# 编译原理2-2 词法分析器

## 实验目的:

设计、编制并调试一个自定义语言C--的词法分析程序，加深对词法分析原理的理解。

## 程序的有限状态自动机

我在规划程序时将所有的词法单元分为标识符，常量，关键字，运算符，分节符，但是实际上我在程序中将类别分的非常细，返回的基本上就是记号的类别，这样虽然说写起来会更加的复杂，但是词法分析器得到的结果会更准确；

在这个程序中最复杂的实际上就是有限状态自动机的获得，而数字字面量实际上是相对比较复杂的一类，因为要处理八进制、十进制以及16进制这三种输入；另一方面，对于>,>=这种符号，我们采取最长匹配,遇到这些字符查看其下一个字符看能否组成更长的合法输入；

除此之外，只要状态机得到了标识之外的输入，即为语法错误。

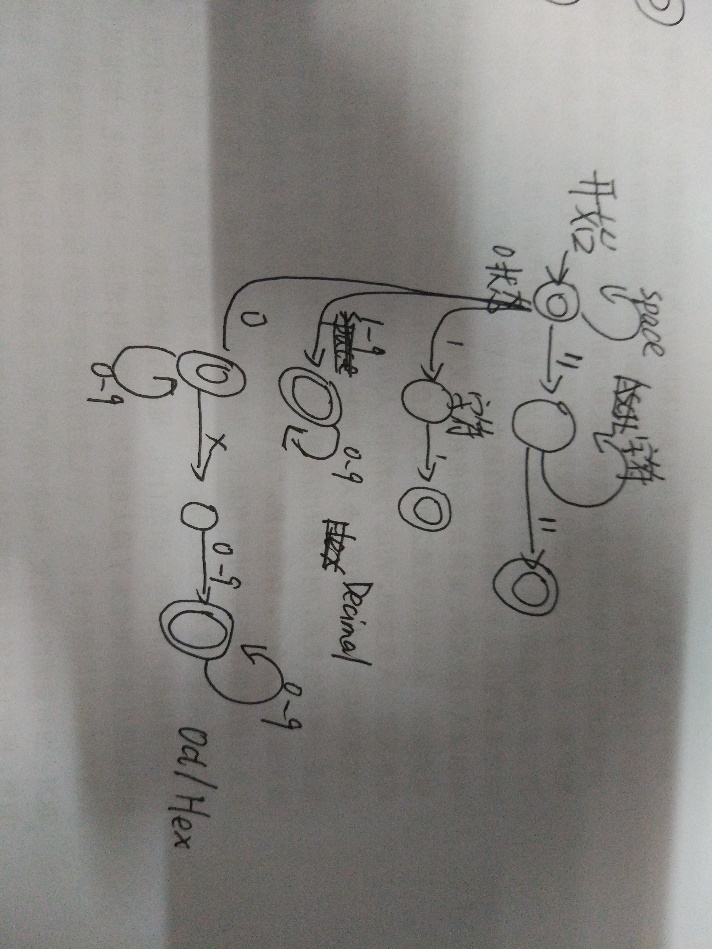
状态机在识别完一个串或者出现错误之后，返回0状态(初始状态)

