# **词法分析概要设计**

## 整体设计图

## 各个cpp的内容介绍

1. 自动机类 Automaton（定义于Lex.h）

存储自动机的节点和边信息。

边的保存方法为：(((uint64)(x)<<32)+(uint64)(y))

主要方法为：

Automaton()：构造一个空自动机

Automaton(char ):构造一个仅包含一个字符的自动机

Automaton(Automaton\*)复制一个自动机

1. 各类工具函数（LexTool.cpp）

全局变量stateIndex 用于不重复的命名所有节点

Automaton\* merge(Automaton\*,Automaton\*)合并两个数据库

Automaton\* closure(Automaton\*)为当前自动机生成闭包

Automaton\* connect(Automaton\*,Automaton\*)连接两个自动机

edges getEdged(int,map<unsigned long long ,bitset<256>>从边的集合中查询以指定节点为起点的边。

1. 由正规式生成NFA（REtoNFA.cpp）

readRE(string) 从指定路径中读取词法

GRC(string) 生成可识别字符的位图

REtoNFA(vector<int>)递归的将正规式转换为自动机

processLex()：控制整个过程，将每个自动机和终态配对

mergeMultiA(map<int,string>)将生成的自动机并联起来，生成一个大自动机。

1. 由NFA生成DFA（NFAtoDFA.cpp）

findEqualS(map<uint64,bitset>,unordered\_set<int>,int)：从边集中找到所给点的所有等价状态，保存到指定集合中。

findTargetS(map<uint64,bitset>,unordered\_set,int,char)：从边集中找到指定点经过指定字符到达的点的集合

isSubsetOfOne(vector<unordered\_set<int>>,unordered\_set<int>)：查询指定点集是否是所给点集的子集。

checkFinal(unoirdered\_set,map<int,string>,map<int,string>,int)：检查所给点集是否包含终态，是则将终态和点集配对并添加到新终态集中。

NFAtoDFA(Automaton,map<int,string>,map<int,string>):化简自动机后生成一个新自动机和相应的终态集。

1. DFA的化简（SimplifyDFA.cpp）

getAllChars(Automaton)：从DFA中获取到所有的可转移字符，存入指定集合中。

getAllStates(Automaton)：从DFA中获取到所有的状态，存入指定集合中。

getEndState(Automaton, int, int)：查询起始状态接受字符转移到的状态。

getSplitSetsIndex()：重新记录所有状态所在集合的下标，便于状态集合的划分。

split(Automaton, unordered\_set<int>)：分离输入的状态集合，成功分离返回true，状态集合不可分返回false。

initSplitSets(const map<int, string>)：对总的状态集合进行初始分配，划分为非终态集和各不同终态分别对应的终态集，为化简做准备。

simplifyDFA(Automaton, map<int, string>, map<int, string>)：化简DFA后生成一个新DFA和相应的终态集。

1. 名字表项类NametabItem（定义于Lex.h）

存储扫描代码文件过程中识别到的词。

index：名字表项序号，-1代表可识别字符（如运算符、分号等）。

name：识别出的词的名字。

type：识别出的词的类型，由输入的文法定义。

value：识别出的词的值。

1. readNext函数（Scanner.cpp）

bitset<MAXCH>recognizeCh：全局变量，用于存储可识别字符。

vector<NametabItem>nametab：全局变量，用于存储名字表。

findName(string)：在名字表中查找是否已有所输入名字，是则返回其下标，否则返回-1。

readnext(NametabItem, Automaton,map<int,string>, string)：每调用一次，在代码文件中扫描，识别并返回一个词，并存储到名字表中。

