编译原理project4

13331125 李天培

语言

```
program    → block
block      → {decls stmts}
decls      → decls decl | ∈
decl       → type id ;
type       → type [ num ] | basic
stmts      → stmts stmt | ∈
stmt       → loc = bool;
           | if ( bool ) stmt
           | if ( bool ) stmt else stmt
           | while ( bool ) stmt
           | do stmt while ( bool );
           | break;
           | block
loc        → loc [ bool ] | id
bool       → bool || join | join
join       → join && equality | equality
equality   → equality == rel | equality != rel | rel
rel        → expr < expr | expr <= expr
           | expr >= expr | expr > expr | expr
expr       → expr + term | expr – term | term
term       → term * unary | term / unary | unary
unary      → !unary | –unary | factor
factor     → (bool) | loc | num | real | true | false
```

对于语言中的所有非终结符，在parser中都是使用一个函数作为代表。语言本身并不带有二义性，同时通过表达式的产生式处理了运算符的结合性和优先级

lexer

词法分析部分主要包括词法单元的定义，词法分析器，保留字，标识符等

1. **Token**：作为编译过程中的最小操作单元，所有的操作符、变量、保留字，都被转化为一个Token，但
   I. Token类本身并没有值（value）。
   II. 只包含一个tag，作为token的说明。
   III. 同时定义每一个token需要一个toString方法，作为中间代码（本程序的最终结果）。
2. **Tag**：在这个类中，定义了大量的保留字的tag值。范围[256, 275]，小于256的为各单字符操作符的ASCII码范围，利用ASCII码直接作为tag值。
3. **继承Token**：包含三个类，Num（整数），Real（浮点数），Word（保留字和操作符，如&&为操作符，true、t（临时变量）为保留字）。
4. **Lexer**：
   I. 词法分析器保存了代码行数信息，保留字存储在words哈希表中，用来在遇到保留字的时候方便查询。
   II. scan函数是Lexer中的主要函数：
      i. 处理多余信息，包括空格，换行（行数会增加），缩进。
      ii. 处理运算符：主要处理的是由两个字符组成的运算符（对于只生成中间代码而言，对每一个运算符而言，只是一个字符，如+在这个程序的结果中只是打印一个'+'。所以此类运

算符都归结到scan最后一部分，但对于“<=”，则需要保存两个以上的字符，所以是新建一个Word，但对于“<”，则只是用一个Token，tag则是“<”的ASCII码值）。

iii. 处理数字：主要判断是否为浮点数或整数，并分别返回不同的Token子类。

iv. 处理字符串：对于字符串，有可能是保留字（if、int、true），也有可能是自定义变量。

v. 什么都是不则直接返回一个新的token。

5. 词法分析器的主要作用就是将原来由字符代码转换成token组成的代码，方便生成中间代码。同时提供token在生成中间代码时候，应该以返回（return）的形式。

symbols

1. symbols包中主要是定义了Env，Env用来保存token。在语法分析器中，每一个block都有一个对应的Env，当查找某一个token时，先查找最内层block，再依次向外查找。
2. Array和Type类定义了数组和类型，分别继承自Type和Word。对于Type，int等可以作为一个Word处理，对于Array，基本为一个Type。在此处的Type和Array主要应用于定义时。

inter

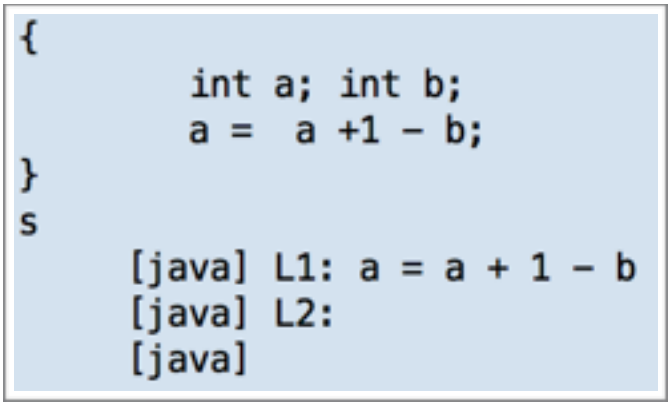
中间代码部分表示为打印中间代码所需要的所有代码。包括赋值语句、条件判断、循环等。

