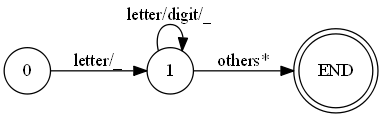
**词法分析程序设计文档**

**一、识别各类符号的DFA**

我们令state = 0时作为系统的初态，此时自动机处于初始状态，自动机应读入一个字符，并根据这个字符做状态转移，进入到相应的子自动机。我们根据DNF在初态输入的第一个符号来作为以下说明的分类。

1. **letter / \_**

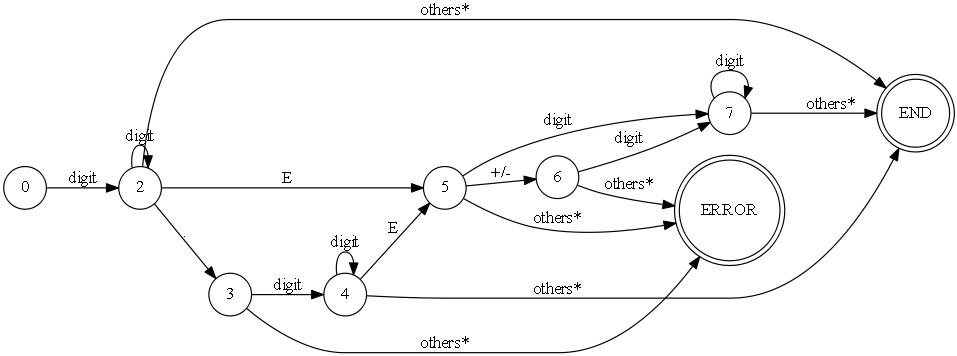
此时进入标识符或关键字状态，除添加下划线与letter等价之外，其余部分均与教材中的自动机相同。在遇到other\*时输出标识符或关键字，具体取决于在关键字表中查找的结果。关键字表以文件的形式存储，在程序开始时被自动载入。



1. **digit**

此时进入无符号数识别状态。与教材中的状态机相同。请注意，我们在这里定义一个错误处理过程errorProcess，某一状态是可能出错的，当且仅当这个状态射出字符的集合的并集与本程序识别的符号集不相等，也就是说，这个状态的射出字符集是不完全的。当在此状态下输入无射出的字符，那么将会进入到错误处理阶段。

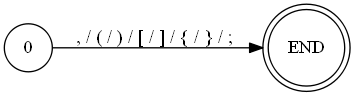
在errorProcess中，我们定义这样的过程：状态回零并向输入流回送一个字符，并打印错误报文。



如上，我们利用python调用graphviz绘制DFA，其基本思想与我们的设计是一致的，除了开始状态（状态0）以外，有责任为每一个子状态设置出口，这个出口只能是ERROR或者是END（这是由于射出集合的完备性，即对于任何中间状态，其射出字符集合的并应与识别的字符集相等），其中ERROR代表执行过程errorProcess，而END代表renturn一个token，如果label带有‘others\*’的标志则还需执行otherProcess，并完成相关的处理工作。

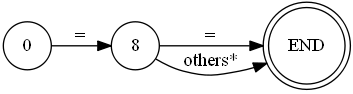
1. **, / ( / ) / [ / ] / { / } / ;**

此时进入单符号识别状态，这些符号在本程序的符号集中不需要超前识别，因此遇到它们直接输出相应的token即可。



1. **=**

此时需要超前识别，如果下一个字符为“=”，输出关系运算符“==”。否则，进入otherProcess过程，输出一个赋值运算符“=”。请注意，我们定义了otherProcess对应于DFA中的other\*射出态，这个过程需要向输入流返回一个字符，向token中添加行号，状态回零并输出token（为了减少参数传递的麻烦，本程序直接在otherProcess外部完成token的初始化和输出）。



从graphviz的图形中不难看出类似于识别‘=’这样符号的设计，此类符号大多需要超前读入，但最后都归结为END状态，区别仅限于是否执行otherProcess（当然它们的输出也不同）。

