编程语言要素

Daniel P. Friedman Mitchell Wand

September 10, 2024

目录

译者的话

这里介绍给读者的，是丹尼尔·弗里德曼 (Daniel P. Friedman) 教授的著作。

弗里德曼教授的著作，除了早年论文以及“小人书”系列（*The Little Schemer* 及其它）之外，译者所见，尚有《编程语言要素》（*Essentials of Programming Languages*，简称 *EoPL*）和《*Scheme* 与编程艺术》(*Scheme and the Art of Programming*)。这些书中，当属《编程语言要素》和《*Scheme* 与编程艺术》最具教材性质。两书所述话题既多，行文又平白易懂，读者循着作者铺就的道路漫步，自能一窥编程语言的门径。较晚出的“小人书”系列，虽然是夫子自道，却有柏拉图对话录的意味。每本专涉一个主题，薄薄一本，不像市面上常见的计算机书籍，动辄千页。但是每一本都言简意赅，值得反复阅读。

The Little Schemer 因为用新颖（就计算机科学来说）的形式讲授最基本的计算机科学原理，早成为同类图书中的名著。不过这类图书似乎像夹生牛排一样，不大对我国读者的胃口，所以它的翻译和引进，还是近几年的事，是否引起消化不良，大概仍待观察。至于之后讲逻辑编程的 *The Reasoned Schemer*，讲设计模式的 *A Little Java, A Few Patterns*，讲定理证明的 *The Little Prover*，讲依赖类型 (*dependent types*) 的 *The Little Typer*，更乏人问津。这也许不能怪读者：这些著作所涉的主题，本来就鲜有广泛传播的中文译著（设计模式除外）。作者不过是将它们反刍提炼，以简洁而富有创造性的方式加以介绍。既然这些图书的主题，本不为人所熟知，又不像热门技术一样，堪做高薪职位的敲门砖，怎能期望读者对它们发生兴趣呢?

同样的理由，却不适用于《编程语言要素》。如果说编程语言世界是一座无尽宽广的城堡，“小人书”系列就是介绍城堡中设计最精巧的房间；*EoPL* 介绍的，是造城堡用的砂石。房间的布局装饰往往变化很快，造房间用的砂石变动

得却极慢。虽然新的编程语言仍在源源不断地设计出来，本书介绍的大多数概念仍将存在于许多望得见望不见的编程语言之中。即使读者并不打算做计算机领域的“泥瓦匠”，仔细阅读本书，仍能对众多习见的概念加深理解。

尽管计算机科学也称科学，从大量流行的书籍之中，我们很难看到它的科学性自何而来，读过本书，我们能够清晰地看到：看似淆乱的编程语言世界背后，也存在某些牢固的运行规律。如同地心说给人带来的困扰一样，违背这些规律，将给编程语言的用户带来无穷的麻烦。理解它们，就能更轻松地理解一门编程语言。而这样的理解，势必能消除许多无谓的争论。

说起争论，本书出版已有三十年。三十年间，无数的争论来了又去了，本书却从未能在中国找到它的读者。它今天就能吗？天暗下来，大风如酒，绯红的晚霞从苍郁的树木后直升起来，满路汽车唱着急促的歌谣，像是催着我，快点将这本书译出来。虽仍抱着这样的疑问，我还是愿意把这本书介绍给读者。

译名表

下表列出本书专有名词及其翻译，并对有疑义者略加说明。

Abstract data type (ADT)	抽象数据类型
Abstract syntax tree	抽象语法树
Abstract syntax	抽象语法
Abstract type	抽象类型
Abstraction boundary	抽象边界
Activation record	活跃记录表
Actual parameter	实际参数，简称实参（同 *argument*）
Advertise	公布
Alternation	并联
Analyzer	分析器
Ancestor	祖先
Antecedent	前件
Application expression	调用表达式
Apply	应用
Argument	实参（文中有时也用这一术语泛指参数）
Association-list	关联列表
Axiom	公理
Backus-Naur Form	巴科斯-诺尔范式
β-reduction	β-推导
Bigits	大位

Bignum representation	大数表示法
Binary method problem	二元方法问题
Binary semaphore	二元信号量
Binding	绑定
Blocked	受阻塞
Body	主体（一般指某种语法结构（`let` 定义、过程定义、类方法定义，甚至整个程序）之中，变量声明包裹起来的、执行期望动作的部分）
Bottom-up	自底向上
Bounce	弹球（参见 *trampoline*）
Bound variable	绑定变量
Bound	绑定
Byte code	字节码
Call by name	按名调用
Call by need	按需调用
Call by reference	按指调用
Call by value result	按值和结果调用
Call by value	按值调用
car	首项，或不译
Casting	强制转换
cdr	余项，或不译
Child	子类
Class environment	类环境
Class	类，或类别（在第 *9* 章，*class* 特指面向对象语言中的类；在附录 *B*，*class* 指词牌的种类。）
Client	客户
Closed over, closed in	闭合于
Closure	闭包
Coinduction	余归纳
Concatenation	串联
Conclusion	结论

Concrete syntax	具体语法
Concrete type	具体类型
Consequent	后件
Constructor	构造器
Context	上下文
Continuation-passing style	续文传递风格
Continuation	续文（这一术语或译作“继续”，或译作“续延”，或译作“(计算）续体”；这里译作“续文”，是将程序类比为文章，那么作为程序中任意位置后续内容的抽象，*continuation* 即为“续文”）
Contour diagram	等深线
Contract	合约
Contravariant	逆变的
Control context	控制上下文
Coroutine	协程
Covariant	协变的
Critical region	关键区域
Curried	咖喱式
Currying	咖哩化
Data abstraction	数据抽象
De Bruijn index	德布鲁金索引
Declaration	声明
Deduction tree	推理树
Deference	解引用
Defined language	被定语言
Defining language	定义语言
Defunctionalization	消函
Delegate	委托
Denoted value	指代值
Derivation	推导

Descendant	后代
Domain equation	定义域方程
Domain-specific language	特定领域语言
Double dispatch	双派发
Dynamic assignment	动态赋值
Dynamic binding	动态绑定
Dynamic dispatch	动态分发
Dynamic extent	动态期限
Dynamic scoping	动态定界
Dynamic	动态
Eager	即时
Effect	效果（*effect* 常常与 *side* 连用，通译为“副作用”；在本书中，*effect* 从不与 *side* 连用，或许是暗示：作为程序效果的 *effect*，不仅仅是一种“副产品”）
Environment	环境
Exception handling	异常处理
Expanded type	展开类型
Explicit reference	显式引用
Export	输出
Expressed value	表达值
Expression-oriented	面向表达式
Extend	扩展
Extent	期限
External	外在
Extractor	提取器（提取数据结构中某一部分内容的过程统称）
Field	字段
Flowchart program	流程图程序
Fluid binding	流式绑定
For effect	求效果

Form	形式（在本书中，这一术语和 *construct* 含意相近）
Formal parameter	形式参数，简称形参
Frame	帧
Front end	前端
Frozen	冻结
Generalization	泛化
Generalize	放宽
Global	全局性
Grammar	语法
Grammatical	语法
Hard-coded	硬编码
Host class	持有类
Hypothesis	假设
Ill-typed	异常类型
Implementation language	实现语言
Implementation	实现
Implicit reference	隐式引用
Inclusive or	涵盖或
Infer	推断
Inherit from	继承于
Inheritance	继承
Inherited attribute	继承属性
Inlining	内联
Input expression	输入表达式
Instance variable	实例变量
Instance	实例
Interface polymorphism	接口多态
Interface	接口
Internal	内在
Interpreter recipe	解释器秘方

Invariant	不变式
Iterative control behavior	迭代性控制行为
Kleene Plus	克莱尼加号
Kleene Star	克莱尼星号
Kleene closure	克莱尼闭包
L-value	左值
Lambda calculus	Lambda 演算
Lazy evaluation	懒求值
Lexeme	词素
Lexical address	词法地址
Lexical depth	词深
Lexical item	词条
Lexical scoping	词法定界
Lexical specification	词法规范
Lexical variable	词法变量
List	列表
Location	位置（特指存储器中的位置）
Member	成员
Memoization	助记法
Message-psasing	消息传递
Metacircular interpreter	自循环解释器
Method name	方法名
Method var	方法变量
Method	方法
Module definition	模块定义
Module procedure	模块过程
Module	模块
Multiple inheritance	多继承
Mutable	可变的
Mutex exclusion, mutex	互斥锁
Mutually recursive	互递归

Name mangling	名称混淆
Natural parameter passing	自然式传参（相对于以“*call by*”开头的几种参数传递机制）
Object-oriented programming	面向对象编程
Object	对象
Observer	观测器
Occurrence check	验存
Occur free	自由出现
Offer	提出
Opaque type	模糊类型
Operand position	操作数位置
Operand	操作数
Operator	操作符
Overloading	重载
Override	覆盖
pair	序对
Parameterized module	参数化模块
Parent	父类
Parser	解析器
Parsing	解析
Polish prefix notation	波兰前缀表示法
Polymorphic	多态
Pool	池（特指线程池）
Pre-emptive scheduling	抢占式调度
Predicate	谓词
Prefix list	前缀列表
Private	私有的
Procedural	过程式

Procedure	过程（文中使用 *procedure* 表示编程语言中的函数；使用 *function* 时，一般表示数学中的函数，这里将前者译作“过程”，以示区别）
Production	生成式
Promise	承诺
Propagate	传播
Protected	受保护的
Prototype	原型
Pseudo-variable	伪变量
Public	公有的
Qualified	受限变量
Quantum	量子（即时间片）
R-value	右值
Ready queue	就绪队列
Record	记录表
Recursive control behavior	递归性控制行为
Reference	引用
Regexp, regular expression	正则表达式
Registerization	寄存
Representation-independent	表示无关
Ribcage	肋排
Rib	肋骨
Rule (of inference)	（推理）规则
Runnable	可运行
Running	在运行
Safe	安全
Scanning	扫描
Scheduler	调度器
Scope	作用域
Scoping	定界

Semi-infinite	半无限
Separated list	分隔表
Sequentialization	序列化
Shadow	遮蔽
Share	共享
Signature	签名
Simple interface	简单接口
Simple module	简单模块
Simple variable	简单变量
Single-inheritance	单继承
Sound	健壮的
Source language	源语言
Stack	栈
Statement-oriented	面向语句
State	状态
Static depth	静深
Static environment	静态环境
Static method dispatch	静态方法分发
Static	静态
Storable value	可存储值
Store-passing interpreter	传递存储器的解释器
Store-passing specification	存储器传递规范
Store	存储器
Structural induction	结构化归纳法
Subclass polymorphism	子类多态
Subclass	子类
Subgoal induction	子目标归纳
Subroutine	子程序
Substitution	代换，代换式（组）（视上下文，这一术语有时表示类型推导的动作，有时表示动作的结果；表示结果时，有时为单数，有时为复数）

Subtype polymorphism	子类型多态
Subtyping	子类型判定
Super call	超类调用
Superclass	超类
Superprototype	超型
Supply, provide	提供
Symbol table	符号表
Syntactic category	句法类别
Syntactic derivation	句法推导
Table	表（特指哈希表，或与之类似的数据结构。第 2 章中的 *Record* 与之类似。）
Tail call	尾调用
Tail form	尾式
Tail position	尾端
Target language	目标语言
Thawed	解冻
Thread identifier	线程描述符
Thread	线程
Thunk	值箱
Time slice	时间片
Token	词牌
Top-down	自顶向下
Trampoline	跳床（读者或许对 *Microsoft Windows XP* 系统自带的“三维弹球”游戏仍有印象，文中的 *trampoline* 和 *bounce* 可以视为这一游戏的类比••或者说，这一游戏生动说明了 *trampoline* 和 *bounce* 的作用）
Trampolining	跳跃（视语境，有时也将这一术语直接翻译为跳床）
Translator	翻译器
Transparent	透明
Type Checking	类型检查

Type Inference	类型推导
Type abbreviation	类型缩写
Type environment	类型环境
Type error	类型错误
Type expression	类型表达式
Type structure	类型结构
Unary representation	一元表示法
Unification	合一
Value declaration	值声明
Value restriction	值约束
Variable aliasing	变量别名
Variable expression	变量表达式
Variable	变量
Variant	变体
Virtual machine	虚拟机
Well-typed	正常类型

序

本书引你直面计算机编程中最基本的思想：

计算机语言的解释器不过是另一个程序。

听上去显而易见，不是吗？但是含义颇深。若你是计算理论学家，解释器思想会使你想起哥德尔发现的形式逻辑系统局限，图灵的通用计算机概念，还有冯诺依曼存储程序机的基本思想。若你是程序员，掌握解释器思想则是巨大力量的源泉。它如灌顶醍醐，能彻底改变你思考编程的方式。

听说解释器之前，我便没少编程，也开发过一些大型程序。其中一例是由 PL/I 写成的大型数据和信息检索系统。实现系统时，我把 PL/I 看作一批固定的规则，由一些遥不可及的语言设计者订立。我认为我的工作不是修改这些规则，甚或深入理解它们，而是从（极为）浩繁的手册中挑挑拣拣，选出这种那种功能来用。我从没想过有一些潜藏的结构组织语言，也不觉得我可能得推翻语言设计者的一些决定。我不知道如何创造嵌合的子语言来帮我组织系统实现，所以整个程序不是各部分能够灵活组合的几种语言，倒像幅庞杂的马赛克，每片都得小心塑造和放置。不理解解释器，你还是能编程，甚至能成为一名称职的程序员，但你无法成为大师。

作为一名程序员，出于三个原因，你应该了解解释器。

首先，在某些时刻，你得实现解释器••也许不是全能而通用语言的解释器，但终归是解释器。几乎每一个与人灵活交互的复杂计算机系统••例如，计算机绘图工具或信息检索系统••都包含某种解释器来组织交互。这些程序可能包含复杂的单项操作••在显示屏的一个区域加上阴影，或执行数据库搜索••但解释器是胶水，使你将单项操作组合成有用的模式。你能把一项操作的结果当作另

一操作的输入吗？你能给一个操作序列命名吗？名字是局部的还是全局的？你能给一个操作序列加上参数，并给它的输入命名吗？诸如此类。不论单项操作多么复杂精巧，常常是胶水的质量最直截地决定系统的能力。很容易找到单项操作良好，但胶水糟糕的程序；回头来看，我觉得我的 PL/I 数据库程序胶水实在糟糕。

其次，即使程序本身不是解释器，也包含类似解释器的重要片段。细究一个复杂的计算机辅助设计系统，你可能发现协同工作的几何图形识别语言，图形解释器，基于规则的控制解释器，以及面向对象语言解释器。组织复杂程序最有效的方式之一是视之为一组语言，每种语言从不同角度，以不同方式处理程序元素。为合适的目的选择合适的语言，理解实现时的利弊权衡：这就是解释器研究的内容。

学习解释器的第三个原因，是直接涉及语言结构的编程技术日益重要。如今，操作面向对象系统中类的层次备受关注，而这只是该趋势的一个例子。这可能是无可避免的结果，因为我们的程序正日趋复杂••多留心语言可能是应对这种复杂性的最好工具。再想想那个基本思想：解释器本身不过是一个程序。但那程序以某种语言写就，其解释器本身不过是另一程序，以某种语言写就，其解释器本身又是……或许截然区分程序和编程语言是个误导性的想法，未来的程序员不再自视为编写某种程序，而是为每个应用创造新的语言。

弗里德曼和宛德完成了标志性的工作，他们的著作势将改变编程语言课程的图景。他们不只给你讲讲解释器，他们展示给你看。本书的核心是一系列精妙的解释器，始于一种高级抽象语言，逐步揭示语言特性，直至一状态机。事实上你可以运行这些代码，研习和修改它们，改变这些解释器的处理方式，诸如定界、参数传递、控制结构，等等。

作者以解释器研究语言的执行，并展示了如何用同样的思想不加运行地分析程序。在新增的两章中，他们展示了如何实现类型检查器和推导器，以及这些特性在现代面向对象编程语言中如何交互。

这种方法之所以吸引人，部分在于作者选择了优良的工具••Scheme 语言。这种语言结合了 Lisp 的统一语法和数据抽象能力，以及 Algol 的词法定界和块状结构。但利器到了大师手中方能物尽其用。本书的示例解释器足可垂范。确实如此，因为他们是可以运行的示范。我确信这些解释器和分析器将在未来数

年现身于许多编程系统的核心之中。

这不是本容易的书。解释器难以掌握不是没有原因的。语言设计者要比普通应用的开发者更加远离最终用户。设计应用程序时，你考虑将要完成的具体任务，以及将要包含的功能。但设计语言时，你考虑人们想要实现的种种应用，以及实现它们的可能方式。你的语言应该是静态作用域还是动态作用域，或者二者兼有？它应该有继承吗？它传递参数时是传引用还是传值？续文应该是显式的还是隐式的？这些都取决于你想让你的语言如何使用，哪些程序应该很容易编写，哪些程序可以不那么易写。

此外，解释器实在是微妙的程序。解释器中一行代码的简单改变都能使最终语言的行为发生剧变。别想粗读这些程序••即使是相对简单的代码，世上也绝少有人能够一瞥新的解释器就预测它的行为。所以钻研这些程序吧。最好，运行它们••这些是能运行的代码。试着解释一些简单的表达式，然后试试更复杂的。添加错误信息。修改解释器。设计你自己的变体。试着真正掌握这些程序，而不是对之如何工作粗具印象。

如果你这样做了，你对编程和身为程序员的自己看法定会不同。你将自视为语言的设计师，而不仅是语言的用户。你将成为语言组成规则的主人，而不仅是他人所选规则的随从。

三版附言

上述序言写于短短七年之前。在九十年代，难以想象七年之内，信息应用和服务已遍布全球。它们由持续增长的编程语言和编程框架驱动••这一切都基于不断壮大的解释器平台。

想创建 Web 页面吗？在九十年代，那等于格式化静态文本和图像，实际上是创建由浏览器运行的程序，而浏览器仅仅执行一条“print”语句。今天的动态网页充分利用脚本语言（解释性语言的另一名字），如 Javascript。浏览器程序则相当复杂，包含对 Web 服务器的异步调用。而服务器通常运行着另一程序，由完全不同的编程框架写就。框架又可能含有多种服务，每种服务都有各自的语言。

或者你要创建一个机器人，在大型多人在线游戏••如魔兽世界••中改善形

象。这时，你可能使用 Lua 之类的脚本语言••还可能支持面向对象扩展••来协助表达各类行为。

又或者你在为大型计算机集群编程，进行全球规模的索引和搜索。这时，你可能用函数式编程中的 map-reduce 范式来编写程序，以减轻明确安排处理器调度细节的负担。

又或者你在为传感器网络开发新的算法，并且尝试用懒求值更好地解决并行和数据汇集问题。又或者你在探索 XSLT 那样的转换系统来控制 Web 页面。又或者你在设计转换和重混多媒体流的框架。又或者……

多少新应用！多少新语言！多少崭新的解释器！

依旧，编程新手，甚至有经验的人，可以分别看待每种框架，在既定的规则内工作。但创造新框架要求大师级技能：理解语言运行背后的原理，挑选最适合每种应用的语言特性，熟知如何创造带给语言生命的解释器。这些是你能从本书学到的技能。

Hal Abelson
剑桥，马萨诸塞州
2007 年 9 月

前言

目标

本书是对编程语言的分析性研究。我们的目标是深入有效地理解编程语言的基本概念。这些概念的重要性久经证实。它们是理解编程语言未来发展的基础。

这些概念多关乎程序元素的语义，或称含义。含义反映了元素在程序执行时应当如何解释。名为解释器的程序最直接地表现了程序语义，且能执行。它们通过直接分析程序文本的抽象表示来处理程序。因此我们以解释器为主要载体来表达编程语言元素的语义。

程序作为对象，最富趣味的话题是：“它做什么?”解释器研究向我们阐明这点。解释器至关重要，因为它发隐抉微，又是更高效编译和其他程序分析的不二法门。

有一大类系统，根据语法结构将信息从一种形式转换为另一形式，解释器也是它们的示例。例如，编译器将程序转换为适合硬件或虚拟机解释的形式。虽然通用的编译技术超出本书范围，但我们确实开发出了几个基本的程序翻译系统。它们反映了编译中典型的程序分析，比如控制转换，变量绑定解析，以及类型检查。

以下是我们方法的一些独特策略：

1. 每一新概念都用一种小型语言解释。这些语言通常是渐进式的：后面的语言依赖前面语言的特性。

2. 用语言处理器••如解释器和类型检查器••解释指定语言所写程序的行为。它们以形式化（无歧义并且完备）和可执行的方式表现语言设计中的决定。

3. 适当的时候，我们用接口和规范建立数据抽象。如此，我们可以改变数据的表示而不必更改程序。我们以此来研究不同的实现策略。

4. 我们的语言处理器在较高层次写就，用以简括地表示语义；同时也在极低层次写就，用以理解实现策略。

5. 我们展示了如何用简单的代数操作预测程序行为，推导它们的性质。但我们通常很少采用数学符号，而是偏爱研究程序的行为，它们是我们语言实现的组成部分。

6. 正文解释关键概念，习题探讨备选设计和其它问题。例如，正文阐述静态绑定，习题讨论动态绑定。一系列习题将词法寻址的概念用于本书设计的各种语言。

通过层次丰富的抽象，我们能从多个角度看待编程语言。我们的解释器常常提供高层次观点，用十分简洁的方式表现语言的语义，这与形式数学语义相去不远。另一个极端是，我们展示了程序如何转换为汇编语言特有的极低层次形式。通过一些小的步骤完成这一转换，我们维持了高低层次观点之间的清晰联系。

我们对本版做了一些重大改动。对所有重要定义，我们都引入了非正式合约。这有助于阐明所做的抽象。此外，新增关于模块的章节。为了使实现更容易，第 3、4、5、7 和第 8 章的源语言假定只能给函数传递一个参数；我们增加了支持多参数过程的习题。第 6 章是新增的，因为我们选择了一阶组合式续文传递风格变换，而非关系式的。同时，由于*尾式* (*tail-form expression*) 的性质，我们在这一章使用多参数过程。在对象和类一章同样如此，但这并非必然。每章都经过修订，新增了很多习题。

组织

前两章为深入学习编程语言奠定了基础。第 1 章强调数据的归纳式定义法和递归编程之间的联系，介绍了几个与变量作用域相关的概念。第 2 章介绍了数据类型工具，由此引出关于数据抽象的讨论和表示方式转换的例子，这种转换将在后面的章节中使用。

第 3 章使用这些基础描述编程语言的行为。本章介绍解释器，作为解释语言运行时行为的机制，并为一门简单的词法作用域语言设计了解释器，该语言支持一等过程和递归。这个解释器是本书后续部分许多材料的基础。本章结尾详细讨论了一种用索引代替变量的语言，结果便是，变量查询可由列表引用完成。

第 4 章介绍的新组件••状态••将位置与值对应。加入了它，我们就能研究关于表示方式的各种问题。此外，它还使我们能够探索按指调用、按名调用和按需调用的参数传递机制。

第 5 章用续文传递风格重写我们的基础解释器。运行解释器所需的控制结构随之从递归转变为迭代。这揭示了解释性语言的控制机制，强化了对控制问题的整体直觉。它也使我们能够用蹦床、异常处理和多线程机制扩展语言。

第 6 章是上一章的续篇。上一章我们展示了如何把我们熟悉的解释器转换为续文传递风格，本章我们展示了如何为更大的一类程序完成转换。续文传递风格是一种强大的编程工具，因为它使几乎所有语言都能实现任意的顺序控制机制。这一算法也是抽象描述源到源程序转换的例子。

第 7 章将第 3 章的语言转换为有类型的语言。首先我们实现一个类型检查器。然后我们展示如何用基于合一的类型推导算法推断程序中的类型。

第 8 章建立类型模块，这极度依赖对前一章的理解。模块既使我们能够建立和强化抽象边界，又提供了新的定界方式。

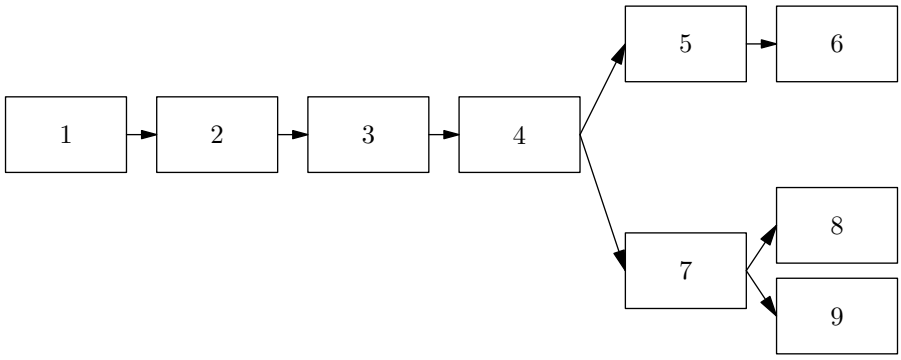
第 9 章以类为中心，展示了面向对象编程语言的基本概念。我们首先设计了高效的运行时结构，作为本章第二部分内容的基础。第二部分将第 7 章的类型检查器思想和第一部分的面向对象编程语言结合起来，结果便是传统的有类型面向对象语言。这要求引入新的概念，包括接口、抽象方法和强制转换。

“扩展阅读”解释了本书的每一思想源于何处。虽然有时我们只列出了能接

触到的来源，读者泛览本章，仍能窥源访本。

最后，附录 B 介绍了我们的 SLLGEN 解析系统。

各章的依赖关系如下图所示。



用法

本书在本科生和研究生课程中均已使用，也已在职业程序员的持续教育课程中使用。我们假定的背景知识有数据结构，过程式语言（如 C、C++、Java）和 Scheme、ML、Python 或 Haskell 的编程经验。

习题是文本的重要部分，散见于各处。它们难易有别，简单者，理解相关材料便轻而易举（标为 [⋆]）；困难者，需要花费大量思考和编程工作（标为 [⋆⋆⋆]）。大量关于应用，历史以及理论的材料潜藏其间。我们建议读一读每道习题，想一想如何解决它们。虽然我们用 Scheme 编写解释程序和转换系统，任何支持一等过程和赋值的语言（ML、Common Lisp、Python、Ruby 等等）都足以完成本书练习。

练习 0.1 [⋆] 我们常说“某语言具有某性质”。为每种说法找出一种或多种具有该性质的语言，以及一种或多种不具有该性质的语言。请随意搜索这些信息，不拘任何编程语言的介绍性书籍（比如 Scott(2005), Sebesta (2007), 或者 Pratt & Zelkowitz(2001)）。

这是本实践性的书：本书讨论的一切都可在通常的大学课程限度内完成。因为函数式语言的抽象特性尤其适合这类编程，我们可以写出强大的语言处理系统，既简洁，又能以适当的努力掌握。

网站由出版社提供，包含本书所有解释器和分析器的完整 Scheme 代码。代码用 PLT Scheme 写成。[1]我们选择这种 Scheme 实现，是因为它的模块系统和编程环境对学生助益良多。代码多半兼容于 R^5RS，当能轻易移植到任何功能完整的 Scheme 实现。

[1]本书网站已迁移至 `http://www.eopl3.com`，代码改用 Racket 实现，网址为 `https://github.com/mwand/eopl3`。••译注

致谢

感谢无数同事同学，他们使用和评论了本书的前两版，又为第三版的漫长构思提供了无价帮助。特别感激以下诸君的贡献，对他们我们深表谢忱。Oliver Danvy 鼓励我们思考一阶组合式续文传递算法，并提出了一些饶有趣味的练习。Matthias Felleisen 切中肯綮的分析改善了一些章节的设计。Amr Sabry 提出了许多有用的建议，在第 9 章的草稿中发现了至少一处极难察觉的问题。Benjamin Pierce 自使用本书第一版授课以来提出了一系列深入见解，其中大半已为我们采用。Gary Leavens 对第二版的初稿提出了极为细致而珍贵的评论，包括许多详细的修改建议。Stephanie Weirich 在本书第二版第 7 章的类型推导代码中发现了一个不易察觉的问题。Ryan Newton 除了阅读第 2 章草稿之外，又承担了一个艰巨任务：为该版的每道练习题推荐难度等级。Chung-chieh Shan 从第三版初稿开始授课，提供了大量有益的评论。

Kevin Millikin、Arthur Lee、Roger Kirchner、Max Hailperin 和 Erik Hilsdale 使用了第二版初稿。Will Clinger、Will Byrd、Joe Near 和 Kyle Blocher 使用了本版手稿。他们的评论弥足珍贵。Ron Garcia、Matthew Flatt、Shriram Krishnamurthi、Steve Ganz、Gregor Kiczales、Marlene Miller、Galen Williamson、Dipanwita Sarkar、Steven Bogaerts、Albert Rossi、Craig Citro、Christopher Dutchyn、Jeremy Siek 和 Neil Ching 也认真阅读并做了评论。

尤其感谢这几位对本书的帮助。感谢 Neil Ching 编制了索引。在我们尝试设计 `define-datatype` 和 `cases` 语法扩展时，Jonathan Sobel 和 Erik Hilsdale 完成了一些原型实现,贡献了很多想法。程序语言小组,尤其是 Matthias Felleisen、Matthew Flatt、Robby Findler 和 Shriram Krishnamurthi，热心帮忙兼容他们的 DrScheme 系统。Kent Dybvig 开发了极为高效和健壮的 Chez

Scheme 实现，本书作者已使用多年。Will Byrd 在整个过程中都提供了无价帮助。Matthias Felleisen 强烈推荐我们兼容 DrScheme 的模块系统，这在位于 `http://www.eopl3.com` 的实现中显而易见。

特别值得一提的是体贴和关心我们进度的人。George Springer 和 Larry Finkelstein 提供了无价支持。特别感谢 Bob Prior，我们 MIT 出版社的优秀编辑，是他鼓励我们攻下本版的写作。同样感谢 Bob 的继任 Ada Brunstein，帮我们流畅过渡到新的编辑。印第安纳大学信息学院和东北大学计算机信息科学学院为我们进行这一工程创造了环境。Mary Friedman 在数周的写作谈话中热情招待，大大加快了我们的进程。

我们感谢 Christopher T. Haynes 在前两版中的合作。很遗憾，他已志不在此，没有继续同我们参与本版的编写。

最后，感谢我们的家人，包容我们写作本书时的热情。谢谢你，Rob、Shannon、Rachel、Sara，还有 Mary；谢谢你，Rebecca 和 Joshua，Jennifer 和 Stephen，Joshua 和 Georgia，还有 Barbara。

本版筹备良久，我们可能忽视了一路帮助过我们的人。我们为任何忽视道歉。您总在书中看到这样的话，可能会奇怪为什么有人会这样写。忽视了别人，你当然感到抱歉。但是，当你有了一众帮手（得一个村子才装得下），你确实有种责任感：任何人都不容忽视。所以，如果您被忽视了，我们深表歉意。

•• D.P.F，M.W.

1 归纳式数据集

解释器与检查器一类的程序是编程语言处理器的核心，本章介绍写这些用到的基本编程工具。

因为编程语言的语法通常为嵌套或者树状结构，递归将是我们的主要技巧。 1.1 节和 1.2 节介绍归纳定义数据结构的方法，并展示如何用这类定义指导递归程序的编写。 1.3 节展示如何将这些技巧推广到更为复杂的程序。本章以大量练习作结。这些练习是本章的核心。欲掌握本书余下部分依赖的递归编程技巧，得自它们的经验不可或缺。

1.1 递推定义的数据

编写过程代码时，必须明确知道什么样的值能作为过程的参数，什么样的值是过程的合法返回值。这些值的集合通常很复杂。本节介绍定义值集合的形式化技术。

1.1.1 归纳定义法

归纳定义法是定义值集合的有效方法。为解释这一方法，我们用它来描述自然数 $N = 0, 1, 2, ...$ 的某一子集 S。

定义 1.1.1 *自然数 n 属于 S，当且仅当：*

1. $n = 0$，*或*

2. $n - 3 \in S$

来看看如何用这一定义判断哪些自然数属于 S。已知 $0 \in S$，因此 $3 \in S$，因为 $(3-3)=0$，且 $0 \in S$。同样地，$6 \in S$，因为 $(6-3)=3$，且 $3 \in S$。依此类推，可得结论：所有 3 的整数倍都属于 S。

其他自然数呢？$1 \in S$ 吗？已知 $1 \neq 0$，所以条件一不满足。此外，$(1-3)=-2$，不是自然数，故不是 S 的元素，因此条件二不满足。因为 1 不满足任一条件，所以 $1 \notin S$。同样地，$2 \notin S$。4 呢？仅当 $1 \in S$ 时 $4 \in S$。但 $1 \notin S$，所以 $4 \notin S$。同理可得，如果 n 是自然数且不是 3 的整数倍，则 $n \notin S$。

据此推论，可得 S 是 3 的整数倍自然数集合。

可以用该定义编写一个函数，判断一个自然数 n 是否属于 S。

in-S? : $N \to Bool$
用法: `(in-S? n) = #t 若 n 属于 S，否则 #f`

```
(define in-S?
  (lambda (n)
    (if (zero? n) #t
        (if (>= (- n 3) 0)
            (in-S? (- n 3))
            #f))))
```

这里根据定义，我们用 Scheme 编写了一个递归过程。符号 `in-S? : N → Bool` 是一条注释，称为该函数的合约 (*contract*)。它表示 `in-S?` 应为一过程，取一自然数，产生一布尔值。这样的注释对阅读和编写代码很有帮助。

要判断是否 $n \in S$，先判断是否 $n = 0$。如果是，那么答案为真。否则，判断是否 $n-3 \in S$。欲知此，首先判断是否 $(n-3) \geqslant 0$。如果是，那么可以用我们的过程判断它是否属于 S。如果不是，那么 n 不可能属于 S。

S 又能够定义为：

定义 1.1.2 集合 S 为 N 所包含的集合中，满足如下两条性质的最小集合：

1. $0 \in S$，且
2. 若 $n \in S$，则 $n+3 \in S$。

“最小集合”是指该集合满足性质 1 和 2，并且是其他任何满足性质 1 和 2 的集合的子集。易知只能有一个这样的集合：如果 S_1 和 S_2 都满足性质 1 和 2，

并且都为最小，那么 $S_1 \subseteq S_2$（因为 S_1 最小）且 $S_2 \subseteq S_1$（因为 S_2 最小），因此 $S_1 = S_2$。之所以需要这一额外条件，是因为否则的话将有许多集合满足其他两个条件（见）。

该定义还能表示为：

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

这只是前一定义的简便表示。每个条目称为一条推理规则 (*rule of inference*)，或称规则 (*rule*)；水平线读作“若-则”。线上部分称作假设 (*hypothesis*) 或者前件 (*antecedent*)；线下部分称作结论 (*conclusion*) 或者后件 (*consequent*)。罗列两个或更多假设时，它们以隐含的“与”连接（见）。不含假设的规则称作公理 (*axiom*)。写公理时通常不加水平线，如：

$$0 \in S$$

该规则意为，自然数 n 属于 S，当且仅当能用有限次推理规则，从公理推得陈述“$n \in S$”。这一解释自然使 S 成为闭合于该规则的最小集合。

这些定义意思相同。我们把版本一称作自顶向下 (*top-down*) 的定义，版本二称作自底向上 (*bottom-up*) 的定义，版本三称作推理规则定义。

再来看几个运用这些的例子。

定义 1.1.3（整数列表，自顶向下） *Scheme* 列表是整数列表，当且仅当：

1. 列表为空，或
2. 列表为序对，首项为整数，余项为整数列表。

我们用 *Int* 表示所有整数的集合，用 *List-of-Int* 表示所有整数列表的集合。

定义 1.1.4（整数列表，自底向上） 集合 *List-of-Int* 是满足如下两条性质的最小 *Scheme* 列表集合：

1. `()` ∈ *List-of-Int*，或
2. 若 $n \in$ *Int* 且 $l \in$ *List-of-Int*，则 `(`n `.` l`)` ∈ *List-of-Int*。

这里，我们用中缀“`.`”代表 Scheme 中 `cons` 操作的结果。式子 `(`n `.` l`)` 代表 Scheme 序对的首项为 n，余项为 l。

定义 1.1.5（整数列表，推理规则）

$$\frac{}{\texttt{()} \in \textit{List-of-Int}}$$

$$\frac{n \in \textit{Int} \quad l \in \textit{List-of-Int}}{\texttt{(}n\ \texttt{.}\ l\texttt{)} \in \textit{List-of-Int}}$$

这三个定义等价。来看看如何用它们生成一些 *List-of-Int* 的元素。

1. 由的性质 1 或的规则 1，`()` 是整数列表。
2. 由的性质 2，`(14 . ())` 是整数列表。因为 `14` 是整数，`()` 是整数列表。写成 *List-of-Int* 规则二的形式，就是

$$\frac{\texttt{14} \in \textit{Int} \quad \texttt{()} \in \textit{List-of-Int}}{\texttt{(14 . ())} \in \textit{List-of-Int}}$$

3. 由的性质 2，`(3 . (14 . ()))` 是整数列表。因为 `3` 是整数，`(14 . ())` 是整数列表。仍写成 *List-of-Int* 规则二的形式，是

$$\frac{\texttt{3} \in \textit{Int} \quad \texttt{(14 . ())} \in \textit{List-of-Int}}{\texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}$$

4. 由的性质 2，`(-7 . (3 . (14 . ())))` 是整数列表。因为 `-7` 是整数，`(3 . (14 . ()))` 是整数列表。再次写成 *List-of-Int* 规则二的形式，是

$$\frac{\texttt{-7} \in \textit{Int} \quad \texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}{\texttt{(-7 . (3 . (14 . ())))} \in \textit{List-of-Int}}$$

5. 不按照这种方式得到的都不是整数列表。

改句点表示法为列表表示法，可知 `()`、`(14)`、`(3 14)` 以及 `(-7 3 14)` 都是 *List-of-Int* 的元素。

还可以结合各条规则来证明 `(-7 . (3 . (14 . ())))` $\in$ *List-of-Int*，以见出整个推理过程。下面的树状图叫做推导 (*derivation*) 或推理树 (*deduction tree*)。

$$\dfrac{\texttt{-7} \in \mathit{Int} \qquad \dfrac{\texttt{3} \in \mathit{Int} \qquad \dfrac{\texttt{14} \in \mathit{Int} \qquad \texttt{()} \in \mathit{List\text{-}of\text{-}Int}}{\texttt{(14 . ())} \in \mathit{List\text{-}of\text{-}Int}}}{\texttt{(3 . (14 . ()))} \in \mathit{List\text{-}of\text{-}Int}}}{\texttt{(-7 . (3 . (14 . ())))} \in \mathit{List\text{-}of\text{-}Int}}$$

练习 1.1 [⋆] 写出下列集合的归纳定义。以三种方式（自顶向下，自底向上，推理规则）写出每个定义，并用你的规则推导出各集合的一些元素。

1. $\{3n+2 \mid n \in N\}$
2. $\{2n+3m+1 \mid n, m \in N\}$
3. $\{(n, 2n+1) \mid n \in N\}$
4. $\{(n, n^2) \mid n \in N\}$。不要在你的规则中使用平方。提示：想一想方程 $(n+1)^2 = n^2 + 2n + 1$。

练习 1.2 [⋆⋆] 下面的几对规则分别定义了什么集合？给出解释。

1. $(0,1) \in S \qquad \dfrac{(n,k) \in S}{(n+1, k+7) \in S}$
2. $(0,1) \in S \qquad \dfrac{(n,k) \in S}{(n+1, 2k) \in S}$
3. $(0,0,1) \in S \qquad \dfrac{(n,i,j) \in S}{(n+1, j, i+j) \in S}$

4. [⋆⋆⋆] $(0,1,0) \in S$ $\quad \dfrac{(n,i,j) \in S}{(n+1, i+2, i+j) \in S}$

练习 1.3 [⋆] 找出自然数的子集 T，满足 $0 \in T$，且对任何 $n \in T$，都有 $n+3 \in T$，但 $T \neq S$，S 是由给出的集合。

1.1.2 语法定义法

前述例子较为直观，但是不难想象，描述更复杂的数据类型会有多麻烦。为了方便，我们展示如何用*语法* (*grammar*) 定义集合。语法通常用来指定字符串的集合，但也能用来定义值的集合。

例如，集合 *List-of-Int* 可用语法定义为：

List-of-Int ::= **(** **)**
List-of-Int ::= **(** *Int* **.** *List-of-Int* **)**

这两条规则对应上述中的两条性质。规则一是说空表属于 *List-of-Int*；规则二是说，若 n 属于 *Int* 且 l 属于 *List-of-Int*，则 **(** n **.** l **)** 属于 *List-of-Int*。这些规则叫做*语法*。

来看看该定义的各个部分，其中有：

- **非终结符**。这些是所定义的集合名。本例中只定义了一个集合，但是通常，可能会定义数个集合。这些集合有时称为*句法类别* (*syntactic category*)。

  依照惯例，我们将非终结符和集合名的首字母大写，在文中提及它们的元素时，则用小写。这要比听起来容易。例如，*Expression* 是非终结符，但我们写作 $e \in$ *Expression* 或“e 是一个 expression”。

  另一常见写法，名叫*巴科斯-诺尔范式* (*Backus-Naur Form*) 或 *BNF*，是在词周围加尖括号，如 ⟨expression⟩。

- **终结符**。这些是集合外在表示中的字符，在本例中，是“**.**”、“**(**”和“**)**”。这些常用打字机字体写出，如 `lambda`。

- **生成式**。规则叫做*生成式* (*production*)。每个生成式的左边是一个非终结符，右边包含终结符和非终结符。左右两边通常用符号 ::= 分隔，读作*是*或*可以*

是。式子右边用其他句法类别和终结符（如左括号、右括号和句点）指定一种方法，用以构建当前句法类别的元素。

如果某些句法类别的含义在上下文中足够清晰，在生成式中提到它们时通常不作定义，如 *Int*。

语法常常简写。当一个生成式的左边与前一生成式相同时，一般会略去。根据这一惯例，我们的语法可以写作

$$\begin{aligned} \textit{List-of-Int} &::= (\,) \\ &::= (\textit{Int} \;.\; \textit{List-of-Int}) \end{aligned}$$

给同一句法类别编写一组规则时，也可以只写一次 ::= 和左边内容，随后的各个右边内容用特殊符号“|”（竖线，读作或）分隔。用“|”，*List-of-Int* 的语法可写成：

$$\textit{List-of-Int} ::= (\,) \mid (\textit{Int} \;.\; \textit{List-of-Int})$$

另一种简写是克莱尼星号 (*Kleene Star*)，写作 $\{\ldots\}^*$。当它出现在右边时，表示一个序列，由任意多个花括号之间的内容组成。用克莱尼星号，*List-of-Int* 的定义可以简写为

$$\textit{List-of-Int} ::= (\{\textit{Int}\}^*)$$

这也包含没有任何内容的情况。如果内容出现 0 次，得到的是空字符串。

星号的变体是克莱尼加号 (*Kleene Plus*) $\{\ldots\}^+$，表示一个或多个内容的序列。把上例中的 * 换成 $^+$，定义的句法类别是非空整数列表。

星号的另一变体是分隔表 (*separated list*) 表示法。例如，$\textit{Int}^{*(c)}$ 表示一个序列，包含任意数量的非终结符 *Int* 元素，以非空字符序列 c 分隔。这也包含没有元素的情况。如果有 0 个元素，得到的是空字符串。例如，$\textit{Int}^{*(,)}$ 包含字符串

```
8
14, 12
7, 3, 14, 16
```

$\textit{Int}^{*(;)}$ 包含字符串

```
8
14; 12
7; 3; 14; 16
```

这些简写不是必需的，总能够不用它们重写语法。

对由语法定义的集合，可以用*句法推导* (*syntactic derivation*) 证明给定值是其元素。这样的推导从集合对应的非终结符开始，在由箭头 $\Rightarrow$ 指示的每一步中，如果非终结符对应的句法类别未做定义，则将其代换为该类别的已知元素，否则代换为对应规则右边的内容。例如，前述证明“`(14 . ())` 是整数列表”，可以用句法推导化为

$$\begin{aligned} \textit{List-of-Int} &\Rightarrow \texttt{(}\textit{Int}\ \texttt{.}\ \textit{List-of-Int}\texttt{)} \\ &\Rightarrow \texttt{(14 . }\textit{List-of-Int}\texttt{)} \\ &\Rightarrow \texttt{(14 . ())} \end{aligned}$$

非终结符的替换顺序无关紧要，所以 `(14 . ())` 的推导也可以写成：

$$\begin{aligned} \textit{List-of-Int} &\Rightarrow \texttt{(}\textit{Int}\ \texttt{.}\ \textit{List-of-Int}\texttt{)} \\ &\Rightarrow \texttt{(}\textit{Int}\ \texttt{. ())} \\ &\Rightarrow \texttt{(14 . ())} \end{aligned}$$

练习 1.4 [⋆] 写出从 *List-of-Int* 到 `(-7 . (3 . (14 ())))` 的推导。

再来看一些有用集合的定义。

1. 许多符号操作过程用于处理只包含符号和具有类似限制的列表。我们把这些叫做 `s-list`，定义如下：

定义 1.1.6 （s-list，s-exp）

$$\begin{aligned} \textit{S-list} &::= \texttt{(}\{\textit{S-exp}\}^{*}\texttt{)} \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list} \end{aligned}$$

s-list 是 s-exp 的列表，s-exp 或者是 s-list，或者是一个符号。这里是一些 s-list。

```
(a b c)
(an (((s-list)) (with () lots) ((of) nesting)))
```

有时也使用更宽松的 s-list 定义，既允许整数，也允许符号。

2. 使用三元素列表表示内部节点，则以数值为叶子，以符号标示内部节点的二叉树可用语法表示为：

定义 1.1.7（二叉树）

$$\mathit{Bintree} ::= \mathit{Int} \mid (\mathit{Symbol}\ \ \mathit{Bintree}\ \ \mathit{Bintree})$$

这是此类树的几个例子：

```
1
2
(foo 1 2)
(bar 1 (foo 1 2))
(baz
 (bar 1 (foo 1 2))
 (biz 4 5))
```

3. *lambda 演算* (*lambda calculus*) 是一种简单语言，常用于研究编程语言理论。这一语言只包含变量引用，单参数过程，以及过程调用，可用语法定义为：

定义 1.1.8（lambda 演算）

$$\begin{aligned}\mathit{LcExp} &::= \mathit{Identifier}\\ &::= (\texttt{lambda}\ (\mathit{Identifier})\ \ \mathit{LcExp})\\ &::= (\mathit{LcExp}\ \ \mathit{LcExp})\end{aligned}$$

其中，*identifier* 是除 `lambda` 之外的任何符号。

第二个生成式中的 identifier 是 `lambda` 表达式主体内的变量名。这一变量叫做表达式的*绑定变量* (*bound variable*)，因为它绑定（或称捕获）主体内出现的任何同名变量。出现在主体内的同名变量都指代这一个。

要明白这怎么用，考虑用算术操作符扩展的 lambda 演算。在这种语言里，

```
(lambda (x) (+ x 5))
```

是一表达式，`x` 是其绑定变量。这式子表示一个过程，把它的参数加 5。因此，在

```
((lambda (x) (+ x 5)) (- x 7))
```

中，最后一个出现的 `x` 不是指 `lambda` 表达式中绑定的 `x`。1.2.4 节中介绍了 `occurs-free?`，到时我们再讨论这个问题。

该语法定义 *LcExp* 的元素为 Scheme 值，因此很容易写出程序来处理它们。

这些语法叫做*上下文无关* (*context-free*) 语法，因为一条规则定义的句法类别可以在任何引用它的上下文中使用。有时这不够严格。考虑二叉搜索树。其节点或者为空，或者包含一个整数和两棵子树

Binary-search-tree ::= **(** **)** | **(** *Int* *Binary-search-tree* *Binary-search-tree* **)**

这如实反映了每个节点的结构，但是忽略了二叉搜索树的一个要点：所有左子树的键值都小于（或等于）当前节点，所有右子树的键值都大于当前节点。

因为这条额外限制，从 *Binary-search-tree* 得出的句法推导并不都是正确的二叉搜索树。要判定某个生成式能否用于特定的句法推导，必须检查生成式用在哪种上下文。这种限制叫做*上下文敏感限制* (*context-sensitive constraints*)，或称*不变式* (*invariants*)。

定义编程语言的语法也会产生上下文敏感限制。例如，在许多编程语言中变量必须在使用之前声明。对变量使用的这一限制就对其上下文敏感。虽然可以用形式化方法定义上下文敏感限制，但这些方法远比本章考虑的复杂。实际中，常用的方法是先定义上下文无关语法，随后再用其他方法添加上下文敏感限制。第 7 章展示了这种技巧的一个例子。

1.1.3 归纳证明法

用归纳法描述的集合，其定义有两种用法：证明关于集合元素的定理，写出操作集合元素的程序。这里给出一个此类证明的例子，写程序留作下节的主题。

定理 1.1.1 令 t 为二叉树，形如，则 t 包含奇数个节点。

用归纳法证明 t 的大小。令 t 的大小等于 t 中节点的个数。归纳假设为 $IH(k)$：树的大小 $\leq k$ 时有奇数个节点。依照归纳法的惯例：先证明 $IH(0)$ 为真，然后证明若对任一整数 k，IH 为真，则对 $k+1$，IH 也为真。

1. 没有哪棵树只有 0 个节点，所以 $IH(0)$ 显然成立。

2. 设 k 为整数时，$IH(k)$ 成立，即，任何树的节点数 $\leq k$ 时，其实际数目为奇数。需证明 $IH(k+1)$ 也成立：任何树的节点数 $\leq k+1$ 时，节点数为奇数。若 t 有 $\leq k+1$ 个节点，根据二叉树的定义，只有两种可能：

   (a) t 形如 n，n 为整数。此时 t 只有一个节点，1 为奇数。

   (b) t 形如 $(sym\ \ t_1\ \ t_2)$，其中，sym 是一符号，t_1 和 t_2 是树。此时 t_1 和 t_2 节点数少于 t。因为 t 有 $\leq k+1$ 个节点，t_1 和 t_2 一定有 $\leq k$ 个节点。因此它们符合 $IH(k)$，一定各有奇数个节点，不妨分别设为 $2n_1+1$ 和 $2n_2+1$。则算上两棵子树和根，原树中的节点总数为

   $$(2n_1+1)+(2n_2+1)+1 = 2(n_1+n_2+1)+1$$

   也是一个奇数。

   陈述“$IH(k+1)$ 成立”证毕，归纳完成。 □

证明的关键是树 t 的子结构总是比 t 自身小。这种证明模式叫做结构化归纳法 (*structural induction*)。

结构化归纳证明

欲证明假设 $IH(s)$ 对所有结构 s 为真，需证明：

1. *IH* 对简单结构（没有子结构）为真。
2. 若 *IH* 对 s 的子结构为真，则对 s 本身也为真。

练习 1.5 [⋆⋆] 证明：若 $e \in LcExp$，则 e 中的左右括号数量相等。

1.2 推导递归程序

我们已经用归纳定义法描述了复杂集合。我们能够分析归纳式集合的元素，观察如何从较小元素构建集合。我们用这一想法写出了过程 `in-S?`，用以判断自然数是否属于集合 S。现在，我们用同样的想法定义更通用的过程，以便对归纳式集合做运算。

递归过程依赖于一条重要原则：

较小子问题原则

若能化问题为较小子问题，则能调用解决原问题的过程解决子问题。

已求得的子问题解随后可用来求解原问题。这可行，因为每次过程调用都是针对较小的子问题，直至最终调用，针对一个可以直接求解的问题，不需再次调用自身。

我们用一些例子解释这一想法。

1.2.1 list-length

标准的 Scheme 程序 `length` 求出列表中的元素个数。

```
> (length '(a b c))
3
> (length '((x) ()))
2
```

我们来写出自己的过程 `list-length`，做同样的事。

先来写出过程的*合约*。合约指定了过程可取参数和可能返回值的集合。合约也可以包含过程的期望用法或行为。这有助于我们在编写时及以后追踪我们的意图。在代码中，这是一条注释，我们用打字机字体示之，以便阅读。

```
list-length : List → Int
用法: (list-length l) = l 的长度
(define list-length
  (lambda (lst)
    ...))
```

列表的集合定义为

$$List ::= \texttt{()} \mid \texttt{(}Scheme\ value\ \texttt{.}\ List\texttt{)}$$

因此，考虑列表的每种情况。若列表为空，则长度为 0。

```
list-length : List → Int
用法: (list-length l) = l 的长度
(define list-length
  (lambda (lst)
    (if (null? lst)
        0
        ...)))
```

若列表非空，则其长度比其余项长度多 1。这就给出了完整定义。

list-length : *List* → *Int*
用法: `(list-length` l`)` = l 的长度

```
(define list-length
  (lambda (lst)
    (if (null? lst)
        0
        (+ 1 (list-length (cdr lst))))))
```

通过 `list-length` 的定义，我们可以看到它的运算过程。

```
  (list-length '(a (b c) d))
= (+ 1 (list-length '((b c) d)))
= (+ 1 (+ 1 (list-length '(d))))
= (+ 1 (+ 1 (+ 1 (list-length '()))))
= (+ 1 (+ 1 (+ 1 0)))
= 3
```

1.2.2 nth-element

标准的 Scheme 过程 `list-ref` 取一列表 `lst` 和从 0 开始计数的索引 `n`，返回 `lst` 的第 `n` 个元素。

```
> (list-ref '(a b c) 1)
'b
```

我们来写出自己的过程 `nth-element`，做同样的事。

仍沿用上述 *List* 的定义。

当 lst 为空时，`(nth-element` lst n`)` 应当返回什么？这种情况下，`(nth-element` lst n`)` 想要取空列表的元素，所以报错。

当 lst 非空时，`(nth-element` lst n`)` 应当返回什么？答案取决于 n。若 $n = 0$，答案是 lst 的首项。

当 lst 非空，且 $n \neq 0$ 时，`(nth-element` lst n`)` 应当返回什么？这种情况下，答案是 lst 余项的第 $(n-1)$ 个元素。由 $n \in N$ 且 $n \neq 0$，可知 $n-1$ 一定属于 N，因此可以递归调用 `nth-element` 找出第 $(n-1)$ 个元素。

这就得出定义

```
nth-element : List × Int → SchemeVal
用法: (nth-element lst n) = lst 的第 n 个元素
(define nth-element
  (lambda (lst n)
    (if (null? lst)
      (report-list-too-short n)
      (if (zero? n)
        (car lst)
        (nth-element (cdr lst) (- n 1))))))

(define report-list-too-short
  (lambda (n)
    (eopl:error 'nth-element
      "List too short by ~s elements.~%" (+ n 1))))
```

这里的注释 **nth-element** ： $List \times Int \to SchemeVal$ 表示 **nth-element** 是一个过程，取两个参数，一个为列表，一个为整数，返回一个 Scheme 值。这与数学中的表示 $f : A \times B \to C$ 相同。

过程 `report-list-too-short` 调用 `eopl:error` 来报告错误，后者会终止计算。它的首个参数是一符号，用于在错误信息中指示调用 `eopl:error` 的过程。第二个参数是一个字符串，会打印为错误信息。对应于字符串中的每个字符序列 `~s`，都必须有一个额外参数。打印字符串时，这些参数的值会替换对应的 `~s` 。`~%` 代表换行。错误信息打印后，计算终结。过程 `eopl:error` 并非标准 Scheme 的一部分，但大多数 Scheme 实现提供这样的组件。在本书中，我们以类似方式，用名字含 `report-` 的过程报告错误。

来看看 `nth-element` 如何算出答案：

```
  (nth-element '(a b c d e) 3)
= (nth-element   '(b c d e) 2)
= (nth-element     '(c d e) 1)
= (nth-element       '(d e) 0)
= d
```

这里，`nth-element` 递归处理越来越短的列表和越来越小的数字。

如果排除错误检查，我们得靠 `car` 和 `cdr` 报错来获知传递了空列表，但它们的错误信息无甚帮助。例如，当我们收到 `car` 的错误信息，可能得找遍整个程序中使用 `car` 的地方。

练习 1.6 [⋆] 如果调换 `nth-element` 中两个条件的顺序，会有什么问题?

练习 1.7 [⋆⋆] `nth-element` 的错误信息不够详尽。重写 `nth-element`，给出更详细的错误信息，像是“`(a b c)` 不足 8 个元素”。

1.2.3 remove-first

过程 `remove-first` 取两个参数：符号 s 和符号列表 los。它返回一个列表，除了不含第一个出现在 los 中的符号 s 外，所含元素及其排列顺序与 los 相同。如果 s 没有出现在 los 中，则返回 los。

```
> (remove-first 'a '(a b c))
'(b c)
> (remove-first 'b '(e f g))
'(e f g)
> (remove-first 'a4 '(c1 a4 c1 a4))
'(c1 c1 a4)
> (remove-first 'x '())
'()
```

写出此过程之前，我们先要定义符号列表集合 *List-of-Symbol*，以便给出问题的完整描述。不像上一节介绍的 s-lists，符号列表不包含子列表。

$$\textit{List-of-Symbol} ::= \texttt{()} \mid \texttt{(}\textit{Symbol}\ \texttt{.}\ \textit{List-of-Symbol}\texttt{)}$$

符号列表或者是空列表，或者是首项为符号，余项为符号列表。

如果列表为空，不需要移除 s，则答案为空列表。

remove-first : $Sym \times Listof(Sym) \to Listof(Sym)$
用法: `(remove-first` s los`)` 返回一列表，除了不含第一个出现在 los 中的符号 s 外，元素及其排列顺序与 los 相同。

```
(define remove-first
  (lambda (s los)
    (if (null? los)
        '()
        ...)))
```

写合约时，我们用 *Listof*(*Sym*) 而不是 *List-of-Symbol*。用这种写法可以免除许多上面那样的定义。

如果 los 非空，有没有哪种情况可以立刻得出答案？如果 los 的第一个元素是 s，比如 $los = (s\ \ s_1\ \ \ldots\ \ s_{n-1})$，$s$ 首次出现时是 los 的第一个元素，那么把它删除之后的结果是 $(s_1\ \ \ldots\ \ s_{n-1})$。

remove-first : $Sym \times Listof(Sym) \to Listof(Sym)$

```
(define remove-first
  (lambda (s los)
    (if (null? los)
        '()
        (if (eqv? (car los) s)
            (cdr los)
            ...))))
```

如果 los 的第一个元素不是 s，比如 $los = (s_0\ \ s_1\ \ \ldots\ \ s_{n-1})$，可知 s_0 不是第一个出现的 s，因此答案中的第一个元素一定是 s_0，即表达式 `(car los)` 的值。而且，los 中的首个 s 一定在 $(s_1\ \ \ldots\ \ s_{n-1})$ 中。所以答案的余下部分一定是移除 los 余项中首个 s 的结果。因为 los 的余项比 los 短，我们可以递归调用 `remove-first`，从 los 的余项中移除 s，即答案的余项可用 `(remove-first s (cdr los))` 求得。已知如何找出答案的首项和余项，可以用 `cons` 结合二者，通过表达式 `(cons (car los) (remove-first s (cdr los)))` 求得整个答案。由此，`remove-first` 的完整定义为

remove-first : $Sym \times Listof(Sym) \to Listof(Sym)$

```
(define remove-first
  (lambda (s los)
    (if (null? los)
        '()
        (if (eqv? (car los) s)
            (cdr los)
            (cons (car los) (remove-first s (cdr los)))))))
```

练习 1.8 [⋆] 如果把 `remove-first` 定义中的最后一行改为 `(remove-first s (cdr los))`，得到的过程做什么运算？对修改后的版本，给出合约，包括用法。

练习 1.9 [⋆⋆] 定义 `remove`。它类似于 `remove-first`，但会从符号列表中移除所有给定符号，而不只是第一个。

1.2.4 occurs-free?

过程 `occurs-free?` 取一个变量 var，由 Scheme 符号表示；一个 lambda 演算表达式 exp，形如；判断 var 是否自由出现于 exp。如果一个变量出现于表达式 exp 中，但不在某一 `lambda` 绑定之内，我们说该变量*自由出现* (*occur free*) 于表达式 exp 中。例如，

```
> (occurs-free? 'x 'x)
#t
> (occurs-free? 'x 'y)
#f
> (occurs-free? 'x '(lambda (x) (x y)))
#f
> (occurs-free? 'x '(lambda (y) (x y)))
#t
> (occurs-free? 'x '((lambda (x) x) (x y)))
#t
> (occurs-free? 'x '(lambda (y) (lambda (z) (x (y z)))))
#t
```

我们可以遵照 lambda 演算表达式的语法解决此问题：

$$\begin{aligned} \mathit{LcExp} &::= \mathit{Identifier} \\ &::= \texttt{(lambda (}\mathit{Identifier}\texttt{)}\ \mathit{LcExp}\texttt{)} \\ &::= \texttt{(}\mathit{LcExp}\ \ \mathit{LcExp}\texttt{)} \end{aligned}$$

我们可以总结出规则的各种情况：

- 若表达式 e 是一变量，则当且仅当 x 与 e 相同时，变量 x 自由出现于 e。

- 若表达式 e 形如 `(`$lambda$ `(`y`)` e'`)`，则当且仅当 y 不同于 x 且 x 自由出现于 e' 时，变量 x 自由出现于 e。

- 若表达式 e 形如 `(`e_1 e_2`)`，则当且仅当 x 自由出现于 e_1 或 e_2 时，x 自由出现于 e。这里的“或”表示*涵盖或* (*inclusive or*)，意为它包含 x 同时自由出现于 e_1 和 e_2 的情况。我们通常用“或”表示这种意思。

你可以说服自己，这些规则涵盖了“x 不在某一 lambda 绑定之中”表示的所有意思。

练习 1.10 [⋆] 我们常用“或”表示“涵盖或”。“或”还有什么含义?

