



# 编译原理

2020年3月



上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

# 编译原理

## 第二章

上海交通大学

张冬莱

Email: [zhang-dm@cs.sjtu.edu.cn](mailto:zhang-dm@cs.sjtu.edu.cn)

2020年3月

## 第2章 一个简单的语法制导翻译器



本章的重点：介绍语法在编译器的构造过程中的理论和实际应用的总体技术，包括：

- 语法的定义方法
- 语法制导翻译与中间代码生成



## 2.1 引言



- 程序设计语言可以通过描述以下两个方面来定义：
  - 第一方面是程序模式，即语言的语法；
  - 第二方面是程序含义，即语言的语义。
- 描述语言的语法，我们可以使用形式化方法，例如：
  - 上下文无关文法或者BNF（Backus-Naur）范式
- 描述语言的语义，我们只能使用非形式化方法和启发性实例。
- 上下文无关文法除了可以用于定义语言的语法之外，还可用于指导源程序的翻译。面向语法的编译技术，如语法制导翻译技术，对于组织编译器的前端十分有用



在编译器里，词法分析器首先把输入字符流转换成记号流。然后，记号流作为下一个阶段的输入，产生源程序的中间表示，这个过程如图2-1所示。图中的“语法制导翻译器”由一个语法分析器和一个中间代码生成器构成。

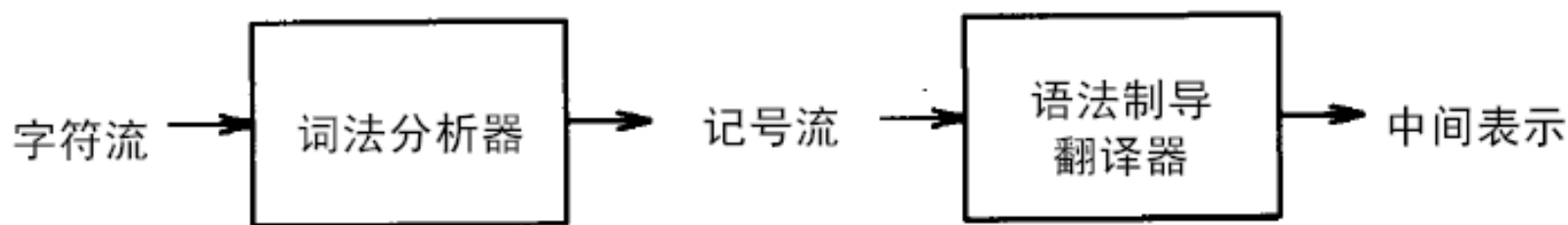


图2-1 我们的编译器前端的结构

## 2.2 语法定义



- 上下文无关文法（简称文法）是一种定义语言语法的表示法，上下文无关文法将贯穿本书始末，作为编译器要实现的语言定义部分。

例如：if-else语句的构造规则可以表达为：

$$stmt \rightarrow \text{if} ( expr ) stmt \text{ else } stmt \quad (2-1)$$


这里，箭头“->”可以读作“可以具有形式”。这样的规则称为产生式 (production)。在一个产生式中，像关键字if，else和括号这样的词法元素称为记号(token)或终结符(terminal)，像expr和stmt这样的变量表示一个记号序列，并称之为非终结符(nonterminal)。



## 2.2.1 文法定义



上下文无关文法包含如下四个部分：

- 一个记号集合，称为终结符号。
- 一个非终结符集合。
- 一个产生式集合。每个产生式具有一个左部和一个右部，左部和右部由箭头连接，左部是一个非终结符。右部是记号和（或）非终结符序列。
- 一个开始符号。 开始符号是一个指定的非终结符。

## 2.2.1 文法定义



我们约定:

定义语法时只需列出文法的产生式，并把以开始符号为左部的产生式列在最前面。在以后的讨论中我们假设：数字、类似于 $\leq$ 的符号、类似于while的黑体字符串均为终结符，斜体名字表示非终结符，任何非斜体的名字或者符号都是记号。为了表示上的方便，我们常把具有相同左部的产生式合并，写成一个产生式，其左部为所有产生式共有的那个非终结符，右部为所有产生式右部的组合，每个右部用“|”分隔。“|”读做“或”



