**Compiler document**

**content 目录**

[1.简介(introduction) 4](#_Toc99313798)

[2.编译(Compiler) 5](#_Toc99313799)

[2.1 词法分析(Lexer Analyse) 5](#_Toc99313800)

[2.1.1 感性认识 5](#_Toc99313801)

[2.1.2 理论部分 6](#_Toc99313802)

[(a).正则表达式(Regular Expression) 6](#_Toc99313803)

[(b).有穷自动机(Finite Automata,FA) 6](#_Toc99313804)

[(c).有穷自动机分类 8](#_Toc99313805)

[(d).DFA 8](#_Toc99313806)

[(e). NFA 9](#_Toc99313807)

[(f). DFA与NFA的等价性 10](#_Toc99313808)

[2.1.3 编程设计 12](#_Toc99313809)

[2.2 语法分析(GrammarAnalyse) 14](#_Toc99313810)

[2.2.1 感性认识 14](#_Toc99313811)

[2.2.2 理论部分 15](#_Toc99313812)

[(a).部分概念 15](#_Toc99313813)

[(b). 文法形式化定义(Grammar) 16](#_Toc99313814)

[(c). 推导(Derivations)及归约(Reductions) 17](#_Toc99313815)

[(d). 句型(Sentential Form)和句子(Sentence) 18](#_Toc99313816)

[(e). 语言(Language) 18](#_Toc99313817)

[(f). 文法分类 19](#_Toc99313818)

[(f). CFG文法分析树 20](#_Toc99313819)

[(g). 二义性文法(Ambiguous Gramma) 22](#_Toc99313820)

[(h). 自顶向下分析(Top-Down Parsing) 22](#_Toc99313821)

[(i). 递归下降分析(Recursive-Decent Parsring) 23](#_Toc99313822)

[(j). 左递归(Left Recursive)及消除(Remove) 25](#_Toc99313823)

[(k). 左公因子提取(Left Fctoring) 25](#_Toc99313824)

[(l). S\_文法 26](#_Toc99313825)

[(m). q\_文法 27](#_Toc99313826)

[(n). LL(1)文法 28](#_Toc99313827)

[2.2.3 编程设计 32](#_Toc99313828)

[(a).总体设计概述 33](#_Toc99313829)

[(b). 逐产生式分析 34](#_Toc99313830)

[2.3 语义分析(Semantic) 39](#_Toc99313831)

[2.4 汇编代码生成 40](#_Toc99313832)

[(a). 概述 40](#_Toc99313833)

[(b) if-else语句汇编 41](#_Toc99313834)

[(c). for 循环语句汇编 42](#_Toc99313835)

[(d). whie循环语句汇编 43](#_Toc99313836)

[(e). switch-case语句汇编 43](#_Toc99313837)

[3.汇编(Assemble) 45](#_Toc99313838)

[3.1 词法分析(Lexer Analyse) 45](#_Toc99313839)

[3.2 语法分析(Parser Analyse) 45](#_Toc99313840)

[(a). 概述 45](#_Toc99313841)

[(b). 逐产生式分析 46](#_Toc99313842)

[3.3 语义分析(Semantic Analyse) 48](#_Toc99313843)

[3.4 目标代码生成 48](#_Toc99313844)

[3.4.1 ELF 格式 48](#_Toc99313845)

[4.链接(Linker) 49](#_Toc99313846)

[5.可执行文件(Executive File) 49](#_Toc99313847)

[Reference 49](#_Toc99313848)

## 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。

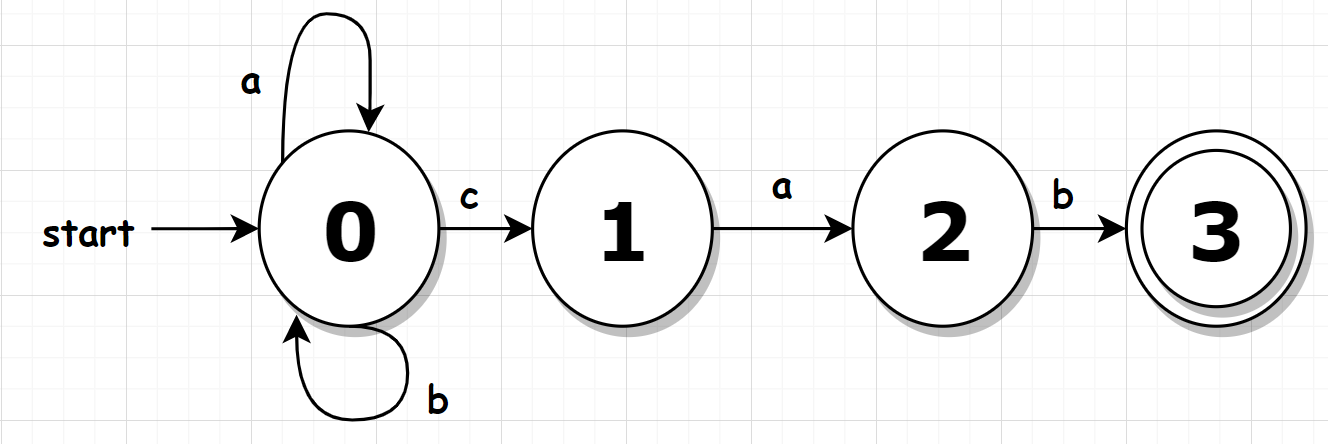


图1 FA 转换图(Transition Graph)

