第三章 语法分析

Identifier	Operator	Identifier	Operator	Identifier	Operator	Literal	Separator
x	:=	y	+	z	*	60.0	;

CFL : Set of sentences

Syntactic Analyzer

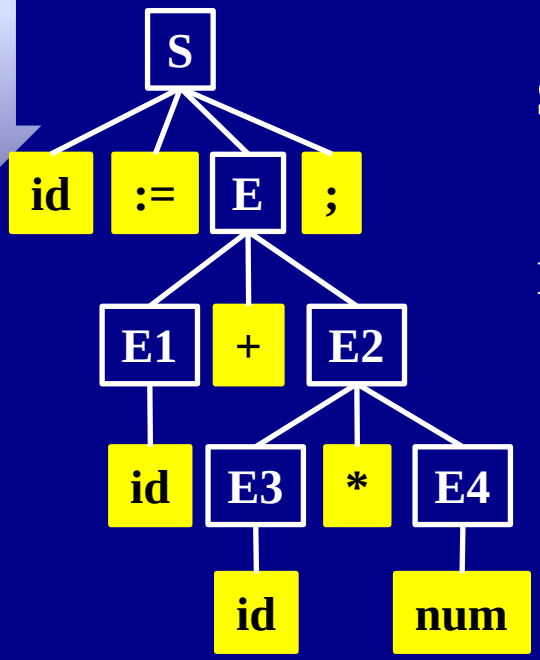
Pushdown Automata

Generation of a unique syntax tree

Description of syntax rules

Context Free Grammar (CFG)

S→id := E;
E→ E + E
| E * E
| id
| num



Start symbol (S)

Derive

Reduce

Token list
id:=id+id*num;

