可以:

a `Canon.Canon.linearize (Tree.Stm s)`

b `Canon.BasicBlocks. BasicBlocks(Tree.StmList stms)`

c `Canon.TraceSchedule(BasicBlocks b)`

# 第 8 部分 指令选择

## 8.1 常用 MIPS 汇编指令

| | |
|---|---|
| `sw reg, addr` | 将寄存器 reg 保存到内存地址 addr 中 |
| `lw reg, addr` | 将内存地址 addr 中的内容保存到寄存器 reg 中 |
| `li reg, imm` | 将立即值 imm 保存到寄存器 reg 中 |
| `la reg, addr` | 将地址 addr (而不是其中的内容) 保存到寄存器 reg 中 |
| `move reg1, reg2` | 将寄存器 reg2 移动到寄存器 reg1 中 |
| `jal label` | 无条件转移到 label, 返回地址 (下一指令) 存于$Ra 寄存器 |
| `jr reg` | 无条件转移到 reg 中所保存的地址, 返回地址存于$Ra |
| `subu reg1,reg2,CONST` | 将 `reg2` 的值减去 `CONST` 后放入 `reg1` |
| `addu reg1,reg2,CONST` | 将 `reg2` 的值加上 `CONST` 后放入 `reg1` |
| `beq reg, src, label` | 当 reg=src 时跳转到 label |
| `bge reg, src, label` | 当 reg>=src 时跳转到 label |
| `bgt reg, src, label` | 当 reg>src 时跳转到 label |
| `ble reg, src, label` | 当 reg<=src 时跳转到 label |
| `blt reg, src, label` | 当 reg<src 时跳转到 label |
| `bne reg, src, label` | 当 reg!=src 时跳转到 label |
| `add reg1, reg2, src` | 将 reg2+src 送到 reg1 |
| `sub reg1, reg2, src` | 将 reg2-src 送到 reg1 |
| `mul reg1, reg2, src` | 将 reg2*src 送到 reg1 |
| `div reg1, reg2, src` | 将 reg2/src 送到 reg1 |

## 8.2 MIPS 中的寄存器

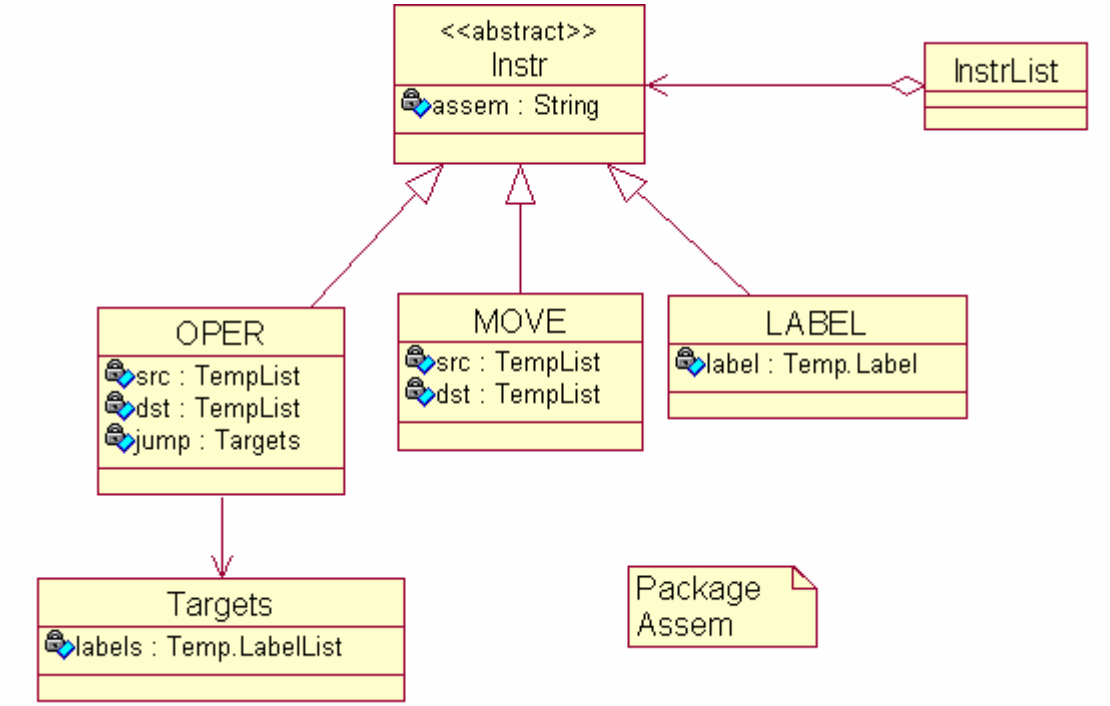
| Register name | Number | Usage |
|---|---|---|
| $zero | 0 | constant 0 |
| $at | 1 | reserved for assembler |
| $v0 | 2 | expression evaluation and results of a function |
| $v1 | 3 | expression evaluation and results of a function |
| $a0 | 4 | argument 1 |
| $a1 | 5 | argument 2 |
| $a2 | 6 | argument 3 |
| $a3 | 7 | argument 4 |
| $t0 | 8 | temporary (not preserved across call) |
| $t1 | 9 | temporary (not preserved across call) |
| $t2 | 10 | temporary (not preserved across call) |
| $t3 | 11 | temporary (not preserved across call) |
| $t4 | 12 | temporary (not preserved across call) |
| $t5 | 13 | temporary (not preserved across call) |
| $t6 | 14 | temporary (not preserved across call) |
| $t7 | 15 | temporary (not preserved across call) |
| $s0 | 16 | saved temporary (preserved across call) |
| $s1 | 17 | saved temporary (preserved across call) |
| $s2 | 18 | saved temporary (preserved across call) |
| $s3 | 19 | saved temporary (preserved across call) |
| $s4 | 20 | saved temporary (preserved across call) |
| $s5 | 21 | saved temporary (preserved across call) |
| $s6 | 22 | saved temporary (preserved across call) |
| $s7 | 23 | saved temporary (preserved across call) |
| $t8 | 24 | temporary (not preserved across call) |
| $t9 | 25 | temporary (not preserved across call) |
| $k0 | 26 | reserved for OS kernel |
| $k1 | 27 | reserved for OS kernel |
| $gp | 28 | pointer to global area |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | return address (used by function call) |

其中$s0 到$s7 是 callee-save 寄存器, $t0 到$t9 是 caller-save 寄存器

### 8.3 Assem.Instr 数据类型

可以为无需进行寄存器分配的那些汇编语言指令构造一种称为 Instr 的数据类型

OPER, MOVE, LABEL 派生自 Instr,它们均被封装在 Assem 包中




OPER 方法中包含汇编语言指令 assem,一个操作数寄存器列表 src 和一系列的结果寄存器 dst,这些寄存器列表都可以是空的.方法 OPER(assem, dst, src) 将构造跳转至下一条指令的操作,并且 jumps()方法的返回值为 null;OPER(assem, dst, src, jump) 还拥有目标标记列表并通过查找这个列表进行跳转

LABEL 表示程序将要跳转的地点.其中包括一个 assem,指明在汇编语言中如何找到标号,有一个 label 参数,用于确定使用那个标号符号

MOVE 与 OPER 很接近,但 MOVE 必须进行数据转换.如果 dst 和 src 被分配给了同一个寄存器,MOVE 指令将被删除

Instr.Format(m) 将汇编指令转化成为字符串形式:m 是一个实现 TempMap 接口的对象,在 TempMap 接口中,存在一个方法可以为每个临时变量分配一个寄存器或者为不同的变量分配相同的寄存器 (给出名字) .请参阅 Temp.TempMap

Instr 是与特定的机器无关的

### 8.4 生成汇编指令

这些工作在 Mips.Codegen 中完成

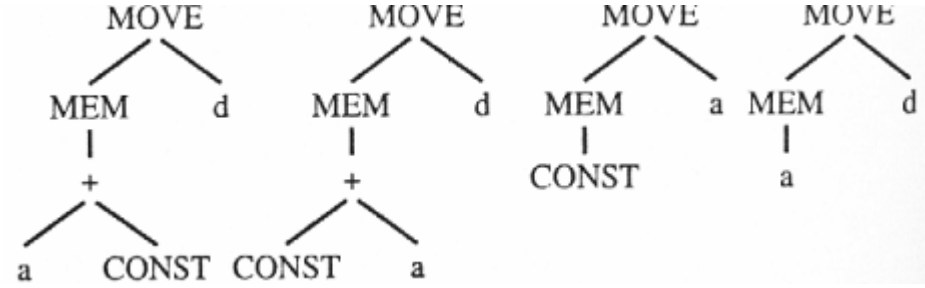
*a 以 Tree.MOVE 为根:*

```
public void munchStm(MOVE s) {
        Expr dst = s.dst;
        Expr src = s.src;
        if (dst instanceof MEM) {
            MEM dst1 = (MEM) dst;
            if (Expr.isBINOP(dst1.exp) //情况 1
                    && ((BINOP) dst1.exp).binop == BINOP.PLUS
                    && Expr.isCONST(((BINOP) dst1.exp).right)) {
                Temp t1 = src.munchExp(this);
                Temp t2 = ((BINOP) dst1.exp).left.munchExp(this);
```

