# **编译原理课程设计**

**项目报告**

# **王瑞吉**

学号5140219030

# 

# 简介

# 1.目标

# 制作Tiger语言编译器

# 2.项目组成Screen Shot 2016-12-05 at 4

# 1.词法分析：分割源程序生成tokens，利用JFlex将自动机文件生成词法分析器程序

# 2.语法分析：根据程序的语法结构检查语法错误，利用Java Cup将文法文件生成语法分析器程序

# 3.抽象语法树：基于语法结构生成抽象语法树

# 4.语义分析：进行变量和类型检查等

# 5.活动记录：记录函数调用

# 6.翻译：翻译生成中间表示树IR Trees（与机器语言无关）

# 7.规范化：优化表达式&分支条件等，为下节做铺垫

# 8.指令选择：处理IR Trees节点，生成MIPS汇编指令

# 9.寄存器分配：选择合适寄存器存储变量，并检查变量的活性

# 10.最终整合：生成完整编译器

# 3.类关系图

# Screen Shot 2016-12-05 at 4词法分析

1.章节目标

运行jflex.bat，根据修改过后的tiger.flex 文件，生成yylex.java为文件，即词法分析器

2.Tiger.flex代码详解

//程序一共分为三大部分，被％％分隔，分别为用户代码，声明和自动机规则

package Parse;

import ErrorMsg.ErrorMsg;

import java\_cup.runtime.\*;

%%

%class yyLex //实现的接口名

%implements Lexer //扫描函数的名称初始为yylex

%function nextToken //扫描元素的返回类型

%type java\_cup.runtime.Symbol //用yychar来统计符号数

%char

//定义函数，以及声明errorMsg

%{

StringBuffer string = new StringBuffer();

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;

}

%}

//处理特殊需要报错的情况

%eofval{

{

if (yystate()==STRING) err("String input error!");

if (yystate()==STRINGNEW) err("String input error!");

return tok(sym.EOF, null);

}

%eofval}

//定义各种符号以及两种状态

LineTerminator = \r|\n|\r\n

InputCharacter = [^\r\n]

WhiteSpace = {LineTerminator} | [ \t\f]

Comments = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}

TraditionalComment = "/\*" [^\*] ~"\*/" | "/\*" "\*"+ "/"

EndOfLineComment = "//" {InputCharacter}\* {LineTerminator}

DocumentationComment = "/\*\*" {CommentContent} "\*"+ "/"

