南京理工大学计算机科学与工程学院

编译原理 课程设计报告

学生姓名 严宇昂

专　　业 计算机科学与技术

学　　号 919106840540

课程名称 软件课程设计（II）

指导教师 项欣光

2022.4

南京理工大学计算机科学与工程学院制

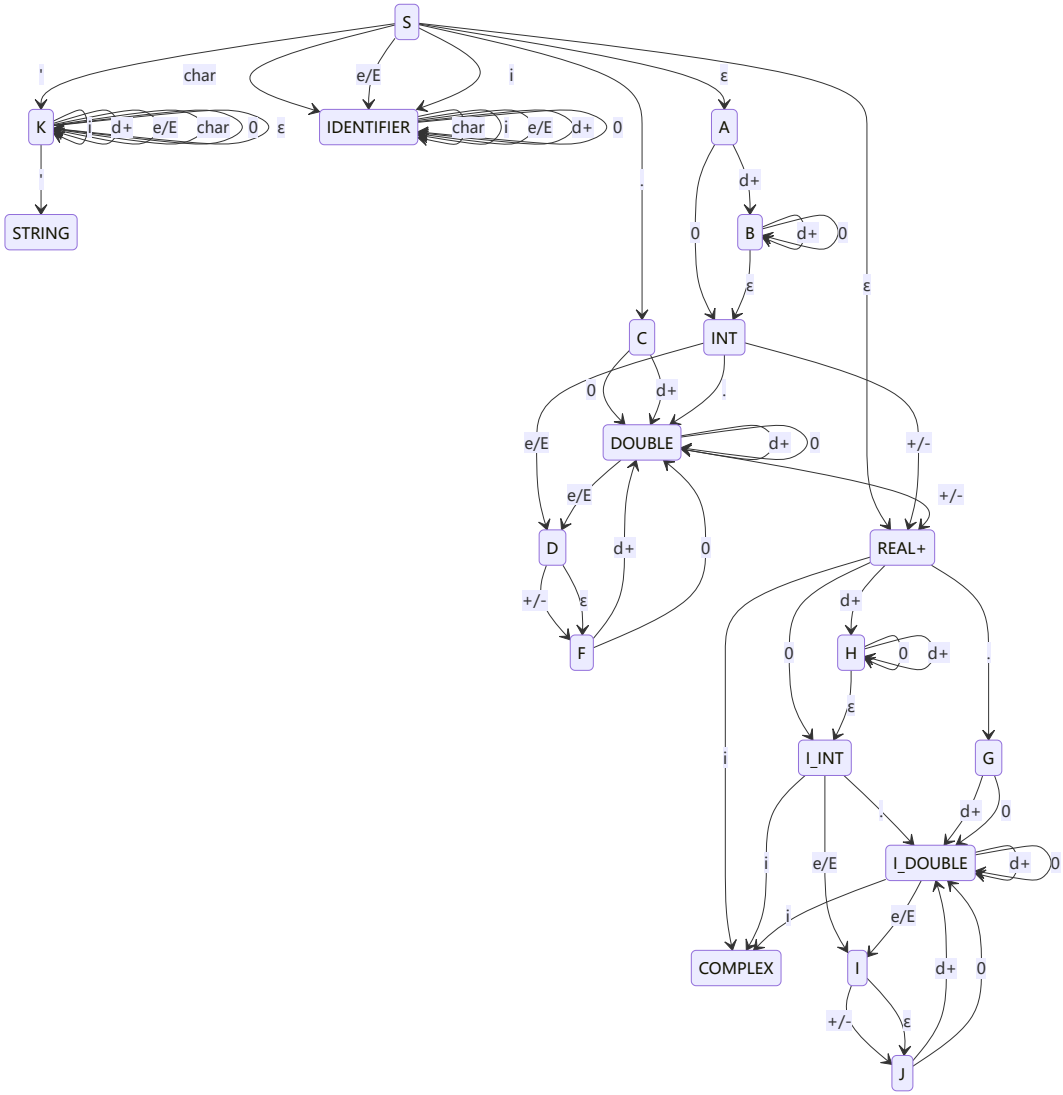
任务一 词法分析

1 设计思路

1.1 词法设计

1.1.1 NFA设计

在词法分析程序中，为分析器设计的文法NFA如图所示：



其中可接受状态为`IDENTIFIER`，`STRING`，`DOUBLE`，`COMPLEX`，分别为标识符，字符串，浮点数，复数

转换边解释：

`i`：字母i

`e/E`：小写字母e或大写字母E

`char`：其他字母字符

`d+`：1-9的数字字符

`0`：字符0

`+/-`：字符+或字符-

`ε`：空输入

在处理前先对所有字符进行分类，简化了NFA的复杂程度

1.1.2 关键词，操作符，限定符设置

关键词：`for`, `while`, `if`, `else`, `return`, `break`, `continue`, `def`, `class`, `int`, `double`

赋值操作符：`+=`, `-=`, `*=`, `/=`, `*=`, `//=`, `%=`, `=`

二元运算符：`and`, `or`, `xor`, `==`, `**`, `+`, `-`, `*`, `/`, `//`, `%`, `in`, `<`, `>`, `<=`, `>=`

单目运算符：`!`, `not`

限定符：`.` `:` `,` `;` `[` `]` `{` `}` `(` `)`

1.2 数据结构的设计

本程序的NFA使用python的字典，是一种索引数据结构.

