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

1.1 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:

```
CMPL_UNIT     : FUNC_LIST
FUNC_LIST     : FUNC_DEF FUNC_LIST | ε
FUNC_DEF      : TYPE_SPEC ID ( ARG_LIST ) CODE_BLOCK
TYPE_SPEC     : int | void
PARA_LIST     : ARGUMENT | ARGUMENT , PARA_LIST | ε
ARGUMENT      : TYPE_SPEC ID
CODE_BLOCK    : { STMT_LIST }
STMT_LIST     : STMT STMT_LIST | ε
STMT          : RTN_STMT | ASSIGN_STMT
RTN_STMT      : return EXPR
ASSIGN_STMT   : ID = EXPR
EXPR          : TERM EXPR2
EXPR2         : + TERM EXPR2 | - TERM EXPR2 | ε
TERM          : FACTOR TERM2
TERM2         : * FACTOR TERM2 | / FACTOR TERM2 | ε
FACTOR        : ID | CONST | ( EXPR )
```

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)

```
int main ( ) {
    int a = 5 ;
    int i ;
    for ( i = 0 ; i < 3 ; i = i + 1 ) {
        if ( a > 2 ) {
            a = a - 1 ;
        }
    }
    return a ;
}
```

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：

```
# INPUT: grammar file -> grammar object
def read_grammar(gramfile):
    with open(gramfile,'r') as f:
        g = [line[:-1] for line in f.readlines() if '->' in line and line[0] != '#']
    gram = dict()
    for line in g:
        l,r = line.split('->')
        l = l.split()[0]
        r = r.split()
        if l in gram.keys():
            gram[l].append(r)
        else:
            gram[l] = [r,]
    # start symbol
    S = g[0].split()[0]
    tmp = [rule(k, gram[k], False) for k in gram.keys() if k != S]
    tmp.append(rule(S, gram[S], True))
    rst = grammar(tmp)
    return rst
```

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：

```
# production rule
class rule:
    def __init__(self, left, right, start):
        self.start = start
        self.left = left
        self.right = right
        self.first = set()
        # init FIRST-set: relation diagram of first-set
        for i in right:
            for c in i:
                if c != 'epsilon':
                    self.first.add(c)
                    eps = False
                    break
                else:
                    eps = True
            if eps:
                self.first.add('epsilon')
        # init FOLLOW-set
        self.follow = set()
        if self.start:
            self.follow.add('#')
```

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 V_T 以及非终结符号集 V_N：

```
# grammar - a set of rules
class grammar:
    def __init__(self, rules):
        self.rules = rules
        self.V  = set()
        self.Vn = set()
        self.Vt = set()
        for rule in self.rules:
            self.Vn.add(rule.left)
            self.V.add(rule.left)
            for i in rule.right:
                for c in i:
                    self.V.add(c)
        self.V.discard('epsilon')
        self.Vt = self.V - self.Vn
        # update first
        for r in self.rules:
            r.update_first(self)
        # init and update follow
        for r in self.rules:
            r.init_follow(self)
```

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。

```
# INPUT: xml file -> input stream object
def read_XML(xmlfile):
    tree = ET.ElementTree(file = xmlfile)
    root = tree.getroot()
    rst = stream([char(t.find("value").text, t.find("type").text) for t in root.iter("token")])
    return rst
```

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。

```
# input stream
class stream:
    def __init__(self, r):
        self.r = r
        self.d = {'epsilon':('eps','epsilon')} # namelist
        for i in r:
            self.d[i.value] = (i.text, i.type)
    def show(self):
        rst = ''
        for c in self.r:
            rst += c.value + ' '
        return rst
    def move(self):
        self.r.pop(0)
    def p(self):
        return self.r[0].value
```

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 V_N 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：

```
# init FIRST-set: relation diagram of first-set
for i in right:
    for c in i:
        if c != 'epsilon':
            self.first.add(c)
            eps = False
            break
        else:
            eps = True
    if eps:
        self.first.add('epsilon')
```