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。

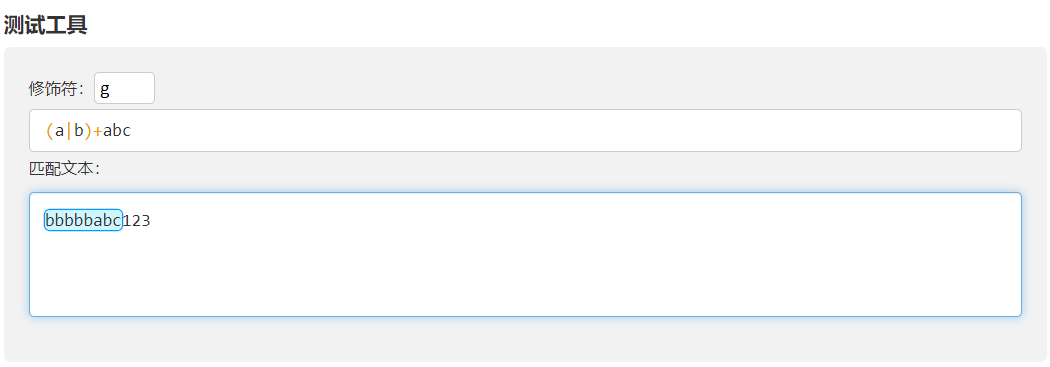


图2 在线平台测试结果

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。

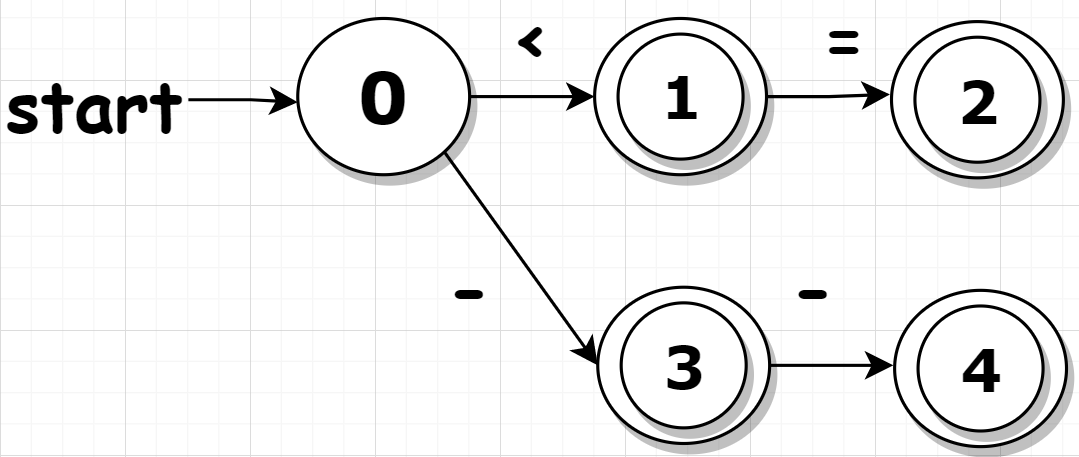


图3 最长匹配原则示例

##### (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。

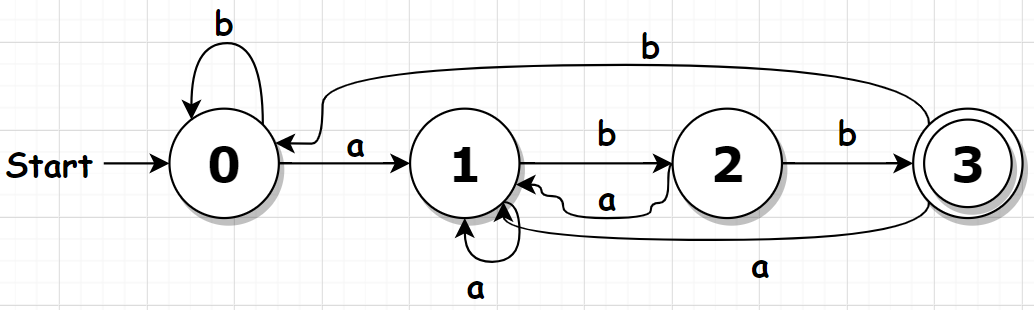


图4 DFA例子

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。

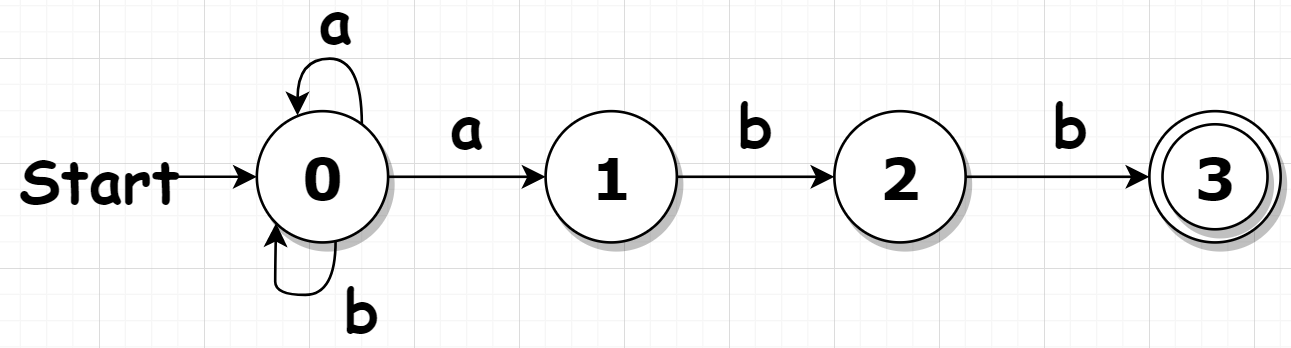


图5 NFA例子

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。

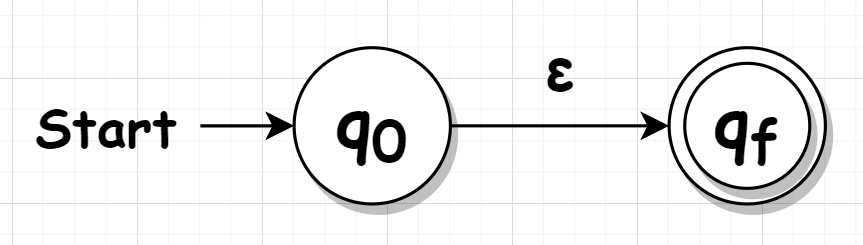


图6 对应的NFA示意图

* 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。

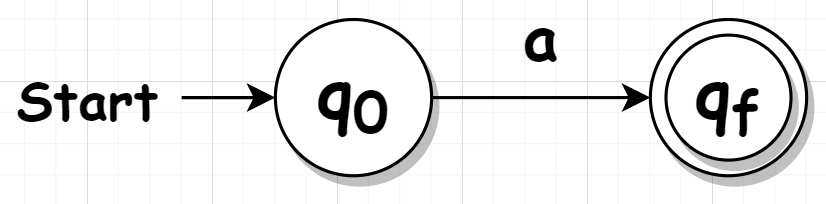


图7 符号对应的NFA示意图

* 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。

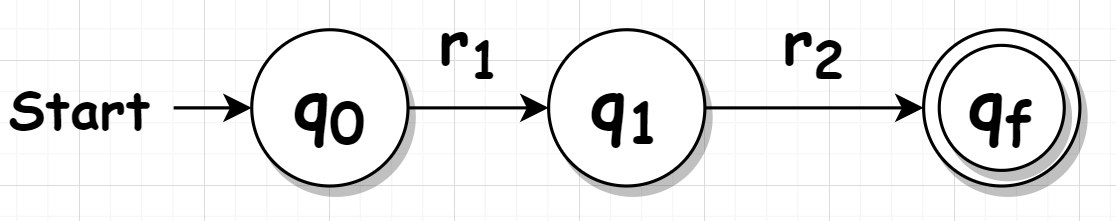


图8 对应的NFA

* 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。

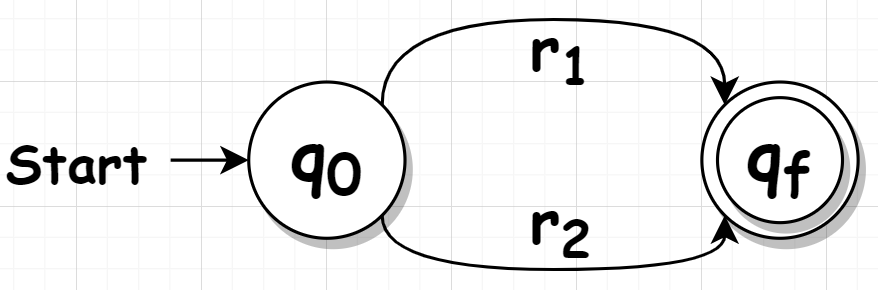


图9 对应的NFA

* ，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。

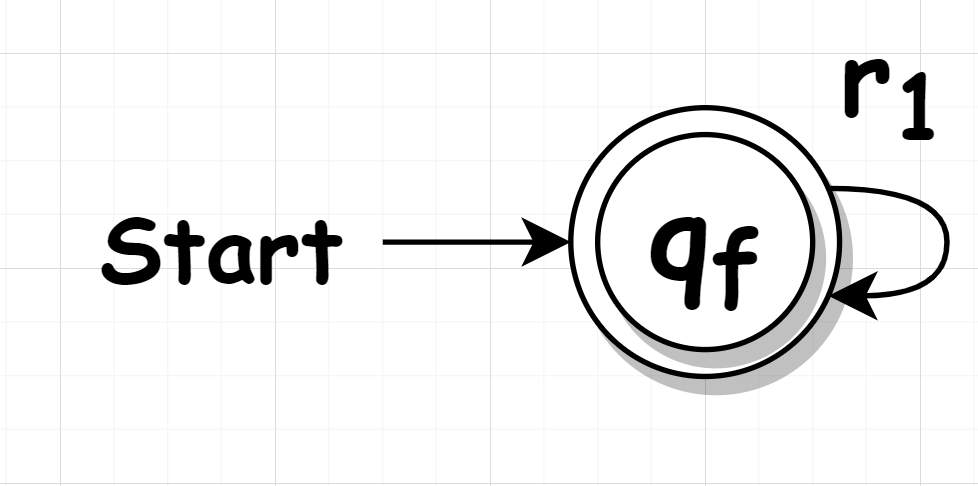


图10 对应的NFA

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。

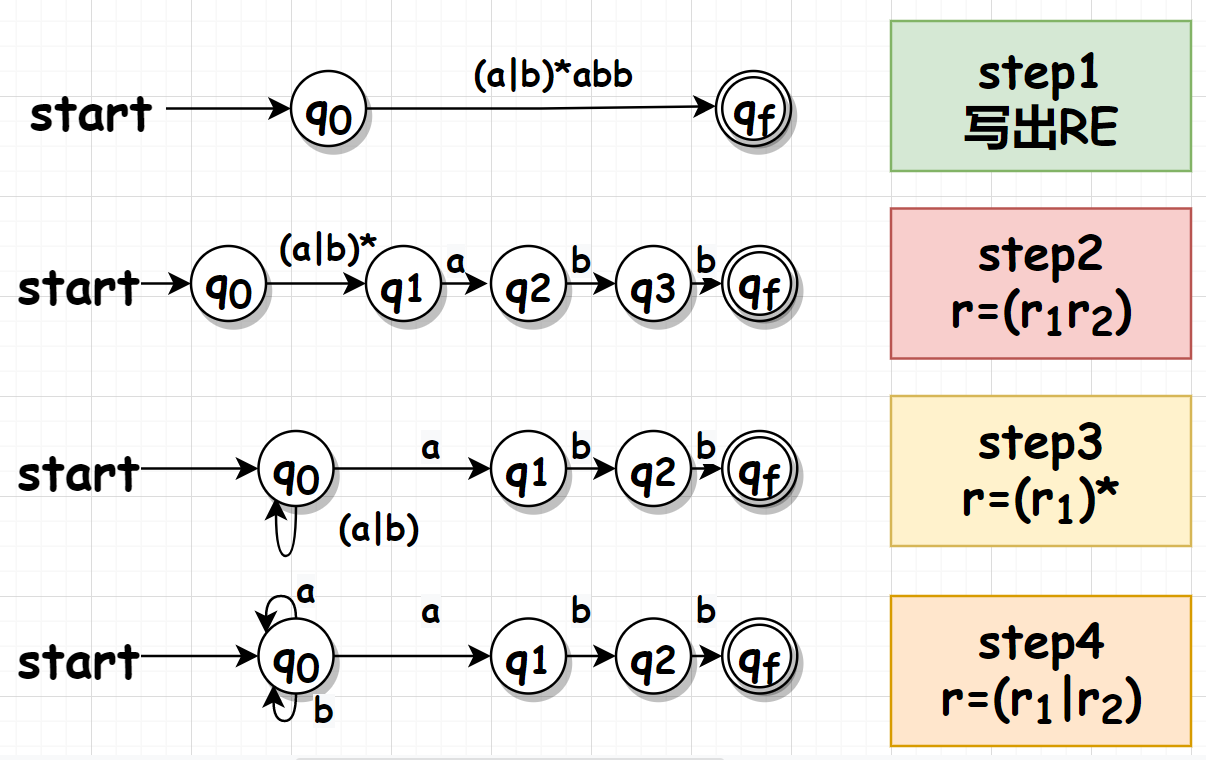


图11 转换为NFA过程

从NFA到DFA的转换，不妨也使用这个例子。如图12所示。

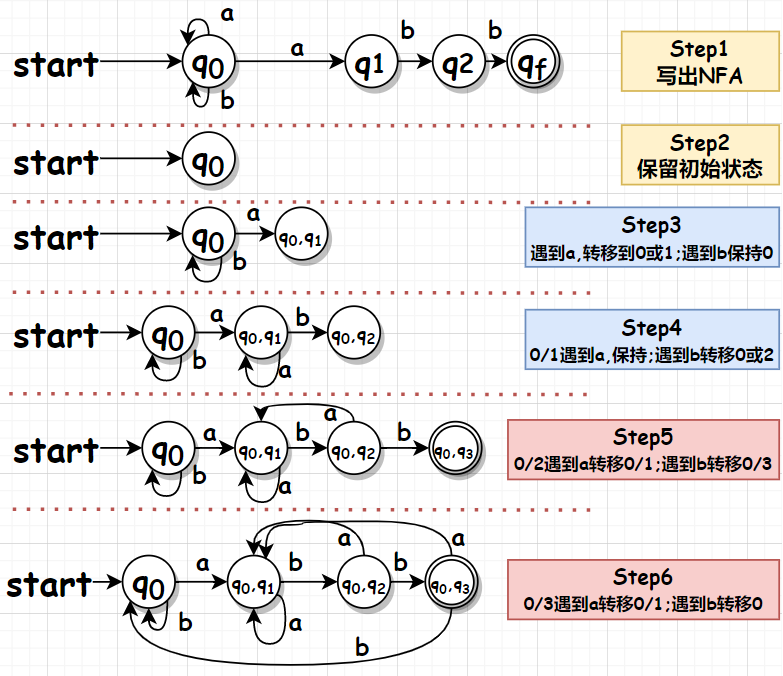


图12 NFA转DFA过程

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。

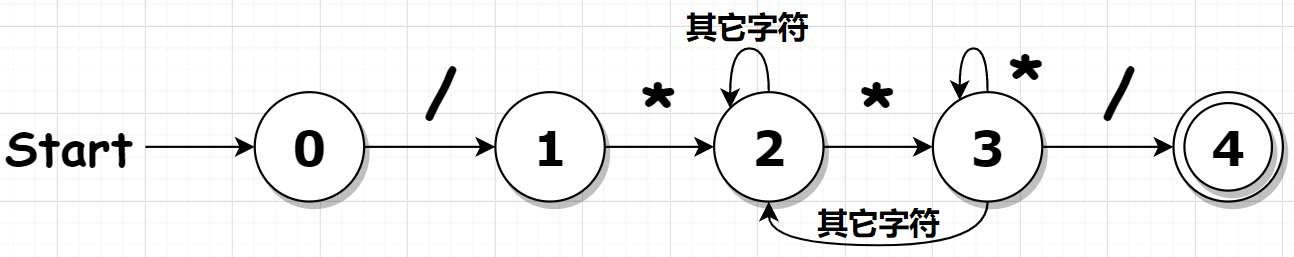


图13 C语言多行注释识别对应的DFA

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、for、switch、case、default |
| 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。

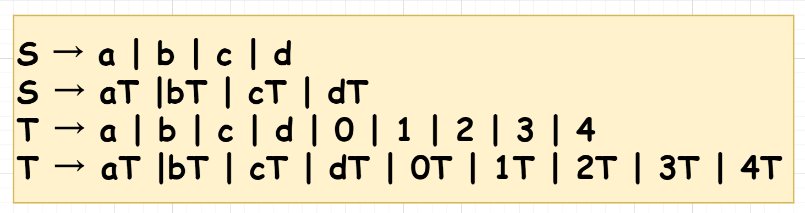


图14 右线性文法示例

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:

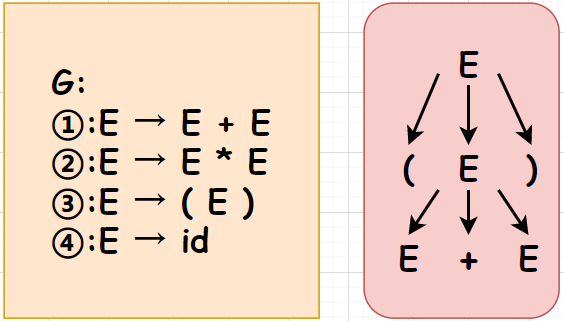


图15 语法分析树示例

* 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。

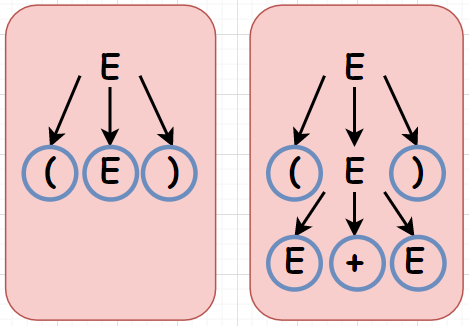


图16 推导过程中分析树对应的边缘

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。

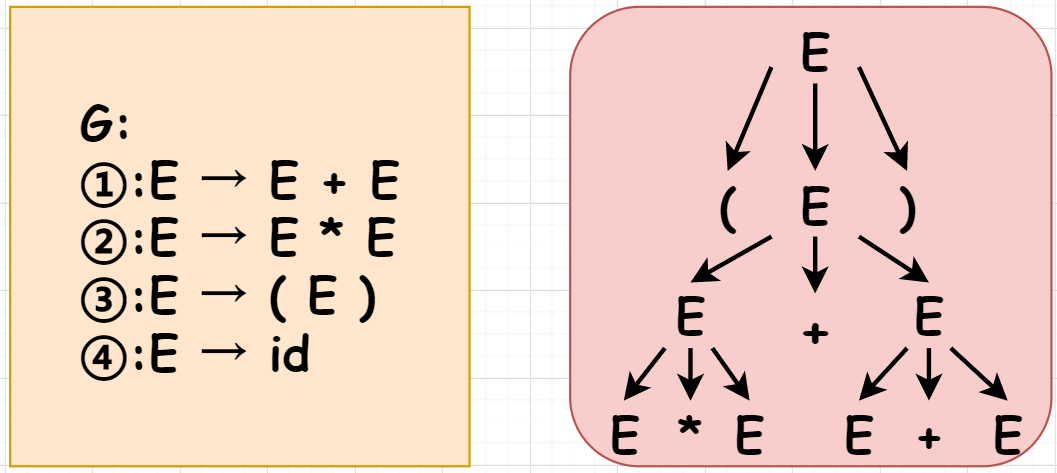


图17 文法G的某句型对应的分析树

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产生多颗分析树，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。

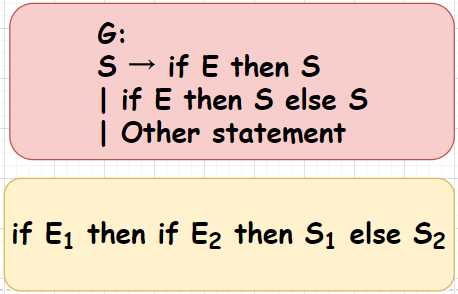


图18 *if-else*文法示例

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。

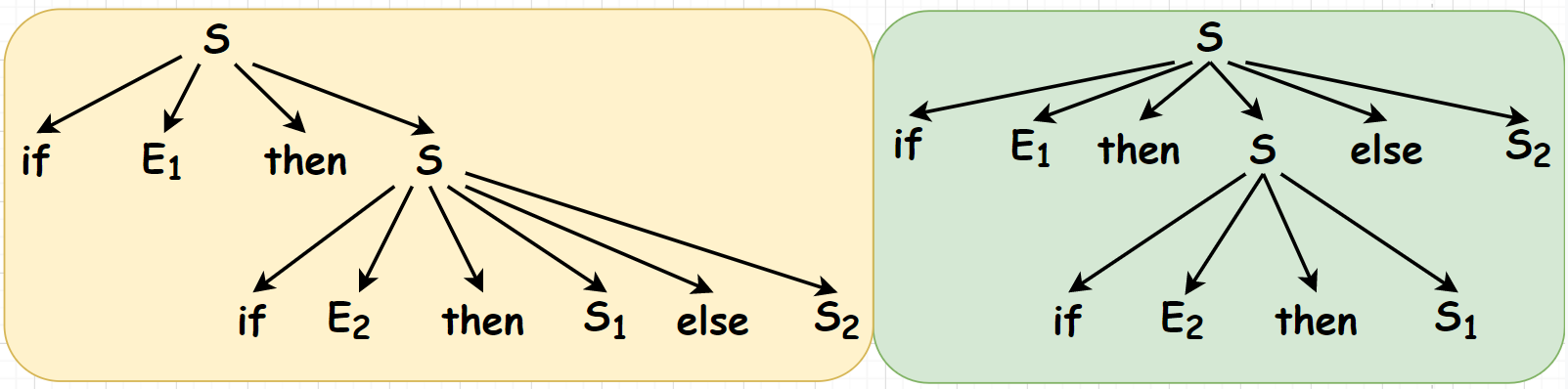


图19 句型对应的两棵分析树

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是否存在二义性。**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。

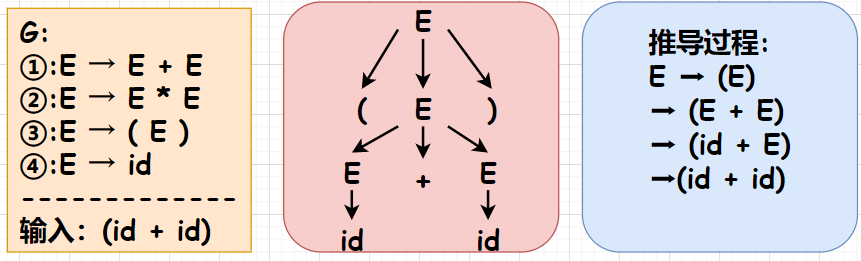


图20 给定字符串，文法推导过程示例

* **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。

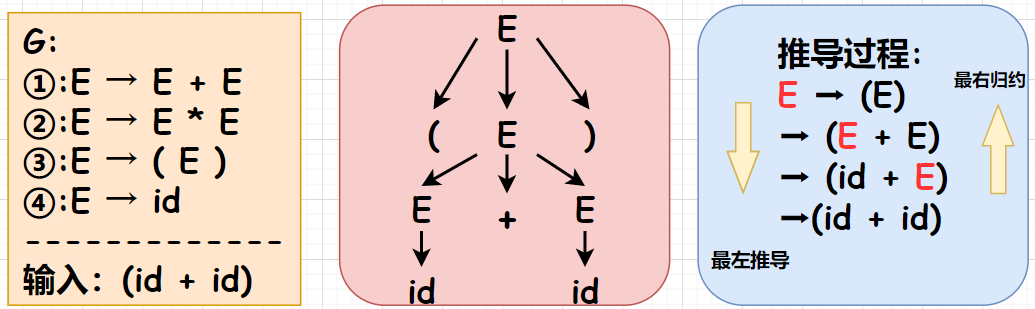


图21 最左推导示例

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。

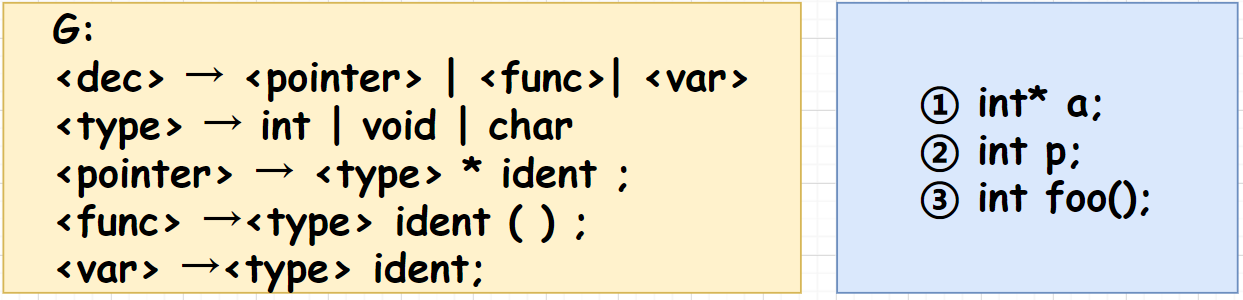


图22 向前看*K*个符号示例

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。

对于一个直接左递归的产生式，不以开头。这个产生式最后产生的符号串对应的正则表达式为。

为了消除直接左递归，我们不妨改写这个产生式，改写为:

这样一来，直接左递归就被消除了，上下两种产生式的正则表达式是一致的。这样消除的方式，其实就是把左递归变成了右递归的形式，即形式，这样程序能够根据输入的字符，判断出递归是否结束，以免造成循环递归。

对于更加一般的产生式而言，消除的直接左递归的方式如下:

不以开头，

只需要类似的转换为:

消除左递归是需要代价的，即我们引入了更多的非终结符和空产生式。

对于**间接**左递归的消除，例如有这样一个文法，如图23所示。



图23 间接左递归文法示例

给出的两个产生式都不存在直接左递归，但是带入以后，发现存在直接左递归，因此这里存在间接左递归文法。消除方式也是类似的，带入变为直接左递归后，再使用直接消除左递归的方法进行消除即可。**具体算法略**。

##### (k). 左公因子提取(Left Fctoring)

对于之前提到的回溯问题，对于不能决定使用非终结符的哪一个产生式，本质是因为这些非终结符的产生式有**公共前缀**(图22)。

这个时候，可以对左公因子进行提取，例如，我们还是以图22给出的例子为例，对比提取后的文法差异。如图24所示，通过把左公因子<type>和ident提取出来，这样程序每一步都无需回溯。

