微型计算机程序设计

〔丹麦〕P. B. 汉森 著

科学出版社

微型计算机程序设计

〔丹麦〕P. B. 汉森 著

林定基 译

科学出版社

1987

内 容 简 介

本书介绍了一个功能较强而又相当简单的 Edison 系统，这是一个在微型计算机上实现的可移植性较好的软件系统。它包括操作系统及其内核、Edison 语言的编译系统、屏幕编辑、文本格式化及打印程序等。作者通过对这个典型系统的分析，把软件系统的抽象的概念教学与具体工程实现结合起来，使读者易于理解并可尽快掌握小型系统软件的结构、设计与实现。

全书共分十章。第一章阐明基本的设计思想，第二、三章介绍程序语言的设计与 Edison 报告，第四章介绍操作系统的主要设计准则，第五章描述 Edison 系统的操作与程序设计的约定，第六章讨论能被内核解释的指令系统，第七章阐述用于实现内核的 Alva 语言，第八、九、十章提供了内核、操作系统和编译程序等的全部文本及其注释。

本书对于学习系统程序设计、数据结构、程序设计语言、计算机体系结构、编译程序、操作系统等课程均有一定参考价值，可作为从事计算机及其应用的软件人员的进修读物，亦可作为大专院校高年级学生和研究生的教材。

Per Brinch Hansen
PROGRAMMING A PERSONAL COMPUTER
Prentice-Hall, Inc., 1982

微型计算机程序设计

〔丹麦〕P. B. 汉森 著
林定基 译
责任编辑 孙月湘
科学出版社出版
北京朝阳门内大街 137 号
中国科学院印刷厂印刷
新华书店北京发行所发行 各地新华书店经售
*
1987 年 9 月第 一 版 开本：787×1092 1/16
1987 年 9 月第一次印刷 印张：20 3/4
印数：0001—5,700 字数：477,000
统一书号：15031·849
本社书号：5354·15—8
定价：4.90 元

译　者　序

近几年来，我国微型计算机的装机数量正以成倍的速度急剧增长．微型计算机的应用已开始遍及国民经济的各个领域，初步显示出它在当代新技术革命中所起的非凡作用．

随着计算机应用水平的不断提高，广大用户为了更充分地发挥计算机的效能，除了要熟悉各种应用软件外，常常需要对系统软件有较深入的了解；一些高等院校的师生已不再满足于一般的抽象的概念教学，而是要求进一步掌握具体的结构与实现技巧．然而，这些问题的解决并不是很容易的．通常，系统软件是用汇编语言写成的，可读性较差；此外，加之工程实现中还夹杂着某些特殊的实际问题，这就给初学者阅读程序文本带来很大困难，许多人甚至望而生畏，不敢深究．这对计算系统的研制及应用水平的提高都是不利的．

本书作者 P. B. 汉森以他丰富的经验，在微型计算机上实现了一个十分简洁而又完整的 Edison 系统．这个系统有一个规模不大的内核，还有用 Edison 写成的操作系统、Edison 编译程序、屏幕编辑程序、文本格式化以及打印程序等．Edison 语言是一种类似 Pascal 和并发 Pascal 的语言，用它写成的系统软件具有较好的可读性和可修改性，因而便于读者阅读和理解，并掌握系统软件的整体概念、具体细节和实现技巧．

本书以 Edison 系统为典型范例，全面介绍了系统内核、操作系统及编译程序等的设计与实现．书中提供了详细的程序文本并带有必要的注释．

本书原名为“Programming a Personal Computer”，直译应为《个人计算机程序设计》；但译者考虑到本书内容实际上适合各种规模的微型计算机，因此改译成《微型计算机程序设计》．

最后，译者感谢刘捷等同志在整理本书译稿时所给予的帮助．

序　　言

在一个很短的时期内，个人计算机已经为程序员制作高质量的小型软件系统创造了条件，这种小型软件使用了目前最好的程序设计语言和设计方法.