第三章 语法分析

词法分析：元素是字母表，组成字符串，线性结构，单词的集合

语法分析：元素是终结符，组成句子，树结构，句子的集合

语法的双重含意：

1. 语法规则：上下文无关文法（子集－LL文法或LR文法）
2. 语法分析：下推自动机（LL或LR分析器），自上而下和自下而上分析

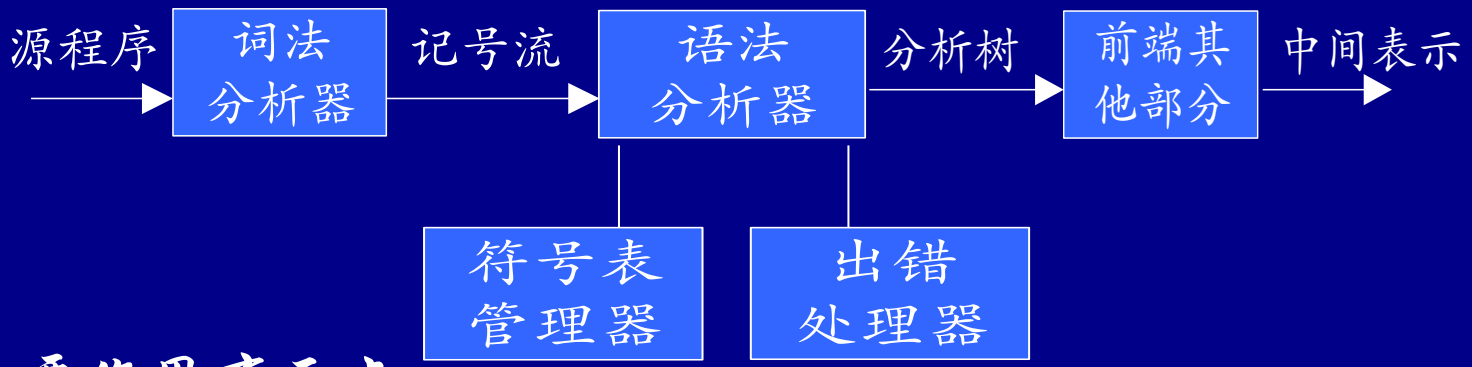
本章主要内容：

1. 与语法分析有关的基本概念和相关问题
2. 上下文无关文法
3. 自上而下分析
4. 自下而上分析

3.1 语法分析的基本术语

3.1.1 语法分析器的作用

语法分析器是编译器前端的重要组成部分，许多编译器，特别是由自动生成工具构造的编译器，往往其前端的中心部件就是语法分析器。语法分析器在编译器中的位置和作用下：



它的主要作用有两点：

<1> 根据词法分析器提供的记号流，为语法正确的输入构造语法树，这是本章的重点，在以后各节中详细讨论；

<2> 检查输入中的语法（可能包括词法）错误，并调用出错处理器进行适当处理，下边简单介绍语法错误处理的基本原则，而在以后的讨论中忽略此问题。

3.1.2 语法错误的处理原则

<1> 源程序中可能出现的错误

两类：语法错误和语义错误

① 词法错误如非法字符或拼写错关键字、标识符等

@　　intege　　20times

② 语法错误是指语法结构出错，如少分号、begin/end不配对等

begin　　x:=a+b　　y:=x;

③ 静态语义错误：如类型不一致、参数不匹配等

a,b:integer;　　x:array[1..10] of integer;

x:=a+b;

④ 动态语义错误(逻辑错误)：如无穷递归、变量为零时作除数等

while (t) { ...};　　a:=a/b;

大多数错误的诊断和恢复集中在语法分析阶段。一个原因是大多数错误是语法错误，另一个原因是语法分析方法的准确性，它们能以非常有效的方法诊断语法错误。

编译时想要准确诊断动态语义或逻辑错误有时是很困难的。

<2> 语法错误处理的目标

对语法错误的处理，一般希望达到以下基本目标：

1. 清楚而准确地报告错误的出现，地点正确、不漏报、不错报也不多报；
2. 迅速地从每个错误中恢复过来（以便分析继续进行）；
3. 不应使对语法正确源程序的分析速度降低太多。

<3> 语法错误的基本恢复策略

1. 紧急方式恢复：抛弃若干输入，直到遇到同步记号。
2. 短语级恢复：采用串替换的方式对剩余输入进行局部纠正（抛弃＋插入）。
3. 出错产生式：用出错产生式捕捉错误（预测错误）。预置型的短语级恢复方式。
4. 全局纠正：对错误输入序列x，找相近序列y，使得x变换成y所需的修改、插入、删除次数最少。

例3.1 下述两条是有语法错误的语句，其中第一条赋值句结束时忘记加分号，采用紧急恢复方式和短语级恢复方式的可能结果分别如下所示。

```
x := a + b
y := c + d;
```

1. 紧急方式：
```
x := a + b + d;
```
—— 丢弃b后若干记号，直到遇到+

2. 短语级恢复：
```
x := a + b;
```
—— 加入分号，使之成为一个赋值句
```
y := c + d;
```

3.2 上下文无关文法(Context Free Grammar, CFG)

3.2.1 CFG的定义与表示

定义3.1 CFG是一个四元组G = (N, T, P, S)，其中

(1) N是非终结符（Nonterminals）的有限集合；

(2) T是终结符（Terminals）的有限集合，且N∩T=Φ；

(3) P是产生式（Productions）的有限集合，

A→α，其中A∈N（左部），α∈(N∪T)*（右部），

若α=ε，则称A→ε为空产生式（也可以记为A →）；

(4) S是非终结符，称为文法的开始符号（Start symbol）。■

例3.2 简单算术表达式的上下文无关文法可表示如下：

N={E}　T={+, *, (,), -, id}　S=E

P:

E → E + E　(1)

E → E * E　(2)

E → (E)　(3)　(G3.1)

E → -E　(4)

E → id　(5)