其实提取公共前缀的本质是延迟决定，程序需要读取足够的信息才能够走到对应的分支。

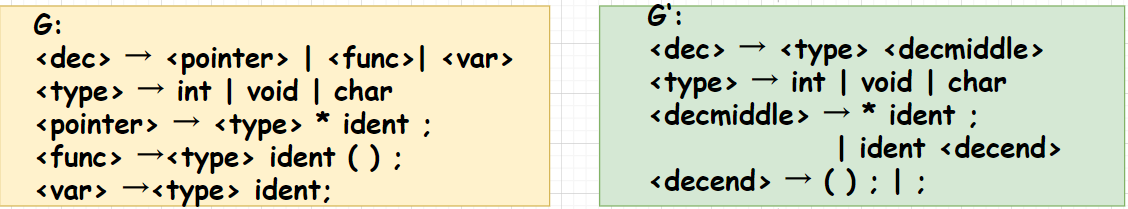


图24 提取左公因子示例

##### (l). S\_文法

预测分析不需要回溯，因此它能够根据当前的最左非终结符*A*，和当前输入的符号，选择正确的产生式。

为了选择正确的产生式，此时能够选择的候选式必须唯一确定，不能有随机性。因此有了S\_文法(简单的确定文法)。其要求为:

* 每个产生式的右部都以终结符开始
* 同一个产生式的各个候选式的首个终结符都不相同

对于这样的文法，每次能够选择的产生式是唯一的，同时S\_文法是不可以有空产生式的。如图25所示。

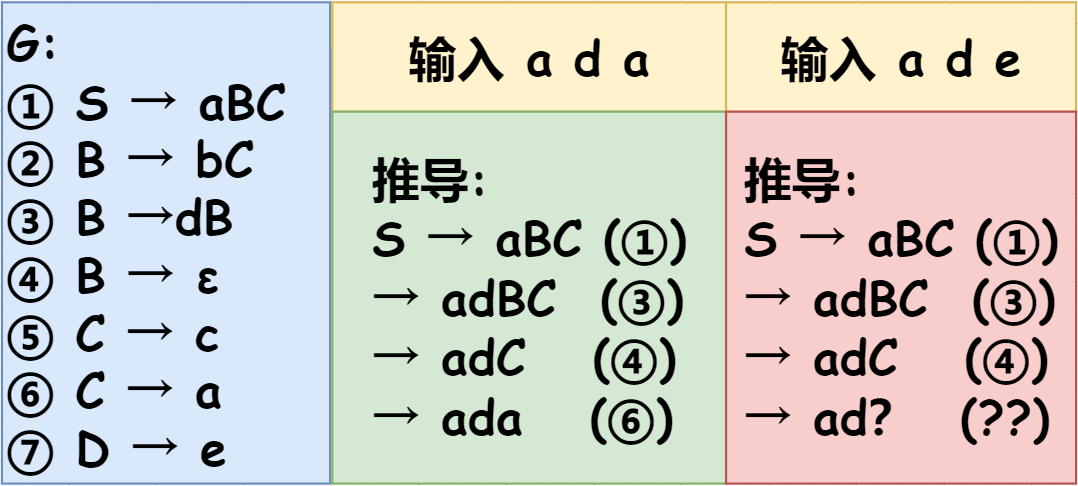


图25 空产生式()的问题

这里给了两个不同的输入，对于第一个输入串，ada，从文法开始S进行推导，依次使用①③④⑥条推导式，输入的ada能够匹配成功；根据当前最左非终结符和输入字符，每次选择的候选式是唯一的。

但对于第二个输入，前面都是相同的道理，但是最后一步非终结符C和当前输入符号e，没有规则是匹配的，因此前面使用第④条规则，即空产生式是徒劳的，此时不应该使用空产生式，而应该直接报错。

但文法中如果没有空产生式，受限会较多，因此我们需要知道，什么情况下需要使用空产生式。事实上，能够使用符号*B*的空产生式，关键看非终结符*B*的后面能够紧跟哪些终结符。

由此我们需要定义一些符号来说明。

* **FOLLOW集合**:非终结符*A*的后继符号集合，记作FOLLOW(A)。

这里的后继符号是终结符，因此FOLLOW(A)可以描述为:

如果*A*是某个句型的最右符号，则将结束符$添加到FOLLOW(A)中。

对于上面那个例子，非终结符*B*的FOLLOW集合通过观察可以看出{*a,c*}。

* **SELECT集合**

对于产生式的可选集，顾名思义，就是遇到这个集合中的符号，就能够选择这个产生式，一个产生式的可选集记为。

(1). 若产生式的右部的第一个符号是终结符，则产生式的可选集只有这一个符号，即。这也很好理解，只有遇到这个终结符时，才能够选择这个产生式，其它时候都不可以。

(2) 如果产生式的右部是空串，则。这也比较好理解，是能够直接跟在非终结符后的终结符；只有遇到这些终结符时，选择产生式，之后才能够可能匹配成功(图25例子)。

对于具有相同左部的各个产生式，如果它们的SELECT集合交集为空，此时遇到一个任何一个终结符，都有唯一的产生式可选(或者直接报错)。

##### (m). q\_文法

由于S\_文法的局限性，其右侧不可以为空，因此有了q\_文法。q文法的要求为:

* 每个产生式的右部以终结符开始，或者其右部是空串
* 具有相同左部的产生式，其SELECT集相交为空

对于相同左部的产生式，SELECT集相交为空的意义在于，此时遇到一个终结符*a*，能够根据*a*属于哪个SELECT集，选择对应的产生式进行推导；试想若相同左部的产生式的SELECT集有交叉，那么遇到交叉的元素意味者选择哪个产生式都可以，这是一种不确定性，也可以理解为二义性，这是我们不希望的。

但q\_文法也有一定的局限性，虽然其产生式的右部可以是空，但必须以终结符开始(或者为空)，不能以非终结符打头。

为此，科学家们引入了LL(1)文法。由于LL(1)文法中，产生式的右部可以是非终结符，需要进一步引入一些概念。

* **FIRST集合**

串首终结符，顾名思义，是字符串的第一个符号，并且是终结符。对于一个文法符号，其串首终结符集的包含的定义是:

可以从推导出所有的**串首终结符**构成的集合。如果，则也应该在中，即。即，对于，

特别地，如果，则。

此时，产生式的可选集的定义如下:

* 如果，则。

即，如果文法符号的FIRST集中没有空串，那么对于一个产生式的可选集就是的FIRST集。即遇到这个中的元素，便能够选择这个产生式。

* 如果，则。

即如果符号的FIRST集中存在空串，也就是里面的非终结符都能够推出空串，那么此时对于产生式，其可选集除了中的终结符(不包括空串)，还应该有的FOLLOW集；是能够直接跟在非终结符后的终结符构成的集合。可以参考q\_文法部分理解。

##### (n). LL(1)文法

LL(1)，第一个L表示从**左**(Left)往右扫描；第二个L表示最**左**(Left)推导；1表示每次向前看1个符号就能够决定词法分析器的下一步。

如果一个文法*G*是LL(1)文法，**当且仅当***G*的任意两个具有相同左部的产生式满足:

* 不存在终结符***a***使得和都能够推导出以开头的***a***串。
* 和**至多**有一个能推导出空串。
* 如果，则；
* 如果，则；

我们逐条解释，对于第一条，如果，使得和都能够推导出开头的串，则和之间有交集，则，的集和的集也会有交集。

对于第二条，二者至多有一个能够推导出空串；试想如果二者都能够推导出空串，则它们的SELECT集都会包含，显然SELECT集交集不为空。

第三条/第四条，如果，则的SELECT集包含，此时的FIRST集合就不可以再包含的元素，否则它们的SELECT集会有交集；第四条同理。

其实第3/4条件是为了保证左部相同的非终结符的各个产生式的SELECT集不相交。

**FIRST集**求解例子:

以简化版的四则运算为例，首先对各个非终结符的FIRST集清空，如图26所示。

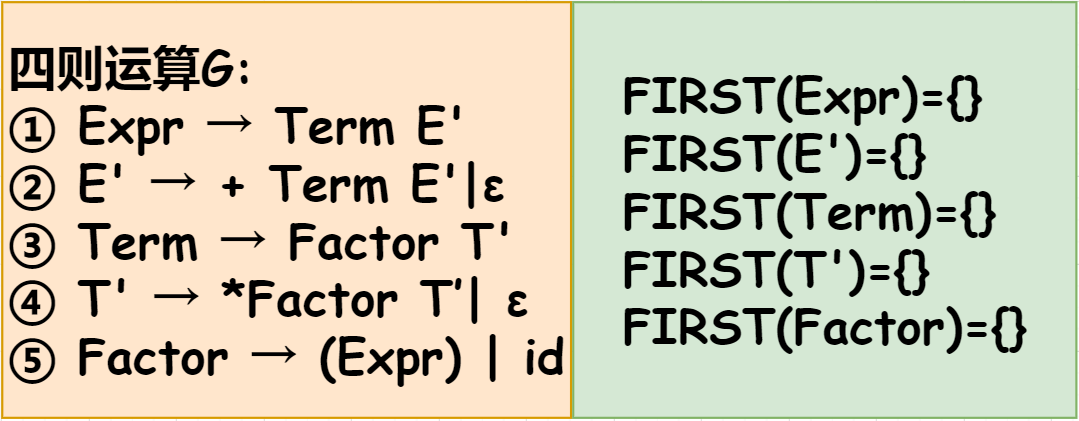


图26 求非终结符的FIRST集(Step1)

接下来，观察每个产生式的右侧串首字符是否为终结符，如果是，则添加到对应非终结符的FIRST集中，这里串首的非终结符只有id、(、\* +。添加完成后，如图27所示。

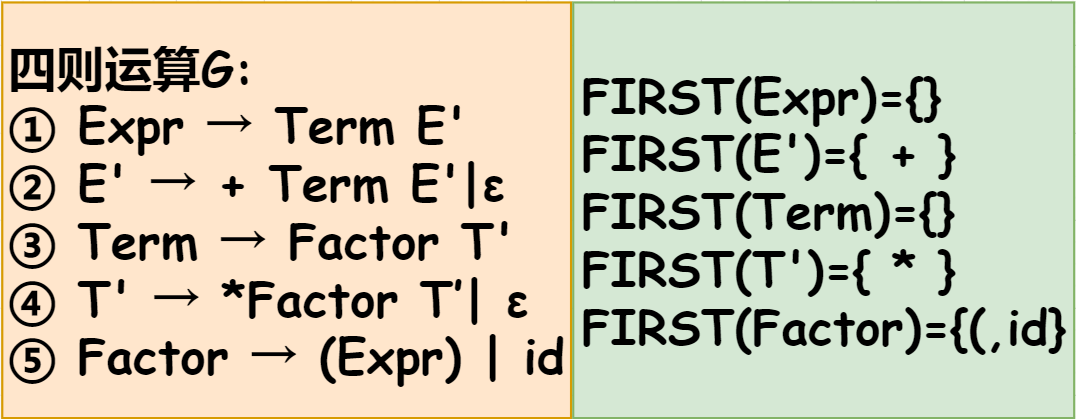


图27 求非终结符的FIRST集(Step2)

如果一个产生式能够推导出空串，那么空串也应该是它的FIRST集中的元素，对于上述文法*G*，第②和第④个产生式能够推导出空串，因此添加到对应非终结符的FIRST集中。如图28所示。



图28 求非终结符的FIRST集(Step3)

接下来，由于不是q\_文法，因此产生式的右部可以是非终结符打头，对于这样的产生式，如①和③，那么它们右部的首个非终结符的FIRST集中的终结符，也应该是这些产生式FIRST集中的元素；即Expr的FIRST集依赖于Term的FIRST集；Term的FIRST集依赖于Factor的FIRST，因此把Factor的FIRST中的终结符添加到FIRST(Term)中，把Term的FIRST的终结符添加到FIRST(Expr)中。如图29所示。

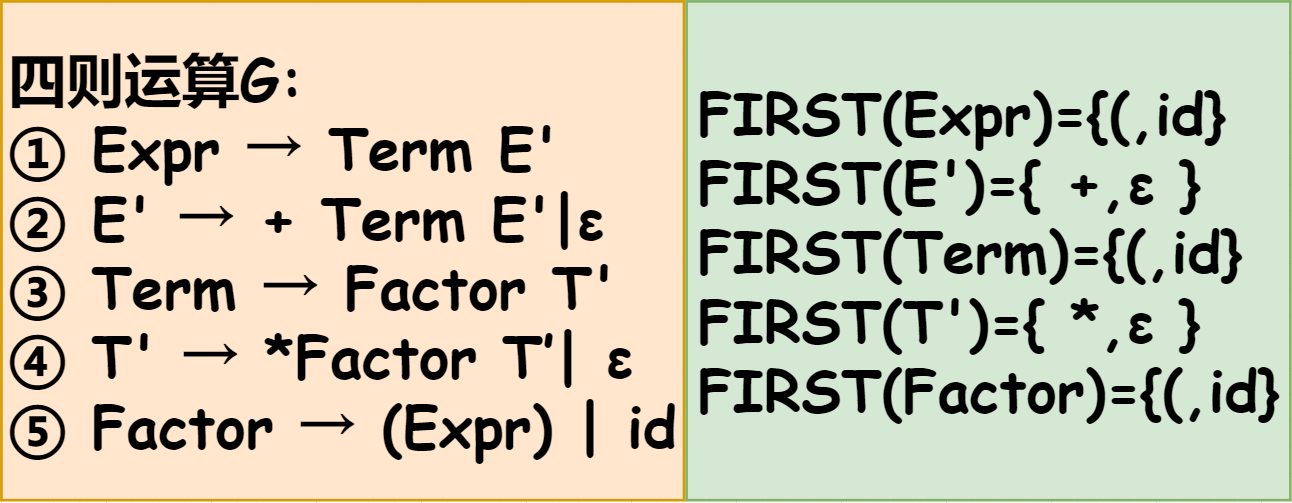


图29 求非终结符的FIRST集(Step4)

对于**FIRST集**的求解，可以简单描述为:

* 如果是一个终结符，那么毫无疑问
* 如果是一个非终结符，并且有，如果对于某个，，，即,…，那么需要把添加到中；如果，有，那么应该把添加到中。
* 如果(是产生式)，那么把添加到中。

重复上述几个步骤，直到FIRST集中没有元素更新。

上述例子，我们继续求解非终结符的**FOLLOW集**。再次看看定义。同时若是某个句型的最右符号，则把结束符$加入到中。

如图30所示，首先初始化待求符号的FOLLOW集。

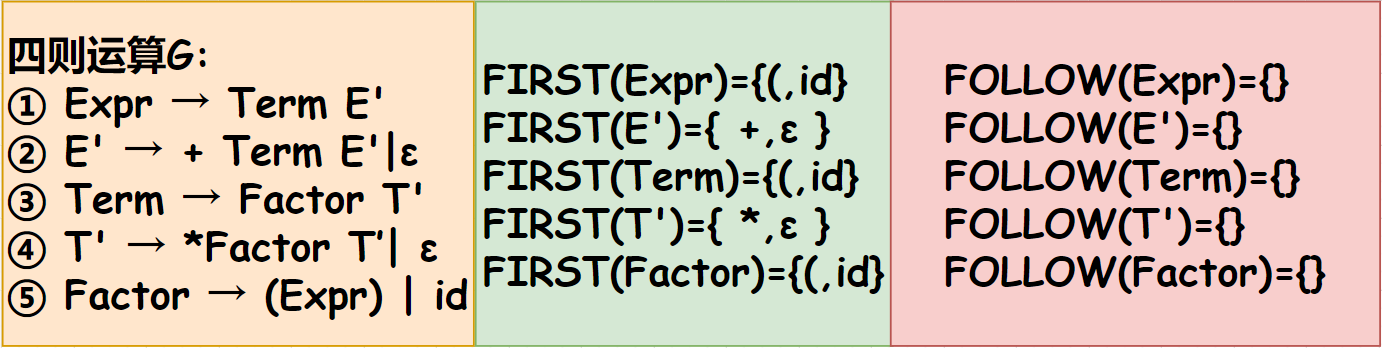


图30 初始化FOLLOW集

观察定义，这里Expr既是文法的开始符号(S)，也是文法的最右符号，因此把$添加到FOLLOW(Expr)中；对于①，的FIRST集中的终结符能够紧跟在Term的后面，因此需要将其加入Term的FOLLOW集中。如图31所示。