## 例2.1



我们想要描述一个由数字、加号和减号组成的表达式，如9-5+2、3-1、7等。因为加号和减号必须出现在两个数字之间，所以我们称这样的表达式为“用加号和减号分隔的数字序列”。下面的文法描述了这些表达式的语法。产生式为：

$$list \rightarrow list + digit \quad (2-2)$$

$$list \rightarrow list - digit \quad (2-3)$$

$$list \rightarrow digit \quad (2-4)$$

$$digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \quad (2-5)$$

左部的非终结符皆为 *list*（列表）的三个产生式的右部可以合并成：

$$list \rightarrow list + digit | list - digit | digit$$



- 按照我们前边的约定，文法的记号是下列符号：

+ - 0 1 2 3 4 5 6 7 8 9

斜体词 *list* 和 *digit* 是非终结符，*list* 是开始非终结符，因为它所对应的产生式列在最前面。

- 如果一个非终结符出现在一个产生式的左部，该产生式称为该非终结符的产生式。记号串是零个或者多个记号的序列。一个包含零个记号的记号串称为空串，记为 $\epsilon$ 。

## 2.2.2 推导



- 根据文法推导符号串时，首先从开始符出发，不断将某个非终结符替换为该非终结符的某个产生式的右部，每替换一次，被称为一次**推导**。由文法的开始符经过**至少一步**推导的多步推导得到的所有终结符串的集合称为该文法所描述的**语言**。

## 例2.2



由例2.1的文法定义的语言是由加号和减号分隔的数字序列。非终结符 *digit* 的10个产生式表示 *digit* 可以代表0,1,...,9中的任何记号。

产生式(2-4)表明单个数字本身是一个列表。

产生式(2-2)和(2-3)表示：当我们碰到一个列表后面跟着一个加号或者减号，随后是另外一个数字，则我们得到了一个新的列表。

显然，产生式(2-2)到产生式(2-5)足以定义我们需要的语言。

例如：我们可以按下面的方法推导出9-5+2是一个 *list*：

- a) 由于9是一个 *digit*，根据产生式(2-4)，9是一个 *list*；
- b) 由于9是一个 *list*，5是一个 *digit*，根据产生式(2-3)，9-5是一个 *list*；
- c) 由于9-5是一个 *list*，2是一个 *digit*，根据产生式(2-2)，9-5+2是一个 *list*。

上述推导的过程可以表示为：

$$\begin{aligned} list &\Rightarrow list + digit \Rightarrow list - digit + digit \Rightarrow digit - digit + digit \\ &\Rightarrow 9 - digit + digit \Rightarrow 9 - 5 + digit \Rightarrow 9 - 5 + 2 \end{aligned}$$

## 例2.3



设标识符是字母开始的字母数字串，

如果：用L表示 <字母>，D表示<数字>，T表示<字母或数字>，  
则有：

$$L \rightarrow a \mid b \mid \cdots \mid z$$
$$D \rightarrow 0 \mid 1 \mid \cdots \mid 9$$
$$T \rightarrow L \mid D$$

如果：用S表示<字母数字串>，

则： <字母或数字>是一<字母数字串>； <字母数字串>后跟一个<字母或数字>，也是<字母数字串>，

所以有：  $S \rightarrow T \mid ST$

其中产生式  $S \rightarrow T \mid ST$  是一种左递归形式，由它可产生一串T。因此作为<标识符>的非终结符id，它或为单个字母，或为字母后跟字母数字串，  
故有：  $id \rightarrow L \mid LS$





我们想要描述的标识符文法就可以定义为：

$$id \rightarrow L \mid LS$$
$$S \rightarrow T \mid ST$$
$$T \rightarrow L \mid D$$
$$L \rightarrow a \mid b \mid \cdots \mid z$$
$$D \rightarrow 0 \mid 1 \mid \cdots \mid 9$$

## 例2.4



设param为C++语言的实际参数，id为函数的名字，我们想要描述C++语言的函数调用语句的文法就可以定义为：

$$call \rightarrow id (optparams)$$
$$optparams \rightarrow params \mid \epsilon$$
$$params \rightarrow params, param \mid param$$

## 2.2.3 语法分析树



- 分析树描绘了如何从文法的开始符号开始推导出它的语言中的一个语句。
- 形式地说，给定一个上下文无关文法，分析树是具有如下特性的树：
  - 树根标记为开始符号。
  - 每个叶节点由记号或者 $\epsilon$ 标记。
  - 每个内节点由一个非终结符标记。
  - 如果 $A$ 是某个内节点的非终结符标记， $X_1, X_2, \dots, X_n$ 是该节点从左到右排列的所有子节点的标记，则 $A \rightarrow X_1 X_2 \dots X_n$ 是一个产生式。这里， $X_1, X_2, \dots, X_n$ 是一个终结符或非终结符。对于 $A \rightarrow \epsilon$ ，分析树中标记为 $A$ 的节点只有一个标记为 $\epsilon$ 的子节点。



