**基于框架的语法分析器**

**实验报告**

姓名：刘天祺

班级：07121502

学号：1320151097

日期：2018年5月23日

**目录**

[一、实验目的和内容 1](#_Toc1914419661)

[1.1 实验目的 1](#_Toc55198913)

[1.2 实验内容 1](#_Toc50384421)

[二、实现过程和步骤 1](#_Toc1678677273)

[2.1 实验环境 1](#_Toc1168006113)

[2.2 实现过程 2](#_Toc1412076198)

[2.2.1 分析方法选取 2](#_Toc876262894)

[2.2.2 准备工作 2](#_Toc1127655900)

[2.2.2.1 LL(1)文法 2](#_Toc302390982)

[2.2.2.2 属性字流 4](#_Toc1711717341)

[2.2.2.3 FIRST集合 4](#_Toc137737671)

[2.2.2.4 FOLLOW集合 5](#_Toc636256071)

[2.2.2.5 LL(1)分析栈 6](#_Toc693373605)

[2.2.3 构造LL(1)分析表 7](#_Toc486184815)

[2.2.4 LL(1)分析器总控程序 8](#_Toc983638762)

[2.2.5 输出 9](#_Toc1963815370)

[2.2.5.1 语法分析过程(xls形式) 9](#_Toc982933347)

[2.2.5.1 语法分析树(xml形式) 9](#_Toc1910633748)

[三、运行效果 10](#_Toc1025229698)

[四、实验心得体会 12](#_Toc1067119686)

[附录1：实验中使用的C语言文法 13](#_Toc119500707)

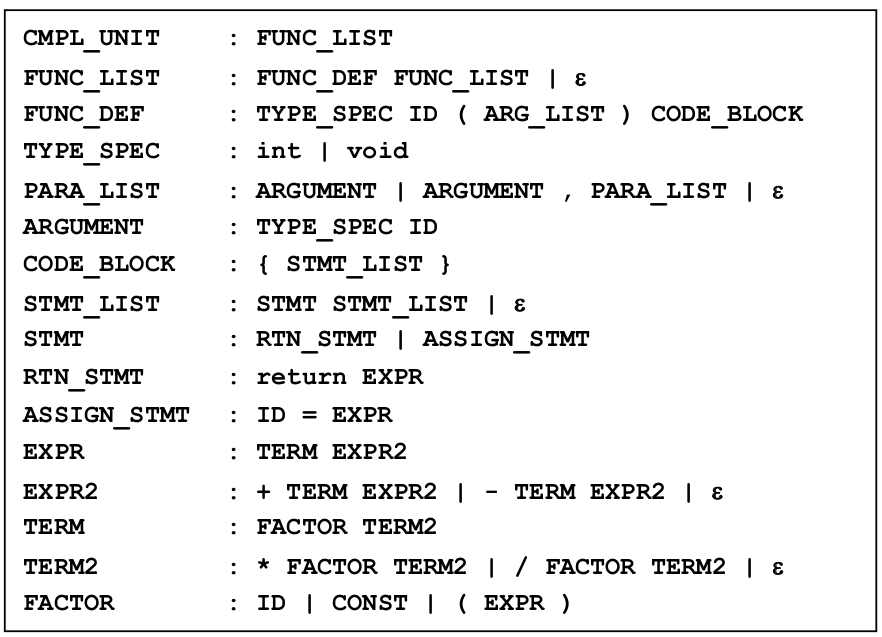
**一、实验目的和内容**

1.1 实验目的

通过实践环节深入理解与编译实现有关的形式语言理论基本概念，掌握编译程序构造的一般原理、基本设计方法和主要实现技术，并通过运用自动机理论解决实际问题，从问题定义、分析、建立数学模型和编码的整个实践活动中逐步提高软件设计开发的能力。

1.2 实验内容

该实验选择C语言的一个子集，基于BIT-MiniCC构建C语法子集的语法分析器，该语法分析器能够读入XML文件形式的属性字符流，进行语法分析并进行错误处理，如果输入正确时输出XML形式的语法树，输入不正确时报告语法错误。

如下为 C 语言文法的一个子集:  
 

在此基础之上进行文法扩充，包括全局变量声明，循环语句、分支语句、函数调用语句以及 switch 语句等。要求至少包括局部变量声明语句、赋值语句、返回语句、一种分支语句(if, if-else, switch等)和一种循环语句(for, while,do-while等)。

**二、实现过程和步骤**

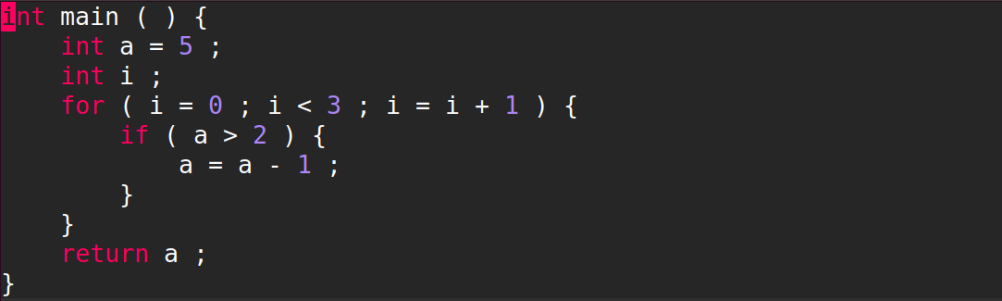
2.1 实验环境

操作系统：Ubuntu 18.04 LTS 64bit 4.15.0-21-generic

Java版本：openjdk version 10.0.1

宿主语言：Python 2.7.15 rc1

测试代码：a.c (作为词法分析器scan.py的输入，输出属性字流a.token.xml)