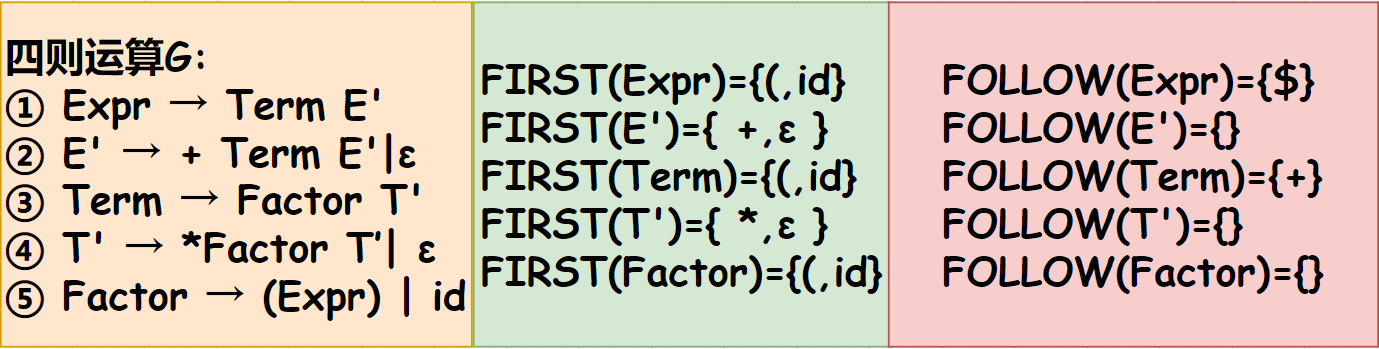


图31 求FOLLOW集(Step1)

同时我们注意到①，能够推导出空串，因此，能够紧跟在Expr后的终结符也能够出现在Term后，因此需要将Expr的FOLLOW集的元素添加到Term中。同时，由于在最右侧，同样，能够出现在Expr后的符号也能出现在之后，因此需要将Expr的FOLLOW集的元素添加到中。如图32所示。

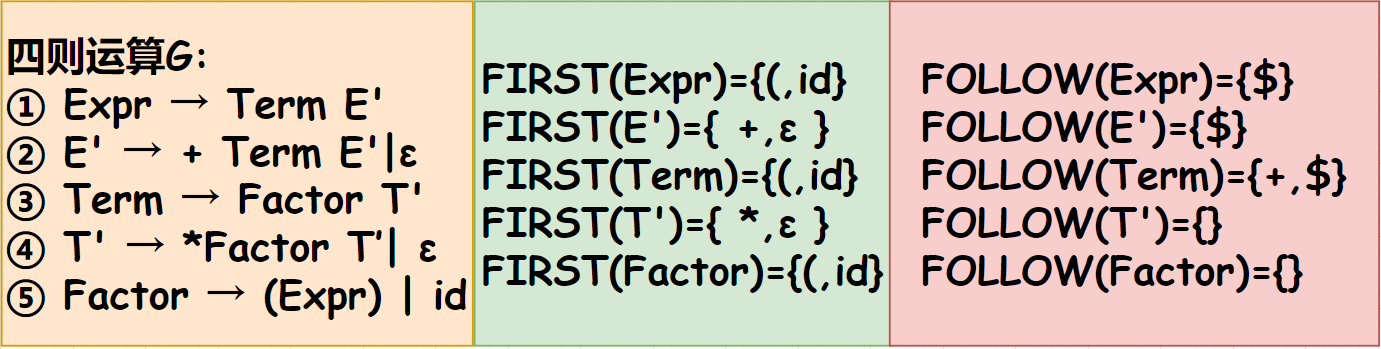


图32 求FOLLOW集(Step2)

观察③产生式，在Factor后，因此的FIRST中的终结符可以出现在Factor后，将其加入FOLLOW(Factor)；同时能够推导出空串，因此能出现在Term后的符号也可以出现在Factor后，将将FOLLOW(Term)的元素加入FOLLOW()中；此外在最右侧，因此能够出现在Term后的符号，也能出现在后，将FOLLOW(Term)的元素加入FOLLOW()。如图33所示。

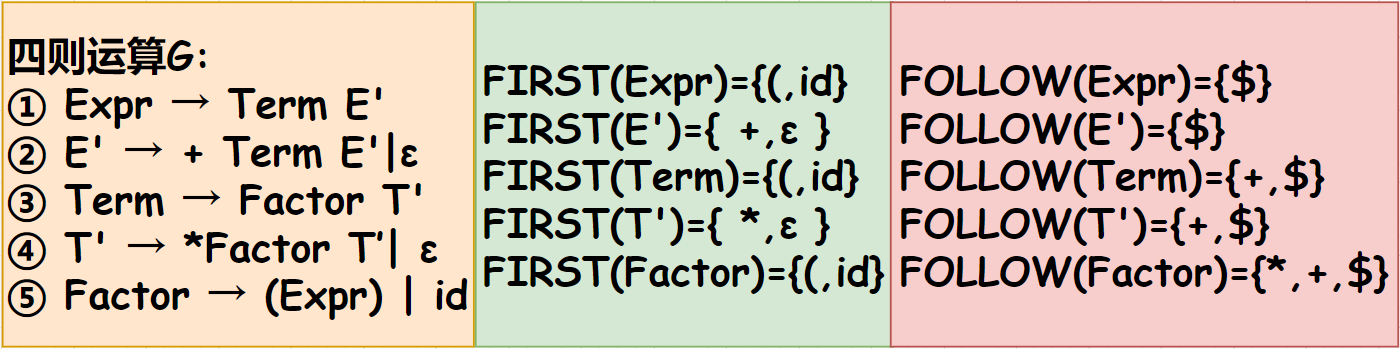


图33 求FOLLOW集(Step3)

观察⑤式，Expr后存在终结符)，将其加入FOLLOW(Expr)中，如图34所示，至此，FOLLOW集求解还没有结束，还需要重复上述步骤，直到没有新的元素加入。

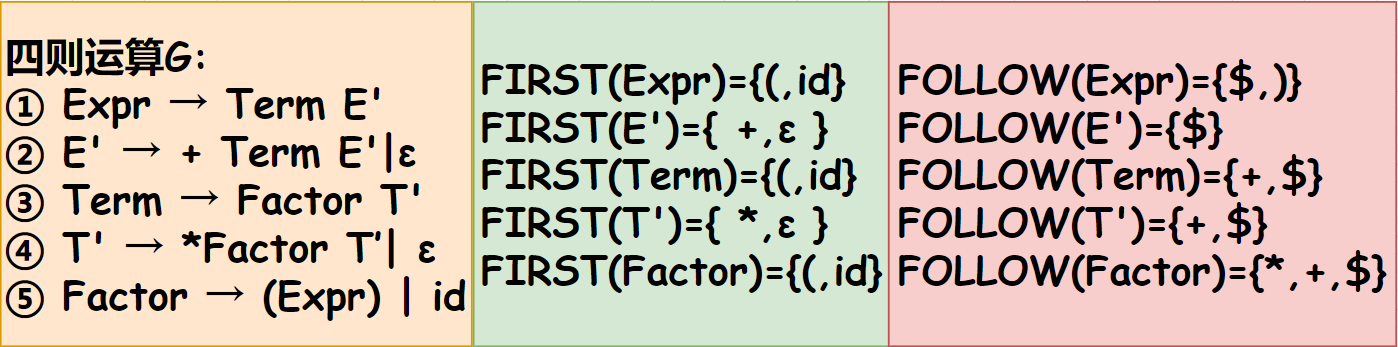


图34 求FOLLOW集(Step4)

重复上述的分析步骤，最后得到的结果如图35所示。

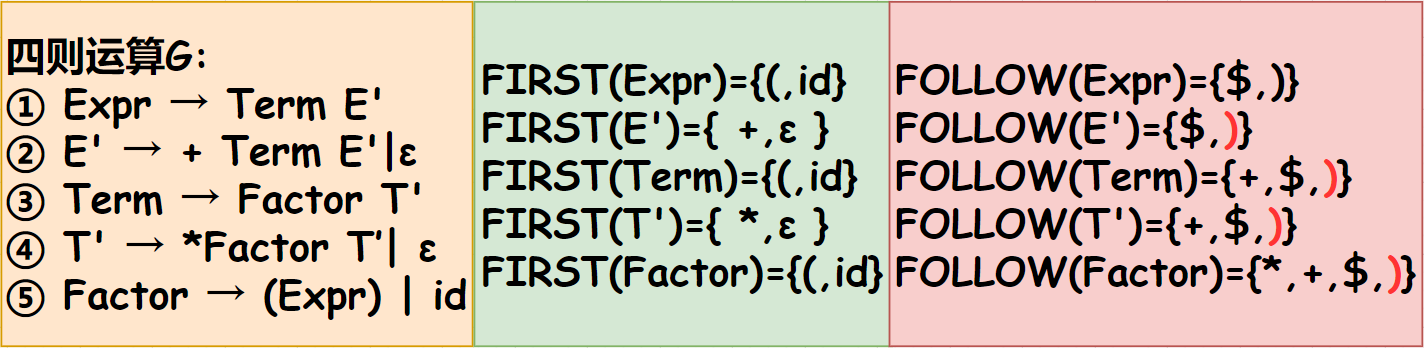


图35 求FOLLOW集(Step5)

对于**FOLLOW集**求解，可以描述为:

* 将$加入FOLLOW(S)中，其中S是文法开始的符号
* 如果在一个产生式，则FIRST()中的终结符应该添加到FOLLOW()中。(是非终结符)
* 如果一个产生式，或并且，则FOLLOW(*A*)的元素都哦应该被添加到FOLLOW(B)中。

重复上述步骤，直到没有新的符号添加到各个集合中。

有了FOLLOW集合和FIRST集合，便能够求出各个产生式的**SELECT集合**。如右图所示，我们求取各个产生式的SELECT集。

* 对于产生式①，其SELECT集是FIRST(Term)，SELECT(①)={(,id}。
* 对于产生式②，由于其右部以终结符打头，所以SELECT(②)={+}。
* 对于产生式③，其右部为空串，因此其SELECT集应该是FOLLOW(E)，即SELECT(③)={$,)}。
* 对于产生式④，其SELECT集是FIRST(Factor)，SELECT(④)={(,id}。
* 对于产生式⑤，右部打头的是终结符，因此SELECT(⑤)={\*}。
* 对于产生式⑥，右部是空串，因此其SELECT集应该是FOLLOW(T’)，即SELECT(⑥)={+,$,)}。
* 类似的，产生式⑦⑧，SELECT(⑦)={(}。SELECT(⑧)={id}。

观察具有相同左部的产生式，即②③、⑤⑥和⑦⑧，它们二者的SELECT集都没有交集，因此这是一个**LL(1)文法**。LL(1)文法可以使用预测分析法，它无需回溯，通过构造预测分析表，在给定当前的产生式和输入符号(k=1)的同时，有唯一可选分支(或者报错)。

类似的除了LL(1)文法，其实还有很多其它的文法，这里就不逐一介绍了。接下来的部分是结合一个简化版的C语言文法，理解推导的过程。

#### 2.2.3 编程设计

编程设计这一部分，主要对得到的token进行词法分析，采用了递归下降的方式，虽然递归下降的方式解析效率不是最高的，但从程序编写和理解的角度是容易的，因此我们采取了这一方法。这一部分对应源文件**Compiler文件夹**下的**Parser**类。

各个类(class)相对独立，但是语法分析的过程中其实也伴随着中间代码的生成，因此类中出现使用其它类的对象和方法也是有可能的，但这不影响整个递归下降的理解。

##### (a).总体设计概述

这里我们设计的文法如图36所示。简化版本的C语言文法似乎也“不简单”。看来其有些冗长，但其实内在的逻辑并不是很复杂，下面会逐一分析。这里我们约定，被<>括起来的符号都是非终结符；直接以英文出现的字符是终结符，表示空串；其中一些界符，如()、{}使用了英文缩写；| 表示或的关系，即这些产生式的右部具有相同的产生式子左部，一般在实际书写时，会将其使用 | 连接，便于阅读。

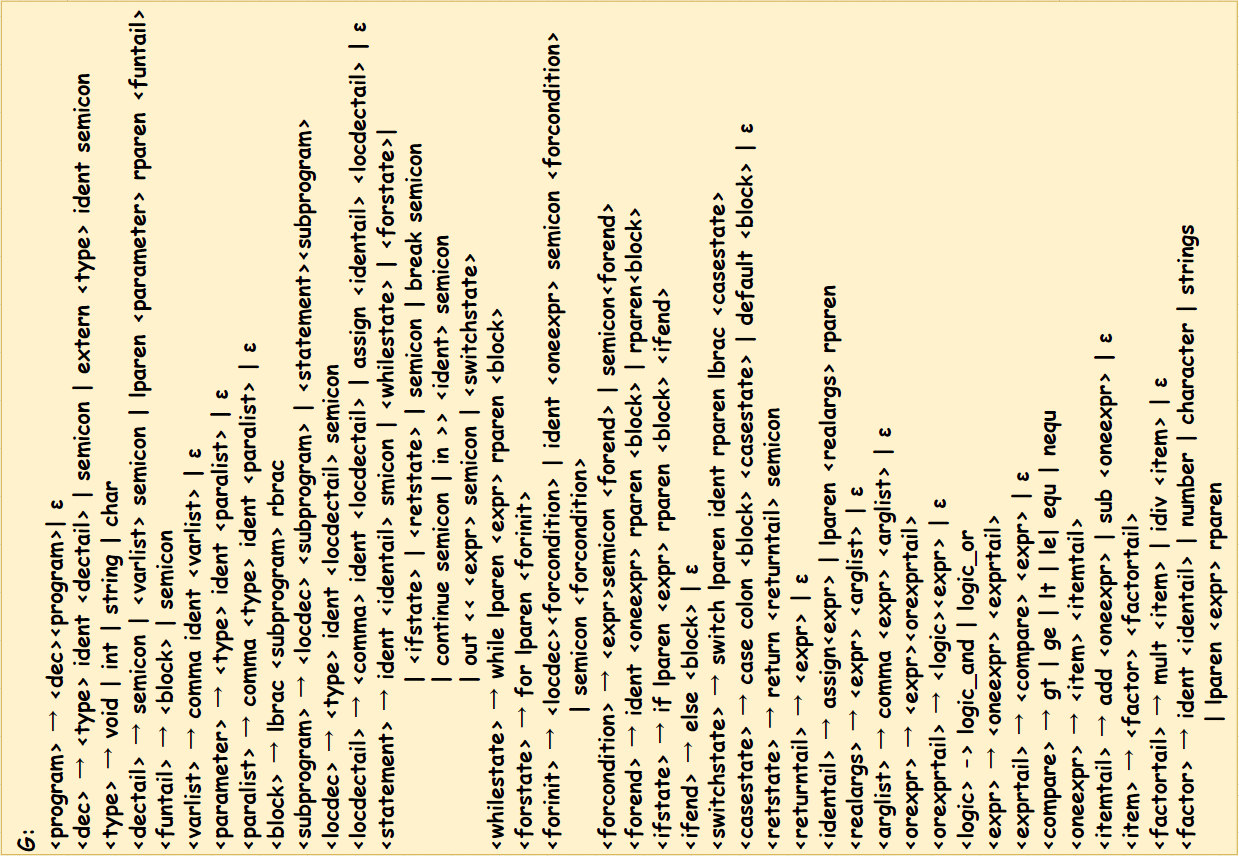


图36 简化版C语言文法

这一文法包含了是C语言文法的一个真子集(保留字少数有差异)，实现了C语言的大部分逻辑框架和运算符，但复杂的数据结构暂时没有纳入其中，例如指针、结构体等都没有考虑，同时由于没有使用堆区内存，所有的变量都是预分配栈空间，因此没有动态类型。

