**Compiler document**

[1.简介(introduction) 2](#_Toc98145700)

[2.编译(Compiler) 3](#_Toc98145701)

[2.1 词法分析(Lexer Analyse) 3](#_Toc98145702)

[2.1.1 感性认识 3](#_Toc98145703)

[2.1.2 理论部分 4](#_Toc98145704)

[(a).正则表达式(Regular Expression) 4](#_Toc98145705)

[(b).有穷自动机(Finite Automata,FA) 4](#_Toc98145706)

[(c).有穷自动机分类 6](#_Toc98145707)

[(d).DFA 6](#_Toc98145708)

[(e). NFA 7](#_Toc98145709)

[(f). DFA与NFA的等价性 8](#_Toc98145710)

[2.1.3 编程设计 10](#_Toc98145711)

[2.2 语法分析(GrammarAnalyse) 12](#_Toc98145712)

[2.2.1 感性认识 12](#_Toc98145713)

[2.2.2 理论部分 13](#_Toc98145714)

[(a).部分概念 13](#_Toc98145715)

[(b). 文法形式化定义(Grammar) 14](#_Toc98145716)

[(c). 推导(Derivations)及归约(Reductions) 15](#_Toc98145717)

[(d). 句型(Sentential Form)和句子(Sentence) 16](#_Toc98145718)

[(e). 语言(Language) 16](#_Toc98145719)

[(f). 文法分类 17](#_Toc98145720)

[(f). CFG文法分析树 18](#_Toc98145721)

[(g). 二义性文法(Ambiguous Gramma) 19](#_Toc98145722)

[(h). 自顶向下分析(Top-Down Parsing) 20](#_Toc98145723)

[(i). 递归下降分析(Recursive-Decent Parsring) 21](#_Toc98145724)

[2.2.3 编程设计 22](#_Toc98145725)

[2.3 语义分析(Semantic) 23](#_Toc98145726)

[2.4 汇编代码生成 23](#_Toc98145727)

[3.汇编(Assemble) 23](#_Toc98145728)

[3.1 词法分析(Lexer Analyse) 23](#_Toc98145729)

[3.2 语法分析(Parser Analyse) 23](#_Toc98145730)

[4.链接(Linker) 23](#_Toc98145731)

[5.可执行文件(Executive File) 23](#_Toc98145732)

[Reference 23](#_Toc98145733)

## 1.简介(introduction)

## 2.编译(Compiler)

程序员完成源代码的输入后，首先需要对源程序进行编译，编译过程分为词法分析和语法分析，最后生成中间代码。当然生成中间代码的前提还要保证没有语义错误。这里的中间代码指汇编代码。这里提到的编译和我们平时谈到的编译器可能还是有些不一样，平时所说的编译器是一种概述，其实是一套编译系统，但这个地方提到的编译，应该是指狭义上的编译过程。

### 2.1 词法分析(Lexer Analyse)

#### 2.1.1 感性认识

词法分析是将输入的源程序切分为一系列的token。示例程序如p1所示。

*.e.g.*

// 示例程序p1

*int main(){*

*printf(“hello Compiler”);*

*return 0;*

*}*

对于p1程序，经过切分，能够得到如下token序列。如表1所示。

表1 程序p1对应的token序列

|  |  |
| --- | --- |
| **Token** | **Value** |
| int | 保留字/关键字 |
| main | 变量名 |
| { | 左花括弧 |
| printf | 变量名 |
| ( | 左括号 |
| “hello Compiler” | 字符串 |
| ) | 右括号 |
| ; | 分号 |
| return | 保留字/关键字 |
| 0 | 数字 |
| ; | 分号 |
| } | 右花括弧 |

从表1中能够看出，对于程序p1而言，token主要有几种类型，首先是保留字，这些保留字程序员不可以用作变量名，类似的保留字还有if、else、return等等；其次是数字，程序的主要功能其实是运算，运算离不开各种数字；再次是变量名，程序中可以存在许多变量名；同时，程序的输出除了运算的结果，还需要一定的提示信息，这些信息通常是用字符串表示，例如“hello Compiler”；最后是一些界符，如分号、逗号以及各种括号等。

设计词法分析的程序，需要从源程序中提取出这些token，源程序本质是一个很长的字符串，通过词法分析后，得到一个个有具体含义的token序列。

#### 2.1.2 理论部分

##### (a).正则表达式(Regular Expression)

要提取出这样的各类token，首先能够想到的是正则表达式(Regular Expression)[1]，例如C语言**标识符**(ident)的正则表达式有如下定义:

其中，digit是数字，即0至9；letter是字母和下划线；最后ident由字母或者下划线开头，由下划线字母或者数字组成。\*表示前面的符号出现0次、1次或者多次;|表示或关系(多选一)。

类似的，可以写数字(number)的正则表达式:

digit依旧是数字；digits是长度大于等于1的数字，注意两个符号直接串行排列，没有|，此时不是二选一的关系(**或**)，是依次匹配关系(**与**)；表示可选的小数部分，其可以是一个小数点打头，后面跟上长度大于等于1的数字串，也可以是一个空串(,下同)，因此称为可选的小数部分；类似地，表示可选的指数部分；最后number由这些符号共同组成。

特别地，当和都取空串时，number此时就是一个整数，如2；若可选指数部分为空，则表示一般意义的浮点数，如9.9；若三个部分都不为空，则类似于3.14E3、3.14E+3、3.14E-3这样的数字都能够被上述正则表达式解析。

##### (b).有穷自动机(Finite Automata,FA)

FA这样一种装置，它的特点在于，它并不需要记忆先前的信息，只需要根据当前所处的状态和输入的信息，转移到下一个状态。

例如一个电梯装置，它并不需要记忆走过的楼层，它只需要根据当前的状态(**当前位于的楼层数**)和输入的信息(**用户按下的按钮**),然后转移到下一个状态(**按钮对应的楼层**)。

类似的，指令执行的虚拟机，它无需记忆自己执行过的指令，只需要根据当前状态(**PC指针，当然还包括一些寄存器/内存**)，和输入的信息(**本条指令**)，然后转移到下一个状态(**新的PC指针和寄存器/内存的值**)。

FA的转换图如图1所示，其中初始状态是start,只有一个；终止状态(接收态)可以有多个，由双圈表示；而有向边表示对于给定的输入(如*a*、*b*)，可以由一个状态转移到另外一个状态。

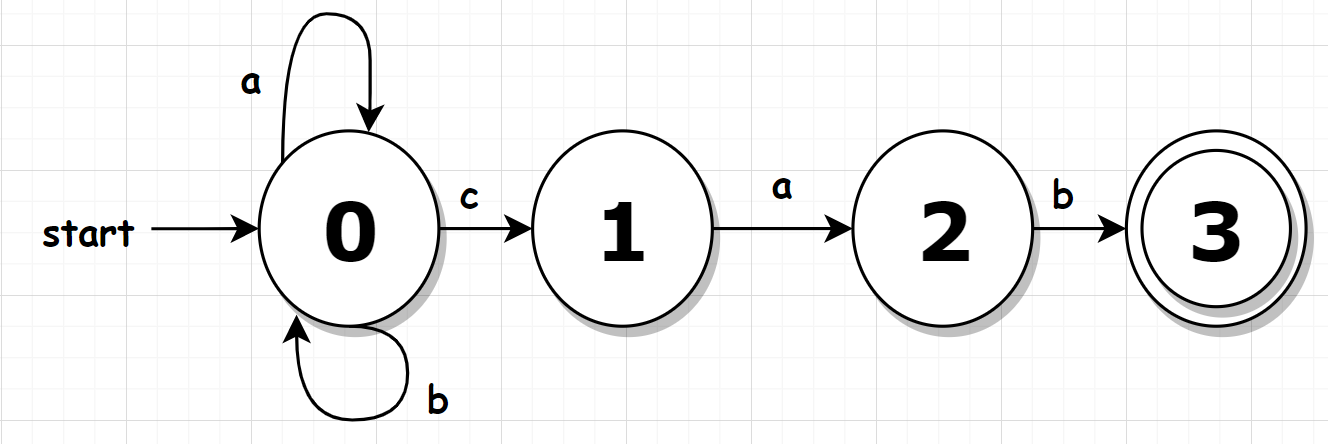


图1 FA 转换图(Transition Graph)