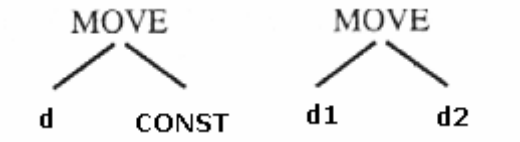
```
                emit(new OPER("sw `s0, "
                        + ((CONST) ((BINOP) dst1.exp).right).value + "(`s1)",
                        null, new TempList(t1, new TempList(t2, null))));
            } else if (Expr.isBINOP(dst1.exp) //情况 2
                    && ((BINOP) dst1.exp).binop == BINOP.PLUS
                    && Expr.isCONST(((BINOP) dst1.exp).left)) {
                Temp t1 = src.munchExp(this);
                Temp t2 = ((BINOP) dst1.exp).right.munchExp(this);
                emit(new OPER("sw `s0, "
                        + ((CONST) ((BINOP) dst1.exp).left).value + "(`s1)",
                        null, new TempList(t1, new TempList(t2, null))));
            } else if (Expr.isCONST(dst1.exp)) { //情况 3
                Temp t1 = src.munchExp(this);
                emit(new OPER("sw `s0, " + ((CONST) dst1.exp).value, null,
                        new TempList(t1, null)));
            } else { //情况 4
                Temp t1 = src.munchExp(this);
                Temp t2 = dst1.exp.munchExp(this);
                emit(new OPER("sw `s0, (`s1)", null, new TempList(t1,
                        new TempList(t2, null))));
            }
        } else if (Expr.isTEMP(dst))
            if (Expr.isCONST(src)) { //情况 5
                emit(new OPER("li `d0, " + ((CONST) src).value, new TempList(
                        ((TEMP) dst).temp, null), null));
            } else { //情况 6
                Temp t1 = src.munchExp(this);
                emit(new OPER("move `d0, `s0", new TempList(((TEMP) dst).temp,
                        null), new TempList(t1, null)));
            }
    }
```

情况 1 到 4 为下面四种 IR 树形式 (写入内存) :

**注意:本部分中的 IR 树用 d 标记数据寄存器,a 标记地址寄存器,后同**




情况 5 和 6 为下面两种 IR 树形式 (寄存器或常数写入寄存器):




情况 1:`sw d, CONST (a)` `//CONST 为常数偏移量, a 为内存地址,下同`
情况 2:`sw d, CONST (a)`
情况 3:`sw d, CONST`
情况 4:`sw d, CONST`
情况 5:`li d, CONST` `//CONST 写入寄存器 d`
情况 6:`move d1, d2` `//d2 写入寄存器 d1`

*b 以 Tree.LABEL 为根:很简单,直接输出标号的字符串表示*

翻译成:`LABEL:`

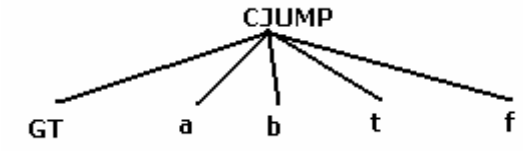
```
public void munchStm(LABEL l) {
    emit(new Assem.LABEL(l.label.toString() + ":", l.label));
}
```

*c 以 Tree.JUMP 为根:强制跳转*

翻译成 `JAL LABEL`

```
public void munchStm(JUMP j) {
    emit(new OPER("j " + j.targets.head, null, null, j.targets));
}
```

*d 以 Tree.CJUMP 为根:*




这样翻译:
`BGT A, B, T`
因为经过第 7 部分规范化处理,所以下一条指令肯定为 `F`
其它关系运算同理

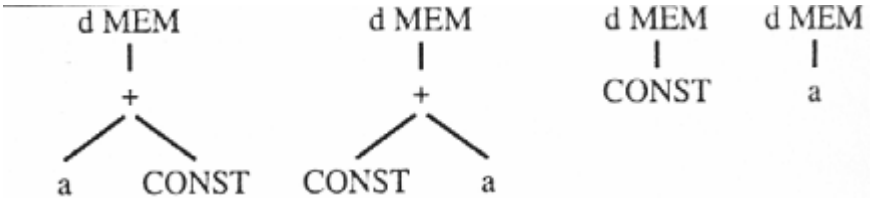
```
public void munchStm(CJUMP j) {
    String oper = null;
    switch (j.relop) {
    case CJUMP.EQ:
        oper = "beq";
        break;
    case CJUMP.NE:
        oper = "bne";
        break;
    case CJUMP.GT:
        oper = "bgt";
        break;
```

```
        case CJUMP.GE:
            oper = "bge";
            break;
        case CJUMP.LT:
            oper = "blt";
            break;
        case CJUMP.LE:
            oper = "ble";
            break;
        }
        Temp t1 = j.left.munchExp(this);
        Temp t2 = j.right.munchExp(this);
        emit(new OPER(oper + " `s0, `s1, `j0", null, new TempList(t1,
                new TempList(t2, null)), new LabelList(j.iftrue, new LabelList(
                j.iffalse, null))));
    }
```

*e 以 Tree.EXP 为根:继续翻译具体的表达式*

```
    public void munchStm(EXP e) {
        e.exp.munchExp(this);
    }
```

*f 以 Tree.MEM 为根,有以下四种情况:*




情况 1:`lw d, CONST (a)`

情况 2:`lw d, CONST (a)`

情况 3:`lw d, CONST //不能使用 li 指令！因为 CONST 是地址不是值,相当于 C 中的 *ptr`

情况 4:`lw d, a`

```
    public Temp munchExp(MEM e) {
        Temp ret = new Temp();
        if (Expr.isBINOP(e.exp) && ((BINOP) e.exp).binop == BINOP.PLUS
                && Expr.isCONST(((BINOP) e.exp).right)) {
            Temp t1 = ((BINOP) e.exp).left.munchExp(this);
            emit(new OPER("lw `d0, " + ((CONST) ((BINOP) e.exp).right).value
                    + "(`s0)", new TempList(ret, null), new TempList(t1, null)));
//情况 1
        } else if (Expr.isBINOP(e.exp) && ((BINOP) e.exp).binop == BINOP.PLUS
                && Expr.isCONST(((BINOP) e.exp).left)) {
```

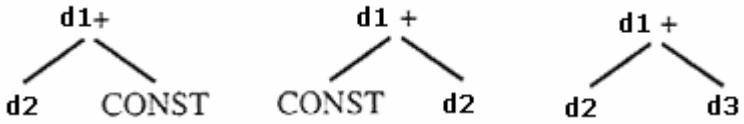
```
            Temp t1 = ((BINOP) e.exp).right.munchExp(this);
            emit(new OPER("lw `d0, " + ((CONST) ((BINOP) e.exp).left).value
                    + "(`s0)", new TempList(ret, null), new TempList(t1, null)));
            //情况 2
        } else if (Expr.isCONST(e.exp)) {
            emit(new OPER("lw `d0, " + ((CONST) e.exp).value, new TempList(ret,
                    null), null)); //情况 3
        } else {
            Temp t1 = e.exp.munchExp(this);
            emit(new OPER("lw `d0, (`s0)", new TempList(ret, null),
                    new TempList(t1, null))); //情况 4
        }
        return ret;
    }
```

*g 以 Tree.CONST 为根:IR 树表示为一个结点(用 d 标记)*


翻译成:li d, CONST

```
    public Temp munchExp(CONST e) {
        Temp ret = new Temp();
        emit(new OPER("li `d0, " + e.value, new TempList(ret, null), null));
        return ret;
    }
```

*h 以 Tree.BINOP 为根:*




以加法为例,以上三种情况分别翻译为:

```
ADD d1, d2, CONST //CONST 在这里是常数非地址
ADD d1, d2, CONST
ADD d1, d2, d3
```

//代码如下:

```
    public Temp munchExp(BINOP e) {
        Temp ret = new Temp();
        String oper = null;
        switch (e.binop) {
        case BINOP.PLUS:
            oper = "add";
            break;
        case BINOP.MINUS:
```

```
            oper = "sub";
            break;
        case BINOP.MUL:
            oper = "mul";
            break;
        case BINOP.DIV:
            oper = "div";
            break;
        }
        if (Expr.isCONST(e.right)) { //情况 1
            Temp t1 = e.left.munchExp(this);
            emit(new OPER(oper + " `d0, `s0, " + ((CONST) e.right).value,
                    new TempList(ret, null), new TempList(t1, null)));
        } else if (Expr.isCONST(e.left)) { //情况 2
            Temp t1 = e.right.munchExp(this);
            emit(new OPER(oper + " `d0, `s0, " + ((CONST) e.left).value,
                    new TempList(ret, null), new TempList(t1, null)));
        } else { //情况 3
            Temp t1 = e.left.munchExp(this);
            Temp t2 = e.right.munchExp(this);
            emit(new OPER(oper + " `d0, `s0, `s1", new TempList(ret, null),
                    new TempList(t1, new TempList(t2, null))));
        }
        return ret;
    }
```

*i 以 Tree.TEMP 为根:直接返回寄存器*

```
    public Temp munchExp(TEMP t) {
        return t.temp;
    }
```

*j 以 Tree.NAME 为根:读入字符串的段地址到寄存器*


翻译为:LA d, MALLOC //注意是读入地址

```
    public Temp munchExp(NAME t) {
        Temp ret = new Temp();
        emit(new OPER("la `d0, " + t.label, new TempList(ret, null), null));
        return ret;
    }
```

