南京理工大学计算机科学与工程学院

编译原理 报告

班　　级 xxxxxxxxxx

学生姓名 xx

学　　号 xxxxxxxxxxx

指导教师 项欣光

南京理工大学计算机科学与工程学院制

目 录

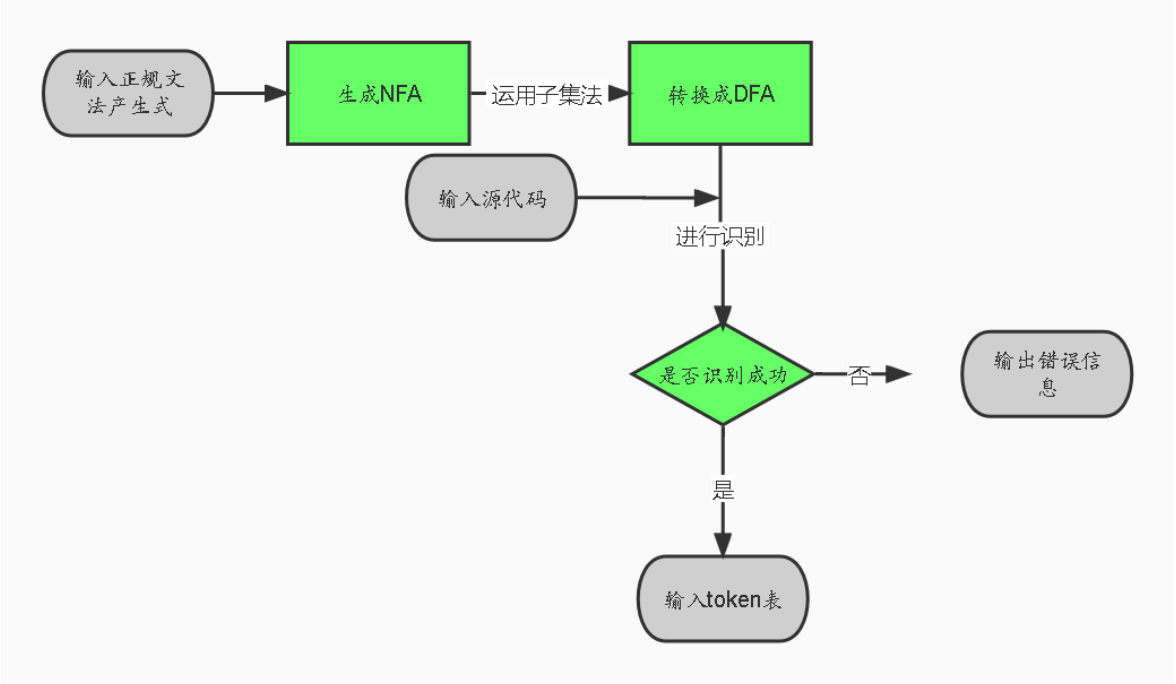
1 词法分析程序

1.1 功能

支持分析 c++语法的常规单词。程序输入：正规文法产生式文件和源代码文件；程序输出：token 表文件，该表由五种 token 组成：关键词，标识符，常量，限定符，运算符。表项结构为三元组：所在行号，类别，token 内容。

1.2 总体方案

1.2.1 总体流程图



图表 1 词法分析程序流程图

1. 正规文法转换成 NFA：

一个产生式 A→tB,可对应转换函数 f(A, t) = B

一个产生式 A→Ɛ,可对应 A 为终态

2. NFA 转换成 DFA：

子集法

两个基本运算：

1. 状态集合 I 的Ɛ闭包，表示为Ɛ-closure(I)，定义为一个状态集，是状态集 I 中的任何状

态 S 经过任意条ε弧而能够到达的状态的集合。

2. 状态集合 I 的 a 弧转换，表示为 move(I, a)，定义为状态集合 J，其中集合 J 是所有那些可从 I 中的某一状态经过一条 a 弧而到达的状态的全体。

1.2.2 相关数据结构

[]表示列表结构
()表示集合结构
{}表示字典结构

```
class NFA:
    data = {}  # 子集之间联系 {子集下标: {弧 : [到达的子集下标]}}
    K = {}  # 状态集合（非终结符集）  {状态: {弧 : [到达的状态]}}
    S = []  # 初态集合  [初态,初态,...]
    Z = []  # 终态集合  [终态,终态,...]
    sigma = ''  # 弧集合（终结符集）
    closures = []  # 子集集合 [子集,子集,... ] 子集表示为[状态,状态,...]
    closuresTemp = []  # 子集集合缓存 保存尚为被标记的子集

    # 从正规文法转换成NFA
    def __init__(self, filename):...

    # 初始化e-closure
    def initClosure(self):...

    # 创建e-closure()
    def createNode(self, keys: list, s, route):...

    # 遍历所有弧，进行move计算
    def move(self):...

    # 转化成DFA
    def toDFA(self):...
```

图表 2 类 NFA

```
class DFA:
    K = {}  # 状态集合（非终结符集）  {状态: {弧 : [到达的状态]}}
    S = []  # 初态集合  [初态,初态,...]
    Z = []  # 终态集合  [终态,终态,...]
    sigma = ''  # 弧集合（终结符集）
    closures = []  # 子集集合 [子集,子集,... ] 子集表示为列表[状态,状态,...]

    # 初始化DFA，对输入的数据进行简单处理罢了
    def __init__(self, sigma, S, Z, closures: list, data):...

    # DFA工作处理输入字符串 使用bfs算法
    def work(self, string):...
```

