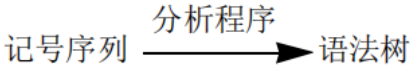
第三章 上下文无关文法及分析

分析的任务是确定程序的语法，或称作结构，也正是这个原因，它又被称作语法分析（syntax analysis）。

程序设计语言的语法通常是由上下文无关（ context-free grammar）的文法规则（grammar rule）给出。

上下文无关文法的规则是递归的（recursive），因此，用作表示语言语义结构的数据结构现在也必须是递归的。称作分析树（parse tree）或语法树（syntax tree）。

分析过程



记号序列通常不是显式输入参数，但是当分析过程需要下一个记号时，分析程序就调用诸如getToken的扫描程序过程以从输入中获得它。

上下文无关文法

上下文无关文法说明程序设计语言的语法结构，并涉及到了递归规则。

eg:

$$exp \rightarrow exp\ op\ exp \mid (exp) \mid \mathbf{number}$$
$$op \rightarrow + \mid - \mid *$$

BNF

- 名字用斜体表示
- 竖线表示作为选择的元符号
- 并置用作一种标准运算
- 没有重复的元符号
- 用箭头符号“→”代替了等号来表示名字的定义
- 文法规则将正则表达式作为部件
- 称作Backus - Naur范式（Backus-Naur form）或BNF文法。

上下文无关文法规则说明

文法规则定义在一个字母表或符号集之上，符号通常是表示字符串的记号。

BNF中的上下文无关文法规则是由符号串组成：

$$exp \rightarrow exp\ op\ exp \mid (exp) \mid \mathbf{number}$$
$$op \rightarrow + \mid - \mid *$$

- 结构名字
- 元字符 →
- 符号串，每个符号都是字母表中的一个符号（即一个结构的名字）或是元符号 |

规则定义了在箭头左边名字的结构。这个结构被定义为由被竖线分隔开的选择右边的一个选项组成。每个选项中的符号序列和结构名字定义了结构的布局。

推导及由文法定义的语言

文法规则通过推导确定记号符号的正规串。推导（derivation）是在文法规则的右边进行选择的一个结构名字替换序列。推导以一个结构名字开始并以记号符号串结束。在推导的每一个步骤中，使用来自文法规则的选择每一次生成一个替换。

eg：注意这里不能再继续写（34 - 3）* 42了，推导到第八步就已经ok了

Example of Derivation

exp → *exp op exp* | (*exp*) | **number**
op → + | - | *

(34 –3) * 42

1. *exp*=> *exp op exp*
2. => *exp op* **number**
3. => *exp* * **number**
4. => (*exp*) * **number**
5. => (*exp op exp*) * **number**
6. => (*exp op* **number**) * **number**
7. => (*exp* – **number**) * **number**
8. => (**number** – **number**) * **number**
9. => (34 – 3) * 42

Grammar rules define →

Derivation steps construct by replacement =>

通过推导得到的所有记号符号的串集就是文法定义的语言

文法规则也被称为产生式，因为他们通过推导产生正规串

文法中的每一个结构名定义了符合语法的记号串的自身语言，第一个规则称为开始符号

结构名：由于在推导中必须被进一步替换（不终结推导），所以结构名也称作**非终结符**（nonterminals）

符号：由于字母表中的符号终结推导，所以被称为**终结符**（terminals）

eg：箭头左边的非终结符，除了非终结符全是终结符

Example 3.1

- Let G be a grammar defined by the rule
  E → (E) | a
- This grammar has one nonterminal E and three terminals (,) , and a.
- This grammar generates language:
- L(G) = { a, (a), ((a)), (((a))), ...}
- Derivation for ((a))
- E => (E)
- => ((E))
- => ((a))

乔姆斯基文法分类

来源：https://blog.csdn.net/ynd_sg/article/details/78760408

一般的文法至少都是0型文法，也就是说0型文法限制最少，而1,2,3型文法都是在0型文法基础上加以限制形成。

若将0型文法比作基类的话，1,2,3就是不断继承并加以限制得到的子类。

① 0型文法

设G=（VN，VT，P，S），如果它的每个产生式α→β是这样一种结构：α∈(VN∪VT)*且至少含有一个非终结符，而β∈(VN∪VT)*，则G是一个0型文法。0型文法也称短语文法。**0型文法的能力相当于图灵机(Turing)。**