*k 以 Tree.CALL 为根*

先向参数寄存器中填入参数:如果是常数参数,采用 li 指令:

```
li $ax, CONST // x 为 0, 1, 2, 3
```

否则采用 move 指令

```
move $ax, dy // x 为 0, 1, 2, 3
```

这里采用了简化的方法,由于 MIPS 之多有 4 个寄存器,所以参数个数不超过 4 个

最后执行跳转:

```
jal func.label
```

注意返回值,返回专用寄存器$V0,它保存函数返回值

```
    public Temp munchExp(CALL c) {
        TempList list = null;
        int i = 0;
        for (ExpList a = c.args; a != null; a = a.tail, ++i) {
            Temp t = null;
            if (a.head instanceof CONST)
                emit(new OPER("li $a" + i + ", " + ((CONST) a.head).value,
                        null, null));
            else {
                t = a.head.munchExp(this);
                emit(new OPER("move $a" + i + ", `s0", null, new TempList(t,
                        null)));
            }
            if (t != null)
                list = new TempList(t, list);
        }
        emit(new OPER("jal " + ((NAME) c.func).label, MipsFrame.callDefs, list));
        return MipsFrame.v0;
    }
```

### 8.5 寄存器的临时表示

在上面程序看出,寄存器在 OPER 中的表示方式为:如果是系统寄存器则直接使用其名称.例如$a0.否则采用如下表示方法:`sn 或 `dn 或 `jn.其中 n 为一个数字.前面的字符 s, d, j 和 OPER 类中的 src (源), dst (目的), jump (跳转) 相对应,后面的数字表示该寄存器列表中的第几个寄存器.表达的方法可以多样.例如下面的代码:

```
        emit(new OPER(oper + " `d0, `s0, `s1", new TempList(ret, null),
    new TempList(t1, new TempList(t2, null))));
```

也可以写成:

```
        emit(new OPER(oper + " `d0, `d1, `d2", new TempList(ret, new
TempList(t1, new TempList(t2, null))),null));
```

这里 d, s, j 仅仅有概念上的意义.

而在 Assem.Instr 中,函数 Format 负责对这种表示方法进行解析, 转换成真正意义上的寄存器名称(通过 Temp.TempMap):

```
public String format(Temp.TempMap m) {
```

```
        Temp.TempList dst = def();
        Temp.TempList src = use();
        Targets j = jumps();
        Temp.LabelList jump = (j == null) ? null : j.labels;
        StringBuffer s = new StringBuffer();
        int len = assem.length();
        for (int i = 0; i < len; i++)
            if (assem.charAt(i) == '`')
                switch (assem.charAt(++i)) {
                case 's': { //source
                    int n = Character.digit(assem.charAt(++i), 10);
                    s.append(m.tempMap(nthTemp(src, n)));
                }
                    break;
                case 'd': { //dest
                    int n = Character.digit(assem.charAt(++i), 10);
                    s.append(m.tempMap(nthTemp(dst, n)));
                }
                    break;
                case 'j': { //jump
                    int n = Character.digit(assem.charAt(++i), 10);
                    s.append(nthLabel(jump, n).toString());
                }
                    break;
                case '`':
                    s.append('`');
                    break;
                default:
                    throw new Error("bad Assem format");
                }
            else
                s.append(assem.charAt(i));

        return s.toString();
    }
```

### 8.6 MIPS 汇编语言语法

注释: 以 # 号开始

指令: 见 8.1

标识符: 以字母,下划线,点号,数字序列(不以数字开头)

标号 :标识符后加冒号

例如:

```
        .data
        item: .word 1
        .text
        .globl main # Must be global
        main: lw $t0, item
```

数值可以为十进制或十六进制,例如 255 或 0xff

字符串用引号括起来,可以使用如下转义字符:\n (换行) \t (tab) \" (引号)

还有如下会用到的汇编指令:

| | |
|---|---|
| .globl symbol | 声明 symbol 是全局的,main 函数必须是全局的(globl main) |
| .text | 声明指令区:后面跟上连续的汇编指令 |
| .data | 声明数据区:后面跟上连续的数据 |
| .word w1,w2… | 在数据区中存储连续的 32 位整数 |
| .asciiz str | 在数据区中存储字符串, 并用 null 结尾 |

# 第 9 部分 活性分析与寄存器分配

## 9.1 抽象的图结构

包 Graph 中封装了一个抽象的图结构,它支持通常图结构中结点和边的大多数操作.数据结构采用结点的邻接表表示,支持有向图

## 9.2 流图

该部分的第一步是根据汇编指令 (instrs) 生成流图

包 FlowGraph 中封装了流图结构 FlowGraph (抽象的) 和 AssemFlowGraph (面向汇编指令的) .在 FlowGraph 中定义了以下三个接口:

```
    public abstract class FlowGraph extends Graph.Graph {
        public abstract TempList def(Node node);    //定义的变量
        public abstract TempList use(Node node);    //使用的变量
        public abstract boolean isMove(Node node); //是否表示 move 操作
    }
```

在 AssemFlowGraph 中,每个结点代表一个汇编指令,边为可能的控制转移.并具体生成流图.

```
public class AssemFlowGraph extends FlowGraph {
    //根据汇编指令创建流图
    public AssemFlowGraph(InstrList instrs) {
        Dictionary labels = new Hashtable();    //标号表
        //添加结点和标号
        for (InstrList i = instrs; i != null; i = i.tail) {
            Node node = newNode();
            represent.put(node, i.head);
            if (i.head instanceof LABEL)
                labels.put(((LABEL) i.head).label, node);
        }
        //添加边
        for (NodeList node = nodes(); node != null; node = node.tail) {
            Targets next = instr(node.head).jumps(); //跳转标号表
            if (next == null) { //没有跳转,则当前指令和下一指令连一条边
                if (node.tail != null) addEdge(node.head, node.tail.head);
            } else //否则当前指令和所有跳转标号连边
                for (LabelList l = next.labels; l != null; l = l.tail)
                    addEdge(node.head, (Node) labels.get(l.head));
        }
    }
    //返回某结点表示的汇编指令
    public Instr instr(Node n) {return (Instr) represent.get(n);}
```

```
    //返回某结点定义的变量
    public TempList def(Node node) {return instr(node).def();}
    //返回某结点使用的变量
    public TempList use(Node node) {return instr(node).use();}
    //是否是 move 指令
    public boolean isMove(Node node) {
        Instr instr = instr(node);
        return instr.assem.startsWith("move");
    }
}
```

### 9.3 活性分析

第二步是进行活性分析

对于流图中的每个结点,用下面的结构(四个集合)来描述它的结点信息:

```
class NodeInfo {
    Set in = new HashSet();     //来指令前的活性变量
    Set out = new HashSet();    //出指令后的活性变量 — 即活跃变量
    Set use = new HashSet();    //某指令使用的变量 - 赋值号右边
    Set def = new HashSet();    //某指令定义的变量 - 赋值号左边
}
```

所谓活性分析,就是求出每个结点的 out 变量,即活跃变量

按下面的步骤求:

首先,对每个结点的 use 和 def 初始化,这在 flowgraph 中已经提供,并且置 in 和 out 为空,在 RegAlloc.Liveness.initNodeInfo 中实现

然后按下面的步骤不断迭代,直到 in 和 out 不变为止

$$in[n] = use[n] \bigcup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \in succ[n]} (in[s])$$

最后将生成的活跃变量记录到活跃变量表中.

以上两步在 RegAlloc.Liveness. calculateLiveness 中实现

```
//计算活性 live-in 和 live-out
//数据流等式:
// in[n]=use[n] U (out[n]-def[n])
//out[n]=U (in[s]) , for each s in succ[n]
//反复迭代,直到不变
    void calculateLiveness() {
        boolean done = false;
        do {
            done = true;
            for(NodeList node=flowGraph.nodes();node != null; node=node.tail) {
                //遍历流图所有指令
```

```
                NodeInfo inf = (NodeInfo) info.get(node.head);//更新前的活性信息

                //等式 1
                Set in1 = new HashSet(inf.out);
                in1.removeAll(inf.def);
                in1.addAll(inf.use);
                if (!in1.equals(inf.in)) done = false; //测试是否完成
                inf.in = in1; //更新 in

                //等式 2
                Set out1 = new HashSet();
                for (NodeList succ = node.head.succ(); succ != null; succ =
                succ.tail) {
                    NodeInfo i = (NodeInfo) info.get(succ.head);
                    out1.addAll(i.in);
                }
                if (!out1.equals(inf.out)) done = false; //测试是否完成
                inf.out = out1; //更新 out
            }
        } while (!done);

        //生成 liveMap
        for (NodeList node = flowGraph.nodes();node != null;node = node.tail) {
            TempList list = null;
            //得到活性信息中活跃变量的迭代器
            Iterator i = ((NodeInfo) info.get(node.head)).out.iterator();
            while (i.hasNext()) list = new TempList((Temp) i.next(), list);
            if (list != null) liveMap.put(node.head, list);
        }
    }
```

### 9.4 干扰图

第三步是生成干扰图.