图表 3 类 DFA

1.3 详细设计

1.3.1 NFA 函数

```
# 从正规文法转换成NFA
def __init__(self, filename):
    f = open(filename, 'r', encoding='utf-8')
    content = f.readlines()
    f.close()
    for x in content:
        x = x.replace('\n', '')
        if x.find(':') != -1:
            temp = x.split(':')
            if temp[0] == 'K':
                for y in temp[1]:
                    self.K[y] = {}
            elif temp[0] == 'S':
                self.S = temp[1]
            elif temp[0] == 'sigma':
                temp[1] += temp[2]
                self.sigma = temp[1]
        elif x.find('->') != -1:
            temp = x.split('->')
            if len(temp[1]) == 1:
                if '' in self.K[temp[0]]:
                    self.K[temp[0]][''].append(temp[1][0])
                else:
                    self.K[temp[0]][''] = [temp[1][0]]
            elif len(temp[1]) == 2:
                if temp[1][0] in self.K[temp[0]]:
                    self.K[temp[0]][temp[1][0]].append(temp[1][1])
                else:
                    self.K[temp[0]][temp[1][0]] = [temp[1][1]]
            else:
                self.Z += temp[0]
    pass
```

```
# 创建e-closure()
def createNode(self, keys: list, s, route):
    d = keys
    for key in keys:
        for x in self.K[key]:
            if x == '':
                d += self.K[key][x]
```

```
    d.sort()  # 新子集
    isRepeat = False
    i = 0
    j = 0
    for i in range(0, len(self.closures)):  # 判断子集是否已经存在在子集集合
        if operator.eq(self.closures[i], d):
            isRepeat = True
            break
    if not isRepeat:
        if len(self.closures) > 0:
            i = i + 1
        for j in range(0, len(self.closuresTemp)):  # 判断子集是否已经存在在子集集合缓存
            if operator.eq(self.closuresTemp[j], d):
                isRepeat = True
                break
    if not isRepeat:
        if len(self.closuresTemp) > 0:
            j = j + 1
        self.closuresTemp.append(d)  # 添加到子集缓存
    if s not in self.data:
        self.data[s] = {}
    if route not in self.data[s]:
        self.data[s][route] = []
    self.data[s][route].append(i + j)  # 生成一条弧
```

```
# 遍历所有弧，进行 move 计算
def move(self):
    for route in self.sigma:
        t = []
        for x in self.closuresTemp[0]:  # 取缓存栈底
            if route in self.K[x]:
                t += self.K[x][route]
        if len(t) > 0:
            self.createNode(t, len(self.closures), route)  # 生成新子集到缓存
    self.closures.append(self.closuresTemp.pop(0))  # 弹出缓存栈底到子集集合
```

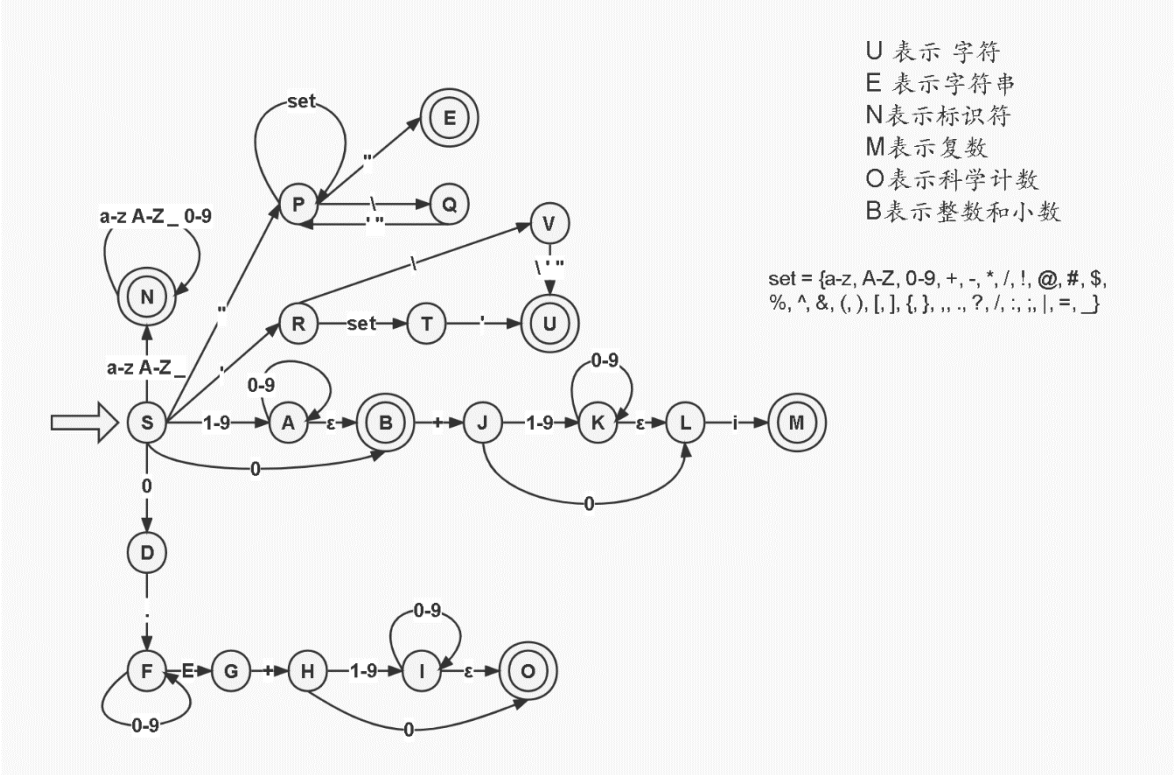
```
# 转化成 DFA
def toDFA(self):
    self.initClosure()
    while len(self.closuresTemp) > 0: # 直到缓存为空
        self.move()
    pass
    return DFA(self.sigma, self.S, self.Z, self.closures, self.data)
```

1.3.2 DFA 函数

```
# 初始化 DFA，对输入的数据进行简单处理罢了
def __init__(self, sigma, S, Z, closures: list, data):
    self.sigma = sigma
    self.closures = closures
    self.K = data
    for x in closures:
        if S in x:
            self.S.append(closures.index(x))
    for x in closures:
        for y in Z:
            if y in x:
                self.Z.append(closures.index(x))
```

```
# DFA 工作处理输入字符串 使用 bfs 算法
def work(self, string):
    q = queue.Queue()  # 队列
    q.put([self.S[0], 0])
    while not q.empty():
        u = q.get()
        if u[1] == len(string):  # 是否到达字符串末端
            if u[0] in self.Z:  # 是否处于终态
                return True
            else:
                return False
        c = string[u[1]]
        t = u[0]
        if t in self.K and c in self.K[t]:  # 是否存在路径
            res = self.K[t][c]
            for y in res:
                q.put([y, u[1] + 1])
        else:
            return False
```