然后，定义 `occurs-free?` 就很容易了。因为有三种情况要检查，我们不用 Scheme 的 `if`，而是用 `cond`。在 Scheme 中，若 exp_1 或 exp_2 返回真值，则 `(or` exp_1 exp_2`)` 返回真值。

occurs-free? : $Sym \times LcExp \to Bool$
用法: *若符号* var *自由出现于* exp，*返回* `#t`，*否则返回* `#f`

```
(define occurs-free?
  (lambda (var exp)
    (cond
      ((symbol? exp) (eqv? var exp))
      ((eqv? (car exp) 'lambda)
        (and
          (not (eqv? var (car (cadr exp))))
          (occurs-free? var (caddr exp))))
      (else
        (or
          (occurs-free? var (car exp))
          (occurs-free? var (cadr exp)))))))
```

这一过程略显晦涩。比如，很难弄明白 `(car (cadr exp))` 指代 `lambda` 表达式中的变量声明，或者 `(caddr exp)` 指代 `lambda` 表达式的主体。在 2.5 节，我们展示如何显著改善这种情况。

1.2.5 subst

过程 `subst` 取三个参数：两个符号 `new` 和 `old`，一个 s-list，`slist`。它检查 `slist` 的所有元素，返回类似 `slist` 的新列表，但把其中所有的 `old` 替换为 `new`。

```
> (subst 'a 'b '((b c) (b () d)))
'((a c) (a () d))
```

因为 `subst` 定义于 s-list 上，它的结构应当反映 s-list 的定义 ()：

$$\begin{aligned} \textit{S-list} &::= (\{\textit{S-exp}\}^*) \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list} \end{aligned}$$

克莱尼星号简洁地描述了集合 s-list，但对写程序没什么用。因此我们的第一步是抛开克莱尼星号重写语法。得出的语法表明，我们的过程应当该递归处理 s-list 的首项和余项。

$$\begin{aligned} \textit{S-list} &::= () \\ &::= (\textit{S-exp} \ . \ \textit{S-list}) \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list} \end{aligned}$$

这一例子比之前的复杂，因为它的语法输入包含两个非终结符，$S\text{-}list$ 和 $S\text{-}exp$。因此，我们需要两个过程，一个处理 $S\text{-}list$，另一个处理 $S\text{-}exp$。

subst : $\textit{Sym} \times \textit{Sym} \times \textit{S-list} \to \textit{S-list}$

```
(define subst
  (lambda (new old slist)
    ...))
```

subst-in-s-exp : $\textit{Sym} \times \textit{Sym} \times \textit{S-exp} \to \textit{S-exp}$

```
(define subst-in-s-exp
  (lambda (new old sexp)
    ...))
```

我们首先处理 `subst`。如果列表为空，不需要替换 `old`。

subst : *Sym* × *Sym* × *S-list* → *S-list*

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
        '()
        ...)))
```

如果 `slist` 非空，它的首项是一个 *S-exp*，余项是另一 s-list。这时，答案应当是一个列表，它的首项是把 `slist` 首项中的 `old` 替换为 `new` 的结果，它的余项是把 `slist` 余项中的 `old` 替换为 `new` 的结果。因为 `slist` 的首项是 *S-exp* 的元素，我们用 `subst-in-s-exp` 解决这一子问题。因为 `slist` 的余项是 *S-list* 的元素，我们递归调用 `subst` 处理它。

subst : *Sym* × *Sym* × *S-list* → *S-list*

```
(define subst
  (lambda (new old slist)
    (if (null? slist)
        '()
        (cons
         (subst-in-s-exp new old (car slist))
         (subst new old (cdr slist))))))
```

现在来处理 `subst-in-s-exp`。由语法，可知符号表达式 `sexp` 或者是符号，或者是 s-list。如果它是符号，那么得检查它与符号 `old` 是否相同。如果是，答案为 `new`；否则，答案还是 `sexp`。如果 `sexp` 是一个 s-list，那么我们递归调用 `subst` 找出答案。

subst-in-s-exp : *Sym* × *Sym* × *S-exp* → *S-exp*

```
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
        (if (eqv? sexp old) new sexp)
        (subst new old sexp))))
```

因为我们严格依照 *S-list* 和 *S-exp* 的定义,这个递归一定会终止。因为 `subst` 和 `subst-in-s-exp` 递归调用彼此，我们称之为*互递归* (*mutually recursive*)。

把 `subst` 拆解为两个过程••每个处理一种句法类别••是个重要技巧。对更为复杂的程序，我们得以每次考虑一个句法类别，从而化繁为简。

练习 1.11 [⋆] `subst-in-s-exp` 的最后一行中，递归是针对 `sexp` 而非更小的子结构，为什么一定能终止?

练习 1.12 [⋆] 用 `subst-in-s-exp` 的定义替换 `subst` 中的调用，从而排除这次调用，然后简化得到的过程。结果中的 `subst` 应当不需要 `subst-in-s-exp`。这种技巧叫做*内联* (*inlining*)，用于优化编译器。

练习 1.13 [⋆⋆] 在我们的例子中，我们从排除 *S-list* 语法内的克莱尼星号开始。依照原本的语法，用 `map` 重写 `subst`。

现在，我们有了编写过程处理归纳数据集的窍门，来把它总结成一句口诀。

遵循语法！

定义过程处理归纳式数据时，程序的结构应当反映数据的结构。

更准确地说:

- 为语法中的每个非终结符编写一个过程。这一过程负责处理相应非终结符的数据，不做其他。

- 在每个过程中，为相应非终结符的每一生成式写一分支。你可能需要额外的分支结构，但这样才能起步。对生成式右边出现的每个非终结符，递归调用相应的过程。

1.3 辅助过程和上下文参数

窍门*遵循语法*很有效，有时却还是不够。考虑过程 `number-elements`。这一过程取任何列表 `(`v_0 v_1 v_2 `...)`，返回一列表 `((0` v_0`) (1` v_1`) (2` v_2`) ...)`。

我们用过的那种直拆法不凑效，因为没有明显的方法能从 `(number-elements (cdr lst))` 得出 `(number-elements lst)`（但是，看看）。

要解决这个问题，我们*放宽* (*generalize*) 问题。我们写一个过程 `number-elements-from`，它取一个额外参数 n，指定起始编号。用递归处理列表，这个过程很容易写。

number-elements-from : $Listof(SchemeVal) \times Int \to Listof(List(Int, SchemeVal))$
用法 : `(number-elements-from '(`v_0 v_1 v_2 `...) n)`
`=((`n v_0`) (`$n+1$ v_1`) (`$n+2$ v_2`) ...)`

```
(define number-elements-from
  (lambda (lst n)
    (if (null? lst) '()
        (cons
         (list n (car lst))
         (number-elements-from (cdr lst) (+ n 1))))))
```

合约的标题告诉我们这个过程取两个参数，一个列表（包含任意 Scheme 值）和一个整数，返回一个列表，列表中的每个元素是包含两个元素的列表：一个整数，一个 Scheme 值。

一旦我们定义了 `number-elements-from`，很容易写出所需的过程。

number-elements : $Listof(SchemeVal) \to Listof(List(Int, SchemeVal))$

```
(define number-elements
  (lambda (lst n)
    (number-elements-from lst 0)))
```

这里有两个要点。首先，过程 `number-elements-from` 的定义独立于 `number-elements`。程序员经常要写一个过程，只调用某个辅助过程，并传

递额外的常量参数。除非我们理解辅助过程对参数的每个值做什么，我们很难理解调用它的过程做什么。这给了我们一条口诀：

> **避免神秘小工具！**
>
> 定义辅助过程时，总是指明它对所有参数值做什么，而不只是初始值。

其次，`number-elements-from` 的两个参数各有作用。第一个参数是我们要处理的列表，随每一次递归调用而减小。而第二个参数，则是对我们当前任务上下文 (*context*) 的抽象。在本例中，当调用 `number-elements` 时，我们最终调用 `number-elements-from` 处理原列表的每个子列表。第二个参数告知我们子列表在原列表中的位置。随递归调用，它不减反增，因为我们每次经过原列表的一个元素。有时我们称之为上下文参数 (*context argument*)，或者继承属性 (*inherited attribute*)。

另一个例子是向量求和。

要求列表中各项的和，我们可以遵循语法，递归处理列表的余项。那么我们的过程看起来像是：

list-sum : $Listof(Int) \to Int$

```
(define list-sum
  (lambda (loi)
    (if (null? loi)
        0
        (+ (car loi)
           (list-sum (cdr loi))))))
```

但是无法按照这种方式处理向量，因为它们不能够很方便地拆解。

因为我们无法拆解向量，我们放宽问题，为向量某一部分求和。问题定义为，计算

$$\sum_{i=0}^{i=length(v)-1} v_i$$

其中，v 是整数向量。通过把上界改为一个参数 n，我们放宽了原问题，所以新的任务是计算

$$\sum_{i=0}^{i=n} v_i$$

其中，$0 \leq n < length(v)$。

按照定义，用归纳法处理第二个参数 n，可以直接写出此过程。

partial-vector-sum : $Vectorof(Int) \times Int \to Int$
用法: 若 $0 \leq n < length(v)$，则

$$\texttt{(partial-vector-sum } v \ n\texttt{)} = \sum_{i=0}^{i=n} v_i$$

```
(define partial-vector-sum
  (lambda (v n)
    (if (zero? n)
        (vector-ref v 0)
        (+ (vector-ref v n)
           (partial-vector-sum v (- n 1))))))
```

由于 n 一定会递减至零，证明此程序的正确性需要用归纳法处理 n。由 $0 \leq n$ 且 $n \neq 0$，可得 $0 \leq (n-1)$，所以递归调用过程 `partial-vector-sum` 仍然满足其合约。

现在，要解决原问题就简单多了。因为向量长度为 0 时无法使用过程 `partial-vector-sum`，所以得另行处理这种情况。

vector-sum : $Vectorof(Int) \to Int$
用法: `(vector-sum` v`)` = $\sum_{i=0}^{i=length(v)-1} v_i$

```
(define vector-sum
  (lambda (v)
    (let ((n (vector-length v)))
      (if (zero? n)
          0
          (partial-vector-sum v (- n 1))))))
```

还有许多情况下，引入辅助变量或过程来解决问题会有帮助，甚至必不可少。只要能对新过程做什么给出独立的定义，尽可以如此。

练习 1.14 [⋆⋆] 若 $0 \leq n < length(v)$，证明 `partial-vector-sum` 的正确性。

1.4 练习

学写递归程序需要练习，那么我们拿几道习题结束本章。

每道习题都假定 `s` 是一个符号，`n` 是一个非负整数，`lst` 是一个列表，`loi` 是一个整数列表，`los` 是一个符号列表，`slist` 是一个 s-list，`x` 是任意 Scheme 值；类似地，`s1` 是一个符号，`los2` 是一个符号列表，`x1` 是一个 Scheme 值，等等。还假定 `pred` 是一个谓词 (*predicate*)，即一个过程，取任意 Scheme 值，返回 `#t` 或者 `#f`。除非某个具体问题另有限制，不要对数据作其他假设。在这些习题中，不需要检查输入是否符合描述；对每个过程，都假定输入值是指定集合的成员。

定义，测试和调试每个过程。你的定义应当有本章这种合约和用法注释。可以随便定义辅助过程，但是你定义的每个辅助过程都应该有其说明，如同 1.3 节那样。

测试这些程序时，先试试所有给出的例子，然后用其他例子测试，因为给定的例子不足以涵盖所有可能的错误。

练习 1.15 [⋆] `(duple n x)` 返回包含 `n` 个 `x` 的列表。

```
> (duple 2 3)
'(3 3)
> (duple 4 '(ha ha))
'((ha ha) (ha ha) (ha ha) (ha ha))
> (duple 0 '(blah))
'()
```

练习 1.16 [⋆] `lst` 是由二元列表（长度为 2 的列表）组成的列表，`(invert lst)` 返回一列表，把每个二元列表反转。

```
> (invert '((a 1) (a 2) (1 b) (2 b)))
'((1 a) (2 a) (b 1) (b 2))
```

练习 1.17 [⋆] `(down lst)` 给 `lst` 的每个顶层元素加上一对括号。

```
> (down '(1 2 3))
'((1) (2) (3))
> (down '((a) (fine) (idea)))
'(((a)) ((fine)) ((idea)))
> (down '(a (more (complicated)) object))
'((a) ((more (complicated))) (object))
```

练习 1.18 [⋆] `(swapper s1 s2 slist)` 返回一列表，将 `slist` 中出现的所有 `s1` 替换为 `s2`，所有 `s2` 替换为 `s1`。

```
> (swapper 'a 'd '(a b c d))
'(d b c a)
> (swapper 'a 'd '(a d () c d))
'(d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
'((y) x (z (y)))
```

练习 1.19 [⋆⋆] `(list-set lst n x)` 返回一列表，除第 `n` 个元素（从零开始计数）为 `x` 外，与 `lst` 相同。

```
> (list-set '(a b c d) 2 '(1 2))
'(a b (1 2) d)
> (list-ref (list-set '(a b c d) 3 '(1 5 10)) 3)
'(1 5 10)
```

练习 1.20 [⋆] `(count-occurrences s slist)` 返回 `slist` 中出现的 `s` 个数。

```
> (count-occurrences 'x '((f x) y (((x z) x))))
3
> (count-occurrences 'x '((f x) y (((x z) () x))))
3
> (count-occurrences 'w '((f x) y (((x z) x))))
0
```

练习 1.21 [⋆⋆] sos1 和 sos2 是两个没有重复元素的符号列表，(product sos1 sos2) 返回二元列表的列表，代表 sos1 和 sos2 的笛卡尔积。二元列表排列顺序不限。

```
> (product '(a b c) '(x y))
'((a x) (a y) (b x) (b y) (c x) (c y))
```

练习 1.22 [⋆⋆] (filter-in pred lst) 返回的列表，由 lst 中满足谓词 pred 的元素组成。

```
> (filter-in number? '(a 2 (1 3) b 7))
'(2 7)
> (filter-in symbol?  '(a (b c) 17 foo))
'(a foo)
```

练习 1.23 [⋆⋆] (list-index pred lst) 返回 lst 中第一个满足谓词 pred 的元素位置，从零开始计数。如果 lst 中没有元素满足谓词，list-index 返回 #f。

```
> (list-index number? '(a 2 (1 3) b 7))
1
> (list-index symbol?  '(a (b c) 17 foo))
0
> (list-index symbol?  '(1 2 (a b) 3))
#f
```

练习 1.24 [⋆⋆] 若 lst 中的任何元素不满足 pred，(every? pred lst) 返回 #f，否则返回 #t。

```
> (every? number? '(a b c 3 e))
#f
> (every? number? '(1 2 3 4 5))
#t
```

练习 1.25 [⋆⋆] 若 lst 中的任何元素满足 pred，(exists? pred lst) 返回 #t，否则返回 #f。

```
> (exists? number? '(a b c 3 e))
#t
> (exists? number? '(a b c d e))
#f
```

练习 1.26 [⋆⋆] (up lst) 移除 lst 中每个顶层元素周围的一对括号。如果顶层元素不是列表，则照原样放入结果中。(up (down lst)) 的结果与 lst 相同，但 (down (up lst)) 不一定是 lst（参见）。

```
> (up '((1 2) (3 4)))
'(1 2 3 4)
> (up '((x (y)) z))
'(x (y) z)
```

练习 1.27 [⋆⋆] (flatten slist) 返回一列表，由 slist 中的符号按出现顺序组成。直观上，flatten 移除参数内的所有内层括号。

```
> (flatten '(a b c))
'(a b c)
> (flatten '((a) () (b ()) () (c)))
'(a b c)
> (flatten '((a b) c (((d)) e)))
'(a b c d e)
> (flatten '(a b (() (c))))
'(a b c)
```

练习 1.28 [★★] `loi1` 和 `loi2` 是元素按照升序排列的整数列表，`(merge loi1 loi2)` 返回 `loi1` 和 `loi2` 中所有整数组成的的有序列表。

```
> (merge '(1 4) '(1 2 8))
'(1 1 2 4 8)
> (merge '(35 62 81 90 91) '(3 83 85 90))
'(3 35 62 81 83 85 90 90 91)
```

练习 1.29 [★★] `(sort loi)` 返回一列表，将 `loi` 中的元素按照升序排列。

```
> (sort '(8 2 5 2 3))
'(2 2 3 5 8)
```

练习 1.30 [★★] `(sort/predicate pred loi)` 返回一列表，将 `loi` 的元素按照谓词指定的顺序排列。

```
> (sort/predicate < '(8 2 5 2 3))
'(2 2 3 5 8)
> (sort/predicate > '(8 2 5 2 3))
'(8 5 3 2 2)
```

练习 1.31 [★] 写出如下过程，对二叉树 () 做运算：`leaf` 和 `interior-node` 生成二叉树，`leaf?` 检查二叉树是否是一片叶子，`lson`、`rson` 和 `contents-of` 取出一个节点的各部分。`contents-of` 应对叶子和内部节点都适用。

练习 1.32 [★] 写出过程 `double-tree`，它取一棵二叉树，形如，生成另一棵二叉树，把原二叉树中的所有整数翻倍。

练习 1.33 [★★] 写出过程 `mark-leaves-with-red-depth`，它取一棵二叉树 ()，生成与原树形状相同的另一棵二叉树，但在新的二叉树中，每个叶子中的整数表示它和树根之间含有 `red` 符号的节点数。例如，表达式

```
(mark-leaves-with-red-depth
 (interior-node 'red
  (interior-node 'bar
   (leaf 26)
   (leaf 12))
  (interior-node 'red
   (leaf 11)
   (interior-node 'quux
    (leaf 117)
    (leaf 14)))))
```

使用中定义的过程，应返回二叉树

```
(red
 (bar 1 1)
 (red 2 (quux 2 2)))
```

练习 1.34 [★★★] 写出过程 `path`，它取一个整数 `n` 和一棵含有整数 `n` 的二叉搜索树（bst）`bst`，返回由 `left` 和 `right` 组成的列表，表示如何找到包含 `n` 的节点。如果在树根处发现 `n`，它返回空列表。

```
> (path 17 '(14 (7 () (12 () ()))
                (26 (20 (17 () ())
                        ())
                    (31 () ()))))
'(right left left)
```

练习 1.35 [★★★] 写出过程 `number-leaves`，它取一棵二叉树，生成与原树形状相同的二叉树，但叶子的内容从 0 开始计的整数。例如，

```
(number-leaves
 (interior-node 'foo
  (interior-node 'bar
   (leaf 26)
```

```
 (leaf 12))
(interior-node 'baz
 (leaf 11)
 (interior-node 'quux
  (leaf 117)
  (leaf 14)))))
```

应返回

```
(foo
 (bar 0 1)
 (baz
  2
  (quux 3 4)))
```

练习 1.36 [⋆⋆⋆] 写出过程 g，则 n-e 的 number-elements 可以定义为：

```
(define number-elements
  (lambda (lst)
    (if (null? lst) '()
        (g (list 0 (car lst))
           (number-elements (cdr lst))))))
```

2 数据抽象

2.1 用接口定义数据

每当我们想以某种方式表示一些量时，我们就新定义了一种数据类型：它的取值是其表示，它的操作是处理其实体的过程。

这些实体的表示通常很复杂，所以如能避免，我们不愿关心其细节。我们可能想改变数据的表示。最高效的表示往往难以实现，所以我们可能希望先简单实现，只在确知系统的整体性能与之攸关时，才改用更高效的表示。不管出于什么原因，如果我们决定改变某些数据的表示方式，我们得能定位程序中所有依赖表示方式的部分。这就需要借助*数据抽象* (*data abstraction*) 技术。

数据抽象将数据类型分为两部分：*接口* (*interface*) 和*实现* (*implementation*)。*接口*告诉我们某类型表示什么数据，能对数据做什么操作，以及可由这些操作得出的性质。*实现*给出数据的具体表示，以及处理数据表示的代码。

这样抽象出的数据类型称为*抽象数据类型* (*abstract data type*)。程序的其余部分••数据类型的*客户* (*client*) ••只能通过接口中指定的操作处理新数据。这样一来，如果我们希望改变数据的表示，只需改变数据处理接口的实现。

这一想法并不陌生：我们写程序处理文件时，多数时候只关心能否调用过程来打开，关闭，读取文件或对文件做其他操作。同样地，大多数时候，我们不关心整数在机器中究竟怎样表示，只关心能否可靠地执行算术操作。

当客户只能通过接口提供的过程处理某类型的数据时，我们说客户代码与*表示无关* (*representation-independent*)，因为这些代码不依赖数据类型值的表示。

那么所有关于数据表示的信息必然在实现代码之中。实现最重要的部分就

是表示数据的规范。我们用符号 $\lceil v \rceil$ 指代“数据 v 的表示”。

要说得更明白些，来看一个简单例子：自然数类型。待表示的数据是自然数。接口由四个过程组成：`zero`、`is-zero?`、`successor` 和 `predecessor`。当然，不是随便几个过程都可以作为这一接口的实现。当且仅当一组过程满足如下四个方程时，可以作为 `zero`、`is-zero?`、`successor` 和 `predecessor` 的实现：

$$\texttt{(zero)} = \lceil 0 \rceil$$

$$\texttt{(is-zero?}\ \lceil n \rceil\texttt{)} = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases}$$

$$\texttt{(successor}\ \lceil n \rceil\texttt{)} = \lceil n+1 \rceil \quad (n \geq 0)$$

$$\texttt{(predecessor}\ \lceil n+1 \rceil\texttt{)} = \lceil n \rceil \quad (n \geq 0)$$

这一定义没有指明自然数应当如何表示，它只要求这些过程都产生指定的行为。即，过程 `zero` 必须返回 0 的表示；给定数字 n 的表示，过程 `successor` 必须返回数字 $n+1$ 的表示，等等。这个定义没对 `(predecessor (zero))` 做出说明，所以按这个定义，任何行为都是可以接受的。

现在可以写出处理自然数的客户程序，而且不论用哪种表示方式，都保证能得出正确的结果。例如，不论怎样实现自然数，

```
(define plus
  (lambda (x y)
    (if (is-zero? x)
      y
      (successor (plus (predecessor x) y)))))
```

都满足 `(plus` $\lceil x \rceil$ $\lceil y \rceil$`)` $= \lceil x+y \rceil$。

大多数接口都包含：若干*构造器* (*constructor*)，用来产生数据类型的元素；若干*观测器* (*observer*)，用来从数据类型的值中提取信息。这里有三个构造器，`zero`、`successor` 和 `predecessor`；一个观测器，`is-zero?`。

可以用多种方式表示这套接口，我们考虑其中三种。

1. *一元表示法* (*Unary representation*)：在一元表示法中，自然数 n 由 n 个 `#t` 组成的列表表示。所以，0 表示为 `()`，1 表示为 `(#t)`，2 表示为 `(#t #t)`，等等。可以用归纳法定义这种表示方式：

$\lceil 0 \rceil =$ `()`
$\lceil n+1 \rceil =$ `(#t . `$\lceil n \rceil$`)`

要满足该表示的定义，数据处理过程可以写成：

```
(define zero (lambda () '()))
(define is-zero? (lambda (n) (null? n)))
(define successor (lambda (n) (cons #t n)))
(define predecessor (lambda (n) (cdr n)))
```

2. *Scheme 数字表示法* (*Scheme number representation*)：在这种表示中，只需用 Scheme 内置的数字表示法（本身可能十分复杂!）。令 $\lceil n \rceil$ 为 Scheme 整数 `n`，则所需的四个过程可以定义为：

```
(define zero (lambda () 0))
(define is-zero? (lambda (n) (zero? n)))
(define successor (lambda (n) (+ n 1)))
(define predecessor (lambda (n) (- n 1)))
```

3. *大数表示法* (*Bignum representation*)：在大数表示法中，数值以 N 进制表示，N 是某个大整数。该方法以 0 到 $N-1$ 之间的数字（有时不称数位，而称*大位* (*bigits*)）组成的列表表示数值，这就很容易表示远超机器字长的整数。这里，为了便于使用，我们把最低位放在列表最前端。这种表示法可用归纳法定义为：

$$\lceil n \rceil = \begin{cases} \texttt{()} & n = 0 \\ \texttt{(}r\ \texttt{.}\ \lceil q \rceil\texttt{)} & n = qN + r, 0 \leqslant r < N \end{cases}$$

所以，如果 $N = 16$，那么 $\lceil 33 \rceil =$ `(1 2)`，$\lceil 258 \rceil =$ `(2 0 1)`，因为：

$$258 = 2 \times 16^0 + 0 \times 16^1 + 1 \times 16^2$$

这些实现都没有强制数据抽象：无法防止客户程序查验表示用的是列表还是 Scheme 整数。与之相对，有些语言直接支持数据抽象：它们允许程序员创建新的接口，确保只能通过接口提供的过程处理新数据。如果类型的表示隐藏起来，不会因任何操作而暴露（包括打印），那就说该类型是*模糊* (*opaque*) 的，否则称之为*透明* (*transparent*) 的。

Scheme 没有提供标准机制来创建新的模糊类型，所以我们退而求其次：定义接口，靠客户程序的作者小心行事，只使用接口中定义的过程。

在第 8 章中，我们讨论一些方法，以便强化语言的这种协议。

练习 2.1 [⋆] 实现大数表示法的四种操作。然后用你的实现计算 10 的阶乘。随着参数改变，执行时间如何变化？随着进制改变，执行时间如何变化？解释原因。

练习 2.2 [⋆⋆] 详加分析上述表示。从满足数据类型定义的角度来说，它们在何种程度上是成功或失败的？

练习 2.3 [⋆⋆] 用差分树表示所有整数（负数和非负数）。差分树是一列表，可用语法定义如下：

Diff-tree ::= `(one)` | `(diff` *Diff-tree* *Diff-tree*`)`

列表 `(one)` 表示 1。如果 t_1 表示 n_1，t_2 表示 n_2，那么 `(diff` n_1 n_2`)` 表示 $n_1 - n_2$。所以，`(one)` 和 `(diff (one) (diff (one) (one)))` 都表示 1；`(diff (diff (one) (one)) (one))` 表示 -1。

1. 证明此系统中，每个数都有无限种表示方式。
2. 实现这种整数表示法：写出 nat 定义的 `zero`、`is-zero?`、`successor` 和 `predecessor`，此外还要能表示负数。这种方式下，整数的任何合法表示都应该能作为你过程的参数。例如，你的过程 `successor` 可以接受无限多种 1 的合法表示，且都应给出一个 2 的合法表示。对 1 的不同合法表示，可以给出不同的 2 的合法表示。
3. 写出过程 `diff-tree-plus`，用这种表示做加法。你的过程应针对不同的差分树进行优化，并在常数时间内得出结果（即与输入大小无关）。注意，不可使用递归。

2.2 数据类型的表示策略

使用数据抽象的程序具有表示无关性：与用来实现抽象数据类型的具体表示方式无关，甚至可以通过重新定义接口中的一小部分过程来改变表示。在后面的章节中我们常会用到这条性质。

本节介绍几种数据类型的表示策略。我们用数据类型环境 (*environment*) 解释这些选择。对有限个变量组成的集合，环境将值与其中的每个元素关联起来。在编程语言的实现之中，环境可用来维系变量与值的关系。编译器也能用环境将变量名与变量信息关联起来。

只要能够检查两个变量是否相等，变量能够用我们喜欢的任何方式表示。我们选用 Scheme 符号表示变量，但在没有符号数据类型的语言中，变量也可以用字符串，哈希表引用，甚至数字表示（见 3.6 节）。

2.2.1 环境的接口

环境是一函数，定义域为有限个变量的集合，值域为所有 Scheme 值的集合。数学上常说的有限函数是指有序数对组成的有限集合，我们采用这一含义，就得表示形如 $\{(var_1, val_1), ..., (var_n, val_n)\}$ 的所有集合。其中，var_i 是某一变量，val_i 是任意 Scheme 值。有时称环境 env 中变量 var 的值 val 为其在 env 中的绑定 (*binding*)。

这一数据类型的接口有三个过程，定义如下：

$$
\begin{aligned}
&\texttt{(empty-env)} && = \lceil\emptyset\rceil \\
&\texttt{(apply-env } \lceil f\rceil \ \ var\texttt{)} && = f(var) \\
&\texttt{(extend-env } var \ \ v \ \lceil f\rceil\texttt{)} && = \lceil g\rceil
\end{aligned}
$$

$$
\text{其中，}\quad g(var_1) = \begin{cases} v & \text{若 } var_1 = var \\ f(var_1) & \text{否则} \end{cases}
$$

过程 `empty-env` 不带参数，必须返回空环境的表示；`apply-env` 用环境对变量求值；`(extend-env` var val env`)` 产生一个新环境，将变量 var 的值设为 val，此外与 env 相同。例如，表达式

```
> (define e
    (extend-env 'd 6
      (extend-env 'y 8
        (extend-env 'x 7
          (extend-env 'y 14
            (empty-env))))))
```

定义了一个环境 e，使 $e(\mathtt{d}) = 6$，$e(\mathtt{x}) = 7$，$e(\mathtt{y}) = 8$，且对任何其他变量，e 未定义。本例中，`y` 先绑定到 14，随后绑定到 8。这当然只是生成该环境的众多方法之一。

如同前一个例子，可以将接口中的过程分为构造器和观测器。本例中，`empty-env` 和 `extend-env` 是构造器，`apply-env` 是唯一的观测器。

练习 2.4 [⋆⋆] 考虑数据类型*栈* (*stack*)，其接口包含过程 `empty-stack`、`push`、`pop`、`top` 和 `empty-stack?`。按照示例中的方式写出这些操作的定义。哪些操作是构造器？哪些是观测器？

2.2.2 数据结构表示法

观察可知，每个环境都能从空环境开始，n 次调用 `extend-env` 得到，其中 $n \geqslant 0$。例如，

```
(extend-env var_n val_n
  ...
  (extend-env var_1 val_1
    (empty-env)))
```

由此可得一种环境的表示方法。

每个环境都能用下列语法定义的表达式生成：

Env-exp ::= `(empty-env)`
::= `(extend-env` *Identifier* *Scheme-value* *Env-exp*`)`

可以用描述列表集合的语法表示环境，由此得出中的实现。过程 `apply-env` 查看表示环境的数据结构 `env`，判断它表示哪种环境，并做适当操作。如果它表示空环境，那就报错；如果它表示 `extend-env` 生成的环境，那就判

断要查找的变量是否与环境中绑定的某一变量相同，如果是，则返回保存的值，否则在保存的环境中查找变量。

这是一种常见的代码模式。我们叫它*解释器秘方* (*interpreter recipe*)：

解释器秘方

1. 查看一段数据。
2. 判断它表示什么样的数据。
3. 提取数据的各个部分，对它们做适当操作。

Env = `(empty-env)` | `(extend-env` *Var SchemeVal Env*`)`
Var = *Sym*

empty-env : () → *Env*

```
(define empty-env
  (lambda () (list 'empty-env)))
```

extend-env : *Var* × *SchemeVal* × *Env* → *Env*

```
(define extend-env
  (lambda (var val env)
    (list 'extend-env var val env)))
```

apply-env : *Env* × *Var* → *SchemeVal*

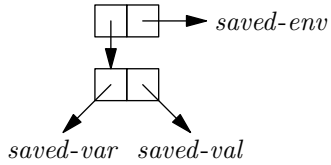
```
(define apply-env
  (lambda (env search-var)
    (cond
      ((eqv? (car env) 'empty-env)
        (report-no-binding-found search-var))
      ((eqv? (car env) 'extend-env)
        (let ((saved-var (cadr env))
              (saved-val (caddr env))
              (saved-env (cadddr env)))
          (if (eqv? search-var saved-var)
            saved-val
            (apply-env saved-env search-var))))
      (else
        (report-invalid-env env)))))

(define report-no-binding-found
  (lambda (search-var)
    (eopl:error 'apply-env "~s 未绑定" search-var)))

(define report-invalid-env
  (lambda (env)
    (eopl:error 'apply-env "非法环境: ~s" env)))
```

图 2.1 环境的数据结构表示

练习 2.5 [⋆] 只要能区分空环境和非空环境，并能从后者中提取出数据片段，就能用任何数据结构表示环境。按这种方式实现环境：空环境由空列表表示，`extend-env` 生成如下环境：



这叫 *a-list* 或关联列表 (*association-list*) 表示法。

练习 2.6 [⋆] 发明三种以上的环境接口表示，给出实现。

练习 2.7 [⋆] 重写中的 `apply-env`，给出更详细的错误信息。

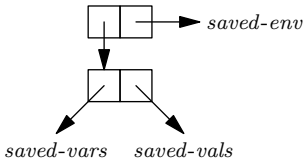
练习 2.8 [⋆] 给环境接口添加观测器 `empty-env?`，用 a-list 表示法实现它。

练习 2.9 [⋆] 给环境接口添加观测器 `has-binding?`，它取一环境 env，一个变量 s，判断 s 在 env 中是否有绑定值。用 a-list 表示法实现它。

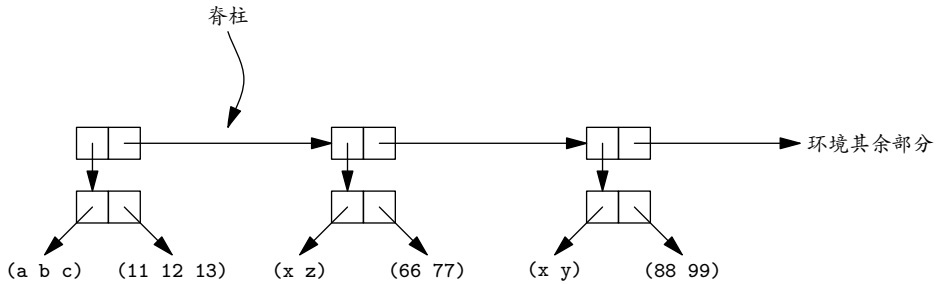
练习 2.10 [⋆] 给环境接口添加构造器 `extend-env*`，用 a-list 表示法实现它。这个构造器取一变量列表和一长度相等的值列表，以及一环境，其定义为：

$$\texttt{(extend-env* (}var_1\ \ldots\ var_k\texttt{) (}val_1\ \ldots\ var_k\texttt{) }\lceil f\rceil\texttt{)} = \lceil g\rceil,$$
$$\text{其中，}g(var) = \begin{cases} val_i & \text{若 } var = var_i \text{ 对某个 } i \text{ 成立，} 1 \leqslant i \leqslant k \\ f(var) & \text{否则} \end{cases}$$

练习 2.11 [⋆⋆] 前一题中的 `extend-env*` 实现比较拙劣的话，运行时间与 k 成正比。有一种表示可使 `extend-env*` 的运行时间为常数：用空列表表示空环境，用下面的数据结构表示非空环境：



那么一个环境看起来像是这样：



这叫做*肋排* (*ribcage*) 表示法。环境由名为*肋骨* (*rib*) 的序对列表表示；每根左肋是变量列表，右肋是对应的值列表。

用这种表示实现 `extend-env*` 和其他环境接口。

2.2.3 过程表示法

环境接口有一条重要性质：它只有 `apply-env` 一个观测器。这样就能用取一变量，返回绑定值的 Scheme 过程表示环境。

要这样表示，定义 `empty-env` 和 `extend-env` 的返回值为过程，调用二者的返回值就如同调用上一节的 `apply-env` 一样。由此得出下面的实现。

$$Env = Var \to SchemeVal$$
$$Var = Sym$$

empty-env : $() \to Env$

```
(define empty-env
  (lambda ()
    (lambda (search-var)
      (report-no-binding-found search-var))))
```

extend-env : $Var \times SchemeVal \times Env \to Env$

```
(define extend-env
  (lambda (saved-var saved-val saved-env)
    (lambda (search-var)
      (if (eqv? search-var saved-var)
        saved-val
        (apply-env saved-env search-var)))))
```

apply-env : $Env \times Var \to SchemeVal$

```
(define apply-env
  (lambda (env search-var)
    (env search-var)))
```

`empty-env` 创建的空环境收到任何变量都会报错，表明给定的变量不在其中。过程 `extend-env` 返回的过程代表扩展而得的环境。这个过程收到变量 `search-var` 后，判断该变量是否与环境中绑定的相同。如果相同，就返回保存的值；否则，就在保存的环境中查找它。

这种表示法中，数据由 `apply-env` 执行的动作表示，我们称之为*过程表示法* (*procedural representation*)。

数据类型只有一个观测器的情形并非想象中那般少见。比如，当数据是一组函数，就能用调用时执行的动作表示。这种情况下，可以按照下列步骤提炼出接口和过程表示法：

1. 找出客户代码中求取指定类型值的 lambda 表达式。为每个这样的 lambda 表达式写一个构造器过程。构造器过程的参数用作 lambda 表达式中的自由变量。在客户代码中，用构造器调用替换对应的 lambda 表达式。

2. 像定义 `apply-env` 那样定义一个 `apply-` 过程。找出客户代码中所有使用指定类型值的地方，包括构造器过程的主体。所有使用指定类型值的地方都改用 `apply-` 过程。

一旦完成这些步骤，接口就包含所有的构造器过程和 `apply-` 过程，客户代码则与表示无关：它不再依赖表示，我们将能随意换用另一套接口实现，正如 2.2.2 节所述。

如果用于实现的语言不支持高阶过程，那就得再做一些步骤，用数据结构表示法和解释器秘方实现所需接口，就像上一节那样。这一操作叫做消函(*defunctionalization*)。环境的数据结构表示中，各种变体都是消函的简单例子。过程表示法和消函表示法的关系将是本书反复出现的主题。

练习 2.12 [⋆] 用过程表示法实现 ex2.4 中的栈数据类型。

练习 2.13 [⋆⋆] 扩展过程表示法，用两个过程组成的列表表示环境，实现 `empty-env?`。一个过程像前面那样，返回变量的绑定值；一个返回环境是否为空。

练习 2.14 [⋆⋆] 扩展前一题中的表示法，加入第三个过程，用它来 `has-binding?`（见 ex2.9）。

2.3 递推数据类型的接口

第 1 章大部分都在处理递推数据类型。例如，给出了 lambda 演算表达式的语法：

$$\begin{aligned} \textit{Lc-Exp} &::= \textit{Identifier} \\ &::= \texttt{(lambda (}\textit{Identifier}\texttt{)}\ \textit{Lc-Exp}\texttt{)} \\ &::= \texttt{(}\textit{Lc-Exp}\ \ \textit{Lc-Exp}\texttt{)} \end{aligned}$$

我们还写出了过程 `occurs-free?`。像当时提到的，1.2.4 节中 `occurs-free?` 的定义不大容易读懂。比如，很难搞明白 `(car (cadr exp))` 指代 `lambda` 表达式中的变量声明，或者 `(caddr exp)` 指代表达式的主体。

要改善这种情况，可以给 lambda 演算表达式添加一套接口。我们的接口有几个构造器，以及两种观测器：谓词和提取器。

构造器有：

var-exp： $\textit{Var} \to \textit{Lc-Exp}$
lambda-exp： $\textit{Var} \times \textit{Lc-Exp} \to \textit{Lc-Exp}$
app-exp： $\textit{Lc-Exp} \times \textit{Lc-Exp} \to \textit{Lc-Exp}$

谓词有：

var-exp?： *Lc-Exp* → *Bool*
lambda-exp?： *Lc-Exp* → *Bool*
app-exp?： *Lc-Exp* → *Bool*

提取器有：

var-exp->var： *Lc-Exp* → *Var*
lambda-exp->bound-var： *Lc-Exp* → *Var*
lambda-exp->body： *Lc-Exp* → *Lc-Exp*
app-exp->rator： *Lc-Exp* → *Lc-Exp*
app-exp->rand： *Lc-Exp* → *Lc-Exp*

每个提取器对应 lambda 演算表达式中的一部分。现在可以写出一版只依赖接口的 `occurs-free?`。

occurs-free? : $Sym \times LcExp \rightarrow Bool$

```
(define occurs-free?
  (lambda (search-var exp)
    (cond
      ((var-exp? exp) (eqv? search-var (var-exp->var exp)))
      ((lambda-exp? exp)
        (and
          (not (eqv? search-var (lambda-exp->bound-var exp)))
          (occurs-free? search-var (lambda-exp->body exp))))
      (else
        (or
          (occurs-free? search-var (app-exp->rator exp))
          (occurs-free? search-var (app-exp->rand exp)))))))
```

只要使用上述构造器，怎样表示 lambda 演算表达式都可以。
我们可以写出设计递推数据类型接口的一般步骤：

设计递推数据类型的接口

1. 为数据类型的每种变体加入一个构造器。
2. 为数据类型的每种变体加入一个谓词。
3. 为传给数据类型构造器的每段数据加入一个提取器。

练习 2.15 [⋆] 上述语法指定了 lambda 演算表达式的表示方式，实现其接口。

练习 2.16 [⋆] 修改实现，换用另一种表示，去掉 `lambda` 表达式绑定变量周围的括号。

练习 2.17 [⋆] 再发明至少两种方式来表示数据类型 lambda 演算表达式，实现它们。

练习 2.18 [⋆] 我们常用列表表示值的序列。在这种表示法中，很容易从序列中的一个元素移动到下一个，但是不借助上下文参数，很难从一个元素移动到上一个。实现非空双向整数序列，语法为：

$$\mathit{NodeInSequence} ::= (\mathit{Int}\ \ \mathit{Listof}(\mathit{Int})\ \ \mathit{Listof}(\mathit{Int}))$$

第一个整数列表是当前元素之前的序列，反向排列。第二个列表是当前元素之后的序列。例如，`(6 (5 4 3 2 1) (7 8 9))` 表示列表 `(1 2 3 4 5 6 7 8 9)`，当前元素为 6。

用这种表示实现过程 `number->sequence`，它取一数字，生成只包含该数字的序列。接着实现 `current-element`、`move-to-left`、`move-to-right`、`insert-to-left`、`insert-to-right`、`at-left-end?` 和 `at-right-end?`。

例如：

```
> (number->sequence 7)
```

```
'(7 () ())
> (current-element '(6 (5 4 3 2 1) (7 8 9)))
6
> (move-to-left '(6 (5 4 3 2 1) (7 8 9)))
'(5 (4 3 2 1) (6 7 8 9))
> (move-to-right '(6 (5 4 3 2 1) (7 8 9)))
'(7 (6 5 4 3 2 1) (8 9))
> (insert-to-left 13 '(6 (5 4 3 2 1) (7 8 9)))
'(6 (13 5 4 3 2 1) (7 8 9))
> (insert-to-right 13 '(6 (5 4 3 2 1) (7 8 9)))
'(6 (5 4 3 2 1) (13 7 8 9))
```

如果参数在序列最右端，过程 `move-to-right` 应失败。如果参数在序列最左端，过程 `move-to-left` 应失败。

练习 2.19 [⋆] 空二叉树和用整数标记内部节点的二叉树可以用语法表示为：

$$BinTree ::= \texttt{()} \mid \texttt{(}Int\ \ BinTree\ \ BinTree\texttt{)}$$

用这种表示实现过程 `number->bintree`，它取一个整数，产生一棵新的二叉树，树的唯一节点包含该数字。接着实现 `current-element`、`move-to-left-son`、`move-to-right-son`、`at-leaf?`、`insert-to-left` 和 `insert-to-right`。例如：

```
> (number->bintree 13)
'(13 () ())
> (define t1 (insert-to-right 14
               (insert-to-left 12
                 (number->bintree 13))))
> t1
'(13 (12 () ()) (14 () ()))
> (move-to-left-son t1)
'(12 () ())
> (current-element (move-to-left-son t1))
12
> (at-leaf? (move-to-right-son (move-to-left-son t1)))
```

```
#t
> (insert-to-left 15 t1)
'(13 (15 (12 () ()) ()) (14 () ()))
```

练习 2.20 [⋆⋆⋆] 按照 ex2.19 中的二叉树表示，很容易从父节点移到某个子节点，但是不借助上下文参数，无法从子节点移动到父节点。扩展 ex2.18 中的列表表示法，用以表示二叉树中的节点。提示：想想怎样用逆序列表表示二叉树在当前节点以上的部分，就像 ex2.18 那样。

用这种表示实现 ex2.19 中的过程。接着实现 `move-up` 和 `at-root?`。

2.4 定义递推数据类型的工具

对复杂的数据类型，按照上述步骤设计接口很快就会使人厌倦。本节介绍用 Scheme 自动设计和实现接口的工具。这个工具产生的接口与前一节的虽不完全相同，却很类似。

仍考虑前一节讨论的数据类型 lambda 演算表达式。lambda 演算表达式的接口可以这样写：

```
(define-datatype lc-exp lc-exp?
  (var-exp
   (var identifier?))
  (lambda-exp
   (bound-var identifier?)
   (body lc-exp?))
  (app-exp
   (rator lc-exp?)
   (rand lc-exp?)))
```

这里的名字 `var-exp`、`var`、`bound-var`、`app-exp`、`rator` 和 `rand` 分别是变量表达式 (*variable expression*)、变量 (*variable*)、绑定变量 (*bound variable*)、调用表达式 (*application expression*)、操作符 (*operator*) 和操作数 (*operand*) 的缩写。

这些表达式声明了三种构造器：`var-exp`、`lambda-exp` 和 `app-exp`，以及一个谓词 `lc-exp?`。三个构造器用谓词 `identifier?` 和 `lc-exp?` 检查它们的参数，确保参数合法。所以，如果生成 lc-exp 时只用这些构造器，可以确保表达式及其所有子表达式合法。如此一来，处理 lambda 表达式时就能跳过许多检查。

我们用形式 `cases` 代替谓词和提取器，判断数据类型的实例属于哪种变体，并提取出它的组件。为解释这一形式，我们用数据类型 `lc-exp` 重写 `occurs-free?`（occurs-free）：

occurs-free? : $Sym \times LcExp \to Bool$

```
(define occurs-free?
  (lambda (search-var exp)
    (cases lc-exp exp
      (var-exp
        (var) (eqv? var search-var))
      (lambda-exp
        (bound-var body)
        (and
          (not (eqv? search-var bound-var))
          (occurs-free? search-var body)))
      (app-exp
        (rator rand)
        (or
          (occurs-free? search-var rator)
          (occurs-free? search-var rand))))))
```

要理解它，假设 `exp` 是由 `app-exp` 生成的 lambda 演算表达式。根据 `exp` 的取值，分支 `app-exp` 将被选中，`rator` 和 `rand` 则绑定到两个子表达式，接着，表达式

```
(or
  (occurs-free? search-var rator)
  (occurs-free? search-var rand))
```

将会求值，就像我们写：

```
(if (app-exp? exp)
```

```
    (let ((rator (app-exp->rator exp))
          (rand (app-exp->rand exp)))
      (or
        (occurs-free? search-var rator)
        (occurs-free? search-var rand)))
  ...)
```

递归调用 `occurs-free?` 像这样完成运算。

一般的 `define-datatype` 声明形如：

```
(define-datatype type-name type-predicate-name
  {(variant-name {(filed-name predicate)}*)}+)
```

这新定义了一种数据类型，名为 *type-name*，它有一些变体 (*variants*)。每个变体有一变体名，以及 0 或多个字段，每个字段各有其字段名和相应的谓词。不论是否属于不同的类型，变体都不能重名。类型也不能重名，且类型名不能用作变体名。每个字段的谓词必须是一个 Scheme 谓词。

每个变体都有一个构造器过程，用于创建该变体的值。这些过程的名字与对应的变体相同。如果一个变体有 n 个字段，那么它的构造器取 n 个参数，用对应的谓词检查每个参数值，并返回变体值，值的第 i 个字段为第 i 个参数值。

type-predicate-name 绑定到一个谓词。这个谓词判断其参数值是否是相应的类型。

记录表 (*record*) 可以用只有一种变体的数据类型定义。为了区分只有一种变体的数据类型，我们遵循一种命名惯例：当只有一个变体时，我们以 `a-`*type-name* 或 `an-`*type-name* 命名构造器；否则，以 *variant-name*-*type-name* 命名构造器。

由 `define-datatype` 生成的数据结构可以互递归。例如， 1.1 节中的 s-list 语法为：

$$\begin{aligned} \textit{S-list} &::= (\{\textit{S-exp}\}^*) \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list} \end{aligned}$$

s-list 中的数据可以用数据类型 `s-list` 表示为：

```
(define-datatype s-list s-list?
  (empty-s-list)
  (non-empty-s-list
    (first s-exp?)
    (rest s-list?)))

(define-datatype s-exp s-exp?
  (symbol-s-exp
    (sym symbol?))
  (s-list-s-exp
    (slst s-list?)))
```

数据类型 `s-list` 用 `(empty-s-list)` 和 `non-empty-s-list` 代替 `()` 和 `cons` 来表示列表。如果我们还想用 Scheme 列表，可以写成：

```
(define-datatype s-list s-list?
  (an-s-list
    (sexps (list-of s-exp?))))

(define list-of
  (lambda (pred)
    (lambda (val)
      (or (null? val)
        (and (pair? val)
          (pred (car val))
          ((list-of pred) (cdr val)))))))
```

这里 `(list-of` *pred*`)` 生成一个谓词，检查其参数值是否是一个列表，且列表的每个元素都满足 *pred*。

`cases` 语法的一般姓形式为：

```
(cases type-name expression
  {(variant-name ({filed-name}*) consequent)}*
  (else default))
```

该形式指定类型，一个待求值和检查的表达式，以及一些从句。每个从句以指定类型的某一变体名及相应字段名为标识。`else` 从句可有可无。首先求 *expression* 的值，得到 *type-name* 的某个值 v。如果 v 是某个 *variant-name* 的变体，那就选中对应的从句。各 *type-name* 绑定到 v 中对应的字段值。然后在

这些绑定的作用域内求取并返回 *consequent* 的值。如果 v 不属于任何变体，且有 `else` 从句，则求取并返回 *default* 的值。如果没有 `else` 从句，必须为指定数据类型的每个变体指定从句。

形式 `cases` 根据位置绑定变量：第 i 个变量绑定到第 i 个字段。所以，我们可以用：

```
(app-exp (exp1 exp2)
  (or
    (occurs-free? search-var exp1)
    (occurs-free? search-var exp2)))
```

代替

```
(app-exp (rator rand)
  (or
    (occurs-free? search-var rator)
    (occurs-free? search-var rand)))
```

`define-datatype` 和 `cases` 形式提供了一种简洁的方式来定义递推数据类型，但这种方式并不是唯一的。根据使用场景，可能得用专门的表示方式，它们利用数据的特殊性质，更紧凑或者更高效。获得这些优势的代价是必须动手实现接口中的过程。

`define-datatype` 形式是特定领域语言 (*domain-specific language*) 的例子。特定领域语言是一种小巧的语言，用来描述小而明确的任务中的单一任务。本例中的任务是定义一种递推数据类型。这种语言可能像 `define-datatype` 一样，存在于通用语言中；也可能是一门单独的语言，别有一套工具。一般来说，创造这类语言首先要找出任务的不同变体，然后设计语言，描述这些变体。这种策略通常非常有效。

练习 2.21 [⋆] 用 `define-datatype` 实现 2.2.2 节中的环境数据类型。然后实现 ex2.9 中的 `has-binding?`。

练习 2.22 [⋆] 用 `define-datatype` 实现 ex2.4 中的栈数据类型。

练习 2.23 [⋆] `lc-exp` 的定义忽略了中的条件：“*Identifier* 是除 `lambda` 之外的

任何符号。”修改 identifier? 的定义，补充这一条件。提示：任何谓词都能在 define-datatype 中使用，你定义的也能。

练习 2.24 [⋆] 这是用 define-datatype 表示的二叉树：

```
(define-datatype bintree bintree?
  (leaf-node
   (num integer?))
  (interior-node
   (key symbol?)
   (left bintree?)
   (right bintree?)))
```

实现操作二叉树的过程 bintree-to-list，则 (bintree-to-list (interior-node 'a (leaf-node 3) (leaf-node 4))) 应返回列表：

```
(interior-node
  a
  (leaf-node 3)
  (leaf-node 4))
```

练习 2.25 [⋆⋆] 用 cases 写出 max-interior，它取至少有一个内部节点的整数二叉树（像前一道练习那样），返回叶子之和最大的内部节点对应的标签。

```
> (define tree-1
    (interior-node 'foo (leaf-node 2) (leaf-node 3)))
> (define tree-2
    (interior-node 'bar (leaf-node -1) tree-1))
> (define tree-3
    (interior-node 'baz tree-2 (leaf-node 1)))
> (max-interior tree-2)
'foo
> (max-interior tree-3)
'baz
```

最后一次调用 `max-interior` 也可能返回 `foo`，因为节点 `foo` 和 `baz` 的叶子之和都为 5。

练习 2.26 [⋆⋆] ex1.33 还有一种写法。树的集合可以用下列语法定义：

$$
\begin{aligned}
\textit{Red-blue-tree} &::= \textit{Red-blue-subtree} \\
\textit{Red-blue-subtree} &::= \texttt{(red-node } \textit{Red-blue-subtree}\ \ \textit{Red-blue-subtree}\texttt{)} \\
&::= \texttt{(blue-node } \{\textit{Red-blue-subtree}\}^{*}\texttt{)} \\
&::= \texttt{(leaf-node } \textit{Int}\texttt{)}
\end{aligned}
$$

用 `define-datatype` 写出等价定义，用得到的接口写出一个过程，它取一棵树，生成形状相同的另一棵树，但把每片叶子的值改为从当前叶子节点到树根之间红色节点的数目。

2.5 抽象语法及其表示

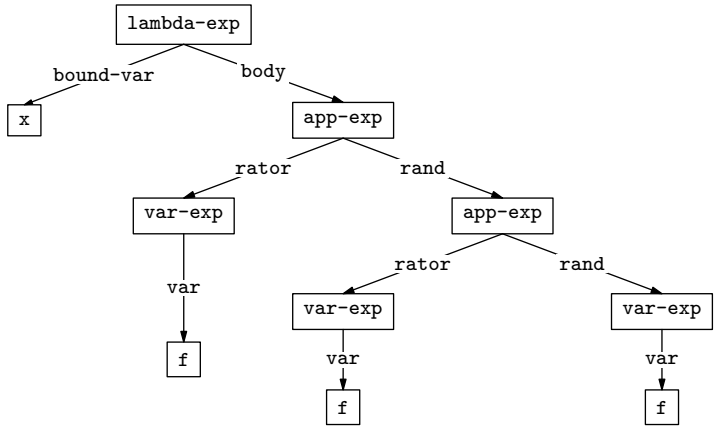


图 2.2 `(lambda (x) (f (f x)))` 的抽象语法树

语法通常指定归纳式数据类型的某一具体表示，后者使用前者生成的字符串或值。这种表示叫做*具体语法* (*concrete syntax*)，或*外在* (*external*) 表示。

例如，指定集合 lambda 演算表达式，用的就是 lambda 演算表达式的具体语法。我们可以用其他具体语法表示 lambda 演算表达式。例如，可以用

$$
\begin{aligned}
\textit{Lc-exp} &::= \textit{Identifier} \\
&::= \texttt{proc}\ \textit{Identifier}\ \texttt{=>}\ \textit{Lc-exp} \\
&::= \textit{Lc-exp}\ \ \texttt{(}\textit{Lc-exp}\texttt{)}
\end{aligned}
$$

把 lambda 演算表达式定义为另一个字符串集合。

为了处理这样的数据，需要将其转换为*内在* (*internal*) 表示。`define-datatype` 形式提供了一种简洁的方式来定义这样的内在表示。我们称之为*抽象语法* (*abstract syntax*)。在抽象语法中，不需要存储括号之类的终止符，因为它们不传达信息。另一方面，我们要确保数据结构足以区分它所表示的 lambda 演算表达式，并提取出各部分。lc-exp 的数据类型 `lc-exp` 助我们轻松实现这些。

将内在表示形象化为*抽象语法树* (*abstract syntax tree*) 也很不错。展示了一棵抽象语法树，它代表数据类型 `lc-exp` 表示的 lambda 演算表达式 `(lambda (x) (f (f x)))`。树的每个内部节点以相应的生成式名字为标识。树枝以所出现的非终结符名字为标识。叶子对应终止符字符串。

要为某种具体语法设计抽象语法，需要给其中的每个生成式，以及生成式中出现的每个非终止符命名。很容易将抽象语法写成 `define-datatype` 声明。我们为每个非终结符添加一个 `define-datatype`，为每个生成式添加一个变体。

中挑出的内容可以精确表示如下：

Lc-Exp ::= *Identifier*
`var-exp (var)`

::= `(lambda (`*Identifier*`)` *Lc-Exp*`)`
`lambda-exp (bound-var body)`

::= `(`*Lc-Exp* *Lc-Exp*`)`
`app-exp (rator rand)`

本书采用这种表示，同时指明具体语法和抽象语法。

具体语法主要供人使用，抽象语法主要供计算机使用，既已区分二者，现在来看看如何将一种语法转换为另一种。

当具体语法是个字符串集合，推导出对应的抽象语法树可能相当棘手。这一任务叫做*解析* (*parsing*)，由*解析器* (*parser*) 完成。因为写解析器通常比较麻烦，所以最好借由工具*解析器生成器* (*parser generator*) 完成。解析器生成器以一套语法作为输入，产生一个解析器。由于语法是由工具处理的，它们必需以某种机器能够理解的语言写成，即写语法用的特定领域语言。有很多现成的解析器生成器。

如果具体语法以列表集合的形式给出，解析过程就会大大简化。比如，和 define-datatype `define-datatype` 的语法类似，本节开头的 lambda 演算表达式指定了一个列表集合。这样，Scheme 过程 `read` 会自动把字符串解析为列表和符号。然后，把这些列表结构解析为抽象语法树就容易多了，就像 `parse-expression` 这样。

parse-expression : *SchemeVal* → *LcExp*

```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
        (if (eqv? (car datum) 'lambda)
          (lambda-exp
            (car (cadr datum))
```

```
            (parse-expression (caddr datum)))
          (app-exp
            (parse-expression (car datum))
            (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum)))))
```

通常，很容易把抽象语法树重新转换为列表-符号表示。我们这样做了，Scheme 的打印过程就会将其显示为列表形式的具体语法。这由 `unparse-lc-exp` 完成：

unparse-lc-exp : *LcExp* → *SchemeVal*

```
(define unparse-lc-exp
  (lambda (exp)
    (cases lc-exp exp
      (var-exp (var) var)
      (lambda-exp
        (bound-var body)
        (list 'lambda
          (list bound-var)
          (unparse-lc-exp body)))
      (app-exp
        (rator rand)
        (list (unparse-lc-exp rator)
          (unparse-lc-exp rand))))))
```

练习 2.27 [⋆] 画出下面 lambda 演算表达式的抽象语法树：

```
((lambda (a) (a b)) c)

(lambda (x)
  (lambda (y)
    ((lambda (x)
       (x y))
     x)))
```

练习 2.28 [⋆] 写出反向解析器，将 lc-exp 的抽象语法转换为符合本节第二个语法（lc-grammar2）的字符串。

练习 2.29 [⋆] 当具体语法使用克莱尼星号或加号（kleene-star）时，生成抽象语法树时最好使用相应子树的列表。例如，如果 lambda 演算表达式的语法为：

$$
\begin{aligned}
\textit{Lc-Exp} ::= & \ \textit{Identifier} \\
& \boxed{\texttt{var-exp (var)}} \\
::= & \ \texttt{(lambda (}\{\textit{Identifier}\}^*\texttt{)}\ \textit{Lc-Exp}\texttt{)} \\
& \boxed{\texttt{lambda-exp (bound-vars body)}} \\
::= & \ \texttt{(}\textit{Lc-Exp}\ \ \{\textit{Lc-Exp}\}^*\texttt{)} \\
& \boxed{\texttt{app-exp (rator rands)}}
\end{aligned}
$$

那么字段 `bound-vars` 的谓词可写作 `(list-of identifier?)`，`rands` 的谓词可写作 `(list-of lc-exp?)`。以这种方式写出该语法的 `define-datatype` 和解析器。

练习 2.30 [⋆⋆] 上面定义的过程 `parse-expression` 很不可靠：它检查不到某些可能的语法错误，例如 `(a b c)`，并且因其他表达式终止时给不出恰当的错误信息，如 `(lambda)`。修改一下，使之更健壮，可接受任何 s-exp，并且对不表示 lambda 演算表达式的 s-exp 给出恰当的错误信息。

练习 2.31 [⋆⋆] 有时，把具体语法定义为括号包围的符号和整数序列很有用。例如，可以把集合前缀列表 (*prefix list*) 定义为：

$$
\begin{aligned}
\textit{Prefix-list} ::= & \ \texttt{(}\textit{Prefix-exp}\texttt{)} \\
\textit{Prefix-exp} ::= & \ \textit{Int} \\
::= & \ \texttt{-}\ \ \textit{Prefix-exp}\ \ \textit{Prefix-exp}
\end{aligned}
$$

那么 `(- - 3 2 - 4 - 12 7)` 是一个合法的前缀列表。有时为纪念其发明者 Jan •ukasiewicz，称之为波兰前缀表示法 (*Polish prefix notation*)。写一个解析器，将前缀列表表示法转换为抽象语法：

```
(define-datatype prefix-exp prefix-exp?
  (const-exp
   (num integer?))
  (diff-exp
   (operand1 prefix-exp?)
   (operand2 prefix-exp?)))
```

使上例与这几个构造器生成相同抽象语法树：

```
(diff-exp
 (diff-exp
  (const-exp 3)
  (const-exp 2))
 (diff-exp
  (const-exp 4)
  (diff-exp
   (const-exp 12)
   (const-exp 7))))
```

提示：想想如何写一个过程，取一列表，产生一个 `prefix-exp` 和列表剩余元素组成的列表。

3 表达式

本章研究变量绑定及其作用域。我们用一系列小型语言解释这些概念。我们为这些语言写出规范，遵照第 1 章的解释器秘方实现其解释器。我们的规范和解释器取一名为*环境* (*environment*) 的上下文参数，以记录待求值的表达式中各个变量的含义。

3.1 规范和实现策略

我们的规范包含若干断言，形如：

$$\texttt{(value-of } exp \ \rho\texttt{)} = val$$

意为在环境 ρ 中，表达式 exp 的值应为 val。我们像第 1 章那样，写出推理规则和方程，以便推导出这样的断言。我们手写规则和方程，找出某些表达式的期望值。

而我们的目标是写出程序，实现语言。概况如所示。首先是程序，由我们要实现的语言写出。这叫做*源语言* (*source language*) 或*被定语言* (*defined language*)。前端接收程序文本（由源语言写成的程序），将其转化为抽象语法树，然后将语法树传给解释器。解释器是一程序，它查看一段数据结构，根据结构执行一些动作。当然，解释器自身也由某种语言写成。我们把那种语言叫做*实现语言* (*implementation language*) 或*定义语言* (*defining language*)。我们的大多数实现都遵照这种方式。

另一种常见的组织方式如所示。其中，编译器替代了解释器，将抽象语法树翻译为另一种语言（称为*目标语言* (*target language*)）写成的程序，然后执行。目标语言可能像那样，由一个解释器执行，也可能翻译成更底层的语言执行。

通常，目标语言是一种机器语言，由硬件解释。但目标语言也可能是一种特定用途的语言，比原本的语言简单，为它写一个解释器相对容易。这样，程序可以编译一次，然后在多种不同的硬件平台上执行。出于历史原因，常称这样的目标语言为*字节码* (*byte code*)，称其解释器称为*虚拟机* (*virtual machine*)。

编译器常常分为两部分：*分析器* (*analyzer*)，尝试推断关于程序的有效信息；*翻译器* (*translator*)，执行翻译，可能用到来自分析器的信息。这些阶段既能用推理规则指定，也能用专写规范的语言指定。然后是实现。第 6 章和第 7 章探讨了一些简单的分析器和翻译器。

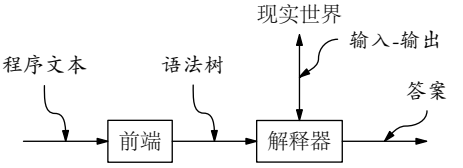
不论采用哪种实现策略，我们都需要一个*前端* (*front end*)，将程序转换为抽象语法树。因为程序只是字符串，我们的前端要将这些字符组成有意义的单元。分组通常分为两个阶段：*扫描* (*scanning*) 和*解析* (*parsing*)。

扫描就是将字符序列分为单词、数字、标点、注释等等。这些单元叫做*词条* (*lexical item*)、*词素* (*lexeme*)、或者最常见的*词牌* (*token*)。把程序分为词牌的方式叫做语言的*词法规范* (*lexical specification*)。扫描器取一字符序列，生成词牌序列。

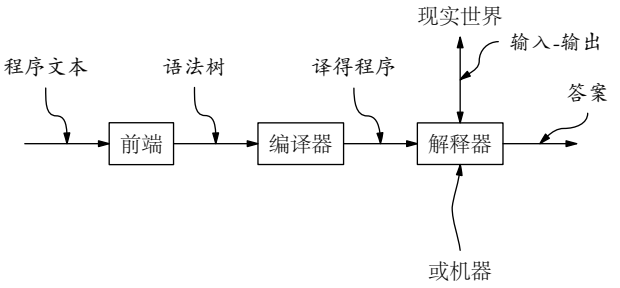
解析就是将词牌序列组成有层次的语法结构，如表达式、语句和块。这就像用从句组织（或称图解[1]）句子。我们称之为语言的*句法* (*syntactic*) 或*语法* (*grammatical*) 结构。解析器取一词牌序列（由扫描器给出），生成一棵抽象语法树。

设计前端的标准方式是使用*解析器生成器* (*parser generator*)。解析器生成器是一程序，取一词法规范和语法，生成一个扫描器和解析器。

[1]西方有 diagram sentence 之说，以树状图表示句子结构，如我国中学生学习英文之主、谓、宾。••译注



(a) 由解释器执行



(b) 由编译器执行

图 3.1 语言处理系统块状图

大多数主流语言都有解析器生成系统。如果没有解析器生成器，或者没有合适的，可以手写扫描器和解析器。编译器教材描述了这一过程。本书使用的解析技术及相关语法设计从简，专为满足我们的需求。

另一种方式是忽略具体语法的细节，把表达式写成列表结构，就像 2.5 节和 ex2.31 中，处理 lambda 演算表达式那样。

3.2 LET：一门简单语言

我们先来定义一种非常简单的语言，根据它最有趣的特性，将其命名为 LET。

3.2.1 定义语法

展示了我们这门简单语言的语法。在这种语言中，程序只能是一个表达式。表达式是个整数常量、差值表达式、判零表达式、条件表达式、变量、或 `let` 表达式。

这里是本门语言写成的一个简单表达式，及其抽象语法表示。

```
(scan&parse "-(55, -(x,11))")
#(struct:a-program
  #(struct:diff-exp
    #(struct:const-exp 55)
    #(struct:diff-exp
      #(struct:var-exp x)
      #(struct:const-exp 11))))
```

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `(-` *Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `(zero?` *Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

图 3.2 LET 语言的语法

3.2.2 定义值

任何编程语言的规范之中，最重要的一部分就是语言能处理的值的集合。每种语言至少有两个这样的集合：表达值 (*expressed values*) 和指代值 (*denoted values*)。表达值是指表达式可能的取值，指代值是指可以绑定到变量的值。

本章的语言中，表达值和指代值总是相同。它们是：

$$ExpVal = Int + Bool$$
$$DenVal = Int + Bool$$

第 4 章展示表达值和指代值不同的语言。

要使用这个定义，我们要有表达值数据类型的接口。我们有这几个接口：

num-val : $Int \to ExpVal$
bool-val : $Bool \to ExpVal$
expval->num : $ExpVal \to Int$
expval->bool : $ExpVal \to Bool$

我们假定，当传给 `expval->num` 的参数不是整数值，或传给 `expval->bool` 的参数不是布尔值时，二者行为未定义。

3.2.3 环境

若要求取表达式的值，我们得知道每个变量的值。我们靠环境记录这些值，就像 2.2 节那样。

环境是一函数，定义域为变量的有限集合，值域为指代值。我们用一些缩写表示环境。

- ρ 表示任一环境。
- [] 表示空环境。
- $[var = val]\rho$ 表示 `(extend-env` var val ρ`)`。
- $[var_1 = val_1, var_2 = val2]\rho$ 是 $var_1 = val_1([var_2 = val_2]\rho)$ 的缩写，其余同理。

- $[var_1 = val_1, var_2 = val2, \ldots]$ 表示的环境中，var_1 的值为 val_1，其余同理。

  我们偶尔用不同缩进表示复杂环境，以便阅读。例如，我们可能把

```
(extend-env 'x 3
  (extend-env 'y 7
    (extend-env 'u 5 ρ)))
```

缩写为

```
[x=3]
 [y=7]
  [u=5]ρ
```

3.2.4 定义表达式的行为

我们语言中的六种表达式各对应一个左边为 *Expression* 的生成式。表达式接口包含七个过程，六个是构造器，一个是观测器。我们用 *ExpVal* 表示表达值的集合。

构造器：

const-exp： $Int \to Exp$
zero?-exp： $Exp \to Exp$
if-exp： $Exp \times Exp \times Exp \to Exp$
diff-exp： $Exp \times Exp \to Exp$
var-exp： $Var \to Exp$
let-exp： $Var \times Exp \times Exp \to Exp$

观测器：

value-of： $Exp \times Env \to ExpVal$

实现之前，我们先写出这些过程的行为规范。依照解释器秘方，我们希望 `value-of` 查看表达式，判断其类别，然后返回恰当的值。

`(value-of (const-exp` n`) ρ) = (num-val` n`)`

```
(value-of (var-exp var) ρ) = (apply-env ρ var)

(value-of (diff-exp exp1 exp2) ρ)
= (num-val
    (-
      (expval->num (value-of exp1 ρ))
      (expval->num (value-of exp2 ρ))))
```

任何环境中，常量表达式的值都是这个常量。变量引用的值从某一环境中查询而得。差值表达式的值为第一个操作数在某一环境中的值减去第二个操作数在同一环境中的值。当然，准确来说，我们得确保操作数的值是数字，且结果是表示为表达值的数字。

展示了如何结合这些规则求取构造器生成的表达式的值。在本例以及其他例子中，我们用 $\bullet exp\bullet$ 表示表达式 exp 的抽象语法树，用 $\lceil n \rceil$ 表示 `(num-val` n`)`，用 $\lfloor val \rfloor$ 表示 `(expval->num` val`)`。我们还运用了一点事实：$\lfloor \lceil n \rceil \rfloor = n$。

练习 3.1 [⋆] 列出中所有应用 $\lfloor \lceil n \rceil \rfloor = n$ 的地方。

练习 3.2 [⋆⋆] 给出一个表达值 $val \in \mathit{ExpVal}$，且 $\lceil \lfloor val \rfloor \rceil \neq val$。

3.2.5 定义程序的行为

在我们的语言中，整个程序只是一个表达式。要找出这个表达式的值，我们要定义程序中自由变量的值。所以程序的值就是在适当的初始环境中求出的该表达式的值。我们把初始环境设为 `[i=1,v=5,x=10]`。

```
(value-of-program exp)
= (value-of exp [i=⌈1⌉,v=⌈5⌉,x=⌈10⌉])
```

3.2.6 定义条件

接下来是这门语言的布尔值接口。这门语言有一个布尔值构造器 `zero?`，一个布尔值观测器 `if` 表达式。

当且仅当操作数的值为 0，`zero?` 表达式的值为真。像那样，可将其写成一条推理规则。我们以 `bool-val` 为构造器，把布尔值转换为表达值；以 `expval->num` 为提取器，判断表达式的值是否为整数，如果是，则返回该整数。

令 ρ = [i=1,v=5,x=10]。

```
(value-of
  <<-(-(x,3), -(v,i))>>
  ρ)

= ⌈(-
    ⌊(value-of <<-(x,3)>> ρ)⌋
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      ⌊(value-of <<x>> ρ)⌋
      ⌊(value-of <<3>> ρ)⌋)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      10
      ⌊(value-of <<3>> ρ)⌋)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
      10
      3)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    7
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    7
    (-
      ⌊(value-of <<v>> ρ)⌋
      ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
    7
    (-
      5
      ⌊(value-of <<i>> ρ)⌋))⌉

= ⌈(-
    7
    (-
      5
      1))⌉

= ⌈(-
    7
    4)⌉

= ⌈3⌉
```

图 3.3 按照规范做简单运算

$$\frac{\texttt{(value-of}\ exp_1\ \rho\texttt{)} = val_1}{\texttt{(value-of (zero?-exp}\ exp_1\texttt{)}\ \rho\texttt{)} = \begin{cases}\texttt{(bool-val \#t)} & \text{若}\texttt{(expval->num}\ val_1\texttt{)} = 0\\ \texttt{(bool-val \#f)} & \text{若}\texttt{(expval->num}\ val_1\texttt{)} \neq 0\end{cases}}$$

一个 `if` 表达式就是一个布尔值观测器。欲求 `if` 表达式 `(if-exp` exp_1 exp_2 exp_3`)` 的值，首先判断子表达式 exp_1 的值；若该值为真，整个表达式的值应为子表达式 exp_2 的值，否则为子表达式 exp_3 的值。这也很容易写成推理规则。就像在前一个例子中使用 `expval->num` 一样，我们用 `expval->bool` 提取表达值的布尔部分。

$$\frac{\texttt{(value-of}\ exp_1\ \rho\texttt{)} = val_1}{\texttt{(value-of (if-exp}\ exp_1\ exp_2\ exp_3\texttt{)}\ \rho\texttt{)} = \begin{cases}\texttt{(value-of}\ exp_2\ \rho\texttt{)} & \text{若}\texttt{(expval->bool}\ val_1\texttt{)} = \texttt{\#t}\\ \texttt{(value-of}\ exp_3\ \rho\texttt{)} & \text{若}\texttt{(expval->bool}\ val_1\texttt{)} = \texttt{\#f}\end{cases}}$$

用这种推理规则很容易指定任意表达式的期望行为，但却不适合展示推理过程。像 `(value-of` exp_1 ρ`)` 这样的前件表示一部分计算，所以一个计算过程应该是一棵树，就像 deriv-tree 那种。很不幸的是，这样的树极为晦涩。因此，我们经常把规则转为方程，然后就能用相等代换展示计算过程。

`if-exp` 的方程式规范是：

```
(value-of (if-exp exp1 exp2 exp3) ρ)
= (if (expval->bool (value-of exp1 ρ))
    (value-of exp2 ρ)
    (value-of exp3 ρ))
```

展示了用这些规则进行简单运算的过程。

令 ρ = [x=⌈33⌉,y=⌈22⌉]。

```
(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)

= ⌈18⌉
```

图 3.4 条件表达式的简单计算过程

3.2.7 定义 let

接下来我们处理用 `let` 表达式创建新变量绑定的问题。我们给这门解释性语言添加语法，以关键字 `let` 起始，然后是一个声明，关键字 `in`，及其主体。例如，

```
let x = 5
in -(x, 3)
```

像 `lambda` 变量绑定一样（见 1.2.4 节），`let` 变量绑定于主体之中。

如同其主体，整个 `let` 形式也是一个表达式，所以 `let` 表达式可以嵌套，例如

```
let z = 5
in let x = 3
   in let y = -(x, 1)    % 这里 x = 3
      in let x = 4
         in -(z, -(x,y)) % 这里 x = 4
```

在本例中，第一个差值表达式中使用的 `x` 指代外层声明，另一个差值表达式中使用的 `x` 指代内层声明，所以整个表达式的值是 3。

`let` 声明的右边也是一个表达式，所以它可以任意复杂。例如

```
let x = 7
in let y = 2
   in let y = let x = -(x,1)
              in -(x,y)
      in -(-(x,8), y)
```

这里第三行声明的 `x` 绑定到 6, 所以 `y` 的值是 4, 整个表达式的值是 $((-1)-4) = -5$。

可以将规范写成一条规则。

$$\frac{\texttt{(value-of } exp_1 \ \rho\texttt{)} = val_1}{\begin{array}{c}\texttt{(value-of (let-exp } var \ exp_1 \ body\texttt{)} \ \rho\texttt{)} \\ = \texttt{(value-of } body \ \texttt{[}var\texttt{=}val_1\texttt{]}\rho\texttt{)}\end{array}}$$

像之前那样，将其转为方程通常更方便。

$$\texttt{(value-of (let-exp } var \ exp_1 \ body\texttt{)} \ \rho\texttt{)} = \texttt{(value-of } body \ \texttt{[var=(value-of } exp_1 \ \rho\texttt{)]}\rho\texttt{)}$$

展示了一个例子，其中 ρ_0 表示任意环境。

```
(value-of
  <<let x = 7
    in let y = 2
       in let y = let x = -(x,1) in -(x,y)
          in -(-(x,8),y)>>
  ρ0)

= (value-of
    <<let y = 2
      in let y = let x = -(x,1) in -(x,y)
         in -(-(x,8),y)>>
    [x=⌈7⌉]ρ0)

= (value-of
    <<let y = let x = -(x,1) in -(x,y)
      in -(-(x,8),y)>>
    [y=⌈2⌉][x=⌈7⌉]ρ0)
```

令 ρ_1 = [y=⌈2⌉][x=⌈7⌉]ρ_0。

```
= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<let x = -(x,1) in -(x,y)>> ρ1)]
    ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=(value-of <<-(x,1)>> ρ1)]ρ1)]
    ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=(value-of <<-(x,2)>> [x=⌈6⌉]ρ1)]
    ρ1)

= (value-of
    <<-(-(x,8),y)>>
    [y=⌈4⌉]ρ1)

= ⌈(- (- 7 8) 4)⌉

= ⌈-5⌉
```

图 3.5 let 一例

```
(define-datatype program program?
  (a-program
   (exp1 expression?)))

