语法产生式逻辑解释

产生式

```
/*0*/              "<<ACC>>-><<START>>",
=> 这个其实是增广文法里添加的开始符号的产生式，并不会用到

/*1*/              "A-><<ID>>:=<<EXPR>>",
=> id := E ，赋值语句

/*2*/              "<<BAND>>-><<BNOT>>",
=> not的布尔表达式归约为and的布尔表达式
/*3*/              "<<BAND>>-><<BAND>>andM<<BNOT>>",
=> and的布尔表达式和not的布尔表达式一起归约为and的布尔表达式（其实就是多个布尔表达式连在一起的递归归约）

/*4*/              "<<BCMP>>-><<EXPR>><<RELOP>><<EXPR>>",
=> 表达式(expression)和表达式一起做布尔运算(relop)，归约为布尔比较运算表达式(boolean_comparison)
/*5*/              "<<BCMP>>-><<EXPR>>",
=> 表达式归约为布尔比较运算表达式（为什么看起来归约反了？因为这是为了解决括号等优先级问题产生的辅助用的式子）
/*6*/              "<<BCMP>>->(<<BEXPR>>)",
=> 带括号的布尔表达式(boolean_expression)归约为布尔比较运算式

/*7*/              "<<BEXPR>>-><<BAND>>",
=> 布尔and表达式归约为布尔表达式
/*8*/              "<<BEXPR>>-><<BEXPR>>orM<<BAND>>",
=> 布尔表达式or布尔and表达式归约为布尔表达式（同3，递归归约，这两个结构不同的原因是and运算符优先级高于or）

/*9*/              "<<BNOT>>-><<BCMP>>",
=> 布尔比较运算表达式归约为布尔not表达式（这也是解决优先级问题的）
/*10*/              "<<BNOT>>->not<<BNOT>>",
=> not布尔not表达式（即取反）归约为布尔not表达式（即可以反复取反）

/*11*/              "<<ELIST>>-><<EXPR>>",
=> 算法4.7不做可以不用理解这条
/*12*/              "<<ELIST>>-><<ELIST>>,<<EXPR>>",
=> 同上

/*13*/              "<<EXPR>>-><<NEG>>",
=> negative表达式（这个算是一个中间辅助非终结符？用来下面给term取反用的）归约为表达式
/*14*/              "<<EXPR>>-><<EXPR>>+<<TERM>>",
=> 表达式+项目（可以在23和24里取反）归约为表达式

/*15*/              "<<FACTOR>>-><<ID>>",
```

=> ID（可以是用户自己定义的变量or常量or数字等，反正用户输入的不是已有的终结符都会被规约成ID）规约成因子（factor=>后面用来乘法运算）

```
/*16*/              "<<FACTOR>>->(<<EXPR>>)",
```

=> 带括号的表达式归约为因子（也是乘法用）

```
/*17*/              "<<ID>>->i", // variable_name
```

=> 用户输入的变量or常量or数字等被识别为i，然后规约为ID

```
/*18*/              "L->L;MS",
```

=> 书上的label语句

```
/*19*/              "L->S",
```

=> 同上，label语句

```
/*20*/              "<<LABEL>>-><<ID>>:",
```

=> 用户输入一个词语归约为label语句的label

```
/*21*/              "M->null",
```

=> 书上的，记录下一条语句入口地址所以用空字多出来的非终结符

```
/*22*/              "N->null",
```

=> 书上的，记录下一条语句入口地址所以用空字多出来的非终结符

```
/*23*/              "<<NEG>>-><<TERM>>",
```

=> 项目归约为negative表达式

```
/*24*/              "<<NEG>>->-<<TERM>>",
```

=> 带单目运算符-的项目归约为negative表达式

```
/*25*/              "<<OPENSTMT>>->if<<BEXPR>>thenM<<STMT>>",
```

=> if表达式归约为开放语句（open_statement，专门为if语句设置的中间过程）

```
/*26*/              "<<OPENSTMT>>->if<<BEXPR>>thenMSNelseM<<OPENSTMT>>",
```

=> 同上，多了个else

```
/*27*/              "<<RELOP>>->=",
/*28*/              "<<RELOP>>->!=",
/*29*/              "<<RELOP>>-><",
/*30*/              "<<RELOP>>-><=",
/*31*/              "<<RELOP>>->>",
/*32*/              "<<RELOP>>->>=",
```

=> 以上都是布尔运算符的规约过程

```
/*33*/              "S->if<<BEXPR>>thenMSNelseMS",
```

=> 这个跟26有什么关系我也不清楚，，反正都是中间过程。。。自己写的时候反正按照产生式表一步一步写总能写对的

```
/*34*/              "S->A",
```

=> 赋值语句(assignment)归约为语句(statement，和上面的EXPR优先级不同的另一种表达式，下面还有个优先级更低的STMT)

=> 顺便说下在规约过程中越靠近开始符号的非终结符优先级越低。(因为越远的越先规约)