个人计算机吸引了新一代的程序设计者，他们对能否使用为已往的计算机所编写的软件很少或根本不感兴趣. 因此，计算机制造商们可以自由地选择新的程序设计语言并开发合理的计算机结构，以适合个人计算的要求.

个人计算机的简单操作步骤和小容量存贮器使软件的复杂程度受到限制.

由于这些原因，个人计算机为提高软件的质量提供了很好的机会. 这个目的确实已经被 UCSD Pascal 系统所实现 [Bowles, 1980]. 但是，它还没有出现工业上广泛采用的趋势. 大多数微型计算机的操作系统仍然是用汇编语言写成的.

最近发展的复杂的程序设计语言 Ada 是与新型的带有较大容量存贮器的微处理器相联系的，不久的将来，难以理解的、不绝对可靠的软件的开发甚至对个人计算机也是不可避免的.

在这些情况发生以前，我愿意为软件设计者们提供一个清晰的方案：这是一个功能很强但却简单的软件系统，它足以支持在个人计算机上开发较复杂的程序，而且足以在程序设计的各个水平上加以细致地研究.

本书描述 Edison 系统，该系统支持用程序设计语言 Edison 编写的程序的开发；这种语言是为微处理机设计的一种类似于 Pascal 的语言. Edison 系统是在带软盘的 PDP-11/23 微型计算机上开发的. 但是，它可以移植到其他微处理机上[1].

软件设计的主要困难就是人们理解大量细节的困难. 一般的教科书忽略了琐碎的具体内容，而把注意力集中在过于简化的(但却有效的)设计原理上. 然而，在本书中，我不能压缩程序设计的细节，因为我的目的在于阐明如何建立所有细节都能理解的软件.

本书解释了上述程序设计语言和软件系统是怎样设计和实现的. 它包括了操作系统和编译程序的程序文本，两者都是用 Edison 写成的. 它还包括了用于 PDP-11 计算机的系统内核的文本内容.

本书并不需要从头到尾阅读，除非你希望把这个软件系统移植到另一种计算机上. 作为 Edison 系统的新用户，你可以从系统报告(第五章)开始阅读，这章解释如何运行这个系统. 在你可以为系统编写程序之前，有必要读一下与程序设计语言 Edison 有关的第二章和第三章.

本书为几乎所有有关计算机程序设计的大学课程的学习提供了实际范例. 这些课程是:

系统程序设计

---

1) 包括 IBM 公司的个人计算机.

数据结构
程序设计语言
计算机体系结构
编译程序
操作系统

本书是为专业程序设计人员和学过程序设计语言、编译程序以及操作系统的大学生编写的. 必要的背景已在 Wirth [1973, 1976a] 和我本人 [Brinch Hansen, 1973] 的著作中描述过. 如果你要了解其中的某些资料,而不是全部,那么你就可以跳过那些标有星号的章节.

我已将本书在 USC (南加里福尼亚大学)用作大学教材, 让学生们体会到如何把在计算机科学的主要课程中所学到的原理用到一个完整软件系统的设计中去.

我的早期著作《并发程序的结构》(The Architecture of Concurrent Programs, Prentice-Hall, 1977) 描述了一个被称为 Solo 的单用户操作系统, 它是用程序设计语言 Pascal 和并发 Pascal 写成的. Solo 系统是为了评价监控概念的用途而进行的实验性研究.

Edison 系统是一个旨在进一步简化熟知设计思想的工程课题. 它由一种单一的程序设计语言写成. 这种语言比 Pascal 要小,但却比 Pascal 和并发 Pascal 的组合更有效.该系统可在微型计算机上运行,所占用的内存和磁盘容量分别只有 Solo 系统的 2/3 和 1/10.

John Wiley and Sons 公司热情地给予翻印了以下文章的大部分内容:

Edison——一种多处理器语言.

Edison 的设计.

Edison 程序.

软件——实践和经验, 1981. 4.