(define-datatype expression expression?
  (const-exp
   (num number?))
  (diff-exp
   (exp1 expression?)
   (exp2 expression?))
  (zero?-exp
   (exp1 expression?))
  (if-exp
   (exp1 expression?)
   (exp2 expression?)
   (exp3 expression?))
  (var-exp
   (var identifier?))
  (let-exp
   (var identifier?)
   (exp1 expression?)
   (body expression?)))
```

图 3.6 LET 语言的语法数据类型

3.2.8 实现 LET 规范

接下来的任务是用一组 Scheme 过程实现这一规范。我们的实现以 SLLGEN[1] 为前端，表达式用中的数据类型表示。在我们的实现中，表达值的表示如所示。数据类型声明了构造器 `num-val` 和 `bool-val`，用来将整数值和布尔值转换为表达值。我们还定义了提取器，用来将表达值转为整数或布尔值。如果表达值类型不符预期，提取器报错。

[1] 见 附录 B。••译注

```
(define-datatype expval expval?
  (num-val
   (num number?))
  (bool-val
   (bool boolean?)))
```

expval->num : *ExpVal* → *Int*

```
(define expval->num
  (lambda (val)
    (cases expval val
      (num-val (num) num)
      (else (report-expval-extractor-error 'num val)))))
```

expval->bool : *ExpVal* → *Bool*

```
(define expval->bool
  (lambda (val)
    (cases expval val
      (bool-val (bool) bool)
      (else (report-expval-extractor-error 'bool val)))))
```

图 3.7 LET 语言的表达值

只要满足 2.2 节中的定义，任意一种环境实现都可使用。过程 `init-env` 创建指定的初始环境，供 `value-of-program` 使用。

init-env : () → *Env*
用法: `(init-env)` = `[i=⌈1⌉,v=⌈5⌉,x=⌈10⌉]`

```
(define init-env
  (lambda ()
    (extend-env
     'i (num-val 1)
     (extend-env
      'v (num-val 5)
      (extend-env
       'x (num-val 10)
       (empty-env))))))
```

现在我们可以写出解析器，如和 fig-3.9 所示。主过程是 `run`，它取一个字符串，解析它，把结果传给 `value-of-program`。最令人感兴趣的过程是 `value-of`，它取一表达式和一环境，用解释器秘方计算规范所要求的答案。在代码中，我们插入了相关的推理规则定义，以便观察 `value-of` 的代码如何与规范对应。

在下面的练习以及全书之中，短语“扩展语言，添加……”表示向语言规范添加规则或者方程，并增改相应的解释器，实现指定特性。

run : *String* → *ExpVal*
```
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))
```

value-of-program : *Program* → *ExpVal*
```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

value-of : *ExpVal* × *Env* → *Bool*
```
(define value-of
  (lambda (exp env)
    (cases expression exp

      (value-of (const-exp n) ρ) = n
      (const-exp (num) (num-val num))

      (value-of (var-exp var) ρ) = (apply-env ρ var)
      (var-exp (var) (apply-env env var))

      $alignedat−1 (value-of (diff-exp exp1 exp2) ρ) = ⌈(- ⌊(value-of exp1 ρ)⌋
      (diff-exp (exp1 exp2)
        (let ((val1 (value-of exp1 env))
              (val2 (value-of exp2 env)))
          (let ((num1 (expval->num val1))
                (num2 (expval->num val2)))
            (num-val
              (- num1 num2)))))
```

图 3.8 LET 语言的解释器

(value-of exp_1 ρ) = val_1

(value-of (zero?-exp exp_1) ρ)

$$= \begin{cases} \texttt{(bool-val \#t)} & \text{若} \texttt{(expval->num } val_1\texttt{)} = 0 \\ \texttt{(bool-val \#f)} & \text{若} \texttt{(expval->num } val_1\texttt{)} \neq 0 \end{cases}$$

```
(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let (num1 (expval->num val1))
      (if (zero? num1)
        (bool-val #t)
        (bool-val #f)))))
```

(value-of exp_1 ρ) = val_1

(value-of (if-exp exp_1 exp_2 exp_3) ρ)

$$= \begin{cases} \texttt{(value-of } exp_2\ \rho\texttt{)} & \text{若} \texttt{(expval->bool } val_1\texttt{)} = \texttt{\#t} \\ \texttt{(value-of } exp_3\ \rho\texttt{)} & \text{若} \texttt{(expval->bool } val_1\texttt{)} = \texttt{\#f} \end{cases}$$

```
(if-exp (exp1 exp2 exp3)
  (if (expval->bool (value-of exp1 env))
    (value-of exp2 env)
    (value-of exp3 env)))
```

(value-of exp_1 ρ) = val_1

(value-of (let-exp var exp_1 $body$) ρ)

= (value-of $body$ [var=val_1]ρ)

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))))
```

图 3.9 LET 语言的解释器，续

练习 3.3 [⋆] 我们只有一个算术操作，选减法为什么比加法好?

练习 3.4 [⋆] 把中的推导写成 deriv-tree 那样的推理树。

练习 3.5 [⋆] 把中的推导写成 deriv-tree 那样的推理树。

练习 3.6 [⋆] 扩展语言, 添加新操作符 `minus`, 它取一参数 n, 返回 $-n$。例如, `minus(-(minus(5), 9))` 的值应为 14。

练习 3.7 [⋆] 扩展语言，添加加法、乘法和整数除法操作。

练习 3.8 [⋆] 向该语言添加等值比较谓词 `equal?`，以及顺序比较谓词 `greater?` 和 `less?`。

练习 3.9 [⋆⋆] 向该语言添加列表处理操作，包括 `cons`、`car`、`cdr`、`null?` 和 `emptylist`。列表可以包含任何表达值，包括其他列表。像 3.2.2 节那样，给出语言表达值和指代值的定义。例如:

```
let x = 4
in cons(x,
        cons(cons(-(x,1),
                  emptylist),
             emptylist))
```

应返回一表达值，表示列表 `(4 (3))`。

练习 3.10 [⋆⋆] 向该语言添加操作 `list`。该操作取任意数量的参数，返回一表达值，包含由参数值组成的列表。例如:

```
let x = 4
in list(x, -(x,1), -(x,3))
```

应返回一表达值，表示列表 `(4 3 1)`。

练习 3.11 [⋆] 真正的语言可能有很多上述练习那样的操作符。调整解释器代码，以便添加新操作符。

练习 3.12 [⋆] 向该语言添加 cond 表达式。语法为：

$$\textit{Expression} ::= \texttt{cond}\ \{\textit{Expression}\ \texttt{==>}\ \textit{Expression}\}^*\ \texttt{end}$$

在这种表达式里，==> 左边的表达式按序求值，直到其中一个返回真。整个表达式的值是真值表达式右边对应表达式的值。如果没有条件为真，该表达式应报错。

练习 3.13 [⋆] 修改语言，把整数作为唯一的表达值。修改 if，以 0 为假，以所有其他值为真。相应地修改谓词。

练习 3.14 [⋆⋆] 前一题的另一做法是给语言添加新的非终结符 *Bool-exp*，作为布尔值表达式。修改条件表达式的生成式：

$$\textit{Expression} ::= \texttt{if}\ \textit{Bool-exp}\ \texttt{then}\ \textit{Expression}\ \texttt{else}\ \textit{Expression}$$

为 *Bool-exp* 写出适当的生成式，实现 value-of-bool-exp。按这种方式，ex3.8 中的谓词应放在哪里?

练习 3.15 [⋆] 扩展语言，添加新操作 print，它取一参数，打印出来，返回整数 1。在我们的规范框架下，为什么不能表示这一操作?

练习 3.16 [⋆⋆] 扩展语言，允许 let 声明任意数量的变量，语法为：

$$\textit{Expression} ::= \texttt{let}\ \{\textit{Identifier} = \textit{Expression}\}^*\ \texttt{in}\ \textit{Expression}$$

像 Scheme 中的 let 那样，声明右边在当前环境中求值，每个新变量绑定到对应声明右边的值，然后求主体的值。例如：

```
let x = 30
in let x = -(x,1)
       y = -(x,2)
```

```
      in -(x,y)
```

值应为 1。

练习 3.17 [⋆⋆] 扩展语言，添加表达式 `let*`，像 Scheme 的 `let*` 那样。则：

```
let x = 30
in let* x = -(x,1) y = -(x,2)
   in -(x,y)
```

值应为 2。

练习 3.18 [⋆⋆] 向该语言添加表达式：

Expression ::= `unpack` {*Identifier*}* `=` *Expression* `in` *Expression*

则：如果 `lst` 恰好是三元素的列表，`unpack x y z = lst in ...` 将 `x`、`y` 和 `z` 绑定到 `lst` 的各元素；否则报错。例如：

```
let u = 7
in unpack x y = cons(u,cons(3,emptylist))
   in -(x,y)
```

值应为 4。

3.3 PROC：有过程的语言

目前为止，我们的语言只能进行定义好的操作。要想让这种解释性语言更有用，必须能创建新过程。我们把新语言叫做 PROC。

按照 Scheme 的设计，我们把过程作为语言的表达值，则：

$$ExpVal = Int + Bool + Proc$$
$$DenVal = Int + Bool + Proc$$

其中，*Proc* 是一值集合，表示过程。我们把 *Proc* 作为一种抽象数据类型。下面我们考虑它的接口和规范。

我们还需要语法来创建和调用过程。对应的生成式为：

Expression ::= `proc (`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `letrec(`*Expression* *Expression*`)`
`call-exp (rator rand)`

在 `(proc` *var* *body*`)` 中，变量 *var* 是绑定变量 (*bound variable*) 或形参 (*formal parameter*)。在过程调用 `(call-exp` exp_1 exp_2`)` 中，表达式 exp_1 是操作符 (*operator*)，表达式 exp_2 是操作数 (*operand*) 或实际参数 (*actual parameter*)。我们用实参 (*argument*) 指代实际参数的值。

这是这种语言的两个小例子。

```
let f = proc (x) -(x,11)
in (f (f 77))

(proc (f) (f (f 77))
 proc (x) -(x,11))
```

第一个程序创建过程 `f`，将实参减 11，然后用 77 两次调用 `f`，得到的答案为 55。第二个程序创建的过程取一参数，连续两次用 77 调用该参数；然后将减 11 的过程传给创建的过程，结果仍为 55。

现在我们来看数据类型 *Proc*。它的接口包含构造器 `procedure`，用于创建过程值；观测器 `apply-procedure`，用于调用过程值。

接下来的任务是，明确表示过程的值需要包含哪些信息。欲知此，我们考虑在程序中任意位置写 `proc` 表达式时发生了什么。

词法作用域规则告诉我们，调用一个过程时，过程的形参绑定到调用时的实参，然后在该环境内求值过程的主体。过程中出现的自由变量也应该遵守词法绑定规则。考虑表达式：

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

这里我们两次求值表达式 `proc (z) -(z,x)`。第一次求值时，`x` 绑定到 200，所以根据词法绑定规则，得到的过程将实参减 200，我们将其命名为 `f`。第二次求值时，`x` 绑定到 100，得出的过程应将实参减 100，我们将该过程命名为 `g`。

这两个过程由同一个表达式生成，而行为不同。由此可得，`proc` 表达式的值一定以某种方式依赖求值环境。因此，构造器 `procedure` 必定取三个参数：绑定变量、主体以及环境。`proc` 表达式定义为：

```
(value-of (proc-exp var body) ρ)
= (proc-val (procedure var body ρ))
```

像 `bool-val` 和 `num-val` 一样，`proc-val` 是一构造器，生成一个 *Proc* 的表达值。

调用过程时，我们要找出操作符和操作数的值。如果操作符是一个 `proc-val`，那么我们要以操作数的值调用它。

```
(value-of (call-exp rator rand) ρ)
= (let ((proc (expval->proc (value-of rator ρ)))
        (arg (value-of rand ρ)))
    (apply-procedure proc arg))
```

这里，我们用了一个判断式 `expval->proc`，像 `expval->num`，它判断表达值 `(value-of rator ρ)` 是否由 `proc-val` 生成，如果是，则从中提取出包含的过程。

最后，我们考虑调用 `apply-procedure` 时发生了什么。词法作用域规则告诉我们，调用过程时，过程主体的求值环境将其形参绑定到调用时的实参；而且，任何其他变量的值必须和过程创建时相同。因此，这些过程应满足条件

(apply-procedure (procedure var $body$ ρ) val)
= (value-of $body$ [$var = val$]ρ)

3.3.1 一个例子

我们用一个例子展示定义的各部分是如何配合的。由于我们还没有写出过程的实现，这个计算过程用规范表示。令 ρ 为任一环境。

```
(value-of
  <<let x = 200
    in let f = proc (z) -(z,x)
       in let x = 100
          in let g = proc (z) -(z,x)
             in -((f 1), (g 1))>>
  ρ)

= (value-of
    <<let f = proc (z) -(z,x)
      in let x = 100
         in let g = proc (z) -(z,x)
            in -((f 1), (g 1))>>
    [x=⌈200⌉]ρ)

= (value-of
    <<let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))>>
    [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
     [x=⌈200⌉]ρ)

= (value-of
    <<let g = proc (z) -(z,x)
      in -((f 1), (g 1))>>
```

```
    [x=⌈100⌉]ρ
     [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
      [x=⌈200⌉]ρ)

= (value-of
    <<let g = proc (z) -(z,x)
      in -((f 1), (g 1))>>
    [g=(proc-val (procedure z <<-(z,x)>>
                            [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
     [x=⌈100⌉]ρ
      [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
       [x=⌈200⌉]ρ)

= ⌈(-
    (value-of <<(f 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                              [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
       [x=⌈100⌉]ρ
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ)
    (value-of <<(g 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                              [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
       [x=⌈100⌉]ρ
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ))⌉

= ⌈(-
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈200⌉]ρ)
      ⌈1⌉)
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈100⌉][f=...][x=⌈200⌉]ρ)
```

```
      ⌈1⌉))⌉

= ⌈(-
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈200⌉]ρ)
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈100⌉][f=...][x=⌈200⌉]ρ))⌉

= ⌈(- -199 -99)⌉

= ⌈-100⌉
```

其中，绑定到的 `f` 过程将实参减 200，绑定到 `g` 的过程将实参减 100，所以 `(f 1)` 的值是 -199，`(g 1)` 的值是 -99。

3.3.2 表示过程

根据 2.2.3 节中介绍的方法，我们可以按照过程表示法，用过程在 `apply-procedure` 中的动作表示它们。欲如此，我们将 `procedure` 的值定义为实现语言的过程，它取一实参，返回下面规范要求的值：

`(apply-procedure (procedure` *var body* ρ`)` *val*`)`
`= (value-of` *body* `(extend-env` *var val* ρ`))`

因此，完整的实现是：

proc? : *SchemeVal* → *Bool*

```
(define proc?
  (lambda (val)
    (procedure? val)))
```

procedure : *Var* × *Exp* × *Env* → *Proc*

```
(define procedure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))
```

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

```
(define apply-procedure
```

```
  (lambda (proc1 val)
    (proc1 val)))
```

这里定义的函数 `proc?` 不完全准确，因为不是每个 Scheme 过程都能作为我们语言中的过程。我们需要它，只是为了定义数据类型 `expval`。

另一种方式是用 2.2.2 节那样的数据结构表示法。

proc? : *SchemeVal* → *Bool*
procedure : *Var* × *Exp* × *Env* → *Proc*

```
(define-datatype proc proc?
  (procedure
   (var identifier?)
   (body expression?)
   (saved-env environment?)))
```

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

这些数据结构常称为闭包 (*closure*)，因为它们自给自足，包含过程调用所需要的一切。有时，我们说过程闭合于 (*closed over, closed in*) 创建时的环境。

显然，这些实现都满足过程接口的定义。

不论哪种实现，我们都要给数据类型 `expval` 添加变体：

```
(define-datatype exp-val exp-val?
  (num-val
    (val number?))
  (bool-val
    (val boolean?))
  (proc-val
    (val proc?)))
```

同时给 `value-of` 添加两条新语句：

```
      (proc-exp (var body)
        (proc-val (procedure var body env)))
```

```
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc arg)))
```

记住：一定要给语言的每个扩展写出规范。参见 ex-note 的说明。

练习 3.19 [⋆] 在很多语言中，过程创建和命名必须同时进行。修改本节的语言，用 `letproc` 替换 `proc`，以支持此性质。

练习 3.20 [⋆] 在 PROC 中，过程只能有一个参数，但是可以用返回其他过程的过程来模拟多参数过程。例如，可以写出这样的代码：

```
let f = proc (x) proc (y) ...
in ((f 3) 4)
```

这个小技巧叫做咖哩化 (*Currying*)，该过程则称作咖喱式 (*Curried*) 的。写出一个咖喱式的过程，它取两个参数，返回二者之和。在我们的语言中，可以把 $x + y$ 写成 `-(x,-(0,y))`。

练习 3.21 [⋆⋆] 扩展本节的语言，添加多参数过程及其调用，语法为：

Expression ::= `proc (`{*Identifier*}$^{*(,)}$`)` *Expression*
Expression ::= `(`*Expression* {*Expression*}*`)`

练习 3.22 [⋆⋆⋆] 本节的内置操作（如差值）和过程调用使用不同的具体语法。修改具体语法，不要让语言的用户区分哪些是内置操作，哪些是自定义的过程。根据所使用的解析技术，这道练习可能很容易，也可能非常难。

练习 3.23 [⋆⋆] 下面的 PROC 程序值是什么？

```
let makemult = proc (maker)
                proc (x)
                 if zero?(x)
```

```
                then 0
                else -(((maker maker) -(x,1)), -4)
in let times4 = proc (x) ((makemult makemult) x)
   in (times4 3)
```

用这个程序里的小技巧写出 PROC 阶乘过程。提示：你可以使用咖喱化（ex3.20）定义双参数过程 `times`。

练习 3.24 [⋆⋆] 用上述程序里的小技巧写出 ex3.32 中的互递归程序 `odd` 和 `even`。

练习 3.25 [⋆] 提炼上述练习中的技巧，用它在 PROC 中定义任意递归过程。考虑下面的代码：

```
let makerec = proc (f)
               let d = proc (x)
                         proc (z) ((f (x x)) z)
               in proc (n) ((f (d d)) n)
in let maketimes4 = proc (f)
                     proc (x)
                      if zero?(x)
                      then 0
                      else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

证明它返回 12。

练习 3.26 [⋆⋆] 我们用数据结构表示过程时，在闭包中记录了整个环境。但是显然，我们只需要自由变量的绑定。修改过程的表示，只保留自由变量。

练习 3.27 [⋆] 给语言添加另一种过程 `traceproc`。`traceproc` 类似 `proc`，但会在入口和出口处打印跟踪消息。

练习 3.28 [⋆⋆] 设计过程的另一种方法是*动态绑定* (*dynamic binding*)（或称*动态定界* (*dynamic scoping*)）：求值过程主体的环境由调用处的环境扩展而得。例如，在

```
let a = 3
```

```
in let p = proc (x) -(x,a)
       a = 5
   in -(a, (p 2))
```

中，过程主体内的 a 绑定到 5，而不是 3。修改语言，使用动态绑定。做两次，一次用过程表示法表示过程，一次用数据结构表示法。

练习 3.29 [⋆⋆] 很不幸的是，使用动态绑定的程序很可能异常晦涩。例如，在词法绑定中，批量替换过程的绑定变量，决不会改变程序的行为：我们甚至可以像 3.6 节那样，删除所有变量，将它们替换为词法地址。但在动态绑定中，这种转换就危险了。

例如，在动态绑定中，过程 `proc (z) a` 返回调用者环境中的变量 a。那么程序

```
let a = 3
in let p = proc (z) a
   in let f = proc (x) (p 0)
      in let a = 5
         in (f 2)
```

返回 5，因为调用处 a 的值为 5。如果 f 的形参为 a 呢?

3.4 LETREC：支持递归过程的语言

现在我们来定义支持递归的新语言 LETREC。因为我们的语言只有单参数过程，所以我们降低难度，只让 letrec 表达式声明一个单参数过程，例如：

```
letrec double (x)
        = if zero?(x) then 0 else -((double -(x,1)), -2)
in (double 6)
```

递归声明的左边是递归过程的名字和绑定变量，= 右边是过程主体，其生成式为：

Expression ::= `letrec` *Identifier* `(`*Identifier*`)` `=` *Expression* `in` *Expression*

```
letrec-exp (p-name b-var p-body letrec-body)
```

letrec 表达式的值是其主体的值，在符合期望行为的环境中求出：

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

这里，我们给环境接口新增一个过程 extend-env-rec。但我们仍然得回答这个问题：(extend-env-rec *proc-name bound-var proc-body* ρ) 的期望行为是什么?

我们定义该环境的行为如下：设 ρ_1 为 (extend-env-rec *proc-name bound-var proc-body* ρ) 产生的环境。那么 (apply-env ρ_1 *var*) 应返回什么?

1. 如果变量 *var* 与 *proc-name* 相同，那么 (apply-env ρ_1 *var*) 应返回一个闭包，其绑定变量为 *bound-var*，主体为 *proc-body*，环境为绑定 *proc-name* 时所在的环境。而我们已经有这个环境了：就是 ρ_1！所以：

   ```
   (apply-env ρ1 proc-name)
   = (proc-val (procedure bound-var proc-name ρ1))
   ```

2. 如果 *var* 与 *proc-name* 不同，那么：

   ```
   (apply-env ρ1 var) = (apply-env ρ var)
   ```

和 fig-3.11 展示了一个例子。的最后一行递归调用 double，找出了原来的 double，正合所愿。

满足这些要求的任何方法都能够用来实现 extend-env-rec。这里我们使用对应于抽象语法表示的方法。练习讨论了其他实现策略。

如, 在抽象语法表示中, 我们为 extend-env-rec 新增一种变体。apply-env 倒数第二行的 env 对应上述 ρ_1。

```
(value-of <<letrec double(x) = if zero?(x)
                               then 0
                               else -((double -(x,1)), -2)
           in (double 6)>> ρ0)

= (value-of <<(double 6)>>
    (extend-env-rec double x <<if zero?(x) ...>> ρ0))

= (apply-procedure
    (value-of <<double>> (extend-env-rec double x
                           <<if zero?(x) ...>> ρ0))
    (value-of <<6>> (extend-env-rec double x
                      <<if zero?(x) ...>> ρ0)))

= (apply-procedure
    (value-of <<double>> (extend-env-rec double x
                           <<if zero?(x) ...>> ρ0))
    ⌈6⌉)

= (value-of
    <<if zero?(x) ...>>
    [x=⌈6⌉](extend-env-rec
                  double x <<if zero?(x) ...>> ρ0))

...

= (-
    (value-of
      <<(double -(x,1))>>
      [x=⌈6⌉](extend-env-rec
                     double x <<if zero?(x) ...>> ρ0))
    -2)
```

图 3.10 extend-env-rec 计算过程

```
= (-
    (apply-procedure
      (value-of
        <<double>>
        [x=⌈6⌉](extend-env-rec
                       double x <<if zero?(x) ...>> ρ0))
      (value-of
        <<(double -(x,1))>>
        [x=⌈6⌉](extend-env-rec
                       double x <<if zero?(x) ...>> ρ0)))
    -2)

= (-
    (apply-procedure
      (procedure x <<if zero?(x) ...>>
        (extend-env-rec double x <<if zero?(x) ...>> ρ0))
      ⌈5⌉)
    -2)

= ...
```

图 3.11 `extend-env-rec` 计算过程，续

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))

(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name b-var p-body saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var))))))
```

图 3.12 向环境添加 `extend-env-rec`

练习 3.30 [⋆] `apply-env` 倒数第二行调用 `proc-val` 的目的是什么?

练习 3.31 [⋆] 扩展上面的语言，允许声明任意数量参数的过程，像 ex3.21 那样。

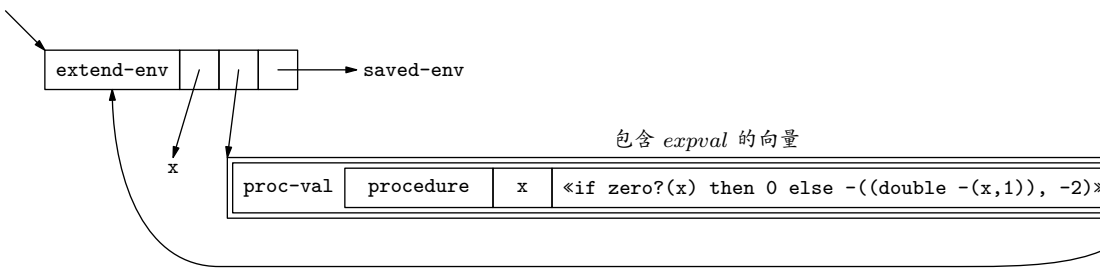
练习 3.32 [⋆] 扩展上面的语言，允许声明任意数量的单参数互递归过程，例如:

```
letrec
  even(x) = if zero?(x) then 1 else (odd -(x,1))
  odd(x)  = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

练习 3.33 [⋆⋆] 扩展上面的语言，允许声明任意数量的互递归过程，每个过程的参数数量也任意，就像 ex3.21 那样。

练习 3.34 [⋆⋆⋆] 用 2.2.3 节中环境的过程表示法实现 `extend-env-rec`。

练习 3.35 [⋆] 目前为止，我们看到的表示法都很低效，因为每次查找过程时，它们都要新创建一个闭包，但每次的闭包都相同。我们可以只创建一次闭包，把值放入长度为 1 的向量，再将其放入一个显式循环结构中，像这样:



这里是创建这种数据结构的代码:

```
(define extend-env-rec
```

```
(lambda (p-names b-vars bodies saved-env)
  (let ((vec (make-vector (length p-names))))
    (let ((new-env (extend-env p-names vec saved-env)))
      (vector-set! vec 0
        (proc-val (procedure b-var body new-env)))
      new-env))))
```

修改环境数据类型和 `apply-env` 的定义，完成这种表示的实现。确保 `apply-env` 总是返回表达值。

练习 3.36 [⋆⋆] 扩展这种实现，处理 ex3.32 中的语言。

练习 3.37 [⋆⋆] 使用动态绑定（ex3.28），不需要任何特殊机制，靠 `let` 就能创建递归过程。这是出于历史兴趣。在早年的编程语言设计中， 3.4 节讨论的那些方法还鲜为人知。要验证动态绑定实现的递归，试试程序：

```
let fact = proc (n) add1(n)
in let fact = proc (n)
                if zero?(n)
                then 1
                else *(n, (fact -(n,1)))
   in (fact 5)
```

试试词法绑定，再试试动态绑定。用动态绑定的被定语言写出 3.4 节中的互递归过程 `even` 和 `odd`。

3.5 定界和变量绑定

我们已经在很多地方见到过变量的声明和使用，现在我们来系统讨论这些思想。

在大多数编程语言中，变量只能以两种方式出现：*引用* (*reference*) 或*声明* (*declaration*)。变量引用就是使用变量。例如，在 Scheme 表达式

```
(f x y)
```

中出现的变量 `f`、`x` 和 `y` 均为引用。但在

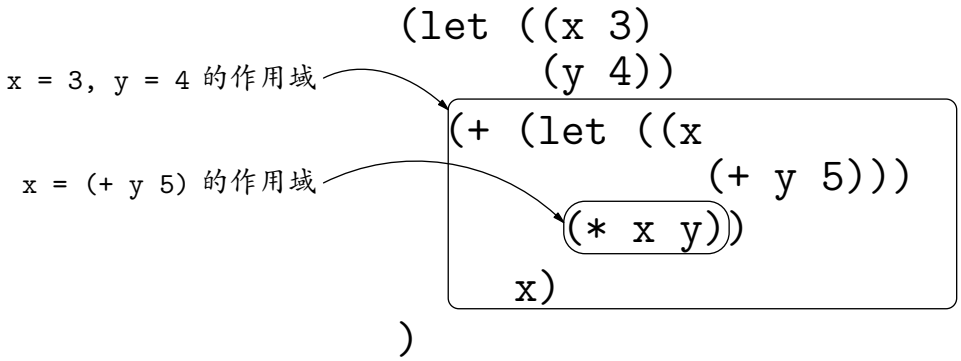


图 3.13 简单等深线

```
(lambda (x) (+ x 3))
```

或

```
(let ((x (+ y 7))) (+ x 3))
```

中，第一个出现的 `x` 是声明：引入一个变量，作为某个值的名字。在 `lambda` 表达式中，变量的值在过程调用时提供。在 `let` 表达式中，变量的值由表达式 `(+ y z)` 求得。

我们说变量引用由对应的声明绑定 (*bound*)，且绑定到它的值。在 1.2.4 节，我们已经见过用声明绑定变量的例子。

大多数编程语言中的声明都有有限的作用域，所以同一个变量名在程序的不同部分可用于不同的目的。例如，我们反复把 `lst` 用作绑定变量，它的作用域每次都限制在对应的 `lambda` 表达式主体内。

每种编程语言都有一些规则来判断各个变量引用指代哪一声明。这些规则通常叫做*定界* (*scoping*) 规则。声明在程序中生效的部分叫做声明的*作用域* (*scope*)。

我们可以不加执行地判断程序中的各个变量引用对应于哪个声明。这样的性质不必执行程序就能计算出来，名为*静态* (*static*) 性质。

要找出某个变量引用对应于哪一声明，我们*向外* (*outward*) 查找，直到找出变量的声明。这里是一个简单的 Scheme 示例。

```
(let ((x 3)                  称之为 x1
      (y 4))
  (+ (let ((x                称之为 x2
             (+ y 5)))
       (* x y))              这个 x 指代 x2
     x))                     这个 x 指代 x1
```

在这个例子中，内层的 `x` 绑定到 9，所以表达式的值为

```
(let ((x 3)
      (y 4))
  (+ (let ((x
             (+ y 5)))
       (* x y))
     x))

= (+ (let ((x
             (+ 4 5)))
       (* x 4))
     3)

= (+ (let ((x 9))
       (* x 4))
     3)

= (+ 36
     3)
```

```
= 39
```

这样的定界规则叫做词法定界 (*lexical scoping*) 规则，这样声明的变量叫做词法变量 (*lexical variable*)。

使用词法定界，我们可以重新声明一个变量，给作用域戳个“洞”。这样的内层声明遮蔽 (*shadow*) 外层声明。例如，在上例的乘式 `(* x y)` 中，内层的 `x` 遮蔽了外层的。

词法作用域是嵌套式的：每个作用域完全包裹在另一个里面。我们用等深线 (*contour diagram*) 解释这点。展示了上例的等深线。每个作用域用一个框圈起来，箭头连接声明与其作用域。

展示了一个更复杂的程序和它的等深线。其中的第 5 行、第 7 行和第 8 行，表达式 `(+ x y z)` 出现了三次。第 5 行在 `x2` 和 `z2` 的作用域内，`x2` 和 `z2` 的作用域在 `z1` 的作用域内，`z1` 的作用域在 `x1` 和 `y1` 的作用域内。所以，第 5 行的 `x` 指代 `x2`，`y` 指代 `y1`，`z` 指代 `z2`。第 7 行在 `x4` 和 `y2` 的作用域内，`x4` 和 `y2` 的作用域在 `x2` 和 `z2` 的作用域内，`x2` 和 `z2` 的作用域在 `z1` 的作用域内，`z1` 的作用域在 `x1` 和 `y1` 的作用域内。所以，第 7 行的 `x` 指代 `x4`，`y` 指代 `y2`，`z` 指代 `z2`。最后，第 8 行在 `x3` 的作用域内，`x3` 的作用域在 `x2` 和 `z2` 的作用域内，`x2` 和 `z2` 的作用域在 `z1` 的作用域内，`z1` 的作用域在 `x1` 和 `y1` 的作用域内。所以，第 8 行的 `x` 指代 `x3`，`y` 指代 `y1`，`z` 指代 `z2`。

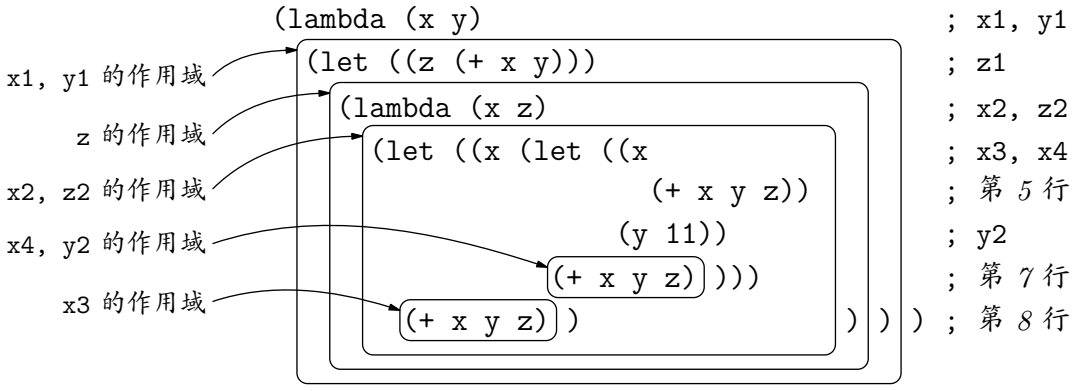


图 3.14 较复杂的等深线

变量与值的对应关系叫做绑定 (*binding*)。我们可以通过规范来理解如何创建绑定。

由 `proc` 声明的变量在过程调用时绑定。

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))
```

`let` 声明的变量绑定到声明右边的值。

```
(value-of (let-exp var val body) ρ)
= (value-of body (extend-env var val ρ))
```

`letrec` 声明的变量也要绑定到声明右边的值。

```
(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

绑定的期限 (*extent*) 指绑定保持的时长。在我们的语言中，就像在 Scheme 中一样，所有的绑定都是半无限 (*semi-infinite*) 的，意思是变量一旦绑定，该绑定就要（至少是有可能）无限期地保留。这是因为绑定可能隐藏在已返回的闭包之中。在半无限的语言中，垃圾回收器收集不能再访问的绑定。这只能在运行时判定，因此我们说这是一条动态 (*dynamic*) 性质。

很可惜的是，“动态”有时表示“在表达式求值期间”，有时又表示“无法事先计算”。如果我们不允许 `let` 的值为过程，那么 let 绑定会在 `let` 主体求值结束时到期。这叫做动态期限，而它是一条静态性质。因为这种期限是一条静态性质，所以我们可以准确预测绑定何时可以抛弃。ex3.28 等几道练习中的动态绑定表现类似。

3.6 消除变量名

定界算法的执行过程可以看作始自变量引用的外出旅行。在旅途中，到达对应的声明之前可能会跨越多条等深线。跨越的等深线数目叫做变量引用的词

深 (*lexical depth*)（或静深 (*static depth*)）。由于惯用“从 0 开始的索引”，所以不计最后跨过的等深线。例如，在 Scheme 表达式

```
(lambda (x)
  ((lambda (a)
     (x a))
   x))
```

中，最后一行 `x` 的引用以及 `a` 的引用词深均为 0，而第三行 `x` 的引用词深为 1。

因此，我们可以完全消除变量名，写成这样

```
(nameless-lambda
  ((nameless-lambda
     (#1 #0))
   #0))
```

这里，每个 `nameless-lambda` 都声明了一个新的无名变量，每个变量引用由其词深替代；这个数字准确标示了要使用的声明。这些数字叫做词法地址 (*lexical address*) 或德布鲁金索引 (*de Bruijn index*)。编译器例行计算每个变量引用的词法地址。除非用来提供调试信息，计算一旦完成，变量名即可丢弃。

这样记录信息有用，因为词法地址预测了怎样从环境中找出某个变量。

考虑我们语言中的表达式

```
let x = exp1
in let y = exp2
   in -(x,y)
```

在差值表达式中，`y` 和 `x` 的词深分别为 0 和 1。

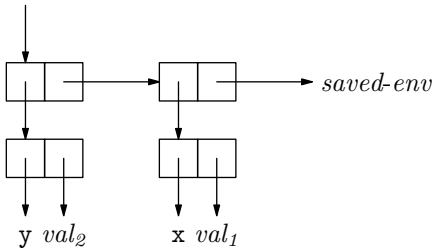
现在假设在某个适当环境中求得 exp_1 和 exp_2 的值，分别为 val_1 和 val_2，那么这个表达式的值为

```
(value-of
  <<let x = exp1
    in let y = exp2
       in -(x,y)>>
  ρ)
=
(value-of
  <<let y = exp2
    in -(x,y)>>
  [x=val1]ρ)
```

```
=
(value-of
  <<-(x,y)>>
  [y=val2][x=val1]ρ)
```

所以，求差值表达式的值时，y 深度为 0，x 深度为 1，正如词深预测的那样。

如果用关联列表表示环境（见 ex2.5），那么环境看起来像是



所以不论 val_1 和 val_2 值为何，x 和 y 的值都是取环境中第 1 个元素的余项和第 0 个元素的余项。

过程的主体也是这样。考虑

```
let a = 5
in proc (x) -(x,1)
```

在过程的主体中，x 的词深是 0，a 的词深是 1。

这个表达式的值为：

```
(value-of
  <<let a = 5 in proc (x) -(x,a)>>
  ρ)
= (value-of <<proc (x) -(x,a)>>
    (extend-env a ⌈5⌉ ρ))
= (proc-val (procedure x <<-(x,a)>> [a=⌈5⌉]ρ))
```

这个过程的主体只能通过 apply-procedure 求值：

```
(apply-procedure
  (procedure x <<-(x,a)>> [a=⌈5⌉]ρ)
  ⌈7⌉)
= (value-of <<-(x,a)>>
    [x=⌈7⌉][a=⌈5⌉]ρ)
```

每个变量又一次出现在词深预测的环境位置。

3.7 实现词法地址

现在，我们来实现上述词法地址分析。我们写出过程 translation-of-program，它取一程序，从声明中移除所有变量，并将每个变量引用替换为词深。

例如，程序

```
let x = 37
in proc (y)
    let z = -(y,x)
    in -(x,y)
```

将翻译为

```
#(struct:a-program
   #(struct:nameless-let-exp
      #(struct:const-exp 37)
      #(struct:nameless-proc-exp
         #(struct:nameless-let-exp
            #(struct:diff-exp
               #(struct:nameless-var-exp 0)
               #(struct:nameless-var-exp 1))
            #(struct:diff-exp
               #(struct:nameless-var-exp 2)
               #(struct:nameless-var-exp 1))))))
```

然后，我们写出新版的 value-of-program 来求无名程序的值，它不需要把变量放入环境中。

3.7.1 翻译器

因为是写翻译器，我们得知道源语言和目标语言。目标语言中的某些部分源语言中没有，例如 nameless-var-exp 和 nameless-let-exp；源语言中的某些部分目标语言中没有，它们由后者中的对应结构取代，例如 var-exp 和 let-exp。

我们可以为每种语言写一个 `define-datatype`，也可以让二者共用一个。因为我们使用的前端是 SLLGEN，后者更容易。我们给 SLLGEN 的语法添加生成式：

Expression ::= `%lexref` *number*
`nameless-var-exp (num)`

Expression ::= `%let` *Expression* `in` *Expression*
`nameless-let-exp (exp1 body)`

Expression ::= `%lexproc` *Expression*
`nameless-proc-exp (body)`

新的构造器名字用 `%` 开头，因为在我们的语言中，`%` 通常是注释字符。

我们的翻译器将拒绝任何含有无名结构（`nameless-var-exp`、`nameless-let-exp` 或 `nameless-proc-exp`）的程序。我们的解释器将拒绝任何含有原先具名结构（`var-exp`、`let-exp` 或 `proc-exp`）的程序，因为它们理应被替换。

要计算任何变量引用的词法地址，我们需要它所在的作用域。这是一种上下文 (*context*) 信息，所以它和 1.3 节的继承属性类似。

所以 `translation-of-program` 将取两个参数：一个表达式和一个静态环境 (*static environment*)。静态环境是一个变量列表，表示当前表达式所在的作用域。最内部作用域声明的变量成为列表的第一个元素。

例如，翻译上例中的最后一行时，静态环境为：

```
(z y x)
```

所以，在静态环境中搜索变量就是查找它在静态环境中的位置，也就是词法地址：查得 `x` 为 2，`y` 为 1，`z` 为 0。

Senv = *Listof*(*Sym*)
Lexaddr = *N*

empty-senv : () → *Senv*

```
(define empty-senv
  (lambda ()
    '()))
```

extend-senv : *Var* × *Senv* → *Senv*

```
(define extend-senv
  (lambda (var senv)
    (cons var senv)))
```

apply-senv : *Senv* × *Var* → *Lexaddr*

```
(define apply-senv
  (lambda (senv var)
    (cond
      ((null? senv)
       (report-no-binding-found var))
      ((eqv? var (car senv))
       0)
      (else
        (+ 1 (apply-senv (cdr senv) var))))))
```

图 3.15 实现静态环境

进入新的作用域就要给静态环境添加一个新元素。我们添加过程 `extend-senv` 来完成这一步。

由于静态环境只是变量列表，这些过程很容易实现，如所示。

翻译器有两个过程：`translation-of` 处理表达式，`translation-of-program` 处理程序。

`senv` 表示一些声明，我们从中翻译表达式 `e`。要完成这点，我们像 ex1.33 或 ex2.26 那样递归复制语法树，除了

1. 调用 `apply-senv`,用正确的词法地址,把每个 `var-exp` 替换为 `nameless-var-exp`。
2. 把每个 `let-exp` 替换为一个 `nameless-let-exp`。新表达式的右侧由旧表达式的右侧译得。它与原式的作用域相同，所以我们在同样的静态环境 `senv` 中翻译它。新表达式的主体由旧表达式的主体译得。但是主体位于新的作用域内，多了一个绑定变量 var。所以我们在静态环境 `(extend-senv` var $senv$`)` 中翻译主体。
3. 把每个 `proc-exp` 替换为一个 `nameless-proc-exp`，主体在新的作用域内译得，该作用域由静态环境 `(extend-senv` var `senv)` 表示。

`translation-of` 代码如所示。

过程 `translation-of-program` 在适当的初始静态环境中执行 `translation-of`。

translation-of : $Exp \times Senv \to Nameless\text{-}exp$

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num)
        (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp)))))
```

图 3.16 词法地址翻译器

translation-of : *Program* → *Nameless-exp*

```
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))
```

translation-of : () → *Senv*

```
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

3.7.2 无名解释器

凭借词法地址分析器的预测，我们的解释器不必在运行时直接搜索变量。

由于我们的程序中没有任何变量，我们不能把变量放入环境中；但是因为我们明确知道如何从环境中寻找它们，我们不需要！

我们的顶层过程是 `run`：

run : *String* → *ExpVal*

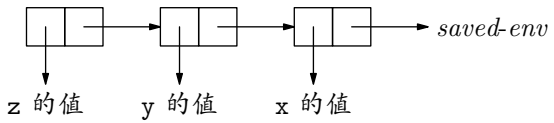
```
(define run
  (lambda (string)
    (value-of-program
     (translation-of-program
      (scan&parse string)))))
```

我们不用全功能的环境，而是用无名环境，其接口如下：

nameless-environment?： *SchemeVal* → *Bool*
empty-nameless-env： () → *Nameless-env*
extend-nameless-env： *ExpVal* × *Nameless-env* → *Nameless-env*
apply-nameless-env： *Nameless-env* × *Lexaddr* → *DenVal*

我们可以用指代值列表实现无名环境，这样 `apply-nameless-env` 只需调用 `list-ref`。这种实现如所示。

s3.7-eg 例子中最后一行的无名环境如下



由于更改了环境接口，我们需要查看代码中所有依赖这套接口的地方。我们的解释器中使用环境的只有两处：过程和 `value-of`。

修改过程规范时，只需把旧规范中的变量名移除：

```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-nameless-env val ρ))
```

nameless-environment? : *SchemeVal* → *Bool*

```
(define nameless-environment?
  (lambda (x)
    ((list-of exp-val?) x)))
```

empty-nameless-env : () → *Nameless-env*

```
(define empty-nameless-env
  (lambda () '()))
```

extend-nameless-env : *Expval* × *Nameless-env* → *Nameless-env*

```
(define extend-nameless-env
  (lambda (val nameless-env)
    (cons val nameless-env)))
```

apply-nameless-env : *Nameless-env* × *Lexaddr* → *DenVal*

```
(define apply-nameless-env
  (lambda (nameless-env n)
    (list-ref nameless-env n)))
```

图 3.17 无名环境

这一规范的实现可定义为：

procedure : *Nameless-exp* × *Nameless-env* → *Proc*

```
(define-datatype proc proc?
  (procedure
   (body expression?)
   (saved-nameless-env nameless-environment?)))
```

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure
        (body saved-nameless-env)
        (value-of body
          (extend-nameless-env val saved-nameless-env))))))
```

现在，我们可以写出 `value-of`。它的大部分与前一个解释器相同，只是原先使用 `env` 的地方现在用 `nameless-env`。但我们要处理新的部分：`nameless-var-exp`、`nameless-let-exp` 和 `nameless-proc-exp`，它们分别取代对应的 `var-exp`、`let-exp` 和 `proc-exp`。其实现如所示。`nameless-var-exp` 用于环境查询。`nameless-let-exp` 先求出式子右边的 exp_1，然后用式子右边的值扩展环境，并在新环境内求主体的值。这和 `let` 所做相同，只是没有变量。`nameless-proc` 生成一个 `proc`，随后可供 `apply-procedure` 调用。

value-of : *Nameless-exp* × *Nameless-env* → *ExpVal*

```
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (const-exp (num) ... 同前 ...)
      (diff-exp (exp1 exp2) ... 同前 ...)
      (zero?-exp (exp1) ... 同前 ...)
      (if-exp (exp1 exp2 exp3) ... 同前 ...)
      (call-exp (rator rand) ... 同前 ...)

      (nameless-var-exp (n)
        (apply-nameless-env nameless-env n))

      (nameless-let-exp
        (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp)))))
```

图 3.18 无名解释器的 `value-of`

最后是新的 value-of-program：

value-of-program : *Nameless-program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-nameless-env))))))
```

练习 3.38 [⋆] 扩展词法地址翻译器和解释器，处理 ex3.12 中的 cond。

练习 3.39 [⋆] 扩展词法地址翻译器和解释器，处理 ex3.18 中的 unpack。

练习 3.40 [⋆⋆] 扩展词法地址翻译器和解释器，处理 letrec。修改 translation-of 的上下文参数，不仅记录每个绑定变量的名字，也记录变量是否由 letrec 绑定。对 letrec 绑定变量的引用，生成一种新的引用，名为 nameless-letrec-var-exp。然后你可以继续用上面的无名环境表示，解释器则要以适当的方式处理 nameless-letrec-var-exp 。

练习 3.41 [⋆⋆] 修改词法地址翻译器和解释器，像 ex3.21 那样处理多参数的 let 表达式、过程和过程调用。用肋排表示法（ex2.21）表示无名环境。在这种表示法中，词法地址包含两个非负数：词深，指明跨越的等深线数目，与之前相同；位置，指明变量在声明中的位置。

练习 3.42 [⋆⋆⋆] 修改词法地址翻译器和解释器，使用 ex3.26 中的瘦身过程表示法。要如此，你不能在 (extend-senv *var senv*) 中翻译过程的主体，而是在一个新的静态环境中，它指明了各个变量在瘦身表示中的位置。

练习 3.43 [⋆⋆⋆] 翻译器不止能记录变量的名字。例如，考虑程序

```
let x = 3
in let f = proc (y) -(y,x)
   in (f 13)
```

这里，不必运行我们就能看出：在过程调用处，`f` 绑定到一个过程，其主体为 `-(y,x)`，`x` 的值与过程创建处相同。因此我们完全可以避免在环境中查找 `f`。扩展翻译器，记录“已知过程”，生成代码，避免在调用这样的过程时搜索环境。

练习 3.44 [⋆⋆⋆] 在前一个例子中，`f` 的唯一用途是作为一个已知过程。因此，由表达式 `proc (y) -(y,x)` 产生的过程从未使用。修改翻译器，避免产生这样的过程。

4 状态

4.1 计算的效果

到目前为止，我们只考虑了计算产生的*值* (*value*)，但是计算也有*效果* (*effect*)：它可以读取，打印，修改内存或者文件系统的状态。在现实世界中，我们*总是*对效果很感兴趣：如果一次计算不显示答案，那对我们完全没用！

产生值和产生效果有何区别？效果是*全局性* (*global*) 的，整个计算都能看到。效果*感染*整个计算（故意用双关语）。

我们主要关心一种效果：给内存中的位置赋值。赋值与绑定有何区别？我们已经知道，绑定是局部的，但变量赋值有可能是全局的。那是在本不相关的几部分计算之间*共享* (*share*) 值。如果两个过程知道内存中的同一位置，它们就能共享信息。如果把信息留在已知位置，同一个过程就能在当前调用和后续调用之间共享信息。

我们把内存建模为从*位置* (*location*) 到值集合的的有限映射，称值集合为*可存储值* (*storable values*)。出于历史原因，我们称之为*存储器* (*store*)。通常，一种语言中的可存储值与表达值相同，但不总是这样。这个选择是语言设计的一部分。

代表内存位置的数据结构叫做*引用* (*reference*)。位置是内存中可用来存值的地方，引用是指向那个地方的数据结构。位置和引用的区别可以这样类比：位置就像文件，引用就像一个 URL。URL 指向一个文件，文件包含一些数据。类似地，引用指代一个位置，位置包含一些数据。

引用有时候又叫*左值* (*L-values*)。这名字反映了这种数据结构与赋值语句

左边变量的联系。类似地，表达值，比如赋值语句右边表达式的值，叫做*右值* (*R-values*)。

我们考虑两种带有存储器的语言设计。这些设计叫做*显式引用* (*explicit reference*) 和*隐式引用* (*implicit reference*)。

4.2 EXPLICIT-REFS：显式引用语言

在这种设计中，我们添加引用，作为另一种表达值。那么，我们有：

$$\begin{aligned} ExpVal &= Int + Bool + Proc + Ref(ExpVal) \\ DenVal &= ExpVal \end{aligned}$$

这里，$Ref(ExpVal)$ 表示包含表达值的位置引用集合。

我们沿用语言中的绑定数据结构，但是添加三个新操作，用来创建和使用引用。

- `newref`，分配新的位置，返回其引用。
- `deref`，*解引用* (*deference*)：返回引用指向位置处的内容。
- `setref`，改变引用指向位置处的内容。

我们把得到的语言称作 EXPLICIT-REFS。让我们用这些结构写几个程序。

下面是两个过程 `even` 和 `odd`。它们取一参数，但是忽略它，并根据位置 `x` 处的内容是偶数还是奇数返回 1 或 0。它们不是通过直接传递数据来通信，而是改变共享变量的内容。

这个程序判断 13 是否为奇数，并返回 1。过程 `even` 和 `odd` 不引用它们的实参，而是查看绑定到 `x` 的位置中的内容。

```
let x = newref (0)
in letrec even(dummy)
            = if zero? (deref(x))
              then 1
              else begin
```

```
                    setref(x, -(deref(x), 1));
                    (odd 888)
                   end
            odd(dummy)
             = if zero? (deref(x))
               then 0
               else begin
                    setref(x, -(deref(x), 1));
                    (even 888)
                   end
   in begin setref(x,13); (odd 888) end
```

这个程序使用多声明的 `letrec`（ex3.32）和 `begin` 表达式（ex4.4）。`begin` 表达式按顺序求每个子表达式的值，并返回最后一个表达式的值。

为了同我们的单参数语言保持一致，我们给 `even` 和 `odd` 传一个无用参数；如果我们的过程支持任意数量的参数（ex3.21），这些过程的参数就可以去掉。

当两个过程需要分享很多量时，这种通信方式很方便；只需给某些随调用而改变的量赋值。同样地，一个过程可能通过一长串调用间接调用另一过程。二者可以通过一个共享变量直接交换数据，居间的过程不需要知道它。因此，以共享变量通信可作为一种隐藏信息的方式。

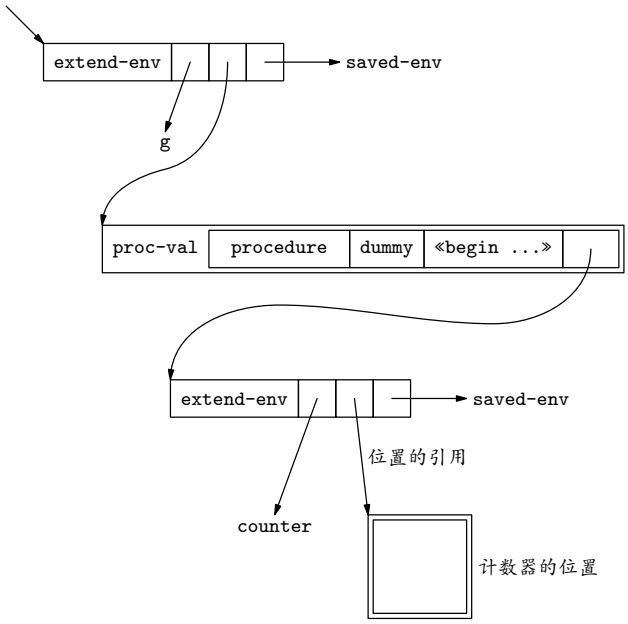
赋值的另一用途是通过私有变量创建隐藏状态。例如：

```
let g = let counter = newref(0)
        in proc (dummy)
            begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

这里，过程 `g` 保留了一个私有变量，用来存储 `g` 被调用的次数。因此，第一次调用 `g` 返回 1，第二次返回 2，整个程序的值为 -1。

下图是 `g` 绑定时所在的环境。可以认为，这是在 `g` 的不同调用之间共享信息。Scheme 过程 `gensym` 用这种技术创建唯一符号。

[!ht]



练习 4.1 [⋆] 这个程序如果写成下面这样会怎样?