1. **!**

与（4）情况类似，唯一的区别在于识别的是“！ ”还是“！=”，在此不再赘述。

1. **>**

与（4）情况类似，唯一的区别在于识别的是“>”还是“>=”，在此不再赘述。

1. **<**

与（4）情况类似，唯一的区别在于识别的是“<”还是“<=”，在此不再赘述。

1. **+**

与（4）情况类似，唯一的区别在于识别的是“+”还是“+=”还是“++”，在此不再赘述。

1. **–**

与（4）情况类似，唯一的区别在于识别的是“-”还是“-=”还是“--”，在此不再赘述。

1. **\***

与（4）情况类似，唯一的区别在于识别的是“\*”还是“\*=”，在此不再赘述。

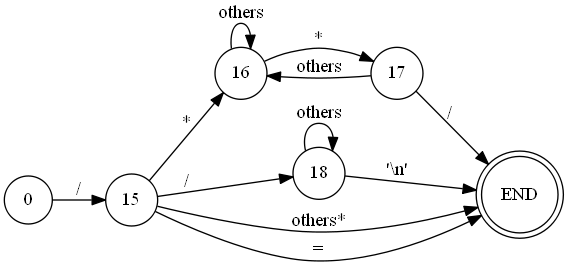
1. **/**

此时进入/符号的识别状态，如果下一个字符是others\*，则进入otherProcess过程，并输出一个算数运算符“/”，如果下一个字符是“=”，那么直接输出赋值运算符“/=”。

如果下一个字符是“\*”，我们将进入注释处理状态。在这个状态下，只要不识别到“\*”，程序一直做跳过字符的操作，直到遇到疑似的结束符“\*”，程序进入结束判定状态。此时若下一个字符是“/”，则结束注释识别过程并返回，否则，返回上一个状态继续进行注释处理。

如果下一个字符是“/”则进入单行注释处理状态，此时的过程与上述的多行注释类似，只不过注释的结束符被唯一标定为字符“/n”，否则一直进行单行注释处理。

可用DFA表达上述过程如下：



这里我们第一次出现了others而非others\*，顾名思义，这里的others代表从输入流中读入一个字符后，不必再回退给输入流一个字符，这是因为我们正在做的是排除注释，而非超前读入。

1. **&**

与（4）情况类似，唯一的区别在于识别的是“&”还是“&=”，还是“&&”，在此不再赘述。

1. **|**

与（4）情况类似，唯一的区别在于识别的是“|”还是“|=”，还是“||”’，在此不再赘述。

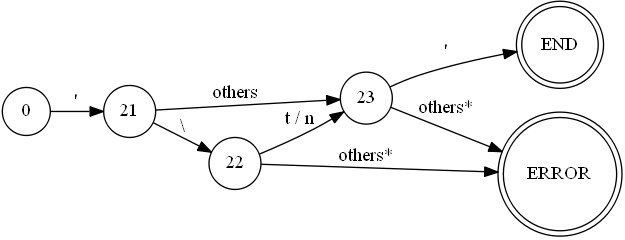
1. **‘**

进入单字符char识别状态，需要注意的就是，在C中一些与\组合起来的转义字符也属于单字符，为了简单起见，我们规定此词法分析器仅可以识别C转义字符的一个子集，只有‘\n’和‘\t’是合法的转义字符，其他转义字符不作考虑。

如果输入字符是‘\’，我们进入转义字符识别状态，此时只有再输入t或n才是有效的转义字符，其他的均进入到错误处理过程。

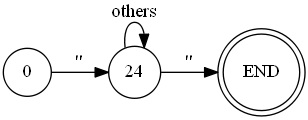
如果输入字符是其他，那么我们进入结束等待态，在这个状态中，只有输入“ ' ”（另一半的单引号）才算识别字符结束，否则进入错误处理。

以上叙述可以总结为如下的DFA：

****

1. **“**

进入字符串string识别状态，需要注意的是，字符串的识别远比单个字符限制要少，因此也简单的多，在这个状态下，只需要不断读入字符，直到遇到另一个双引号即可。



**二、Token的类别以及属性**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **记号** | **种别码** | **属性** | **模式** | **范例** |
| KEY | 1 | 关键字表项指针 | C语言关键字符集 | int , char , return , void ,main … |
| ID | 2 | 标识符表项指针 | 首字母为下划线，后跟若干字母、下划线或数字的字符串 | sum , \_init , word84 |
| NUM | 3 | 常数值 | 任何C标准的数值常量 | -1.4E-2 , 0.0001 ... |
| LITERAL | 4 | 字符串值 | “”之间的任何字符串 | "hello, world \n " |
| CHAR | 5 | 字符值 | ‘’之间的任何字符（转义字符仅是子集） | c' , '0' , '\n' , '\t' |
| SINGLE\_OP | 6 | 符号本身 |  | , ( ) [ ] { } |
| ASSIGN\_OP | 7 | 符号本身 |  | = += -= /= &= … |
| LOGIC\_OP | 8 | 符号本身 |  | && || ! |
| RELATION\_OP | 9 | 符号本身 |  | > < == != >= <= |
| ARITH\_OP | 10 | 符号本身 |  | + ++ - -- \* / |
| BITWISE\_OP | 11 | 符号本身 |  | & | |