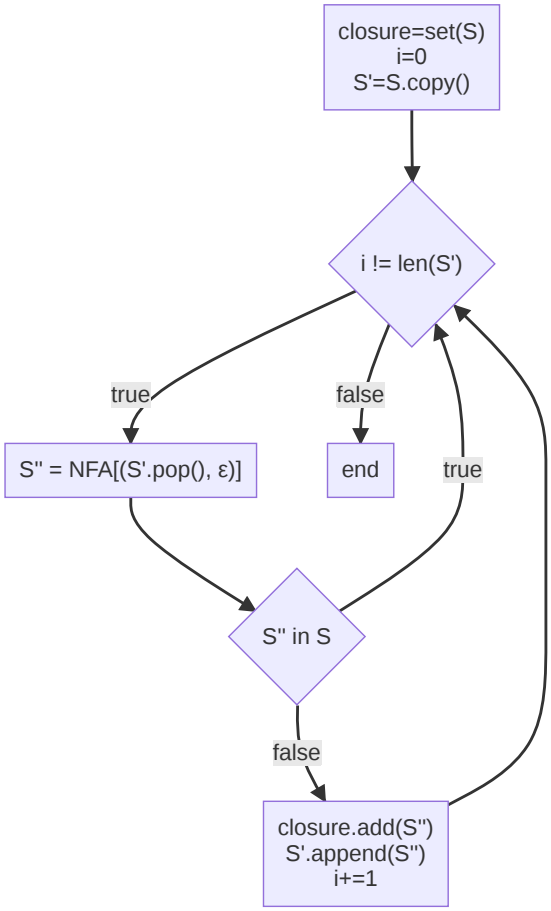
设一条正规产生式为$f(A,t)=Z$

具体格式为NFA[(A, t)] = Z. A, t作为键，Z作为值存储，方便查找，降低算法复杂度。由于NFA中$f(A,t)$可能有不止一种的状态，Z用set类保存所有f(A,t)的状态