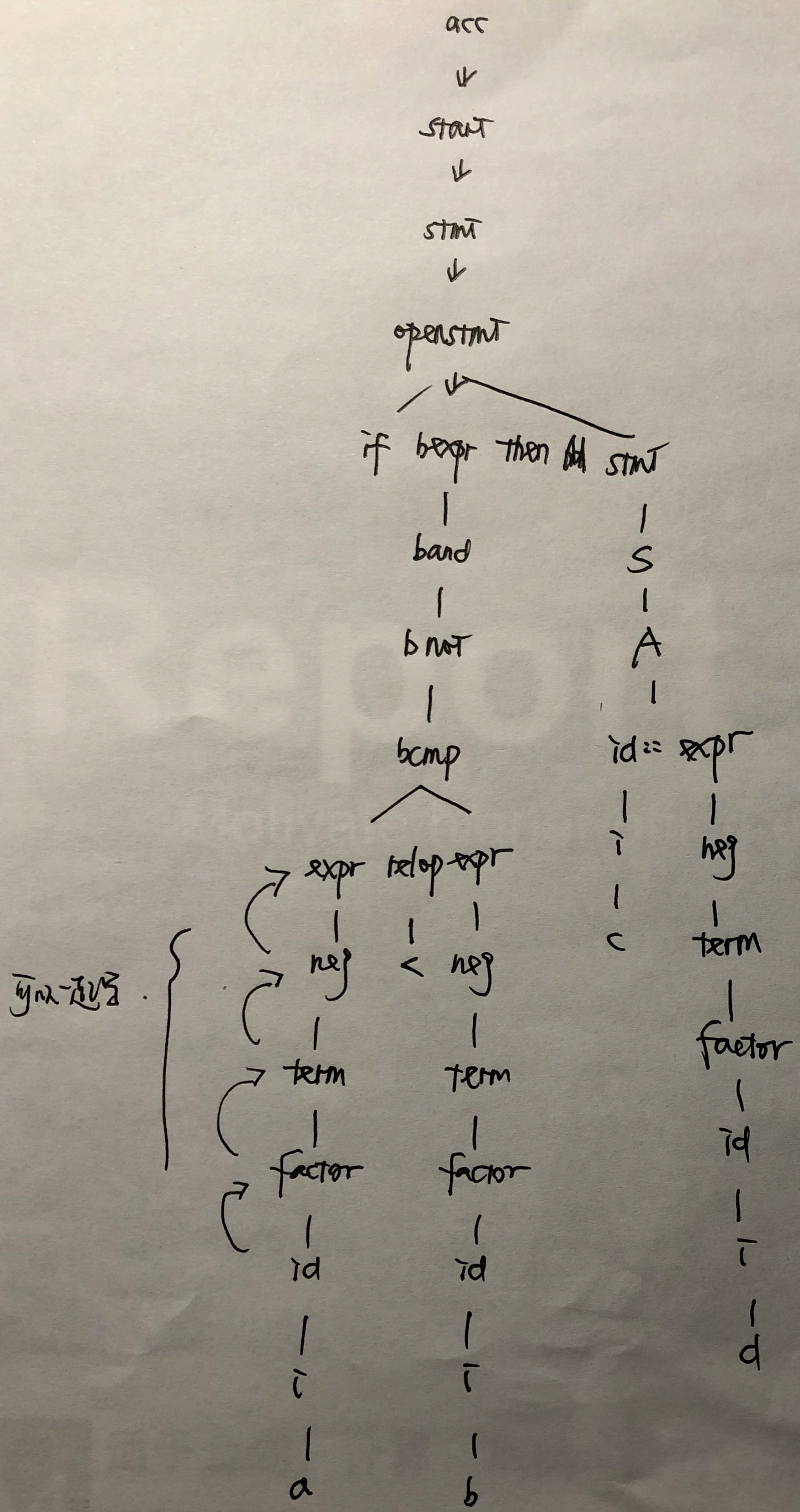
```
/*35*/              "S->{L}",
```

=> label语句的嵌套归约

```
/*36*/            "S->whileM<<BEXPR>>doMS",
=> while语句归约为statement
/*37*/            "S->call<<ID>>(<<ELIST>>)",
=> 过程调用，算法4.7的不用管
/*38*/            "S->labelS",
=> label语句归约为statement
/*39*/            "S->goto<<ID>>",
=> goto语句归约为statem

/*40*/            "<<START>>-><<STMT>>",
=> 开始符号归约为语句

/*41*/            "<<STMT>>->S",
=> 解决优先级的语句归约为语句
/*42*/            "<<STMT>>-><<OPENSTMT>>",
=> 开放语句归约为语句

/*43*/            "<<TERM>>-><<FACTOR>>",
=> 因子归约为项目
/*44*/            "<<TERM>>-><<TERM>>*<<FACTOR>>"
=> 乘号的递归，项目*因子归约为项目
```

举例

验证自己写的内容能不能正确识别一个表达式，可以像这样画一个语法树然后观察测试的结果跟这个过程是不是一致。通过画语法树也可以知道要实现一个功能需要实现哪些产生式的规约才能满足。


acc
start
stmt
openstmt
if bexpr then stmt
band
bnot
bcmp
expr relop expr
neg < neg
term term
factor factor
id id
i i
a b
S
A
id := expr
i
c
neg
term
factor
id
i
d

⇒ if a<b then c:=d