```
let g = proc (dummy)
          let counter = newref(0)
          in begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
          end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

在 EXPLICIT-REFS 中, 我们可以存储任何表达值。引用也是表达值。这意味着我们可以在一个位置存储引用。考虑下面的程序:

```
let x = newref(newref(0))
```

```
in begin
    setref(deref(x), 11);
    deref(deref(x))
end
```

这段程序分配了一个新位置，内容为 0。然后，它将 `x` 绑定到一个位置，其内容为指向第一个位置的引用。因此，`deref(x)` 的值是第一个位置的引用。那么程序求 `setref` 的值时，会修改第一个位置，整个程序返回 11。

4.2.1 存储器传递规范

在我们的语言中，任何表达式都可以有效果。要定义这些效果，我们需要描述每次求值使用什么样的存储器，以及求值如何修改存储器。

在规范中，我们用 σ 表示任一存储器，用 $[l = v]\sigma$ 表示另一存储器，除了将位置 l 映射到 v 外，它与 σ 相同。有时，涉及 σ 的某个具体值时，我们称之为存储器的*状态* (*state*)。

我们使用*存储器传递规范* (*store-passing specifications*)。在存储器传递规范中，存储器作为显式参数传递给 `value-of`，并作为 `value-of` 的结果返回。那么我们可以写：

$$\texttt{(value-of } exp_1 \ \rho \ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\sigma_1\texttt{)}$$

它断言在环境为 ρ，存储器状态为 σ_0 时，表达式 exp_1 的返回值为 val_1，并且可能把存储器修改为另一状态 σ_1。

这样我们就能写出 `const-exp` 之类的无效果操作：

$$\texttt{(value-of (const-exp } n\texttt{)} \ \rho \ \sigma\texttt{)} = \texttt{(}n\texttt{,}\sigma\texttt{)}$$

以此表明求表达式的值不会修改存储器。

`diff-exp` 的规范展示了如何定义有顺序的行为。

$$\frac{\begin{array}{l}\texttt{(value-of (diff-exp } exp_1\texttt{)} \ \rho \ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\sigma_1\texttt{)} \\ \texttt{(value-of (diff-exp } exp_2\texttt{)} \ \rho \ \sigma_1\texttt{)} = \texttt{(}val_2\texttt{,}\sigma_2\texttt{)}\end{array}}{\texttt{(value-of (diff-exp } exp_1 \ exp_2\texttt{)} \ \rho \ \sigma_0\texttt{)} = \texttt{(}\lceil\lfloor val_1\rfloor - \lfloor val_2\rfloor\rceil\texttt{,}\sigma_2\texttt{)}}$$

这里，我们从状态为 σ_0 的存储器开始，首先求 exp_1 的值。exp_1 返回值为 val_1，但它可能有效果，把存储器状态修改为 σ_1。然后我们从 exp_1 修改过的存储器••也就是 σ_1••开始，求 exp_2 的值。exp_2 同样返回一个值 val_2，并把存储器状态修改为 σ_2。之后，整个表达式返回 $val_1 - val2$，对存储器不再有任何效果，所以存储器状态留在 σ_2。

再来试试条件表达式。

$$\frac{\texttt{(value-of}\ exp_1\ \rho\ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\sigma_1\texttt{)}}{\begin{array}{l}\texttt{(value-of (if-exp}\ exp_1\ exp_2\ exp_3\texttt{)}\ \rho\ \sigma_0\texttt{)} \\ \quad = \begin{cases}\texttt{(value-of}\ exp_2\ \rho\ \sigma_1\texttt{)} & \text{若}\texttt{(expval->bool}\ val_1\texttt{)} = \texttt{\#t} \\ \texttt{(value-of}\ exp_3\ \rho\ \sigma_1\texttt{)} & \text{若}\texttt{(expval->bool}\ val_1\texttt{)} = \texttt{\#f}\end{cases}\end{array}}$$

一个 `if-exp` 从状态 σ_0 开始，求条件表达式 exp_1 的值，返回值 val_1，将存储器状态修改为 σ_1。整个表达式的结果可能是 exp_2 或 exp_3 的结果，二者都在当前环境 ρ 和 exp_1 留下的存储器状态 σ_1 中求值。

练习 4.2 [⋆] 写出 `zero?-exp` 的规范。

练习 4.3 [⋆] 写出 `call-exp` 的规范。

练习 4.4 [⋆⋆] 写出 `begin` 表达式的规范。

$$\textit{Expression} ::= \texttt{begin}\ \textit{Expression}\ \{\texttt{;}\ \textit{Expression}\}^*\ \texttt{end}$$

`begin` 表达式包含一个或多个分号分隔的子表达式，按顺序求这些子表达的值，并返回最后一个的结果。

练习 4.5 [⋆⋆] 写出 `list`（ex3.10）的规范。

4.2.2 定义显式引用操作

在 EXPLICIT-REFS 中，我们必须定义三个操作：`newref`、`deref` 和 `setref`。它们的语法为：

Expression ::= `newref (`*Expression*`)`
`newref-exp (exp1)`

Expression ::= `deref (`*Expression*`)`
`deref-exp (exp1)`

Expression ::= `setref (`*Expression* `, ` *Expression*`)`
`setref-exp (exp1 exp2)`

这些操作的行为定义如下。

$$\frac{\texttt{(value-of } exp\ \rho\ \sigma_0\texttt{)} = \texttt{(}val\texttt{,}\sigma_1\texttt{)} \quad l \notin \mathrm{dom}(\sigma_1)}{\texttt{(value-of (newref-exp } exp\texttt{)}\ \rho\ \sigma_0\texttt{) = ((ref-val } l\texttt{),[}l\texttt{=}val\texttt{]}\sigma_1\texttt{)}}$$

这条规则是说：`newref-exp` 求出操作数的值，得到一个存储器，然后分配一个新位置 l，将参数值 val 放到这一位置，以此来扩展那个存储器。然后它返回新位置 l 的引用。这意味着 l 不在 σ_1 的定义域内。

$$\frac{\texttt{(value-of } exp\ \rho\ \sigma_0\texttt{)} = \texttt{(}val\texttt{,}\sigma_1\texttt{)}}{\texttt{(value-of (deref-exp } exp\texttt{)}\ \rho\ \sigma_0\texttt{) = (}\sigma_1(l)\texttt{,}\sigma_1\texttt{)}}$$

这条规则是说：`deref-exp` 求出操作数的值，然后把存储器状态改为 σ_1。参数的值应是位置 l 的引用。然后 `deref-exp` 返回 σ_1 中 l 处的内容，不再更改存储器。

$$\frac{\begin{array}{c}\texttt{(value-of } exp_1\ \rho\ \sigma_0\texttt{)} = \texttt{(}l\texttt{,}\sigma_1\texttt{)} \\ \texttt{(value-of } exp_2\ \rho\ \sigma_1\texttt{)} = \texttt{(}val\texttt{,}\sigma_2\texttt{)}\end{array}}{\texttt{(value-of (setref-exp } exp_1\ exp_2\texttt{)}\ \rho\ \sigma_0\texttt{) = (}\lceil 23\rceil\texttt{,[}l\texttt{=}val\texttt{]}\sigma_2\texttt{)}}$$

这条规则是说：`setref-exp` 从左到右求操作数的值。第一个操作数的值必须是某个位置 l 的引用；然后 `setref-exp` 把第二个参数的值 val 放到位置 l 处，以此更新存储器。`setref-exp` 应该返回什么呢？它可以返回任何值。为了强调这一选择的随意性，我们让它返回 23。因为我们对 `setref-exp` 的返回值不感兴趣，我们说这个表达式的执行*求效果* (*for effect*) 而不求值。

练习 4.6 [⋆] 修改上面的规则，让 `setref-exp` 返回右边表达式的值。

练习 4.7 [⋆] 修改上面的规则，让 `setref-exp` 返回位置的原内容。

4.2.3 实现

迄今为止，我们使用的规范语言可以轻松描述有效果计算的期望行为，但是它没有体现存储器的一个要点：引用最终指向现实世界的内存中某一真实的位置。因为我们只有一个现实世界，我们的程序只能记录存储器的一个状态 σ。

在我们的实现中，我们利用这一事实，用 Scheme 中的存储器建模存储器。这样，我们就能用 Scheme 中的效果建模效果。

我们用一个 Scheme 值表示存储器状态，但是我们不像规范建议的那样直接传递和返回它，相反，我们在一个全局变量中记录状态，实现代码中的所有过程都能访问它。这很像示例程序 `even/odd` 使用共享位置，而不是直接传递参数。使用单一全局变量时，我们也几乎不需要理解 Scheme 中的效果。

我们还是要选择如何用 Scheme 值建模存储器。我们选择的可能是最简单的模型：以表达值列表作为存储器，以代表列表位置的数字表示引用。分配新引用就是给列表末尾添加新值；更新存储器则建模为按需复制列表的一大部分。代码如和 fig-4.2 所示。

empty-store : () → *Sto*
```
(define empty-store
  (lambda () '()))
```

用法 : *Scheme* 变量，包含存储器当前的状态。初始值无意义。
```
(define the-store 'uninitialized)
```

get-store : () → *Sto*
```
(define get-store
  (lambda () the-store))
```

initialize-store! : () → *Unspecified*
用法 : `(initialize-store!)` 将存储器设为空。
```
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))
```

reference? : *SchemeVal* → *Bool*
```
(define reference?
  (lambda (v)
    (integer? v)))
```

newref : *ExpVal* → *Ref*
```
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref)))
```

deref : *Ref* → *ExpVal*
```
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

图 4.1 拙劣的存储器模型

setref! : *Ref* × *ExpVal* → *Unspecified*
用法 : 除了把位置 `ref` 的值设为 `val`，`the-store` 与原状态相同。

```
(define setref!
  (lambda (ref val)
    (set! the-store
      (letrec
        ((setref-inner
           用法 : 返回一列表，除了位置 ref1 处
           值为 val，与 store1 相同。
           (lambda (store1 ref1)
             (cond
               ((null? store1)
                 (report-invalid-reference ref the-store))
               ((zero? ref1)
                 (cons val (cdr store1)))
               (else
                 (cons
                   (car store1)
                   (setref-inner
                     (cdr store1) (- ref1 1)))))))))
        (setref-inner the-store ref)))))
```

图 4.2 拙劣的存储器模型，续

这种表示极其低效。一般的内存操作大致在常数时间内完成，但是采用我们的表示，这些操作所需的时间与存储器大小成正比。当然，真正实现起来不会这么做，但这足以达到我们的目的。

我们给表达值数据类型新增一种变体 `ref-val`，然后修改 `value-of-program`，在每次求值之前初始化存储器。

value-of-program : *Program* → *SchemeVal*

```
(define value-of-program
  (lambda (pgm)
    (initialize-store!)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

现在，我们可以写出 `value-of` 中与 `newref`、`deref` 和 `setref` 相关的语句。这些语句如所示。

我们可以给该系统添加一些辅助过程，把环境、过程和存储器转换为更易读的形式，也可以改善系统，在代码中的关键位置打印消息。我们还使用过程把环境、过程和存储器转换为更易读的形式。得出的日志详细描述了系统的动作。典型例子如和 fig-4.5 所示。此外，这一跟踪日志还表明，差值表达式的参数按从左到右的顺序求值。

练习 4.8 [⋆] 指出我们实现的存储器中，到底是哪些操作花费了线性时间而非常数时间。

练习 4.9 [⋆] 用 Scheme 向量表示存储器，从而实现常数时间操作。用这种表示会失去什么?

练习 4.10 [⋆] 实现 {ex4.4} 中定义的 `begin` 表达式。

练习 4.11 [⋆] 实现 ex4.5 中的 `list`。

练习 4.12 [⋆⋆⋆] 像解释器中展示的，我们对存储器的理解基于 Scheme 效果的含义。具体地说，我们得知道在 Scheme 程序中这些效果何时产生。我们可以写出更贴合规范的解释器，从而避免这种依赖。在这一解释器中，`value-of` 同时返回值和存储器，就像

规范中那样。这一解释器的片段如所示。我们称之为*传递存储器的解释器* (*store-passing interpreter*)。补全这个解释器，处理整个 EXPLICIT-REFS 语言。

过程可能修改存储器时，不仅返回通常的值，还要返回一个新存储器。它们包含在名为 `answer` 的数据类型之中。完成这个 `value-of` 的定义。

练习 4.13 [⋆⋆⋆] 扩展前一道练习中的解释器，支持多参数过程。

```
(newref-exp
 (exp1)
 (let ((v1 (value-of exp1 env)))
   (ref-val (newref v1))))

(deref-exp
 (exp1)
 (let ((v1 (value-of exp1 env)))
   (let ((ref1 (expval->ref v1)))
     (deref ref1))))

(setref-exp
 (exp1 exp2)
 (let ((ref (expval->ref (value-of exp1 env))))
   (let ((val2 (value-of exp2 env)))
     (begin
       (setref! ref val2)
       (num-val 23)))))
```

图 4.3 `value-of` 的显式引用操作语句

```
> (run "
let x = newref(22)
in let f = proc (z) let zz = newref(-(z,deref(x)))
                    in deref(zz)
   in -((f 66), (f 55))")

进入 let x
newref: 分配位置 0
进入 let x 主体, 环境 =
((x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
存储器 =
((0 #(struct:num-val 22)))

进入 let f
进入 let f 主体, 环境 =
((f
  (procedure
   z
   ...
   ((x #(struct:ref-val 0))
    (i #(struct:num-val 1))
    (v #(struct:num-val 5))
    (x #(struct:num-val 10)))))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
存储器 =
((0 #(struct:num-val 22)))

进入 proc z 主体, 环境 =
((z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
存储器 =
((0 #(struct:num-val 22)))
```

图 4.4 EXPLICIT-REFS 的求值跟踪日志

```
进入 let zz
newref: 分配位置 1
进入 let zz 主体, 环境 =
((zz #(struct:ref-val 1))
 (z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
存储器 =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

进入 proc z 主体, 环境 =
((z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
存储器 =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

进入 let zz
newref: 分配位置 2
进入 let zz 主体, 环境 =
((zz #(struct:ref-val 2))
 (z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
存储器 =
((0 #(struct:num-val 22))
 (1 #(struct:num-val 44))
 (2 #(struct:num-val 33)))

#(struct:num-val 11)
>
```

图 4.5 EXPLICIT-REFS 的求值跟踪日志，续

```
(define-datatype answer answer?
  (an-answer
    (val exp-val?)
    (store store?)))
```

value-of : $Exp \times Env \times Sto \to ExpVal$

```
(define value-of
  (lambda (exp env store)
    (cases expression exp
      (const-exp (num)
        (an-answer (num-val num) store))
      (var-exp (var)
        (an-answer
          (apply-store store (apply-env env var))
          store))
      (if-exp (exp1 exp2 exp3)
        (cases answer (value-of exp1 env store)
          (an-answer (val new-store)
            (if (expval->bool val)
              (value-of exp2 env new-store)
              (value-of exp3 env new-store)))))
      (deref-exp
        (exp1)
        (cases answer (value-of exp1 env store)
          (an-answer (v1 new-store)
            (let ((ref1 (expval->ref v1)))
              (an-answer (deref ref1) new-store)))))
      ...)))
```

图 4.6 ex4.12，传递存储器的解释器

4.3 IMPLICIT-REFS：隐式引用语言

显式引用设计清晰描述了内存的分配、解引用和变更，因为显而易见，这些操作都在程序员的代码之中。

大多数编程语言都用共同的方式处理分配、解引用和变更，并把它们打包为语言的一部分。这样，由于这些操作存在于语言内部，程序员不需要担心何时执行它们。

在这种设计中，每个变量都表示一个引用。指代值是包含表达值的位置的引用。引用不再是表达值，只能作为变量绑定。

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$

每次绑定操作都会分配一个位置：在每个过程调用处，在 `let` 和 `letrec` 中。

当变量出现在表达式中，我们首先在环境中查找标识符，找到绑定的位置，然后在存储器中找出那个位置的值。因此对 `var-exp`，我们有个“二级”系统。

一个位置的内容可用 `set` 表达式修改，语法为：

Expression ::= `set` *Identifier* `=` *Expression*
`assign-exp (var exp1)`

这里的 *Identifier* 不是表达式的一部分，所以无法解引用。在这种设计中，我们说变量是可变的 (*mutable*)，意为可以修改。

这种设计叫做按值调用 (*call-by-value*)，或隐式引用 (*implicit reference*)。大多数编程语言，包括 Scheme，都采纳这一设计的某种变体。

是这种设计的两个示例程序。因为引用不再是表达值，我们不能像 4.2 节中的例子那样做链式引用。

```
let x = 0
in letrec even(dummy)
           = if zero?(x)
             then 1
             else begin
                    set x = -(x,1);
                    (odd 888)
                  end
          odd(dummy)
           = if zero?(x)
             then 0
             else begin
                    set x = -(x,1);
                    (even 888)
                  end
   in begin set x = 13; (odd -888) end

let g = let count = 0
        in proc (dummy)
            begin
             set count = -(count,-1);
             count
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

图 4.7 IMPLICIT-REFS 中的 odd 和 even

4.3.1 规范

我们可以轻松写出解引用和 `set` 的规则。现在，环境总是把变量绑定到位置，所以当变量作为表达式时，我们需要将其解引用：

$$\texttt{(value-of (var-exp } var\texttt{) } \rho\ \sigma\texttt{)} = \texttt{(}\sigma(\rho(var))\texttt{,}\sigma\texttt{)}$$

赋值就像我们预想的那样：我们在环境中查找式子左侧的标识符，获取一个位置，在环境中求右边表达式的值，修改指定位置的内容。就像 `setref`，`set` 表达式的返回值任意。我们让它返回表达值 27。

$$\frac{\texttt{(value-of } exp_1\ \rho\ \sigma_0\texttt{)} = \texttt{(}val_1\texttt{,}\sigma_1\texttt{)}}{\texttt{(value-of (assign-exp } var\ exp_1\texttt{) } \rho\ \sigma_0\texttt{) = (}\lceil 27 \rceil\texttt{,[}\rho(var)\texttt{=}val_1\texttt{]}\sigma_1\texttt{)}}$$

我们还要重写过程调用和 `let` 规则，体现出对存储器的修改。对过程调用，规则变成：

$$\texttt{(apply-procedure (procedure } var\ body\ \rho\texttt{) } val\ \sigma\texttt{)}$$
$$\texttt{= (value-of } body\ \texttt{[}var = l\texttt{]}\rho\ \texttt{[}l = val\texttt{]}\sigma\texttt{)}$$

其中，l 是不在 σ 定义域中的某一位置。

`(let-exp` var exp_1 $body$`)` 的规则类似。我们首先求右边 exp_1 的值，然后将该值放入一个新位置，将变量 var 绑定到这个位置，在得到的环境中求 `let` 主体的值，作为整个表达式的值。

练习 4.14 [⋆] 写出 `let` 的规则。

4.3.2 实现

现在我们着手修改解释器。在 `value-of` 中，我们取出每个 `var-exp` 的值，就像规则描述的那样：

```
(var-exp (var) (deref (apply-env env var)))
```

`assign-exp` 的代码也显而易见：

```
(assign-exp (var exp1)
  (begin
    (setref!
```

```
          (apply-env env var)
          (value-of exp1 env))
        (num-val 27)))
```

创建引用呢？新的位置应在每一新绑定处创建。这门语言中只有四个地方创建新绑定：初始环境中、`let` 中、过程调用以及 `letrec` 中。

在初始环境中，我们直接分配新位置。

对 `let`，我们修改 `value-of` 中相应的行，分配包含值的新位置，并把变量绑定到指向该位置的引用。

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var (newref val1) env))))
```

对过程调用，我们同样修改 `apply-procedure`，调用 `newref`。

apply-procedure : $Proc \times ExpVal \to ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

最后，要处理 `letrec`，我们替换 `apply-env` 中的 `extend-env-rec` 从句，令其返回一个引用，指向包含适当闭包的位置。由于我们使用多声明的 `letrec`（ex3.32），`extend-env-rec` 取一个过程名列表，一个绑定变量列表，一个过程主体列表，以及已保存的环境。过程 `location` 取一变量，一个变量列表。若变量存在于列表中，`location` 返回变量在列表中的位置；若不存在，返回 `#f`。

```
(extend-env-rec (p-names b-vars p-bodies saved-env)
  (let ((n (location search-var p-names)))
```

```
(if n
  (newref
    (proc-val
      (procedure
        (list-ref b-vars n)
        (list-ref p-bodies n)
        env)))
  (apply-env saved-env search-var))))
```

用前面介绍的辅助组件，展示了 IMPLICIT-REFS 求值的简单例子。

```
> (run "
let f = proc (x) proc (y)
         begin
          set x = -(x,-1);
          -(x,y)
         end
in ((f 44) 33)")
newref: 分配位置 0
newref: 分配位置 1
newref: 分配位置 2
进入 let f
newref: 分配位置 3
进入 let f 主体, 环境 =
((f 3) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

newref: 分配位置 4
进入 proc x 主体, 环境 =
((x 4) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44)))

newref: 分配位置 5
进入 proc y 主体, 环境 =
((y 5) (x 4) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44))
 (5 #(struct:num-val 33)))

#(struct:num-val 12)
>
```

图 4.8 IMPLICIT-REFS 的简单求值

练习 4.15 [⋆] 在中，环境中的变量为什么绑定到一般的整数，而不是中那样的表达值?

练习 4.16 [⋆] 既然变量是可变的，我们可以靠赋值产生递归过程。例如：

```
letrec times4(x) = if zero?(x)
                   then 0
                   else -((times4 -(x,1)), -4)
in (times4 3)
```

可以替换为：

```
let times4 = 0
in begin
    set times4 = proc (x)
                   if zero?(x)
                   then 0
                   else -((times4 -(x,1)), -4);
    (times4 3);
   end
```

手动跟踪这个程序，验证这种翻译可行。

练习 4.17 [⋆⋆] 写出规则并实现多参数过程和声明多变量的 `let`。

练习 4.18 [⋆⋆] 写出规则并实现声明多过程的 `letrec` 表达式。

练习 4.19 [⋆⋆] 修改声明多过程的 `letrec` 实现，让每个闭包只生成一次，并且只分配一个位置。本题类似 ex3.35。

练习 4.20 [⋆⋆] 在本节的语言中，就像在 Scheme 中一样，所有变量都是可变的。另一种设计是同时允许可变和不可变的变量绑定：

$$ExpVal = Int + Bool + Proc$$
$$DenVal = Ref(ExpVal) + ExpVal$$

只有变量绑定可变时，才能赋值。当指代值是引用时，解引用自动进行。

修改本节的语言，让 `let` 像之前那样引入不可变变量，可变变量则由 `letmutable` 表达式引入，语法为：

Expression ::= `letmutable` *Identifier* `=` *Expression* `in` *Expression*

练习 4.21 [⋆⋆] 之前，我们建议两个相去很远的过程通过赋值交换信息，避免居间的过程知晓，从而使程序更加模块化。这样的赋值常常应该是临时的，只在执行函数调用时生效。向语言添加动态赋值 (*dynamic assignment*)（又称流式绑定 (*fluid binding*)）组件，完成这一操作。生成式为：

Expression ::= `setdynamic` *Identifier* `=` *Expression* `during` *Expression*

`setdynamic-exp (var exp1 body)`

`setdynamic` 表达式的效果是把 exp_1 的值临时赋给 var，求 *body* 的值，重新给 var 赋其原值，然后返回 *body* 的值。变量 var 必需已绑定。例如，在下列表达式中：

```
let x = 11
in let p = proc (y) -(y,x)
   in -(setdynamic x = 17 during (p 22),
        (p 13))
```

x 是过程 `p` 中的自由变量，在调用 `(p 22)` 中值为 17，在调用 `(p 13)` 中重设为 11，所以表达式的值为 $5 - 2 = 3$。

练习 4.22 [⋆⋆] 迄今为止，我们的语言都是面向表达式 (*expression-oriented*) 的：我们感兴趣的主要是表达式这种句法类别和它们的值。扩展语言，建模简单的面向语句 (*statement-oriented*) 的语言，其规范概述如下。一定要遵循语法，分别写出过程来处理程序、语句和表达式。

值 同 IMPLICIT-REFS。

语法 使用下列语法：

$$
\begin{aligned}
\textit{Program} &::= \textit{Statement} \\
\textit{Statement} &::= \textit{Identifier} \texttt{ = } \textit{Expression} \\
&::= \texttt{print } \textit{Expression} \\
&::= \{\{\textit{Statement}^{*(;)}\}\} \\
&::= \texttt{if } \textit{Expression } \textit{Statement } \textit{Statement} \\
&::= \texttt{while } \textit{Expression } \textit{Statement} \\
&::= \texttt{var } \{\textit{Identifier}\}^{*(;)} \texttt{ ; } \textit{Statement}
\end{aligned}
$$

非终结符 *Expression* 指的是 IMPLICIT-REFS 语言中的表达式，可能稍有扩展。

语义 程序是一个语句。语句不返回值，而是修改和打印存储器。

赋值语句仍按通常的方式执行。打印语句求出实参的值，打印结果。`if` 语句仍按通常的方式执行。块语句定义见 *Statement* 中的最后一个生成式。它把每个声明变量绑定到一个未初始化的引用，然后执行块主体。这些绑定的作用域是块主体。

用如下断言写出语句的规范：

$$\texttt{(result-of } stmt \ \rho \ \sigma_0\texttt{)} = \sigma_0$$

例子 这里是一些例子。

```
(run "var x,y; {x = 3; y = 4; print +(x,y)}") % 例 1
7
(run "var x,y,z; {x = 3;                      % 例 2
                  y = 4;
                  z = 0;
                  while not(zero?(x))
                    {z = +(z,y); x = -(x,1)};
                  print z}")
12
(run "var x; {x = 3;                          % 例 3
              print x;
```

```
              var x; {x = 4; print x};
              print x}")
3
4
3
(run "var f,x; {f = proc(x,y) *(x,y);          % 例 4
                x = 3;
                print (f 4 x)}")
12
```

例 3 解释了块语句的作用域。

例 4 解释了语句和表达式的交互。过程值创建并存储于变量 `f`。最后一行用实参 4 和 `x` 调用这个过程；因为 `x` 绑定到一个引用，解引用得 3。

练习 4.23 [⋆] 给 ex4.22 中的语言添加 `read` 语句，形如 `read` var。这一语句从输入读取一个非负数，存入指定的变量中。

练习 4.24 [⋆] `do-while` 语句类似 `while`，但是条件判断在其主体之后执行。给 ex4.22 中的语言添加 `do-while` 语句。

练习 4.25 [⋆] 扩展 ex4.22 语言中的块语句，允许初始化变量。在你的解答中，变量的作用域是否包含同一个块语句中后续声明的变量?

练习 4.26 [⋆⋆⋆] 扩展前一道练习中的解答，允许同一块语句中声明的过程互递归。考虑给语言增加限制，块中的过程声明要在变量声明之后。

练习 4.27 [⋆⋆⋆] 扩展前一道练习中的解答，增加子程序 (*subroutine*)。我们把子程序当过程用，但是它不返回值，且其主体为语句而非表达式。然后增加子程序调用，作为一种新语句。扩展块的语法，允许同时声明过程和子程序。这将如何影响指代值和表达值? 如果在子程序调用中使用过程会怎样？反过来呢?

4.4 MUTABLE-PAIRS：可变序对语言

在 ex3.9 中，我们给语言添加了列表，但它们是不可变的：不像 Scheme 中，有 `set-car!` 和 `set-cdr!` 处理它们。

现在，我们给 IMPLICIT-REFS 添加可变序对。序对是表达值，具有如下操作：

$$\begin{aligned}
\textbf{make-pair} &: \mathit{Expval} \times \mathit{Expval} \to \mathit{MutPair} \\
\textbf{left} &: \mathit{MutPair} \to \mathit{Expval} \\
\textbf{right} &: \mathit{MutPair} \to \mathit{Expval} \\
\textbf{setleft} &: \mathit{MutPair} \times \mathit{Expval} \to \mathit{Unspecified} \\
\textbf{setright} &: \mathit{MutPair} \times \mathit{Expval} \to \mathit{Unspecified}
\end{aligned}$$

序对包含两个位置，二者可以分别赋值。由此得出语言值的*定义域方程* (*domain equations*)：

$$\begin{aligned}
\mathit{ExpVal} &= \mathit{Int} + \mathit{Bool} + \mathit{Proc} + \mathit{MutPair} \\
\mathit{DenVal} &= \mathit{Ref}(\mathit{ExpVal}) \\
\mathit{MutPair} &= \mathit{Ref}(\mathit{ExpVal}) \times \mathit{Ref}(\mathit{ExpVal})
\end{aligned}$$

我们把这种语言叫做 MUTABLE-PAIRS。

4.4.1 实现

我们可以直接用前例中的 `reference` 数据类型实现可变序对。代码如所示。

完成图中的代码，给语言添加这些就很直接了。如所示，我们给表达值数据类型新增一种变体 `mutpair-val`，给 `value-of` 新增 5 行代码。我们让 `setleft` 返回任意值 82，让 `setright` 返回 83。用前述辅助组件得到的示例跟踪日志如所示。

```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))

make-pair : ExpVal × ExpVal → MutPair
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))

left : MutPair → ExpVal
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))

right : MutPair → ExpVal
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc)))))

setleft : MutPair × ExpVal → Unspecified
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val)))))

setright : MutPair × ExpVal → Unspecified
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))
```

图 4.9 可变序对的拙劣实现

```
(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (left p1))))

(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1))))

(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))

(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))
```

图 4.10 给解释器添加可变序对模块

4.4.2 可变序对的另一种表示

把可变序对表示为两个引用没有利用与 `MutPair` 相关的已知信息。序对中的两个位置虽然能够各自赋值，但它们不是独立分配的。我们知道它们会一起分配：如果序对的左侧是一个位置，那么右侧是下一个位置。所以我们还可以用左侧的引用表示序对。代码如所示，不需再做其他修改。

```
> (run "let glo = pair(11,22)
in let f = proc (loc)
            let d1 = setright(loc, left(loc))
            in let d2 = setleft(glo, 99)
               in -(left(loc),right(loc))
   in (f glo)")
;; 为 init-env 分配单元
newref: 分配位置 0
newref: 分配位置 1
newref: 分配位置 2
进入 let glo
;; 为序对分配单元
newref: 分配位置 3
newref: 分配位置 4
;; 为 glo 分配单元
newref: 分配位置 5
进入 let glo 主体, 环境 =
((glo 5) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4))))

进入 let f
;; 为 f 分配单元
newref: 分配位置 6
进入 let f 主体, 环境 =
((f 6) (glo 5) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2)))))
```

图 4.11 MUTABLE-PAIRS 求值的跟踪日志

```
;; 为 loc 分配单元
newref: 分配位置 7
进入 proc loc 主体, 环境 =
((loc 7) (glo 5) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2))))
 (7 #(struct:mutpair-val #(struct:a-pair 3 4))))

#(struct:num-val 88)
>
```

图 4.12 MUTABLE-PAIRS 求值的跟踪日志，续

与之类似，堆中的任何聚合对象都可以用其第一个位置的指针表示。但是，如果不提供区域的长度信息，指针本身无法指明一片内存区域（见 ex4.30）。缺乏长度信息是经典安全问题的一大来源，比如写数组越界。

---

mutpair? : *SchemeVal* → *Bool*

```
(define mutpair?
  (lambda (v)
    (reference? v)))
```

make-pair : *ExpVal* × *ExpVal* → *MutPair*

```
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1))))
```

left : *MutPair* → *ExpVal*

```
(define right
  (lambda (p)
    (deref (+ 1 p))))
```

right : *MutPair* → *ExpVal*

```
(define left
  (lambda (p)
    (deref p)))
```

setleft : *MutPair* × *ExpVal* → *Unspecified*

```
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

setright : *MutPair* × *ExpVal* → *Unspecified*

```
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

图 4.13 可变序对的另一种表示

练习 4.28 [⋆⋆] 写出五个可变序对操作的推理规则规范。

练习 4.29 [⋆⋆] 给语言添加数组。添加新操作符 newarray、arrayref 和 arrayset，用它们来创建、解引用和更新数组。这需要：

$$\begin{aligned} ArrVal &= (Ref(ExpVal)) \\ ExpVal &= Int + Bool + Proc + ArrVal \\ DenVal &= Ref(ExpVal) \end{aligned}$$

由于数组中的位置是连续的，用上述第二种表示。下面程序的结果是什么？

```
let a = newarray(2,-99)
    p = proc (x)
          let v = arrayref(x,1)
          in arrayset(x,1,-(v,-1))
in begin arrayset(a,1,0); (p a); (p a); arrayref(a,1) end
```

这里 newarray(2,-99) 要创建长度为 2 的数组，数组中的每个位置都包含 -99。begin 表达式定义同 ex4.4。令数组索引从 0 开始，所以长度为 2 的数组索引为 0 和 1。

练习 4.30 [⋆⋆] 给 ex4.29 的语言添加过程 arraylength，返回数组的长度。你的过程运行时间应为常数。arrayref 和 arrayset 一定要查验索引，确保索引值在数组长度之内。

4.5 传参变体

当过程主体执行时，其形参绑定到一个指代值。那个值从哪儿来？它一定是过程调用传入实参的值。我们已见过两种传参方式：

- 自然传参，指代值与实参的表达值相同（pass-by-value）。
- 按值调用，指代值是一个引用，指向一个位置，该位置包含实参的表达值（4.3 节）。这是最常用的传参方式。

本节探讨其他一些传参机制。

4.5.1 按指调用

考虑下面的表达式：

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

按值调用时，绑定到 x 的指代值是一个引用，它包含的初始值与绑定到 a 的引用相同，但这些引用互不相干。所以赋值给 x 不会影响引用 a 的内容，整个表达式的值是 3。

按值调用时，当过程给某个参数赋新值，调用者无法获悉。当然，如果传给调用者的参数像 4.4 节那样包含可变序对，那么调用者能看到 setleft 和 setright 的效果，但看不到 set 的效果。

虽然这样隔离调用者和被调者常合所愿，但有些时候，给过程传递一个位置，并让过程给这个位置赋值也不无好处。要这样做，可以给过程传递一个引用，该引用指向调用者变量的位置，而不是变量的内容。这种传参机制叫做*按指调用* (*call-by-reference*)。如果操作数正是变量引用，那就传递变量位置的引用。然后，过程的形参绑定到这个位置。如果操作数是其他类型的表达式，那么形参绑定到一个新位置，该位置包含操作数的值，就像按值调用一样。在上例中使用按指调用，把 4 赋给 x 等效于把 4 赋给 a，所以整个表达式返回 4，而不是 3。

按指调用过程，且实参为变量时，传递的不是按值调用中变量所在位置的内容，而是那个变量的*位置*。例如，考虑：

```
let p = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin (g z); z end
```

调用过程 g 时，y 绑定到 z 的位置，而不是那个位置的内容。类似地，调用 f 时，x 绑定到同一个位置。所以，x、y 和 z 都绑定到同一位置，set x = 44 的效果是把那个位置的内容设为 44。因此，整个表达式的值是 44。执行这个表达式的跟踪日志如和 fig-4.15 所示。在本例中，x、y 和 z 最终都绑定到位置 5。

按指调用的常见用法是返回多个值。过程以通常方式返回一个值，并给按指传递的参数赋其他值。另一种例子，对换变量的值：

```
let swap = proc (x) proc (y)
            let temp = x
            in begin
                set x = y;
                set y = temp
               end
in let a = 33
   in let b = 44
      in begin
          ((swap a) b);
          -(a,b)
         end
```

采用按指调用，这会对换 a 和 b 的值，所以它返回 11。但如果用已有的按值调用解释器执行这个程序，它会返回 -11，因为在对换过程的内部赋值对变量 a 和 b 毫无影响。

就像在按值调用中一样，在按指调用中，变量仍然指代表达值的引用：

$$ExpVal = Int + Bool + Proc$$
$$DenVal = Ref(ExpVal)$$

唯一需要改变的是新位置的分配。按值调用中，求每个操作数的值都要分配新位置；按指调用中，只在求非变量操作数的值时才分配新位置。

这很容易实现。函数 `apply-procedure` 必需修改，因为不是每个过程调用都要分配新位置。那份责任必须上移至 `value-of` 中的 `call-exp`，因为它才具有做出判断所需的信息。

apply-procedure : $Proc \times ExpVal \to ExpVal$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var val saved-env))))))
```

然后我们修改 value-of 中的 call-exp，引入新函数 value-of-operand 来做必要的判断。

```
(call-exp
 (rator rand)
 (let ((proc (expval->proc (value-of rator env)))
       (arg (value-of-operand rand env)))
   (apply-procedure proc arg)))
```

过程 value-of-operand 检查操作数是否为变量。如果是，则返回那个变量指代的引用，然后传给过程 apply-procedure；否则，它求出操作数的值，在新位置放入那个值，并返回指向该位置的引用。

value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (value-of exp env))))))
```

我们也可以按照同样的方式修改 let，但我们不这样做，因此语言中仍然保留按值调用的功能。

多个按指调用参数可以指向同一个位置，如下面的程序所示。

```
let b = 3
in let p = proc (x) proc(y)
            begin
             set x = 4;
             y
            end
   in ((p b) b)
```

它的值为 4，因为 x 和 y 指向同一个位置，即 b 的绑定。这种现象叫做变量别名 (*variable aliasing*)。这里的 x 和 y 是同一个位置的别名（名字）。通常，我们在给一个变量赋值时并不想改变另一个变量的值，所以别名会导致程序难以理解。

```
> (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin
          (g z);
          z
         end")
newref: 分配位置 0
newref: 分配位置 1
newref: 分配位置 2
进入 let f
newref: 分配位置 3
进入 let f 主体, 环境 =
((f 3) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

进入 let g
newref: 分配位置 4
进入 let g 主体, 环境 =
((g 4) (f 3) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
(1 #(struct:num-val 5))
(2 #(struct:num-val 10))
(3 (procedure x ... ((i 0) (v 1) (x 2))))
(4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))

进入 let z
newref: 分配位置 5
进入 let z 主体, 环境 =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))
```

图 4.14 CALL-BY-REFERENCE 的简单求值

```
进入 proc y 主体, 环境 =
((y 5) (f 3) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

进入 proc x 主体, 环境 =
((x 5) (i 0) (v 1) (x 2))
存储器 =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

#(struct:num-val 44)
>
```

图 4.15 CALL-BY-REFERENCE 的简单求值，续

练习 4.31 [⋆] 写出 CALL-BY-REFERENCE 的推理规则规范。

练习 4.32 [⋆] 扩展语言 CALL-BY-REFERENCE，支持多参数过程。

练习 4.33 [⋆⋆] 扩展语言 CALL-BY-REFERENCE，也令其支持按值调用的过程。

练习 4.34 [⋆] 给语言添加按指调用的 `let`，名为 `letref`。写出规范并实现。

练习 4.35 [⋆⋆] 在按值调用框架下,我们仍能享受按指调用的便利。扩展语言 IMPLICIT-REF，添加新表达式：

Expression ::= `ref` *Identifier*

`ref-exp (var)`

这与语言 EXPLICIT-REF 不同。因为引用只能从变量获得。这就使我们能用按值调用语言写出类似 `swap` 的程序。下面表达式的值是什么?

```
let a = 3
    b = 4
in let swap = proc (x) proc (y)
                let temp = deref(x)
                in begin
                    setref(x,deref(y));
                    setref(y,temp)
                   end
   in begin ((swap ref a) ref b); -(a,b) end
```

此处使用支持多声明的 `let`（ex3.16）。这种语言的表达值和指代值是什么?

练习 4.36 [⋆] 大多数语言支持数组，在按指调用中，数组引用通常以类似变量引用的方式处理。如果操作数是数组引用，那就不给被调过程传递它的内容，而是传递引用指向的位置。比如，需要调用对换过程的常见情形是交换数组元素，传递数组引用就能用上对换过程。给按指调用语言添加 ex4.29 中的数组操作符，扩展 `value-of-operand`，处理这种情况，使下例中的过程调用能够如愿执行：

```
((swap (arrayref a i)) (arrayref a j))
```

要是下面这样呢?

```
((swap (arrayref a (arrayref a i))) (arrayref a j))
```

练习 4.37 [⋆⋆] *按值和结果调用* (*call-by-value-result*) 是按指调用的一种变体。在按值和结果调用中，实参必需是变量。传递参数时，形参绑定到新的引用，初值为实参的值，就像按值调用一样。然后过程主体照常执行。但过程主体返回时，新引用处的值复制到实参指代的引用中。因为这样可以改进内存分配，所以可能比按指调用更高效。实现按值和结果调用，写出一个过程，使之在按指调用与按值和结果调用中产生不同的答案。

4.5.2 懒求值：按名调用和按需调用

迄今为止，我们讨论的所有参数传递机制都是*即时* (*eager*) 的：它们总是找出每个操作数的值。现在我们来看另一种截然不同的传参机制，名叫*懒求值* (*lazy evaluation*)。在懒求值中，操作数的值直到过程主体需要时才会求取。如果过程主体从未引用相关参数，就不需求操作数的值。

这可能使程序免于永不终止。例如，考虑:

```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

这里的 `infinite-loop` 是一个过程，调用时永不终止。`f` 是一个过程，调用时不引用它的参数，总返回 11。我们考虑过的任何一种传参机制都无法使这个程序终止。但在懒求值中，这个程序将返回 11，因为操作数 `(infinite-loop 0)` 没有求值。

现在，我们使用懒求值修改我们的语言。在懒求值中，如无必要，我们不求操作数表达式的值。因此，我们将过程的绑定变量与未求值的操作数关联起来。当过程主体需要绑定变量的值时，我们先求对应操作数的值。有时，我们把不求操作数的值而传给过程叫做*冻结* (*frozen*)，把过程求操作数的值叫做*解冻* (*thawed*)。

当然，我们还要加入过程求值时的环境。要这样，我们引入一种新的数据类型，值箱 (*thunk*)。值箱包含一个表达式、一个环境。

```
(define-datatype thunk thunk?
  (a-thunk
   (exp1 expression?)
   (env environment?)))
```

当过程需要使用绑定变量的值时，会求相应值箱的值。

我们面对的情况稍稍复杂一些，因为我们需要同时容纳懒求值、计算效果和即时求值（`let` 需要）。因此，我们把指代值定为内存位置的引用，位置包含表达值或者值箱。

$$DenVal = Ref(ExpVal + Thunk)$$
$$ExpVal = Int + Bool + Proc$$

我们的位置分配策略与按指调用类似：如果操作数是变量，那么我们传递指代的引用；否则，我们给未求值的参数在新位置放一个值箱，传递该位置的引用。

value-of-operand : $Exp \times Env \to Ref$

```
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (a-thunk exp env))))))
```

求 `var-exp` 的值时，我们首先找到变量绑定的位置。如果该位置是一个表达值，那么返回这个值，作为 `var-exp` 的值。如果它包含一个值箱，那么我们求取并返回值箱的值。这叫做按名调用 (*call by name*)。

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? val)
        val
        (value-of-thunk val)))))
```

过程 value-of-thunk 定义如下：

value-of-thunk : *Thunk* → *ExpVal*
```
(define value-of-thunk
  (lambda (th)
    (cases thunk th
      (a-thunk (exp1 saved-env)
        (value-of exp1 saved-env)))))
```

或者，一旦发现值箱的值，我们可以把表达值放到同一个位置，这样就不需要再次求值箱的值。这种方式叫做*按需调用* (*call by need*)。

```
    (var-exp (var)
      (let ((ref1 (apply-env env var)))
        (let ((w (deref ref1)))
          (if (expval? w)
            w
            (let ((val1 (value-of-thunk w)))
              (begin
                (setref! ref1 val1)
                val1))))))
```

这里用到了名为*助记法* (*memoization*) 的通用策略。

各种形式的懒求值引人之处在于，即使没有效果，通过它也能以相当简单的方式思考程序。把过程调用替换为过程的主体，把过程主体内对每个形参的引用替换为对应的操作数，就能建模过程调用。这种求值策略是 lambda 演算的基础，在 lambda 演算中，它叫做 β-*推导* (β-*reduction*)。

不幸的是，按名调用和按需调用使求值顺序难以确定，而这对理解有效果的程序至关重要。但没有效果时，这不成问题。所以懒求值盛行于函数式编程语言（没有计算效果的那些），在别处却杳无踪影。

练习 4.38 [⋆] 下面的例子展示了 ex3.25 在按需调用中的变体。ex3.25 中的原始程序在按需调用中可行吗？如果下面的程序在按值调用中运行呢？为什么？

```
let makerec = proc (f)
               let d = proc (x) (f (x x))
```

```
                in (f (d d))
in let maketimes4 = proc (f)
                     proc (x)
                      if zero?(x)
                      then 0
                      else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

练习 4.39 [⋆] 没有计算效果的话，按名调用和按需调用总是给出同样的答案。设计一个例子，让按名调用和按需调用给出不同的答案。

练习 4.40 [⋆] 修改 `value-of-operand`，避免为常量和过程生成值箱。

练习 4.41 [⋆⋆] 写出按名调用和按需调用的推理规则规范。

练习 4.42 [⋆⋆] 给按需调用解释器添加懒求值的 `let`。

5 传递续文的解释器

在第 3 章，我们用环境的概念探讨绑定行为，建立每部分程序执行的数据上下文。这里，我们将用类似方式探讨每部分程序执行的控制上下文 (*control context*)。我们将介绍续文 (*continuation*) 的概念，用来抽象控制上下文。我们将要编写的解释器会取一续文参数，从而彰显控制上下文。

考虑下面的 Scheme 阶乘函数定义。

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

我们可以用推导建模 `fact` 的计算过程：

```
  (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

这是阶乘的自然递归定义。每次调用 `fact` 都保证返回值与调用处的 `n` 相乘。这样，随着计算进行，`fact` 在越来越大的控制上下文中调用。比较这一行为与下列过程。

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

用这个定义，我们计算：

```
  (fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

这里，`fact-iter-acc` 总是在同样的控制上下文内调用：在本例中，是没有任何上下文。当 `fact-iter-acc` 调用自身时，它在 `fact-iter-acc` 执行的“尾端”，除了把返回值作为 `fact-iter-acc` 调用的结果，不需再做任何保证。我们称之为*尾调用* (*tail call*)。这样，上述推导中的每一步都形如 `(fact-iter-acc` n a`)`。

当 `fact` 这样的过程执行时，每次递归调用都要记录额外的控制信息，此信息保留到调用返回为止。在上面的第一个推导中，这反映了控制上下文的增长。这样的过程呈现出*递归性控制行为* (*recursive control behavior*)。

与之相对，`fact-iter-acc` 调用自身时，不需记录额外的控制信息。递归调用发生在表达式的同一层（上述推导的最外层）反映了这一点。在这种情况下，当递归深度（没有对应返回的递归调用数目）增加时，系统不需要不断增长的内存存放控制上下文。只需使用有限内存存放控制信息的过程呈现出*迭代性控制行为* (*iterative control behavior*)。

为什么这些程序呈现出不同的控制行为呢？在阶乘的递归定义中，过程 `fact` 在*操作数位置* (*operand position*) 调用。我们需要保存这个调用的上下文，因为我们需要记住，过程调用执行完毕之后，仍需求出操作数的值，并执行外层调用，在本例中，是完成待做的乘法。这给出一条重要原则：

不是过程调用，而是操作数的求值导致控制上下文扩大。

本章，我们学习如何跟踪和操作控制上下文。我们的核心工具是名为续文(*continuation*) 的数据类型。就像环境是数据上下文的抽象表示，续文是控制上下文的抽象表示。我们将探索续文，编写直接传递续文参数的解释器，就像之前直接传递环境参数的解释器。一旦处理了简单情况，我们就能明白如何给语言添加组件，以更加复杂的方式处理控制上下文，譬如异常和线程。

在第 6 章，我们展示如何用转换解释器的技术转换所有程序。我们说以这种方式转换而得的程序具有续文传递风格 (*continuation-passing style*)。第 6 章还展示了续文的其他一些重要应用。

5.1 传递续文的解释器

在我们的新解释器中，`value-of` 等主要过程将取第三个参数。这一参数••续文••用来抽象每个表达式求值时的控制上下文。

我们从，即 3.4 节中的 LETREC 语言解释器入手。我们把 `value-of-program` 的结果叫做 *FinalAnswer*，以强调这个表达值是程序的最终值。

FinalAnswer = *ExpVal*

value-of-program : *Program* → *FinalAnswer*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

value-of : *Exp* × *Env* → *ExpVal*

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (var-exp (var) (apply-env env var))
      (diff-exp (exp1 exp2)
        (let ((num1 (expval->num (value-of exp1 env)))
              (num2 (expval->num (value-of exp2 env))))
          (num-val (- num1 num2))))
      (zero?-exp (exp1)
        (let ((num1 (expval->num (value-of exp1 env))))
          (if (zero? num1) (bool-val #t) (bool-val #f))))
      (if-exp (exp1 exp2 exp3)
        (if (expval->bool (value-of exp1 env))
          (value-of exp2 env)
          (value-of exp3 env)))
      (let-exp (var exp1 body)
        (let ((val1 (value-of exp1 env)))
          (value-of body (extend-env var val1 env))))
      (proc-exp (var body)
        (proc-val (procedure var body env)))
      (call-exp (rator rand)
        (let ((proc1 (expval->proc (value-of rator env)))
              (arg (value-of rand env)))
          (apply-procedure proc1 arg)))
      (letrec-exp (p-name b-var p-body letrec-body)
        (value-of letrec-body
          (extend-env-rec p-name b-var p-body env))))))
```

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body (extend-env var val saved-env))))))
```

图 5.1 传递环境的解释器

我们的目标是重写解释器，避免在调用 value-of 时产生控制上下文。当控制上下文需要增加时，我们扩展续文参数，就像在第 3 章中，程序产生数据上下文时，扩展解释器的环境一样。彰显控制上下文，我们就能看到它如何消长。之后，从 5.4 节到 5.5 节，我们将用它给我们的语言添加新的控制行为。

现在，我们知道环境表示一个从符号到指代值的函数。续文表示什么呢？表达式的续文表示一个过程，它取表达式的结果，完成计算。所以我们的接口必须包含一个过程 apply-cont，它取一续文 cont，一个表达值 val，完成由 cont 指定的计算。apply-cont 的合约为：

FinalAnswer = *ExpVal*
apply-cont : *Cont* × *ExpVal* → *FinalAnswer*

我们把 apply-cont 的结果叫做 *FinalAnswer* 是为了提醒自己，它是计算最终的值：程序的其他部分都不用它。

接口应该包含什么样的续文构造器？随着我们分析解释器，这些续文构造器自会显现。首先，我们需要一个续文构造器，在不需再对计算值进行操作时生成上下文。我们把这个续文叫做 (end-cont)，其定义为：

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "计算结束.~%")
    val)
```

调用 (end-cont) 打印出一条计算结束消息，并返回程序的值。因为 (end-cont) 打印出一条消息，我们可以看出它调用了多少次。在正确的计算中，它只应调用一次。

我们把 value-of-program 重写为：

value-of-program : *Program* → *FinalAnswer*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))
```

现在我们可以写出 value-of/k。我们依次考虑 value-of 的各个分支。value-of 的前几行只是算出一个值，然后返回，不会再次调用 value-of。

在传递续文的解释器中，这些行调用 `apply-cont`，把对应的值传给续文：

value-of/k : $Exp \times Env \times Cont \to ExpVal$

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val (procedure var body env))))
      ...)))
```

目前，`cont` 唯一可能的值是终止续文，但这马上就会改变。很容易看出，如果程序为上述表达式之一，表达式的值将传给 `end-cont`（通过 `apply-cont`）。

`letrec` 的行为也不复杂：它不调用 `value-of`，而是创建一个新环境，然后在新环境中求主体的值，主体的值就是整个表达式的值。这表明主体和整个表达式在同样的控制上下文中执行。因此，主体的值应返回给整个表达式的续文。所以我们写：

```
(letrec-exp (p-name p-var p-body letrec-body)
  (value-of/k letrec-body
    (extend-env-rec p-name p-var p-body env)
    cont))
```

这解释了一条通用原则：

尾调用不扩大续文

若 exp_1 的值作为 exp_2 的值返回，则 exp_1 和 exp_2 应在同样的续文中执行。

写成这样是不对的：

```
(letrec-exp (p-name p-var p-body letrec-body)
  (apply-cont cont
```

```
          (value-of/k letrec-body
            (extend-env-rec p-name p-var p-body env)
            (end-cont))))
```

因为调用 value-of/k 是在操作数位置：它要作为 apply-cont 的操作数。另外，由于使用续文 (end-cont) 会在计算完成之前打印出计算结束消息，这种错误很容易排查。

接下来我们考虑 zero? 表达式。在 zero? 表达式中，我们得求出实参的值，然后返回给依赖该值的续文。所以我们要在新的续文中求实参的值，这个续文会取得返回值，然后做适当处理。

那么，在 value-of/k 中，我们写：

```
    (zero?-exp (exp1)
      (value-of/k exp1 env
        (zero1-cont cont)))
```

其中，(zero1-cont cont) 是一续文，具有如下性质：

```
(apply-cont (zero1-cont cont) val)
= (apply-cont cont
    (bool-val
      (zero? (expval->num val))))
```

就像 letrec，我们不能把 value-of/k 写成：

```
    (zero?-exp (exp1)
      (let ((val (value-of/k exp1 env (end-cont))))
        (apply-cont cont
          (bool-val
            (zero? (expval->num val))))))
```

因为调用 value-of/k 是在操作数位置。let 声明的右边是在操作数位置，因为 (let ((*var* exp_1)) exp_2) 等效于 ((lambda (*var*) exp_2) exp_1)。value-of/k 调用的值最终成为 expval->num 的操作数。像之前那样，如果我们运行这段代码，计算结束消息会出现两次：一次在计算中间，一次在真正结束时。

let 表达式只比 zero? 表达式稍微复杂一点：求出声明右侧的值之后，我们在适当的扩展环境内求主体的值。原来的 let 代码为：

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))
```

在传递续文的解释器中，我们在完成剩余计算的上下文中求 exp_1 的值。所以，在 `value-of/k` 中我们写：

```
(let-exp (var exp1 body)
  (value-of/k exp1 env
    (let-exp-cont var body env cont)))
```

然后我们给续文的接口添加规范：

```
(apply-cont (let-exp-cont var body env cont) val)
= (value-of/k body (extend-env var val env) cont)
```

`let` 表达式主体的值成为 `let` 表达式的值，所以求 `let` 表达式主体时的续文与求整个 `let` 表达式的相同。这是**尾调用不扩大续文**的又一例子。

下面我们处理 `if` 表达式。在 `if` 表达式中，我们首先求条件的值，但条件的结果不是整个表达式的值。我们要新生成一个续文，查看条件表达式的结果是否为真，然后求真值表达式或假值表达式的值。所以在 `value-of/k` 中我们写：

```
(if-exp (exp1 exp2 exp3)
  (value-of/k exp1 env
    (if-test-cont exp2 exp3 body env cont)))
```

其中，`if-test-cont` 是另一个续文构造器，满足如下规范：

```
(apply-cont (if-test-cont exp2 exp3 env cont) val)
= (if (expval->bool val)
    (value-of/k exp2 env cont)
    (value-of/k exp3 env cont))
```

现在，我们有了四个续文构造器。我们可以用过程表示法或者数据结构表示法实现它们。过程表示法如所示，数据结构表示法使用 `define-datatype`，如所示。

Cont = ExpVal -> FinalAnswer

end-cont : () → *Cont*

```
(define end-cont
  (lambda ()
    (lambda (val)
      (begin
        (eopl:printf "计算结束.~%")
        val))))
```

zero1-cont : *Cont* → *Cont*

```
(define zero1-cont
  (lambda (cont)
    (lambda (val)
      (apply-cont cont
        (bool-val
          (zero? (expval->num val)))))))
```

let-exp-cont : *Var* × *Exp* × *Env* × *Cont* → *Cont*

```
(define let-exp-cont
  (lambda (var body env cont)
    (lambda (val)
      (value-of/k body (extend-env var val env) cont))))
```

if-test-cont : *Exp* × *Exp* × *Env* × *Cont* → *Cont*

```
 : Exp × Exp × Env × Cont → Cont
(define if-test-cont
  (lambda (exp2 exp3 env cont)
    (lambda (val)
      (if (expval->bool val)
        (value-of/k exp2 env cont)
        (value-of/k exp3 env cont)))))
```

apply-cont : *Cont* × *ExpVal* → *FinalAnswer*

```
(define apply-cont
  (lambda (cont v)
    (cont v)))
```

图 5.2 用过程表示续文

```
(define-datatype continuation continuation?
  (end-cont)
  (zero1-cont
    (cont continuation?))
  (let-exp-cont
    (var identifier?)
    (body expression?)
    (env environment?)
    (cont continuation?))
  (if-test-cont
    (exp2 expression?)
    (exp3 expression?)
    (env environment?)
    (cont continuation?)))
```

apply-cont : $Cont \times ExpVal \to FinalAnswer$

```
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont ()
        (begin
          (eopl:printf "计算结束.~%")
          val))
      (zero1-cont (saved-cont)
        (apply-cont saved-cont
          (bool-val
            (zero? (expval->num val)))))
      (let-exp-cont (var body saved-env saved-cont)
        (value-of/k body
          (extend-env var val saved-env) saved-cont))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (if (expval->bool val)
          (value-of/k exp2 saved-env saved-cont)
          (value-of/k exp3 saved-env saved-cont))))))
```

图 5.3 用数据结构表示续文

下面这个简单算例展示了各部分如何配合。像 3.3 节那样，我们用 •exp• 指代表达式 exp 的抽象语法树。设 ρ_0 是一环境，b 在其中绑定到 (bool-val #t)；$cont_0$ 是初始续文，即 (end-cont) 的值。注释说明不是正式的，应与 value-of/k 的定义和 apply-cont 的规范对照阅读。这个例子是预测性的，因为我们让 letrec 引入了过程，但还不知道如何调用它。

```
(value-of/k <<letrec p(x) = x in if b then 3 else 4>>
  ρ0 cont0)
= 令 ρ0 为 (extend-env-rec ... ρ0)
(value-of/k <<if b then 3 else 4>> ρ1 cont0)
= 然后，求条件表达式的值
(value-of/k <<b>> ρ1 (test-cont <<3>> <<4>> ρ1 cont0))
= 把 b 的值传给续文
(apply-cont (test-cont <<3>> <<4>> ρ1 cont0)
            (bool-val #f))
= 求真值表达式
(value-of/k <<3>> ρ1 cont0)
= 把表达式的值传给续文
(apply-cont cont0 (num-val 3))
= 在最后的续文中处理最终答案
(begin (eopl:printf ...) (num-val 3))
```

差值表达式给我们的解释器带来了新困难，因为它得求两个操作数的值。我们还像 if 那样开始，先求第一个实参：

```
(diff-exp (exp1 exp2)
  (value-of/k exp1 env
    (diff1-cont exp2 env cont)))
```

当 (diff1-cont exp2 env cont) 收到一个值，它应求 exp2 的值，求值时的上下文应保存 exp1 的值。我们将其定义为：

```
(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env
    (diff2-cont val1 cont))
```

当 (diff2-cont val1 cont) 收到一个值，我们得到了两个操作数的值，所以，我们可以把二者的差继续传给等待中的 cont。定义为：

```
    (apply-cont (diff2-cont val1 cont) val2)
    = (let ((num1 (expval->num val1))
            (num2 (expval->num val2)))
        (apply-cont cont
          (num-val (- num1 num2))))
```

让我们看看该系统的例子。

```
(value-of/k
  <<-(-(44,11),3)>>
  ρ0
  #(struct:end-cont))
= 开始处理第一个操作数
(value-of/k
  <<-(44,11)>>
  ρ0
  #(struct:diff1-cont <<3>> ρ0
     #(struct:end-cont)))
= 开始处理第一个操作数
(value-of/k
  <<44>>
  ρ0
  #(struct:diff1-cont <<11>> ρ0
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont))))
= 把 <<44>> 的值传给续文
(apply-cont
  #(struct:diff1-cont <<11>> ρ0
    #(struct:diff1-cont <<3>> ρ0
       #(struct:end-cont)))
  (num-val 44))
= 现在, 开始处理第二个操作数
(value-of/k
  <<11>>
  ρ0
  #(struct:diff2-cont (num-val 44)
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont))))
= 把值传给续文
(apply-cont
  #(struct:diff2-cont (num-val 44)
     #(struct:diff1-cont <<3>> ρ0
```

```
        #(struct:end-cont)))
  (num-val 11))
= 44 − 11 等于 33，传给续文
(apply-cont
  #(struct:diff1-cont <<3>> ρ0
     #(struct:end-cont))
  (num-val 33))
= 开始处理第二个操作数 <<3>>
(value-of/k
  <<3>>
  ρ0
  #(struct:diff2-cont (num-val 33)
     #(struct:end-cont)))
= 把值传给续文
(apply-cont
  #(struct:diff2-cont (num-val 33)
     #(struct:end-cont))
  (num-val 3))
= 33 − 3 等于 30，传给续文
(apply-cont
  #(struct:end-cont)
  (num-val 30))
```

`apply-cont` 打印出消息“计算结束”，返回计算的最终结果 `(num-val 30)`。

我们的语言中最后要处理的是过程调用。在传递环境的解释器中，我们写：

```
(call-exp (rator rand)
  (let ((proc1 (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc1 arg)))
```

就像在 `diff-exp` 中一样，这里要处理两个调用。所以我们必须先择其一，然后转换余下部分来处理第二个。此外，我们必须把续文传给 `apply-procedure`，因为 `apply-procedure` 要调用 `value-of/k`。

我们选择先求操作符的值，所以在 `value-of/k` 中我们写：

```
(call-exp (rator rand)
  (value-of/k rator
    (rator-cont rand env cont)))
```

就像 `diff-exp`，`rator-cont` 在适当的环境中求操作数的值：

```
(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env
    (rand-cont val1 cont))
```

当 rand-cont 收到一个值，它就可以调用过程了：

```
(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))
```

最后，我们还要修改 apply-procedure，以符合续文传递风格：

apply-procedure/k : $Proc \times ExpVal \times Cont \to FinalAnswer$

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```

传递续文的解释器展示完毕。完整的解释器如和 fig-5.5 所示。续文的完整规范如所示。

value-of-program : *Program* → *FinalAnswer*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))
```

value-of/k : *Exp* × *Env* × *Cont* → *FinalAnswer*

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val
            (procedure var body env))))
      (letrec-exp (p-name b-var p-body letrec-body)
        (value-of/k letrec-body
          (extend-env-rec p-name b-var p-body env)
          cont))
      (zero?-exp (exp1)
        (value-of/k exp1 env
          (zero1-cont cont)))
      (if-exp (exp1 exp2 exp3)
        (value-of/k exp1 env
          (if-test-cont exp2 exp3 env cont)))
      (let-exp (var exp1 body)
        (value-of/k exp1 env
          (let-exp-cont var body env cont)))
      (diff-exp (exp1 exp2)
        (value-of/k exp1 env
          (diff1-cont exp2 env cont)))
      (call-exp (rator rand)
        (value-of/k rator env
          (rator-cont rand env cont))))))
```

图 5.4 传递续文的解释器（第 1 部分）

apply-procedure : $Proc \times ExpVal \times FinalAnswer \to FinalAnswer$

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```

图 5.5 传递续文的解释器（第 2 部分）

现在我们可以验证断言：不是过程调用，而是实参的求值扩大了控制上下文。具体来说，如果我们在某个续文 $cont_1$ 中求过程调用 (`(`exp_1 exp_2`)`) 的值，求 exp_1 得到的过程主体也将在 $cont_1$ 中求值。

但过程调用本身不会增大控制上下文。考虑 (`(`exp_1 exp_2`)`) 的求值，其中 exp_1 的值是一个过程 $proc_1$，exp_2 的值是某个表达值 val_2。

```
(value-of/k <<(exp1 exp2)>> ρ1 cont1)
= 求操作符的值
(value-of/k <<exp1>> ρ1
  (rator-cont <<exp2>> ρ1 cont1))
= 把过程值传给续文
(apply-cont
  (rator-cont <<exp2>> ρ1 cont1)
  proc1)
= 求操作符的值
(value-of/k <<exp2>> ρ1
  (rand-cont <<proc1>> cont1))
= 把参数值传给续文
(apply-cont
  (rand-cont <<proc1>> cont1)
  val2)
= 调用过程
(apply-procedure/k proc1 val2 cont1)
```

所以，过程调用时，过程主体在过程调用所在的续文中求值。操作数的求值需要控制上下文，进入过程主体则不需要。

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf
      "计算结束.~%")
    val)

(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env (diff2-cont val1 cont))

(apply-cont (diff2-cont val1 cont) val2)
= (let ((num1 (expval->num val1))
        (num2 (expval->num val2)))
    (apply-cont cont (num-val (- num1 num2))))

(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env (rand-cont val1 cont))

(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))

(apply-cont (zero1-cont cont) val)
= (apply-cont cont (bool-val (zero? (expval->num val))))

(apply-cont (if-test-cont exp2 exp3 env cont) val)
= (if (expval->bool val)
    (value-of/k exp2 env cont)
    (value-of/k exp3 env cont))

(apply-cont (let-exp-cont var body env cont) val1)
= (value-of/k body (extend-env var val1 env) cont)
```

图 5.6 中续文的规范

练习 5.1 [⋆] 用过程表示法实现续文数据类型。

练习 5.2 [⋆] 用数据结构表示法实现续文数据类型。

练习 5.3 [⋆] 给解释器添加 `let2`。`let2` 表达式就像 `let` 表达式，但要指定两个变量。

练习 5.4 [⋆] 给解释器添加 `let3`。`let3` 表达式就像 `let` 表达式，但要指定三个变量。

练习 5.5 [⋆] 给语言添加 ex3.9 中的列表。

练习 5.6 [⋆⋆] 给语言添加 ex3.10 中的 `list` 表达式。提示：添加两个续文构造器，一个用来求列表首元素的值，一个用来求列表剩余元素的值。

练习 5.7 [⋆⋆] 给解释器添加多声明的 `let`（ex3.16）。

练习 5.8 [⋆⋆] 给解释器添加多参数过程（ex3.21）。

练习 5.9 [⋆⋆] 修改这个解释器，实现 IMPLICIT-REFS 语言。提示：添加新的续文构造器 `(set-rhs-cont env var cont)`。

练习 5.10 [⋆⋆] 修改前一题的解答，不要在续文中保存环境。

练习 5.11 [⋆⋆] 给传递续文的解释器添加 ex4.4 中的 `begin` 表达式。确保调用 `value-of` 和 `value-of-rands` 时不需要生成控制上下文。

练习 5.12 [⋆] 给•fig-5.6 的解释器添加辅助过程，生成类似 cps-computation 计算的输出。

练习 5.13 [⋆] 把 `fact` 和 `fact-iter` 翻译为 LETREC 语言。你可以给语言添加乘法操作符。然后，用前一道练习中带有辅助组件的解释器计算 `(fact 4)` 和 `(fact-iter 4)`。将它们和本章开头的计算比较。在 `(fact 4)` 的跟踪日志中找出 `(* 4 (* 3 (* 2 (fact 1))))`。调用 `(fact 1)` 时，`apply-procedure/k` 的续文是什么？

练习 5.14 [⋆] 前面练习中的辅助组件产生大量输出。修改辅助组件，只跟踪计算过程中

最大续文的尺寸。我们用续文构造器的使用次数衡量续文的大小，所以 cps-computation 的计算中，续文最大尺寸是 3。然后，用 `fact` 和 `fact-iter` 计算几个操作数的值。验证 `fact` 使用的续文最大尺寸随其参数递增，但 `fact-iter` 使用的续文最大尺寸是常数。

练习 5.15 [⋆] 我们的续文数据类型只有一个常量 `end-cont`，所有其他续文构造器都有一个续文参数。用列表表示和实现续文。用空列表表示 `end-cont`，用首项为其他数据结构（名为帧 (*frame*) 或活跃记录表 (*activation record*)），余项为已保存续文的非空列表表示其他续文。观察可知，解释器把这些列表当成（帧的）堆栈。

练习 5.16 [⋆⋆] 扩展传递续文的解释器，处理 ex4.22 中的语言。给 `result-of` 传递一个续文参数，确保 `result-of` 不在扩大控制上下文的位置调用。因为语句不返回值，需要区分普通续文和语句续文；后者通常叫命令续文 (*command continuation*)。续文接口应包含过程 `apply-command-cont`，它取一命令续文并使用它。用数据结构和无参数过程两种方式实现命令续文。

5.2 跳跃式解释器

有人可能想用普通的过程式语言转译解释器，使用数据结构表示续文，从而避免高阶函数。但是，用大多数过程式语言做这种翻译都很困难：它们不只在必要时才扩大控制上下文，而且在每个函数调用处扩大控制上下文（即堆栈！）。在我们的系统中，由于过程调用在计算结束之前不返回，在那之前，系统的堆栈将一直增高。

这种行为不无道理：在这种语言中，几乎所有的过程调用都出现在赋值语句的右边，所以几乎所有过程调用都要扩大控制上下文，以便记录待完成的赋值。因此，体系结构为这种最常见的情形做了优化。而且，由于大多数语言在堆栈中存储环境信息，所有过程调用生成的控制上下文都不能忘了移除这一信息。