1.4 测试与运行



图表 4 文法->NFA

K 表示非终结符集V_N

S 表示开始符 S

sigma 表示终结符集V_T

＜非终结符＞ → ＜终结符＞＜非终结符＞表示规则 P

```
K:SABCDFGHIJKLMNOPQRSTUVE

S:S
sigma:0123456789abcdefghijklmnopqrstuvwxyz.
+_ABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()-=/:;"'{}[]|\,<>?~`
S->0B
S->0D
S->1A
S->2A
S->3A
S->4A
S->5A
S->6A
S->7A
```

```
S->8A
S->9A
A->0A
A->1A
A->2A
A->3A
A->4A
A->5A
A->6A
A->7A
A->8A
A->9A
A->B
B->.C
C->A
D->.F
F->0F
F->1F
F->2F
F->3F
F->4F
F->5F
F->6F
F->7F
F->8F
F->9F
F->EG
G->+H
H->0O
H->1I
H->2I
H->3I
H->4I
H->5I
H->6I
H->7I
H->8I
H->9I
I->0I
I->1I
I->2I
I->3I
I->4I
I->5I
```

```
I->6I
I->7I
I->8I
I->9I
I->O
B->+J
J->1K
J->2K
J->3K
J->4K
J->5K
J->6K
J->7K
J->8K
J->9K
K->0K
K->1K
K->2K
K->3K
K->4K
K->5K
K->6K
K->7K
K->8K
K->9K
K->L
J->0L
L->iM
S->aN
S->bN
S->cN
S->dN
S->eN
S->fN
S->gN
S->hN
S->iN
S->jN
S->kN
S->lN
S->mN
S->nN
S->oN
S->pN
```

```
S->qN
S->rN
S->sN
S->tN
S->uN
S->vN
S->wN
S->xN
S->yN
S->zN
S->_N
S->AN
S->BN
S->CN
S->DN
S->EN
S->FN
S->GN
S->HN
S->IN
S->JN
S->KN
S->LN
S->MN
S->NN
S->ON
S->PN
S->QN
S->RN
S->SN
S->TN
S->UN
S->VN
S->WN
S->XN
S->YN
S->ZN
N->aN
N->bN
N->cN
N->dN
N->eN
N->fN
N->gN
```

```
N->hN
N->iN
N->jN
N->kN
N->lN
N->mN
N->nN
N->oN
N->pN
N->qN
N->rN
N->sN
N->tN
N->uN
N->vN
N->wN
N->xN
N->yN
N->zN
N->AN
N->BN
N->CN
N->DN
N->EN
N->FN
N->GN
N->HN
N->IN
N->JN
N->KN
N->LN
N->MN
N->NN
N->ON
N->PN
N->QN
N->RN
N->SN
N->TN
N->UN
N->VN
N->WN
N->XN
N->YN
```

```
N->ZN
N->0N
N->1N
N->2N
N->3N
N->4N
N->5N
N->6N
N->7N
N->8N
N->9N
N->_N
S->"P
P->aP
P->bP
P->cP
P->dP
P->eP
P->fP
P->gP
P->hP
P->iP
P->jP
P->kP
P->lP
P->mP
P->nP
P->oP
P->pP
P->qP
P->rP
P->sP
P->tP
P->uP
P->vP
P->wP
P->xP
P->yP
P->zP
P->AP
P->BP
P->CP
P->DP
P->EP
```

```
P->FP
P->GP
P->HP
P->IP
P->JP
P->KP
P->LP
P->MP
P->NP
P->OP
P->PP
P->QP
P->RP
P->SP
P->TP
P->UP
P->VP
P->WP
P->XP
P->YP
P->ZP
P->0P
P->1P
P->2P
P->3P
P->4P
P->5P
P->6P
P->7P
P->8P
P->9P
P->:P
P->;P
P-> P
P->/P
P->[P
P->]P
P->{P
P->}P
P->|P
P->!P
P->@P
P->#P
P->$P
```

```
P->%P
P->^P
P->&P
P->*P
P->(P
P->)P
P->-P
P->+P
P->_P
P->=P
P->.P
P->~P
P->`P
P-><P
P->>P
P->,P
P->?P
R->aT
R->bT
R->cT
R->dT
R->eT
R->fT
R->gT
R->hT
R->iT
R->jT
R->kT
R->lT
R->mT
R->nT
R->oT
R->pT
R->qT
R->rT
R->sT
R->tT
R->uT
R->vT
R->wT
R->xT
R->yT
R->zT
R->AT
```

```
R->BT
R->CT
R->DT
R->ET
R->FT
R->GT
R->HT
R->IT
R->JT
R->KT
R->LT
R->MT
R->NT
R->OT
R->PT
R->QT
R->RT
R->ST
R->TT
R->UT
R->VT
R->WT
R->XT
R->YT
R->ZT
R->0T
R->1T
R->2T
R->3T
R->4T
R->5T
R->6T
R->7T
R->8T
R->9T
R->:T
R->;T
R->/T
R->[T
R->]T
R->{T
R->}T
R->|T
R->!T
```

```
R->@T
R->#T
R->$T
R->%T
R->^T
R->&T
R->*T
R-> T
R->(T
R->)T
R->-T
R->+T
R->_T
R->=T
R->.T
R->~T
R->`T
R-><T
R->>T
R->,T
R->?T
P->\Q
Q->'P
Q->"P
P->"E
S->'R
R->\V
V->\T
V->'T
V->"T
T->'U
```

```
void myPrint(int i,int j){
    return i+j;
}
int main(){
    int n = 5,m = 3;
    int*p=&n;
    bool isOK = true;
    n+=(3*m);
    myPrint(n,m);
    double xs = 156.154;
    double d=0.123E+5;
    double fs=( 5+9i )+5;
    string s = "hello \"world!";
    char c = s[1];
    if(c == 'a'&&(n==m || isOK)){
        n++;
    }
    return 0;
}
```

图表 5 源代码

输出：

```
1 关键词 void
1 标识符 myPrint
1 限定符 (
1 关键词 int
1 标识符 i
1 限定符 ,
1 关键词 int
1 标识符 j
1 限定符 )
1 限定符 {
2 关键词 return
2 标识符 i
2 运算符 +
2 标识符 j
2 限定符 ;
3 限定符 }
4 关键词 int
4 标识符 main
4 限定符 (
4 限定符 )
4 限定符 {
5 关键词 int
5 标识符 n
5 运算符 =
5 常量 5
```

```
5 限定符 ,
5 标识符 m
5 运算符 =
5 常量 3
5 限定符 ;
6 关键词 int
6 运算符 *
6 标识符 p
6 运算符 =
6 运算符 &
6 标识符 n
6 限定符 ;
7 关键词 bool
7 标识符 isOK
7 运算符 =
7 关键词 true
7 限定符 ;
8 标识符 n
8 运算符 +=
8 限定符 (
8 常量 3
8 运算符 *
8 标识符 m
8 限定符 )
8 限定符 ;
9 标识符 myPrint
9 限定符 (
9 标识符 n
9 限定符 ,
9 标识符 m
9 限定符 )
9 限定符 ;
10 关键词 double
10 标识符 xs
10 运算符 =
10 常量 156.154
10 限定符 ;
11 关键词 double
11 标识符 d
11 运算符 =
11 常量 0.123E+5
11 限定符 ;
12 关键词 double
12 标识符 fs
```

```
12 运算符 =
12 限定符 (
12 常量 5+9i
12 限定符 )
12 运算符 +
12 常量 5
12 限定符 ;
13 关键词 string
13 标识符 s
13 运算符 =
13 常量 "hello \"world!"
13 限定符 ;
14 关键词 char
14 标识符 c
14 运算符 =
14 标识符 s
14 限定符 [
14 常量 1
14 限定符 ]
14 限定符 ;
15 关键词 if
15 限定符 (
15 标识符 c
15 运算符 ==
15 常量 'a'
15 限定符 &&
15 限定符 (
15 标识符 n
15 运算符 ==
15 标识符 m
15 限定符 ||
15 标识符 isOK
15 限定符 )
15 限定符 )
15 限定符 {
16 标识符 n
16 运算符 ++
16 限定符 ;
17 限定符 }
18 关键词 return
18 常量 0
18 限定符 ;
19 限定符 }
```

