# 第 1 章　编译器概述

从理论上说，构造专用计算机来直接执行某种高级语言写的程序是可能的。但是，实际上目前的计算机能执行的都是非常低级的机器语言。那么，一个基本的问题是：高级语言的程序最终是怎样在计算机上执行的。

能够完成从一种语言到另一种语言变换的软件称为翻译器，这两种语言分别叫做该翻译器的源语言和目标语言。编译器是一种翻译器，它的特点是目标语言比源语言低级。

本章通过描述编译器的各个组成部分来介绍编译这个课题。该课题涉及程序设计语言、机器结构、形式语言理论、类型论、算法和软件工程等方面的知识。

编译器的工作可以分成若干阶段，每个阶段把源程序从一种表示变换成另一种表示。编译过程的一种典型分解见图 1.1，图中的每个方框表示它的一个阶段。

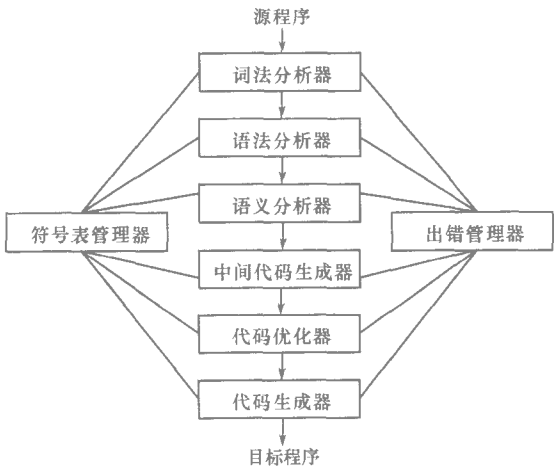


图 1.1　编译的阶段

本节以 Pascal 语言的赋值语句

$$\text{position} := \text{initial} + \text{rate} * 60 \tag{1.1}$$

的翻译（假定变量都是实型）为例，概要介绍编译的各个阶段。

## 1.1 词法分析

词法分析逐个读构成源程序的字符，把它们组成词法记号（*token*）流。赋值语句(1.1)的字符流在词法分析时被组成下面的词法记号流：

(1) 标识符 (position)
(2) 赋值号 (:=)
(3) 标识符 (initial)
(4)加 号( + )
(5) 标识符 (rate)
(6)乘 号( * )
(7) 数(60)

分隔记号的空格通常在词法分析时被删去。

词法单元 position、initial 和 rate 属于同样的记号，因此需要为某些记号增加一个属性来区分属于同一记号的不同词法单元。例如，发现 rate 这样的标识符时，词法分析器不仅产生一个记号,如 id,还把当前词法单元 rate 填入符号表，如果表中还没有它的话。 id 这次出现的属性是符号表中 rate 条目的指针。

用 $\mathbf{id}_1$、$\mathbf{id}_2$ 和 $\mathbf{id}_3$ 分别表示 position、initial 和 rate ，以强调标识符的内部表示是有别于形成标识符的字符序列的。于是，语句（1.1）在词法分析后的表示是

$$\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60 \qquad (1.2)$$

多字符算符 := 和数 60 也应该有它们的内部表示，本书将在第 2 章词法分析中讨论，为直观起见，这里直接用它们在源程序中的字符序列。

编译器的词法分析也叫做线性分析或扫描。

## 1.2 语法分析

语法分析（syntax analysis ）简称为分析(parsing)，它把词法记号流依照语言的语法结构按层次分组，以形成语法短语。因此语法分析也称为层次分析。源程序语法短语常用分析树表示,图 1.2 便是一例。

在表达式 initial + rate * 60 中,短语 rate * 60 是一个逻辑单位，因为按算术表达式的一般习惯,乘比加先完成 由于 initial + rate 后面是乘号,所以 initial + rate 不能组成一个短语。

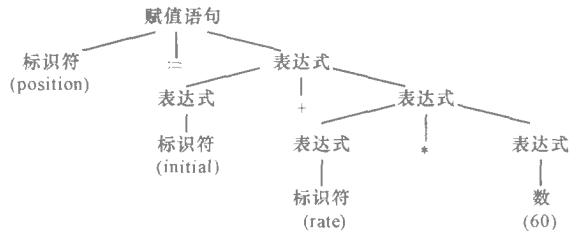


图 1.2 position := initial + rate * 60 的分析树

程序的层次结构通常由递归的规则表示，例如，可以用如下规则作为表达式定义的一部分：

(1) 任何一个标识符都是表达式；

(2) 任何一个数都是表达式；

(3) 如果 $e_1$ 和 $e_2$ 都是表达式，那么

$e_1 + e_2$

$e_1 * e_2$

$(e_1)$

也都是表达式。

规则(1)和(2)都是（非递归的）基本规则，而规则(3)是通过把算符作用于表达式来定义更复杂的表达式。这样，由规则(1)，initial 和 rate 都是表达式；由(2)，60 是表达式；由(3)，首先可推出 rate * 60 是表达式，然后 initial + rate * 60 也是表达式。