CommentContent = ( [^\*] | \\*+ [^/\*] )\*

Identifier = [:jletter:] [:jletterdigit:]\*

DecIntegerLiteral = 0 | [0-9]\*

%state STRING

%state STRINGNEW

%%

<YYINITIAL>

//自动机部分代码，YYINITIAL是初始状态，可以返回token或者进入其他状态

{

" " {}

{WhiteSpace} {}

{Comments} {}

"let" {return tok(sym.LET, null);}

"in" {return tok(sym.IN, null);}

"end" {return tok(sym.END, null);}

"function" {return tok(sym.FUNCTION, null);}

"var" {return tok(sym.VAR, null);}

"type" {return tok(sym.TYPE, null);}

"int" {return tok(sym.INT, null);}

"string" {return tok(sym.STRING, null);}

"array" {return tok(sym.ARRAY, null);}

"of" {return tok(sym.OF, null);}

"for" {return tok(sym.FOR, null);}

"while" {return tok(sym.WHILE, null);}

"to" {return tok(sym.TO, null);}

"do" {return tok(sym.DO, null);}

"break" {return tok(sym.BREAK, null);}

"if" {return tok(sym.IF, null);}

"then" {return tok(sym.THEN, null);}

"else" {return tok(sym.ELSE, null);}

":=" {return tok(sym.ASSIGN, null);}

"&" {return tok(sym.AND, null);}

"|" {return tok(sym.OR, null);}

"+" {return tok(sym.PLUS, null);}

"-" {return tok(sym.MINUS, null);}

"\*" {return tok(sym.TIMES, null);}

"/" {return tok(sym.DIVIDE, null);}

">=" {return tok(sym.GE, null);}

">" {return tok(sym.GT, null);}

"<=" {return tok(sym.LE, null);}

"<" {return tok(sym.LT, null);}

"<>" {return tok(sym.NEQ, null);}

"=" {return tok(sym.EQ, null);}

\n {newline();}

"," {return tok(sym.COMMA, null);}

"." {return tok(sym.DOT, null);}

";" {return tok(sym.SEMICOLON, null);}

":" {return tok(sym.COLON, null);}

"{" {return tok(sym.LBRACE, null);}

"}" {return tok(sym.RBRACE, null);}

"(" {return tok(sym.LBRACK, null);}

")" {return tok(sym.RBRACK, null);}

"[" {return tok(sym.LPAREN, null);}

"]" {return tok(sym.RPAREN, null);}

"nil" {return tok(sym.NIL,null);}

\" {string.setLength(0); yybegin(STRING);}

{Identifier} {return tok(sym.ID, yytext());}

{DecIntegerLiteral} {return tok(sym.NUM, new Integer(yytext()));}

[^] {err("Illegal character < "+yytext()+" >!");}

}

<STRING>

{

\" {yybegin(YYINITIAL); return tok(sym.STR,string.toString());}

//需要特殊处理的转义字符\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\r\"\\]+ {string.append(yytext());}

\\t {string.append('\t');}

\\n {string.append('\n');}

\\r {string.append('\r');}

\\\" {string.append('\"');}

\\\\ {string.append('\\');}

{LineTerminator} {err("String Input Error");}

\\ {yybegin(STRINGNEW);}

}

<STRINGNEW>

{ //匹配斜杠

{WhiteSpace} {}

" " {}

\\ {yybegin(STRING);}

\" {err(" \\ isn't match");}

[^] {string.append(yytext());}

}

# 

# 语法分析以及语法分析树

1.Symbol包

Symbol.Binder 起连接作用

Symbol.Symbol 符号

Symbol.Table 符号表

2.Absyn包

包含抽象语法树的结点，比如：

Dec：声明函数／类型／变量（通过abstract抽象类调用）

Absyn.Print：输出抽象语法树

3.Grm.cup 代码

package Parse;

import Absyn.\*;

//代码必须在两个冒号之间

action code {:

static Symbol.Symbol sym(String s) {

return Symbol.Symbol.symbol(s);

}

:};

parser code {:

public Exp parseResult;

Lexer lexer;

ErrorMsg.ErrorMsg errorMsg;

//构造函数

public parser(Lexer l, ErrorMsg.ErrorMsg err) {

this();

errorMsg=err;

lexer=l; //传入词法分析器

}

public void synerr\_print(java\_cup.runtime.Symbol s) {

err\_print("Syntax error (" + s.sym + ")", s);

}

public void err\_print(String message, java\_cup.runtime.Symbol info) {

errorMsg.error(info.left, message);

}

:};

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

//定义终结符,字符串类型，整数类型，以及其它类型

terminal String ID, STR;

terminal Integer NUM;

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, UMINUS;

//定义非终结符，来自于抽象语法树结点的类型（调用Absyn）

non terminal Exp expr, program;

non terminal ExpList expr\_seq, expr\_list;

non terminal Dec decl;

non terminal DecList decl\_list;

non terminal TypeDec type\_decl, type\_decllist;

non terminal VarDec variable\_decl;

non terminal FunctionDec function\_decl, function\_decllist;

non terminal Ty type;

non terminal FieldList type\_fields;

non terminal FieldExpList field\_list;

non terminal Var lvalue;

//定义优先级

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;

//文法开始，文法规则＋抽象语法树结点

start with program;

program ::= expr: e

{: parser.parseResult = (Exp)e; :};