我们发现0型文法是限制最少的一种文法，平时见到的文法几乎都属于0型文法。

② 1型文法

1型文法也叫上下文有关文法，此文法对应于线性有界自动机。它是在0型文法的基础上每一个α→β,都有|β|>=|α|。这里的|β|表示的是β的长度。

注意：虽然要求|β|>=|α|，但有一特例：α→ε也满足1型文法。

如有A->Ba则|β|=2,|α|=1符合1型文法要求。反之,如aA->a，则不符合1型文法。

③ 2型文法

2型文法也叫上下文无关文法，它对应于下推自动机。2型文法是在1型文法的基础上,再满足：每一个α→β都有α是非终结符。如A->Ba,符合2型文法要求。

如Ab->Bab虽然符合1型文法要求,但不符合2型文法要求，因为其α=Ab，而Ab不是一个非终结符。

④ 3型文法

3型文法也叫正规文法，它对应于有限状态自动机。它是在2型文法的基础上满足:A→α|αB（右线性）或A→α|Bα（左线性）。

如有：A->a,A->aB,B->a,B->cB，则符合3型文法的要求。但如果推导为:A->ab,A->aB,B->a,B->cB或推导为:A->a,A->Ba,B->a,B->cB则不符合3型方法的要求了。具体的说，例子A->ab,A->aB,B->a,B->cB中的A->ab不符合3型文法的定义,如果把后面的ab,改成“一个非终结符＋一个终结符”的形式（即为aB）就对了。例子A->a,A->Ba,B->a,B->cB中如果把B->cB改为B->Bc的形式就对了，

因为A→α|αB（右线性）和A→α|Bα（左线性）两套规则不能同时出现在一个语法中,只能完全满足其中的一个,才能算3型文法。

其实四种文法就是规定产生式的左和右边的字符的组成规则不同而已，于是我们便可以严格根据左边和右边规则的不同来加以判断。

首先，应该明确，四种文法，从0型到3型，其规则和约定越来越多，限制条件也越来越多，

所以我们应该按照3->2->1->0的顺序去判断，一旦满足前面的规则，就不用去判断后面的了。

1.先来看看3型文法的判断规则

①：左边必须**只有一个字符**，且必须是**非终结符**；

②：其右边最多只能有**两个字符**，要么是一个**非终结符+终结符**（终结符+非终结符），要么是一个**终结符**。

③：对于3型文法中的所有产生式，若其右边有两个字符的产生式，这些产生式右边两个字符中终结符和非终结符的**相对位置**一定要固定，也就是说如果一个产生式右边的两个字符的排列是：终结符＋非终结符，那么所有产生式右边只要有两个字符的，都必须满足终结符+非终结符。反之亦然。

2.再看看2型文法判断规则

①：与3型文法的第一点相同，即：左边必须有且仅有一个非终结符。

②：2型文法所有产生式的右边可以含有若干个终结符和非终结符（只要是有限的就行，没有个数限制）。

3.最后再看看1型文法判断规则

①：1型文法所有产生式左边可以含有一个、两个或两个以上的字符，但其中必须至少有一个非终结符。

②：与2型文法第二点相同，但需要满足|α|≤|β|.

0型文法不需要判断了，一般的文法都是0型文法。

总结:

乔姆斯基分类法则

	0型文法	1型文法	2型文法	3型文法
文法名称	非限制性文法	上下文有关文法	上下文无关文法	正则文法
对应机器	图灵机	线性自动机	下推自动机	有穷自动机
识别对象	自然语言	受限自然语言	程序语言	单词

分析树和抽象语法树

左递归和右递归：看和左侧非终结符一样的非终结符写在左还是右

Left and Right Recursion

- Left recursive grammar:
- A → A a | a
- A → A α | β
  - Equivalent to β α *
  - β、βα、βαα 、βααα 、……
- Right recursive grammar:
- A → a A | a
- A → α A | β
  - Equivalent to α * β
  - β、αβ、ααβ、αααβ 、……