同样地，许多语言用类似如下的规则递归地定义语句：

(1) 如果 identifier 是标识符，expression 是表达式，那么

identifier := expression

是语句。

(2) 如果 expression 是表达式，statement 是语句，那么

**while** ( expression ) **do** statement

**if** ( expression ) **then** statement

也都是语句。

图 1.2 的分析树描绘了 position := initial + rate * 60 的语法结构，这种语法结构更常见的内部表示由图 1.3(a)的语法树给出。语法树是分析树的浓缩表示，其中内部结点都是算符，内部结点的后代是它的运算对象。图 1.3(b)这样的树结构将在第 4 章 4.2 节讨论。在第 4 章的语法制导翻译中，将详细讨论编译器如何利用输入所含的层次结构来产生语法树 。

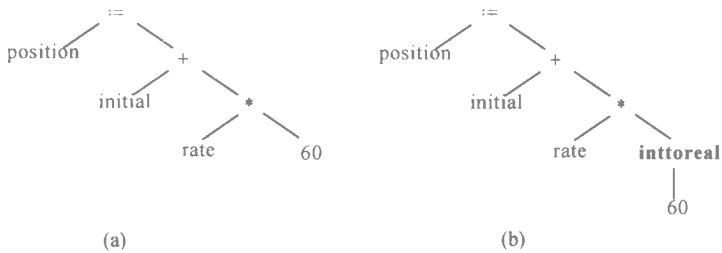


图 1.3 语义分析插入了类型转换

## 1.3 语义分析

语义分析阶段检查程序的语义正确性，以保证程序各部分能有意义地结合在一起，并为以后的代码生成阶段收集类型信息。

语义分析的一个重要部分是类型检查，编译器检查每个算符的运算对象，看它们的类型是否适当。例如，当实数作为数组的下标时，许多语言的规定是要求编译器报告错误；当然也有些语言允许运算对象的类型隐式转换，例如二元的算术算符作用于一个整数和一个实数的时候。类型检查和语义分析在第 5 章讨论。

例 1.1 在机器内部，整数的二进制表示和实数的二进制表示是有区别的，不论它们是否有相同的值。在图 1.3 中，所有的标识符都声明为实型，另外，由 60 本身可知它是整数。对图 1.3(a)进行类型检查会发现 * 作用于实型变量 rate 和整数 60，通常的办法是把整数转变为实数。可以建立一个额外的算符结点 inttoreal(见图 1.3(b))，它显式地把整数转变为实数。

编译器的前三个阶段对源程序分别进行不同的分析，以揭示源程序的基本数据和结构，决定它们的含义，建立源程序的中间表示。许多处理源程序的软件工具都要完成某类分析，下面给出几个例子。

(1) 格式打印程序　它以源程序为输入，以程序结构清晰可见的方式输出源程序。例如：注释可以以特殊的字体或颜色出现，语句可以按它们嵌套的层次阶梯式地显示出来。显然，格式打印程序需要对源程序进行词法分析和语法分析，但不需要对源程序进行语义分析，因为一种书写格式只和语法有关。

(2) 文档抽取程序　它抽取程序文件中的注释和函数首部等信息，形成一份程序文档。它需要对程序进行词法分析和语法分析，至少是部分的语法分析。要使得这样抽取的信息

形成一份有用的文档，注释很有讲究，否则抽取出来的文档没有什么用处。

(3 静态检查程序　静态检查程序读入程序，分析它，并试图不运行程序而发现一些潜在的错误。它的分析部分和第 9 章优化编译器的分析部分类似。例如，它可以检查出源程序的某些部分决不会执行，或者某个变量在赋值前可能被引用。此外，使用比第五章更精致的类型检查和类型推导技术，它还能捕捉程序的安全隐患，例如通过指针使用已经释放了的存储空间。

(4 解释器　纯解释器在执行源程序语句时，都需分析构成该语句的字符串，以便识别和执行它指定的计算。如果给定的语句仅执行一次，纯解释的方法是所有方法中代价最小的一种，例如，它常用于交互语言的“立即命令”。如果语句重复执行，较好途径是分析源程序的字符流仅一次，用一串更适于解释的符号序列或其他的形式来代替它。因此解释器往往也做某种程度的翻译，例如，对于赋值语句，解释器可能建立像图 1.3(a) 那样的树，在遍历树时执行结点的操作。在树根，它发现必须完成赋值，因此调用子例程来计算右部表达式的值，然后把值存到为标识符 position 分配的存储单元。计算树根右子树的子例程发现必须计算两个表达式的和，它递归地调用自己来计算表达式 rate * 60 的值，然后再加上变量 initial 的值。