在图1中，给定一个串，如*abcab*，上述有穷自动机能够从初始态到终止态，则称串*abcab*能够被该FA接收。

一个有穷自动机(Machine,M)能够接收的所有串构成的集合，成为该FA构成的语言，记为L(M)。

上述有穷自动机能够接收的语言其实是由*cab*结尾，*a*或者*b*打头构成的串集合，其实也可以用下面的正则表达式描述:

+表示至少重复1次。现在也有相当多的在线平台[2]供大家测试，如图2所示，*bbbbabc*被这个正则表达式接收了。

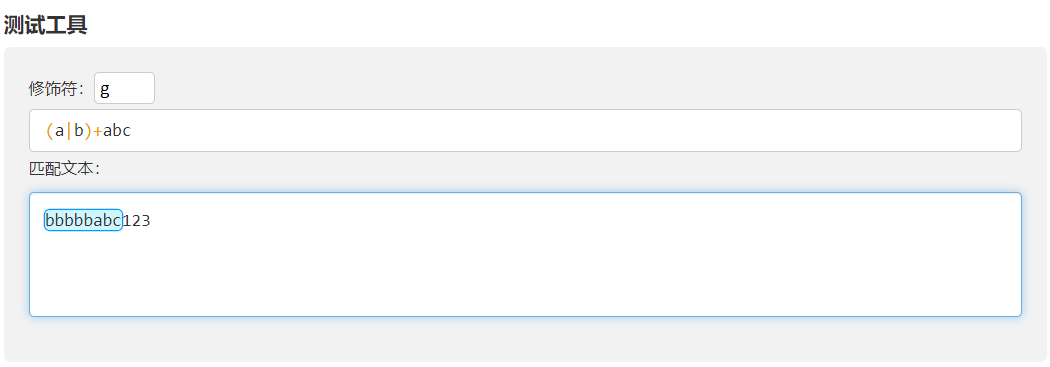


图2 在线平台测试结果

另外一个例如如图3所示。这个自动机有多个终止态，此时遵循最长字符串匹配原则(Long String Match Principle)。简单来说，最后匹配的结果选择某最长的前缀进行匹配。例如输入字符串为<=时，虽然<符号已经匹配到终止态1，但是后面的=匹配到终止态2，根据最长匹配原则，应该匹配<=，到达终态2。下面的减号是类似的。

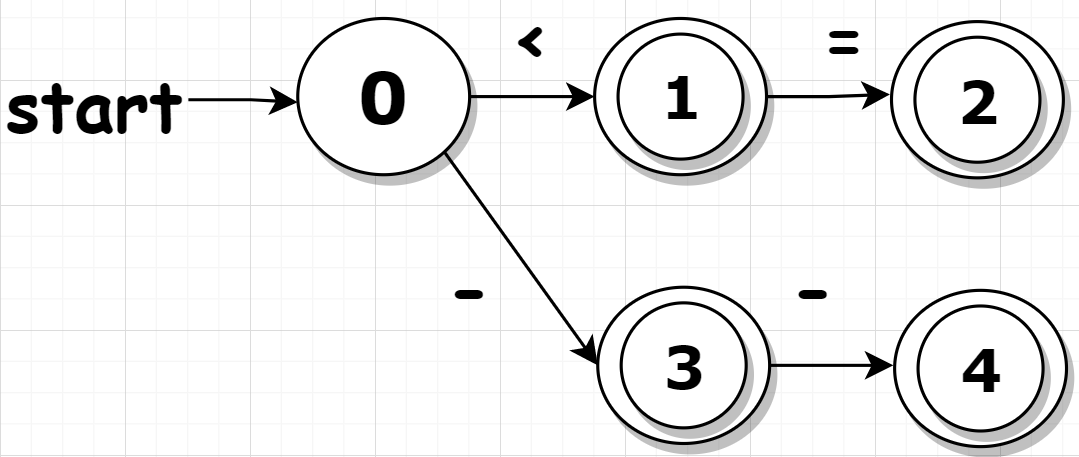


图3 最长匹配原则示例

##### (c).有穷自动机分类

有穷自动机分为确定有穷自动机(Deterministic FA,DAF)[3]和不确定的有穷自动机(NonDeterministic FA,NFA)[4]。

DFA，之所以叫确定的有穷自动机，因为处于某个状态时，给定一个输入，它达到的下一个状态是唯一确定的；与之相对的NFA，在某个状态给定一个输入，它可能有多个可转移的状态；或者NFA也可以这样理解，对于给定的一个输入，有多个状态可以转移，但具体转移至哪个状态，还取决于之后的输入信息。

##### (d).DFA

一般会使用五元组描述一个自动机。对于DFA而言，其中各个符号的含义如下所示。

* :有穷状态集合，即所有状态构成的集合。
* :输入的符号构成的集合。
* :状态转移函数，即表示给定一个输入符号，对于当前所在状态能够到达的下一状态。科学一点的表示是，。即一种映射，结合下面的例子就明白了。
* 初始状态，。
* 接收状态构成的集合，也叫终止状态，显然。

下面是一个简单的有穷自动机的例子，如图4所示。

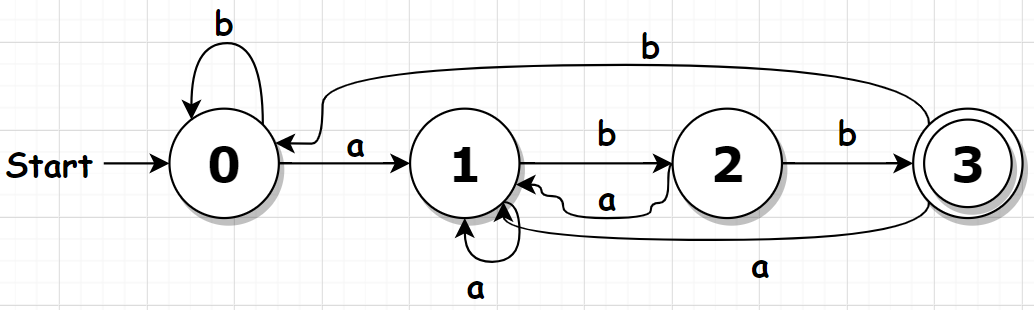


图4 DFA例子

这里；；；。而其实描述的就是图中的各种状态转换的曲线，当然也可以用转换表的表示。如表2所示。

表2 DFA状态转移表

|  |  |  |
| --- | --- | --- |
| **输入 状态** |  |  |
| 0\*(起始) | 1 | 0 |
| 1 | 1 | 2 |
| 2 | 1 | 3 |
| 3\*(终止) | 1 | 0 |

从转换表可以看到各个状态之间转移的条件，这里也能明确看到，从一个状态，给定一个输入，能转移到唯一确定的状态。

这个自动机构成的语言L(M)是怎样的呢？

* **状态0**:遇到*b*则一直处于状态0，直到输入*a*，转移到状态1。
* **状态1**:遇到字母*a*则一直保持，直到输入符号*b*，说明状态1是以*a*结尾的字符串。
* **状态2**:以*ab*结尾的字符串，若再次输入*a*时，打破了这一规则，转移到状态1，此时字符串以*a*结尾；若输入*b*，此时转移到状态3。
* **状态3**:以abb结尾的字符串被最后接收。若再次输入*a*，则转移到1(以*a*结尾状态)，输入*b*则回退到0(初始状态)。

经过上述分析，L(M)表示的是任何以*abb*结尾的字符串，如果用正则表达式表示，则应该是。

##### (e). NFA

NFA对应的五元组和DFA基本一致，有一点**不同**，NFA中的描述的不再是单一元素，而是一个集合，即下一个状态可能的集合。映射关系被修改为了:。从DFA中可以看到，，而NFA。