Node：对于中间代码而言，程序会生成一棵语法树。其中所有的结点对应的就是相应的一个子类。Node类包含了最基本的将代码转换成中间代码的函数，包括行号，生成语句块（emitlabel），生成中间代码（emit）

1. **Stmt**：语句结点。类中包含了gen函数，表示语句的范围，b为起始控制流编号，a为语句结束后的控制流编号。该函数是作为生成中间代码时使用，但Stmt中的gen并无实际作用。after参数是为break准备的，after可以视作循环语句结束后的第一条语句，after的目的是当循环中有break时，其跳转到的目标语句。
    1. 循环（While，Do）：循环控制中有两大部分，条件判断（表达式形式Expr）和循环代码（语句形式Stmt）。都设置了after作为break的跳出，并且设置了条件判断的跳转代码（jumping）。
    2. 赋值（Set，SetElem）：赋值类中主要是判断类型是否符合，同时gen函数只是打印相应的赋值语句，对数组的赋值进行规约。
    3. 控制流语句（If，Else）：生成控制流序号，并生成对应的跳转语句。之后生成相应的内部代码。在else中，if的非出口为else入口，所以跳转的目的标号不是a。同时在执行完if内容后跳过else内容。
    4. Seq：简单的将多个语句连接起来，各个语句的中间代码由各语句生成。gen中只是调用语句的gen。
    5. Break：根据语句的after进行跳转。
2. **Expr**：表达式节点。和Stmt最明显的区别有两个（1）表达式中不会有嵌套，而stmt中会嵌套表达式（2）Expr可以规约为一个Token。对于一个Expr，包含一个Token和Type，作为该表达式最基本的信息。跳转函数包含两个值t和f，表示表达式的真假出口，用来将跳转代码翻译中间代码形式。gen返回三地址码形式，reduce返回单地址码形式。

```
public void jumping(int t, int f) {
    int label = t != 0 ? t : newlabel();
    expr1.jumping(label, 0);
    expr2.jumping(t, f);
    if (t == 0) emitlabel(label);
}
```

1. Logical：布尔判断的父类，其中会检验类型是否正确。
    1. Rel、Or、Not、And：分别是大小判断、或、非、且。其中只是实现表达式与运算符的关系所产生的不同的跳转形式，如：Or中，expr1真出口为原真出口，假出口为继续执行，expr2的真假出口皆为原真假出口，和或运算的行为一致，当第一个表达式为真时，直接执行代码而不需要判断后面的表达式，否则判断后面表达式，但是为了跳过判断后面的表达式，当t为0时（if中t即为0），需要newlabel提供跳转到内部代码的控制流编号。
2. Op：包含一个reduce函数，为的是产生一个临时变量，将较长的表达转换成多个三地址码形式的表达式。在子类中会调reduce函数，当expr为Op或其子类，则会产生一个临时变量，或为Id、Constant，直接产生对应的变量或值。这样就完成了表达式的拆分。
    1. Unary、Arith、Access：每一个的gen函数都会返回一个新的类，为的是子表达式expr规约成一个单地址码，如果只是返回this，则不能返回三地址码形式。



3. Temp：每一个临时变量都有自己的编号，同时Temp管理编号顺序。
4. Id：offset为在Id管理中的偏移量。
5. Constant：将数值转换为token，并且将true和false转换为对应的跳转。

parser

根据程序顺序，先调用block()完成语法树的构建，再调用gen，逐步生成中间代码。

1. block()：返回一个Stmt，提供相当于语法数的根结点。根据语言，在parser中定义了不同的函数，如bool()，join()。程序会先到达最底层，即factor()，返回一个特定的Node，不断向上返回，并判断是否符合该函数，每个函数都会返回相应的新的Node，这样形成了一个自底向上的翻译树。
2. s.gen()：根据语法树进行打印中间代码，s作为Stmt，不断地调用其子结点的gen函数，生成相应的中间代码。