## 1.4　中间代码生成

语法分析和语义分析后，某些编译器产生源程序的显式中间表示，可以认为这种中间表示是一种抽象机的程序。中间表示必须具有两个性质：它易于产生并且易于翻译成目标程序。

中间表示有各种形式。在第 7 章，我们把中间表示看成三地址代码，它们像机器的汇编语言，这种机器的每个存储单元的作用类似于寄存器。三地址代码由三地址语句序列组成，每个三地址语句最多有三个操作数，语句 (1.1)的三地址代码可以如下：

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2                    (1.3)
id1 := temp3
```

这种中间形式有它的特点。首先，除了赋值算符外，每个语句至多只有一个算符，因此，在生成这些语句时，编译器必须决定运算完成的次序，语句 (1.1)的乘优先于加。其次，编译器必须产生临时变量名，用以保留每个语句的计算结果。第三，某些三地址语句没有三个运算对象，例如语句(1.3)的第一个和最后一个语句。

本书在第 7 章叙述编译器时用的主要是中间表示。通常，除了计算表达式外，这些中间

表示还必须做其他事情，它们必须处理控制流结构和过程调用。第 4 章和第 7 章提供为程序设计语言典型结构产生中间代码的一些算法。

## 1.5 代码优化

代码优化阶段试图改进代码，以产生执行较快的机器代码。如果中间代码生成算法比较简单的话，它给代码优化留了很多机会。例如，产生中间代码的一个比较自然的算法是为语义分析后的树上的每个算符产生一条指令，因而得到语句（1.3)的中间代码。然而，还存在更好的算法，例如使用两条指令

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```
(1.4)

也可以完成同样的计算。用简单的中间代码生成算法是可以的，因为产生较优代码这个问题可以在代码优化阶段得以解决，也就是，优化器会推断出，60 从整型表示变为实型表示可以在编译时完成，从而 inttoreal 操作可以删去。此外， temp3 只使用一次，即把它的值传给 id1，所以用 id1 代替 temp3 是妥当的，从而(1.3)的最后一个语句不必存在，这样就得到(1.4)的结果。

不同的编译器完成不同程度的优化，能完成大多数优化的编译器叫做“优化编译器”，但是编译的相当大一部分时间都消耗在这种优化上。简单的优化也可以使目标程序的运行时间大大缩短，而编译速度并没有降低太多。第 9 章将讨论优化问题。

## 1.6 代码生成

编译的最后一个阶段是目标代码生成，它生成可重定位的机器代码或汇编码。此阶段为源程序所用的每个变量选择存储单元，并且把中间代码翻译成等价的机器指令序列。此阶段的一个关键问题是寄存器分配。

例如，使用寄存器 R1 和 R2，(1.4)的代码可以翻译成：

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```
(1.5)

每条指令的第一个和第二个操作数分别代表源和目的操作数。每条指令的 F 告诉我们指令处理浮点数。这段代码把地址 id3 的内容取入寄存器 R2(我们暂且认为指令中 id3 代表对象 id3 的地址。变量的存储分配在第 章讨论），然后把它乘上实数 #号代表 作为常数处理。第三条指令把 id2 取入寄存器 R1,再把原先寄存器 R2 的值加上去，最后寄存器 R1 的值存入地址 id1。这样，该段代码便实现了图 1.4 的赋值。代码生成在第 8 章讨论。

中间代码生成、代码优化和代码生成三个阶段合称为对源程序进行综合，它们从源程序的中间表示建立起和源程序等价的目标程序。

## 1.7 符号表管理

符号表管理和出错管理是编译过程中的两项重要工作，它们与词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成这六个阶段相互作用。下面我们简单介绍这两项工作。

编译器的一项重要工作是记录源程序中使用的标识符，并收集每个标识符的各种属性。这些属性提供标识符的存储分配、类型和作用域信息。如果是过程标识符，还有参数的个数和类型，参数传递方式和返回值类型（如果有的话）。

符号表是为每个标识符保存一个记录的数据结构，记录的域是标识符的属性。该数据结构允许我们迅速地找到一个标识符的记录，在此记录中存储和读取数据。符号表在第 7 章讨论。

词法分析器发现源程序的标识符时，把该标识符填入符号表。但是，一般来说。词法分析期间不能确定一个标识符的属性。例如，像