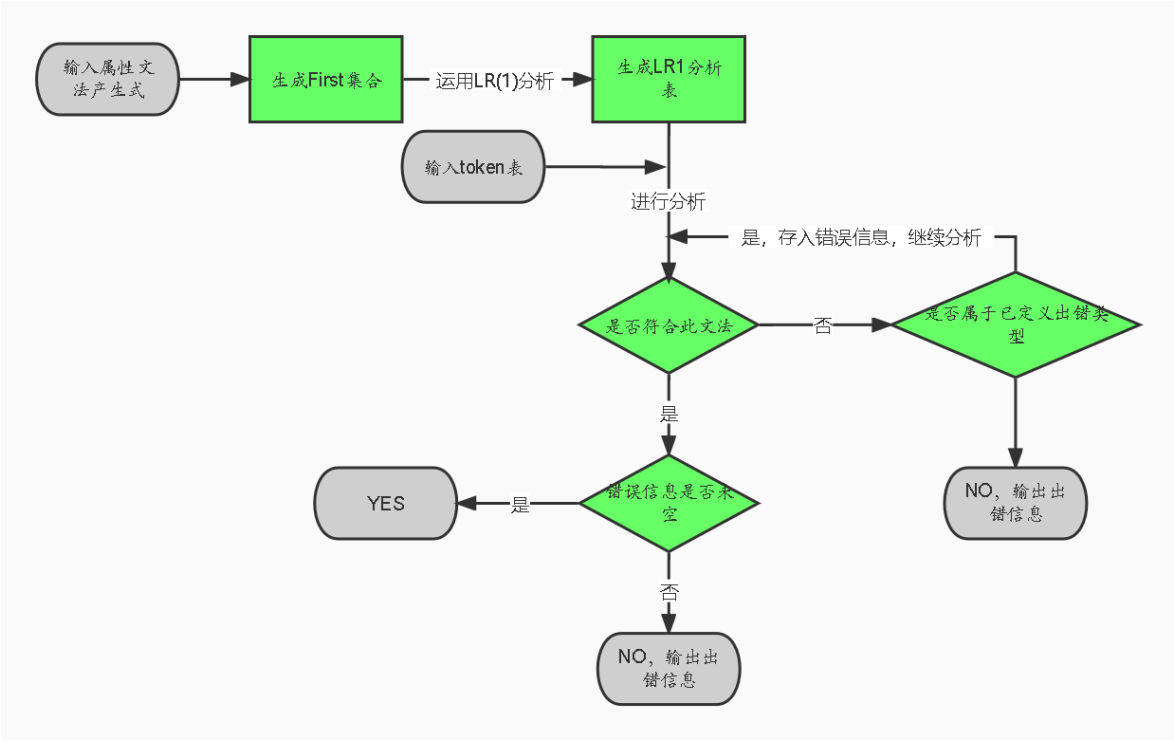
2 语法分析程序

2.1 功能

支持分析 c++简单语法。程序输入：上下文无关文法产生式和词法分析程序输出的 token 表；程序输出：YES 或 NO；错误提示，有语法错则标志出错行号和大致出错原因。期间会打印相关信息和分析过程，同时生成 html 文件。

2.2 总体方案

2.2.1 总体流程图



图表 6 语法分析程序流程图

1. 生成 Fisrt 集
   持续使用下面的方式，直到每个 First 集合不再扩大。

   ①. 若 X 属于 V_T，则 First(X) = {X}

   ②. 若 X 属于 V_N，且有产生式 X→a…，则把 a 加入到 First（X）中；若 X→ε，则把ε也加入到 First（X）中

③．若 X→Y…是一个产生式且 Y 属于V_N，则把 First（Y）中的所有非ε元素都加入到 First（X）中；若 X→$Y_1Y_2...Y_k$是一个产生式，$Y_1,Y_2,...,Y_{i-1}$都是非终结符，而且，对于任何 j, 1 <= j <= i-1，First (Y_j)都含有ε（即$Y_1,Y_2,...,Y_{i-1}$=- - - >ε）,则把 First (Y_i)中的所有非ε元素都加入到 First(X)中；特别的，若所有的 First (Y_j)都含有ε, j=1,2,3…k,则把ε加入到 First (X)中

2. 构造 LR（1）项目集的闭包函数

①．假定 I 是一个项目集，I 的任何项目都属于 CLOSURE（I）

②．若有项目 A→α · Bβ,a 属于 CLOSURE(I)，B→γ是一个产生式，那么，b∈First（β a）， 则 B→ · γ，b 也属于 CLOSURE(I)

③．重复步骤 2，直到 CLOSURE(I)不再增大为止

3. 构造转换函数

GO(I，X)=CLOSUER(J)

其中 I 是 LR(1)项目集，X 是文法符号，J={任何形如[$A\to\alpha X\cdot\beta,a$] 的项目| [$A\to\alpha X\cdot\beta,a$] ∈ I }

4. 构造 LR(1)分析表

①．若项目 [$A\to\alpha\cdot a\beta,b$]属于I_k，且 GO(I_k, a) = I_j，其中 a∈V_T，则置 ACTION[k,a]=S_j。移进

②．若项目 [$A\to\alpha\cdot,a$]属于I_k，则置 ACTION[k,a]=r_j，其中 a∈V_T。归约

③．若项目 [$S'\to S\cdot,\#$]属于I_k，则置 ACTION[k,#]=acc。接受

④．若 GO(I_k, A) = I_j,其中 A∈V_N，则置 GOTO[k, A]=j。

⑤．凡不能用规则 1-4 填入分析表中的元素，均置报错标志。

2.2.2 相关数据结构

[]表示列表结构

()表示集合结构

{}表示字典结构

```
class DATA:
    rules = []  # 文法产生式集合
    first = {}  # first集合 {非终结符:(终结符, 终结符,...)}
    items = []  # 项目集集合
    d = []  # 项目集未进行闭包时的集合
    K = {}  # 非终结符集
    S = ''  # 开始符
    sigma = []  # 终结符集
```

图表 7 类 DATA

```
class Item:
    child = {}  # 表示ACTION和GOTO {GOTO,ACTION} GOTO表示为{非终结符:项目集下标} ACTION表示为{终结符:文法产生式}
    content = {}  # 处理过后的内容 {活前缀后面一个符号:{非终结符:[[文法产生式,活前缀长度,向前搜索符集合()]]}}
    data = {}  # 内容 {非终结符:{文法产生式第一个符号:[[文法产生式,活前缀长度,向前搜索符集合()]]}}

    # 进行闭包处理生成项目集
    def __init__(self, d):...

    # 创建新项目集
    def createChildren(self):...
```

图表 8 类 Item

```
message = [
    'except ;',
    'declaration does not declare anything',
    'except }',
    'except )',
    'except ]',
]
```