<1> 由产生式集表示CFG

前提：若文法正确，第一个产生式的左部是文法开始符号S

则：　N是可以出现在产生式左边符号的集合，

T是绝不出现在产生式左边符号的集合（记号）

P: E → E + E　（1）

E → E * E　（2）　S=E

E → (E)　（3）　N={E}

E → -E　（4）　T={+, *, (,), -, id}

E → id　（5）

<2> 产生式的一般读法

可以将产生式中的记号→读作“定义为”或者“可导出”。

更一般的，“E → E + E”可用自然语言表述为“算术表达式定义为两个算术表达式相加”，

或者“一个算术表达式加上另一个算术表达式，仍然是一个算术表达式”。

<3> 终结符与非终结符书写上的区分

(a) 用大小写区分： E → id

(b) 用 “” 区分： E → "id"　　E → E "+" E

(c) 用<>区分： 〈E〉 → 〈E〉 + 〈E〉

约定：大写英文字母A、B、C表示非终结符；

小写英文字母a、b、c表示终结符；

小写希腊字母α、β、δ表示任意文法符号序列。

<4> 产生式的缩写形式

当若干个产生式具有相同的左部非终结符时，可以将它们简写为以下形式：

该产生式的左部是此非终结符，

右部是所有原来的右部用“|”分隔开，

例3.3 G3.1可以重写为如下形式：

E →	E + E	(1)		P:	E → E + E	(1)
	| E * E	(2)			E → E * E	(2)
	| (E)	(3)	(G3.2)		E → (E)	(3)
	| -E	(4)			E → -E	(4)
	| id	(5)			E → id	(5)

严格的巴克斯范式（BNF）写法：

E::= E + E | E * E | (E) | -E | id

用“|”连接的每个右部称为一个候选项，具有平等的权利。即id是一个表达式，-E也是一个表达式。

3.2.2 CFG产生语言的基本方法－推导

CFG（产生式）通过推导的方法产生语言。

通俗地讲，产生式产生语言的过程是从开始符号S开始，对产生式左部的非终结符反复地使用产生式；

将产生式左部的非终结符替换为右部的文法符号序列（展开产生式，用标记=>表示），直到得到一个终结符序列。

例3.4 用(G3.2)产生终结符序列-(id+id)可如下：

E → E + E (1)
| E * E (2)
| (E) (3)
| -E (4)
| id (5)

(G3.2)

E => -E by(4)
=> -(E) by(3)
=> -(E+E) by(1)
=> -(id+E) by(5)
=> -(id+id) by(5)

定义3.2 利用产生式产生句子的过程中，将产生式A→γ的右部代替文法符号序列αAβ中的A得到αγβ的过程，称为αAβ直接推导出αγβ，记作：αAβ⇒αγβ。

若对于任意文法符号序列α1，α2，...αn，均有α1⇒α2⇒...⇒αn，则称此过程为零步或多步推导，记为：$\alpha 1 \Rightarrow^{*} \alpha n$，其中，n=1时称为零步推导。

若α1≠αn，即推导过程中至少使用一次产生式，则称此过程为至少一步推导，记为：$\alpha 1 \Rightarrow^{+} \alpha n$。 ■

定义3.2强调了两点：

- 对于任意文法符号序列α，有$\alpha \Rightarrow^{*} \alpha$，即推导具有自反性；
- 若$\alpha \Rightarrow^{*} \beta$，$\beta \Rightarrow^{*} \gamma$，则$\alpha \Rightarrow^{*} \gamma$，即推导具有传递性。

定义3.3 由CFG G所产生的语言L(G)被定义为：

$$L(G) = \{ \omega \mid S \Rightarrow^{+} \omega \text{ and } \omega \in T^{*} \},$$

L(G)称为**上下文无关语言**(Context Free Language, CFL)，ω称为**句子**。

若$S \Rightarrow^{*} \alpha$，$\alpha \in (N \cup T)^{*}$，则称α为G的一个**句型**。 ■

定义3.4 在推导过程中，若每次直接推导均替换句型中最左边的非终结符，则称为**最左推导**，由最左推导产生的句型被称为**左句型**。 ■

类似的可以定义最右推导与右句型，最右推导也被称为**规范推导**。

再考察-(id+id)的推导过程（这是一个最左推导）：

E => -E => -(E) => -(E+E) => -(id+E) => -(id+id)
α1 α2 α3 α4 α5 α6

其中，α1是文法开始符号，α6是句子，其他αi (i=2、3、4、5)均是句型。

句型是一个相当广泛的概念，根据定义3.3可知，α1和α6同样也是句型。

E → E + E (1)
| E * E (2)
| (E) (3) (G3.2)
| -E (4)
| id (5)

