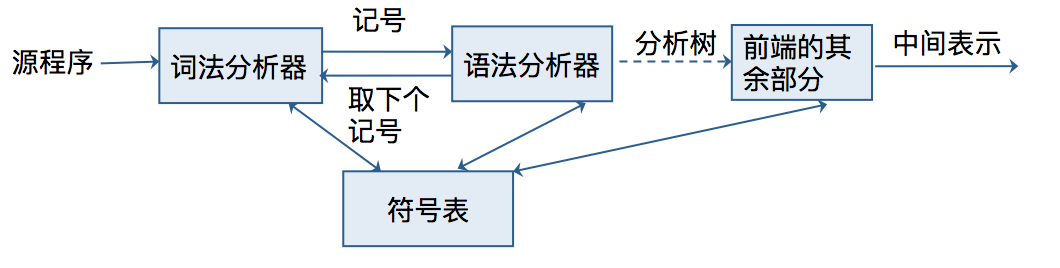
第四章 语法分析 →∪Σε

词法分析器的作用:

语法分析器的作用: 分析一个词法单元序列的语法



语法分析算法: 通用的、自顶向下的、自底向上的.

怎么来系统地描述程序设计语言的构造（比如表达式和语句）？？

怎么来描述一个合法的if语句 (C)?

*stmt* → **if** ( *expr* ) *stmt* **else** *stmt*

上下文无关文法的定义:

* 上下文无关文法，Context Free Grammar，**CFG**
* CFG是一个四元组（*VT , VN , S, P*）
  + *VT* : 终结符的有限集合 Terminal
  + *VN*: 非终结符的有限集合，Non-terminal 且*VN ∩VT* = *Ø*
  + *S* : 开始符号，非终结符中的一个
  + *P* : 产生式的有限集合，产生式形式:*A* →α ，其中*A∈VN ，α∈(VN∪VT )\**

如何来描述一个算术表达式（+-\*/和括号)

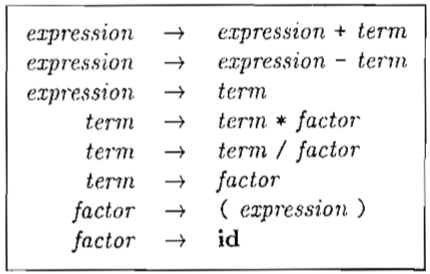
可以用正则表达式实现吗?

例(1) 一个简单的算术表达式上下文无关文法

终结符: {**id**, +, \*, −, /, (, )} abc \* efg + hij id \* id + id

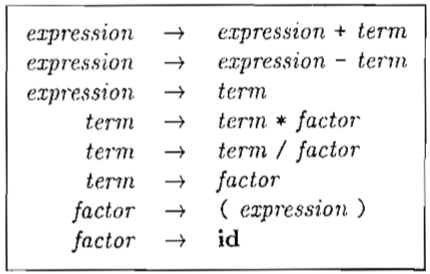
非终结符: { *expression*, *term, factor*} abc efg + hij id id + id

起始符: *expression*



符号表示的约定:

* 终结符
  + 字母表中的小写字母，如 a，b，c
  + 黑体串，如 **id, while**
  + 数字 0, 1, … , 9
  + 标点符号，如括号，逗号等
  + 运算符号，如+, -等
* 非终结符
  + 字母表中的大写字母，如 A, B, C
  + 字母 S，并且通常代表开始符号
  + 小写字母的名字（斜体），如 *expr*, *stmt*
  + 字母表中后面的大写字母，如X, Y, Z, 可以是终结符或非终结符
  + 字母表中后面的小写字母，如u, v,*…* z, 可代表终结符号串
  + 小写希腊字母，如，，等可代表文法的终结符号串
  + 对于A → 1,A → 2,… A → n可以写成A → 1|2|…|n
  + 除非特别说明，第一个产生式的头就是开始符号



→⇒

可简写为:

4.2.3 推导

* 把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替
  + *E* → *E* + *E* | *E* \* *E* | (*E* ) | − *E* | id
  + *E* ⇒ −*E* ⇒ −(*E*) ⇒ −(*E* + *E*) ⇒ −(id + *E*) ⇒ −(id + id)