而DFA中f(A, t)只有一种状态，因此Z不再用set保存，直接赋值。

1.3 ε-closure的计算

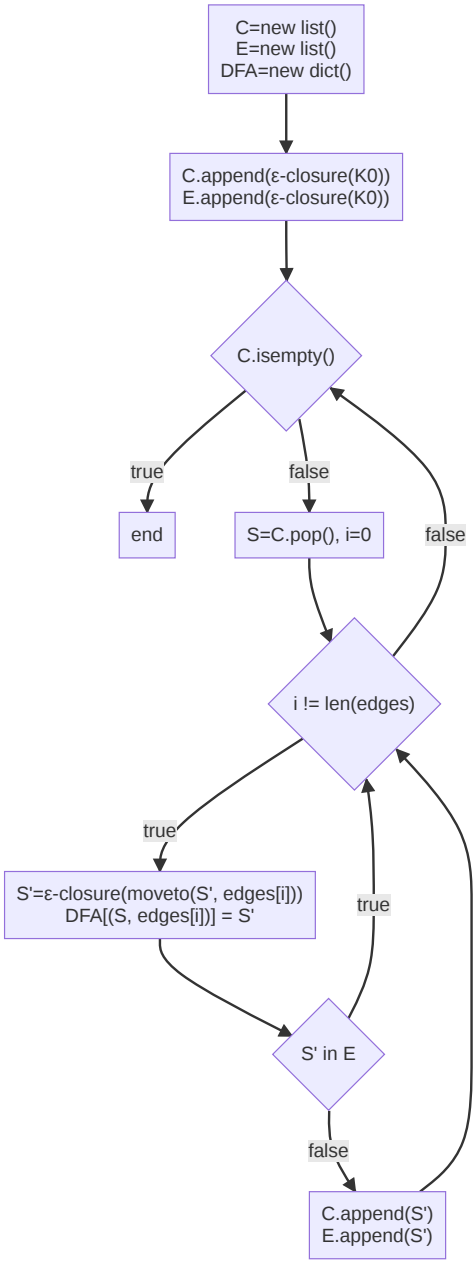
算法流程图如下：



遍历集合中的每一个元素，查找对应的ε边转换，加入到集合中，循环操作，直到集合不再增大为止

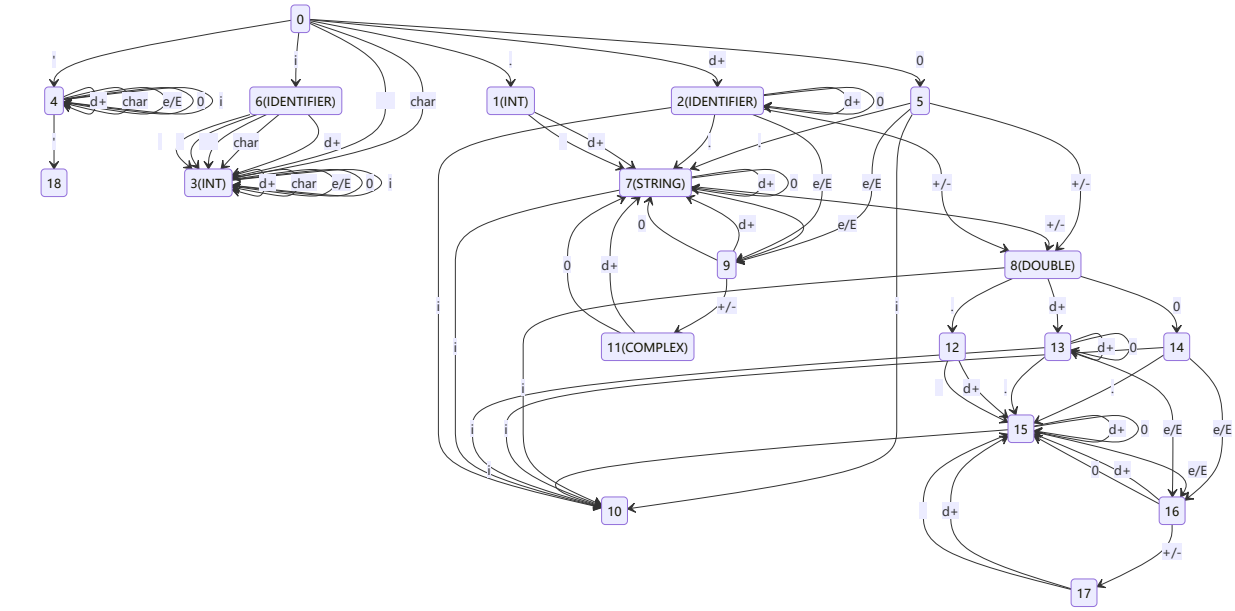
1.4 NFA到DFA转换

算法流程图如下：



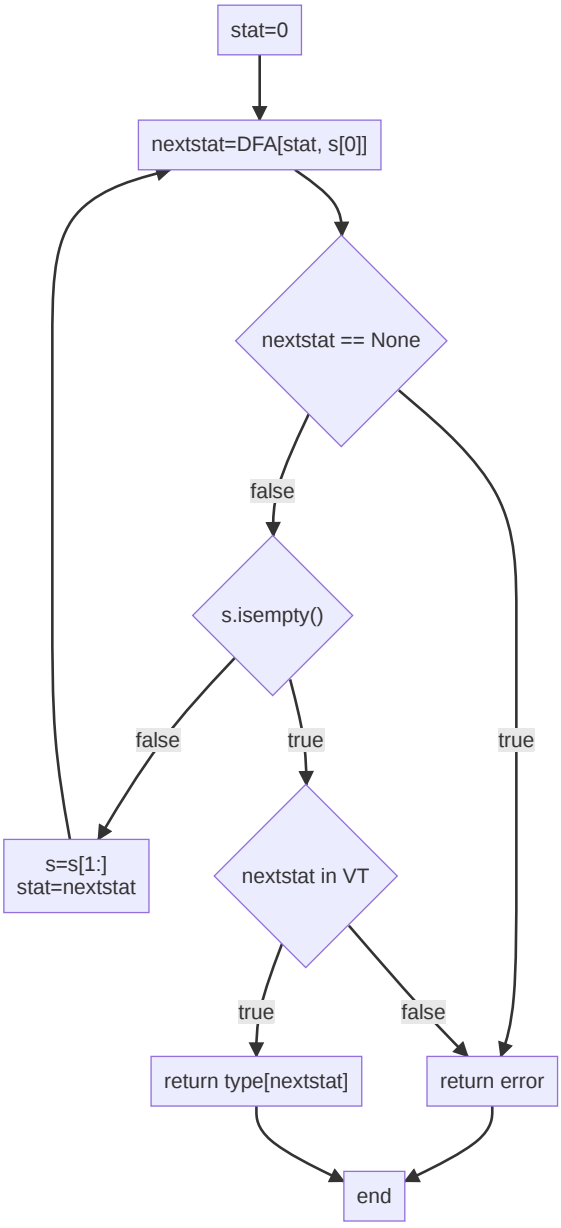
NFA到DFA转换采用了图遍历DFS算法，在这种情况下，图有可能是有环的。因此除了需要用一个队列C维护未被访问的状态，还要用一个队列维护已经访问过的数组E，用来防止回边导致的死循环。

NFA经过转换后的DFA如图所示：



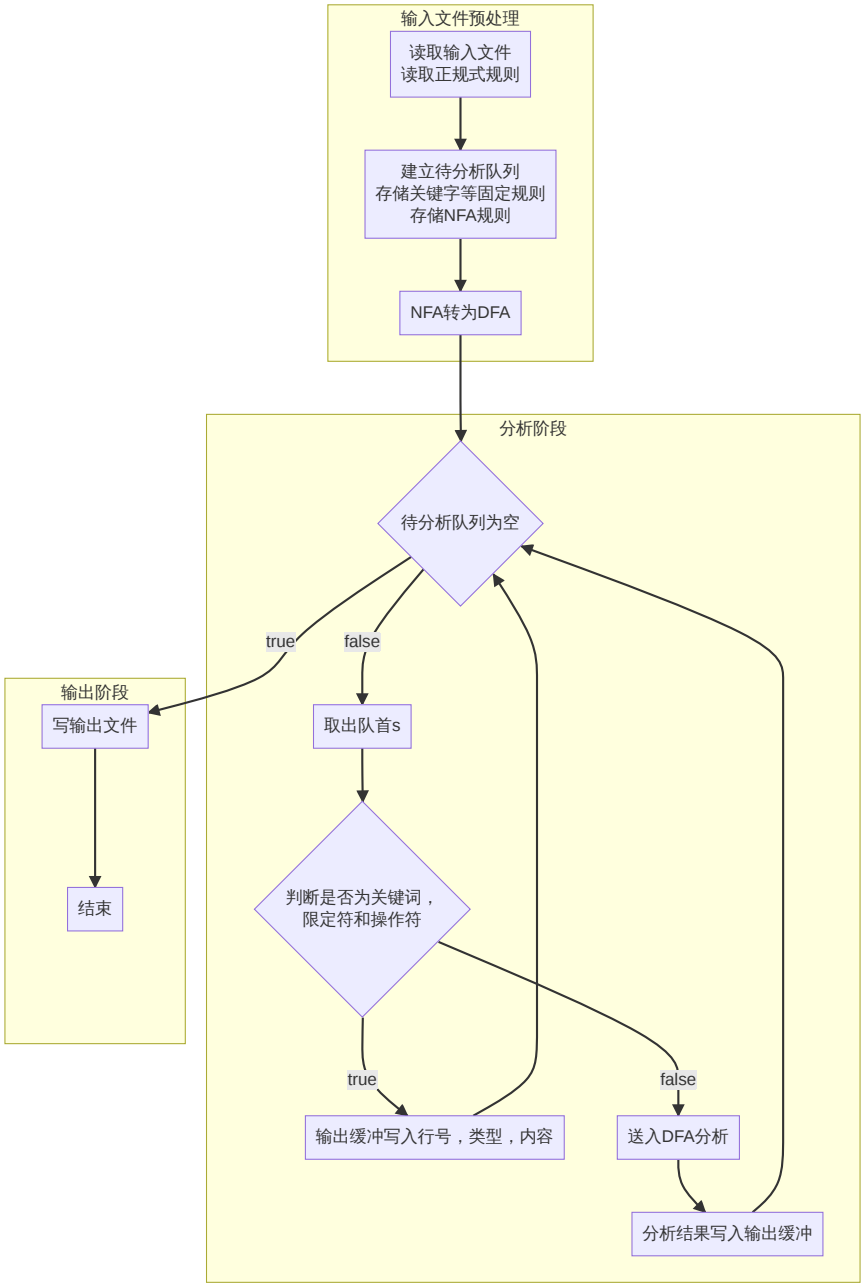
1.5 DFA和输入字符串的匹配

算法流程图如下



DFA的可接受状态为串中所有字符处理完毕，并且最终状态必须为可接受状态。其他情况（1. 查询不到转换状态 2. 最终状态不为可接受状态）均返回error

1.6 总流程图



2 代码函数解释

`def typeof(ch)`: 对字符进行分类，分类为d+,0,char,e/E,i类型（详见2.1.1节）
`def closure(NFA, depts)` 计算ε-closure，NFA：NFA语法，depts原始集合
`def NFA2DFA(NFA)` 计算NFA转换的DFA
`def Parse(DFA, s)` 通过NFA判断待分析串的类型