考察控制和数据流图,可以画出一张干扰图.干扰图中的每个结点代表临时变量的值,每条边(t1,t2)代表了一对不能分配到同一个寄存器中的临时变量.它的抽象接口在 RegAlloc.InterferenceGraph 中实现

生成干扰图的算法如下:

a 在任何定义变量 a 且没有转移的指令中,其中非活跃变量包括 b1,b2…bj, 添加干扰边(a,b1),…(a,bj)

b 在转移指令 a←c 中,其中非活跃变量为 b1,…,bj,为每一个与 c 不同的 bi 添加干扰边(a,b1),…(a,bj) (即 move 指令要特殊考虑)

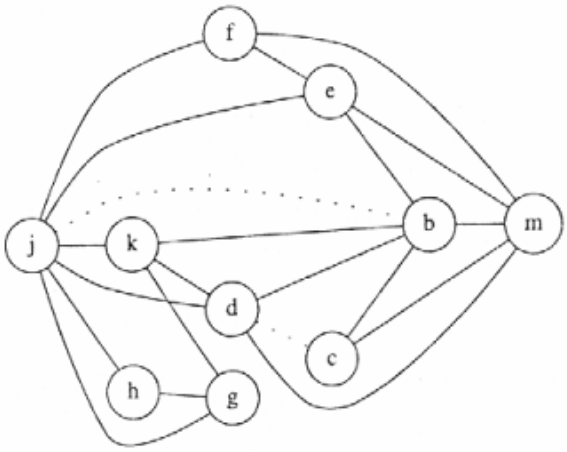
以上步骤在 RegAlloc.Liveness.buildGraph 中实现

```
    //生成干扰图
```

```
    void buildGraph() {
        Set temps = new HashSet(); //包含流图中所有有关的变量(使用的和定义的)
        //生成 temps
        for (NodeList node = flowGraph.nodes(); node != null; node = node.tail)
{
            for (TempList t = flowGraph.use(node.head); t != null; t = t.tail)
                temps.add(t.head);
            for (TempList t = flowGraph.def(node.head); t != null; t = t.tail)
                temps.add(t.head);
        }

        //生成 tnode
        Iterator i = temps.iterator();
        while (i.hasNext()) add(newNode(), (Temp) i.next());

        //生成干扰图
        for (NodeList node = flowGraph.nodes(); node != null; node = node.tail)
        //遍历流图中每一条指令
            for (TempList t = flowGraph.def(node.head); t != null; t = t.tail)

            //遍历指令中定义的所有变量
                for (TempList t1 = (TempList) liveMap.get(node.head); t1 != null;
t1 = t1.tail)
                //遍历指令的所有活跃变量
                    //应用以上规则 a,b,尝试添加边
                    if (t.head != t1.head //防止自回路
                            && !(flowGraph.isMove(node.head)
                            && flowGraph.use(node.head).head == t1.head)) {
                        addEdge(tnode(t.head), tnode(t1.head));
                        addEdge(tnode(t1.head), tnode(t.head)); //无向图,双向加
边
                    }
    }
```

下面是一个例子:

```
live-in: k j
        g := mem[j+12]
        h := k - 1
        f := g * h
        e := mem[j+8]
        m := mem[j+16]
        b := mem[f]
        c := e + 8
        d := c
        k := m + 4
        j := b
live-out: d k j
```




### 9.5 寄存器分配

第四步是根据干扰图分配寄存器.

这其实是根据干扰图进行的一个着色 (coloring) 过程－使干扰图中相邻的两个结点着不同的颜色.它是 NPC 问题.这里采用近似的贪心算法如下:

1 扫描干扰图中的每个结点, 把已经分配好寄存器的变量推入堆栈, 并删除从这些结点出发的边. (注意这时应把原来无向的干扰图看成有向的, 即 a→b 和 b→a 不是同一条边.)

2 再扫描剩下结点 (设为 n 个), 它们都是还没有被分配寄存器的 (不在堆栈中). 每次扫描找出一个出度最大且小于寄存器数目的结点. 并删除那些不在堆栈中的结点指向这个结点的边. 然后把这个结点推入堆栈. 重复上述过程, 直到堆栈中装满了结点. 如果某次扫描找不到这样的结点, 说明寄存器溢出 (spill), 报错. (这里不进行额外处理了)

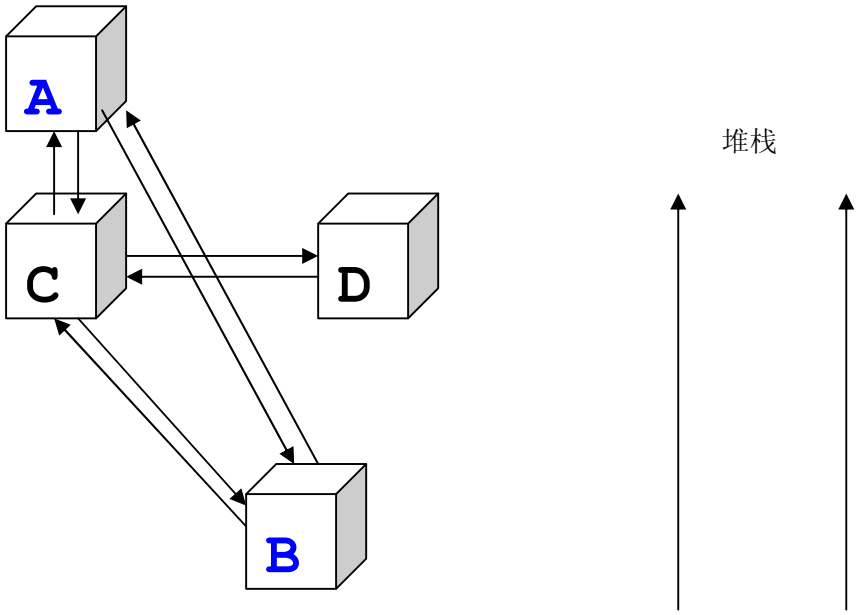
3 接下来为那 n 个还没有分配寄存器的结点分配寄存器 (它们处在栈顶). 对弹出堆栈的某个结点 node, 先设集合 available 为所有可供分配的寄存器, 然后对于在当前干扰图中每个条从 node 出发到某个结点 node1 的边, 从 available 中删除 node1 所代表的寄存器. 最后, 再取剩下寄存器中的一个为 node 分配.

以下代码描述了这个过程:

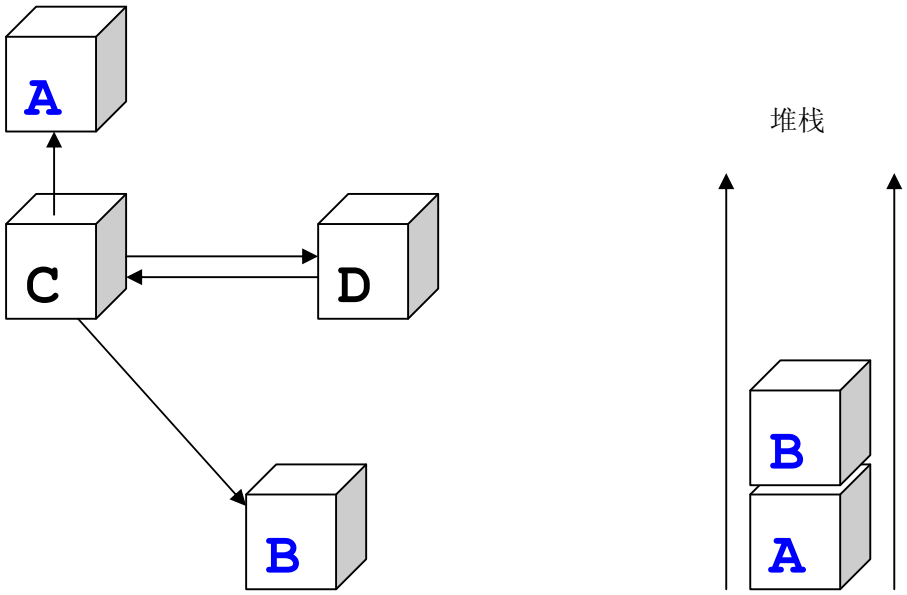
```
    void color() {
        int number = 0;
        //遍历每个临时变量结点
        for (NodeList node=interGraph.nodes(); node != null;node = node.tail) {
            ++number;
            //得到结点所对应的临时变量 temp
            Temp temp = interGraph.gtemp(node.head);
            //如果 temp 已经被分配了寄存器
            if (init.tempMap(temp) != null) {
                --number;
                pushNode(node.head); //将改结点压入堆栈
                map.put(temp, temp); //放入分配列表 map 中，它们的寄存器就是本身了
                //删除从该结点出发的所有边
                for (NodeList adj = node.head.succ(); adj != null; adj = adj.tail)
```

```
                    interGraph.rmEdge(node.head, adj.head);
                }
            }
            //剩下是 number 个还没有被分配寄存器的
            for (int i = 0; i < number; ++i) {
                Node node = null;
                int max = -1;
                //再次遍历每个临时变量结点
                for (NodeList n = interGraph.nodes(); n != null; n = n.tail)
                    if (init.tempMap(interGraph.gtemp(n.head)) == null
                            && !isInStack(n.head)) { //没有被分配寄存器且不在堆栈中
                        int num = n.head.outDegree(); //出度
                        if (max < num && num < regs.size()) {
                        //找到一个度最大的且小于寄存器数目的结点
                            max = num;
                            node = n.head;
                        }
                    }
                if (node == null) { //度大于等于寄存器数目，溢出
                    System.err.println("Color.color() : register spills.");
                    break;
                }
                //否则继续推入堆栈并移去从不在堆栈中的结点指向该结点的所有边
                pushNode(node);
                for (NodeList adj = node.pred(); adj != null; adj = adj.tail)
                    if (!isInStack(adj.head))
                        interGraph.rmEdge(adj.head, node);
            }
            //接下来开始分配那 number 个没有分配寄存器的临时变量，它们处在栈顶
            for (int i = 0; i < number; ++i) {
                Node node = popNode(); //弹出一个
                Set available = new HashSet(regs); //可供分配寄存器列表
                for (NodeList adj = node.succ(); adj != null; adj = adj.tail) {
                    //从可供分配寄存器列表中移除该结点指向的某个结点所代表的寄存器
                    available.remove(map.get(interGraph.gtemp(adj.head)));
                }
                //取剩下的一个作为寄存器
                Temp reg = (Temp) available.iterator().next();
                //加入寄存器分配表
                map.put(interGraph.gtemp(node), reg);
            }
        }
```