依旧举一个例子，如图5所示。这里；；；。与前面的DFA是相同的。

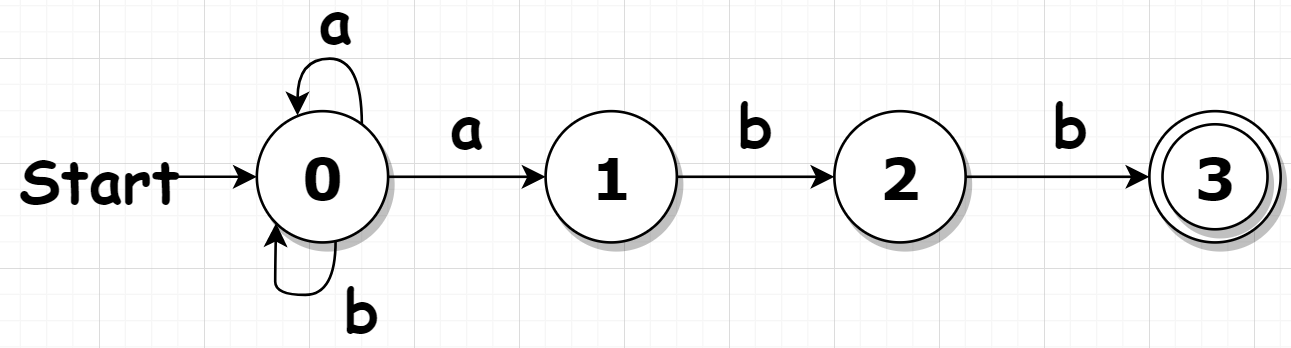


图5 NFA例子

如果观察状态转移表，类似的有表3。

表3 NFA状态转移表

|  |  |  |
| --- | --- | --- |
| **输入 状态** |  |  |
| 0\*(起始) | {1,0} | {0} |
| 1 |  | {2} |
| 2 |  | {3} |
| 3\*(终止) |  |  |

这里对比便能够看出与DFA的区别，状态表中都是一个个集合，而非元素。由于一些状态转换函数在图中没有给出，因此用空集()表示。

这个NFA接收的字符串是什么呢，不难看出，它接收的同样是以*abb*结尾的字符串，当输入字符*a*时，它可能进入状态0，也可以进入状态1，这是不确定的，甚至再输入字符*b*时，它也可以是状态0，也可以是状态2；但这样的字符串是能够进入终止态(状态3)，虽然它也可以是状态0，但因为它能够被这个NFA接收，因此NFA应该接收这个字符串，因此转移到确定的状态3。

事实上，还有一种NFA，被称为带边的NFA()，它与一般的NFA不同在于，它可以从某个状态不消耗符号，直接转移到下一个状态。

##### (f). DFA与NFA的等价性

从上面的两个例子中，可以看出DFA和NFA似乎可以表示同一种接收字符。事实上，二者表达能力是一样的，两者是等价的，每一个DFA可以转化为NFA;反之亦然。为什么要讨论DFA和NFA的等价性，因为直观上，NFA的表述形式我们更容易理解，但在计算机编程实现上，显然DFA更容易实现，在实际编程种，我们也采用了DFA的形式。一般而言，在编程中，会首先书写正则表达式，然后将正则表达式转化为NFA，最后转化为DFA。**二者等价性严格证明略**。虽然定理的证明略显复杂，但其应用很容易理解，还是举例子说明。

正则表达式转化为NFA规则如下:

* 对应的NFA，如图6所示。此时从初始状态不需要接收任何字符，便可以转移到终止状态。表示空串。

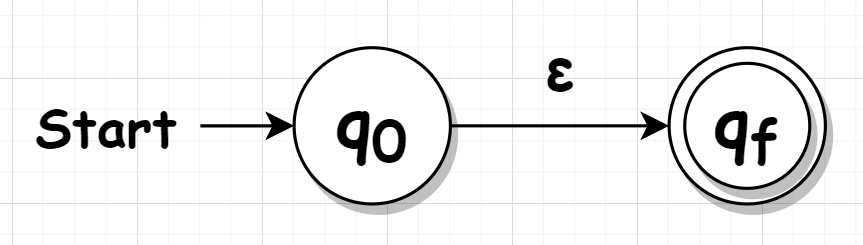


图6 对应的NFA示意图

* 对于，其对应的NFA如图7所示。此时初始状态接收一个字符，便转移到终止状态。

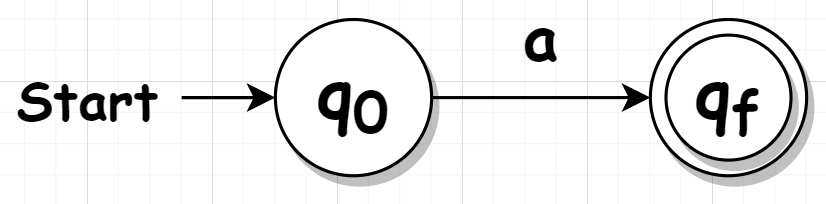


图7 符号对应的NFA示意图

* 对于这样的正则表达式，其对应的NFA如图8所示。由于是串行排列，表示先匹配再匹配。

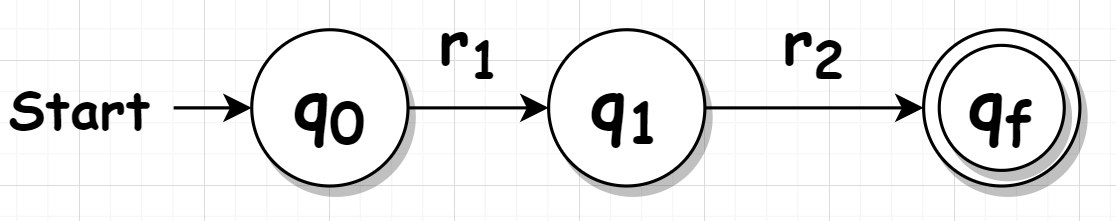


图8 对应的NFA

* 类似的，对于这样的正则表达式，其表示一种或的关系，所以两个状态之间是并行排列，如图9所示。

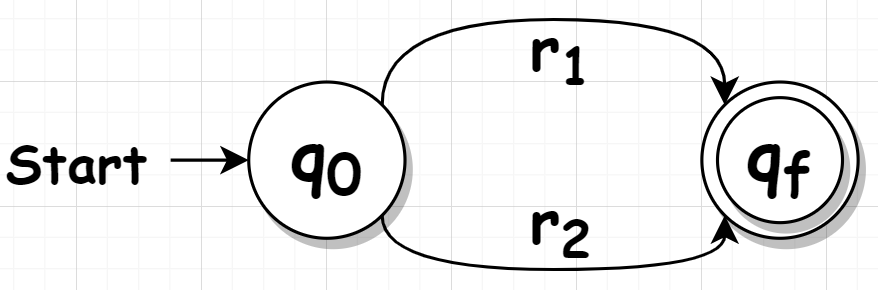


图9 对应的NFA

* ，一个正则表达式的克林闭包(Kleene)[5]依旧是一个正则表达式。简单理解就是重复0次或者1次或者多次；或者表述为将若干个连接起来，因此其对应的NFA如图10所示。

