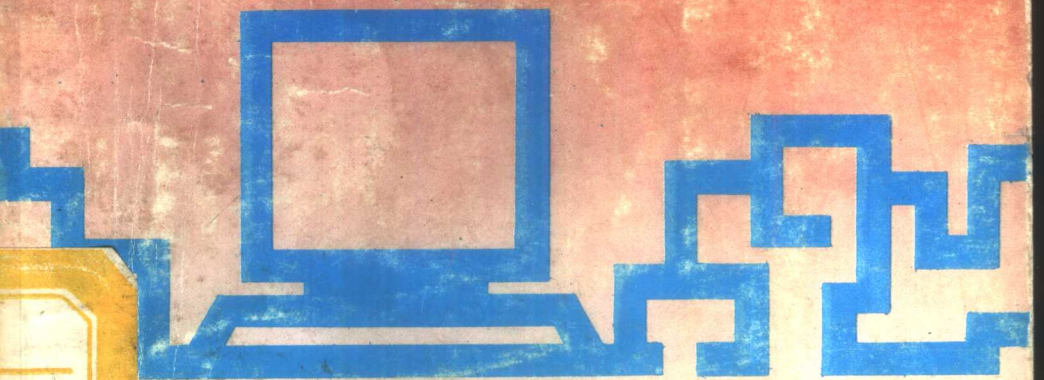
編譯程序

通過 Pascal 的設計與構造

(英) Robin Hunter 著
蘇運霖 陳力 譯



OMPILERS Their Design and Construction Using Pascal

暨南大學出版社

编译程序

通过 Pascal 的设计与构造

〔英〕Robin Hunter 著

苏运霖　阵力　译

暨南大学出版社

1991·广州

内 容 简 介

编译方法是计算机科学中最重要的内容之一。本书全面地介绍了设计与构造一个可靠的编译程序的理论与方法。内容包括基本理论、词法分析、语法分析、代码生成、优化、总体设计及出错恢复与诊断等。内容主次分明，反映了80年代的最新成果。

本书可作高等院校师生、计算机工作者和对此感兴趣的其他人员自学的编译课教材或参考书。

编译程序——通过 Pascal 的设计与构造

*

〔英〕Robin Hunter 著

苏运霖　陈力　译

*

暨南大学出版社

（广州　石牌）

广东省新华书店经销

华南师范大学印刷厂印刷

*

开本：850×1168　1/32　印张：10.5　字数：270千字

1991年3月第1版　1991年3月第1次印刷

印数：1—5000册

ISBN　7—81029—060—6/T·1

定价：4.90元

译 者 的 话

编译程序是计算机专业最重要的专业课程之一。为教好、学好这门课程，专业教师希望有一本适宜教学的，系统性强的，并能反映该专业最新成就的教材；学生亦希望得到一本易懂、易学、便于实际掌握其内容的学习参考书。好的编译程序教材不仅适用于教学部门，对于离开学校的计算机工作者也是一本好的参考书，"学而时习之"。现在我们向读者郑重介绍由 **Robin Hunter** 著的 **"COMPILERS Their Design and Construction Using Pascal"** 一书，因为我们感到它是一本国内还未曾见过的极佳教材。通过数年以该书原版作为教材的教学实践，我们对此书有了深入的了解；学生也普遍对此书表示欢迎。本书重点在于讲述如何设计与构造一个编译程序，并介绍了几个最新的算法，由浅入深，由表及里，层层剖析，重点突出，详简得体，作者以其精练的笔触把现代编译技术的精华全面烘托出来，使读者能立即掌握编译技术之精髓，迅速步入编译技术之前沿。我们觉得，把这本书翻译出来，既可为更多院校提供现成的中文教材；也可使拥有英文原版的读者对照学习。

本书由苏运霖教授主持翻译的全部工作。第1至第6章和第10至第13章由陈力翻译；第7至第9章由苏运霖翻译；习题解释由二人合译。最后由苏运霖对全书译稿作了详细地校对与修改。对书中发现的错误也一一作了改正。谢宜晖、周子兴、孔伟、姚志阳等同志，对本书的翻译与出版做了大量的工作。暨南大学出版社的领导