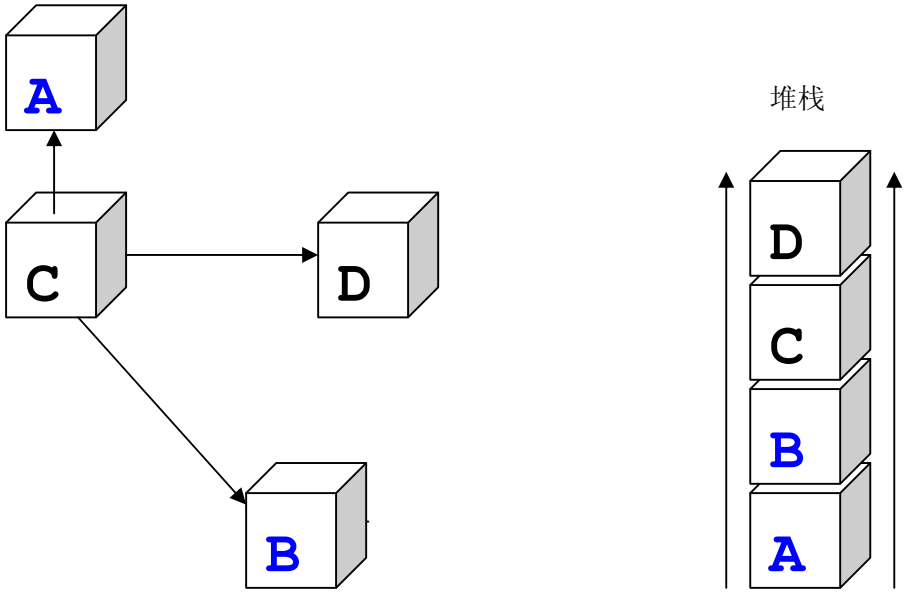
下面是一个例子: 开始 A-R1, B-R2




经过步骤 1 后：A-R1, B-R2




经过步骤 2 后：A-R1, B-R2



为变量 D 分配寄存器:
Available = {R1, R2, R3} → {R1, R2, R3} (当前干扰图中没有从 D 出发的边)
为 D 分配 R1
为变量 C 分配寄存器
Available = {R1, R2, R3} → {R2, R3} (边 CA) → {R2, R3} (边 CD) → {R3} (边 CB)
为 C 分配 R3
最后分配结果: A-R1, B-R2, C-R3, D-R1

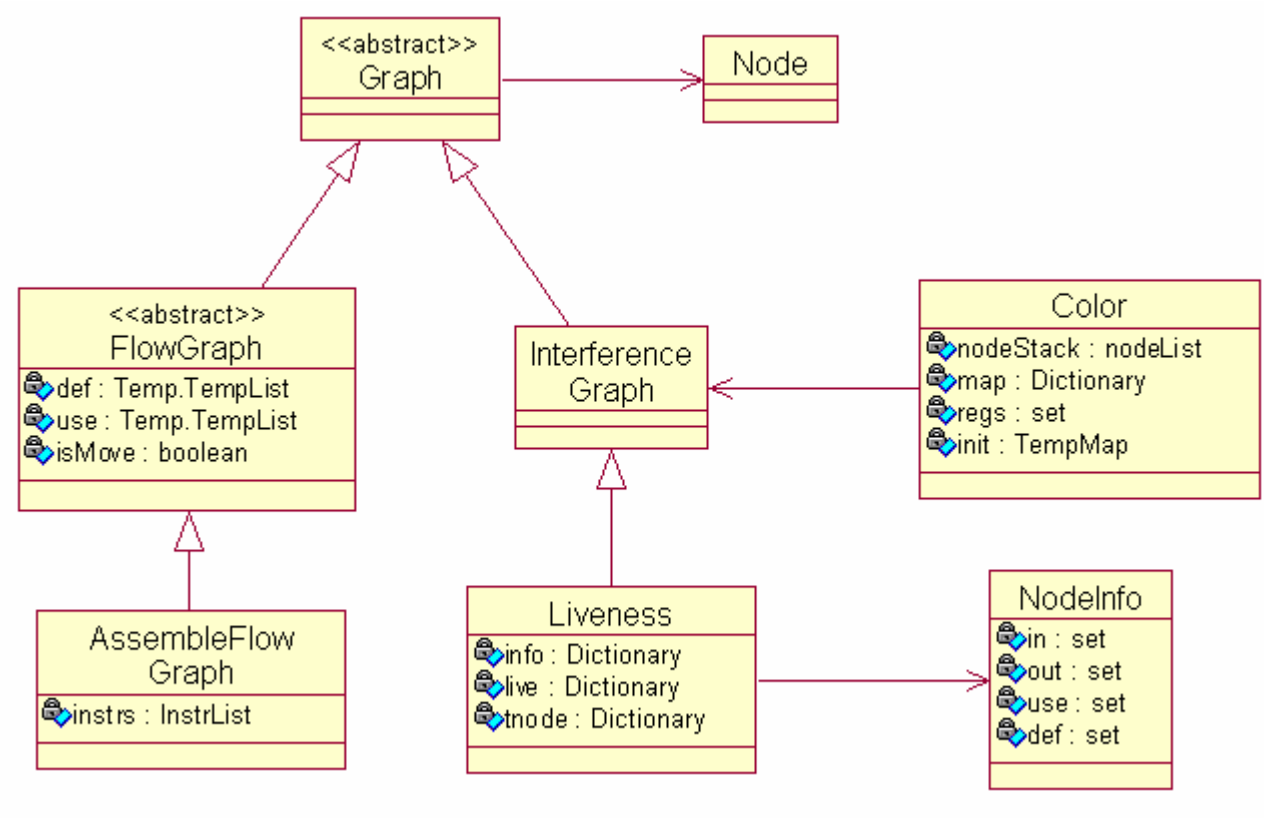
以上步骤在 RegAlloc. Color.color 中实现

### 9.6 把以上步骤连起来

RegAlloc.RegAlloc 提供了以上步骤 (9.2 到 9.5) 的一个接口如下:

```
    public RegAlloc(Frame f, InstrList instrs) {
        this.instrs = instrs;
        FlowGraph flowGraph = new AssemFlowGraph(instrs);//根据汇编指令生成流图
        InterferenceGraph interGraph=new Liveness(flowGraph);//活性分析,干扰图
        color = new Color(interGraph, f, f.registers());//着色法分配寄存器
    }

    //RegAlloc.Liveness 的对外接口,输入流图,进行活性分析,并输出干扰图
    public Liveness(FlowGraph flowGraph) {
        this.flowGraph = flowGraph;
        initNodeInfo();          //初始化
        calculateLiveness();     //计算活性变量
        buildGraph();            //生成干扰图
    }
```

### 9.7 本部分中的类关系图


<<abstract>>
Graph
Node
<<abstract>>
FlowGraph
def : Temp.TempList
use : Temp.TempList
isMove : boolean
Interference
Graph
Color
nodeStack : nodeList
map : Dictionary
regs : set
init : TempMap
AssembleFlow
Graph
instrs : InstrList
Liveness
info : Dictionary
live : Dictionary
tnode : Dictionary
NodeInfo
in : set
out : set
use : set
def : set

# 第 10 部分 使成为整体

## 10.1 错误报告

ErrorMsg.ErrorMsg 封装了编译器的错误报告机制,它是一个链式系统.调用 error (pos, msg) 能输出出错位置和信息

对于严重的错误,直接抛出异常,终止程序

## 10.2 编译器的输入和输出

输入文件:tiger 语言源文件,文件名为 inFilename,它传入主程序的第一个参数中

输出文件:

astOut – 抽象语法树 (inFilename.abs)

irOut – IR 树 (inFilename.ir)

out – MIPS 指令文件 (inFilename.s)

## 10.3 编译全过程

编译器入口:Main.java 的 Main 函数

*a 词法分析、语法分析和构造抽象语法树*

Main 函数中:

```
Parse parse = new Parse(inFilename);
```

进而调用 Parse 的构造函数中:

```
//Yylex 执行词法分析
//Grm 执行语法分析
Grm parser = new Grm(new Yylex(inp, errorMsg), errorMsg);
parser.parse();
//返回抽象语法树的根
absyn = parser.parseResult;
```

回到 Main 函数:

```
//第一个输出－抽象语法树
new Absyn.Print(astOut).prExp(parse.absyn, 0);
```

*b 语义分析和 IR 树构造*

在 Main 函数中新建一个 frame

```
static Frame frame = new MipsFrame();
```

根据这个 frame 创建 translate 对象

再根据 translate 对象创建 semant 对象,注意出错信息的传递

```
Semant semant = new Semant(translate, parse.errorMsg);
```

语义检查是和 IR 树的翻译绑定在一起的,由下面的代码驱动:

```
Frag frags = semant.transProg(parse.absyn);
```

在 semant.transProg 函数中:

```
public Frag transProg(Exp e) {
    level = new Level(level, Symbol.symbol("main"), null); //新建层－作为 main
函数
```

```
        //从这里开始正式翻译主函数,进而递归完成整个程序的翻译 (包括语义和 IR 树)
        ExpTy et = e.translate(this);
        translator. procEntryExit (level, et.exp, false); //添加函数调用部分代码
        level = level.parent;
        return translator.getResult(); //返回翻译完的段链表
    }
```