##### (b). 逐产生式分析

* <program> → <dec><program> |

<program>是文法的开始符号(S)，从这里开始递归。<program>递推的结果可以是<dec><program>，也可以是空串，空串其实就是递归的出口，而这个产生式得到的是符号串；想想我们的C语言程序，是不是由一个个函数或者变量声明/定义组成的。如图37所示。

这里我们截取一段C+的代码块，忽略及类型差异，能够看到这里的视角(view)就是变量和函数的声明/定义，每行代码形式如出一辙。

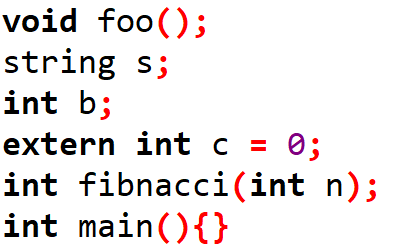


图37 程序分析起始状态

* <dec> → <type> ident <dectail> | semicon | extern <type> ident semicon

第二步推导<dec>能够推出的分支就比较多了，可以是类型打头的分支，这里上图中的代码块应该都属于这一分支；还可以是一个分号终结符(semicon)，表示空语句；也可以是extern打头的外部变量声明。

这里看程序就更细致了，第一步只是笼统地看待程序都是一系列的声明/定义语句，这里就要具体区分是外部声明语句还是一般的声明/定义语句还是空语句。这里具体选择哪一个分支，取决于读取的终结符是哪一个(红色)，根据读取到的符号判断进入哪一个分支，如图38所示。

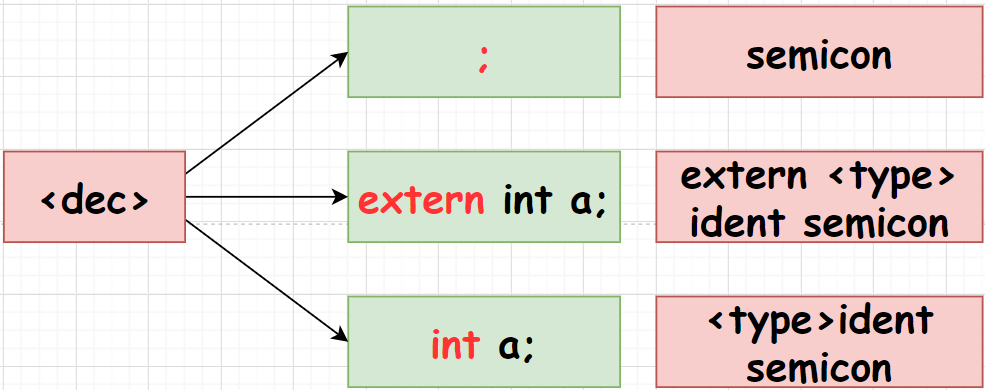


图38 <dec>推导示例

* <type> → void | int | string | char

这是类型<type>的推断结果，后面都是终结符，这里其实就是判断不同的类型。

* <dectail> → semicon | <varlist> semicon | lparen <parameter> rparen <funtail>

这里是在做进一步区分，当你已经匹配到int foo这样的串的时候，接下来你要判断，它是一个变量的声明，还是变量的定义，还是函数。这三个选择刚好对应<dectail>的三个推出式。如图39所示。

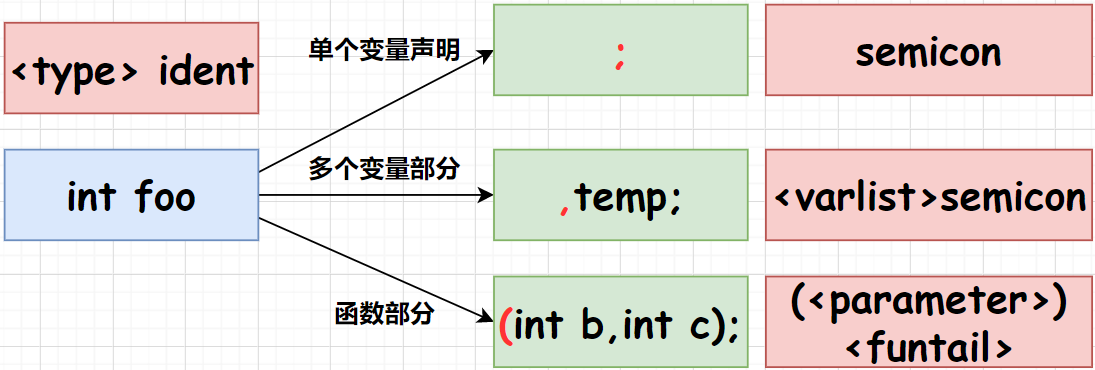


图39 <dectail>推导示例

这里可以看到，绿色表示后续输入的代码(示例)，红色是候选的产生式(下同)，则目前这一步在确定是单个变量的声明；还是多个变量的声明；或者是函数的声明/定义。

* <funtail> → <block> | semicon

<funtail>实际是39图进入了第三个分支，这里其实在进一步区分是函数定义还是函数声明，如图40所示。

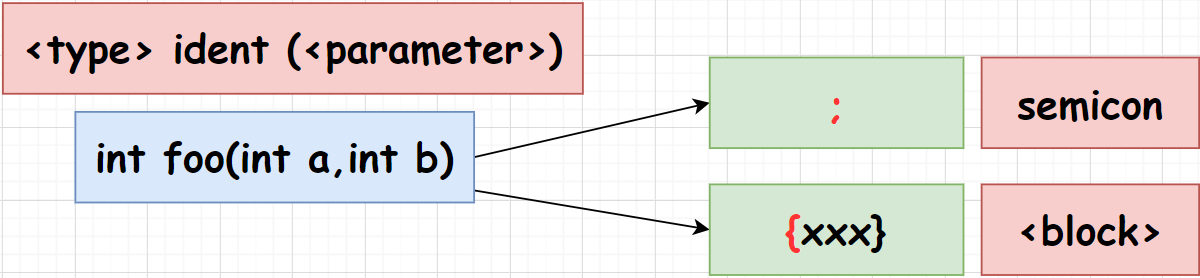


图40 <funtail>推导示例

* <varlist> → comma ident <varlist> |

这是图39的第二个分支，进一步解析多变量定义。多个变量之间用逗号隔开，一旦遇到逗号就循环进入<varlist>进行解析，直到遇到分号，解析结束。这里其实是右递归的形式，由于我们是LL(1)文法，因此不存在无限循环的情况。

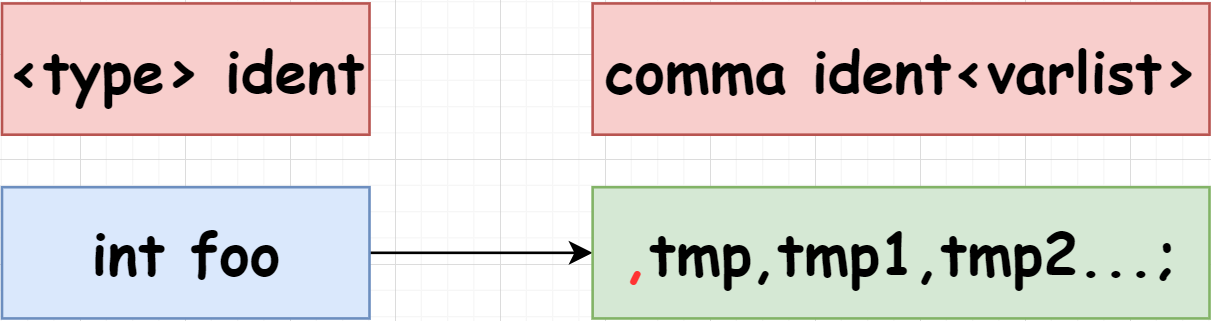


图41 <varlist>推导示例

* <parameter> → <type> ident <patalist> |

这是图39第三个分支，此时说明这是一个函数，<parameter>是解析函数形参的符号，这里形参如果为空，会匹配到右括号，进入第一个分支；如果有参数，那么进入第二个分支，这里形参的解析和<varlist>是不是有些相似，都是右递归的解析方式，如图42。

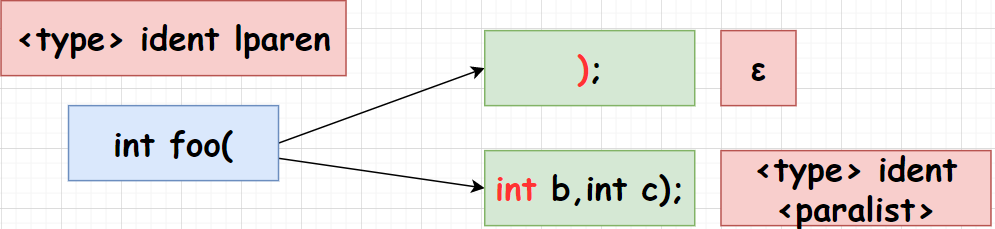


图42 <parameter>推导示例

* < patalist > → comma <type> ident < patalist > |

< patalist >推导是右递归的形式，如图43所示。

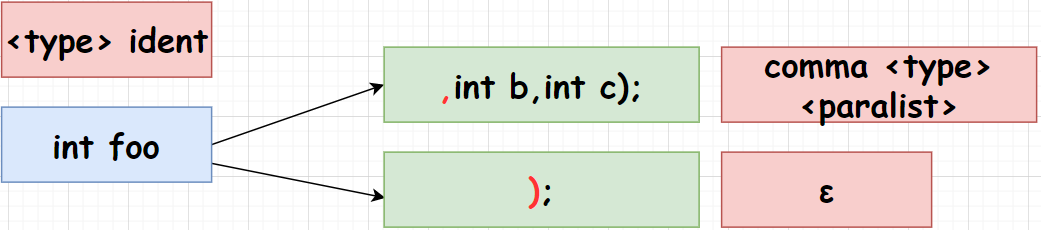


图43 < patalist > 推导示例

* <block> → lbrac <subprogram> rbrac

这里是解析代码块，也就是被{}包裹起来的部分，被包裹起来的部分是子程序，里面的变量都是局部变量，之前的变量和函数都是全局变量。

* <subprogram> → <locdec> <subprogram> | <statement> <subprogram>

子程序可以被解析为两部分，一个是局部变量的声明/定义，对应于<locdec>；另一个是语句解析，对应于<statement>。这里有些抽象，可以往下继续看两部分的解析过程。

* <locdec> → <type> ident <locdectail> |

这个产生式是不是有些熟悉，其实和全局变量的解析没有任何区别，只是因为block中不可以嵌套函数声明/定义，因此只能够对变量进行声明和解析！这里解析的变量都是局部变量。

* <locdectail> → <comma> ident <locdectail> | assign <identail> <locdectail> |

这里与全局变量的声明略有不同，如图44所示，这里一共有三条分支，可以是单个变量定义，也可以是多个变量定义，还可以是定义的同时就赋值。这和全局变量有一些不同，全局变量的语法中不可以直接赋值，也不能定义完之后赋值，但局部变量可以。

读者可以自行修改语法，使得语法分析器能够支持你想要的解析类型。



图44 <locdectail>推导示例

* <statement> → ident <identail> semicon | <whilestate> | <forstate> | <ifstate> | <retstate> | semicon | break semicon | continue semicon | in >> <ident> semicon | out << <expr> semicon | <switchstate>

语句解析的内容有些长，但其实逻辑是一样的。如图45所示。这里分支较多，但每一个分支都是可能在块中出现的，例如局部变量的处理，各种逻辑语句(while,for,if-else,switch-case)，以及输入输出等。

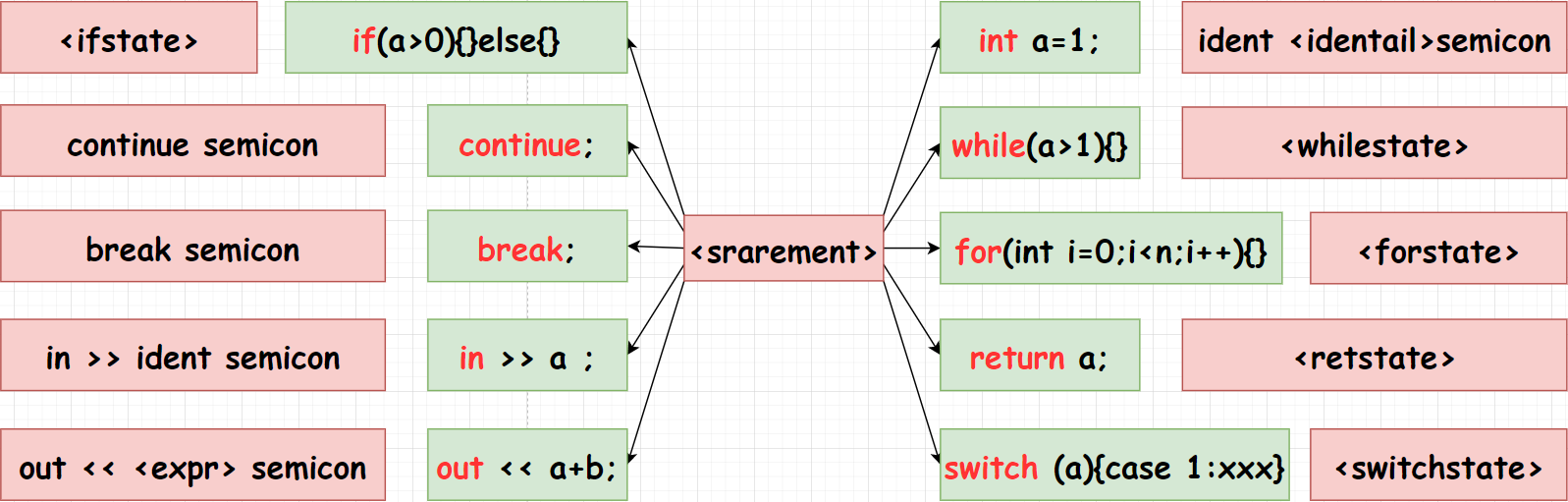


图45 <statement> 推导示例

* <whilestate> → while lparen <expr> rparen <block>

这里没有多余的分支，对于一个while语句而言，它只能是条件和后面从属的块构成，这里其实看出程序有交叉，因为<block>本身也可以包含while，即while中可以有更小的代码块<block>，这样就实现了这些条件语句的嵌套解析。

* <forstate> → for lparen <forinit>

这是for语句开始的部分，匹配到关键字后，再匹配一个终结符左括号，进入<forinit>分支。

* <forinit> → <locdec><forcondition> | <ident> <oneexpr> semicon <forcondition> | semicon <forcondition>

这里有三个分支，如图46所示。for循环的初始化部分，既可以是定义一个变量，也可以是变量的赋值，或者为空。

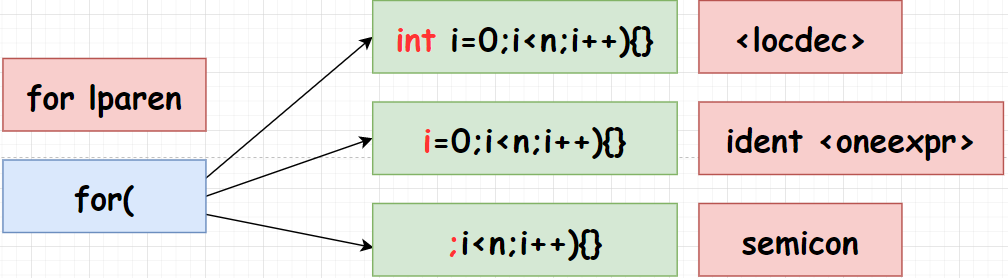


图46 <forinit>推导示例

* <forcondition> → <expr> semicon <forend> | semicon <forend>

在for循环的条件部分，可以是一个条件表达式，也可以为空，如图47所示。

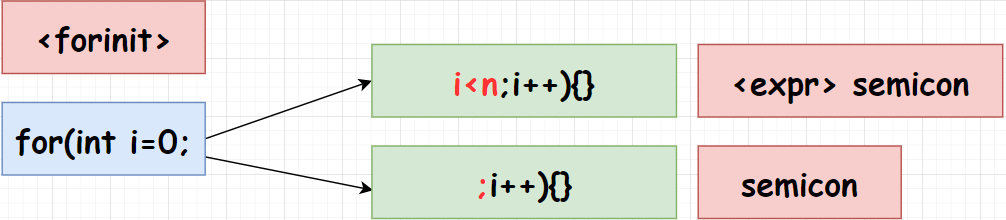


图47 <forcondition> 推导示例

* <forend> → ident <oneexpr> rparen <block> | rparen <block>

最后是<forend>，同样，forend可以是一条赋值运算语句也可以为空。这里if-else语句和while语句和switch语句是类似的，限于篇幅就不重复示例。

