

编译原理实验报告

实验名称： 词法分析程序的设计与实现
实验类型： 设计性实验
指导教师： 付永钢
专业班级： 计算 2112
姓 名： 蔡俊志
学 号： 202121331040
电子邮件： chickenbilibili@outlook.com
实验地点： 陆大 0206
实验成绩：

日期： 2023 年 11 月 10 日

一、实验目的

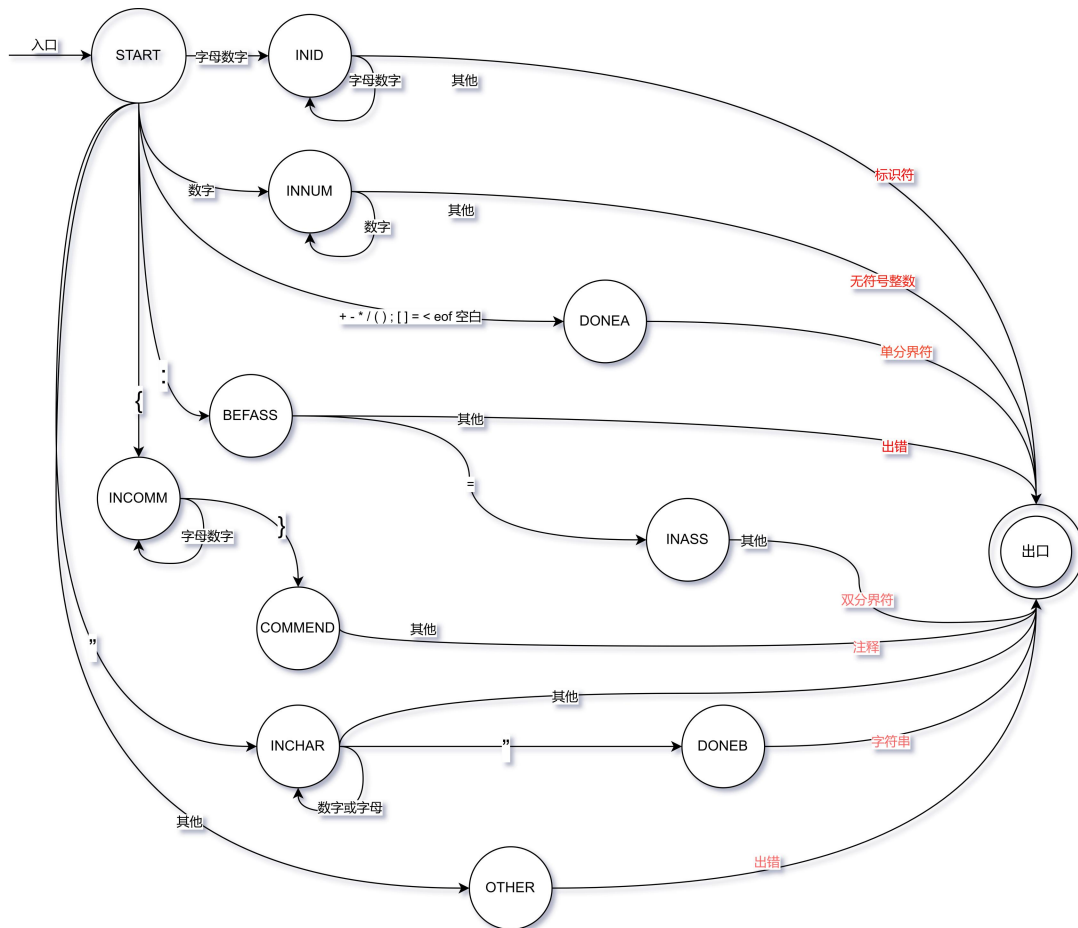
1. 深入理解有限自动机及其应用
2. 掌握词法分析程序的开发。
3. 掌握根据语言的词法规则构造识别其单词的有限自动机的方法
4. 深入理解词法分析程序自动生成原理

二、实验过程

第一部分：构造识别单词的自动机

为这是我在实验说明文件提供的 DFA（下面简称原 DFA）的基础上进行微调和修改的新版本，我之后的代码将会对这个改良后的 DFA 进行实现。改动的地方有：

1. 字符串的识别、读取部分原由三个状态来表示，现精简为 2 个。
2. 对一些没有命名的状态进行了命名，以便更方便的编写代码。比如这里的 DONEA，BEFCOMM（Before comment）等。