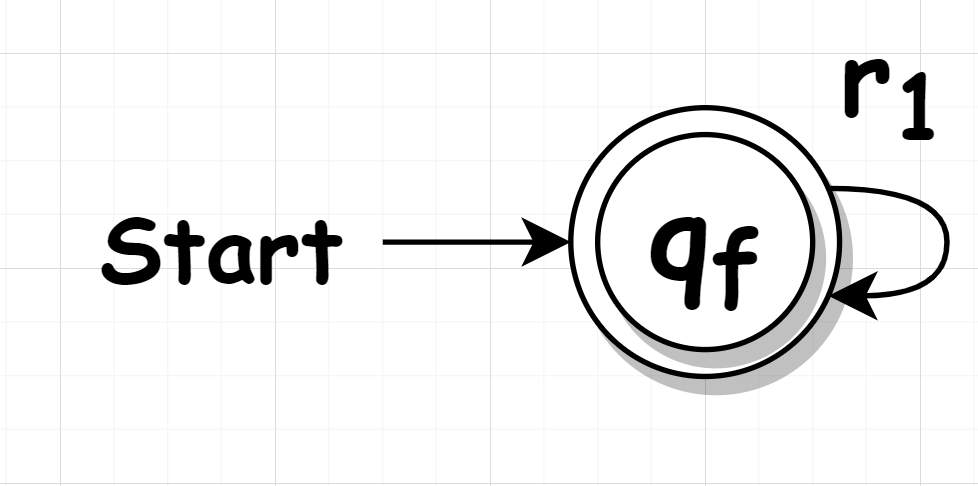


图10 对应的NFA

以刚才的正则表达式为例，要画出其对应的NFA，可以根据上述的规则，一步一步转换。如图11所示。

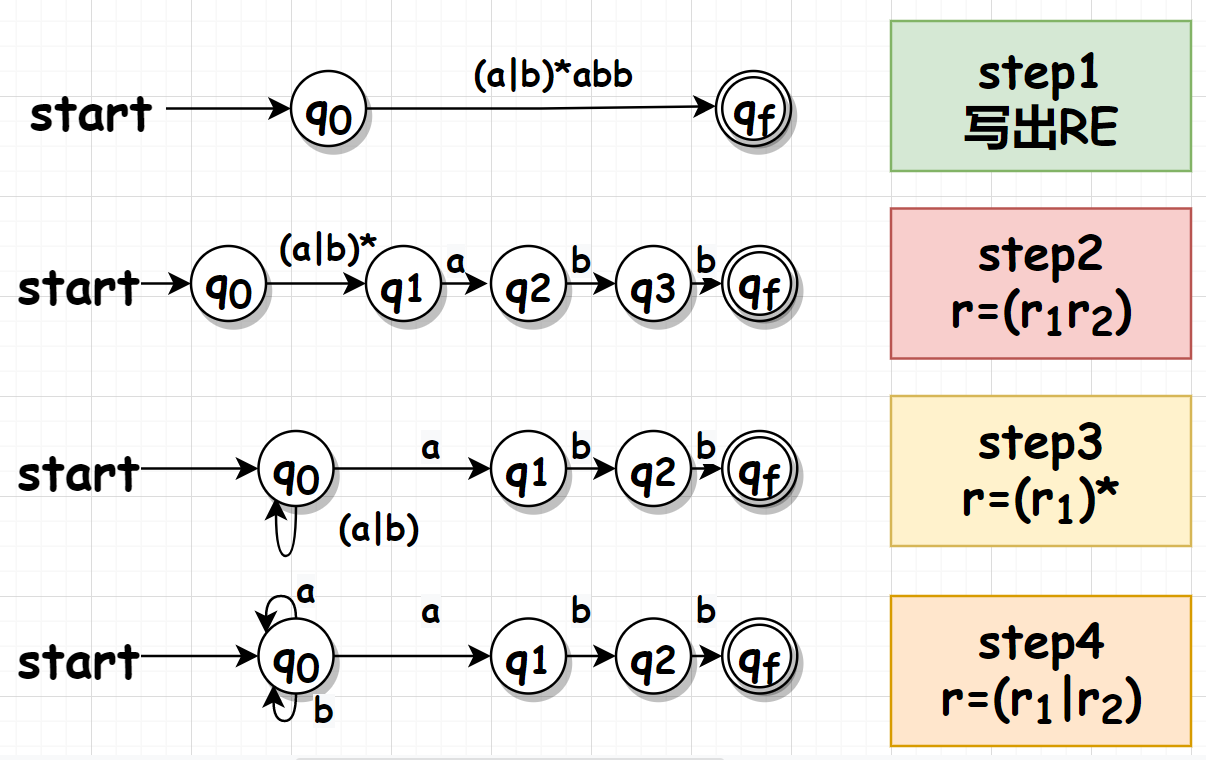


图11 转换为NFA过程

从NFA到DFA的转换，不妨也使用这个例子。如图12所示。

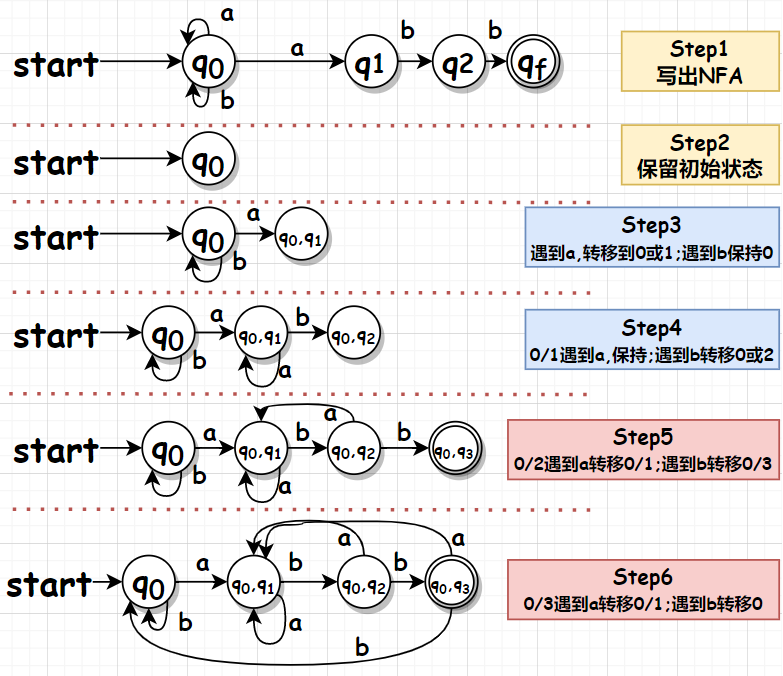


图12 NFA转DFA过程

从NFA到DFA的转换过程中，我们需要参考转换表，也就是表3。为了方便查阅，这里我们再次列出，如表4所示。

表4 NFA状态转换表

|  |  |  |
| --- | --- | --- |
| **输入 状态** |  |  |
| (起始) | {,} | {} |
|  |  | {} |
|  |  | {} |
| (终止) |  |  |

* 从初始状态开始，也就是开始，若输入字符为*a*，则查阅表4的第一行一列(简写为(1,1),下同)，其可能转移到或者。于是这里给出一个新的状态，用于表示这种中间状态；若输入*b*，则保持。(对应Step3)
* 从状态开始，若输入*a*，则查阅表4(1,1)和(2,1)，其依旧保持状态；而输入字符*b*，则查阅表4(1,2)和(2,2)，其可能转移到状态0，也可以是状态2，于是构造一个新的状态。(对应Step4)
* 从状态开始，若输入*a*，查阅表4(1,1)和(3,1)，其转移到状态；若输入*b*，查阅表4(1,2)和(3,2)，其可能转移到状态0或者状态3，因此构造一个新的状态。(对应Step5)
* 最后从状态开始，输入*a*，查阅表4(1,1)和(1,4)，其转移到状态；若输入*b*，查阅表4(1,2)和(4,2)，其转移到状态0。(对应Step6)

最后观察Step6得到的DFA结果，其实和最初我们绘制的图4是一样的！因此这样的方法能够得到等价的DFA。这个方法称为子集构造法(Subset Construction)[6]。

#### 2.1.3 编程设计

在实际设计中，我们需要解析的是C语言词法的一个很小的子集，所以对于上述理论部分，没有用到很复杂的部分，很多待识别的token都是一步到位。例如加减乘除的符号以及一些界符。这里标识符(ident)和注释(多行注释)会相对复杂一点，对于标识符的识别，前文已经提及过；而众多关键字void/int这些不过是一类特殊的标识符而已。对于注释的识别，这里给出一个DFA的示意图，如图13所示。

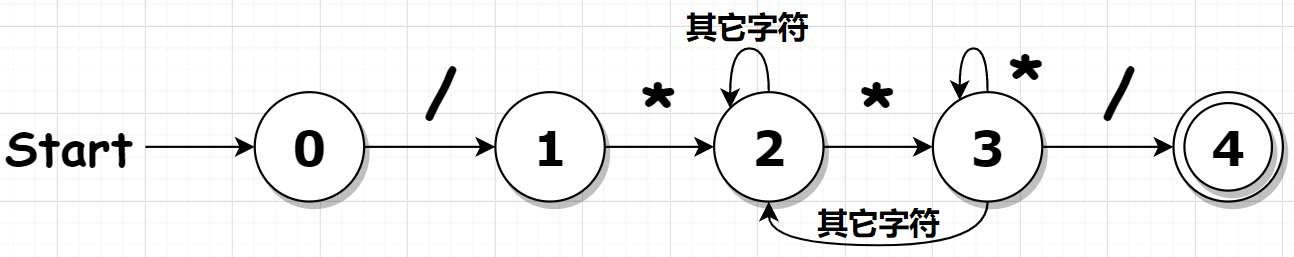


图13 C语言多行注释识别对应的DFA

词法提取的框架程序p2所示。

*.e.g.*

// 提取token函数 p2

*void GetSymbol(){*

*while(current\_char == ‘ ’ || current\_char == ‘\n’ || current\_char == 9){*

*if(current\_char >= ‘a’ && current\_char <= ‘z’ ||*

*current\_char >= ‘A’ && current\_char <= ‘Z’ ||*

*current\_char == ‘\_’){*

**// 标识符ident**

*}else if(current\_char >= ’0’ && current\_char <= ‘9’){*

**// 数字 number**

*}else{*

*switch(current\_char){*

*case ‘+’:* **//+识别**

*case ‘-’:* **//-识别**

*… …*

*default:* **//非法字符**

*} // end switch*

*} // end else*

*} // end while*

*}*

函数每次识别出一个token，并标记这个token所属的类别和值，出现非法字符时，需要跳过，并给出适当的错误提示；同样，若一次识别过程中，没有办法抵达终止态，如字符串缺少了右引号，需要采用一定的恢复策略，例如给予用户一定的提示，然后重置识别器到初始状态，开启下一个token的识别。