## 2.2.3 语法分析树



- 一棵分析树从左到右的叶节点是这棵分析树生成的结果。分析树生成的结果是由根节点的非终结符生成或导出的串。
- 任何树的叶节点都满足从左到右排列的自然顺序，即如果 $a$ 和 $b$ 具有相同的父节点，且 $a$ 在 $b$ 的左部，则 $a$ 和 $a$ 的所有后代都在 $b$ 和 $b$ 的所有后代的左部。
- 使用分析树的概念，我们可以定义：一个文法生成的语言是它的某个分析树生成的串的集合。为给定的记号串找到一个分析树的过程称为这个串的语法分析（parsing）。

## 例2.5



由例2.2推理过程如图2-2中的树所示。树中的每个节点用一个文法符号标记。一个内节点和它的所有子节点对应一个产生式。内节点对应产生式的左部，子节点对应产生式的右部。这样的树称作分析树。

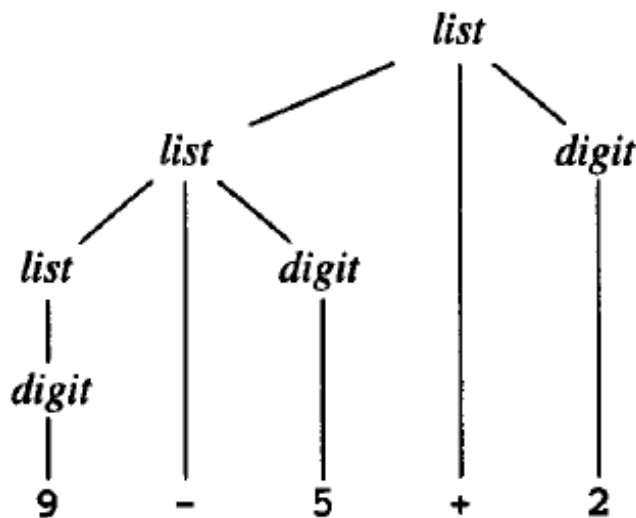


图2-2 与例2.1中文法对应的9-5+2 的分析树



## 2.2.4 二义性



- 一棵分析树读完它的叶节点只能生成惟一的一个串，但是，一个文法可能有多棵分析树生成相同的记号串。这样的文法称为具有二义性的文法。
- 判断一个文法是否具有二义性，我们只需检查是否存在一个具有多棵分析树的记号串。



## 例2.6



如果不区分例2.1中的 *digit* 和 *list*, 例2.1中的文法可以改写成如下形式:

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

把 *digit* 和 *list* 的概念合成一个非终结符串 *string* 是有意义的, 因为一个 *digit* 是 *list* 的特例。



- 从图2-3中可以看到，表达式 $9-5+2$ 现在有了不止一棵分析树，分别对应着不同的带括号表达式 $(9-5)+2$ 和 $9-(5+2)$ 。这两个表达式的值是不同的，分别是6和2。例2.1的文法不允许这样的解释。