第二部分：用直接转向法实现有限自动机的代码，生成对应的词法分析程序

代码实现部分，我使用 Python 的类来描述上述图片的 DFA，在类中定义状态转换函数从而实现直接转向法，下面是我对代码更多细节的描述部分：

1. 主函数

主函数负责以下逻辑：对文件进行读取，存放 DFA 类的实例，以及把文件中的字符一个一个送进 DFA 实例中。**伪代码**如下：

```
dfa = DFA()
for symbol in "code.txt":
    dfa.run(symbol)
    ...
```

当然实际上 main 函数需要做的不止伪代码展示的这么简单，伪代码展示的是 main 函数的核心，也就是将字符整理好送进 DFA 的逻辑。除此之外，main 函数还要负责对 DFA 的整体控制等。Python 代码如下：

```
start_state = 'START'
accept_state = 'ACCEPT'
dfa = DFA(start_state, accept_state)
i = 0

# Open file
with open("code1.txt", "r") as file:
    input_string = file.read()

while i < len(input_string):

    # 往 DFA 里输入一个字符
    dfa.run(input_string[i])

    # 如果 DFA 目前状态已经 accept
    if dfa.current_state == accept_state:
        dfa.conclude()
        # conclude()用于将当前 DFA 的 Buffer 整合成 (30, "end")并输出
        dfa.refresh()
        # refresh()用于清空 DFA 的 Buffer, 重置各种属性位
    else:
        i += 1
    if i == len(input_string):
```

```
dfa.conclude()
```

2. 如何设计 DFA 类的属性部分

为了保存 DFA 运行时的各种变量，我们需要合理设计 DFA 类的属性部分。需要保存的变量如下：

- a. 初始状态（需要在实例 DFA 时传入）
- b. 接收状态（需要在实例 DFA 时传入）
- c. 当前状态（DFA 运行时的内部状态）
- d. 接受状态提示（这对应着第一部分的 DFA 图中的红字部分，如：以字符串接受 DFA 就可以利用这个变量来显式提示当前单词类型，从而加快处理速度）
- e. 字符缓冲（这是一个 list，用于在 DFA 尚未接受时存放输入的历史字符）
- f. 转换字典（引用并指示每一个状态需要调用的函数）

类的初始化 Python 代码如下：

```
def __init__(self, start_state, accept_states):
    self.transitions = {
        'START': self.state_start,
        'INID': self.state_inid,
        'INNUM': self.state_innum,
        'BEFASS': self.state_befass,
        'INCOMM': self.state_incomm,
        'COMMEND': self.state_commend,
        'INCHAR': self.state_inchar,
        'OTHER': self.state_other,
        'DONEA': self.state_donea,
        'DONEB': self.state_doneb,
        'INASS': self.state_inass,
    }
    self.start_state = start_state
    self.current_state = start_state
    self.accept_states = accept_states
    self.buffer = []
    self.hint = ""
```

值得一提的是，hint 是存放到一个字典里面的。这个字典长这样：

```
HINT_DICT = {
```

```
"ID": 1,  
"INT": 2,  
"ERROR": 100,  
"STRING": 31,  
"COMMENT": 32,  
}
```

3. 如何设计 DFA 类的方法部分

我设计的 DFA 类中的函数有两类。一类是用于实现直接转向法的函数(也可以称作状态转换函数)它们是 DFA 的最基本函数。另一类函数用于控制 DFA 的输入, 输出, 管理一些属性等等, 在接下来的两小节我会介绍他们。

下面展示了 DFA() 中的所有函数的列表以及它们功能的简述

```
run(self, symbol_) #将字符输入 DFA 的窗口  
conclude(self) #DFA 进入 accept 状态之后, 对 buffer 的内容赋予 token 并输出到控制台  
refresh(self) #DFA 进入 accept 状态之后, 对其所有属性重置, 迎接下一个词  
set_current_state(self, state) # 设置当前状态  
lookup_for_token(self, string_in) # 从单词 token 表里面找 token  
  
# 下面这些全部都是每个状态对应的处理函数, 返回下一个状态  
state_start(self, symbol_in)  
state_inid(self, symbol_in)  
state_innum(self, symbol_in)  
state_inchar(self, symbol_in)  
state_donea(self, symbol_in)  
state_doneb(self, symbol_in)  
state_befass(self, symbol_in)  
state_other(self, symbol_in)  
state_incomm(self, symbol_in)  
state_inass(self, symbol_in)  
state_commend(self, symbol_in)
```

4. DFA 类的状态转换函数

这个部分的函数可以简单总结为一句话: 根据当前输入字符和当前状态。返回 DFA 应当转换的下一个状态。故此时使类似 **switch case** 语句将会比较方便(虽然 Python 里没有)。下面是一个状态转换函数的伪代码

```
func state(symbol_in):
```

```
buffer += symbol_in
switch symbol_in:
    case ... :
        return "State1"
    case ... :
        return "State2"
    case ... :
        return "State3"
    case ... :
        return "State4"
    ...
```

下面是 Python 的代码, 这里以 INCHAR 的状态函数为例。

```
def state_inchar(self, symbol_in):
    if symbol_in.isalnum():
        # 如果是一个数字或者字符
        self.buffer.append(symbol_in)
        return 'INCHAR'
        # 状态不变
    elif symbol_in == "\":
        # 如果是一个引号
        self.buffer.append(symbol_in)
        return 'DONEB'
        # 进入下一步状态
    else:
        # 其他字符
        self.hint = "ERROR"
        return 'ACCEPT'
        # 结束 DFA 并且把 hint 文本设置为 ERROR
```

5. DFA 类的控制函数

这里展示一下 DFA 的输入管理函数的实现。也就是 DFA 处理每一个字符输入的窗口。
Python 代码如下：

```
def run(self, symbol_):
    if self.current_state == self.accept_states:
        # 如果 DFA accept 那么直接返回
        return

    target_state = self.transitions[self.current_state](symbol_)
    # 在 Transition 字典中找到当前状态对应的状态函数, 然后用执行这个函数返回的值更新当前状态
    self.current_state = target_state
```

这里再展示一下当 DFA accept 的时候的输出函数, Python 代码如下:

```
def conclude(self):
    string = ''.join(self.buffer)
    # 把 Buffer 里面的字符拼成一个字符串
    if string not in [" ", "\n", "\t"]:
        # 忽略空格回车和制表符
        token = self.lookup_for_token(string)
        # 从 hint 字典和单词字典中找 token
        print(f'({token}, "{string}")')
    # 格式输出
```

第三部分: 用 flex 生成上述给定 DFA 所对应的 PL0 语言的词法分析程序

flex 是一个强大的词法分析生成程序, 我会手动指定识别到保留字段时的输出形式。它的语法类似于需求描述语言, 准确描述了词法分析的行为。代码如下:

```
%{
# include <stdio.h>
%}

%%

"+"      { printf("(3: +)\n"); }
"-"      { printf("(4: -)\n"); }
"*"      { printf("(5: *)\n"); }
"/"      { printf("(6: /)\n"); }
"="      { printf("(7: =)\n"); }
">"      { printf("(8: >)\n"); }
```

```

"<"      { printf("(9: <)\n"); }
"<>"     { printf("(10: <>)\n");}
"<="     { printf("(11: <=)\n");}

""
int main(){
    yylex();
    return 0;
}

int yywrap(){
    return 1;
}

```

三、实验结果

实验测试数据：

测试了四份 PLO 代码，其中一份有错误

Code1.txt

```

procedure divide;
var w;
begin
    r := x; q := 0; w := y;
end

```

Code2.txt

```

const m = 7, n = 85;
var x, y, z, q, r;

procedure multiply;
var a, b;
begin
    a := x; b := y; z := 0;
    while b > 0 do
    begin
        if odd b then z := z + a;
        a := 2 * a; b := b / 2;
    end
end;

```

Code3.txt

```

procedure divide;

```



```

var w;

begin

    r := x; q := 0; w := y;

    while w > y do

        begin

            q := 2 * q; w := w / 2;

            if w <= r then

                begin

                    r := r - w;

                    q := q + 1;

                end;

            end

        end;

end;

```

Code1(contain ERROR).txt

```

procedure divide;

var w;

begin

    r := x; q := 0; w := y;

end

```

对于 **code1.txt** 的测试

使用直接转向法编写的

python 代码

```

Run main x
C:\Users\chicken\anaconda3\python.exe C:\Users\chicken\PycharmProjects\Lex\main.py
(29, "procedure")
(1, "divide")
(17, ";")
(21, "var")
(1, "w")
(17, ";")
(27, "begin")
(1, "r")
(20, ":=")
(1, "x")
(17, ";")
(1, "q")
(20, ":=")
(2, "0")
(17, ";")
(1, "w")
(20, ":=")
(1, "y")
(17, ";")
(30, "end")
Process finished with exit code 0

```

使用 flex 自动生成的代码

```

C:\Users\chicken\Desktop\Co x + v
(29, "procedure")
(1, "divide")
(17, ";")
(21, "var")
(1, "w")
(17, ";")
(27, "begin")
(1, "r")
(20, ":=")
(1, "x")
(17, ";")
(1, "q")
(20, ":=")
(2, "0")
(17, ";")
(1, "w")
(20, ":=")
(1, "y")
(17, ";")
(30, "end")

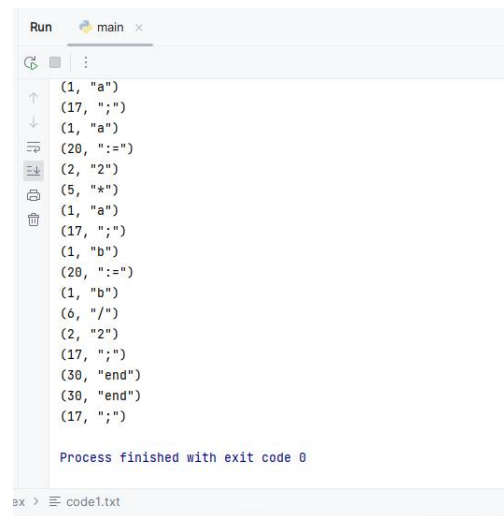
-----
Process exited after 0.01485 seconds with return value 0 (0 ms cpu time)
Press ANY key to exit...|

```

对于 **code2.txt** 的测试

使用直接转向法编写的

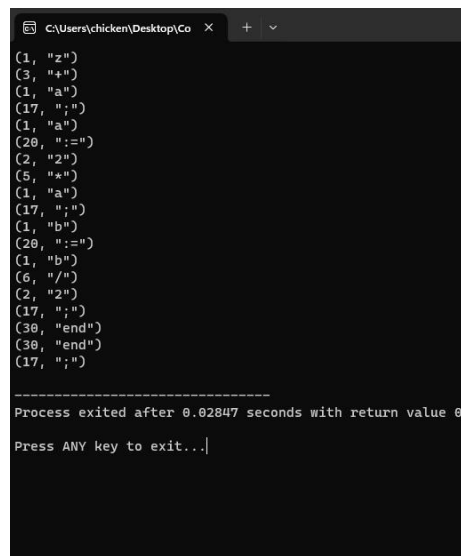
python 代码



```
Run main x
(1, "a")
(17, ";")
(1, "a")
(20, ":=")
(2, "2")
(5, "*")
(1, "a")
(17, ";")
(1, "b")
(20, ":=")
(1, "b")
(6, "/")
(2, "2")
(17, ";")
(30, "end")
(30, "end")
(17, ";")

Process finished with exit code 0
```

使用 flex 自动生成的代码

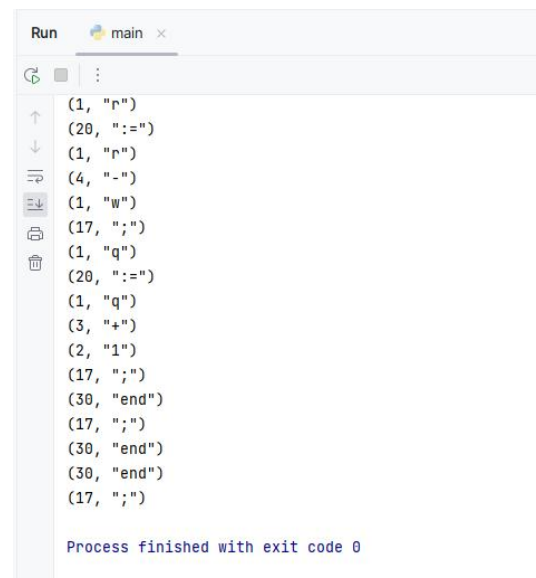


```
C:\Users\chicken\Desktop\Co x + v
(1, "a")
(17, ";")
(1, "a")
(20, ":=")
(2, "2")
(5, "*")
(1, "a")
(17, ";")
(1, "b")
(20, ":=")
(1, "b")
(6, "/")
(2, "2")
(17, ";")
(30, "end")
(30, "end")
(17, ";")

Process exited after 0.02847 seconds with return value 0
Press ANY key to exit...
```

对于 code3.txt 的测试

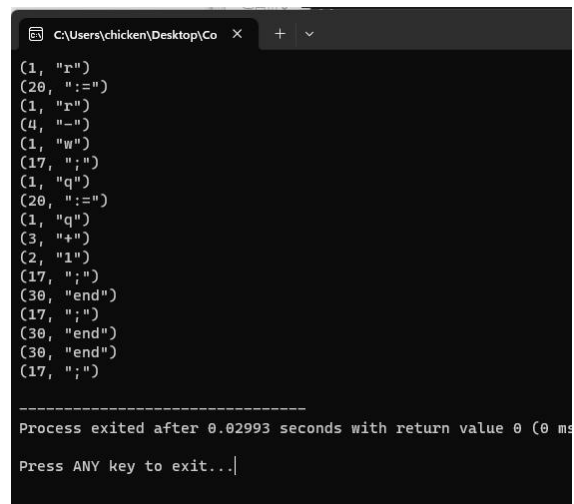
Python 代码



```
Run main x
(1, "r")
(20, ":=")
(1, "r")
(4, "-")
(1, "w")
(17, ";")
(1, "q")
(20, ":=")
(1, "q")
(3, "+")
(2, "1")
(17, ";")
(30, "end")
(17, ";")
(30, "end")
(30, "end")
(17, ";")

Process finished with exit code 0
```

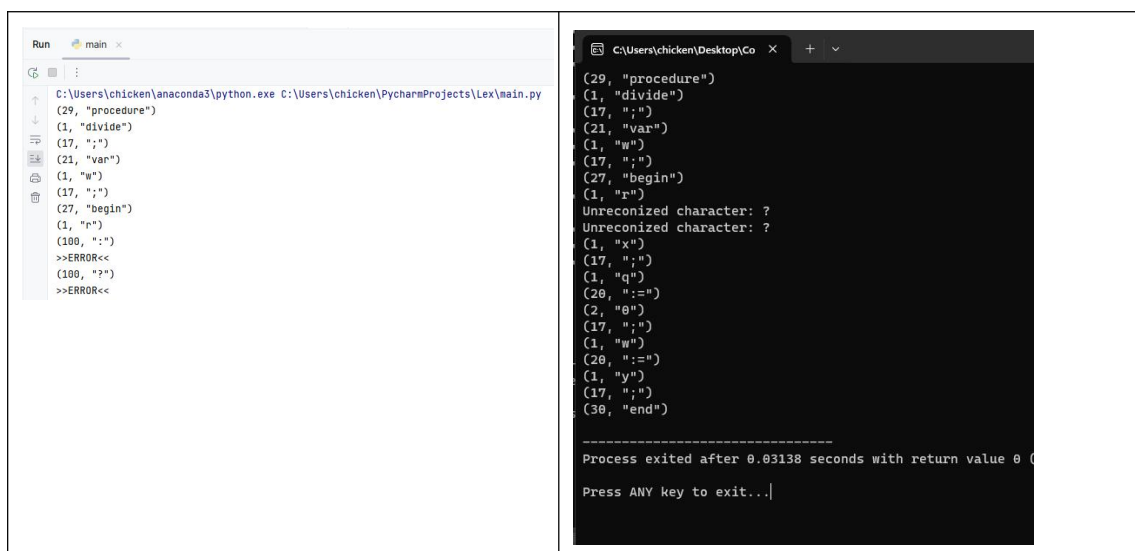
使用 flex 自动生成的代码



```
C:\Users\chicken\Desktop\Co x + v
(1, "r")
(20, ":=")
(1, "r")
(4, "-")
(1, "w")
(17, ";")
(1, "q")
(20, ":=")
(1, "q")
(3, "+")
(2, "1")
(17, ";")
(30, "end")
(17, ";")
(30, "end")
(30, "end")
(17, ";")

Process exited after 0.02993 seconds with return value 0 (0 ms)
Press ANY key to exit...
```

对于 Code1(contain ERROR).txt 的测试



四、讨论与分析

根据结果对比，我们可以观察到以下几个方面的差异。首先，对于正常的程序，直接转向法和 Flex 自动生成的代码在结果输出上是一致的，即直接转向法所生成的代码能够完整地识别和解析正常的程序。

然而，在处理错误的程序时，直接转向法的输出结果仅提供了错误发生的行数，而没有给出具体的错误信息。而 Flex 生成的代码则具备更为健全的错误处理机制，能够在发生错误后继续对字符进行识别，直到程序结束，并提供详细的错误原因。因此，就诊断错误和处理错误的能力而言，Flex 生成的代码更为出色。

最终结论是，直接转向法生成的代码的处理逻辑是正确的，但对于错误信息的定义相对简略。此外，为了提高程序的检测效率，在发生错误后需要设计一个恢复机制也许会更好，使得程序能够在检测到错误后不受其影响，继续识别其他正确的字符流或继续检测其他可能存在的错误，但由于时间紧张，就没有做尝试。

对于这个实验的感悟，详见实验者自评

五、附录：

Github Repository Link: [点我](#)

```
dfa.py

from dict import TOKEN_DICT, HINT_DICT
class DFA:
    def __init__(self, start_state, accept_states):
        self.transitions = {
            'START': self.state_start,
            'INID': self.state_inid,
            'INNUM': self.state_innum,
```

```

        'BEFASS': self.state_befass,
        'INCOMM': self.state_incomm,
        'COMMEND': self.state_commend,
        'INCHAR': self.state_inchar,
        'OTHER': self.state_other,
        'DONEA': self.state_donea,
        'DONEB': self.state_doneb,
        'INASS': self.state_inass,
    }

    self.start_state = start_state
    self.current_state = start_state
    self.accept_states = accept_states
    self.buffer = []
    self.hint = ""

def run(self, symbol_):
    if self.current_state == self.accept_states:
        # 如果 DFA accept 那么直接返回
        return

    target_state = self.transitions[self.current_state](symbol_)
    # 在 Transition 字典中找到当前状态对应的状态函数, 然后更新当前状态
    self.current_state = target_state

def conclude(self):
    string = ''.join(self.buffer)
    # 把 Buffer 里面的字符拼成一个字符串
    if string not in [" ", "\n", "\t"]:
        # 忽略空格回车和制表符
        token = self.lookup_for_token(string)
        # 从 hint 字典和单词字典中找 token
        print(f'({token}, "{string}")')
        # 格式输出
        if token == 100:
            print('>>ERROR<<')

def refresh(self):
    self.buffer.clear()
    self.current_state = 'START'

```

```

self.hint = ""

def set_current_state(self, state):
    self.current_state = state

def lookup_for_token(self, string_in):
    if self.hint:
        if self.hint == "ID" and string_in in TOKEN_DICT:
            return TOKEN_DICT[string_in]
        else:
            return HINT_DICT[self.hint]
    else:
        return TOKEN_DICT[string_in]

# 以下都是 STATE METHOD
# 实际意义是 DFA 的跳转逻辑

def state_start(self, symbol_in):

    self.buffer.append(symbol_in)
    if symbol_in.isalpha():
        return 'INID'
    elif symbol_in.isdigit():
        return 'INNUM'
    elif symbol_in in ["+", "-", "*", "/", "(", ")", ";", "[", "]", "="]:
        return 'DONEA'
    elif symbol_in == ":":
        return 'BEFASS'
    elif symbol_in == "{":
        return 'INCOMM'
    elif symbol_in == "\'":
        return 'INCHAR'
    else:
        return 'OTHER'

def state_inid(self, symbol_in):
    self.hint = "ID"
    if symbol_in.isalnum():

```

```

        self.buffer.append(symbol_in)
        return 'INID'
    else:
        return 'ACCEPT'

def state_innum(self, symbol_in):
    self.hint = "INT"
    if symbol_in.isdigit():
        self.buffer.append(symbol_in)
        return 'INNUM'
    else:
        return 'ACCEPT'

def state_inchar(self, symbol_in):
    if symbol_in.isalnum():
        # 如果是一个数字或者字符
        self.buffer.append(symbol_in)
        return 'INCHAR'
        # 状态不变
    elif symbol_in == "\"":
        # 如果是一个引号
        self.buffer.append(symbol_in)
        return 'DONEB'
        # 进入下一步状态
    else:
        # 其他字符
        self.hint = "ERROR"
        return 'ACCEPT'
        # 结束 DFA 并且把 hint 文本设置为 ERROR

def state_donea(self, symbol_in):
    return 'ACCEPT'

def state_doneb(self, symbol_in):
    self.hint = "STRING"
    return 'ACCEPT'

def state_befass(self, symbol_in):

```

```

    if symbol_in == "=":
        self.buffer.append(symbol_in)
        return 'INASS'
    else:
        self.hint = "ERROR"
        return 'ACCEPT'

def state_other(self, symbol_in):
    self.hint = "ERROR"
    return 'ACCEPT'

def state_incomm(self, symbol_in):
    if symbol_in.isalnum():
        self.buffer.append(symbol_in)
        return 'INCOMM'
    elif symbol_in == "}":
        self.buffer.append(symbol_in)
        return 'COMMEND'

def state_inass(self, symbol_in):
    return 'ACCEPT'

def state_commend(self, symbol_in):
    self.hint = "COMMENT"
    return 'ACCEPT'

```

dict.py

```

TOKEN_DICT = {
    "+": 3,
    "-": 4,
    "*": 5,
    "/": 6,
    "=": 7,
    ">": 8,
    "<": 9,
    "<>": 10,
    "<=": 11,
    ">=": 12,
    "(": 13,

```

```
")": 14,  
"{": 15,  
"}": 16,  
";": 17,  
",": 18,  
"\\": 19,  
":=": 20,  
"var": 21,  
"if": 22,  
"then": 23,  
"else": 24,  
"while": 25,  
"for": 26,  
"begin": 27,  
"writeln": 28,  
"procedure": 29,  
"end": 30  
}
```

```
HINT_DICT = {  
    "ID": 1,  
    "INT": 2,  
    "ERROR": 100,  
    "STRING": 31,  
    "COMMENT": 32,  
}
```

main.py

```
from DFA import DFA  
  
start_state = 'START'  
accept_state = 'ACCEPT'  
dfa = DFA(start_state, accept_state)  
i = 0  
  
# Open file  
with open("code1.txt", "r") as file:
```



```
input_string = file.read()

while i < len(input_string):

    # 往 DFA 里输入一个字符
    dfa.run(input_string[i])

    # 如果 DFA 目前状态已经 accept
    if dfa.current_state == accept_state:
        dfa.conclude()
        # 用于将当前 DFA 的 Buffer 整合成 (30, "end")并输出
        dfa.refresh()
        # 清空 DFA 的 Buffer
    else:
        i += 1
        if i == len(input_string):
            dfa.conclude()
```

六、实验者自评

实验态度：这个实验本人秉承着严谨的态度，为了保证直接转向法代码的正确性，进行过多次改良。但是由于时间不多，代码中仍存在一些已知或者未知的错误，但暂未发现重大的影响功能的 Bug。本人还将整个项目开源至 GitHub，寻求他人批评指正。随后，将自己的代码和生成的代码进行详细对比，发现自身不足的同时继续做了些许改进。

实验方法：本人有和其他同学讨论过直接转向法编程的实现。大多数同学选择使用 C 语言进行面向过程编程。我认为这个问题应该最适合使用面向对象的编程方法，就像我上面的代码实现一样。本人编写了有穷自动机的类，将属性和方法打包进去，这让程序的可读性和可维护性达到了一个新的高度。得益于 Python 语法的灵活性，其中的一些 DFA 中的转换函数十分的直白，把 DFA 的精髓体现的很完美。

效果：手动编写的词法分析器效果和自动生成的代码效力一致，实现了题目的要求效果。