图表 9 定义的错误信息

2.3 详细设计

```
# 创建 First 集合
def createFirst():
    s = {}
    t = []  # First 集合全为终结符的非终结符集合
    for x in DATA.K:
        isOk = True
        for y in DATA.K[x]:
            if y in DATA.K and y != x:  # first 集合中加入非终结符，则存到 s 中
                if x not in DATA.first:
                    DATA.first[x] = set()
                if x in s:
                    s[x].append(y)
                else:
                    s[x] = [y]
                isOk = False

            elif y in DATA.sigma:
                if x not in DATA.first:
                    DATA.first[x] = set()
                DATA.first[x].add(y)
        if isOk:
            t.append(x)
```

```
    while len(s) > 0:
        for x in s:
            if len(s[x]) == 0:  # first集合中无非终结符，则加入t,并从s中弹出
                s.pop(x)
                t.append(x)
                break
            else:
                for y in s[x]:
                    if y in t:
                        DATA.first[x] = DATA.first[x].union(DATA.first[y])
                        s[x].remove(y)
    printFirst()  # 打印First集合
```

```
# 获得当前符号列表的First
def getFirst(s) → set:
    need = set()
    i = 0
    while i < len(s):
        if type(s[i]) == set:
            need = need.union(s[i])  # 合并
            break
        else:
            if s[i] in DATA.sigma and s[i] != '$':  # 如果是空集，则继续
                need.add(s[i])
                break
            elif s[i] in DATA.K:
                need = need.union(DATA.first[s[i]])
                if '$' not in DATA.K[s[i]]:  # 如果可以产生空集，则继续
                    break
        i = i + 1
    return need
```

```
class Item:

    def __init__(self, d):
        self.child = {}  # action or goto {非终结符:项目集下标goto,终结符:文法产生式action}
        self.content = {}  # 处理过后的内容 {活前缀后面一个符号:{非终结符:[[文法产生式,活前缀长度,向前搜索符集合()]]}}
        self.data = deepcopy(d)  # 内容 {非终结符:{文法产生式第一个符号:[[文法产生式,活前缀长度,向前搜索符集合()]]}}
        DATA.d.append(deepcopy(d))
        # 进行闭包处理
        while True:
            isOK = True
```

```
        Temp = {} # 一轮中新增的项目
        for xx in d:
            for yy in d[xx]:
                for zz in d[xx][yy]:
                    if zz[1] == len(zz[0]):  # 已经到顶了
                        continue
                    c = zz[0][zz[1]]  # 活前缀后面第一个符号
                    if c in DATA.K:
                        if c not in Temp:
                            Temp[c] = {}
                        if c not in self.data:
                            self.data[c] = {}
                        for z in DATA.K[c]:  # 遍历 c 的文法产生式
                            if z not in Temp[c]:
                                Temp[c][z] = []
                            if z not in self.data[c]:
                                self.data[c][z] = []
                            for k in DATA.K[c][z]:
                                te = zz[0][zz[1] + 1:]
                                te.append(zz[2])
                                l = [k, 0, getFirst(te)]
                                Temp[c][z].append(l)
                                try:
                                    self.data[c][z].index(l)
                                except ValueError:
                                    self.data[c][z].append(l)
                                    if k[0] in DATA.K:
                                        isOK = False
        d = deepcopy(Temp) # 深复制新增项目，继续进行闭包处理
        if isOK:
            break
    # 为生成新/子项目集做准备，生成 self.content
    for x in self.data:
        for y in self.data[x]:
            for z_ in self.data[x][y]:
                if z_[1] == len(z_[0]):
                    for k in z_[2]:
                        if k not in self.child:
                            self.child[k] = ''
                        if x == DATA.S:
                            self.child[k] = 'acc'
                        else:
                            self.child[k] = x + '→' + ' '.join(z_[0])
                    continue
                c = z_[0][z_[1]]
```

```
                if c not in self.content:
                    self.content[c] = {}
                if x not in self.content[c]:
                    self.content[c][x] = []
                self.content[c][x].append(z_)
        DATA.items.append(self)
        self.createChildren()  # 生成新/子项目集
        pass

    # 创建子项目集
    def createChildren(self):
        for x in self.content:  # x→弧
            d = {}
            # 创建一个项目集（未进行闭包处理）
            for y in self.content[x]:
                for z in self.content[x][y]:
                    if y not in d:
                        d[y] = {}
                    if z[0][0] not in d[y]:
                        d[y][z[0][0]] = []
                    d[y][z[0][0]].append([z[0], z[1] + 1, z[2]])
            if len(d) > 0:  # 查找是否已经存在与项目集集合
                try:
                    index = DATA.d.index(d)
                    self.child[x] = index
                except ValueError:
                    self.child[x] = len(DATA.items)
                    Item(d)
```

```
# 分析 Token 表
def analysisTokens(string: list, mes: str):
    errors = []  # 错误信息集合
    printTitle('分析过程')
    string.append('#')  # 输入栈
    state = [0]  # 状态栈
    fhs = ['#']  # 符号栈
    i = 0
    index = 0
    a = ''
    while len(state) > 0:
        index = index + 1
        c = string[i]
        state_cur = state[-1]
        if c not in DATA.items[state_cur].child:  # 判断输入符号是否在预测表中,
```

```
若不在，换成空字符串试试
            if '$' in DATA.items[state_cur].child:
                c = '$'
        if c in DATA.items[state_cur].child:
            temp = DATA.items[state_cur].child[c]
            if type(temp) == str: # 归约
                print('步骤:%d' % index)
                print('状态栈:%s' % str(state))
                print('符号栈:%s' % ','.join(fhs))
                print('输入栈:%s' % string[i:])
                print('Action:%s' % temp)
                a += '<tr class=\'bz\'><td>步骤</td><td>%d</td></tr>' % index
                a += '<tr><td>状态栈</td><td>%s</td></tr>' % str(state)
                a += '<tr><td>符号栈</td><td>%s</td></tr>' % ','.join(fhs)
                a += '<tr><td>输入栈</td><td>%s</td></tr>' % string[i:]
                a += '<tr><td>Action</td><td>%s</td></tr>' % temp
                if temp == 'acc':  # 接受，成功
                    print('GOTO:%s\n' % 'None')
                    a += '<tr><td>GOTO</td><td>None</td></tr>'
                    if len(errors) == 0:
                        return True, mes.replace('(process2)', a)
                    else:
                        return errors, mes.replace('(process2)', a)
                else:  # 归约
                    t = temp.split('→')
                    tt = t[1].split(' ')
                    ii = len(tt)
                    fhs = fhs[:-ii] + [t[0]]
                    while ii > 0: # 状态弹栈
                        ii = ii - 1
                        state.pop()
                    if t[0] not in DATA.items[state[-1]].child:
                        if '$' in DATA.items[state[-1]].child:
                            t[0] = '$'
                    if t[0] in DATA.items[state[-1]].child:
                        state.append(DATA.items[state[-1]].child[t[0]])  #
压入新状态
                    else:
                        errMes = judge(DATA.items[state[-1]].child)  # 判断错
误信息
                        errors.append([i, errMes[0]])
                        if errMes[1] is None:  # 若为未定义错误信息，直接报错
                            return errors, mes.replace('(process2)', a)
                        string.insert(i, errMes[1])
                    a += '<tr><td>GOTO</td><td>%d</td></tr>' % state[-1]
```

```
                print('GOTO:%d\n' % state[-1])

            elif type(temp) == int:  # 移进
                print('步骤:%d' % index)
                print('状态栈:%s' % str(state))
                print('符号栈:%s' % ','.join(fhs))
                print('输入栈:%s' % string[i:])
                print('Action:%s' % 'None')
                print('GOTO:%d\n' % temp)
                a += '<tr class=\'bz\'><td>步骤</td><td>%d</td></tr>' % index
                a += '<tr><td>状态栈</td><td>%s</td></tr>' % str(state)
                a += '<tr><td>符号栈</td><td>%s</td></tr>' % ','.join(fhs)
                a += '<tr><td>输入栈</td><td>%s</td></tr>' % string[i:]
                a += '<tr><td>Action</td><td>None</td></tr>'
                a += '<tr><td>GOTO</td><td>%d</td></tr>' % temp
                state.append(temp)
                fhs.append(c)
                if c != '$':
                    i = i + 1
            else:
                errMes = judge(DATA.items[state[-1]].child)
                errors.append([i, errMes[0]])
                if errMes[1] is None:
                    return errors, mes.replace('(process2)', a)
                string.insert(i, errMes[1])
```