在这种语言中，一种解决方案是使用跳跃 (*trampolining*) 技术。为了避免产生无限长的调用链，我们把调用链打断，让解释器中的某个过程返回一个无参数过程。这个过程在调用时继续计算。整个计算由一个名叫跳床 (*trampoline*)

的过程驱动，它从一个过程调用弹射到另一个。例如，我们可以在 apply-procedure/k 的主体周围插入一个 (lambda () ...)，因为在我们的语言中，只要不执行过程调用，表达式的运行时间就是有限的。

得出的代码如所示，它也展示了解释器中所有的尾调用。因为我们修改了 apply-procedure/k，不再让它返回一个 *ExpVal*，而是返回一个过程，我们得重写它和它所调用所有过程的合约。所以，我们必须检查解释器中所有过程的合约。

我们从 value-of-program 开始。由于这是调用解释器的过程，它的合约保持不变。它调用 value-of/k，把结果传给 trampoline。因为我们要操作 value-of/k 的结果，所以它不是 *FinalAnswer*。我们明明没有修改 value-of/k 的代码，怎么会这样呢？过程 value-of/k 在尾部递归调用 apply-cont，apply-cont 在尾部递归调用 apply-procedure/k，所以 apply-procedure/k 的任何结果都可能成为 value-of/k 的结果。而我们修改了 apply-procedure/k，它的返回值与之前不同。

我们引入*弹球* (*Bounce*)，作为 value-of/k 的可能结果（我们叫它弹球，因为它是跳床的输入）。这一集合的值是什么呢？value-of/k 在尾部递归调用自身和 apply-cont，这些是它里面所有的尾递归。所以能成为 value-of/k 结果的值只能是 apply-cont 的结果。而且，apply-procedure/k 在尾部递归调用 value-of/k，所以不论 *Bounce* 是什么，它是 value-of/k、apply-cont 和 apply-procedure/k 结果的集合。

过程 value-of/k 和 apply-cont 只是在尾部调用其他过程。真正把值放入 *Bounce* 中的是 apply-procedure/k。这些是什么样的值呢？我们来看代码。

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (... (value-of/k body ...)))))
```

已知 apply-procedure/k 返回无参数的过程，该过程在调用时返回一个 *ExpVal*，或调用 value-of/k、apply-cont 或 apply-procedure/k 之

一的结果，也就是 *Bounce*。所以，`apply-procedure/k` 可能的取值由如下集合描述：

$$ExpVal \cup (() \to Bounce)$$

这和 `value-of/k` 的可能结果相同，所以我们得出结论：

$$Bounce = ExpVal \cup (() \to Bounce)$$

合约为：

value-of-program : $Program \to FinalAnswer$
trampoline : $Bounce \to FinalAnswer$
value-of/k : $Exp \times Env \times Cont \to Bounce$
apply-cont : $Cont \times ExpVal \to Bounce$
apply-procedure/k : $Proc \times ExpVal \times FinalAnswer \to Bounce$

过程 `trampoline` 满足其合约：首先给它传入一个 *Bounce*。如果其参数是一个 *ExpVal*（也是 *FinalAnswer*），那么返回；否则，参数一定是一个返回值为 *Bounce* 的过程。所以，它调用这个无参数过程，然后调用自身处理其结果，返回值总是一个 *Bounce*（在 7.4 节我们将看到如何自动完成这个推理过程）。

`apply-procedure/k` 返回的每个无参数过程都表示计算流程中的一个快照。我们可以在计算中的不同位置返回这样的快照。在 5.5 节，我们将看到如何用这一思想模拟多线程程序中的原子操作。

Bounce = *ExpVal* ∪ (() → *Bounce*)

value-of-program : *Program* → *FinalAnswer*
```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp)
        (trampoline
          (value-of/k exp (init-env) (end-cont)))))))
```

trampoline : *Bounce* → *FinalAnswer*
```
(define trampoline
  (lambda (bounce)
    (if (expval? bounce)
      bounce
      (trampoline (bounce)))))
```

value-of/k : *Exp* × *Env* × *Cont* → *Bounce*
```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (... (value-of/k ...))
      (... (apply-cont ...))))))
```

apply-cont : *Cont* × *ExpVal* → *Bounce*
```
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (... val)
      (... (value-of/k ...))
      (... (apply-cont ...))
      (... (apply-procedure/k ...))))))
```

apply-procedure/k : *Proc* × *ExpVal* × *Cont* → *Bounce*
```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (lambda ()
      (cases procedure proc1
        (... (value-of/k ...))))))
```

图 5.7 用过程表示跳床

练习 5.17 [⋆] 修改跳跃式解释器，把所有调用 `apply-procedure/k` 的地方（只有一处）放入 `(lambda () ...)` 中。这一修改需要更改合约吗?

练习 5.18 [⋆] 中的跳床系统使用过程表示 *Bounce*。改用数据结构表示法。

练习 5.19 [⋆] 不要在 `apply-procedure/k` 主体周围插入 `(lambda () ...)`，改为在 `apply-cont` 的主体周围插入。修改合约，使之符合这一更改。*Bounce* 的定义需要修改吗？然后，用数据结构表示法替换过程表示法表示 *Bounce*，像 ex5.18 那样。

练习 5.20 [⋆] 在 ex5.18 中，`trampoline` 返回 *FinalAnswer* 之前的最后一颗弹球形如 `(apply-cont (end-cont)` *val*`)`，其中，*val* 是 *ExpVal*。利用这一点优化 ex5.19 中弹球的表示。

练习 5.21 [⋆⋆] 用普通的过程式语言实现跳跃式解释器。用 ex5.18 中的数据结构表示快照，把 `trampoline` 中对自身的递归调用替换为普通的 `while` 或其它循环结构。

练习 5.22 [⋆⋆⋆] 有人可能想用普通的过程式语言转译第 3 章中传递环境的解释器。同样是因为上述原因，除了最简单的情况，这种转换都会失败。跳跃技术在这种情况下也有效吗?

5.3 指令式解释器

在第 4 章中我们看到，给共享变量赋值有时可以替代绑定。考虑顶部的老例子 `even` 和 `odd`。

可以用中间的程序替代它们。其中，共享变量 `x` 供两个过程交换信息。在顶部的例子中，过程主体在环境中查找相关数据；在另一个程序中，它们从存储器中查找相关数据。

考虑底部的计算跟踪日志。它可能是二者中任一计算跟踪日志。当我们记录调用的过程和实参时，它是第一个计算的跟踪日志；当我们记录调用的过程和寄存器 `x` 的值时，它是第二个计算的跟踪日志。

而当我们记录程序计数器的位置和寄存器 **x** 的内容时，这又可以解释为 *goto*（名为流程图程序 (*flowchart program*)）的跟踪日志。

```
letrec
 even(x) = if zero?(x)
           then 1
           else (odd sub1(x))
 odd(x) = if zero?(x)
          then 0
          else (even sub1(x))
in (odd 13)
```

```
let x = 0
in letrec
    even() = if zero?(x)
             then 1
             else let d = set x = sub1(x)
                  in (odd)
    odd() = if zero?(x)
            then 0
            else let d = set x = sub1(x)
                 in (even)
   in let d = set x = 13
      in (odd)
```

```
      x = 13;
      goto odd;
even: if (x=0) then return(1)
               else {x = x-1;
                     goto odd;}
odd:  if (x=0) then return(0)
               else {x = x-1;
                     goto even;}
```

```
  (odd 13)
= (even 12)
= (odd 11)
...
= (odd 1)
= (even 0)
= 1
```

图 5.8 跟踪日志相同的三个程序

能如此，只是因为原代码中 `even` 和 `odd` 的调用不扩大控制上下文：它们是尾调用。我们不能这样转换 `fact`，因为 `fact` 的跟踪日志无限增长：“程序计数器”不是像这里一样出现在跟踪日志的最外层，而是出现在控制上下文中。

任何不需要控制上下文的程序都可以这样转换。这给了我们一条重要原理：

> **无参数的尾调用等同于跳转。**

如果一组过程只通过尾调用互相调用，那么我们可以像像那样，翻译程序，用赋值代替绑定，然后把赋值程序转译为流程图程序。

本节，我们用这一原理翻译传递续文的解释器，将其转换为适合无高阶过程语言的形式。

我们首先从和 fig-5.5 中的解释器开始，用数据结构表示续文。续文的数据结构表示如和 fig-5.10 所示。

```
(define-datatype continuation continuation?
  (end-cont)
  (zero1-cont
    (saved-cont continuation?))
  (let-exp-cont
    (var identifier?)
    (body expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (if-test-cont
    (exp2 expression?)
    (exp3 expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (diff1-cont
    (exp2 expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (diff2-cont
    (val1 expval?)
    (saved-cont continuation?))
  (rator-cont
    (rand expression?)
    (saved-env environment?)
    (saved-cont continuation?))
  (rand-cont
    (val1 expval?)
    (saved-cont continuation?)))
```

图 5.9 用数据结构实现的续文（第 1 部分）

我们的第一个任务是列出需要通过共享寄存器通信的过程。这些过程及其形参为：

```
(value-of/k exp env cont)
(apply-cont cont val)
(apply-procedure/k proc1 val cont)
```

所以我们需要五个寄存器：`exp`、`env`、`cont`、`val` 和 `proc1`。上面的三个过程各改为一个无参数过程，每个实参的值存入对应的寄存器，调用无参数过程，换掉上述过程的调用。所以，这段代码

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      ...)))
```

可以替换为：

```
(define value-of/k
  (lambda ()
    (cases expression exp
      (const-exp (num)
        (set! cont cont)
        (set! val (num-val num))
        (apply-cont))
      ...)))
```

apply-cont : $Cont \times ExpVal \to Bounce$

```
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont ()
        (begin
          (eopl:printf
            "计算结束.~%")
          val))
      (zero1-cont (saved-cont)
        (apply-cont saved-cont
          (bool-val
            (zero? (expval->num val)))))
      (let-exp-cont (var body saved-env saved-cont)
        (value-of/k body
          (extend-env var val saved-env) saved-cont))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (if (expval->bool val)
          (value-of/k exp2 saved-env saved-cont)
          (value-of/k exp3 saved-env saved-cont)))
      (diff1-cont (exp2 saved-env saved-cont)
        (value-of/k exp2
          saved-env (diff2-cont val saved-cont)))
      (diff2-cont (val1 saved-cont)
        (let ((num1 (expval->num val1))
              (num2 (expval->num val)))
          (apply-cont saved-cont
            (num-val (- num1 num2)))))
      (rator-cont (rand saved-env saved-cont)
        (value-of/k rand saved-env
          (rand-cont val saved-cont)))
      (rand-cont (val1 saved-cont)
        (let ((proc (expval->proc val1)))
          (apply-procedure/k proc val saved-cont))))))
```

图 5.10 用数据结构实现的续文（第 2 部分）

现在，我们依次转换四个过程。我们还要修改 value-of-program 的主体，因为那是最初调用 value-of/k 的地方。只有三点小麻烦：

1. 存储器在过程调用之间往往保持不变。这对应上例中的赋值 (set! cont cont)。我们大可移除这样的赋值。

2. 我们必须确保 cases 表达式中的字段不与寄存器重名。否则字段会遮蔽寄存器，寄存器就无法访问了。例如，在 value-of-program 中，如果我们写：

```
(cases program pgn
  (a-program (exp)
    (value-of/k exp (init-env) (end-cont))))
```

那么 exp 绑定到局部变量，我们无法给全局寄存器 exp 赋值。解决方法是重命名局部变量，避免冲突：

```
(cases program pgn
  (a-program (exp1)
    (value-of/k exp1 (init-env) (end-cont))))
```

然后，可以写：

```
(cases program pgn
  (a-program (exp1)
    (set! cont (end-cont))
    (set! exp exp1)
    (set! env (init-env))
    (value-of/k)))
```

我们已仔细挑选数据类型中的字段名，避免这种冲突。

3. 一次调用中如果两次使用同一寄存器，又会造成一点麻烦。考虑转换 (cons (f (car x)) (f (cdr x))) 中的第一个调用，其中，x 是 f 的形参。不做过多考虑的话，这个调用可以转换为：

```
(begin
  (set! x (car x))
  (set! cont (arg1-cont x cont))
  (f))
```

但这是不对的，因为它给寄存器 `x` 赋了新值，但 `x` 原先的值还有用。解决方法是调整赋值顺序，把正确的值放入寄存器中，或者使用临时变量。大多情况下，要避免这种问题，可以先给续文变量赋值：

```
(begin
  (set! cont (arg1-cont x cont))
  (set! x (car x))
  (f))
```

有时临时变量无法避免；考虑 `(f y x)`，其中 `x` 和 `y` 是 `f` 的形参。我们的例子中还未遇到这种麻烦。

翻译完的解释器如•fig-5.14 所示。这个过程叫做寄存 (*registerization*)。很容易用支持跳转的指令式语言翻译它。

```
(define exp 'uninitialized)
(define env 'uninitialized)
(define cont 'uninitialized)
(define val 'uninitialized)
(define proc1 'uninitialized)

value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (set! cont (end-cont))
        (set! exp exp1)
        (set! env (init-env))
        (value-of/k)))))

value-of/k : () → FinalAnswer
用法: 依赖寄存器
exp : Exp
env : Env
cont : Cont
(define value-of/k
  (lambda ()
    (cases expression exp
      (const-exp (num)
        (set! val (num-val num))
        (apply-cont))
      (var-exp (var)
        (set! val (apply-env env var))
        (apply-cont))
      (proc-exp (var body)
        (set! val (proc-val (procedure var body env)))
        (apply-cont))
      (letrec-exp (p-name b-var p-body letrec-body)
        (set! exp letrec-body)
        (set! env (extend-env-rec p-name b-var p-body env))
        (value-of/k))
```

图 5.11 指令式解释器（第 1 部分）

```
(zero?-exp (exp1)
  (set! cont (zero1-cont cont))
  (set! exp exp1)
  (value-of/k))
(let-exp (var exp1 body)
  (set! cont (let-exp-cont var body env cont))
  (set! exp exp1)
  (value-of/k))
(if-exp (exp1 exp2 exp3)
  (set! cont (if-test-cont exp2 exp3 env cont))
  (set! exp exp1)
  (value-of/k))
(diff-exp (exp1 exp2)
  (set! cont (diff1-cont exp2 env cont))
  (set! exp exp1)
  (value-of/k))
(call-exp (rator rand)
  (set! cont (rator-cont rand env cont))
  (set! exp rator)
  (value-of/k)))))
```

图 5.12 指令式解释器（第 2 部分）

apply-cont : () → *FinalAnswer*
用法: 读取寄存器
`cont` : *Cont*
`val` : *ExpVal*

```
(define apply-cont
  (lambda ()
    (cases continuation cont
      (end-cont ()
        (eopl:printf "计算结束.~%")
        val)
      (zero1-cont (saved-cont)
        (set! cont saved-cont)
        (set! val (bool-val (zero? (expval->num val))))
        (apply-cont))
      (let-exp-cont (var body saved-env saved-cont)
        (set! cont saved-cont)
        (set! exp body)
        (set! env (extend-env var val saved-env))
        (value-of/k))
      (if-test-cont (exp2 exp3 saved-env saved-cont)
        (set! cont saved-cont)
        (if (expval->bool val)
          (set! exp exp2)
          (set! exp exp3))
        (set! env saved-env)
        (value-of/k))
```

图 5.13 指令式解释器（第 3 部分）

```
(diff1-cont (exp2 saved-env saved-cont)
  (set! cont (diff2-cont val saved-cont))
  (set! exp exp2)
  (set! env saved-env)
  (value-of/k))
(diff2-cont (val1 saved-cont)
  (let ((num1 (expval->num val1))
        (num2 (expval->num val)))
    (set! cont saved-cont)
    (set! val (num-val (- num1 num2)))
    (apply-cont)))
(rator-cont (rand saved-env saved-cont)
  (set! cont (rand-cont val saved-cont))
  (set! exp rand)
  (set! env saved-env)
  (value-of/k))
(rand-cont (rator-val saved-cont)
  (let ((rator-proc (expval->proc rator-val)))
    (set! cont saved-cont)
    (set! proc1 rator-proc)
    (set! val val)
    (apply-procedure/k))))))
```

apply-procedure/k : () → *FinalAnswer*
用法: 依赖寄存器
`proc1` : *Proc*
`val` : *ExpVal*
`cont` : *Cont*

```
(define apply-procedure/k
  (lambda ()
    (cases proc proc1
      (procedure (var body saved-env)
        (set! exp body)
        (set! env (extend-env var val saved-env))
        (value-of/k)))))
```

图 5.14 指令式解释器（第 4 部分）

练习 5.23 [⋆] 如果删去解释器某一分支中的“goto”会怎样？解释器会出什么错？

练习 5.24 [⋆] 设计一些例子，解释上文提到的每个麻烦。

练习 5.25 [⋆⋆] 寄存支持多参数过程的解释器（ex3.21）。

练习 5.26 [⋆] 用跳床转换这个解释器，用 `(set! pc apply-procedure/k)` 替换 `apply-procedure/k` 的调用，并使用下面这样的驱动器：

```
(define trampoline
  (lambda (pc)
    (if pc (trampoline (pc)) val)))
```

练习 5.27 [⋆] 设计一个语言特性，导致最后给 `cont` 赋值时，必须用临时变量。

练习 5.28 [⋆] 给本节的解释器添加 ex5.12 中的辅助组件。由于续文表示方式相同，可以复用那里的代码。验证本节的指令式解释器生成的跟踪日志与 ex5.12 中的解释器完全相同。

练习 5.29 [⋆] 转换本节的 `fact-iter`（fact-iter）。

练习 5.30 [⋆⋆] 修改本节的解释器，让过程使用 ex3.28 中的动态绑定。提示：像本章这样转换 ex3.28 中的解释器；二者不同的部分转换后才会不同。像 ex5.28 那样给解释器添加辅助组件。观察可知，就像当前状态中只有一个续文，当前状态只会压入或弹出一个环境，而且环境与续文同时压入或弹出。由此我们得出结论，动态绑定具有动态期限 (*dynamic extent*)：即，形参的绑定保留到过程返回为止。词法绑定则与之不同，绑定包裹在闭包内时可以无限期地保留。

练习 5.31 [⋆] 添加全局寄存器，排除本节代码中剩余的 `let` 表达式。

练习 5.32 [⋆⋆] 改进前一题的解答，尽可能减少全局寄存器的数量。不到 5 个就可以。除了本节解释器中已经用到的，不要使用其他数据结构。

练习 5.33 [⋆⋆] 把本节的解释器翻译为指令式语言。做两次，一次使用宿主语言中的无参数过程调用，一次使用 goto。计算量增加时，这二者性能如何?

练习 5.34 [⋆⋆] 如 imperative-lang 所述，用大多数指令式语言都难以完成这种翻译，因为它们在所有过程调用中使用堆栈，即使是尾调用。而且，对大型解释器，由 goto 链接的代码可能太过庞大，以致某些编译器无法处理。把本节的解释器翻译为指令式语言，用 ex5.26 中的跳跃技术规避这一难题。

5.4 异常

迄今为止，我们只用续文管理语言中的普通控制流。但是续文还能让我们修改控制上下文。让我们来给我们的语言添加异常处理 (*exception handling*)。我们给语言新增两个生成式:

Expression ::= `try` *Expression* `catch` `(` *Identifier* `)` *Expression*
`try-exp (exp1 var handler-exp)`

Expression ::= `raise` *Expression*
`raise-exp (exp)`

try 表达式在 catch 从句描述的异常处理上下文中求第一个参数的值。如果该表达式正常返回，它的值就是整个 try 表达式的值，异常处理器（即 handler-exp）则被移除。

raise 表达式求出参数的值，以该值抛出一个异常。这个值会传给最接近的异常处理器，并绑定到这个处理器的变量。然后，处理器主体将被求值。处理器主体可以返回一个值，这个值称为对应 try 表达式的值；或者，它可以抛出另一个异常，将异常传播 (*propagate*) 出去；这时，该异常会传给第二接近的异常处理器。

这里是一个例子（暂时假设我们给语言添加了字符串）。

```
let list-index =
     proc (str)
      letrec inner (lst)
```

```
          = if null?(lst)
            then raise("ListIndexFailed")
            else if string-equal?(car(lst), str)
                 then 0
                 else -((inner cdr(lst)), -1)
```

过程 `list-index` 是个咖喱式过程，它取一个字符串，一个字符串列表，返回字符串在列表中的位置。如果找不到期望的列表元素，`inner` 抛出一个异常，跳过所有待做的减法，把 `"ListIndexFailed"` 传给最接近的异常处理器。

这个异常处理器可以利用调用处的信息对异常做适当处理。

```
let find-member-number =
     proc (member-name)
      ... try ((list-index member-name) member-list)
            catch (exn)
             raise("CantFindMemberNumber")
```

过程 `find-member-number` 取一字符串，用 `list-index` 找出字符串在列表 `member-name` 中的位置。`find-member-number` 的调用者没办法知道 `list-index`，所以 `find-member-number` 把错误消息翻译成调用者能够理解的异常。

根据程序的用途，还有一种可能是，元素名不在列表中时，`find-member-number` 返回一个默认值。

```
let find-member-number =
     proc (member-name)
      ... try ((list-index member-name) member-list)
           catch (exn)
            the-default-member-number
```

在这些程序中，我们忽略了异常的值。在其他情况下，`raise` 传出的值可能包含一部分可供调用者利用的信息。

用传递续文的解释器实现这种异常处理机制直截了当。我们从 `try` 表达式开始。在续文的数据结构表示中，我们添加两个构造器：

```
(try-cont
  (var identifier?)
  (handler-exp expression?)
  (env environment?)
  (cont continuation?))
```

```
(raise1-cont
  (saved-cont continuation?))
```

在 value-of/k 中，我们给 try 添加下面的从句：

```
(try-exp (exp1 var handler-exp)
  (value-of/k exp1 env
    (try-cont var handler-exp env cont)))
```

求 try 表达式主体的值时会发生什么呢？如果主体正常返回，那么这个值应该传给 try 表达式的续文，也就是此处的 cont：

(apply-cont (try-cont *var handler-exp env cont*) *val*)
= (apply-cont *cont val*)

如果一个异常抛出了呢？首先，我们当然得求出 raise 参数的值。

```
(raise-exp (exp1)
  (value-of/k exp1 env
    (raise1-cont cont)))
```

当 exp1 的值返回给 raise1-cont 时，我们要查找续文中最接近的异常处理器，即最上层的 try-cont 续文。所以，我们把续文规范写成：

(apply-cont (raise1-cont *cont*) *val*)
= (apply-handler *val cont*)

其中，apply-handler 是一过程，它找出最接近的异常处理器，然后调用它()。

apply-handler : $ExpVal \times Cont \to FinalAnswer$

```
(define apply-handler
  (lambda (val cont)
    (cases continuation cont
      (try-cont (var handler-exp saved-env saved-cont)
        (value-of/k handler-exp
          (extend-env var val saved-env)
          saved-cont))
      (end-cont ()
        (report-uncaught-exception))
      (diff1-cont (exp2 saved-env saved-cont)
        (apply-handler val saved-cont))
      (diff2-cont (val1 saved-cont)
        (apply-handler val saved-cont))
      ...)))
```

图 5.15 过程 `apply-handler`

要明白怎样将这些结合到一起，我们考虑用被定语言实现的 index。令 exp_0 指代表达式:

```
let index
     = proc (n)
         letrec inner (lst)
           = if null? (lst)
             then raise 99
             else if zero?(-(car(lst), n))
                  then 0
                  else -((inner cdr(lst)), -1)
           in proc (lst)
               try (inner lst)
                catch (x) -1
in ((index 5) list(2, 3))
```

我们从任意环境 ρ_0 和续文 $cont_0$ 开始求 exp_0 的值，只展示计算的关键部分，并插入注释。

```
(value-of/k
  <<let index = ... in ((index 5) list(2, 3))>>
  ρ0
  cont0)
= 执行 let 主体
(value-of/k
  <<((index 5) list(2, 3))>>
  ((index                称之为ρ1
     #(struct:proc-val
        #(struct:procedure n <<letrec ...>> ρ0)))
   (i #(struct:num-val 1))
   (v #(struct:num-val 5))
   (x #(struct:num-val 10)))
  #(struct:end-cont))
= 最后求 try 的值
(value-of/k
  <<try (inner lst) catch (x) -1>>
  ((lst                  称之为ρlst=(2 3)
     #(struct:list-val
        (#(struct:num-val 2) #(struct:num-val 3))))
   (inner ...)
   (n #(struct:num-val 5))
```

```
   ρ0)
  #(struct:end-cont))
= 在 try-cont 续文中求 try 主体的值
(value-of/k
  <<(inner lst)>>
  ρlst=(2 3)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= lst 绑定到(2 3)，求 inner 主体的值
(value-of/k
  <<if null?(lst) ...>>
  ρlst=(2 3)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= 求条件的值，进入递归所在的行
(value-of/k
  <<-((inner cdr(lst)), -1)>>
  ρlst=(2 3)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= 求 diff-exp 第一个参数的值
(value-of/k
  <<(inner cdr(lst))>>
  ρlst=(2 3)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
= lst 绑定到(3)，求 inner 主体的值
(value-of/k
  <<if null?(lst) ...>>
  ((lst #(struct:list-val (#(struct:num-val 3)))) 称之为ρlst=(3)
   (inner ...)
   ρ0)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
= 求条件的值，进入递归所在的行
(value-of/k
  <<-((inner cdr(lst)), -1)>>
  ρlst=(3)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
```

```
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
= 求 diff-exp 第一个参数的值
(value-of/k
  <<(inner cdr(lst))>>
  ρlst=(3)
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:diff1-cont <<-1>> ρlst=(2 3)
        #(struct:try-cont x <<-1>> ρlst=(2 3)
           #(struct:end-cont)))))
= lst 绑定到(), 求 inner 主体的值
(value-of/k
  <<if null?(lst) ... >>
  ((lst #(struct:list-val ()))      称之为ρlst=()
   (inner ...)
   (n #(struct:num-val 5))
   ...)
  #(struct:diff1-cont <<-1>> ρlst=(3)
     #(struct:diff1-cont <<-1>> ρlst=(2 3)
        #(struct:try-cont x <<-1>> ρlst=(2 3)
           #(struct:end-cont)))))
= 求 raise 表达式参数的值
(value-of/k
  <<99>>
  ρlst=()
  #(struct:raise1-cont
     #(struct:diff1-cont <<-1>> ρlst=(3)
        #(struct:diff1-cont <<-1>> ρlst=(2 3)
           #(struct:try-cont x <<-1>> ρlst=(2 3)
              #(struct:end-cont))))))

= 用 apply-handler 展开续文, 直到找出一个异常处理器
(apply-handler
  #(struct:num-val 99)
     #(struct:diff1-cont <<-1>> ρlst=(3)
        #(struct:diff1-cont <<-1>> ρlst=(2 3)
           #(struct:try-cont x <<-1>> ρlst=(2 3)
              #(struct:end-cont)))))
=
(apply-handler
  #(struct:num-val 99)
```

```
  #(struct:diff1-cont <<-1>> ρlst=(2 3)
     #(struct:try-cont x <<-1>> ρlst=(2 3)
        #(struct:end-cont))))
=
(apply-handler
  #(struct:num-val 99)
  #(struct:try-cont x <<-1>> ρlst=(2 3)
     #(struct:end-cont)))
= 找到异常处理器；把异常值绑定到 x
(value-of/k
  #(struct:const-exp -1)
  ((x #(struct:num-val 99))
  ρlst=(2 3) ...)
  #(struct:end-cont))
=
(apply-cont #(struct:end-cont) #(struct:const-exp -1))
=
#(struct:const-exp -1)
```

如果列表包含期望值，那么我们不需调用 `apply-handler`，而是调用 `apply-cont`，并执行续文中所有待完成的 `diff`。

练习 5.35 [⋆⋆] 这种实现很低效，因为异常抛出时，`apply-handler` 必须在续文中线性查找处理器。让所有续文都能直接使用 `try-cont` 续文，从而避免这种查找。

练习 5.36 [⋆] 另一种避免 `apply-handler` 线性查找的设计是使用两个续文，一个正常续文，一个异常续文。修改本节的解释器，改用两个续文，实现这一目标。

练习 5.37 [⋆] 修改被定语言，在过程调用的实参数目错误时抛出异常。

练习 5.38 [⋆] 修改被定语言，添加除法表达式，并在被零除时抛出异常。

练习 5.39 [⋆⋆] 目前，异常处理器可以重新抛出异常，把它传播出去；或者返回一个值，作为 `try` 表达式的值。还可以这样设计语言：允许计算从异常抛出的位置继续。修改本节的解释器，在 `raise` 调用处的续文中运行异常处理器的主体，完成这种设计。

练习 5.40 [⋆⋆⋆] 把 `raise` 异常处的续文作为第二个参数传递，使被定语言中的异常处理器既能返回也能继续。这可能需要把续文作为一种新的表达值。为用值调用续文设计

恰当的语法。

练习 5.41 [⋆⋆⋆] 我们展示了如何用数据结构表示的续文实现异常。我们没办法马上用 2.2.3 节中的步骤得到过程表示法，因为我们现在有两个观测器：`apply-handler` 和 `apply-cont`。用一对过程实现本节的续文：一个单参数过程，表示 `apply-cont` 中续文的动作；一个无参数过程，表示 `apply-handler` 中续文的动作。

练习 5.42 [⋆⋆] 前一道练习只在抛出异常时捕获续文。添加形式 `letcc` *Identifier* `in` *Expression*，允许在语言中的任意位置捕获续文，其规范为：

```
(value-of/k (letcc var body) ρ cont)
= (value-of/k body (extend-env var cont ρ) cont)
```

这样捕获的续文可用 `throw` 调用：表达式 `throw` *Expression* `to` *Expression* 求出两个子表达式的值。第二个表达式应返回一续文，应用于第一个表达式的值。`throw` 表达式当前的续文则被忽略。

练习 5.43 [⋆⋆] 修改前一道练习被定语言中的 `letcc`，把捕获的续文作为一种新的过程，这样就能写 `(`exp_1 exp_2`)`，而不必写 `throw` *Expression* `to` *Expression*。

练习 5.44 [⋆⋆] 前面练习里的 `letcc` 和 `throw` 还有一种设计方式，只需给语言添加一个过程。这个过程在 Scheme 中叫做 `call-with-current-continuation`，它取一个单参数过程 `p`，并给 `p` 传递一个单参数过程，这个过程在调用时，将其参数传递给当前的续文 `cont`。`call-with-current-continuation` 可用 `letcc` 和 `throw` 定义如下：

```
let call-with-current-continuation
      = proc (p)
          letcc cont
          in (p proc (v) throw v to cont)
in ...
```

给语言添加 `call-with-current-continuation`。然后写一个翻译器，用只有 `call-with-current-continuation` 的语言翻译具有 `letcc` 和 `throw` 的语言。

5.5 线程

许多编程任务中，可能需要一次进行多项计算。当这些计算作为同一进程的一部分，运行在同一地址空间，通常称它们为*线程* (*thread*)。本节，我们将看到如何修改解释器，模拟多线程程序的执行。

我们的多线程解释器不做单线程计算，而且维护多个线程。就像本章之前展示的那样，每个线程包含一项正在进行的计算。线程使用第 4 章中的赋值，通过共享内存通信。

在我们的系统中，整个计算包含一个*线程池* (*pool*)。每个线程*在运行* (*running*)、*可运行* (*runnable*) 或者*受阻塞* (*blocked*)。在我们的系统中，一次只能有一个线程在运行。在多 CPU 系统中，可以有若干线程同时运行。可运行的线程记录在名为*就绪队列* (*ready queue*) 的队列中。还有些线程因为种种原因未能就绪，我们说这些线程*受阻塞*。本节稍后介绍受阻塞线程。

线程调度由*调度器* (*scheduler*) 执行，就绪队列为其状态的一部分。此外，它保存一个计时器，当一个线程执行若干步骤（即线程的*时间片* (*time slice*) 或*量子* (*quantum*)）时，它中断线程，将其放回就绪队列，并从就绪队列中选出一个新的线程来运行。这叫做*抢占式调度* (*pre-emptive scheduling*)。

我们的新语言基于 IMPLICIT-REFS，名叫 THREADS。在 THREADS 中，新线程由名为 `spawn` 的结构创建。`spawn` 取一参数，该参数的值应为一个过程。新创建的线程运行时，给那个过程传递一个任意参数。该线程不是立刻运行，而是放入就绪队列中，轮到它时才运行。`spawn` 的执行只求效果；在我们的系统中，我们为它任选一个返回值 73。

我们来看这种语言的两个示例程序。定义了一个过程 `noisy`，它取一个列表，打印出列表的第一个元素，然后递归处理列表的剩余部分。这里，主体中的表达式创建两个线程，分别打印列表 `[1,2,3,4,5]` 和 `[6,7,8,9,10]`。两个列表究竟如何穿插取决于调度器；在本例中，在被调度器打断之前，每个线程打印出列表中的两个元素，

```
test: two-non-cooperating-threads

letrec
  noisy (l) = if null?(l)
              then 0
              else begin print(car(l)); (noisy cdr(l)) end
in
   begin
    spawn(proc (d) (noisy [1,2,3,4,5]));
    spawn(proc (d) (noisy [6,7,8,9,10]));
    print(100);
    33
   end

100
1
2
6
7
3
4
8
9
5
10
正确结果: 33
实际结果: #(struct:num-val 33)
正确
```

图 5.16 两个交错运行的线程

展示了一个生产者和一个消费者，由初始值为 0 的缓存相联系。生产者取一参数 `n`，进入 `wait`，循环 5 次，然后把 `n` 放入缓存。每次进入 `wait` 循环，它打印一个倒数计时器（以 200s 为单位）的值。消费者取一参数（但忽略它），进入一循环，等待缓存变成非零值。每次进入循环时，它打印一个计数器（以 100s 为单位）的值，以展示它等结果等了多久。主线程将生产者放入就绪队列，打印出 300，并在自身中启动消费者。所以，前两项，300 和 205，分别由主线程和生产者线程打印。[1]就像前一个例子那样，在被打断之前，消费者线程和生产者线程各自循环大约两次。

[1]原文为 So the first two items, 300 and 205, are printed by the main thread. 实则 205 是生产者所在线程打印。

```
let buffer = 0
in let producer = proc (n)
        letrec
         wait(k) = if zero?(k)
                   then set buffer = n
                   else begin
                         print(-(k,-200));
                         (wait -(k,1))
                        end
         in (wait 5)
   in let consumer = proc (d)
           letrec busywait (k) = if zero?(buffer)
                                 then begin
                                       print(-(k,-100));
                                       (busywait -(k,-1))
                                      end
                                 else buffer
            in (busywait 0)
      in begin
          spawn(proc (d) (producer 44));
          print(300);
          (consumer 86)
         end

300
205
100
101
204
203
102
103
202
201
104
105
正确结果: 44
实际结果: #(struct:num-val 44)
正确
```

图 5.17 由缓存连接的生产者和消费者

实现从 IMPLICIT-REFS 语言传递续文的解释器开始。这与 5.1 节中的类似，只是多了 IMPLICIT-REFS 中的存储器（当然!），以及 ex5.9 中的续文构造器 `set-rhs-cont`。

我们给这个解释器添加一个调度器。调度器状态由四个值组成，接口提供六个过程来操作这些值，如所示。

展示了本接口的实现。这里 `(enqueue` q val`)` 返回一队列，除了把 val 放在末尾之外，与 q 相同。`(dequeue` q f`)` 取出队头及剩余部分，将它们作为参数传递给 f。

我们用无参数且返回表达值的 Scheme 过程表示线程：

$$Thread = () \to ExpVal$$

如果就绪队列非空，那么过程 `run-next-thread` 从就绪队列取出第一个线程并运行，赋予它一个大小为 `the-max-time-slice` 的新时间片。如果还有就绪线程，它还把 `the-ready-queue` 设置为剩余线程队列。如果就绪队列为空，`run-next-thread` 返回 `the-final-answer`，计算至此全部终止。

然后我们来看解释器。`spawn` 表达式在某个续文中求参数的值，这个续文执行时，将一个新线程放入就绪队列，并将 73 返回给 `spawn` 的调用者。新的线程执行时，将一个任意值（这里选 28）传给 `spawn` 参数求值得到的过程。要完成这些，我们给 `value-of/k` 新增从句：

```
(spawn-exp (exp)
  (value-of/k exp env
    (spawn-cont cont)))
```

给 `apply-cont` 新增从句：

```
(spawn-cont (saved-cont)
  (let ((proc1 (expval->proc val)))
    (place-on-ready-queue!
      (lambda ()
        (apply-procedure/k proc1
          (num-val 28)
          (end-subthread-cont))))
    (apply-cont saved-cont (num-val 73))))
```

调度器的内部状态

`the-ready-queue`	就绪队列
`the-final-answer`	主线程结束时的值
`the-max-time-slice`	每个线程运行的步数
`the-time-remaining`	当前运行线程剩余的步数

调度器的接口

initialize-scheduler!	: *Int* → *Unspecified* 初始化调度器状态
place-on-ready-queue!	: *Thread* → *Unspecified* 把线程放入就绪队列
run-next-thread	: () → *FinalAnswer* 运行下一个线程。如果没有就绪线程，返回最终答案。
set-final-answer!	: *ExpVal* → *Unspecified* 设置最终答案
time-expired?	: () → *Bool* 判断计时器是否为 0
decrement-timer!	: () → *Unspecified* 递减 `the-time-remaining`

图 5.18 调度器的状态和接口

initialize-scheduler! : *Int* → *Unspecified*
```
(define initialize-scheduler!
  (lambda (ticks)
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))
```

place-on-ready-queue! : *Thread* → *Unspecified*
```
(define place-on-ready-queue!
  (lambda (th)
    (set! the-ready-queue
      (enqueue the-ready-queue th))))
```

run-next-thread : () → *FinalAnswer*
```
(define run-next-thread
  (lambda ()
    (if (empty? the-ready-queue)
        (begin
          (when (debug-mode?)
            (eopl:printf "计算结束.~%"))
          the-final-answer)
        (begin
          (when (debug-mode?)
            (eopl:printf "切换到另一线程.~%"))
          (dequeue the-ready-queue
            (lambda (first-ready-thread other-ready-thread)
              (set! the-ready-queue other-ready-thread)
              (set! the-time-remaining the-max-time-slice)
              (first-ready-thread)))))))
```

set-final-answer! : *ExpVal* → *Unspecified*
```
(define set-final-answer!
  (lambda (val)
    (set! the-final-answer val)))
```

time-expired? : *ExpVal* → *Bool*
```
(define time-expired?
  (lambda ()
    (zero? the-time-remaining)))
```

decrease-timer! : () → *Unspecified*
```
(define decrease-timer!
  (lambda ()
    (set! the-time-remaining (- the-time-remaining 1))))
```

图 5.19 调度器

```
let x = 0
in let mut = mutex()
   in let incr_x = proc (id)
                    proc (dummy)
                     set x = -(x,-1)
      in begin
          spawn((incr_x 100));
          spawn((incr_x 200));
          spawn((incr_x 300))
         end
```

图 5.20 不安全的计数器

跳跃式解释器生成快照时也要做这样：它将计算打包（这里的 `(lambda () (apply-procedure/k ...))`），然后把它传给另一个过程处理。在跳床的例子中，我们把线程传给跳床，后者直接执行前者。这里，我们把新线程放入就绪队列，继续我们的现有计算。

这带来一个关键问题：每个线程应在什么续文中运行？

- 运行主线程的续文应记录主线程的值，作为最终答案，然后运行剩余线程。
- 子线程结束时无法报告它的值，所以运行子线程的续文应忽略其值，然后直接运行剩余线程。

由此我们得出两种新续文，其行为由 `apply-cont` 中的以下几行实现：

```
(end-main-thread-cont ()
  (set-final-answer! val)
  (run-next-thread))

(end-subthread-cont ()
  (run-next-thread))
```

我们从 `value-of-program` 入手整个系统：

value-of-program : $Int \times Program \to FinalAnswer$

```
(define value-of-program
  (lambda (timeslice pgm)
    (initialize-store!)
    (initialize-scheduler! timeslice)
    (cases program pgm
      (a-program (exp1)
        (value-of/k
          exp1
          (init-env)
          (end-main-thread-cont))))))
```

最后，我们修改 `apply-cont`，让它在每次调用时递减计时器。如果计时器到期，那就中止当前计算。在实现时，我们先把一个线程放入就绪队列，它用调用 `run-next-thread` 时恢复的计时器再次调用 `apply-cont`。

apply-cont : $Cont \times ExpVal \to FinalAnswer$

```
(define apply-cont
  (lambda (cont val)
    (if (time-expired?)
      (begin
        (place-on-ready-queue!
          (lambda () (apply-cont cont val)))
        (run-next-thread))
      (begin
        (decrement-timer!)
        (cases continuation cont
          ...)))))
```

共享变量不是可靠的通信方式，因为多个线程可能试图写同一变量。例如，考虑中的程序。这里，我们创建了三个线程，试图累加同一个计数器 `x`。如果一个线程读取了计数器，但在更新计数器之前被打断，那么两个线程将把计数器设置成同样的值。因此，计数器可能变成 2，甚至是 1，而不是 3。

我们想要确保不会发生这种混乱。同样地，我们想要组织我们的程序，避免中的程序空转。恰恰相反，它应该能够进入休眠状态，并在生产者向共享缓存插入值时唤醒。

有许多方式设计这类同步组件。一种简单的方式是使用*互斥锁* (*mutex exclusion, mutex*) 或*二元信号量* (*binary semaphore*)。

互斥锁可能*打开* (*open*) 或*关闭* (*closed*)。它还包含一个等待互斥锁打开的线程队列。互斥锁有三种操作：

- `mutex`，无参数操作，创建一个初始状态为打开的互斥锁。
- `wait`，单参数操作，线程用它表明想要访问某一互斥锁。它的参数必须是一把互斥锁。它的行为取决于互斥锁的状态。
  - 若互斥锁关闭，当前线程放入互斥锁的等待队列中，并中止。我们说当前线程*受阻塞*，等待这把互斥锁。
  - 若互斥锁打开，则将其关闭，并让当前线程继续执行。

`wait` 的执行只求效果；它的返回值未定义。

- `signal`，单参数操作，线程用它表明准备释放某一互斥锁。它的参数必须是一把互斥锁。

  - 若互斥锁关闭，且等待队列中没有线程，则将其打开，并让当前线程继续执行。
  - 若互斥锁关闭，且等待队列中仍有线程，则从中选出一个线程，放入调度器的等待队列，互斥锁则保持关闭。执行 `signal` 的线程继续计算。
  - 若互斥锁打开，则保持打开，线程继续执行。

  `signal` 的执行只求效果；它的返回值未定义。`signal` 操作总是成功：执行它的线程仍然是运行线程。

这些属性保证在一对连续的 `wait` 和 `signal` 之间，只有一个线程可以执行。这部分程序叫做关键区域 (*critical region*)。在关键区域内，两个线程不可能同时执行。例如，展示了我们之前的例子，只是在关键行周围插入了一把互斥锁。在这个程序中，一次只有一个线程可以执行 `set x = -(x,-1)`；所以计数器一定能够到达终值 3。

```
let x = 0
in let mut = mutex()
   in let incr_x = proc (id)
                     proc (dummy)
                       begin
                         wait(mut);
                         set x = -(x,-1);
                         signal(mut)
                       end
      in begin
          spawn((incr_x 100));
          spawn((incr_x 200));
          spawn((incr_x 300))
         end
```

图 5.21 使用互斥锁的安全计数器

我们用两个引用模拟互斥锁：一个指向其状态（开启或关闭），一个指向等待这把锁的线程列表。我们还把互斥锁作为一种表达值。

```
(define-datatype mutex mutex?
  (a-mutex
    (ref-to-closed? reference?)
    (ref-to-wait-queue reference?)))
```

我们给 `value-of/k` 添加适当的行：

```
    (mutex-exp ()
      (apply-cont cont (mutex-val (new-mutex))))
```

其中：

new-mutex : () → *Mutex*

```
(define new-mutex
  (lambda ()
    (a-mutex
     (newref #f)
     (newref '()))))
```

`wait` 和 `signal` 作为新的单参数操作，只是调用过程 `wait-for-mutex` 和 `signal-mutex`。`wait` 和 `signal` 都求出它们唯一参数的值，所以，在 `apply-cont` 中我们写：

```
    (wait-cont
      (saved-cont)
      (wait-for-mutex
        (expval->mutex val)
        (lambda () (apply-cont saved-cont (num-val 52)))))

    (signal-cont
      (saved-cont)
      (signal-mutex
        (expval->mutex val)
        (lambda () (apply-cont saved-cont (num-val 53)))))
```

现在，我们可以写出 `wait-for-mutex` 和 `signal-mutex`。这些过程取两个参数：一个互斥锁，一个线程，其工作方式如上所述 ()。

wait-for-mutex : *Mutex* × *Thread* → *FinalAnswer*
用法: 等待互斥锁开启，然后关闭它

```
(define wait-for-mutex
  (lambda (m th)
    (cases mutex m
      (a-mutex (ref-to-closed? ref-to-wait-queue)
        (cond
          ((deref ref-to-closed?)
            (setref! ref-to-wait-queue
              (enqueue (deref ref-to-wait-queue) th))
            (run-next-thread))
          (else
            (setref! ref-to-closed? #t)
            (th)))))))
```

signal-mutex : *Mutex* × *Thread* → *FinalAnswer*

```
(define signal-mutex
  (lambda (m th)
    (cases mutex m
      (a-mutex (ref-to-closed? ref-to-wait-queue)
        (let ((closed? (deref ref-to-closed?))
              (wait-queue (deref ref-to-wait-queue)))
          (if closed?
            (if (empty? wait-queue)
              (begin
                (setref! ref-to-closed? #f)
                (th))
              (begin
                (dequeue
                  wait-queue
                  (lambda (first-waiting-th other-waiting-ths)
                    (place-on-ready-queue!
                      first-waiting-th)
                    (setref!
                      ref-to-wait-queue
                      other-waiting-ths)))
                (th)))
            (th)))))))
```

图 5.22 `wait-for-mutex` 和 `signal-mutex`

练习 5.45 [⋆] 给本节的语言添加形式 `yield`。线程不论何时执行 `yield`，都将自身放入就绪队列之中，就绪队列头部的线程接着执行。当线程继续时，就好像调用 `yield` 返回了 99。

练习 5.46 [⋆⋆] 在 ex5.45 的系统中，线程放入就绪队列，既可能是因为耗尽时间片，也可能是因为它选择让步。在后一种情况下，线程会以一个完整的时间片重启。修改系统，让就绪队列记录每个线程的剩余时间片（如果有的话），在线程重启时仍使用剩余的时间片。

练习 5.47 [⋆] 如果剩余两个子线程，二者都在等待另一个子线程持有的互斥锁会怎样?

练习 5.48 [⋆] 我们用过程表示线程。将其改为数据结构表示法。

练习 5.49 [⋆] 为 THREADS 完成 ex5.15（用堆栈上的帧表示续文）。

练习 5.50 [⋆⋆] 寄存本节的解释器。必须寄存的互递归尾调用过程有哪些?

练习 5.51 [⋆⋆⋆] 我们想要组织我们的程序，避免中的程序空转。恰恰相反，它应该能够进入休眠状态，并在生产者向共享缓存插入值时唤醒。用具有互斥锁的程序完成这些，或者实现一种同步操作符完成这些。

练习 5.52 [⋆⋆⋆] 写出使用互斥锁的程序，如，但主线程等待所有三个子线程结束，然后返回 `x` 的值。

练习 5.53 [⋆⋆⋆] 修改线程的表示，添加*线程描述符* (*thread identifier*)。每个新线程都有一个新的线程描述符。当子线程创建时，传给它的不是本节中的任意值 28，而是它的线程描述符。子线程的描述符也作为 `spawn` 表达式的值返回给父线程。给解释器添加辅助组件，跟踪线程描述符的创建。验证就绪队列中一个线程描述符至多出现一次。子线程如何获知父线程的描述符？原程序的线程描述符应如何处理?

练习 5.54 [⋆⋆] 给 ex5.53 的解释器添加组件 `kill`。`kill` 结构取一线程号，在就绪队列或所有等待队列中找出对应的线程，然后删除它。此外，当它找到目标线程时，返回

真；在所有队列中都找不到指定线程号时，返回假。

练习 5.55 [⋆⋆] 给 ex5.53 的解释器添加线程通信组件，一个线程可以用另一线程的描述符给它发送一个值。线程可以选择接收消息，没有线程给它发消息时可以阻塞。

练习 5.56 [⋆⋆] 修改 ex5.55 的解释器，不要使用共享存储器，而是让每个线程具有自己的存储器。在这种语言中，几乎可以排除互斥锁。重写本节语言的示例程序，但不用互斥锁。

练习 5.57 [⋆⋆⋆] 在你最爱的操作系统教材中，有各种各样的同步机制。挑出三种，在本节的框架下实现它们。

练习 5.58 [绝对⋆] 和朋友吃些披萨吧，但是一人一次一定只拿一块！

6 续文传递风格

在第 5 章，我们把解释器中的所有主要过程调用重写成*尾调用*。这样，我们保证任何时候，不论执行的程序多大或多复杂，解释器只使用有限的控制上下文。这种性质叫做*迭代性控制行为*。

我们通过给每个过程多传一个*续文*参数实现这一目标。这种编程风格叫做*续文传递风格* (*continuation-passing style*) 或 *CPS*，且不限于解释器。

本章，我们介绍一种系统性的方法，将任一过程转换为具有迭代性控制行为的等效过程。实现这一点需要将过程转换为续文传递风格。

6.1 写出续文传递风格的程序

除了写解释器，CPS 还有别的作用。我们考虑 •老熟人• 阶乘程序：

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

阶乘的传递续文版本是：

```
(define fact
  (lambda (n)
    (fact/k n (end-cont))))

(define fact/k
  (lambda (n cont)
    (if (zero? n)
```

```
        (apply-cont cont 1)
        (fact/k (- n 1) (fact1-cont n cont)))))
```

其中，

(apply-cont (end-cont) *val*) = *val*

(apply-cont (fact1-cont *n cont*) *val*) = *val*
= (apply-cont *cont* (* *n val*))

在这一版内，`fact/k` 和 `apply-cont` 中的所有调用都在末尾，因此不产生控制上下文。

我们可以用数据结构实现这些续文：

```
(define-datatype continuation continuation?
  (end-cont)
  (fact1-cont
   (n integer?)
   (cont continuation?)))

(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (end-cont () val)
      (fact1-cont (saved-n saved-cont)
        (apply-cont saved-cont (* saved-n val))))))
```

我们还能以多种方式转换这一程序，比如寄存它，如所示。

我们甚至能将其转为跳跃式，如所示。如果用普通的指令式语言，我们自然能将跳床替换为适当的循环。

但是，本章我们主要关心，用过程表示法（如）时会发生什么。回忆一下，在过程表示法中，续文用它在 `apply-cont` 中的动作表示。过程表示为：

```
(define end-cont
  (lambda ()
    (lambda (val) val)))

(define fact1-cont
```

```
      (lambda (n saved-cont)
        (lambda (val)
          (apply-cont saved-cont (* n val)))))

    (define apply-cont
      (lambda (cont val)
        (cont val)))
```

```
(define n 'uninitialized)
(define cont 'uninitialized)
(define val 'uninitialized)

(define fact
  (lambda (arg-n)
    (set! cont (end-cont))
    (set! n arg-n)
    (fact/k)))

(define fact/k
  (lambda ()
    (if (zero? n)
      (begin
        (set! val 1)
        (apply-cont))
      (begin
        (set! cont (fact1-cont n cont))
        (set! n (- n 1))
        (fact/k)))))

(define apply-cont
  (lambda ()
    (cases continuation cont
      (end-cont () val)
      (fact1-cont (saved-n saved-cont)
        (set! cont saved-cont)
        (set! val (* saved-n val))
        (apply-cont)))))
```

图 6.1 寄存后的 `fact/k`

我们还可以更进一步，将程序中所有调用续文构造器的地方替换为其定义。因为定义在行内展开，这一转换叫做内联 (*inlining*)。我们还要内联 `apply-cont` 的调用，不再写 `(apply-cont cont val)`，而是直接写 `(cont val)`。

```
(define n 'uninitialized)
(define cont 'uninitialized)
(define val 'uninitialized)
(define pc 'uninitialized)

(define fact
  (lambda (arg-n)
    (set! cont (end-cont))
    (set! n arg-n)
    (set! pc fact/k)
    (trampoline!)
    val))

(define trampoline!
  (lambda ()
    (if (procedure? pc)
      (begin
        (pc)
        (trampoline!)))))


(define fact/k
  (lambda ()
    (if (zero? n)
      (begin
        (set! val 1)
        (set! pc apply-cont))
      (begin
        (set! cont (fact1-cont n cont))
        (set! n (- n 1))
        (set! pc fact/k)))))

(define apply-cont
  (lambda ()
    (cases continuation cont
      (end-cont
        ()
        (set! pc #f))
      (fact1-cont
        (n saved-cont)
        (set! cont saved-cont)
        (set! val (* n val))
        (set! pc apply-cont)))))
```

图 6.2 寄存后的跳跃式 `fact/k`

如果我们按这种方式内联所有用到续文的地方，我们得到：

```
(define fact
  (lambda (n)
    (fact/k n (lambda (val) val))))

(define fact/k
  (lambda (n cont)
    (if (zero? n)
      (cont 1)
      (fact/k (- n 1) (lambda (val) (cont (* n val)))))))
```

`fact/k` 的定义可以读作：

> 若 n 为 *0*，将 *1* 传给续文；否则，求 $n-1$ 的 `fact`，求值所在的续文，取其结果 `val`，然后将 `(* n val)` 的值传给当前续文。

过程 `fact/k` 具有性质 `(fact/k n g) = (g n!)`。对 n 使用归纳法很容易证明这条性质。第一步，当 $n=0$，我们计算：

```
(fact/k 0 g) = (g 1) = (g (fact 0))
```

归纳步骤中，对某个 n，设 `(fact/k n g) = (g n!)`，试证明 `(fact/k` $(n+1)$ `g) = (g` $(n+1)$`!)`。要证明它，我们计算：

```
(fact/k n+1 g)
= (fact/k n (lambda (val) (g (* n+1 val))))
= ((lambda (val) (g (* n+1 val)))     （根据归纳假设）
   (fact n))
= (g (* n+1 (fact n)))
= (g (fact n+1))
```

归纳完毕。

像 1.3 节一样，这里的 g 是上下文参数；性质 `(fact/k n g) = (g n!)` 作为独立规范，遵循我们的原则**避免神秘小工具**。

现在，我们用同样的方式转换计算斐波那契数列的 `fib`。我们从下面的过程开始：

```
(define fib
  (lambda (n)
    (if (< n 2)
      1
      (+
        (fib (- n 1))
        (fib (- n 2))))))
```

这里我们两次递归调用 `fib`，所以我们需要一个 `end-cont` 和两个续文构造器，二者各对应一个参数，就像处理 5.1 节中的差值表达式那样。

```
(define fib
  (lambda (n)
    (fib/k n (end-cont))))

(define fib/k
  (lambda (n cont)
    (if (< n 2)
      (apply-cont cont 1)
      (fib/k (- n 1) (fib1-cont n cont)))))

(apply-cont (end-cont) val) = val

(apply-cont (fib1-cont n cont) val1)
= (fib/k (- n 2) (fib2-cont val1 cont))

(apply-cont (fib2-cont val1 cont) val2)
= (apply-cont cont (+ val1 val2))
```

在过程表示法中，我们有：

```
(define end-cont
  (lambda ()
    (lambda (val) val)))

(define fib1-cont
  (lambda (n cont)
    (lambda (val1)
      (fib/k (- n 2) (fib2-cont val1 cont)))))
```

```
(define fib2-cont
  (lambda (val1 cont)
    (lambda (val2)
      (apply-cont cont (+ val1 val2)))))

(define apply-cont
  (lambda (cont val)
    (cont val)))
```

如果我们内联所有使用这些过程的地方，可得：

```
(define fib
  (lambda (n)
    (fib/k n (lambda (val) val))))

(define fib/k
  (lambda (n cont)
    (if (< n 2)
      (cont 1)
      (fib/k (- n 1)
        (lambda (val1)
          (fib/k (- n 2)
            (lambda (val2)
              (cont (+ val1 val2)))))))))
```

类似阶乘，我们可以把这个定义读作：

> 若 $n < 2$，将 *1* 传给续文。否则，处理 $n - 1$，求值所在的续文，取其结果 `val1`，然后处理 `val2`，求值所在的续文，取其结果 `val2`，然后将 `(+ val1 val2)` 的值传给当前续文。

用推导 `fact` 的方式，很容易得出，对任意 g，`(fib/k` n g`)` = `(`g `(fib` n`))`。这里有个假想的例子，推广了这一想法：

```
(lambda (x)
  (cond
   ((zero? x) 17)
   ((= x 1) (f x))
```

```
((= x 2) (+ 22 (f x)))
((= x 3) (g 22 (f x)))
((= x 4) (+ (f x) 33 (g y)))
(else (h (f x) (- 44 y) (g y)))))
```

变成：

```
(lambda (x cont)
  (cond
   ((zero? x) (cont 17))
   ((= x 1) (f x cont))
   ((= x 2) (f x (lambda (v1) (cont (+ 22 v1)))))
   ((= x 3) (f x (lambda (v1) (g 22 v1 cont))))
   ((= x 4) (f x (lambda (v1)
                   (g y (lambda (v2)
                          (cont (+ v1 33 v2)))))))
   (else (f x (lambda (v1)
                (g y (lambda (v2)
                       (h v1 (- 44 y) v2 cont)))))))))
```

其中，过程 `f`、`g` 和 `h` 都以类似方式转换。

- 在 `(zero? x)` 这一行，我们将 17 返回给续文。
- 在 `(= x 1)` 这一行，我们按尾递归的方式调用 `f`。
- 在 `(= x 2)` 这一行，我们在加法的操作数位置调用 `f`。
- 在 `(= x 3)` 这一行，我们在过程调用的操作数位置调用 `f`。
- 在 `(= x 4)` 这一行，有两个过程调用在加法的操作数位置。
- 在 `else` 这一行，有两个过程调用在另一个过程调用内的操作数位置。

从这些例子中浮现出一种模式。

CPS 秘方

要将程序转换为续文传递风格：

1. 给每个过程传一个额外参数（通常是 `cont` 或 `k`）。
2. 不论过程返回常量还是变量，都将返回值传给续文，就像上面的 `(cont 17)`。
3. 过程调用在尾部时，用同样的续文 `cont` 调用过程。
4. 过程调用在操作数位置时，在新的续文中求过程调用的值，这个续文给调用结果命名，继续进行计算。

这些规则虽不正式，但足以解释这种模式。

练习 6.1 [⋆] 考虑,为什么移除 `fact/k` 定义中的 `(set! pc fact/k)` 和 `apply-cont` 中定义的 `(set! pc apply-cont)`，程序仍能正常运行?

