编译原理

Tiger Compiler

Java 实现

韩雨桐

5130309482

ivyhan2013@126.com

2015/12/11

上海交通大学电子信息与电气工程学院计算机系大三上编译原理课程设计

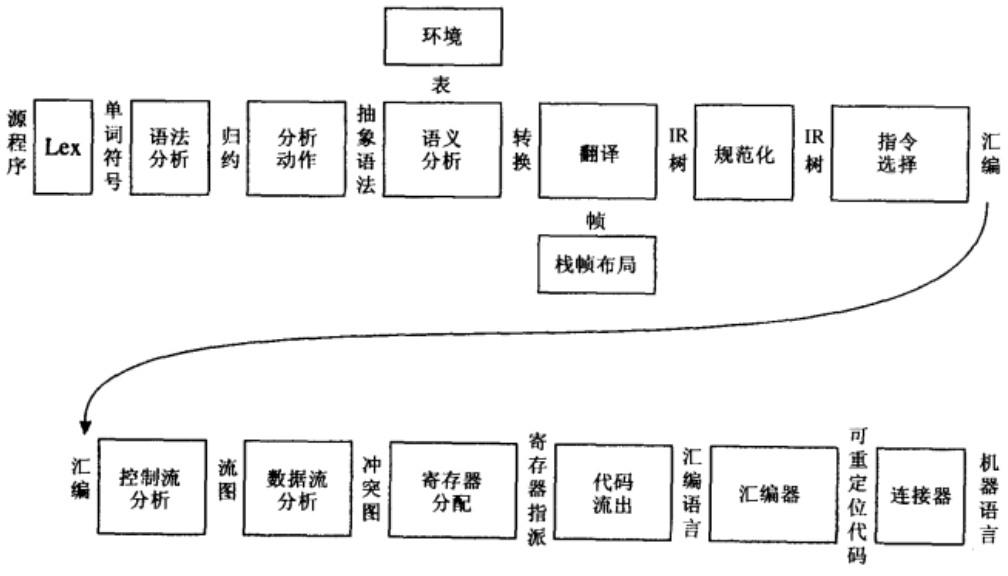
目录

第1章 项目简介

1.1 项目目的

本项目使用《Modern Compiler Implementation in Java》，俗称虎书。虎书讲述了将程序设计语言转换成可执行代码是使用的技术，数据结构和算法。为了阐明实际编译时会遇到的问题，虎书以 Tiger 语言为例说明如何编译一种语言。虎书每章讲解编译器的一个模块，每章后面的程序设计都要求实现相应的编译阶段。本项目的目的就是按照虎书给出的顺序实现一个 Tiger 编译器。

1.2 主要模块



编译器的各个阶段以及他们的接口

1) 词法分析：使用工具 jflex 生成一个词法分析器；

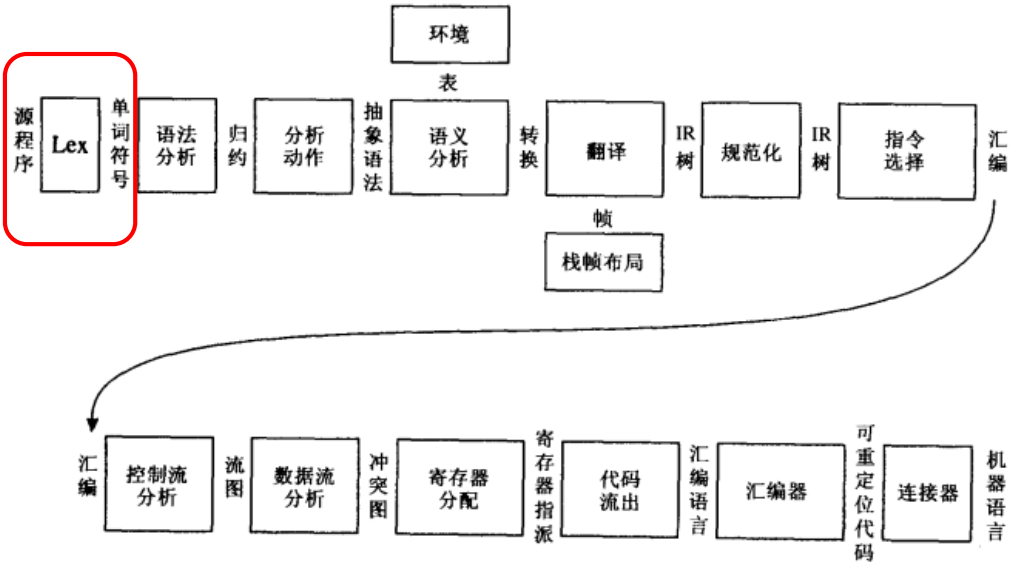
2) 语法分析：使用工具 java_cup 生成一个语法分析器；

3) 抽象语法分析：生成抽象语法分析树；

第2章 词法分析

2.1 本章任务

了解 JFlex 的使用方法，正确书写 JFlex 生成词法分析器。



例如：源程序为 A:= B+1;

经过词法分析之后：

tok（ sym.ID,"X" ）

tok(sym.ASSIGN,null)
tok(sym.ID,"B")
tok(sym.PLUS,null)
tok(sym.INT,1)
tok(sym.SEMICOLON,null)

2.2 JFlex 的使用

配置文件以.flex 为扩展名，整个文档分为三个部分，使用%%划分

用户代码

选项与声明

词法规则

形式形如：

```
用户代码
……
……
%%
选项与声明
……
……
%%
词法规则
……
```

2.2.1用户代码

JFLEX 直接将这部分代码拷贝到生成词法分析器 Java 源文件中 ,通常在这里我们只定义一些类注释信息以及 package 和 import 的引用。

本项目中我这样定义用户代码

```
    package tiger.parse;
import tiger.errormsg.ErrorMsg;
import java_cup.runtime.*;
```

2.2.2选项与声明

在这一部分，**选项**用来定制词法分析器，**声明**则是声明一些能够在第三部分(词法规则定义)使用的宏定义和词法状态，其中宏大多由正则表达式定义。

所有选项都要由一个“%”符号开头，本次用到的**选项**有：

%char

打开%char 开关后，生成的程序中 int yychar 的值为当前的字符的个数（从 0 计数）

```
%char
```

%line

打开%line 开关后，生成的程序中 int yyline 的值为当前的字符的行数（从 0 计数）

```
%line
```

%column

打开%column 开关后，生成的程序中 int yycolumn 的值为当前的字符的列数（从 0 计数）

```
%column
```

%unicode

支持字符集中 0-65535#字符，使用该字符集不会出现运行时的溢出现象。

%cup

该指令等同于以下指令集

```
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
return new java_cup.runtime.Symbol(<CUPSYM>.EOF);
%eofval}
%eofclose
```

```
%cup
```

%{.....用户代码.....%} 类代码指令

其中用户代码将被直接复制到生成类文件中，在这里你可以定义自己的成员变量和方法。此规范描述中出现多个类代码指令，那么 JFLEX 将根据这些类代码指令出现的先后顺序将他们拼接起来

```
%{

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) {
                return    new    java_cup.runtime.Symbol(kind,    yychar,
yychar+yylength(), null);
}

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


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

private ErrorMsg errorMsg;

  StringBuffer string = new StringBuffer();
  int commentstate=0;
```

```
  private Symbol symbol(int type) {
    return new Symbol(type, yyline, yycolumn);
  }
  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline, yycolumn, value);
  }

%}
```

本次用的**声明**有：

```
LineTerminator = \r\n|\r|\n
InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]

Identifier = [:jletter:] [:jletterdigit:]*
DecIntegerLiteral =[0-9]+
```