2.4 测试与运行

输入：
上下文无关文法产生式：
K 表示非终结符集V_N
S 表示开始符 S
sigma 表示终结符集V_T

$\alpha \to \beta$表示规则 P

$表示$\varepsilon$

```
K:S' S define func_ret func_define paras block para dv_op sv_op pre_op
assigning_op exp logic_exp conjunction define_statement if_statement
else_statement while_statement ret_statement for_statement func_call ids
define_para valuation normal_statement exps id_add valuation_add para_add
exp_add exps const_add
```

```
S:S'
sigma:( ) [ ] ; , { } # + - * / & ^ && || | if else return for while void
int double float char string ++ -- ~ # = < > <= >= != == $ id const
S'→S
S→$
func_ret→void
func_ret→define
define→int
define→double
define→flout
define→string
define→char
define→bool
define→define *
func_define→func_ret id ( paras ) { block }
func_define→para ( paras ) { block }
func_define→func_ret id ( ) { block }
func_define→para ( ) { block }
para→define id
paras→para para_add
para_add→, paras
para_add→$
assigning_op→=
assigning_op→+=
assigning_op→-=
assigning_op→*=
assigning_op→/=
dv_op→>
dv_op→<
dv_op→==
dv_op→<=
dv_op→>=
dv_op→!=
dv_op→+
dv_op→-
dv_op→*
dv_op→/
dv_op→^
dv_op→&
dv_op→|
sv_op→++
sv_op→--
pre_op→~
define_statement→define valuation valuation_add ;
```

```
define_statement→define_para ;
define_para→define ids
valuation→id assigning_op exp
valuation→id [ const ] assigning_op { const const_add }
valuation→id assigning_op { const const_add }
const_add→, const const_add
const_add→$
valuation_add→, valuation valuation_add
valuation_add→$
exp→exp dv_op exp
exp→exp sv_op
exp→sv_op exp
exp→pre_op exp
exp→func_call
exp→const
exp→( exp )
exp→id
exp→id [ const ]
exp→& id
normal_statement→exp ;
normal_statement→valuation ;
normal_statement→func_call ;
func_call→id ( ids )
func_call→id ( )
ids→id id_add
id_add→, ids
id_add→$
for_statement→for ( define_statement logic_exp ; exp ) { block }
if_statement→if ( logic_exp ) { block } else_statement
if_statement→if ( logic_exp ) { block }
else_statement→else { block }
while_statement→while ( logic_exp ) { block }
ret_statement→return exp ;
conjunction→&&
conjunction→||
logic_exp→exps
exps→exp exp_add
exp_add→conjunction exps
exp_add→$
exp→exp conjunction exp
block→$
block→{ block }
block→define_statement block
block→if_statement block
block→else_statement block
```

```
block→for_statement block
block→while_statement block
block→ret_statement block
block→normal_statement block
S→func_define S
S→define_statement S
```

源代码：

```
void myPrint(int i,int j){

   return i+j;
}
int main(){
   int n = 5,m = 3;
   int*p=&n;
   int pp[3]={1,2,3};
   bool isOK = true;
   n+=(3*m;
   myPrint(n,m)
   double xs = 156.154;
   double d=0.123E+5;
   double fs=( 5+9i )+5;
   string s = "hello \"world!";
   char c = s[1;
   if(c == 'a'&&(n==m || isOK)){
      n++;
   }
   return 0;
```

输出：

```
NO
0. 8:11: error:except )
    n+=(3*m;
          ^
1. 10:4: error:except ;
    double xs = 156.154;
    ^
2. 14:16: error:except ]
    char c = s[1;
               ^
3. 18:12: error:except }
    return 0;
           ^
```

图表 10 输出错误信息

```
请输入想获得的项目集：6

----------------------------------------------------状态6----------------------------------------------------
S->define_statement S · , #

----------------------------------------------------子结点----------------------------------------------------

----------------------------------------------------结束----------------------------------------------------

请输入想获得的项目集：8

----------------------------------------------------状态8----------------------------------------------------
func_define->func_ret id · ( paras ) { block } , void/char/string/double/$/int/#
func_define->func_ret id · ( ) { block } , void/char/string/double/$/int/#

----------------------------------------------------子结点----------------------------------------------------
->          (           -> 9

----------------------------------------------------结束----------------------------------------------------
```

图表 11 查询某个项目集

源代码

```
void myPrint(int i,int j){
    return i+j;
}
int main(){
    int n = 5,m = 3;
    int*pp={1,2,3};
    int*p=&n;
    bool isOK = true;
    n+=(3*m;
    myPrint(n,m)
    double xs = 156.154;
    double d=0.123E+5;
    double fs=( 5+9i )+5;
    string s = "hello \"world!";
    char c = s[1;
```

Token表

```
1 关键词 void
1 标识符 myPrint
1 限定符 (
1 关键词 int
1 标识符 i
1 限定符 ,
1 关键词 int
1 标识符 j
1 限定符 )
1 限定符 {
2 关键词 return
2 标识符 i
2 运算符 +
2 标识符 j
2 限定符 ;
```