3 程序说明

词法分析的程序的可执行文件名称为`lex_parser.exe`，源代码文件为`lex_parser.py`

3.1 操作说明

在命令行输入命令 ./lex_parser -h获取帮助信息：

```
PS C:\Users\Anderson\Desktop\Gitworkspace\Compiler> ./lex_parser -h
usage:
lex parser by yuangyan
Courseworkwork for semester2, 2022
 [-h] [-s] [-i] [-r] [-o]

optional arguments:
  -h, --help      show this help message and exit
  -s, --showDFA   show DFA coverted from input NFA
  -i , --input    input file name, 'lex.txt' by default
  -r , --rule     rule file name, 'lex_rule.txt' by default
  -o , --output   output file name, '<input file name>_parsed.txt' by default
```

在词法分析程序中共有5个可选参数，分别为：

-h, --help: 显示帮助信息
-s, --showDFA: 在程序运行的过程中打印从输入NFA转换成的DFA
-i, --input: 选择输入待分析文本路径，不选该参数则默认为同目录下'lex.txt'文件
-r, --rule: 选择词法分析规则路径，不选该参数则默认为同目录下'lex_rule.txt'文件
-o, --output: 选择输出token表路径，不选该参数则默认为同目录下'<输入文件名>_parsed.txt'文件

输入命令格式为
./lex_parser (-i [待分析文本路径]) (-r [词法分析规则文件]) (-o [输出token路径]) (–showDFA)
例如：./lex_parser -i input.txt -o output.txt -r rule.txt --showDFA

3.2 输入输出样例

输入文件lex.txt:

> identifier123
> 123identifier
> $identifier
> 0.12
> .12
> 12
> 12.
> 12.00
> 1.2e+3
> 3i
> 12+3i
> 1.2+3.4e+5i
> 'YuangYan'
> 'YuangYan
> while(a<b){
> c[d] = e.getX()
> }

输出的token list，路径为lex_parsed.txt

```
1 IDENTIFIER identifier123
2 invalid_syntax 123identifier
3 invalid_syntax $identifier
4 DOUBLE 0.12
5 DOUBLE .12
6 INT 12
7 DOUBLE 12.
8 DOUBLE 12.00
9 DOUBLE 1.2e+3
10 COMPLEX 3i
11 COMPLEX 12+3i
12 COMPLEX 1.2+3.4e+5i
13 STRING 'YuangYan'
14 invalid_syntax 'YuangYan
15 KEYWORD while
15 DELIMITER (
15 IDENTIFIER a
15 BINARY_OPERATOR <
15 IDENTIFIER b
15 DELIMITER )
15 DELIMITER {
16 IDENTIFIER c
16 DELIMITER [
16 IDENTIFIER d
16 DELIMITER ]
16 ASSIGNMENT_OPERATOR =
16 IDENTIFIER e
16 DELIMITER .
16 IDENTIFIER getX
16 DELIMITER (
16 DELIMITER )
17 DELIMITER }
```

3.3 输入样例解释

在lex_parsed.txt文件中每行格式为[行号] [类型] [内容]
在设计的文法规则中
1） 标识符只能由字母开头，因此123identifier和$identifier被识别为invalid_syntax
2） 0.12，12.，12.00 12e+3均能识别为`double`类型
3） 复数的实部和虚部均可为`int`和`double`类型，兼容科学计数法，如1.2+3.4e+5i
4） 字符串常量由成对的单引号标识，'YuangYan缺少一个单引号，因此被识别为`nvalid_syntax`
5） 可以识别连在一起的不同类型，如while(a<b)被识别为关键字，限定符，标识符，赋值操作符，限定符

当加入`–showDFA`参数时，程序运行过程中会打印转换后的DFA，如下：

```
PS C:\Users\Anderson\Desktop\Gitworkspace\Compiler> ./lex_parser --showDFA
DFA:
0--d+-->1   (INT)
0--.-->2
```

```
0--e/E-->3   (IDENTIFIER)
0--i-->4   (COMPLEX)   (IDENTIFIER)
0--0-->5   (INT)
0--'-->6
0--char-->3   (IDENTIFIER)
6--d+-->6
6--e/E-->6
6--i-->6
6--0-->6
...
```

3.4 输入规则解释

在输入规则中，每行的第一列标识规则类型，分别为NFA，Keyword，AssignmentOperator，UnaryOperator，Delimiter和DFAType

1） 规则为`NFA`：

设NFA产生式为$f(A,t)=Z$，A在第二列，t在第三列，Z从第四列开始

2） 规则为`Keyword` `AssignmentOperator` `UnaryOperator` `Delimiter` `DFAType`均在第二列添加指定的字符串