2.2 实现过程

2.2.1 分析方法选取

本次实验中采取**LL(1)分析法**对xml格式的属性字流进行语法分析，最终生成xml格式的语法分析树，故需要构造LL(1)分析器——parser.py。

LL(1)分析器由**总控程序**、**LL(1)分析表**和**分析栈**大三部分构成。

2.2.2 准备工作

2.2.2.1 LL(1)文法

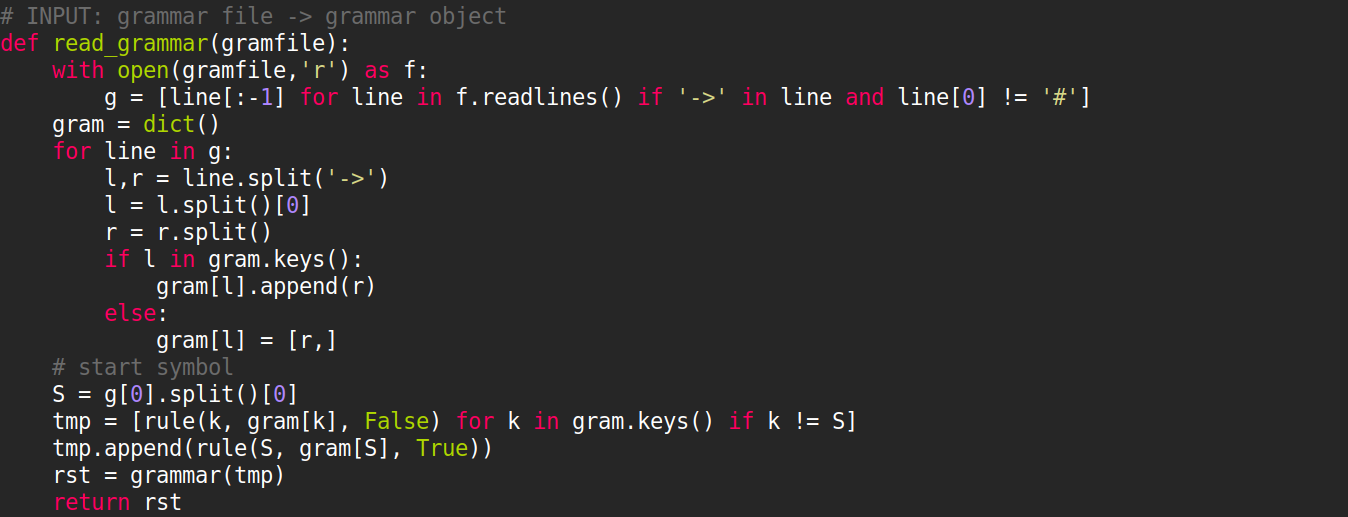
LL(1)分析器构造的关键是构造LL(1)分析表，成功构造LL(1)分析表需要文法是LL(1)文法。本次实验中分析的是C语言文法，因此在给定的C 语言文法子集的基础上进行扩充并消除文法中出现的左递归和回溯。

为了方便文法的修改与扩充，本次实验中以文本文件的形式存储文法。文法采取巴科斯-诺尔范式(BNF)描述，规定产生式的左部有且仅有一个非终结符号，对于存在多个候选式的产生式规则，将其拆分为多条，以便分析器的读入，文法文件中每行以‘#’开头的内容为注释。

本次实验中使用的C语言文法如下(节选，完整文法请参见附录1)：

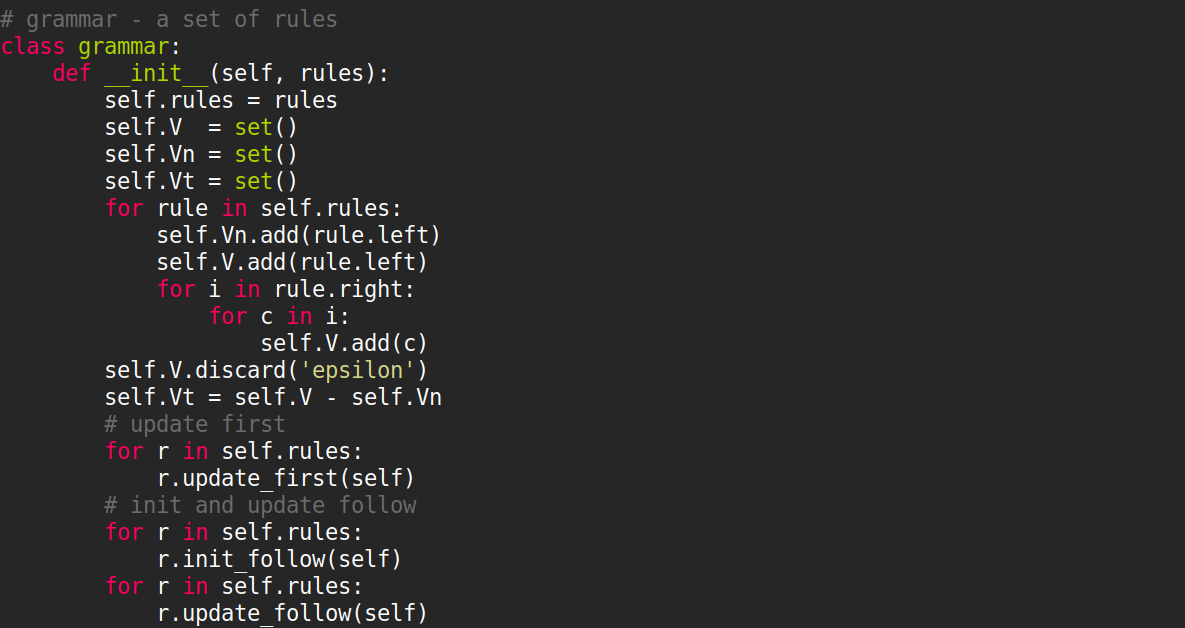
|  |
| --- |
| # 编译单元  CMPL\_UNIT -> FUNC\_LIST  # 函数列表  FUNC\_LIST -> FUNC\_DEF FUNC\_LIST  FUNC\_LIST -> epsilon  # 函数定义  FUNC\_DEF -> TYPE\_SPEC ID ( ARG\_LIST ) CODE\_BLOCK  # 类型声明  TYPE\_SPEC -> int  TYPE\_SPEC -> void  # 参数列表  ARG\_LIST -> ARGUMENT ARG\_LIST1  ARG\_LIST -> epsilon  ARG\_LIST1 -> , ARG\_LIST  ARG\_LIST1 -> epsilon  # 参数  ARGUMENT -> TYPE\_SPEC ID  # 代码块  CODE\_BLOCK -> { STMT\_LIST }  # 语句：变量声明、返回、赋值、循环、分支、比较、函数调用  STMT -> DECL\_STMT  STMT -> RTN\_STMT  STMT -> ITER\_STMT  STMT -> SLCT\_STMT  STMT -> ASSIGN\_OR\_CMP\_STMT |