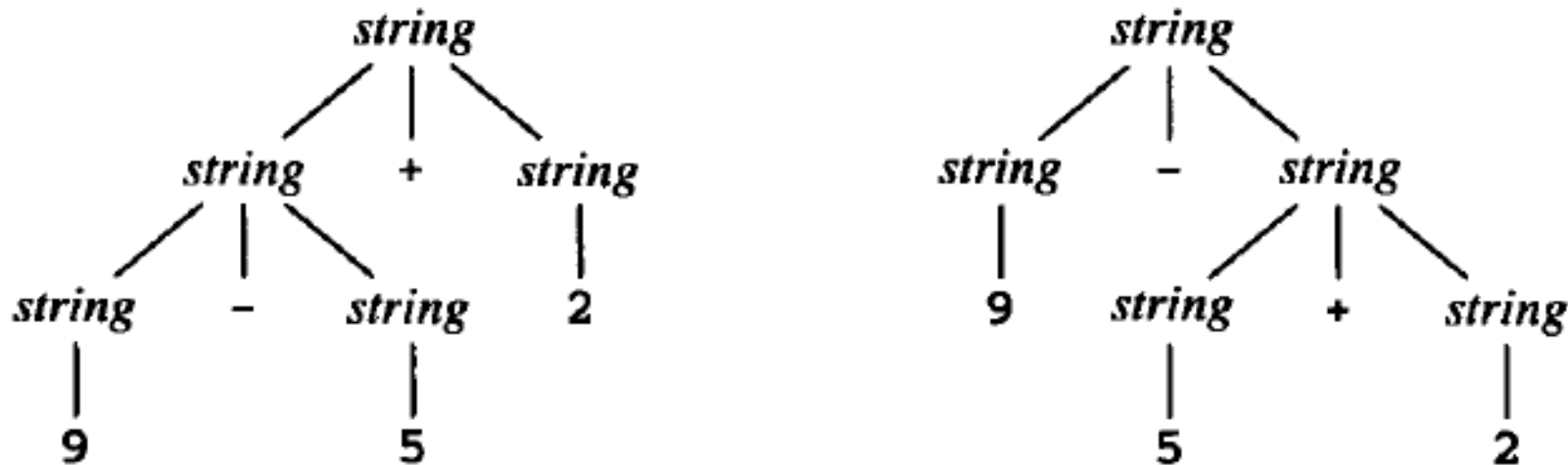


图2-3  $9-5+2$  的两棵分析树

## 2.2.5 运算符的结合性



- 当同一种操作符连续出现的时候，我们需要确定操作的优先顺序，这就是该操作符的结合性。
- 在大多数的程序设计语言中，加、减、乘、除四种算术操作符都是左结合的。
- 某些常用操作符是右结合的，如C语言中的赋值运算操作符“=”号是右结合的。
- 注意：这些操作符的结合性实际上都要通过语法的定义才能实现。

## 2.2.6 运算符的优先级



- 当不止一种操作符出现的时候，我们需要确定操作符之间的优先顺序，即运算符的优先级。
- 如：普通的算术运算中，乘法和除法比加法和减法具有较高的优先级。
- 注意：这些操作符的优先级实际上也都要通过语法的定义才能实现。



## 例2.7



- 算术表达式的文法可以根据操作符的结合性和优先级表来构建。
- 考虑下面的算术表达式文法：

1:  $E \rightarrow E + T \mid E - T \mid T$

2:  $T \rightarrow T * F \mid T / F \mid F$

3:  $F \rightarrow (E) \mid id$

- 该文法中，确定操作符：

优先级递增的次序排列： + - 低于 \* / 低于 ( )

相同优先级的出现在同一行上 + - 相同， \* / 相同

左结合： + - \* /

## 2.3 语法制导翻译



- 为了翻译程序设计语言的某个结构，除了为该结构生成的代码以外，编译器还需要保存许多信息。例如，编译器可能要知道这个结构的类型、目标代码中第 1 条指令的位置、生成的指令个数等等。我们抽象地称这些信息为与该结构相关的属性。
- 本节给出一种称为语法制导定义的形式化方法，用以说明程序设计语言中各种结构的翻译。一个语法制导定义根据与其语义部分相关联的属性说明了程序设计语言的一个结构的翻译。
- 我们还要介绍一个更加过程化的概念，叫做翻译模式，用来描述翻译过程。本章我们将翻译模式用于把中缀表达式翻译成后缀表达式。



## 2.3.1 后缀表示



- 一个表达式 $E$ 的后缀表示可以归纳地定义如下：
  - 如果 $E$ 是一个变量或者常量，则 $E$ 的后缀表示是 $E$ 本身。
  - 如果 $E$ 是形如 $E1 \ op \ E2$ 的表达式，其中 $op$ 是一个二元操作符，则 $E$ 的后缀表示是 $E1 \ E2 \ op$ ，这里 $E1$ 和 $E2$ 分别是 $E1$ 和 $E2$ 的后缀表示。
  - 如果 $E$ 是形如 $(E1)$ 的表达式，则 $E1$ 的后缀表示是 $E$ 的后缀表示。
- 例如， $(9-5)+2$ 的后缀表示是 $95-2+$ ， $9-(5+2)$ 的后缀表示是 $952+-$ 。