3）规则为`DFAType`

用于表示DFA的可接收状态，在第二列添加指定的字符串，下图为部分lex_rule.txt内容

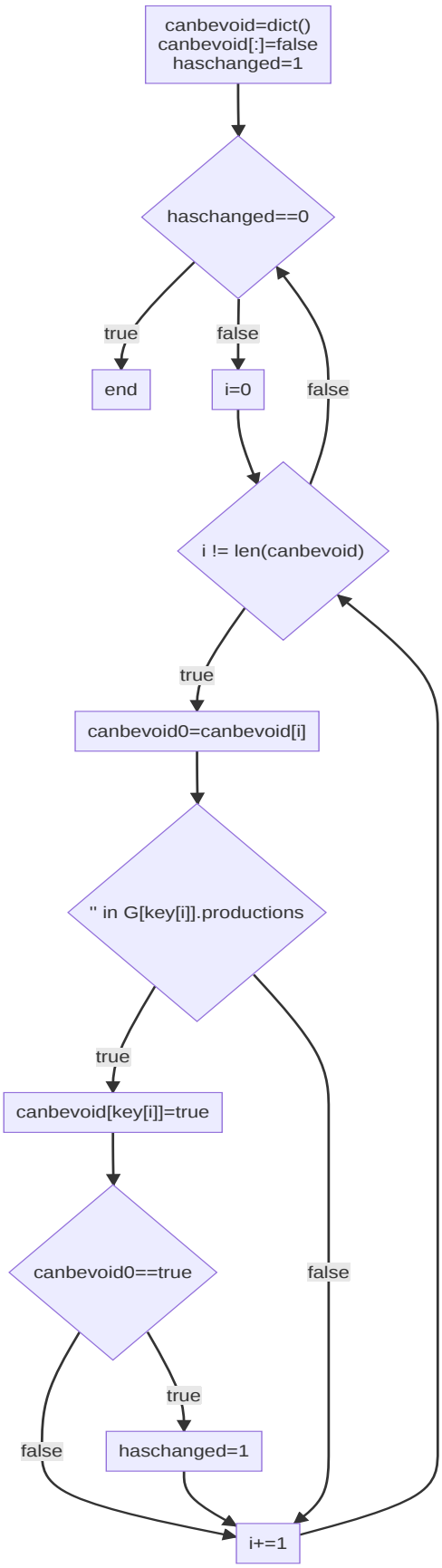
> NFA IDENTIFIER char IDENTIFIER
> Keyword while
> AssignmentOperator +=
> BinaryOperator and
> UnaryOperator !
> Delimiter .
> DFAType INT

任务二 文法分析

1 设计思路

1.1 判断非终结符是否能推出空

算法流程图如下



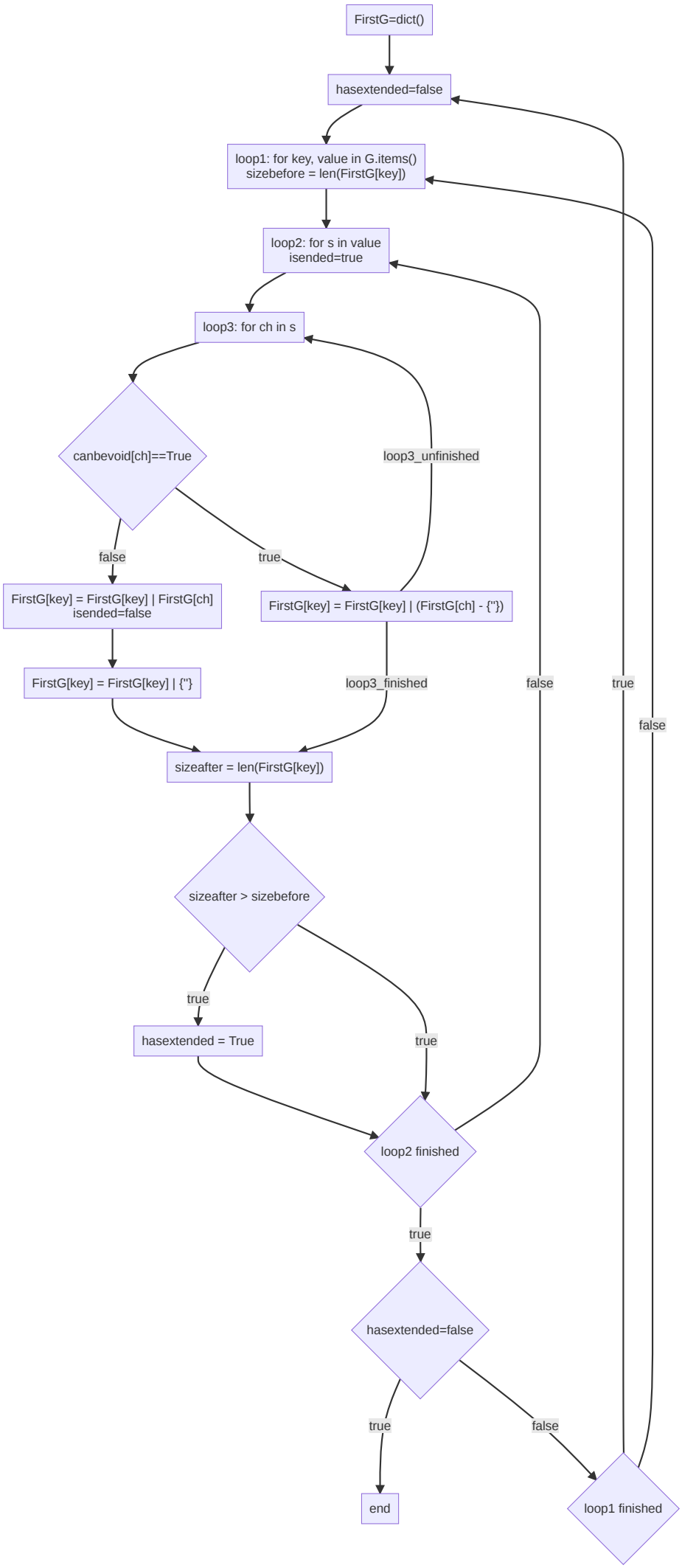
书中给出的判断是否能推出空产生式的算法有一些问题，可能会陷入死锁，故采用改进的判断空产生式算法。
与书中算法不一样的是，该算法将所有canbevoid全部置为false，即算法初始化将所有非终结符标为“不可产生空产生式”，之后循环遍历canbevoid数组，当非终结符产生式右侧能退出空时修改状态为true。当一轮循环后canbevoid数组没有变化则退出该循环，否则，则继续循环，直到canbevoid数组不再变化为止，这个状态量用haschanged保存。