FIRST集合

```
FIRST( S' )=double  void  string  $  int  char
FIRST( S )=void  double  string  $  int  char
FIRST( define )=double  string  int  char
FIRST( func_ret )=double  void  string  int  char
FIRST( func_define )=void  double  string  int  char
FIRST( paras )=double  string  int  char
FIRST( block )=for  --  char  {  id  return  ~  double  const  &  if  string  (  $  ++  while  else  int
FIRST( para )=double  string  int  char
```

LR1预测分析表

状态	(	)	[	]	;	,	{	}	#	+	-	*	/	&	^	&&	||	|	if	else	return
0																					
1									acc												
2									r_1												
3																					
4									r_90												
5																					
6									r_91												
7																					
8	S_9																				
9		S_306																			
10		S_11																			

LR1预测分析过程

步骤	1
状态栈	[0]
符号栈	#
输入栈	['void', 'id', '(', 'int', 'id', ',', 'int', 'id', ')', '{', 'return', 'id', '+', 'id', ';', '}', 'int', 'id', '(', ')', '{', 'int', 'id', '=', 'const', ';', 'id', '=', 'const', ';', 'int', '*', 'id', '=', '{', 'const', ',', 'const', ',', 'const', '}', ';', 'int', '*', 'id', '=', '&', 'id', ';', '+=', '(', 'const', '*', 'id', ';', 'id', '(', 'id', ',', 'id', ')', 'double', 'id', '=', 'const', ';', 'double', 'id', '=', 'const', ';', 'double', 'id', '=', '(', 'const', ')', '+', 'const', ';', 'string', 'id', '=', 'const', ';', 'char', 'id', '=', 'id', '[', 'const', ';', '(', 'id', '==', 'id', '||', 'id', ')', ')', '{', 'id', '++', ';', '}', 'return', 'const', ';', '#']
Action	None
GOTO	335

语法分析结果

NO

```
9:11: error:except )
    n+=(3*m;
          ^
11:4: error:except ;
    double xs = 156.154;
    ^
15:16: error:except ]
    char c = s[1;
```

图表 12 生成的 html 文件

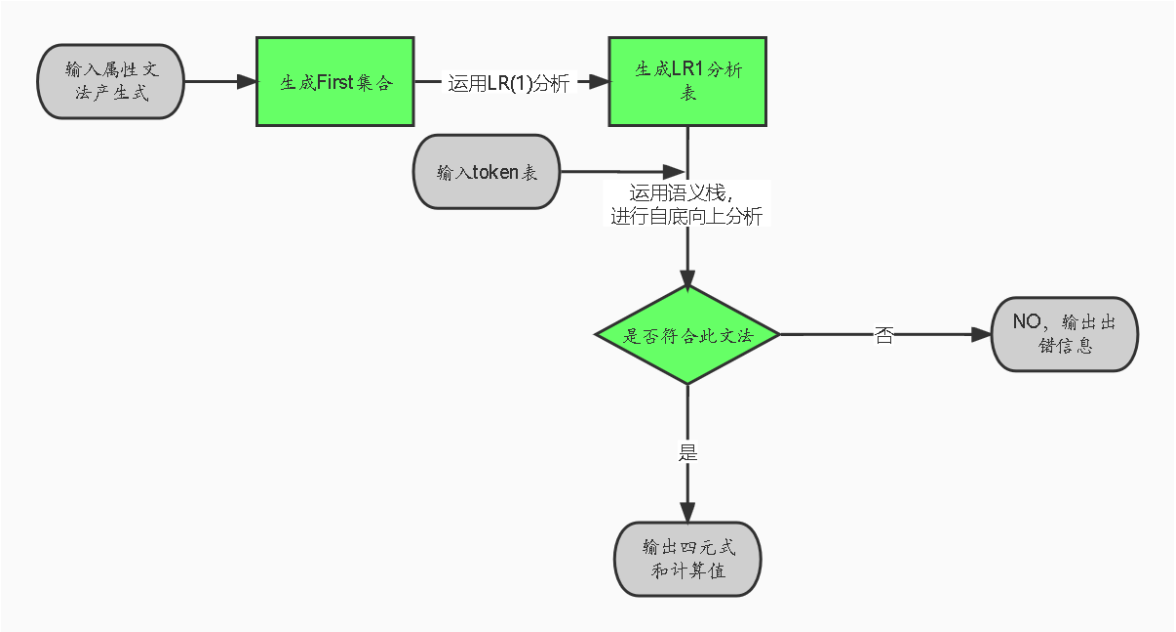
3 语义分析程序

3.1 功能

支持计算简单表达式。程序输入：属性文法产生式和词法分析程序输出的 token 表；程序输出：四元式，形式为 (op, y, z, x)，其中 op 为运算符，y 和 z 为运算数，x 为运算结果。和计算值。基于 S-属性文法的语义计算，采用自底向上的方式进行，采用 LR(1)技术进行语法分析，通过扩充分析栈中的域，形成语义栈来存放综合属性的当前取值。

3.2 总体方案

总体流程图



图表 13 语法分析程序流程图

语义栈(状态，符号，语义值)，符号'-'表示为定义的语义值。

3.2.1 相关数据结构

语义分析程序也是基于 LR(1)分析，所以数据结构和函数与语法分析程序类似。

```
TAC = []  # 四元组


class DATA:
    yuyi = []  # 语义函数集合
    rules = []  # 文法产生式集合
    first = {}  # first集合 {非终结符:(终结符，终结符,...)}
    items = []  # 项目集集合
    d = []  # 项目集未进行闭包时的集合
    K = {}  # 非终结符集
    S = ''  # 开始符
    sigma = []  # 终结符集
```

图表 14 类 DATA 和列表四元组

四元组形式表示为(op, y, z,x)，其中 op 为运算符，y 和 z 为运算数，x 为运算结果。

```
class Item:...


def LR1(filename):...


# 创建First集合
def createFirst():...


# 获得当前字符串的First
def getFirst(s) -> set:...
```