//表达式文法

//RESULT为结果，left代表符号左侧，right代表符号右侧

expr ::= LET: l decl\_list: dl IN expr\_seq: es END

{: RESULT = new LetExp(lleft, dl, new SeqExp(esleft, es)); :}

| LET: l decl\_list: dl IN END

{: RESULT = new LetExp(lleft, dl, null); :}

| BREAK: b

{: RESULT = new BreakExp(bleft); :}

| 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); :}

| WHILE: w expr: e1 DO expr: e2

{: RESULT = new WhileExp(wleft, e1, e2); :}

| 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); :}

| ID: i LBRACE field\_list: fl RBRACE

{: RESULT = new RecordExp(ileft, Symbol.Symbol.symbol(i), fl); :}

| ID: i LBRACE RBRACE

{: RESULT = new RecordExp(ileft, Symbol.Symbol.symbol(i), null); :}

| ID: i LPAREN expr: e1 RPAREN OF expr: e2

{: RESULT = new ArrayExp(ileft, Symbol.Symbol.symbol(i), e1, e2); :}

| ID: i LBRACK expr\_list: el RBRACK

{: RESULT=new CallExp(ileft, Symbol.Symbol.symbol(i), el);:}

| ID: i LBRACK RBRACK

{: RESULT = new CallExp(ileft, Symbol.Symbol.symbol(i), null); :}

| LBRACK: l expr\_seq: es RBRACK

{: RESULT = new SeqExp(lleft, es); :}

| LBRACK: l RBRACK

{: RESULT = new SeqExp(lleft, null); :}

| lvalue: l

{: RESULT = new VarExp(l.pos, l); :}

| lvalue: l ASSIGN expr: e

{: RESULT = new AssignExp(lleft, l, e); :}

//处理二元运算符

| 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)); :}

| 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); :}

| 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); :}

| MINUS: m expr: e

{: RESULT = new OpExp(mleft, new IntExp(e.pos, 0), OpExp.MINUS, e); :} %prec UMINUS

| NIL: n

{: RESULT = new NilExp(nleft); :}

| STR: s

{: RESULT = new StringExp(sleft, s); :}

| NUM: i

{: RESULT = new IntExp(ileft, i); :}

;

//逗号，分号表达式，以及域表达式等等

expr\_seq ::= expr: e

{: RESULT = new ExpList(e, null); :}

| expr: e SEMICOLON expr\_seq: es

{: RESULT = new ExpList(e, es); :}

;

expr\_list ::= expr: e

{: RESULT = new ExpList(e, null); :}

| expr: e COMMA expr\_list: el

{: RESULT = new ExpList(e, el); :}

;

field\_list ::= ID: id EQ expr: e

{: RESULT = new FieldExpList(idleft, Symbol.Symbol.symbol(id), e, null); :}

| ID: id EQ expr: e COMMA field\_list: fl

{: RESULT = new FieldExpList(idleft, Symbol.Symbol.symbol(id), e, fl); :}

;

lvalue ::= ID: id

{: RESULT = new SimpleVar(idleft, Symbol.Symbol.symbol(id)); :}

| lvalue: l DOT ID: id

{: RESULT = new FieldVar(lleft, l, Symbol.Symbol.symbol(id));:}

| lvalue: l LPAREN expr: e RPAREN

{: RESULT = new SubscriptVar(lleft, l, e); :}

| ID: i LPAREN expr: e RPAREN

{: RESULT = new SubscriptVar(ileft,new SimpleVar(ileft,Symbol.Symbol.symbol(i)),e); :}

;

decl\_list ::= decl: d

{: RESULT = new DecList(d, null);:}

| decl: d decl\_list: dl

{: RESULT = new DecList(d, dl);:}

;

decl ::= type\_decllist: td

{: RESULT = td; :}

| variable\_decl: vd

{: RESULT = vd; :}

| function\_decllist: fd

{: RESULT = fd; :}

;

type\_decllist ::= type\_decl: t

{: RESULT = t; :}

| type\_decl: t type\_decllist: tl

{: RESULT = new TypeDec(t.pos, t.name, t.ty, tl); :}

;

type\_decl ::= TYPE: t ID: id EQ type: ty

{: RESULT = new TypeDec(tleft, Symbol.Symbol.symbol(id), ty, null); :}

;

type ::= ID: id

{: RESULT = new NameTy(idleft, Symbol.Symbol.symbol(id)); :}

| LBRACE: l type\_fields: tf RBRACE

{: RESULT = new RecordTy(lleft, tf); :}

| LBRACE: l RBRACE

{: RESULT = new RecordTy(lleft, null); :}

| ARRAY: a OF ID: i

{: RESULT = new ArrayTy(aleft, Symbol.Symbol.symbol(i)); :}

|ARRAY: a OF INT

{: RESULT = new ArrayTy(aleft, Symbol.Symbol.symbol("int")); :}

|ARRAY: a OF STRING

{: RESULT = new ArrayTy(aleft, Symbol.Symbol.symbol("string")); :}

| INT: i

{: RESULT = new NameTy(ileft, Symbol.Symbol.symbol("int")); :}

| STRING: s

{: RESULT = new NameTy(sleft, Symbol.Symbol.symbol("string")); :}

;