和工作人员也对本书的出版予以很大支持与帮助，谨此一并致谢。因译者水平所限，错漏及不妥之处在所难免，恳请读者不吝指正。

译者识
1990年11月于
广州暨南大学

原　　序

如书名所指出的那样，本书是一本重点在于如何设计与构造的编译程序教材，书中所描述的各个算法都是用 **Pascal** 写的——因此，标题中最后的短语为“通过 **Pascal**”。另外，从实现的观点来看，**Pascal** 也是所考虑的主要语言之一。

本书部分地以1981年首次出版的《编译程序的设计和构造》为基础，也是它的继续。鉴于在 **Strathclyde** 大学计算机科学系所进行的编译程序工作，前一本教材是面向 **ALGOL** 68 语言的，其中 **ALGOL** 68 不仅用来描述算法而且用来阐述实现问题。现在这本教材仍同前一本保持相同的基本方法，但从编译的各方面来看，语言的范围更宽了，所考虑的主要语言有 **FORTRAN**、**ALGOL** 60、**ALGOL** 68、**PASCAL** 和 **Ada**。

本书可用作计算机科学专业本科生或其他对编译程序感兴趣的人的编译程序课程的基础。各章后配备的习题帮助学生把问题考虑得更清楚。本书末尾给出了第 1—12 章习题的解答要点。

第 1 章通过描述典型的高级语言和机器的特征，以及通过讨论编译程序项目可能的设计目标来介绍编译过程。第 2 章考虑了程序设计语言语法的形式定义，其中一节是关于属性文法的。词法分析包括在第 3 章中，而自顶向下和自底向上的语法分析方法在第 4 章和第 5 章中给出。

第 6 章介绍了在上下文无关文法中嵌入编译程序中的动作的思想；而第 7 章讨论了编译程序的总体设计。符号表和类型表在

第 8 章中讨论；第 9 章论述存储分配和废料收集。

第 10 章是关于代码生成的，并引入了 **Pascal P**—码作为中间语言。**P**—码翻译成汇编码在第 11 章中讨论。第 12 章考虑了出错恢复技术，而第 13 章讨论了生成可靠的编译程序的问题。

致　　谢

我非常感谢 **A. D. McGettrick** 教授和 **R. R. Patel** 先生对编译程序问题的许多有用的讨论；也感谢 **C. Conlan** 小姐帮助打印本教材的一部分，以及我的妻子 **Kate** 在准备原稿中所提供的帮助。

目　　录

第一章　编译过程

用高级（或面向问题的）程序设计语言写成的程序必须转换成等价的机器码程序后才能在计算机上执行。自 1950 年代中期就已经有了高级语言，早期的语言有 FORTRAN 和 COBOL．更近一点的有 ALGOL 68、Pascal 和 Ada。有一种程序，它能够把某种高级语言写的任何程序翻译成与之等价的其它某种语言的程序（常为某具体机器的代码），这种程序叫**编译程序**。编译一个程序包括**分析**——确定程序的预期作用；和**综合**——产生等价的机器码程序两个部分。在分析过程中，编译程序应能够检查输入程序是否有任何非法情况（即不属于该编译程序所支持的语言），如果有，应该给程序员返回一个适当的信息。编译的这一方面叫**出错检查**。

过去 25 年来，编译程序的技术已有了相当大的进展。早期构造的编译程序常用凑和的方法，倾向于进展速度慢，且缺乏结构性。（关于构造编译程序的简史见 Bauer[1974]）。较现代的编译程序一般使用更系统的方法，因而进展相对地要快一些，并且结构性好，以便尽可能地把编译的不同方面分开。

还有一种方法，不用把每个程序翻译成机器码然后再执行它；而是首先把程序翻译成中间语言，然后每当遇到中间语言语句时再翻译并执行它。一个程序若以这种方式翻译并执行高级语言程序，就叫**解释程序**。解释程序优于编译程序之处是：

1．常常更容易根据源程序给用户传送出错信息。

2．程序的中间语言版本往往比编译程序产生的机器码更紧凑。

3. 对源程序的部分修改不必重新编译整个程序。

交互式语言，如 BASIC，常用解释程序来实现，普通的中间语言乃是某种形式的逆波兰表示（见 10.1 节）。实现交互式语言的较好的教材是 Brown [1979]。解释程序的主要缺点是程序运行较慢，这是因为中间代码的语句每执行一次都要先翻译一次，尽管如果中间语言设计得好的话这种时间消耗可能不是很大。一个折衷方法是 Dakin 和 Poole [1973] 的**混合编码方法**，程序中最常执行的部分被编译而其余部分被解释。这可节省空间，因为被解释的那部分程序很可能比编译后的代码更紧凑得多。作为这个思想的扩展的是所谓的 '抛弃式编译'，它是由 Brown [1976] 提出的。