readCode(string)：从输入的文件路径读取代码文件，为扫描做准备。

## 词法分析类的实现

1. 词法分析总体流程

* 读取词法
* 根据正规式构造NFA
* 通过子集法将NFA转化为DFA
* 利用等价状态法化简DFA

1. 正规式构造NFA

* 从词法中读取一条正规式。
* 将正规式中的字符逐个转换，非运算符或转义字符转化为一个自动机，并用占位符替代，运算符号则拓展为int。如遇到“{”，找到它之后的第一个“}”，从终态集中查找对应的正规式，将其复制一份，插入占位符。
* 在处理完的数组中，找到最后一个“（”及他之后的第一个“）”，之间的部分即为一个“子正规式”，将其取出，对其从头进行这一步，返回最后的自动机，将占位符放在原位；接着处理“|”，此时没有了“（）”的干扰，只需对运算符两侧的自动机做并运算；最后处理连接，从左往右，将相邻的两个自动机连接起来并用一个结果替代，如此重复，直至只剩下一个自动机。
* 处理完所有的正规式后，将所有的自动机的起点连起来，就得到了词法对应的一个完整的NFA

1. NFA转DFA

* 用一个点集的数组来保存所有的子集。
* 先寻找起点的所有等价状态，构成一个集合，保存到数组中，作为接下来循环的起点。
* 进入循环，结束条件为指针指向数组的末尾，指针所指为本次运算处理的点集。对于该点集，先遍历所有的可识别字符，计算能抵达的状态的集合；再计算集合的等价状态的集合；检查该集合是否是已知集合数组中某个集合的子集，是则构建一条由当前正计算的状态到该状态的边，否则将该集合插入数组中，同时检查集合中是否包含终态，是则构建终态到该集合对应关系。

1. DFA的化简

* 首先，从化简前的DFA中记录所有发生转移的字符和所有状态。
* 将初始的状态集合分成终态集和非终态集。其中若有多个不同含义的终态，也应划分为不同的集合。用一个集合数组存储所有划分后的状态集合。
* 记录所有状态所在集合的下标，为划分集合做准备。
* 进入循环，结束条件为状态集合数不再增长，每次处理集合数组中的一个状态集合。对于该状态集合，遍历所有发生转移的字符，对集合内所有状态进行一次转移，然后记录所有状态分别转移到了什么状态，分别在哪些子集中；如果到达状态出现在两个以上的不同子集中，那么将原集合划分为相应数量的集合，插入回集合数组中，进行下一次划分，同时清空被划分的集合；如果到达状态均在同一个子集中，那么不进行划分。
* 去除空集，化简前DFA初态所在的集合作为新自动机的初态，化简前自动机终态所在的集合为新DFA的终态。之后再根据化简前DFA的状态转移情况（边），构造出化简后的DFA。