我很感谢 Mostek 协会的 L. J. Sevin, Steve Goings, Nick Matelan 和 Vincent Prothro; 他们为开发程序设计语言 Edison 提供了机会. 语言报告由于 Peter Naur 的结构性的注释而得到显著的改进. 文本中一些有益的注释也是由 Dines Bjorner, Jon Fellows, David Gries, Tony Hoare, Peter Lyngbaek, Habib Maghami 和 Harlan Mills 提供的.

本书用来纪念丹麦计算机发展的创始人 Niels Ivar Bech. 在 Bech 的热情领导下,年轻一代的丹麦人为计算机程序设计做出了极大的贡献, 例如 Algol 60 报告、Gier Algol 编译程序和 RC 4000 多道程序设计系统.

P. B. 汉森

目　录

第一章 软件质量

本书描述的 Edison 系统是面向专业程序设计人员的个人软件系统. 这个系统包括一个编译程序、一个操作系统、一个屏幕编辑程序、一个文本格式化程序以及少量的其它程序;所有这些都是用程序设计语言 Edison 写成的.

我们从研究简单性在软件设计中的决定性作用着手. 历史学家 William McNeill [1967]指出: "不可理解的事物是没有意义的,正常人一般不会把注意力放在无意义的事情上."

一个专业程序人员很容易将一台小型计算机用到它的极限. 在 Edison 系统设计期间我就做过尝试. 在 PDP-11 微型计算机上,系统可以在 28K 字的存贮器内编译 Edison 编译程序最大的一次扫描,仅留下 200 个字未用! 这不是偶然的,而是我故意用一些有用的功能来对操作系统加以扩充,直至达到存贮器的极限.

在重要的用户程序设计中,程序设计人员也必须清楚地意识到系统的极限. 否则,这样的程序也许不能在一个小型计算机上运行. 在某些应用中, 甚至有必要使系统适应不同种类的后备存贮器.

除非程序设计人员充分理解了该软件系统,不然这些事一件也不可能实现.

努力追求极端的简单性才能精通软件的一些细节.

尽管软件设计的实际需要是要使简单性成为基本要求, 但一个更深刻的原因可以从创造性工作的特点中找到. 发明的喜悦和实施某种事情的乐趣在科学和工程中是最有力的动力. 要承认这一点,一个软件工程师必须寻找惊人的简单性和完美的设计方案.

简单性是人们建立、理解和使用一个系统所需的努力的度量. 我们对简单性的度量是: 如果一个个人软件系统,在不到一天内就能学会使用、一月内就能详细地理解它、一年内可以建立它, 那么这个系统就是简单的. 个人计算的吸引力就在于一个人能完全精通!

下面,我将提供几条设计准则,以帮助读者使软件系统简单化.

1.1 使用的简单性

以下三条基本设计规则对那些极不熟悉软件系统的用户来说是有帮助的.

1. 明确规定系统的用途

在一个带有字母显示终端、双软盘驱动器和打印机的个人计算机上, Edison 系统支持对 Edison 程序的开发和编制说明文件. 系统能使单用户从终端调用下列操作:

文本输入
文本存贮
文本编辑
文本格式化

文本打印
程序编译
程序存贮
程序执行

2. 精确描述操作约定

尽管一个专业程序设计人员能够了解一个个人软件系统的完整细节，但显然他（或她）也能够忽略程序设计的细节而完全依据一份系统命令及其作用的精确描述来工作。Edison 系统报告就是这样一种描述(见第五章)。

3. 使系统易于使用

Edison 系统实现的所有操作都是由简单命令调用的。用户只需记住操作的名字并在终端上键入。系统将提醒用户完整地定义这个操作所需要的参量。

少数键入错误可以用一组统一的行编辑约定来校正，这些约定贯穿整个系统。文本编辑程序允许用户将一个文本文件在屏幕上来回移动并进行修改、插入和删除，这里只用了少数几条编辑命令，这些命令的约定很象普通的键入。