```
var position, initial, rate : real ;
```

这样的 Pascal 声明，词法分析器读过 position、initial 和 rate 时，它们的类型还是未知的。

其余的阶段把标识符的信息填入符号表，然后以不同的方式使用这些信息。例如，语义分析和中间代码生成需要知道标识符的类型，才能检查源程序是否以合法的方式使用它们，才能为它们产生适当的操作。代码生成需要使用标识符存储分配信息以产生正确的指令。

## 1.8 错误诊断和报告

每个阶段都有可能发现源程序的错误。在发现错误后，该阶段必须处理此错误，使得编译可以继续进行，以便进一步发现源程序的其他错误。发现一个错误便停下来的编译器是

没有尽到责任的，除非它是集成在一个好的编程环境中，使得编译和编辑之间的切换相当方便和迅速。

语法分析和语义分析通常处理编译器能发现的绝大部分错误。词法分析阶段能发现的错误类型是，当前被扫描的字符串不能形成语言的词法记号。记号流违反语言的语法规则是由语法分析阶段诊断。语义分析时，编译器试图找出语法正确但对所含的操作来说是无意义的结构，如相加的两个标识符，其一是数组名，另一个是过程名。本书把每个阶段对错误的处理放在与该阶段有关的章节介绍。

语句(1.1)的编译过程总结在图 1.4 中。

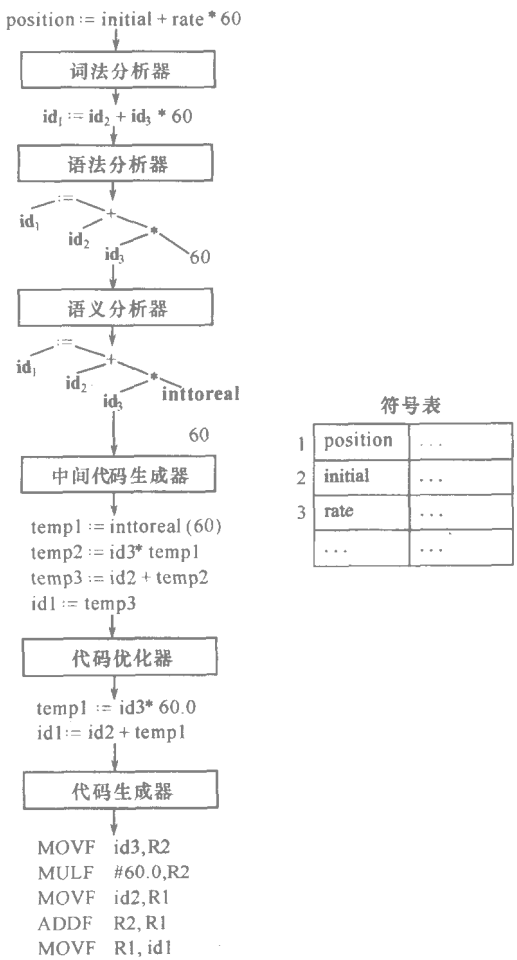


图 1.4 一个语句的翻译

## 1.9 阶段的分组

在实际的编译器中，若干阶段可以组合在一起，各阶段之间的中间表示也无需显式构造。

通常，所有的阶段被分成前端和后端两部分。前端只依赖于源语言，由几乎独立于目标机器的阶段或阶段的一部分组成。通常，它们是词法和语法分析、符号表建立、语义分析和中间代码生成，有些优化也可以在前端完成。前端还包括与这些阶段同时完成的错误处理。

后端是指编译器中依赖于目标机器的部分，它们一般独立于源语言，而与中间语言有关。后端包括代码优化、代码生成和伴随这些阶段的错误处理和符号表操作。

取一个编译器前端，重写它的后端以产生同一源语言在另一机器上的编译器已经是件普通的事情。如果这些后端是很仔细设计的，甚至不需要对它做很多的重新设计。

把几种不同的语言编译成同一种中间语言，让不同的前端使用同一后端，从而得到一台机器上的几个编译器，也是很吸引人的事。但是，由于不同语言的着眼点有区别，在这个方向上只取得了有限的成功。

编译的几个阶段常用一遍（pass）扫描来实现，一遍扫描包括读一个输入文件和写一个输出文件。在工程实践中，有很多不同的方式把编译器的阶段组成遍，因此我们更愿意围绕着阶段而不是围绕遍来组织对编译的讨论。

把几个阶段组成一遍，并且这些阶段的活动在该遍中交错进行是屡见不鲜的。例如，词法分析、语法分析、语义分析和中间代码生成可以组成一遍。如果这样，第一遍扫描可以直接从字符流翻译成中间代码。更具体一些，可以把语法分析器看成是“主导”的，它试图在所看见的记号流上找出语法结构。当它需要记号时，调用词法分析器取下一个记号。如果已看出一个语法结构，语法分析器则激活中间代码生成器，以完成语义分析和生成中间代码。

## 习 题 1

1.1 解释下列名词：

源语言　　目标语言　翻译器　编译器　解释器

1.2 典型的编译器可以划分成哪几个主要的逻辑阶段，各阶段的主要功能是什么？

# 第2章　词法分析

词法分析器的任务是把构成源程序的字符流翻译成词法记号流。构造词法分析器的一种简单办法是用状态转换图来描述源语言词法记号的结构，然后手工把这种状态转换图翻译成为识别词法记号的程序。用这种方式可以产生高效的词法分析器。

本章重点围绕词法分析器的自动生成展开，先介绍与之有关的正规式和有限自动机概念，以及词法分析器的自动生成方法，最后介绍一个词法分析器自动生成工具 Lex。

## 2.1　词法记号及属性

词法分析是编译的第一阶段，它的主要任务是读输入字符流，产生用于语法分析的词法记号序列。在第 1 章中提到，编译器一些阶段的活动会交错进行，图 2.1 给出了词法分析器和语法分析器的一种典型关系，即把词法分析器作为语法分析器的一个子程序来实现。当收到来自语法分析器的“取下一个记号”命令时，词法分析器读输入字符直到它能够确认下一个词法记号为止。

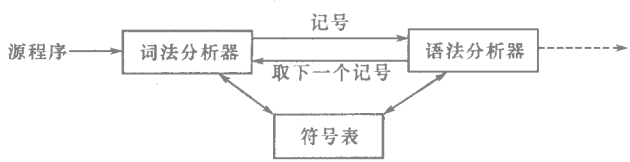


图 2.1　词法分析器和语法分析器的相互作用

词法分析器是编译器中读源程序的部分，因而它还可以完成和用户接口的一些其他任务。其一是剥去源程序的注解和（由空格、制表或换行符等引起的）空白。另一任务是把来自编译器各个阶段的错误信息和源程序联系起来，例如，词法分析器记住当前处理的字符行的行号，从而源程序行号可以和错误信息联系到一起。在某些编译器中，复制源程序并把错误信息嵌在其中是词法分析器的责任。通常在词法记号流中已经没有行的概念，因此这样的事情一般由词法分析器来完成。如果源语言支持对宏定义的预处理，该功能可以在词法分析的时候实现。

### 2.1.1 词法记号、模式、词法单元

在谈论词法分析时,使用术语‘词法记号’(简称记号)‘模式’和‘词法单元’表示特定的含义。表 2.1 是使用它们的例子。一般来说，在输入的字符流中有很多字符串，它们的记号是一样的。这样的字符串集合由叫做模式的规则来描述，模式匹配对应集合的任一字符串。模式的形式描述将在下一节讨论。词法单元（lexeme），又称单词，是源程序的字符串，它由模式匹配为记号。例如， Pascal 语句