返回语句的推导也是类似的，可以返回空(void)，也可以返回一个表达式。

* <identail> → assign <expr> | lparen <realargs> rparen

<identail>也就是跟在ident后面的部分，可以是赋值语句，也可以是函数调用，如图48所示。

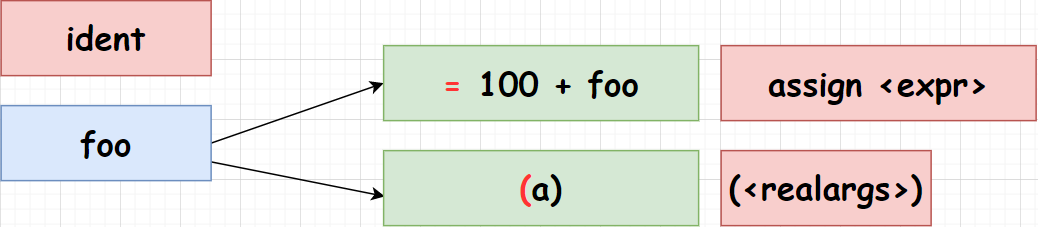


图48 <idential>推导示例

* <realargs> → <expr> <arglist> |
* <arglist> → comma <expr> <arglist> |

把两个连起来看，这与<parameter>和< patalist>的推导示例如出一辙，一个是函数形参的推导，一个是函数调用的实参的推导，形参比实参其实只多了一个类型<type>。这样右递归的形式，解决了不同函数形参数量不同的问题；同时一些函数支持不定参数传参，例如经典的printf函数，<arglist>推导形式支持不定参数的传参。

* <orexpr> → <expr> <orexprtail>
* <orexprtail> → <logic> <expr> |

进入运算的阶段，首先是逻辑运算。这里<logic>可以推导出逻辑与(AND)和逻辑或(OR)，当然有些表达式本身没有逻辑运算，因此<orexprtail>可以推导出空串。

* <expr> → <oneexpr> <exprtail>
* <exprtail> → <cmp><expr> |

表达式<expr>的推导，被拆成了两部分，前一部分可以理解为简化版的四则运算，后一部分是运算结果的比较(可选)。同样的逻辑，有些表达式不存在运算结果比较，因此<exprtail>可以推导出空串。<cmp>可以推导出多种比较运算，例如大于、小于和不等于等等。

* <oneexpr> → <item> <itemtail>
* itemtail → add <oneexpr> | sub <oneexpr> |

进入算数运算阶段后，接下来是优先级较低的加减运算，同理，由于一个表达式可以没有加减运算，因此itemtail可以推导出空串。

* <item> → <factor><factortail>
* <factortail> → mult <item> | idiv <item> mod <item> |

之后是运算优先级较高的乘法/除法/模运算，同理，<factortail>可以推导出空串。以上四个部分的形式都很类似，事实上这些运算的优先级刚好是从高到低[10]；即在这样一个运算体系中，乘法一类的运算会被优先计算(括号除外)。这里我们没有添加C语言的位运算，位运算大多是二元运算，其形式与上述定义类似，优先级介于关系比较运算<cmp>和逻辑连接运算<logic>之间。

* <factor> → ident <identail> |number | character | strings

| lparen <expr> rparen

最后是运算体系中最小的单元，一个因子<factor>，它可以是一个数字(包括字符和字符串)；也可以是一个变量；还可以是由括号包裹的表达式！这里便体现了表达式的递归思想，表达式本身可以推导出<factor>，但<factor>又可以包含更小的表达式。尽管是递归，但这是一个右递归，因此在LL(1)文法中不会出现无限循环的情况。

这里我们没有添加一元运算符，它们的优先级低于括号，但高于乘法一类的运算符。

至此，文法部分我们就解决了，这里每一个非终结符(个别除外)在实际的代码中都对应着一个函数签名，函数调用结构与推导过程是一致的，在关注语法部分时，可以忽略错误处理和中间代码生成的部分。

### 2.3 语义分析(Semantic)

经过了词法检查，经过语法检查，程序是否就能够被正确编译呢？显然不是。例下面给出示例程序p4。

*.e.g.*

// 示例程序p4

*void main(){*

*1=1;*

*int a = “hello world!”*

*return 0*

*}*

例如给常量赋值、赋值变量类型不统一、函数返回值不匹配还有函数形参和实参匹配问题等等，这些都是属于语义分析的部分，上述程序从语法上没有问题，但就像最开始那个小猫吃鱼的例子，通过文法G，我们能够产生无数的句子，但有些句子是不符合语义的，例如“小鱼吃猫”。这里列举的几个错误类型是类似的道理，不是所有的句子都是有意义的，因此这里要对一些常见的错误类型进行检查，减少程序运行时crash的可能性。

事实上，即便通过了语义检查，程序运行时依旧可能崩溃或者不能达到设计者预期的目标，常见的IDE基本都能对静态语义做很好的检查，在你编写程序时，及时给出提醒，让我们及时修改；即便这样，有时依旧会看到弹出assertion的窗体。因此编译器只能够最大限度地保证程序的正确性，但程序中一定有bug需要动态调试才能发现。

语义部分对应Compiler文件夹下Semantic文件，其中VarRecord是对任意变量的抽象，FunRecord是对任意函数的抽象，最后还有一个Table，用来存储全局符号，避免符号重定义/未定义等问题。

语义部分的检查伴随着中间代码的生成，即在产生中间代码之前进行检查。

### 2.4 汇编代码生成

##### (a). 概述

Compiler的最后一步是汇编代码的生成，这里汇编代码其实就是中间代码，中间代码的好处在于起到桥梁的作用，就像数据库三级模式和两级映像，能够很好地屏蔽不同内模式的差异；这里汇编代码作为桥梁，能够在必要时，产生不同平台下的机器代码，从C语言程序员角度看，所编写的代码是一致的，最后运行的结果也是一致的，但实际的二进制文件是不一致的，屏蔽这些细节的桥梁是中间代码。理论上也可以一步到位，但直接从C语言翻译为机器指令有些费劲；汇编和指令基本是一一对应的关系，而C语言的一条语句基本对应2~10条左右的汇编；汇编比机器指令的可读性要高很多，这也方便我们调试一些bug。

这里的汇编是x86汇编，并且也只用到基本的简单指令，用到的寄存器如表6所示。

表6 汇编部分使用的寄存器

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| AL | CL | DL | BL | AH | CH | DH | BH |
| EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |

上述寄存器之所以是这个顺序，与它们的编号有关系(第一行)，这个编号和后面指令翻译也是相关的，因此我们也按照这一顺序表示。

汇编翻译时，主要用到32位寄存器，部分时候时8位寄存器，这里没有用到16位寄存器，理论上可以使用，但最后汇编代码翻译为机器码时，会比较复杂，IA-32架构[11]对应的是CISC[12](复杂指令计算机)，其指令集非常多，多到你难以想象，因此汇编中用到的是少部分指令集。

这里对应的代码是Compiler文件夹下的Generator类。这里有几个约定，其实是参考了x86的调用约定[13]。

* 函数的返回值保存在EAX寄存器中；在x86中，可能会因为字长关系，被保存在AX/AL/EAX中，这里我们统一为32bit。
* 函数实参通过栈传递，从右至左依次入栈；
* 名字修饰，这里x86的cdecl约定要求编译后的函数名的前缀以下划线打头；由于后续的汇编器是属于开发的一部分，这里命名修饰我们并没有和x86保持一致。

命名修饰其实比较有趣，尤其是C++，和C的修饰差别很大，C++支持不同作用域下同名变量的存在，这给了程序员更多操作的空间，例如在for循环时，可以每次都int i= 0；这样的操作在C++中是合法的，因为这个i的作用域从属于这个for循环，程序员无需每次在程序的开头定义好全部的变量；但另外一方面，这对于编译器提出了更高的要求，对于不同作用域，以及不同命名空间中的同名变量，需要通过命名修饰，保证它们的唯一性。

如图49所示。在Generator类中有两个静态的变量，分别是buffer\_flag和id。如果我们没有在使用前声明它们，编译器会报错，找不到符号。

仔细观察编译器具体给出的符号名不是简单的buffer\_flag和id，而是有很多奇怪的字符如@、?连接在一起构成的，这其实就是编译器的命名修饰，具体修饰的规则因编译器不同而异，具体可以参考《程序员的自我修养》一书(86-91)。

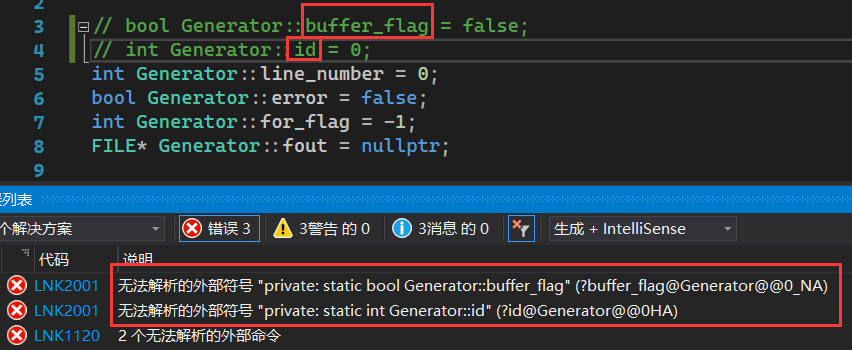


图49 C++符号修饰示例

我们对符号修饰的规则相对简单，在不同类型的符号前添加对应的前缀，例如变量前添加对应前缀，变量名后添加编号后缀进行修饰。支持即用即定义，但是不存在作用域的修饰，因此函数内的变量名不允许冲突，否则会编译失败；当然，既然不允许任何同名变量的存在，其实也可以不对变量名修饰，因为它们之间本身就没有冲突。

* 调用者负责清栈，即将栈中的实参弹出；同时调用者负责保护EAX寄存器。这一点与x86调用约定保持一致。

下面主要是汇编语句的翻译流程，首先是逻辑框架的翻译。

##### (b) if-else语句汇编

这里汇编代码加粗，伪代码正常斜体(下同)。这里对于if-else语句，首先会根据条件产生条件代码，将最后条件运算的结果存入寄存器eax中，通过比较eax和0的值，判断执行if对应的block块(BLOCK 1)还是else对应的block块(BLOCK 2)，这里BlOCK 2是可选，因为else分支可以省略。如程序p5所示。

##### (c). for 循环语句汇编

*.e.g.*

// if-else条件语句示例程序p5

*IF\_START:*

*Condition → eax*

***cmp eax,0***

***je IF\_MIDDLE***

*BLCOK 1*

***jmp IF\_END***

*IF\_MIDDLE:*

*[BLOCK 2]*

*IF\_END:*

for循环的执行时，首先执行①，也就是初始化部分，紧接着是②，然后跳转到④执行，然后是③，之后又从②开始。如图50所示。

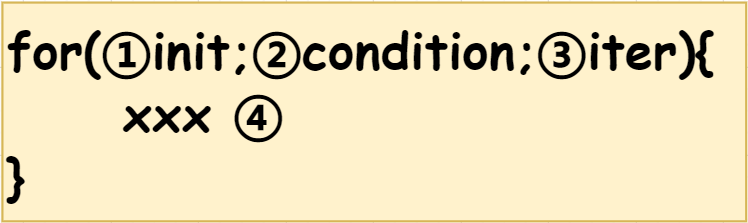


图50 for循环代码四部分

其中①只会被执行一次，其它部分是循环，因此汇编代码结构如程序p6。

*.e.g.*

// for循环示例程序p6

*FOR\_START:*

*[FOR\_INIT] ; for*循环初始化部分

*FOR\_LOOP: ; for*条件判断部分

*Condition → eax*

***cmp eax,0***

***je FOR\_EXIT*** *; for* 循环退出部分

***jmp FOR\_BLOCK***

*FOR\_ITER:*

*iter statement ; for* 循环迭代部分

***jmp FOR\_LOOP***

***FOR\_BLOCK:***

*BLOCK ; for* 循环块

***jmp FOR\_ITER***

*FOR\_EXIT:*

这里之所以没有按照顺序产生汇编代码，而是大量使用跳转，是因为程序扫描的时候，是按照顺序扫描，即先扫描的FOR\_ITER部分并不是先被执行的程序，因此需要使用跳转指令跳过这部分，之后再去执行。如果希望按照汇编顺序产生汇编代码，减少使用跳转指令，那么语法分析程序可能需要回溯，即先分析BLCOCK块中的代码，再分析ITER部分，两种方式或许各有利弊。

##### (d). whie循环语句汇编

while循环语句比for更简单，并且执行顺序与分析顺序基本一致。

*.e.g.*

// while循环语句示例程序p7

*WHILE\_LOOP:*

*Condition → eax*

***cmp eax,0***

***je WHILE\_EXIT***

*BLCOK*

***jmp IF\_WHILE\_LOOP***

*WHILE\_EXIT:*

这里while语句没有太大难度，do-while语句也是类似的，但在语法分析部分我们没有给出do-while的实现。

##### (e). switch-case语句汇编

switch-case语句和if-else语句从C语言的角度看，没有太大的区别，似乎switch-case是多个连续的if-else，但是连续的if-else效率其实是很低的，试想有100个case标签，采用if-else语句的翻译模式，那么最坏的情况需要比较100次，时间复杂度是线性增长的，即。但在C语言中，switch-case语句的case标签要求是数值类型常量，即在程序运行前，编译器其实就知道所有标签的分布情况，因此对于case标签较多时，gcc会进行优化。优化的大致思路如下:

(1). case标签数量较少，例如不多于4个。

对于这一类型，其实和连续的if-else没有太大的差异，因为标签本身较少，比较次数也不多，因此可以不做优化，采用cmp和je指令，将其翻译为连续比较语句即可。

(2) case标签较少，多于4个，连续性较好，即最大值max和最小值min之差不大。

例如case标签的分布在1-10之间，缺失了4，5，7，8。此时gcc会产生一个长度为10的跳转表，跳转表的每一项都是一个地址，如果存在对应的case标签，则跳转表这一项就填写对应case标签的地址，否则就填写default的默认跳转地址；这样一来，拿到一个变量，判断它是否在大于max和小于区间标签的min，如果落入这个区间，可以直接根据作为数组的索引，立即跳转到对应的分支执行语句。执行时间可以看作常量，比连续if-else高不少。

(3)其它情况，例如case标签实在比较多，或者连续性很差，例如标签值为1、100、5000等。如果使用第二种方式，会造成内存大量浪费，这样换来时间上的节约是不值得的，因此gcc会采用对半搜索，将命中的时间复杂度控制在。对半搜索其实就是构建一个二叉树，先对case标签排序，然后对半搜索即可。