type\_fields ::= ID: i1 COLON ID: i2

{: RESULT = new FieldList(i1left, Symbol.Symbol.symbol(i1), Symbol.Symbol.symbol(i2), null); :}

| ID: i1 COLON INT

{: RESULT = new FieldList(i1left, Symbol.Symbol.symbol(i1), Symbol.Symbol.symbol("int"), null); :}

| ID: i1 COLON STRING

{: RESULT = new FieldList(i1left, Symbol.Symbol.symbol(i1), Symbol.Symbol.symbol("string"), null); :}

| ID: i1 COLON ID: i2 COMMA type\_fields: tf

{: RESULT = new FieldList(i1left, Symbol.Symbol.symbol(i1), Symbol.Symbol.symbol(i2), tf); :}

| ID: i1 COLON INT COMMA type\_fields: tf

{: RESULT = new FieldList(i1left, Symbol.Symbol.symbol(i1), Symbol.Symbol.symbol("int"), tf); :}

| ID: i1 COLON STRING COMMA type\_fields: tf

{: RESULT = new FieldList(i1left, Symbol.Symbol.symbol(i1), Symbol.Symbol.symbol("string"), tf); :}

;

variable\_decl ::= VAR: v ID: id1 COLON ID: id2 ASSIGN expr: e

{: RESULT = new VarDec(vleft, Symbol.Symbol.symbol(id1), new NameTy(id1left, Symbol.Symbol.symbol(id2)), e); :}

| VAR: v ID: id ASSIGN expr: e

{: RESULT = new VarDec(vleft, Symbol.Symbol.symbol(id), null, e); :}

| VAR: v ID: id1 COLON INT ASSIGN expr: e

{: RESULT = new VarDec(vleft, Symbol.Symbol.symbol(id1), new NameTy(id1left, Symbol.Symbol.symbol("int")), e); :}

| VAR: v ID: id1 COLON STRING ASSIGN expr: e

{: RESULT = new VarDec(vleft, Symbol.Symbol.symbol(id1), new NameTy(id1left, Symbol.Symbol.symbol("string")), e); :}

;

//函数声明列表

function\_decllist ::= function\_decl: f

{:RESULT = f; :}

| function\_decl: f function\_decllist: fl

{: RESULT = new FunctionDec(f.pos, f.name, f.params, f.result, f.body, fl); :}

;

function\_decl ::= FUNCTION: f ID: id1 LBRACK type\_fields: tf RBRACK COLON ID: id2 EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id1), tf, new NameTy(id2left, Symbol.Symbol.symbol(id2)), e, null); :}

| FUNCTION: f ID: id1 LBRACK type\_fields: tf RBRACK COLON INT: i EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id1), tf, new NameTy(ileft, Symbol.Symbol.symbol("int")), e, null); :}

| FUNCTION: f ID: id1 LBRACK type\_fields: tf RBRACK COLON STRING: s EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id1), tf, new NameTy(sleft, Symbol.Symbol.symbol("string")), e, null); :}

| FUNCTION: f ID: id LBRACK type\_fields: tf RBRACK EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id), tf, null, e, null); :}

| FUNCTION: f ID: id1 LBRACK RBRACK COLON ID: id2 EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id1), null, new NameTy(id2left, Symbol.Symbol.symbol(id2)), e, null); :}

| FUNCTION: f ID: id1 LBRACK RBRACK COLON INT: i EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id1), null, new NameTy(ileft, Symbol.Symbol.symbol("int")), e, null); :}

| FUNCTION: f ID: id1 LBRACK RBRACK COLON STRING: s EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id1), null, new NameTy(sleft, Symbol.Symbol.symbol("string")), e, null); :}

| FUNCTION: f ID: id LBRACK RBRACK EQ expr: e

{: RESULT = new FunctionDec(fleft, Symbol.Symbol.symbol(id), null, null, e, null); :}