这一阶段关注的问题时正确识别出输入程序中的token序列，至于是否是合法的语句，或者有误语义错误，这些都不是这一阶段关注的问题！一个函数完成它应该完成的任务即可。

另外，在实际编程中，可能有边界的问题，例如扫描到文件末尾依旧未匹配，此时不能再读取下一个字符，否则会产生越界访问等问题，这些与上述理论部分关系不大，属于实际编程中需要注意的边界问题。

词法分析部分对应于源文件中**Compiler文件夹下Lexer类**，这个类会对给定的文本文件进行词法分析，也可以输出词法分析的结果。完成了词法分析后，可以单独对词法分析模块进行测试，一方面可以测试程序输出的词法分析结果是否和预期的一样；另外一方面是对程序的健壮性进行测试，面对非法的用户输入，甚至是精心构造的恶意输入，程序是否能够给出必要的错误信息，并正常退出。

我们给出的词法分析器，分析的词法相当有限，一些C支持的高级语法，例如结构体、指针等等，其实并不困难，但后续对它们汇编和进一步目标代码生成会有一定的难度，因此也就没有过于细化的考虑；另外对于C支持的位运算，其实质和四则运算没有区别，并不影响后续的扩展，因此在源程序中也没有全部分析和实现。

最后我们给出源程序能够识别的词法表，如表5所示。

表5 词法分析表

|  |  |
| --- | --- |
| **Token** | **Value** |
| ident(保留字) | int、void、char、string、while、if  else、in、out、return、extern、continue |
| ident | 变量名 |
| 比较运算符 | >、<、>=、<=、==、!=、 |
| number(数字) |  |
| char(字符) |  |
| strings(字符串) | (.\*代表任意字符；红色表示按字符识别，下同) |
| 数值运算符 | 、、、 |
| Assign(赋值) | = |
| 界符 | (、)、{、}、,、; |
| 注释 |  |

至此，词法分析部分结束。

### 2.2 语法分析(GrammarAnalyse)

语法分析部分是对这些token做进一步的分析，找出它们的内在逻辑结构。事实上，语法分析和我们学英语的短语结构很类似，一个个单词可以看作token，单词也有不同的词性；它们如何组装成“符合要求”的短语，那些语法是符合要求的，这些问题都是语法分析部分要解决的问题。

#### 2.2.1 感性认识

语法分析部分的理论相对还要复杂一点，但直观上，我们应该有如下的感受:对于下面的示例C语言程序p3，你一眼能够看出问题。这个源程序能够顺利通过词法分析，因为每一个token都是合法的；但在语法分析上，语法分析器认为最后一个return 0不是一个完整的语句，因为缺少了分号。

每条语句结束后，需要添加分号作为语句结束的标志，这就是C语言设计的一条语法规则。

*.e.g.*

// 示例程序p3

*int main(){*

*printf(“hello Compiler”);*

*return 0*

*}*

在实际的自然语言中，我们可以给出一个简化语言规则模型，例如:

*p*1: **<句子> <名词短语><动词短语>**

*p*2:**<名词短语> <形容词短语><名词短语>**

*p*3:**<名词短语> <名词>**

*p*4:**<动词短语> <动词><名词短语>**

*p*5:**<形容词> 小**

*p*6:**<动词> 吃**

*p*7:**<名词> 猫 | 鱼**

那么句子“小猫吃鱼”就符合上面的句子规则，同样“猫吃小鱼”也符合上面的句子规则，类似地，我们还可以写出“小鱼吃猫”和“鱼吃小猫”等。虽然有些句子在我们看来不符合逻辑，但这个并不属于语法分析的阶段，这个是语义部分要解决的内容。一个句子为什么符合上面的文法规则，后续理论部分还会用到这个例子解释说明。

从上面的例子中，那些不能再分割的单元，例如“猫”、“鱼”等是构成语言的基本元素；而可以被继续解析和分割的单元，即被<>括起来的部分，是语法的成分。

#### 2.2.2 理论部分

语法分析的本质是从词法分析器输出的token序列中，识别它们组合出的短语，并构造语法分析树(parser tree)。

##### (a).部分概念

* **字母表(Alphabet)**

字母表是一个**有穷**符号集合，这个字母表和英文的字母表略有不同，这里是数学上的概念，其实就是一个包含有限元素的一个集合。每个元素我们称为一个串(String)。

* **字母表乘积运算(Product)**

给定两个字母表和，它们的乘积运算可以表示为:

这其实有点类似于笛卡尔乘积[7]，只不过笛卡尔乘积展现的形式是有序对，本质上它们都是集合上的乘法运算。这里给一个例子:

{*0,1*} {*f,g*}*=*{*af,0g,1f,1g*}

* **字母表的幂运算(Power)**

幂运算其实就是一种特殊的乘法，有如下定义:

这其实就是一个递归的乘法定义式，其运算结果同样类似于笛卡尔乘积。例子如下:

={000,001,…,111}

* **字母表的正闭包(Positive Closure)**

正闭包的定义如下:

字母表的正闭包其实就是长度为正的字符串构成的集合，这个集合的元素应该是无穷的。例如。

* **字母表的克林闭包(Kleene Closure)**

字母表的克林闭包其实就是正闭包并上，即。其实比正闭包多了一个空串。例如

##### (b). 文法形式化定义(Grammar)

文法和这里我们说的语法因该是一个意思，在不引发歧义时，可以混用。经过上面自然语言的理解，文法(Grammar)其实在数学上也有严格定义，如下四元组:

* **:终结符集合(Terminal Symbol)**

终结符其实是文法定义语言的基本符号，有时候也被称为token。在之前的例子中，那些名词(**猫、鱼**)、形容词(**小**)以及动词(**吃**)等词语都属于终结符集合。而在实际的C语言中，那些我们看到的不可再拆分的单元，如保留字int、一个数字等这些也属于终结符集合，注意这里的是一个**集合**。

* **:非终结符集合(Nonterminal Symbol)**

与终结符对应，那些表示句子成分的符号，即被尖括号括起来的成分，都属于非终结符集合，它们有待进一步解析，注意这里的同样是一个**集合**。既然有这样一个分类，那么必然也有以下关系:

以及，

文法符号集合

即两个集合是对立的，互不相交，构成了整个文法符号。

* **:产生式集合(Production Set)**

产生式描述的是终结符和非终结符组合成串(String)的方法，其一般形式为:

即定义为，被称为的候选式(candidate)。既然和与终结符和非终结符有关，则；。由于定义为，那么至少包含一个非终结符。在上述例子中，每一条规则都是一个产生式。

* **:开始符号(Start Symbol)**

显然，。是一个特殊的非终结符，因为它是文法推导的开始，它也表示了文法中最大的成分。在上述案例中，就是<句子>。

文法定义完成后，再看看编程中的一个例子。最经典的是四则运算的文法，其中产生式的定义如下:

这里我们可以看到，非终结符只有一个，即，其它都是终结符；产生式集合有4个元素，其含义也不难理解，一个表达式(Expression,E)可以定义为表达式之和，或者表达式的乘积或者带括号的表达式，当然一个数字本身也可以被定义为一个表达式。

这里的产生式左部其实相同，即对于这样的产生式子，可以简记为:

##### (c). 推导(Derivations)及归约(Reductions)

在一个文法，如果有。对于一个符号串，可以将其中的替换为，于是符号串被重写为。记为。这个过程称为文法符号串的直接推导(directly derive)。

其实上面的直接推导过程，就是**用产生式的右部()替换产生式的左部()**。那么既然有直接推导，也可以多次间接推导，例如，那么经过步推导得出。如果，其实就是直接推导，当然也必然成立，相当于没有推导。

类似的，可以用(正闭包)表示经过正数步的推导；(克林闭包)表示经过若干步推导(包含0步)。

再回到“小猫吃鱼”的例子。

我们可以尝试进行文法的推导。从开始符号<句子>进行推导。

* Step1 **<句子> <名词短语><动词短语>**

接下来对<名词短语>做进一步推导，使用*p*2规则:

* Step2 **<形容词短语><名词短语><动词短语>**

接下来对<形容词短语>做进一步推导，使用*p*5规则，将其用“小”替换。

* Step3 小**<名词短语><动词短语>**

接下来，对<名词短语>做进一步推导，使用*p*3规则:

* Step4 小**<名词><动词短语>**

紧接着，对<名词>做进一步推导，使用*p*7规则:

* Step5 小猫**<动词短语>**

使用*p*4规则，对<动词短语>进行推导:

* Step6 小猫**<动词><名词短语>**

使用*p*6规则，对<动词>进行推导:

* Step7 小猫吃**<名词短语>**

继续使用*p*3和*p*7规则，最后能够得到完整的“小猫吃鱼”的句子。那么从Step1到结束得到句子“小猫吃鱼”的过程称为推导；反之从句子出发，判断其是否属于这个文法，自底向上的过程称为归约。这是一对逆过程，类似于微分和积分的关系。即归约的过程是用产生式的左部替换产生式的右部。

这个地方需要强调的是，比如在Step5的推导过程中，为什么选择了*p*7规则中的终结符“猫”，而没有选择终结符“狗”。这里需要解释:

如果你是应用上述文法产生句子，那没有特殊规定的情况下，自然可以选择任意合法的推导，而上述推导只是一个例子，之前我们也提到过，满足上述推导的句子不止一个，你甚至可以尝试写出其它满足这个语言的句子(无需考虑语义)。

一般而言，从生成语言的角度，使用的是推导；而从语言识别的角度，判断一个句子是否属于一个语言，采用的是归约的方式。

##### (d). 句型(Sentential Form)和句子(Sentence)

在数学上，对于句子和句型其实也有明确的定义，还记得文法四元组中的开始符号吧，如果经过若干步推导出一个串，并且是中的元素，那么称是文法的一个**句型**。

本质就是一个符号串，既有可能有终结符也可能包含非终结符，由于，后者是文法符号集合的克林闭包，那么还可以是空串。

特别地，如果中只有终结符，即，那么此时的称为文法的一个**句子**。

根据上述的定义，可以看出**句子其实是一种特殊的句型**，它不包含非终结符。

通俗来讲，句子是一个完整的结构，而句型更像是一种框架和中间产物，例如我们常说的一些搭配，“xxx是xxx”，其中xxx是名词，这个就是一个句型；而如果把xxx具体化，那么它就被称为了一个句子。

在我们讲述的“小猫吃鱼”的案例中，前面的推导过程结果产生的都是句型，只有最后一步的推导，得到的“小猫吃鱼”才是一个句子。

##### (e). 语言(Language)

根据文法推导出的所有句子，构成了一个集合，我们把这个集合称为文法产生的语言，记做。

对于刚才的“小猫吃鱼”例子中，我们能够得到很多句子，例如“小猫吃鱼”、“猫吃小鱼”、“小鱼吃猫”、“鱼吃小猫”、“猫吃小猫”、“猫吃猫”等等，事实上能产生的句子的数量是无穷的，即这个集合的数量是无穷的；例如“猫吃小小鱼”这样的句子也是符合上述文法的！而对于之前所列举的算数表达式的文法，其产生的句子也是无穷的。

一般对于无穷的集合我们很难把它说明清楚，但是文法集合的数量往往是有限且较少的，因此**文法的意义在于解决了无穷语言的有穷表示问题**！

##### (f). 文法分类

文法分为以下四类:分别是0型文法、1型文法、2型文法和3型文法。这是乔姆斯基文法分类体系(Chomsky)[8]。不同文法的限制是不一样的，学过数据库的同学们应该了解，我们谈到的第一范式、第二范式以及BC范式(BCNF)等等是一个道理，不同的范式规范级别不一样，不同的文法也是这个道理。越高等级的范式其限制越多，但是拥有良好的性质也越多，但范式也不是越高越好，文法也是类似的道理。

这与离散数学中提到的群、半群、阿贝尔群，似乎也有些类似，将一部分的元素抽取出来，它们构成一个新的集合，这个集合拥有更多的良好的性质。

* **0型文法(Type-0 Grammaer)**

0型文法的限制最少，它被称为无限制文法，或者也叫短语结构文法(Phrase Structure Gramma PSG)，类似于第一范式，范围最广，但限制最少。其文法要求是，，至少包含一个非终结符即可。

由0型文法生成的语言称为0型语言，记为L(G)。

* **1型文法(Type-1 Grammaer)**

1型文法称为上下文有关文法(Context-Sensitive Grammar,CSG)，其定义为:

其实就是在0型文法的基础上，对产生式进行限制，要求产生式的左部符号个数不得超过右部中符号个数。

这一类文法产生式的形式多为，即在上下文为和的情况下，可以把替换为；也就是说替换的过程需要依赖于上下文，所以也称为上下文有关文法。

从1型文法定义中也能看出CSG不包含产生式，原因是的限制，，而至少包含一个非终结符，即，显然不等式不可能成立。

由上下文有关文法产生的语言称为上下文有关语言。

* **2型文法(Type-2 Grammaer)**

2型文法也称为上下文无关文法(Context-Free Grammar,CFG)，定义为:

这要求其产生式的左部必须是一个非终结符，其产生式形式多为类型。即定义(这里*A*表示非终结符)，将替换为无需考虑上下文，所以也称为上下文无关文法(CFG)。

同理，由上下文无关文法生成的语言称为上下文无关语言。

* **3型文法(Type-3 Grammaer)**

3型文法又称为正则文法(Regular Grammar,RG)，又分为两种，分别是右线性(Right Linear)文法和左线性文法(Left Linear)。

右线性文法形如或者，右线性文法其实是在2型文法基础上，对产生式右部做进一步限制，要求右侧是一个终结符，或者是终结符右侧添加一个非终结符*B*。

而左线性文法形如或者。其也是类似的道理，要求产生式子的右侧是终结符，或者是终结符左侧添加一个非终结符*B*。

简单概括来说，正则文法要求产生式的右部只有一个终结符，并且只能在同一侧，不能一会儿左一会儿右。如图14所示，这是一个右线性文法。

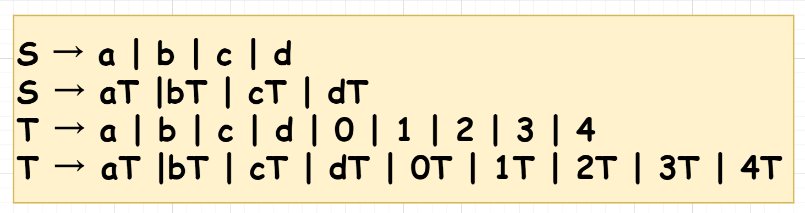


图14 右线性文法示例

首先这个文法产生式左侧都是一个非终结符，所以首先符合2型文法，其次其右侧要么是终结符，要么终结符都在非终结符的右侧，所以是右线性文法。这里大写字母*S、T*都是非终结符，而小写字母和数字都是终结符。

这其实就是一个标识符的文法，最后产生的句子都是字母开头，由字母和数字组成的串。

总结:四级文法是逐步限制的关系，越来越严格；同时它们也是真子集关系。即0型文法集合包含1型文法集合；1型文法集合包含2型文法集合；2型文法集合包含了3型文法集合。

##### (f). CFG文法分析树

CFG文法是被研究的较多的一类文法，我们的编译器也使用了这一文法。为了引入概念，首先让我们来看一棵语法分析树的样子，如图15所示。从图中我们容易观察到几个现象:

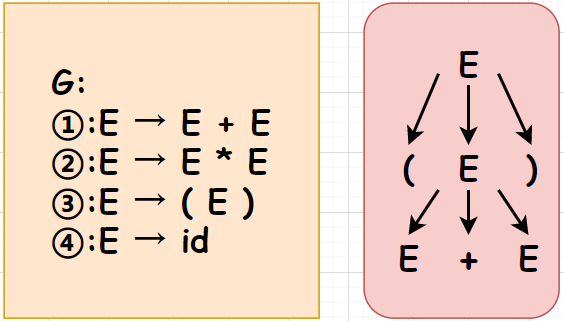


图15 语法分析树示例

* 根结点的符号是文法开始的符号，也就是*E*
* 每一个内部结点其实是对某个产生式的应用

该结点是这个产生式的左部*A*，其子结点是产生式的右部。例如在语法树的第2层，其实就是使用了第③个推导式，因此其结点为E(产生式的左部)，它的三个孩子(、E、)是产生式的右部。