LL(1)分析器通过read\_grammar函数读取文法规则，并将其中的各条产生式规则存储在一个grammar对象中：



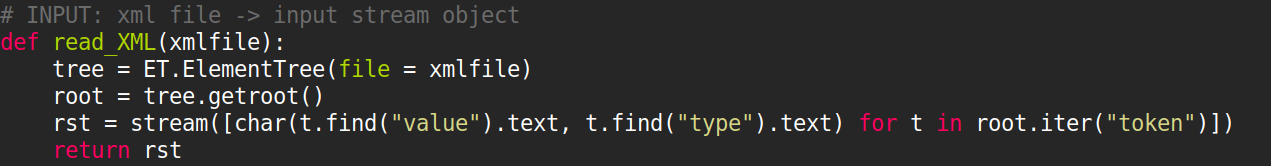
其中，每条产生式规则都为一个rule对象，其左部为一个非终结符号，右部为一张列表，存储多个候选式：

 grammar对象中包含产生式规则对象的列表，并由此可计算出文法的符号集V、终结符号即VT以及非终结符号集VN：

注：rule与grammar对象在初始化过程中还涉及与FIRST集、FOLLOW集有关的操作， 这些内容将在2.2.2.3与2.2.2.4中小节详细说明。

2.2.2.2 属性字流

LL(1)分析器通过read\_XML函数，对xml格式属性字流中的每个单词的**类型**与**值**进行提取，将之转存于一个属性字流对象stream中，以便于分析器使用。



stream对象中包含一个列表r和一个符号表d。r记录了每个单词的类型与值，即输入的符号串；d记录了所有单词值对应的类型（如标识符、常量），以便于语法树生成；move方法用于移动符号串指针，p方法用于返回符号串指针指向的单词值。



2.2.2.3 FIRST集合

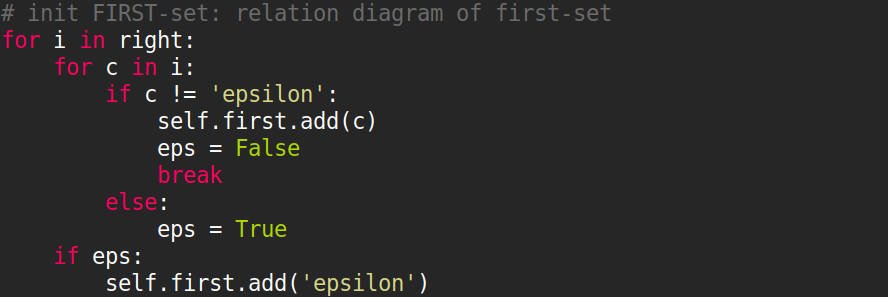
检测文法是否为LL(1)文法（是否含有回溯）以及构造LL(1)分析表的过程中需要用到FIRST集。对于任一终结符号a而言，FIRST(a)={a}，因此只需考虑非终结符号的FIRST集合的构建。

对于一个文法而言，其产生式规则集中的每条产生式规则与非终结符号集VN中的每个非终结符号一一对应，因此可以将FIRST集合存储于产生式规则对象rule中。

由于计算一个非终结符号的FIRST集时可能会需要另一个非终结符号的FIRST集，因此在产生式规则对象初始化的时候不能完成FIRST集合的计算工作，需要等到所有产生式规则对象初始化完毕并存入grammar对象以后能够开始相应FIRST集合的计算工作。

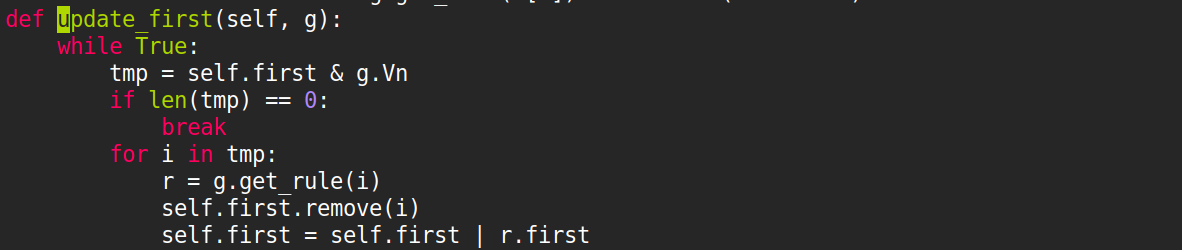
为了解决上述问题，本次实验中采取**关系图法**来求解每个非终结符的FIRST集合：

1. 初始化FIRST集合关系图  
    对每条产生式而言，将其右部候选式的**首个字符**加入到其左部所对应的非终结符的 FIRST集合中，若候选式为空串，则将epsilon加入到FIRST集合中：



注：该步骤在每个rule对象初始化时完成

1. 更新FIRST集合关系图  
    对每条产生式i而言，如果其FIRST集合中含有非终结符号，则将该非终结符号 对应的产生式j的FIRST集合中的所有元素加入i的FIRST集合，并将该非终结 符号从i的FIRST集合中移除。重复上述操作，直至其FIRST集合中不含非终结 符。



注：更新关系图需要由文法配合，所以更新在文法对象初始化时完成，如下图：



2.2.2.4 FOLLOW集合

构造LL(1)分析表的过程中，如果某产生式P->γ，且γ=>ε，则需要用到P的FOLLOW集来构造预测函数。每一个非终结符号都对应一个FOLLOW集合，因此将FOLLOW集合存储于产生式规则对象rule中。

由于计算一个非终结符号的FOLLOW集时可能会需要另一个非终结符号的FOLLOW集或是FIRST集，因此在产生式规则对象初始化的时候不能完成FOLLOW集合的计算工作，需要等到所有产生式规则对象初始化完毕并存入grammar对象以后能够开始相应FOLLOW集合的计算工作。