;

//之后使用java\_cup进行规约，生成parser.java（语法分析器）,sym.java（常数表）两个文件

4. Parse.java

提供parse包的接口，调用yylex执行词法分析，调用Grm执行语法分析，并且返回抽象语法树。当语法分析需要下一个符号时，会通过调用lexer.nextToken()返回词法分析，获得相应符号

# 语义分析

1.Types包

封装了类型信息，包括INT，NIL，STRING，VOID，NAME，ARRAY，RECORD，以及判断函数如isIntType，静态常量类型\_int，还有强制转换的内容：Type.Java中的CoreTo(Type)

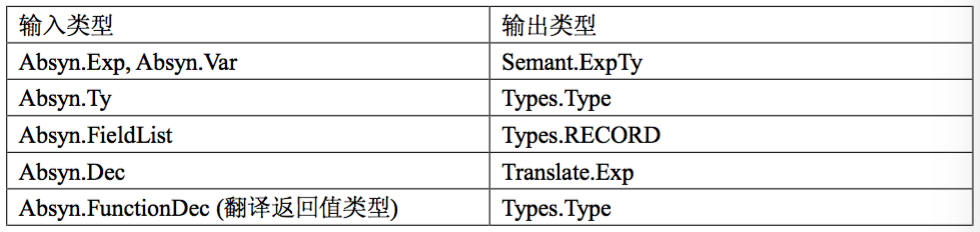
其中，NAME表示还未被识别出的类型，比如赋值语句，因此真正的类型被确定必须要使用binding方法将类型绑定到NAME类中，再调用NAME.actual方法（其他类也有配套actual方法，返回本身的类型）。

NAME也具有检查循环定义的功能（isLoop函数），具体实现过程注释可参见源代码文件。

2.语义分析过程

语义分析和翻译中间代码是相互联系的。语义分析时，需要检查语法结点，即需要递归的检查其子语法成分的正确性（并且需要翻译）。当检查正确，并且翻译完成后，再调用Translate对整个表达式进行翻译。

类型检查的输入和输出：



3.Semant包

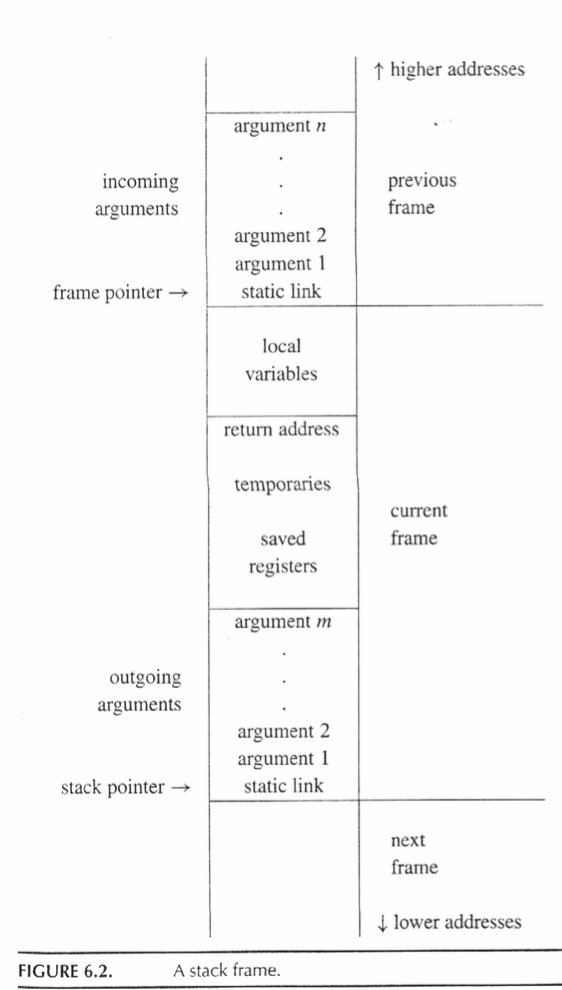
Entry.java 描述变量和函数的种类，包括：普通变量VarEntry，库函数StdFuncEntry，循环变量LoopVarEntry，普通函数FuncEntry