1.2 计算first集

First集的计算使用python的数据结构set，set是一个不允许内容重复的组合，而且set里的内容位置是随意的数据结构

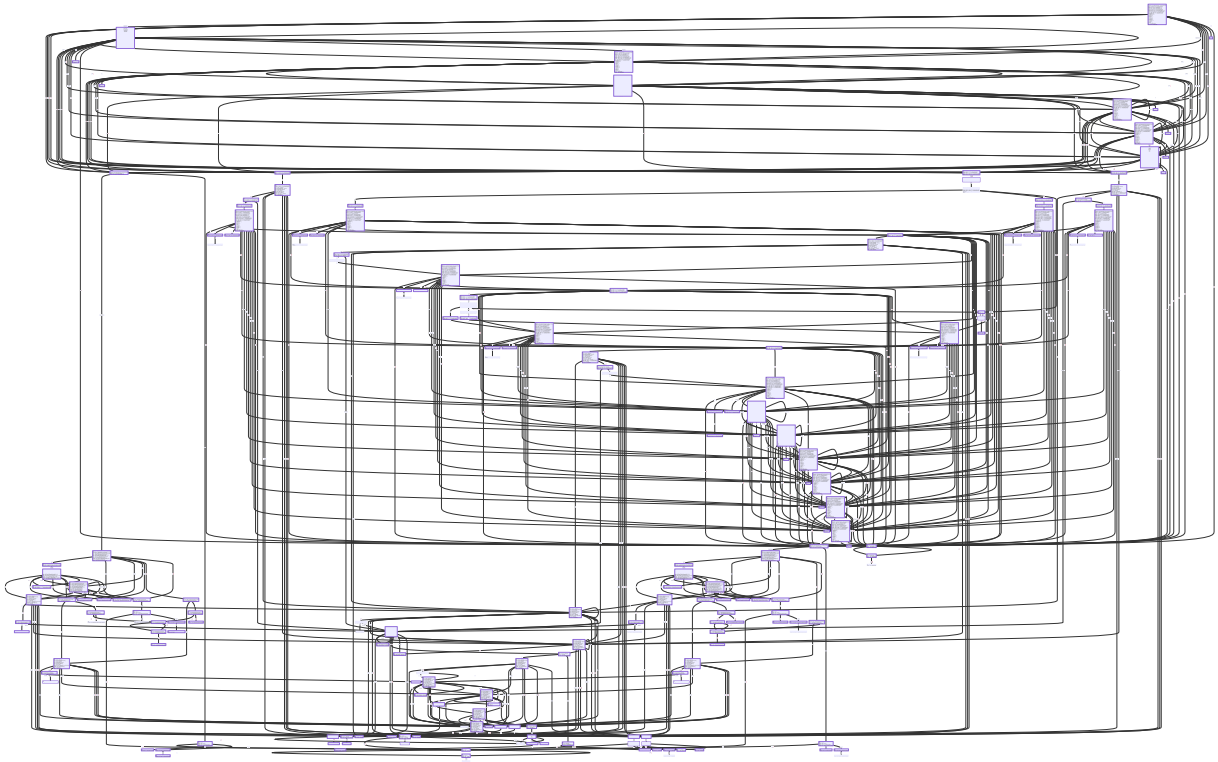
计算first集时要对右侧的产生式第一个符号进行分类.
1）当右侧的第一个符号不能推出空产生式，直接取原first集和右侧符号并集
2）当右侧第一个符号能推出空产生式，first集和右侧所有连续的VN且能能推出空产生式VN取并集
3）若右侧所有产生式均能推出空产生式，first集合加入”
4）否则first集并入右侧第一个不能推出空产生式的first集

算法流程图如下：

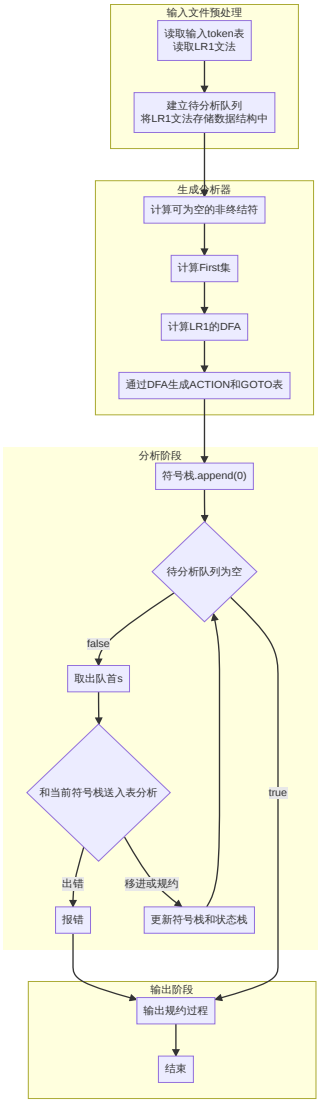

FirstG=dict()
hasextended=false
loop1: for key, value in G.items()
sizebefore = len(FirstG[key])
loop2: for s in value
isended=true
loop3: for ch in s
canbevoid[ch]==True
loop3_unfinished
false
true
FirstG[key] = FirstG[key] | FirstG[ch]
isended=false
FirstG[key] = FirstG[key] | (FirstG[ch] - {''})
FirstG[key] = FirstG[key] | {''}
loop3_finished
false
true
false
sizeafter = len(FirstG[key])
sizeafter > sizebefore
true
hasextended = True
true
loop2 finished
true
hasextended=false
true
false
end
loop1 finished

1.3 计算LR1的DFA

生成DFA的方式和词法分析中的方法类似。同样采用字典数据结构保存DFA，用DFS遍历转换边和状态，LR1_rule的转换表如图所示(可放大)



1.4 算法总流程图



2 代码函数解释

`def sortanddeduplicate(l)`：对于一个列表去重排序

`def checkcanbevoid(G)`：计算可以产生空产生式的VN

`def Firstset(G)`：计算First集

`def printerror(index)` 打印错误信息，index是分析队列中的位置

`def Formstat(rawstat)`：生成状态闭包

`def DFA(G)`：生成LR1转换表

`def LR1table(dfa: dict)`：通过DFA生成GOTO和ACTION表

`def LR1(s: list)`：进行LR1语法分析

3 程序说明

词法程序分析采用LR1文法分析。
程序的可执行文件名称为LR1_parser.exe，源代码文件为LR1_parser.py

3.1 操作说明

在命令行输入命令 ./LR1_parser -h获取帮助信息：

```
PS C:\Users\Anderson\Desktop\Gitworkspace\Compiler> ./LR1_parser -h
usage:
LR1 grammar parser by yuangyan
Courseworkwork for semester2, 2022
 [-h] [-s] [-i] [-r] [-o]

optional arguments:
  -h, --help         show this help message and exit
  -s, --showParsing  show Parsing Process
  -i , --input       input file name, 'lex_parsed.txt' by default
  -r , --rule        rule file name, 'grammar_rule.txt' by default
  -o , --output      output file name, '<input file name>_LR1.txt' by default
```