* 叶结点既可以是终结符也可以是非终结符

从左向右排列的**叶结点**构成的符号串称为是这棵树的**产出**(yeid)，或者也称为**边缘**(frontier)。例如这棵树的产出其实就是(E+E)。

对于推导过程中的每一个句型，都可以构造出一个**边缘**为的分析树。例如在上述推导中产生的句型和，在推导过程中都可以构造出边缘为的分析树。如图16所示。其中蓝色部分表示分析树的边缘。

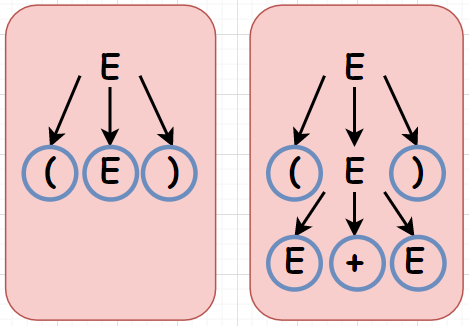


图16 推导过程中分析树对应的边缘

那么给定一个句型，其分析树中每一棵**子树**的边缘称为该**句型**的一个短语(phrase)。如图17所示，还是这个文法*G*，给定一个句型对应的分析树。这个句型拥有的短语有(E\*E+E+E)、E\*E、E+E。

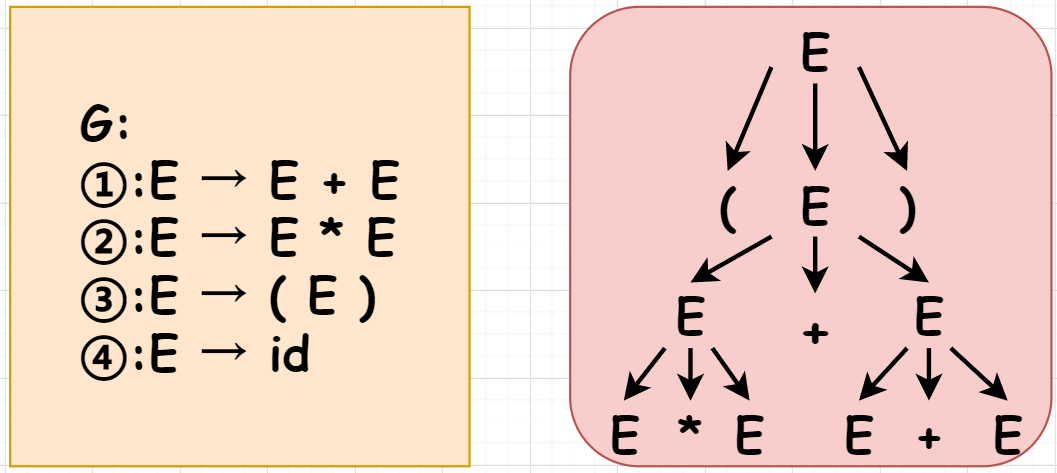


图17 文法G的某句型对应的分析树

如果一棵子树只有父子两代结点，即子树**高度为2**，那么这个子树的边缘称为该句型的直接短语(immediate phase)。例如这里的E\*E和E+E就是直接短语。

由于子树的高度为2，结合之前提到的“每一个内部结点都是对产生式的应用”，那么直接短语一定是某产生式子的右部。例如这里给出的两个短语分别是产生式②和产生式①的右部。**反之不成立**，即产生式的右部不一定是直接短语。

例如这里的(E)是产生式③的右部，但不是这个句型的直接短语，但它可以是其它句型的直接短语。

##### (g). 二义性文法(Ambiguous Gramma)

所谓二义性，其实就是一个文法可以为某个句型生产多颗分析树，这样的文法就称为二义性文法。

最典型的例子就是**else悬空问题**[9]。给定一个文法和一个它的句型，如图18所示。

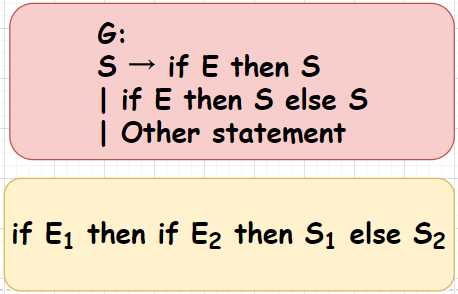


图18 *if-else*文法示例

这里我们能够根据这个文法，给出这个句型两棵不同的分析树，如图19所示。显然这个问题是我们遇到的else与哪个if匹配的问题！以C语言为例，为了消除这样的二义性，C语言规定else与最近的尚未匹配的if进行匹配，从而消除这样的二义性。

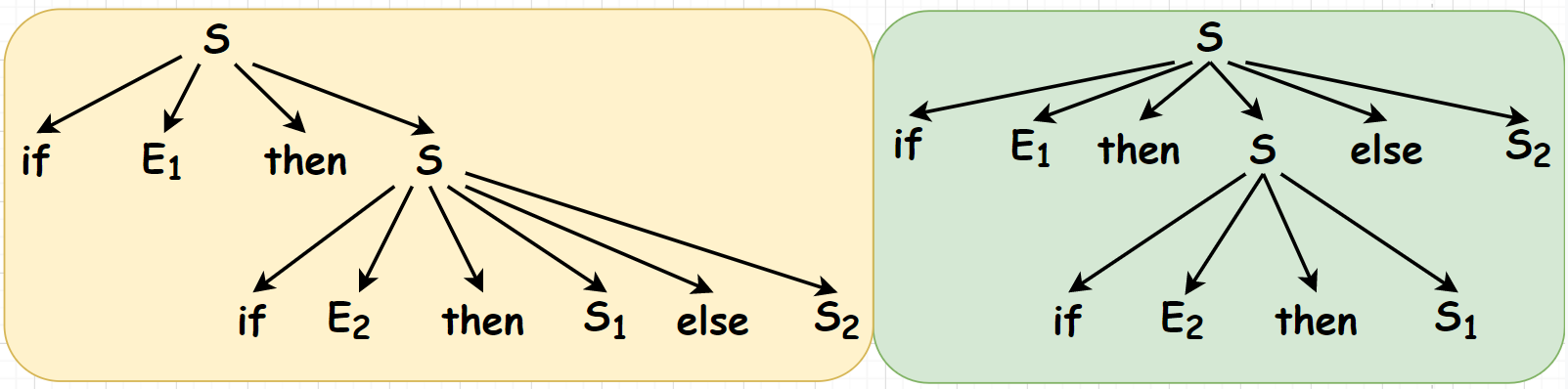


图19 句型对应的两棵分析树

事实上，为了消除二义性，许多程序设计语言做法与C语言类似，就近原则；或者通过从属关系的来控制，例如花括号不能省略，此时也可以消除二义性(本质是修改文法规则)；甚至一些语言要求else不可以缺失，这样也避免了二义性。

最后是文法二义性的**判断**，对于一个CFG，不存在一个算法判断它是否为无二义性文法；但存在一组**充分条件**，满足这个条件的文法是无二义性的文法，不满足这个条件的文法无法判断！**严格证明略**。

##### (h). 自顶向下分析(Top-Down Parsing)

看完了文法部分，或许你依旧对于分析过程不是很明白，下面是构造分析树的一些细节问题。我们程序会输入一个个字符串，我们需要遍历整个字符串的每一个token，不断推导出属于这个语言的句子。

首先还是一个简单的例子，自顶向下，顾名思义就是从开始符号S开始推导，从分析树的根结点开始构造分析树，如图20所示。

相当于模拟程序的输入字符串是(id+id)，我们需要不断应用产生式，进行自上而下的推导。

看起来似乎没有什么问题，我们也容易读懂最右侧的推导过程，但是这里面有几个问题还需要考虑，(1)每一步**替换哪个非终结符**；(2)该非终结符**用哪个候选式替换**。为了解决这个模糊的问题，需要引入几个简单的概念。