**三、程序的设计与实现**

1. **全局过程和变量**

|  |  |  |
| --- | --- | --- |
| **名字** | **类型** | **用途** |
| Token | class | 用来存储一个token，本质上是一个二元组，形式为<类别，属性>。为了简化程序，我们默认一个Token具有所有的属性值，只是当我们可以确定其类别时（本程序是如此的），我们根据其类别判断哪个属性值是有效的 |
| buffer | char | 由于可能需要预读，因此设置输入缓冲区 |
| token | Token\* | 为简单起见，Token 使用全局单例模式，token是它的一个指针 |
| key\_table | vector<string \*> | 关键字表 |
| sym\_table | vector<string \*> | 标识符表 |
| cur\_row | int | 当前行号 |
| in | ifstream | 输入流 |
| ch | char | 当前的输入字符 |
| str | string | 当前的已识别到的符号串 |
| state | int | DNF状态 |
| pf | int | 缓冲区指针 |
| type\_count | int[] | 类型统计数组 |
| char\_count | int | 字符统计量 |
| void initKeyTable() | func | 过程，以文件的形式初始化关键字表 |
| void Token::output() | func | 输出一个已经成型的token，若涉及语法程序，可看作向语法分析程序返回一个Token的引用或指针，在本程序中只做简单的输出 |
| char get\_char() | func | 读入一个字符，同时完成缓冲区溢出判断等操作 |
| void skipSpace() | func | 跳过空白符 |
| void retract() | func | 缓冲区指针退回一个字符 |
| string \* searchKeyTable() | func | 查找关键字表，返回一个匹配的指针 |
| string \* searchSymTable() | func | 查找标识符表，返回一个匹配的指针 |
| void otherProcess() | func | 对应于自动机的other\*过程 |
| void errorProcess() | func | 错误处理过程，涉及到状态回零，retract和打印操作 |
| void tokenizer() | func | 词法分析器的主过程 |

1. **简要设计思想**

从DFA中可以得出本程序的所有控制结构，需要说明的是，本程序识别的符号集是C语言的一个子集，这个子集中所涉及到的符号单元为所有状态射出边上label内容的并集。

词法分析器采用基于DFA的逻辑控制结构，对应于C语言的IF-ELSE分支和SWITCH-CASE分支。每一个DFA的状态都对应自己相应的一段程序，每一个子自动机负责识别一类符号，这些符号拥有相同的起始字符。所有的子自动机都是并行的，这也就要求新添加一个起始状态作为初始状态，初始状态下需要对str进行初始化，并对缓冲区的情况进行判断。

缓冲区的设计并没用采用书上的二分结构，而是将其中的一个分支简化为“哨兵单元”，这个单元的作用是简化retract的判断工作。get\_char过程负责读入输入流中的一个字符并前移pf，但前提是缓冲区没有溢出。为了减少判断工作，我们让每一次的缓冲区载入从buffer[1]开始并让之前的buffer最后一个字符载入到buffer[0]之中并置pf为1，这样，在进行retract过程时，无论任何情况，只需要进行pf- -的操作即可，不需要在判断pf的位置。