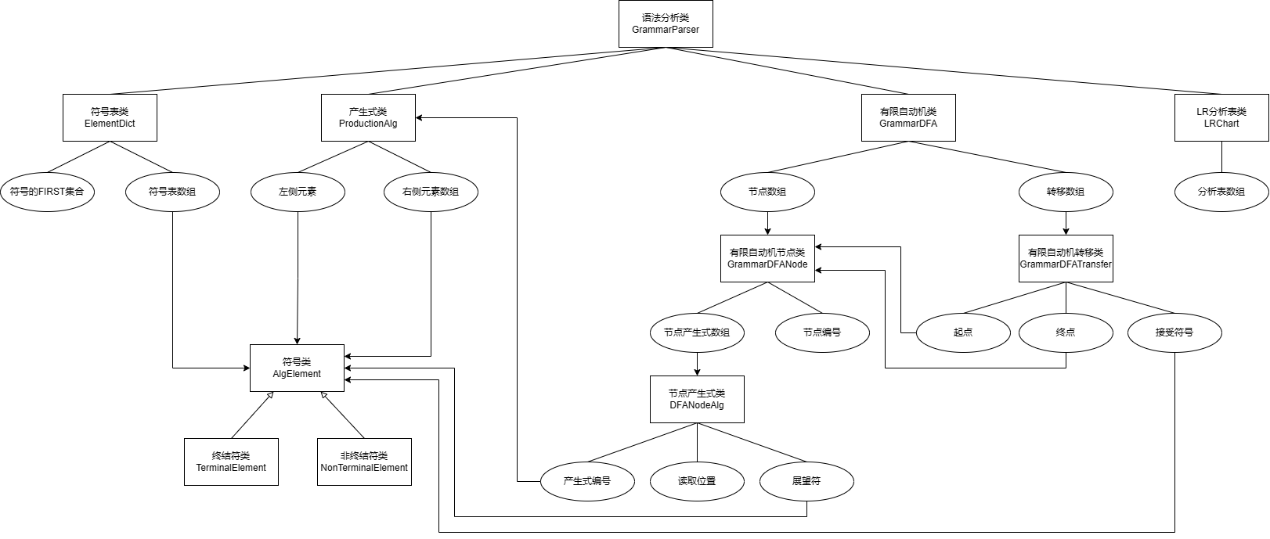
1. readNext函数

* 从输入的代码文件路径中提取出每一行的字符串。
* 进入循环，指针所指为将要扫描的字符，结束条件为指针位置超过了本行的长度。对于每个扫描到的字符，检查自动机当前状态能否接受这个字符；如果可以接受，那么将这个字符添加到识别出的词字符串的尾部，状态更新为转移到的新状态，同时指针后移；如果不能接受，分为以下4种情况：
  + 当前自动机处于初态，扫描到了空格字符：指针后移；
  + 当前自动机处于初态，扫描到了可识别字符：指针后移，返回识别的字符；
  + 当前自动机处于终态：将识别出的词字符串作为名字，在名字表中查找是否已有该名字，若已有，则返回已有名字及其信息；若没有，则作为新名字插入名字表中。之后，返回识别到的名字。
  + 当前自动机不处于终态：报错，并输出具体报错信息。

# **语法分析**

## 概要设计

## 整体设计图



## 语法分析类介绍

1. 语法分析类 GrammarParser

语法分析的主类，协调各个类完成语法分析工作，主要方法包括：

processGrammarRule()：读取并分析语法规则

LR1Main()：根据LR(1)分析表执行语法分析

1. 符号表类 ElementDict

存储语法分析过程中所有出现的符号的信息，包括终结符和非终结符，主要方法包括：

createElem()：在符号字典中添加一个新符号

findElem()：查找符号字典中是否存在某个符号

calculateFirst()：计算所有符号的FIRST集合

classifyAlg()：将产生式按照左侧符号进行分类

1. 产生式类 ProductionAlg

存储产生式的信息，包括产生式左侧的符号与产生式右侧的符号串，主要方法包括：

readFromStr(): 从用户的一行输入读取一个产生式

splitAlg()：将产生式按照候选式符进行拆分

1. 符号类 AlgElement

存储语法分析过程中的符号，包括终结符和非终结符，主要方法包括：

operator==()：判断两个符号是否为同一个符号

1. 有限自动机类 GrammarDFA

存储语法有限自动机的信息，包括节点数组和转移数组，主要方法包括：

buildDFA()：创建有限自动机

expandNodeAlg()：扩展结点的等价产生式

getProspectCh()：计算结点产生式的展望符

1. 有限自动机节点类 GrammarDFANode

存储有限自动机的节点信息，包括该节点对应的状态ID和该节点的产生式数组，主要方法包括：

addAlg()：向节点中添加产生式

1. 有限自动机转移类 GrammarDFATransfer

存储有限自动机的转移信息，包括开始状态、接受字符和结束状态

1. 节点产生式类 DFANodeAlg

存储有限自动机中节点的产生式信息，包括产生式编号、当前读取的位置和展望符，主要方法包括：

createNext()：根据该产生式创建读取一个字符后下一个产生式的信息

isAtEnd()：产生式是否读取到了末尾

1. LR分析表类 LRChart

存储LR分析表的信息，主要方法包括：

build()：构建LR分析表

## 语法分析类的实现

## 语法分析总体流程

* 读取用户输入
* 构建产生式数组，根据识别出的终结符与非终结符元素构建符号表
* 对产生式进行处理，包括拆分含有多个候选式的产生式以及将产生式按照左侧的非终结符进行分类
* 根据产生式数组计算符号表每个元素的FIRST集合
* 从起始符开始构建LR(1)有限自动机，展望符通过FIRST集合计算得到
* 根据有限自动机构建LR(1)分析表