注：该步骤在每个 rule 对象初始化时完成

（2）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。

```
def update_first(self, g):
    while True:
        tmp = self.first & g.Vn
        if len(tmp) == 0:
            break
        for i in tmp:
            r = g.get_rule(i)
            self.first.remove(i)
            self.first = self.first | r.first
```

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：

```
# update first
for r in self.rules:
    r.update_first(self)
```

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 $P->X_1X_2...X_k$：

当 $X_i \in V_N$ 时(1<=i<=k)时：

如果 $X_{i+1} \in V_T$，则将 X_{i+1} 加入 X_i 的 FOLLOW 集合；

如果 $X_{i+1} \in V_N$，则将 X_{i+1} 的 FIRST 集中的所有元素加入 X_i 的 FOLLOW 集合；

若 $X_{i+1}X_{i+2}...X_{i+j}*=> \varepsilon$ 且 $X_{i+j+1} \in V_T$，则将 X_{i+j+1} 加入 X_{i+j+1} 的 FOLLOW 集合；

若 $X_{i+1}X_{i+2}...X_{i+j}*=> \varepsilon$ 且 $X_{i+j+1} \in V_N$，则将 X_{i+j+1} 的 FIRST 集中的所有元素加入 X_i 的 FOLLOW 集合；

若 $X_{i+1}X_{i+2}...X_k*=> \varepsilon$ 且 $X_{i+j+1} \in V_T$，则将 P 的 FOLLOW 集中的所有元素加入 X_i 的 FOLLOW 集合；

```
def init_follow(self, g): # relation diagram of follow-set
    for X in self.right:
        for i in range(len(X)): # rule: self -> X[0]X[1]...X[len]
            if X[i] in (g.Vt | {'epsilon'}):
                pass
            elif X[i] in g.Vn:
                if i == len(X)-1: # no X[i+1]
                    if X[i] != self.left :
                        g.get_rule(X[i]).follow.add(self.left)
                else: # has X[i+1]
                    eps = True
                    for j in range(i+1,len(X)): # the rest
                        if X[j] in g.Vt:
                            g.get_rule(X[i]).follow.add(X[j])
                            eps = False
                            continue
                        if ['epsilon'] in g.get_rule(X[j]).right: # X[j] => epsilon
                            if len(g.get_rule(X[j]).right) > 1:
                                g.get_rule(X[i]).follow = g.get_rule(X[i]).follow | (g.get_rule(X[j]).first - {'epsilon'})
                        else:
                            eps = False
                            g.get_rule(X[i]).follow = g.get_rule(X[i]).follow | (g.get_rule(X[j]).first - {'epsilon'})
                    if eps: # rest => eps
                        if X[i] != self.left :
                            g.get_rule(X[i]).follow.add(self.left)
```

（2）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。

```
def update_follow(self, g):
    while True:
        tmp = self.follow & g.Vn
        for i in tmp:
            r = g.get_rule(i)
            self.follow = self.follow | r.follow
        new_tmp = self.follow & g.Vn
        if new_tmp == tmp:
            break
    self.follow = self.follow - g.Vn
```

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：

```
# init and update follow
for r in self.rules:
    r.init_follow(self)
for r in self.rules:
    r.update_follow(self)
```

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：

```
# LL(1) parsing stack
class stack:
    def __init__(self, g):
        for rule in g.rules:
            if rule.start:
                S = rule.left
                break
        self.s = ['#',S]
    def pop(self):
        return self.s.pop(-1)
    def push(self, l):
        self.s.extend(l[::-1])
        if 'epsilon' in self.s:
            self.s.remove('epsilon')
```

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-> $\gamma_1 | \gamma_2 | \cdots | \gamma_n$：
>
> 如果 a∈FIRST(γ_i)，则 M(P,a) = P-> γ_i ;
>
> 如果 ε ∈FIRST(γ_i)，则对于 FOLLOW(P)中的终结符 b_j，M(P,b_j) = P-> γ_i ；