1.2 程序设计的简单性

下面的设计规则使用户能够很容易地为一个系统添加新的程序并了解现有的程序。这些规则本身也简化了建立系统本身的任务。

1. 保持系统小型化

如果软件是由一个人在一年内为一台小机器开发的，那么它必然是小型的。因此，这样做是合理的，对一台个人计算机不必更复杂。不用说，一个小系统只能做少量的事情，但这也没有办法。

2. 自始至终使用同一种语言

本书描述了一个程序文本约有 10,000 行的软件系统。如果你想在一个月之内了解它，那你必须每天(包括周末)阅读 500 行。设计人员在做这件事时，为了减轻负担，最低限度要求整个系统使用同一种语言。

Edison 系统包括一个操作系统、一个编译程序、一个屏幕编辑程序、一个文本格式化程序、一个打印程序和一个 PDP-11 汇编程序。所有这些程序都是用程序设计语言 Edison 写成的。

唯一用汇编语言写成的程序是系统内核（1800 字），它解释编译程序所生成的代码。这个内核最初是用 Edison 编写和测试的，后来用人工的方法将它翻译成汇编语言，而用 Edison 语句作为注释。

3. 不考虑机器的细节

大多数计算机仅仅在一些无关紧要的细节上有所区别，其中只有少数细节会影响到程序设计语言的系统实现。因此，尽可能地忽略机器的某些细节是完全有理由的。这仅能通过使用抽象程序设计语言来实现，其中存贮器地址、寄存器和位模式可以用变量、表达式和值的概念来代替。

此外，我们可以使编译程序为一台适合于该程序设计语言的理想机器产生抽象代码，

然后用一个较小的、与机器有关的程序(系统内核)有效地解释抽象代码.

4. 使用统一接口

最复杂的程序设计任务之一是设计一个文件系统，该系统应使所有的程序都能有效地利用软磁盘，而不必考虑空间分配和数据存取的一些细节. 软盘驱动器要比微处理器慢几个数量级. 由于这类设备的特性严重地限制了整个系统的性能,因而不能忽视它们. 人们希望最好能找到一组统一的管理磁盘的过程，它可以由该操作系统一次性地加以实现,而且可由所有其他程序来调用.

然而,在操作系统和所有其他程序之间的统一接口只有当它较小时才便于简化. Edison 操作系统实现了一组过程，它使其他程序能使用终端、磁盘和打印机，这组过程共 33 个.

1.3 程序设计语言的简单性

在大多数情况下,一个聪明的软件设计者将选用一种现有的程序设计语言,而不企图去设计一种新的语言. 因此不论在哪种情况下,都可以提出如下建议:

1. 保持语言的小型化

要判断一种程序设计语言是否是小型的，只需注意语言报告和编译程序. 定义程序设计语言 Pascal 和 Edison 的报告还不到 50 页. 用 Edison 编写的 Edison 编译程序只包含 4200 行程序文本.

从现有的一种较好的语言（如 Pascal）着手，同时省略所有并非绝对必要的语言特性，就能使一种新的程序设计语言小型化. 你可能需要进一步改进所保留的该语言的一些特性，使之能简明地表达某些已被取消的概念. 但是，该语言的任何扩展都应该是最小的.

2. 精确地定义语言

一种抽象程序设计语言的主要价值在于它具有不必考虑硬件和软件细节的能力. 然而,只有当该语言报告以与系统无关的项来精确地描述每个语言特性的作用时,程序员才能安全地不顾及计算机、操作系统和编译程序的特性. 写这种报告的困难将在第二章中讨论.

1.4 其他软件特性

前面关于简单性的讨论看来是不够全面的,软件的其他一些特性显然也是重要的.

1. 可靠性

经验表明：小型软件系统能制作得比运行它的硬件更可靠. 一旦发生什么错误，那么对于简单而且资料完善的系统要寻找错误显然是很容易的，所以简单性和可靠性是相辅相成的.