## 2.3.2 综合属性



- 如果分析树的某个节点的属性值是由其子节点的属性值确定的，则我们称该属性为综合属性。一棵分析树的所有综合属性值的计算只需要分析树的一次自底向上遍历。
- 例2.6 把一个由加号和减号分隔的数字序列组成的表达式翻译成后缀表示的语法制导定义如图2-5所示。图中与每个非终结符相关联的是一个具有字符串值的属性 $t$ ，属性 $t$ 表示该非终结符产生的表达式的后缀表示。



- 一个数字的后缀形式是该数字本身。例如，与产生式  $term \rightarrow 9$  相关联的语义规则定义：当该产生式在分析树的节点上被使用时， $term.t$  的值是9。当产生式  $expr \rightarrow term$  被应用时， $term.t$  的值成为  $expr.t$  的值。

产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
...	...
$term \rightarrow 9$	$term.t := '9'$

图2-5 中缀到前缀翻译的语法制导定义



- 产生式  $expr \rightarrow expr1 + term$  导出一个包含加号的表达式。加操作符的左操作数由  $expr1$  给出，右操作数由  $term$  给出。产生式中  $expr1$  的下标是为了区别产生式左右两侧的  $expr$ 。与这个产生式关联的语义规则

$expr.t := expr1.t || term.t || '+'$

定义了属性  $expr.t$  值的计算方式，即首先连接左右操作数的后缀形式  $expr1.t$  和  $term.t$ ，然后连接上加号。语义规则中的操作符  $||$  表示字符串的连接。

- 图2-6给出了对应于图2-2中分析树的注释分析树。每个节点上的 $t$ 属性的值用与该节点的产生式相关联的语义规则计算。根节点的属性值是该分析树生成的串的后缀表示。