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：

```
# create LL(1) parsing table
class LL_table:
    def __init__(self, g):
        self.table = dict()
        for rule in g.rules:
            for P in rule.right:
                for X in P: # P -> X0X1...Xn
                    if X in g.Vt:
                        self.table[(rule.left,X)] = (rule, rule.right.index(P))
                        break
                    elif X in g.Vn:
                        no_eps = True
                        for a in g.get_rule(X).first:
                            if a == 'epsilon':
                                no_eps = False
                            else:
                                self.table[(rule.left,a)] = (rule, rule.right.index(P))
                        if no_eps:
                            break
                for X in P: # P -> X0X1...Xn
                    if X in g.Vt:
                        break
                    elif X in g.Vn:
                        if 'epsilon' not in g.get_rule(X).first:
                            break # if eps not all Xi.first, then eps not in X0X1...Xn.first
                    for b in rule.follow:
                        self.table[(rule.left,b)] = (rule, rule.right.index(P))
    def query(self, Vn, Vt):
        try:
            rst = self.table[(Vn,Vt)]
        except KeyError, err:
            error(err)
        else:
            return rst
```

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)。

```
# Entry point
def main():
    # init input stream
    r = read_XML(sys.argv[1]) # iFile
    # init production rules
    g = read_grammar('./grammar')
    # init LL(1) parsing table
    t = LL_table(g)
    # init LL(1) parsing stack
    s = stack(g)
    # init List of parsing procedure
    output = [['step', 'stack', 'rest string', 'action']]
    # init step
    step = 1
    # init parser tree
    new_xml = ET.Element("ParserTree",attrib={"name":sys.argv[1].split('.')[0]+'.tree.xml'})
    # init List of each element in parser tree
    l = [new_xml,]
```

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 $X \in V_T$，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 $X \in V_N$，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：

```
# mainloop
while True:
    line = [step, s.show(), r.show()]
    tmp = s.pop()
    if tmp in g.Vt | {'#'}:
        text = r.p()
        if tmp == r.d[text][0]:
            if tmp == '#':
                line.append('done')
                output.append(line)
                break
            else:
                if r.d[text][0] in ['ID', 'CONST']:
                    get_element(l, r.d[text][0]).text = r.p()
                r.move()
        action = 'pop \''+r.d[text][0]+'\':\''+text+'\', p++'
    else:
        rule, index = t.query(tmp, r.d[r.p()][0])
        if rule.start:
            l.append(ET.SubElement(new_xml, rule.left))
        for item in rule.right[index]:
            if item == 'epsilon':
                continue
            if item in g.Vn:
                l.append(ET.SubElement(get_element(l, rule.left), item))
            if item in ['ID', 'CONST']:
                l.append(ET.SubElement(get_element(l, rule.left), item))
            elif item in g.Vt:
                l.append(ET.SubElement(get_element(l, rule.left), r.d[item][1]))
                l[-1].text = r.d[item][0]
        action = rule.show_rule(index)
        s.push(rule.right[index])
    line.append(action)
    step += 1
    output.append(line)
```

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：

```
# OUTPUT: parsing procedure in xls format
workbook = xlwt.Workbook()
sheet1 = workbook.add_sheet('sheet1',cell_overwrite_ok=True)
for r in range(len(output)):
    for c in range(len(line)):
        sheet1.write(r,c,output[r][c])
workbook.save(sys.argv[1].split('.')[0]+'.xls')
```

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。

（3）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：

```
# OUTPUT: parsing tree in xml format
et = ET.ElementTree(new_xml)
tmp = ET.tostring(et.getroot())
root = etree.fromstring(tmp)
res = etree.tostring(root, pretty_print=True)
with open(sys.argv[1].split('.')[0]+'.tree.xml', 'w+') as f:
    f.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    f.write(res)
```

