程序语言的形式规范概论

〔美〕FRANK G·PAGAN 著

周之英 译 孙永强 校

清华大学出版社

程序语言的形式规范概论

〔美〕 Frank G.Pagan
周之英 译
孙永强 校

清华大学出版社

内 容 简 介

本书较全面地介绍了程序设计语言形式规范方面的技术。包括以语法定义为主的 BNF，属性文法，二级文 法，抽象语法等。并讨论了以语义定义为主的运算性方法，指称方法和公理方法。此外，还 介绍了把程序设计语言作为元语言的形式规范方法。全书着重介绍各种方法的基本概念、特点及应 用 范围。因此，它是这一领域很好的入门书，可供高等院校有关计算机软件专业学生或研究生的教科 书 或教学参考书，也可供从事计算机理论工作或实际工作的同志学习参考。

TP345/122

程序语言的形式规范概论

〔美〕 Frank G. Pagan

周之英 译

孙永强 校

☆

清华大学出版社出版

（北京 清华园）

北京昌平环球科技印刷厂印刷

新华书店北京发行所发行 各地新华书店经售

☆

1986年9月第1版 1986年9月第1次印刷

开本：787×1092 1/16 印张：11 字数：274千字

印数：0001—8000

书号：15235·230 定价：1.85元

译 者 的 话

不久前，程序语言的形式定义似乎还只是局限于计算机理论界的一种深奥技术。随着“软件工程”的兴起，越来越多的人注意到在实际工作中应用这方面研究成果的可能性。广大的软件研制人员和计算机用户迫切要求有一本较好的入门书，一方面要能帮助他们迅速地了解这一领域的全面技术背景及可能的应用方向。另一方面又可作为深入研究的出发点。F.G.PAGAN 所著《FORMAL SPECIFICATION OF PROGRAMMING LANGUAGES: A Panoramic Primer》可以说就是这样一本书。这本书除指称方法方面的介绍略欠不足外，全面地、较完整地介绍了各种形式规范技术。书中收集了丰富的文献资料，可供进一步深入学习。

由于计算机新词汇术语层出不穷，国内尚无统一译法，我们尽可能遵照通用的译法，并在许多术语的译文后附原文以便对照。在原文中发现的错误已作了修改，并加以译注。

周巢尘和蒋国南同志阅读了全文，并提出了宝贵的意见，在此谨向他们表示感谢。

限于水平，肯定有不少错误缺点，恳请读者批评指正。

序　　言

我希望通过这本书逐步地、较全面地给读者介绍一个重要的技术领域，介绍为计算机程序语言的语法和语义的形式规范所需要的元语言（从 BNF 到公理语义学）。形式语言定义好像总是被人们看作一种复杂而神秘的技术；人们往往认为只有寻找，并阅读了极大量很分散的专门著作，才可能得到形式语义学的一般知识。本书把具有各种特点的定义方法的基本材料和应用例子都集中在一起。为了尽量减少复杂的数学论证及避免过于拘泥于形式，我们虽然很详细地叙述了各种元语言，但这种叙述是非形式化的。我们在定义程序语言时强调怎样使用元语言，而不是强调他们的理论基础或数学背景。现在已有好几本书非形式地研究并选择一些程序语言来说明算法的形式规范（即程序的编码）。和这类书一样，本书非形式地研究并选择一些元语言来说明程序语言的形式规范。

对高级程序语言感兴趣的大多数计算机科学家将会发现本书汇集的材料十分有用。我们假定读者已具有程序设计的一般知识，熟悉两种以上高级语言及与计算机科学有关的内容，如树及其他数据结构和递归概念。还需要了解离散数学中各种课题的一些知识，如集合论、函数、逻辑和近世代数。本书可以作为研究生或大学高年级学生学习程序语言的结构、设计和理论方面的教科书。

我们在全书中用两个小型的专门设计的语言作为各种规范技术实例研究的对象，以此作为基本评述工具。其中一个语言（“Pam”）有整数算术运算和流程控制（条件句及循环结构）的功能；而另一个（“Eva”）有块结构，递归过程和结构数据类型（字符串）。所有这些特点都和众所周知的 Pascal 及 PL/I 等语言中的某些特点相类似。两个小型语言结合在一起提供了解释如何使用各种元语言的良好基础。事实上，为避免实例研究变得过长或过分复杂，每个语言本身必然是又小、又弱而不能实用。

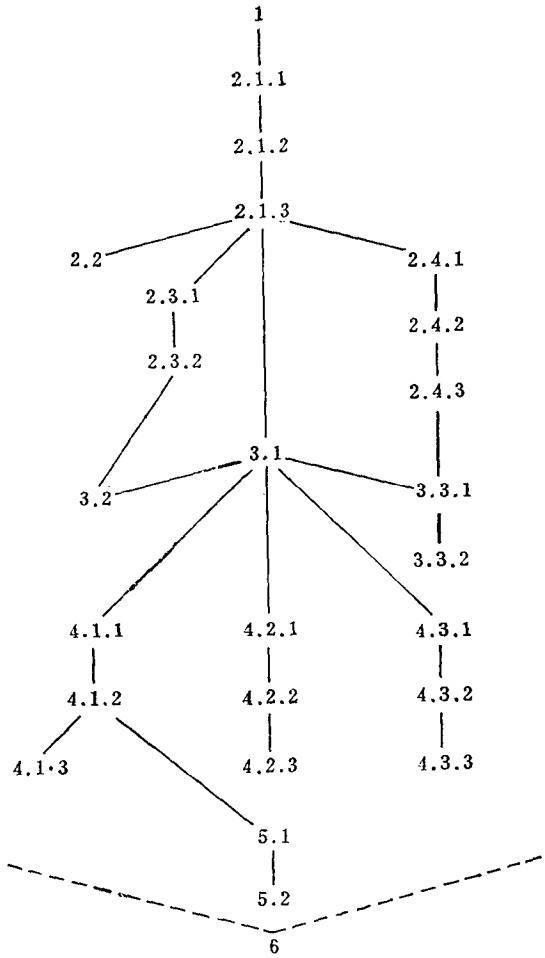
每一节结尾处简短地列出指导进一步学习的资料，尽可能只引用读者容易找到的刊物。在大多数情况下，读者还可从那些列出的书目中找到更为完整的参考书目。

本书内容的前后次序不甚重要，我所选择的结构只是反映在形式语言规范问题这一复杂领域中我个人的观点。第二章涉及语法的形式规范，包括程序文本的一切上下文有关特性，也包括 BNF 及其变型(2.1，2.2节），属性文法（2.3节），和二级文法（2.4节）。我们在BNF 一节中也介绍了语言 Pam 和 Eva。第四章涉及形式语义学，介绍运算性方法和维也纳定义语言（4.1节），指称方法（4.2节）和公理方法（4.3节）。第三章完成形式语法学向形式语义学的过渡，也包括了利用文法来说明语义的技术。第五章讨论了把程序语言作为元语言的概念，很可能元语言本身恰好就是要定义的语言。

我想让读者有尽可能多的灵活性来选择阅读章节的次序。下图指出了阅读某一章节的条件，在你要读某一节前，必须先读完有直线与这节相连的较早章节。因而，对用二级文法作语义规范感兴趣的人可按 1，2.1.1，2.1.2，2.1.3，2.4.1，2.4.2，2.4.3，3.1，3.3.1，3.3.2这样的次序来阅读，也可以选择读完 2.1.3 后立即读 3.1 节。根据不同的兴趣或不同的课程要求也可以沿其他路径进行阅读。结束语（第 6 章）中提到所有其他章节的内容，但

此为试读，需要完整PDF请访问：www.ertongbook.com

即使没有读完全书，也可以阅读这一章。



阅读课文各章节的必要条件结构图

目　　录

第一章 引 言

程序设计语言确实是人类智慧的结晶。它帮助人们准备计算机程序，表达并交流计算机算法。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更令人拍案叫绝。经过三十年来努力研究和发展，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。因而，计算机科学家采用有丰富含义的术语——‘语言’，而不用术语‘记号’来描述这些工具是很合适的，他们还借用了自然语言研究中语言学家采用的一些概念和术语。

当涉及语言描述时，基本上可分类为语法（syntax）、语义(semantics)和语用(Pragmatics)三个方面。粗略地讲，语法涉及语言外表形式有关的问题；语义是指所包含的意义；语用就是牵涉到的实际应用。这些说法非常笼统而有些含糊不清，因为根据对这些概念的不同解释，就有不同方法来描述语言。特别对语义更是如此，语义肯定有不止一种意义的‘意义’。

拿普遍的赋值语句

$$a := b + \sin(x) + 2$$

可以算作赋值语句的一个特例。我们用下列陈述概括了赋值语句的非形式描述。

·语法：赋值语句由一个变量，后面跟一个符号‘:=’，再在后面跟一个表达式组成。

·语义：赋值语句的执行过程是，先对语句右部表达式求值，再把得到的结果值与语句左部的变量相结合，并取代了与此变量结合过的任何先前值。

·语用：赋值语句可以用来计算和保存不变动的表达式的值；（这个值在程序中不止一个地方需要用到）或可以通过某一变量的先前值来更新它的值；或……。

当然，我们还必须补充说明“变量”及“表达式”。下面还有另二种办法来解释语义：

·语义：若在赋值语句执行后，某逻辑断言（assertion）成立，则在赋值语句执行前对断言中变量的所有自由出现都用表达式来置换，该断言仍保持成立。

·语义：赋值语句代表了把内存状态映射到内存状态的一种函数，即由其表达式代表的函数。把这种函数施加于某特定内存状态而产生的结果内存状态中，除变量值将由函数对应的值所置换外，其他内存状态保持不变。

大多数语言描述都是非形式的，用叙述性方式而不是用严格的记号方法来表示。这样做对某些问题是足够了。毕竟形式记号是深奥的（有些更抽象难懂）。而大家都能懂得自然语言。例如，在教程序设计引论时，可能根本不需要明确区分语法、语义和语用。但当需要来写语言的定义性规范或语言的标准定义时，大家都知道非形式方法是有缺陷的，主要问题是缺乏清晰性和准确性：经验告诉我们，这时我们要做到完整地、严格地、无二义性地表达程序设计语言的内部关系及其细节，而对这一点来说自然语言过于含糊而不准确。我们越想搞得严格，术语越为冗长罗唆。规范精确性是好的（至少在一个方面）但用叙述性说明是不能实现的。甚至在上述简短的例子中，我们只使定义稍许精确一点，就可看出这些缺点了。

如果还有什么希望来写出一个具有完整性、一致性、精确性，无二义性、简洁性、可读

性和有用的高质量的语言规范的话，我们只有考虑形式方法。当然，必须记住形式方法也许不可能同时达到上述所有目标。在开发及利用通用的形式方法去说明程序设计语言语法及语义方面已经做了大量工作，还有更大量工作留待今后去做。至今还搞不清究竟是否能够或是否应该对语用进行形式化，所以从现在起我们将限于讨论语法及语义问题。

我们也许要问语言规范的用途是什么，换句话说，为什么我们想方设法精确地定义程序语言呢？差不多每一本以形式定义为主题的书或文章都会讨论这一基本问题。我们的观点和其他文献中的观点很接近。下面列出几条形式规范的目的及其优点：

1. 程序设计语言标准化：很早就认识到要对“老”的语言定义标准版本及标准扩充。一些广泛使用的语言如 Fortran，Cobol 及 PL/I 原先大部分用非形式方式定义的。现在已成立了专门的委员会，花了很多时间及精力对它们进行标准化工作。如果很好地设计并真正实现了标准化，就会制止胡乱地编造定义有毛病或不相容的方言，这种方言往往是由不同的实现及用不同的注解来说明语言所造成的。标准化使语言成为交流的媒介和编写可移植程序的工具，进而增加语言的价值。现在已经公认，只有在合理的技术基础上，标准化的努力才能真正成功。这也意味着标准化工作在一定程度上必须运用形式规范技术。

2. 为用户提供参考材料：程序设计语言的用户需要有一套明确的文件资料。在这些文件资料中，用户能查到语言设施的合法规定及语言含义等全部详细而准确的材料。形式规范技术在原则上是提供这种文件资料的最好方法。

3. 程序证明：对认为只靠调整及测试程序是不够的观点，就往往要求用推理证明来验证程序有关的一些性质，特别是程序的正确性。如果这个证明在数学上是严格的，那么此证明所涉及的语言结构的性质也必须是严格地形式化的。从而可用计算工具帮助检查程序的正确性，甚至还可能自动产生证明过程。语言形式定义另一个略为不同的目的是要提出用系统性的方法来构造程序，以保证程序与它的规范相符合，这样，正确性证明实际上寓于构造过程之中。

4. 为程序编制人员提供参考材料：程序语言的编译程序或解释程序的编制人员必须通晓语言的全部性能。只要一个人已经编制过那怕一个很小的“玩具”语言，他就会同意这一观点。如果这个语言不是严格地、无二义性地定义，不同编制人员的理解就会不一样，而最终将生产出不相容的处理程序[1]。

5. 证明语言的实现：为了证明程序的正确性，必须把它的功能要求严格地形式化。语言实现就是具有所要实现的语言规范所指定功能要求的一个程序。因此，语言规范的形式化是保证语言实现正确性的一个必要条件。

6. 语言实现自动化：使用了形式规范技术提供了自动或半自动地构造编译程序或解释程序的可能性。通常要利用一个特殊的程序，把语言规范作为输入，产生的输出就是语言的处理或部分处理程序。从（较简单的）形式语法规范，自动生成语法分析程序就是利用上述思想而得到成功的一个领域。另一方面，某些形式定义方法得到的一组语言规范就已经是一个真正的语言实现或部分实现。如果这个处理程序对实际使用来说效率太低，问题就转化成如何把它变换成（或优化）一个高效率的语言处理程序。

7. 改进语言设计：“好”的程序语言设计要求同时达到人们所期望的一切质量标准，

注1：假定编制人员本身不承担责任去编制语言的‘更好’的变型。

如自然性，概念清晰性，功能强有力及有用的各种特点。在很多人看来，这种“好”的语言设计目前还难于达到。语言结构的形式处理帮助我们洞察程序语言的基本性质，而用其他方法则做不到。形式方法能揭露表面上看不到的不合理因素；发现那些表面上不同的语言间内在相似点及表面上相似语言间的不同点。还能查出为什么把某些功能组合在一起会极大地增加问题复杂程度；或不能协调地相互作用；为什么程序中所使用的某些特征是难于证明是正确的；等等。在设计阶段使用形式工具将有力地帮助人们得到较好的程序设计语言。

形式定义以不同方式影响着许多不同的人，包括语言设计者、实现语言的程序编制人员或使用者。我们应该记住，设计单一的方法同时满足全部要求是不太可能的，也根本没有必要这样做。现存的方法是否已经令人满意地解决了所有这些要求了呢？这个问题是有争论的。但不管怎么说，即使研究出更好的方法，本书叙述的方法中大部分基本概念仍会有用。

任一种语言描述或语言规范一定要使用某种元语言，就是用来描写程序语言的某种语言（或记号）。用作非形式描述的元语言就是一些自然语言，可能再附加少许数学公式。形式描述需要形式元语言。在本书中常把被定义的语言称为主语言（subject language）。（在其他文献中经常采用术语“目标语言”——object language，用这种术语的缺点是可能与编译程序的目标语言相混淆）。

本书的其余部分讨论了形式元语言的非形式解释及在定义程序设计语言中的应用事例。要详细讨论形式定义如何用于（或不能用于）上述各种可能的应用领域则大大超出本书范围。大多数情况下，可以在每章末的参考书目中找到进一步的资料。

深入学习的资料

要列出目前已提到的问题的全部资料来源是不实际的，也是无用的。这样，会把文献目录中大部分（甚至可能更多）的书刊文章都列进去。Marcotty 等（1976）所写的长篇综述论文中序言部分（p.192—194）及结论部分（p.272—274）所包含的内容和本章最接近。在那篇文章中作者对名为 ASPLE 的小语言阐述了各种完整的形式定义。

此为试读，需要完整PDF请访问：www.ertongbook.com

第二章　形式语法

从根本上来说，语法形式化比语义形式化要简单。我们研究一下计算机发展史，可以看到很早就开展了语法形式化的研究工作，并很快取得不少进展。而语义形式化方面的工作却开始得晚些，且进展很慢。象早期的 Algol 60 等许多语言，就是用形式的语法说明和非形式的语义说明来进行定义的。实际上，也许把其中的一些语义说明看作是语法问题会更合理（即语法也常常只是部分形式化）。到目前为止，不论从满足形式语言的目的或从可能的应用领域来看，形式语言规范比形式语义规范都成功得多。

本章讨论了形式语法规范方面大多数杰出的方法。考虑到大多数读者已经熟悉我们要介绍的第一种元语言（BNF），所以顺便在这一节里介绍两个微型语言，我们给它起个名字叫 Pam 和 Eva。这本书自始至终将把这两个语言作为例子来解释各种形式方法。第二节说明 BNF 的一些简单而常用的变种及其推广形式。第三节涉及属性文法。第四节与二级文法有关。这两种文法比早期的形式化方法要强有力得多。

2.1　巴科斯—瑙尔范式

2.1.1　BNF 元语言

我们要讨论的最早期的形式化方法巴科斯—瑙尔范（BACKUS—NAUR）式或 BNF。BNF 是大家所熟知且一直广泛地使用着的。BNF 简单、自然，有相当好的表达能力，因此值得花时间搞清楚它。本质上，BNF 是说明生成文法（generative grammar）的一种记号系统。产生式文法定义了用主语言写的程序的全部可能的符号串集合及其语法结构。BNF 和自然语言的某些性质，以及形式语言理论有密切关系。用这些领域的术语来说，BNF 定义的语言类就是上下文无关语言类或二型语言类。用 BNF 可以表达的文法构成上下文无关语法类或二型文法类。

BNF 文法由一组产生规则组成。每个产生规则有一个左部及一个右部，二者用元符号‘::=’分隔开来。左部由一个非终结符组成。非终结符是用‘〈’和‘〉’括起来的一个或多个字符的字符串，代表主语言的结构类型或语法范畴的名字。符号‘::=’可以读作‘由…组成’或‘被定义为…’，也就是说产生规则定义了产生式左部的非终结符。文法中每一个非终结符都应该对应一个产生规则。产生规则的右部由一个或多个不同的说明项目组成，用元符号‘|’分隔（读作‘或’）。每个说明项目是一个非终结符和（或）终结符序列，其中终结符是主语言的标记符号（token 即字符或不可分割的字符组）。例如，产生规则

〈条件句〉::= **if**〈比较式〉**then**〈序列〉**fi** | **if**〈比较式〉**then**〈序列〉**else**〈序列〉**fi**

表示〈条件句〉是由符号 **if**，后面跟一个〈比较式〉，再跟一个符号 **then**，再跟一个〈序列〉，最后以符号 **fi** 结束；或者由符号 **if**，后面跟一个〈比较式〉，再跟符号 **then**，再跟着一个〈序列〉，后面跟符号 **else**，再跟一个〈序列〉，最后以 **fi** 结束。其中非终结符〈比较式〉，〈序

列〉必须由其他规则来定义；**if**，**then**，**else** 和 **fi** 是终结符。

以上的〈条件句〉规则和〈条件句〉规则中所用到的非终结符规则，以及涉及的非终结符规则中用到的非终结符规则等等，一起确定了全体终结符字符串的集合（终结符串）。每个终结符串是这类结构的一个特例。文法的非终结符〈程序〉是所有非终结符中“最上层”的符号，或称作“识别”符号。这个符号确定了全体主语言。如果这个识别符号不出现在任何产生规则的右部，则我们能形式地唯一地识别这个符号。假如〈序列〉是个识别符号，但又出现在某些产生规则的右部，这时我们可以补充一条规则。

〈程序〉::=〈序列〉

这样〈序列〉就是可以唯一地识别的识别符号了。实际的程序语言的文法也许会有几百条规则，例如，Algol 60 的定义有 117 条规则。

任何一种程序语言都允许无限多个可能的程序结构。文法中使用了递归就能使有限个（或少量）产生规则生成无限多个终结符串。例如，规则

〈序列〉::=〈语句〉|〈序列〉；〈语句〉

是递归的，因为右部用了正在定义的非终结符。按规则右部第一个说明项，〈序列〉可以由单个〈语句〉组成。按第二个说明项，〈序列〉也可以由以分号（终结符）隔开的二个语句组成。因此，再根据第二个说明项，〈序列〉又可以由用二个分号分隔开的三个〈语句〉组成。根据归纳法，可以看到，事实上〈序列〉可以由无限多的〈语句〉组成，各个〈语句〉由分号分隔。此时，规则

〈序列〉::=〈语句〉|〈语句〉；〈序列〉

也可以同样做到这一点。我们称以上规则是对〈序列〉的右递归，而第一个规则称左递归。

我们能用语法树来描述由特定文法产生的特定终结符串的语法结构（有时称派生树或分析树）。如果我们的文法包括下述规则

〈条件句〉::=**if**〈比较式〉**then**〈序列〉**fi**|…

〈序列〉::=〈语句〉|〈序列〉；〈语句〉

则图 2.1 是语法树的一部分。一个代表完整程序的语法树中，用终结符标记所有叶子结点，用非终结符标记所有其他结点，用文法的识别符号标记根结点。对某一非终结符N的结点，其直接后代从左到右所标记的符号构成N的产生规则中某一种定义。因此，图 2.1 中的结构

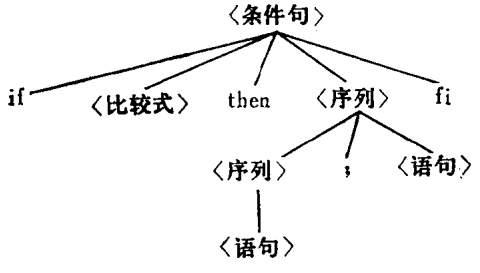


图 2.1

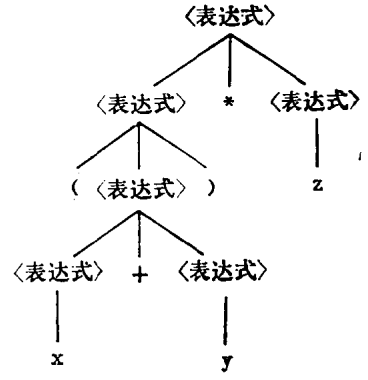


图 2.2

就是某一个标有〈条件句〉规则的非终结符结点以下的子树部分。从左到右扫描〈条件句〉子树的叶子，得到的终结符串将构成〈条件句〉的一个特例。而扫描某一完整树的叶子所得到的终结符串就形成〈程序〉的一个特例。

现在，让我们来研究一下只有操作符‘+’和‘*’、括号以及基本操作数x、y和z构成的一类非常简单的算术表达式。大家也许首先会想到用下述产生规则

〈表达式〉::=x|y|z|（〈表达式〉）|〈表达式〉+〈表达式〉|〈表达式〉*〈表达式〉

来定义所要求的表达式语法。由此，对终结符串（x+y)*z，我们可以得到图2.2中的唯一语法树。但正如图2.3所示，终结符串x+y*z却有二个可能的语法树。人们把有多种语法树的一种终结符串称为有二义性。而允许有这种情况存在的文法也称为二义性文法。

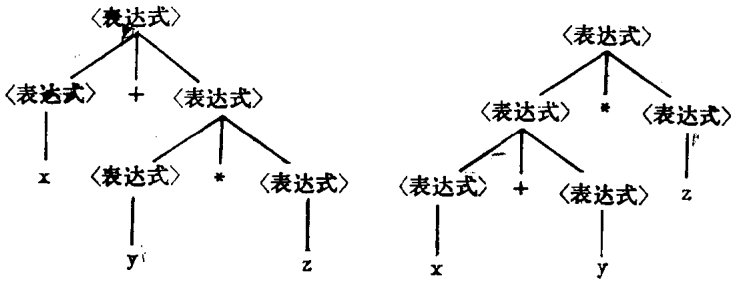


图 2.3

有时二义性是无害的，不影响结构的意义。但在上述例子中，第一种结构意味着先作y与z的乘法，而在第二种结构中却先作x与y的加法，因此二种结构的结果往往不相同。

要一般地确定任一给定文法是否有二义性理论上是不可解的。但实际上，通常可以设法避免二义性。特别当我们只限于讨论上下文无关文法时更是如此。我们说，如果左递归和右递归都对应于同一非终结符，则文法是有二义性的。这是一条确定文法是否有二义性的简单而有用的规则。上述例子就属这种情况。我们可以引入另一个非终结符来消除二义性：

〈表达式〉::=〈元素〉|〈表达式〉+〈元素〉|〈表达式〉*〈元素〉

〈元素〉::=x|y|z|（〈表达式〉）

现在，图2.4代表了x+y*z的唯一结构。〈表达式〉的左递归规则意味着从左到右计算（括号的影响除外）。右递归规则

〈表达式〉::=〈元素〉|〈元素〉+〈表达式〉|
〈元素〉*〈表达式〉

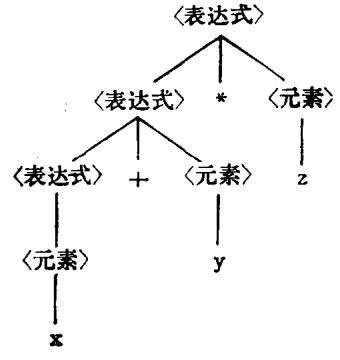


图 2.4

意味着从右到左计算。

请注意，在大多数程序语言中，乘法优先于加法，这里并没有保证这一点。为此，我们可以增加第三个非终结符进一步改进这一文法：

〈表达式〉::=〈项〉|〈表达式〉+〈项〉

〈项〉::=〈元素〉|〈项〉*〈元素〉

〈元素〉::=x|y|z|（〈表达式〉）

图2.5是x+y*z的唯一语法树，而图2.6是y*z+x的树。

下一步要简单介绍两个小型样本语言Pam和Eva，结合这两个语言将给出完整的BNF文法例的子。

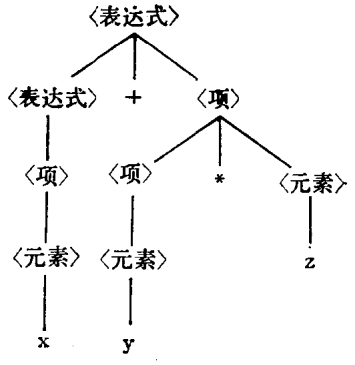


图 2.5

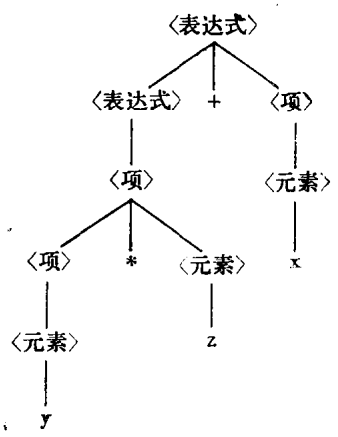


图 2.6

练　　习

1. 为许多常用语言中形式为

〈名字〉(〈下标〉, …, 〈下标〉)

的下标变量写出一组 BNF 的产生规则。

2. 已知一个文法有下列产生规则

〈语句〉::=〈条件句〉|

begin 〈语句〉**end**|…

〈条件句〉::=**if**〈比较式〉**then**〈语句〉|

if〈比较式〉**then**〈语句〉**else**〈语句〉

这个文法会造成所谓“尾随 **else**”式的二义性。

试用部分语法树为工具：

（1）说明如何产生二义性。

（2）解释应如何改变文法来消除二义性。这就要求在符号 **else** 后面跟一个〈无条件句〉(就是除了条件句以外的任何一种〈语句〉)。

（3）解释如何通过引入一个类似 **fi** 的闭符号来消除二义性。

2.1.2 Pam语言

Pam 是一种只能用于说明整数算术运算的非常简单的语言，类似于 Algol 语言。下一节我们将介绍另外一种语言称 Eva。二者都不是实用的语言。我们只把它们作为媒介来介绍及说明各种各样的形式规范方法。Pam 中有允许任意层次嵌套的结构化条件句及循环句，但不包括说明句及程序块结构；在 Pam 源程序中不同位置出现的相同标识符一定表示同一变量（整数变量）。

表 2.1 是 Pam 完整的文法。从中可以清楚地看出该语言的语法。请注意，操作符‘*’和‘/’优先于‘+’和‘-’；〈常数〉(整数) 是由一个或多个数字串组成；〈变量〉的第一个字符是

字母，后面可有任意多个字母和数字。

虽然文法中没有明确涉及语言的语义，但根据我们在其他语言方面所具有的经验，可以很容易猜出该语言的大部分语义。例如，形式为

while C **do** S **end**

的循环句中，C是〈比较式〉，S是〈序列〉，该句的作用就是：只要C成立，就要重复执行S（S可能执行0次或多次）。形为

to E **do** S **end**

的循环句中，根据〈表达式〉E-开始求出的值决定S重复的次数。除此，还能想到，当且仅当〈变量〉赋值后，〈变量〉的值才真正确定；对未赋值的〈变量〉进行运算都是语义错误。

下列例子是计算正输入值n的阶乘的一个Pam程序：

```
read n;
to n do
    read x;
    if x>0 then
        y:=1; z:=1;
        while z<>x do
            z:=z+1;
            y:=y*z
        end;
        write y
    fi
end
```

表2.1　Pam的BNF文法

〈程序〉::=〈序列〉
〈序列〉::=〈语句〉|〈序列〉;〈语句〉
〈语句〉::=〈输入句〉|〈输出句〉|〈赋值句〉|〈条件句〉|〈确定循环〉|〈不确定循环〉
〈输入句〉::=**read**〈变量表〉
〈输出句〉::=**write**〈变量表〉
〈变量表〉::=〈变量〉|〈变量表〉,〈变量〉
〈赋值句〉::=〈变量〉:=〈表达式〉
〈条件句〉::=**if**〈比较式〉**then**〈序列〉**fi**|**if**〈比较式〉**then**〈序列〉**else**〈序列〉**fi**
〈确定循环〉::=**to**〈表达式〉**do**〈序列〉**end**
〈不确定循环〉::=**while**〈表达式〉**do**〈序列〉**end**
〈比较式〉::=〈表达式〉〈关系符〉〈表达式〉
〈表达式〉::=〈项〉|〈表达式〉〈弱操作符〉〈项〉
〈项〉::=〈元素〉|〈项〉〈强操作符〉〈元素〉

续表

〈元素〉::=〈常数〉|〈变量〉|(〈表达式〉)
〈常数〉::=〈数字〉|〈常数〉〈数字〉
〈变量〉::=〈字母〉|〈变量〉〈字母〉|〈变量〉〈数字〉
〈关系符〉::= =|=〈|〈|〉|〉=|〈 〉
〈弱操作符〉::= +|-
〈强操作符〉::= *|/
〈数字〉::= 0|1|2|3|4|5|6|7|8|9|
〈字母〉::= *a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*|

表 2.1 所示文法生成了所有的合法 Pam 程序。其中的终结符号串中没有非法的 Pam 程序，从这个意义上，此文法就是 Pam 的完整的且严格的语法说明。

练　　习

1. 参考表 2.1，请你对一个简短的 Pam 程序画出完整的语法树。

2. 试写出一个 Pam 程序，该程序读入一个正整数 *n* 及打印出大于 *n* 的最小完全数。〔完全数等于它的所有因数（包括 1）的和。如 6（=1+2+3)是一个完整数。〕一定要保证你的程序满足表 2.1 的语法。

3. 请你用尽可能少的非终结符（当然也是尽可能少的产生规则）重写表 2.1 的文法。

4. 在 Pam 及一些实用的类似于 Algol 语言中，分号作为语句间的分隔符。而 PL/I 等语言用分号作为语句的结束号。即作为每个语句的最后一个符号。请你改写表 2.1 的文法，来定义 Pam 的一种方言，使分号成为语句的结束号。

2.1.3 Eva 语言

Eva 具有 Pam 所没有的一些重要概念及特点：不同类型（**Char**，**String** 和 **Proc**）的标识符，复合数据结构（字符串值）及其操作（**head**，**tail**，**cons**），说明，程序块结构，递归过程及参数传送。语言仍然是十分少的。我们能从表中看到这一点。

表 2.2　Eva 的 BNF 文法

〈程序〉::=〈块〉
〈块〉::=**begin**〈说明序列〉〈语句序列〉**end**
〈说明序列〉::=〈说明〉|〈说明序列〉〈说明〉
〈说明〉::=〈说明符〉〈名字表〉|**proc**〈名字〉=〈语句〉|
　　proc〈名字〉(〈参数表〉)=〈语句〉
〈参数表〉::=〈说明符〉〈名字表〉|〈参数表〉，〈说明符〉〈名字表〉

此为试读，需要完整PDF请访问：www.ertongbook.com