为了解决上述问题，本次实验中采取**关系图法**来求解每个非终结符的FOLLOW集合：

1. 初始化FOLLOW集合关系图

对grammar对象中的每一个rule对象中的每一条产生式P->X1X2…Xk：

当Xi∈VN时(1<=i<=k)时：

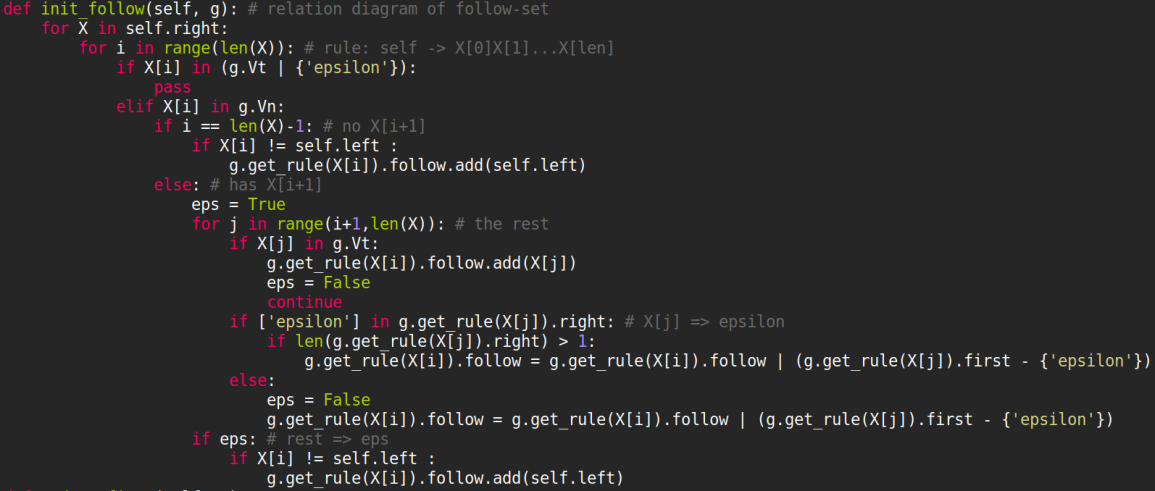
如果Xi+1∈VT，则将Xi+1加入Xi的FOLLOW集合；

如果Xi+1∈VN，则将Xi+1的FIRST集中的所有元素加入Xi的FOLLOW集合；

若Xi+1Xi+2…Xi+j\*=>ε且Xi+j+1∈VT，则将Xi+j+1加入Xi+j+1的FOLLOW集合；

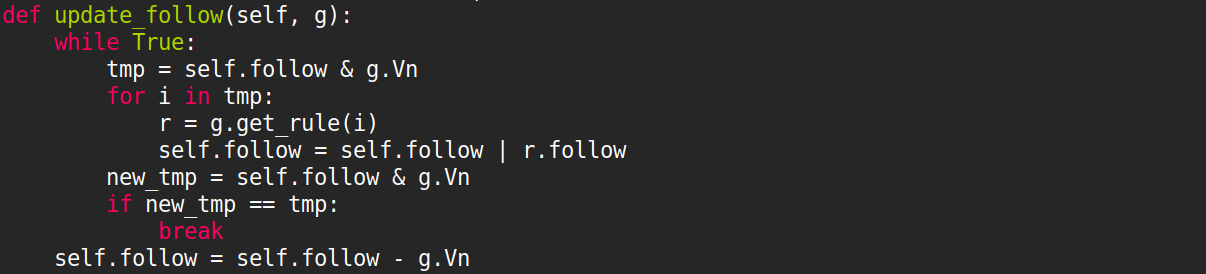
若Xi+1Xi+2…Xi+j\*=>ε且Xi+j+1∈VN，则将Xi+j+1的FIRST集中的所有元素  
 加入Xi的FOLLOW集合；

若Xi+1Xi+2…Xk\*=>ε且Xi+j+1∈VT，则将P的FOLLOW集中的所有元素  
 加入Xi的FOLLOW集合；



1. 更新FOLLOW集合关系图

对每条产生式i而言，如果其FOLLOW集合中含有非终结符号，则将该非终结符号对应的产生式j的FOLLOW集合中的所有元素加入i的FOLLOW集合。重复上述操作，直至没有新的元素可添加进其FOLLOW集合中。移除所有FOLLOW集合中的非终结符号，所得的集合即为该产生式左部非终结符号的FOLLOW集合。



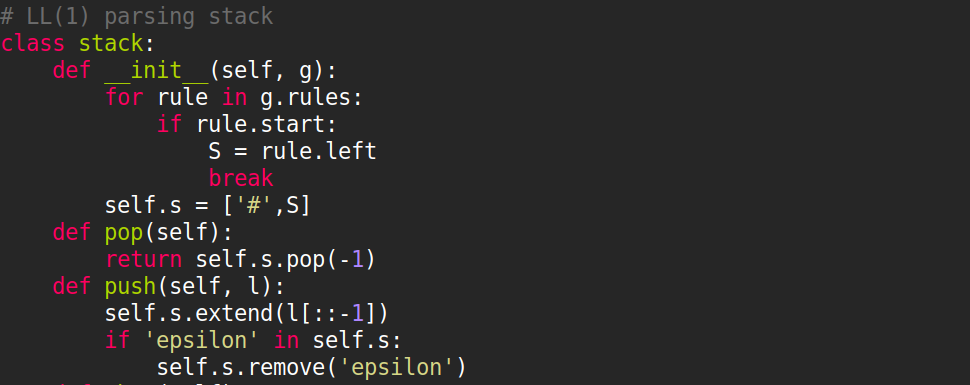
注：初始化和更新FOLLOW集合关系图均依赖于文法，因此在文法对象初始化时，

分别执行FOLLOW集合关系图的初始化与更新操作，如下图所示：



2.2.2.5 LL(1)分析栈

LL(1)分析栈用于临时存放语法分析过程中的文法符号，分析栈在初始化时将'#'与文法的开始符号S依次压入栈：



2.2.3 构造LL(1)分析表

构造LL(1)分析表的关键在于构造预测函数。**预测函数**M(P,a)根据分析栈栈顶的非终结符P和输入字符串扫描指针所指向的终结符a来选取候选式。

LL(1)分析表的构造方法如下：

|  |
| --- |
| 对文法中的每一条产生式P->γ1|γ2|…|γn：  如果a∈FIRST(γi)，则M(P,a) = P->γi ;  如果ε∈FIRST(γi)，则对于FOLLOW(P)中的终结符bj，M(P,bj) = P->γi ； |