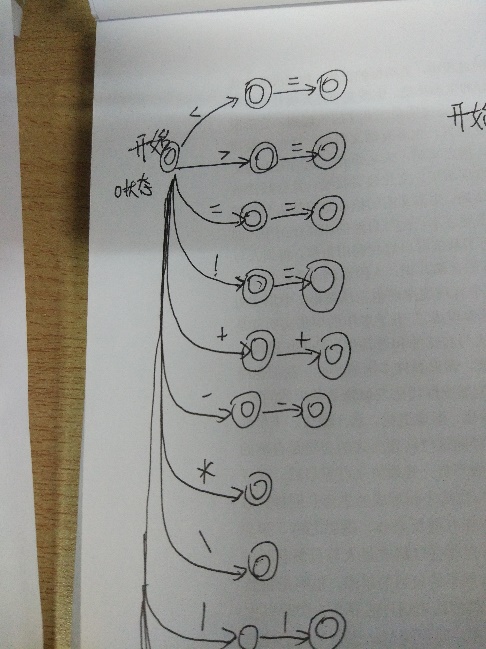
程序使用Python3.6编写，主要是因为其键值对操作非常方便。

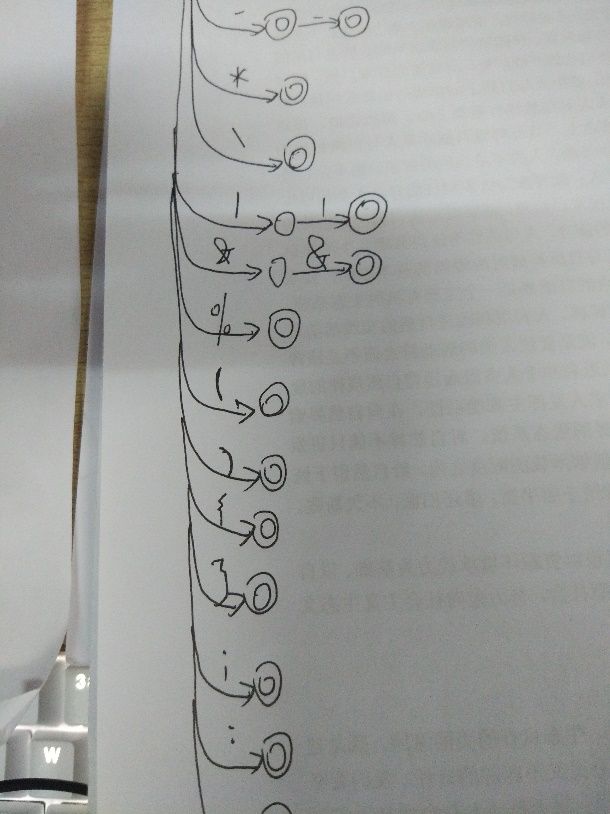
### 程序框图

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| lex\_parser(input\_str) | | | | | | | |
| parse\_space | parse\_seperator | parse\_operator | parse\_char | parse\_str | Parse\_num | | |
|  |  |  |  |  | decimal | hex | oct | |

### DFA：







.3.具体实现:

首先是处理各种空格的函数:

**def** parse\_space(x):  
 **if** x == **'\t' or** x == **' ' or** x == **'\n'**:  
 **return True  
 return False**

**然后判断是否是有效的分隔符的函数**

**def** parse\_seperator(x):  
 seperator\_dict = {  
 **','**: **'comma'**,  
 **':'**: **'colon'**,  
 **';'**: **'simcon'**,  
 **'('**: **'lparen'**,  
 **')'**: **'rparen'**,  
 **'{'**: **'lbrac'**,  
 **'}'**: **'rbrac'** }  
 print(**"< '{}'\t:\t'{}'\t >"**.format(seperator\_dict.get(x, -1), x))

**判断是否是有效的运算符的函数:**

**ef** parse\_operator(x):  
 operator\_dict = {  
 **'++'**: **'inc'**,  
 **'--'**: **'dec'**,  
 **'+'**: **'add'**,  
 **'-'**: **'minus'**,  
 **'\*'**: **'mul'**,  
 **'/'**: **'div'**,  
 **'%'**: **'mod'**,  
 **'<='**: **'le'**,  
 **'<'**: **'lt'**,  
 **'>='**: **'ge'**,  
 **'>'**: **'gt'**,  
 **'=='**: **'eq'**,  
 **'='**: **'assign'**,  
 **'!'**: **'not'**,  
 **'!='**: **'nequ'**,  
 **'&&'**: **'and'**,  
 **'||'**: **'or'** }  
 **if** operator\_dict.get(x, -1) != -1:  
 print(**"< '{}'\t:\t'{}'\t >"**.format(operator\_dict.get(x, -1), x))  
 **else**:  
 **return** -1  
 **return** 0