*c 规范化、指令生成、数据流分析、寄存器分配等*

先向 out 输出汇编文件头 ".globl main"

然后 Main 函数检查每个段,如果是数据段,则向 out 输出 ".data" 和数据信息

如果是函数段,则进入 emitProc 函数

```
    out.println(".globl main");
    for (Frag f = frags; f != null; f = f.next)
        if (f instanceof ProcFrag)
            emitProc((ProcFrag) f);
        else if (f instanceof DataFrag)
            out.println(".data\n" + ((DataFrag) f).data);
```

在 emitProc 函数中:

```
    //根据函数段输出该函数段的 IR 树和汇编指令
    static void emitProc(ProcFrag f) {
        //输出 IR 树
        Tree.Print print = new Tree.Print(irOut);
        irOut.println("function " + f.frame.name);
        print.prStm(f.body);

        //规范化
        //IR 树被写成一个没有 SEQ 和 ESEQ 结点的规范树表
        Tree.StmList stms = Canon.Canon.linearize(f.body);

        //根据该表划分基本块,每个基本块中不包含内部跳转和标号
        Canon.BasicBlocks b = new Canon.BasicBlocks(stms);

        //基本块被顺序放置,所有的 CJUMP 都跟有 false 标号
        Tree.StmList traced = (new Canon.TraceSchedule(b)).stms;

        //生成汇编代码
        Assem.InstrList instrs = codegen(f.frame, traced);
        instrs = frame.procEntryExit2(instrs);

        //寄存器分配
        //这一步具体 3 步见 9.6
        RegAlloc regAlloc = new RegAlloc(f.frame, instrs);
```

```
        //添加函数调用和返回的代码
        instrs = f.frame.procEntryExit3(instrs);
        Temp.TempMap tempmap = new Temp.CombineMap(f.frame, regAlloc);

        //输出 Mips 指令
        out.println(".text");
        for (Assem.InstrList p = instrs; p != null; p = p.tail)
            out.println(p.head.format(tempmap));
    }
```

*d 追加标准函数库*

将标准函数库 (runtime.s) 的汇编代码添加到生成好的汇编源文件尾部即可

## 10.4 函数声明、调用的翻译

这部分比较复杂,单独列出.请先参考第五部分:活动记录的相关内容

考虑下面的 Tiger 程序,它计算 1+2 的值

```
let
/* calculate a+b */
function calc(a: int, b:int): int =
    a+b
in
    calc(1,2)
end
```

首先,是并不困难的词法、文法分析和构建抽象语法树.

其次,在语义分析阶段:

函数声明的参数列表 a:int, b:int 被翻译成 RECORD 类型 (a, int) → (b, int) 类型为 (Symbol.Symbol, Types.Type)

返回类型被翻译成 Types.Type 类型的 int

以上在 Semant.transDec (FunctionDec) 中进行,其中的各种检查都未发现错误

然后进入 IR 树翻译阶段,:

Semant 在当前层 (main 函数) 上建立一个新层

在 Level 的构造函数中:

参数列表的开头被增加了一个参数, 用于存放静态链接

Level 的构造函数为自己建造一个新帧

在建造新帧的过程 newFrame 中:

为帧创建一个 AccessList (formals)

产生把参数放入 frame.Access 的汇编指令

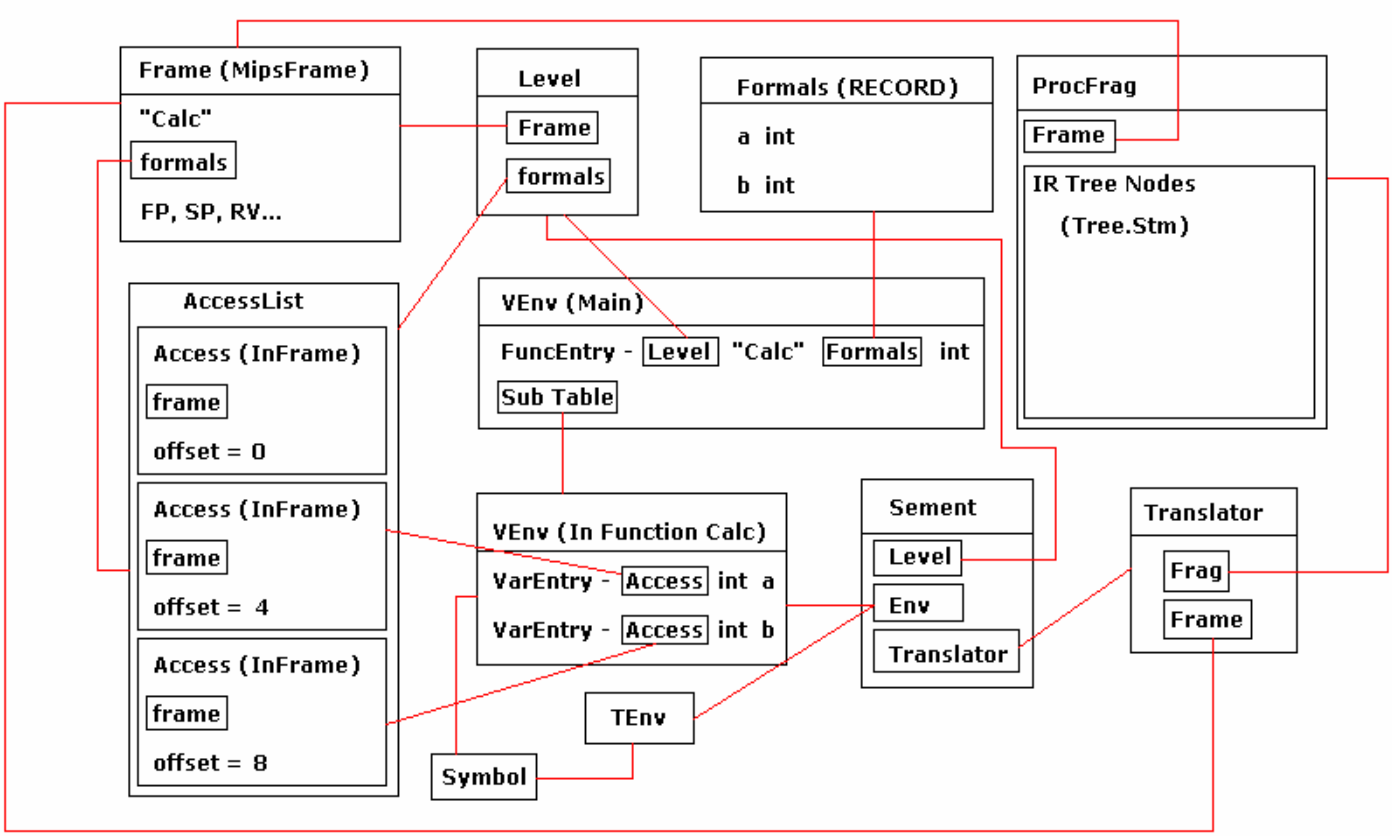
回到 Level 的构造函数,构建一个与 frame 完全相同的 AccessList (formals)

回到 Semant.transDec (FunctionDec) 中,将刚才建好的 Level 添加到 vEnv 中

然后用 BeginScope 打开新的 vEnv 符号表

将函数的参数放入新符号表中

翻译函数体 (语义检查+IR 树生成)
然后调用 procEntryExit 方法给函数加上重要的保存/恢复 FP、Callee 寄存器等指令
把完整的函数指令和帧放入段中
用 EndScope 结束当前 vEnv 符号表
最后通过规范化、指令选择、寄存器分配等步骤,完成最后的翻译
下面是中间部分的对象图




下面看函数调用部分: 翻译 calc (1,2)
首先在 Semant.java 中:
找出函数在 vEnv 中的入口, 得到函数的层 (Level 类型)、名称 (Label 类型)、参数(RECORD 类型转 ArrayList 类型)、返回值类型 (Type 类型).
然后以这些参数调用 transCallExp

在 Translate 的 transCallExp 函数中:
先逐个翻译参数成 Tree.Expr 类型, 将参数连接成 Tree.ExpList 类型的链表
然后找到 calc 函数上层的静态链接, 将它作为第一个参数隐式地加入参数表
最后返回 IR 树的 CALL 结点, 它有如下子结点: Tree.NAME 作为函数名, Tree.ExpList 的 Args 作为函数参数

最后同样通过规范化、指令选择、寄存器分配等步骤,完成最后的翻译

## 10.5 汇编源程序在虚拟机上的运行

使用工具 PCSpim, 它是在 Windows 环境下模拟 Mips 机的软件
打开编译器生成的 .s 文件后,在 Simulator 菜单中选择 Go 即可运行

# 第 11 部分 FAQ

这里汇集了一些问题, 以加深对项目的理解

**Q: 你认为抽象语法树的作用是什么?它是自然需要完成的,还是人为地需要完成?**

**A:** 对一个合法的程序,抽象语法树清晰且唯一地表示表示了程序的语法结构,用树这种数据结构表示是很显然的事情.事实上,几乎所有的编译器都用到了这种技术.它为以后 IR 树的翻译提供了良好的接口—抽象语法树结点与 IR 树结点间几乎可以建立起映射关系.