这些宏定义都使用了正则表达式定义。

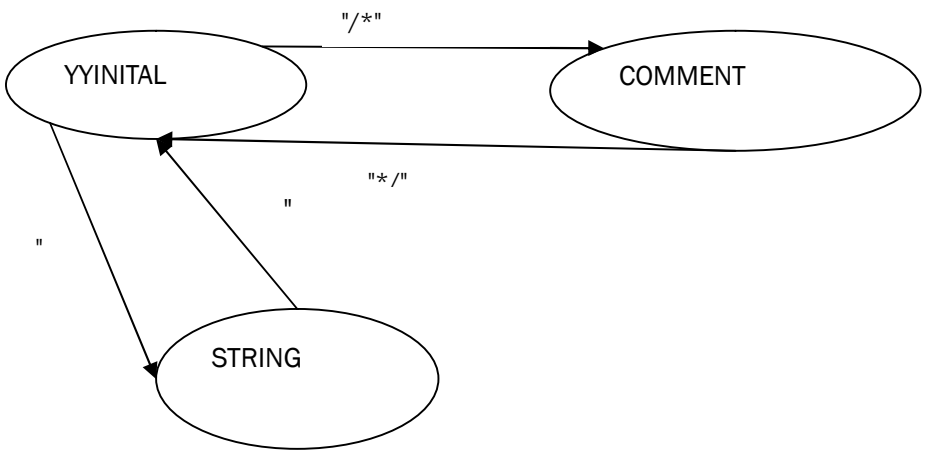
2.2.3词法规则

我将通过回答几个问题的形式来描述我是如何组织语法规则的

1. 总共有几个状态？

   一共有：YYINITIAL、STRING、COMMENT 三个状态。

下面是一个简单的状态转换图，**并不是完全正确的**，但是可以描述出大致的转换。



2. 如何处理错误？

在词法分析部分，错误可能是由于注释，字符串没有闭合导致，也就是/* */ " " \\没有匹配导致的。

所以当文件结束的时候，一定是出于 YYINITAL 状态，如果出于 STRING 状态说明字符串没有闭合，如何出于 COMMENT 状态说明注释没有匹配。

3. 如何处理注释？

Tiger 注释支持嵌套，所以引入一个 int commentstate 量，模拟栈。

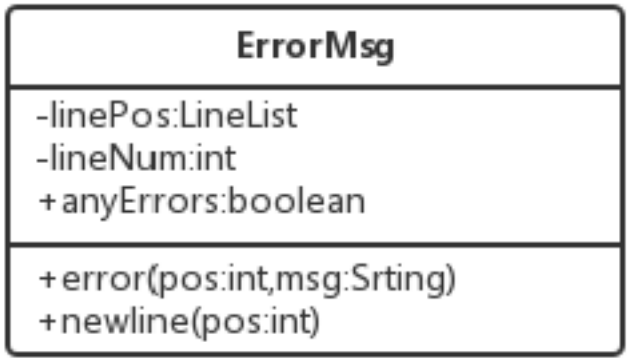
```
"/*"            {commentstate++;}
"*/"            {commentstate--;if (count==0) {yybegin(YYINITIAL);}}
```

4. 如何处理转义字符？

```
[^\n\"\\]+                  { string.append( yytext() ); }
\\t                         { string.append('\t'); }
\\n                         { string.append('\n'); }
\\\"                         { string.append('\"'); }
\\                          { string.append('\\'); }
\\[0-3]?[0-9][0-9]           { char val = (char)
Integer.parseInt(yytext().substring(1),8);
                                string.append( val ); }
```

5. 如何处理报错，显示行数？

虎书中提供了 ErrorMsg 包，ErrorMsg 类图如下：



error 用于输出错误，而我们可以在程序中显式的调用 newline 来计算行数。在每个

```
{LineTerminator} { newline(); }
```

处调用 newline ( )；但是这样当在 STRING 状态中当处理换行的字符串时，需要判断何时换行。

同时开启%line 和 %colum 用 Symbol 记录行数和列数

```
private Symbol symbol(int type) {
  return new Symbol(type, yyline, yycolumn);
}
private Symbol symbol(int type, Object value) {
  return new Symbol(type, yyline, yycolumn, value);

}
```

2.3完成情况

1. 完成.flex 文件编写
2. 可以过滤注释
3. 可以将程序流切割成 token
4. 可以处理长字符串

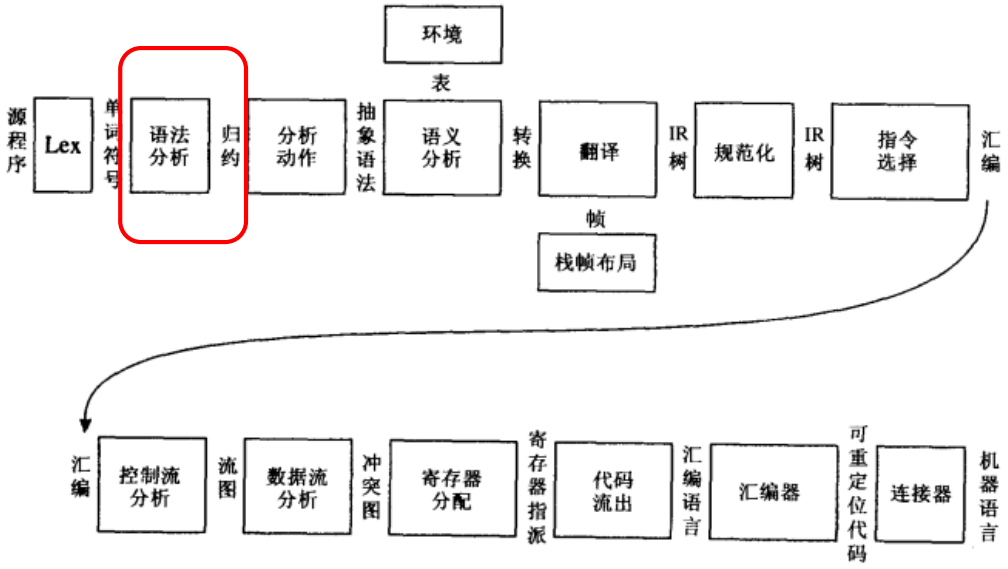
2.4参考资料

JFlex User's Manual

第3章 语法分析

3.1本章任务

了解 java_cup 的使用，对词法分析输出的单词符号进行语法分析。



3.2CUP 语法

在最上方定义 package 和 import 等用户代码；

将代码部分放在{: :}之中；

定义终结符：terminal

```
terminal String ID, STRING;
terminal Integer INT;
terminal COMMA;
```

定义非终结符：non terminal

```
non terminal Exp program, expr;
```

定义优先级（优先级由高到低）：

```
precedence left ELSE;
precedence left type_declaration,
    variable_declaration,function_declaration;
precedence left OR;
precedence left AND;
precedence nonassoc EQ,LE,GE,LT,GT,NEQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left DOT;
```

语法规则定义方式如下：

```
exp :: = exp PLUS exp {:semantic action:}
```

3.3语法规则

program	→ *exp*
dec	→ *tyDec* | *varDec* | *funDec*
tyDec	→ **type tyId** = *ty*
ty	→ **tyId** | *arrTy* | *recTy*
arrTy	→ **array of tyId**
recTy	→ { *fieldDec**, }
fieldDec	→ **id : tyId**
funDec	→ **function id** (*fieldDec**,) = *exp* | **function id** (*fieldDec**,) **: tyId** = *exp*
varDec	→ **var id** := *exp* | **var id : tyId** := *exp*
lValue	→ **id** | *subscript* | *fieldExp*
subscript	→ *lValue* [*exp*]
fieldExp	→ *lValue* . **id**
exp	→ *lValue* | **nil** | **intLit** | **stringLit** | *seqExp* | *negation* | *callExp* | *infixExp* | *arrCreate* | *recCreate* | *assignment* | *ifThenElse* | *ifThen* | *whileExp* | *forExp* | **break** | *letExp*
seqExp	→ (*exp*$^{*;}$)
negation	→ - *exp*
callExp	→ **id** (*exp**,)
infixExp	→ *exp* **infixOp** *exp*
arrCreate	→ **tyId** [*exp*] **of** *exp*
recCreate	→ **tyId** { *fieldCreate**, }
fieldCreate	→ **id** = *exp*
assignment	→ *lValue* := *exp*
ifThenElse	→ **if** *exp* **then** *exp* **else** *exp*
ifThen	→ **if** *exp* **then** *exp*
whileExp	→ **while** *exp* **do** *exp*
forExp	→ **for id** := *exp* **to** *exp* **do** *exp*
letExp	→ **let** *dec*$^{+}$ **in** *exp*$^{*;}$ **end**

根据上表总结出来的语法规则，按照 cup 语法进行编写 cup 程序。

3.4FAQ

1. 终结符为什么要分 String 和 Integer 和其他？

   cup 最后被编译成 java 的时候，标记为 string 的终结符会对应 String 类型；Integer 会对应 int 类型；其他会对应 Object 类型。

   所以 ID 和 String 要注明为 terminal String ，int 要注明 terminal Integer。

2. 如何处理异常？

   在语法分析阶段，官方代码中给出的异常处理的方式是系统抛出异常，而不是通过 ErrorMsg 进行管理。

3. 如何将 cup 和 jflex 进行连接？

   在 jflex 中打开%cup 开关，这会让 Yylex implement java_cup.runtime.Scanner

3.5完成情况

1. 完成.cup 文件
2. 使用 java_cup 进行编译 (java j ava_cup. Main Tiger_Parse. cup) 得到 parser 类,冲突个数为零