在实际编程中，我们采用了(1)和(3)，即当case数量不多于4时，逐个比较，多于4个时，采用对半搜索。程序p8是case数目小于等于4时对应的汇编代码。

*.e.g.*

// switch-case语句示例程序p8

*SWITCH\_START:*

***jmp CASE\_TEABLE***

*CASE\_LAB1: BLOCK 1*

*CASE\_LAB2: BLOCK 2*

*… …*

*CASE\_LAB5: BLOCK 5*

*DEFAULT\_LAB:*

*[BLOCK\_DEFAULT]*

***jmp SWTICH\_END***

*SWITCH\_END:*

***jmp CASE\_TABLE\_END***

*CASE\_TABLE:*

***cmp eax,CASE\_LABE1***

***je CASE\_LAB1***

***… …***

***cmp eax,CASE\_LAB5***

***je CASE\_LAB5***

***jmp DEFAULT\_LAB***

*CASE\_TABLE\_END:*

当case标签较多时，此时需要产生对半搜索的代码。算法伪代码见示例程序p9。只看黑体部分，其实就是一个递归对半搜索。

*.e.g.*

// switch-case语句示例程序p9

***void Generator::GenerateCaseTable(int start,int end,int[] element) {***

*// int jmp\_id = start + ((end - start) >> 1);*

*// if (start <= end) {*

*// label CASE\_LAB(jmp\_id):*

*// cmp eax, element[jmp\_id]*

*//* ***je ..***

*// gen = false;*

*// if (jmp\_id != start)* ***jl ..***

*// else gen = true*

*// if (jmp\_id != end)* ***jg ..***

*// else gen = true;*

*// if(gen) jmp DEFAULT\_END*

***GenerateCaseTable(start, jmp\_id - 1, element);***

***GenerateCaseTable(jmp\_id + 1, end, element);***

***// }***

***// else return;***

***}***

## 3.汇编(Assemble)

汇编器的作用是对产生的汇编代码(.s文件)进行分析，输出目标代码。汇编器同样会有词法、语法、语义和代码生成这几个部分，理论部分参考第二章的介绍。汇编器虽然也有这这个模块，但是相对要简化很多，首先汇编本身词法就不多，大部分都是保留字，尤其是各种寄存器和指令；其次汇编对应的语法也相对单一，基本以mov eax,ecx这类形式为主；再者，一条汇编语句对应着一条机器指令代码，不像C语言到汇编那样，对应关系较为复杂。

### 3.1 词法分析(Lexer Analyse)

汇编器的词法分析相对简单，基本是编译器词法分析的一个缩略版，大部分都是保留字，分析的符号很有限。如表7所示。

表7 汇编词法表

|  |  |
| --- | --- |
| **Token** | **Value** |
| ident(保留字) | global、\_start、section、db、dd、dw、equ、times |
| ident | 变量名 |
| regs(寄存器) | al、cl、dl、bl、ah、ch、dh、bh  eax、ecx、edx、ebx、esp、ebp、esi、edi |
| number(数字) |  |
| 指令(instruction) | add、sub、mov、cmp、and、or、lea  call、int、mult、idiv  neg、inc、dec、jmp  je、jg、jl、jge、jle、jne、jna  push、pop、ret |
| 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 | 、 |
| 界符 | ,、[、]、: |
| 注释 | ; |

这里指令占据了大部分，这些指令都是很常见的汇编指令，其含义也通俗易懂，例如JA(Jmp Above)、JE(Jmp Equal)等，前者表示大于跳转，后者表示等于跳转，其它的跳转指令是类似的助记含义。另外，一些跳转指令没有实现，例如JZ，它其实和JE指令是一样的，因为CMP本身是将两个操作数相减，如果相等，Z标自然也是0，JZ同样会跳转。其它没有用到的汇编指令这里就省略没有分析，程序会认为是非法指令，可以自行添加需要的实现的指令。

### 3.2 语法分析(Parser Analyse)

##### (a). 概述

由于在汇编代码产生部分，我们仅仅使用了少量的汇编指令，因此在汇编器的实现中，对应的词法分析会简短很多，比起C语言子集的文法G，汇编的文法G短了不少。如图51所示。

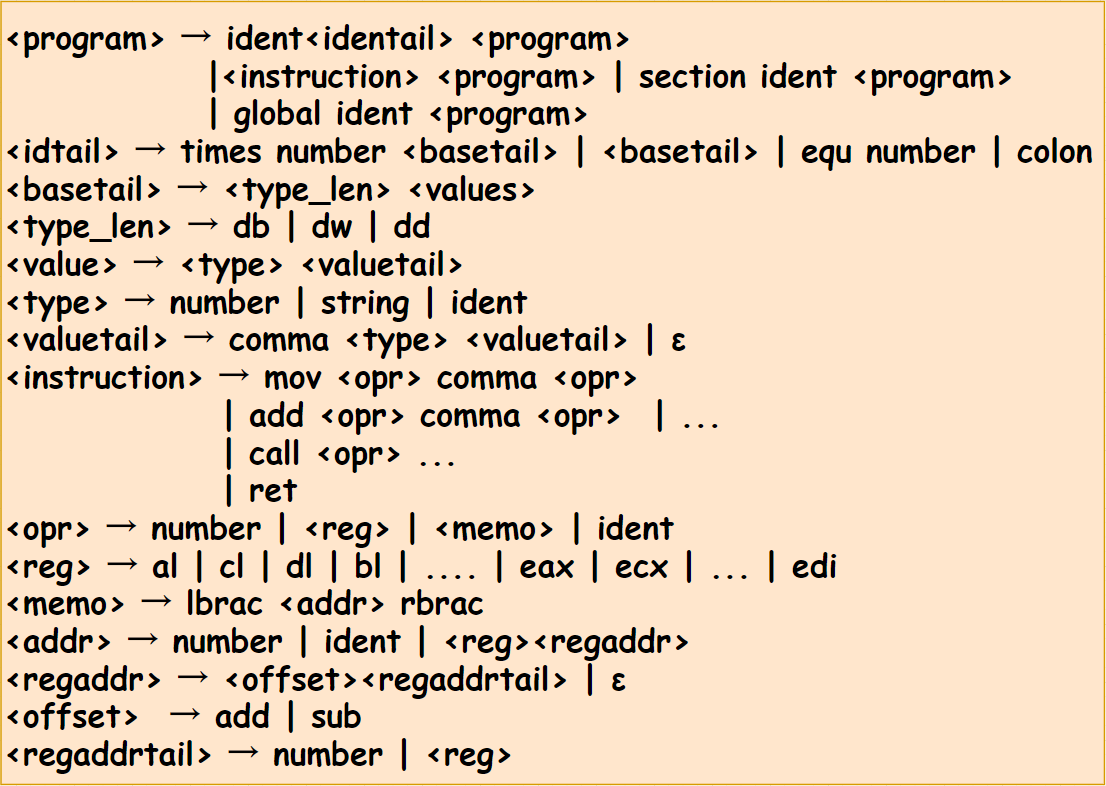


图51 汇编文法

##### (b). 逐产生式分析

* <program> → ident<identail><program> | <instruction> <program>

| section ident <program> | global ident <program>

文法开始符号S为<program>，这是一个典型的右递归，分支示意图如图52所示。这里一共分为4个分支，分别是段名、指令、标号名以及全局符号。根据读入的终结符，可以判断程序进入哪一个分支。

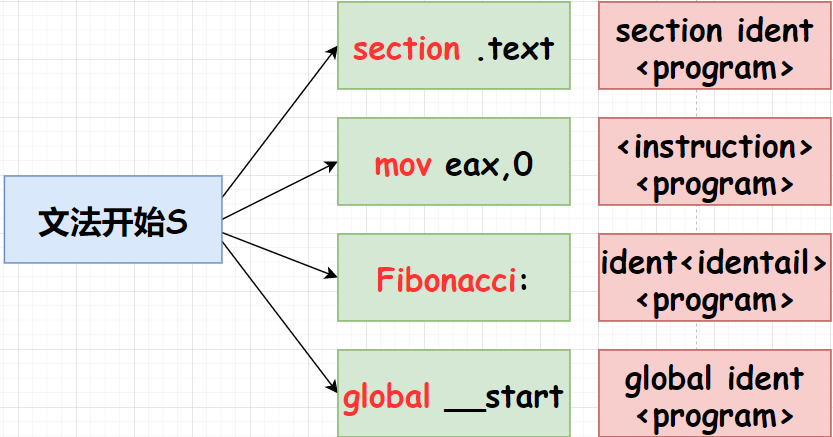


图52 <program>推导示例

* <idtail> → times number <basetail> | <basetail> | equ number | colon

这里是图52的第3个分支，当读入了一个标识符ident后，后面会进一步区分不同的分支。如图53所示。

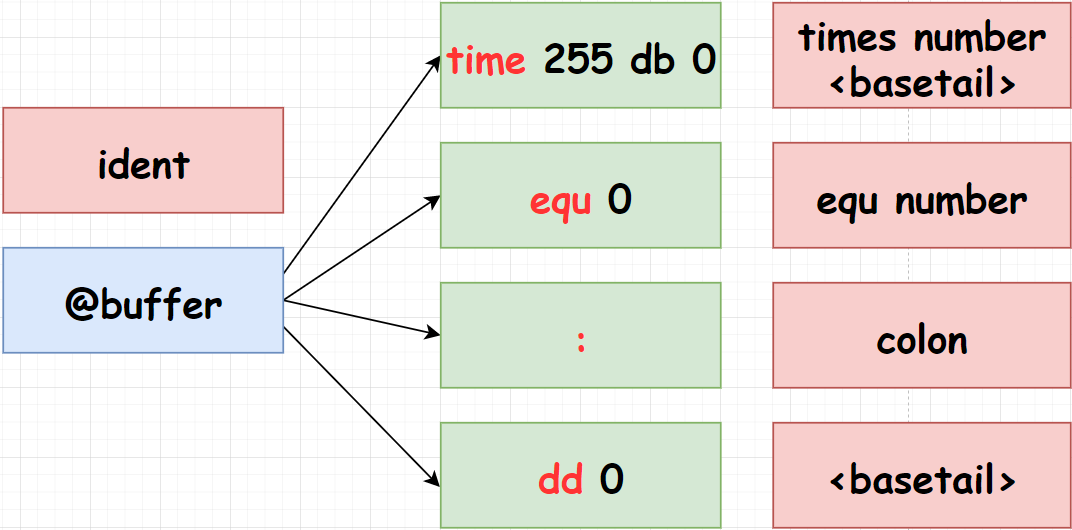


图53 <idtail> 推导示例

* <basetail> → <type\_len> <values>

<basetail> 会进一步定义符号的字长和类型。其中<type\_len>包括三种字长，db、dw和dd。分别是1、2和4字节。

* <values> → <type> <valuetail>
* <valuetail> → comma <type> <valuetail> |

这里类似于连续符号定义的文法，通过逗号分割，由于定义的数量是不定长，因此是右递归形式。<type>包括了立即数、字符串或者是标识符。

* <instruction> → mov <opr> comma <opr> … | …

| call <opr> … | …

| ret

接下来指令集，指令主要分为三类，二元操作符，以mov为代表；一元操作符，以call为代表，最后是零元操作符，只有一个，就是ret。这里<opr>是操作数，操作数可以是一个寄存器，可以是一个内存单元，包括立即寻址、偏移寻址、寄存器寻址等。

* <opr> → number | <reg> | <memory> | ident

这里操作数可以有4中类型，根据不同类型的终结符选择不同的分支。示例如图54所示。这里只是举一个mov的例子，虽然mov 1234,eax这个例子的语义是非法的，但它符合这个文法；与之前提到的语义检查道理一样，在文法分析阶段，并不检查语义的正确性。

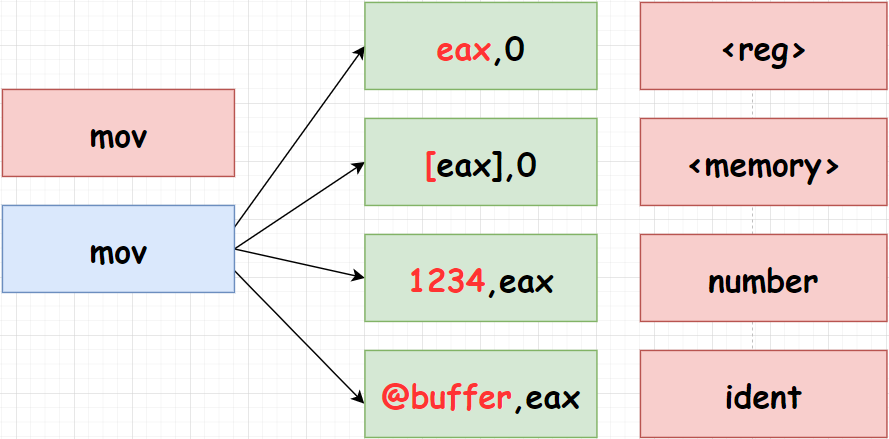


图54 <opr> 推导示例

其中<reg>是各种寄存器，这里一共是8 + 8，16种。分别是8 bit寄存器8种和32 bit寄存器8种。<memory>是内存，访问内存用一对[]括起来，这里也分为几种访问的方式，如下所示:

* <memory> → lbrac <addr> rbrac
* <addr> → number | ident | <reg><regaddr>

这里分支如图55所示，注意有两个分支都是以寄存器<reg>开头，它们在这里无法区分，因此会进一步进入到<regaddr>进行进一步推导。这其实是内存访问的几种方式，寄存器寻址，立即数寻址(标识符本质也是一个立即数)以及偏移寻址。这里识别不了[1234 + eax]这种模式，也就是说这个文法要求寄存器必须写在前面，否则认为是非法的文法，读者也可以自行修改文法，使得文法分析器支持这一文法。

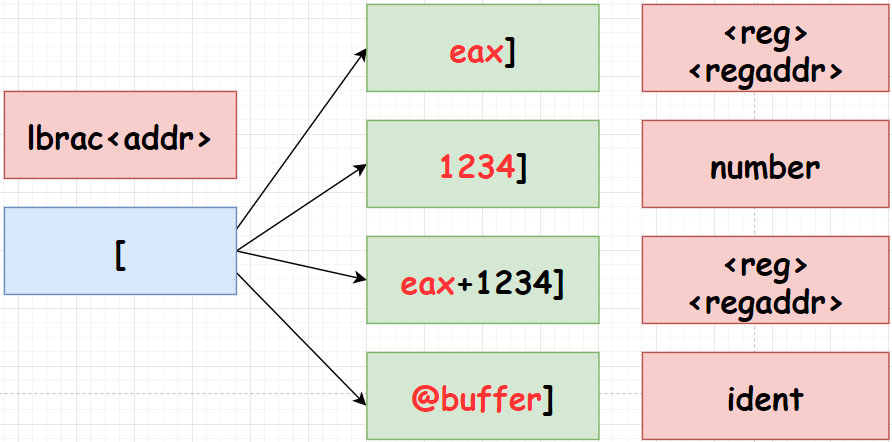


图55 <memory>推导示例

* <regaddr> → <offset> <regaddrtail> |
* <offset> → add | sub
* <regaddrtail> → number | reg

这里其实就是在进一步区分图55里第1个和第3个分支。它们对应不同的偏移寻址模式。

至此，我们的汇编器文法部分就介绍完了。

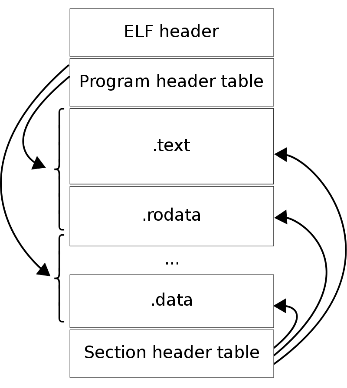
### 3.3 语义分析(Semantic Analyse)

语义部分的检查内容较少，这里只简单检查了二元操作数的类型，例如要求第一个操作数不能是立即数；两个操作数不能都是内存。

### 3.4 目标代码生成

#### 3.4.1 ELF 格式

这里的目标代码是x86-elf格式的代码。这里首先要简单了解ELF文件的格式[14]。ELF的全称是Excutable and Linkable Format，这是Unix的标准二进制文件格式。在Windows下标准可执行文件格式是PE格式。

如右图所示，ELF文件包括了ELF的头部和数据部分组成。数据部分包括程序头表(Program Header Table)和段头表(Section Header Table)。这两个表其实是看ELF的视图不同，前者是装载器(Loader)视角，即把多个属性相似的section看作一个部分，因为要装载进内存，分配相应的地址；后者是链接器(Linker)看待程序的视角，链接器需要合并不同二进制程序的相同的节。