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：

step	stack	rest string	action
1	# CMPL_UNIT	int main ( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	CMPL_UNIT -> FUNC_LIST
2	# FUNC_LIST	int main ( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	FUNC_LIST -> FUNC_DEF FUNC_LIST
3	# FUNC_LIST FUNC_DEF	int main ( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	FUNC_DEF -> TYPE_SPEC ID (ARG_LIST) CODE_BLOCK
4	# FUNC_LIST CODE_BLOCK) ARG_LIST (ID TYPE_SPEC	int main ( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TYPE_SPEC -> int
5	# FUNC_LIST CODE_BLOCK) ARG_LIST (ID int	int main ( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'int':'int', p++
6	# FUNC_LIST CODE_BLOCK) ARG_LIST (ID	main ( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'ID':'main', p++
7	# FUNC_LIST CODE_BLOCK) ARG_LIST (	( ) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop '(':'(', p++
8	# FUNC_LIST CODE_BLOCK) ARG_LIST	) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ARG_LIST -> epsilon
9	# FUNC_LIST CODE_BLOCK)	) { int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop ')':')', p++
10	# FUNC_LIST CODE_BLOCK	{ int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	CODE_BLOCK -> { STMT_LIST }
11	# FUNC_LIST) STMT_LIST {	{ int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop '{':'{', p++
12	# FUNC_LIST) STMT_LIST	int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	STMT_LIST -> STMT STMT_LIST
13	# FUNC_LIST) STMT_LIST STMT	int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	STMT -> DECL_STMT
14	# FUNC_LIST) STMT_LIST DECL_STMT	int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	DECL_STMT -> TYPE_SPEC ID DECL_STMT1 ;
15	# FUNC_LIST) STMT_LIST ; DECL_STMT1 ID TYPE_SPEC	int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TYPE_SPEC -> int
16	# FUNC_LIST) STMT_LIST ; DECL_STMT1 ID int	int i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'int':'int', p++
17	# FUNC_LIST) STMT_LIST ; DECL_STMT1 ID	i = 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'ID':'i', p++
18	# FUNC_LIST) STMT_LIST ; DECL_STMT1	= 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	DECL_STMT1 -> = EXPR
19	# FUNC_LIST) STMT_LIST ; EXPR =	= 0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop '=':'=', p++
20	# FUNC_LIST) STMT_LIST ; EXPR	0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	EXPR -> TERM EXPR2
21	# FUNC_LIST) STMT_LIST ; EXPR2 TERM	0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TERM -> FACTOR TERM2
22	# FUNC_LIST) STMT_LIST ; EXPR2 TERM2 FACTOR	0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	FACTOR -> CONST
23	# FUNC_LIST) STMT_LIST ; EXPR2 TERM2 CONST	0 ; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'CONST':'0', p++
24	# FUNC_LIST) STMT_LIST ; EXPR2 TERM2	; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TERM2 -> epsilon
25	# FUNC_LIST) STMT_LIST ; EXPR2	; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	EXPR2 -> epsilon
26	# FUNC_LIST) STMT_LIST ;	; for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop ';':';', p++
27	# FUNC_LIST) STMT_LIST	for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	STMT_LIST -> STMT STMT_LIST
28	# FUNC_LIST) STMT_LIST STMT	for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	STMT -> ITER_STMT
29	# FUNC_LIST) STMT_LIST ITER_STMT	for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ITER_STMT -> for (STMT STMT STMT) CODE_BLOCK
30	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT STMT (for	for (i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'for':'for', p++
31	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT STMT (	(i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop '(':'(', p++
32	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT STMT	i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	STMT -> ASSIGN_OR_CMP_STMT
33	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT ASSIGN_OR_CMP_STMT	i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT -> ID ASSIGN_OR_CMP_STMT1
34	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT ASSIGN_OR_CMP_STMT1 ID	i = 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'ID':'i', p++
35	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT ASSIGN_OR_CMP_STMT1	= 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT1 -> ASSIGN_STMT
36	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT ASSIGN_STMT	= 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_STMT -> = EXPR SUFFIX
37	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR =	= 1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop '=':'=', p++
38	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR	1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	EXPR -> TERM EXPR2
