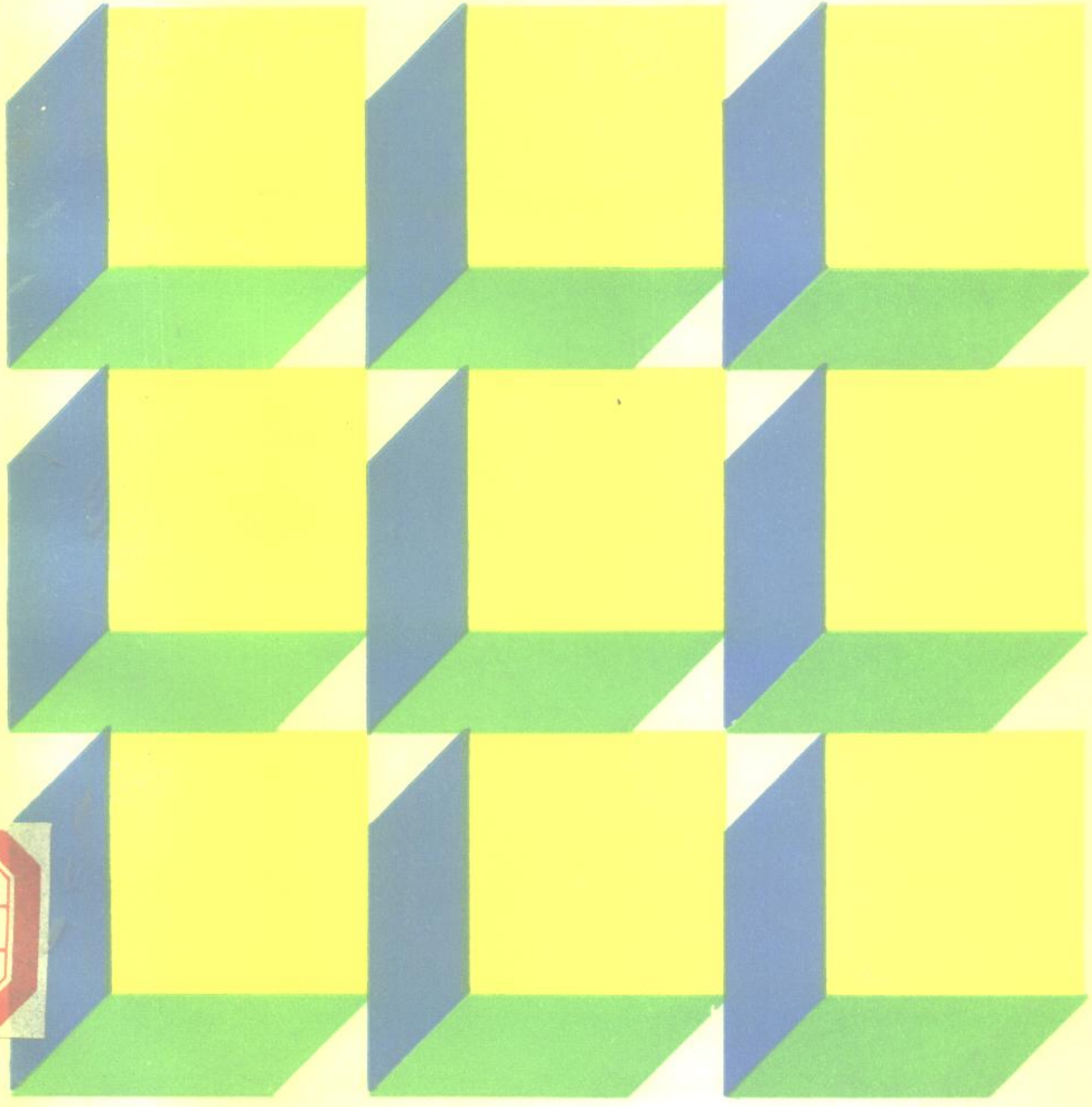
形式语义学基础

陈意云　编著



中国科学技术大学出版社

形式语义学基础

陈意云　编著

中国科学技术大学出版社
1994·合肥

(皖)新登字 08 号

形式语义学基础
陈慧云 编著
*
中国科学技术大学出版社出版发行
(安徽省合肥市金寨路 96 号,230026)
中国科学技术大学印刷厂印刷
全国新华书店经销
*
开本:787×1092/16 印张:15.5 字数:397 千
1994 年 3 月第 1 版 1994 年 3 月第 1 次印刷
印数:1—3000 册
ISBN 7-312-00533-0/TP·72 定价:7.50 元

内　容　简　介

形式语义是计算机科学和软件工程的重要研究领域之一. 本书较全面地介绍了程序设计语言形式语义方面的技术，包括指称语义、公理语义、指称语义和公理语义间的联系、代数规范的初始语义和等词逻辑程序的说明语义. 对于与指称语义有关的λ演算和论域理论，本书也作了较深入的介绍. 各章后均附有习题.

本书致力于使更多的读者接受和运用形式语义学方面的成果，它直接面向两类读者. 首先，从事实际应用工作的软件工程师会喜欢它，如果他们希望懂得更多的专业理论基础的话. 另外，本书可作为程序设计语言形式语义学的教材，适用于高年级大学生和研究生.

前　言

为了规范、设计、实现、阅读、理解、归档、欣赏、评论、测试、调试、维护、移植、改编或改进程序，需要掌握用于表示程序最后形式的符号表示法——程序设计语言.

程序设计语言虽然已经历了 40 年的研究和发展，有使用价值的新型语言设计和对语言结构宝贵的理论见解仍在源源不断地涌现.

尽管有严谨的，有时是煞费苦心的详尽描述，程序设计语言的多样性、复杂性以及往往深奥莫测的结构和概念经常使初学者困惑不解，有时也使专家感到糊涂. 过去 30 年来，计算机科学家已经开拓了一些数学的模型技术，它们提供了对程序设计语言和程序设计的深刻理解，但是，形式的语言定义好象总是被人们看作一种复杂而神秘的技术.

本书致力于使更多的读者接受和运用这方面的成果，因而不是专为理论计算机科学的读者编写的. 本书直接面向两类读者. 首先，从事实践的软件工程师会喜欢它，如果他们希望懂得更多的专业理论基础. 另外，本书可作为程序设计语言形式语义的教材，适用于高年级大学生和研究生.

读者必须具备程序设计语言的基本知识，若熟悉编译技术，将有助于理解. 本书并不使用高深的数学，所有必需的概念都定义在第 2 章，并且都是基于初等集合论.

本书的组织如下：

第 1 章讨论形式定义的概念，尤其是为什么需要这样的定义，同时也提到形式定义的局限. 这一章强调程序设计语言和数学记号间的区别. 本章还简要介绍语义描述的各种方法.

第 2 章介绍必要的数学背景：关系、部分函数、全函数和有限函数. 这些概念是简单的，任何软件工程师或大学生都可以很快掌握它们.

第 3 章致力于抽象语法概念，把它作为程序设计语言规范的基础. 该章介绍抽象文法和抽象语法表达式概念，讨论抽象语法在软件工具和语言设计方面的应用，并且给抽象语法记号以数学解释.

第 4 章介绍 λ 演算的理论，在指称语义中，λ 演算作为对付高阶函数的工具是很有用的. 另外，λ 演算是作用式语言的一个例子.

第 5 章致力于指称语义，给出一个简单而有代表性的程序设计语言的完整描述.

第 6 章说明在同一框架中如何覆盖更多的语言特征：记录、指针、输入和输出、分程序结构和例程.

第 7 章解释所需的数学概念，以说明递归定义是正确的. 熟悉指称语义的早期文献的读者将发现这种方法有点不正统. 本章的目的是给出不动点理论的一个较简单的观点. 这个讨论完全建立在集合和部分函数的简单性质之上. 它并不需要专门的底元、完全的偏序和格等概念.

第 8 章讨论不确定的程序构造和并行程序构造的指称语义.

第 9 章提出公理语义，包括 Hoare 理论和 Dijkstra 的最弱前条件. 最后一节提出基于断言的程序构造方法，讨论设计循环算法的系统策略.

第 10 章讨论指称语义和公理语义两种语义定义的一致性，说明公理语义的规则如何证明为指称理论中的定理.

第 11 章介绍代数规范的初始语义.

第 12 章介绍一种等词逻辑程序的说明语义，它是用程序的最小 Herbrand 模型作为语义的.

由于编者水平有限，书中难免有些缺点和错误，恳请广大读者批评指正.

编著者

1993 年 5 月于合肥

目　　次

1 基本概念

形式语义学是对形式语言及其句子采用形式系统方法进行语义定义的学问. 本书中提出的形式描述是数学的理论，它们用于模型和分析程序设计语言与程序的根本性质.

本章解释为什么这样的描述是有用的，以及这些描述包含的成分. 为了避免被模型的语言和用于模型它们的记号间的混淆，本章指出我们必须注意的东西. 此外，本章还概述程序设计语言语义的主要形式描述方法.

若无特别说明，本书中的“语言”就是指“程序设计语言”.

1.1 为什么需要形式描述

形式描述服务于下面这样一些目的：

(1)帮助理解语言；

(2)支持语言标准化；

(3)指导语言设计；

(4)帮助编写编译器和语言系统；

(5)支持程序验证和软件可靠性；

(6)有助于软件规范.

下面逐点讨论这些好处，然后分析一些反对形式规范的议论.

1.1.1 帮助理解语言

写形式规范的最基本好处是使我们能洞察被说明的对象. 数学的方法有助于提出问题，而这些问题在非形式的方式下可能不会被发现. 因此，写形式规范的过程是对被规范的对象理解得越来越透彻的过程.

不仅对软件系统的形式规范是这样，对语言的形式规范也是这样. 语言的一些主要概念(如数据类型、分程序结构、并行进程和递归等等)的精致的数学模型使我们能彻底理解这些概念. 熟悉形式规范技术的程序员对语言的理解比一般的程序员要透彻.

1.1.2 支持语言标准化

程序员面临的最烦恼的问题之一是程序的可移植性，即怎样编程序，使之能适应各种操作环境(计算机、终端和操作系统等)而不必过度地努力.

可移植性的关键是标准化. 虽然语言的标准化问题不足以解决移植性问题，但它显然是关键的.

Fortran, Cobol, C, Pascal 和 Ada 等语言都是标准化工作致力的对象，但是这些语言也未能

解决标准化问题.这是因为一个完整的实际语言的描述包含了许多细致的问题,它们很难在文本中用自然语言解释到足够清楚的程度.形式规范能解决这个问题.在要求绝对精确而无二义的情况下,数学技巧特别有效.

如果想用形式规范代替语言的自然语言描述,这是不切实际的.自然语言是人们的交谈用语,任何语言标准应该有一个供日常使用的自然语言版本.形式和非形式的版本起着互补的作用.当自然语言的文档出现二义或有冲突的解释时,形式规范可作为最后手段的文档.

这个相互补充还有另一个重要方面.从形式规范中得到的透彻理解使你能产生较好的自然语言描述.按这种方式,形式规范是编写或改进非形式规范的手册的理想起点.即使把这种方法只运用到语言的子集也是挺有益的.如果你发现很难解释语言的某些特征,在编写你的非形式描述之前,用本书的技术,尝试用形式方法描述它们是个好主意.

1.1.3 指导语言设计

语言的恰当设计是重要的,因为它的结构和形式一旦固定下来,将影响许多用户.不仅语言的设计者对此有浓厚的兴趣,而且很多经常为他们自己建立的系统设计用户界面的软件设计者对此也感兴趣.这样的界面是一些小语言.不管大多数程序员是否认识到,他们自己都是语言设计者.

语言设计不是一门科学,语言的质量大部分取决于设计者的天才和经验.数学上的简单性是件重要的事情.经验表明,越是难数学化地模型的概念,一旦移植为语言的特征,就越难讲授和(或)越难实现.

1.1.4 有助于编写编译器和语言系统

不管近年来有多大的进步,有多少可用的工具出现(尤其是编译器前端的工具),构造非常工程化的编译器仍需付出很大的努力.

形式描述为设计编译器,说得更一般些是语言系统,提供了稳固的基础.这里所说的语言处理系统不仅包含编译器,还包括许多支持使用高级语言的工具,如解释器、运行系统、程序检查器和调试器等.语言的形式描述可以看成该语言的抽象语言系统的规范.形式语义,尤其是指称语义,可以理解为语言的抽象编译器的高级描述.

1.1.5 支持程序验证和软件可靠性

程序的正确性和健壮性是软件工程的基本要点.为了保证这两点,研究工作者已花了很多精力来开发程序正确性证明的技术.基本的思路是,把程序和数学变换联系起来,并用数学证明技术去检查这个变换是否实现所阐明的程序的目的.

程序正确性的证明涉及三类问题:

(1)必须精确地论述每个程序或程序元素的目的.这是软件规范问题.

(2)必须开拓适当的数学理论,去进行有关程序的推理和证明程序行为的性质.

(3)因为任何实际系统的证明是冗长的并易于出错的过程,因而必须有有效的工具支持这个过程.

语言的形式描述本质上是为了达到上述第二个目标,因为它们为用这些语言写的程序进行有关的推理提供了数学基础.

1.1.6 有助于软件规范

语言的形式规范对上面三个目标的第一个目标(软件规范)有间接而重要的影响,怎样精确,无二义地描述一个软件产品的目的而不必提供一个实现呢?

本书的形式描述的目的只是在于规范语言,而不是任意的软件系统.但是,本书所用的方法也能使你对软件系统的规范得到较深刻的理解.许多基本要点和技术是一样的,这两个领域的研究紧密相关.

本书中的许多技术都可用于这两个领域,如第3章的抽象语法,第9章的公理规范技术等.

1.1.7 形式规范的局限性

形式规范不是程序设计所有弊病的灵丹妙药.反对形式规范的主要议论是说它们太困难,难学、难写和难读.尽管近年来的努力使得形式规范比以往容易理解和使用,但是,要达到生产和使用形式规范的目标,仍需很大的努力.

反对形式规范的另一种议论是说它们只适用于一些小例子,而不能成功地描述实际语言.近年来,在完整地描述 Algol 60,Pascal、PL/I 和 Ada 等语言上的成功,在很大程度上证明这种议论是错误的.但是,语言的某些特征仍很难模型到满意的程度,如并行、浮点算术和复杂的数据结构等.不过,可以这么说,这种技术已到了可用于实际语言的程度.本书虽然没有形式地描述完整的实际语言,但是它包含了如何这样做的基本技术.

1.2 程序与数学

任何语言都由它的语法和语义性质来刻画,数学记号和程序设计语言都是这样.形式地定义纯数学语言的语法和语义的确是富有成果的,如第4章要研究的 λ 演算(日常的数学实践并不需要对它的记号进行这样的形式分析).

对程序设计语言,有些东西是特殊的:它们大多数是命令式的.这个特征使得形式描述成为必要但又非常困难.为了避免潜在的混淆,在介绍形式描述前,我们必须仔细考虑它.

1.2.1 命令式的特征

程序设计的命令式(或操作式)特征由描述给机器发命令的构造来刻画.在高级语言的情况下,这机器不是实际的物理硬件,而是提供通过编译器和操作系统访问这个硬件的虚拟机器.一个命令式构造表示由这样的机器执行的命令.

最能反映语言的命令式特征的构造是**命令**.在常用的语言中,程序是命令序列,每条命令描述要完成的一组动作.这里所说的"命令"就是常用语言中的语句,在本书中用"命令"一词更为合适.

常用语言的命令包含两个明显的命令式构造:控制流和赋值.

(1)程序用显式的控制结构来精确地说明其命令的执行次序.

(2)赋值命令 $x:=e$ 不是断言一个性质,而是要求机器改变内部状态的某个东西,即对应 x 的值.赋值是最简单的命令,机器状态的改变表示赋值命令有**副作用**.

在通常的数学中,没有副作用概念.如果写 $x=e$,那是指两个对象相等,并不试图改变任何

东西. 甚至程序设计中的变量概念也和数学上的不一样.

1.2.2 引用透明性

在数学记号和程序的根本区别上，你会发现一个重要的概念：**引用透明性**. 这是数学记号所具有的等量代换性质：如果 $a=b$，那么从任何为真的性质中用 b 代换 a 后仍为真. 例如，若下列两个算术性质都是真：

$$x=2$$

$$x+y>5$$

那么，从引用透明性可以推出

$$2+y>5$$

标准的数学记号是引用透明的. 这个性质在数学推理中起着奠基的作用. 例如，代数、算术和逻辑中的公式演算都依赖于它.

然而，如果允许副作用的话，语言不具备引用透明性. 考虑下面的 Pascal 例程：

```
function f (x: integer): integer;
begin
    y:=y+1;
    f:=y+x
end
```

虽然 Pascal 的术语把 f 称为“函数”，但它不是数学意义上的函数. 因为 f 每次被调用时，不仅仅从其变元产生一个结果，还通过改变 y（这里假定 y 是一个全局整型变量）的值而改变程序的状态.

结果是，依赖于这个“函数”的表达式将不是引用透明的. 假定在某一点 $y=z=0$，其中 z 是调用例程的变量. $f(z)=1$ 是指调用 $f(z)$ 将返回 1. 引用透明性将蕴涵下面两个表达式：

$$f(z)+f(z) \quad 和 \quad 1+f(z)$$

相等，但是情况不是这样，因为 f 的每次调用增大 y，第一个表达式的值是 3，而第二个表达式的值是 2.

别名也会破坏引用透明性. 每当一对象可通过多个名字访问时，别名就出现. 参数的引用调用是引起别名的机制之一，在 Pascal 中，如果把 f 的参数声明为

```
var x: integer
```

（表示子程序可以直接访问调用的实在参数），那么当 y 和 z 的值都是 0 时，调用 $f(y)$ 和 $f(z)$ 有不同的结果. 前一个返回 2，后一个返回 1.

当然，允许在全局变量上有副作用的函数，或者允许在全局变量和子程序参数间有别名，不是一种好的程序设计实践. 但是，只要语言支持变量和赋值语句，这个问题就会出现. 如果允许对同一变量 x 多次赋值，那么就不能使用在程序的某一执行点的事实 $x=a$，从 a 的性质去推导在任何其它点的 x 的性质. 全局变量上的别名和副作用只是使问题变得更严重.

甚至一些表面上看起来无害的构造，如象 C 语言的 *getint* 这样的输入函数，也有这样的问题. 例如，表达式

$$2 * getint() \quad 和 \quad getint()+getint()$$

通常是有区别的. 第一个表示在一给定的文件上读入下一个整数，并把它乘以 2；而第二个表示从这个文件上读入下两个整数，并把它们相加. 函数 *getint* 产生副作用，因为它修改了这个计

算的全局状态，而该状态是包含了输入文件的读位置的.

所以，命令式构造危及对数学对象推理的许多基本技术. 在语言理论上的许多工作可以看成试图根据意义明确的数学构造来解释语言的引用不透明的特征，给它们以数学的体面. 例如，指称语义中基于函数复合的数学概念提供命令序列的模型. 在公理语义中，赋值的影响通过代换的数学概念来描述.

通过对语言特征提供基于标准数学概念的描述，语言的理论使得我们能够使用精确和严格的技术去操作程序和进行有关的推理，这些技术类似于那些用于算术公式运算的技术. 这个活动需要数学的引用透明性.

1.2.3 非命令式的语言

不要从前面的讨论得出误解，认为副作用、赋值、显式的执行次序和其它令人反感的特征是语言必须有的. 其实，没有人相信这些不好的东西是必须的.

基于上面的原因(也还有其它原因)，人们一直致力于数学上比较清晰的语言，没有引用不透明的特征. 这样的语言称为**函数式的**或**作用式的**，大多数这样的语言以函数作用而不是赋值作为计算值的基本技术.

这类语言的例子有 FP 和 Miranda. 有关的研究包括 Lucid 这样的数据流语言和 Prolog 这样的逻辑语言.

用这样的语言编程确实大大接近传统的数学. 例如，Miranda 程序设计已被授予“指称式程序设计”的称号，因为它的表示法和精神十分接近第 5 章介绍的形式描述技术.

如果程序设计都用这样的语言完成，形式描述的需要性也就不会象现在这样迫切了. 然而，在计算机技术的目前状况下，由于效率上的原因，这些语言还不能广为传播供实际使用. 说得根本一些，这些语言是否会通用还值得怀疑，因为它们往往忽略了一个困难而又不可避免的程序设计的特征，即软件工程和纯数学间的本质区别，因而需要改编问题求解的表述，使得它们能够在物理硬件上高效地运行.

研究一下 Lisp 和 Prolog 的实际使用是有益的. 它们是含较少命令特征而又得到足够广泛使用的仅有例子. 但是，真正实际有用的 Lisp 或 Prolog 程序(而不是教科书上的例子)似乎又经常使用命令式特征，如 Lisp 的 *PROG*，*SETQ* 和 *RPLACA*(它们对应显式的执行次序、赋值和副作用)，以及 Prolog 的 *cut* 机制(非正规的执行次序). 虽然这个现象还需要进一步调查，但是我们怀疑，性能上的限制不是命令式构造仍然存在的唯一原因.

不管你是否相信函数语言和逻辑语言将广为使用，都不能改变短期的观点：当今大多数程序设计仍含有命令式特征. 为了明白它的基础是什么，需要一个依赖于引用透明的表示法的数学分析.

1.2.4 语言和元语言

我们已经说过，“语言”一词仅用于考虑其形式语义的程序设计语言，而用于描述语义的形式体系叫做**元语言**.

虽然数学和语言间的区别是明显的，但是，用于形式规范的元语言也倾向于使用许多最初是用于程序设计的记号. 因此必须注意区别元语言和它描述的语言.

举个简单的例子. 在元语言中，通过分情形分析来定义函数是很有用的(函数是数学对象，属于这个元语言)，其形式类似于：

$$
\begin{aligned}
&f(p)=expr_1(p) \qquad \text{如果 } p \text{ 的类型是 } type_1\\
&f(p)=expr_2(p) \qquad \text{如果 } p \text{ 的类型是 } type_2\\
&\cdots\\
&f(p)=expr_n(p) \qquad \text{如果 } p \text{ 的类型是 } type_n
\end{aligned}
$$

其中，变元有几种不同的情况，$expr_i$ 给出第 i 种情况下的函数值. 如果元语言的记号允许把此函数定义写成

$$
\begin{aligned}
&f(p)=\\
&\quad \textbf{case } type \textit{ of } p \textbf{ of}\\
&\qquad type_1 \rightarrow expr_1(p) \mid\\
&\qquad type_2 \rightarrow expr_2(p) \mid\\
&\qquad \cdots \mid\\
&\qquad type_n \rightarrow expr_n(p)\\
&\quad \textbf{end}
\end{aligned}
$$

的形式，则更漂亮.

虽然这个分情形表示法直接取自程序设计语言，但它有完整的数学意义(第 3 章将给出它的形式定义). 当元语言使用这种表达式时，不要和被描述语言中可能有的分情形命令混淆. 例如，Pascal 和 Ada 有这样的构造，其语法和上面的类似.

本书所用的元语言叫做 Meta，它确实使用了这样的分情形表达式和其它一些由程序设计语言启发的构造. 必须记住，它们是数学的记号，不要与程序设计中的类似东西混淆.

本书中用的描述语义的方法应该有助于避免混淆. 当需要引用某个程序片断，如 Pascal 的命令

```
case p of
    1,3,5:x:=a+1;
    2,4,6:x:=a-1
end
```

时，形式描述不是引用这种标准的表示法，而是相当不同的形式，叫做“抽象语法”，将在第 3 章介绍，并嵌在 Meta 元语言中.

1.3 语义描述的主要方法

本节介绍语言的语义描述的主要方法.

1.3.1 属性方法

属性规范通过拓广语法描述的文法来描述语义，这个拓广是增加覆盖语义的成分，虽然具体语法也可用作拓广的基础，但是抽象语法明显地更加适用.

把语义加到语法描述上的过程通常称为**修饰**. 文法是用一组产生式来定义语言的语法. 可以通过同时修饰文法符号和产生式，把文法拓广成语法和语义的完整描述.

(1)为了修饰文法符号，可以定义描述其实例的语义性质的**属性**.

(2)为了修饰产生式，可以定义若干**规则**，它们用来表达产生式左部的文法符号的实例的属性和产生式右部的文法符号的实例的属性间的联系.

任何特定对象(文法符号的实例)的语义由计算这个对象的属性得到.

属性文法技术是简单的和有效的,它非常适用于编译器的设计,许多编译器的输入是属性文法.但是,属性文法的算法味道太浓,而且该方法也不够完整,它在很大程度上依赖于属性和规则的选择.有关属性文法的介绍,可以参考编译原理和技术方面的专著.

1.3.2 翻译语义

翻译语义用翻译方案表示语言的语义.对该语言的任何程序,翻译语义产生用较简单的和易理解的语言编写的一个程序.

翻译语义的好处是可以深刻理解该语言,并且可能有助于编译器的编写.但是,该方法缺乏抽象性和一般性.翻译规范太依赖于目标语言的选择,以致难以满足语言标准化和程序验证等形式规范的其它方面的要求.

仔细考虑一下,翻译语义有助于编译器的编写这个优点也是有问题的,因为对不同的目标机器,你需要重新构造一个翻译定义.

这种方法还受到一个根本的限制.该方法是根据较简单的语言来定义一个语言,一个自然的问题是:这个目标语言是怎样定义的?一个可接受的答案要求目标语言的语义是不容怀疑地定义的.这可由指称方法完成.1.3.4 节的指称方法也是一种翻译方法,但是翻译的目标是数学对象的集合而不是程序设计语言.

考虑这些局限性的结论是,翻译方法并不适合作为语义描述的一般方法.但是它也有两方面的用处.

其一是,它可以作为编译器设计的形式工具,你可以通过写出一组翻译函数来精确地说明编译器必须完成的翻译.这在翻译的目标语言是 Fortran 或 C 这样的移植性较好的语言时比较有用.

其二是,翻译语义可基于其它方法来简化语言的规范.大多数语言包含一些概念上冗余的构造,它们可以根据本语言的其它构造来定义.例如 Pascal,C,Ada 以及其它语言的 **repeat…until…**和 **for…**循环只是表示上的方便,它们没有贡献任何新的概念,因为它们都可以用 **while…** 循环来表示.这样,出于语义规范的目的,可以把语言分成两部分.核心部分用一种高级的方法规范,而扩展部分的语义通过翻译方法用核心构造表示.这种二级规范比把所有特征堆在一起的规范容易理解.

1.3.3 操作语义

如果把语言的翻译语义看成它的编译器的话,那么它的操作语义就象它的一个解释器.在操作语义的情况下,语言语义的表达是通过给出能确定该语言的任何程序的执行效果的一种机制,这样的机制是一个"抽象自动机":一种能够形式地执行程序的形式设备.操作语义的刻画是通过描述语言的各种构造在解释自动机上的执行效果来实现的.解释自动机的选择是多种多样的,有的相当具体,接近于可执行的语言,而有的相当数学化.

由于操作语义给出了语言的具体和直观的描述,并且这种描述相当接近于实际程序,因此,程序员对此有兴趣.另外,有了操作语义,解释器的设计变得容易,因而便于对程序实例执行这样的描述.这使得操作语义在测试新语言或新语言特征方面富有吸引力,因为它们的效果可以在编译器构造之前,通过实际执行语义描述来模拟.

但是,操作语义失去了规范的一个本质特征:独立于实现.操作描述所规范的是执行程序

的一种特定方式，和它在翻译语义中的对应物一样，操作语义冒着过分约束的危险，因此它对1.1节所说的语义的各种运用几乎没有帮助.本书不打算介绍操作语义.

1.3.4 指称语义

指称语义可以看成是翻译语义的衍变.使用这种方法，我们可以由翻译方案表达语言的语义，该方案把每个程序和一个**指称**(含义)联系起来，不同的是翻译的结果.在翻译语义中，程序的含义是另一个程序；而在指称语义中，它是数学对象.

语言的指称描述由一组含义函数 M 给出.每个含义函数的形式是

$$M_T : T \to D_T,$$

其中 T 是程序或命令等构造.大多数这样的函数是高阶函数，它们产生函数作为结果.指称集合 D_T 可能随构造 T 不同而不同，它们叫做**语义论域**，而语法构造叫做**语法论域**.

第5章到第8章将详细讨论指称语义.

1.3.5 公理语义

公理方法对语言的定义取另外一种观点：语言的公理规范是该语言的一个数学理论，也就是说，是一个系统.在这个系统中，你可以用该语言表示与程序有关的陈述，证明或反驳这样的陈述.

这些陈述是叫做**公式**的形式表达式.公式可以是真或假的，真的公式叫做**定理**.一个理论的目标是定义哪些公式是定理.

一个数学理论由三部分组成：

(1)**语法规则**.它用来确定什么是合式公式；

(2)**公理**.它是不加证明地接受的基本定理；

(3)**推理规则**.它是从已确立的定理演绎新定理的机理.

这些成分使得我们有可能证明某些公式是定理.一个证明是从公理开始，由若干次使用推理规则组成的.

在第9和第10章，我们将详细讨论公理语义以及公理语义和指称语义的联系.

可以认为，公理方法比指称方法更加抽象和实际.它更抽象是因为它并不试图弄清程序的"含义"，而是弄清从程序能证明什么.这使得它从软件工程的观点看显得更实际，因为对于实际的软件构造，我们对程序的形式模型的兴趣通常小于从程序所能得知的重要事实的兴趣，如程度是否终止和它计算什么样的值.

1.3.6 代数规范的初始语义

数据类型是程序设计和软件系统规范的基本概念.数据论域及其上的运算合起来就称为数据类型.从这种观点，可以说数据类型是代数，但代数不一定是数据类型，因为代数可以有不可数的基集.

按照算法和软件系统设计的观点，人们的兴趣主要不是在数据类型的具体表示上，而是在于更抽象级别上的它的性质.这就导致了抽象数据类型的概念.

在第11章，我们将介绍描述抽象数据类型和软件系统的一种方法，称为代数规范.这种代数规范是基于描述抽象数据类型的运算之间的联系来刻画该类型的.代数规范的语义是它的一类同构的模型(每个模型是一个代数)，叫做初始模型，因而这种语义也叫做初始语义.