Env.java 包含了两个符号表，tEnv记录了当前类型名称（关联符号和类型），并且需要添加int和string进行初始化；vEnv记录了当前变量等信息（关联符号和入口），需要添加库函数进行初始化

Semant.java 具体表达式&声明&变量的检查。每种类型检查完之后会调用translator中对应的函数进行翻译（例如IntExp会调用transIntExp，如果没有翻译动作调用transNoOp）。具体类型检查可查看代码注释。

# 活动记录

1. Frame的结构



上图为Frame的结构

2.Frame 包

Frame.java 定义了Frame结构，用来存放函数的相关信息，详细情况见下面代码注释：

public abstract class Frame implements TempMap{

public Label name = null;//帧名称

public AccessList formals = null;//变量列表

public abstract Frame newFrame(Label name, Util.BoolList formals);//建立新帧(名称、逃逸信息表)

public abstract Access allocLocal(boolean escape); //分配新本地变量(是否逃逸，即Access是放在Frame－返回true中还是在Register－返回false中)

public abstract Tree.Exp externCall(String func, Tree.ExpList args);//调用外部函数

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 java.util.HashSet registers(); //寄存器列表

public abstract Tree.Stm procEntryExit1(Tree.Stm body); //添加额外函数调用指令

public abstract Assem.InstrList procEntryExit2(Assem.InstrList body);

public abstract Assem.InstrList procEntryExit3(Assem.InstrList body);

public abstract String string(Label label, String value);

public abstract Assem.InstrList codegen(Tree.Stm s); //生成 MIPS 指令

}

Access.java 描述了形参和局部变量，是抽象数据类型，代码解释如下：

public abstract class 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

public abstract Tree.Exp exp(Tree.Exp framePtr);//以 fp 为起始地址返回变量

public abstract Tree.Exp expFromStack(Tree.Exp stackPtr); //以 sp 为起始地址返回变量

}

AccessList.Java 链表结构，用于把Access连接成链表

3.Mips包

InFrame.java 记录了存放在Frame中的变量地址，可以在Frame包中被引用

InReg.java 记录了存放在Register中的变量地址，可以在Frame包中被引用

MipsFrame.java 描述了Frame在Mips中的存储信息，包括寄存器的初始化（寄存器信息见下述代码注释），建立帧（newFrame函数）并分配Access（allocLocal方法），procEntryExit1（增加一些对Callee的处理指令），procEntryExit2（空指令），procEntryExit3（增加Callee的一些处理指令），（procEntryExit具体功能可见下表），产生数据段汇编代码（string函数），具体可见代码注释。

public int allocDown = 0;//栈偏移量

public TempList argRegs = new TempList(new Temp(5), new TempList(new Temp(6), new TempList(new Temp(7), new TempList(new Temp(8), null))));

//参数寄存器$a0~$a3

public java.util.ArrayList<MOVE> saveArgs = new java.util.ArrayList<MOVE>();//用于保存参数

private Temp fp = new Temp(0);//帧指针

private Temp sp = new Temp(1);//栈指针

private Temp ra = new Temp(2);//返回指针

private Temp rv = new Temp(3);//返回值

private Temp zero = new Temp(4);//保存零值

private TempList calleeSaves = null;//寄存器$s0~$s7，callee是被调用的函数，calleeSave被用来保存长期变量（如全局变量）

public TempList callerSaves = null;//寄存器$t0~$t9，caller是调用者，callerSave被用来保存短期变量（如局部变量）

private int numOfcalleeSaves = 8;//用于callee的寄存器的数量,共8个

ProcEntryExit功能简要说明：

ProcEntryExit在translate.java中被调用，并调用了ProcEntryExit1，而ProcEntryExit2被Main.java调用，最终生成函数调用的汇编代码。

功能具体功能可见下表：

|  |  |
| --- | --- |
| 汇编代码内容 | 实现该部分的函数 |
| 设置函数体 | ProcEntryExit3 |
| 计算$sp,并分配空间 | ProcEntryExit3 |
| 保存老$fp | ProcEntryExit2 |
| 计算新$fp(=$sp-frame空间大小+4bytes) | ProcEntryExit2 |
| 保存$ra | ProcEntryExit2 |
| 保存Callee-Saved寄存器 | ProcEntryExit2 |
| 保存参数$A0~A3 | ProcEntryExit2 |
| (以下为函数体部分) |  |
| 写入返回值到$rv | ProcEntryExit1 |
| 恢复Callee－Saved寄存器 | ProcEntryExit2 |
| 恢复$ra | ProcEntryExit2 |
| 恢复$fp | ProcEntryExit2 |
| 恢复$sp,$sp+=相应frame大小 | ProcEntryExit3 |
| 跳转，返回 | ProcEntryExit3 |