*E* ⇒ –*E* 读作 E推导出-E

* 产生式A→γ, 且α、是文法的任意符号串，则αA ⇒ αγ
* 产生式A→γ, 且α、是文法的任意符号串，则αA ⇒ αγ称为**直接推导**
* 若α1⇒α2⇒*…*⇒αn，则为**零步**或**多步推导**，记为：α1⇒\*αn
  + 若α1=αn，为**零步推导**
  + 若α1≠αn，则为**至少一步推导**，记为：α1⇒+αn
* 两点注意：
  + ∀α，有α ⇒\*α，即推导具有**自反性**
  + 若α ⇒\* ，且⇒ γ, 则α ⇒\* γ ，即推导具有**传递性**

上下文无关文法 VS. 正则表达式 都是用来描述语言 (串的集合)

* 上下文无关语言
  + 由上下文无关文法生成的语言是上下文无关语言(CFL)。
  + 等价的文法
  + 如果两个文法产生同样的语言，则两个文法等价。
* 句型
  + 文法 G 的开始符为 S，S ⇒\* α，α可能含有非终结符，
  + 则 α 叫做文法 G 的句型。句子是只含终结符的句型。
* 例 E → E + E | E \* E | (E ) | − E | id ，写出串 −(id + id) 的推导
* 最左推导 (leftmost)：每次直接推导均替换句型中最左边的非终结符

E ⇒lm −E ⇒lm −(E) ⇒lm −(**E** + E) ⇒lm −(**id** + E) ⇒lm  −(id + id)

* 最右推导：每次直接推导均替换句型中最右边的非终结符

E ⇒rm −E ⇒rm −(E) ⇒rm −(E + **E**) ⇒rm −(E + **id**) ⇒rm  −(id + id)

练习1: 考虑下下文无关文法: S → S S + | S S \* | a

以及串aa+a\*

给出这个串的一个最左推导和一个最右推导.

S ⇒lm S S \* ⇒lm SS+ S \* ⇒lm a S+ S \* ⇒lm a a+ S \*⇒lm a a+ a \*

练习2: 求最左推导,

(1) S → S ( S ) S | e ( () () ) (匹配的括号序列)

(2) S → + SS | \* SS | a + \* aaa (前缀表达式)

(3) S → S + S | S S | ( S ) | S \* | a (a + a) \* a

(4) S → a S b S | b S a S | e aabbab (表述的语言是???)

所以由ab构成的字符串，并且a的出现次数和b的出现次数一致 (包括空串)

练习3: 为下面语言设计文法

(1) 所有由0和1组成的串的集合(包括空串);

(2) 所有由0和1组成的并且每个0之后都至少跟着一个1的串的集合;

(3) 所有由0和1组成的回文的集合, 即从前面和从后面读结果都是相同的串的集合;

(4) 所有由0和1组成的具有相同多个0和1的串的集合 (包括空串);

(5) 所有由0和1组成的，并且1的个数多于0的个数的串的集合;

(6) 所有由0和1组成的并且0的个数和1的个数不相同的串的集合.

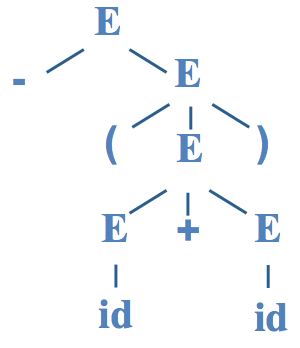
4.2.4 语法分析树

语法分析树是推导的图形表示形式. 它过滤掉了推导过程中对非终结符号应用产生式的顺序.

例(2) 文法E → E + E | E \* E | (E ) | − E | id

求串 –(id + id)的推导过程

*E* ⇒ −*E* ⇒ −(*E*) ⇒ −(*E* + *E*) ⇒ −(id + *E*) ⇒ −(id + id)



语法分析树的属性:

* 根结点是开始符号
* 叶子结点是终结符或ε
* 内部结点是一个非终结符
* 若A→x1x2…xn，则A是一个非终结符，x1x2…xn 是终结符或非终结符。

语法分析(Syntactic Parsing): 为给定的符号串找到一棵分析树的过程称为串的**语法分析**。

练习(3)

考虑下下文无关文法: S → S S + | S S \* | a

以及串aa+a\*

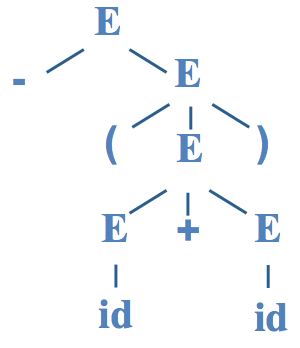
给出这个串的一个最左推导和一个最右推导. 并画出分析树.