## 产生式的读取与符号表的识别

对于用户的每一行输入，按照以下流程读取：

* 每次读入一个符号，该符号是一个非终结符或由一对大括号构成的终结符
* 查找该符号是否在符号表中，如果不在符号表中则在符号表中插入该符号
* 先读取产生式的左侧符号，再逐个读入产生式右侧的符号

按照以上方法读入多行输入，可以得到一个产生式数组，然后对产生式数组进行拆分：

* 检查产生式每一个右侧符号，如果符号为候选式符( | )，则将后续的符号串拆分成一个新的产生式
* 根据产生式左侧的符号对产生式进行归类

## FIRST集合的计算

1. 原理描述：

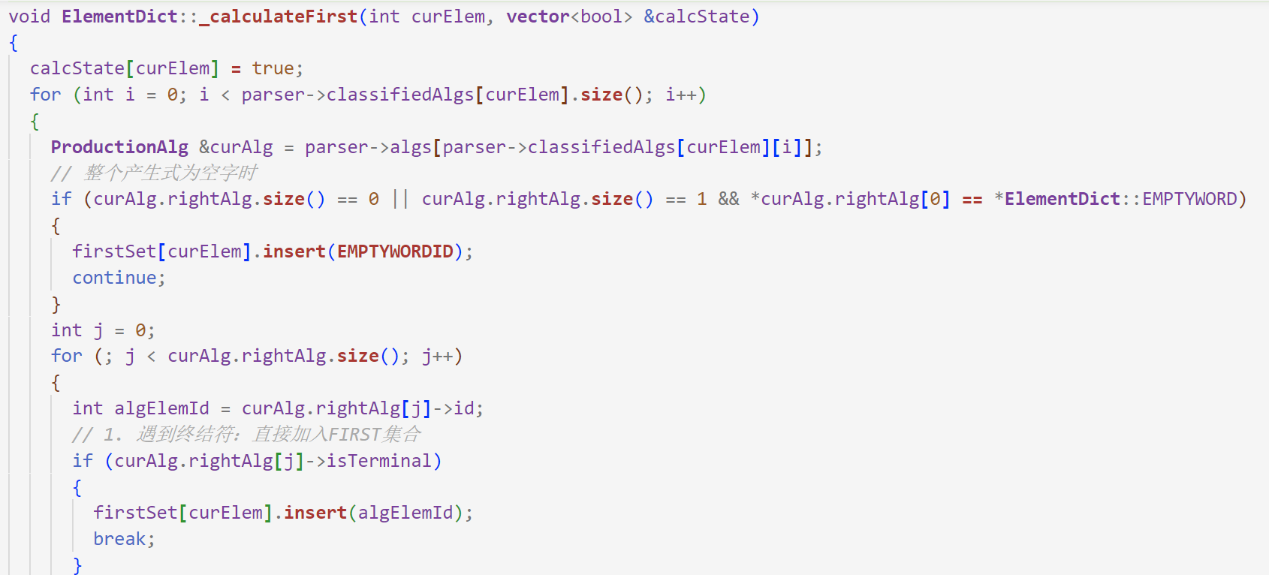
若X为终结符，则 FIRST(X) = X；对于符号串 Y=X1X2...Xn，构造 FIRST(Y)集合：初始化FIRST(Y) = FIRST(X1) - {ε}，找到Xi使得对所有Xj，1 <= j <= i-1, ε∈FIRST(Xj), 把 FIRST(Xj) - {ε} 加入 FIRST(Y)中；将 FIRST(Xi) 加入 FIRST(Y) 中

1. 对于每一个非终结符，按照以下流程计算该终结符的FIRST集合：

* 获取该非终结符的所有产生式
* 逐条读取产生式，从产生式的右侧第一个元素开始向后扫描每个产生式元素
* 如果当前元素是终结符，则将该终结符加入FIRST集合，并结束该产生式的扫描
* 如果当前元素是非终结符，则检查该元素的FIRST集合是否已经计算过，若没有计算过则递归计算该元素的FIRST集合
* 将该非终结符FIRST集合中除了空字以外的所有元素添加到结果FIRST集合中
* 如果该非终结符的FIRST集合包含空字，则继续扫描产生式的下一个元素；否则，结束该产生式的扫描
* 如果整个产生式扫描完成，说明产生式中的所有元素都可以推导为空字，则将空字加入结果集合中