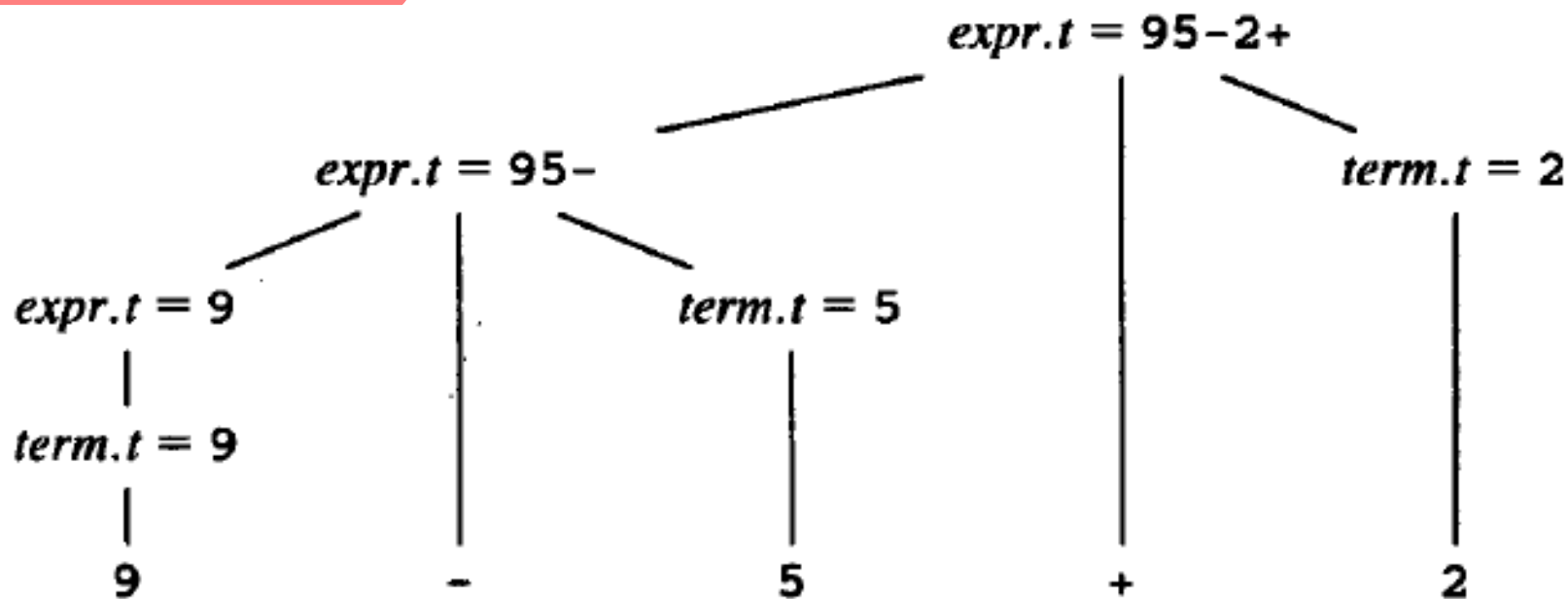


图2-6 分析树各节点的属性值

## 2.3.3 简单语法制导翻译定义



- 语法制导定义使用上下文无关文法来说明输入的语法结构。它通过每个文法符号和一个属性集合相关联，通过每一个产生式和一个语义规则集合相关联。语义规则用来计算与产生式中出现的符号相关联的属性的值。文法和语义规则集合构成了语法制导定义。
- 翻译是一个输入到输出的映射。每个输入 $x$ 的输出用下面的方式来说明。首先，构建 $x$ 的分析树。假定分析树的节点 $n$ 用文法符号 $X$ 标识。我们用 $X.a$ 表示节点 $n$ 上 $X$ 的属性 $a$ 的值。节点 $n$ 上的 $X.a$ 的值是使用与 $X$ 产生式相关联的属性 $a$ 的语义规则来计算的。每个节点都具有属性值的分析树称为注释分析树。





- 语法制导定义具有如下特性：每个产生式左部的非终结符的翻译是将该产生式右部的非终结符的翻译按照它们在右部出现的次序连接起来得到的，在连接过程中可能还需要附加（也可能不需要）一些额外的串。具有这样特性的语法制导定义称之为简单的语法制导定义。

## 2.3.4 树的遍历



- 树的遍历是指从根开始，以某种顺序访问树的每一个节点。本章使用图2-10定义的深度优先遍历计算语义规则。它从根开始，从左到右递归访问每个节点的子节点，如图2-11所示。一旦给定节点的所有后代都被访问，则该节点的语义规则将被计算。之所以称之为“深度优先”遍历，是因为它尽可能地访问一个节点的未访问的子节点，于是它尽可能快地访问离根最远的节点。这可以快速获取语法树节点上的综合属性。

```
procedure visit(n: node);  
begin  
  for n 的每个子节点 m, 从左到右 do  
    visit(m);  
  计算节点 n 处的语义规则  
end
```

图2-10 树的深度优先遍历

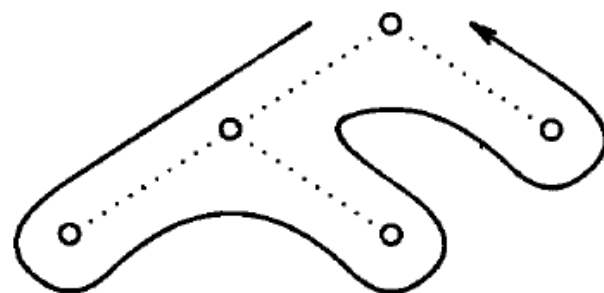


图2-11 树的深度优先遍历示例

## 2.3.5 翻译方案



- 一个翻译模式是一个上下文无关文法，其中被称为语义动作的程序段被嵌入到产生式右部。一个翻译模式类似于语法翻译制导定义，只是语义规则的计算顺序是显式给出的。一个语义动作的执行位置通过用括号把语义动作括起来并将其放在产生式右部来表示，如  $rest \rightarrow + term$   
 $\{print( '+' )\} rest1$
- 翻译模式对于由基本语法产生的每个语句  $x$  都产生一个输出，方法是：  
按照  $x$  的分析树的深度优先遍历顺序执行语义动作。

## 2.3.6 翻译的输出



- 翻译模式的语义动作把翻译的输出以一次一个字符或一个字符串的形式写入一个文件。
- 语法制导定义具有如下特性：每个产生式左部的非终结符的翻译是将该产生式右部的非终结符的翻译按照它们在右部出现的次序连接起来得到的，在连接过程中可能还需要附加（也可能不需要）一些额外的串。具有这样特性的语法制导定义称之为简单的语法制导定义。