分析树

与推导相对应的分析树是一个作了标记的树

- 内部节点由非终结符标出

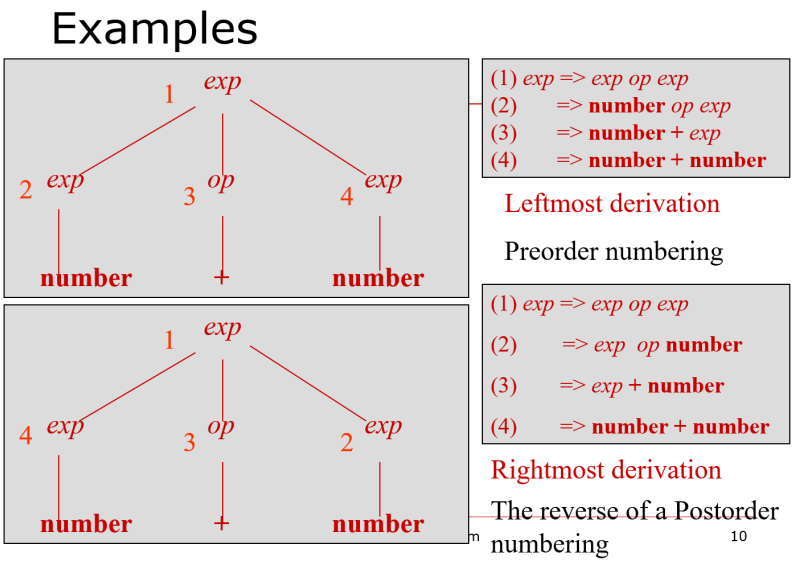
- 树叶节点由终结符标出
- 每个内部节点的子节点都表示推导的一个步骤中相关非终结符的替换

分析树可能与许多推导相对应，但仍有可能找出那个与分析树唯一相关的推导。

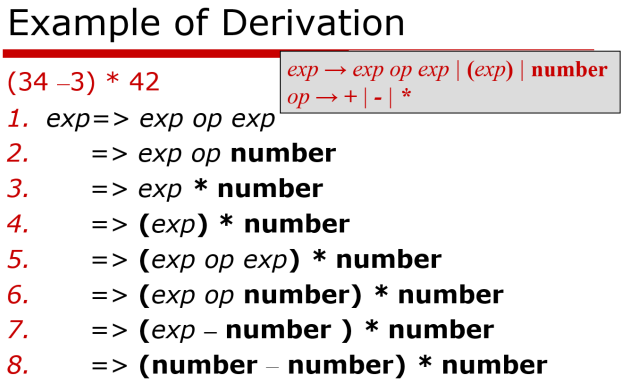
最左推导：每一步中，最左的非终结符都要被替换

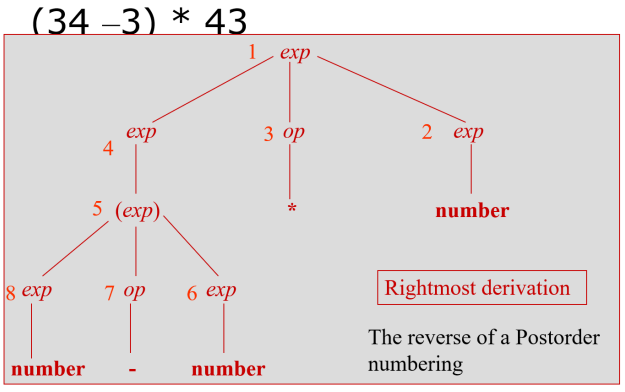
最右推导：每一步中，最右的非终结符都要被替换

最左推导和分析树的前序遍历对应，最右推导和后序遍历的逆序对应



eg:

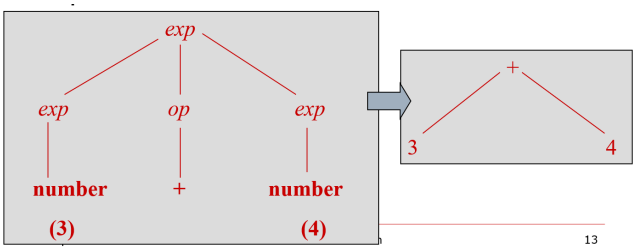


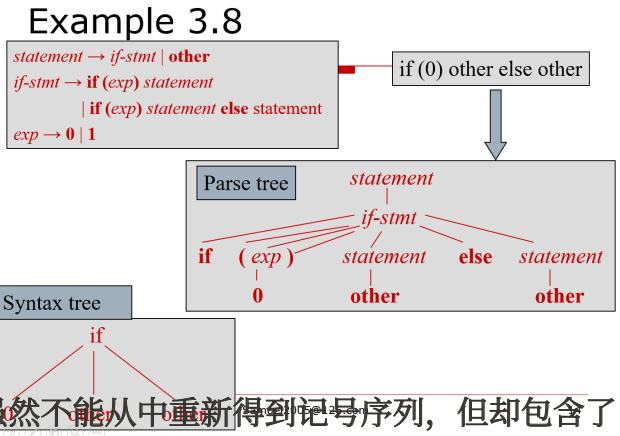


抽象语法树

分析树包括了比纯粹为编译生成可执行代码所需更多的信息

eg:





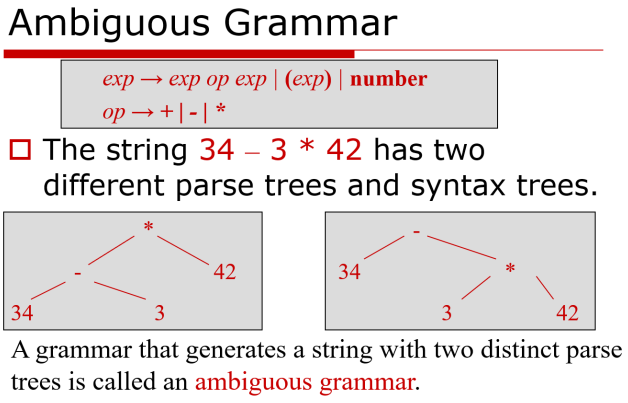
抽象语法树：是真正的源代码记号序列的抽象表示。虽然不能从中重新得到记号序列，但却包含了转换所需的所有信息，而且比分析树效率更高。

二义性

二义性文法

注意：不能通过算法判断是否有二义性

eg：34 - 3 * 42有两个不同的分析树和抽象语法树



可生成带有两个不同分析树的串的文法称为二义性文法（ambiguous grammar）

两个消除二义性的方法：

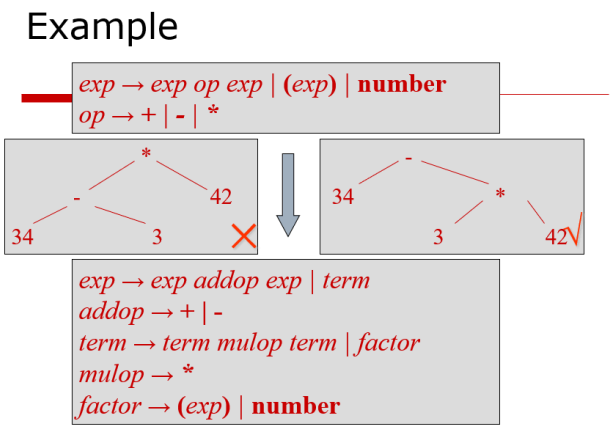
- 消除二义性规则：设置一个规则，该规则可在每个二义性情况下指出哪一个分析树或语法树是正确的
  - 优点：无需修改文法就可以消除二义性
  - 缺点：语言的语法结构再也不能由文法单独提供了
- 将文法改变成一个强制正确分析树的构造的格式

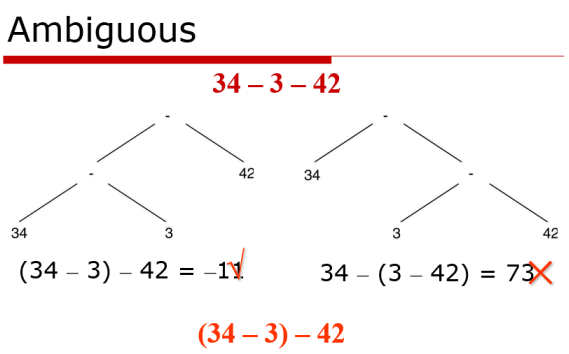
两种办法中，都必须确定在二义性情况下哪一个树是正确的。

优先权和结合性

有些计算比其他计算拥有更高的优先级，比如一般乘法比加法有更高的优先级。为了解决语法中运算符的优先程度，我们将具有相同优先级的运算符分成一组。

eg:





结合性

有些运算式是左结合的，比如一系列减法运算从左到右依次进行。

- exp → exp addop exp | term

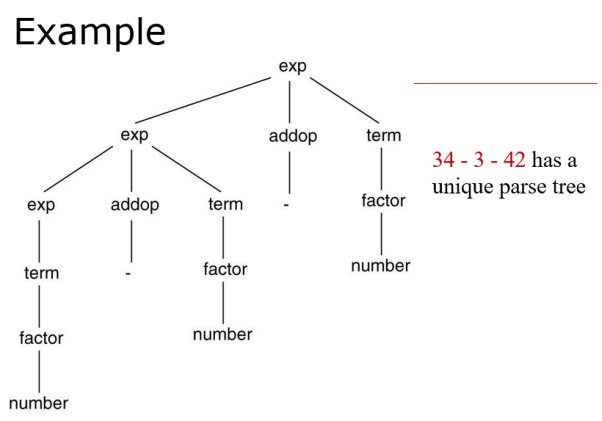
  运算符两边的递归允许任何一边在产生式匹配运算符的重复。

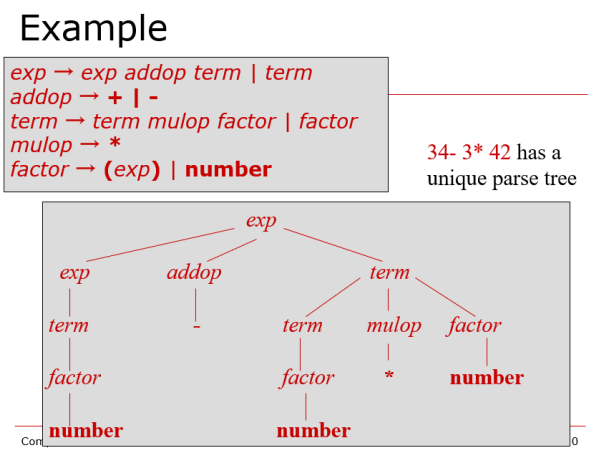
- exp → exp addop term | term

  右结合

总的来说，左递归规则使得算符在左边结合，右递归规则使得它们在右边结合

eg:



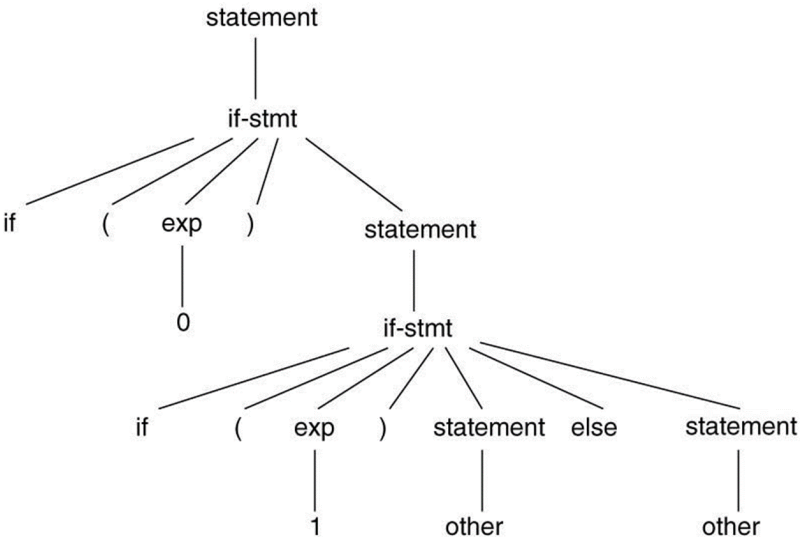


else悬挂问题

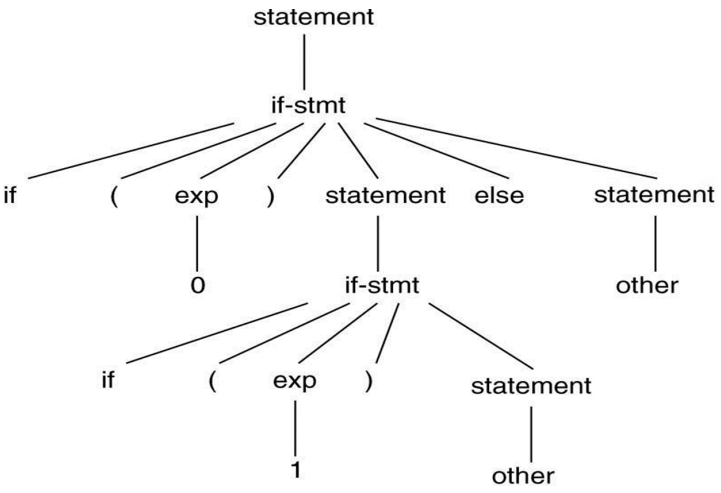
statement → *if-stmt* | **other**
if-stmt → **if** (*exp*) *statement*
| **if** (*exp*) *statement* **else** *statement*
exp → **0** | **1**

if (0) if (1) other else other 有两棵含义不同的分析树

if (0) if (1) other else other



if (0) if (1) other **else** other



1.修改语法

statement → *if-stmt* | **other**
if-stmt → **if** (*exp*) *statement*
| **if** (*exp*) *statement* **else** *statement*
exp → **0** | **1**

2.消除二义性规则：最近嵌套规则 if (0) if (1) other else other 是对的

statement → *matched-stmt* | *unmatched-stmt*
matched-stmt → **if** (*exp*) *matched-stmt* **else** *matched-stmt* | **other**
unmatched-stmt → **if** (*exp*) *statement* | **if** (*exp*) *matched-stmt* **else** *unmatched-stmt*
exp → **0** | **1**