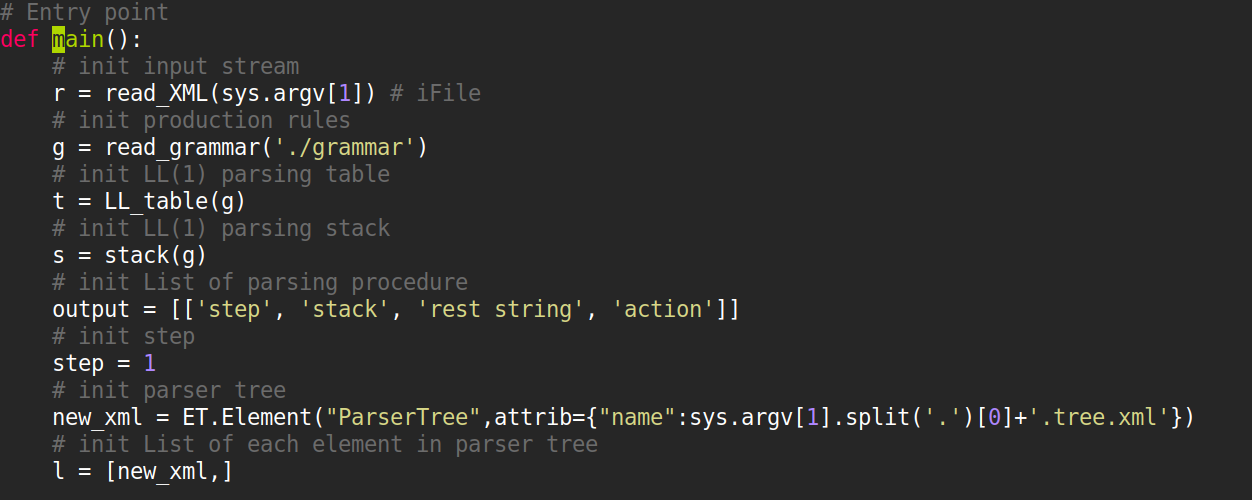
LL(1)分析表对象提供一个查询接口——query函数，LL(1)分析器可以通过调用query函数来查询LL(1)分析表，得到M(P,a)对应的候选式。

LL(1)分析表对象如下图所示：



2.2.4 LL(1)分析器总控程序

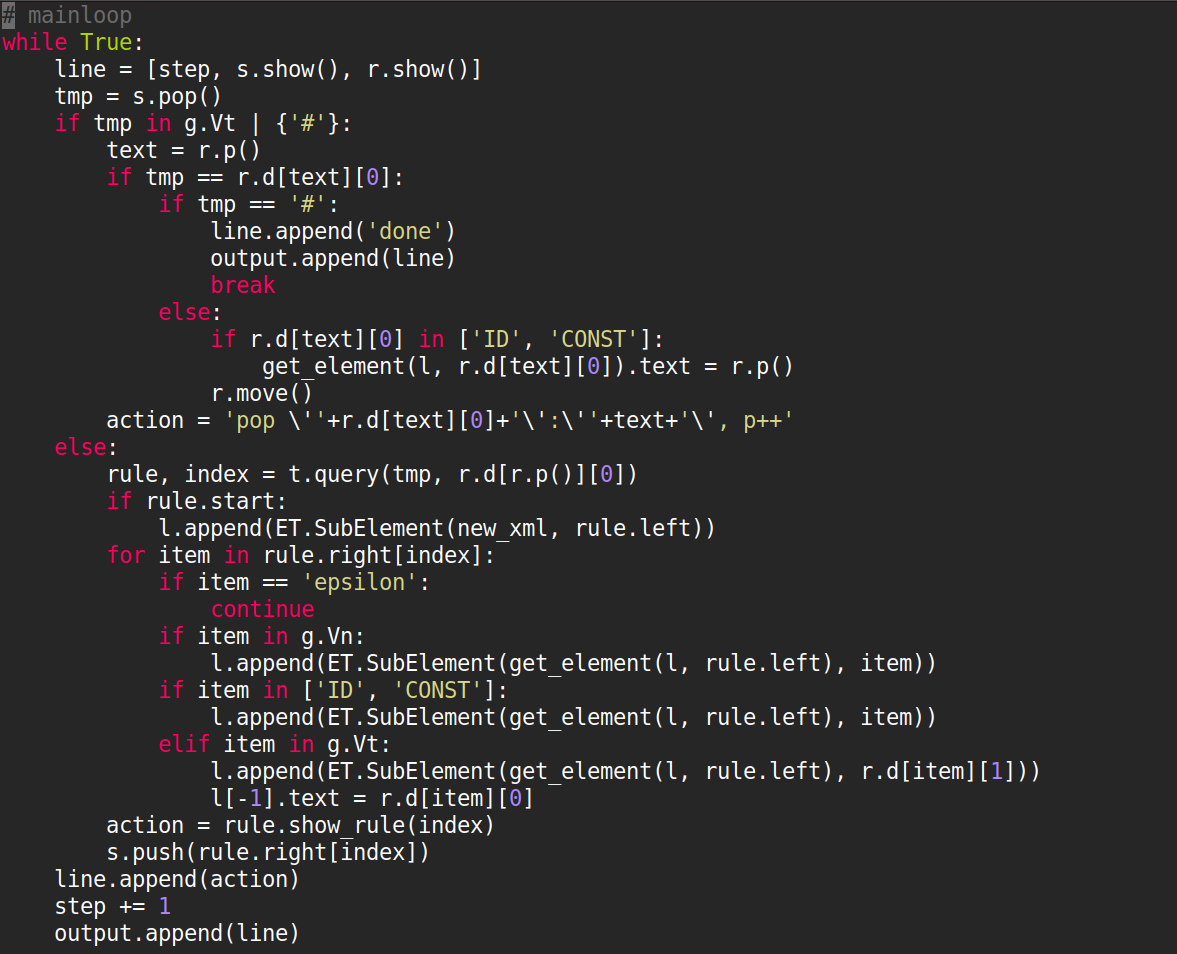
LL(1)分析器首先完成初始化工作，包括读取属性字流、读取文法、构造LL(1)分析表、初始化LL(1)分析栈、初始化分析过程列表(输出)、初始化语法分析树(输出)。