2. 效率

通过选择直接在现有计算机上能实现的语言结构，以及通过在编译期间完成对代码的简单优化,我们就能达到有效的运行. 然而,影响效率的主要因素是设计一个完善的文

件系统，使磁头的移动最少。

3. 可扩展性

用一些新程序来扩展一个软件系统的能力是由统一的程序接口设计来支持的。

4. 可移植性

一个执行与机器无关的代码的系统可以用重写内核的办法移植到其他计算机中。要让内核尽可能小就必须选择一种小型的程序设计语言。

因此，在许多情况下，简单性直接影响到其他所希望的软件特性。然而，与简单性有严重矛盾的那些软件特性必须舍弃。

关于软件质量的其他讨论可以在 Hoare [1972a] 和 Brinch Hansen [1977]的著作中查到。

第二章　程序语言的设计

2.1 背　景

程序设计语言 Pascal 的发展是软件技术中的一个里程碑 [Wirth, 1971]. 首先，一个程序员现在已经能够使用抽象表示法来描述数据类型和操作. 这种与机器无关的表示法使一个程序员有可能在一年内写出一个正确、易懂的可移植的编译程序.

面对硬件成本的急剧下降和软件成本的持续上升，计算机工业最终将不可避免地采用这种优越的程序设计工具，以支持新的微处理器技术.

自从 Pascal 语言诞生以来，在软件技术方面已出现了两个有效的突破：

(1) 现在已经广泛地认识到，把数据结构和操作分组成为一些程序模块，可以更清楚地阐明大型程序的内容.

(2) 这种用抽象语言编写并发程序的能力正在成为新一代以微处理器为基础的多处理器体系结构的基本能力.

监控概念的诞生导致了支持模块化和并行化的新一代程序语言的开发 [Brinch Hansen, 1973; Hoare, 1974]. 程序设计语言并发 Pascal 和 Modula 对我们为小型计算机研制简单而正确的操作系统和实时系统的能力已产生了引人注目的影响[Brinch Hansen, 1975, 1977; Wirth, 1977]. 但必须承认，这些语言是基于某些复杂概念的.

将近代软件技术中的一些主要优点合并到一种程序设计语言中，在这方面，Edison 语言要算是一种尝试. 这种语言比 Pascal 简单，它支持用于微型计算机的顺序和并发程序的模块化构造.

Edison 语言的简单性是通过取消许多众所周知的语言概念，以及澄清那些早先混淆的问题而得以实现的. 本章描述了设计 Edison 语言的一些考虑，也讨论了如何写一个简明语言报告的语言学问题. 我假定读者已经熟悉 Pascal 语言.

2.2 数 据 类 型

Pascal 的最有效的进展是引入了一组数据类型，这组数据类型使系统程序设计能用抽象程序设计语言来实现. 尽管如此，Pascal 的某些数据类型仍然很复杂，而且定义得并不准确.

在 Edison 语言中，我力求简化并阐明有关数据类型的问题.

2.2.1 类型概念

通过观察 Edison 语言的标准类型，即整数、布尔和字符，就能很好地阐明类型的概念.

