上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

编译原理课程设计

Practicum in Principles of Compilers

项目报告

Project Report

姓　　名: 包伟铭

学　　号: 5140219191

专　　业: 计算机科学与技术

指导教师: 沈耀

学　　院: 电子信息与电气工程学院

目录

1 项目概览

1.1 综述

本课程项目旨在实现一个简单 Tiger 语言编译器。

1.1.1 开发语言

JAVA

1.1.2 工具

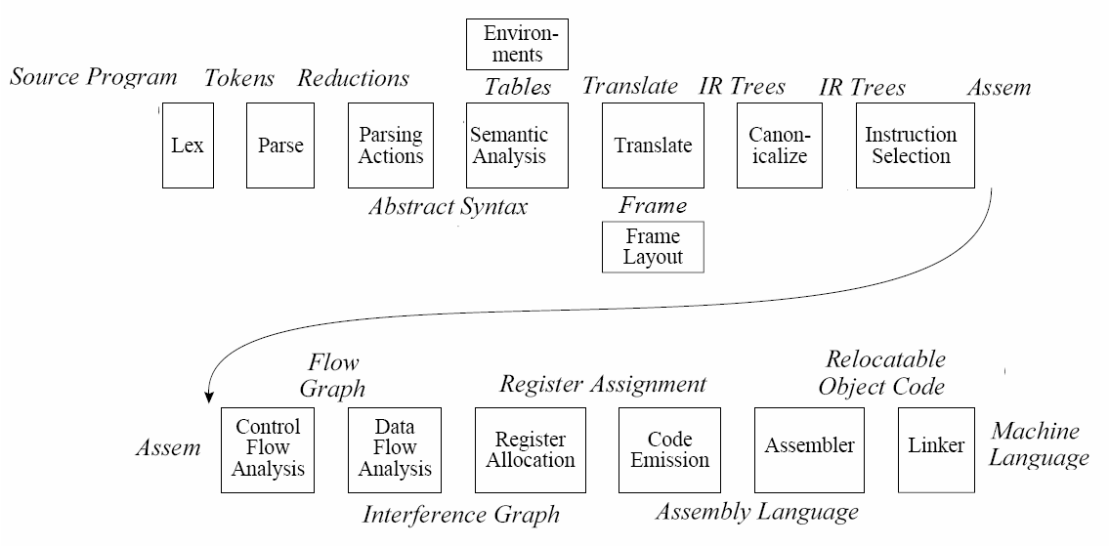
- Java 开发环境：Eclipse；
- 词法分析器生成工具：JFlex；
- 语法分析器生成工具：JAVACUP；
- MIPS 语言模拟器：QtSpim

1.1.3 项目完成情况

符合计划要求，已完成最终目标代码生成，可将较复杂的 tiger 程序（如八皇后等）编译生成 MIPS 汇编语言文件，在模拟器上运行，输出正确结果。

1.2 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。



【图 1】编译器主要步骤

1.3 TIGER 语言介绍

Tiger 语言是一个简单小巧的指令式编程语言，有整形数 (Integer) 和字符串 (String) 变量，数组 (Array)，记录(Record)，可嵌套的函数 (Function)。该语言没有语句的概念，程序由表达式组成。

Tiger 由 Andrew Appel 在著名的"虎书"——《Modern Compiler Implementation in Java[1]》一书中定义，具体规范细节参见 Prof. Stephen Edwards 撰写的《Tiger Language Reference Manual》。

- 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：type typeA={a:typeB}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

[1] Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

- 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** …

2 词法分析

2.1 词法分析目标

词法分析将读入的程度源代码划分为 Token，供之后的语法分析使用。在部分编译器中，词法分析同时，初步构建符号表。

以一段简单的 Tiger 代码为例：

```
let
    var a := 1
    var b := 2
in
    a+b
end
```

进过词法分析后，将输出如下的 Token 流：

```
LET 0                INT 28
VAR 6                IN 31
ID 10                ID 36
ASSIGN 12            PLUS 37
INT 15               ID 38
VAR 19               END 41
ID 23                EOF 44
ASSIGN 25
```

2.2 实现方案 – JFLEX

2.2.1 JFlex 简介

JFlex 是一个面向 JAVA 的词法分析器生成工具，同时也由 JAVA 语言实现。通过输入特定的正则表达式及其对应的行为代码，JFlex 可以生成对应的词法分析工具。

JFlex 基于确定有穷自动机（DFAs）。

2.2.2 .flex 文件书写

JFlex 代码分为以下三个块，块与块之间由%%隔开：

- **保留代码段**——该段代码会保留到生成的词法分析器中。Tiger 语言词法分析器中对应代码如下：

```
package tiger.parse;
import tiger.errormsg.ErrorMsg;
```

- **JFlex 定义段**

```
%%
%function next_token
%type java_cup.runtime.Symbol

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

/* JFLex definitions: */
%cup // 配合JAVACUP，供语法分析使用
%char
%state STRING
%state STRING1
%state COMMENT

%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}
```

```
LineTerminator = \r|\n|\r\n|\n\r
Identifier = [a-zA-Z][:jletterdigit:]*
DecIntegerLiteral = [0-9]+
WhiteSpace = [ \t\f]
```

- **正则表达式及行为列表段**

```
%%
/* Regular expressions and actions: */
<YYINITIAL> {
    {LineTerminator}   { newline(); }
    {WhiteSpace}   { /* do nothing */ }

    /* Token : Keywords */
    "array" { return tok(sym.ARRAY, null); }
    "break" { return tok(sym.BREAK, null); }
    "do" { return tok(sym.DO, null); }
    "else" { return tok(sym.ELSE, null); }
    "end" { return tok(sym.END, null); }
    "for"  { return tok(sym.FOR, null); }
    "function" { return tok(sym.FUNCTION, null); }
    "if"  { return tok(sym.IF, null); }
    "in" { return tok(sym.IN, null); }
    "let" { return tok(sym.LET, null); }
    "nil" { return tok(sym.NIL, null); }
    "of" { return tok(sym.OF, null); }
    "then" { return tok(sym.THEN, null); }
    "to" { return tok(sym.TO, null); }
    "type" { return tok(sym.TYPE, null); }
    "var" { return tok(sym.VAR, null); }
    "while" { return tok(sym.WHILE, null); }

    /* Token : Identifiers */
    {Identifier} { return tok(sym.ID, yytext()); }

    /* Token : Integer */
    {DecIntegerLiteral} { return tok(sym.INT, new Integer(yytext())); }
    // should we check very long integer in there?
    /*[0-9]+ { return tok(sym.INT, new Integer(yytext())); }*/
    // or not?
    /*[0-9]+ { return tok(sym.INT, new String(yytext())); }*/

    /* Token : String */
    \" { string.setLength(0);yybegin(STRING); }
```

```
    /* Token : SEPARATORS AND OPERATORS */
    "," { return tok(sym.COMMA, null); }
    ":" { return tok(sym.COLON, null); }
    ";" { return tok(sym.SEMICOLON, null); }
    "(" { return tok(sym.LPAREN, null); }
    ")" { return tok(sym.RPAREN, null); }
    "[" { return tok(sym.LBRACK, null); }
    "]" { return tok(sym.RBRACK, null); }
    "{" { return tok(sym.LBRACE, null); }
    "}" { return tok(sym.RBRACE, null); }
    "." { return tok(sym.DOT, null); }
    "+" { return tok(sym.PLUS, null); }
    "-" { return tok(sym.MINUS, null); }
    "*" { return tok(sym.TIMES, null); }
    "/" { return tok(sym.DIVIDE, null); }
    "=" { return tok(sym.EQ, null); }
    "<>" { return tok(sym.NEQ, null); }
    "<" { return tok(sym.LT, null); }
    "<=" { return tok(sym.LE, null); }
    ">" { return tok(sym.GT, null); }
    ">=" { return tok(sym.GE, null); }
    "&" { return tok(sym.AND, null); }
    "|" { return tok(sym.OR, null); }
    ":=" { return tok(sym.ASSIGN, null); }

    "/*" { count=1; yybegin(COMMENT); }
    "*/" { err("Comment symbol don't match!"); }
    [^] { return tok(sym.error, yytext()); /* err("Illegal character <
"+yytext()+" >!");*/ }
}

<STRING> {
    \" { yybegin(YYINITIAL); return tok(sym.STRING,string.toString()); }
    \\[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} {}
    " " {}
    \\ { yybegin(STRING); }
    \" { err("\\dont match"); }
    [^] { string.append(yytext()); }
}

<COMMENT> {
    "/*" { count++; }
    "*/" { count--; if (count==0) { yybegin(YYINITIAL); } }
    [^] {}
}
```

2.2.3 使用.flex 文件生成 java 类

JFlex 自带 GUI。安装完成后，可运行 bin/JFlex.bat 批处理文件，在 GUI 中选择源.flex 文件及相关设置即可。JFlex 生成 Yylex.java 类供后续操作使用。

2.3 TIGER 词法 DFA

JFlex 工具可生成定义语言的 DFA 图，Tiger 语言的最小化 DFA 包括 109 个状态节点，1599 条边，Graphviz dot 文件见“$工程目录/tiger/parser/dfa-min.dot”。

2.4 难点与思考

词法分析是编译器的第一步，同时也是整个编译器实现过程中最简单的一步。这一步中只需要学习 JFlex 文件的写法，配合 Tiger 语言 Reference Manual 中的细节，仔细即可。

本项目解决的问题包括：

- **注释的嵌套**——通过状态变量 COMMENT 和计数器 count 实现，前者当读入“/*”时进入，开始计数，count 设为 1。注释层次由 count 增减记录。当 count 为 0 时，即退出 COMMENT 状态。
- **转义字符**——转义字符在 STRING 状态中处理，对于“\n”，“\t”等直接转义处理，“\122”等判断数字序号是否超出边界[0, 255]，若无则转义，否则报错。

在项目过程中注意到：

- **%line 与%column——**在 Tiger.flex 中声明%line 和%column 两个开关量后，JFlex 会在扫描时自动维护两个对应的变量 yyline 和 yycolumn，以表示当前扫描到的行与列，方便调试。