3.2.3 推导与语法树

对于推导：

E => -E => -(E) => -(E+E) => -(id+E) => -(id+id)

它产生句子的方式很不直观，看起来十分困难。

语法树是推导的图形表示，它的表示很直观，并且同时反映语言结构的实质和推导过程。

定义3.5 对CFG G的句型，语法树被定义为具有下述性质的一棵树。

(1) 根由开始符号所标记；

(2) 每个叶子由一个终结符，或一个非终结符，或 ε 标记；

(3) 每个内部结点由一个非终结符标记；

(4) 若A是某内部节点的标记，且X1，X2，...，Xn是该节点从左到右所有孩子的标记，则A→X1X2...Xn是一个产生式。若A→ ε，则标记为A的结点可以仅有一个标记为 ε 的孩子。

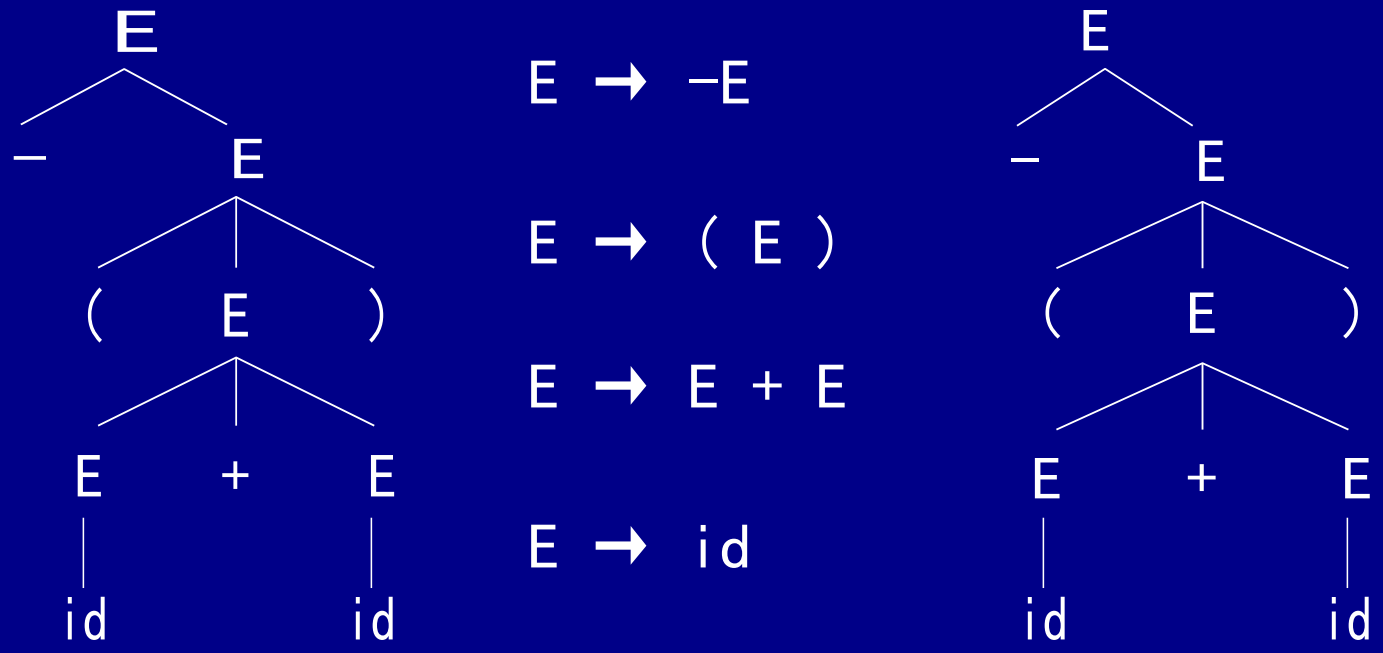
语法树与语言和文法的关系：

- 与文法的关系：每一直接推导（每个产生式），对应一棵仅有父子关系的子结构，即产生式左部非终结符“长出”右部的孩子；
- 与语言的关系：语法树的叶子，从左到右构成G的一个句型。若叶子仅由终结符标记，则构成一个句子。