```
var count : integer;
```

的子串 count 是记号“标识符”的一个词法单元。

本书用黑体字来表示记号，它们将作为源语言语法的终结符。

在大多数程序设计语言中，下列结构处理为记号：关键字、算符、标识符、常数、文字串（字符串）和标点符号。在上面的例子中，当字符串 count 出现在源程序中时，词法分析器将记号 id 返回给语法分析器。记号的返回由传递一个代表这个记号的整数来实现。

表 2.1 记号的例子

| 词法记号 | 词法单元举例 | 模式的非形式描述 |
|---|---|---|
| **var** | var | var |
| **for** | for | for |
| **relation** | <, <=, =, <>, >, >= | <或 <= 或 = 或 <> 或 > 或 >= |
| **id** | sum, count, D5 | 以字母开头的字母数字串 |
| **num** | 3.1416, 10, 2.08 E12 | 任何数值常数 |
| **literal** | "segmentation error" | 引号"和"之间的任意字符串,但引号本身除外 |

一个模式是描述源程序中某个记号的词法单元集合的规则。表 2.1 中记号 **var** 的模式是字符串 var，它是这个关键字的拼写。记号 **relation** 的模式是 Pascal 的 6 个关系算符的集合。为了更精确地描述更复杂的记号（如 **id** 和 **num**）,我们将使用 2.2 节介绍的正规式。

某些语言的一些规定给词法分析带来了困难。例如，对待空格，不同的语言有不同的规定。在早先的一些语言（如 FORTRAN 和 Algol 68)中,空格无意义(文字串中的除外),而不是作为词法单元的分隔符，它可以随便加入，以改变程序的可读性。空格的这种约定增加了处理标识符记号的复杂性，典型例子是 FORTRAN 的 DO 语句,在语句