练习 6.2 [⋆] 用归纳法证明：对任意 g，`(fib/k` n g`)` = `(`g `(fib` n`))`，归纳变量为 n。

练习 6.3 [⋆] 把下面每个 Scheme 表达式重写为续文传递风格。假设所有未知函数都已经重写成 CPS 风格。

1. `(lambda (x y) (p (+ 8 x) (q y)))`
2. `(lambda (x y u v) (+ 1 (f (g x y) (+ u v))))`
3. `(+ 1 (f (g x y) (+ u (h v))))`
4. `(zero? (if a (p x) (p y)))`
5. `(zero? (if (f a) (p x) (p y)))`
6. `(let ((x (let ((y 8)) (p y)))) x)`
7. `(let ((x (if a (p x) (p y)))) x)`

练习 6.4 [⋆⋆] 把下面的所有过程重写为续文传递风格。表示每个过程的续文时，先用数据结构表示法，然后用过程表示法，然后用内联过程表示法。最后，写出寄存版本。照第 5 章那样定义 `end-cont`，验证你实现的这四个版本是尾调用：

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "计算结束.~%")
    (eopl:printf "这句话只能出现一次.~%")
    val)
```

1. `remove-first`（1.2.3 节）
2. `list-sum`（1.3 节）
3. `occurs-free?`（1.2.4 节）
4. `subst`（1.2.5 节）

练习 6.5 [⋆] 当我们把表达式重写为 CPS 时，我们就为表达式中的过程选择了一种求值顺序。把前面的每个例子重写为 CPS，且使过程调用从右向左求值。

练习 6.6 [⋆] 在 `(lambda (x y) (+ (f (g x)) (h (j y))))` 中，过程调用有多少种不同的求值顺序？对每种求值顺序，写出对应的 CPS 表达式。

练习 6.7 [⋆⋆] 写出、fig-5.5 和 fig-5.6 中解释器的过程表示和内联过程表示。

练习 6.8 [⋆⋆⋆] 写出 5.4 节解释器的过程表示和内联过程表示。这极富挑战性，因为我们实际上有两个观测器，`apply-cont` 和 `apply-handler`。提示：考虑修改 cps-recipe 的秘方，给每个过程添加两个参数，一个表示 `apply-cont` 中续文的行为，一个表示 `apply-handler` 中续文的行为。

有时，我们能发现更巧妙的方式表示续文。我们重新考虑用过程表示续文的 `fact`。其中，我们有两个续文构造器，写作：

```
(define end-cont
  (lambda ()
    (lambda (val) val)))

(define fact1-cont
  (lambda (n cont)
    (lambda (val) (cont (* n val)))))
```

```
(define apply-cont
  (lambda (cont val)
    (cont val)))
```

在这个系统中，所有续文都用某个数乘以参数。`end-cont` 将其参数乘 1，若 *cont* 将参数乘 k，那么 `(fact1` n *cont*`)` 将其值乘 $k * n$。

所以每个续文都形如 `(lambda (val) (*` k `val))`。这意味着我们可以用每个续文仅有的自由变量••数字 k••表示它。用这种表示方式，我们有：

```
(define end-cont
  (lambda ()
    1))

(define fact1-cont
  (lambda (n cont)
    (* cont n)))

(define apply-cont
  (lambda (cont val)
    (* cont val)))
```

如果我们在 `fact/k` 的原始定义中内联这些过程，并使用性质 `(*` *cont* `1) =` *cont*，可得：

```
(define fact
  (lambda (n)
    (fact/k n 1)))

(define fact/k
  (lambda (n cont)
    (if (zero? n)
      cont
      (fact/k (- n 1) (* cont n)))))
```

但是这和 `fact-iter`（fact-iter）完全相同！所以我们明白了，累加器通常只是续文的一种表示方式。这令人印象深刻。相当一部分经典的程序优化问题原来是这一思想的特例。

练习 6.9 [⋆] 乘法的什么性质使这种程序优化成为可能?

练习 6.10 [⋆] 给 `list-sum` 设计一种简便的续文表示方式，就像上面的 `fact/k` 那样。

6.2 尾式

要写出程序来做续文传递风格变换，我们需要找出输入和输出语言。我们选择 LETREC 作为输入语言，并补充多参数过程和多声明的 `letrec` 表达式。其语法如所示，我们称之为 CPS-IN。为了区分这种语言和输出语言的表达式，我们把这些叫做*输入表达式* (*input expression*)。

要定义 CPS 变换算法的可能输出，我们要找出 CPI-IN 的子集，在这个集合中，过程调用不产生任何控制上下文。

回忆第 5 章中的原则:

> **不是过程调用，而是操作数的求值导致控制上下文扩大。**

那么，在

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

中，是 `fact` 的调用位置*作为操作数*导致了控制上下文的产生。相反，在

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

中，过程调用都不在操作数位置。我们说这些调用在*尾端* (*tail position*)，因为它们的值就是整个调用的结果。我们称之为*尾调用* (*tail call*)。

Program ::= *InpExp*
`a-program (exp1)`

InpExp ::= *Number*
`const-exp (num)`

InpExp ::= `(-` *InpExp* `,` *InpExp*`)`
`diff-exp (exp1 exp2)`

InpExp ::= `(zero?` *InpExp*`)`
`zero?-exp (exp1)`

InpExp ::= `if` *InpExp* `then` *InpExp* `else` *InpExp*
`if-exp (exp1 exp2 exp3)`

InpExp ::= *Identifier*
`var-exp (var)`

InpExp ::= `let` *Identifier* `=` *InpExp* `in` *InpExp*
`let-exp (var exp1 body)`

InpExp ::= `letrec` {*Identifier* `(`{*Identifier*}$^{*(,)}$`)` `=` *InpExp*}* `in` *InpExp*
`letrec-exp (p-names b-varss p-bodies letrec-body)`

InpExp ::= `proc (`{*Identifier*}$^{*(,)}$`)` *InpExp*
`proc-exp (vars body)`

InpExp ::= `(`*InpExp* {*InpExp*}*`)`
`call-exp (rator rands)`

图 6.3 CPS-IN 的语法

再回忆一下原则**尾调用不扩大续文**：

> **尾调用不扩大续文**
>
> 若 exp_1 的值作为 exp_2 的值返回，则 exp_1 和 exp_2 应在同样的续文中执行。

若所有过程调用和所有包含过程调用的子表达式都在尾端，我们称一个表达式为*尾式* (*tail form*)。这个条件表明所有过程调用都不会产生控制上下文。

因此，在 Scheme 中，

```
(if (zero? x) (f y) (g z))
```

是尾式，

```
(if b
  (if (zero? x) (f y) (g z))
  (h u))
```

也是尾式，但

```
(+
  (if (zero? x) (f y) (g z))
  37)
```

不是。因为 `if` 表达式包含一个不在尾端的过程调用。

通常，要判定语言的尾端，我们必须理解其含义。处于尾端的子表达式具有如下性质：该表达式求值后，其值随即成为整个表达式的值。一个表达式可能有多个尾端。例如，`if` 表达式可能选择真值分支，也可能选择假值分支。对尾端的子表达式，不需要保存信息，因此也就不会产生控制上下文。

CPS-IN 中的尾端如所示。尾端每个子表达式的值都可能成为整个表达式的值。在传递续文的解释器中，操作数位置的子表达式会产生新的续文。尾端的子表达式在原表达式的续文中求值，如 tail-call-explain 所述。

```
zero?(O)
-(O,O)
if O then T else T
let Var = O in T
letrec {Var ({Var}*(,)) = T}* in T
proc ({Var}*(,)) = T
(O O ... O)
```

图 6.4 CPS-IN 中的尾端和操作数位置。尾端记为 T。操作数位置记为 O。

我们根据这种区别设计 CPS 转换算法的目标语言 CPS-OUT。这种语言的语法如所示。这套语法定义了 CPS-IN 的子集，但略有不同。生成式的名字总以 `cps-` 开头，这样它们不会与 CPS-IN 中生成式的名字混淆。

新的语法有两个非终结符，*SimpleExp* 和 *TfExp*。这种设计中，*SimpleExp* 表达式不包含任何过程调用，*TfExp* 表达式一定是尾式。

因为 *SimpleExp* 表达式不包含任何过程调用，它们大致可以看成只有一行的简单代码，对我们来说，它们简单到不需使用控制堆栈。简单表达式包括 `proc` 表达式，因为 `proc` 表达式直接返回一个过程值，但过程的主体必须是尾式。

尾表达式的传递续文解释器如所示。由于这种语言的过程取多个参数，我们用 ex2.10 中的 `extend-env*` 创建多个绑定，并用类似方式扩展 `extend-env-rec`，得到 `extend-env-rec*`。

在这个解释器中，所有递归调用都在（Scheme 的）尾端，所以运行解释器不会在 Scheme 中产生控制上下文（不全是这样：过程 `value-of-simple-exp`（ex6.11）会在 Scheme 中产生控制上下文，但这可以避免（参见 ex6.18））。

更重要的是，解释器不会产生新的续文。过程 `value-of/k` 取一个续文参数，原封不动地传给每个递归调用。所以，我们可以很容易地移除续文参数。

当然，没有通用的方式判断一个过程的控制行为是否是迭代式的。考虑

```
(lambda (n)
  (if (strange-predicate? n)
    (fact n)
    (fact-iter n)))
```

只有 `strange-predicate?` 对所有足够大的 `n` 都返回假时，这个过程才是迭代式的。但即使能查看 `strange-predicate?` 的代码，也可能无法判断这一条件的真假。因此，我们最多只能寄希望于程序中的过程调用不产生控制上下文，而不论其是否执行。

Program ::= *TfExp*
```
cps-a-program (exp1)
```

SimpleExp ::= *Number*
```
cps-const-exp (num)
```

SimpleExp ::= *Identifier*
```
cps-var-exp (var)
```

SimpleExp ::= `(-` *SimpleExp* `,` *SimpleExp*`)`
```
cps-diff-exp (simple1 simple2)
```

SimpleExp ::= `(zero?` *SimpleExp*`)`
```
cps-zero?-exp (simple1)
```

SimpleExp ::= `proc (`{*Identifier*}$^{*(,)}$`)` *TfExp*
```
cps-proc-exp (vars body)
```

TfExp ::= *SimpleExp*
```
simple-exp->exp (simple-exp1)
```

TfExp ::= `let` *Identifier* `=` *SimpleExp* `in` *TfExp*
```
cps-let-exp (var simple1 body)
```

TfExp ::= `letrec` {*Identifier* `(`{*Identifier*}$^{*(,)}$`)` `=` *TfExp*}* `in` *TfExp*
```
cps-letrec-exp (p-names b-varss p-bodies body)
```

TfExp ::= `if` *SimpleExp* `then` *TfExp* `else` *TfExp*
```
cps-if-exp (simple1 body1 body2)
```

TfExp ::= `(`*SimpleExp* {*SimpleExp*}*`)`
```
call-exp (rator rands)
```

图 6.5 CPS-OUT 的语法

value-of/k : $TfExp \times Env \times Cont \to FinalAnswer$

```
(define value-of/k
  (lambda (exp env cont)
    (cases tfexp exp
      (simple-exp->exp (simple)
        (apply-cont cont
          (value-of-simple-exp simple env)))
      (cps-let-exp (var rhs body)
        (let ((val (value-of-simple-exp rhs env)))
          (value-of/k body
            (extend-env (list var) (list val) env)
            cont)))
      (cps-letrec-exp (p-names b-varss p-bodies letrec-body)
        (value-of/k letrec-body
          (extend-env-rec** p-names b-varss p-bodies env)
          cont))
      (cps-if-exp (simple1 body1 body2)
        (if (expval->bool (value-of-simple-exp simple1 env))
          (value-of/k body1 env cont)
          (value-of/k body2 env cont)))
      (cps-call-exp (rator rands)
        (let ((rator-proc
                (expval->proc
                  (value-of-simple-exp rator env)))
              (rand-vals
                (map
                  (lambda (simple)
                    (value-of-simple-exp simple env))
                  rands)))
          (apply-procedure/k rator-proc rand-vals cont))))))
```

apply-procedure/k : $Proc \times ExpVal \times Cont \to ExpVal$

```
(define apply-procedure/k
  (lambda (proc1 args cont)
    (cases proc proc1
      (procedure (vars body saved-env)
        (value-of/k body
          (extend-env* vars args saved-env)
          cont)))))
```

图 6.6 CPS-OUT 中的尾式解释器

练习 6.11 [⋆] 写出 value-of-simple-exp，完成中的解释器。

练习 6.12 [⋆] 判断下列表达式是否是简单的。

1. `-((f -(x,1)),1)`
2. `(f -(-(x,y),1))`
3. `if zero?(x) then -(x,y) else -(-(x,y),1)`
4. `let x = proc (y) (y x) in -(x,3)`
5. `let f = proc (x) x in (f 3)`

练习 6.13 [⋆] 用上面 cps-recipe 的 CPS 秘方，把下列 CPS-IN 表达式翻译为续文传递风格。用中的解释器运行转换后的程序，测试它们，确保原程序和转换后的版本对所有输入都给出同样的结果。

1. removeall。

```
letrec
 removeall(n,s) =
  if null?(s)
  then emptylist
  else if number?(car(s))
       then if equal?(n,car(s))
            then (removeall n cdr(s))
            else cons(car(s),
                      (removeall n cdr(s)))
       else cons((removeall n car(s)),
                 (removeall n cdr(s)))
```

2. occurs-in?。

```
letrec
 occurs-in?(n,s) =
  if null?(s)
  then 0
  else if number?(car(s))
       then if equal?(n,car(s))
            then 1
```

```
                else (occurs-in? n cdr(s))
           else if (occurs-in? n car(s))
                then 1
                else (occurs-in? n cdr(s))
```

3. remfirst。它使用前面例子中的 occurs-in?。

```
letrec
 remfirst(n,s) =
  letrec
   loop(s) =
    if null?(s)
    then emptylist
    else if number?(car(s))
         then if equal?(n,car(s))
              then cdr(s)
              else cons(car(s),(loop cdr(s)))
         else if (occurs-in? n car(s))
              then cons((remfirst n car(s)),
                        cdr(s))
              else cons(car(s),
                        (remfirst n cdr(s)))
  in (loop s)
```

4. depth。

```
letrec
 depth(s) =
  if null?(s)
  then 1
  else if number?(car(s))
       then (depth cdr(s))
       else if less?(add1((depth car(s))),
                     (depth cdr(s)))
            then (depth cdr(s))
            else add1((depth car(s)))
```

5. depth-with-let。

```
letrec
 depth(s) =
  if null?(s)
```

```
        then 1
        else if number?(car(s))
             then (depth cdr(s))
             else let dfirst = add1((depth car(s)))
                      drest = (depth cdr(s))
                  in if less?(dfirst,drest)
                     then drest
                     else dfirst
```

6. map。

```
letrec
 map(f, l) = if null?(l)
             then emptylist
             else cons((f car(l)),
                       (map f cdr(l)))
 square(n) = *(n,n)
in (map square list(1,2,3,4,5))
```

7. fnlrgtn。n-list 类似 s-list（s-list），只不过其中的元素不是符号，而是数字。fnlrgtn 取一 n-list，一个数字 n，返回列表中（从左向右数）第一个大于 n 的数字。一旦找到结果，就不再检查列表中剩余元素。例如，

```
(fnlrgtn list(1,list(3,list(2),7,list(9)))
 6)
```

返回 7。

8. every。这个过程取一谓词，一个列表，当且仅当谓词对列表中所有元素都为真时，返回真。

```
letrec
 every(pred, l) =
  if null?(l)
  then 1
  else if (pred car(l))
       then (every pred cdr(l))
       else 0
in (every proc(n) greater?(n,5) list(6,7,8,9))
```

练习 6.14 [⋆] 补充 value-of-program 和 apply-cont，完成中的解释器。

练习 6.15 [⋆] 观察前一道练习中的解释器可知，`cont` 只有一个值。根据这一观察完全移除 `cont` 参数。

练习 6.16 [⋆] 寄存中的解释器。

练习 6.17 [⋆] 把中的解释器转换为跳跃式。

练习 6.18 [⋆⋆] 修改 CPS-OUT 的语法，把简单 `diff-exp` 和 `zero?-exp` 的参数限制为常量和变量。这样，语言中的 `value-of-simple-exp` 就不必递归。

练习 6.19 [⋆⋆] 写出 Scheme 过程 `tail-form?`，它取一 CPS-IN 程序的语法树，语法如所示，判断同一字符串是否是中语法定义的尾式。

6.3 转换为续文传递风格

本节，我们开发算法，将任意程序从 CPS-IN 转换为 CPS-OUT。

就像传递续文的解释器一样，我们的翻译器跟随语法。也像传递续文的解释器一样，我们的翻译器再取一个表示续文的参数。多出的这个参数是一个表示续文的简单表达式。

像之前那样，我们首先给出例子，然后提出规范，最后写出程序。展示了与前一节类似的 Scheme 例子，只是更加详细。

```
(lambda (x)
  (cond
    ((zero? x) 17)
    ((= x 1) (f (- x 13) 7))
    ((= x 2) (+ 22 (- x 3) x))
    ((= x 3) (+ 22 (f x) 37))
    ((= x 4) (g 22 (f x)))
    ((= x 5) (+ 22 (f x) 33 (g y)))
    (else (h (f x) (- 44 y) (g y)))))
```

变换为

```
(lambda (x k)
  (cond
    ((zero? x) (k 17))
    ((= x 1) (f (- x 13) 7 k))
    ((= x 2) (k (+ 22 (- x 3) x)))
    ((= x 3) (f x (lambda (v1) (k (+ 22 v1 37)))))
    ((= x 4) (f x (lambda (v1) (g 22 v1 k))))
    ((= x 5) (f x (lambda (v1)
                    (g y (lambda (v2)
                           (k (+ 22 v1 33 v2)))))))
      (else (f x (lambda (v1)
                   (g y (lambda (v2)
                          (h v1 (- 44 y) v2 k)))))))))
```

图 6.7 CPS 变换示例（Scheme）

第一种情况是常量。常量直接传给续文，就像上面 `(zero? x)` 这一行。

`(cps-of-exp` n K`) = (`K n`)`

其中，K 表示续文，是一个 simple-exp。

同样，变量直接传给续文。

`(cps-of-exp` var K`) = (`K var`)`

当然，我们算法的输入输出都是抽象语法树，所以我们应该用抽象语法构造器，而不是具体语法，就像：

```
(cps-of-exp (const-exp n) K)
= (make-send-to-cont K (cps-const-exp n))

(cps-of-exp (var-exp var) K)
= (make-send-to-cont K (cps-var-exp var))
```

其中：

make-send-to-cont : *SimpleExp* × *SimpleExp* → *TfExp*

```
(define make-send-to-cont
  (lambda (k-exp simple-exp)
    (cps-call-exp k-exp (list simple-exp))))
```

我们需要 `list`，因为在 CPS-OUT 中，每个调用表达式都取一个操作数列表。

但是在规范中，我们仍然使用具体语法，因为具体语法通常更容易读懂。

过程呢？我们转换中那样的 `(lambda (x) ...)` 过程时，为其新增一个参数 `k`，然后转换主体，并将主体的值传给续文 `k`。我们在中正是这样做的。所以

`proc (`var_1`, ..., `var_n`)` exp

变成：

`proc (`var_1`, ..., `var_n`, k) (cps-of-exp` exp `k)`

就像图中那样。但是，这还没完。我们的目标是生成代码，求 `proc` 表达式的值，并将结果传给续文 K。所以 `proc` 表达式的完整规范为：

```
(cps-of-exp <<proc (var_1, ..., var_n) exp>> K)
= (K <<proc (var_1, ..., var_n, k) (cps-of-exp exp k)>>)
```

其中，`k` 是新变量，K 表示续文，是任意简单表达式。

有操作数的表达式呢？我们暂时给语言添加任意多个操作数的求和表达式。要添加这种表达式，我们给 CPS-IN 的语法添加生成式：

$$\textit{InpExp} ::= \texttt{+(}\{\textit{InpExp}\}^{*(,)}\texttt{)}$$
`sum-exp (exps)`

给 CPS-OUT 的语法添加生成式：

$$\textit{SimpleExp} ::= \texttt{+(}\{\textit{SimpleExp}\}^{*(,)}\texttt{)}$$
`cps-sum-exp (simple-exps)`

这个新生成式仍保持性质：简单表达式内不会出现过程调用。

`(cps-of-exp •+(`exp_1`, ..., `exp_n`)• `K`)` 可能是什么呢？可能 exp_1, ..., exp_n 都是简单的，就像中的 `(= x 2)`。那么，整个表达式都是简单的，我们可以将它的值接传给续文。设 $simp$ 为一简单表达式，那么有：

```
(cps-of-exp <<+(simp_1, ..., simp_n)>> K)
= (K <<+(simp_1, ..., simp_n)>>)
```

如果操作数不是简单的呢？那么求值续文需要给其值命名，然后继续求和，就像上面 `(= x 3)` 这行。其中的第二个是首个复杂操作数。那么我们的 CPS 转换器应具有性质：

```
(cps-of-exp <<+(simp_1, exp_2, simp_3, ..., simp_n)>> K)
= (cps-of-exp exp_2
    <<proc (var_2) (K +(simp_1, var_2, simp_3, ..., simp_n))>>)
```

如果 exp_2 只是一个过程调用，那么输出和图中相同。但 exp_2 可能更复杂，所以我们递归调用 `cps-of-exp` 处理 exp_2 和更大的续文：

```
proc (var_2) (K +(simp_1, var_2, simp_3, ..., simp_n))
```

而求和表达式中，还有另一种复杂操作数，就像 `(= x 5)` 这种。所以，不是直接使用续文

```
proc (var2) (K +(simp1, var2, ..., simpn))
```

我们还要递归处理更大的参数。我们可以把这条规则总结为:

```
(cps-of-exp <<+(simp1, exp2, exp3, ..., expn)>> K)
= (cps-of-exp exp2
    <<proc (var2)
       (cps-of-exp <<+(simp1, var2, exp3, ..., expn))>> K)
```

`cps-of-exp` 的每个递归调用都保证会终止。第一个调用会终止是因为 exp_2 比原表达式小。第二个调用会终止是还是因为其参数也比原来的小: var_2 总是比 exp_2 小。

例如, 查看 `(= x 5)` 这一行, 并使用 CPS-IN 的语法, 我们有:

```
(cps-of-exp <<+((f x), 33, (g y))>> K)
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<+(v1, 33, (g y))>> K)>>)
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<(g y)>>
         <<proc v2
            (cps-of-exp <<+(v1, 33, v2)>> K)))
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<(g y)>>
         <<proc v2
            (K <<+(v1, 33, v2)>>)))
= (f x
   proc (v1)
    (g y
     proc (v2)
      (K <<+(v1, 33, v2)>>)))
```

过程调用与之类似。如果操作符和操作数都是简单的, 那么我们添加续文参数, 直接调用过程, 就像 `(= x 2)` 这行。

```
(cps-of-exp <<(simp0 simp1 ... simpn)>> K)
= (simp0 simp1 ... simpn K)
```

另一方面，如果某个操作数是复杂的，那么我们必须先求它的值，像 `(= x 4)` 这行。

```
(cps-of-exp <<(simp0 simp1 exp2 exp3 ... expn)>> K)
= (cps-of-exp exp2
    <<proc (var2)
       (cps-of-exp <<(simp0 simp1 var2 exp3 ... expn)>> K)>>)
```

而且，像之前那样，`cps-of-exp` 的第二个调用递归进入过程调用之中，为每个复杂参数调用 `cps-of-exp`，直到所有参数都是简单参数。

写成 CPS-IN 的例子 `(= x 5)` 可用这些规则处理如下：

```
(cps-of-exp <<(h (f x) -(44,y) (g y))>> K)
= (cps-of-exp <<(f x)>>
    <<proc (v1)
       (cps-of-exp <<(h v1 -(44,y) (g y))>> K)>>)
= (f x
   proc (v1)
    (cps-of-exp <<(h v1 -(44,y) (g y))>> K))
= (f x
   proc (v1)
    (cps-of-exp <<(g y)>>
      <<proc (v2)
         (cps-of-exp <<(h v1 -(44,y) v2)>> K)))
= (f x
   proc (v1)
    (g y
     proc (v2)
      (cps-of-exp <<(h v1 -(44,y) v2)>> K)))
= (f x
   proc (v1)
    (g y
     proc (v2)
      (h v1 -(44,y) v2 K)))
```

求和表达式和过程调用的规范遵循同样的模式：找出第一个复杂操作数，递归处理那个操作数和修改过的操作数列表。这对任何求值操作数的表达式都有效。如果 `complex-exp` 是某个需要求值操作数的 CPS-IN 表达式，那么我们有：

```
(cps-of-exp (complex-exp simp_0 simp_1 exp_2 exp_3 ... exp_n) K)
= (cps-of-exp exp_2
    <<proc (var_2)
       (cps-of-exp
         (complex-exp simp_0 simp_1 exp_2 exp_3 ... exp_n)
         K)>>)
```

其中，var_2 是一个新变量。

处理求和表达式和过程调用的唯一不同之处是在所有参数都简单时。在这种情况下，我们要把每个参数转换为 CPS-OUT 中的 `simple-exp`，并用转换结果生成一个尾式。

我们可以把这种行为封装到过程 `cps-of-exps` 中，如所示。它的参数是输入表达式的列表和过程 `builder`。它用 ex1.23 中的 `list-index`，找出列表中第一个复杂表达式的位置。如果有这样的复杂表达式，那么它转换该表达式，转换所在的续文给表达式的结果命名（绑定到 `var` 的标识符），然后递归处理修改后的表达式列表。

如果不存在复杂表达式，那么我们用 `builder` 处理表达式列表。但这些表达式虽是简单的，它们仍属于 CPS-IN 的语法。因此，我们用过程 `cps-of-simple-exp` 把每个表达式转换为 CPS-OUT 的语法。然后，我们把 *SimpleExp* 的列表传给 `builder`（`list-set` 如 ex1.19 所述）。

过程 `inp-exp-simple?` 取一 CPS-IN 表达式，判断表示它的字符串能否解析为 *SimpleExp*。它使用 ex1.24 中的过程 `every?`。若 lst 中的所有元素满足 $pred$，`(every?` $pred$ lst`)` 返回 `#t`，否则返回 `#f`。

`cps-of-simple-exp` 的代码直截了当，如所示。它还将 `proc-exp` 的主体翻译做 CPS 变换。若要使输出为 *SimpleExp*，这是必要的。

我们可以用 `cps-of-exps` 生成求和表达式和过程调用的尾式。

cps-of-exps : $Listof(InpExp) \times (Listof(InpExp) \to TfExp) \to TfExp$

```
(define cps-of-exps
  (lambda (exps builder)
    (let cps-of-rest ((exps exps))
      cps-of-rest : Listof(InpExp) → TfExp
      (let ((pos (list-index
                  (lambda (exp)
                    (not (inp-exp-simple? exp)))
                  exps)))
        (if (not pos)
          (builder (map cps-of-simple-exp exps))
          (let ((var (fresh-identifier 'var)))
            (cps-of-exp
              (list-ref exps pos)
              (cps-proc-exp (list var)
                (cps-of-rest
                  (list-set exps pos (var-exp var))))))))))))
```

inp-exp-simple? : $InpExp \to Bool$

```
(define inp-exp-simple?
  (lambda (exp)
    (cases expression exp
      (const-exp (num) #t)
      (var-exp (var) #t)
      (diff-exp (exp1 exp2)
        (and (inp-exp-simple? exp1) (inp-exp-simple? exp2)))
      (zero?-exp (exp1) (inp-exp-simple? exp1))
      (proc-exp (ids exp) #t)
      (sum-exp (exps) (every? inp-exp-simple? exps))
      (else #f))))
```

图 6.8 cps-of-exps

cps-of-sum-exp : $Listof(InpExp) \times SimpleExp \to TfExp$

```
(define cps-of-sum-exp
  (lambda (exps k-exp)
    (cps-of-exps exps
      (lambda (simples)
        (make-send-to-cont
          k-exp
          (cps-sum-exp simples))))))
```

---

cps-of-simple-exp : $InpExp \to SimpleExp$

用法: 设 `(inp-exp-simple? exp) = #f`

```
(define cps-of-simple-exp
  (lambda (exp)
    (cases expression exp
      (const-exp (num) (cps-const-exp num))
      (var-exp (var) (cps-var-exp var))
      (diff-exp (exp1 exp2)
        (cps-diff-exp
          (cps-of-simple-exp exp1)
          (cps-of-simple-exp exp2)))
      (zero?-exp (exp1)
        (cps-zero?-exp (cps-of-simple-exp exp1)))
      (proc-exp (ids exp)
        (cps-proc-exp (append ids (list 'k%00))
          (cps-of-exp exp (cps-var-exp 'k%00))))
      (sum-exp (exps)
        (cps-sum-exp (map cps-of-simple-exp exps)))
      (else
        (report-invalid-exp-to-cps-of-simple-exp exp)))))
```

图 6.9 `cps-of-simple-exp`

---

cps-of-call-exp : $InpExp \times Listof(InpExp) \times SimpleExp \to TfExp$

```
(define cps-of-call-exp
  (lambda (rator rands k-exp)
    (cps-of-exps (cons rator rands)
      (lambda (simples)
        (cps-call-exp
          (car simples)
          (append (cdr simples) (list k-exp)))))))
```

现在，我们可以写出 CPS 翻译器的剩余部分（•fig-6.12）。它跟随语法。当表达式总是简单的，如常量、变量和过程，我们直接用 `make-send-to-cont` 生成代码。否则，我们调用辅助过程，每个辅助过程都调用 `cps-of-exps` 求子表达式的值，用适当的生成器构造 CPS 输出的最内部。一个例外是 `cps-of-letrec-exp`，它没有紧邻的子表达式，所以它直接生成 CPS 输出。最后，我们调用 `cps-of-exps` 翻译整个程序，它取一生成器，该生成器直接返回一个简单表达式。

在下面的练习中，用 CPS-OUT 的语法和解释器运行输出表达式，确保它们是尾式。

cps-of-exp : *InpExp* × *SimpleExp* → *TfExp*

```
(define cps-of-exp
  (lambda (exp k-exp)
    (cases expression exp
      (const-exp (num)
        (make-send-to-cont k-exp (cps-const-exp num)))
      (var-exp (var)
        (make-send-to-cont k-exp (cps-var-exp var)))
      (proc-exp (vars body)
        (make-send-to-cont k-exp
          (cps-proc-exp (append vars (list 'k%00))
            (cps-of-exp body (cps-var-exp 'k%00)))))
      (zero?-exp (exp1)
        (cps-of-zero?-exp exp1 k-exp))
      (diff-exp (exp1 exp2)
        (cps-of-diff-exp exp1 exp2 k-exp))
      (sum-exp (exps)
        (cps-of-sum-exp exps k-exp))
      (if-exp (exp1 exp2 exp3)
        (cps-of-if-exp exp1 exp2 exp3 k-exp))
      (let-exp (var exp1 body)
        (cps-of-let-exp var exp1 body k-exp))
      (letrec-exp (p-names b-varss p-bodies letrec-body)
        (cps-of-letrec-exp
          p-names b-varss p-bodies letrec-body k-exp))
      (call-exp (rator rands)
        (cps-of-call-exp rator rands k-exp)))))
```

cps-of-zero?-exp : *InpExp* × *SimpleExp* → *TfExp*

```
(define cps-of-zero?-exp
  (lambda (exp1 k-exp)
    (cps-of-exps (list exp1)
      (lambda (simples)
        (make-send-to-cont
          k-exp
          (cps-zero?-exp
            (car simples)))))))
```

图 6.10 `cps-of-exp`，第 1 部分

cps-of-diff-exp : $InpExp \times InpExp \times SimpleExp \to TfExp$

```
(define cps-of-diff-exp
  (lambda (exp1 exp2 k-exp)
    (cps-of-exps
      (list exp1 exp2)
      (lambda (simples)
        (make-send-to-cont
          k-exp
          (cps-diff-exp
            (car simples)
            (cadr simples)))))))
```

cps-of-if-exp : $InpExp \times InpExp \times SimpleExp \to TfExp$

```
(define cps-of-if-exp
  (lambda (exp1 exp2 exp3 k-exp)
    (cps-of-exps (list exp1)
      (lambda (simples)
        (cps-if-exp (car simples)
          (cps-of-exp exp2 k-exp)
          (cps-of-exp exp3 k-exp))))))
```

cps-of-let-exp : $Var \times InpExp \times InpExp \times SimpleExp \to TfExp$

```
(define cps-of-let-exp
  (lambda (id rhs body k-exp)
    (cps-of-exp
      (call-exp
        (proc-exp (list id) body)
        (list rhs))
      k-exp)))
```

cps-of-letrec-exp :
$Listof(Var) \times Listof(Listof(Var)) \times Listof(InpExp) \times InpExp \times SimpleExp \to TfExp$

```
(define cps-of-letrec-exp
  (lambda (p-names b-varss p-bodies letrec-body k-exp)
    (cps-letrec-exp
      p-names
      (map
        (lambda (b-vars) (append b-vars (list 'k%00)))
        b-varss)
      (map
        (lambda (p-body)
          (cps-of-exp p-body (cps-var-exp 'k%00)))
        p-bodies)
      (cps-of-exp letrec-body k-exp))))
```

图 6.11 `cps-of-exp`，第 2 部分

cps-of-program : *InpExp* → *TfExp*

```
(define cps-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cps-a-program
          (cps-of-exps (list exp1)
            (lambda (new-args)
              (simple-exp->exp (car new-args)))))))))
```

图 6.12 `cps-of-exp`，第 3 部分

练习 6.20 [⋆] 我们的过程 `cps-of-exps` 迫使子表达式按从左向右的顺序求值。修改 `cps-of-exps`，使子表达式从右向左求值。

练习 6.21 [⋆] 修改 `cps-of-call-exp`，先从左向右求出操作数的值，再求操作符的值。

练习 6.22 [⋆] 有时，当我们生成 (*K simp*)，*K* 已经是一个 `proc-exp`。所以，不是生成：

```
(proc (var1) ... simp)
```

而应生成：

```
let var1 = simp
in ...
```

那么，CPS 代码具有性质：形如

```
(proc (var) exp1
 simp)
```

的子表达式若不在原表达式中，则不会出现在 CPS 代码中。

修改 `make-send-to-cont`，生成更好的代码。新的规则何时生效?

练习 6.23 [⋆⋆] 观察可知，`if` 的规则导致续文 *K* 复制两次，所以在嵌套的 `if` 中，转换后的代码尺寸呈指数增长。运行一个例子，验证这一观察。然后，修改转换，把 *K* 绑定到新的变量，从而避免这种增长。

练习 6.24 [⋆⋆] 给语言添加列表（ex3.10）。记住：列表的参数不在尾端。

练习 6.25 [⋆⋆] 扩展 CPS-IN，让 `let` 表达式声明任意数量的变量（ex3.16）。

练习 6.26 [⋆⋆] 由 `cps-of-exps` 引入的续文变量在续文中只会只会出现一次。修改 `make-send-to-cont`，不是生成 ex6.22 中的

```
let var1 = simp1
in T
```

而是生成 $T[simp_1/var_1]$。其中，符号 $E_1[E_2/var]$ 意为，把表达式 E_1 中自由出现的所有变量 var 替换为 E_2。

练习 6.27 [⋆⋆] 按当前方式，`cps-of-let-exp` 生成一个无用的 `let` 表达式（为什么?）。修改这个过程，直接把 `let` 变量作为续文变量。那么，若 exp_1 是复杂的，

```
(cps-of-exp <<let var1 = exp1 in exp2>> K)
= (cps-of-exp exp1 <<proc (var1) (cps-of-exp exp2 K)>>)
```

练习 6.28 [⋆] 试想：有一个 Scheme 程序的 CPS 转换器，用它转换第 3 章中的第一个解释器，结果会怎样?

练习 6.29 [⋆⋆] 考虑 `cps-of-exps` 的变体。

```
(define cps-of-exps
  (lambda (exps builder)
    (let cps-of-rest ((exps exps) (acc '()))
      cps-of-rest : Listof(InpExp) × Listof(SimpleExp) → TfExp
      (cond
        ((null? exps) (builder (reverse acc)))
        ((inp-exp-simple? (car exps))
          (cps-of-rest (cdr exps)
            (cons
              (cps-of-simple-exp (car exps))
              acc)))
        (else
          (let ((var (fresh-identifier 'var)))
            (cps-of-exp (car exps)
              (cps-proc-exp (list var)
                (cps-of-rest (cdr exps)
                  (cons
                    (cps-of-simple-exp (var-exp var))
                    acc)))))))))
```

为什么 cps-of-exp 的这种变体比中的更高效?

练习 6.30 [⋆⋆] 调用 cps-of-exps 处理长度为 1 的表达式列表可以简化如下:

```
(cps-of-exps (list exp) builder)
= (cps-of-exp/ctx exp (lambda (simp) (builder (list simp))))
```

其中,

cps-of-exp/ctx : $InpExp \times (SimpleExp \square TfExp) \to TfExp$

```
(define cps-of-exp/ctx
  (lambda (exp context)
    (if (inp-exp-simple? exp)
      (context (cps-of-simple-exp exp))
      (let ((var (fresh-identifier 'var)))
        (cps-of-exp exp
          (cps-proc-exp (list var)
            (context (cps-var-exp var))))))))
```

这样，由于列表的参数数量已经确定，我们可以简化出现 (cps-of-exps (list ...)) 的地方。那么，诸如 cps-of-diff-exp 可以用 cps-of-exp/ctx 定义，而不需要 cps-of-exps。

```
(define cps-of-diff-exp
  (lambda (exp1 exp2 k-exp)
    (cps-of-exp/ctx exp1
      (lambda (simp1)
        (cps-of-exp/ctx exp2
          (lambda (simp2)
            (make-send-to-cont k-exp
              (cps-diff-exp simp1 simp2))))))))
```

对 cps-of-call-exp 中用到的 cps-of-exps，我们可以用 cps-of-exp/ctx 处理 rator，但仍需使用 cps-of-exps 处理 rands。删除翻译器中其他地方出现的 cps-of-exps。

练习 6.31 [⋆⋆⋆] 写一个翻译器，它取 `cps-of-program` 的输出，生成一个等价程序，其中所有的续文都用第 5 章中的数据结构表示。用列表表示那些用 `define-datatype` 生成的数据结构。由于我们的语言不支持符号，你可以在首项位置使用整数标签，以此区分数据类型的变体。

练习 6.32 [⋆⋆⋆] 写一个翻译器，它类似 ex6.31，但把所有过程表示为数据结构。

练习 6.33 [⋆⋆⋆] 写一个翻译器，它取 ex6.32 的输出，将其转换为那样的寄存器程序。

练习 6.34 [⋆⋆] 我们把程序转换为 CPS 时，不仅将程序中的控制上下文变为显式的，而且还确定了操作的执行顺序，以及所有中间结果的名字。后者叫做*序列化* (*sequentialization*)。如果我们不关心能否获得迭代性控制行为，我们序列化程序时可将其转换为 *A-normal form*，或称 *ANF*。这里是一个 ANF 程序的例子。

```
(define fib/anf
  (lambda (n)
    (if (< n 2)
      1
      (let ((val1 (fib/anf (- n 1))))
        (let ((val2 (fib/anf (- n 2))))
          (+ val1 val2))))))
```

CPS 程序传递命名中间结果的续文，从而序列化计算；ANF 程序用 `let` 表达式命名所有中间结果，从而序列化计算。

重写 `cps-of-exp`, 生成 ANF 程序而非 CPS 程序（对不在尾端的条件表达式, 用 ex6.23 中的方法处理）。然后，用修改后的 `cps-of-exp` 处理例 `fib` 的定义，验证其结果是否为 `fib/anf`。最后，验证对已经是 ANF 的输入程序，你的翻译器产生的程序与输入只有绑定变量名不同。

练习 6.35 [⋆] 用几个例子验证：若采用 ex6.27 中的优化方法，对 ANF 转换器（ex6.34）的输入和输出程序进行 CPS 变换，所得结果相同。

6.4 建模计算效果

CPS 的另一重要应用是提供模型，将计算效果变为显式的。计算效果••像是打印或给变量赋值••很难用第 3 章使用的方程推理建模。通过 CPS 变换，我们可以将这些效果变为显式的，就像我们在第 5 章中处理非局部控制流一样。

用 CPS 建模效果时，我们的基本原则是简单表达式不应有任何效果。简单表达式不应含有过程调用也是基于这一原则，因为过程调用可能不终止（这当然是一种效果!）。

本节，我们研究三种效果：打印，存储器（用显式引用模型），以及非标准控制流。

我们首先考虑打印。打印当然是一种效果：

```
(f print(3) print(4))
```

和

```
(f 1 1)
```

即使返回同样的答案，也具有不同效果。效果还取决于参数的求值顺序。迄今为止，我们的语言总是按从左向右的顺序求参数的值，但其他语言可能不是这样。

要建模这些想法，我们按照下面的方式修改 CPS 变换：

- 我们给 CPS-IN 添加 `print` 表达式：

  InpExp ::= `print (`*InpExp*`)`
  `print-exp (exp1)`

  我们尚未写出 CPS-IN 的解释器，但解释器应当扩展，从而处理 `print-exp`；它打印出参数的值，返回某个值（我们选任意值 38）。

- 我们给 CPS-OUT 添加 `printk` 表达式：

  TfExp ::= `printk (`*SimpleExp*`) ;` *TfExp*
  `cps-printk-exp (simple-exp1 body)`

表达式 printk(*simp*);*exp* 有一种效果：打印。因此，它必须是一个 *TfExp*，而非 *SimpleExp*，且只能出现在尾端。*exp* 的值成为整个 printk 表达式的值，所以 *exp* 本身在尾端，可以是一个 tfexp。那么，这部分代码可以写作：

```
proc (v1)
 printk(-(v1,1));
  (f v1 K)
```

要实现它，我们给 CPS-OUT 的解释器添加：

```
(printk-exp (simple body)
  (begin
    (eopl:printf "~s~%"
      (value-of-simple-exp simple env))
    (value-of/k body env cont)))
```

- 我们给 cps-of-exp 添加一行代码，把 print 表达式翻译为 printk 表达式。我们为 print 选择任意返回值 38。所以，我们的翻译应为：

  (cps-of-exp <<print(*simp*$_1$)>> *K*) = printk(*simp*$_1$) ; (*K* 38)

  然后，由于 print 的参数可能是复杂的，我们用 cps-of-exps 处理。这样，我们给 cps-of-exp 新增这几行代码：

```
(print-exp (rator)
  (cps-of-exps (list rator)
    (lambda (simples)
      (cps-printk-exp
        (car simples)
        (make-send-to-cont k-exp
          (cps-const-exp 38))))))
```

来看一个更复杂的例子。

```
(cps-of-exp <<(f print((g x)) print(4))>> K)
= (cps-of-exp <<print((g x))>>
    <<proc (v1)
       (cps-of-exp <<(f v1 print(4))>> K)>>)
= (cps-of-exp <<(g x)>>
    <<proc (v2)
       (cps-of-exp <<(print v2)>>
         <<proc (v1)
            (cps-of-exp <<(f v1 print(4))>> K)>>)>>)
= (g x
   proc (v2)
    (cps-of-exp <<(print v2)>>
      <<proc (v1)
        (cps-of-exp <<(f v1 print(4))>> K)>>))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in (cps-of-exp <<(f v1 print(4))>> K))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in (cps-of-exp <<print(4)>>
         <<proc (v3)
            (cps-of-exp <<(f v1 v3)>> K)>>))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in printk(4);
       let v3 = 38
       in (cps-of-exp <<(f v1 v3)>> K))
= (g x
   proc (v2)
    printk(v2);
    let v1 = 38
    in printk(4);
       let v3 = 38
       in (f v1 v3 k))
```

这里，我们调用 `g`，调用所在的续文把结果命名为 `v2`。续文打印出 `v2` 的值，把 38 传给下一续文，下一续文将 `v1` 绑定到实参 38，打印出 4，然后调用下一续文，下一续文把 `v2` 绑定到实参（也是 38），然后用 `v1`，`v3` 和 K 调用 `f`。

我们按照同样的步骤建模显式引用（4.2 节）。我们给 CPS-IN 和 CPS-OUT 添加新的语法，给 CPS-OUT 的解释器添加代码，处理新的语法，给 `cps-of-exp` 添加代码，将新的 CPS-IN 语法翻译为 CPS-OUT。对显式引用，我们需要添加创建引用，解引用和赋值的语法。

- 我们给 CPS-IN 添加语法：

InpExp ::= `newref (`*InpExp*`)`
```
newref-exp (exp1)
```

InpExp ::= `deref (`*InpExp*`)`
```
deref-exp (exp1)
```

InpExp ::= `setref (`*InpExp* `,` *InpExp*`)`
```
setref-exp (exp1 exp2)
```

- 我们给 CPS-OUT 添加语法：

TfExp ::= `newrefk (`*simple-exp*`,` *simple-exp*`)`
```
cps-newrefk-exp (simple1 simpe2)
```

TfExp ::= `derefk (`*simple-exp*`,` *simple-exp*`)`
```
cps-derefk-exp (simple1 simpe2)
```

TfExp ::= `setrefk (`*simple-exp*`,` *simple-exp*`) ;` *TfExp*
```
cps-setrefk-exp (simple1 simpe2)
```

`newrefk` 表达式取两个参数：放入新分配单元的值，接收新位置引用的续文。`derefk` 与之类似。由于 `setrefk` 的执行通常只求效果，`setrefk` 的设计与 `printk` 类似。它将第二个参数的值赋给第一个参数的值，后者应是一个引用，然后尾递归式地执行第三个参数。

在这门语言中，我们写：

```
newrefk(33, proc (loc1)
             newrefk(44, proc (loc2)
                          setrefk(loc1,22);
                          derefk(loc1, proc (val)
                                       -(val,1))))
```

这个程序新分配一个位置，值为 33，把 `loc1` 绑定到那个位置。然后，它新分配一个位置，值为 44，把 `loc2` 绑定到那个位置。然后，它把位置 `loc1` 的内容设为 22。最后，它取出 `loc1` 的值，把结果（应为 22）绑定到 `val`，求出并返回 `-(val,1)` 的结果 21。

要得到这种行为，我们给 CPS-OUT 的解释器添加这几行代码：

```
(cps-newrefk-exp (simple1 simple2)
  (let ((val1 (value-of-simple-exp simple1 env))
        (val2 (value-of-simple-exp simple2 env)))
    (let ((newval (ref-val (newref val1))))
      (apply-procedure/k
        (expval->proc val2)
        (list newval)
        k-exp))))

(cps-derefk-exp (simple1 simple2)
  (apply-procedure/k
    (expval->proc (value-of-simple-exp simple2 env))
    (list
      (deref
        (expval->ref
          (value-of-simple-exp simple1 env))))
    k-exp))

(cps-setrefk-exp (simple1 simple2 body)
  (let ((ref (expval->ref
               (value-of-simple-exp simple1 env)))
        (val (value-of-simple-exp simple2 env)))
    (begin
      (setref! ref val)
      (value-of/k body env k-exp))))
```

- 最后，我们给 `cps-of-exp` 添加这几行代码来做翻译：

```
(newref-exp (exp1)
  (cps-of-exps (list exp1)
    (lambda (simples)
      (cps-newrefk-exp (car simples) k-exp))))

(deref-exp (exp1)
  (cps-of-exps (list exp1)
    (lambda (simples)
      (cps-derefk-exp (car simples) k-exp))))

(setref-exp (exp1 exp2)
  (cps-of-exps (list exp1 exp2)
    (lambda (simples)
      (cps-setrefk-exp
        (car simples)
        (cadr simples)
        (make-send-to-cont k-exp
          (cps-const-exp 23))))))
```

在最后一行，我们让 `setref` 返回 23，这与 EXPLICIT-REFS 一致。

练习 6.36 [⋆⋆] 给 CPS-IN 添加 `begin` 表达式（ex4.4）。CPS-OUT 应该不需要修改。

练习 6.37 [⋆⋆⋆] 给 CPS-IN 添加隐式引用（ 4.3 节）。用和显式引用相同的 CPS-OUT，确保翻译器在适当的地方插入分配和解引用。提示：回忆一下，在隐式引用出现的地方，`var-exp` 不再是简单的，因为它需要读取存储器。

练习 6.38 [⋆⋆⋆] 如果一个变量不会出现在 `set` 表达式的左边，它是不可变的，因此可以视为简单的。扩展前一题的解答，按简单表达式处理所有这样的变量。

最后是非局部控制流。我们来考虑 ex5.42 中的 `letcc`。`letcc` 表达式 `letcc` *var* `in` *body* 将当前续文绑定到变量 *var*。*body* 为该绑定的作用域。续文的唯一操作是 `throw`。我们用语法 `throw` *Expression* `to` *Expression*，它

需要求出两个子表达式的值。第二个表达式应返回一个续文，该续文作用于第一个表达式的值。`throw` 当前的续文则被忽略。

我们首先按照本章的方式分析这些表达式。这些表达式一定是复杂的。`letcc` 的主体部分在尾端，因为它的值就是整个表达式的值。由于 `throw` 中的两个位置都需求值，且都不是 `throw` 的值（确实，`throw` 没有值，因为它不会返回到紧邻的续文），因此它们都在操作数位置。

现在，我们可以写出转换这两个表达式的规则。

(cps-of-exp <<letcc var in $body$>> K)
= let var = K
  in (cps-of-exp $body$ var)

(cps-of-exp <<throw $simp_1$ to $simp_2$>> K)
= ($simp_2$ $simp_1$)

我们仍用 `cps-of-exps` 处理 `throw` 可能含有的复杂参数。这里，K 如期望的那样忽略。

这个例子中，我们不需要给给 CPS-OUT 添加语法，因为我们操作的正是控制结构。

练习 6.39 [⋆] 在 CPS 翻译器中实现 `letcc` 和 `throw`。

练习 6.40 [⋆⋆] 在 CPS 翻译器中添加和实现 5.4 节中的 `try/catch` 和 `throw`。CPS-OUT 应该不需要添加任何东西，而 `cps-of-exp` 改取两个续文：一个成功续文，一个错误续文。

7 类型

我们已理解如何用解释器建模程序的运行时行为。现在，我们用同样的技术不加运行地分析或预测程序的行为。

我们已见识过一些了：我们的词法地址翻译器在分析阶段预测程序在运行时如何从环境中找出各个变量。而且，翻译器本身看起来就像一个解释器，只是我们传递的不是环境，而是静态环境。

我们的目标是分析程序，预测程序求值是否安全 (*safe*)，即，求值过程是否能避免某些类型的错误，但安全的含义视语言而不同。如果我们能保证求值是安全的，我们就能确保程序满足其合约。

本章，我们考虑类似第 3 章 LETREC 的语言。这些语言求值安全，当且仅当：

1. 每个待求值的变量 var 都已绑定。

2. 每个待求值的差值表达式 (`diff-exp` exp_1 exp_2) 中，exp_1 和 exp_2 的值都是 `num-val`。

3. 每个待求值的表达式 (`zero?-exp` exp_1) 中，exp_1 的值都是 `num-val`。

4. 每个待求值的条件表达式 (`if-exp` exp_1 exp_2 exp_3) 中，exp_1 的值都是 `bool-val`。

5. 每个待求值的过程调用 (`call-exp` $rator$ $rand$) 中，$rator$ 的值都是 `proc-val`。

这些条件确保每个操作符都作用于正确类型的操作数。因此，我们说违反这些条件是*类型错误* (*type error*)。

安全的求值仍可能因为其他原因而失败：除以零，取空列表的 `car`，等等。我们不把这些算作安全的定义，因为在预测安全性时，保证这些条件要比上面列出的难得多。同样地，安全的求值可能永远运行。我们的安全定义不包含不终止，因为检查程序是否终止也很困难（事实上，这一般是无法判定的）。有些语言的类型系统给出比上述更强的保证，但它们要比我们这里考虑的复杂得多。

我们的目标是写出过程，查看程序文本，接受或者拒绝它。而且，我们希望我们的分析过程保守一点：如果分析接受程序，那么我们确保求程序的值是安全的。如果分析不能确定求值是否安全，它必须拒绝程序。我们称这样的分析是*健壮的* (*sound*)。

拒绝所有程序的分析仍是健壮的，可我们还是想让我们的分析接受一大批程序。本章的分析将接受足够多的程序，因此是有用的。

这里是一些示例程序，以及它们应被分析拒绝或接受：

```
if 3 then 88 else 99          拒绝：条件非布尔值
proc (x) (3 x)                拒绝：rator 非过程值
proc (x) (x 3)                接受
proc (f) proc (x) (f x)       接受
let x = 4 in (x 3)            拒绝：rator 非过程值

(proc (x) (x 3)               拒绝：同前例
 4)

let x = zero?(0)              拒绝：diff-exp 参数非整数
in -(3, x)

(proc (x) -(3,x)              拒绝：同前例
 zero?(0))

let f = 3                     拒绝：rator 非过程值
in proc (x) (f x)

(proc (f) proc (x) (f x)      拒绝：同前例
 3)

letrec f(x) = (f -(x,-1))     接受，不终止，但是安全
```

```
in (f 1)
```

虽然最后一个例子求值不终止，但根据上述定义，求值仍是安全的，所以我们的分析可以接受它。之所以接受它，是因为我们的分析器不够好，不足以判定这个程序不会终止。

7.1 值及其类型

由于安全条件只涉及 `num-val`、`bool-val` 和 `proc-val`，有人可能以为记录这三种类型就足够了。但那是不够的：如果我们只知道 `f` 绑定到一个 `proc-val`，我们根本无法确认 `(f 1)` 的值。从这个角度来看，我们需要更细致地记录与过程相关的信息。这些更细致的信息叫做语言的*类型结构* (*type structure*)。

我们的语言将有一种非常简单的类型结构。现在，考虑 LETREC 中的表达值。这些值只包含单参数过程，但处理中的多参数过程也很明了：只需做些额外工作，没有任何新思想。

类型语法

Type ::= *int*
`int-type ()`

Type ::= *bool*
`bool-type ()`

Type ::= `(`*Type* `->` *Type*`)`
`proc-type (arg-type result-type)`

要理解这个系统如何工作，让我们来看些例子。

值及其类型的例子

`3` 的值类型为 `int`。

`-(33,22)` 的值类型为 `int`。

`zero?(11)` 的值类型为 `bool`。

`proc (x) -(x,11)` 的值类型为 `int -> int`，因为给定一个整数时，它返回一个整数。

`proc (x) let y = -(x,11) in -(x,y)`
的值类型为 `int -> int`，因为给定一个整数时，它返回一个整数。

`proc (x) if x then 11 else 22`
的值类型为 `bool -> int`，因为给定一个布尔值时，它返回一个整数。

`proc (x) if x then 11 else zero?(11)` 在我们的类型系统中没有类型，因为给定一个布尔值时，它既可能返回一个整数，也可能返回一个布尔值，而我们没有描述这种行为的类型。

`proc (x) proc (y) if y then x else 11`
的值类型为 `(int -> (bool -> int))`，因为给定一个布尔值时，它返回一个过程，该过程取一布尔值，返回一整数。

`proc (f) (f 3)` 的值类型为 `((int -> `t`) -> `t`)`，t 是任意类型，因为给定一个类型为 `(int -> `t`)` 的过程，它返回类型为 t 的值。

`proc (f) proc (x) (f (f x))` 的值类型为 `((`t `-> `t`) -> (`t `-> `t`))`，t 是任意类型，因为给定一个类型为 `(`t `-> `t`)` 的过程，它返回另一过程，该过程取一类型为 t 的参数，返回一类型为 t 的值。

我们用下面的定义解释这些例子。

定义 7.1.1 性质“表达值 v 的类型为 t”由对 t 进行归纳得到：

- 当且仅当表达值是一个 `num-val`，其类型为 `int`。
- 当且仅当表达值是一个 `bool-val`，其类型为 `bool`。

- 当且仅当表达值是一个 `proc-val`，且给定类型为 t_1 的参数时，发生如下之一：

  1. 返回值类型为 t_2
  2. 不终止
  3. 发生类型错误之外的错误

  其类型为 $(t_1 \to t_2)$。

有时，我们不说“v 类型为 t”，而说“v 具有类型 t”。

此定义归纳自 t。但是它依赖于上面另行定义的类型错误。

在该系统中，值 v 可以有多个类型。比如，值 `proc (x) x` 类型为 $(t \to t)$，t 是任意类型。有些值可能没有类型，比如 `proc (x) if x then 11 else zero?(11)`。

练习 7.1 [⋆] 下面是一些含有闭包的表达式。想想每个表达式的值。每个值的类型是什么（可能不止一个）？有些值的类型在我们的有类型语言中可能无法描述。

1. `proc (x) -(x,3)`
2. `proc (f) proc (x) -((f x), 1)`
3. `proc (x) x`
4. `proc (x) proc (y) (x y)`
5. `proc (x) (x 3)`
6. `proc (x) (x x)`
7. `proc (x) if x then 88 else 99`
8. `proc (x) proc (y) if x then y else 99`
9. ```
   (proc (p) if p then 88 else 99
    33)
   ```
10. ```
    (proc (p) if p then 88 else 99
     proc (z) z)
    ```

11.
```
proc (f)
  proc (g)
   proc (p)
    proc (x) if (p (f x)) then (g 1) else -((f x),1)
```
12.
```
proc (x)
  proc(p)
   proc (f)
    if (p x) then -(x,1) else (f p)
```
13.
```
proc (f)
  let d = proc (x)
           proc (z) ((f (x x)) z)
  in proc (n) ((f (d d)) n)
```

练习 7.2 [⋆⋆] 根据，有没有表达值恰好有两种类型?

练习 7.3 [⋆⋆] 在语言 LETREC 中，能否判定表达值 val 的类型为 t?

7.2 赋予表达值类型

现在，我们只解决了表达值的类型。为了分析程序，我们要写出过程，预测表达式值的类型。

更准确地说，我们的目标是写出过程 `type-of`。给定一个表达式（名为 exp）和一个将变量映射到某一类型的类型环境 (*type environment*)（名为 $tenv$），它赋给 exp 一个类型 t，且 t 具有性质：

type-of 规范

不论何时求 exp 的值，若环境中所有变量对应值的类型都由 $tenv$ 指定，则发生如下之一：

- 结果类型为 t，
- 求值不终止，或

- 求值因类型错误之外的原因失败。

如果我们可以赋予表达式一个类型，我们说该表达式是*正常类型* (*well-typed*) 的，否则说它是*异常类型* (*ill-typed*) 或*无类型*的。

我们的分析基于以下原则：如果我们能预测表达式中所有子表达式的值类型，就能预测表达式的值类型。

我们用这一想法写出 `type-of` 遵循的一些规则。设 $tenv$ 为*一类型环境*，将各个变量映射到类型。那么我们有：

简单判类规则

$$\texttt{(type-of (const-exp } num\texttt{) } tenv\texttt{) = int}$$

$$\texttt{(type-of (var-exp } num\texttt{) } tenv\texttt{) = } tenv(var)$$

$$\frac{\texttt{(type-of } exp_1\ tenv\texttt{) = int}}{\texttt{(type-of (zero?-exp } exp_1\texttt{) } tenv\texttt{) = bool}}$$

$$\frac{\texttt{(type-of } exp_1\ tenv\texttt{) = int} \quad \texttt{(type-of } exp_2\ tenv\texttt{) = int}}{\texttt{(type-of (diff-exp } exp_1\ exp_2\texttt{) } tenv\texttt{) = int}}$$

$$\frac{\texttt{(type-of } exp_1\ tenv\texttt{) = } t_1 \quad \texttt{(type-of } body\ \texttt{[}var\texttt{=}t_1\texttt{]}tenv\texttt{) = } t_2}{\texttt{(type-of (let-exp } var\ exp_1\ body\texttt{) } tenv\texttt{) = } t_2}$$

$$\frac{\begin{array}{l}\texttt{(type-of } exp_1\ tenv\texttt{) = bool}\\ \quad \texttt{(type-of } exp_2\ tenv\texttt{) = } t\\ \quad \texttt{(type-of } exp_2\ tenv\texttt{) = } t\end{array}}{\texttt{(type-of (if-exp } exp_1\ exp_2\ exp_3\texttt{) } tenv\texttt{) = } t}$$

$$\frac{\texttt{(type-of } rator\ tenv\texttt{) = } t_1 \to t_2 \quad \texttt{(type-of } rand\ tenv\texttt{) = } t_1}{\texttt{(type-of (call-exp } rator\ rand\texttt{) } tenv\texttt{) = } t_2}$$

若我们在适当的环境中求类型为 t 的表达式 exp 的值，我们不仅知道值的类型为 t，而且知道与这个值有关的历史信息。因为求 exp 的值保证是安全的，

我们知道 exp 的值一定是由符合类型 t 的操作符产生的。在第 8 章，我们更细致地思考数据抽象时，这种观点会很有帮助。

过程呢？如果 `proc(`var`)` $body$ 类型为 $t_1 \to t_2$，那么应该用类型为 t_1 的参数调用它。求 $body$ 的值时，绑定到变量 var 的值类型为 t_1。

这意味着如下规则：

$$\frac{\texttt{(type-of}\ body\ \texttt{[}var\texttt{=}t_1\texttt{]}tenv\texttt{)} = t_2}{\texttt{(type-of (proc-exp}\ var\ body\texttt{)}\ tenv\texttt{)} = t_1 \to t_2}$$

这条规则是健壮的：如果 `type-of` 对 $body$ 做出了正确预测，那么它也能对 `(proc-exp` var $body$`)` 做出正确预测。

只有一个问题：如果我们要计算 `proc` 表达式的类型，我们怎么找出绑定变量的类型 t_1？它无处可寻。

要解决这个问题，有两种标准设计：

- 类型检查 (*Type Checking*)：按这种方法，程序员需要指出缺失的绑定变量类型，类型检查器推断其他表达式的类型，检查它们是否一致。
- 类型推导 (*Type Inference*)：按这种方法，类型检查器根据程序中变量的使用，尝试推断 (*infer*) 绑定变量的类型。如果语言设计得当，类型检查器可以推断出大多数甚至所有这样的类型。

我们依次研究它们。

练习 7.4 [⋆] 用本节的规则，像 deriv-tree 那样，写出 `proc (x) x` 和 `proc (x) (x y)` 的类型推导。运用规则，给每个表达式赋予至少两种类型。这些表达式的值类型相同吗？

7.3 CHECKED：带有类型检查的语言

除了要求程序员写出所有绑定变量的类型之外，我们的新语言和 LETREC 相同。对由 `letrec` 绑定的变量，我们还要求程序员指定过程结果的类型。

这里是一些 CHECKED 程序例子。

```
proc (x : int) -(x,1)

letrec
 int double (x : int) = if zero?(x)
                        then 0
                        else -((double -(x,1)), -2)
in double

proc (f : (bool -> int)) proc (n : int) (f zero?(n))
```

double 结果的类型为 int，但 double 本身的类型为 (int -> int)，因为它是一个过程，取一整数，返回一整数。

要定义这种语言的语法，我们改变 proc 和 letrec 表达式的生成式。

对指定绑定变量类型的 proc 表达式，规则变为：

$$\frac{\texttt{(type-of}\ body\ \texttt{[}var\texttt{=}t_{var}\texttt{]}tenv\texttt{)} = t_{res}}{\texttt{(type-of (proc-exp}\ var\ t_{var}\ body\texttt{)}\ tenv\texttt{)} = t_{var} \to t_{res}}$$

letrec 呢？典型的 letrec 如下：

letrec
  t_{res} p (*var* : t_{var}) = $e_{proc\text{-}body}$
in $e_{letrec\text{-}body}$

该表达式声明一个名为 p 的过程，其形参是类型为 t_{var} 的变量 var，主体为 $e_{proc\text{-}body}$。因此，p 的类型应为 $t_{var} \to t_{res}$。

检查 letrec 中的表达式 $e_{proc\text{-}body}$ 和 $e_{letrec\text{-}body}$ 时，类型环境中的所有变量都必须有正确的类型。我们可以用定界规则判断当前作用域属于哪些变量，并由此判断变量对应的类型。

$e_{letrec\text{-}body}$ 在过程名 p 的作用域内。如上所述，p 的类型声明为 $t_{var} \to t_{res}$。因此，检查 $e_{letrec\text{-}body}$ 时的类型环境应为：

$$tenv_{letrec\text{-}body} = [p = (t_{var} \to t_{res})]tenv$$

$e_{proc\text{-}body}$ 呢？$e_{proc\text{-}body}$ 在变量 p 的作用域内，p 的类型为 $t_{var} \to t_{res}$；$e_{proc\text{-}body}$ 也在变量 var 的作用域内，var 的类型为 t_{var}。因此，$e_{proc\text{-}body}$ 的类型环境应为：

$$tenv_{proc\text{-}body} = [var = t_{var}]tenv_{letrec\text{-}body}$$

而且，在这个类型环境中，$e_{proc\text{-}body}$ 的结果类型应为 t_{res}。
把这些写成一条规则，我们有：

$$\frac{\texttt{(type-of }e_{proc\text{-}body}\texttt{ [}var\texttt{=}t_{var}\texttt{][}p\texttt{=(}t_{var}\to t_{res}\texttt{)]}tenv\texttt{) = }t_{res} \qquad \texttt{(type-of }e_{letrec\text{-}body}\texttt{ [}p\texttt{=(}t_{var}\to t_{res}\texttt{)]}tenv\texttt{) = }t}{\texttt{(type-of (letrec-exp}t_{res}p\texttt{(}var : t_{var}\texttt{)=}e_{proc\text{-}body}\texttt{in}e_{letrec\text{-}body}\texttt{)}tenv\texttt{)=}t}$$

现在我们已经写出了所有规则，可以实现语言的类型检查器了。

7.3.1 检查器

我们需要比较类型是否相等。我们用过程 `check-equal-type!` 做比较，它比较两个类型，若二者不等则报错。`check-equal-type!` 的第三个参数是一表达式，指明类型不等的位置。

check-equal-type! : *Type* × *Type* × *Exp* → *Unspecified*

```
(define check-equal-type!
  (lambda (ty1 ty2 exp)
    (if (not (equal? ty1 ty2))
      (report-unequal-types ty1 ty2 exp))))
```

report-unequal-types : *Type* × *Type* × *Exp* → *Unspecified*

```
(define report-unequal-types
  (lambda (ty1 ty2 exp)
    (eopl:error 'check-equal-type!
      "Types didn't match: ~s != ~a in~%~a"
      (type-to-external-form ty1)
      (type-to-external-form ty2)
      exp)))
```

我们不使用 `check-equal-type!` 调用的返回值，因此如同 4.2.2 节中的 `setref` 那样，`check-equal-type!` 的执行只求效果。

过程 `report-unequal-types` 用 `type-to-external-form`，将类型转换为易读的列表。

type-to-external-form : *Type* → *List*

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type))))))
```

现在，我们可以将规则转换为程序，就像处理第 3 章中的解释器那样。结果如•fig-7.3 所示。

$Tenv = Var \to Type$

type-of-program : $Program \to Type$

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1) (type-of exp1 (init-tenv))))))
```

type-of : $Exp \times Tenv \to Type$

```
(define type-of
  (lambda (exp tenv)
    (cases expression exp
      [(type-of num tenv) = int]
      (const-exp (num) (int-type))
      [(type-of var tenv) = tenv(var)]
      (var-exp (var) (apply-tenv tenv var))
      [(type-of e1 tenv) = int  (type-of e2 tenv) = int
       ---------------------------------------------
            (type-of (diff-exp e1 e2) tenv) = int]
      (diff-exp (exp1 exp2)
        (let ((ty1 (type-of exp1 tenv))
              (ty2 (type-of exp2 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (check-equal-type! ty2 (int-type) exp2)
          (int-type)))
      [(type-of e1 tenv) = int
       ---------------------------------
       (type-of (zero?-exp e1) tenv) = bool]
      (zero?-exp (exp1)
        (let ((ty1 (type-of exp1 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (bool-type)))
      [(type-of e1 tenv) = bool
        (type-of e2 tenv) = t
        (type-of e3 tenv) = t
       ---------------------------------
       (type-of (if-exp e1 e2 e3) tenv) = t]
      (if-exp (exp1 exp2 exp3)
        (let ((ty1 (type-of exp1 tenv))
              (ty2 (type-of exp2 tenv))
              (ty3 (type-of exp3 tenv)))
          (check-equal-type! ty1 (bool-type) exp1)
          (check-equal-type! ty2 ty3 exp)
          ty2)))
```

图 7.1 CHECKED 的 `type-of`

$$\frac{\texttt{(type-of } body\ \texttt{[}var{=}t_1\texttt{]}tenv\texttt{)} = t_2 \quad \texttt{(type-of } e_1\ tenv\texttt{)} = t_1}{\texttt{(type-of (let-exp } var\ e_1\ body\texttt{) } tenv\texttt{)} = t_2}$$

```
(let-exp (var exp1 body)
  (let ((exp1-type (type-of exp1 tenv)))
    (type-of body
      (extend-tenv var exp1-type tenv))))
```

$$\frac{\texttt{(type-of } body\ \texttt{[}var{=}t_{var}\texttt{]}tenv\texttt{)} = t_res}{\texttt{(type-of (proc-exp } var\ t_{var}\ body\texttt{) } tenv\texttt{)} = \texttt{(}t_{var} \to t_{res}\texttt{)}}$$

```
(proc-exp (var var-type body)
  (let ((result-type
          (type-of body
            (extend-tenv var var-type tenv))))
    (proc-type var-type result-type)))
```

$$\frac{\texttt{(type-of } rator\ tenv\texttt{)} = \texttt{(}t_1 \to t_2\texttt{)} \quad \texttt{(type-of } rand\ tenv\texttt{)} = t_1}{\texttt{(type-of (call-exp } rator\ rand\texttt{) } tenv\texttt{)} = t_2}$$

```
(call-exp (rator rand)
  (let ((rator-type (type-of rator tenv))
        (rand-type (type-of rand tenv)))
    (cases type rator-type
      (proc-type (arg-type result-type)
        (begin
          (check-equal-type! arg-type rand-type rand)
          result-type))
      (else
        (report-rator-not-a-proc-type
          rator-type rator)))))
```

图 7.2 CHECKED 的 `type-of`，续

$$\frac{\texttt{(type-of } e_{\textit{proc-body}} \texttt{ [}var\texttt{=}t_{var}\texttt{][}p \texttt{ =(}t_{var} \to t_{res}\texttt{)]}tenv\texttt{) = } t_{res} \qquad \texttt{(type-of } e_{\textit{letrec-body}} \texttt{ [}p \texttt{ = (}t_{var} \to t_{res}\texttt{)]}tenv\texttt{) = } t}{\texttt{(type-of letrec } t_{res}\ p\ \texttt{(}var\texttt{:}t_{var}\texttt{) =}\ e_{\textit{proc-body}}\ \texttt{in}\ e_{\textit{letrec-body}}\ tenv\texttt{) =}\ t}$$

```
(letrec-exp (p-result-type p-name b-var b-var-type
               p-body letrec-body)
  (let ((tenv-for-letrec-body
          (extend-tenv p-name
            (proc-type b-var-type p-result-type)
            tenv)))
    (let ((p-body-type
            (type-of p-body
              (extend-tenv b-var b-var-type
                tenv-for-letrec-body))))
      (check-equal-type!
        p-body-type p-result-type p-body)
      (type-of letrec-body tenv-for-letrec-body)))))))
```

图 7.3 CHECKED 的 `type-of`，续

练习 7.5 [⋆⋆] 扩展检查器，处理多声明 `let`、多参数过程、以及多声明 `letrec`。你需要添加形如 t_1 `*` t_2 `*` ... `*` t_n `->` t 的类型来处理多参数过程。