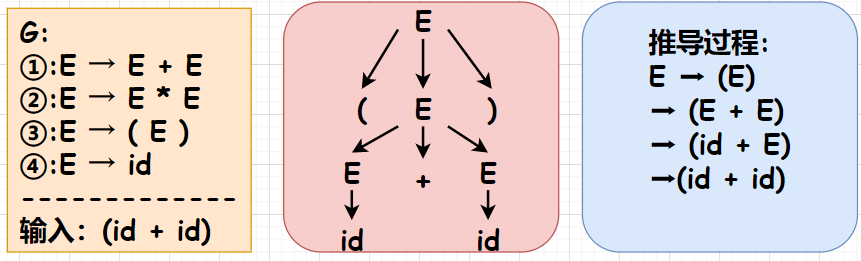


图20 给定字符串，文法推导过程示例

* **最左推导(Left-most Derivation)**

最左推导是指每次推导总是选择每个句型的最左非终结符进行替换。如图21所示，红色部分就是每一步选择推导的非终结符，它是这个句型的最左端的非终结符。最左推导的逆过程是最右归约。

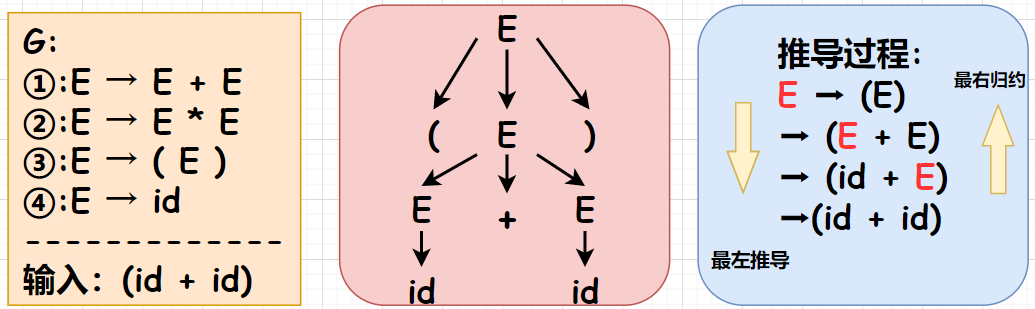


图21 最左推导示例

从文法开始符号推导出的句型称为最左句型(left-sentential form)。

* **最右推导(Right-most Derivation)**

类似的，还有最右推导的方式，其实就是每次都选择句型的最右非终结符进行替换。它的逆过程称为最左归约。

这两个推导方式回答了第一个问题:每一步替换哪个非终结符。一般而言，在**自底向上**的分析中，会采用最左归约的方式，因此最左归约是**规范归约**，与之对应的最右推导称为**规范推导**。而在**自顶向下**的分析中，采用最左推导的方式。

一个文法可以有很多种推导，但是最左推导和最右推导的结果是唯一的。

##### (i). 递归下降分析(Recursive-Decent Parsring)

计算机如何实现自顶向下的分析？这里我们采用递归下降的分析方式，最后编程也是用这种方式实现。

对于每一个非终结符，都有一个过程(process)与之对应，递归调用其它非终结符的过程，直到完成语法分析。但是我们只回答了第一个问题，对于第二个问题，**当一个非终结符有多个候选式时，我们如何选择**？

一种简单的方式是逐个尝试，直到尝试成功。但这一来，会存在回溯问题，分析器的效能会下降。导致不能确定选择哪个候选式子的原因其实很简单，就是目前扫描的字符串有限，得到的信息有限，因此不知道如何选择同一个非终结符的哪个候选式。

对于这样的问题，其实如果能多读取一些字符，自然也就知道它应该去往哪一个分支，因此有了预测分析(Predictive Parsing)。

预测分析其实是递归下降技术的一个特例，它通过对输入的字符串向前看(Look ahead)**固定个数**(例如1个)的符号来选择正确的非终结符*A*的产生式。

那么向前看一个符号得到的信息有限，也可以构造向前看*k*个符号的语法分析器，那么与之对应的文法称为LL(k)文法。

有了预测分析，我们就不需要回溯，因为每一步都是确定的，这也就回答了第二个问题:对于一个非终结符的多个产生式，如何选择。这里略有点抽象，我们给出一个小例子。如图22所示，对于这样三个不同的输入，程序需要向前看多少个token才能区分。这里给出了一个声明变量和函数的简单文法*G*，其中ident是标识符。

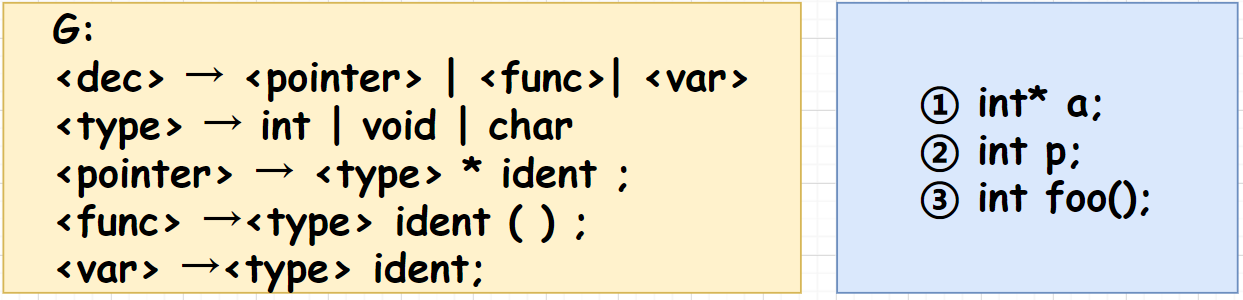


图22 向前看*K*个符号示例

我们不妨简单分析以下，对于程序扫描到int这个token后，它有三条分支可以进入，此时如果不向前看，那么不能确定选择1~3中的哪一个分支；

如果向前看1个token，即LL(1)文法，此时文法分析器能够区分①和②③，因为①中int的下一个字符是\*；而②③都是标识符(ident)，这样一旦向前看的字符是\*，则选择第一个分支，否则选择②③分支；

但是这样也区分不了②③，因为都是标识符，因此当指针指向int这个token时，需要向前看两个token，此时对于②来说是分号，对于③来说是左括号，此时才能够区分②③。

仔细观察文法*G*，产生不确定性的原因其实是因为有**公共前缀**<type>和<type>ident，因此如果不向前看便不知道选择哪一个分支。如果可以消除公共前缀，是不是意味者能够减少向前看的个数？

对于这样的文法，如果不希望回溯，则至少需要设计LL(2)文法才可以保证。后续会继续分析如何设计LL(k)文法，以及文法之间的转换，由于想使用递归下降的方式分析程序，这里会考虑到递归的出口问题，如何解决可能存在的无限递归问题，后续会进一步分析。

##### (j). 左递归(Left Recursive)及消除(Remove)

对于含有形式的产生式的文法称为**直接左递归**(immediate left recursive)文法。同样，如果经过多步推导产生了这样形式的产生式，，即，那么这个文法就是**左递归文法**。

左递归是导致文法分析器无限循环的根本原因。我们的算术表达式的文法中这类产生式，其实这就是一个左递归文法。如何消除左递归？

首先是消除直接左递归。

#### 2.2.3 编程设计

这里我们设计的文法如图X所示。

### 2.3 语义分析(Semantic)

### 2.4 汇编代码生成

## 3.汇编(Assemble)

### 3.1 词法分析(Lexer Analyse)

### 3.2 语法分析(Parser Analyse)

## 4.链接(Linker)

## 5.可执行文件(Executive File)

## Reference

1. https://zh.wikipedia.org/wiki/%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F
2. https://www.runoob.com/regexp/regexp-syntax.html
3. https://zh.wikipedia.org/wiki/%E7%A1%AE%E5%AE%9A%E6%9C%89%E9%99%90%E7%8A%B6%E6%80%81%E8%87%AA%E5%8A%A8%E6%9C%BA
4. https://zh.wikipedia.org/wiki/%E9%9D%9E%E7%A1%AE%E5%AE%9A%E6%9C%89%E9%99%90%E7%8A%B6%E6%80%81%E8%87%AA%E5%8A%A8%E6%9C%BA
5. https://zh.wikipedia.org/wiki/%E5%85%8B%E8%8E%B1%E5%B0%BC%E6%98%9F%E5%8F%B7
6. https://zh.wikipedia.org/wiki/%E5%B9%82%E9%9B%86%E6%9E%84%E9%80%A0
7. https://zh.wikipedia.org/wiki/%E7%AC%9B%E5%8D%A1%E5%84%BF%E7%A7%AF
8. https://zh.wikipedia.org/wiki/%E4%B9%94%E5%A7%86%E6%96%AF%E5%9F%BA%E8%B0%B1%E7%B3%BB
9. https://en.wikipedia.org/wiki/Dangling\_else