1. 类型由一些值的有限集合组成

整数类型包含连续整数的一个有限集合：

$$\cdots, -2, -1, 0, 1, 2, \cdots$$

布尔类型包含两个真值：

false, true

字符类型包含字符的一个有限集合：

$$\cdots, \text{'a'}, \text{'b'}, \text{'c'}, \cdots$$

2. 每种类型都有一个名字

标准类型被命名为

int, boot, char

3. 每个值属于且仅属于一种类型

或者换一种说法，类型是不相交的值的集合。

4. 每个常量、变量和表达式属于一个固定的类型

在程序执行中，操作数的类型限制了它的可能值。操作数的类型可以在程序文本尚未执行之前确定。

常量的类型可由它的语法或名字给定：

15 'a' false

或通过它的说明给定：

```
const max = 100
```

变量的类型由它的说明给定，例如

```
vax x:int
```

表达式的类型可由它的操作数类型和已知的运算结果的类型来决定。

5. 每个操作要使用一些固定类型的操作数并产生一些固定类型的结果

作为一个例子，我们来比较两个整数是否相等，由此产生了一个布尔结果。

除了 (2) 和 (3) 以外，Pascal 中的类型概念也都具有同样的性质。这就导致了 2.2.3 节和 2.2.16 节要讨论的那些概念上的难点。

2.2.2 基本类型

整数、布尔和字符类型都有类似的性质。这些类型均包含值的一个有限的有序集合。这基本上是修订的 Pascal 报告中所用的定义，在那里，这些类型被称为纯量类型或标量类型 [Jensen and Wirth, 1975] (不幸的是，这个报告把实数类型也看做是纯量类型，这就无意中使报告的另一处出现数组下标可以是实数类型的迹象)。

在 Edison 中，这些类型称为基本类型，它们与整数的相似性叙述如下：基本类型的连续值能被映象到一组有限的称为有序值的连续整数上。

一个基本值 x 的有序值记作 int(x)。对任何整数值，如 5，其有序值为该整数自身，即

$$\text{int}(5) = 5$$

对布尔值，我们规定

$$\text{int}(\text{false}) = 0 \qquad \text{int}(\text{true}) = 1$$

字符的有序值是依赖于系统的。对 ASCII 字符集来说，某些有序值的例子如

$$\text{int}(\text{'a'}) = 97 \qquad \text{int}(\text{'b'}) = 98 \qquad \text{int}(\text{'c'}) = 99$$

从整数到布尔值的逆映象定义如下：

bool(0) = false　　bool(1) = true

同样，对于字符来说

char(97) = 'a'　　char(98) = 'b'　　char(99) = 'c'

把 0—9 范围中的整数值 x 变为打印的数字字符，这个常见的问题是由映象

char(x + int('0'))

解决的.

Pascal 使用类似的表示法

chr(x + ord('0'))

但当 x 是一个整数表达式时，不能描述逆映象 bool(x).

整数、布尔和字符类型是 Edison 语言的标准类型. 其他基本类型，如枚举类型，可以通过类型说明来导出，如

enum task(flow, scan, log)

这个说明引出了一个名为 task 的枚举类型，它包括三个相继的值，分别称为 flow, scan 和 log.

这些值有相应的序值

int(flow) = 0　　int(scan) = 1　　int(log) = 2

从整数到 task 各个值的逆映象定义如下：

task(0) = flow　　task(1) = scan　　task(2) = log

在 Pascal 中，这种类型是这样说明的：

type task = (flow, scan, log)

Pascal 中不能描述逆映象.

在 Pascal 中，纯量值 x 的前趋和后继记为 pred(x) 和 succ(x)，例如

pred('b')　　succ(scan)

在 Edison 中，这些值通过下列映象来定义：

char(int('b') − 1)　　task(int(scan) + 1)

尽管这种表示法不太美观，但它却使语言省略了两个标准函数.

算术运算符

+ − * div mod

适用于整数值且产生整数结果(如同 Pascal).

布尔运算符

not and or

适用于布尔值并产生布尔结果(如同 Pascal).

这些运算符有它们的常规含义.

顺序关系

= <> < <= > >=

适用于相同基本类型的操作数并产生布尔结果(如同 Pascal). 对于整数来说，这些关系有它们的常规含义. 其他基本值的顺序与它们序值的顺序是一样的；如，只有当 x < y 时，才有 int(x) < int(y)，对其他关系也类似.

Edison 对这些熟知概念的唯一贡献是，在任何基本类型的值与整数之间定义了映象(反之亦然)．这种思想可根据整数的性质来定义所有基本类型的性质．

2.2.3 子界类型

Pascal 的子界类型可用来导出一些不相交的、包含在其他子界中的或甚至有覆盖的类型，如下面的例子所示．

```
var x1:1···10; x2:11···20; x3:5···9; x4:0···15
```

这就产生了一些复杂的问题，如表达式 x1 + x2 有效吗？如果有效，它的类型是什么(1…10，11…20，1…20 还是 12…30)？类型 1…10 的值是否也是类型 0…15 的值？类型 0…15 的某些值是否是类型 1…10 的值，而其他值是否又是类型 11…20 的值呢？一般说来，这些子界类型的值是否能与整数值兼容，或者说它们是否属于不同的类型呢？在 Pascal 报告中，这些问题没有一个得到回答．