**Q: 你能为语义分析与中间代码的翻译找到明确的界限吗?这里你认为还有什么需要讨论的地方?**

**A:** 语义分析是以抽象语法树为输入进行出错检查.几乎所有的编译错误(如类型不匹配,变量未定义等等)都能在语义分析中被发现.而中间代码的翻译是在正确语义的基础上将抽象语法树结点翻译成 IR 树结点.

在具体实现时语义分析部分在 Semant 类中完成,翻译在 Translate 类中完成.它们存在明显的区别, 是编译过程中的两个不同的阶段.但由于树的递归性质,它们在实现时是交替进行的.例如下面的代码用于检查算术运算表达式中的错误,这里没有与 IR 树有关的具体代码:

```
//Semant.java
public ExpTy transExp(CalcExp e) {
    ExpTy left = e.left.translate(this);    //左边的检查的翻译
    ExpTy right = e.right.translate(this); //右边的检查和翻译
    if (!isBothInt(left.ty, right.ty))      //语义分析:两边都要是整数
        error(e.left.pos, "ERROR: Both Integers are required!");
    //中间代码翻译
    Translate.Exp ex = translator.transCalcExp(e.oper, left.exp, right.exp);
    return new ExpTy(ex, Type._int);
}
```

而下面的代码来自于 translate.java,它仅负责翻译中间代码:

```
//translate.java
public Exp transCalcExp(int oper, Exp left, Exp right) {
    int binOp = 0;
    switch (oper) {
    case CalcExp.PLUS:
        binOp = BINOP.PLUS;
        break;
    case …
    //IR 树结点群
    return new Ex(new BINOP(binOp, left.unEx(), right.unEx()));
}
```

**Q: 你是怎么理解语义分析和中间代码翻译的?它们能被简化吗?**

**A:** 两者的不同作用在上一问中已经给出.由于它们的不同点, 将它们加以区分体现了结构化程序设计的模块化思想和面向对象程序设计的封装思想.在工业界,可以让不同的专家单独

负责某块.而如果将它们简化成一个阶段,可能会变得复杂和混乱.

**Q: 为什么只有中间树还不能足够地生成汇编代码?**

**A:** 有以下原因:

1 中间树是不面向任何具体机器的.中间树可以适应运行在不同机器 (如 MIPS,Pentium, AMD)上的指令集.因此,为了生成具体的汇编代码, 还需要知道具体指令集的细节.例如对于简单的中间树结点 CALL,在不同机器上的实现方式有可能不同

2 中间树默认有无穷多个寄存器可供使用.而实际的机器并不是这样.因此还需要经过寄存器分配

3 中间树仅提供了一般处理器所共有的指令,并没有提供某些特殊的优化指令(如 MMX 指令),因此还有优化的空间

**Q: 如果不允许有用户自定义类型,即 Tiger 语言仅支持整型与字符串两种类型,那么编译器的哪些部分需要改动?如何改动?**

**A:** 词法分析阶段:有关用户类型的关键字 (如 Type 等) 需要移除.有限自动机将会变得简单

语法分析阶段:有关用户类型的文法(如 FieldList)也将被删除,抽象语法树的结点数将减少

翻译成中间代码阶段:有关用户类型的抽象语法树结点将不再被翻译

以后阶段(指令生成、寄存器分配等)将不受影响.因为它们属于编译器的后端,不依赖高级语言

**Q: 如果不允许函数的嵌套,即所有的函数都在同一层中(如 C 和 java),那么编译器的哪些部分需要改动?如何改动?**

**A:** 词法、语法分析阶段:有关函数嵌套的关键字和文法将被删除,抽象语法树将变得简单

语义分析阶段:如果函数嵌套,编译器将报错

最重要的改变来自于中间代码翻译阶段:Level 类将不再有用,因为所有函数的静态链接都是 main 函数.此外,这样还有可能增加全局变量的概念,那么需要代码对全局变量进行管理—或许需要一张全局量符号表和在帧中增加其指针

以后阶段(指令生成、寄存器分配等)将不受影响.因为它们属于编译器的后端,不依赖高级语言

**Q: 请估计你的编译器中各部分的时间复杂度:**

**A:** 词法分析:线性时间(编译器对程序进行线性扫描)

语法分析:线性时间(每个抽象语法树结点只会被处理一次)

语义分析:线性时间(同上)

中间代码:线性时间(每个 IR 树结点只会被处理一次)

指令选择:线性时间(同上)

生成流图:不会超过 N*J,其中 N 为指令数,J 为每条指令跳转的最多分支数.由于 J 是常数,所以它也是线性时间的

计算 Live-in 和 Live-out:N*D,其中 N 为指令数,D 为数据流公式迭代次数

生成干扰图:N 平方,其中 N 为指令数.从干扰图的生成算法中可以看出

寄存器分配:N 平方,其中 N 为寄存器数.从寄存器分配算法中可以看出

**Q: 考虑在其它平台(如 Pentium 和 AMD)上运行 Tiger 程序,你的编译器哪些部分需要修改?**

**给出理由.**

**A:** 编译器的前端(词法、文法、语义、中间表示)不需要修改.需要修改 Codegen,它与指令的生成相关;派生自 Frame.Frame 的 MipsFrame 也需要修改,因为帧格式可能有所不同.

**Q: 寄存器个数与寄存器分配算法有关系吗?换句话说,是不是对于寄存器较少的平台需要复杂的寄存器分配算法而对于寄存器较多的平台仅需要简单的算法?给出理由.**

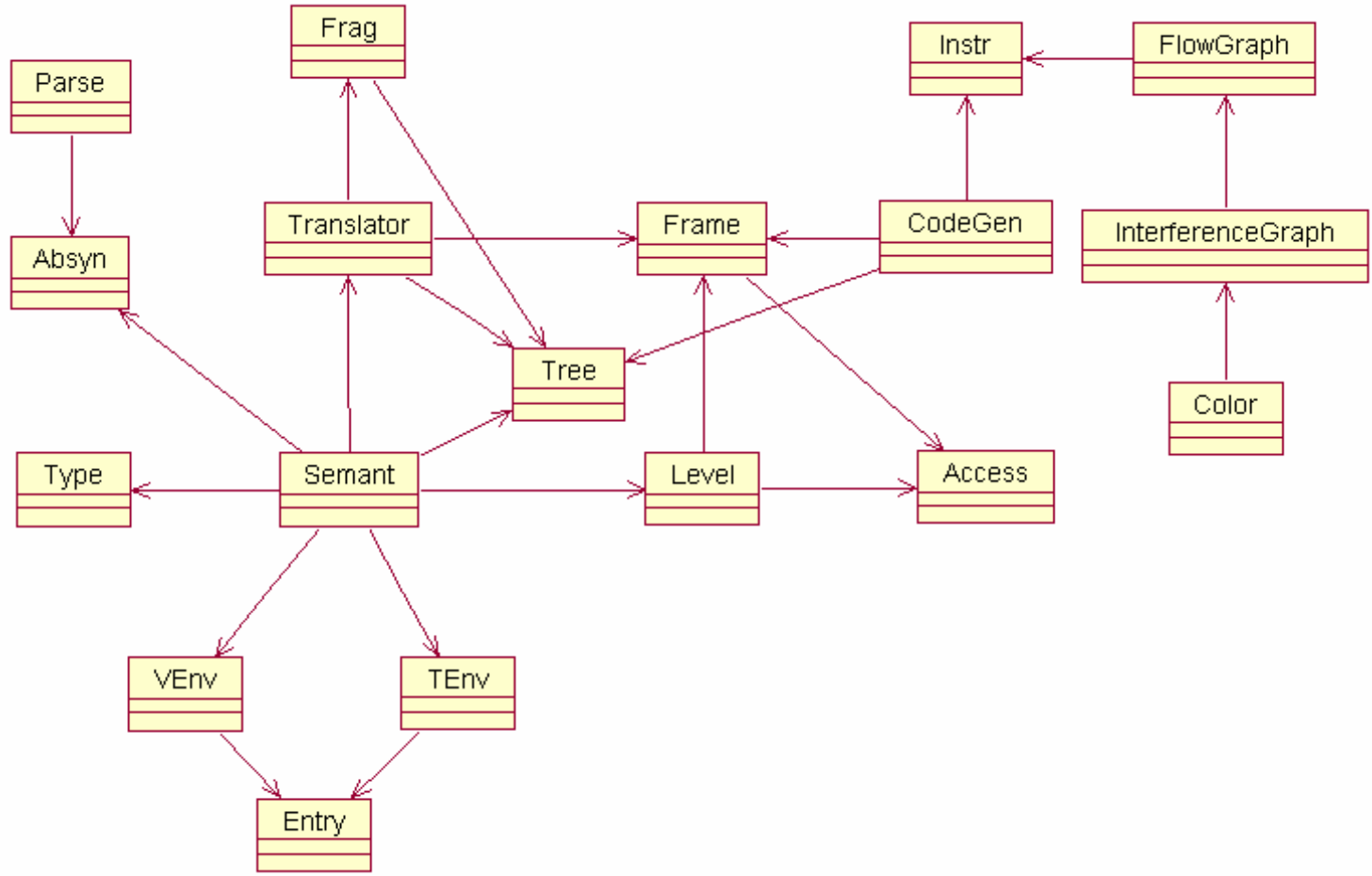
**A:** 两者之间没有必然联系.优秀的寄存器分配算法可以使用也应该使用于任何平台.例如,尽管某种机器拥有大量寄存器,但运行于它上的程序可能也非常复杂,如果不优化算法的话,同样可能出现寄存器溢出的现象.