最左推导: S → SS\* → SS+S\* → aS+S\* → aa+S\* → aa+a\*

最右推导: S → SS\* → Sa\* → SS+a\* → Sa+a\* → aa+a\*

每个语法分析树都和唯一的最左推导及唯一的最右推导相关联。

E → E + E | E \* E | (E ) | − E | id



即根据上图所示语法树，可以得到唯一的最左推导和唯一的最右推导。

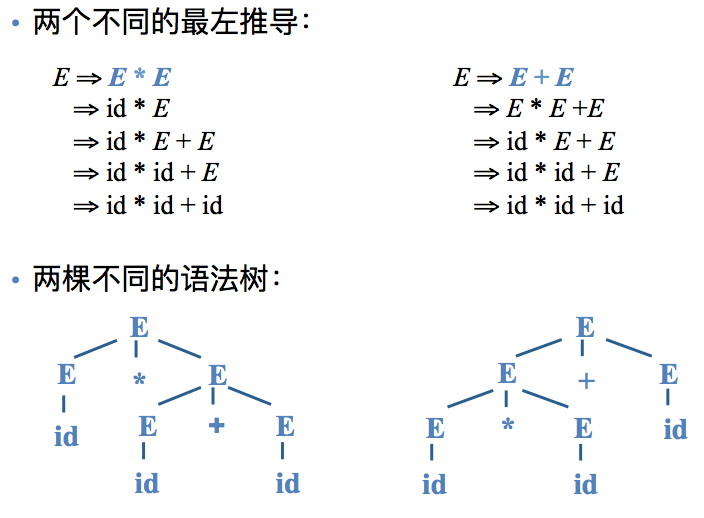
4.2.5 二义性

一个文法，如果存在某个句子不止一棵分析树，或者说这个句子存在不止一种最左（最右）推导，那么称这个文法是二义的 (ambiguous)。

(注：并不是对所有的句子都存在不止一棵分析树)

例(3) 给定文法, E → E + E | E \* E | (E ) | − E | id

求句子 id  id + id 的分析树



一个句子有多于一棵分析树，仅与文法和句子有关，与采用的推导方法无关；

练习(4) 文法

S → a S b S | b S a S | ε

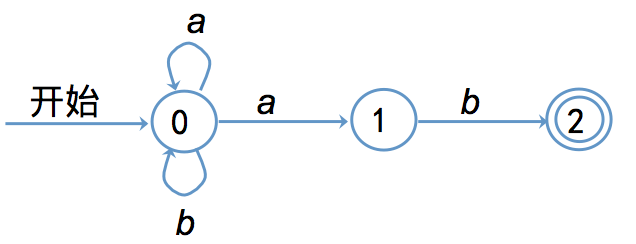
是否具有二义性?

4.2.7 上下文无关文法和正则表达式

正则式描述的语言结构均可用上下文无关文法描述，反之不一定

* 从正则式到上下文无关文法的转换步骤：
  1. 构造正规式的 NFA；
  2. 若 0 为初态，则A0为开始符号；
  3. 对于move(i,a)=j，引入产生式 Ai→aAj；
  4. 对于move(i,ε)=j，引入产生式 Ai→Aj；
  5. 若 i 是终态，则引入产生式 Ai →ε。

例(4) 求正则式 (a|b)\*ab等价的文法



文法

A0 → a A0 | b A0 | a A1

A1 → b A2

A2 → ε

例(5) 假设L ={anbn | n ≥ 1}，即表示由n 个a后面跟n个b的所有串的集合。为什么说该语言不能由正则表达式表示?

**文法: S → aSb | ab**

* L是不能用正则式描述的语言的一个范例
  1. 若存在接受L的DFA D，状态数为k个
  2. 设D读完ε, a, aa, …, ak 分别到达状态s0, s1, …, sk
  3. 至少有两个状态相同，例如是si和sj，则ajbi属于L