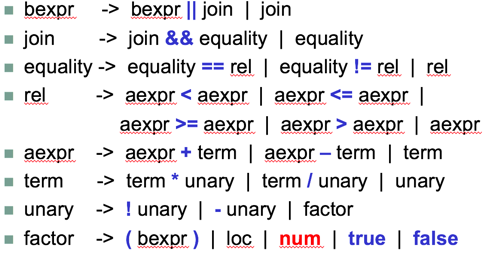
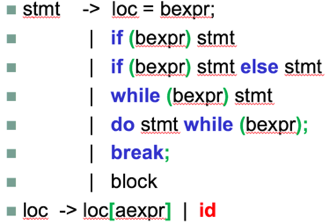
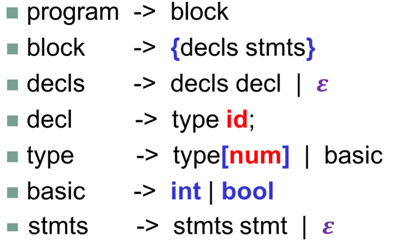
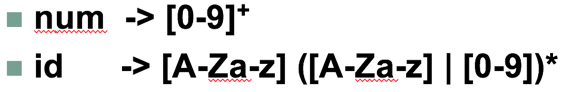
# 一、原文法及词法

考虑如下源语言

以及词法



可以看出该语言的所有语句必须放在一个block里，即一对花括号间，仅有int型和bool型，有数组，也可以对数组进行算数及布尔运算，如有条件语句有循环语句以及跳转语句(break)，表达式大致可分为布尔表达式(bexpr)与算数表达式(aexpr)，且两者之间可以相互嵌套。

注意到数组的申明方式类似于JAVA，而不同于C，即 且所有变量的申明必须放在block的最前面，不能申明时就赋初值，也不能一次申明多个变量。没有函数。

# 二、词法分析

通过对正则词法建立一个确定有限自动机，对每次输入的字符进行相应的状态转换，并将识别出的id标识符与保留的单词作对比以正确识别关键字。最后将输入流统一转换成以词为单元的数组，送入语法分析程序。

在做词法分析前可以先对输入流进行预处理，以去掉注释内容（以双斜线开始至行尾）。

# 三、语法分析

### 消除左递归并提取左公因子

首先对原文法消除左递归并提取左公因子，然后计算文法的候选首符集，判断是否所有产生式的首符集都两两不相交，最后还要计算非终结符的Follow集，以判断是否有First-Follow冲突。满足以上条件的才是LL(1)文法。

产生的新文法如下:

1. program -> block
2. block -> {decls stmts}
3. decls -> decl decls | ε
4. decl -> type id;
5. type -> basic type’
6. type’ -> [num]type’ | ε
7. basic -> int | bool
8. stmts -> stmt stmts | ε
9. **stmt -> loc = bexpr;**

**| if (bexpr) stmt elsmt**

**| while (bexpr) stmt**

**| do stmt while (bexpr);**

**| break;**

**| block**

1. **elsmt -> else stmt | ε**
2. loc -> id loc’
3. loc’ -> [aexpr]loc’ | ε
4. bexpr -> join bexpr’
5. bexpr’ -> || join bexpr’ | ε
6. join -> equality join’
7. join’ -> && equality join’ | ε
8. equality -> rel equality’
9. equality’ -> == rel equality’ | != rel equality’ | ε
10. rel -> aexpr rel’
11. rel’ -> < aexpr | <= aexpr | >= aexpr | > aexpr | ε
12. aexpr -> term aexpr’
13. aexpr’ -> +term aexpr’ | -term aexpr’ | ε
14. term -> unary term’
15. term’ -> \*unary term’ | /unary term’ | ε
16. unary -> !unary | -unary | factor
17. factor -> (bexpr) | loc | num | true | false

可以发现，因此该文法并不是严格的LL(1)文法，但对嵌套的if-else语句规定就近结合，即可消除该产生式的歧义性，不妨碍继续进行LL(1)的文法分析。

### 递归下降分析

对于该文法，我们都可以构造一个由一组递归过程组成的不带回溯的自上而下的分析程序，其中每个过程对应于文法的一个非终结符。建立一个由单词组成的序列，用sym指针指向该序列中的当前正在分析的单词，每当与产生式的某一终结符匹配，便令sym指向序列中的下一个单词。具体过程如同课本上的样例，此处不再赘述。

# 四、符号表建立

由于变量应当仅在当前的block及更内层的block起作用，且一个程序会有很多同级或者多层嵌套的block，因此需要通过一个全局变量topTable指针，指向当前的block的符号表。每次遇到block的产生式，便需要在当前block下建立一个子block，并将topTable指针指向该子block的符号表，每当退出该block时，便又将指针指回上一层block。

具体建表时，根据属性文法的规则，给每个非终结符加上相关的综合属性和继承属性，不妨以i存储继承属性，tp存储标识符的类型，space存储该变量所占的字节数，遇到与符号表相关的产生式时便依照下面花括号中相应的动作进行操作。使用指向一个Type结构体的全局指针变量currType来指代当前产生式，

1. **struct** Type{
2. **int** typeCode;
3. **int** size;       //元素个数
4. Type \*elem, \*parent;
5. Type();
6. **void** clear();
7. **int** space();   //计算所占空间大小
8. **int** dim() **const**;    //计算维数 非零即数组
9. Type\* derive(**int** size\_);   //以自身为蓝本衍生新的结构体变量
10. };

使用其方法derive()进行继承操作，用其成员typeCode代表类型。

### 类型推断

### 变量使用检查

其中array函数构造一个数组类型，且可以嵌套构造；add函数则向当前符号表添加变量及相关信息；search函数则在当前符号表寻找该变量，未找到则继续在上一层符号表查找，知道找到该变量为止，否则该变量未经定义，报错。

# 五、属性文法翻译

同符号表建立过程类似，为各个产生式引入继承属性和综合属性，以i存储继承属性，s存储综合属性，offset存储偏移量，place存储变量的地址。通过一个指向Syntax结构体的全局指针currSyntax指代当前产生式，

1. **struct** Syntax{
2. Operand \*i, \*s;
3. Operand \*place, \*offset;  // offset用于数组元素的引用
4. Syntax \*parent;
5. Syntax();
6. **void** clear();
7. Syntax\* derive();
8. };

### 算数表达式的翻译

其中每次产生一个新临时变量时，都会将该变量插入到相应的符号表，函数emit产生一个四元式并插入存储四元式序列的全局变量sequence中。

### 数组元素的引用

根据书上的内容，一个数组的相对地址可通过如下公式计算，

代表数组在该维度的大小。由于下标从0开始，得出。以i存储继承属性，place存储变量的地址。

其中函数dimsize计算得出该数组当前维度的大小。isInRight代表当前变量是否在赋值号右边，若是则需要先将该数组元素的值先存入一个新的临时变量。

### 赋值语句的翻译

对于普通变量与数组的赋值语句翻译略有不同。

### 布尔表达式的翻译

与或非的翻译方式与算数表达式大体一样，对关系运算的翻译则用到了跳转语句，需要根据具体情况对一个新变量赋值为真或假，其中nextquad返回下一条四元式的序号。可以发现本程序是采用的数值表示法来翻译布尔表达式，该方法对于此文法来说较为方便。由于根据文法，bexpr无法产生出赋值表达式，且该语法也不含有函数，因此不论是用数值表示法或是将布尔式作为条件控制语句翻译，其结果是一致的，不会有副作用。

### 控制语句的翻译

其中quadNum用来储存某条四元式的序号，用于回填时的定位。Backpatch(quad1, quad2)函数会将序号为quad1的跳转四元式的转移地址填为quad2。可以发现上面的控制语句的翻译过程与书本上的有些不同，但都用到了回填机制。对于break语句的翻译，需要用到一个list容器储存一个block或stmt里面的所有最上层的break所产生的无条件跳转语句，然后在循环语句结束后回填breaklist，使得break能直接跳出当前循环。

# 六、代码优化

代码优化则主要是先将四元式序列划分为一系列的基本块，通过删除公共子表示式，删除无用代码，合并已知量来实现，使得代码更精简，运行效率也更高。

可以发现在生成数组引用产生式的代码时，如对于，引用时总会在最开始生成两个形如

的冗余赋值代码，因为完全可以只生成一条代码即可，甚至这一条也可能是一条复写传播的冗余代码。

除此之外，如果产生式的非终结符，则在非终结符末产生的一条无条件转移语句则会仅会跳转到下一句四元式，相当于正常的顺序执行，因此删掉也不会影响程序的执行过程。

虽然在语义分析的时候也可以通过一些技术上的修改使得不产生这些冗余代码，但将这种冗余代码顺便交给代码优化处理也不失为一种简便的方法。

# 七、程序说明

该文法的编译器通过C++实现，为了使得代码结构更清晰，将代码划分为几个不同的源文件，分别负责词法分析，语法分析，符号表建立，以及语义分析。**由于源码较多，不便在文档中展示，故仅简单介绍各文件，所有源代码可在附件中查看。**

其中头文件分别为*global.h word.h syntax.h table.h semantic.h common.h*。*global.h*中是一系列的枚举型变量的申明，用以表示终结符、非终结符、运算操作、操作数类型。*common.h*则是语法分析及语义分析都要用到的全局变量的申明。*word.h syntax.h table.h semantic.h*分别是词法分析、语法分析、符号表建立、语义分析所用到的结构体及函数的申明。

源文件则有 *parser.cpp word.cpp syntax.cpp table.cpp semantic.cpp common.cpp*。其中main函数在*parser.cpp*中，由*parser.cpp*调用词法、语法、语义分析，先通过词法分析将输入代码转换为单词序列，传给语法分析，通过一次遍历完成四元式的产生。

编译则通过如下命令：

1. g++ -o parser.exe parser.cpp word.cpp table.cpp syntax.cpp semantic.cpp common.cpp

程序在g++ 4.9.2下编译通过，生成可执行文件parser.exe，输入待分析的程序代码，输出四元式序列。

如果词法分析时遇到了非法字符，将会报错并提示。语法分析时遇到不满足语法的语句同样也会报错并提示。如果出现的变量未经申明便使用，或者数组变量的错误引用，同样也会报错。

对于如下满足文法的示例代码（注释将会在词法分析的预处理时删除），

1. {   //矩阵乘法解析示例 //input\_matrix\_mul.txt
2. **int**[4][6] matrix1;
3. **int**[6][7] matrix2;
4. **int**[4][7] result;
5. **int** row1;
6. **int** row2;
7. **int** col1;
8. **int** col2;
9. row1 = 4;
10. col1 = 6;
11. row2 = 6;
12. col2 = 7;
13. //省略矩阵赋值操作
14. **if**(col1==row2){  //两矩阵可相乘的条件
15. **int** i;      //每个block前都可以声明新变量
16. i = 0;
17. **while**(i<row1){
18. **int** j;
19. j = 0;
20. **while**(j<col2){
21. **int** k;
22. k = 0;
23. result[i][j] = 0;
24. **while**(k<row2){
25. result[i][j] = result[i][j]+matrix1[i][k]\*matrix2[k][j];
26. k = k+1;
27. }
28. j = j+1;
29. }
30. i = i+1;
31. }
32. }
33. }

上方的序列并未进行优化处理，每行的第一个数字代表该四元式的序号，随后便是一个四元式。其中代表将赋值号右边数组的元素赋值给变量，代表将变量赋值给赋值号左边数组的元素。

JZ,JE,JMP,JL等转移指令与汇编一致，但对JZ与JE稍作区分，JZ代表单个操作数为0（false）则跳转，JE代表两个操作数相等则跳转。

生成的四元式序列将会输出到一个文件交由代码优化器继续进行处理。