问这个问题就是好比在问"现在的计算机的运算速度大大加快就不需要好的算法吗?"答案当然是否定的.算法课上学过的各种排序算法的比较就是一个典型的例子.

# 第 12 部分 附录

## 12.1 主要类图

下面给出本工程的重要类之间的关系图 (仅给出类名,省略属性和方法) .详细类图请参阅附件的 Rational Rose 模型




## 12.2 压缩包内容

```
tiger.bat                    //启动编译器
readme.txt                   //说明文件
<CLASS>                      //编译后的 java 字节码
<DOC>                        //作业文档
<STEPBYSTEP>                 //本文档
<UML>                        //UML 模型, 用 Rational Rose 打开
<SRC>                        //原代码
    Main.java                //主程序, 编译程序入口
    <ABSYN>                  //抽象语法树结点
    <SYMBOL>
        Symbol.java          //抽象符号
        Table.java           //抽象符号表
    <PARSE>
```

```
    Grm.java                //文法分析器,由 CUP 生成
    Sym.java                //文法分析器常数表,由 CUP 生成
    Yylex.java              //词法分析器,由 Jflex 生成
    Parse.java              //词法、文法分析主程序
    Lexer.java              //词法分析器接口
<TREE>                      //IR 树结点和输出
<TYPES>                     //类型信息
<UTIL>
    BoolList.java           //布尔链表
<GRAPH>
    Graph.java              //抽象图
    Node.java               //抽象结点
    NodeList.java           //抽象结点列表
<JAVA_CUP>                  //java_cup
<ERRORMSG>
    ErrorMsg.java           //出错信息处理
<ASSEM>                     //抽象汇编指令
<SEMANT>
    Semant.java             //语义分析器
    *Entry.java             //各种入口类,用于符号表中
    Env.java                //符号表
    ExpTy.java              //表达式的 IR 树翻译结果
<TRANSLATE>
    Translate.java          //IR 树翻译器
    *Frag.java              //段
    Level.java              //层
    Access.java             //Access (带上 Frame)
    AccessList.java         //Access 列表
    其它文件                //宏观 IR 树结点
<FLOWGRAPH>
    FlowGraph.java          //抽象流图
    AssemFlowGraph.java     //汇编指令流图
<CANON>                     //规范化
<REGALLOC>
    Color.java              //寄存器分配
    InterferenceGraph.java  //干扰图
    Liveness.java           //活性分析
    Movelist.java           //Move 指令链表
    RegAlloc.java           //RegAlloc 包对外接口
<FRAME>
    Frame.java              //抽象帧
    Access.java             //抽象的 Access
    AccessList.java         //抽象的 Access 链表
<MIPS>
```

```
        CodeGen.java            //MIPS 汇编指令生成器
        InFrame.java            //MIPS 机存放于帧中的 Access
        InReg.java              //MIPS 机存放于寄存器中的 Access
        MipsFrame.java          //MIPS 机的 Frame
    <TEMP>
        *Map.java               //映射表
        Label*.java             //标号和标号链表
        Temp*.java              //临时变量（寄存器）和临时变量链表
```

### 12.3 编译器使用方法

在根目录下输入：

```
java -cp class; Main 文件名.tig
```

在同一目录下产生三个输出文件

也可以用批处理方式：Tiger 文件名.tig

### 12.4 一个范例

一个很简单的 Tiger 程序:

```
/* test.tig */
let
    var a:=1
    var b:=2
in
    a+b
end
```

抽象语法树 test.abs:

```
LetExp(
DecList(
 VarDec(a,
  IntExp(1),
  true),
 DecList(
  VarDec(b,
   IntExp(2),
   true),
  DecList())),
SeqExp(
 ExpList(
  CalcExp(
   PLUS,
   varExp(
    SimpleVar(a)),
```

```
   varExp(
    SimpleVar(b))))))
```

IR 树 test.ir

```
function main
SEQ(
 SEQ(
  SEQ(
   SEQ(
    SEQ(
     SEQ(
      SEQ(
       SEQ(
        SEQ(
         SEQ(
          MOVE(
           MEM(
            BINOP(PLUS,
             TEMP t1,
             CONST 36)),
           TEMP t0),
          SEQ(
           MOVE(
            TEMP t0,
            BINOP(PLUS,
             TEMP t1,
             CONST 48)),
           SEQ(
            MOVE(
             MEM(
              BINOP(PLUS,
               TEMP t0,
               CONST -16)),
             TEMP t2),
            SEQ(
             MOVE(
              MEM(
               BINOP(PLUS,
                TEMP t0,
                CONST -48)),
              TEMP t19),
             SEQ(
              MOVE(
               MEM(
                BINOP(PLUS,
                 TEMP t0,
                 CONST -44)),
               TEMP t20),
              SEQ(
               MOVE(
                MEM(
                 BINOP(PLUS,
                  TEMP t0,
                  CONST -40)),
                TEMP t21),
               SEQ(
                MOVE(
                 MEM(
                  BINOP(PLUS,
                   TEMP t0,
                   CONST -36)),
                 TEMP t22),
                SEQ(
                 MOVE(
                  MEM(
                   BINOP(PLUS,
                    TEMP t0,
                    CONST -32)),
                  TEMP t23),
                 SEQ(
                  MOVE(
                   MEM(
                    BINOP(PLUS,
                     TEMP t0,
                     CONST -28)),
                   TEMP t24),
                  SEQ(
                   MOVE(
                    MEM(
                     BINOP(PLUS,
                      TEMP t0,
                      CONST -24)),
                    TEMP t25),
                   SEQ(
                    MOVE(
                     MEM(
                      BINOP(PLUS,
                       TEMP t0,
                       CONST -20)),
                     TEMP t26),
                    SEQ(
                     MOVE(
                      MEM(
                       BINOP(PLUS,
                        TEMP t0,
                        CONST 0)),
                      TEMP t8),
                     EXP(
                      ESEQ(
                       SEQ(
                        MOVE(
                         MEM(
                          BINOP(PLUS,
                           TEMP t0,
                           CONST -4)),
                         CONST 1),
                        MOVE(
                         MEM(
                          BINOP(PLUS,
```

```
                          TEMP t0,
                          CONST -8)),
                        CONST 2)),
                      BINOP(PLUS,
                      MEM(
                       BINOP(PLUS,
                       TEMP t0,
                       CONST -4)),
                      MEM(
                       BINOP(PLUS,
                       TEMP t0,
                       CONST
-8))))))))))))))))),
         MOVE(
          TEMP t26,
          MEM(
           BINOP(PLUS,
            TEMP t0,
            CONST -20)))),
        MOVE(
         TEMP t25,
         MEM(
          BINOP(PLUS,
           TEMP t0,
           CONST -24)))),
       MOVE(
        TEMP t24,
        MEM(
         BINOP(PLUS,
          TEMP t0,
          CONST -28)))),
      MOVE(
       TEMP t23,
       MEM(
        BINOP(PLUS,
         TEMP t0,
         CONST -32)))),
     MOVE(
      TEMP t22,
      MEM(
       BINOP(PLUS,
        TEMP t0,
        CONST -36)))),
    MOVE(
     TEMP t21,
     MEM(
      BINOP(PLUS,
       TEMP t0,
       CONST -40)))),
   MOVE(
    TEMP t20,
    MEM(
     BINOP(PLUS,
      TEMP t0,
      CONST -44)))),
  MOVE(
   TEMP t19,
   MEM(
    BINOP(PLUS,
     TEMP t0,
     CONST -48)))),
 MOVE(
  TEMP t2,
  MEM(
   BINOP(PLUS,
    TEMP t0,
    CONST -16)))),
MOVE(
 TEMP t0,
 MEM(
  BINOP(PLUS,
   TEMP t1,
   CONST 36))))
```

Mips 汇编程序 test.s:

```
    .globl main
.text
main:
subu $sp, $sp, 52
L1:
sw $fp, 36($sp)
add $t2, $sp, 48
move $fp, $t2
sw $ra, -16($fp)
sw $s0, -48($fp)
sw $s1, -44($fp)
sw $s2, -40($fp)
sw $s3, -36($fp)
sw $s4, -32($fp)
sw $s5, -28($fp)
sw $s6, -24($fp)
sw $s7, -20($fp)
sw $a0, 0($fp)
li $s4, 1
sw $s4, -4($fp)
li $s4, 2
sw $s4, -8($fp)
lw $t2, -4($fp)
lw $s4, -8($fp)
add $s4, $t2, $s4
lw $s4, -20($fp)
move $s7, $s4
lw $s4, -24($fp)
move $s6, $s4
lw $s4, -28($fp)
move $s5, $s4
lw $s4, -32($fp)
move $s4, $s4
lw $t2, -36($fp)
move $s3, $t2
lw $t2, -40($fp)
move $s2, $t2
lw $t2, -44($fp)
move $s1, $t2
lw $t2, -48($fp)
move $s0, $t2
lw $t2, -16($fp)
```

```
move $ra, $t2
lw $t2, 36($sp)
move $fp, $t2
j L0
L0:
addu $sp, $sp, 52
jr $ra
```

## 12.5 参考资料和工具

**资料:**

《现代编译器的 Java 实现》Modern Compiler Implementation in Java 2nd Edition

Assemblers, Linkers, and the SPIM Simulator

The MIPS Info Sheet

Tiger Language Reference Manual

**工具:**

JDK 1.5.0

JCreator 3.50

Jflex-1.4

JavaCup 0.10

PCSpim 7.0

Rational Rose 2003

UltraEdit 10.20

Adobe Acrobat 7.0