初始化完成后，LL(1)分析器开始进行语法分析：

|  |
| --- |
| 设X为栈顶符号，a为扫描指针指向的终结符：  若X∈VT，则：  若X=a='#'，则表示分析成功，停止分析过程；  若X=a≠'#'，则X出栈，移动扫描指针；  若X≠a，则表示分析出错；  若X∈VN，则查LL(1)分析表，若无错，则将查询到的候选式中的文法符号倒序入栈。  重复执行上述操作。 |

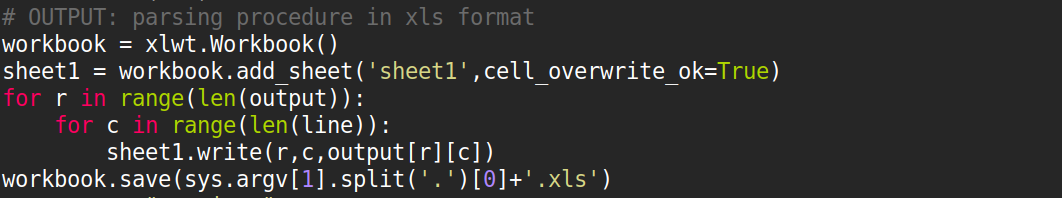
总控程序的代码如下图所示：



2.2.5 输出

2.2.5.1 语法分析过程(xls形式)

在2.2.4节的语法分析过程中，将每一步的分析结果保存在分析过程列表中，语法分析结束后，使用xlwt模块将列表中的语法分析过程输出到xls文件中，代码如下：

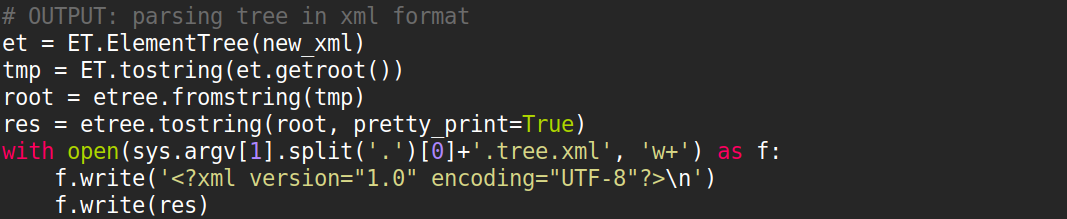


2.2.5.1 语法分析树(xml形式)

在2.2.4节的语法分析过程中，逐步建立语法分析树，建树的规则如下：

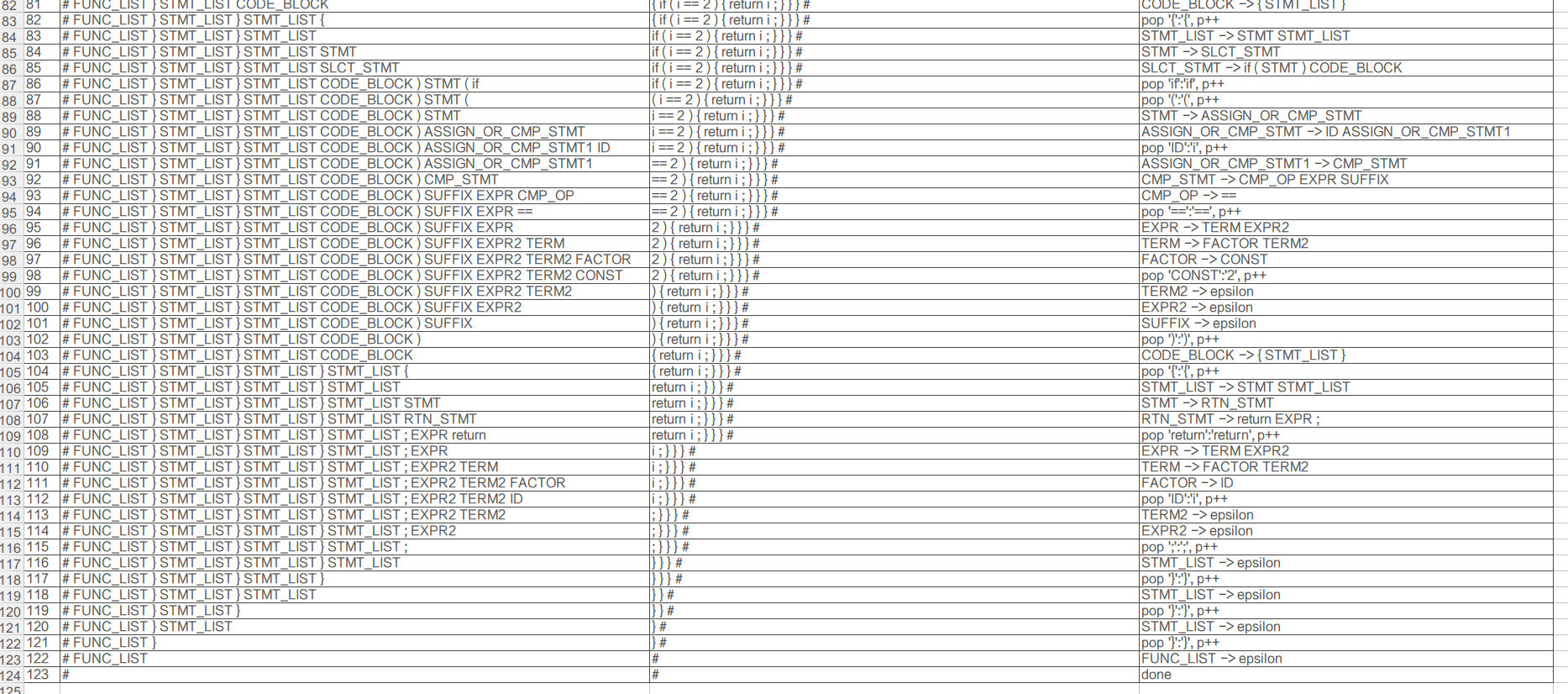
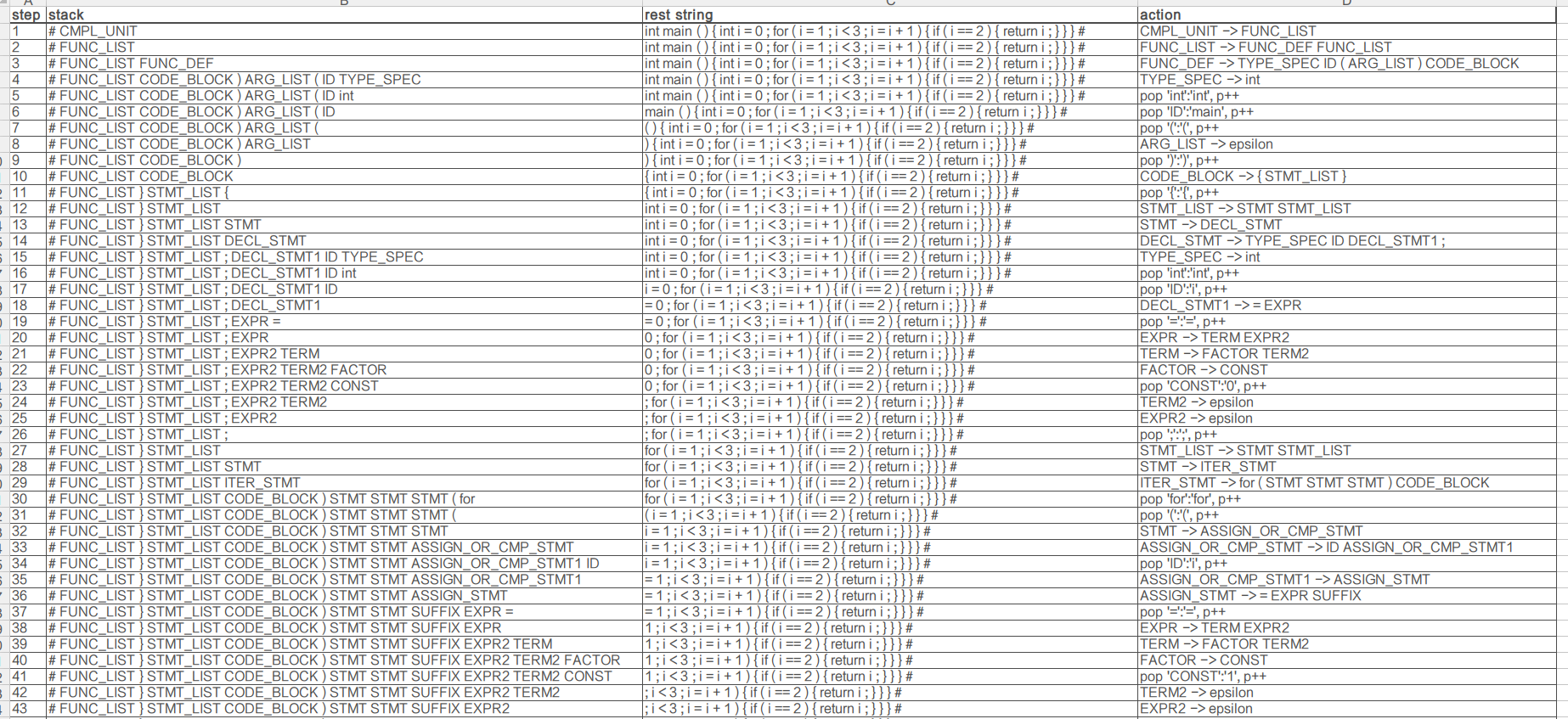
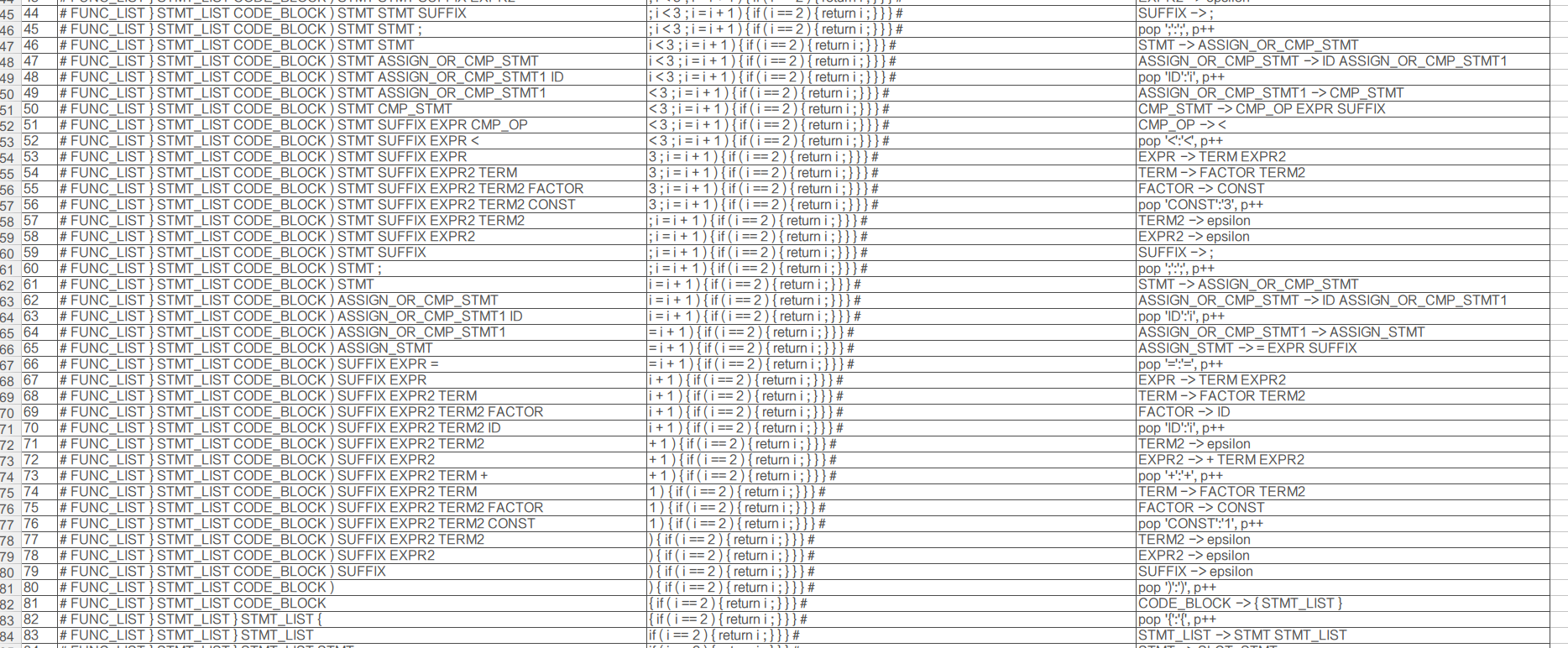
|  |
| --- |
| 1. 语法分析树的根节点为文法的开始符号。 2. 查询LL(1)分析表后，为候选式中的每个文法符号建立一个节点，其父亲   节点为产生式左部终结符号对应的节点。   1. 所有子树的xml标签为其类型，若其类型为标识符或是常量，则查询属  性字流中的符号表，将标识符或常量的值写入节点text字段。 |