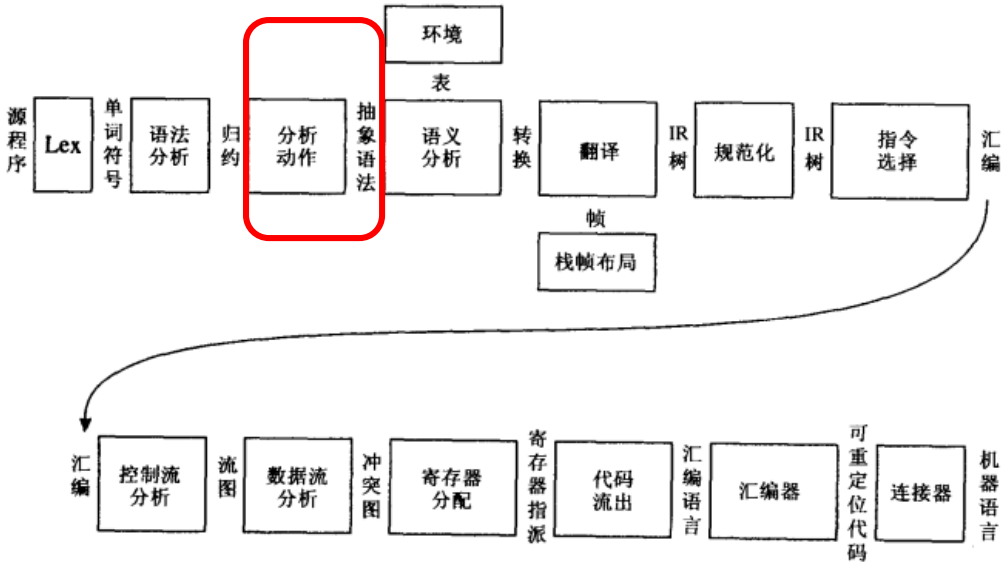
3.6参考资料

1. *Tiger Language Specification* by Martin Hirzel and Kristoffer H. Rose ,NYU ,2013 ***Section 2 Syntax***
2. *MCIJ* chapter 3
3. *CUP′s Manual*

第4章 抽象语法树

4.1本章任务

在上一章的基础上添加抽象语法树，并输出抽象语法树



4.1.1抽象语法树

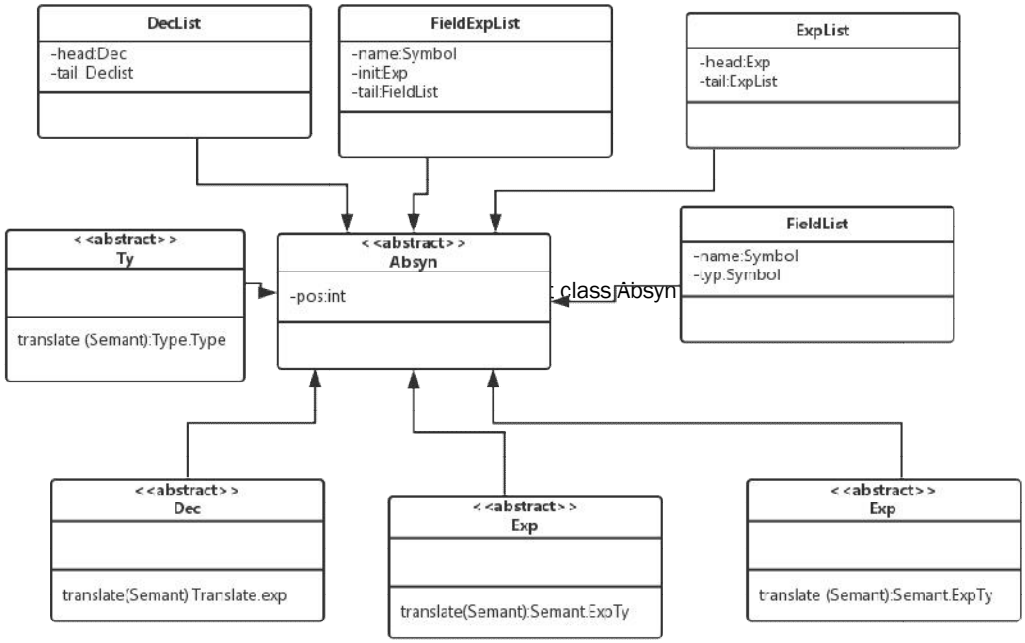
为了有利于模块化，最好将语法问题（语法分析）与语义问题（类型检查和翻译成及其代码）分开处理。达到此目的的一种方法是由语法分析器生成语法分析树，即一种数据结构，编译器在叫后的阶段可对器进行遍历.

抽象语法起到了在语法分析其和编译器的较后阶段建立一个清晰

接口的作用。抽象语法树传递源程序的短语结构，其中已解决了所有语法分析问题，但是不带有任何语义解释。

4.2Absyn 包

Absyn 包中包含了 Tiger 抽象语法声明，在本章中他提供了 AST 的根节点，因为抽象语法是语法和语义的接口，所以弄清 Absyn 的结构是十分重要的。在这里列出 Absyn 的类图。



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）

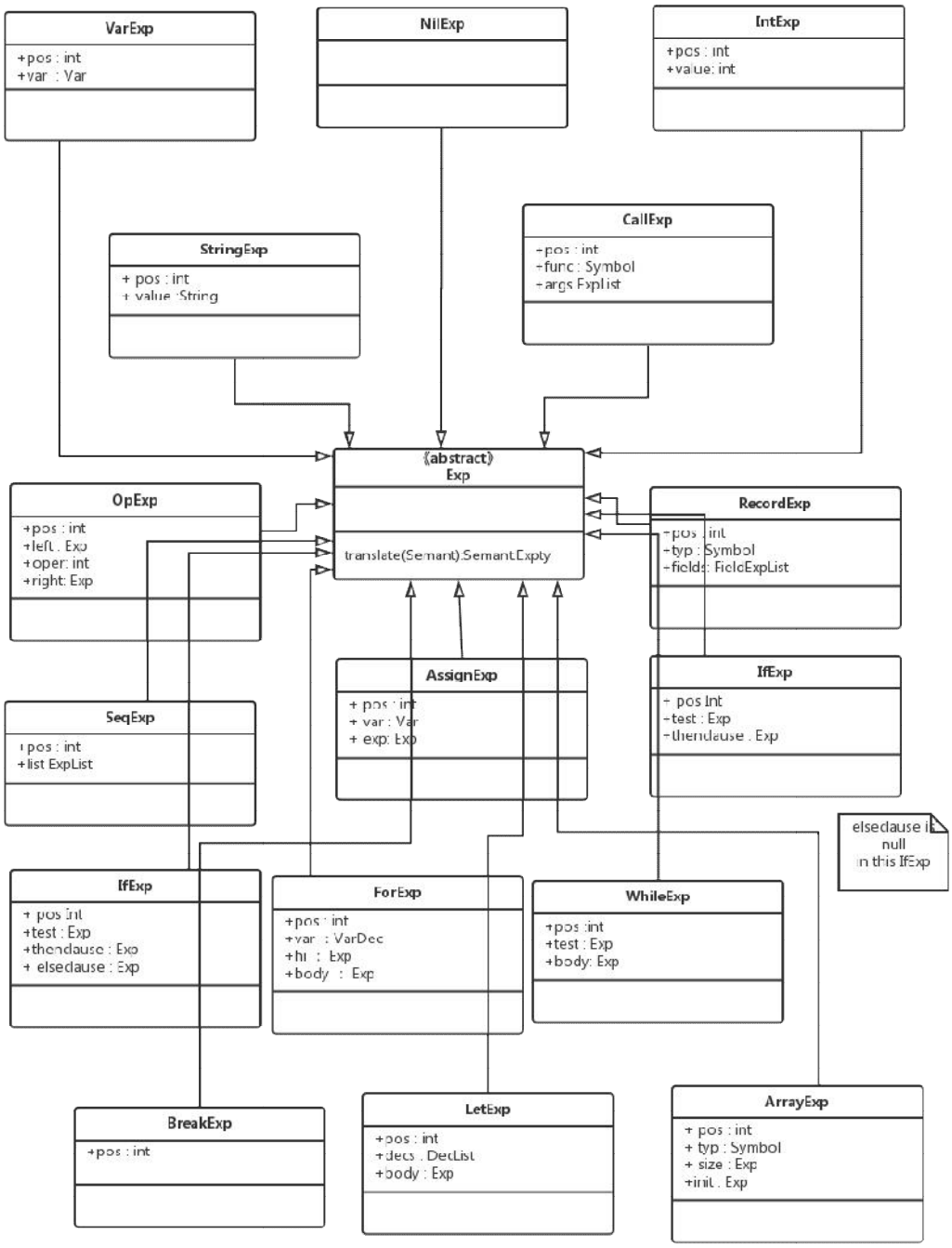
Lists:

DecList: 声明列表,用于 Tiger 语言的声明块结构

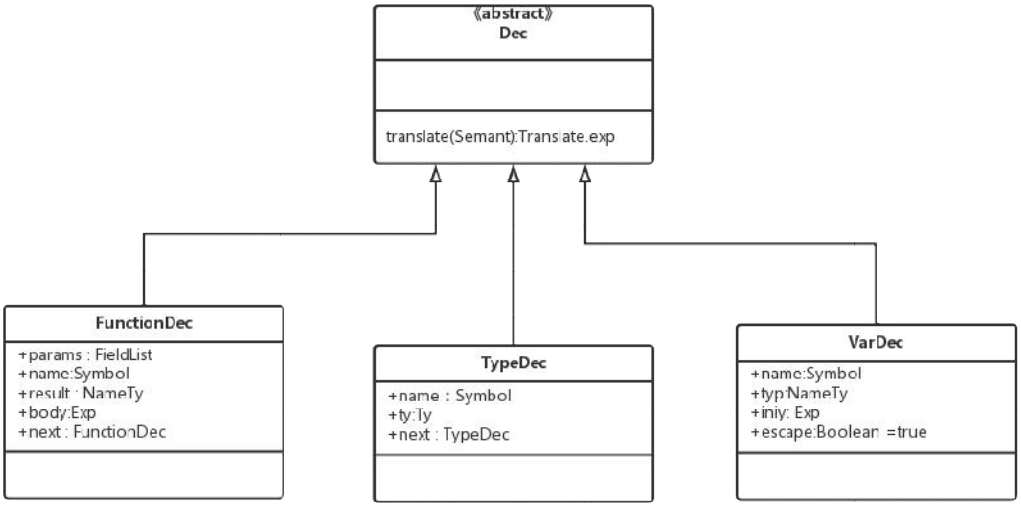
FieldExpList: 用逗号分割的表达式列表,用于函数调用.例如:func(a,b,c)