39	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR2 TERM	1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TERM -> FACTOR TERM2
40	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR2 TERM2 FACTOR	1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	FACTOR -> CONST
41	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR2 TERM2 CONST	1 ; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'CONST':'1', p++
42	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR2 TERM2	; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TERM2 -> epsilon
43	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX EXPR2	; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	EXPR2 -> epsilon
44	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT SUFFIX	; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	SUFFIX -> ;
45	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT ;	; i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop ';':';', p++
46	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT STMT	i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	STMT -> ASSIGN_OR_CMP_STMT
47	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT ASSIGN_OR_CMP_STMT	i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT -> ID ASSIGN_OR_CMP_STMT1
48	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT ASSIGN_OR_CMP_STMT1 ID	i < 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'ID':'i', p++
49	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT ASSIGN_OR_CMP_STMT1	< 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT1 -> CMP_STMT
50	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT CMP_STMT	< 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	CMP_STMT -> CMP_OP EXPR SUFFIX
51	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR CMP_OP	< 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	CMP_OP -> <
52	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR <	< 3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop '<':'<', p++
53	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR	3 ; i = i + 1) { if (i == 2) { return i ; } } } #	EXPR -> TERM EXPR2
54	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR2 TERM	3 ; i = i + 1) { if (i == 2) { return i ; } } } #	TERM -> FACTOR TERM2
55	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR2 TERM2 FACTOR	3 ; i = i + 1) { if (i == 2) { return i ; } } } #	FACTOR -> CONST
56	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR2 TERM2 CONST	3 ; i = i + 1) { if (i == 2) { return i ; } } } #	pop 'CONST':'3', p++
57	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR2 TERM2	; i = i + 1) { if (i == 2) { return i ; } } } #	TERM2 -> epsilon
58	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX EXPR2	; i = i + 1) { if (i == 2) { return i ; } } } #	EXPR2 -> epsilon
59	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT SUFFIX	; i = i + 1) { if (i == 2) { return i ; } } } #	SUFFIX -> ;
60	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT ;	; i = i + 1) { if (i == 2) { return i ; } } } #	pop ';':';', p++
61	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT	i = i + 1) { if (i == 2) { return i ; } } } #	STMT -> ASSIGN_OR_CMP_STMT
62	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_OR_CMP_STMT	i = i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT -> ID ASSIGN_OR_CMP_STMT1
63	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_OR_CMP_STMT1 ID	i = i + 1) { if (i == 2) { return i ; } } } #	pop 'ID':'i', p++
64	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_OR_CMP_STMT1	= i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT1 -> ASSIGN_STMT
65	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_STMT	= i + 1) { if (i == 2) { return i ; } } } #	ASSIGN_STMT -> = EXPR SUFFIX
66	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR =	= i + 1) { if (i == 2) { return i ; } } } #	pop '=':'=', p++
67	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR	i + 1) { if (i == 2) { return i ; } } } #	EXPR -> TERM EXPR2
68	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM	i + 1) { if (i == 2) { return i ; } } } #	TERM -> FACTOR TERM2
69	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2 FACTOR	i + 1) { if (i == 2) { return i ; } } } #	FACTOR -> ID
70	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2 ID	i + 1) { if (i == 2) { return i ; } } } #	pop 'ID':'i', p++
71	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2	+ 1) { if (i == 2) { return i ; } } } #	TERM2 -> epsilon
72	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2	+ 1) { if (i == 2) { return i ; } } } #	EXPR2 -> + TERM EXPR2