整个过程可以以伪代码描述如下：

*char get\_char()*

*{*

*ch = buffer中pf的内容*

*if(pf == BUF\_SIZE)*

*{*

*buffer[0] = buffer[pf];*

*读入下一个块，置于&buffer[1]中*

*len = 读入的块大小*

*总字符数 += len;*

*if(len < BUF\_SIZE)*

*{*

*在buffer中添加结束标记EOF;*

*}*

*pf = 1;*

*}*

*else pf++;*

*return ch*

*}*

然而对于errorProcess过程，本程序简单地向输入流退回一个字符并报错，令状态归零。编译器不需要对程序员的行为做额外猜测，这不是编译器应该做的事。

在报错的尺度设计上，重点在于是否应该报出**所有的**词法错误。例如这样的声明int 123abc = 100;显然地，这是不符合C标识符格式的非法标识符，但是这样的“词法错误”一定要被词法分析器抛出吗？按照设计好的DFA我们很容易看到，词法分析器先识别123为常量，再识别abc为标识符。在GCC编译器编译此语句后，报出的错误是语法分析错误，即无符号数后面不可能跟一个标识符。倘若设计词法分析器抛出这个错误，那要将DFA的很多部分修正，考虑到很多种额外的情况来面面俱到。但仔细分析一下可知，修改后的设计反而是不合理的，**词法分析器应该专注其本职工作而不能越界**！它的任务**只是**作为语法分析的子程序向语法分析程序提供token，123abc这类的非法标识符识别代价很高，但是如果交给语法分析器做，是一件很简单自然的事情。同样的道理，括号匹配等任务也不应该出现在一个好的词法分析器中。

对于otherProcess的核心思想就是区分贪婪模式和非贪婪模式，很显然我们的词法分析器都应该是非贪婪的，但是这样的区别在正则文法中不能很好体现，因此教材中的others隔断设计是以DFA为核心逻辑的词法分析器最精妙的地方，以others作为隔断这在文法中没有体现，需要动一点脑筋才能解其意，本程序沿用这样的思想。

1. **对于无法用正则文法识别符号的处理**

仔细观察不难发现，C语言的符号集的属性不可能全部可以用DFA识别，举个最简单的例子，符号\*有两种含义，一个是算术运算符的“乘号”，另一个是地址运算符的“地址索引”，仅仅依靠DFA是不能区分它们的，而+符号就不涉及这个问题，通过DFA很容易识别它，这是因为它在DFA中不构成歧义。

因此如何处理类似\*这样的符号？经过思考后，我认为利用DFA进行词法分析这个大原则不变的基础上，给类似\*的符号划定一个集合A，集合A中的符号输出的二元组中，属性应该是UNKNOWN的，即遇到\*时应该输出<UNKNOWN,\*>，类似的token的种别识别过程交由语法分析程序处理。这也就说明，并非任何token在输出时都应该保证有其确定的种别和属性。对于集合B，也即符号集U对于A的补集，这些符号是可以被DFA唯一识别的，可以由原来的策略输出<TYPE,ATTRIBUTE>。

而在本程序中，根据题目要求，上述思想被简化为：输入集的符号仅仅局限于B集合，这样可以给程序的输出带来清晰性，又可以满足题目的要求，因此识别的是C语言符号集的一个子集。

**四、总结与反思**

在编程时涉及到思想问题如“贪婪模式”、“错误处理过程和other过程的设计”、“token结构的设计”都已经得到了初步的解决，DFA的绘制问题采用graphviz的脚本编程方式也很好地呈现在文档里。最重要的是，本次编程明确了一个思想：词法分析器究竟该做什么？它作为被语法分析器调用的子过程，它每次被调用后都输出什么？词法分析器如何定义所谓的“词法错误”？像123abc这类的标识符错误是否必须要被词法分析器所识别？这些问题折射出对程序设计和维护方面的理解，在上述的文档中**均可以**得到解答。

但是，有些问题解决的并不完美。最大的问题在于无法用正则文法识别的token该如何处理（也即三.3的内容）。为了简洁的表述，我们采取的方法是一个折衷——取C语言的字符子集，这些子集有这样的特点：它们都可以被正则文法无二义性地识别。典型的例子如符号“\*”，我们强制定义为它是乘号，而不能是寻址运算符。

但事实上，经过DevC中GUN GCC编译器的词法分析过程，我们发现这样的过程必然是不合理的，词法分析器简单地抛出一个token，即<UNKNOWN,\*>，并将UNKNOWN的确定交给语法分析器处理。这同时引发出了一个问题，对于C语言符号集U，存在一个集合A是“\*”的同类，它们不能由正则文法唯一识别，可是如何确定这个A呢？这恐怕就要追溯到语言的设计者上了。我们可以这样来较好的处理这个问题，即对于任意符号，如果属于集合A，那么输出的属性是UNKNOWN，交由语法分析器处理，否则，输出一个完整的token。然而在本程序角度上来讲，**没有必要**将集合A完全挖掘出来，**但是我们理应对这方面的问题给予一定的思考**。