实现算法的代码如下：







## LR(1)有限自动机的构建

1. 整体流程

按照以下流程构建LR(1)有限自动机：

* 创建有限自动机的起始节点。之前读入产生式时，自动添加了编号为0的产生式为 [起始符] -> [非终结符]，起始节点中的内容为该产生式，展望符为#（结束符）
* 使用一个队列记录操作，将起始节点入队，并重复执行以下步骤直到队列为空：
* 将队首的节点出队，扩展该结点的等价产生式，扩展方法会在后面详细描述
* 检查有限自动机中是否已经有该节点。代码中重载了相等比较运算符，判断节点相等的依据是两个节点中所有产生式的内容、当前位置和展望符都相同
* 根据有限自动机中是否存在该节点，确定该节点的编号或者为该节点分配新的节点编号。由于该节点是由另一个节点转移产生的，需要更新转移关系的目标节点编号为该节点的编号
* 如果有限自动机中不存在该节点，则将该节点加入有限自动机中，并计算该节点的转移状态：
* 建立一个映射，记录该节点接受符号转移到的状态
* 遍历该节点所有产生式，确定每一个产生式下一个位置的符号
* 如果下一个位置的符号没有对应的转移节点，则创建一个新节点并入队，该符号转移到这个新节点
* 在新节点中添加该产生式，新节点中产生式当前符号位置指向下一个位置





1. 节点产生式的扩展

节点产生式扩展的目标是把所有等价产生式加入节点，具体流程如下：

* 遍历该节点的所有产生式
* 如果产生式的下一个位置是非终结符，则引入该非终结符的所有产生式，并计算展望符
* 新产生式的当前位置设置为0
* 重复上述操作直到产生式集合不再增长





1. 根据FIRST集合计算展望符

节点中引入新的产生式时，可以按照以下规则计算展望符：

* 对于产生式 ，还未读取的符号串为，要引入的产生式为 ，则从Xm+2开始逐个检查符号Xi
* 如果Xi为终结符，则将Xi加入新产生式的展望符集合中，展望符集合计算完成
* 如果Xi为非终结符，则将FIRST(Xi)加入新产生式的展望符集合中
* 如果FIRST(Xi)包含空字，则继续检查符号Xi+1，否则展望符集合计算完成
* 如果Xm+2, …, Xn均可以推导为空字，则将原产生式的展望符加入新产生式的展望符中

## LR(1)分析表的构建

按照以下流程构建LR(1)分析表：

* 遍历有限自动机每一个转移，记该转移的信息为i状态接受字符ch转移到j状态
* 如果ch为终结符，设置ACTION[i][ch] = j
* 如果ch为非终结符，设置GOTO[i][ch] = j
* 遍历有限自动机的每一个节点
* 对于状态i节点中每一个处于结束位置的产生式，设该产生式编号为j，对该产生式的每一个展望符ch，设置REDUCE[i][ch] = j
* 由于起始节点（0状态）的第一个产生式为 [起始符] -> [非终结符]，可以保证1状态一定为ACCEPT状态，设置ACTION[1][#] = acc
* 其余位置均设置为ERROR