练习 7.6 [⋆] 扩展检查器，处理赋值（4.3 节）。

练习 7.7 [⋆] 修改检查 `if-exp` 的代码，若条件不是布尔值，则不检查其他表达式。给出一个表达式，使新旧两个版本的检查器表现出不同的行为。

练习 7.8 [⋆⋆] 给语言添加类型 `pairof`。比如，当且仅当一个值是序对，且所含值类型为 t_1 和 t_2 时，其类型为 `pairof` t_1 `*` t_2。给语言添加下列生成式：

$$
\begin{aligned}
\textit{Type} ::= & \texttt{pairof } \textit{Type} \texttt{ * } \textit{Type} \\
& \boxed{\texttt{pair-type (ty1 ty2)}} \\
\textit{Expression} ::= & \texttt{pair (}\textit{Expression}\texttt{ , }\textit{Expression}\texttt{)} \\
& \boxed{\texttt{pair-exp (exp1 exp2)}} \\
\textit{Expression} ::= & \texttt{unpair } \textit{Identifier}\ \textit{Identifier}\texttt{ = }\textit{Expression} \\
& \texttt{in } \textit{Expression} \\
& \boxed{\texttt{unpair-exp (var1 var2 exp body)}}
\end{aligned}
$$

`pair` 表达式生成一个序对，`unpair` 表达式（同）将两个变量绑定到表达式的两部分。这些变量的作用域是 `body`。`pair` 和 `unpair` 的判类规则为：

$$
\frac{\begin{array}{c}\texttt{(type-of }e_1\ tenv\texttt{) = }t_1 \\ \texttt{(type-of }e_2\ tenv\texttt{) = }t_2\end{array}}{\texttt{(type-of (pair-exp }e_1\ e_2\texttt{) }tenv\texttt{) = pairof }t_1\texttt{ * }t_2}
$$

$$
\frac{\begin{array}{c}\texttt{(type-of }e_{pair}\ tenv\texttt{) = (pairof }t_1\ t_2\texttt{)} \\ \texttt{(type-of }e_{body}\ \texttt{[}var_1\texttt{=}t_1\texttt{][}var_2\texttt{=}t_2\texttt{]}tenv\texttt{) = }t_{body}\end{array}}{\texttt{(type-of (unpair-exp }var_1\ var_2\ e_1\ e_{body}\texttt{) }tenv\texttt{) = }t_{body}}
$$

扩展 CHECKED，实现这些规则。在 `type-to-external-form` 中，用列表 `(pairof` t_1 t_2`)` 表示序对。

练习 7.9 [⋆⋆] 给语言添加类型 `listof`，其操作与类似。当且仅当值是列表，且所有元素类型均为 t 时，值类型为 `listof` t。用下列生成式扩展语言：

$$\begin{aligned}
\textit{Type} ::= & \texttt{ listof } \textit{Type} \\
& \boxed{\texttt{list-type (ty)}} \\
\textit{Expression} ::= & \texttt{ list (}\textit{Expression}\ \{\texttt{,}\textit{Expression}\}\texttt{\^{}\{*\})} \\
& \boxed{\texttt{list-exp (exp1 exps)}} \\
\textit{Expression} ::= & \texttt{ cons (}\textit{Expression}\ \texttt{ , }\ \textit{Expression}\texttt{)} \\
& \boxed{\texttt{cons-exp (exp1 exp2)}} \\
\textit{Expression} ::= & \texttt{ null? (}\textit{Expression}\texttt{)} \\
& \boxed{\texttt{null-exp (exp1)}} \\
\textit{Expression} ::= & \texttt{ emptylist_}\textit{Type} \\
& \boxed{\texttt{emptylist-exp (ty)}}
\end{aligned}$$

以及四条与类型相关的规则：

$$\frac{\begin{array}{c}\texttt{(type-of }e_1\ \textit{tenv}\texttt{) = }t \\ \texttt{(type-of }e_2\ \textit{tenv}\texttt{) = }t \\ \vdots \\ \texttt{(type-of }e_n\ \textit{tenv}\texttt{) = }t\end{array}}{\texttt{(type-of (list-exp }e_1\ \texttt{(}e_2\ \texttt{...}\ e_n\texttt{))}\ \textit{tenv}\texttt{) = listof }t}$$

$$\frac{\texttt{(type-of } e_1 \ tenv\texttt{) = } t \qquad \texttt{(type-of } e_2 \ tenv\texttt{) = listof } t}{\texttt{(type-of cons(}e_1\texttt{, } e_2\texttt{) } tenv\texttt{) = listof } t}$$

$$\frac{\texttt{(type-of } e_1 \ tenv\texttt{) = listof } t}{\texttt{(type-of null?(}e_1\texttt{) } tenv\texttt{) = bool}}$$

$$\texttt{(type-of emptylist[}t\texttt{] } tenv\texttt{) = listof } t$$

虽然 cons 和 pair 类似，它们的判类规则却完全不同。

为 car 和 cdr 写出类似的规则，扩展检查器，处理这些和上述表达式。用中的小技巧避免与 proc-type-exp 的冲突。这些规则应确保 car 和 cdr 应用于列表，但它们无法保证列表非空。为什么让规则确保列表非空不合理？为什么 emptylist 中的类型参数是必需的？

练习 7.10 [⋆⋆] 扩展检查器，处理 EXPLICIT-REFS。你需要这样做：

- 给类型系统添加类型 refto t，其中，t 是任意类型。这个类型表示引用，指向的位置包含类型为 t 的值。那么，若 e 类型为 t，则 (newref e) 类型为 refto t。
- 给类型系统添加类型 void。seref 的返回值为此类型。对类型为 void 的值，不能进行任何操作，所以 setref 返回什么值都不要紧。这是把类型作为信息隐藏机制的例子。
- 写出 newref、deref 和 setref 的判类规则。
- 在检查器中实现这些规则。

练习 7.11 [⋆⋆] 扩展检查器，处理 MUTABLE-PAIRS。

7.4 INFERRED：带有类型推导的语言

在程序中写出类型虽然有助于设计和文档，但很耗时。另一种设计是让编译器根据变量的使用以及程序员可能给出的信息，推断出所有变量的类型。令人惊讶的是，对设计严谨的语言，编译器总能推断出变量的类型。这种策略叫做*类型推导*。它适用于 LETREC 这样的语言，也适用于比较大型的语言。

我们从语言 CHECKED 入手研究类型推导的实例。然后，我们修改语言，令所有类型表达式成为可选项。我们用标记 `?` 替代缺失的类型表达式。因此，典型的程序看起来像是：

```
letrec
 ? foo (x : ?) = if zero?(x)
                 then 1
                 else -(x, (foo -(x,1)))
in foo
```

每个问号（当然，除了 `zero?` 结尾那个）指出所在之处有一个待推导的类型。

由于类型表达式是可选的，我们可以用类型替代某些 `?`，例如：

```
letrec
 ? even (x : int) = if zero?(x) then 1 else (odd -(x,1))
 bool odd (x : ?) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

要定义这种语法，我们新添一个非终结符，*Optional-type*，并修改 `proc` 和 `letrec` 的生成式，令其用可选类型替代类型。

排除的类型就是需要我们找出的类型。要找出它们，我们遍历抽象语法树，生成类型之间的方程，方程中可能含有这些未知类型。然后，我们求解含有未知类型的方程。

要理解这一流程，我们需要给未知类型起名字。对每个表达式 e 或绑定变量 var，设 t_e 或 t_{var} 表示表达式或绑定变量的类型。

对表达式抽象语法树中的每个节点，类型规则决定了类型之间必须成立的某些方程。对我们的 PROC 语言，这些方程是：

$$\begin{aligned}
\texttt{(diff-exp } e_1 \ e_2\texttt{)} &: t_{e_1} = \texttt{int} \\
&\quad\ t_{e_2} = \texttt{int} \\
&\quad\ t_{\texttt{(diff-exp } e_1 \ e_2\texttt{)}} = \texttt{int} \\
\\
\texttt{(zero?-exp } e_1\texttt{)} &: t_{e_1} = \texttt{int} \\
&\quad\ t_{\texttt{(zero?-exp } e_1\texttt{)}} = \texttt{bool} \\
\\
\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)} &: t_{e_1} = \texttt{bool} \\
&\quad\ t_{e_2} = t_{\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)}} \\
&\quad\ t_{e_3} = t_{\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)}} \\
\\
\texttt{(proc-exp } var \ body\texttt{)} &: t_{\texttt{(proc-exp } var \ body\texttt{)}} = (t_{var} \to t_{body}) \\
\\
\texttt{(call-exp } rator \ rand\texttt{)} &: t_{rator} = (t_{rand} \to t_{\texttt{(call-exp } rator \ rand\texttt{)}})
\end{aligned}$$

- 第一条规则是说，`diff-exp` 的参数和结果类型均为 `int`。
- 第二条规则是说，`zero?-exp` 的参数为 `int`，结果为 `bool`。
- 第三条规则是说，`if` 表达式中的条件类型必须为 `bool`，两个分支的类型必须与整个 `if` 表达式的类型相同。
- 第四条规则是说，`proc` 表达式的类型是一过程，其参数类型为绑定变量的类型，其结果类型为主体的类型。
- 第五条规则是说，在过程调用中，操作符类型必须是一过程，其参数类型必须与操作数相同，其结果类型与整个调用表达式的类型相同。

要推导表达式的类型，我们为所有子表达式和绑定变量分配一个类型变量，给出所有子表达式的约束条件，然后求解得出的方程。要理解这一流程，我们来推导几个示例表达式的类型。

我们从表达式 `proc(f) proc(x) -((f 3),(f x))` 开始。我们首先做一张表，涵盖这个表达式中的所有绑定变量、`proc` 表达式、`if` 表达式和过程调用，并给它们分别分配一个变量。

表达式	类型变量
`f`	t_f
`x`	t_x
`proc(f)proc(x)-((f 3),(f x))`	t_0
`proc(x)-((f 3),(f x))`	t_1
`-((f 3),(f x))`	t_2
`(f 3)`	t_3
`(f x)`	t_4

现在，对每个复杂表达式，都可以根据上述规则写出一个类型方程。

表达式	方程
`proc(f)proc(x)-((f 3),(f x))`	1. $t_0 = t_f \to t_1$
`proc(x)-((f 3),(f x))`	2. $t_1 = t_x \to t_2$
`-((f 3),(f x))`	3. $t_3 = \texttt{int}$
	4. $t_4 = \texttt{int}$
	5. $t_2 = \texttt{int}$
`(f 3)`	6. $t_f = \texttt{int} \to t_3$
`(f x)`	7. $t_f = t_x \to t_4$

- 方程 1 是说，整个表达式生成一个过程，其参数类型为 t_f，结果类型与 `proc(x)-((f 3),(f x))` 相同。
- 方程 2 是说，`proc(x)-((f 3),(f x))` 产生一过程，其参数类型为 t_x，结果类型与 `-((f 3),(f x))` 相同。
- 方程 3-5 是说，减法操作 `-((f 3),(f x))` 的参数和结果都是整数。
- 方程 6 是说，`f` 期望的参数类型为 `int`，返回值类型与 `(f 3)` 相同。
- 类似地，方程 7 是说，`f` 期望的参数类型与 `x` 相同，返回值类型与 `(f x)` 相同。

只要满足如下方程，t_f、t_x、t_0、t_1、t_2、t_3 和 t_4 的解可以是任意值：

$t_0 = t_f \to t_1$
$t_1 = t_x \to t_2$
$t_3 = \texttt{int}$
$t_4 = \texttt{int}$
$t_2 = \texttt{int}$
$t_f = \texttt{int} \to t_3$
$t_f = t_x \to t_4$

我们的目标是找出变量的值，使所有方程成立。我们可以把这样的解表示为一组方程，方程的左边都是变量。我们称这组方程为一组代换式 (*substitution*)，称代换式方程左边的变量绑定 (*bound*) 于代换式。

我们可以按部就班地求解这些方程。这一过程叫做合一 (*unification*)。

我们把计算分为两种状态，一种是待求解的方程，一种是已发现的代换式。最开始，所有方程都待求解，没有一个代换式。

方程	代换式
$t_0 = t_f \to t_1$	
$t_1 = t_x \to t_2$	
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \to t_3$	
$t_f = t_x \to t_4$	

我们依次考虑每个方程。如果方程左边是一个变量，我们将其移到代换式组中。

方程	代换式
$t_1 = t_x \to t_2$	$t_0 = t_f \to t_1$
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \to t_3$	
$t_f = t_x \to t_4$	

但是，这样做可能会改变代换式组。例如，下一个方程给出了 t_1 的值。代换式 t_0 右边的值包含 t_1，我们要在其中使用这一信息。所以，我们把代换式右

边出现的每个 t_1 换掉。那么，我们有：

方程	代换式
$t_3 = \texttt{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_4 = \texttt{int}$	$t_1 = t_x \to t_2$
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \to t_3$	
$t_f = t_x \to t_4$	

如果方程右边是一变量，我们调换两侧，然后仍照上面操作。我们可以按照这种方式，继续处理下面的的三个方程。

方程	代换式
$t_4 = \texttt{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_2 = \texttt{int}$	$t_1 = t_x \to t_2$
$t_f = \texttt{int} \to t_3$	$t_3 = \texttt{int}$
$t_f = t_x \to t_4$	

方程	代换式
$t_2 = \texttt{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_f = \texttt{int} \to t_3$	$t_1 = t_x \to t_2$
$t_f = t_x \to t_4$	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$

方程	代换式
$t_f = \texttt{int} \to t_3$	$t_0 = t_f \to (t_x \to \texttt{int})$
$t_f = t_x \to t_4$	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

现在，下一个要处理的方程含有 t_3，已经在代换式组中绑定到 `int`。所以，我们用 `int` 替换方程中的 t_3。方程中的其他类型变量也这样处理。我们称之为对方程*应用* (*apply*) 代换式。

方程	代换式
$t_f = \texttt{int} \to \texttt{int}$	$t_0 = t_f \to (t_x \to \texttt{int})$
$t_f = t_x \to t_4$	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

我们把得到的方程移入代换式组中，并更新需要更新的代换式。

方程	代换式
$t_f = t_x \to t_4$	$t_0 = (\texttt{int} \to \texttt{int}) \to (t_x \to \texttt{int})$
	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$

下一个方程，$t_f = t_x \to t_4$，包含 t_f 和 t_4，均已绑定于代换式，所以我们对该方程应用代换式，得:

方程	代换式
$\texttt{int} \to \texttt{int} = t_x \to \texttt{int}$	$t_0 = (\texttt{int} \to \texttt{int}) \to (t_x \to \texttt{int})$
	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$

如果方程两边都不是变量，我们可以将其化简，得到两个方程:

方程	代换式
$\texttt{int} = t_x$	$t_0 = (\texttt{int} \to \texttt{int}) \to (t_x \to \texttt{int})$
$\texttt{int} = \texttt{int}$	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$

我们还是照常处理：像之前那样，对调第一个方程的两侧，加入代换式组，更新代换式组。

方程	代换式
$\texttt{int} = \texttt{int}$	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

最后一个方程 $\texttt{int} = \texttt{int}$ 总是成立，所以可以丢弃。

方程	代换式
	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

没有方程了，所以我们已完成。从这个计算，我们得出结论：原表达式 `proc (f) proc (x) -((f 3),(f x))` 的类型应为：

$$((\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}))$$

这是合理的：`f` 的第一个参数必须是一个 `int`，因为它接受 `3` 做参数。它必须生成一个 `int`，因为它的值用作减法操作的参数。`x` 也必须是一个 `int`，因为它也用作 `f` 的参数。

我们再看另一个例子：`proc (f) (f 11)`。我们仍从分配类型变量开始。

表达式	类型变量
`f`	t_f
`proc (f) (f 11)`	t_0
`(f 11)`	t_1

接下来我们写出方程：

表达式	方程
`proc(f)(f 11)`	$t_0 = t_f \to t_1$
`(f 11)`	$t_f = \texttt{int} \to t_1$

然后求解：

方程	代换式
$t_0 = t_f \to t_1$	
$t_f = \texttt{int} \to t_1$	

方程	代换式
$t_f = \texttt{int} \to t_1$	$t_0 = t_f \to t_1$

方程	代换式
	$t_0 = (\texttt{int} \to t_1) \to t_1$
	$t_f = \texttt{int} \to t_1$

这意味着可以给 `proc (f) (f 11)` 赋予类型 $(\texttt{int} \to t_1) \to t_1$，其中 t_1 是任何类型。这也是合理的：我们可以推出 `f` 必须取一 `int` 参数，但对 `f` 结果的类型一无所知。而且，对任何 t_1，这个代码都切实可行，只要 `f` 取一 `int` 参数，返回一类型为 t_1 的值。我们称 t_1 对它是多态 (*polymorphic*) 的。

再来看一个例子。考虑 `if x then -(x,1) else 0`。我们还是给每个不是常数的子表达式分配一个类型变量。

表达式	类型变量
`x`	t_x
`if x then -(x,1) else 0`	t_0
`-(x,1)`	t_1

然后给出方程：

表达式	方程
`if x then -(x,1) else 0`	$t_x = \texttt{bool}$
	$t_1 = t_0$
	$\texttt{int} = t_0$
`-(x,1)`	$t_x = \texttt{int}$
	$t_1 = \texttt{int}$

像之前那样处理这些方程，我们有：

方程	代换式
$t_x = \texttt{bool}$	
$t_1 = t_0$	
$\texttt{int} = t_0$	
$t_x = \texttt{int}$	
$t_1 = \texttt{int}$	

方程	代换式
$t_1 = t_0$	$t_x = \texttt{bool}$
$\texttt{int} = t_0$	
$t_x = \texttt{int}$	
$t_1 = \texttt{int}$	

方程	代换式
$\texttt{int} = t_0$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

方程	代换式
$t_0 = \texttt{int}$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

方程	代换式
$t_x = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = t_0$
	$t_0 = \texttt{int}$

由于 t_x 已经绑定于代换式组，我们对下一方程应用代换，得：

方程	代换式
$\texttt{bool} = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = t_0$
	$t_0 = \texttt{int}$

怎么回事？从这些方程，我们推出 `bool = int`。所以在这些方程中的解中，均有 `bool = int`。但 `bool` 和 `int` 不可能相等。因此，这些方程无解，也

就无法赋予这个表达式类型。这是合理的，因为表达式 `if x then -(x,1) else 0` 中，`x` 同时用作布尔值和整数值，而在我们的类型系统中，这是不允许的。

再来看一个例子。考虑 `proc (f) zero?((f f))`。我们仍像之前那样处理。

表达式	类型变量
`proc (f) zero?((f f))`	t_0
`f`	t_f
`zero?((f f))`	t_1
`(f f)`	t_2

表达式	方程
`proc (f) zero?((f f))`	$t_0 = t_f \to t_1$
`zero?((f f))`	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$
`(f f)`	$t_f = t_f \to t_2$

然后，我们仍像之前那样求解：

方程	代换式
$t_0 = t_f \to t_1$	
$t_1 = \texttt{bool}$	
$t_2 = \texttt{int}$	
$t_f = t_f \to t_2$	

方程	代换式
$t_1 = \texttt{bool}$	$t_0 = t_f \to t_1$
$t_2 = \texttt{int}$	
$t_f = t_f \to t_2$	

方程	代换式
$t_2 = \texttt{int}$	$t_0 = t_f \to \texttt{bool}$
$t_f = t_f \to t_2$	$t_1 = \texttt{bool}$

方程	代换式
$t_f = t_f \to t_2$	$t_0 = t_f \to \texttt{bool}$
	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$

方程	代换式
$t_f = t_f \to \texttt{int}$	$t_0 = t_f \to \texttt{bool}$
	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$

问题来了。我们推导出 $t_f = t_f \to \texttt{int}$。但没有一种类型具有这种性质，因为这个方程的右边总是比左边大：如果 t_f 的语法树包含 k 个节点，那么方程右边总是包含 $k+2$ 个节点。

所以，如果我们推导的方程形如 $tv = t$，且类型变量 tv 出现在类型 t 中，我们只能得出结论：原方程无解。这个附加条件叫做*验存* (*occurrence check*)。

这个条件也意味着我们生成的代换式应满足如下不变式：

无存不变式

代换式中绑定的变量不应出现在任何代换式的右边。

我们解方程的代码极度依赖这个不变式。

练习 7.12 [⋆] 用本节的方法，推导中每个表达式的类型，或者判定表达式没有类型。就像本节的其他练习那样，假设每个绑定变量都有对应的 `?`。

练习 7.13 [⋆] 写出 `let` 表达式的类型推导规则。用你的规则，推导下列各表达式的类型，或者判定表达式无类型。

1. `let x = 4 in (x 3)`
2. `let f = proc (z) z in proc (x) -((f x), 1)`
3. `let p = zero?(1) in if p then 88 else 99`
4. `let p = proc (z) z in if p then 88 else 99`

练习 7.14 [⋆] 下面的表达式有何问题?

```
letrec
 ? even(odd : ?) =
    proc (x : ?)
     if zero?(x) then 1 else (odd -(x,1))
in letrec
    ? odd(x : bool) =
       if zero?(x) then 0 else ((even odd) -(x,1))
    in (odd 13)
```

练习 7.15 [⋆⋆] 写出 letrec 表达式的类型推导规则。你的规则应能处理多声明的 letrec。用你的规则推导下列每个表达式的类型，或者判定表达式无类型。

1. ```
   letrec ? f (x : ?)
            = if zero?(x) then 0 else -((f -(x,1)), -2)
   in f
   ```

2. ```
   letrec ? even (x : ?)
              = if zero?(x) then 1 else (odd -(x,1))
            ? odd (x : ?)
              = if zero?(x) then 0 else (even -(x,1))
   in (odd 13)
   ```

3. ```
   letrec ? even (odd : ?)
              = proc (x) if zero?(x)
                         then 1
                         else (odd -(x,1))
   in letrec ? odd (x : ?) =
                 if zero?(x)
                 then 0
                 else ((even odd) -(x,1))
      in (odd 13)
   ```

练习 7.16 [⋆⋆⋆] 修改 INFERRED 的语法，排除缺失类型，不再用 ? 做标记。
   ```

7.4.1 代换式

我们按自底向上的方式实现。我们首先来考虑代换式。

我们将类型变量表示为数据类型 `type` 的新变体。这里用到的技术和 3.7 节中处理词法地址的相同。我们给语法添加生成式：

$$\textit{Type} ::= \texttt{\%tvar-type}\ \textit{Number}$$
```
tvar-type (serial-number)
```

我们把这些扩展后的类型称为*类型表达式* (*type expression*)。类型表达式的基本操作是用类型代换类型变量，由 `apply-one-subst` 定义。`(apply-one-subst` t_0 tv t_1`)` 将 t_0 中出现的每个 tv 代换为 t_1，返回代换后的表达式。有时，这写作 $t_0\texttt{[}tv = t_1\texttt{]}$。

apply-one-subst : $\textit{Type} \times \textit{Tvar} \times \textit{Type} \to \textit{Type}$
```
(define apply-one-subst
  (lambda (ty0 tvar ty1)
    (cases type ty0
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (arg-type result-type)
        (proc-type
          (apply-one-subst arg-type tvar ty1)
          (apply-one-subst result-type tvar ty1)))
      (tvar-type (sn)
        (if (equal? ty0 tvar) ty1 ty0)))))
```

这个过程用来代换单个类型变量。它不能像上节中描述的那样处理所有代换。

代换式组是一个方程列表，方程两边分别为类型变量和类型。该列表也可视为类型变量到类型的函数。当且仅当类型变量出现于代换式组中某个方程的左侧时，我们说该变量*绑定*于代换式。

我们用序对 `(`*类型变量*`.` *类型*`)` 的列表表示代换式组。代换式组的必要观测器是 `apply-subst-to-type`。它遍历类型 t，把每个类型变量替换为代换

式组 σ 中的绑定。如果一个变量未绑定于代换式，那么保持不变。我们用 $t\sigma$ 表示得到的类型。

这一实现用 Scheme 过程 `assoc` 在代换式组中查找类型变量。若给定类型是列表中某个序对的首项，`assoc` 返回对应的（类型变量，类型）序对，否则返回 `#f`。我们将它写出来：

apply-subst-to-type : $Type \times Subst \to Type$

```
(define apply-subst-to-type
  (lambda (ty subst)
    (cases type ty
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (t1 t2)
        (proc-type
          (apply-subst-to-type t1 subst)
          (apply-subst-to-type t2 subst)))
      (tvar-type (sn)
        (let ((tmp (assoc ty subst)))
          (if tmp
            (cdr tmp)
            ty))))))
```

代换式组的构造器有 `empty-subst` 和 `extend-subst`。`(empty-subst)` 生成空代换式组的表示。`(extend-subst` σ tv t`)` 取一代换式组 σ，像上节那样给它添加方程 $tv = t$。这个操作分两步：首先把代换式组中所有方程右边的 tv 替换为 t，然后把方程 $tv = t$ 添加到列表中。用公式表示为：

$$\begin{pmatrix} tv_1 = t_1 \\ \vdots \\ tv_n = t_n \end{pmatrix} [tv = t] = \begin{pmatrix} tv = t \\ tv_1 = t_1[tv = t] \\ \vdots \\ tv_n = t_n[tv = t] \end{pmatrix}$$

该定义具有如下性质：对任意类型 t，

$$(t\sigma)[tv = t'] = t(\sigma[tv = t'])$$

`extend-subst` 的实现依照上式。它把 σ_0 所有绑定中的 t_0 代换为 tv_0。

empty-subst : () → *Subst*
```
(define empty-subst (lambda () '()))
```

extend-subst : *Subst* × *Tvar* × *Type* → *Subst*
用法: `tvar` 尚未绑定于 `subst`。
```
(define extend-subst
  (lambda (subst tvar ty)
    (cons
      (cons tvar ty)
      (map
        (lambda (p)
          (let ((oldlhs (car p))
                (oldrhs (cdr p)))
            (cons
              oldlhs
              (apply-one-subst oldrhs tvar ty))))
        subst))))
```

这一实现保持无存不变式，但既不依赖它，也不强制它。那是下一节中合一器的工作。

练习 7.17 [⋆⋆] 在我们的实现中，当 σ 很大时，`extend-subst` 要做大量工作。实现另一种表示，则 `extend-subst` 变成：

```
(define extend-subst
  (lambda (subst tvar ty)
    (cons (cons tvar ty) subst)))
```

其余工作移至 `apply-subst-to-type`，而性质 $t(\sigma[tv = t']) = (t\sigma)[tv = t']$ 仍然满足。这样定义 `extend-subst` 还需要无存不变式吗?

练习 7.18 [⋆⋆] 修改前一道练习中的实现,则对任意类型变量,`apply-subst-to-type` 最多只需计算一次代换。

7.4.2 合一器

合一器的主要过程是 `unifier`。合一器执行上述推导流程中的这一的步骤：取两个类型 `t_1` 和 `t_2`，满足无存不变式的代换式组 σ，以及表达式 `exp`,

将 $t_1 = t_2$ 添加到 σ，返回得到的代换式组。这是合并 $t_1\sigma$ 和 $t_2\sigma$ 后所得的最小 σ 扩展。这组代换式仍满足无存不变式。若添加 $t_1 = t_2$ 导致矛盾，或者违反了无存不变式，那么合一器报错，指出错误所在的表达式 `exp`。这通常是得出方程 $t_1 = t_2$ 的表达式。

这个算法用 `cases` 来写十分不便，所以我们改用类型的谓词和提取器。算法如所示，其流程如下：

- 首先，像上面那样，我们对类型 t_1 和 t_2 分别应用代换式。
- 如果结果类型相同，我们立即返回。这一步对应上面的删除简单方程。
- 如果 `ty1` 为未知类型，那么无存不变式告诉我们，它未绑定于代换式。由于它未绑定，我们尝试把 $t_1 = t_2$ 添加到代换式组。但我们要验存，以保证无存不变式成立。当且仅当类型变量 tv 不在 t 中时，调用 `(no-occurrence?` tv t`)` 返回 `#t` ()。
- 如果 t_2 为未知类型，则对调 t_1 和 t_2，也照这样处理。
- 如果 t_1 和 t_2 都不是类型变量，那么我们再做进一步分析。

  如果它们都是 `proc` 类型，那么我们化简方程，在两个参数类型之间建立方程，得到一组代换式，然后用这组代换式在结果类型之间建立方程。

  否则，t_1 和 t_2 中一个是 `int`，另一个是 `bool`，或一个是 `proc`，另一个是 `int` 或 `bool`。不管是哪种情况，方程都无解，引发报错。

从另一种角度来思考这些有助于理解。代换式组是一个*存储器*，未知类型是指向存储器中某位置的*引用*。`unifier` 把 `ty1 = ty2` 添加到存储器中，得到一个新的存储器。

最后，我们必须验存。直接递归处理类型即可，如所示。

练习 7.19 [⋆] 我们说：“如果 `ty1` 为未知类型，那么无存不变式告诉我们，它未绑定于代换式。”详细解释为什么如此。

练习 7.20 [⋆⋆] 修改合一器，不是对合一器的实参，而是只对类型变量调用 `apply-subst-to-type`。

unifier : $Type \times Type \times Subst \times Exp \to Subst$

```
(define unifier
  (lambda (ty1 ty2 subst exp)
    (let ((ty1 (apply-subst-to-type ty1 subst))
          (ty2 (apply-subst-to-type ty2 subst)))
      (cond
        ((equal? ty1 ty2) subst)
        ((tvar-type? ty1)
          (if (no-occurrence? ty1 ty2)
            (extend-subst subst ty1 ty2)
            (report-no-occurrence-violation ty1 ty2 exp)))
        ((tvar-type? ty2)
          (if (no-occurrence? ty2 ty1)
            (extend-subst subst ty2 ty1)
            (report-no-occurrence-violation ty2 ty1 exp)))
        ((and (proc-type? ty1) (proc-type? ty2))
          (let ((subst (unifier
                         (proc-type->arg-type ty1)
                         (proc-type->arg-type ty2)
                         subst exp)))
            (let ((subst (unifier
                           (proc-type->result-type ty1)
                           (proc-type->result-type ty2)
                           subst exp)))
              subst)))
        (else (report-unification-failure ty1 ty2 exp))))))
```

图 7.4 合一器

练习 7.21 [⋆⋆] 我们说代换式组就像存储器。用中的代换式组表示实现合一器，用全局 Scheme 变量记录代换式组，就像和 fig-4.2 那样。

练习 7.22 [⋆⋆] 优化前一道练习的实现，在常数时间内获取类型变量的绑定。

no-occurrence? : $Tvar \times Type \to Bool$

```
(define no-occurrence?
  (lambda (tvar ty)
    (cases type ty
      (int-type () #t)
      (bool-type () #t)
      (proc-type (arg-type result-type)
        (and
          (no-occurrence? tvar arg-type)
          (no-occurrence? tvar result-type)))
      (tvar-type (serial-number) (not (equal? tvar ty))))))
```

图 7.5 验存

7.4.3 找出表达式的类型

我们用 `otype->type` 为每个 `?` 定义一个新类型变量，把可选类型转换为未知类型。

otype->type : *OptionalType* → *Type*
```
(define otype->type
  (lambda (otype)
    (cases optional-type otype
      (no-type () (fresh-tvar-type))
      (a-type (ty) ty))))
```

fresh-tvar-type : () → *Type*
```
(define fresh-tvar-type
  (let ((sn 0))
    (lambda ()
      (set! sn (+ sn 1))
      (tvar-type sn))))
```

把类型转换为外在表示时，我们用包含序号的符号表示类型变量。

type-to-external-form : *Type* → *List*
```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type)))
      (tvar-type (serial-number)
        (string->symbol
          (string-append
            "ty"
            (number->string serial-number)))))))
```

现在我们可以写 `type-of` 了。它取一表达式，一个将程序变量映射到类型表达式的类型环境，和一个满足无存不变式的代换式组，返回一个类型和满足无存不变式的新代换式组。

类型环境将各类型表达式与程序变量对应起来。代换式组解释了类型表达式中每个类型变量的含义。我们把代换式组比作*存储器*，把类型变量比作存储器*引用*。因此，`type-of` 返回两个值：一个类型表达式，和一个解释表达式中类型变量的代换式组。像那样，我们在实现时新定义一种包含两个值的数据类型，用作返回值。

`type-of` 的定义如•fig-7.8 所示。对每个表达式，我们递归处理子表达式，一路传递代换式组参数中现有的解。然后，我们根据规范，为当前表达式建立方程，调用 `unifier`，在代换式组中记录这些。

Answer = *Type* → *Subst*

```
(define-datatype answer answer?
  (an-answer
    (ty type?)
    (subst substitution?)))
```

type-of-program : *Program* → *Type*

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cases answer (type-of exp1
                               (init-tenv) (empty-subst))
          (an-answer (ty subst)
            (apply-subst-to-type ty subst)))))))
```

type-of : *Exp* × *Tenv* × *Subst* → *Answer*

```
(define type-of
  (lambda (exp tenv subst)
    (cases expression exp
      (const-exp (num) (an-answer (int-type) subst))

      (zero?-exp (exp1)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst1)
            (let ((subst2
                    (unifier ty1 (int-type) subst1 exp)))
              (an-answer (bool-type) subst2)))))))
```

图 7.6 INFERRED 的 `type-of`，第 1 部分

```
(diff-exp (exp1 exp2)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (ty1 subst1)
      (let ((subst1
              (unifier ty1 (int-type) subst1 exp1)))
        (cases answer (type-of exp2 tenv subst1)
          (an-answer (ty2 subst2)
            (let ((subst2
                    (unifier ty2 (int-type)
                      subst2 exp2)))
              (an-answer (int-type) subst2)))))))))

(if-exp (exp1 exp2 exp3)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (ty1 subst)
      (let ((subst
              (unifier ty1 (bool-type) subst exp1)))
        (cases answer (type-of exp2 tenv subst)
          (an-answer (ty2 subst)
            (cases answer (type-of exp3 tenv subst)
              (an-answer (ty3 subst)
                (let ((subst
                        (unifier ty2 ty3 subst exp)))
                  (an-answer ty2 subst)))))))))))
(var-exp (var)
  (an-answer (apply-tenv tenv var) subst))
(let-exp (var exp1 body)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (exp1-type subst)
      (type-of body
```

```
(extend-tenv var exp1-type tenv)
subst))))
```

(proc-exp *var body*) : $t_{\texttt{(proc-exp } var\ body\texttt{)}}$ = ($tvar \rightarrow t_{body}$)

```
(proc-exp (var otype body)
  (let ((var-type (otype->type otype)))
    (cases answer (type-of body
                           (extend-tenv var var-type tenv)
                           subst)
      (an-answer (body-type subst)
        (an-answer
          (proc-type var-type body-type)
          subst)))))
```

(call-exp *rator rand*) : t_{rator} = ($t_{rand} \rightarrow t_{\texttt{(call-exp } rator\ rand\texttt{)}}$)

```
(call-exp (rator rand)
  (let ((result-type (fresh-tvar-type)))
    (cases answer (type-of rator tenv subst)
      (an-answer (rator-type subst)
        (cases answer (type-of rand tenv subst)
          (an-answer (rand-type subst)
            (let ((subst
                    (unifier
                      rator-type
                      (proc-type
                        rand-type result-type)
                      subst
                      exp)))
              (an-answer result-type subst))))))))
```

图 7.7 INFERRED 的 `type-of`，第 3 部分

$aligned at -1letrec $t_{proc\text{-}result}$ p (*var* : t_{var}) = $e_{proc\text{-}body}$ in $e_{letrec\text{-}body}$: $t_p = t_{var} \to t_{e_{pro}}$

```
(letrec-exp (p-result-otype p-name b-var b-var-otype
               p-body letrec-body)
  (let ((p-result-type (otype->type p-result-otype))
        (p-var-type (otype->type b-var-otype)))
    (let ((tenv-for-letrec-body
           (extend-tenv p-name
             (proc-type p-var-type p-result-type)
             tenv)))
      (cases answer (type-of p-body
                      (extend-tenv b-var p-var-type
                        tenv-for-letrec-body)
                      subst)
        (an-answer (p-body-type subst)
          (let ((subst
                  (unifier p-body-type p-result-type
                    subst p-body)))
            (type-of letrec-body
              tenv-for-letrec-body
              subst)))))))))
```

图 7.8 INFERRED 的 `type-of`，第 4 部分

因为多态的缘故，测试推导器比测试之前的解释器稍微麻烦。例如，如果给推导器输入 `proc (x) x`，它给出的外在表示可能是 `(tvar1 -> tvar1)`、`(tvar2 -> tvar2)` 或 `(tvar3 -> tvar3)`，等等。每次调用推导器结果都可能不同，所以我们写测试项时不能直接使用它们，否则就无法比较推导出的类型和正确类型。我们需要接受上述所有可能，但拒绝 `(tvar3 -> tvar4)` 或是 `(int -> tvar17)`。

要比较两种类型的外在表示，我们统一未知类型的名字，遍历每个外在表示，给类型变量重新编号，使之从 `ty1` 开始。然后，我们就能用 `equal?` 比较重新编号的类型（•fig-7.11）。

要逐个命名所有未知变量，我们用 `canonical-subst` 生成代换式组。我们用 `table` 做累加器，即可直接递归。`table` 的长度告诉我们已找出多少个不同的未知类型，我们可以用其长度给“下一个”`ty` 符号编号。这和我们在中使用的 `length` 类似。

TvarTypeSym = 含有数字的符号

A-list = *Listof*(*Pair*(*TvarTypeSym*, *TvarTypeSym*))

equal-up-to-gensyms? : *S-exp* × *S-exp* → *Bool*

```
(define equal-up-to-gensyms?
  (lambda (sexp1 sexp2)
    (equal?
      (apply-subst-to-sexp (canonical-subst sexp1) sexp1)
      (apply-subst-to-sexp (canonical-subst sexp2) sexp2))))
```

canonical-subst : *S-exp* → *A-list*

```
(define canonical-subst
  (lambda (sexp)
    loop : S-exp × A-list → A-list
    (let loop ((sexp sexp) (table '()))
      (cond
        ((null? sexp) table)
        ((tvar-type-sym? sexp)
          (cond
            ((assq sexp table) table)
            (else
              (cons
                (cons sexp (ctr->ty (length table)))
                table))))
        ((pair? sexp)
          (loop (cdr sexp)
            (loop (car sexp) table)))
        (else table)))))
```

图 7.9 `equal-up-to-gensyms?`，第 1 部分

tvar-type-sym? : *Sym* → *Bool*
```
(define tvar-type-sym?
  (lambda (sym)
    (and (symbol? sym)
      (char-numeric? (car (reverse (symbol->list sym)))))))
```

symbol->list : *Sym* → *List*
```
(define symbol->list
  (lambda (x)
    (string->list (symbol->string x))))
```

apply-subst-to-sexp : *A-list* × *S-exp* → *S-exp*
```
(define apply-subst-to-sexp
  (lambda (subst sexp)
    (cond
      ((null? sexp) sexp)
      ((tvar-type-sym? sexp)
        (cdr (assq sexp subst)))
      ((pair? sexp)
        (cons
          (apply-subst-to-sexp subst (car sexp))
          (apply-subst-to-sexp subst (cdr sexp))))
      (else sexp))))
```

ctr->ty : *N* → *Sym*
```
(define ctr->ty
  (lambda (n)
    (string->symbol
      (string-append "tvar" (number->string n)))))
```

图 7.10 `equal-up-to-gensyms?`，第 2 部分

练习 7.23 [⋆⋆] 扩展推导器，像那样处理序对类型。

练习 7.24 [⋆⋆] 扩展推导器，处理多声明 `let`、多参数过程和多声明 `letrec`。

练习 7.25 [⋆⋆] 扩展推导器，像那样处理列表类型。修改语言，用生成式

Expression ::= `emptylist`

代替

Expression ::= `emptylist_`*Type*

提示：考虑用类型变量代替缺失的 *_t*。

练习 7.26 [⋆⋆] 扩展推导器，像那样处理 EXPLICIT-REFS。

练习 7.27 [⋆⋆] 重写推导器，将其分为两步。第一步生成一系列方程，第二步重复调用 `unify` 求解它们。

练习 7.28 [⋆⋆] 我们的推导器虽很有用，却不够强大，不允许程序员定义多态过程，像定义多态原语 `pair` 或 `cons` 那样，适用于多种类型。例如，即使执行是安全的，我们的推导器也会拒绝程序

```
let f = proc (x : ?) x
in if (f zero?(0))
   then (f 11)
   else (f 22)
```

因为 `f` 既是 `(bool -> bool)` 也是 `(int -> int)`。由于本节的推导器至多只能找出 `f` 的一种类型，它将拒绝这段程序。

更实际的例子是这样的程序

```
letrec
 ? map (f : ?) =
    letrec
     ? foo (x : ?) = if null?(x)
```

```
                        then emptylist
                        else cons((f car(x)),
                                  (foo cdr(x)))
     in foo
in letrec
     ? even (y : ?) = if zero?(y)
                        then zero?(0)
                        else if zero?(-(y,1))
                             then zero?(1)
                             else (even -(y,2))
   in pair(((map proc(x : int) -(x,1))
            cons(3,cons(5,emptylist))),
            ((map even)
             cons(3,cons(5,emptylist))))
```

这个表达式用了两次 map，一次产生 int 列表，一次产生 bool 列表。因此，两次使用它需要两个不同的类型。由于本节的推导器至多只能找出 map 的一种类型，它检测到 int 和 bool 冲突，拒绝程序。

避免这个问题的一种方法是只允许 let 引入多态，然后在类型检查时区分 (let-exp var e_1 e_2) 和 (call-exp (proc-exp var e_2) e_1)。

给推导器添加多态绑定，处理表达式 (let-exp var e_1 e_2) 时，把 e_2 中自由出现的每个 var 代换为 e_1。那么，在推导器看来，let 主体中有多个不同的 e_1 副本，它们可以有不同的类型，上述程序就能通过。

练习 7.29 [⋆⋆⋆] 前一道练习指出的类型推导算法会多次分析 e_1，每次对应 e_2 中出现的一个 e_1。实现 Milner 的 W 算法，只需分析 e_1 一次。

练习 7.30 [⋆⋆⋆] 多态和副作用之间的相互作用很微妙。考虑以下文开头的一段程序

```
let p = newref(proc (x : ?) x)
in ...
```

1. 完成这段程序，使之通过推导器的检查，但根据本章开头的定义，求值不安全。

2. 限制 `let` 声明的右边，不允许出现作用于存储器的效果，从而避免这一问题。这叫做*值约束* (*value restriction*)。

8 模块

要构建只有几百行代码的系统，我们介绍的语言特性已非常强大。如果我们要构建更大的系统，有数千行代码，我们就还需要一些别的佐料。

1. 我们需要一种好的方式，将系统分为相对独立的部分，并能编写文档，解释各部分之间的依赖关系。

2. 我们需要一种更好的方式来控制名字的作用域和绑定。词法定界是命名控制的强大工具，但不足以应对更大或分割为多份源代码的程序。

3. 我们需要一种方式强化抽象边界。在第 2 章，我们介绍了抽象数据类型的思想。在类型的实现中，我们可以对值做任意操作，但在实现之外，类型的值只能通过类型接口中的过程创建和操作。我们把这叫做*抽象边界* (*abstraction boundary*)。如果程序遵守这一界限，我们可以改变数据类型的实现。但是，如果某些代码打破了抽象，依赖实现细节，我们就无法任意修改实现而不破坏其他代码。

4. 最后，我们需要一种方式，将这些部分灵活组合，让同一部分可复用于不同地方。

本章，我们介绍*模块* (*module*)，以满足这些需求。具体来说，我们展示了如何用类型系统创建和强化抽象边界。

我们的模块语言中，程序包含一系列*模块定义* (*module definition*)，后跟一个待求值的表达式。每个模块定义把一个名字绑定到一个*模块*。创建的模块

可能是简单模块 (*simple module*)，类似环境，是一组绑定；也可能是模块过程 (*module procedure*)，取一模块，生成另一模块。

每个模块都有一套接口 (*interface*)。作为一组绑定的模块具有简单接口 (*simple interface*)，接口列出模块提供的绑定及其类型。模块过程的接口指定模块的参数接口和返回模块的接口，就像过程的类型指定参数和结果的类型。

这些接口就像类型一样，决定了模块组合的方式。因为求出示例程序的值非常直接，因此我们关注其类型。如前所见，理解这种语言的定界和绑定规则是程序分析和求值的关键。

8.1 简单模块系统

我们的第一种语言名叫 SIMPLE-MODULES，只有简单模块。它没有模块过程，只创建非常简单的抽象边界。几种流行语言使用与之类似的模块系统。

8.1.1 例子

设想一个软件项目中有三名开发者：爱丽丝、鲍伯和查理。爱丽丝、鲍伯和查理正在开发项目中相对独立的几部分。这些开发者散居各地，时区都可能不同。项目的每部分都要实现一套 2.1 节那样的接口，但接口的实现可能涉及大量其他过程。而且，当各部分集成到一起时，开发者需要确保不存在干扰项目其他部分的命名冲突。

要实现这一目标，开发者们需要公布一套接口，列出每个供他人使用的过程名字。模块系统要保证这些名字是公开的，而它们使用的其他名字则是私有的，且不会被项目中的其他代码修改。

我们可以用第 3 章中的定界技术，但这些无法应对更大的工程。相反，我们使用模块系统。开发者们各设计一个模块，包含公开接口和私有实现。他们能看到自己模块的接口和实现，但爱丽丝只能看到他人模块的接口。她的所做所为不会影响其他模块的实现，其他模块的实现也不会影响她的（如所示）。

这里是 SIMPLE-MODULES 的简短例子。

例 8.1

```
module m1
 interface
  [a : int
   b : int
   c : int]
 body
  [a = 33
   x = -(a,1)  % = 32
   b = -(a,x)  % = 1
   c = -(x,b)] % = 31
let a = 10
in -(-(from m1 take a,
       from m1 take b),
     a)
```

类型为 `int`，值为 $((33-1)-10)=22$。

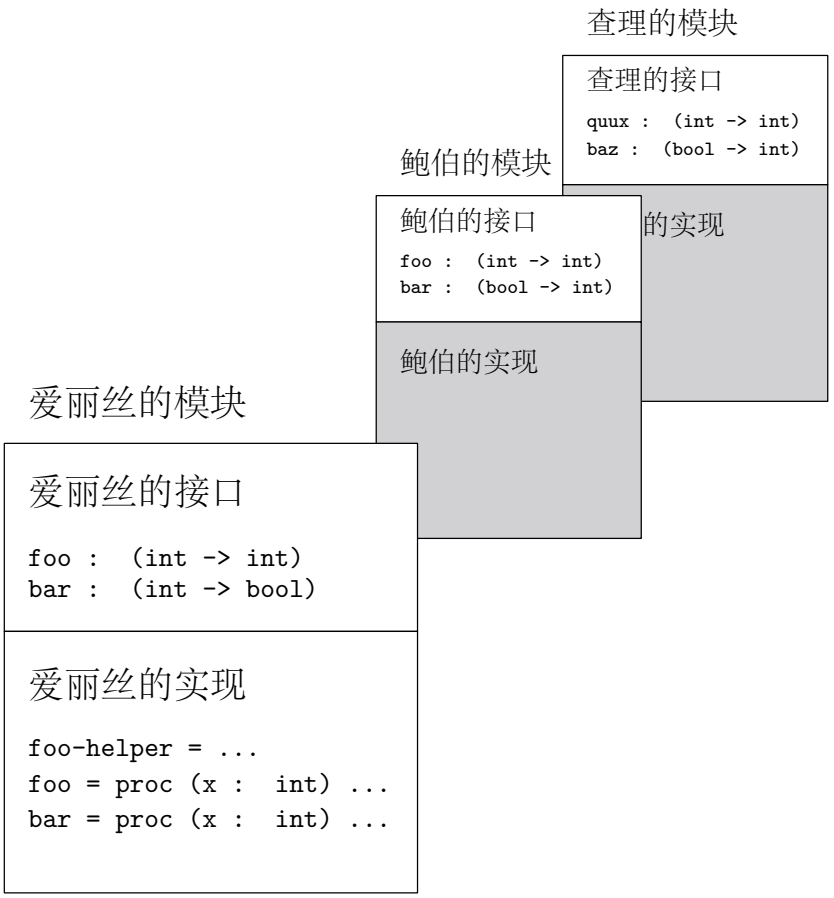


图 8.1 项目中，爱丽丝所见的三个模块

这个程序从名为 `m1` 的模块定义开始。像任何模块一样，它有*接口*和*主体*。主体*实现*接口。接口*声明*变量 `a`、`b` 和 `c`。主体*定义了* `a`、`x`、`b` 和 `c` 的绑定。

当我们求程序的值时，也会求出 `m1` 主体中表达式的值。变量 `from m1 take a`、`from m1 take b` 和 `from m1 take c` 绑定到适当的值，它们的作用域为模块定义之后。由于 `from m1 take x` 未在接口中声明，所以它的作用域不包含模块定义之后。

为了同*简单变量* (*simple variables*) 区别，我们称这些新变量为*受限变量* (*qualified variables*)。在一般的语言中，受限变量可能写作 `m1.a`、`m1:a` 或 `m1::a`。在第 9 章探讨的面向对象语言中，`m1.a` 通常另有含义。

我们说接口*提出* (*offer*)（或称*公布* (*advertise*)，或称*承诺* (*promise*)）三个整型值，主体*提供* (*supply, provide*)（或称*输出* (*export*)）这些值。当模块主体提供的值类型与接口命名变量时公布的类型相符时，主体*满足*其接口。

在主体中，定义具有 `let*` 那样的作用域，所以 `x`、`b` 和 `c` 的定义在 `a` 的作用域内。一些作用域如所示。

本例中，以 `let a = 10` 开头的表达式是*程序主体* (*program body*)。它的值即程序的值。

每个模块都在模块主体和程序其余部分之间建立了抽象边界。模块主体中的表达式在抽象边界*之内*，其他部分在抽象边界*之外*。模块主体也可以提供不在接口中的名字绑定，但那些绑定在程序主体和其他模块中不可见，正如所示。在我们的例子中，程序主体不在 `from m1 take x` 的作用域内。如果我们写 `-(from m1 take a, from m1 take x)`，程序就会类型异常。

例 8.2 程序

```
module m1
 interface
  [u : bool]
 body
  [u = 33]

44
```

类型异常。即使程序的其他部分不使用那些值，模块主体也要将接口中的名字

与适当类型的值关联起来。

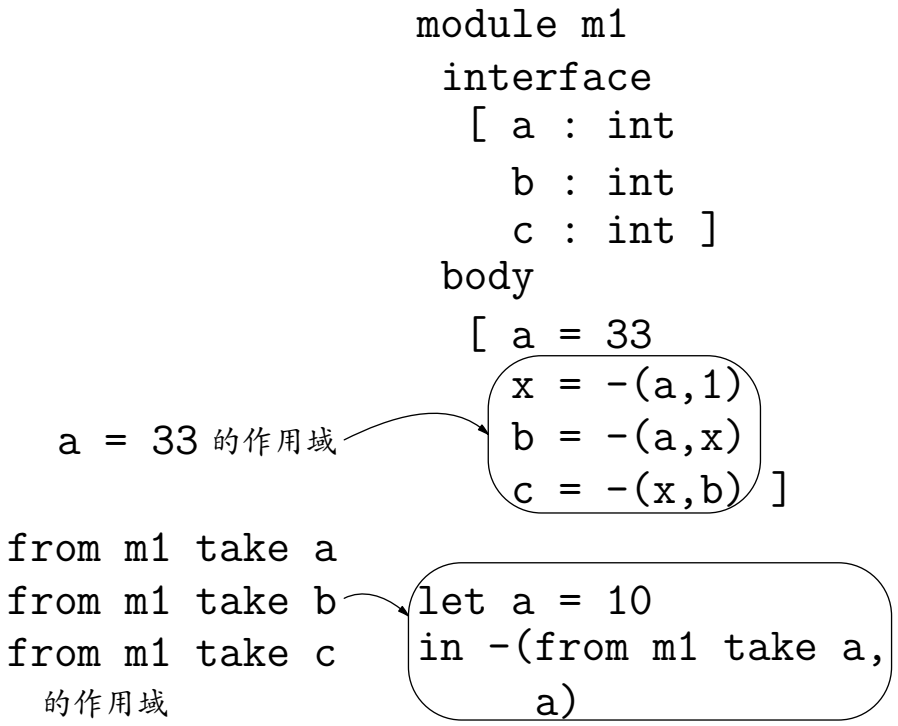


图 8.2 简单模块中的一些作用域

例 8.3 模块主体必须提供接口中声明的所有绑定。例如，

```
module m1
 interface
  [u : int
   v : int]
 body
  [u = 33]

44
```

类型异常，因为 `m1` 的主体没有提供接口中公布的所有值。

例 8.4 为了让实现简单一点，我们的语言要求模块主体按照接口声明的顺序给出各值。因此

```
module m1
 interface
  [u : int
   v : int]
 body
  [v = 33
   u = 44]

from m1 take u
```

类型异常。可以免除这一限制（ex8.8、ex8.17）。

例 8.5 在我们的语言中，模块具有 `let*` 式的作用域（ex3.17）。例如，

```
module m1
 interface
  [u : int]
 body
  [u = 44]

module m2
 interface
  [v : int]
 body
  [v = -(from m1 take u,11)]
```

```
-(from m1 take u, from m2 take v)
```

类型为 int。但如果我们交换定义的顺序，

```
module m2
 interface
  [v : int]
 body
  [v = -(from m1 take u,11)]

module m1
 interface
  [u : int]
 body
  [u = 44]

-(from m1 take u, from m2 take v)
```

则类型异常，因为 m2 主体中使用 from m1 take u 的地方不在后者的作用域内。

8.1.2 实现简单模块系统

语法

SIMPLE-MODULES 的程序包含一串模块定义，然后是一个表达式。

Program ::= {*ModuleDefn*}* *Expression*

```
a-program (m-defs body)
```

模块定义包含名字、接口和主体。

ModuleDefn ::= `module` *Identifier* `interface` *Iface* `body` *ModuleBody*

```
a-module-definition (m-name expected-iface m-body)
```

简单模块的接口包含任意数量的声明。每个声明指定程序中一个变量的类型。我们称之为值声明 (*value declaration*)，因为要声明的变量表示一个值。在后面几节中，我们介绍其他类别的接口和声明。

Iface ::= `[{`*Decl*`}*]`
`simple-iface (decls)`
Decl ::= *Identifier*`:`*Type*
`val-decl (var-name ty)`

模块主体包含任意数量的定义。每个定义将变量和某个表达式的值关联起来。

ModuleBody ::= `[{`*Defn*`}*]`
`defns-module-body (defns)`
Defn ::= *Identifier*`=`*Expression*
`val-defn (var-name exp)`

我们的表达式与 CHECKED（7.3 节）相同，但我们要修改语法，新增一种表达式，以便使用受限变量。

Expression ::= `from`*Identifier*`take`*Identifier*
`qualified-var-exp (m-name var-name)`

解释器

求模块主体的值会得到一个*模块*。在我们的简单模块语言中，模块是一个环境，包含模块输出的所有绑定。我们用数据类型 `typed-module` 表示这些。

```
(define-datatype typed-module typed-module?
  (simple-module
    (bindings environment?)))
```

我们用一种新的绑定在环境中绑定模块名：

```
(define-datatype environment environment?
  (empty-env)
  (extend-env ...as before...)
```

```
  (extend-env-rec ...as before...)
  (extend-env-with-module
    (m-name symbol?)
    (m-val typed-module?)
    (saved-env environment?)))
```

例如，如果我们的程序是

```
module m1
 interface
  [a : int
   b : int
   c : int]
 body
  [a = 33
   b = 44
   c = 55]
module m2
 interface
  [a : int
  b : int]
 body
  [a = 66
   b = 77]
let z = 99
in -(z, -(from m1 take a, from m2 take a))
```

那么声明 z 之后的环境是

```
#(struct:extend-env
   z #(struct:num-val 99)
   #(struct:extend-env-with-module
      m2 #(struct:simple-module
            #(struct:extend-env
               a #(struct:num-val 66)
               #(struct:extend-env
                  b #(struct:num-val 77)
                  #(struct:empty-env))))
      #(struct:extend-env-with-module
         m1 #(struct:simple-module
               #(struct:extend-env
                  a #(struct:num-val 33)
```

```
                    #(struct:extend-env
                       b #(struct:num-val 44)
                       #(struct:extend-env
                          c #(struct:num-val 55)
                          #(struct:empty-env)))))
              #(struct:empty-env))))
```

在这个环境中，`m1` 和 `m2` 各绑定到一个简单模块，二者分别包含一个小环境。

我们用 `lookup-qualified-var-in-env` 求受限变量 `from` m `take` var 引用的值。它在当前环境中查找模块 m，然后在得到的环境中查找 var。

lookup-qualified-var-in-env! : $Sym \times Sym \times Env \to ExpVal$

```
(define lookup-qualified-var-in-env
  (lambda (m-name var-name env)
    (let ((m-val (lookup-module-name-in-env m-name env)))
      (cases typed-module m-val
        (simple-module (bindings)
          (apply-env bindings var-name))))))
```

要求程序的值，我们把所有模块定义加入当前环境中，得到初始环境，然后求程序主体的值。过程 `add-module-defns-to-env` 遍历模块定义，求每个模块定义主体的值，并将得到的模块加入当前环境中，如所示。

最后，要求模块主体的值，我们按照 `let*` 式定界，在适当的环境内求每个表达式的值，得出一环境。过程 `defns-to-env` 生成的环境只包含定义 `defns` 产生的绑定 ()。

value-of-program : $Program \to ExpVal$

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (m-defns body)
        (value-of body
          (add-module-defns-to-env m-defns (empty-env)))))))
```

add-module-defns-to-env : $Listof(Defn) \times Env \to Env$

```
(define add-module-defns-to-env
  (lambda (defns env)
    (if (null? defns)
      env
      (cases module-definition (car defns)
        (a-module-definition (m-name iface m-body)
          (add-module-defns-to-env
            (cdr defns)
            (extend-env-with-module
              m-name
              (value-of-module-body m-body env)
              env)))))))
```

图 **8.3** SIMPLE-MODULES 的解释器，第 1 部分

value-of-module-body : $ModuleBody \times Env \to TypedModule$

```
(define value-of-module-body
  (lambda (m-body env)
    (cases module-body m-body
      (defns-module-body (defns)
        (simple-module
          (defns-to-env defns env))))))
```

defns-to-env : $Listof(Defn) \times Env \to Env$

```
(define defns-to-env
  (lambda (defns env)
    (if (null? defns)
      (empty-env)
      (cases definition (car defns)
        (val-defn (var exp)
          (let ((val (value-of exp env)))
            (let ((new-env (extend-env var val env)))
              (extend-env var val
                (defns-to-env
                  (cdr defns) new-env)))))))))
```

图 8.4 SIMPLE-MODULES 的解释器，第 2 部分

检查器

检查器的工作是确保所有模块主体满足其接口，且所有变量的使用符合其类型。

我们语言的定界规则很简单：模块遵循 `let*` 式定界，依次进入模块输出绑定中的受限变量的作用域。接口告诉我们每个受限变量的类型。声明和定义也都遵循 `let*` 式定界（如）。

就像第 7 章中的检查器那样，我们用类型环境记录与当前作用域内各名字的相关信息。因为我们现在有了模块名，我们要在类型环境中绑定模块名。每个模块名绑定到模块的接口，作为其类型。

```
(define-datatype type-environment type-environment?
  (empty-tenv)
  (extend-tenv ... 同前...)
  (extend-tenv-with-module
    (name symbol?)
    (interface interface?)
    (saved-tenv type-environment?)))
```

要找出受限变量 `from` m `take` var 的类型，我们首先在类型环境中找出 m，然后在得到的接口中查找 var 的类型。

lookup-qualified-var-in-tenv : $Sym \times Sym \times Tenv \to Type$

```
(define lookup-qualified-var-in-tenv
  (lambda (m-name var-name tenv)
    (let ((iface (lookup-module-name-in-tenv tenv m-name)))
      (cases interface iface
        (simple-iface (decls)
          (lookup-variable-name-in-decls var-name decls))))))
```

type-of-program : *Program* → *Type*

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (module-defns body)
        (type-of body
          (add-module-defns-to-tenv module-defns
            (empty-tenv)))))))
```

add-module-defns-to-tenv : *Listof*(*ModuleDefn*) × *Tenv* → *Tenv*

```
(define add-module-defns-to-tenv
  (lambda (defns tenv)
    (if (null? defns)
      tenv
      (cases module-definition (car defns)
        (a-module-definition (m-name expected-iface m-body)
          (let ((actual-iface (interface-of m-body tenv)))
            (if (<:-iface actual-iface expected-iface tenv)
              (let ((new-tenv
                      (extend-tenv-with-module
                        m-name
                        expected-iface
                        tenv)))
                (add-module-defns-to-tenv
                  (cdr defns) new-tenv))
              (report-module-doesnt-satisfy-iface
                m-name expected-iface actual-iface)))))))))
```

图 8.5 SIMPLE-MODULES 的检查器，第 1 部分

就像第 7 章那样，对程序做类型检查的过程类模仿程序的求值，但我们记录的不是值，而是类型。我们用 `type-of-program` 代替 `value-of-program`，用 `add-module-defns-to-tenv` 代替 `add-module-defns-to-env`。过程 `add-module-defns-to-tenv` 用 `<:-iface` 检查各模块主体产生的接口与提出的接口是否相符；如果相符，就将模块加入到类型环境中；否则报错。

模块主体的接口将主体中定义的各个变量与定义中的类型关联起来。例如，如果我们查看第一个例子的主体，

```
[a = 33
 x = -(a,1)
 b = -(a,x)
 c = -(x,b)]
```

可得

```
[a : int
 x : int
 b : int
 c : int]
```

一旦我们建立了一套接口来描述模块主体输出的所有绑定，我们就能将其与模块公布的接口比较。

回忆一下，简单接口包含一个声明列表。过程 `defns-to-decls` 创建这样的列表，调用 `type-of` 找出每个定义的类型。在每一步，它还按 `let*` 定界，扩展局部类型环境（见）。

剩下的只是用 `<:-iface` 比较每个模块的期望类型与实际类型。我们将 `<:` 定义为：若 i_1`<:`i_2，则满足接口 i_1 的任何模块也满足接口 i_2。例如

```
[u : int          [u : int
 v : int    <:     z : int]
 z : int]
```

因为满足接口 `[u : int v : bool z : int]` 的任何模块都提供了接口 `[u : int z : int]` 公布的所有值。

对我们的简单模块语言，`<:-iface` 只需调用 `<:-decls` 比较声明。这些过程取一 `tenv` 参数，简单模块系统不使用它，但 8.2 节需要。见。

过程 `<-decls` 执行主要工作，比较两个声明集合。如果 $decls_1$ 和 $decls_2$ 是两个声明集合，当且仅当任何能提供 $decls_1$ 中声明绑定的模块，也能提供

interface-of : *ModuleBody* × *Tenv* → *Iface*

```
(define interface-of
  (lambda (m-body tenv)
    (cases module-body m-body
      (defns-module-body (defns)
        (simple-iface
          (defns-to-decls defns tenv))))))
```

defns-to-decls : *Listof*(*Defn*) × *Tenv* → *Decl*

```
(define defns-to-decls
  (lambda (defns tenv)
    (if (null? defns)
      '()
      (cases definition (car defns)
        (val-defn (var-name exp)
          (let ((ty (type-of exp tenv)))
            (cons
              (val-decl var-name ty)
              (defns-to-decls
                (cdr defns)
                (extend-tenv var-name ty tenv)))))))))
```

图 8.6 SIMPLE-MODULES 的检查器，第 2 部分

$decls_2$ 中声明的绑定时，我们说 $decls_1 <: decls_2$。如果 $decls_2$ 中的所有声明，在 $decls_1$ 中都有与之匹配的声明，就能保证这一点，就像上面的例子那样。

过程 `<:-decls` 首先检查 `decls1` 和 `decls2`。若 `decls2` 为空，那么它对 `decls1` 无所要求，所以结果为 `#t`。若 `decls2` 非空，但 `decls1` 为空，那么 `decls2` 有所要求，但 `decls1` 无可提供，所以结果为 `#f`。否则，我们比较 `decls1` 和 `decls2` 声明的第一对变量的名字；若二者相同，那么它们的类型必须匹配，然后我们递归处理两个声明列表余下的部分；若他们不同，那么我们递归处理 `decls1` 的 `cdr`，找出匹配 `decls2` 中第一个声明的内容。

这样，简单模块系统就完成了。

<:-iface : $Iface \times Iface \times Tenv \to Bool$

```
(define <:-iface
  (lambda (iface1 iface2 tenv)
    (cases interface iface1
      (simple-iface (decls1)
        (cases interface iface2
          (simple-iface (decls2)
            (<:-decls decls1 decls2 tenv)))))))
```

<:-decls : $Listof(Decl) \times Listof(Decl) \times Tenv \to Bool$

```
(define <:-decls
  (lambda (decls1 decls2 tenv)
    (cond
      ((null? decls2) #t)
      ((null? decls1) #f)
      (else
        (let ((name1 (decl->name (car decls1)))
              (name2 (decl->name (car decls2))))
          (if (eqv? name1 name2)
            (and
              (equal?
                (decl->type (car decls1))
                (decl->type (car decls2)))
              (<:-decls (cdr decls1) (cdr decls2) tenv))
            (<:-decls (cdr decls1) decls2 tenv)))))))
```

图 8.7 SIMPLE-MODULES 的接口比较

练习 8.1 [⋆] 修改检查器，检测并拒绝任何定义两个同名模块的程序。

练习 8.2 [⋆] 过程 `add-module-defns-to-env` 不完全正确，因为它加入了模块定义的所有值，而不只是接口中的值。修改 `add-module-defns-to-env`，只加入接口中声明的值。`add-module-defns-to-tenv` 也有此问题吗?