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

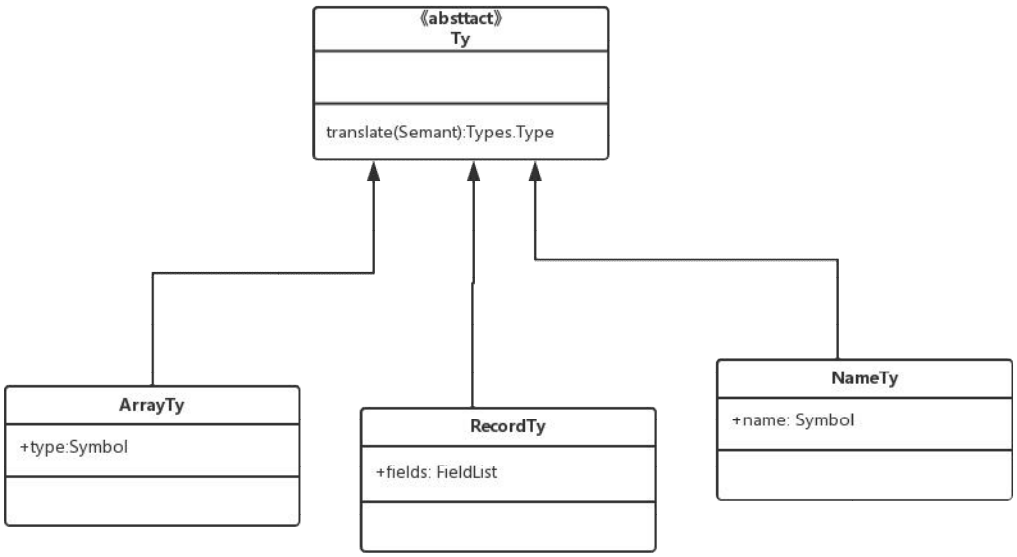
FieldList: 域列表,用在域变量中.例如{name="abc", age=10}中的 list



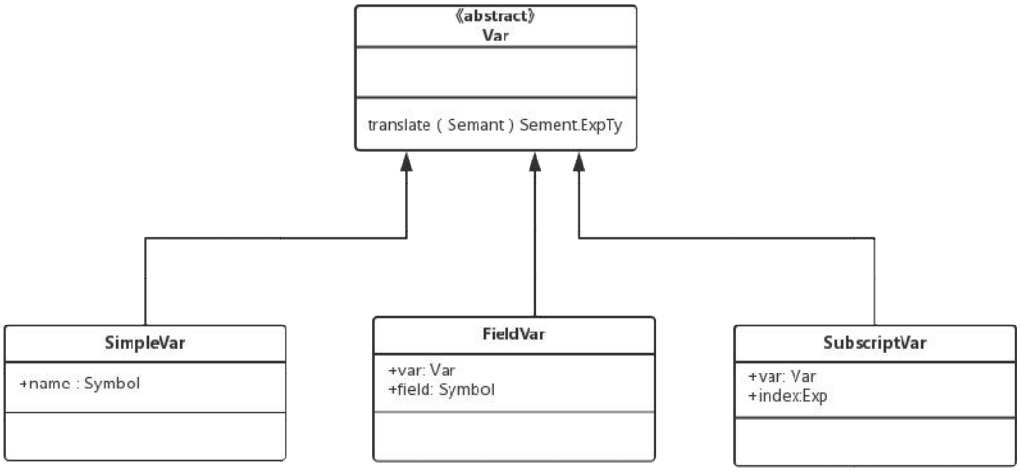
图：abstract Exp class



图：abstract Dec class



图：abstract Ty class



图：abstract Var class

4.3 FAQ

1. 如何构造抽象语法树？

在上一章我们提到 Cup 的语法如下：

```
exp :: = exp PLUS exp {:semantic action:}
```

在 semantic action 中我们进行中抽象语法树的描述，例如：

```
exp  :: =  expr: e1 PLUS: plus expr: e2
{: RESULT=new OpExp(plusleft,e1,OpExp.PLUS,e2); :}
```

2. 如何输出抽象语法树？

在官方框架中，Absyn 中有一个 print 类 用于输出树。

使用方法如下：

```
java_cup.runtime.Symbol s=p.parse();
Print print=new Print(System.out);
print.prExp((Exp)s.value, 0);
```

3. 在 VarDec 中有一个一个成员为“escape”，其含义是什么？

编译器的语义分析阶段需要知道那些局部变量会在被嵌套的函数中使用。varDec 的 escape 成员用于记录这种信息。在构造函数的参数中没有提及这个 escape 域，但是他总是被初始化为 TRUE，这是一个保守的近似值。field 类型既用于形式参数，也用于记录域（record fields）；escape 对形式参数有意义，但是对记录域来说可以忽视。

在抽象语法中使用 escape 域是一种“雇佣（hack）”，因为逃逸是全局的，非句法的属性。如果让 escape 位于 ABsyn 之外会导致需要另外的数据结构来描述逃逸属性。

4.4完成情况

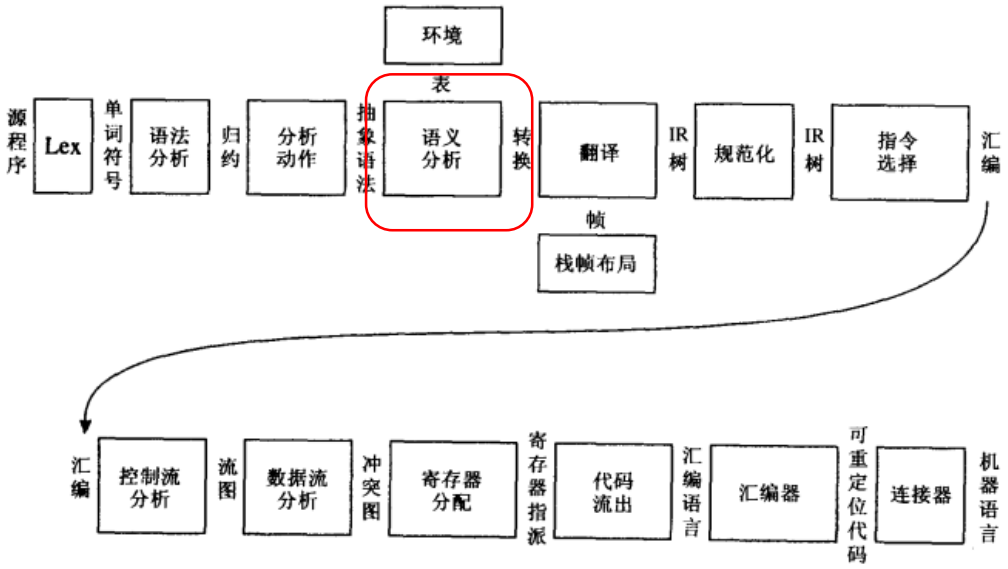
1. 输出抽象语法树
2. 完成了 Parse 模块的建立

4.5 参考资料

MCIJ Chapter 4.