# 中间代码生成

1.Tree包

Tree.Exp & Tree.stm 定义了中间树的结点，并以此派生出Tree包中的其他表达式。

ExpList & StmList 是exp和stm的链表

Tree.Exp派生的表达式（有返回值）：

CONST.java 定义了整数常量

NAME.java 字符常量

TEMP.java 临时变量

BINOP.java 二元运算，比如加减乘除和逻辑运算

MEM.java 制定地址下存储器的内容

CALL.java 函数调用

ESEQ.java 调用stm和exp进行计算

Tree.Stm派生的表达式（无返回值）：

MOVE.java 将某个内容存入TEMP或者MEM

EXP.java 计算

JUMP.java 跳转指令

CJUMP.java 比较关系的判断指令（EQ，NE，LT）

SEQ.java 连接stm

LABEL.java 标记n作为当前地址

2.Translate包

Exp.java 宏观表达式，抽象类，总括各种表达式的实现

Ex.java 有返回值的中间表达式

Nx.java 无返回值的中间表达式

Cx.java 一个判断条件，会调用RelCx.java，返回真or假

IfExp.java if else语句

WhileExp while语句

ForExp.java for循环语句

以上java文件代码分析详见注释

3.Frag包

Frag.java 段数据结构，可以是字符串或函数块，可形成链表，并且在翻译过程中不断杯拓展。可通过AddFrag和GetResult处理链表

DataFrag.java 字符串段，有标号和字符数据

ProcFrag.java 函数块段，包含Tree.Stm和其Frame

4.Translate的实现

整数常量 CONST(value)

字符串常量 NAME（label）

nil空变量 EX(CONST(0))

变量表达式 translate.Exp

运算表达式 EX(BINOP(binop,left,right))

字符串运算 RELCX(OPER,EX(COMP),EX(0))

关系运算 RELEX(BINOP(binop,left,right))

赋值运算 NX(MOVE(lvalue,ex))

获取静态链接 EX(CALL(NAME(func\_name),args))

库函数调用前者 EX(EXTERNAL\_CALL)

连接stm NX（SEQ（NX(E1),NX(E2)））

连接exp EX（ESEQ（NX(E1),EX(E2)））

数组类型 EX(alloc)

判断&循环语句 IfExp,WhileExp,ForExp

break指令 NX(JUMP(LABEL))

下标 EX(MEM(BINOP(BINOP.PLUS, EX(VAR),BINOP(MUL,EX(IDX),CONST(WORDSIZE)))

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

Record类型 EX(ESEQ(SEQ(MOVE(TEMP(addr),alloc),init),TEMP(addr)))

# 规范化

1.Canon包

可以直接使用。主要包含三个接口：

Canon.Canon.linearize (Tree.Stm s)：

将IR写成规范树表（无SEQ和ESEQ）

Canon.BasicBlocks. BasicBlocks(Tree.StmList stms)：

根据规范树表划分基本块，不含内部跳转和标号

Canon.TraceSchedule(BasicBlocks b)

顺序放置基本块，所有CJUMP都有false标号

# 指令选择

1.Mips包

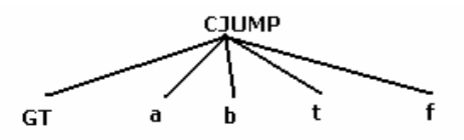
Codegen.java 生成汇编指令（其他文件在前文已经说明过），主要情况如下（详情请看代码注释）：