在词法分析程序中共有5个可选参数，分别为：

-h, --help: 显示帮助信息
-s, --showParsing: 在程序运行的过程中打印分析过程
-i, --input: 选择输入待分析文本路径，不选该参数则默认为同目录下'lex_parsed.txt'文件
-r, --rule: 选择词法分析规则路径，不选该参数则默认为同目录下'grammar_rule.txt'文件
-o, --output: 选择输出路径，不选该参数则默认为同目录下'<输入文件名>_LR1.txt'文件

输入命令格式为

./LR1_parser (-i [待分析文本路径]) (-r [LR1文法分析规则文件]) (-o [输出路径]) (–showParsing)
例如：./LR1_parser -i input.txt -o output.txt -r rule.txt –showParsing

3.2 输入输出样例

输入样例：lex.txt

```
while(a=b){
  a+=1
}
```

经过词法分析后：lex_parsed.txt

```
1 KEYWORD while
1 DELIMITER (
1 IDENTIFIER a
1 ASSIGNMENT_OPERATOR =
1 IDENTIFIER b
1 DELIMITER )
1 DELIMITER {
2 IDENTIFIER a
2 ASSIGNMENT_OPERATOR +=
2 INT 1
3 DELIMITER }
```

将lex_parsed送入分析后的输出

```
PS C:\Users\Anderson\Desktop\Gitworkspace\Compiler> ./LR1_parser
[ERROR] in line 3: while(a=b){
                           ^
invalid syntax: =
```

可见报错。
原来，在文法中`while`里面的语句只能是一个“表达式”，而不能是一个“赋值语句”，所以在这里报错，将a=b修改为a<b后再次输入，并且加入选项`–showParsing`，显示分析过程

输出结果：

```
PS C:\Users\Anderson\Desktop\Gitworkspace\Compiler> ./LR1_parser --showParsing
([0], ['#'], ['while', '(', 'id', 'BinaryOp', 'id', ')', '{', 'id',
'AssignmentOp', 'const', '}', '#'], 'S4')
([0, 4], ['#', 'while'], ['(', 'id', 'BinaryOp', 'id', ')', '{', 'id',
'AssignmentOp', 'const', '}', '#'], 'S21')
([0, 4, 21], ['#', 'while', '('], ['id', 'BinaryOp', 'id', ')', '{', 'id',
'AssignmentOp', 'const', '}', '#'], 'S27')
([0, 4, 21, 27], ['#', 'while', '(', 'id'], ['BinaryOp', 'id', ')', '{', 'id',
'AssignmentOp', 'const', '}', '#'], 'r: <ids> → <id>')
([0, 4, 21, 28], ['#', 'while', '(', 'ids'], ['BinaryOp', 'id', ')', '{', 'id',
'AssignmentOp', 'const', '}', '#'], 'r: <Expr> → <ids>')
([0, 4, 21, 23], ['#', 'while', '(', 'Expr'], ['BinaryOp', 'id', ')', '{', 'id',
'AssignmentOp', 'const', '}', '#'], 'S66')
([0, 4, 21, 23, 66], ['#', 'while', '(', 'Expr', 'BinaryOp'], ['id', ')', '{',
'id', 'AssignmentOp', 'const', '}', '#'], 'S68')
([0, 4, 21, 23, 66, 68], ['#', 'while', '(', 'Expr', 'BinaryOp', 'id'], [')',
'{', 'id', 'AssignmentOp', 'const', '}', '#'], 'r: <ids> → <id>')
([0, 4, 21, 23, 66, 28], ['#', 'while', '(', 'Expr', 'BinaryOp', 'ids'], [')',
'{', 'id', 'AssignmentOp', 'const', '}', '#'], 'r: <Expr> → <ids>')
```

```
([0, 4, 21, 23, 66, 67], ['#', 'while', '(', 'Expr', 'BinaryOp', 'Expr'], [')',
'{', 'id', 'AssignmentOp', 'const', '}', '#'], 'r: <Expr> → <Expr><BinaryOp>
<Expr>')
([0, 4, 21, 23], ['#', 'while', '(', 'Expr'], [')', '{', 'id', 'AssignmentOp',
'const', '}', '#'], 'S72')
([0, 4, 21, 23, 72], ['#', 'while', '(', 'Expr', ')'], ['{', 'id',
'AssignmentOp', 'const', '}', '#'], 'S73')
([0, 4, 21, 23, 72, 73], ['#', 'while', '(', 'Expr', ')', '{'], ['id',
'AssignmentOp', 'const', '}', '#'], 'S12')
([0, 4, 21, 23, 72, 73, 12], ['#', 'while', '(', 'Expr', ')', '{', 'id'],
['AssignmentOp', 'const', '}', '#'], 'r: <ids> → <id>')
([0, 4, 21, 23, 72, 73, 74], ['#', 'while', '(', 'Expr', ')', '{', 'ids'],
['AssignmentOp', 'const', '}', '#'], 'S123')
([0, 4, 21, 23, 72, 73, 74, 123], ['#', 'while', '(', 'Expr', ')', '{', 'ids',
'AssignmentOp'], ['const', '}', '#'], 'S132')
([0, 4, 21, 23, 72, 73, 74, 123, 132], ['#', 'while', '(', 'Expr', ')', '{',
'ids', 'AssignmentOp', 'const'], ['}', '#'], 'r: <Expr> → <const>')
([0, 4, 21, 23, 72, 73, 74, 123, 128], ['#', 'while', '(', 'Expr', ')', '{',
'ids', 'AssignmentOp', 'Expr'], ['}', '#'], 'r: <Assignment> → <ids>
<AssignmentOp><Expr>')
([0, 4, 21, 23, 72, 73, 80], ['#', 'while', '(', 'Expr', ')', '{', 'Assignment'],
['}', '#'], 'r: <S> → <Assignment>')
([0, 4, 21, 23, 72, 73, 78], ['#', 'while', '(', 'Expr', ')', '{', 'S'], ['}',
'#'], 'S92')
([0, 4, 21, 23, 72, 73, 78, 92], ['#', 'while', '(', 'Expr', ')', '{', 'S', '}'],
['#'], 'r: <Loop> → <while><(><Expr><)><{><S><}>')
([0, 10], ['#', 'Loop'], ['#'], 'r: <S> → <Loop>')
([0, 11], ['#', 'S'], ['#'], "r: S'→S")
accepted
```

语句被accepted，并且分析过程中显示了符号栈，已分析栈，待输入串和规约语法
在输出的lex_parsed_LR1.txt中记录了上述过程.

附录1 词法生成规则

lex_rule.txt

```
NFA S ε A REAL+
NFA S . C
NFA S char IDENTIFIER
NFA S e/E IDENTIFIER
NFA S i IDENTIFIER
NFA S ' K
NFA K i K
NFA K d+ K
NFA K e/E K
NFA K char K
NFA K 0 K
NFA K ε K
NFA K ' STRING
NFA IDENTIFIER char IDENTIFIER
NFA IDENTIFIER i IDENTIFIER
NFA IDENTIFIER e/E IDENTIFIER
NFA IDENTIFIER d+ IDENTIFIER
NFA IDENTIFIER 0 IDENTIFIER
NFA A 0 INT
NFA A d+ B
NFA B d+ B
NFA B 0 B
NFA B ε INT
NFA C 0 DOUBLE
NFA C d+ DOUBLE
NFA INT . DOUBLE
NFA DOUBLE d+ DOUBLE
NFA DOUBLE 0 DOUBLE
NFA INT e/E D
NFA DOUBLE e/E D
NFA D +/- F
NFA D ε F
NFA F d+ DOUBLE
NFA F 0 DOUBLE
NFA INT +/- REAL+
NFA DOUBLE +/- REAL+
NFA REAL+ 0 I_INT
NFA REAL+ d+ H
NFA REAL+ . G
NFA REAL+ i COMPLEX
NFA H 0 H
NFA H d+ H
NFA H ε I_INT
NFA I_INT . I_DOUBLE
```

```
NFA G d+ I_DOUBLE
NFA G 0 I_DOUBLE
NFA I_DOUBLE d+ I_DOUBLE
NFA I_DOUBLE 0 I_DOUBLE
NFA I_INT e/E I
NFA I_DOUBLE e/E I
NFA I +/- J
NFA I ε J
NFA J d+ I_DOUBLE
NFA J 0 I_DOUBLE
NFA I_INT i COMPLEX
NFA I_DOUBLE i COMPLEX
Keyword for
Keyword while
Keyword if
Keyword else
Keyword return
Keyword break
Keyword continue
Keyword def
Keyword class
Keyword int
Keyword double
Keyword dict
Keyword list
Keyword tuple
AssignmentOperator +=
AssignmentOperator -=
AssignmentOperator *=
AssignmentOperator /=
AssignmentOperator //=
AssignmentOperator %=
AssignmentOperator =
BinaryOperator and
BinaryOperator or
BinaryOperator xor
BinaryOperator ==
BinaryOperator **
BinaryOperator +
BinaryOperator -
BinaryOperator *
BinaryOperator /
BinaryOperator //
BinaryOperator %
BinaryOperator in
BinaryOperator <
BinaryOperator >
BinaryOperator <=
BinaryOperator >=
UnaryOperator !
```

UnaryOperator not
Delimiter .
Delimiter :
Delimiter (
Delimiter)
Delimiter {
Delimiter }
Delimiter [
Delimiter]
Delimiter ,
DFAType INT
DFAType DOUBLE
DFAType COMPLEX
DFAType IDENTIFIER
DFAType STRING
Expressionrule S identifier A
Expressionrule S keyword A
Expressionrule S assignment_operator Expr1
Expressionrule S binary_operator Expr1
Expressionrule S unary_operator Expr1
Expressionrule S delimiter Expr1
Expressionrule S ε Expr1
Expressionrule A assignment_operator Expr1
Expressionrule A binary_operator Expr1
Expressionrule A unary_operator Expr1
Expressionrule A delimiter Expr1
Expressionrule Expr1 datatype Expr2
Expressionrule Expr1 assignment_operator Expr1
Expressionrule Expr1 binary_operator Expr1
Expressionrule Expr1 unary_operator Expr1
Expressionrule Expr1 delimiter Expr1
Expressionrule Expr1 identifier Expr2
Expressionrule Expr2 ε A

附录2 LR1生成规则

grammar_rule.txt

```
G S' S
G S  Assignment   S
G S  Assignment
G S  Funcdef  S
G S  Funcdef
G S  Func   S
G S  Func
G S  Loop   S
G S  Loop
G S  Branch   S
G S  Branch
G   S   Funcall S
G   S   Funcall
G Expr   Expr   BinaryOp   Expr
G Expr   UnaryOp   Expr
G Expr   (   Expr   )
G Expr   const
G Expr   ids
G Expr   Funcall
G Expr   id  [ Expr ]
G Assignment   ids   AssignmentOp  Expr
G   ids    id  .   ids
G   ids    id
G   Funcall ids ( )
G   Funcall ids ( Consts )
G   Consts const
G   Consts const , Consts
G   Paras id , Paras
G   Paras id
G   Funcdef def id ( ) { S }
G   Funcdef def id ( Paras ) { S }
G   Funcdef def id ( Paras ) { }
G   Funcdef def id (  ) { }
G   Loop while ( Expr ) { S }
G   Loop while ( Expr ) { }
G   Branch if ( Expr ) { S }
G   Branch if ( Expr ) { }
isVN   S'
isVN   S
isVN   Expr
isVN   Assignment
isVN   Funcdef
isVN   Loop
isVN   Branch
```

isVN Paras
isVN Consts
isVN Funcall
isVN ids
filter DOUBLE const
filter INT const
filter COMPLEX const
filter STRING const
filter IDENTIFIER id
filter BINARY_OPERATOR BinaryOp
filter UNARY_OPERATOR UnaryOp
filter ASSIGNMENT_OPERATOR AssignmentOp