本书中我们主要讨论编译程序而不是解释程序，虽然有许多思想二者都适用。在讨论编译程序时我们把要被编译的程序叫**源程序**（或源文本），而产生的机器码叫**目标代码**（图 1.1）。类似地，高级语言可以叫做**源语言**，机器语言叫做**目标语言**。

图 1.1

本章我们讨论高级语言和典型的计算机的特性，以及编译过程的各个方面。这使得我们要去考虑编译程序的总体设计。

1.1 语言和机器的关系

尽管从语言的使用者（和实现者）的观点看，各种高级语言之间的差别是相当大的；但是在这节里我们希望强调的是这些语

言之间的类似性，以便试图来指出编译程序必须实现的任务的类型。我们心目中的典型的高级语言有 BASIC、FORTRAN、PL/1、Pascal、ALGOL 68、Ada 和（可能提到不多的）COBOL。用来阐述我们观点的程序段是用 Pascal 写的，当然用任何其它的语言来写都行，不熟悉 Pascal 的读者阅读它们不应该有大的问题。

语言

大多数高级语言的共同特征是：

1. 表达式和赋值语句

通常能够计算表达式并且（其中一种可能是）把它的值赋给一个变量，例如：

$a := (b+c) * (e+f)$

其中假设 b、c、e 和 f 都具有适当的值。

2. 条件语句

一个语句的结果可能依赖于布尔（真）值：

if $a=b$ **then** $x := x+y$ **else** $x := x-y$

它的作用是：如果 a 等于 b 则把 $x+y$ 的值赋给 x，否则把 $x-y$ 的值赋给 x。

3. 循环语句

一个语句（可能是复合的）可以执行若干次

```
for k := 1 to 10 do
begin   read(i);
        write(i)
end
```

在此复合语句中的语句序列，即

```
read(i);
write(i)
```

被执行 10 次。

4. **输入/输出**

通常有一些简单的实用程序用于从标准的输入/输出通道上读入或写出简单的值，例如：

read(*x*, *y*, *z*)

是读入 *x*、*y* 和 *z* 的值，而

write(*a*, *b*, *c*)

则写出 *a*、*b* 和 *c* 的值。

5. **过程/子程序/函数**

对于用高级语言编写的程序，通过把它划分成一些子程序，并把每个子程序写成为一个独立的过程或函数，可以给它以某种结构。例如，如果所涉及的一个程序多次计算十个实型数的平均值，那么就可以写出如下的函数：

```
function average(row : array10) : real;
var   sum : real;
      i : integer;
begin   sum : =0;
        for i : =1 to 10 do
        sum : =sum+row[i];
        average : =sum/10
end
```

假设 *array*10 已定义如下：

```
type array10=array[1..10] of real
```

则 *average* 可以被调用如下：

write(*average*(*a*))

其中 *a* 已经被说明为

```
var a : array10
```

并且已被赋了 10 个实数值。

在 Pascal 中，象 sum 和 i 这样的变量可以说明为局部于函数或过程中，从而使范围不重叠的变量得以共享存储空间。类型标

识符、过程和函数也可以局部地说明。在象 ALGOL 60、ALGOL 68 和 Ada 这样的语言中标识符甚至可以被更加局部地说明，即在相当于 Pascal 中的复合语句这一级中说明。ALGOL 中带有说明的复合语句叫分程序。分程序结构对存储分配的要求将在第 9 章中讨论。

现时我们不准备多说各高级语言在诸如提供的循环语句和条件语句的方式或可利用的运算符的类型等方面的差别。Aho 和 Ullman[1977]的第二章从编译程序编写者的观点对语言的差别作了很好的概括。