3 语法分析

3.1 语法分析目标

语法分析部分以词法分析的结果作为输入，进行语法分析，输出抽象语法树。

语法分析的中心思想依然是有穷自动机。根据 LR 语法，将对应的语法用实现其相应自动机。首先写出对应语法，而后利用 JAVACUP 来生成对应的自动机。依次读出词法分析结果中的词法单元，在自动机的控制下按照语法规约，并在规约过程中通过 result 的传递得到对应的语法树，生成输出抽象语法树。

以 2.1 词法分析目标中的例子为例，抽象语法分析树如下：

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

3.2 实现方案 –JAVACUP

3.2.1 JAVACUP 简介

JAVACUP 是 Java 语言实现的 CUP 解析器生成器，可大大简化。

词法分析工具的核心是产生式的定义，通过 JAVACUP 工具可实现对应词法分析器（类）的自动生成。

Eclipse 中有 CUP 编写插件 CUP Parser Generator，可高亮语法，提供元数据，检测冲突，生成行为表以及最小化行为表，同时支持简单的 debug 操作。

3.2.2 .cup 书写

.cup 文件中需要定义的包括：上下文无关文法的各要素，包括：终结符，非终结符，产生式及开始符号，以及在“{: :}”指定对应抽象语法树生成的对应操作（该部分需结合抽象语法树维护 Absyn 包及符号表维护 Symbol 包中接口，分别在 3.2.4 及 4.2.1 中介绍）。

.cup 文件结构包括：保留代码段，行为代码段，词法解释器代码段，终结符段，非终结符段，优先级定义，开始符号声明，以及产生式文法段。

- Tiger 语言的终结符定义如下：

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

- Tiger 语言的非终结符定义如下：

```
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;
```

- Tiger 语言优先级定义如下：

```
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 LPAREN;
```

- 开始符号声明：

```
start with program;
```

- 产生式文法：

产生式部分包括上下文无关语法以及对应的翻译动作，翻译动作同样由“{: :}”包围，紧跟对应的产生式。示例代码如下：

```
type::= ID:i {: RESULT = new NameTy(ileft, 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(i)); :}
      ;
```

由于 Tiger 语言支持嵌套的函数这一点需注意。根据 Tiger 的 reference manual，产生式文法部分包括以下 14 项：

1. expr
2. lvalue
3. exprlist
4. exprseq
5. filedlist
6. declarationlist
7. declaration
8. typedeclist
9. typedeclaration
10. type
11. typefields
12. variabledeclaration
13. functiondeclist
14. functiondeclaration

3.2.3 使用.flex 文件生成 java 类

- 安装 JAVACUP，配置完成环境变量后，在对应目录命令行下输入：

```
java java_cup.Main –version
```

可得到当前 JAVACUP 的版本号。

- 在对应目录命令行下输入：

```
java java_cup.Main -expect 2 -dump < <filename>
```

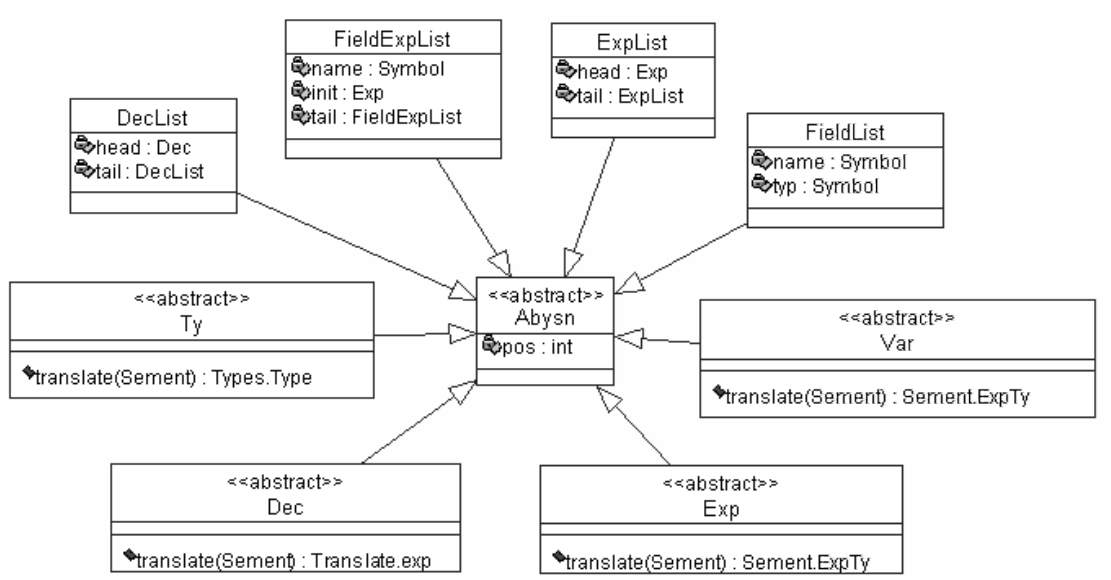
即可在 JAVACUP 目录下生成语法分析器 parser.java 及 sym.java，其中：

-expect	表示允许冲突数量；
-dump	参数显示编译过程；
<	标记表示输入文件（适用于 JAVACUP 0.10k 及之前的版本）；
<filename>	即替换为.cup 文件。