**以上函数不是均返回-1**

**定义判断是否是关键字的函数**

**def** parse\_keyword(x):  
 key\_word\_dict = {  
 **'int'**: **'kw\_int'**,  
 **'char'**: **'kw\_char'**,  
 **'void'**: **'kw\_void'**,  
 **'if'**: **'kw\_if'**,  
 **'else'**: **'kw\_else'**,  
 **'switch'**: **'kw\_switch'**,  
 **'case'**: **'kw\_case'**,  
 **'default'**: **'kw\_default'**,  
 **'while'**: **'kw\_while'**,  
 **'do'**: **'kw\_do'**,  
 **'for'**: **'kw\_for'**,  
 **'break'**: **'kw\_break'**,  
 **'continue'**: **'kw\_continue'**,  
 **'return'**: **'kw\_return'** }  
 **if** key\_word\_dict.get(x, -1) != -1:  
 print(**"< '{}'\t:\t'{}'\t >"**.format(key\_word\_dict.get(x, -1), x))  
 **else**:  
 **return** -1  
 **return** 0

**在主循环中依次判断每个字符该运行哪一条指令流，直到代码流终结;在这里我们使用index来模拟DFA的运行**

**def** lex\_parser(input\_str):  
 i = 0  
 str\_len = len(input\_str)  
 *# 主循环* **while** i < str\_len:  
 **if** parse\_space(input\_str[i]):  
 i = i + 1  
 **continue  
 if** input\_str[i] **in** [**','**, **':'**, **';'**, **'{'**, **'}'**, **'('**, **')'**]:  
 parse\_seperator(input\_str[i])  
 i = i + 1  
 **continue** *# operator* **if** input\_str[i] **in** [**'\*'**, **'/'**, **'%'**]:  
 parse\_operator(input\_str[i])  
 i = i + 1  
 **continue  
 if** input\_str[i] **in** [**'<'**, **'>'**, **'='**, **'!'**, **'+'**, **'-'**, **'&'**, **'|'**]:  
 **if** parse\_operator(input\_str[i:i + 2]) != -1:  
  
 i = i + 2  
 **else**:  
 parse\_operator(input\_str[i:i + 1])  
 i = i + 1  
 **continue** *# char* **if** input\_str[i] == **"'"**:  
 **assert** input\_str[i + 2] == **"'"** print(**"< '{}'\t:\t'{}'\t >"**.format(**'char'**, input\_str[i + 1]))  
 i = i + 3  
 **continue** *# str* **if** input\_str[i] == **'"'**:  
 locate = i + 1  
  
 **while** input\_str[locate] != **'"'**:  
 locate = locate + 1  
  
 print(**"< '{}'\t:\t'{}'\t >"**.format(**'str'**, input\_str[i + 1:locate]))  
 i = locate + 1  
 **continue** *# 数字* **if '1'** <= input\_str[i] <= **'9'**:  
 locate = i + 1  
 **while '9'** >= input\_str[locate] >= **'0'**:  
 locate = locate + 1  
 print(**"< '{}'\t:\t'{}'\t >"**.format(**'num\_literal'**, input\_str[i: locate]))  
 i = locate  
 **continue  
  