能极大地影响编译程序结构的一个语言特点是类型的思想。在大多数语言中每个可能的值都属于有限个或无限个类型中的一种；而每一种类型又对应着有限个或无限个值。例如，所有的整数、实数或某字符集中的字符组成的集合可以看成是类型。在 Pascal 中，我们可以举出定义整数类型的子类型的例子：

type *upto*100=1..100

它是 1 和 100 之间的所有整数值的集合。同大多数类似的语言一样，Pascal 还有对于数组和记录的集聚类型。ALGOL 68 中的方式大体上对应于类型，但又更一般化。例如，有一个对应于具有一个**实型**参数及一个**实型**结果的过程的方式。

有些语言，如 Pascal 和 ALGOL 68，叫做**强类型的**或具有**静态类型**，即程序中所有值的类型在编译时就知道了。这就对语言引进了冗余性，因而增加了编译时可进行的检查。例如，如果 x 是一个字符变量，则编译程序将检测到

$x:=3$

是非法的。PL/1 虽然也是强类型的，但它将不检查上面的‘错误’，这是因为它有许多隐含的类型转换（如整型到字符型）。

象 POP－2、APL 和 LISP 这样的语言叫作**弱类型的**或具有**动态类型**，这是因为类型要到运行时刻才知道。例如在 POP－2 中，说明语句

```
VARS X, Y;
```

意味着 X 和 Y 可取任何类型的值。一个赋值语句，如

```
X→Y
```

(POP－2 中赋值是从左到右进行）总是合法的。但是，在

```
X+Y
```

中，把二元运算符＋应用于 X 和 Y 之上就可能合法也可能不合法了，这取决于 X 和 Y 的**当前**类型。这一点只能在运行时刻检查，所以目标代码中必须包含这种运行时刻的检查，因而不可避免的代价是减缓了目标程序的速度。

静态类型与动态类型之间的折衷方法是使在主要部分中类型是静态的，但还有一个额外的类型（比如叫**任意类型**或**一般类型**），它的值可以取语言中任何其它的类型。这个方法选自 CPL 和 ESPOL（Burroughs ALGOL 60 超集）。

关于类型的第三个方法是由 BCPL 采用的，它只有一种类型，它可被认为是位模式。不同的类型可存在于程序员的头脑之中，但程序或实现机构不知道是什么类型。程序员的责任是不要试图把布尔值和整数相加。在编译时刻和运行时刻都不作类型检查是可能的。

机器

与 BCPL 一样，机器码中不存在类型的概念。计算机处理位模式而不对它们赋予什么意义。和语言一样，计算机的细节差别也很大，但总能够列出一些典型的特性以便阐明编译程序必须完成的任务。对程序员来说，各种机器的特点和它们的有用性关系很小，不过这些特点与计算机总体设计时的价格——性能的折衷考虑关系就大了。大多数机器的共同特点包括：

1. **线性的主存**

主存具有存储大量位（二进制数字）的能力。它们通常被分成 8 位一组的字节并顺序排列以便用 0 到 M－1 中的整数给每个

字节编址，其中M为主存的字节数，通常为2的幂。一个字节可用于表示一个字符。由若干个字节（如2个或4个）组成的一组可以叫一个字，而且在某些机器中字是可寻址的最小单位。

2. **寄存器**

象主存中的字一样，寄存器也可用于存储数据。计算机执行的所有算术运算中通常都涉及到寄存器，而且在寄存器中存取值所花的时间比在主存的字中少得多。在任何计算机中通常都只有少量的（可能8或16个）寄存器。

3. **指令系统**

每个机器都有一指令系统，叫做它的机器码。每个指令由一个2进制序列组成，通常还带几个参数。这些参数也是几个2进制数序列，典型的可能包括一个寄存器名或一个主存地址。每个指令（包括参数）通常刚好占几个字（也可是半字），在程序执行期间它们被放于主存中。某些机器码有定长指令，还有一些有可变长的指令。当使用助忆符代替指令代码的二进制数并用十进制数代替其它地方的二进制数字序列之后，典型的机器码指令可能变成

LDX 1 4444

意为把地址4444中的内容装到寄存器1中。（因为事先可能不知道在执行期间程序在主存的什么地方，因此地址域通常不是绝对的而是相对于某基值的。），

STO 2 2000

意为把寄存器2的内容存到地址2000中，

ADX 2 102

意为把地址102中的内容加到寄存器2上，

BRN 4000

意为跳到地址4000且从这个地址继续执行指令。

机器的其它特点，从编译的观点看，同我们关系不大，它们是：