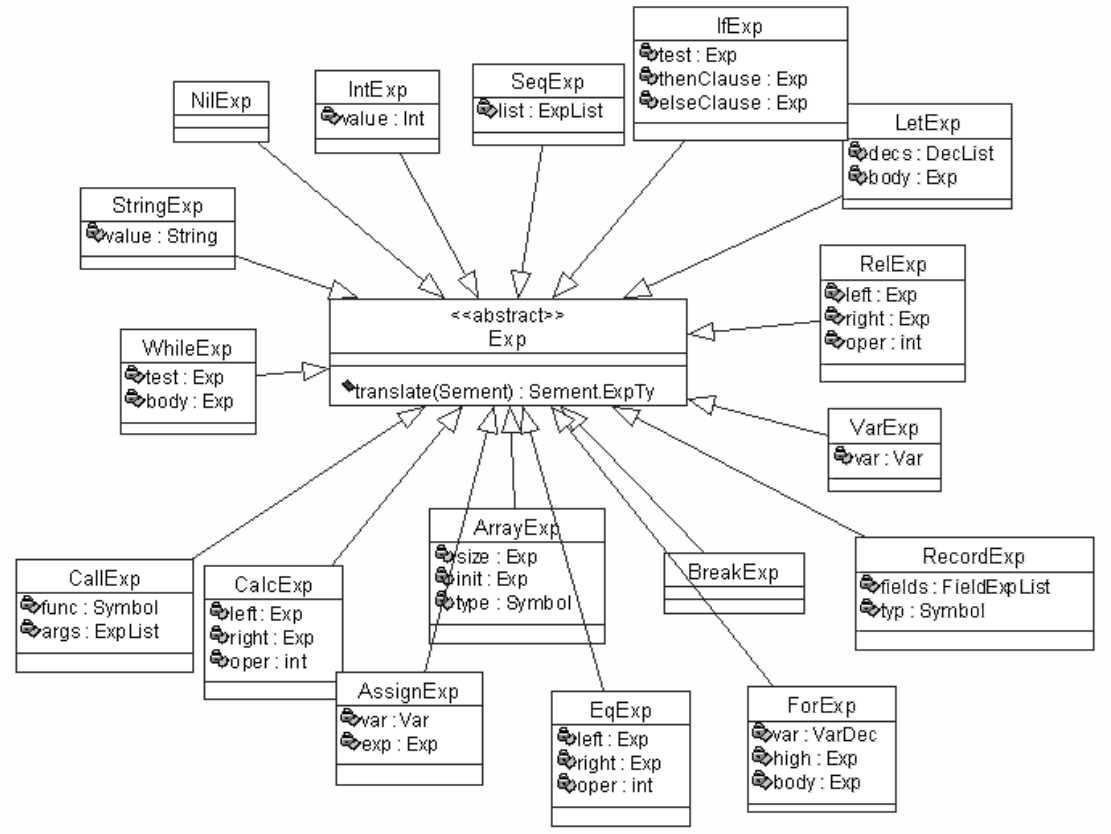
3.2.4 抽象语法树节点维护 – Absyn 包

Absyn 包中维护了抽象语法树的节点信息，所含类及其接口类图如下：

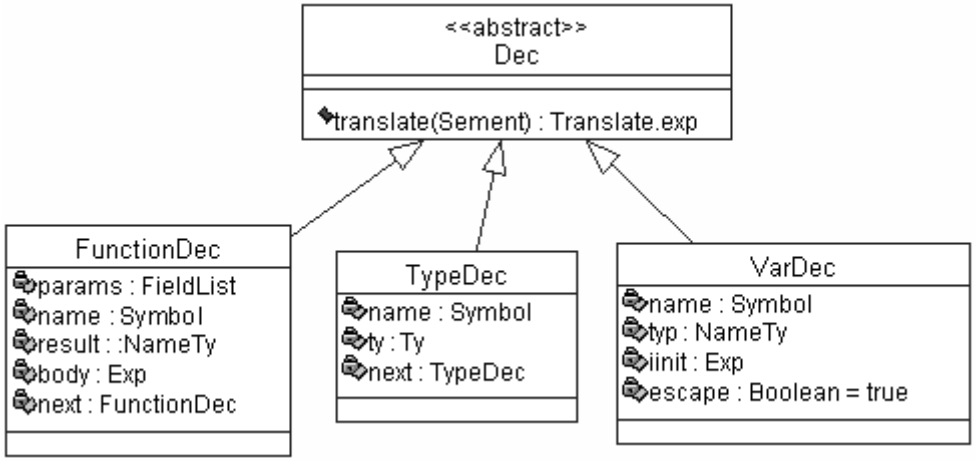
- Absyn 抽象类及接口：



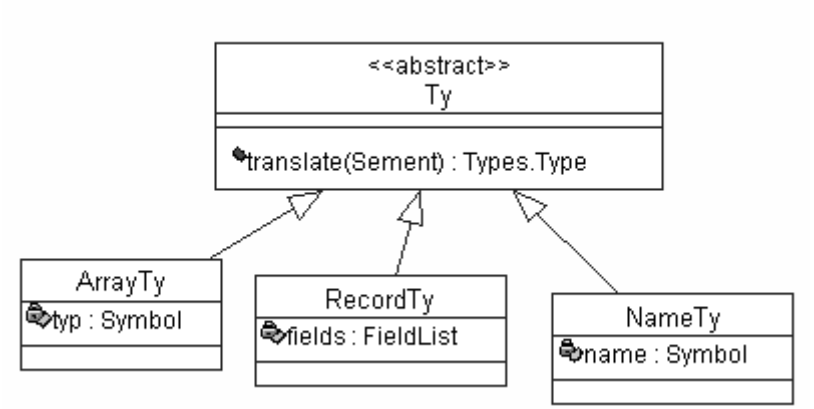
- Exp 抽象类及接口：



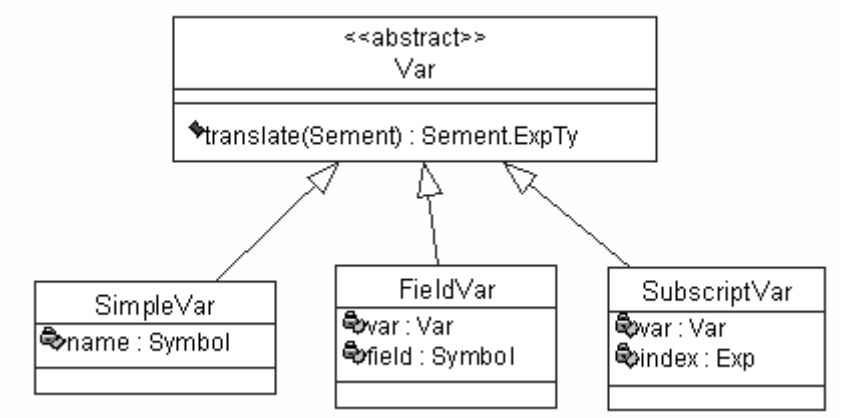
- Dec 抽象类及接口：



- Ty 抽象类及接口：



- Var 抽象类及接口：



3.2.5 词法分析器与语法分析器的链接

- 词法分析器中声明%cup 使 Yylex implements java_cup.runtime 类包，见 2.2.2。
- 语法分析器中声明 scan 函数，调用 Yylex 类中的 next_token()方法。

3.3 难点与思考

尽管 Tiger 近是一门简单的语言，但其所含的产生式也不算少，编写.cup 文件大量的产生式对应代码时时有错误，所幸借助 Eclipse CUP 插件，这一部分的编写少了不少麻烦，使用 tiger_runnable 中提供的 CupTest 类方法测试 testcase 时也相对顺利。

尤其针对 reference manual 中右下角带 opt 角标的产生式，由于参数，表达式列表可以为空，需把为空的情况单独列出处理。如 if-then 与 if-then-else：

```
| 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); :}
```

4 语义分析

4.1 语义分析目标

语义分析旨在对上阶段生成的抽象语法树，即通过语法检查的代码进行逻辑、语义检查。检查过程中，在本项目实现里构建符号表，若检查通过则一并供接下来的中间代码生成模块产生中间代码树（IR Tree），若检查错误，则通过 ErrorMsg 类输出对应语义错误信息。

从功能层面说，语义分析及中间代码（中间代码树）生成模块是分开的，但在实现的时候不分先后，一步完成。语义分析通过的部分立刻翻译，再继续递归的分析翻译接下来的部分。这一章节针对语义分析部分。

4.2 符号表

4.2.1 符号表维护 – Symbol 包

符号表维护由框架代码提供的 Symbol 包实现，含 Symbol 类及 Table 类。

4.2.2 非数据类型标识符入口

非数据类型标识符包括声明的函数、变量等，通过入口维护。Entry 类指明了一个非数据类型标识符 (变量和函数) 的种类。

派生自 Entry，共有 4 个入口如下：

- VarEntry： 用于普通变量；

```
public class VarEntry extends Entry {
    Type Ty;// 变量类型
    Translate.Access acc;// 为变量分配的存储空间
    boolean isFor;// 标记是否为循环变量

    public VarEntry(Type ty, Translate.Access acc) {
        Ty = ty;
        this.acc = acc;
        this.isFor = false;
    }
    public VarEntry(Type ty, Translate.Access acc, boolean isf) {
        Ty = ty;
        this.acc = acc;
        this.isFor = isf;
    }
}
```

- LoopVarEntry： 用于循环变量，继承自 VarEntry；

```
public class LoopVarEntry extends VarEntry {

    public LoopVarEntry(Type ty, Translate.Access acc) {
        super(ty, acc);
    }
    public LoopVarEntry(Type ty, Translate.Access acc, boolean isf) {
        super(ty, acc, isf);
    }
}
```

- FuncEntry： 用于普通函数；

```
public class FuncEntry extends Entry {
    RECORD paramlist;// 参数表
    Type returnTy;// 返回值类型
    public Translate.Level level;// 函数的层
    public Temp.Label label;// 函数的标记名称

    public FuncEntry(Translate.Level level, Temp.Label label, RECORD p,
Type rt) {
        paramlist = p;
        returnTy = rt;
        this.level = level;
        this.label = label;
    }
}
```

- StdFuncEntry： 用于库函数，继承自 FuncEntry。

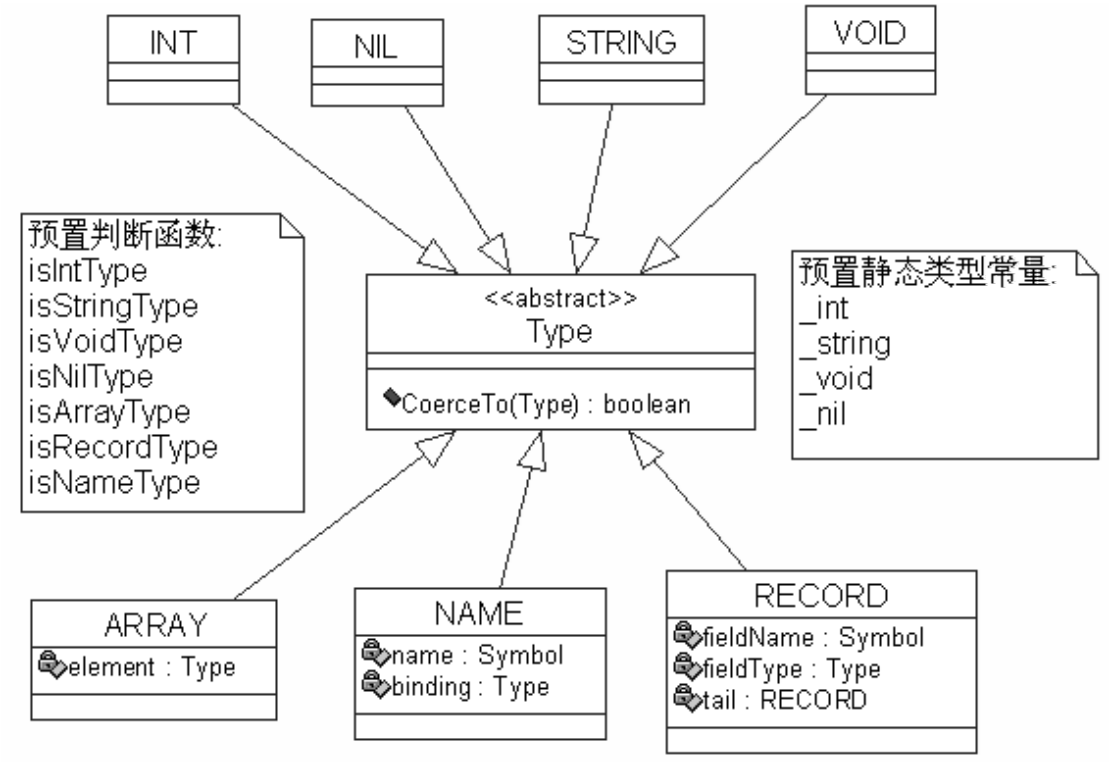
```
public class StdFuncEntry extends FuncEntry {

    public StdFuncEntry(Translate.Level l, Temp.Label lab, RECORD params,
Type rt) {
        super(l, lab, params, rt);
    }
}
```

4.2.3 数据类型标识符维护 – Types 包

Types 包描述数据类型标识符，类名以大写表示，区分 Absyn 中的类型。（Absyn 包中的类型信息是根据程序代码简单翻译的，通过语法检查但没有经过语义检查）。

Types 包类图如下：



Type 抽象类的 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。

NAME 类中就循环定义问题，定义了 isLoop 方法，如下：

```
public class NAME extends Type {
    public Symbol.Symbol name;
    private Type binding;
    public NAME(Symbol.Symbol n) { name = n;}
    public boolean isLoop() {
        Type b = binding;
        boolean any;
        binding = null;
        if (b == null)
            any = true;
        else if (b instanceof NAME)
            any = ((NAME) b).isLoop();
        else any = false;
        binding = b;
        return any;
    }
    public Type actual() {return binding.actual();}
    public boolean coerceTo(Type t) {return this.actual().coerceTo(t);}
    public void bind(Type t) { binding = t;}
}
```

4.3 语义分析一般步骤

语义分析是一个递归过程。

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。

这些主要在 Semant 类中进行的，检查部分与翻译部分在本项目实现中不可割裂。

4.4 语义规则与 SEMANT 类实现

IntExp	
StringExp	
NilExp	
VarExp	
CalcExp	左右均为 int 类型
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
RelExp	左右两边必须全为 int 或 string
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
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
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