1.以Tree.MOVE为根，根据不同6种不同形式的IR树调用emit

2.以Tree.LABEL为根，直接输出标号字符串

3.以Tree.JUMP为根，强制跳转

4.以Tree.CJUMP为根，如下图，翻译为BGT A， B， T



5.以Tree.EXP为根，继续翻译表达式

6.以Tree.MEM为根，根据4种不同情况调用emit

7.以Tree.CONST为根，则IR树为一个节点，直接翻译

8.以Tree.BINOP为根，根据不同运算的不同情况翻译

9.以Tree.TEMP为根，直接返回寄存器

10.以Tree.NAME为根，读入字符串段地址

11.以Tree.CALL为根，填参数寄存器（li或move），之后执行跳转

其中emit指令会用到s、d、j代表src，dst，jump

2.Assem.包

Instr.java 包含Instr数据类型，即无需进行寄存器分配的指令

OPER.java 派生于Instr，包含assem指令，操作数寄存器列表src和结果寄存器dst，均可为空

LABEL.java 表示程序将要跳转的地点，包括assem（指向标号），label参数（确定标号符号）

MOVE.java 必须进行数据转换，src&dst不能在同一个寄存器，否则会被删除

# 指令选择

1.FlowGraph包

目的：根据汇编指令生成流图

FlowGraph.java 封装了抽象的流图结构，并定义了三个接口：

/\*\*

\* 定义的变量

\*/

public abstract TempList def(Node node);

/\*\*

\* 使用的变量

\*/

public abstract TempList use(Node node);

/\*\*

\* 是否是move

\*/

public abstract boolean isMove(Node node);

AssemFlowGraph.java 封装了面向汇编指令的流图结构，每个结点代表一条指令，每条边代表一个可能的转移，并可以据此生成具体的流图。

2.Graph包

该包中封装了一个抽象图的数据结构，包含结点信息有向边的信息，以及大多数对图的操作函数。活性分析（即求出每个结点的out变量）依赖于此包。

GraphNodeInfo.java 描述流图中的结点信息，其数据结构如下：

public Set<Temp.Temp> in = new HashSet<Temp.Temp>();//来指令前的活性变量

public Set<Temp.Temp> out = new HashSet<Temp.Temp>();//出指令后的活跃变量

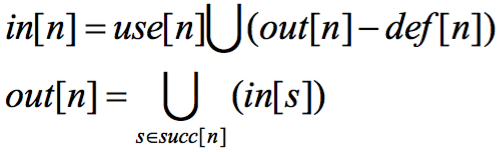
public Set<Temp.Temp> use = new HashSet<Temp.Temp>();//某指令引用的变量,赋值号右边

public Set<Temp.Temp> def = new HashSet<Temp.Temp>();//赋值号左边

3.RegAlloc包

Liveness.java:

是活性分析的实现部分，包含initNodeInfo函数，用于对每个结点进行初始化；calcLiveness函数，按照下图公式不断迭代，直到in，out不变，然后获得活跃变量。



InterferenceGraph.java:

提供了生成干扰图的抽象接口。生成干扰图算法简述（详见代码注释）：1. 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) ； 2.在转移指令 c到a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)

Color.java:

根据干扰图，分配寄存器，主要根据着色算法（使干扰图中相邻两个结点染不同颜色），近似的贪心算法简述如下（具体可见代码）：

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到b 和 b到a 不是同一条边.)

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)

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, 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.

RegAlloc.java:

整体调用包中其他文件，提供统一接口。调用上述文件，先根据汇编指令生成流图（AssemFlowGraph），再生成干扰图（Liveness），进行活性分析（Liveness），最后应用着色算法分配寄存器（Color）

# 最终整合

1.ErrorMsg包

封装了错误信息，可以调用error输出报错位置和报错信息，也可以在遇到严重错误时抛出异常，终止程序

2.Main包

传入输入文件名，输出词法分析结果（yylex\_result）astOut（抽象语法树），irOut（IR树，.ir文件），out（MIPS指令集.s文件）。具体步骤如下：

传入输入文件，调用Parse包（yylex进行词法分析，Grm进行语法分析，返回抽象语法树的根，输出astOut）。新建Frame并创建translate和semant对象，在进行语义检查和IR树翻译（transProg），返回翻译完成的段链表。之后进行规范化，分别处理数据段（输出信息）和函数段（执行emitProc，输出该函数段的ir树和汇编指令），其中emitProc包含输出ir树，规范化，划分基本块，放置基本块，生成汇编代码，寄存器分配，添加调用和返回，输出Mips，追加标准函数库。