73	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM +	+ 1) { if (i == 2) { return i ; } } } #	pop '+':'+', p++
74	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM	1) { if (i == 2) { return i ; } } } #	TERM -> FACTOR TERM2
75	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2 FACTOR	1) { if (i == 2) { return i ; } } } #	FACTOR -> CONST
76	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2 CONST	1) { if (i == 2) { return i ; } } } #	pop 'CONST':'1', p++
77	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2	) { if (i == 2) { return i ; } } } #	TERM2 -> epsilon
78	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2	) { if (i == 2) { return i ; } } } #	EXPR2 -> epsilon
79	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX	) { if (i == 2) { return i ; } } } #	SUFFIX -> epsilon
80	# FUNC_LIST) STMT_LIST CODE_BLOCK)	) { if (i == 2) { return i ; } } } #	pop ')':')', p++
81	# FUNC_LIST) STMT_LIST CODE_BLOCK	{ if (i == 2) { return i ; } } } #	CODE_BLOCK -> { STMT_LIST }
82	# FUNC_LIST) STMT_LIST {	{ if (i == 2) { return i ; } } } #	pop '{':'{', p++
83	# FUNC_LIST) STMT_LIST	if (i == 2) { return i ; } } } #	STMT_LIST -> STMT STMT_LIST
84	# FUNC_LIST) STMT_LIST STMT	if (i == 2) { return i ; } } } #	STMT -> SLCT_STMT
85	# FUNC_LIST) STMT_LIST SLCT_STMT	if (i == 2) { return i ; } } } #	SLCT_STMT -> if (STMT) CODE_BLOCK
86	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT (if	if (i == 2) { return i ; } } } #	pop 'if':'if', p++
87	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT (	(i == 2) { return i ; } } } #	pop '(':'(', p++
88	# FUNC_LIST) STMT_LIST CODE_BLOCK) STMT	i == 2) { return i ; } } } #	STMT -> ASSIGN_OR_CMP_STMT
89	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_OR_CMP_STMT	i == 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT -> ID ASSIGN_OR_CMP_STMT1
90	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_OR_CMP_STMT1 ID	i == 2) { return i ; } } } #	pop 'ID':'i', p++
91	# FUNC_LIST) STMT_LIST CODE_BLOCK) ASSIGN_OR_CMP_STMT1	== 2) { return i ; } } } #	ASSIGN_OR_CMP_STMT1 -> CMP_STMT
92	# FUNC_LIST) STMT_LIST CODE_BLOCK) CMP_STMT	== 2) { return i ; } } } #	CMP_STMT -> CMP_OP EXPR SUFFIX
93	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR CMP_OP	== 2) { return i ; } } } #	CMP_OP -> ==
94	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR ==	== 2) { return i ; } } } #	pop '==':'==', p++
95	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR	2) { return i ; } } } #	EXPR -> TERM EXPR2
96	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM	2) { return i ; } } } #	TERM -> FACTOR TERM2
97	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2 FACTOR	2) { return i ; } } } #	FACTOR -> CONST
98	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2 CONST	2) { return i ; } } } #	pop 'CONST':'2', p++
99	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2 TERM2	) { return i ; } } } #	TERM2 -> epsilon
100	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX EXPR2	) { return i ; } } } #	EXPR2 -> epsilon
101	# FUNC_LIST) STMT_LIST CODE_BLOCK) SUFFIX	) { return i ; } } } #	SUFFIX -> epsilon
102	# FUNC_LIST) STMT_LIST CODE_BLOCK)	) { return i ; } } } #	pop ')':')', p++
103	# FUNC_LIST) STMT_LIST CODE_BLOCK	{ return i ; } } } #	CODE_BLOCK -> { STMT_LIST }
104	# FUNC_LIST) STMT_LIST) STMT_LIST {	{ return i ; } } } #	pop '{':'{', p++
105	# FUNC_LIST) STMT_LIST) STMT_LIST	return i ; } } } #	STMT_LIST -> STMT STMT_LIST
106	# FUNC_LIST) STMT_LIST) STMT_LIST STMT	return i ; } } } #	STMT -> RTN_STMT
107	# FUNC_LIST) STMT_LIST) STMT_LIST RTN_STMT	return i ; } } } #	RTN_STMT -> return EXPR ;
108	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR return	return i ; } } } #	pop 'return':'return', p++
109	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR	i ; } } } #	EXPR -> TERM EXPR2
110	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR2 TERM	i ; } } } #	TERM -> FACTOR TERM2
111	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR2 TERM2 FACTOR	i ; } } } #	FACTOR -> ID
112	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR2 TERM2 ID	i ; } } } #	pop 'ID':'i', p++
113	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR2 TERM2	; } } } #	TERM2 -> epsilon
114	# FUNC_LIST) STMT_LIST) STMT_LIST ; EXPR2	; } } } #	EXPR2 -> epsilon
115	# FUNC_LIST) STMT_LIST) STMT_LIST ;	; } } } #	pop ';':';', p++
116	# FUNC_LIST) STMT_LIST) STMT_LIST	} } } #	STMT_LIST -> epsilon
117	# FUNC_LIST) STMT_LIST }	} } } #	pop '}':'}', p++
118	# FUNC_LIST) STMT_LIST	} } #	STMT_LIST -> epsilon
119	# FUNC_LIST) STMT_LIST }	} } #	pop '}':'}', p++
120	# FUNC_LIST) STMT_LIST	} #	STMT_LIST -> epsilon
121	# FUNC_LIST }	} #	pop '}':'}', p++
122	# FUNC_LIST	#	FUNC_LIST -> epsilon
123	#	#	done

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)：

```
<?xml version="1.0" encoding="UTF-8"?>
<ParserTree name="a.tree.xml">
  <CMPL_UNIT>
    <FUNC_LIST>
      <FUNC_DEF>
        <TYPE_SPEC>
          <keyword>int</keyword>
        </TYPE_SPEC>
        <ID>main</ID>
        <Sep_or_Operator>(</Sep_or_Operator>
        <ARG_LIST/>
        <Sep_or_Operator>)</Sep_or_Operator>
        <CODE_BLOCK>
          <Sep_or_Operator>{</Sep_or_Operator>
          <STMT_LIST>
            <STMT>
              <DECL_STMT>
                <TYPE_SPEC>
                  <keyword>int</keyword>
                </TYPE_SPEC>
                <ID>i</ID>
                <DECL_STMT1>
                  <Sep_or_Operator>=</Sep_or_Operator>
                  <EXPR>
                    <TERM>
                      <FACTOR>
                        <CONST>0</CONST>
                      </FACTOR>
                      <TERM2/>
                    </TERM>
                    <EXPR2/>
                  </EXPR>
                </DECL_STMT1>
                <Sep_or_Operator>;</Sep_or_Operator>
              </DECL_STMT>
            </STMT>
            <STMT_LIST>
              <STMT>
                <ITER_STMT>
                  <keyword>for</keyword>
                  <Sep_or_Operator>(</Sep_or_Operator>
                  <STMT/>
                  <STMT/>
                  <STMT>
                    <ASSIGN_OR_CMP_STMT>
                      <ID>i</ID>
                      <ASSIGN_OR_CMP_STMT1>
                        <ASSIGN_STMT>
                          <Sep_or_Operator>=</Sep_or_Operator>
                          <EXPR>
                            <TERM>
                              <FACTOR>
                                <CONST>1</CONST>
                              </FACTOR>
                              <TERM2/>
                            </TERM>
                            <EXPR2/>
                          </EXPR>
                          <SUFFIX>
                            <Sep_or_Operator>;</Sep_or_Operator>
                          </SUFFIX>
                        </ASSIGN_STMT>
                      </ASSIGN_OR_CMP_STMT1>
                    </ASSIGN_OR_CMP_STMT>
                    <ASSIGN_OR_CMP_STMT>
                      <ID>i</ID>
```

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 )
```