无关紧要的二义性

有时文法可能会有二义性并且总是生成唯一的抽象语法树，这样的二义性称为无关紧要的二义性。

扩展的表示法：EBNF和语法图

EBNF表示法

EBNF : Extended BNF Notation

EBNF使用花括号{ }来表示重复

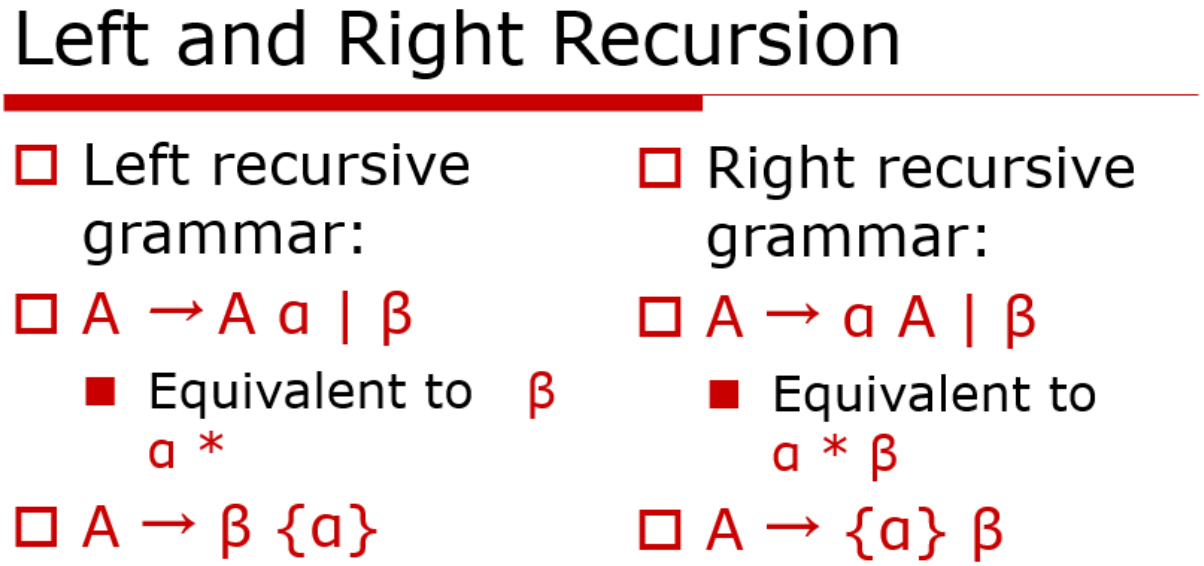
eg:

A → $\beta\alpha*$

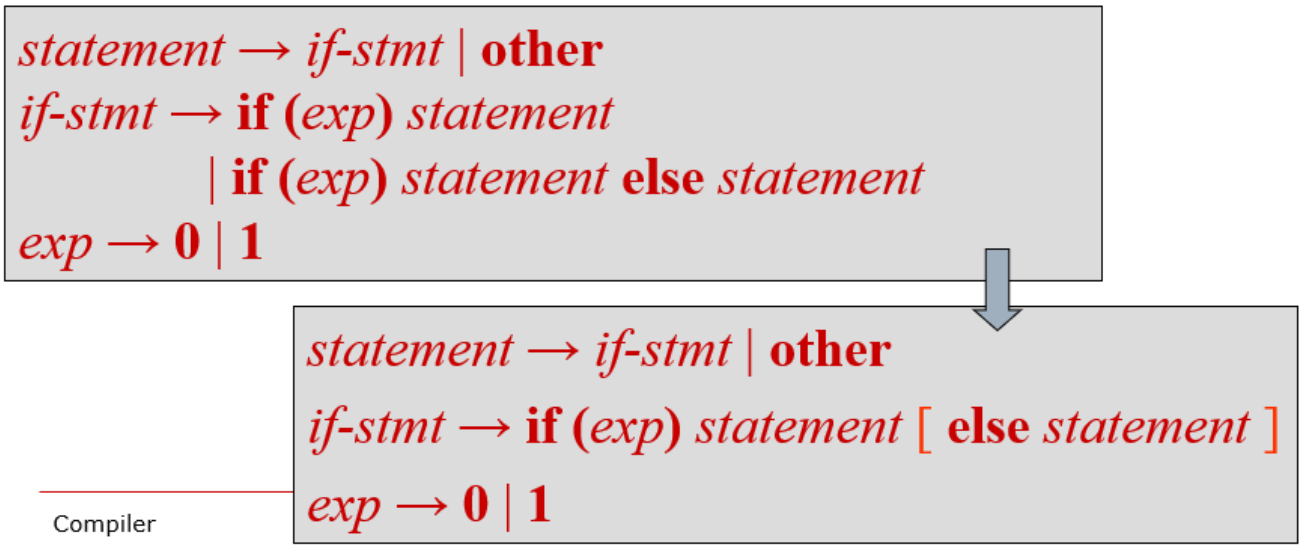
A → $\alpha * \beta$

则可以写成：A → β {α} 和 A → {α} β

左递归和右递归



EBNF中的可选结构可通过前后用方括号[ ]表示出来，eg:



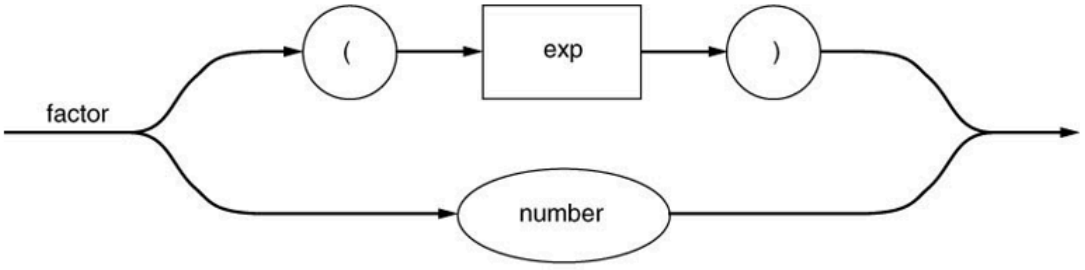
语法图

用作可视地表示EBNF规则的图形表示法称作语法图，它们由表示终结符和非终结符的方框、表示序列和选择的带箭头的线，以及每一个表示文法规则定义该非终结符的图表的非终结符标记组成。

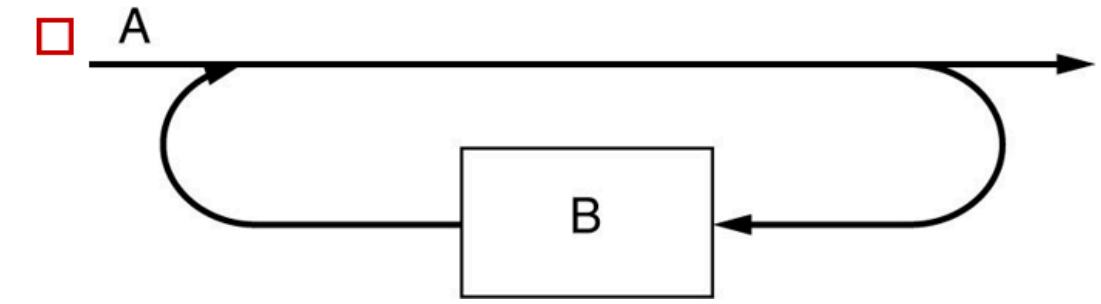
圆形框和椭圆形框用来指出图中的终结符，方形框和矩形框用来指出非终结符

eg:

- As an example, consider the grammar rule
- *factor* → (*exp*) | ***number***
- This is written as a syntax diagram in the following way:

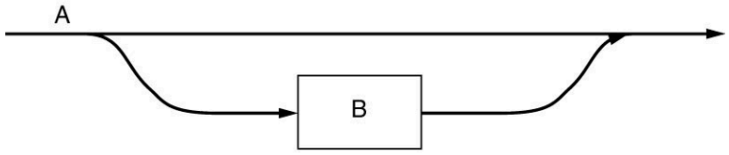


语法图是从EBNF而非BNF中写出来的，所以需要用图来表示重复和可选结构

- Syntax diagrams are written from the EBNF rather than the BNF, so we need diagrams representing repetition and optional constructs. Given a repetition such as
- $A \rightarrow \{ B \}$
- 

可选结构可以画作:

- An optional construct such as
- $A \rightarrow [B]$
- Is drawn as:



eg:

BNF包含结合性和优先权，要先改写成EBNF，改写规则可以参考左递归和右递归这里，对应着写

exp → *exp addop term* | *term*
addop → + | –
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | **number**