第5章 语义分析

5.1本章任务：



语义分析的任务是：将变量的定义与它们各个使用联系起来，检查每个表达式是否有正确的类型，并将抽象语法转换成更加简单的，适合于生成机器代码的表示。

所以这一章的主要工作是符号表的管理，*符号表*，也称为环境，其作用是将标识符映射到他们的类型和存储位置。

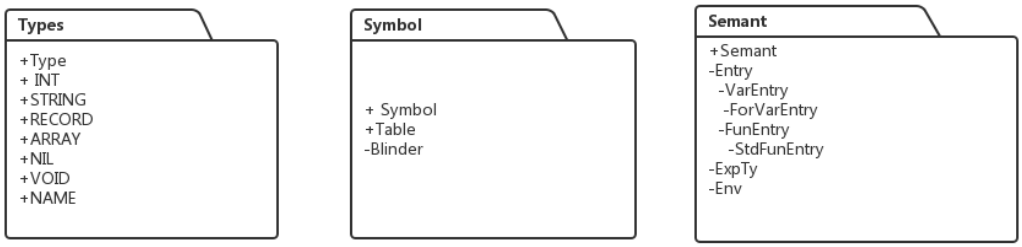
程序中的每一个局部变量都有一个*作用域*，该变量在此作用域中是可见的。当语义分析达到每个作用域结束时，所有局部于此作用域的标识符都将被抛弃。

环境是由一些绑定构成的集合，所谓绑定是指标识符域含义之间的一种映射关系。

我们的任务就是管理这些绑定。

5.1.1Tiger 编译器的符号

5.2 结构框架



图：语义分析用到的包图

5.3管理入口

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。

我使用了两个入口分别是 VarEntry 一个是 FunctionEntry 一个处理变量一个处理函数。

5.4管理环境

Symbol 模块中的类型 Table 提供从符号到其绑定的映射(使用 Blinder 类)。这样，我们将有一个类型环境（type environment）和一个值环境（value environment）

type environment：记录了当前的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

value environment：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 2 种入口

在初始化 tenv 时,要添加两种基本类型 int 和 string

在初始化 venv 时,要先把库函数（如 print, flush 等，详见 MCIJ 的附录）作为库函数添加进去

5.5作用域

每个局部变量都有一个作用域（Scope），变量在他的作用域中可见。当语义分析到达每个作用于的结束时，所有局部于此作用域的标识符都将被抛弃。

我们使用 beginScope 和 endScope 来进行这些操作，同时需要一个来做辅助操作（详见下一章）。

5.6语义分析步骤

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（表现在 AST 上据说其子节点），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.

在类型检查阶段，我们要为每一个标识符绑定一个含义。所以我们要对 AST 的节点进行转换。

AST	OUTPUT	解释
Absyn.EXp; Absyn.Var	Semant.ExpTy	将表达式转换为有类型的表达式
Absyn,Ty	Types.Type	将类型加入 Type
Absyn.FieldList	Types.RECORD	将域表定义加入 RECORD
Absyn.Dec	Translate.Exp	声明被翻译为 Translate.Exp
Absyn,FunctionDec	Types.Type	翻译返回值类型

5.6.1语义错误

语义分析阶段的错误种类很多，篇幅原因就不一一列出了。

大致有需要 int 类型的时候没有 int 类型 ,类型不一致 ,在不能引用 void 类型的时候引用 void 类型，调用的函数未定义等等。

5.7FAQ

1. 为什么要使用两个环境？

   例如对于程序：

```
let type a = int
   var a : a =5
   var b : a =a
   in b+a
end
```

符号 a 在预期类型标识符的语法上下文中表示类型“a”，在预期的变量的语法上下文中表示变量“a”。

2. 如何处理递归声明？

   对于一组相互递归的对象（类型或者函数），其解决方案是首先将所有这些对象的“头”放到环境中，得到一个环境 e1，然后在环境 e1 下处理所有这些对象的“体”。

   为了将这个头送入环境 tenv，我们为其绑定一个空的 NAME 类。

   如果递归声明出现如下情况

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

   含有一个非法的递归（相互递归），这个时候我们只需要调用 NAME 类里的方法 isLoop（）来判断是否成环。

3. 如何处理循环语句，for 和 while

这里我定义了一个 ForState 类 ,通过将循环压入栈弹出栈的方法确定在哪层循环。

4. 如何执行语义分析？

使用函数 transProg ( );

```
public Frag transProg(tiger.Frame.Frame globalFrame, tiger.Absyn.Exp e
xp) {
globalLevZero = new Level(globalFrame);
Level mainLevel = new Level(globalLevZero, tiger.Symbol.Symbol
     .symbol("t_main"), null);
    env.initialize();
ExpTy mainFun = transExp(mainLevel, exp);
translate.procEntryExit(mainFun.exp, mainLevel, false);
return translate.getResult();
```

procEntryExit 函数在后面细讲。

5.8完成情况

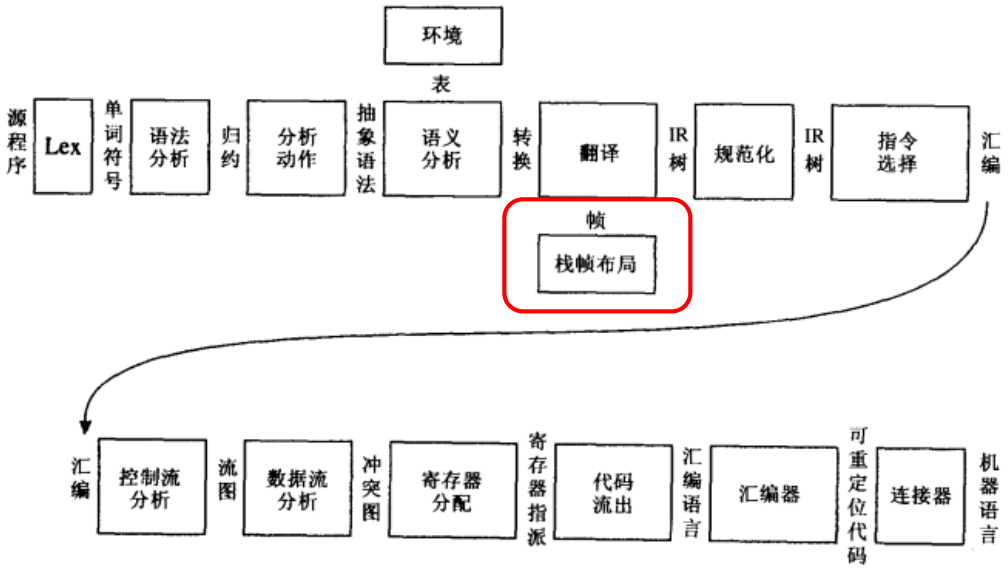
1. 实现一个简单的类型检查器和声明处理器
2. 能处理递归（和相互递归）的函数，（相互）递归的类型说明
3. 保证 break 语句的正确嵌套
4. Offical 的所有错误数据均能报错

5.9参考资料

- MCIJ Chapter 5 &Chapter 6
- *Tiger Language Specification* by Martin Hirzel and Kristoffer H. Rose，NYU，2013 *Section 5 Type Rule*

第6章 活动记录

6.1 本章任务



管理 TIger 的栈帧，为局部变量分配存储空间，并追踪嵌套层。

6.2 栈帧

下图描述了栈帧的基本结构

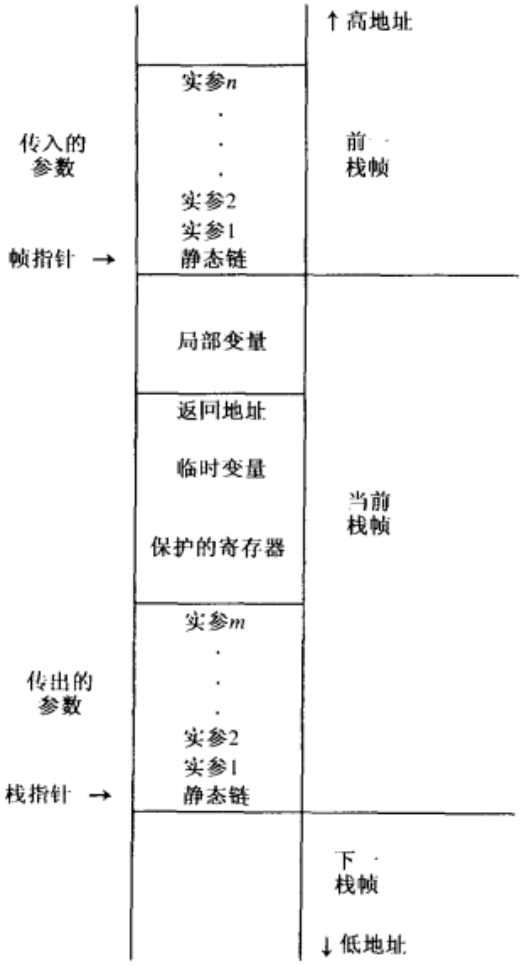


图 6-1 栈帧

6.2.1 栈帧中的变量

函数的参数将通过寄存器进行传递，返回地址将存放在一个寄存器中，函数的结果将被保存在寄存器中而返回。只有在如下状态下，他们才被写到栈帧里。

- 该变量将被作为传地址参数
- 该变量被嵌套在当前的过程内的过程访问
- 该变量的值太大以至于不能将它放到单个寄存器中
- 该变量是一个数组，为了引用器元素需要进行地址运算
- 需要使用存放该变量的寄存器作为特殊用途，如传递参数