语法分析结束后，将总控程序中构建好的语法分析树输出到xml文件中，代码如下：



**三、运行效果**

对输入的属性字流a.token.xml的语法分析流程如下表所示：



对输入的属性字流a.token.xml进行语法分析，得到的语法树如下图所示(节选)：



**四、实验心得体会**

通过本此实验熟悉掌握了LL(1)分析法，通过代码基本实现了一个LL(1)分析器，但是还有很多的不足，比如：1.错误处理方面有所欠缺：仅在查LL(1)分析表查不到候选式时会报错。2.对于给定的文法限制过于严格，使用时必须提供一个LL(1)文法，对给定任意BNF描述的文法，自动消除左递归和回溯的功能没有实现。3.由于第2点，没能写出更全面一点的C语言文法，仅实现了实验的最低要求。4.在计算FIRST集合的时候，没有考虑计算字符串的FIRST集合的这种情况，为后续计算FOLLOW集合以及LL(1)分析表构建增加了麻烦。

附录1：实验中使用的C语言文法

# 编译单元

CMPL\_UNIT -> FUNC\_LIST

# 函数列表

FUNC\_LIST -> FUNC\_DEF FUNC\_LIST

FUNC\_LIST -> epsilon

# 函数定义

FUNC\_DEF -> TYPE\_SPEC ID ( ARG\_LIST ) CODE\_BLOCK

# 类型声明

TYPE\_SPEC -> int

TYPE\_SPEC -> void

TYPE\_SPEC -> char

TYPE\_SPEC -> float

TYPE\_SPEC -> double

# 参数列表

ARG\_LIST -> ARGUMENT ARG\_LIST1

ARG\_LIST -> epsilon

ARG\_LIST1 -> , ARG\_LIST

ARG\_LIST1 -> epsilon

# 参数

ARGUMENT -> TYPE\_SPEC ID

# 代码块

CODE\_BLOCK -> { STMT\_LIST }

# 语句列表

STMT\_LIST -> STMT STMT\_LIST

STMT\_LIST -> epsilon

# 语句：变量声明、返回、赋值、循环、分支、比较、函数调用

STMT -> DECL\_STMT

STMT -> RTN\_STMT

STMT -> ITER\_STMT

STMT -> SLCT\_STMT

STMT -> ASSIGN\_OR\_CMP\_STMT

# STMT -> CALL\_STMT

# 变量声明语句

DECL\_STMT -> TYPE\_SPEC ID DECL\_STMT1 ;

DECL\_STMT1 -> = EXPR

DECL\_STMT1 -> epsilon

# 返回语句

RTN\_STMT -> return EXPR ;

# 消除赋值与比较语句的间接回溯

ASSIGN\_OR\_CMP\_STMT -> ID ASSIGN\_OR\_CMP\_STMT1

ASSIGN\_OR\_CMP\_STMT1 -> ASSIGN\_STMT

ASSIGN\_OR\_CMP\_STMT1 -> CMP\_STMT

# 赋值语句

ASSIGN\_STMT -> = EXPR SUFFIX

# 比较语句

CMP\_STMT -> CMP\_OP EXPR SUFFIX

CMP\_OP -> >

CMP\_OP -> <

CMP\_OP -> >=

CMP\_OP -> <=

CMP\_OP -> ==

CMP\_OP -> !=

# 后缀

SUFFIX -> ;

SUFFIX -> epsilon

# 循环语句 for, while 待修改

ITER\_STMT -> for ( STMT STMT STMT ) CODE\_BLOCK

# ITER\_STMT -> for ( STMT ; STMT ; STMT ) STMT

# ITER\_STMT -> while ( STMT ) CODE\_BLOCK

# ITER\_STMT -> while ( STMT ) STMT

# ITER\_STMT -> do STMT while ( STMT )

# ITER\_STMT -> do CODE\_BLOCK while ( STMT )

# 分支语句 if-else 待修改

SLCT\_STMT -> if ( STMT ) CODE\_BLOCK

# SLCT\_STMT -> if ( STMT ) STMT

# SLCT\_STMT -> if ( STMT ) STMT else CODE\_BLOCK

# SLCT\_STMT -> if ( STMT ) STMT else STMT

# SLCT\_STMT -> if ( STMT ) CODE\_BLOCK else CODE\_BLOCK

# SLCT\_STMT -> if ( STMT ) CODE\_BLOCK else STMT

# 调用语句

# CALL\_STMT -> ID ( ARG\_LIST )

# 表达式

EXPR -> TERM EXPR2

EXPR2 -> + TERM EXPR2

EXPR2 -> - TERM EXPR2

EXPR2 -> epsilon

TERM -> FACTOR TERM2

TERM2 -> \* FACTOR TERM2

TERM2 -> / FACTOR TERM2

TERM2 -> epsilon

FACTOR -> ID

FACTOR -> CONST

FACTOR -> ( EXPR )