图表 15 相关函数和类与语法分析程序相同

3.3 详细设计

```
# 分析 Token 表
def analysisTokens(string: list, mes: str):
    global TAC
    printTitle('分析过程')
    string.append('#')
    stack = [[0, '#', '-']]  # 语义栈
    i = 0
    index = 0
    a = ''
    while len(stack) > 0:
        index = index + 1
        c = string[i]
        state_cur = stack[-1]
        if c[0] not in DATA.items[state_cur[0]].child:
```

```
            if '$' in DATA.items[state_cur[0]].child:
                c[0] = '$'
        if c[0] in DATA.items[state_cur[0]].child:
            temp = DATA.items[state_cur[0]].child[c[0]]
            if type(temp) == str:
                print('步骤:%d' % index)
                print('状态栈:%s' % str(stack))
                print('余留符号串:%s' % string[i:])
                print('分析动作:%s\n' % temp)
                a += '<tr class=\'bz\'><td>步骤</td><td>%d</td></tr>' % index
                a += '<tr><td>状态栈</td><td>%s</td></tr>' % str(stack)
                a += '<tr><td>余留符号串</td><td>%s</td></tr>' % string[i:]
                a += '<tr><td>分析动作</td><td>%s</td></tr>' % temp
                if temp == 'acc':
                    print(state_cur[2])
                    return True, mes.replace('(process2)', a)
                else:
                    t = temp.split('→')
                    tt = t[1].split(' ')
                    ii = len(tt)
                    paras = []  # 弹出的值
                    while ii > 0:
                        ii = ii - 1
                        temp_digit = stack.pop()[2]
                        if type(temp_digit) != str:
                            paras.append(temp_digit)
                    if t[0] not in DATA.items[stack[-1][0]].child:
                        if '$' in DATA.items[stack[-1][0]].child:
                            t[0] = '$'
                    if t[0] in DATA.items[stack[-1][0]].child:
                        try:
                            index1 = DATA.rules.index(temp)
                            m = DATA.yuyi[index1]
                            n = '-'  # 进行语义计算
                            if '+' in m:
                                n = paras[1] + paras[0]
                                TAC.append(('+', paras[1], paras[0], n))
                            elif '-' in m:
                                n = paras[1] - paras[0]
                                TAC.append(('-', paras[1], paras[0], n))
                            elif '*' in m:
                                n = paras[1] * paras[0]
                                TAC.append(('*', paras[1], paras[0], n))
                            elif '/' in m:
                                n = paras[1] / paras[0]
```

```
                        TAC.append(('/', paras[1], paras[0], n))
                    elif ':=' in m:
                        n = paras[0]
                        stack.append([
                                DATA.items[stack[-1][0]].child[t[0]],
                                t[0], n])
                except Exception as e:
                    print(e)
                    return i, mes.replace('(process2)', a)
            else:
                return i, mes.replace('(process2)', a)

        elif type(temp) == int:
            print('步骤:%d' % index)
            print('状态栈:%s' % str(stack))
            print('余留符号串:%s' % string[i:])
            print('分析动作:S_%d\n' % temp)
            a += '<tr class=\'bz\'><td>步骤</td><td>%d</td></tr>' % index
            a += '<tr><td>状态栈</td><td>%s</td></tr>' % str(stack)
            a += '<tr><td>余留符号串</td><td>%s</td></tr>' % string[i:]
            a += '<tr><td>分析动作</td><td>S_%d</td></tr>' % temp
            stack.append([temp, c[0], c[1]])
            if c[0] != '$':
                i = i + 1
    else:
        return i, mes.replace('(process2)', a)
```

3.4 测试与运行

输入：
属性文法：
K 表示非终结符集V_N
S 表示开始符 S
sigma 表示终结符集V_T

$\alpha \to \beta$表示规则 P　{}表示语义动作

```
K:S E T F R Q
S:S
sigma:+ * ( ) const - /
S→E{print(E.val)}
E→E + T{E.val:=E.val+T.val}
E→T{E.val:=T.val}
```

```
T→T * Q{T.val:=T.val*Q.val}
T→Q{T.val:=Q.val}
Q→Q / R{Q.val:=Q.val/R.val}
Q→R{Q.val:=R.val}
R→R - F{R.val:=R.val-F.val}
R→F{R.val:=F.val}
F→( E ){F.val:=E.val}
F→const{F.val:=const.lexval}
```

输入 token 表：

```
1 常量 5
1 运算符 +
1 常量 2
1 运算符 *
1 限定符 (
1 常量 4
1 运算符 -
1 常量 1
1 限定符 )
1 运算符 +
1 常量 8
1 限定符 /
1 限定符 (
1 常量 4
1 运算符 -
1 常量 2
1 限定符 )
1 常量 9
```

输出：

```
15.0

-------------------------------------------------四元式-------------------------------------------------
(- , 4.000 , 1.000 , 3.000)
(* , 2.000 , 3.000 , 6.000)
(+ , 5.000 , 6.000 , 11.000)
(- , 4.000 , 2.000 , 2.000)
(/ , 8.000 , 2.000 , 4.000)
(+ , 11.000 , 4.000 , 15.000)
```

图表 16 语义分析程序输出

```
请输入想获得的项目集：3

-------------------------------------------------状态3-------------------------------------------------
E->E + T · , #
E->E + T · , +
T->T · * Q , #
T->T · * Q , +
T->T · * Q , *

-------------------------------------------------子结点-------------------------------------------------
->          *           -> 4

-------------------------------------------------结束-------------------------------------------------
```

图表 17 显示相应项目集

源代码

5+2*(4-1)+8/(4-2)

Token表

1 常量 5
1 运算符 +
1 常量 2
1 运算符 *
1 限定符 (
1 常量 4
1 运算符 -
1 常量 1
1 限定符)
1 运算符 +
1 常量 8
1 限定符 /
1 限定符 (
1 常量 4
1 运算符 -

FIRST集合

FIRST(S)=(const
FIRST(E)=(const
FIRST(T)=(const
FIRST(F)=(const
FIRST(R)=(const
FIRST(Q)=(const

LR1预测分析表

状态	ACTION							GOTO					
	+	*	(	)	const	-	/	S	E	T	F	R	Q
0			S_10		S_29				1	33	30	31	32
1	S_2												
2			S_10		S_29					3	30	31	32
3	r_1	S_4											
4			S_10		S_29						30	31	5
5	r_3	r_3					S_6						
6			S_10		S_29						30	7	
7	r_5	r_5				S_8	r_5						
8			S_10		S_29						9		
9	r_7	r_7				r_7	r_7						
10			S_21		S_24				11	25	28	27	26
11	S_12			S_12									

LR1预测分析过程

步骤	1
状态栈	[[0, '#', '-']]
余留符号串	[['const', 5], ['+', '-'], ['const', 2], ['*', '-'], ['(', '-'], ['const', 4], ['-', '-'], ['const', 1], [')', '-'], ['+', '-'], ['const', 8], ['/', '-'], ['(', '-'], ['const', 4], ['-', '-'], ['const', 2], [')', '-'], '#']
分析动作	S_29
步骤	2
状态栈	[[0, '#', '-'], [29, 'const', 5]]
余留符号串	[['+', '-'], ['const', 2], ['*', '-'], ['(', '-'], ['const', 4], ['-', '-'], ['const', 1], [')', '-'], ['+', '-'], ['const', 8], ['/', '-'], ['(', '-'], ['const', 4], ['-', '-'], ['const', 2], [')', '-'], '#']

语义分析结果

四元式

-	4.000	1.000	3.000
*	2.000	3.000	6.000
+	5.000	6.000	11.000
-	4.000	2.000	2.000
/	8.000	2.000	4.000
+	11.000	4.000	15.000

图表 18 语义分析程序生成的 html