Edison 中取消了子界类型，从而使这些问题不再存在．

2.2.4 实数

Edison 不包含实数类型，因为实型对于系统程序设计和文本处理不是必要的．包含实数只会对此项工作增加更多的细节，而对主要目的——可理解软件的设计毫无贡献．然而，这是一个有争议的决定，因为它使系统对工程计算不实用．

2.2.5 类型说明

在 Pascal 中，类型说明用一个名字开始，例如

```
type T1 = record x:char; y:integer end;
     T2 = array[1··10] of char
```

因此编译程序必须扫描三个符号

```
类型名＝字符号
```

才能决定一个说明是引入一个记录类型还是一个数组类型．这是编译程序中并不希望的做法，它并非只需要查找一个符号来决定程序文本中下一个句子的语法形式．

编译期间发现的错误也由于这种语法而复杂化．在出现语法错误之后，如

```
type T1 = record x:char; y:integer ned;
     T2 = array[1··10] of char
```

编译程序就不能确定记录说明在何处结束，数组说明在何处开始．所要写的字符号 end 将被解释成一个错误的域名 ned，因而跟在分号后的类型名 T2 也一样．其结果是在涉及所有使用类型 T2 的变量处出现一连串错误信息．

在 Edison 中，每个类型说明都是由一个字符号(代替一个名字)开始的，例如

```
record T1(x:char;y:int)
array T2[1···10](char)
```

在出现语法错误时，如

```
record T1(x:char; y:int)
array T2[1···10] (char)
```

编译程序将跳过右方括号，并确认字 array 是一个新的类型说明的开始。

当然，赋值和过程调用也有同样的出错问题，因为它们都是由名字开头并且可包含作为操作数的名字。但是，编译过程中一个语句的偶然删除并不象一个说明的删除那样严重，因为其他语句不能造成由名字带来的错误。

由于 Pascal 的表示法十分自然，以致程序员似乎没有理由再去寻找比它更好的工具。因此，没有经过太多的考虑，我就在 Edison 中原原本本地使用了 Pascal 关于类型说明的表示法。当我测试用 Pascal 写成的第一个 Edison 编译程序时，已经感到有些不足。但在我提出新的语法表示法之前，这种方法也曾使用了一段时间。

2.2.6 记录

一个名为 time 的记录类型的说明在 Edison 中采用如下形式：

```
record time (hour, minute, second:int)
```

这个类型的值包括三个子值(或域)，分别命名为 hour, minute, second。这些域都是整数类型。

形如

```
time(10, 55, 2)
```

的记录构造符表示类型为 time 的一个值，其中三个域的值分别为 10，55 和 2。这些域以它们说明时的次序列出(即 hour, minute 和 second)。

time 类型的一个记录变量，如

```
var now:time
```

包含三个域变量，分别记作

```
how. hour    now. minute    now. second
```

(如同 Pascal)。

尽管在 Pascal 中，一个数据类型被定义为一些值的集合，但这种语言却没有关于记录和数组值的表示法，这是 Pascal 的一个特有的疏忽。记录的值一次只能计算一个域，例如

```
now. hour: = 10; now. minute: = 55; now. second: = 2
```

在 Edison 中，这仅用一个赋值语句就能表示

```
now: = time(10, 55, 2)
```

可以测试同类型的两个记录值等或不等。如果所有相应的域均相等，则两个记录值相等，否则就不等。

一个记录值中的各个域可以有不同类型，例如

```
record job(kind:task; title:text)
```

2.2.7 数组

一个名为 lastday 的数组类型定义了一年中每月的天数，在 Edison 中可以说明如下：

```
array lastday[1···12] (int)
```

这个类型的值包括 12 个整数类型的子值(或元素)。这些元素是用范围在 1—12 的下标来标识的。