```
DO 8 I = 3.75
```

中，词法分析器只有看见了小数点后，才能确定 DO 不是关键字，而 DO8I 是标识符。但是在表面上类似的语句

```
DO 8 I = 3.75
```

中，DO 是关键字，该语句共有 7 个记号：关键字 DO，语句标号 8，标识符 I，算符(，常数 3，逗号和常数 75。在没有看见逗号之前，词法分析器不敢保证 DO 是关键字。空格的这种规定给词法分析器带来的困难是，需要向前阅读多个字符，才能回过头来确定一个记号。

另一个例子是，关键字是否保留。保留字是语言预先确定了含义的词法单元，程序员不可以对这样的词法单元重新声明它的含义，如 Pascal 语言的 var 和 begin 等称为保留字。

很多语言使用关键字概念，并且它们是保留的，因此和上面的保留字没有区别，如 C 语言和 Java 语言。但是 FORTRAN 语言的关键字不保留，如 IF，当它作为语句的第一个词法单元时，很可能是关键字，因为这是该关键字出现的地方，但也不排除它是一个程序显式声明的标识符。若 IF 出现在语句的其他地方，它一定是程序显式或隐式声明的标识符。这就给词法分析带来很大困难，因为识别一个记号和该记号所处的上下文有关了。

顺便需要区分的是标准标识符概念，标准标识符也是预先确定了含义的标识符，程序也可以重新声明它的含义。在这个声明的作用域内，程序声明的含义起作用，而预先确定的含义消失，在其他地方都是预先确定的含义起作用，如 Pascal 语言的 integer 和 true 等。词法分析器对标准标识符没有什么特别的处理，由符号表管理来解决这件事。

### 2.1.2 词法记号的属性

从上一小节知道，Pascal 的 6 个关系算符都属于记号 **relation**。因为从程序的语法是否正确的角度看，使用哪个关系算符都一样。但是从翻译成目标代码来考虑，不同的关系算符，其翻译结果是不一样的。因此词法分析器需要给记号以属性，用属性来记住记号的附加信息，以便需要时使用它们。概括地说，记号影响语法分析的决策，属性影响记号的翻译。

例 2.1 Pascal 语句