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
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)
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)
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)
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
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
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；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
SimpleVar	检查 vEnv，若没找到变量名或类型不是 VarEntry 则报告变量未定义
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
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
NameTy	检查 tEnv,若没有发现类型,则报告未知类型错误
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
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
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
TypeDec	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否有重复的声明(不同的块中可以有重复的声明的)，若有，报告重定义错误(具体实现采用 HashSet 类)
	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
	检测循环定义问题
	如果以上均正确,添加类型到 tEnv 中
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进行：同类型声明类似，先检查声明块中是否有重复的声明；

	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； 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； 接着检查函数返回值，如果没有返回值则设置成 void； 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： 用 BeginScope 进入 vEnv 的一张子表，并转移到新层； 将函数参数存入子表； 翻译函数体，用 ProcEntryExit 给函数体加上函数调用指令； 检查函数体的返回类型是否和声明部分匹配,若否则报错； 用 EndScope 退出子表,转移到当前层。

4.5 总结与思考

语义检查工作和翻译不可分割，在已经很大的代码量基础上又提升了难度，调试部分和 Translate 一并，非常艰难。

5 活动记录

5.1 活动记录目标

语义分析之后，在翻译之前，针对函数调用问题，通过活动记录记录函数调用关系及变量信息。每个函数调用加入一个 Frame 帧结构来保存状态信息，函数静态链接信息有 Level 描述。

5.2 帧结构与活动记录

5.2.1 抽象帧结构 – Frame 类

Frame 包主要包含以下 3 个类：

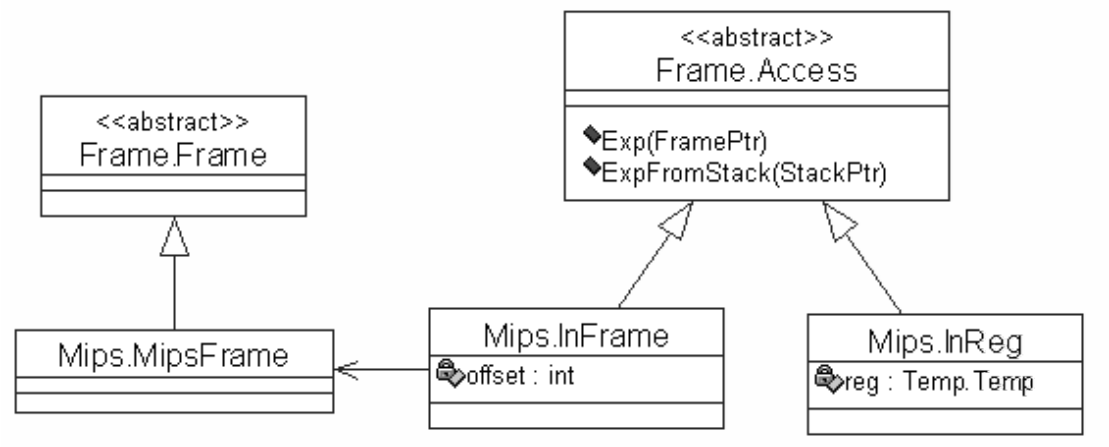
- Frame —— 定义了抽象的程序帧结构，用于存放函数参数，返回地址，寄存器，栈指针，帧指针返回值等重要信息。
- Access —— 用于描述那些存放在帧中或是寄存器中的形式参数和局部变量,也是抽象数据类型。
- AccessList —— 将 Access 串成链表。

5.2.2 用于 MIPS 的帧结构

本项目基于 MIPS，在创建 MIPS 的帧结构时先有：InFrame 类派生自 Access，表示存放在帧中的变量；InReg 类派生自 Access，表示存放在寄存器中的变量。

用于 MIPS 的帧结构 MipsFrame，主要任务为进行初始化寄存器、新建帧、分配 Access 及函数调用处理（procEntryExit1）。

以上类间关系如下图所示：



5.3 静态链接 – LEVEL 类

Level 是用来描述静态连接用的数据结构，它被封装在 Translate.Level 中。

```
    public Level parent;// 上一层level
    Frame.Frame frame;// 层中对应的帧
    public AccessList formals = null;// 参数表
```

Level 类的两个构造函数如下：

```
    public Level(Level parent, Symbol name, BoolList fmls) {
        // 构造函数,带入上一层
        this.parent = parent;
        BoolList bl = new BoolList(true, fmls);
        this.frame = parent.frame.newFrame(new Temp.Label(name), bl);
        for (Frame.AccessList f = frame.formals; f != null; f = f.next)
            this.formals = new AccessList(new Access(this, f.head),
this.formals);
    }

    public Level(Frame.Frame frm) {
        // 新建一个没有上一层的层
        this.frame = frm;
        this.parent = null;
    }
```

5.4 函数调用主要步骤

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：

- 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；
- 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；
- 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：

- 分配帧空间：将$sp 减去帧空间 (如 32byes)；
- 保存 Callee-Saved 寄存器 ($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；

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

第三步，Callee 返回到 Caller 之前：

- 如果函数有返回值,把它放到$v0 中；
- 恢复所有 Callee-Saved 寄存器；
- 将$sp 加上相应的帧空间 (如 32bytes)；
- 跳转回返回地址(存在 $ra 中)。

Caller-Save 和 Callee-Save 是在硬件手册中人为设定的规范，而不是通过硬件实现。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.

MipsFrame 类成员方法包括：

- 新建帧：

```
    public Frame newFrame(Label name, Util.BoolList formals) {
        MipsFrame ret = new MipsFrame();
        ret.name = name; // 传入帧名称
        TempList argReg = argRegs; // 传入参数表
        for (Util.BoolList f = formals; f != null; f = f.tail, argReg =
argReg.tail) {
            // 为每个参数分配存储空间
            Access a = ret.allocLocal(f.head);
            ret.formals = new AccessList(a, ret.formals);
            if (argReg != null) {
                ret.saveArgs.add(new Tree.MOVE(a.exp(new Tree.TEMP(fp)),
new Tree.TEMP(argReg.head)));
                // 产生保存参数的汇编指令,把参数放入 frame 的 Access 中
            }
        }
        return ret;
    }
```

- 分配 Access：

```
    public Access allocLocal(boolean escape) {
        if (escape) {
```

```
            // 若逃逸,则在帧中分配空间
            Access ret = new InFrame(this, allocDown);
            allocDown -= Translate.Library.WORDSIZE;
            // 增加一个机器字的存储空间,注意存储空间向下增长
            return ret;
        } else
            // 否则分配寄存器作为存储空间
            return new InReg();
    }
```

- procEntryExit1 处理：

  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 Tree.Stm procEntryExit1(Tree.Stm body) {
        // 在函数体原指令前加上保存参数的指令
        for (int i = 0; i < saveArgs.size(); ++i)
            body = new Tree.SEQ((Tree.MOVE) saveArgs.get(i), body);
        // 以下为加入保存CalleeSave的指令
        Access fpAcc = allocLocal(true);// 为$fp中的值分配空间
        Access raAcc = allocLocal(true);// 为$ra中的值分配空间
        Access[] calleeAcc = new Access[numOfcalleeSaves];// 为寄存器
$s0~$s7分配空间
        TempList calleeTemp = calleeSaves;// 为寄存器寄存器$t0~$t9分配空间
        for (int i = 0; i < numOfcalleeSaves; ++i, calleeTemp =
calleeTemp.tail) {
            // 将calleeSave寄存器存入帧空间中
            calleeAcc[i] = allocLocal(true);
            body = new Tree.SEQ(new Tree.MOVE(calleeAcc[i].exp(new
Tree.TEMP(fp)), new Tree.TEMP(calleeTemp.head)),
                    body);
        }
        body = new Tree.SEQ(new Tree.MOVE(raAcc.exp(new Tree.TEMP(fp)),
new Tree.TEMP(ra)), body);
        // 在 body 前面加上保存返回地址 $ra 的指令
        body = new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(fp), new
Tree.BINOP(Tree.BINOP.PLUS, new Tree.TEMP(sp),
                new Tree.CONST(-allocDown - Translate.Library.WORDSIZE))),
body);
        // 令$fp=$sp-帧空间+4 bytes
        body = new Tree.SEQ(new Tree.MOVE(fpAcc.expFromStack(new
Tree.TEMP(sp)), new Tree.TEMP(fp)), body);
```

```
        // 在 body 前保存 fp
        calleeTemp = calleeSaves;
        // 在 body 后恢复 callee
        for (int i = 0; i < numOfcalleeSaves; ++i, calleeTemp =
calleeTemp.tail)
            body = new Tree.SEQ(body,
                    new Tree.MOVE(new Tree.TEMP(calleeTemp.head),
calleeAcc[i].exp(new Tree.TEMP(fp))));
        body = new Tree.SEQ(body, new Tree.MOVE(new Tree.TEMP(ra),
raAcc.exp(new Tree.TEMP(fp))));
        // body 后恢复返回地址
        body = new Tree.SEQ(body, new Tree.MOVE(new Tree.TEMP(fp),
fpAcc.expFromStack(new Tree.TEMP(sp))));
        // body 后恢复 fp
        return body;
    }
```

- procEntryExit2 处理：

  函数经 procEntryExit2 处理后保持不变，仅增加一条空指令。

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

- procEntryExit3 处理：

  函数经 procEntryExit3 处理后：分配帧空间:将$sp 减去帧空间 (如 32byes) → 设置函数体标号 → 跳转到返回地址 → 将$sp 加上相应的帧空间 (如 32bytes)。