练习 8.3 [⋆] 修改语言的语法，以 `m.v` 代替 `from m take v` 使用受限变量。

练习 8.4 [⋆] 修改语言的表达式，像 ex7.24 那样，加入多声明 `let`、多参数过程和多声明 `letrec`。

练习 8.5 [⋆] 允许在模块主体中使用 `let` 和 `letrec` 声明。例如，可以写

```
module even-odd
 interface
  [even : (int -> bool)
   odd  : (int -> bool)]
 body
  letrec
   bool local-odd (x : int)  = ... (local-even -(x,1)) ...
   bool local-even (x : int) = ... (local-odd -(x,1)) ...
  in [even = local-even
      odd = local-odd]
```

练习 8.6 [⋆⋆] 允许在模块主体中定义局部模块。例如，可以写

```
module m1
 interface
  [u : int
   v : int]
 body
  module m2
   interface [v : int]
   body [v = 33]
  [u = 44
   v = -(from m2 take v, 1)]
```

练习 8.7 [⋆⋆] 扩展前一题的解答，允许模块将其他模块作为输出的一部分。例如，可以写

```
module m1
 interface
  [u : int
   n : [v : int]]
 body
  module m2
   interface [v : int]
   body [v = 33]
  [u = 44
   n = m2]

from m1 take n take v
```

练习 8.8 [⋆⋆] 在我们的语言中，模块必须按照接口中的顺序产生值，可以取消这种限制。取消它。

练习 8.9 [⋆⋆] 我们说我们的模块系统应当能解释模块之间的依赖关系。给 SIMPLE-MODULES 添加这种能力，在模块主体和程序主体中添加一条 `depends-on` 语句。那么，模块 `m` 不是在之前声明的所有模块的作用域中，而是在自身 `depends-on` 语句中列出模块的作用域中。例如，考虑程序

```
module m1 ...
module m2 ...
module m3 ...
module m4 ...
module m5
 interface [...]
 body
  depends-on m1, m3
  [...]
```

`m5` 的主体仅在来自 `m1` 或 `m3` 的受限变量的作用域中。即使 `m4` 输出了 `x` 的值，使用 `from m4 take x` 也将造成类型异常。

练习 8.10 [★★★] 我们还可以用 `depends-on` 这样的特性控制模块主体求值的时机。给 SIMPLE-MODULES 增加这种能力，在各模块主体和程序主体添加一条 `imports` 语句。`imports` 就像 `depends-on`，不同之处是，仅当（用 `imports` 语句）将模块输入到其他模块时，才求其主体的值。

这样，如果我们的语言有打印表达式，程序

```
module m1
 interface [] body [x = print(1)]
module m2
 interface [] body [x = print(2)]
module m3
 interface []
 body
  import m2
  [x = print(3)]
import m3, m1
33
```

在返回 33 之前，将打印 2、3 和 1。这里的模块接口为空，因为我们只关心它们主体求值的顺序。

练习 8.11 [★★★] 修改检查器，用 INFERRED 作为语言的表达式。这道练习中，你需要修改 `<:-decls`，不能用 `equal?` 比较类型。例如，在

```
module m
 interface [f : (int -> int)]
 body [f = proc (x : ?) x]
```

中，类型推导器报告的 `f` 实际类型可能是 `(tvar07 -> tvar07)`，这应当接受。但是，我们应拒绝模块

```
module m
 interface [f : (int -> bool)]
 body [f = proc (x : ?) x]
```

即使类型推导器报告的类型仍为 `(tvar07 -> tvar07)`。

8.2 声明类型的模块

至今为止，我们的接口只声明了普通变量及其类型。在下面这种模块语言 OPAQUE-TYPES 中，我们还允许接口声明类型。例如，在定义

```
module m1
 interface
  [opaque t
   zero : t
   succ : (t -> t)
   pred : (t -> t)
   is-zero : (t -> bool)]
body
 ...
```

中，接口声明了类型 `t`，以及该类型值的操作 `zero`、`succ`、`pred` 和 `is-zero`。如同 2.1 节，这套接口可能与算术操作的实现相关。这里的声明 `t` 为*模糊类型* (*opaque type*)，意为，模块之外的代码不知道这种类型的值如何表示。所有的外部代码都知道可以用 `from m1 take zero` 和 `from m1 take succ` 等过程处理 `from m1 take t` 类型的值。这样，`from m1 take t` 的表现就像 `int` 和 `bool` 之类的原生类型一样。

我们将介绍两种类型声明：*透明* (*transparent*) 类型和*模糊* (*opaque*) 类型。好的模块系统中，二者缺一不可。

8.2.1 例子

欲知其用途，再想想我们的几位开发者。爱丽丝一直用包含整数对的数据结构表示点的横纵坐标。她使用的语言具有 ex7.8 那样的类型，所以她的模块 `Alices-points` 接口具有如下声明：

```
initial-point : (int -> pairof int * int)
increment-x : (pairof int * int -> pairof int * int)
```

鲍伯和查理对此直发牢骚。他们不想一遍又一遍地写 `pairof int * int`。因此，爱丽丝用透明类型声明重写她的接口。这样，她可以写

```
module Alices-points
 interface
  [transparent point = pairof int * int
   initial-point : (int -> point)
   increment-x : (point -> point)
   get-x : (point -> int)
   ...]
```

这减轻了她的工作，因为她写得更少；这也减轻了她合作者的工作，因为他们在实现中可以写这样的定义：

```
[transparent point = from Alices-points take point
 foo = proc (p1 : point)
        proc (p2 : point) ...
 ...]
```

对某些项目中，这很不错。不过，爱丽丝的项目正好要表示固定形状金属导轨上的点，所以横纵坐标不是相互独立的。[1]爱丽丝实现 `increment-x` 时，要小心翼翼地更新纵坐标，以匹配横坐标的改变。但是鲍伯不知道这点，所以他的过程写作

```
increment-y = proc (p : point)
                unpair x y = p
                in newpair(x, -(y,-1))
```

由于鲍伯的代码修改纵坐标时不随之修改横坐标，爱丽丝的代码就没法正常工作了。

更糟糕的是，如果爱丽丝打算修改点的表示，把纵坐标作为第一部分呢？她可以按照新的表示修改她的代码。但是鲍伯的代码就坏掉了，因为过程 `increment-y` 修改了序对中的错误部分。

爱丽丝可以把 `point` 声明为*模糊*数据类型来解决她的问题。她把接口重写为

```
opaque point
initial-point : (int -> point)
increment-x : (point -> point)
get-x : (point -> int)
```

[1]不妨将金属导轨视为有长度无宽度的圆形轨迹，以圆心为坐标原点。要保证横坐标变化时，得到的点仍在圆上，则纵坐标也要相应改变。反之亦然。••译注

现在鲍伯用过程 `initial-point` 创建新的点，而且他可以用 `from Alices-points take get-x` 和 `from Alices-points take increment-x` 处理点，但是除了爱丽丝接口中的过程外，他无法用其他过程处理点。尤其是，他写不出过程 `increment-y`，因为它用了爱丽丝接口之外的过程处理点。

在本节的剩余部分中，我们探究这些组件的更多例子。

透明类型

我们首先讨论透明类型声明。有时这些又称作*具体* (*concrete*) 类型或*类型缩写* (*type abbreviation*)。

例 8.6 程序

```
module m1
 interface
  [transparent t = int
   z : t
   s : (t -> t)
   is-z? : (t -> bool)]
 body
  [type t = int
   z = 33
   s = proc (x : t) -(x,-1)
   is-z? = proc (x : t) zero?(-(x,z))]

 proc (x : from m1 take t)
  (from m1 take is-z? -(x,0))
```

类型为 `(int -> bool)`。

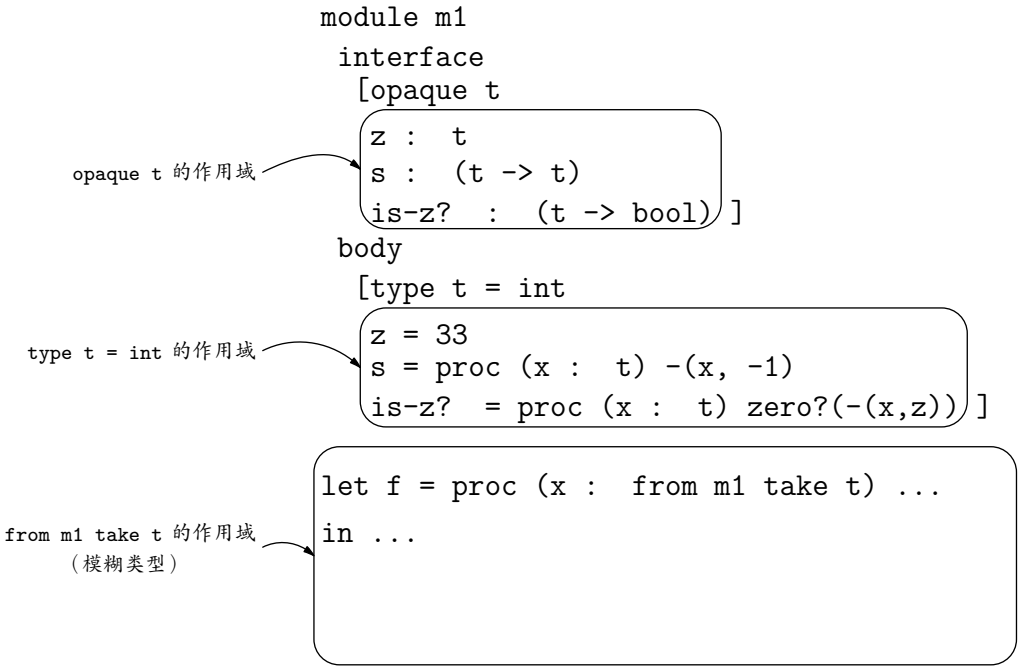


图 8.8 模块类型声明的作用域

在接口的剩余部分中，声明 `transparent t = int` 将 `t` 绑定到类型 `int`，所以我们可以写 `z : t`。更重要的是，在程序的剩余部分中，声明也将 `from m1 take t` 绑定到 `int`。我们称之为*受限类型* (*qualified type*)。这里，我们用它声明了绑定到变量 `z` 的类型。声明的作用域是接口的剩余部分，以及模块定义之后程序的剩余部分。

模块主体中的定义 `type t = int` 在主体的剩余部分中，将 `t` 绑定到 `int`，所以我们可以写 `s = proc (x : t) ...`。像之前那样，定义的作用域是主体的剩余部分（见）。

当然，我们可以给类型起任意名字，也可以声明多个类型。类型声明可以出现在接口中任意位置，只要每个声明都先于使用。

模糊类型

模块还可以用 `opaque-type` 声明输出*模糊*类型。模糊类型有时又称作*抽象类型* (*abstract type*)。

例 8.7 我们把程序中的透明类型替换为模糊类型。得出的程序是

```
module m1
 interface
  [opaque t
   z : t
   s : (t -> t)
   is-z? : (t -> bool)]
 body
  [type t = int
   z = 33
   s = proc (x : t) -(x,-1)
   is-z? = proc (x : t) zero?(-(x,z))]

 proc (x : from m1 take t)
  (from m1 take is-z? -(x,0))
```

接口中的声明 `opaque t` 把 `t` 作为一种新的模糊类型名字。模糊类型的行为就像 `int` 或 `bool` 之类的原生类型。名为 `t` 的类型在接口的剩余部分中绑定到这种模糊类型，而受限类型 `from m1 take t` 在程序的剩余部分中绑

定到同一模糊类型。程序的剩余部分都知道 `from m1 take z` 绑定到一个值，其类型为 `from m1 take t`；`from m1 take s` 和 `from m1 take is-z?` 绑定到过程，用来处理这种类型的值。这就是抽象边界。类型检查器确保表达式的类型为 `from m1 take t` 时，求值是安全的，所以表达式的值只能通过这些操作符生成，如 suitable-env 所述。

与之对应，定义 `type t = int` 在模块主体内部将 `t` 作为 `int` 的名字，但是，由于程序的剩余部分从模块接口获得绑定，所以对此一无所知。

所以 `-(x,0)` 类型异常，因为主程序不知道类型 `from m1 take t` 为的值就是类型为 `int` 的值。

我们改变程序，删掉算术操作，得

```
module m1
 interface
  [opaque t
   z : t
   s : (t -> t)
   is-z? : (t -> bool)]
 body
  [type t = int
   z = 33
   s = proc (x : t) -(x,-1)
   is-z? = proc (x : t) zero?(-(x,z))]

 proc (x : from m1 take t)
  (from m1 take is-z? x)
```

现在，我们的程序类型正常，类型为 `(from m1 take t -> bool)`。

通过强化抽象边界，类型检查器确保程序只能通过接口提供的过程处理接口提供的值。如第 2 章所述，这给我们提供了机制来分离数据类型的用户和实现。接下来，我们给出这一技术的几个例子。

例 8.8 如果程序使用了模块定义

```
module colors
 interface
  [opaque color
   red : color
   green : color
```

```
    is-red? : (color -> bool)]
  body
   [type color = int
    red = 0
    green = 1
    is-red? = proc (c : color) zero?(c)]
```

程序没法知道 `from colors take color` 实际为 `int`，也不知道 `from colors take green` 实际为 1（也许有一个例外：返回颜色作为最终答案，然后打印出来）。

例 8.9 程序

```
module ints1
 interface
  [opaque t
   zero : t
   succ : (t -> t)
   pred : (t -> t)
   is-zero : (t -> bool)]
 body
  [type t = int
   zero = 0
   succ = proc(x : t) -(x,-5)
   pred = proc(x : t) -(x,5)
   is-zero = proc (x : t) zero?(x)]

let z = from ints1 take zero
in let s = from ints1 take succ
   in (s (s z))
```

类型为 `from ints1 take t`，值为 10。但我们只能通过 `ints1` 输出的过程处理这个值。这个模块用表达值 $5 * k$ 表示整数 k。用 2.1 节的表示法，写作 $\lceil k \rceil = 5 * k$。

例 8.10 在这个模块中，$\lceil k \rceil = -3 * k$。

```
module ints2
 interface
  [opaque t
```

```
    zero : t
    succ : (t -> t)
    pred : (t -> t)
    is-zero : (t -> bool)]
  body
   [type t = int
    zero = 0
    succ = proc(x : t) -(x,3)
    pred = proc(x : t) -(x,-3)
    is-zero = proc (x : t) zero?(x)]

let z = from ints2 take zero
in let s = from ints2 take succ
   in (s (s z))
```

类型为 `from ints2 take t`，值为 -6。

例 8.11 在前面的例子中，我们不能直接处理值，但我们能用模块输出的过程处理它们。像第 2 章那样，我们可以结合这些过程做有用的工作。这里，我们将它们结合起来，写出过程 `to-int`，把模块中的值转回 `int` 类型。

```
module ints1 ... 同前...

let z = from ints1 take zero
in let s = from ints1 take succ
in let p = from ints1 take pred
in let z? = from ints1 take is-zero
in letrec int to-int (x : from ints1 take t) =
              if (z? x)
              then 0
              else -((to-int (p x)), -1)
in (to-int (s (s z)))
```

类型为 `int`，值为 2。

例 8.12 这里用到的技术与 `ints2` 中算术操作的实现相同。

```
module ints2 ... 同前...

let z = from ints2 take zero
in let s = from ints2 take succ
```

```
in let p = from ints2 take pred
in let z? = from ints2 take is-zero
in letrec int to-int (x : from ints2 take t) =
              if (z? x)
              then 0
              else -((to-int (p x)), -1)
in (to-int (s (s z)))
```

类型同样为 `int`，值为 2。

在 8.3 节中，我们展示如何将两个抽象出来。

例 8.13 在下面的程序中，我们设计一个模块来封装布尔类型。布尔值用整数值表示，但是像那样，程序的剩余部分对此一无所知。

```
module mybool
 interface
  [opaque t
   true : t
   false : t
   and : (t -> (t -> t))
   not : (t -> t)
   to-bool : (t -> bool)]
 body
  [type t = int
   true = 0
   false = 13
   and = proc (x : t)
          proc (y : t)
           if zero?(x) then y else false
   not = proc (x : t)
           if zero?(x) then false else true
   to-bool = proc (x : t) zero?(x)]

let true = from mybool take true
in let false = from mybool take false
   in let and = from mybool take and
      in ((and true) false)
```

类型为 `from mybool take t`，值为 13。

练习 8.12 [⋆] 在中，and 和 not 的定义可以从模块内部移到外面吗？to-bool 呢？

练习 8.13 [⋆] 写一个模块，用 $5 * k + 3$ 表示整数 k，实现算术操作。

练习 8.14 [⋆] 下面是 mybool () 的另一种定义：

```
module mybool
 interface
  [opaque t
   true : t
   false : t
   and : (t -> (t -> t))
   not : (t -> t)
   to-bool : (t -> bool)]
 body
  [type t = int
   true = 1
   false = 0
   and = proc (x : t)
          proc (y : t)
           if zero?(x) then false else y
   not = proc (x : t)
          if zero?(x) then true else false
   to-bool = proc (x : t)
              if zero?(x) then zero?(1) else zero?(0)]
```

有没有程序类型为 int，用 mybool 原来的定义返回一个值，用新的定义返回另一个值？

练习 8.15 [⋆⋆] 写一个模块，实现抽象表。你实现的表应类似环境，但不是把符号绑定到 Scheme 值，而是把整数值绑定到整数值。接口提供一个值，表示空表；两个过程 add-to-table 和 lookup-in-table 类似 extend-env 和 apply-env。由于我们的语言只有单参数过程，我们用咖喱化（ex3.20）实现等效的多参数过程。你可以用查询任何值都返回 0 的表模拟空表。这是该模块的一个例子：

```
module tables
 interface
```

```
  [opaque table
   empty : table
   add-to-table : (int -> (int -> (table -> table)))
   lookup-in-table : (int -> (table -> int))]
 body
  [type table = (int -> int)
   ...]

let empty = from tables take empty
in let add-binding = from tables take add-to-table
   in let lookup = from tables take lookup-in-table
      in let table1 = (((add-binding 3) 300)
                       (((add-binding 4) 400)
                        (((add-binding 3) 600)
                         empty)))
         in -(((lookup 4) table1),
              ((lookup 3) table1))
```

这个程序类型应为 `int`。表 `table1` 把 4 绑定到 400，把 3 绑定到 300，所以程序的值应为 100。

8.2.2 实现

现在我们来扩展系统，实现透明类型和模糊类型声明，及受限类型的引用。

语法和解释器

我们给两种新类型添加语法：有名类型（如 `t`）和受限类型（如 `from m1 take t`）。

Type ::= *Identifier*

```
named-type (name)
```

Type ::= `from` *Identifier* `take` *Identifier*

```
qualified-type (m-name t-name)
```

我们为模糊类型和透明类型新增两种声明。

Decl ::= `opaque` *Identifier*
`opaque-type-decl (t-name)`

Decl ::= `transparent` *Identifier* `=` *Type*
`transparent-type-decl (t-name ty)`

我们还要新增一种定义：类型定义，用来定义模糊类型和透明类型。

Defn ::= `type` *Identifier* `=` *Type*
`type-defn (name ty)`

解释器不需要查看类型和声明，所以解释器的唯一改动是忽略类型定义。

defns-to-env : $Listof(Defn) \times Env \to Env$

```
(define defns-to-env
  (lambda (defns env)
    (if (null? defns)
      (empty-env)
      (cases definition (car defns)
        (val-defn (var exp) ...as before...)
        (type-defn (type-name type)
          (defns-to-env (cdr defns) env))))))
```

检查器

检查器的改动就多多了，因为所有关于类型的操作都要扩展，以便处理新的类型。

首先，我们介绍一种系统性的方法来处理模糊类型和透明类型。模糊类型就像 `int` 或 `bool` 之类的原生类型一样。而透明类型名副其实，是透明的：它们的行为与定义相同。所以每个类型都等价于下列语法描述的：

Type ::= `int` | `bool` | `from` m `take` t | `(`*Type* `->` *Type*`)`

其中，t 为 m 中的模糊类型声明。我们称这种形式的类型为*展开类型* (*expanded type*)。

接下来我们扩展类型环境，处理新类型。我们的类型环境将每个有名类型或受限类型绑定到一个展开类型。新的类型环境定义为

```
(define-datatype type-environment type-environment?
  (empty-tenv)
  (extend-tenv ... 同前...)
  (extend-tenv-with-module ... 同前...)
  (extend-tenv-with-type
    (t-name symbol?)
    (type type?)
    (saved-tenv type-environment?)))
```

它满足条件：`type` 总是一个展开类型。像 invariant 讨论的，这个条件是一*不变式*。

接着我们写出函数 `expand-type`，它取一个类型和一个类型环境，用后者中的绑定展开前者。根据不变式“结果类型已展开”，它在类型环境中查询有名类型和受限类型，对 `proc` 类型，它递归处理参数和结果类型。

expand-type : $Type \times Tenv \to ExpandedType$

```
(define expand-type
  (lambda (ty tenv)
    (cases type ty
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (arg-type result-type)
        (proc-type
          (expand-type arg-type tenv)
          (expand-type result-type tenv)))
      (named-type (name)
        (lookup-type-name-in-tenv tenv name))
      (qualified-type (m-name t-name)
        (lookup-qualified-type-in-tenv m-name t-name tenv)))))
```

为了维持这一不变式，我们必须保证不论何时扩展类型环境，都要调用 `expand-type`。这种地方有三处：

- 在检查器中的 type-of 内；
- 用 defns-to-decls 处理类型定义列表之处；
- 在 add-module-defns-to-tenv 中，向类型环境添加模块之处。

在检查器中，我们把形如

```
(extend-tenv sym ty tenv)
```

的调用替换为

```
(extend-tenv var (expand-type ty tenv) tenv)
```

在 defns-to-decls 中，当我们遇到类型定义时，我们展开右边的定义，然后将其加入类型环境中。type-of 返回的类型一定是展开的，所以我们不需要再次展开它。由于在模块主体中，所有类型绑定都是透明的，所以我们把类型定义转换为透明类型声明。在 add-module-defns-to-tenv 中，我们调用 extend-tenv-with-module，将接口添加到类型环境中。这里，我们需要展开接口，以确保它包含的所有类型都已展开。要完成这一点，我们修改 add-module-defns-to-tenv，调用 expand-iface。见。

过程 expand-iface（）调用 expand-decls。我们提取出这些过程，为 8.3 节做准备。

defns-to-decls : $Listof(Defn) \times Tenv \to Decl$

```
(define defns-to-decls
  (lambda (defns tenv)
    (if (null? defns)
      '()
      (cases definition (car defns)
        (val-defn (var-name exp)
          (let ((ty (type-of exp tenv)))
            (let ((new-env (extend-tenv var-name ty tenv)))
              (cons
                (val-decl var-name ty)
                (defns-to-decls (cdr defns) new-env)))))
        (type-defn (name ty)
          (let ((new-env
                  (extend-tenv-with-type
                    name (expand-type ty tenv) tenv)))
            (cons
              (transparent-type-decl name ty)
              (defns-to-decls (cdr defns) new-env)))))))))
```

add-module-defns-to-tenv : $Listof(ModuleDefn) \times Tenv \to Tenv$

```
(define add-module-defns-to-tenv
  (lambda (defns tenv)
    (if (null? defns)
      tenv
      (cases module-definition (car defns)
        (a-module-definition (m-name expected-iface m-body)
          (let ((actual-iface (interface-of m-body tenv)))
            (if (<:-iface actual-iface expected-iface tenv)
              (let ((new-env
                      (extend-tenv-with-module m-name
                        (expand-iface
                          m-name expected-iface tenv)
                        tenv)))
                (add-module-defns-to-tenv
                  (cdr defns) new-env))
              (report-module-doesnt-satisfy-iface
                m-name expected-iface actual-iface)))))))))
```

图 8.9 OPAQUE-TYPES 的检查器，第 1 部分

expand-iface : $Sym \times Iface \times Tenv \to Iface$

```
(define expand-iface
  (lambda (m-name iface tenv)
    (cases interface iface
      (simple-iface (decls)
        (simple-iface
          (expand-decls m-name decls tenv))))))
```

expand-decls : $Sym \times Listof(Decl) \times Tenv \to Listof(Decl)$

```
(define expand-decls
  (lambda (m-name decls internal-tenv)
    (if (null? decls) ()
      (cases declaration (car decls)
        (opaque-type-decl (t-name)
          (let ((expanded-type
                  (qualified-type m-name t-name)))
            (let ((new-env
                    (extend-tenv-with-type
                      t-name expanded-type internal-tenv)))
              (cons
                (transparent-type-decl t-name expanded-type)
                (expand-decls
                  m-name (cdr decls) new-env)))))
        (transparent-type-decl (t-name ty)
          (let ((expanded-type
                  (expand-type ty internal-tenv)))
            (let ((new-env
                    (extend-tenv-with-type
                      t-name expanded-type internal-tenv)))
              (cons
                (transparent-type-decl t-name expanded-type)
                (expand-decls
                  m-name (cdr decls) new-env)))))
        (val-decl (var-name ty)
          (let ((expanded-type
                  (expand-type ty internal-tenv)))
            (cons
              (val-decl var-name expanded-type)
              (expand-decls
                m-name (cdr decls) internal-tenv)))))))))
```

图 8.10 OPAQUE-TYPES 的检查器，第 2 部分

过程 expand-decls 遍历声明集合，创建新的类型环境，其中的每个类型和变量名都绑定到一个展开类型。麻烦之处是声明遵循 let* 式作用域：集合中的每个声明的作用域包含它之后的所有声明。

要明白这意味着什么，考虑模块定义

```
module m1
 interface
  [opaque t
   transparent u = int
   transparent uu = (t -> u)
   % A 处
   f : uu
   ...]
 body
  [...]
```

要满足不变式，类型环境中的 m1 应绑定到包含如下声明的接口

```
[transparent t = from m1 take t
 transparent u = int
 transparent uu = (from m1 take t -> int)
 f : (from m1 take t -> int)
 ...]
```

只要我们这样做，不论何时我们从类型环境中查询类型时，得到的都是期望的展开类型。

在 A 处，紧随声明 f 之后，类型环境应绑定到

```
t  绑定到 from m1 take t
u  绑定到 int
uu 绑定到 (from m1 take t -> int)
```

我们把类似上面 A 处的类型环境称为内部类型环境。它作为参数传给 expand-decls。

现在我们可以写出 expand-decls。像 defns-to-decls，这个过程只创建透明声明，因为它的用途就是创建查询受限类型所用的数据结构。

最后，我们修改 <:-decls，处理两种新声明。我们必须处理声明集合内部的作用域关系。例如，如果我们比较

```
[transparent t = int
x : bool                <:    [y : int]
y : t]
```

处理声明 `y` 时，我们需要知道 `t` 指代 `int` 类型。所以，当我们递归向下处理声明列表时，我们需要随之扩展类型环境，就像在 `expand-decls` 中生成 `internal-tenv` 一样。我们调用 `extend-tenv-with-decl` 处理这些，它取一声明，根据类型环境将其展开为适当的类型 ()。

<:-decls : *Listof(Decl)* × *Listof(Decl)* × *Tenv* → *Bool*

```
(define <:-decls
  (lambda (decls1 decls2 tenv)
    (cond
      ((null? decls2) #t)
      ((null? decls1) #f)
      (else
        (let ((name1 (decl->name (car decls1)))
              (name2 (decl->name (car decls2))))
          (if (eqv? name1 name2)
            (and
              (<:-decl
                (car decls1) (car decls2) tenv)
              (<:-decls
                (cdr decls1) (cdr decls2)
                (extend-tenv-with-decl
                  (car decls1) tenv)))
            (<:-decls
              (cdr decls1) decls2
              (extend-tenv-with-decl
                (car decls1) tenv)))))))))
```

extend-tenv-with-decl : *Decl* × *Tenv* → *Tenv*

```
(define extend-tenv-with-decl
  (lambda (decl tenv)
    (cases declaration decl
      (val-decl (name ty) tenv)
      (transparent-type-decl (name ty)
        (extend-tenv-with-type
          name
          (expand-type ty tenv)
          tenv))
      (opaque-type-decl (name)
        (extend-tenv-with-type
          name
          (qualified-type (fresh-module-name '%unknown) name)
          tenv)))))
```

图 8.11 OPAQUE-TYPES 的检查器，第 3 部分

在展开过程中，我们总是用 decls1。欲知其原因，考虑比较

```
[transparent t = int                [opaque t
transparent u = (t -> t)  <:         transparent u = (t -> int)
f : (t -> u)]                        f : (t -> (int -> int))]
```

这一比较应该通过，因为当模块主体提供左侧的绑定时，也是右侧接口的正确实现。

比较类型 u 的两个定义时，我们得知道类型 t 实际上是 int。即使左边的声明没有出现在右边，同样的技巧也适用，就像上面第一个例子中的声明 t 所展示的。我们调用 expand-type 来维持不变式“类型环境中的所有类型均已展开”。extend-tenv-with-decl 最后一句中选什么模块名无关紧要，因为受限类型支持的唯一操作是 equal?。所以用 fresh-module-name 足以保证这一受限类型是新生成的。

现在来处理关键问题：如何比较声明？仅当二者使用相同的名字（变量或类型）时，声明才能匹配。如果一对声明同名，有四种匹配方式：

- 二者均为值声明，且类型匹配。
- 二者均为模糊类型声明。
- 二者均为透明类型声明，且定义匹配。
- decl1 为透明类型声明，decl2 为模糊类型声明。例如，假设有个模块的接口声明了 opaque t，主体中的定义为 type t = int，应当接受这种做法。过程 defns-to-decls 将定义 type t = int 转换为透明类型声明，所以 add-module-defns-to-tenv 中的条件

  ```
  actual-iface <: expected-iface
  ```

  需要检查

  $$\texttt{(transparent } t \texttt{ = int)} <: \texttt{(opaque } t\texttt{)}$$

  是否成立。

  由于这一模块应被接受，该项测试应返回真。

这告诉我们，类型已知的对象总能作为类型未知的对象。反之则不然。例如，

$$\texttt{(opaque } t\texttt{)} <: \texttt{(transparent } t \texttt{ = int)}$$

应为假，因为模糊类型值的实际类型可能不是 `int`，而且满足 `opaque t` 的模块可能无法满足 `transparent t = int`。

这样，我们就得出中的代码。`equiv-type?` 的定义扩展其类型，所以，在上面的例子

```
[transparent t = int x : bool y : t] <: [y : int]
```

中，左边的 `t` 展开为 `int`，匹配成功。

练习 8.16 [⋆] 用 ex7.24 中的语言扩展本节的系统，然后重写 ex8.15，用多参数过程代替返回过程的过程。

练习 8.17 [⋆⋆] 仿照 ex8.8，允许模块以不同于接口声明的顺序产生值。但是记住，定义•• 尤其是类型的定义••必须遵守定界规则。

练习 8.18 [⋆⋆] 我们代码依赖的不变式是：类型环境中的所有类型均已展开。我们在代码中多次调用 `expand-type` 来维持这一不变式。这就很容易因忘记调用 `expand-type` 而破坏系统。重构代码，减少 `expand-type` 的调用，以便更稳定地维持不变式。

<:-decl : *Decl* × *Decl* × *Tenv* → *Bool*

```
(define <:-decl
  (lambda (decl1 decl2 tenv)
    (or
      (and
        (val-decl? decl1)
        (val-decl? decl2)
        (equiv-type?
          (decl->type decl1)
          (decl->type decl2) tenv))
      (and
        (transparent-type-decl? decl1)
        (transparent-type-decl? decl2)
        (equiv-type?
          (decl->type decl1)
          (decl->type decl2) tenv))
      (and
        (transparent-type-decl? decl1)
        (opaque-type-decl? decl2))
      (and
        (opaque-type-decl? decl1)
        (opaque-type-decl? decl2)))))
```

equiv-type? : *Type* × *Type* × *Tenv* → *Bool*

```
(define equiv-type?
  (lambda (ty1 ty2 tenv)
    (equal?
      (expand-type ty1 tenv)
      (expand-type ty2 tenv))))
```

图 8.12 OPAQUE-TYPES 的检查器，第 4 部分

8.3 模块过程

OPAQUE-TYPES 中的程序有固定的依赖关系。模块 `m4` 可能依赖 `m3` 和 `m2`，`m2` 依赖 `m1`。有时，我们说依赖关系是*硬编码* (*hard-coded*) 的。通常，这种硬编码的依赖关系会导致糟糕的程序设计，因为这使模块难以复用。本节，我们给系统添加名为*模块过程* (*module procedure*)（有时又称*参数化模块* (*parameterized module*)）的组件，以便复用模块。我们称这种新语言为 PROC-MODULES。

8.3.1 例子

再来看我们的三位开发者。查理想用爱丽丝模块的某些组件。但爱丽丝的模块使用了鲍伯模块提供的数据库，而查理想用另一数据库，由其他模块提供（戴安娜所写）。

要实现这些，爱丽丝用模块过程重写她的代码。模块过程就像过程，但它处理的是模块，而非表达值。在模块层面上，接口就像类型。就像 CHECKED 中的过程类型指定参数和结果类型，模块过程的接口指定参数的接口和结果的接口。

爱丽丝写出新的模块 `Alices-point-builder`，开头为

```
module Alices-point-builder
 interface
  ((database : [opaque db-type
                opaque node-type
                insert-node : (node-type ->
                               (db-type -> db-type))
                ...])
   => [opaque point
       initial-point : (int -> point)
       ...])
```

这套接口说的是，`Alices-point-builder` 是一模块过程。它期望的参数是一个模块，该模块输出两个类型，`db-type` 和 `node-type`，一个过程，`insert-node`，可能还有其他一些值。给定这个模块，`Alices-point-builder` 应生成一个模块，生成的模块输出模糊类型 `point`，过程 `initial-point`，还可能有其他一些值。`Alices-point-builder` 的接口还指定了参

数的局部名称；稍后我们将看到为何需要它。

爱丽丝的新模块主体开头为

```
body
 module-proc (m : [opaque db-type
                   opaque node-type
                   insert-node : (node-type ->
                                  (db-type -> db-type))
                   ...])
  [type point = ...
   initial-point = ... from m take insert-node ...
   ...]
```

就像一般的过程表达式形如

proc (*var* : *t*) *e*

模块过程形如

module-proc (*m* : [...]) [...]

本例中，爱丽丝选择 m 作为模块过程中绑定变量的名字；它不必和接口中的局部名字相同。我们需要再写一遍参数的接口，因为模块接口的作用域不包含模块主体。可以免除这一限制（见 ex8.27）。

现在，爱丽丝把她的模块重写为

```
module Alices-points
 interface
  [opaque point
  initial-point : (int -> point)
  ...]
 body
  (Alices-point-builder Bobs-db-module)
```

查理的模块写成

```
module Charlies-points
 interface
  [opaque point
   initial-point : (int -> point)
   ...]
 body
  (Alices-point-builder Dianas-db-module)
```

模块 Alices-points 使用 Bobs-db-module 处理数据库。模块 Charlies-points 使用 Dianas-db-module 处理数据库。这样组织可以复用 Alices-point-builder 中的代码。这不仅避免了写两次同样的代码，而且，在代码需要变动时，可以只改一处，自动传播到 Alices-point 和 Charlies-points 中。

另一个例子，考虑和。在这两个例子中，我们用基本相同的代码写 to-int。在中它是

```
letrec int to-int (x : from ints1 take t)
            = if (z? x)
              then 0
              else -((to-int (p x)), -1)
```

在中，x 的类型是 from ints2 take t。所以我们重写为参数化的模块，参数模块产生所需的整数。

例 8.14 声明

```
module to-int-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
    => [to-int : (from ints take t -> int)])
 body
  module-proc (ints : [opaque t
                zero : t
                succ : (t -> t)
                pred : (t -> t)
                is-zero : (t -> bool)])
   [to-int
     = let z? = from ints take is-zero
       in let p = from ints take pred
       in letrec int to-int (x : from ints take t)
                     = if (z? x)
                       then 0
                       else -((to-int (p x)), -1)
       in to-int]
```

定义了一个模块过程。这套接口说的是，该模块取模块 ints，生成另一模块；ints 实现算术操作的接口，生成的模块输出过程 to-int，将 ints 中的类型 t 转换为整数。得出的过程 to-int 不能依赖算术操作的实现，因为我们根本不知道实现是什么！这段代码中，ints 声明了两次：一次在接口中，一次在主体中。像我们之前说的，这是因为接口声明的作用域只限于接口，不包含模块主体。

我们再来看使用 to-int 的一些例子：

例 8.15

```
module to-int-maker ... 同前...

module ints1 ... 同前...

module ints1-to-int
 interface [to-int : (from ints1 take t -> int)]
 body
  (to-int-maker ints1)

let two1 = (from ints1 take succ
            (from ints1 take succ
             from ints1 take zero))
in (from ints1-to-int take to-int
    two1)
```

类型为 int，值为 2。因为我们首先定义了模块 to-int-maker 和 ints1。然后我们用 ints1 调用 to-int-maker，得到模块 ints1-to-int，它输出绑定 from ints1-to-int take to-int。

下面这个例子两次使用 to-int-maker，处理两种不同的算术操作实现。

例 8.16

```
module to-int-maker ... 同前...

module ints1 ... 同前...
```

```
module ints2 ... 同前...

module ints1-to-int
interface [to-int : (from ints1 take t -> int)]
body (to-int-maker ints1)

module ints2-to-int
interface [to-int : (from ints2 take t -> int)]
body (to-int-maker ints2)

let s1 = from ints1 take succ
in let z1 = from ints1 take zero
in let to-ints1 = from ints1-to-int take to-int

in let s2 = from ints2 take succ
in let z2 = from ints2 take zero
in let to-ints2 = from ints2-to-int take to-int

in let two1 = (s1 (s1 z1))
in let two2 = (s2 (s2 z2))
in -((to-ints1 two1), (to-ints2 two2))
```

类型为 `int`，值为 0。如果我们将 `(to-ints2 two2)` 替换为 `(to-ints2 two1)`，则程序类型异常，因为 `to-ints2` 期望的参数类型是 `int2` 表示的算术操作，但值 `two1` 的类型是 `int1` 表示的算术操作。

练习 8.19 [⋆] 中，创建 `two1` 和 `two2` 的代码重复，因此可以抽象出来。完成模块定义

```
module from-int-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
    => [from-int : (int -> from ints take t)])
 body
  ...
```

用模块 `ints` 表示整数表达值。用你的模块重做中的计算。用大于 2 的参数测试。

练习 8.20 [⋆] 完成模块定义

```
module sum-prod-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
    => [plus : (from ints take t
                -> (from ints take t
                    -> from ints take t))
        times : (from ints take t
                 -> (from ints take t
                     -> from ints take t))])
 body
  [plus = ...
   times = ...]
```

定义一个模块过程，它取一算术操作的实现，生成用这种实现的求和和求积的过程。用 plus 中的 `plus` 定义，以及类似的 `times` 定义。

练习 8.21 [⋆] 写一个模块过程，取算术操作的实现 `ints`，返回算术操作的另一种实现，其中，整数 k 以 `ints` 中的 $2 * k$ 表示。

练习 8.22 [⋆] 完成模块定义

```
module equality-maker
 interface
  ((ints : [opaque t
            zero : t
            succ : (t -> t)
            pred : (t -> t)
            is-zero : (t -> bool)])
    => [equal : (from ints take t
```

```
                    -> (from ints take t
                        -> bool))])
  body
   ...
```

定义一个模块过程，它取一算术操作的实现，生成一过程，用这种实现做相等比较。

练习 8.23 [⋆] 写出模块 `table-of`，它与 ex8.15 中的 `table` 模块类似，只是将表的内容参数化，这样就能用

```
module mybool-tables
 interface
  [opaque table
   empty : table
   add-to-table : (int ->
                   (from mybool take t ->
                    (table -> table)))
   lookup-in-table : (int ->
                      (table ->
                       from mybool take t))]
 body
  (table-of mybool)
```

定义包含 `from mybool take t` 类型值的表。

8.3.2 实现

语法

给我们的语言添加模块过程就像添加过程一样。模块过程的接口很像 `proc` 的类型。

Iface ::= `((`*Identifier* `:` *Iface*`))` `=>` *Iface*

```
proc-iface (param-name param-iface result-iface)
```

虽然这套接口看起来像是普通的过程类型，它还是有两处区别。首先，它描述了模块值到模块值的函数，而非表达值到表达值的函数。其次，不像过程

类型，它要给函数的输入命名。这是必须的，因为输出的接口可能依赖输入的值，就像 `to-int-maker` 的类型那样：

```
((ints : [opaque t
          zero : t
          succ : (t -> t)
          pred : (t -> t)
          is-zero : (t -> bool)])
  => [to-int : (from ints take t -> int)])
```

`to-int-maker` 取一模块 `ints`，生成一模块，其类型不仅依赖 `ints` 中的固定类型，也依赖 `ints` 本身。当我们像那样用 `ints1` 调用 `to-int-maker` 时，得到的模块接口是

```
[to-int : (from ints1 take t -> int)]
```

而当我们用 `ints2` 调用时，得到的是另一个接口

```
[to-int : (from ints2 take t -> int)]
```

我们扩展 `expand-iface` 来处理这些新接口，并按已展开处理。这样行得通是因为参数接口和结果接口在需要时自会展开。

expand-iface : $Sym \times Iface \times Tenv \to Iface$

```
(define expand-iface
  (lambda (m-name iface tenv)
    (cases interface iface
      (simple-iface (decls) ... 同前...)
      (proc-iface (param-name param-iface result-iface)
        iface))))
```

我们需要新的模块主体来创建模块过程，引用模块过程的绑定变量，以及调用模块过程。

ModuleBody ::= `module-proc (`*Identifier* `:` *Iface*`)` *ModuleBody*

```
proc-module-body (m-name m-type m-body)
```

ModuleBody ::= *Identifier*

```
var-module-body (m-name)
```

ModuleBody ::= `(`*Identifier* *Identifier*`)`

```
app-module-body (rator rand)
```

解释器

首先，类似过程，我们新增一种模块。

```
(define-datatype typed-module typed-module?
  (simple-module
    (bindings environment?))
  (proc-module
    (b-var symbol?)
    (body module-body?)
    (saved-env environment?)))
```

我们扩展 `value-of-module-body` 处理新的模块主体。代码与表达式中的变量引用和过程调用十分类似 ()。

value-of-module-body : $ModuleBody \times Env \to TypedModule$

```
(define value-of-module-body
  (lambda (m-body env)
    (cases module-body m-body
      (defns-module-body (defns) ...as before...)
      (var-module-body (m-name)
        (lookup-module-name-in-env m-name env))
      (proc-module-body (m-name m-type m-body)
        (proc-module m-name m-body env))
      (app-module-body (rator rand)
        (let ((rator-val
                (lookup-module-name-in-env rator env))
              (rand-val
                (lookup-module-name-in-env rand env)))
          (cases typed-module rator-val
            (proc-module (m-name m-body env)
              (value-of-module-body m-body
                (extend-env-with-module
                  m-name rand-val env)))
            (else
              (report-bad-module-app rator-val)))))))))
```

图 8.13 `value-of-module-body`

检查器

IFACE-M-VAR

$$\texttt{(} \triangleright \;\; m \;\; tenv \texttt{)} = tenv\texttt{(}m\texttt{)}$$

IFACE-M-PROC

$$\frac{\texttt{(} \triangleright \;\; \texttt{body [}m\texttt{=}i_1\texttt{]tenv)} = i'}{\texttt{(} \triangleright \;\; \texttt{(m-proc (}m\texttt{:}i_1\texttt{)} \;\; body\texttt{))} = \texttt{((}m\texttt{:}i_1\texttt{)} \;\; \texttt{=>} \;\; i'_1\texttt{)}}$$

IFACE-M-APP

$$\frac{tenv(m_1) = \texttt{((}m\texttt{:}i_1\texttt{)} \;\; \texttt{=>} \;\; i'_1\texttt{)} \quad tenv(m_2) = i_2 \qquad i_2 <: i_1}{\texttt{(} \triangleright \;\; \texttt{(}m_1 \;\; m_2\texttt{)} \;\; \texttt{tenv)} = \texttt{i((}m\texttt{:}i_1\texttt{)} \;\; \texttt{=>} \;\; i'_1\texttt{)}}$$

图 8.14 新模块主体的判类规则

我们可以给新的模块主体写出 7.2 节那样的规则。这些规则如所示。为了能在一页纸内写下规则，我们用 `(`$\triangleright$ $body$ $tenv$`) = `i 代替 `(interface-of` $body$ $tenv$`) = `i。

正如预想的那样，模块变量的类型从类型环境中取得。就像 CHECKED 中的过程那样，`module-proc` 的类型根据参数类型和主体类型得到。

模块过程的调用很像 CHECKED 中的过程调用，但有两个重要区别。

首先，操作数的类型（规则 IFACE-M-APP 中的 i_2）不必与参数类型 (i_1) 相同。我们只要求 $i_2 <: i_1$。这就够了，因为 $i_2 <: i_1$ 意味着满足接口 i_2 的任意模块都满足接口 i_1，也就能作为模块过程的参数。

其次, 在结果类型 t_1' 中, 我们把 m 代换为操作数 m_2。考虑 module-proc-eg 的例子。其中，我们用 `ints1` 和 `ints2` 调用模块过程 `to-int-maker`，其接口为

```
((ints : [opaque t
          zero : t
          succ : (t -> t)
          pred : (t -> t)
          is-zero : (t -> bool)])
  => [to-int : (from ints take t -> int)])
```

当我们用 `ints1` 调用 `to-int-maker`，代换而得的接口为

```
[to-int : (from ints1 take t -> int)]
```

当我们用 `ints2` 调用 `to-int-maker`，代换而得的接口为

```
[to-int : (from ints2 take t -> int)]
```

正合期望。

interface-of : *ModuleBody* × *Tenv* → *Iface*

```
(define interface-of
  (lambda (m-body tenv)
    (cases module-body m-body
      (var-module-body (m-name)
        (lookup-module-name-in-tenv tenv m-name))
      (defns-module-body (defns)
        (simple-iface
          (defns-to-decls defns tenv)))
      (app-module-body (rator-id rand-id)
        (let ((rator-iface
                (lookup-module-name-in-tenv tenv rator-id))
              (rand-iface
                (lookup-module-name-in-tenv tenv rand-id)))
          (cases interface rator-iface
            (simple-iface (decls)
              (report-attempt-to-apply-simple-module rator-id))
            (proc-iface (param-name param-iface result-iface)
              (if (<:-iface rand-iface param-iface tenv)
                (rename-in-iface
                  result-iface param-name rand-id)
                (report-bad-module-application-error
                  param-iface rand-iface m-body))))))
      (proc-module-body (rand-name rand-iface m-body)
        (let ((body-iface
                (interface-of m-body
                  (extend-tenv-with-module rand-name
                    (expand-iface rand-name rand-iface tenv)
                    tenv))))
          (proc-iface rand-name rand-iface body-iface))))))
```

图 8.15 PROC-MODULES 的检查器，第 1 部分

用这些规则，很容易写出 `interface-of` 的代码 ()。当我们检查 `module-proc` 的主体时，我们把参数添加到类型环境中，就好像它是位于顶层的模块。这段代码用过程 `rename-in-iface` 对得到的接口进行代换。

最后，我们扩展 `<:-iface`，处理新的类型。用来比较 `proc-iface` 的规则为

$$\frac{i_2 <: i_1 \quad i_1'\texttt{[}m'/m_1\texttt{]} <: i_2'\texttt{[}m'/m_2\texttt{]} \quad m'\text{不在}i_1'\text{或}i_2'\text{中}}{\texttt{((}m_1\ \texttt{:}\ i_1\texttt{)}\ \texttt{=>}\ i_1'\texttt{)} <: \texttt{((}m_2\ \texttt{:}\ i_2\texttt{)}\ \texttt{=>}\ i_2'\texttt{)}}$$

要使 `((`m_1 `:` i_1`) =>` i_1'`)`，满足第一个接口的模块 m_0 也必须满足第二个接口。这就是说，接口为 i_2 的任何模块都能作为参数，传给 m_0，m_0 产生的任何模块都满足 i_2'。

为满足第一个要求，我们要求 $i_2 <: i_1$。这保证了满足 i_2 的任何模块都能作为参数传给 m_0。注意逆序：我们说参数类型的*子类型判定* (*subtyping*) 是*逆变的* (*contravariant*)。

结果的类型呢？我们可以要求 $i_1' <: i_2'$。不幸的是，这行不通。i_1' 中，可能出现模块变量 m_1，i_2' 中，可能出现模块变量 m_2 的实例。所以，要比较它们，我们得将 m_1 和 m_2 重命名为新的模块变量 m'。一旦完成这一步，我们就能照常比较它们了。这就得出条件 $i_1'\texttt{[}m'/m_1\texttt{]} <: i_2'\texttt{[}m'/m_2\texttt{]}$。

判断这种关系的代码较为直白 ()。判断 $i_1'\texttt{[}m'/m_1\texttt{]} <: i_2'\texttt{[}m'/m_2\texttt{]}$ 时，我们扩展类型环境，给 m' 添加绑定。由于 i_1 的比 i_2 小，我们将 m' 与它关联起来。我们调用 `extend-tenv-with-module` 比较结果的类型时，还要调用 `expand-iface` 维持不变式。

现在，完成了。吃杯圣代吧，放些奶油口味、热浇汁口味、还有坚果口味的佐料。怎么组合不要紧，好吃就行！

<:-iface : $Iface \times Iface \times Tenv \to Bool$

```
(define <:-iface
  (lambda (iface1 iface2 tenv)
    (cases interface iface1
      (simple-iface (decls1)
        (cases interface iface2
          (simple-iface (decls2)
            (<:-decls decls1 decls2 tenv))
          (proc-iface (param-name2 param-iface2 result-iface2)
            #f)))
      (proc-iface (param-name1 param-iface1 result-iface1)
        (cases interface iface2
          (simple-iface (decls2) #f)
          (proc-iface (param-name2 param-iface2 result-iface2)
            (let ((new-name (fresh-module-name param-name1)))
              (let ((result-iface1
                      (rename-in-iface
                        result-iface1 param-name1 new-name))
                    (result-iface2
                      (rename-in-iface
                        result-iface2 param-name2 new-name)))
                (and
                  (<:-iface param-iface2 param-iface1 tenv)
                  (<:-iface result-iface1 result-iface2
                    (extend-tenv-with-module
                      new-name
                      (expand-iface new-name param-iface1 tenv)
                      tenv)))))))))))
```

图 8.16 PROC-MODULES 的检查器，第 2 部分

练习 8.24 [⋆] 目前，调用模块只能通过标识符。要是我们想检查调用 `(m1 (m2 m3))`，类型规则有什么问题?

练习 8.25 [⋆] 扩展 PROC-MODULES，像 ex3.21 那样，允许模块取多个参数。

练习 8.26 [⋆⋆] 扩展语言的模块主体，将模块调用的生成式改为

ModuleBody ::= `(`*ModuleBody* *ModuleBody*`)`
`app-module-body (rator rand)`

练习 8.27 [⋆⋆⋆] 在 PROC-MODULES 中，我们总要一遍又一遍写这种接口

```
[opaque t
 zero : t
 succ : (t -> t)
 pred : (t -> t)
 is-zero : (t -> bool)]
```

给程序添加语法，支持命名接口，这样我们就能写

```
interface int-interface = [opaque t
                           zero : t
                           succ : (t -> t)
                           pred : (t -> t)
                           is-zero : (t -> bool)]
module make-to-int
 interface
  ((ints : int-interface)
    => [to-int : from ints take t -> int])
 body
  ...
```

9 对象和类

在许多编程工作中，程序都要用接口管理某些状态。例如，文件系统内部状态的访问和修改只能通过系统的接口。状态常常涉及多个变量，为了维护状态的一致性，必须协同修改那些变量。因此，我们需要某种技术，确保组成状态的多个变量能协同更新。*面向对象编程* (*Object-oriented programming*) 技术正是为了完成这一任务。

在面向对象编程中，每个受管理的状态称为一个*对象* (*object*)。一个对象中存有多个量，称为*字段* (*field*)；有多个相关过程，称为*方法* (*method*)，方法能够访问字段。调用方法常被视为将方法名和参数当作消息传给对象；有时，又说这是从*消息传递* (*message-psasing*) 的视角看待面向对象编程。

在第 4 章那样的有状态语言中，过程也能体现用对象编程的优势。过程是一种对象，其状态包含于自由变量之中。闭包只有一种行为：拿参数调用它。例如，g-counter 的 g 控制计数器的状态，此状态的唯一操作就是递增。但更常见的是，一个对象具有多种行为。面向对象的编程语言具有这种能力。

一个方法通常需要管理多重状态，例如多个文件系统或程序中的多个队列。为便于方法共享，面向对象编程系统通常提供名为*类* (*class*) 的结构，用来指定某种对象的字段及方法。每个对象都创建为类的*实例* (*instance*)。

类似地，多个类可能有相似而不相同的字段和方法。为便于共享实现，面向对象编程语言通常支持*继承* (*inheritance*)，允许程序员增改某些方法的行为，添加字段，对现有类小做修改，就能定义新类。这时，由于新类的其他行为从原类继承而得，我们说新类*继承于* (*inherit from*) 或*扩展* (*extend*) 旧类。

不论是用代码建模真实世界中的对象还是人工层面的系统状态，一旦程序能由结合行为和状态的对象组成，其结构通常都清晰明了。将行为类似的对象

与同一个类关联起来，也是自然而然的。

真实世界中的对象通常兼具*状态*和*行为*，后者要么控制前者，要么受前者控制。例如，猫能吃，打呼噜，跳，躺下，这些活动都由猫当前的状态控制，包括有多饿，有多累。

对象和模块既相似，又不同。模块和类都提供了定义模糊类型的机制，但对象是一种具有行为的数据结构，模块只是一组绑定。同一个类可以有很多个对象；大多数模块系统没有提供相仿的能力。但 PROC-MODULES 这样的模块系统提供了更为灵活的方式来控制名字的可见性。模块和类能够相得益彰。

9.1 面向对象编程

本章，我们研究一种简单的面向对象语言，名为 CLASSES。CLASSES 程序包含一些类声明，然后是一个可能用到那些类的表达式。

展示了用这种语言写成的简单程序。它定义了继承于 `object` 的类 `c1`。类 `c1` 的每个对象都包含两个字段，名为 `i` 和 `j`。字段叫做*成员* (*member*) 或*实例变量* (*instance variable*)。类 `c1` 支持三个*方法*或*成员函数* (*member function*)，名为 `initialize`、`countup` 和 `getstate`。每个方法包含*方法名* (*method name*)，若干*方法变量* (*method var*)（又称*方法参数* (*method parameters*)），以及*方法主体* (*method body*)。方法名对应 `c1` 实例能够响应的*消息*种类。有时，我们称之为“`c1` 的方法 `countup`”。

```
class c1 extends object
 field i
 field j
 method initialize (x)
  begin
   set i = x;
   set j = -(0,x)
  end
 method countup (d)
  begin
   set i = +(i,d);
   set j = -(j,d)
  end
 method getstate () list(i,j)
let t1 = 0
    t2 = 0
    o1 = new c1(3)
in begin
    set t1 = send o1 getstate();
    send o1 countup(2);
    set t2 = send o1 getstate();
    list(t1,t2)
   end
```

图 9.1 简单的面向对象程序

本例中，类的每个方法都维护完整性约束或不变式 $i = -j$。当然，真实程序的完整性约束可能复杂得多。

中的程序首先初始化三个变量。t1 和 t2 初始化为 0。o1 初始化为类 c1 的一个对象。我们说这个对象是类 c1 的一个实例。对象通过 new 操作创建。它会触发调用类的方法 initialize，在本例中，是将对象的字段 i 设置为 3，字段 j 设置为 -3。然后，程序调用 o1 的方法 getstate，返回列表 (3 -3)。接着，它调用 o1 的方法 countup，将两个字段的值改为 5 和 -5，然后再次调用 getstate，返回 (5 -5)。最后，值 list(t1,t2)，即 ((3 -3) (5 -5))，成为整段程序的返回值。

```
class interior-node extends object
 field left
 field right
 method initialize (l, r)
  begin
   set left = l;
   set right = r
  end
 method sum () +(send left sum(),send right sum())
class leaf-node extends object
 field value
 method initialize (v) set value = v
 method sum () value
let o1 = new interior-node(
           new interior-node(
            new leaf-node(3),
            new leaf-node(4)),
           new leaf-node(5))
in send o1 sum()
```

图 9.2 求树叶之和的面向对象程序

解释了面向对象编程中的关键思想：*动态分发* (*dynamic dispatch*)。在这段程序中，树有两种节点，`interior-node` 和 `leaf-node`。通常，我们不知道是在给哪种节点发送消息。相反，每个节点接收 `sum` 消息，并用自身的 `sum` 方法做适当操作。这叫做*动态分发*。这里，表达式生成一棵树，有两个内部节点，三个叶节点。它将 `sum` 消息发给节点 `o1`；`o1` 将 `sum` 消息发给子树，依此类推，最终返回 12。这段程序也表明：所有方法都是互递归的。

方法主体可通过标识符 `self`（有时叫做 `this`）调用同一对象的其他方法，`self` 总是绑定于方法调用时的对象。例如，在

```
class oddeven extends object
 method initialize () 1
 method even (n)
  if zero?(n) then 1 else send self odd(-(n,1))
 method odd (n)
  if zero?(n) then 0 else send self even(-(n,1))
let o1 = new oddeven()
in send o1 odd(13)
```

中，方法 `even` 和 `odd` 递归调用彼此，因为它们执行时，`self` 绑定到包含二者的对象。这就像 ex3.37 中，用动态绑定实现递归。

9.2 继承

通过继承，程序员能够渐进地修改旧类，得到新类。在实践中，这十分有用。例如中的经典例子：有色点与点类似，但多了处理颜色的方法。

```
class point extends object
 field x
 field y
 method initialize (initx, inity)
  begin
   set x = initx;
   set y = inity
  end
 method move (dx, dy)
  begin
   set x = +(x,dx);
   set y = +(y,dy)
  end
 method get-location () list(x,y)
class colorpoint extends point
 field color
 method set-color (c) set color = c
 method get-color () color
let p = new point(3,4)
    cp = new colorpoint(10, 20)
in begin
    send p move(3,4);
    send cp set-color(87);
    send cp move(10,20);
    list(send p get-location(),   % 返回 (6 8)
         send cp get-location(),  % 返回 (20 40)
         send cp get-color())     % 返回 87
   end
```

图 9.3 继承的经典例子：colorpoint

```
class c1 extends object
 field x
 field y
 method initialize () 1
 method setx1 (v) set x = v
 method sety1 (v) set y = v
 method getx1 () x
 method gety1 () y
class c2 extends c1
 field y
 method sety2 (v) set y = v
 method getx2 () x
 method gety2 () y
let o2 = new c2()
in begin
    send o2 setx1(101);
    send o2 sety1(102);
    send o2 sety2(999);
    list(send o2 getx1(),  % 返回 101
         send o2 gety1(),  % 返回 102
         send o2 getx2(),  % 返回 101
         send o2 gety2())  % 返回 999
   end
```

图 9.4 字段遮蔽的例子

如果类 c_2 扩展类 c_1，我们说 c_1 是 c_2 的*父类* (*parent*) 或*超类* (*superclass*)，c_2 是 c_1 的*子类* (*child*)。在继承中，由于 c_2 定义为 c_1 的扩展，所以 c_1 必须在 c_2 之前定义。作为起始，语言还包含一个预先定义的类，名为 `object`，它没有任何方法或字段。由于类 `object` 没有 `initialize` 方法，因此无法用它创建对象。除 `object` 之外的所有类都有唯一父类，但可以有多个子类。因此，由 `extends` 得出的关系在类与类之间产生了树状结构，根为 `object`。因为每个类至多只有一个直接超类，这是一种*单继承* (*single-inheritance*) 语言。有些语言允许类继承自多个超类。*多继承* (*multiple inheritance*) 虽然强大，却不无问题。在练习中，我们会看到一些不便之处。

术语*继承*源于宗谱的类比。我们常常引申这一类比，说类的*祖先* (*ancestor*)（从类的父类到根类 `object`）和*后代* (*descendant*)。如果 c_2 是 c_1 的后代，可以说 c_2 是 c_1 的*子类* (*subclass*)，写作 $c_2 < c_1$。

如果类 c_2 继承自 c_1，除非在 c_2 中重新声明，c_1 的所有字段和方法都对 c_2 的方法可见。由于一个类继承了父类的所有方法和字段，任何能够使用父类实例的地方都可以使用子类实例。类似地，任何能够使用类实例的地方都可以使用其后代的实例。有时，这叫做*子类多态* (*subclass polymorphism*)。我们的语言选择这种设计，其他面向对象语言可能选择不同的可见性规则。

接下来，我们考虑重新声明类的字段或方法时会发生什么。如果 c_1 的某个字段在某个子类 c_2 中重新声明，新的声明*遮蔽* (*shadow*) 旧的，就像词法定界一样。例如。类 `c2` 的对象有两个名为 `y` 的字段：`c1` 中声明的和 `c2` 中声明的。`c1` 中声明的方法能看到 `c1` 的字段 `x` 和 `y`。在 `c2` 中，`getx2` 中的 `x` 指代 `c1` 的字段 `x`，但 `gety2` 中的 `y` 指代 `c2` 的字段 `y`。

如果类 c_1 的方法 m 在某个子类 c_2 中重新声明，我们说新的方法*覆盖* (*override*) 旧的方法。我们将方法声明所在的类称为方法的*持有类* (*host class*)。类似地，我们将表达式的持有类定义为表达式所在方法（如果有的话）的持有类。我们还将方法或表达式的超类定义为持有类的父类。

如果给类 c_2 的对象发送消息 m，应使用新的方法。这条规则很简单，结果却不简单。考虑下面的例子：

```
class c1 extends object
 method initialize () 1
 method m1 () 11
 method m2 () send self m1()
class c2 extends c1
 method m1 () 22
let o1 = new c1() o2 = new c2()
in list(send o1 m1(), send o2 m1(), send o2 m2())
```

我们希望 `send o1 m1()` 返回 11，因为 `o1` 是 `c1` 的实例。同样地，我们希望 `send o2 m1()` 返回 22，因为 `o2` 是 `c2` 的实例。那么 `send o2 m2()` 呢？方法 `m2` 直接调用方法 `m1`，但它调用的是哪个 `m1`？

动态分发告诉我们，应查看绑定到 `self` 的对象属于哪个类。`self` 的值是 `o2`，属于类 `c2`。因此，调用 `send self m1()` 应返回 22。

```
class point extends object
 field x
 field y
 method initialize (initx, inity)
  begin
   set x = initx;
   set y = inity
 end
 method move (dx, dy)
  begin
   set x = +(x,dx);
   set y = +(y,dy)
  end
 method get-location () list(x,y)
class colorpoint extends point
 field color
 method initialize (initx, inity, initcolor)
  begin
   set x = initx;
   set y = inity;
   set color = initcolor
  end
 method set-color (c) set color = c
 method get-color () color
let o1 = new colorpoint(3,4,172)
in send o1 get-color()
```

图 9.5 演示 super 必要性的例子

我们的语言还有一个重要特性：*超类调用* (*super call*)。考虑中的程序。我们在类 `colorpoint` 中重写了 `initialize` 方法，同时设置字段 `x`、`y` 和 `color`。但是，新方法的主体复制了原方法的代码。在我们的小例子中，这尚可接受，但在大型例子中，这显然是一种坏的做法（为什么?）。而且，如果 `colorpoint` 声明了字段 `x`，就没法初始化 `point` 的字段 `x`，正如 field-shadowing 的例子中，没法初始化第一个 `y` 一样。

解决方案是，把 `colorpoint` 的 `initialize` 方法主体中的重复代码替换为*超类调用*，形如 `super initialize()`。那么 `colorpoint` 中的 `initialize` 方法写作：

```
method initialize (initx, inity, initcolor)
 begin
  super initialize(initx, inity);
  set color = initcolor
 end
```

方法 m 主体中的超类调用 `super` n`(...)` 使用的是 m 持有类父类的方法 n。这不一定是 `self` 所指类的父类。`self` 所指类总是 m 持有类的子类，但不一定是同一个，[1]因为 m 可能在目标对象的某个祖先中声明。

要解释这种区别，考虑。给类 `c3` 的对象 `o3` 发送消息 `m3`，找到的是 `c2` 的方法 `m3`，它执行 `super m1()`。`o3` 的类是 `c3`，其父类是 `c2`，但方法的持有类是 `c2`，`c2` 的超类是 `c1`。所以，执行的是 `c1` 的方法 `m1`。这是*静态方法分发* (*static method dispatch*) 的例子。虽然进行超类方法调用的对象是 `self`，方法分发却是静态的，因为要调用的方法可以从程序文本中推断出来，与 `self` 所指类无关。

本例中，`c1` 的方法 `m1` 调用 `o3` 的方法 `m2`。这是普通方法调用，所以使用动态分发，找出的是 `c3` 的方法 `m2`，返回 33。

[1]任何类都是自身的子类，故有此说。••译注

```
class c1 extends object
 method initialize () 1
 method m1 () send self m2()
 method m2 () 13
class c2 extends c1
 method m1 () 22
 method m2 () 23
 method m3 () super m1()
class c3 extends c2
 method m1 () 32
 method m2 () 33
let o3 = new c3()
in send o3 m3()
```

图 9.6 解释 `super` 调用与 `self` 相互作用的例子