## 例2.8



- 图2-5是把表达式翻译成后缀形式的一个简单的语法制导定义。由该定义得到的翻译模式如图2-13所示，带语义动作的 $9-5+2$ 的分析树如图2-14所示。注意，尽管图2-6和图2-14描述的是相同的输入-输出映射，但两者构造翻译的过程是不相同的。图2-6是把输出附加在分析树的根，而图2-14输出递增地显示出来。
- 图2-14的根表示图2-13中的第一个产生式。在深度优先遍历中，当我们遍历根节点的最左子树时，先执行左操作数 $expr$ 的子树中的所有语义动作。然后，我们访问没有语义动作的叶节点 $+$ 。接下来，我们执行右操作数 $term$ 的子树中的所有语义动作。最后，执行特殊节点上的语义动作 $\{print( '+' )\}$ 。
- 由于 $term$ 产生式右部只有一个数字，语义动作只显示该数字。产生式 $expr \rightarrow term$ 不产生输出，头两个产生式的语义动作只须显示操作符。在深度优先遍历分析树的过程中执行图2-14中的语义动作显示 $95-2+$ 。

<i>expr</i>	$\rightarrow$	<i>expr</i> + <i>term</i>	{ <i>print</i> ('+') }
<i>expr</i>	$\rightarrow$	<i>expr</i> - <i>term</i>	{ <i>print</i> (' - ') }
<i>expr</i>	$\rightarrow$	<i>term</i>	
<i>term</i>	$\rightarrow$	0	{ <i>print</i> ('0') }
<i>term</i>	$\rightarrow$	1	{ <i>print</i> ('1') }
		...	
<i>term</i>	$\rightarrow$	9	{ <i>print</i> ('9') }

图2-13 把表达式翻译成后缀形式的语义动作

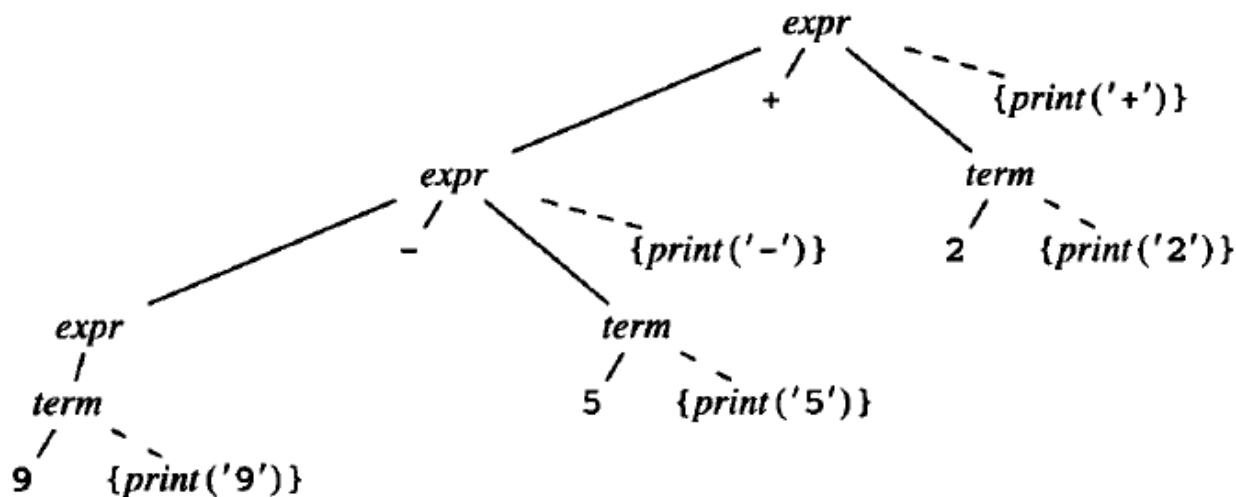


图2-14 把9-5+2 翻译成95-2+ 的语义动作



- 作为一般规则，多数分析方法都以一种“贪心”的方式从左到右地处理输入，即在读入下一个记号之前构造尽可能多的分析树的组成部分。在简单翻译模式（由简单语义制导定义得到的）中，语义动作也是按照从左到右的顺序执行的。因此，实现简单翻译模式时，我们可以在语法分析的时候执行语义动作，完全没有必要构造分析树。



## 第二章 作业



- 1. 写出不以0开头的奇数的上下文无关文法。
- 2. 设param为C++语言的实际参数，小写字母a…z和数字0…9可用的符号，参考例2.3和例2.4写出C++函数调用的完整的上下文无关文法。





## 第二章 总结



主要内容：

- 程序语言的语法定义方法；
- 语法制导翻译与中间代码生成的基本方法；

课后要求：

- 完成课后作业；
- 简单阅读教材P37-66内容，该部分的具体内容将在第三，四，五，六章进行详细讲解

# 谢谢!



上海交通大学

SHANGHAI JIAO TONG UNIVERSITY

上海交通大学

