

A Complete Front End 分析与注解

编译原理 Project4

目录

构建类图.....	2
主程序 MAIN 包.....	2
词法分析器 LEXER 包.....	2
符号表和类型 SYMBOLS 包.....	3
中间代码生成 INTER 包.....	3
语法生成器 PARSER 包.....	4
加入注释.....	5
运行结果.....	6
MAKEFILE 文件.....	6
ECLIPSE 运行.....	7
PARSER 详解.....	7
编程细节.....	8
TOSTRING 函数.....	8
PARSER 中的 MATCH 函数.....	8
文法没有提取左公因子.....	8
ENV 作用域.....	9
一些疑惑.....	9
ID 中的 OFFSET 属性.....	9
生成中间代码时跟踪行号.....	9
PARSER 中的文法与原来文法有差异.....	10
参考文档.....	10

本实验报告是本人阅读《编译原理》龙书附录 A——一个完整的编译器前端的记录。该编译器主要实现将一门简单的编程语言转换为三地址码表示。包括词法分析器，语法分析器以及中间代码生成。其中，词法分析器将所读取到的字符串文本转换为 token 流，输入到语法分析器，语法分析器使用递归下降算法，根据预先定义好的文法，利用 token，自顶向下构造出一棵抽象语法树，最后，遍历生成的语法树，生成并打印出三地址码。整个编译器的代码主要分为 lexer，parser，inter，symbols 以及 main 五个类，结构合理，代码条理清晰，思路简明易懂，具有较高的参考价值。以下，将介绍本人阅读此编译器源码的具体过程。

构建类图

由于此编译器的类相对较多，所以，做出类图对于分析类之间的相互关系，了解类函数的功能等，有着很大的帮助。以下的类图依据附录源码，使用 UMLet 做出。所有的类图都放在文件夹/uml 下，包含 jpg 文件以及 uxf 文件。

主程序 main 包

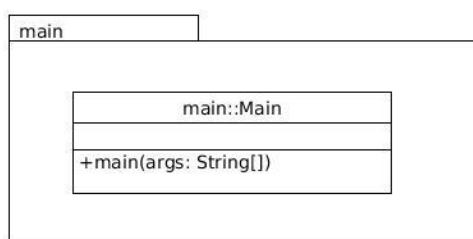


图 1 main package

main 包，仅包含一个类 Main，是程序运行的入口，负责调用 lexer 和 parser。

词法分析器 lexer 包

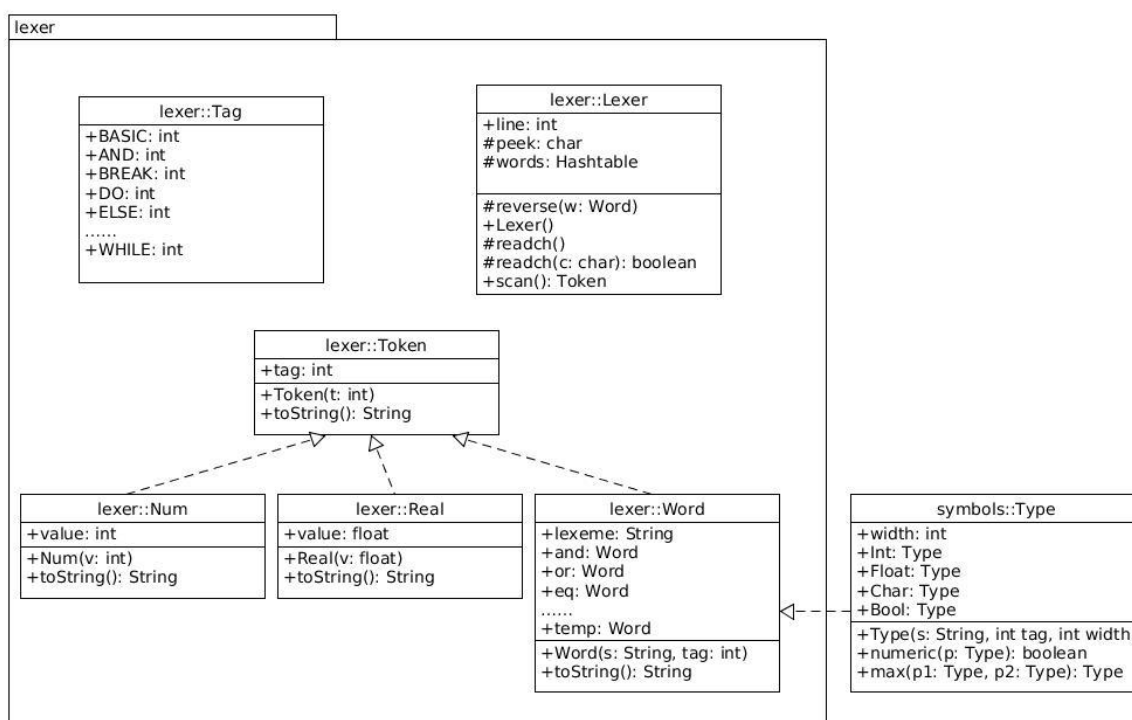


图 2 词法生成器 lexer 包

词法生成器包，包含词法单元以及词法生成器。其中，

- Token 为所有词法单元的父类，其下有三个子类，分别是整数词法单元 Num，浮点数词法单元 Real 以及保留字词法单元 Word。
- Tag 类定义了终结符对应的常量，用于设置 Token 的 tag 属性，在语法生成器中可以根据 tag 来

识别 Token

- Lexer 类是整个 lexer 包的核心，它的功能是读取字符串流，并返回识别到的词法单元。
- Type 类为类型，属于 symbols 包中，但基本类型，如 int, float 等属于保留字，所以继承了保留字 Word 类，故此处一并列出。

符号表和类型 symbols 包

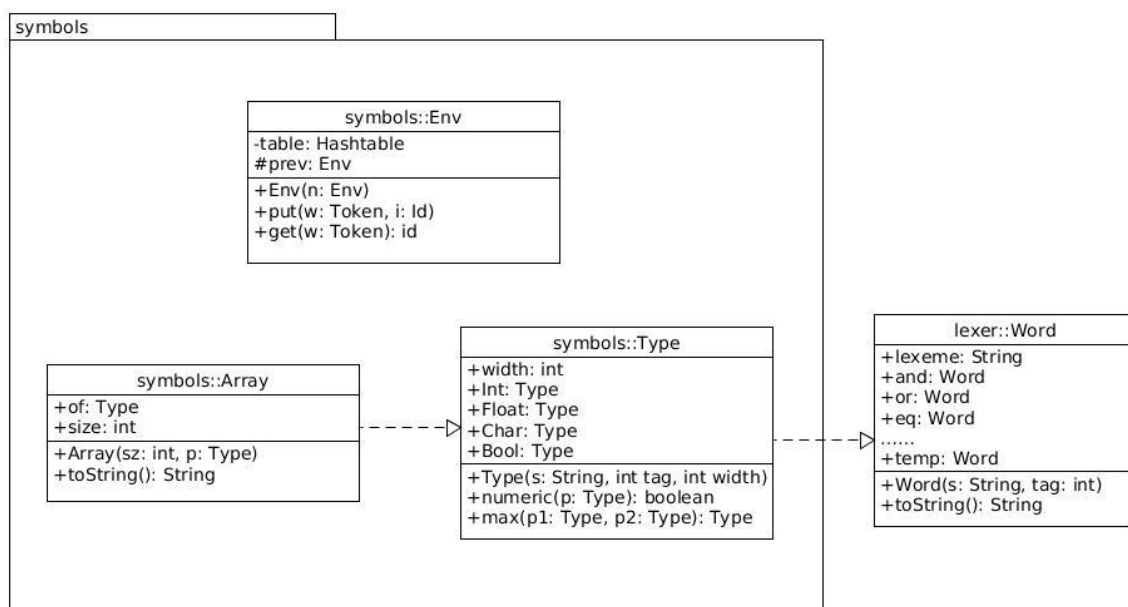


图 3 符号表和类型 symbols 包

符号表和类型包，包含符号表，基本类型以及构造类型。其中，

- Env 类为作用域符号表，它将词法单元映射为标识符条目。在语法分析过程中，实际上会连带使用它来做语义分析，即判断变量在当前作用域是否可见。
- Type 类为基本类型，继承了 Word 类。
- Array 为数组类，它是本语言中唯一的构造类型。其中的 size 属性记录了数组的大小，在生成中间代码时，可用于计算占用的内存空间。

中间代码生成 inter 包

由于 inter 包相对较大，故分三部分来讲。

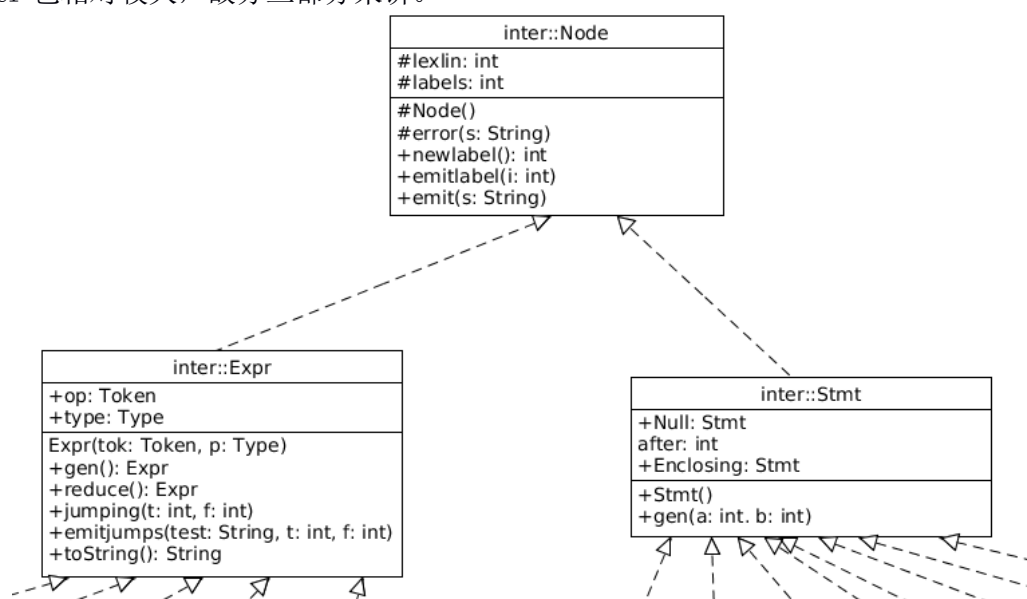


图 4 中间代码生成 inter 包 (1)

- Node 类为中间代码的父类，即抽象语法的节点；
- Expr 类为表达式的父类，表达式包括逻辑运算，算术运算等等；
- Stmt 类为语句的父类，语句包括 if 语句，while 语句等。

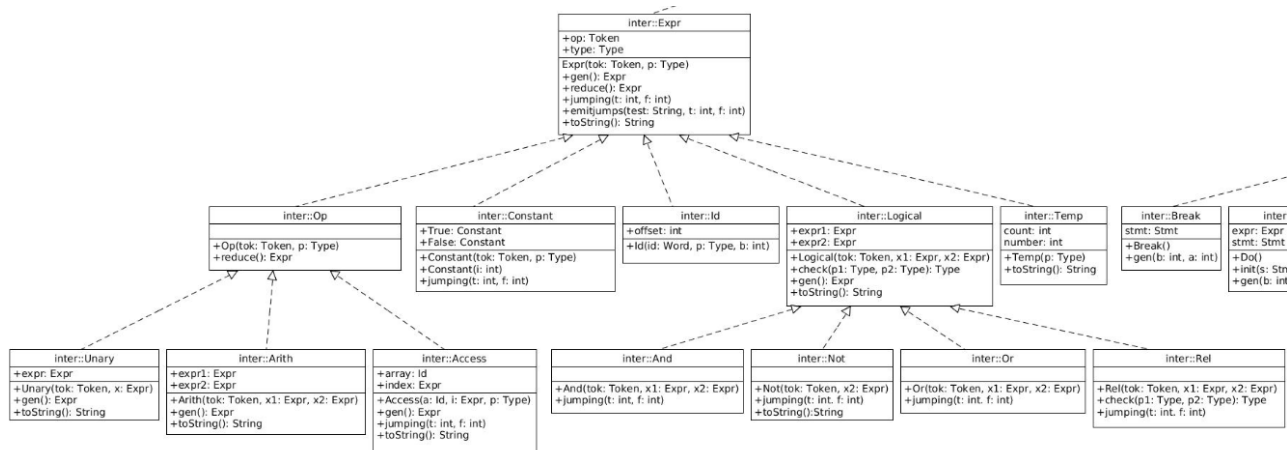


图 5 中间代码生成 inter 包 (2)

- Expr 类下分五个类，分别是算术运算 Op 类，常量 Constant 类，标识符 Id 类，逻辑运算 Logical 类以及临时变量 Temp 类；
- Op 类下分三个类，分别是单目运算 Unary 类，双目运算 Arith 类以及数组运算 Access 类。其中的 Unary 类只负责负号运算，取反运算在 Not 类中进行。
- Logical 类下分四个类，分别是取和运算 And 类，取反运算 Not 类，取或运算 Or 类以及关系运算 Rel 类。

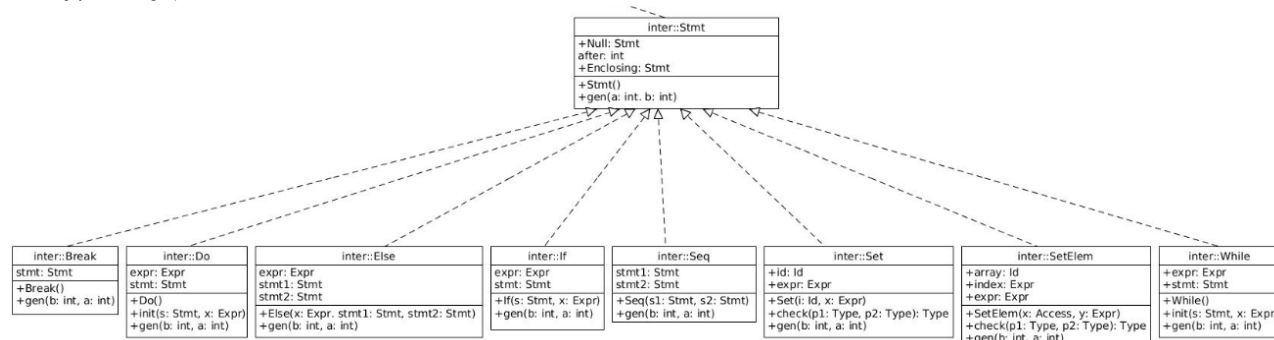


图 6 中间代码生成 inter 包 (3)

Stmt 类下有 8 个类，分别是

- Break 类为 Break 语句
- Do 类为 do stmt while (bool)语句
- Else 类为 if (bool) stmt else stmt 语句
- If 类为 if (bool) stmt 语句
- Seq 类为语句序列，即 stmt; stmts
- Set 类为基本变量的赋值语句，即 Id = expr
- SetElem 类为数组元素的赋值语句，即 id[i] = expr
- Whiel 类为 while (bool) stmt 语句

语法生成器 parser 包

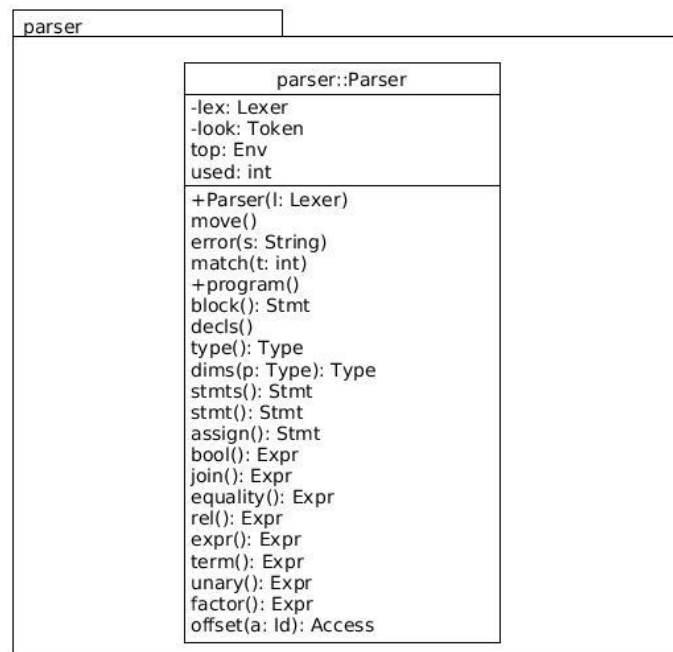


图 7 语法生成器 parser 包

语法生成器为本编译器最核心的部分，它结合了 lexer，inter 以及 symbols 包，使用递归下降算法，构建语法树，最终生成中间代码并打印，后续将会详细分析。

加入注释

本人结合附录 A 文档以及课本第二章，第五章和第六章，为此编译器源码加入 java 风格的注释，并生成了 javadoc 文件，存放在/doc 文件夹下。

Class	Description
Access	数组运算
And	和运算
Arith	双目运算符表达式节点。
Break	Break语句
Constant	布尔常量True和False以及数值常量
Do	Do stmt while (expr)语句
Else	if (expr) stmt1 else stmt2语句
Expr	表达式构造类
Id	标识符
If	if (Expr) S 语句
Logical	逻辑运算表达式，做为AND, OR, NOT的父类，提供公用的功能

图 8 JavaDoc 文档

另外，本项目在 eclipse 下编译运行。使用 Debug 模式，结合断点跟踪，加上 IDE 提供的自动显示注释功能，可以了解代码的详细运作过程提供极大帮助。如下图：

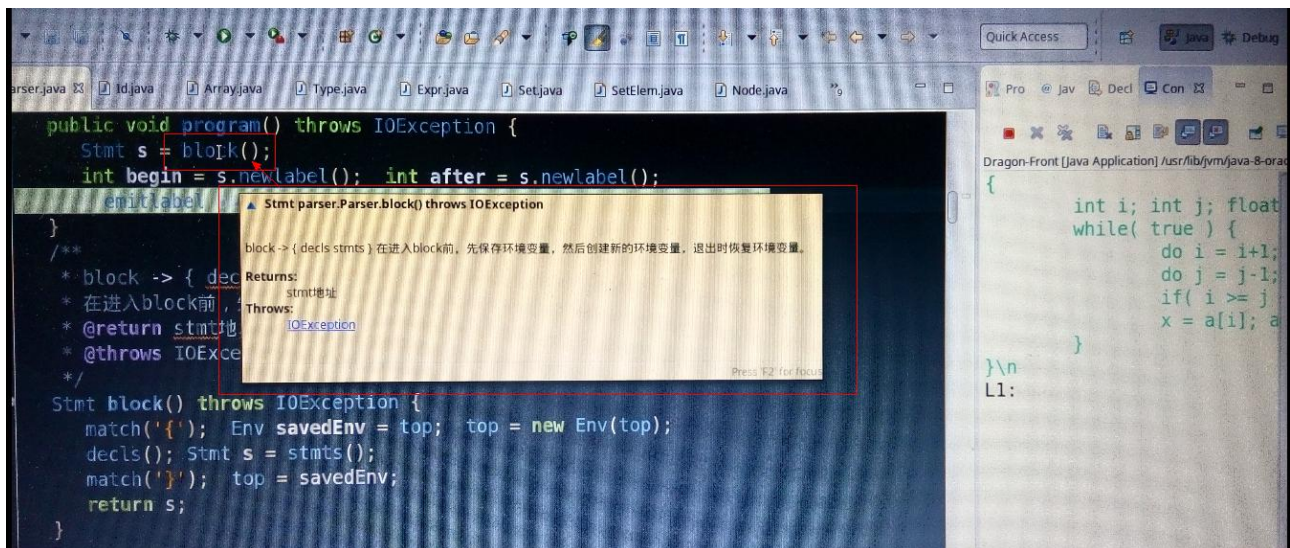


图 9 断点调试阅读源码

运行结果

运行方法有如下两种。

Makefile 文件

项目中提供了 Makefile 文件，在终端下，进入/src 文件夹，运行 make。即可编译代码并运行测试代码。

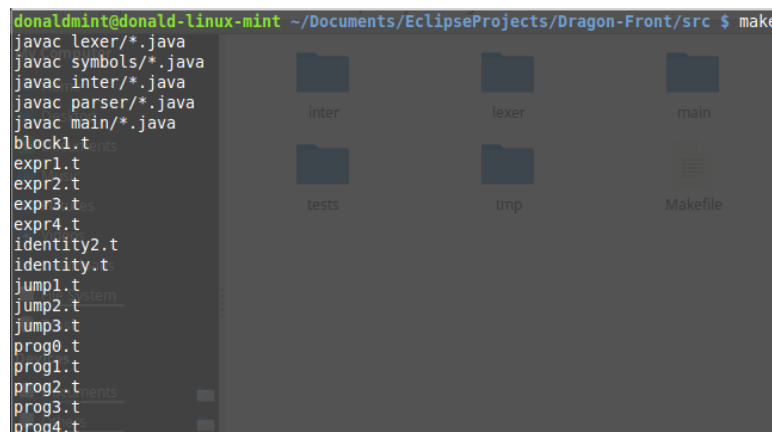


图 10 make 命令

测试文件是/src/tests 文件夹下后缀为.t 的文件，生成的结果为/src/tmp 文件夹下后缀为.i 的文件，另外，/src/tests 文件夹下后缀为.i 的文件存放了正确的三地址码，用于与编译器生成的三地址码进行比较。

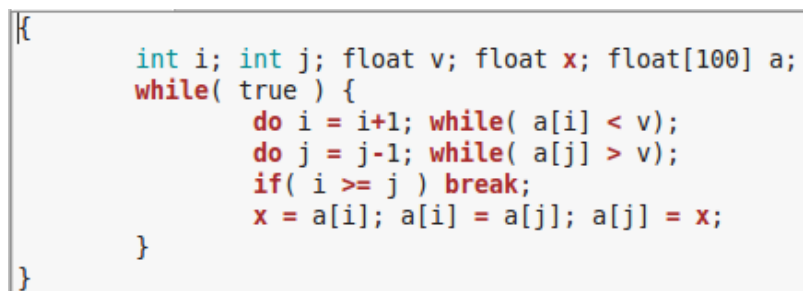


图 11 prog0.t


```

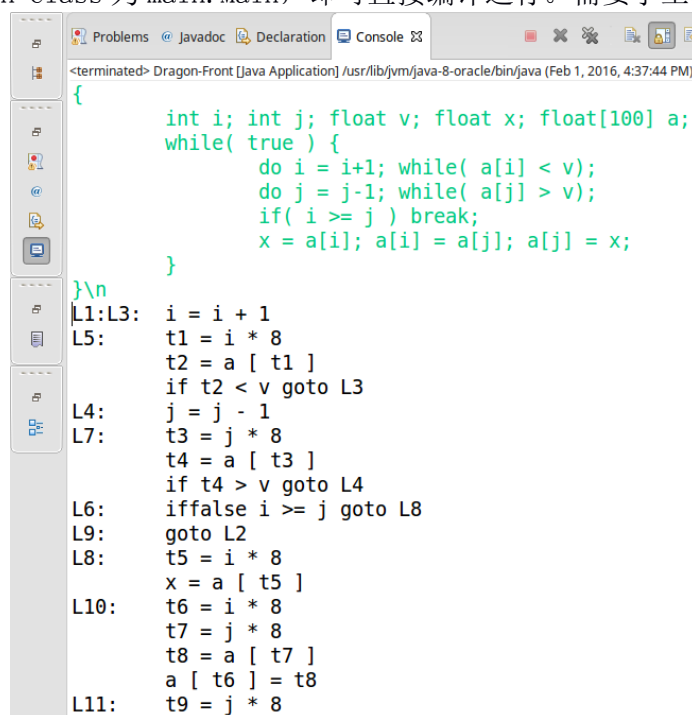
L1:L3:  i = i + 1
L5:      t1 = i * 8
          t2 = a [ t1 ]
          if t2 < v goto L3
L4:      j = j - 1
L7:      t3 = j * 8
          t4 = a [ t3 ]
          if t4 > v goto L4
L6:      iffalse i >= j goto L8
L9:      goto L2
L8:      t5 = i * 8
          x = a [ t5 ]
L10:     t6 = i * 8
          t7 = j * 8
          t8 = a [ t7 ]
          a [ t6 ] = t8
L11:     t9 = j * 8
          a [ t9 ] = x
          goto L1
L2:

```

图 12 prog0.i

Eclipse 运行

在 Eclipse 中配置 Main Class 为 main.Main，即可直接编译运行。需要手工输入代码。



```

<terminated> Dragon-Front [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Feb 1, 2016, 4:37:44 PM)
{
    int i; int j; float v; float x; float[100] a;
    while( true ) {
        do i = i+1; while( a[i] < v);
        do j = j-1; while( a[j] > v);
        if( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
L1:L3:  i = i + 1
L5:      t1 = i * 8
          t2 = a [ t1 ]
          if t2 < v goto L3
L4:      j = j - 1
L7:      t3 = j * 8
          t4 = a [ t3 ]
          if t4 > v goto L4
L6:      iffalse i >= j goto L8
L9:      goto L2
L8:      t5 = i * 8
          x = a [ t5 ]
L10:     t6 = i * 8
          t7 = j * 8
          t8 = a [ t7 ]
          a [ t6 ] = t8
L11:     t9 = j * 8
          a [ t9 ] = x
          goto L1
L2:

```

图 13 prog0.t 在 eclipse 下运行

Parser 详解

附录 A.1 中提供的文法包含有左递归，要使用递归下降方法编写语法分析器，就必须消除左递归。以下是按照 Parser 类的源码分析出来的消除左递归之后的文法。

program -> block
block -> { decls stmts }
decls -> decl decls <i>epsilon</i>
decl -> type id
type -> basic dims
dims -> [num] dims <i>epsilon</i>
stmts -> stmt stmts <i>epsilon</i>
stmt -> ;
if (bool) stmt
if (bool) stmt else stmt

while (bool) stmt do stmt while (bool) ; break ; block assign
assign -> id offset = bool ;
offset -> [bool] offset <i>epsilon</i>
bool -> join join bool
join -> equality equality && join
equality -> rel rel == equality rel != equality
rel -> expr expr < expr expr <= expr expr >= expr expr > expr
expr -> term term + expr term - expr
term -> unary unary * term unary / term
unary -> - unary ! unary factor
factor -> (bool) num real true false id offset

表格 1 消除左递归后的文法，其中 *epsilon* 指的是空语句

递归下降的方法，优点就是代码比较容易实现，基本上按照消除左递归之后的文法，对于每个非终结符创建一个函数，就可以大概实现了。

在 main.Main 中，当创建 lexer 和 parser 之后，就会调用 parser 的 program() 函数，而 program 中又会调用 block，依此递归下降，就可以构建出整棵抽象语法树。

然后，在 program 函数中，当完成 block 的调用之后，就会开始执行三地址代码的生成以及打印过程。方法就是从最后 return 回来的第一个 stmt 语句开始，遍历语法树，调用 gen 函数，逐步生成中间代码。构建完抽象语法树之后，都规约为了 stmt stmts 这样的语句序列，所以从 Seq.gen 开始，不断递归调用 gen，最后完成整个中间代码的生成。

另外，由于 label 并不是要打印时才创建的，而是在进入函数时便开始创建，这就导致了最终 label 的大小并不是逐渐增大的。

编程细节

toString 函数

toString 函数主要是用来返回某个类对象的表示形式的，比如数字等可用在三地址码生成中。在 toString 函数中，可以注意到作者将其他类型的变量转换为 String 类型时，是通过使用 “” + 变量的方法来实现的，比如

```
public String toString() { return "" + value; }
```

这样便省去复杂的转换，值得学习。

Parser 中的 match 函数

```
parser.Parser.java
void match(int t) throws IOException {
    if( look.tag == t ) move();
    else error("syntax error");
}
```

Match 其实是对预读符号的判断，如果读取到的词法单元的 tag look 符号文法中的其他 tag t，那么就调用 move 函数来继续读取下一个词法单元。这样，即起到了匹配作用，又能够继续读取下一个 token 使得 lexer 继续运行，确实是一个很用心的设计。

文法没有提取左公因子

可以看到在最后的文法中，并没有进行左公因子的提取。在源码中，其实是先将公因子规约，然后通过 look 来预读下一位，如果是空的，就 return；而非空，就对符合要求的文法进行展开。而对于

```
expr -> term + expr
```


诸如此类的文法，则是通过一个 while 循环来实现的。

parser.Parser.java

```
Expr expr() throws IOException {
    Expr x = term();
    while( look.tag == '+' || look.tag == '-' ) {
        Token tok = look; move(); x = new Arith(tok, x, term());
    }
    return x;
}
```

先求出前面的算术运算式 Arith，返回一个单一地址来更新 x，然后继续 while 循环。这样便达到了文法中可以不断展开的想法，同时又实现了左结合。

Env 作用域

在进入一个 block 或循环时会先保存之前的环境变量，然后创建新的环境变量，再结束 block 时再恢复之前保存的环境变量。

parser.Parser.java

```
Stmt block() throws IOException {
    match('{'); Env savedEnv = top; top = new Env(top);
    decls(); Stmt s = stmts();
    match('}'); top = savedEnv;
    return s;
}
```

注意到在创建当前的环境时，使用了原先的环境地址 top 作为参数，而该参数用于设置新建的 Env 对象的 prev 属性。这样，便可通过 prev，向上查找外部环境，事实上 Env 中的 get 函数就是这么实现的。

symbols.Env.java

```
public Id get(Token w) {
    for( Env e = this; e != null; e = e.prev ) {
        Id found = (Id) (e.table.get(w));
        if( found != null ) return found;
    }
    return null;
}
```

通过这样的方法，实现了链接符号表。

一些疑惑

Id 中的 offset 属性

在 parser 的 decls() 函数中，会不断累加已经声明的变量所占用的空间 used，然后用来为新创建的变量的 offset 属性赋值。

parser.Parser.java

```
void decls() throws IOException {
    while( look.tag == Tag.BASIC ) { // D -> type ID ;
        Type p = type(); Token tok = look; match(Tag.ID); match(';');
        Id id = new Id((Word)tok, p, used);
        top.put( tok, id );
        used = used + p.width;
    }
}
```

inter.Id.java

```
public Id(Word id, Type p, int b) { super(id, p); offset = b; }
```

然而，在三地址码的生成中并不需要用到这些信息，不太理解作者这样做的意义。

生成中间代码时跟踪行号

在讲解构造抽象语法树创建新节点时，会跟踪文本行号，可用于报告错误。作者在此提到，把生成中间代码时跟踪行号的任务留给读者。

- 如果是跟踪三地址码在原文本的行号，那么只需要在 Node 的 emit 函数中打印出 lexline 就可以了

```
lexer.Node.java
System.out.println("\t" + lexline + "\t" + s);
```

- 如果是跟踪三地址码本身的行号，那么只需给 Node 添加一个静态的 count 属性，然后每次调用 emit 时，将 count 自增一便可以。

```
lexer.Node.java
static int count = 0;
public void emit(String s) {
    count++;
    System.out.println("\t" + count + "\t" + s);
}
```

不过，无论是哪一个方法，都没有任何实质性意义。所以，不太理解所谓的生成中间代码时跟踪行号是什么意思。

Parser 中的文法与原来文法有差异

在 Parser 中，注意到存在如下语句

```
parser.Parser.java
Stmt stmt() throws IOException {
.....
    switch( look.tag ) {
    case ';' :
        move();
        return Stmt.Null;
.....
}
```

也就是说存在文法

```
stmt -> ;
```

但是，在 A.1 中定义的语言文法中，这样的文法是不存在的。后来注意到作者是为了允许存在诸如语句

```
If ( a == b ) ;
```

这样确实合理，但这又会导致语句

```
If ( a == b ) ; else ;
```

以及语句

```
{ ; }
```

同样也是合法的，这就有点不太好理解了。

所以，为了与 A.1 中定义的文法相同，同时又让语言更加合理，斯以为应该将 “;” 这个 case 去除。

参考文档

《编译原理》龙书附录 A——一个完整的编译器前端