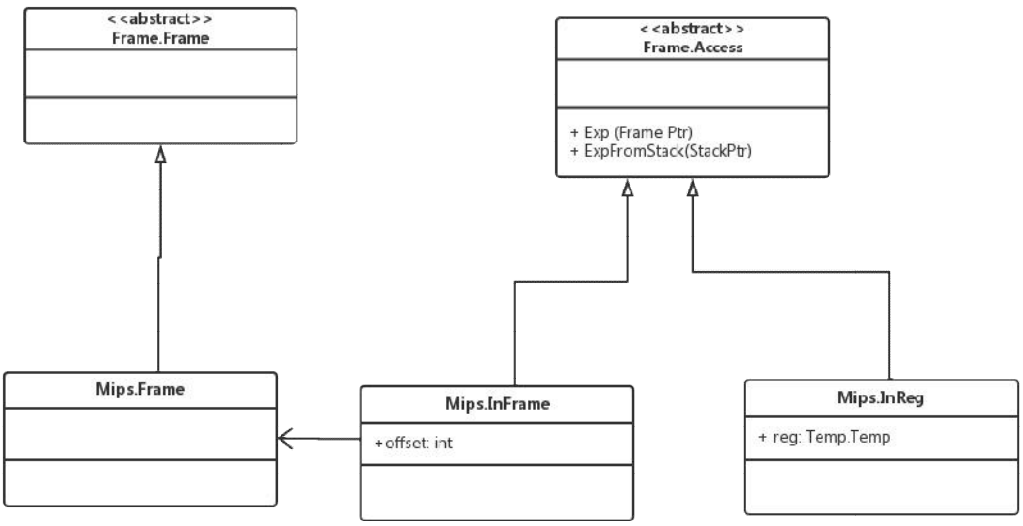
- 存在太多的局部变量和临时变量，溢出寄存器。

如果一个变量是传地址的实参，或者它被取了地址，或者内层的嵌套函数对其进行了访问，那么我们称这个变量是逃逸的。

6.3 Tiger 的栈帧

为了不在语义分析模块的实现中强行塞入任何特定的机器的规定，我们使用使用*抽象方法*。

在后面我们会把这个抽象的 Frame 类实现为一种特定的目标机器。在我们这个项目中我们将它实现为 Mips 在 Package Mips 的 Mips.Frame 进行实现。



因为上一节的 Frame 的结构我们的 Frame 类中应该存储函数的参数、 返回地址、寄存器、帧指针、 栈指针、 返回值等重要信息：

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,它被设置成是抽象数据类型。

因为本节书写的 Frame 只是一个抽象的类，所以我们在第 9 章 指令选择中在仔细的说，

Frame 是如何实现的。

6.4 临时变量(Temp)与标号(Label)

Temp 是局部变量的抽象名

Label 是静态存储器地址的抽象名

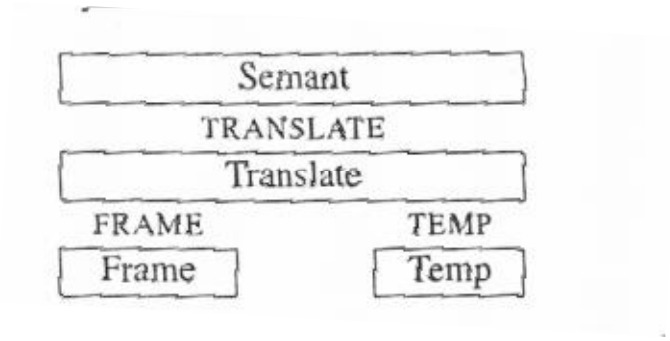
package Temp 管理这两种不同名字组成的集合

new Temp.Temp ()从临时变量的无穷集合中返回一个新的临时变量

new Temp.Label() 从标号的无穷集合中返回一个新的标号。

new Temp.Label(string) 返回一个汇编名为 string 的新标号。

6.5 层



Frame 和 Temp 包提供存储器变量和寄存器变量的与机器无关视图。Translate 模块通过处理嵌套作用域的表示来扩大视图

所以我们需要一个抽象层将源程序的语义与机器相关的栈帧布局分开

在语义分析阶段，transDec 通过调用 new Translate.Level（parent，name，formals）为每个函数创建一个“嵌套层”，而新的 Translate.Level 调用 new Frame.Frame 建立一个新的栈帧。Semant 将这个嵌套层保存在该函数的 FunEntry 数据结构中，以便当他遇到一个函数调

用时，能够将这个被调用的嵌套层传回给 Translate

```
public class FunEntry extends Entry //中加入
   tiger.Translate.Level level;

public class VarEntry extends Entry //中加入
   tiger.Translate.Access access;
```

这里 Access 是由层次 level 和 Frame.Access 组成的对偶对

```
public class Access //中有
   Level home;
   tiger.Frame.Access acc;
```

所以层的属性需要有：

```
tiger.Frame.Frame frame;
// 帧
Level parent;
//直接上级层
public AccessList formals;
//参数
```

为了追踪层次的信息，transDec 需要得到一个指明当前层次的额外参数，同样 transExp 也需要这个参数。同样 transVar 也需要这个参数。

6.6 逃逸变量

非逃逸的局部变量可以分配到寄存器中，而逃逸的变量一定要分配到栈帧中。

使用遍历抽象语法树的方法，寻找每一个变量的逃逸使用。

6.7静态链

Translate 负责管理静态链。

l 是指明其参数是否逃逸的布尔量组成的表，则

l1= new BoolList (true , l) 是一个新表。

true 表示 说明作为“额外参数”的静态链是逃逸的。

6.8 完成情况

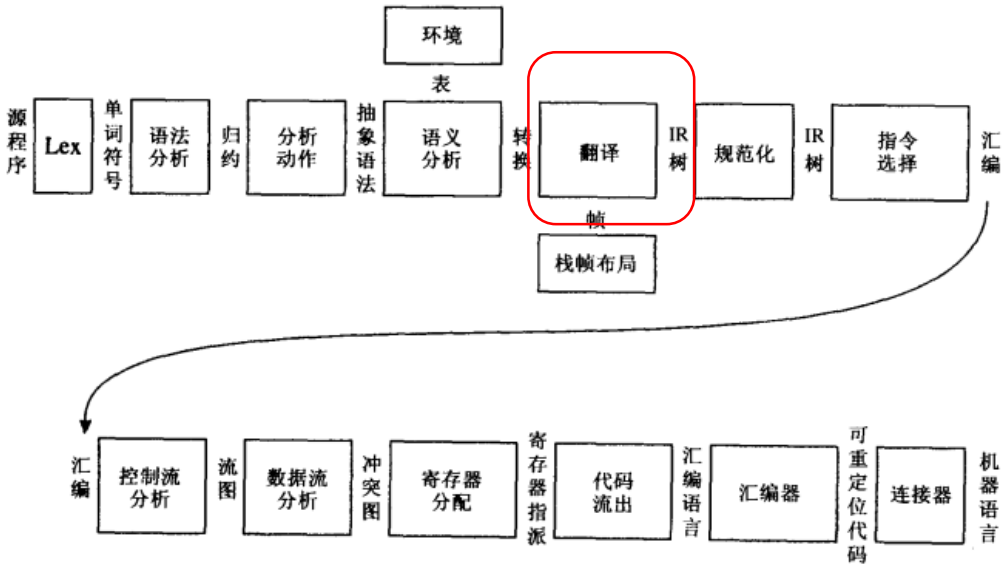
1. 修改了 Semant 为局部变量分配了存储空间，并追踪了嵌套层
2. 实现 Findescape 模块，设置抽象语法中每个变量的 escape 域。

6.9 参考资料

MCIJ Chapter 6

第7章 翻译成中间代码

7.1 本章任务



将抽象语法转换成抽象机器代码。产生中间表示，IR(intermediate representation)树。

完成前端。

7.2 IR tree

官方代码框架中给了 Tree 包其中包含如下类：

Print： 输出 IR 树

AbsExp 和 Stm 是两个抽象类 他们的区别在于有无返回值

AbsExp：代表某个值的计算，是有返回值的。

- CONST (i) 整数常量 i
- NAME (n) 字符常量 n
- TEMP (t) 临时变量 t，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

Stm： 树中间语言的语句 Stm 执行副作用和控制流，无返回值

- 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

AsbExpList 和 StmList 分别是 AbsExp 和 Stm 的链表

7.3 Translate

7.3.1 表达式的种类

- Ex “表达式”，表示 EXP （AbsExp）
- Nx “无结果语句” 表示为 Tree 语句（Stm）
- Cx “条件语句” 表示为一个可能转移到两个标号之一的语句（Stm）

我们还需要处理三种表达式间的转换，函数为转换到 unEx，unCx，unNx；

在进行这些转化时，我做的事情就是把其中的逻辑用 Stm 和 AbsExp 中的表达式表示出来。例如 Cx->Ex 的转化。

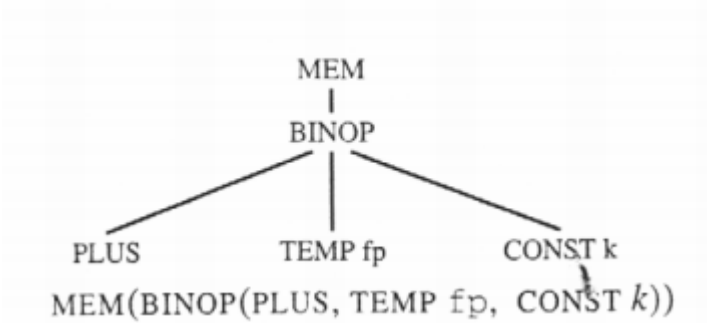
```
Temp r = new Temp();
TEMP rr = new TEMP(r);
Label t = new Label();
Label f = new Label();
//rr=1
//if (exp!=0)goto T else goto F
//LABLE f
//rr=0
//LABEL t
//return rr
return new ESEQ(new SEQ(new MOVE(rr, new CONST(1)),
new SEQ(unCx(t, f),
      new SEQ(new LABEL(f),
new SEQ(new MOVE(rr, new CONST(0)),
           new LABEL(t))))), rr);
```