例3.5　再考察-(id+id)的推导过程。

E => -E => -(E) => -(E+E) => -(id+E) => -(id+id)

用语法树的方式如下：



1. 最左推导和最右推导的中间过程对应的语法树可能不同，因为句型不同：-(id+E) 或 -(E+id)
2. 但是最终的语法树相同，因为最终是同一个句子：-(id+id)
3. 语法树既反映了产生句型的推导过程，又反映了句型的结构。

更多的情况下，仅关注句型结构，而忽略推导过程。

定义3.6 对CFG G的句型，表达式的抽象语法树被定义为具有下述性质的一棵树：

(1) 根与内部节点由表达式中的操作符标记；

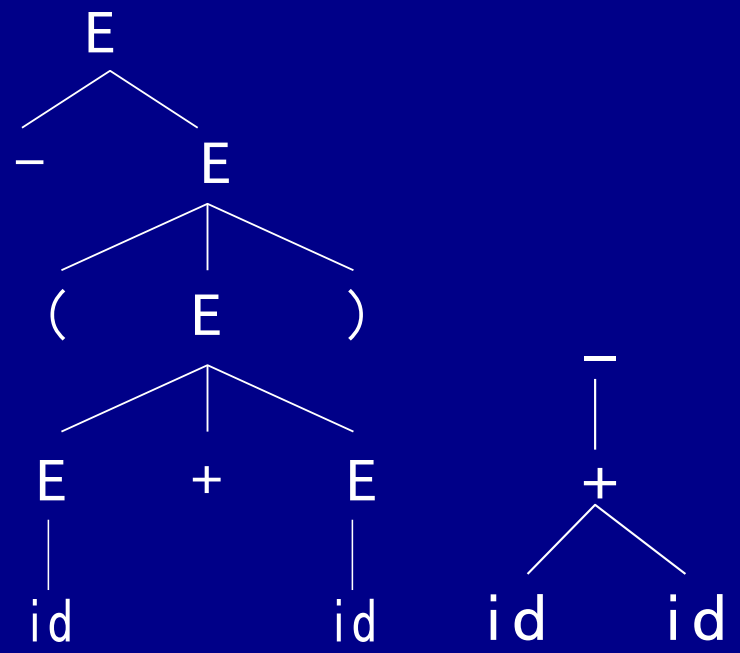
(2) 叶子由表达式中的操作数标记；

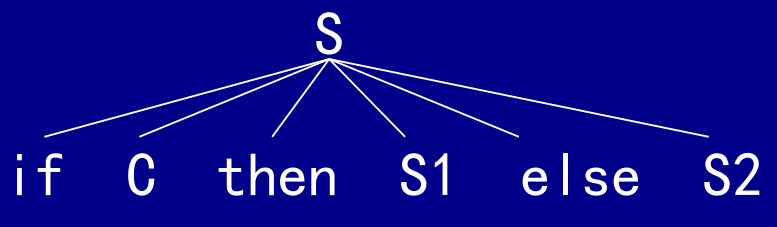
(3) 用于改变运算优先级和结合性的括弧，被隐含在语法树的结构中。 ■

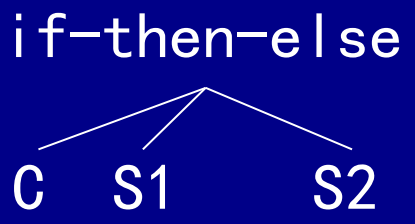
实质上，语法树与抽象语法树的最根本区别在于它们的内部节点（包括根节点）：

- 语法树的内部节点是非终结符；
- 抽象语法树的内部节点是操作符（运算符）；
- 或者说抽象语法树中省略了反映分析过程的非终结符。

例3.6 句子-(id+id)和句型if C then s1 else s2 ：







语法树：左部非终结符“产生”右部文法符号序列；

抽象语法树：操作符（运算）“作用于”操作数（运算对象）；

习惯上：它们分别被称为具体语法树和抽象语法树。