 if** input\_str[i] == **'0'**:  
 *# oct / hex* **if** input\_str[i + 1] == **'x' or** input\_str[i + 1] == **'X'**:  
 locate = i + 2  
 **while '9'** >= input\_str[locate] >= **'0'**:  
 locate = locate + 1  
 print(**"< '{}'\t:\t'{}'\t >"**.format(**'num\_literal'**, input\_str[i: locate]))  
 i = locate  
 **else**:  
 locate = i + 1  
 **while '9'** >= input\_str[locate] >= **'0'**:  
 locate = locate + 1  
 print(**"< '{}'\t:\t'{}'\t >"**.format(**'num\_literal'**, input\_str[i: locate]))  
 i = locate  
 **continue  
 if 'z'** >= input\_str[i] >= **'a' or 'Z'** >= input\_str[i] >= **'A' or** input\_str[i] == **'\_'**:  
 locate = i + 1  
 **while** (**'a'** <= input\_str[locate] <= **'z' or 'A'** <= input\_str[locate] <= **'Z' '0'** <= input\_str[  
 locate] <= **'9' or** input\_str[locate] == **'\_'**):  
 locate = locate + 1  
 lex\_str = input\_str[i: locate]  
 **if** parse\_keyword(lex\_str) != 0:  
 print(**"< '{}'\t:\t'{}'\t >"**.format(**'var'**, input\_str[i: locate]))  
 i = locate  
 **continue** print(**"error in lex"**)  
 i = i + 1

最后是我是用这个Lex为下面这段代码进行词法分析所得到的结果:,其中第一个token是一个非法的字符。

example\_code = **'''  
；  
int main(void)  
{  
 \_\_user asmimage 12\_32;  
 int a = 010;  
 char c = 'x';  
 str s = "As you are";  
 for(int i = 0 ;i <= 0x10; ++i)  
 {  
 void \*m = i % 11;  
 continue;  
 }  
 return 0;  
  
}**

error in lex

< 'kw\_int' : 'int' >

< 'var' : 'main' >

< 'lparen' : '(' >

< 'kw\_void' : 'void' >

< 'rparen' : ')' >

< 'lbrac' : '{' >

< 'var' : '\_\_user' >

< 'var' : 'asmimage' >

< 'num\_literal' : '12' >

< 'var' : '\_' >

< 'num\_literal' : '32' >

< 'simcon' : ';' >

< 'kw\_int' : 'int' >

< 'var' : 'a' >

< 'assign' : '=' >

< 'num\_literal' : '010' >

< 'simcon' : ';' >

< 'kw\_char' : 'char' >

< 'var' : 'c' >

< 'assign' : '=' >

< 'char' : 'x' >

< 'simcon' : ';' >

< 'var' : 'str' >

< 'var' : 's' >

< 'assign' : '=' >

< 'str' : 'As you are' >

< 'simcon' : ';' >

< 'kw\_for' : 'for' >

< 'lparen' : '(' >

< 'kw\_int' : 'int' >

< 'var' : 'i' >

< 'assign' : '=' >

< 'num\_literal' : '0' >

< 'simcon' : ';' >

< 'var' : 'i' >

< 'le' : '<=' >

< 'num\_literal' : '0x10' >

< 'simcon' : ';' >

< 'inc' : '++' >

< 'var' : 'i' >

< 'rparen' : ')' >

< 'lbrac' : '{' >

< 'kw\_void' : 'void' >

< 'mul' : '\*' >

< 'var' : 'm' >

< 'assign' : '=' >

< 'var' : 'i' >

< 'mod' : '%' >

< 'num\_literal' : '11' >

< 'simcon' : ';' >

< 'kw\_continue' : 'continue' >

< 'simcon' : ';' >

< 'rbrac' : '}' >

< 'kw\_return' : 'return' >

< 'num\_literal' : '0' >

< 'simcon' : ';' >

< 'rbrac' : '}' >

## 总结:

这个实现实际上就是实现了一个有限状态自动机，在这里我是利用Python的控制流模拟了这个过程，这样的一个程序是绑定在了这个语法上的，如果说适配另一个语法，基本上需要推倒重写；

如果说写一个通用的Lexer,我的思路是首先实现正则表达式到NFA的程序，然后再实现自动机的确定化与最小化，最后使用这个获得的DFA来接受输入串，判断其是否合法，如果合法，它的二元组又是什么。这应该也是一些成熟的Lexer的做法。当然，一些编译器/编译器前端如GCC,Clang处于效率的考虑，词法分析器都是手工构造的。