```
position := initial + rate * 60
```

的记号和它们的属性值用二元组序列表示如下：

⟨**id**，指向符号表中 position 条目的指针⟩
⟨**assign _ op**，⟩
⟨**id**，指向符号表中 initial 条目的指针⟩
⟨**add _ op**，+⟩
⟨**id**，指向符号表中 rate 条目的指针⟩
⟨**mul _ op**，*⟩
⟨**num**，整数值 60⟩

注意，某些二元组没有属性值，它的第一个成分足以辨别词法单元，例如 **assign _ op**。因为 +，- 和 or 都可归入 **add _ op**，因此 **add _ op** 在此需要第二元。在这个例子中，记号

num 给了一个整数值属性。编译器也可以把形成数的字符串存入数表，让记号 num 的属性值是指向这个条目的指针。□

### 2.1.3 词法错误

词法分析几乎发现不了源程序的错误，因为词法分析器对源程序采取非常局部的观点。像 C 语言的语句

fi( a == f(x) ) …

中,词法分析器把 fi 当作一个普通的标识符交给编译的后续阶段，而不会把它看成是关键字 if 的拼写错。

Pascal 语言要求作为实型常量的小数点后面必须有数字，如果程序中出现小数点后面没有数字情况，它由词法分析器报错。词法分析器也就只能发现这样的错误。

最简单的错误恢复策略是“紧急方式”的恢复。它删掉输入指针当前指向的若干个字符（剩余输入的前缀），直到词法分析器能发现一个正确的记号为止。

另一种策略是进行错误修补尝试。最简单的办法是看一下剩余输入的前缀能否用下面的一个变换变成一个合法的词法单元：

(1) 删除一个多余的字符；

(2) 插入一个遗漏的字符；

(3) 用一个正确的字符代替一个不正确的字符；

(4) 交换两个相邻的字符。

这种策略基于这样的假设，大多数词法错误是多、漏或错了一个字符，或相邻的两个字符错位。这种假设通常是（但不总是）正确的。

## 2.2 词法记号的描述与识别

上一节提到，字符串集合由叫做模式的规则来描述。正规式是表示这些规则的一种重要方法，因此本节围绕正规式来介绍记号的描述与识别。在介绍正规式前，先给“语言”一个形式化的定义。

### 2.2.1 串和语言

术语字母表或字符类表示符号的有限集合，符号的典型例子有英文字母和标点符号。集合 { 0, 1} 是二进制字母表,ASCII 是计算机字母表的一个例子。

字母表上的串是该字母表符号的有穷序列。串 $s$ 的长度是出现在 $s$ 中符号的个数,往往写做 $|s|$。例如 banana 是长度为 6 的串,空串是长度为 0 的特殊串,用 $\varepsilon$ 表示。

术语语言表示字母表上的一个串集，属于该语言的串称为该语言的句子或字。这个定义相当宽,像 $\varnothing$(空集)和 $\{\varepsilon\}$ (仅含空串的集合) 这样的抽象语言也符合这个定义，所有语法正确的 Pascal 程序的集合和所有语法正确的英语句子集合也都分别符合此定义。当然，后两个集合更难描述。注意，这个定义并没有对语言中的串赋予任何意义，这个问题在第 4 章讨论。

如果 $x$ 和 $y$ 都是串,那么 $x$ 和 $y$ 的连接 (写成 $xy$)是把 $y$ 加到 $x$ 后面形成的串。对连接运算而言，空串是一个恒等元素，也就是 $s\varepsilon = \varepsilon s = s$.

如果把连接看成‘积’,那么可以定义串‘指数’。我们定义 $s^0$ 是 $\varepsilon$ ,定义 $s^i$ 为 $s^{i-1}s(i>0)$。因为 $\varepsilon s$ 是 $s$ 本身,所以 $s^2 = ss, s^3 = sss$ ,等等。

有一些重要的运算可以作用于语言。对词法分析而言，我们感兴趣的运算是和、连接和闭包，它们定义在表 2.2 中。我们把‘指数’算符也用于语言,定义 $L^0$ 是 $\{\varepsilon\}$, $L^i$ 是 $L^{i-1}L$ ,即 $L^i$ 是 $L$ 连接它自己 $i-1$ 次。

表 2.2 语言运算的定义

| 运算 | 定义 |
|---|---|
| $L$ 和 $M$ 的和(写成 $L \cup M$) | $L \cup M = \{s \mid s$ 属 $L$ 或 $s$ 属 $M\}$ |
| $L$ 和 $M$ 的连接(写成 $LM$) | $LM = \{st \mid s$ 属 $L$ 且 $t$ 属 $M\}$ |
| $L$ 的闭包(写成 $L^*$) | $L^* = \bigcup_{i=0}^{\infty} L^i$, $L^*$ 表示零个或多个 $L$ 连接的并集 |
| $L$ 的正闭包(写成 $L^+$) | $L^+ = \bigcup_{i=1}^{\infty} L^i$, $L^+$ 表示一个或多个 $L$ 连接的并集 |

例 2.2 令 $L$ 表示集合 $\{A, B, \cdots, Z, a, b, \cdots, z\}$,令 $D$ 表示集合 $\{0, 1, \cdots, 9\}$. 下面是用表 2.2 定义的运算作用于 $L$ 和 $D$ 所得到的新语言的例子。

(1) $L \cup D$ 是字母和数字的集合；

(2) $LD$ 是所有一个字母后跟随一个数字的串的集合；

(3) $L^6$ 是 6 个字母的串的集合；

(4) $L^*$ 是所有字母串 (包括 $\varepsilon$)的集合；

(5) $L(L \cup D)*$ 是以字母开头的所有字母数字串的集合；

(6) $D^+$ 是不含空串的数字串的集合。 □

从这个例子可以看出，从基本集合开始，利用语言上的运算可以定义新的语言。下面将用更有利于计算机处理的形式来定义一类简单的语言。

### 2.2.2 正规式

正规式（又称正规表达式）是按照一组定义规则，由较简单的正规式构成的，每个正规式 $r$ 表示一个语言 $L(r)$。定义规则说明 $L(r)$ 是怎样以各种方式从 $r$ 的子正规式所表示的语言组合而成的。

下面是定义字母表 $\Sigma$ 上正规式的规则，和每条规则相联的是被定义的正规式所表示的语言的描述。

(1) $\varepsilon$ 是正规式,它表示 $\{\varepsilon\}$;

(2) 如果 $a$ 是 $\Sigma$ 上的符号,那么 $a$ 是正规式，它表示 $\{a\}$。虽然都用同样的符号表示，但正规式 $a$ 是不同于串 $a$ 或符号 $a$ 的，从上下文可以清楚地区别所谈到的 $a$ 是正规式、串还是符号；