```
    public InstrList procEntryExit3(InstrList body) {
        body = new InstrList(new OPER("subu $sp, $sp, " + (-allocDown),
new TempList(sp, null), new TempList(sp, null)),
                body);
        body = new InstrList(new OPER(name.toString() + ":", null, null),
body);
        InstrList epilogue = new InstrList(new OPER("jr $ra", null, new
TempList(ra, null)), null);
        epilogue = new InstrList(
                new OPER("addu $sp, $sp, " + (-allocDown), new
TempList(sp, null), new TempList(sp, null)), epilogue);
        body = InstrList.append(body, epilogue);
        return body;
    }
```

5.5 难点与思考

这一章问题很多，Frame 和 Access 在之前理论课程中没有接触过。MipsFrame 实现部分也很复杂，花了很大力气。关键算法在 StepbyStep 中有提供，但所需要声明的成员变量和接口等都没有说明，摸索了很久。

6 中间代码生成

6.1 中间代码生成目标

中间代码生成与语义分析一起，生成了机器无关中间代码，即中间代码树。

6.2 中间代码树维护 – TREE 包

Tree 包定义了中间代码树的节点，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。

中间代码树的节点可以分成 Exp 派生和 Stm 派生两类，直观区别在于 Exp 有返回值而 Stm 没有。

6.3 中间代码翻译 – TRANSLATE 包

6.3.1 表达式翻译过程

在语义分析基础上，生成中间代码树过程分为两步，首先由 Translate 生成树的代理节点（非直接具体节点）Ex，Nx，Cx，然后调用其成员方法来生成具体节点。

代理类继承自 Translate.Exp 抽象类，含三个抽象成员方法，对应三种过程及相应返回值：

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

6.3.2 代理类翻译方法 – Ex, Nx, Cx

三种代理类区别如下：

- Ex 为有返回值的表达式；

```
public class Ex extends Exp {
    Tree.Exp exp;
    public Ex(Tree.Exp e) {
        exp = e;
    }
    Tree.Exp unEx() {
        return exp;
    }// Tree.Exp 本身就是有返回值表达式,无需转换直接返回
```

```
    Tree.Stm unNx() {
        return new Tree.Exp(exp);
    } // Tree.Exp 无返回值表达式
    Tree.Stm unCx(Label t, Label f) {
        return new Tree.CJUMP(Tree.CJUMP.NE, exp, new Tree.CONST(0), t,
f);
    }
}
```

- Nx 为无返回值的表达式；

```
public class Nx extends Exp {
    Tree.Stm stm;
    Nx(Tree.Stm s) {
        stm = s;
    }
    Tree.Exp unEx() {
        // 无返回值表达式不可能被翻译为有返回值表达式,故无操作
        return null;
    }
    Tree.Stm unNx() {
        return stm;
    }// 本身即为无返回值表达式,无需转换
    Tree.Stm unCx(Label t, Label f) {
        // 无法转换
        return null;
    }
}
```

- Cx 表示一个判断条件,有真假两个出口。

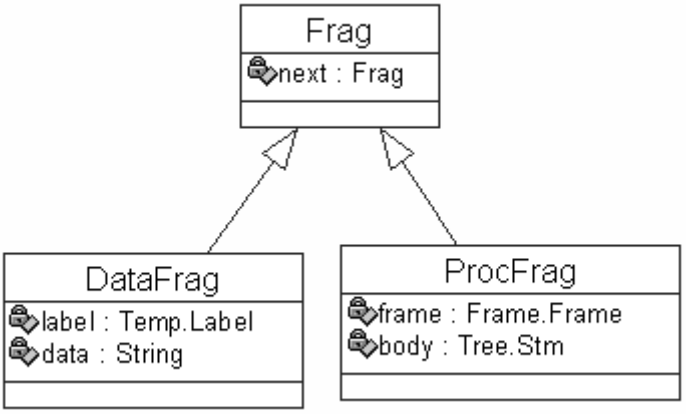
```
public abstract class Cx extends Exp {
    Tree.Exp unEx() {
        Temp r = new Temp();// 返回值
        Label t = new Label();// 真出口
        Label f = new Label();// 假出口
        return new ESEQ(
                new SEQ(new MOVE(new TEMP(r), new CONST(1)),
                        new SEQ(unCx(t, f), // 由子类完成
                                new SEQ(new LABEL(f), new SEQ(new MOVE(new
TEMP(r), new CONST(0)), new LABEL(t))))),
                new TEMP(r));
    }
    abstract Stm unCx(Label t, Label f);
    Stm unNx() {
```

```
        return new Tree.Exp(unEx());
    }
}
```

6.4 段 – FRAG 类

一个 Frag 类可能是一个字符串或一个函数块，因此派生出 DataFrag 和 ProcFrag。DataFrag 中包含标号和字符数据；函数块为 ProcFrag，其中有函数体 (Tree.Stm) 和其 Frame。每个 Frag 都包含一个指向下一个 Frag 的指针，因此可以形成段的链表。

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 AddFrag 和 GetResult 两个接口用于添加段和取回整个段链表。



6.5 各类具体翻译过程

各类的具体翻译过程如下：

情况	结果
整型常数 value	CONST (value)
字符串常数 value	NAME (label)
nil 变量	EX(CONST(0))
变量表达式 translate.Exp	Exp
运算表达式 left oper right	EX(BINOP (binop, left, right))
字符串运算 left oper right	RELCX(OPER, EX (COMP) , EX (0))
其它关系运算 left oper right	RELCX (OPER, LEFT, RIGHT)
赋值运算 lvalue=ex	NX(MOVE (lvalue, ex))
TransCallExp	EX(CALL(NAME(func_name), args))
库函数调用 TransExtCallExp	EX (EXTERNAL_CALL)

stm 的连接 e1, e2	NX(SEQ(NX(E1), NX(E2)))
exp 的连接 e1, e2	EX(ESEQ(NX(E1), EX(E2)))
transRecordExp	EX(ESEQ(SEQ(MOVE(TEMP(addr),alloc),init),TEMP(addr)))
transArrayExp	EX(alloc)
if, while, for	IfExp, WhileExp, ForExp
break	NX (JUMP (LABEL))
简单变量	EX(MEM (BINOP (PLUS, TEMP(FP), CONST(OFFSET))))
下标变量 var[idx]	EX(MEM(BINOP(BINOP.PLUS, EX(VAR),BINOP(MUL,EX(IDX),CONST(WORDSIZE)))
域变量 var[num]	EX(MEM(BINOP(BINOP.PLUS, EX(VAR) , CONST(NUM*WORDSIZE)))
翻译函数体	procEntryExit

Translate 类成员方法实现如下：

- 空操作，整形数，字符型常量，nil 语句：

```
public Exp transNoExp() {
    // 产生一条空语句,实际上和nil语句是同一个效果
    return new Ex(new CONST(0));
}
public Exp transIntExp(int value) {
    // 翻译整形常数
    return new Ex(new CONST(value));
}
public Exp transStringExp(String string) {
    // 翻译字符串常量,产生一个新的数据段
    Temp.Label lab = new Temp.Label();
    addFrag(new Frag.DataFrag(lab, frame.string(lab, string)));
    return new Ex(new NAME(lab));
}
public Exp transNilExp() {
    // 翻译nil语句
    return new Ex(new CONST(0));
}
```

二元运算符，字符串比较，赋值：

```
public Exp transOpExp(int oper, Exp left, Exp right) {
    // 翻译二元运算
    if (oper >= BINOP.PLUS && oper <= BINOP.DIV)
        return new Ex(new BINOP(oper, left.unEx(), right.unEx()));
```

```
        // 加减乘除运算
        return new RelCx(oper, left, right);// 逻辑判断
    }
    public Exp transStringRelExp(Level currentL, int oper, Exp left, Exp
right) {
        // 翻译字符串比较运算,调用标准库函数进行比较,然后进行逻辑判断
        Tree.Exp comp = currentL.frame.externalCall("stringEqual",
                new ExpList(left.unEx(), new ExpList(right.unEx(),
null)));
        return new RelCx(oper, new Ex(comp), new Ex(new CONST(1)));
    }
    public Exp transAssignExp(Exp lvalue, Exp exp) {
        // 翻译赋值表达式,注意赋值表达式无返回值
        return new Nx(new MOVE(lvalue.unEx(), exp.unEx()));
    }
```

函数调用，标准库函数调用：

```
    public Exp transCallExp(Level currentL, Level dest, Temp.Label name,
java.util.ArrayList<Exp> args_value) {
        // 翻译普通函数调用
        ExpList args = null;
        for (int i = args_value.size() - 1; i >= 0; --i) {
            args = new ExpList(((Exp) args_value.get(i)).unEx(), args);
        }
        // 产生实参参数表
        Level l = currentL;
        Tree.Exp currentFP = new TEMP(l.frame.FP());
        while (dest.parent != l) {
            currentFP = l.staticLink().acc.exp(currentFP);
            l = l.parent;
        }
        // 搜索逃逸信息找到静态链所指向的层
        args = new ExpList(currentFP, args);
        // 根据逃逸信息产生逃逸信息,并作为第一个参数即$a0传入函数
        return new Ex(new CALL(new NAME(name), args));
    }
    public Exp transStdCallExp(Level currentL, Temp.Label name,
java.util.ArrayList<Exp> args_value) {
        // 翻译调用标准库函数
        ExpList args = null;
        for (int i = args_value.size() - 1; i >= 0; --i)
            args = new ExpList(((Exp) args_value.get(i)).unEx(), args);
        // 与普通函数调用的区别在于标准库函数不存在函数嵌套定义,即不存在静态链
```

```
        return new Ex(currentL.frame.externalCall(name.toString(),
args));
    }
    public Exp stmcat(Exp e1, Exp e2) {
        // 连接两个表达式,连接后生成无返回值的表达式
        if (e1 == null) {
            if (e2 != null)
                return new Nx(e2.unNx());
            else
                return transNoExp();
        } else if (e2 == null)
            return new Nx(e1.unNx());
        else
            return new Nx(new SEQ(e1.unNx(), e2.unNx()));
    }
```

链接表达式，记录，数组：

```
    public Exp exprcat(Exp e1, Exp e2) {
        // 连接两个表达式,连接后生成有返回值的表达式
        if (e1 == null) {
            return new Ex(e2.unEx());
        } else {
            return new Ex(new ESEQ(e1.unNx(), e2.unEx()));
        }
    }
    public Exp transRecordExp(Level currentL, java.util.ArrayList<Exp>
field) {
        Temp.Temp addr = new Temp.Temp();
        Tree.Exp rec_id = currentL.frame.externalCall("allocRecord",
                new ExpList(new CONST((field.size() == 0 ? 1 :
field.size()) * Library.WORDSIZE), null));
        Stm stm = transNoExp().unNx();
        for (int i = field.size() - 1; i >= 0; --i) {
            Tree.Exp offset = new BINOP(BINOP.PLUS, new TEMP(addr), new
CONST(i * Library.WORDSIZE));
            Tree.Exp value = (field.get(i)).unEx();
            stm = new SEQ(new MOVE(new MEM(offset), value), stm);
            // 为记录中每个域生成 MOVE 指令,将值复制到帧中的相应区域
        }
        // 返回记录的首地址
        return new Ex(new ESEQ(new SEQ(new MOVE(new TEMP(addr), rec_id),
stm), new TEMP(addr)));
    }
```

```
    public Exp transArrayExp(Level currentL, 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
        Tree.Exp alloc = currentL.frame.externalCall("initArray",
                new ExpList(size.unEx(), new ExpList(init.unEx(), null)));
        return new Ex(alloc);
    }
```

If，While，For，Break 语句：

```
    public Exp transIfExp(Exp test, Exp e1, Exp e2) {
        // 将if语句翻译为IR树的节点
        return new IfExp(test, e1, e2);
    }
    public Exp transWhileExp(Exp test, Exp body, Temp.Label out) {
        // 将while语句翻译为IR树的节点,注意只可能是Nx
        return new WhileExp(test, body, out);
    }
    public Exp transForExp(Level currentL, Access var, Exp low, Exp high,
Exp body, Temp.Label out) {
        // 将for语句翻译为IR树的节点,注意只可能是Nx
        return new ForExp(currentL, var, low, high, body, out);
    }
    public Exp transBreakExp(Temp.Label l) {
        // 翻译break语句为IR树的节点,l为loopstack的栈顶标记
        return new Nx(new JUMP(l));
    }
```

简单变量，数组元素，域的成员变量：

```
    public Exp transSimpleVar(Access acc, Level currentL) {
        // 翻译简单变量
        Tree.Exp e = new TEMP(currentL.frame.FP());
        Level l = currentL;
        // 由于可能为外层的变量,故沿着静态链接不断上溯，直到变量的层与当前层相同
        while (l != acc.home) {
            e = l.staticLink().acc.exp(e);
            l = l.parent;
        }
        return new Ex(acc.acc.exp(e));
    }
    public Exp transSubscriptVar(Exp var, Exp index) {
        // 翻译数组元素
        Tree.Exp arr_addr = var.unEx();
        // 数组首地址
        Tree.Exp arr_offset = new BINOP(BINOP.MUL, index.unEx(), new
```

```
CONST(Library.WORDSIZE));
        // 由下标得到偏移量
        return new Ex(new MEM(new BINOP(BINOP.PLUS, arr_addr,
arr_offset)));
        // 产生指令使首地址加上偏移量为数组元素实际地址
    }
    public Exp transFieldVar(Exp var, int fig) {
        // 翻译域的成员变量
        Tree.Exp rec_addr = var.unEx();
        // 首地址
        Tree.Exp rec_offset = new CONST(fig * Library.WORDSIZE);
        // 偏移量,每个成员占一个机器字
        return new Ex(new MEM(new BINOP(BINOP.PLUS, rec_addr,
rec_offset)));
    }
```

6.6 难点与思考

这一部分和语义在实现过程中不可分割，很复杂庞大，难度很高。理论课上没有接触过中间代码树，上手过程相对较慢。

7 规范化

7.1 规范化工作内容

基于已经得到的中间代码树，规范化过程实现:

- 中间代码树树被写成一个没有 SEQ 和 ESEQ 结点的规范树表；
- 根据该表划分基本块，每个基本块中不包含内部跳转和标号；
- 基本块被顺序放置，所有的 CJUMP 都跟有 false 标号。

7.2 实现方法

由于这部分的代码在书所带框架中已经提供。不需知道细节，只需复制 Canon 文件夹，在最后整合时调用以下三个接口即可。

Canon.Canon.linearize (Tree.Stm s)

Canon.BasicBlocks. BasicBlocks(Tree.StmList stms)

Canon.TraceSchedule(BasicBlocks b)

8 指令选择

8.1 指令选择目标

这一部分的目标为：根据中间代码树 IR Tree 的结构，生成 MIPS 汇编指令。但指令中的寄存器均由中间变量表示，具体寄存器将在下一章分配。

8.2 常用 MIPS 汇编指令

常用 MIPS 汇编指令如下：

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

8.3 MIPS 中的寄存器

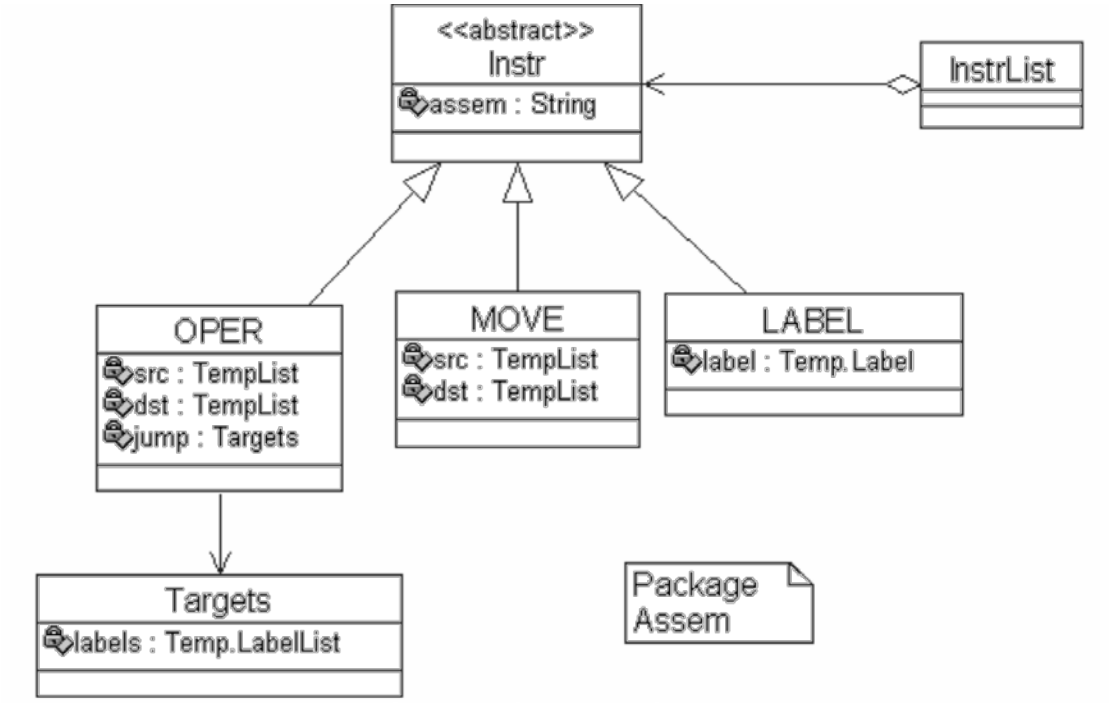
MIPS 中共有 32 号寄存器，如下：

寄存器	名称	用途
$0	$zero	常量 0(constant value 0)

$1	$at	保留给汇编器(Reserved for assembler)
$2-$3	$v0-$v1	函数调用返回值(values for results and expression evaluation)
$4-$7	$a0-$a3	函数调用参数(arguments)
$8-$15	$t0-$t7	暂时的(或随便用的)
$16-$23	$s0-$s7	保存的(或如果用，需要 SAVE/RESTORE 的)(saved)
$24-$25	$t8-$t9	暂时的(或随便用的)
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	返回地址(return address)

8.4 无寄存器分配指令封装 – ASSEM.INSTR 类

Assem 包类图如下：



Instr 类表示汇编语言指令，与特定机器无关，有派生类 OPER，LABEL 和 MOVE。

- OPER 类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，用于确定使用哪个标号符号。
- 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。

8.5 汇编代码生成 – CODEGEN 类

代码生成在 Mips.Codegen 中完成，输入中间代码树节点，将之转为汇编指令，需要实现 Tree 包中各类树的翻译。共需实现以 Tree.MOVE，Tree.JUMP，Tree.LABEL，Tree.CJUMP，Tree.EXP，Tree.MEM，Tree.CONST，Tree.BINOP，Tree.TEMP，Tree.NAME，Tree.CALL 为根的树的翻译。

其中 MOVE 指令含有以下 6 种情况：

- 将常数存入寄存器；
- 将源寄存器的值存入目标寄存器；
- 将源寄存器的值存入内存中，内存首地址在左子树，偏移量在右子树；
- 将源寄存器的值存入内存中，内存首地址在右子树，偏移量在左子树；
- 将寄存器的值存入内存中，内存地址是常数；
- 将寄存器的值存入内存中，内存地址在源寄存器中。

MOVE 指令代码生成对应的 6 种处理方式见下：

```
    private void munchStm(Tree.MOVE m) {
        // 翻译move语句
        if (m.dst instanceof Tree.TEMP) {
            if (m.src instanceof Tree.CONST) {
            // 将常数存入寄存器
                emit(new Assem.OPER("li `d0, " + ((Tree.CONST)
m.src).value, L(((Tree.TEMP) m.dst).temp, null), null));
            } else {
            // 将源寄存器的值存入目标寄存器
                Temp.Temp t1 = munchExp(m.src);
                emit(new Assem.OPER("move `d0, `s0", L(((Tree.TEMP)
m.dst).temp, null), L(t1, null)));
            }
            return;
        }
```

```
        if (m.dst instanceof Tree.MEM) {
            Tree.MEM mem = (Tree.MEM) m.dst;
            if (mem.exp instanceof Tree.BINOP) {
                Tree.BINOP mexp = (Tree.BINOP) mem.exp;
                if (mexp.binop == Tree.BINOP.PLUS && mexp.right instanceof
Tree.CONST) {
            // 将源寄存器的值存入内存中,内存首地址在左子树,偏移量在右子树
                    Temp.Temp t1 = munchExp(m.src);
                    Temp.Temp t2 = munchExp(mexp.left);
                    emit(new Assem.OPER("sw `s0, " + ((Tree.CONST)
mexp.right).value + "(`s1)", null,
                            L(t1, L(t2, null))));
                    return;
                }
                if (mexp.binop == Tree.BINOP.PLUS && mexp.left instanceof
Tree.CONST) {
            // 将源寄存器的值存入内存中,内存首地址在右子树,偏移量在左子树
                    Temp.Temp t1 = munchExp(m.src);
                    Temp.Temp t2 = munchExp(mexp.right);
                    emit(new Assem.OPER("sw `s0, " + ((Tree.CONST)
mexp.left).value + "(`s1)", null,
                            L(t1, L(t2, null))));
                    return;
                }
            }
            if (mem.exp instanceof Tree.CONST) {
            // 将寄存器的值存入内存中,内存地址是常数
                Temp.Temp t1 = munchExp(m.src);
                emit(new Assem.OPER("sw `s0, " + ((Tree.CONST)
mem.exp).value, null, L(t1, null)));
                return;
            }
            // 将寄存器的值存入内存中，内存地址在源寄存器s1中
            Temp.Temp t1 = munchExp(m.src);
            Temp.Temp t2 = munchExp(mem.exp);
            emit(new Assem.OPER("sw `s0, (`s1)", null, L(t1, L(t2,
null))));
        }
    }
```

MEM 指令含 4 种情况：

- 将内存中的值存入目标寄存器，内存地址是常数；
- 内存地址由内存首地址和偏移量(常数)生成，常数在右子树；

- 内存地址由内存首地址和偏移量(常数)生成，常数在左子树；
- 内存地址直接由寄存器给出,无偏移量。

MEM 指令代码生成对应的 4 种处理方式见下：

```
    private Temp.Temp munchExp(Tree.MEM m) {
        // 翻译读取内存的指令
        Temp.Temp ret = new Temp.Temp();
        if (m.exp instanceof Tree.CONST) {
            // 将内存中的值存入目标寄存器,内存地址是常数
            emit(new Assem.OPER("lw `d0, " + ((Tree.CONST) m.exp).value,
L(ret, null), null));
            return ret;
        }
        if (m.exp instanceof Tree.BINOP && ((Tree.BINOP) m.exp).binop ==
Tree.BINOP.PLUS) {
            if (((Tree.BINOP) m.exp).right instanceof Tree.CONST) {
                // 内存地址由内存首地址和偏移量(常数),常数在右子树
                Temp.Temp t1 = munchExp(((Tree.BINOP) m.exp).left);
                emit(new Assem.OPER("lw `d0, " + ((Tree.CONST)
((Tree.BINOP) m.exp).right).value + "(`s0)",
                        L(ret, null), L(t1, null)));
                return ret;
            }
            if (((Tree.BINOP) m.exp).left instanceof Tree.CONST) {
                // 内存地址由内存首地址和偏移量(常数),常数在左子树
                Temp.Temp t1 = munchExp(((Tree.BINOP) m.exp).right);
                emit(new Assem.OPER("lw `d0, " + ((Tree.CONST)
((Tree.BINOP) m.exp).left).value + "(`s0)", L(ret, null),
                        L(t1, null)));
                return ret;
            }
        }
        // 内存地址直接由寄存器给出,无偏移量
        Temp.Temp t1 = munchExp(m.exp);
        emit(new Assem.OPER("lw `d0, (`s0)", L(ret, null), L(t1, null)));
        return ret;
    }
```

8.6 MIPS 汇编语言语法

- 注释： 以 # 号开始
- 指令： 见本章第一节

- 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

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

9 寄存器分配

9.1 寄存器分配目标

寄存器分配顾名思义，其大致步骤为：根据汇编指令生成流图 → 活性分析 → 生成干扰图 → 根据干扰图分配寄存器（着色）。

同样以之前的例子，分配寄存器后生成的代码如下：

```
.globl main
.text
main:
subu $sp, $sp, 56
L1:
sw $fp, 36($sp)
add $t3, $sp, 52
move $fp, $t3
sw $ra, -20($fp)
sw $s0, -52($fp)
sw $s1, -48($fp)
sw $s2, -44($fp)
sw $s3, -40($fp)
sw $s4, -36($fp)
sw $s5, -32($fp)
sw $s6, -28($fp)
sw $s7, -24($fp)
sw $a0, 0($fp)
li $s4, 1
sw $s4, -8($fp)
li $s4, 2
sw $s4, -12($fp)
lw $s4, -8($fp)
lw $t3, -12($fp)
add $s4, $s4, $t3
lw $s4, -24($fp)
move $s7, $s4
lw $s4, -28($fp)
move $s6, $s4
lw $s4, -32($fp)
move $s5, $s4
lw $s4, -36($fp)
move $s4, $s4
lw $t3, -40($fp)
move $s3, $t3
lw $t3, -44($fp)
move $s2, $t3
lw $t3, -48($fp)
move $s1, $t3
lw $t3, -52($fp)
move $s0, $t3
lw $t3, -20($fp)
move $ra, $t3
lw $t3, 36($sp)
move $fp, $t3
j L0
L0:
addu $sp, $sp, 56
jr $ra
```

9.2 抽象图结构 – GRAPH 类

原书框架代码中提供了 Graph 包，其中封装了一个抽象的图结构，支持通常图结构中结点和边的大多数操作，采用结点的邻接表表示。同时支持有向图。

9.3 流图 – FLOWGRAPH 类

活性分析基于程序流图。FlowGraph 包中封装了 FlowGraph 抽象类，其基础上，FlowGraph.AssembleFlowGroup 封装了一个面向汇编指令的流图结构，在这个类中每个结点代表一个汇编指令，边为可能的控制转移。

该类可通过 Instr 构造，构造函数实现如下：

```
    public AssemFlowGraph(InstrList instrs) {
        Dictionary<Label, Node> labels = new Hashtable<Label, Node>();
```

```
        // 把所有指令加为流图节点
        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);
        }
        // 加边
        for (NodeList node = nodes(); node != null; node = node.tail) {
            Targets next = instr(node.head).jumps();
            if (next == null) {
                if (node.tail != null)
                    addEdge(node.head, node.tail.head);
            } else
                for (LabelList l = next.labels; l != null; l = l.tail)
                    addEdge(node.head, (Node) labels.get(l.head));
        }
    }
```

9.4 活性分析 – LIVENESS 类

9.4.1 初始化

对于流图中的每个结点的结点信息，由 NodeInfo 类维护如下：

```
public class NodeInfo {
    public Set<Temp> in = new HashSet<Temp>(); // 来指令前的活性变量
    public Set<Temp> out = new HashSet<Temp>(); // 出指令后的活性变量
    public Set<Temp> use = new HashSet<Temp>(); // 某指令使用的变量
    public Set<Temp> def = new HashSet<Temp>(); // 某指令定义的变量
    NodeInfo(TempList u, TempList d) {
        for (TempList t = u; t != null; t = t.tail)   use.add(t.head);
        for (TempList t = d; t != null; t = t.tail)   def.add(t.head);
    }
    public NodeInfo(Node n) {
        for (TempList t = ((FlowGraph) n.mygraph).use(n); t != null; t =
t.tail) use.add(t.head);
        for (TempList t = ((FlowGraph) n.mygraph).def(n); t != null; t =
t.tail) def.add(t.head);
    }
}
```

活性分析，就是求出每个结点的 out 变量。首先初始化部分如下：

```
    private void initNodeInfo() {
        // 初始化
        for (NodeList nodes = flowGraph.nodes(); nodes != null; nodes =
nodes.tail) {
            // 将每一个点的def与use输入到哈希表中
            NodeInfo ni = new NodeInfo(nodes.head);
            node2nodeInfo.put(nodes.head, ni);
        }
    }
```

9.4.2 计算活性

初始化后，活跃变量迭代计算由下述转移方程实现，直到 in 和 out 不变为止。

$$\text{in}[n] = use[n] \cup (out[n] - \text{def}[n])$$

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

RegAlloc.Liveness. calculateLiveness() 中实现如下：

```
void calculateLiveness() {
        boolean done = false;
        do {
            done = true;
            for (NodeList node = flowGraph.nodes(); node != null; node =
node.tail) {
                NodeInfo inf = node2nodeInfo.get(node.head);
                Set<Temp> in1 = new HashSet<Temp>(inf.out);
                in1.removeAll(inf.def);
                in1.addAll(inf.use);
                if (!in1.equals(inf.in))done = false;
                inf.in = in1;
                Set<Temp> out1 = new HashSet<Temp>();
                for (NodeList succ = node.head.succ(); succ != null; succ
= succ.tail) {
                    NodeInfo i = node2nodeInfo.get(succ.head);
                    out1.addAll(i.in);
                }
                if (!out1.equals(inf.out)) done = false;
                inf.out = out1;
            }
        } while (!done);
        for (NodeList node = flowGraph.nodes(); node != null; node =
node.tail) {
```

```
            TempList list = null;
            Iterator<Temp> i = ((NodeInfo)
node2nodeInfo.get(node.head)).out.iterator();
            while (i.hasNext())
                list = new TempList((Temp) i.next(), list);
            if (list != null)
                liveMap.put(node.head, list);
        }
    }
```

9.4.3 生成干扰图

考察控制流图和数据流图，可以画出干扰图。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，因之后寄存器分配算法，每条无向边维护成一对有向边。其抽象接口在 RegAlloc.InterferenceGraph 中实现。生成干扰图的算法如下:

- 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)。
- 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.buildGrap() 中实现，如下：

```
    private void buildGraph() {
        Set temps = new HashSet();
        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);
        }
        Iterator i = temps.iterator();
        while (i.hasNext())  add(newNode(), (Temp) i.next());
        for (NodeList node = flowGraph.nodes(); node != null; node = node.tail)
            for (TempList t = flowGraph.def(node.head); t != null; t = t.tail)
                for (TempList t1 = (TempList) liveMap.get(node.head); t1 !=
null; t1 = t1.tail)
                    if (t.head != t1.head
                            && !(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));
                    }
    }
```

9.5 着色法寄存器分配 – COLOR 类

根据干扰图分配寄存器，实质是根据干扰图进行的着色过程，即使干扰图中相邻的两个结点着不同的颜色。

这是一个 NP-Hard 问题。本设计中采用了指导文件中的贪心算法，具体如下：

- 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。
- 扫描剩下结点（设为 n 个），它们还没有被分配寄存器，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。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。寄存器溢错误不进行额外处理了，报错。
- 为栈顶 n 个还没有分配寄存器的结点分配寄存器。弹出堆栈的结点 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。

以上步骤在 RegAlloc.Color.color() 方法中实现，如下：

```
public Color(InterferenceGraph interGraph, TempMap init, HashSet
registers) {
        // 用着色法分配寄存器,最终结果放入tempmap中
        HashSet regs = new HashSet(registers);
        this.init = init;
        int number = 0;
        // 预处理,处理已被分配的寄存器的变量
        for (NodeList nodes = interGraph.nodes(); nodes != null; nodes =
nodes.tail) {
            ++number;
            Temp temp = interGraph.gtemp(nodes.head);
            if (init.tempMap(temp) != null) {
                --number;
                selStack.add(nodes.head);
                map.put(temp, temp);
                for (NodeList adj = nodes.head.succ(); adj != null; adj =
adj.tail)
                    interGraph.rmEdge(nodes.head, adj.head);
            }
        }
        // 处理剩下的number个还未分配寄存器的变量
        for (int i = 0; i < number; ++i) {
            Node node = null;
```

```
            int max = -1;
            for (NodeList n = interGraph.nodes(); n != null; n = n.tail)
                if (init.tempMap(interGraph.gtemp(n.head)) == null
&& !selStack.contains(n.head)) {
                    int num = n.head.outDegree();
                    if (max < num && num < regs.size()) {
                        max = num;
                        node = n.head;
                    }
                }
            if (node == null) {
                System.err.println("Color.color() : 溢出");
                break;
            }
            selStack.add(node);
            for (NodeList adj = node.pred(); adj != null; adj = adj.tail)
                if (!selStack.contains(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
        for (int i = 0; i < number; ++i) {
            Node node = (Node) selStack.pop();
            Set available = new HashSet(regs);
            for (NodeList adj = node.succ(); adj != null; adj = adj.tail){
                available.remove(map.get(interGraph.gtemp(adj.head)));
            }
            Temp reg = (Temp) available.iterator().next();
            map.put(interGraph.gtemp(node), reg);
        }
    }
```

9.6 难点与思考

这一部分的组装在 RegAlloc 构造函数中实现，其在对象构造时完成寄存器分配算法，这时候 Color 类的成员对象将记录临时寄存器的映射情况，同时由于 Color 实现了 Temp.TempMap 接口，因此可以直接用于最终的目标代码生成。

这一部分着色算法的编写相对难度较大，本项目实现过程中使用了 java.util 中的标准栈类。Debug 部分花了不少时间。

10 模块整合

10.1 模块整合工作内容

以上部分基本完成了 Tiger 编译器的全部核心功能，最后代码整合工作包括封装的错误报告类及编译器入口类方法。

10.2 错误报告 – ErrorMsg 类

ErrorMsg.ErrorMsg 封装了编译器的错误报告机制，调用 error (position, message) 能输出出错位置和对应信息。

这不包括编译器代码本身的错误，这类错误可能直接抛出异常，程序终止。

10.3 编译器入口 – Main.main()

Main 类作为入口类，链接了编译器的各个部分，main() 方法如下：

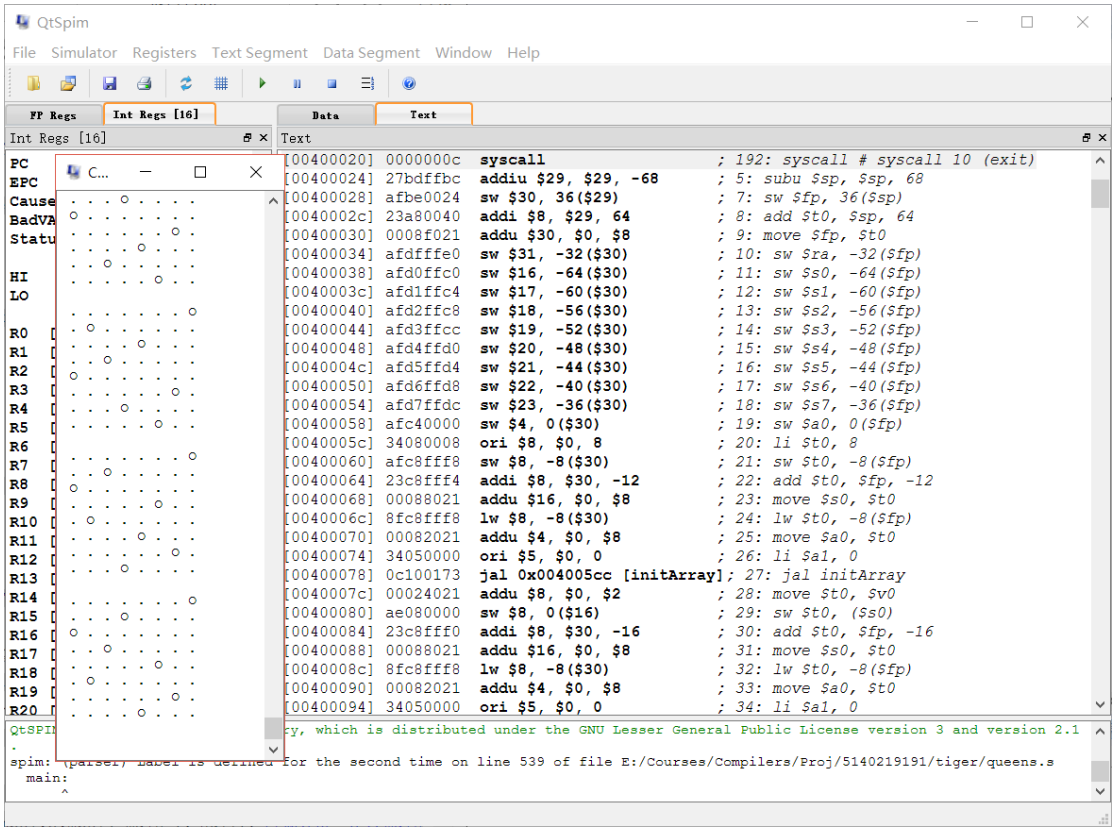
```
    public static void main(String[] argv) throws java.io.IOException {
        String filename = "Testcases/Official/Good/test6.tig";
        ErrorMsg errorMsg = new ErrorMsg(filename);
        InputStream inp = new FileInputStream(filename);
        InputStream inp2 = new FileInputStream(filename);
        // 定义输入流
        PrintStream out = new PrintStream(new
FileOutputStream(filename.substring(0, filename.length() - 4) + ".s"));
        // 定义输出汇编文件
        Yylex lexer = new Yylex(inp, errorMsg);
        java_cup.runtime.Symbol tok;
        Yylex lexer2 = new Yylex(inp2, errorMsg);
        System.out.println("# 词法分析：");
        try {
            do {
                tok = lexer2.next_token();
                System.out.println(symnames[tok.sym] + " " + tok.left);
            } while (tok.sym != sym.EOF);
            inp2.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        // 以上为词法分析
```

```
        System.out.println("\n# 语法分析：");
        Exp result = null;
        parser p = new parser(lexer);
        // 语法分析
        // 打印抽象语法树
        try {
            result = (Exp) p.parse().value;
        } catch (Exception e) {
            e.printStackTrace();
        }
        Print pr = new Print(System.out);
        pr.prExp((Exp) result, 0);
        System.out.println("\n");
        Frame frame = new MipsFrame();
        Translate translator = new Translate(frame);
        Semant smt = new Semant(translator, errorMsg);
        // 语义分析
        Frag.Frag frags = smt.transProg((Exp) result);
        if (ErrorMsg.anyErrors == false)
            System.out.println("无语义语法错误");
        else return;
        // 返回语法分析得到的段,分为数据段和程序段
        irOut = new PrintStream(new
FileOutputStream(filename.substring(0, filename.length() - 4) + ".ir"));
        // 定义中间表示树的输出文件
        out.println(".globl main");
        // 定义main为全局标记
        for (Frag.Frag f = frags; f != null; f = f.next)// 遍历所有的段
            if (f instanceof Frag.ProcFrag)
                emitProc(out, (Frag.ProcFrag) f);// 若为程序段则翻译为
以.text为开头的汇编代码
            else if (f instanceof Frag.DataFrag)
                out.print("\n.data\n" + ((Frag.DataFrag) f).data);// 若为
数据段则翻译为以.data开头的数据或字符串
        BufferedReader runtime = new BufferedReader(new
FileReader("runtime.s"));
        while (runtime.ready())
            out.println(runtime.readLine());
        // 将runtime.s中的库函数汇编代码接到文件末尾
        out.close();
        System.out.println("汇编代码已生成");
        System.exit(0);
    }
```

10.4 运行汇编程序

运行 QtSpim，重置加载汇编文件，单机运行按钮运行。默认位置再文件头，另可指定.globl main 所对应的位置为程序开始位置。

成功编译运行的八皇后问题程序结果见下图：



11体悟收获

完成这一课程设计，可以说心疲力竭，但回顾整个过程，这之中收获了许多。

首次深入接触 Java 语言以及 Eclipse IDE，也是第一次做这样大规模的项目，感觉自己的编程能力有了大幅度的提升；

加深了对编译原理的认识，在搭建 Tiger 编译器的过程中学到了许多理论课未涉及、未深入部分的知识，对于编译器有了更整体性的认识。

感谢沈耀老师及助教在该课程项目中的投入与付出，这一定是我在大学阶段难忘的经历。

12参考资料

[1] Feng Qian. *Compiler Step By Step*

[2] Andrew Appel. *Modern Compiler Implementation in Java* (2nd Edition)

[3] Stephen Edwards. Tiger Language Reference Manual

[4] JFlex User Manual

http://www.jflex.de/manual.html

[5] JAVACUP User Manual

http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html