同样用这样的想法翻译 Translate 包中的其他表达式

7.3.2 翻译过程

1. 简单变量

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

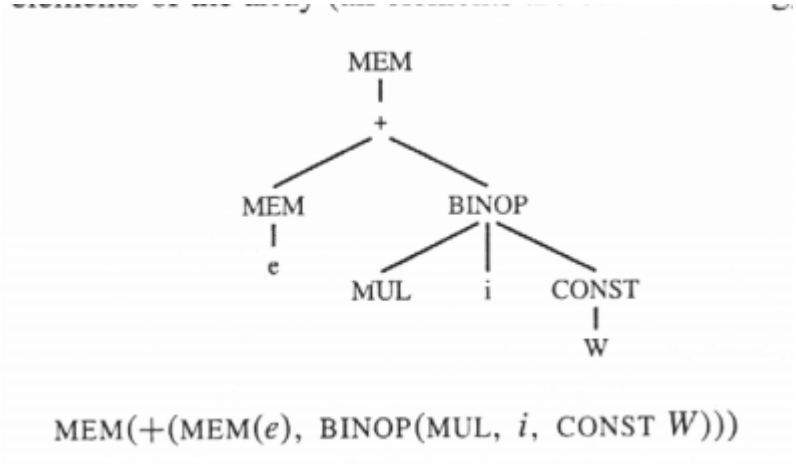


2. 左值

   MEM（+（TEMP fp,CONST kn ）,S ）

3. 下标和域

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

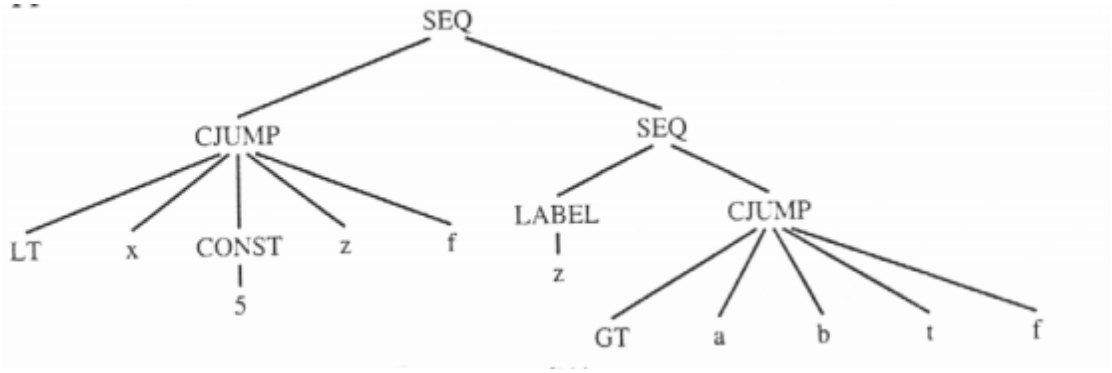


4. 算数操作

   EX（BINOP（op , left , right ））

5. 条件表达式

   if ....then....else,,,,,
   SEQ(s1(z,f),SEQ(LABEL,z,s2(t,f)))



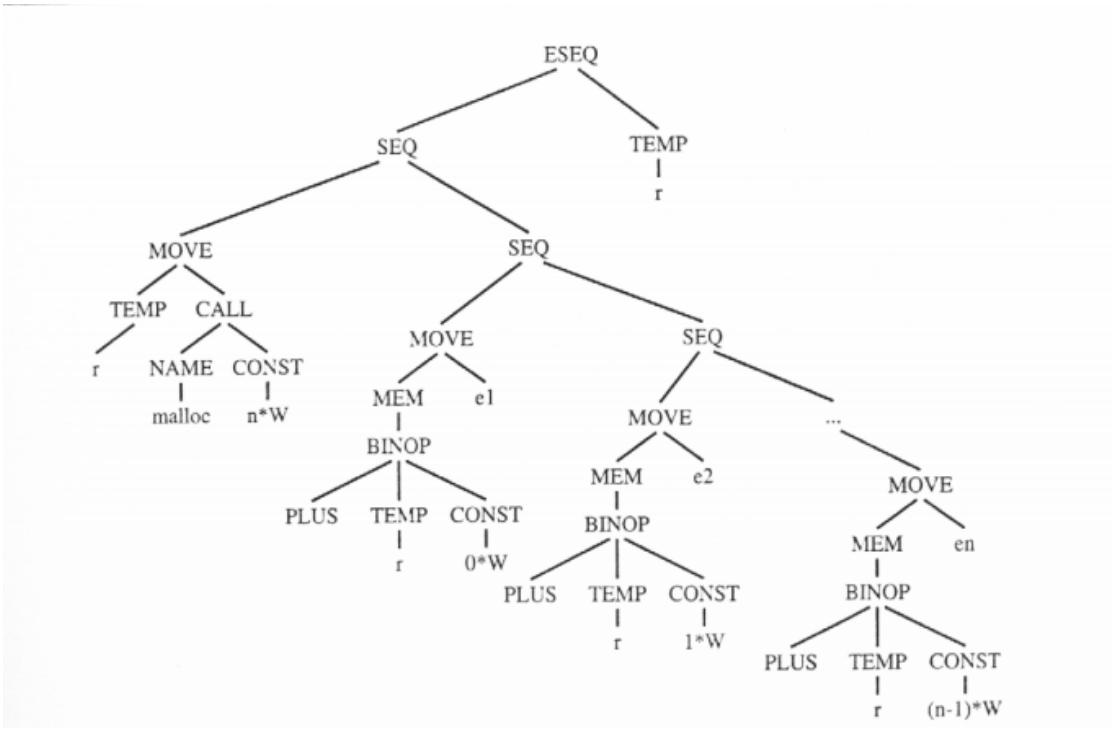
6. 字符串运算

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

其中 COMP 是调用外部函数 stringCompare 得出的比较结果

7. 记录和数组

记录分配



8. while 循环

封装在 WhileExp 中

WhileExp（test，body，done）

9. for 循环

封装在 ForExp 中

ForExp（home，var，low，high，body，done）

10. 函数

将静态链作为一个隐含的参数传递

CALL（NAME $l_f[s_l e_1 e_2 \ldots\ldots e_n]$）

11. 翻译函数体:

由 procEntryExit 函数(在后面详细说这个重要的函数)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

7.3.3 片段（Fragment）

给定一个有嵌套层次 level 和一个已经被翻译好的函数体表达式组成的 Tiger 函数的定义

- 栈帧 一个栈帧描述字，它包含有关局部变量和参数的机器相关信息
- 函数体：从 procEntryExit1 返回的结果

这两个信息成为一个片段

片段可以自成一链

字符串为 DataFrag

```
public class DataFrag extends Frag {
   public String data;
   DataFrag(String d, Frag n) {
      data = d;
      next = n;
   }
}
```

函数块为 ProcFrag

```
public class ProcFrag extends Frag {
```

```
public Stm body;
public Frame frame;
   public ProcFrag(Stm b, Frame f, Frag n) {
          body = b;
          frame = f;
          next = n;}}
```

7.4 FAQ

1. 类型检查与中间代码生成的关系

   在检查代码的时候我们为了不讨论中间代码生成，我们定义了一个虚的 translate 并且对这种类型都使用 null

   但是当我们完成了 translate 之后我们就可以简单的调用 translate 中的相应函数即可。

   而且二者名字上有很好的对应，易于书写。

7.5 完成情况

1. Offical 所有正确代码都可以输出 IR 树
2. 至此前端基本完毕

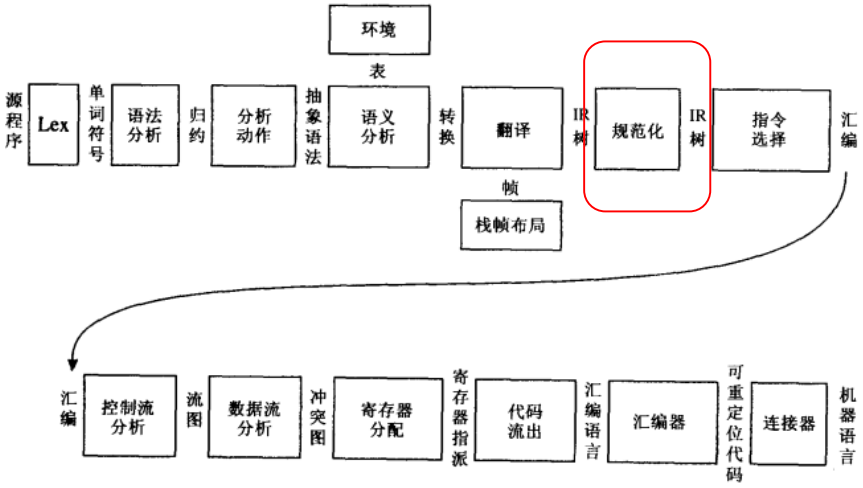
第8章 基本块和轨迹（规范）

8.1 本章任务

将 IR 树写成一个没有 SEQ 和 ESEQ 结点的规范树表

根据该表划分基本块

基本块被顺序放置，所有的 CJUMP 都有 false 标号



8.2 Canon

官方框架中的 Canon 包已经做了这件事情，只需要合理的运用这个包即可。

```
 debug.println ("# After canonicalization: ");
tiger.Tree.StmList stms = tiger.Canon.Canon.linearize(f.body);

debug.println("# Basic Blocks: ");
tiger.Canon.BasicBlocks b = new tiger.Canon.BasicBlocks(stms);

print.prStm(new tiger.Tree.LABEL(b.done));
debug.println("# Trace Scheduled: ");
tiger.Tree.StmList traced = (new tiger.Canon.TraceSchedule(b)).stms;
```