ELF文件的具体结构其实可以去头文件elf.h中阅读，有非常详细的注释。

程序头一共52 Bytes，包括的信息很丰富，在linux下使用readelf命令查看一个二进制文件格式。如图56所示。

这里包括了幻数、一些版本信息以及程序的入口点，拥有的section数量，每个section的大小等等，这些信息都被记录在了ELF头中。

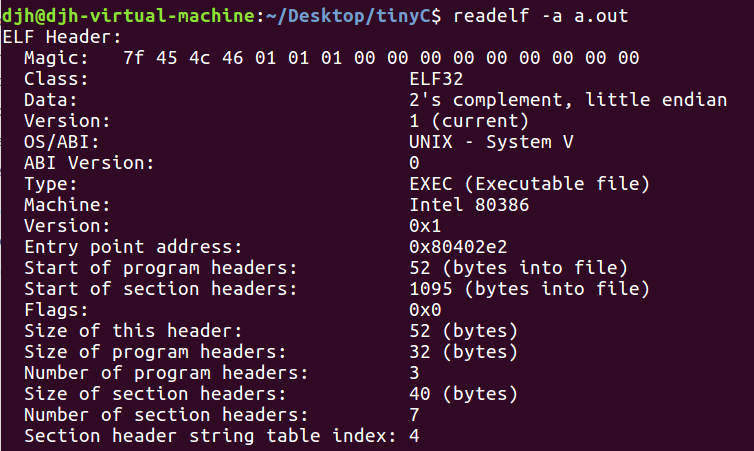


图56 ELF文件头

ELF拥有的section种类其实非常多，甚至你可以自己创建想要的section，但在实际实现中，我们没有使用动态链接、延迟绑定这样的技术，因此许多section就不需要了，产生得到的ELF文件只包含了最基本的section，下面就对这些section进行简单描述，首先还是看看每个Section Header部分，这是描述每个section的一个部分，它的大小是固定的，ELF文件的最后有一个节头表，其中的每一项就是Section Header，大小是40 Bytes。如表8所示。这里section的翻译既可以是段，也可以是节，一般section被翻译为节，segment被翻译为段，在没有歧义时，可以混用。

表8 Section Header内容

|  |  |  |
| --- | --- | --- |
| 类型/字节 | 名字 | 含义 |
| ELF32\_Word/4 | sh\_name | section name,其index在字符串表中的偏移 |
| ELF32\_Word/4 | sh\_type | sectiond类型，比如是字符串段还是符号段等 |
| ELF32\_Word/4 | sh\_flag | section属性值，可写/可执行/可占内存等 |
| ELF32\_Addr/4 | sh\_addr | section地址(address) |
| ELF32\_Off/4 | sh\_offset | section相对文件头的偏移(offset) |
| ELF32\_Word/4 | sh\_size | section的大小(size) |
| ELF32\_Word/4 | sh\_link | 一般为SHN\_UNDEF/,SYM\_TAB等需要特殊考虑 |
| ELF32\_Word/4 | sh\_info | 一般为0,SYM\_TAB等需要特殊考虑 |
| ELF32\_Word/4 | sh\_addralign | section地址对齐约束 |
| ELF32\_Word/4 | sh\_entsize | 如果节包含固定大小的对象，例如符号表，sh\_entsize记录每个对象的大小;否则取0 |

回到图56，上面显示Section Header是40 Bytes，表8中一共有10项，每一项都是4个字节刚好40个字节。这里我们用到了几个节如表9所示。其它常见的节部分与动态链接有关，这里就不展开了。

有一些段相对比较好理解，例如代码段(.text)和数据段(.data)，在汇编文件中也比较明显，.bss段存放未初始化的变量，但.bss不占文件空间，在内存中一般会被初始化为0。其它几个段相对陌生一点，下面会简单说明。

表9 代码生成中涉及的section

|  |  |  |  |
| --- | --- | --- | --- |
| 节名称 | sh\_type | sh\_flag | 备注 |
| .text | SHT\_PROGBITS | SHF\_ALLOC+  SHF\_EXECINSTR | 代码段，可占内存/可执行 |
| .symtab | SHT\_SYMTAB | SHF\_ALLOC/0 | 这个节包含了符号表 |
| .strtab | SHT\_STRTAB | SHF\_ALLOC/0 | 字符串表，存储文件中字符串信息 |
| .shstrtab | SHT\_STRTAB |  | 存储节名称的字符串表 |
| .relname | SHT\_REL | SHF\_ALLOC/0 | 重定位表.rel.text/.rel.data |
| .data | SHT\_PROGBITS | SHF\_ALLOC+  SHF\_WRITE | 数据段，存储初始化了的数据 |
| .bss | SHT\_NOBITS | SHF\_ALLOC+  SHF\_WRITE | 不占用文件空间，内存中初始化为0 |

* **.strtab(字符串表)/.shstrtab(段表字符串表)**

ELF文件中一般会用到很多字符串，例如符号名、段名等，它们长度通常不固定，因此字符串表把它们全部放在一起，用0分割，这样只要给出起始偏移，就能读取相应的字符串内容。这两个段都是存储字符串的，但不同之处在于，一般的字符串表存储一般的字符串，而段表字符串表存储的都是段名(section name)。

对于section name而言，这个偏移量会被存储在sh\_name域中。这样的做法也相对节约空间，没有额外空间浪费。并且如果有相同后缀的字符串，还可以考虑压缩存储，进一步节约空间。

在图56中的最后一个字段，e\_shstrndx(Section header string table index)，也就是段表字符串表(.shstrtab)在**段表**(Section Header Table)中的下标。这样的好处是，当读取ELF文件头时，就可以得到段表和段表字符串表，从而进一步分析整个ELF文件。

* **.symtab(符号表)**

符号表的结构是整齐统一的，它是一个大数组，数组的每个元素都有固定的结构，即ELF32\_Sym，如表10所示。

表10 ELF32\_Sym结构

|  |  |  |
| --- | --- | --- |
| 类型/字节 | 名字 | 含义 |
| ELF32\_Word/4 | sh\_name | section name,其index在字符串表中的偏移 |
| ELF32\_Addr/4 | sh\_value | 符号对应的值 |
| ELF32\_Word/4 | sh\_size | 符号大小(size),即符号占用的字节数 |
| u\_char/1 | sh\_info | 符号类型即绑定信息 |
| u\_char/1 | sh\_other | 0 |
| ELF32\_Half/2 | sh\_shndx | 符号所在段,对应段表中的下标 |

**(1) sh\_info**:低4位标识符号的类型，即这个符号是函数(FUNC)，还是段(SECTION)，还是变量(OBJECT)等情况；高4位表示符号绑定的信息(LOCAL/GLOBAL/WEAK)，这个符号是全局还是局部还是弱符号。

**(2) sh\_shndx**:表示符号所在段对应段表中的下标(index)，例如这个符号在.text段，那么sh\_shndx的值是.text段在段表(Section Header Table)中的下标。比较特殊的情况是，这个符号是外部引用符号，不在此文件中，这一值会被设置位UDEF。例如对于一个hello-world程序，使用了printf函数输出，使用gcc只编译，不链接，最后观察符号表，printf这个符号的sh\_shndx就是SHN\_UDEF；还有一些其它特殊情况，这里就不列举了。

**(3) sh\_value**:符号的值，一般而言，如果这个符号是函数或者变量，那么这个值就是地址，这个地址是该符号在段内的相对偏移量。

* **.relname(重定位表)**

汇编产生的代码本身是可重定位文件，因为它们还没有被正式链接。为什么需要重定位，因为没有正式分配地址之前，一些符号的地址不能确定，需要交给链接器进行地址修正。因此重定位表会记录哪些符号需要重定位，重定位表和符号表有相似之处，它的每个元素都是一个对象，这个类型是ELF32\_Rel，8个字节，如表11所示。

表11 ELF32\_Rel结构

|  |  |  |
| --- | --- | --- |
| 类型/字节 | 名字 | 含义 |
| ELF32\_Addr/4 | r\_offset | 入口地址偏移 |
| ELF32\_Word/4 | r\_info | 低8位表示重定位类型  高24位表示重定位符号在符号表中的下标 |

这里其实涉及了两种重定位类型，一种是绝对地址修正(R\_386\_32)，另外一种是相对地址修正(R\_386\_PC32)。

#### 3.4.2 IA-32指令集

IA-32指令集其实比较复杂，文档很长，但这里涉及到的指令其实比较少。如何阅读i386指令系统，可以参考NJU的一份文档[15]。简言之，整个指令主要由以下几个部分组成。如图57所示。

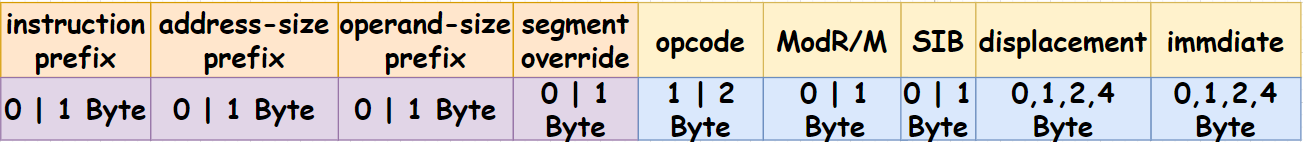


图57 指令构成

尽管指令很复杂，但是这里我们只用到了少量指令，所以前缀部分(橘色)可以忽略，只需要看淡黄色部分的结构即可。

第一个部分opcode是指令的基本编码，长度为1-2个字节，这个域是唯一不可以缺省的，其它部分都可能没有。

后面的两个域分别是MOR/M和SIB，其结构如图58所示。这两个部分的长度为1个字节(或者为0)，这一个字节又被拆成了三个部分。它们的含义后续会介绍。

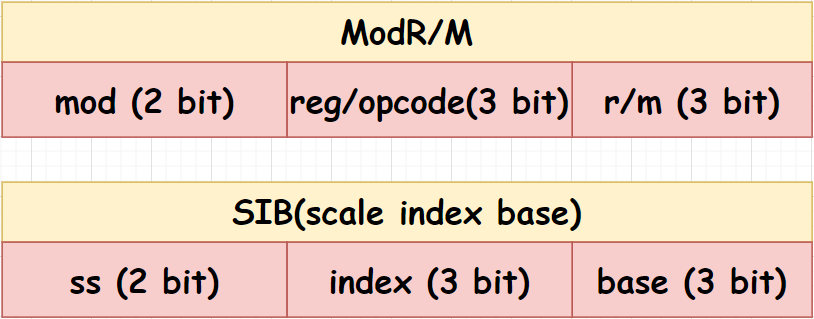


图58 ModR/M 和SIB结构

以ADD指令为例，通过目录索引到ADD指令，可以看到，一条简单的ADD指令，对应很多种情况。如图59所示。我们只关注了8 bit和32 bit这两种情况，其它也是类似的。

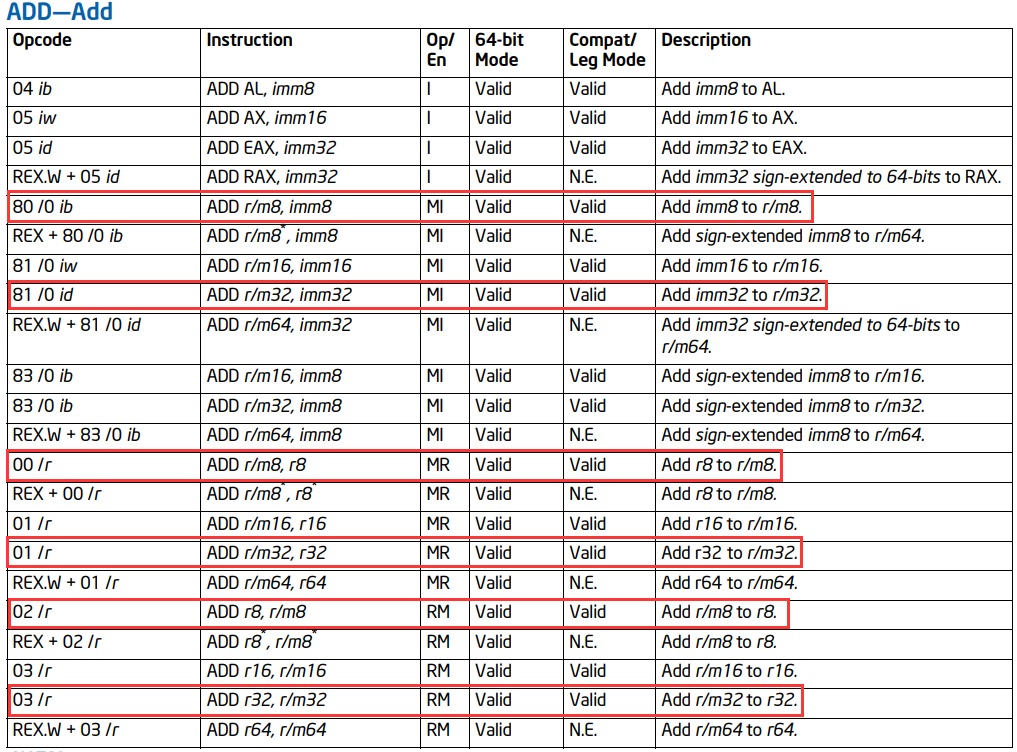


图59 IA-32 ADD指令

第一列是Opcode，如果后面有/r表示Opcode后面会使用ModR/M域，并且ModR/M的reg/opcode域被解析为通用寄存器的编码。reg/opcode的长度是3 bit，因此最多表示8个寄存器，在不同位(8/32)情况下，使用的寄存器刚好也是8个，如表12所示(其实就是表6)。

表12 汇编部分使用的寄存器

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| AL | CL | DL | BL | AH | CH | DH | BH |
| EAX | ECX | EDX | EBX | ESP | EBP | ESI | EDI |

如果Opcode后跟着/ib、/id一类的表示8 bit和32 bit通用寄存器的编码，但是和第一种情况不同，这个值并不是写在ModR/M的reg/opcode域，而是直接加在Opcode中，即Opcode+=寄存器编码。

如果Opcode后面跟着/0，表示会使用ModR/M字节，并且ModR/M字节的reg/opcode域扩展Opcode的一部分，取值为0。/digit同理，这里不仅仅可以是0，还可以是其它数字(0-7)。

第二列是指令类型，imm32表示32 bit的立即数；imm8同理，表示8 bit立即数；r/m表示寄存器或者内存；r表示寄存器。r/m表示寄存器还是内存，需要由ModR/M的mod域区分，当mod=3时，表示寄存器，其余取值表示不同类型的内存访问。具体是一张很大的表，参考指令集文档。

特别地，当r/m域为4时，后面会使用SIB部分，SIB主要解决存在比例因子缩放的指令，例如2 \* EAX + ECX这样的情况。

SIB寻址描述为Base + index \* 2^scale。这里同样有一张大表，参考指令集文档吧。当index = 4时，index实际取0，此时相当于基址寻址(仅Base有效)；当Base=5时，不使用任何基址寄存器，此时只用变址寻址(仅index和scale有效)。实际编程中，没有使用到scale，这里只需要处理[ESI + EDI]这样的指令即可。

## 4.链接(Linker)

链接的本质是合并多个目标文件的相同的section，并修正哪些需要被重定位的符号。一开始会构建两个集合，一个是全局符号集合，即那些可以被导出的符号，这些符号可能会被外部文件引用，另外一个集合是没有被解析的符号。通过对比两个集合的元素，在全局符号集合中查找没有被解析的符号，如果能够找到，这个符号对应的地址被修正，然后标记为定义的状态。如果最后的所有未定义符号都能够被正确解析，那么可以链接成功，否则应该给出找不到某符号的错误。

## 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
10. https://en.wikipedia.org/wiki/Operators\_in\_C\_and\_C%2B%2B
11. https://en.wikipedia.org/wiki/IA-32
12. https://en.wikipedia.org/wiki/Complex\_instruction\_set\_computer
13. https://en.wikipedia.org/wiki/X86\_calling\_conventions
14. https://en.wikipedia.org/wiki/Executable\_and\_Linkable\_Format
15. https://nju-projectn.github.io/ics-pa-gitbook/ics2017/i386-intro.html