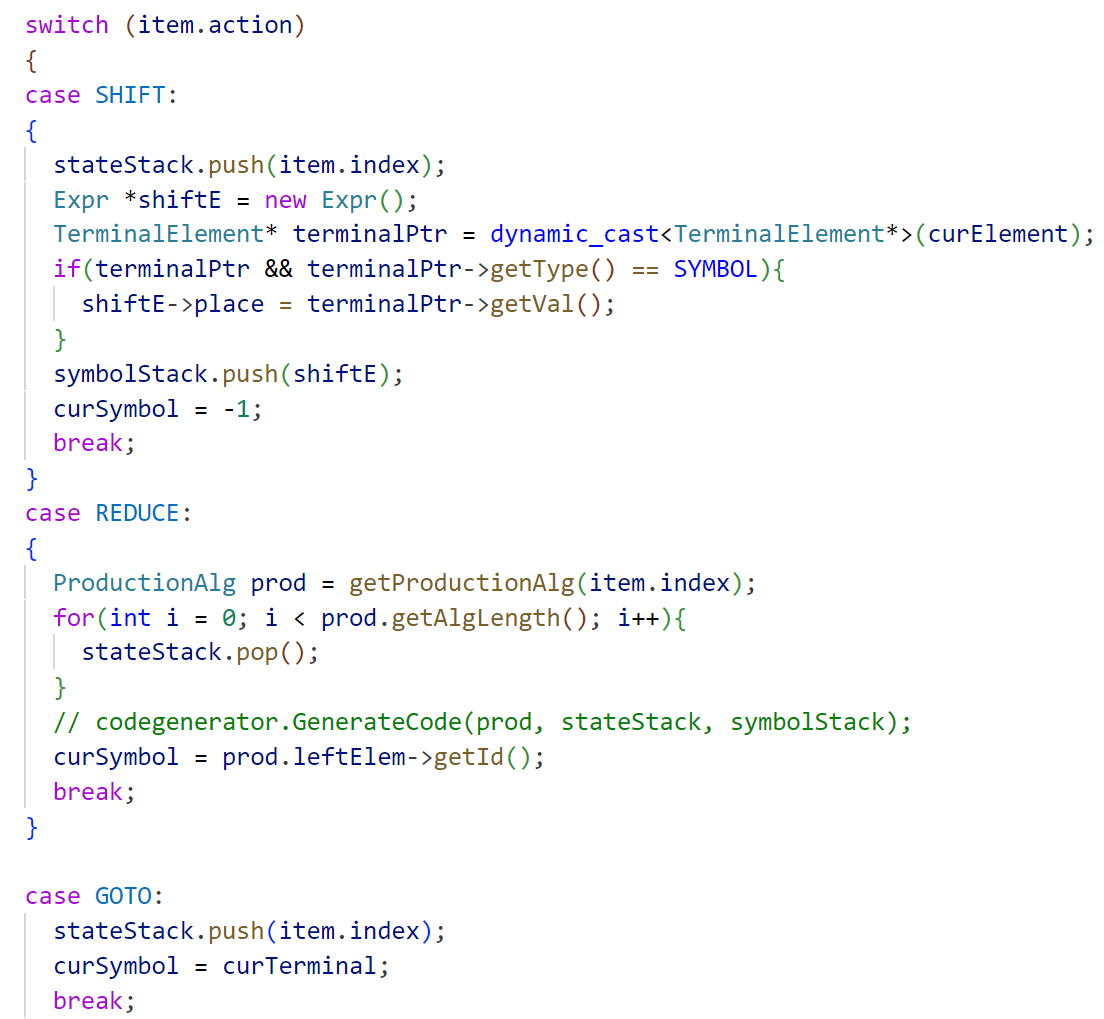


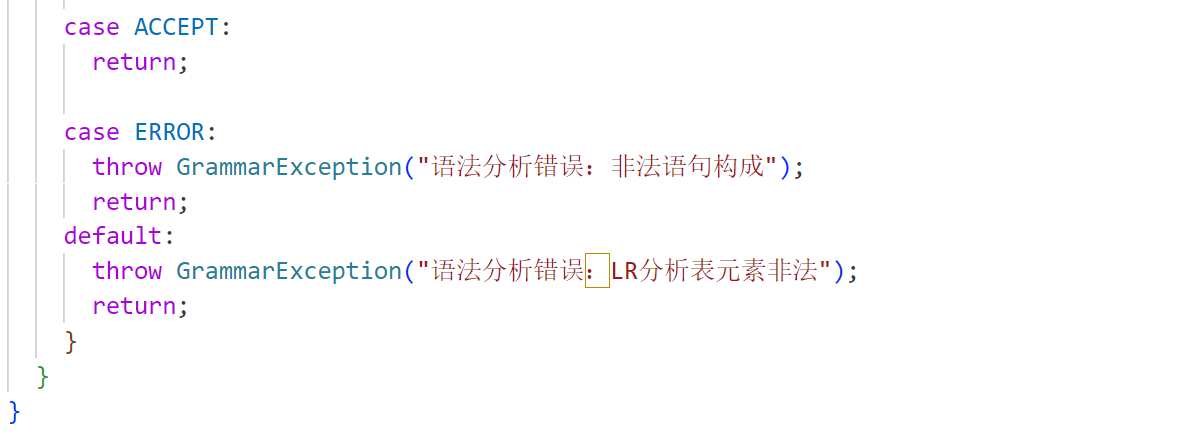
## LR(1)分析主程序

LR(1)分析主程序的逻辑如下：

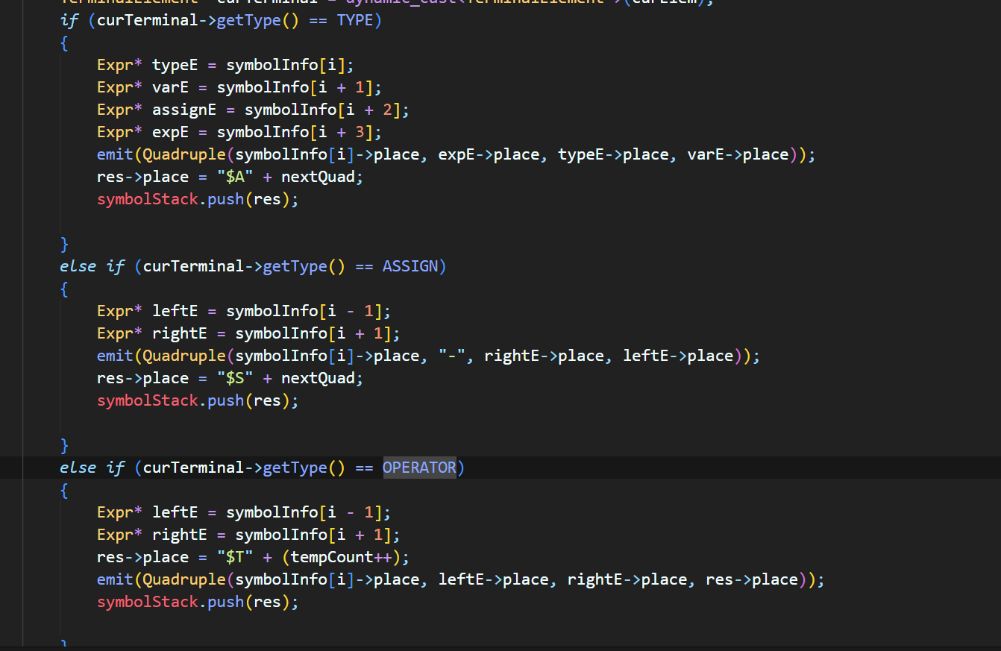
* 设置一个状态栈记录分析状态，执行以下操作：
* 每次取出栈顶状态，根据当前要接受的符号查表
* 如果查表结果为SHIFT，则调用词法分析的readNext函数读取下一个终结符作为下一个待接受的符号
* 如果查表结果为REDUCE，则使用相应产生式进行归约生成中间代码，并根据产生式的长度对状态执行出栈操作，将归约得到的非终结符作为下一个待接受的符号，并执行中间代码生成逻辑
* 如果查表结果为GOTO，则根据GOTO位置进行转移，下一个待接受的符号仍然为正在读取的终结符
* 如果查表结果为ACCEPT，则结束语法分析过程，语法分析成功
* 如果查表结果为ERROR，则抛出语法分析异常







## 中间代码生成



# **四、 系统测试**

测试用例1：

int a;

int b;

int cnt;

bool res;

if (a < 10){

cnt = 1;

}

输出结果1：

0 define int - a

1 define int - b

2 define int - cnt

3 define bool - res

4 j< a 10 $L6

5 j - - $L0

6 = 1 - cnt

由于未确定之后的语句，因此最后一个跳转语句的跳转位置未确定

测试用例2：

int a;

int b;

a = 5;

b = 3;

int c;

c = a + b;

输出结果2：

0 define int - a

1 define int - b

2 = 5 - a

3 = 3 - b

4 define int - c

5 + a b $T0

6 = $T0 - c