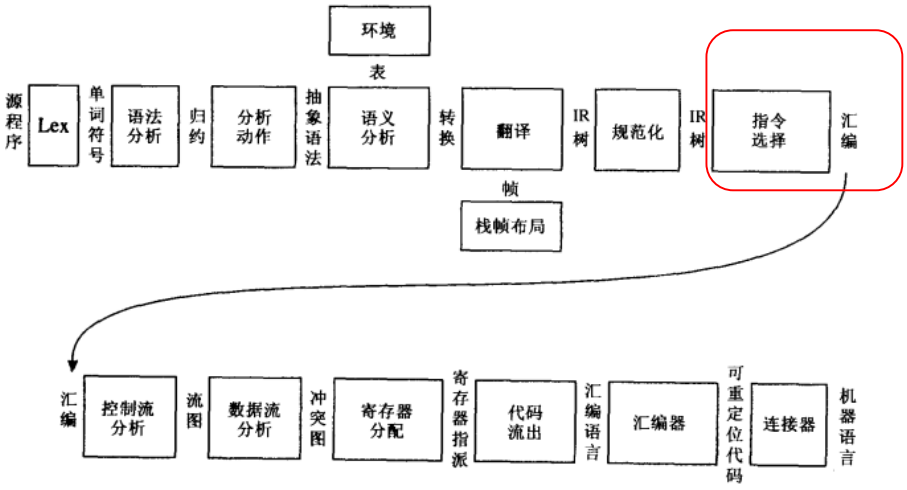
8.3 完成情况

对于官方给的所有正确的数据，均可生成规范化后的 IR 树

与 ir 树一起写入.ir 文件

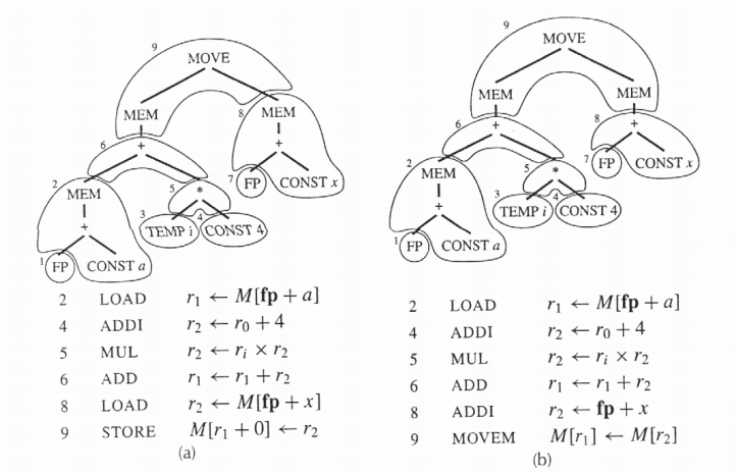
第9章 指令选择

9.1 本章任务



我们需要实现第 6 章虚的 Frame 类，并将找到实现 IR 树的一个机器语言。

可以将一个及其语言指令表示成 IR 树的一段树枝，称之为树形，所以我们这个阶段的任务转换为用树形的最小集合来覆盖一棵树。



9.2 常见 MIPS 指令

存取指令 sw、lw

移动指令 move

算数运算 add、sub、mul、div

逻辑运算 or、and

条件跳转 beq 、bgq、bgt、ble、blt、bne

无条件跳转 jal、 jr

立即数操作 li addi subi

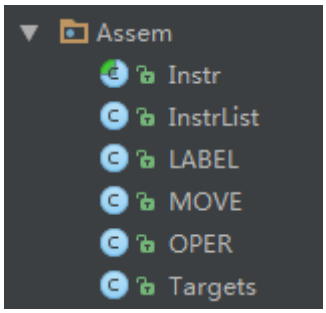
9.3 Maximal Munch 算法

这是一个求覆盖的算法。

主体思想是：从输的根部节点开始，寻找适合他的最大瓦片。用这个瓦片覆盖根节点，同时也可能会覆盖根节点附近的其他几个节点。覆盖根节点后留下若干子树，对每个子树重复相同的算法。

我们用这个算法对经过规范化的 IR 树进行覆盖。

9.4 Assem 包



Instr 是抽象类 派生了 LABEL（跳转地址）、MOVE（转移）OPER（操作），其中 OPER 中含有 Tager 类，用于处理 jump 指令。

9.5 Mips.Frame

需要把第 6 章的 Frame 实例化。

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。

Mips 中的寄存器：如下图描述

在 Mips.Frame 中我们需要对下面这些寄存器进行初始化，并新建帧为帧分配 Access 这个时候我们需要考虑 Access 是不是逃逸的。

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)

9.6 函数调用

调用者为 Caller 被调用者是 Callee

一个函数调用有着如下步骤

- 传递参数（前三个参数放入寄存器$a0-$a3）
- 保存 Caller-Saved Reg（$t0-$t9）和参数 Reg （$a0-$a3）
- 执行 jal 指令 跳转到 Callee 的第一条指令

  进入 Callee
- 分配帧
- 保存 Callee-Saved 寄存器（$s0-$s7），帧寄存器 $fp 和返回地址寄存器 $ra
- $fp=$sp-帧空间+4 byte

  准备返回 Caller
- 返回值放入$v0
- 恢复 Callee-Saved
- 将$sp 加上相应的帧空间
- 跳转返回地址

这些由 ProcEntryExit 函数组来完成。

procEntryExit1：保存/恢复 原 fp；计算新 fp；保存/恢复 ra；保存/恢复 Callee-Saved 保存参数

ProcEntryExit2 增加空指令

ProcEntryExit3 设置函数体标号；分配帧空间；将$sp 加上相应的帧空间；跳转到返回地址

9.7 FAQ

1. tempMap 的意义是什么？

一直没弄清楚 tempMap 的具体意义是什么，虎书第 12 章写到：“tempMap”与该寄存器对应的“预着色临时变量”。tempMap 给出每个预着色临时变量的颜色。

猜测应该与图着色有关。

9.8 完成情况

对于官方给的所有正确的数据，均可生成不需要使用寄存器名的“汇编语言”

与 ir 树和规范 ir 树一起写入.ir 文件

9.9 参考资料

MCIJ *Chapter 9*& *Chapter 6*& *Chapter 12*
The MIPS Info Sheet

计算机组成与设计 *第二章*

第10章 总结

10.1 整合

Driver.java 程序是整个编译器的**入口**

运行后会将**错误**输至显示屏（由 ErrorMsg 提供方法），无错生成**.abs 文**件（抽象语法分析树）；**.ir 文件**（包括 IR 树，规范化的 IR 树，生成的指令）

他使用了 Package Main 中的 **Main** 类，在这里我建立并链接了各个模块

Main 中使用了封装了 parse 模块的 **parse.Runable**，用于检查词法和生产语法分析树。

链接各模块时要注意错误信息的传递。

10.2整体结构

10.3 感想

1. 很佩服设计这个编译器框架的人，我即使只是在框架上添加代码，就已经被折磨的不行了。这种构建整体框架的能力还需要锻炼
2. 因为之前没有详细的学习过 Java，最开始使用 Java 是有些排除的。但是随着工程的进行，尤其还在我使用中文版的《Modern Compil Implementation in C》进行对照的时候，我深刻的感受到了 Java 这个面向对象语言的强大。
3. 课程之间是相通的。数据结构的在工程中的应用是肯定存在的，计算机组成（Mips 知识）、嵌入式（汇编语言，栈帧）课上学的知识也得到了很好的应用。
4. 还是有点遗憾只做到这里，我翻看了后面的书，我感觉前半部分更多的设计了合适的数据结构来存储信息，后面的部分就涉及到了一些很有意思的算法，而这些算法大多都是有着启发性的。所以如果寒假有机会我想我继续写下去。
5. 最后，谢谢助教老师的安排和耐心的看到最后。报告中有一部分是边写工程边学习知识边意识流一般的写到报告中的。所以报告略显繁杂。谢谢阅读~