(3)假定 $r$ 和 $s$ 都是正规式，它们分别表示语言 $L(r)$和 $L(s)$，那么$(r)|(s)$、$(r)(s)$、$(r)^*$和$(r)$ 都是正规式，分别表示语言 $L(r)\cup L(s)$、$L(r)L(s)$、$(L(r))^*$和 $L(r)$。

正规式表示的语言叫做正规集。

如果约定：

(1) 闭包运算（算符是 $^*$）有最高的优先级并且是左结合的运算

(2) 连接运算（两个正规表达式并列）的优先级次之且也是左结合的运算

(3) 或运算(算符是 $|$)的优先级最低且仍然是左结合的运算

那么可以避免正规式中一些不必要的括号。例如，$((a)(b)^*)|(c)$等价于 $ab^*|c$.

例 2.3 令字母表 $\Sigma=\{a, b\}$,那么：

(1)正规式 $a|b$ 表示集合 $\{a, b\}$。

(2) 正规式$(a|b)(a|b)$ 表示 $\{aa, ab, ba, bb\}$ ,即由 $a$ 和 $b$ 构成的所有长度为 2 的串集。表示同样集合的另一正规式是 $aa|ab|ba|bb$。

(3) 正规式 $a^*$ 表示仅由字母 $a$ 构成的所有串的集合，包括空串。

(4) 正规式$(a|b)^*$表示由 $a$ 和 $b$ 构成的所有串的集合，包括空串。 □

如果两个正规式 $r$ 和 $s$ 表示同样的语言，则说 $r$ 和 $s$ 等价，写作 $r=s$。例如,$(a|b)=(b|a)$。

正规式遵守一些代数定律，它们可用于正规式的等价变换，表 2.3列出了正规式 $r$、$s$ 和 $t$ 遵守的代数定律。

表 2.3 正规式的代数性质

| 公理 | 描述 |
| --- | --- |
| $r\|s = s\|r$ | \|是可交换的 |
| $r\|(s\|t) = (r\|s)\|t$ | \|是可结合的 |
| $(rs)t = r(st)$ | 连接是可结合的 |
| $r(s\|t) = rs\|rt$<br>$(s\|t)r = sr\|tr$ | 连接对\|是可分配的 |
| $\varepsilon r = r$<br>$r\varepsilon = r$ | $\varepsilon$ 是连接的恒等元素 |
| $r^* = (r\|\varepsilon)^*$ | * 和 $\varepsilon$ 之间的关系 |
| $r^{**} = r^*$ | * 是幂等的 |

### 2.2.3 正规定义

可以对正规式命名，并用这些名字来引用相应的正规式，以求得表示上的简洁。这些名字也可以像符号一样，出现在正规式中。

如果 $\Sigma$ 是基本符号的字母表，那么正规定义是形式为

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\cdots$$
$$d_n \rightarrow r_n$$

的定义序列,各个 $d_i$ 的名字都不同，每个 $r_i$ 都是 $\Sigma \cup \{d_1, d_2, \cdots, d_{i-1}\}$ 上的正规式。由于每个 $r_i$ 只能含 $\Sigma$ 上的符号和前面定义的名字，因而不会出现递归定义的情况。把这些名字用它们所表示的正规式来代替，就可以为任何 $r_i$ 构造 $\Sigma$ 上的正规式。

为了区别名字和符号，本书在正规定义中用黑体字表示名字。

例 2.4 Pascal 语言的标识符集合含所有以字母开头的字母数字串，下面是这个集合的正规定义。

$$\mathbf{letter} \rightarrow A|B|\cdots|Z|a|b|\cdots|z$$
$$\mathbf{digit} \rightarrow 0|1|\cdots|9$$
$$\mathbf{id} \rightarrow \mathbf{letter}(\mathbf{letter}|\mathbf{digit})^*$$

例 2.5 Pascal 的无符号数是 1946,11.28,63.6E8 和 1.999E－6 这样的串，下面是这样的串集的正规定义。

$$\mathbf{digit} \rightarrow 0|1|\cdots|9$$

digits→digit digit*

optional _ fraction→.digits|ε

optional _ exponent→(E(+|-|ε)digits)|ε

num→digits optional _ fraction optional _ exponent

从这个定义可以知道，无符号数由整数部分、小数部分和指数部分三部分组成，其中小数部分和指数部分都是可能出现或可能不出现的。指数部分如果出现的话，是 E 及可能有的 + 或 - 号，再跟上一个或多个数字。注意小数点后面至少有一个数字，所以 num 能匹配 2.0，但不能匹配 2.。

在正规式中，某些结构频繁出现，为方便起见，用缩写表示它们。

(1) 一个或多个实例　一元后缀算符“ + ”的意思是“一个或多个实例”，即正规式 $a^+$ 表示一个或多个 $a$ 的所有串的集合。算符 + 和算符 * 有同样的优先级和结合性。代数恒等式 $r^* = r_+ | \varepsilon$ 和 $r^+ = rr^*$ 表达了这两个算符之间的关系。

(2) 零个或一个实例　一元后缀算符？的意思是“零个或一个实例”，$r$？是 $r|\varepsilon$ 的缩写。如果 $r$ 是正规式，那么($r$)？是表示语言 $L(r) \cup \{\varepsilon\}$ 的正规式。使用这两种缩写，可以用

num→digit+(.digit+)?(E(+|-)?digit+)?

来描述无符号数。

(3) 字符组　[abc]（其中 a、b 和 c 是字母表的符号）表示正规式 a|b|c。缩写字符组 [a-z] 表示正规式 a|b|…|z。使用字符组，可以用正规式

[A-Za-z][A-Za-z0-9]*

描述标识符。

### 2.2.4 状态转换图

本书以下面正规定义描述的语言为例，讨论怎样识别记号。

例 2.6　考虑下面的正规定义：

while→while

do→do

relop→<|<=|=|<>|>|>=

id→letter(letter|digit)*

num→digit+(.digit+)?(E(+|-)?digit+)?

其中 letter 和 digit 同前面所定义的一样。这是 while 语句中可能出现的部分记号的描述。词法分析器将识别保留字 while 和 do，还有关系算符、标识符和数。假定词法单元之间必要时由空格（或制表符、换行符）分开，词法分析器通过把剩余输入的前缀和下面的正规定义 ws 相比较来完成忽略词法单元之间的空格。