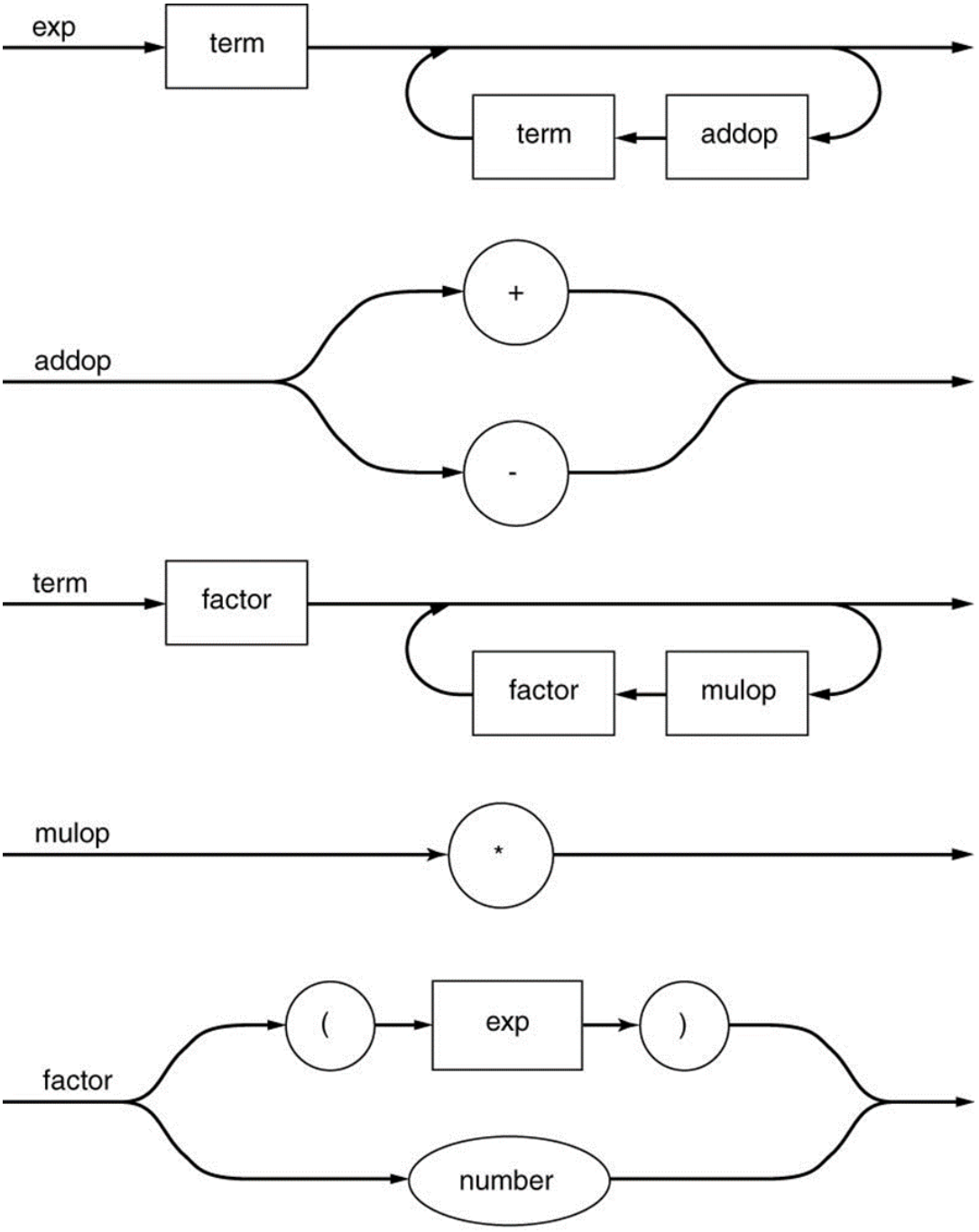
exp → *term* { *addop term* }
addop → + | –
term → *factor* { *mulop factor* }
mulop → *
factor → (*exp*) | **number**

Left and Right Recursion

- Left recursive grammar:
- A → A α | β
  - Equivalent to β α *
- A → β {α}

- Right recursive grammar:
- A → α A | β
  - Equivalent to α * β
- A → {α} β

画出对应的语法图：



eg：

step1：BNF→EBNF

statement → *if-stmt* | **other**
if-stmt → **if (***exp***)** *statement*
| **if (***exp***)** *statement* **else** *statement*
exp → **0** | **1**

statement → *if-stmt* | **other**
if-stmt → **if (***exp***)** *statement* [**else** *statement*]
exp → **0** | **1**

step2: