Principles of Programming Languages
Version 1.0.3

Mike Grant
Zachary Palmer
Scott Smith

http://pl.cs.jhu.edu/pl/book

This document was last compiled on February 13, 2024.

Contents

Chapter 1

Introduction

在这本书中，我们的目标是研究编程语言的基本概念，而不是学习一系列特定的语言。语言容易学习，难的是它们背后的概念。我们依次研究的基本特性包括高阶函数、以记录和变体形式的数据结构、可变状态、异常、对象和类以及类型。我们还研究语言实现，既包括语言解释器也包括语言编译器。全书我们为玩具语言编写了小型的解释器，在第8章我们编写了一个原则性的编译器。我们定义类型检查器来定义哪些程序是良好类型的，哪些不是。我们还通过*operational semantics*和*type systems*的概念，以更精确、数学的方式看待解释器和类型检查器。这两个最后的概念在历史上是从逻辑学家对编程的观点中演变而来的。

材料已从约翰霍普金斯大学为本科生、研究生和博士生开设的编程语言课程中使用的讲义演变而来 [21]。

虽然这本书使用了形式数学技术，如操作语义和类型系统，但它并不强调这些系统的性质证明。然而，我们将概述一些证明的直觉。

The OCaml Language

OCaml 编程语言 [15] 在整本书中都被使用，与本书相关的作业应使用 OCaml 编写。OCaml 是 ML 的一种现代方言，具有可靠、快速、免费，并且可以通过 `http://ocaml.org` 在几乎所有平台上使用的优点。

The FbDK

补充本书的是 **F♭** 开发套件，FbDK。它是一组用于设计和实验书中定义的玩具 **F♭** 和 **F♭SR** 语言的 OCaml 工具和解释器。它可在本书主页 `http://pl.cs.jhu.edu/ pl/book` 获取。

Background Needed

本书假设读者熟悉OCaml的基础知识，包括模块系统（但不包括对象，OCaml中的“O”）。除此之外，没有绝对的前提条件，但了解C、C++和Java有助于理解本书中许多用这些语言实现的主题。第8章中介绍的编译器以C代码为目标，因此需要具备基本的C语言知识来实现编译器。更模糊地说，一定的“数学成熟度”对于理解某些深奥的概念非常有帮助。因此，对数学、形式逻辑以及计算机科学中的其他基础主题（如自动机理论、语法和算法）的研究将大有裨益。

现在，请确保您的安全带已系好，坐直身体，放松，并享受旅程……

Chapter 2

Operational Semantics

2.1 A First Look at Operational Semantics

编程语言的 **syntax** 是该语言中表达式形成的规则集合。编程语言的 **semantics** 是那些表达式的 *meaning*。

存在几种语言语义的形式。公理语义是编程语言中一组公理真理。指称语义涉及将程序建模为静态数学对象，即具有特定属性的集合论函数。然而，我们将关注一种称为操作语义的语义形式。

操作语义是编程语言 *execu- tion* 的数学模型。本质上，它是一个用数学定义的解释器。然而，操作语义比解释器更精确，因为它是在数学上定义的，而不是基于解释器所编写的编程语言的意义。这听起来可能像是一种繁琐的区分，但解释器解释例如语言 `if` 的语句时，使用的是解释器所编写的语言的 `if` 语句。这在某种程度上是对 `if` 的循环定义。正式来说，我们可以如下定义操作语义。

Definition 2.1 (操作语义). *An **operational semantics** for a programming language is a mathematical definition of its computation relation, $e \Rightarrow v$, where e is a program in the language.*

$e \Rightarrow v$ 数学上是一个语言表达式e和语言值v之间的2元关系。整数和布尔值是值。函数也是值，因为它们不计算任何东西。e和v是**metavariables**，这意味着它们表示任意表达式或值，不应与程序中作为一部分的（常规）变量混淆。

编程语言的运算语义是理解语言中表达式精确含义的一种方法。它是编写编译器和解释器时使用的语言形式规范，并允许我们严格验证关于语言的事物。

2.2 BNF grammars and Syntax

在探讨意义之前，我们需要退一步，首先精确地定义语言语法。这是通过形式语法来完成的。*Backus-Naur Form* (BNF) 是一种用于定义语言语法的标准语法形式。你很可能对 BNF 很熟悉，因为它在入门课程中经常被教授，但如果不熟悉，我们提供简要概述。所有 BNF 语法都包含 *terminals*、*nonterminals (aka syntactic categories)* 和产生式规则。终结符传统上使用小写字母标识；非终结符使用大写字母标识。产生式规则描述了非终结符的定义方式。产生式规则的一般形式是：

$$\langle \text{nonterminal} \rangle ::= \langle \text{form 1} \rangle \mid \cdots \mid \langle \text{form n} \rangle$$

每个“形式”都描述了一种特定的语言形式——也就是说，一个由终结符和非终结符组成的字符串。语言中的 *term* 是一个与这些规则之一（传统上是第一个）的描述相匹配的终结符字符串。

例如，考虑语言Sheep。令$\{S\}$为非终结符集合，$\{a, b\}$为终结符集合，语法定义如下：

$$S ::= b \mid Sa$$

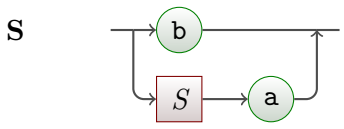
请注意，这是一个递归定义。Sheep中的项的例子有

$$b, ba, baa, baaa, baaaa, \ldots$$

这意味着任何以字符 b 开头，后跟零个或多个 a 字符的字符串都是 Sheep 中的一个项。以下是一些不是 SHEEP 中的项的例子：

- a条款在Sheep中必须以一个b开头。
- $bbaaa$羊不允许一个词中出现多个 b 字符。
- $baahh$ 不是Sheep中的终端。
- $SaaaS$ 是 Sheep 中的一个非终结符。项可能不包含非终结符。

另一种表达语法的方法是使用一个 **syntax diagram**。语法图通过视觉形式描述语法，而不是以文本形式。例如，以下是为语言 Sheep 制作的语法图：



上述语法图描述了Sheep语言的所有术语。要生成S的形式，从图的左侧开始移动，直到到达右侧。矩形节点表示非终结符，而圆形节点表示终结符。到达非终结符节点时，必须使用该非终结符构建一个术语以继续。

作为另一个例子，考虑语言青蛙。令 $\{F, G\}$ 为非终结符集合，$\{r, i, b, t\}$ 为终结符集合，语法定义如下：

$F ::= rF \mid iG ::= bG \mid bF \mid t$

G

请注意，这是一个相互递归的定义。注意，每个产生式规则定义了一个句法类别。FROG中的术语包括：

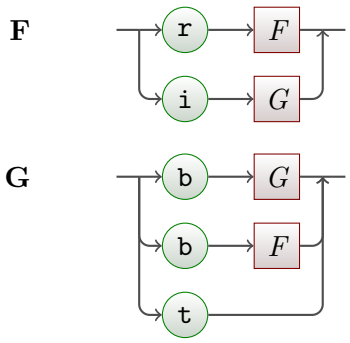
$$ibit, ribbit, ribibibbbit \ldots$$

以下术语不是Frog中的术语：

- rbt当Frog中的一个项以r开头时，下一个非终结符是F。非终结符F只能扩展为rF或iG，它们都不以b开头。因此，没有以rb开头的字符串是Frog中的项。

- $rabbita$ 不是 Frog 中的终端。
- $rrrrrrFF$ 是 Frog 中的一个非终结符；项中可能不包含非终结符。
- bit仅以 b 开头的形式出现在 G 的定义中。由于 F 是第一个定义的非终结符，Frog 中的术语必须匹配 F (，它没有以 b) 开头的任何形式。

以下语法图描述了青蛙：



2.2.1 Operational Semantics for Logic Expressions

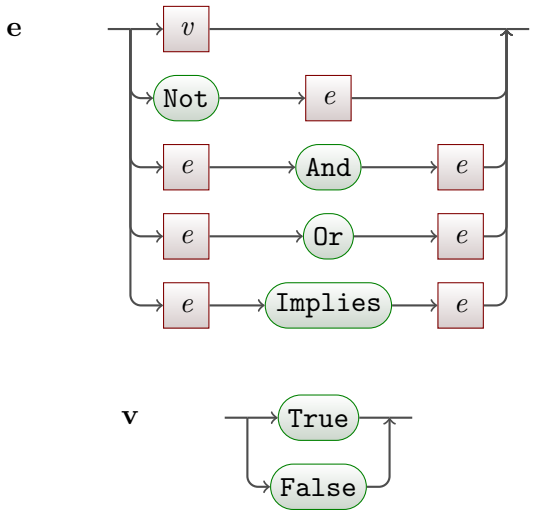
为了了解操作语义是什么以及它是如何定义的，我们现在将考察一个非常简单的语言的操作语义：没有变量的命题布尔逻辑。该语言的语法如下。表达式 e 被递归定义为由值 `True` 和 `False` 以及表达式 e `And` e、e `Or` e、e `Implies` e 和 `Not` e 组成.[1] 这种语法被称为 **concrete syntax**,

[1]Throughout the book we use syntax very similar to OCaml in our toy languages, but with the convention of capitalizing keywords to avoid potential conflicts with the OCaml language.

因为它是描述语言中表达式文本表示的语法。我们可以用以下BNF语法来表示它：

$$
\begin{array}{lcll}
e & ::= & v \mid \texttt{Not}\ e \mid e\ \texttt{And}\ e \mid e\ \texttt{Or}\ e \mid e\ \texttt{Implies}\ e \mid (e) & \textit{expressions} \\
v & ::= & \texttt{True} \mid \texttt{False} & \textit{values}
\end{array}
$$

以下是一个等价的语法图：



注意，上述语法通过使用小写字母表示非终结符而多少打破了传统。终结符以固定宽度字体打印。这样做的原因是与我们在下面的操作语义中将要使用的元变量保持一致，这一点很快就会变得清楚。

我们现在可以讨论布尔语言的运算语义。运算语义以逻辑规则的形式编写，这些规则以一系列位于水平线上的前提条件开头，并在其下方写出结论。例如，逻辑规则

$$\textit{(Apple Rule)} \qquad \frac{\text{Red}(x) \quad \text{Shiny}(x)}{\text{Apple}(x)}$$

指示如果某物是红色且闪亮的，那么那个东西就是苹果。这当然是不正确的；存在许多红色、闪亮的东西，它们不是苹果。尽管如此，这是一个有效的逻辑陈述。在我们的工作中，我们将定义与编程语言相关的逻辑规则；因此，我们控制着规则构建的空间。只要编程语言有数学基础，我们就不必 necessarily关心直观感觉。

操作语义规则讨论代码片段如何评估。例如，让我们考虑 `And` 规则。我们可以为 `And` 定义以下规则：

$$\textit{(And Rule (Try 1))} \qquad \frac{}{\texttt{True And False} \Rightarrow \texttt{False}}$$

此规则表示布尔语言代码 `True And False` 的评估结果为 `False`。上方没有任何先决条件意味着不需要满足任何条件；此操作语义规则始终为真。上方没有任何内容的规则被称为 **axioms**，因为它们没有先决条件，所以结论始终成立。

通常来说，这并不很有用。它只评估一个非常特定的程序。这个规则并没有描述如何评估程序 `True And True`，例如。为了将我们的规则推广到描述整个语言，而不仅仅是语言中的特定术语，我们必须使用元变量。

为了与上述 BNF 语法保持一致，我们使用以 e 开头的元变量来表示表达式，以 v 开头的元变量来表示值。我们现在准备尝试使用以下新规则来描述 `And` 操作符的每个方面：

(And Rule (Try 2)) $$\frac{}{v_1 \texttt{ And } v2 \Rightarrow v_1 \text{ 和 } v_2 \text{ 的逻辑与}}$$

使用此规则，我们可以成功评估 `True And False`、`True and True` 以及如此等等。请注意，我们已使用文本描述来表示表达式 v_1 `And` v_2 的值；这是允许的，尽管在更复杂的语言中，大多数规则不会使用此类描述。

我们很快就会遇到我们方法中的局限性。考虑程序 `True And (False And True)`。如果我们尝试将上述规则应用于该程序，我们将得到 $v_1 =$ `True` 和 $v_2 =$ `(False And True)`。这两个值不能应用于逻辑与，因为 `(False and True)` 不是一个布尔值；它是一个表达式。我们的布尔语言规则不允许操作数是表达式的 `And` 的情况。因此，我们再次尝试该规则：

(And Rule (Try 3)) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1 \texttt{ And } e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

这条规则几乎完全符合我们的期望；事实上，规则本身是完整的。直观上，这条规则表示 e_1 `And` e_2 等于由 e_1 和 e_2 表示的值的逻辑与。但再次考虑程序 `True And False`，我们期望它评估为 `False`。我们可以看到 $e_1 =$ `True` 和 $e_2 =$ `False`，但我们的评估关系并没有将 v_1 或 v_2 与任何值相关联。这是因为，严格来说，我们并不知道 `True` $\Rightarrow$ `True`。

当然，我们希望是这样，并且由于我们正在定义语言，我们可以让它成为现实。我们只需要在操作语义规则中声明它即可。

(Value Rule) $$\frac{}{v \Rightarrow v}$$

上方的值规则是一个公理，声明任何值总是评估为自身。这满足我们的要求，并允许我们使用 `And` 规则。使用这个

形式逻辑方法，我们现在可以证明 `True And (False And True) ⇒ False` 如下：

$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{\dfrac{}{\texttt{False} \Rightarrow \texttt{False}} \quad \dfrac{}{\texttt{True} \Rightarrow \texttt{True}}}{\texttt{False And True} \Rightarrow \texttt{False}}}{\texttt{True And (False And True)} \Rightarrow \texttt{False}}$$

一个人可以将上述 **proof tree** 读取为解释 `True And (False And True)` 评估为 `False` 的原因。我们可以选择如下读取该证明：*"`True And (False And True)` evaluates to `False` by the `And` rule because we know `True` evaluates to `True`, that `False And True` evaluates to `False`, and that the logical and of true and false is false. We know that `False And True` evaluates to `False` by the `And` rule because `True` evaluates to `True`, `False` evaluates to `False`, and the logical and of true and false is false."*

一个与上述类似的非正式格式是：

`True And (False And True) ⇒ False`, 因为根据 `And` 规则
`True ⇒ True`, 并且
`(False And True) ⇒ False`, 后者因为
`True ⇒ True`, 和 `False ⇒ False`

这三个表示法的重要之处在于它们都在描述一个证明树。证明树由节点组成，这些节点代表应用逻辑规则，其子节点为前提条件。为了完成我们的布尔语言，我们使用一套完整的操作语义规则定义了 ⇒ 关系：

(Value Rule) $$\dfrac{}{v \Rightarrow v}$$

(Not Rule) $$\dfrac{e \Rightarrow v}{\texttt{Not}\ \ e \Rightarrow \text{the negation of } v}$$

(And Rule) $$\dfrac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1\ \ \texttt{And}\ \ e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

规则 `Or` 和 `Implies` 的规则留给读者作为练习（参见练习2.4）。这些规则形成了一个在数学逻辑中发现的 **proof system**。逻辑规则表达无可争议的逻辑真理。**proof** $e \Rightarrow v$ 等于构建一个规则应用序列，使得对于任何给定的规则应用，线上方的项目在序列中先出现，并且最终规则应用是 $e \Rightarrow v$。证明在结构上是一棵树，其中每个节点都是一个规则，子树规则有结论，与父规则的假设完全匹配。对于 $e \Rightarrow v$ 的证明树，根规则作为结论有 $e \Rightarrow v$。注意 *all leaves of a proof tree must be axioms*。一个具有非公理叶子的树不是一个证明。

注意上述证明树是如何表达这个逻辑表达式如何计算的。$e \Rightarrow v$的证明与e的执行产生结果v的方式密切相关。唯一的区别是，“执行”从e开始并产生v，而证明树描述的是e和v之间的关系，而不是从e到v的函数。

Lemma 2.1. *The boolean language is* ***deterministic****: if $e \Rightarrow v$ and $e \Rightarrow v'$, then $v = v'$.*

Proof. 通过证明树高度的归纳。 □

Lemma 2.2. *The boolean language is* ***normalizing****: For all boolean expressions e, there is some value v where $e \Rightarrow v$.*

Proof. 通过 e 的大小进行归纳。 □

当可以为某个程序 $e \Rightarrow v$ 构造一个证明 e 时，我们称 e **converges**。当不存在这样的证明时，e **diverges**。因为布尔语言是归一化的，所以该语言中的所有程序都被认为是收敛的。有些语言（如 OCaml）不是归一化的；存在语法上合法的程序，但没有评估证明存在。OCaml 中一个发散的程序示例是 `let rec f x = f x in f 0;;`。

2.2.2 Abstract Syntax

我们的操作语义规则使用元变量以具体语法的形式表达了评估关系。例如，中缀运算符 `And` 以文本格式出现。这对于人类阅读来说是一个很好的表示，因为它符合我们的直觉；然而，它并不是一个理想的计算表示。我们将 `True And False` 读作“对操作数 `True` 和 `False` 执行逻辑与运算”。我们将 `True And (False And True)` 读作“对操作数 `False` 和 `True` 执行逻辑与运算，然后对操作数 `True` 和上一步操作的结果执行逻辑与运算。”如果我们需要编写（例如解释器）与我们的语言一起工作的程序，我们需要一个更准确地描述我们如何思考程序表示。

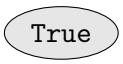
语言中的 **abstract syntax** 是这样的表示。在抽象语法中的一个术语表示为一个 **syntax tree**，其中每个要执行的操作是一个节点，每个操作数是该节点的子节点。为了表示布尔语言的抽象语法树，我们可能使用以下 OCaml 数据类型：

```
type boolexp =
    True | False |
    Not of boolexp |
    And of boolexp * boolexp |
    Or of boolexp * boolexp |
    Implies of boolexp * boolexp;;
```

为了理解抽象和具体语法之间的关系，考虑以下示例：

Example 2.1.

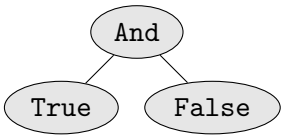
Concrete:
`True`



Abstract:
`True`

Example 2.2.

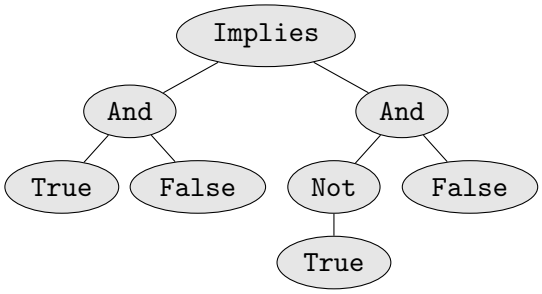
Concrete:
`True And False`



Abstract:
`And(True, False)`

Example 2.3.

Concrete:
`(True And False) Implies`
`((Not True) And False)`



Abstract:
`Implies( And(True,False) ,`
`And(Not(True),False) )`

存在一种简单直接的关系，语言的具体语法和抽象语法之间。如上所述，抽象语法是一种更直接表示正在执行的操作的形式，而具体语法是实际表达操作的形式。编译或解释程序的过程的一部分是将具体语法术语（源文件）转换为抽象语法术语（AST），以便对其进行操作。我们定义一个关系 $[\![c]\!] = a$ 来将具体语法形式 c 映射到抽象语法形式 a (，在这种情况下，对于布尔语言)：

$$
\begin{aligned}
[\![\texttt{True}]\!] &= \texttt{True} \\
[\![\texttt{False}]\!] &= \texttt{False} \\
[\![\texttt{Not } e]\!] &= \texttt{Not}(e) \\
[\![e_1 \texttt{ And } e_2]\!] &= \texttt{And}([\![e_1]\!], [\![e_2]\!]) \\
[\![e_1 \texttt{ Or } e_2]\!] &= \texttt{Or}([\![e_1]\!], [\![e_2]\!]) \\
[\![e_1 \texttt{ Implies } e_2]\!] &= \texttt{Implies}([\![e_1]\!], [\![e_2]\!])
\end{aligned}
$$

例如，此关系表示以下内容：

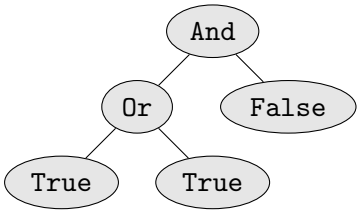
```
⟦(True And False) Implies ((Not True) And False)⟧
  =  Implies( ⟦True And False⟧, ⟦(Not True) And False⟧ )
  =  Implies( And(⟦True⟧, ⟦False⟧), And(⟦Not True⟧, ⟦False⟧) )
  =  Implies( And(True, False), And(Not(⟦True⟧), False) )
  =  Implies( And(True, False), And(Not(True), False) )
```

我们给出的语法在某种程度上是模糊的，因为某些具体表达式的解析树有多个，但我们隐含地假设通常的运算符优先级适用，并且与绑定比或绑定更紧，比蕴含绑定更紧。考虑以下示例：

Example 2.4.

Concrete:
`(True Or True) And False`

Abstract:
`And(Or(True,True),False)`

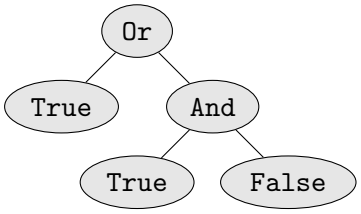


$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{}{\texttt{True} \Rightarrow \texttt{True}}}{\texttt{True Or True} \Rightarrow \texttt{True}} \quad \dfrac{}{\texttt{False} \Rightarrow \texttt{False}}}{\texttt{True Or True And False} \Rightarrow \texttt{False}}$$

Example 2.5.

Concrete:
`True Or (True And False)`

Abstract:
`Or(True,And(True,False))`



$$\dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{\dfrac{}{\texttt{True} \Rightarrow \texttt{True}} \quad \dfrac{}{\texttt{False} \Rightarrow \texttt{False}}}{\texttt{True And False} \Rightarrow \texttt{False}}}{\texttt{True Or (True And False)} \Rightarrow \texttt{True}}$$

表达式在示例2.4中将评估为`False`，因为必须首先评估`Or`操作，然后使用结果评估`And`操作。另一方面，示例2.5执行操作的顺序相反。请注意，尽管如此，在两个示例中，括号本身在抽象语法中不再明显存在。这是因为它们在AST的结构中隐式表示；也就是说，如果形式的具体语法中没有括号，示例2.5中的AST就不会具有现在的形状。

简而言之，括号仅仅改变了表达式的分组方式。在示例2.5中，我们只能将*entire expression*与Or规则相匹配；显然，And规则无法匹配，因为左括号将是e_1的一部分，而右括号将是e_2 (的一部分，并且没有匹配的括号的表达式没有意义)。同样但不太明显的是，示例2.4只能匹配And规则；结合性隐式地强制Or规则首先发生，将整个表达式分配给And运算符进行评估。这种区别在示例的AST中得到了清晰和相应的表示，这是操作语义适用性的关键。

2.2.3 Operational Semantics and Interpreters

如上所述，操作语义与用OCaml编写的实际解释器之间存在非常紧密的关系。给定通过关系 $\Rightarrow$ 定义的运算语义，存在相应的（OCaml）评估函数 `eval`。

Definition 2.2 (忠实实施). *A (OCaml) interpreter function eval faithfully implements an operational semantics $e \Rightarrow v$ if:*
$e \Rightarrow v$ if and only if `eval`$(\llbracket e \rrbracket)$ *returns result* $\llbracket v \rrbracket$.

为了展示这种关系，我们将演示在 OCaml 中创建一个 `eval` 函数。为了简单起见，我们的函数初稿将只包含 `And` 规则和值规则：

```
let eval exp =
  match exp with
    | True -> True
    | False -> False
    | And(exp0,exp1) ->
      begin
        match (exp0, exp1) with
          | (True,True) -> True
          | (_,False) -> False
          | (False,_) -> False
      end
```

首先看起来，这个函数似乎具有我们期望的行为。`True` 等于 `True`，`False` 等于 `False`，`True And False` 等于 `False`。然而，这并不是一个完整的实现。

要找出原因，考虑具体项 `True And True And True`。正如我们之前所看到的，这对应于抽象项 `And(And(True,True),True)`。当 `eval` 函数接收该值作为其参数时，它将该值与情况 `And` 匹配，并将 `exp0` 和 `exp1` 分别定义为 `And(True,True)` 和 `True`。然后我们进入内部匹配，这个匹配失败了！由于 `exp0` 是一个完整的表达式，而不是仅仅一个值，正如匹配表达式所期望的，因此没有任何项可以匹配元组 `(exp0,exp1)`。

回顾我们尝试编写上面的 `And` 规则，我们可以看到为什么这个 `eval` 函数是有缺陷的：这个函数版本没有将 `And` 的操作数视为表达式 - 它期望它们是值。因此，我们可以看到 `And` 子句

在我们的函数是对我们的 And 规则中的 *Try 2* 的忠实实现，我们正是由于它无法处理嵌套表达式而拒绝了这个规则。

我们如何纠正这个问题？我们正在尝试编写我们最终 And 规则的忠实实现，该实现依赖于 And 规则操作数的评估。因此，在我们的实现中，我们必须评估这些操作数；我们通过声明我们的评估函数是递归的来实现这一点。

```
let rec eval exp =
  match exp with
    | True -> True
    | False -> False
    | And(exp0,exp1) ->
      begin
        match (eval exp0, eval exp1) with
          | (True,True) -> True
          | (_,False) -> False
          | (False,_) -> False
      end
```

观察上述代码，我们改动很小。我们将 eval 函数修改为递归。我们还为 And 操作的每个操作数添加了对 eval 的调用。仅此调用就足以解决问题；评估这些参数的过程代表了 And 规则的 $e_1 \Rightarrow v_1$ 和 $e_2 \Rightarrow v_2$ 预条件，而将结果值用于元组中则导致匹配与 v_1 和 v_2 而不是 e_1 和 e_2 进行。上述代码是值规则和 And 规则的忠实实现。

我们现在可以通过继续以相同的形式执行 eval 函数来完成布尔语言解释器:

```
let rec eval exp =
  match exp with
    True -> True
  | False -> False
  | Not(exp0) -> (match eval exp0 with
      True -> False
    | False -> True)
  | And(exp0,exp1) -> (match (eval exp0, eval exp1) with
      (True,True) -> True
    | (_,False) -> False
    | (False,_) -> False)

  | Or(exp0,exp1) -> (match (eval exp0, eval exp1) with
      (False,False) -> False
    | (_,True) -> True
    | (True,_) -> True)

  | Implies(exp0,exp1) -> (match (eval exp0, eval exp1) with
      (False,_) -> True
```

```
  | (True,True) -> True
  | (True,False) -> False)
```

操作语义与解释器的唯一区别是解释器是一个函数。我们从规则的左下角表达式开始，使用解释器递归地产生规则上方的值，最后计算并返回规则下方的值。

注意，上述布尔语言解释器忠实实现了其操作语义：$e \Rightarrow v$ 当且仅当 `eval`($[\![e]\!]$) 返回 $[\![v]\!]$ 作为结果。我们将在整本书中在这两种形式之间来回切换。操作语义形式被使用，因为它独立于任何特定的编程语言。解释器形式是有用的，因为我们可以在非平凡数量的步骤中解释真实程序，这是在操作语义上“纸上谈兵”难以做到的。

Definition 2.3 (元循环解释器)**.** *A* ***metacircular interpreter*** *is an interpreter for (possibly a subset of) a language x that is written in language x.*

元循环解释器让你对一种语言的工作方式有所了解，但遭受了练习2.5中暗示的非基础性问题。Lisp的元循环解释器（即用Lisp编写的Lisp解释器）是经典的编程语言理论练习。

2.3 The F♭ Programming Language

现在我们已经看到如何定义和理解操作语义，我们将开始研究我们的第一种编程语言：**F♭**。**F♭**是一种shunk（扁平化）的纯*func- tional*编程语言。[2]它有整数、布尔值和更高阶的匿名函数。在大多数方面，**F♭**比OCaml要弱得多：没有实数、列表、类型、模块、状态或异常。

F♭ 未类型化，实际上比OCaml更强大。可以在 **F♭** 中编写一些不会产生运行时错误的程序，但在OCaml中无法通过类型检查。类型系统在第6章中讨论。因为没有类型，**F♭** 中可能会发生运行时错误，例如应用 `(5 3)`。

尽管非常简单，**F♭** 仍然是 **Turing-complete**。图灵完备性的概念已经被定义为多种等效方式。以下是一种定义方式：

Definition 2.4 (图灵完备性)**.** *A computational model is* ***Turing-complete*** *if every* ***partial recursive function*** *can be expressed within it.*

当然，这个定义需要部分递归函数（也称为可计算函数）的定义。不深入讨论基础材料，以下这个相对非正式的定义就足够了：

[2]Also, any readers familiar with the programming language C♯ as well as basic music theory should find this at least a bit humorous.

Definition 2.5 (部分递归函数)**.** *A function is a partial recursive function if an algorithm exists to calculate it which has the following properties:*

- *The algorithm must have as its input a finite number of arguments.*
- *The algorithm must consist of a finite number of steps.*
- *If the algorithm is given arguments for which the function is defined, it must produce the correct answer within a finite amount of time.*
- *If the algorithm is given arguments for which the function is not defined, it must either produce a clear error or otherwise not terminate. (That is, it must not appear to have produced an incorrect value for the function if no such value is defined.)*

上述偏递归函数的定义是数学性的，因此不涉及执行特定的细节，如存储空间或实际执行时间。不对计算机可能需要的内存量、参数的范围或函数在宇宙热寂之前终止（只要对于函数定义的所有输入最终都会终止）施加任何限制。

实用意义在于：没有一种计算可以用另一种确定性编程语言表达，而无法在 **F♭**.[3] 中表达。事实上，**F♭** 即使没有数字或布尔值也是图灵完备的。这种只有函数和应用的语言被称为纯λ演算，在2.4.4节中简要讨论。没有确定性编程语言可以计算比部分递归函数更多的内容。

2.3.1 F♭ Syntax

我们将采用与上面定义布尔语言相同的方法来定义 **F♭**。我们首先描述 **F♭** 语言的语法以定义其 **concrete syntax**；**abstract syntax** 将在 2.3.7 节中延迟定义。我们可以使用以下 BNF 定义 **F♭** 的语法：

[3] This does not guarantee that the **F♭** representation will be pleasant. Programs written in **F♭** to perform even fairly simplistic computations such as determining if one number is less than another are excruciating, as we will see shortly.

x	::=	(a | b | ... | z)	*lower-case letters*
		(A | B | ... | Z	*capital letters*
		| a | b | ... | z	*lower-case letters*
		| 0 | 1 | ... | 9	*digits*
		| _ | ' | $\cdots$)$*$	*other characters*
v	::=	x	*variable values*
		| True | False	*boolean values*
		| 0 | 1 | -1 | 2 | -2 | ...	*integer values*
		| Fun $x \to e$	*function values*
e	::=	v	*value expressions*
		| (e)	*parenthesized expressions*
		| e And e | e Or e | Not e	*boolean expressions*
		| e + e | e - e | e = e |	*numerical expression*
		| e e	*application expression*
		| If e Then e Else e	*conditional expressions*
		| Let x = e In e	*let expression*
		| Let Rec f x = e In e	*recursive let expression*

请注意，根据上述BNF，我们将使用元变量 e、v 和 x 分别表示表达式、值和变量。注意最后一点：元变量 x 指的是一个任意的 **F♭** 变量，不一定是指 **F♭** 变量 `x`。

关联性在 **F♭** 中与 OCaml 非常相似。例如，函数应用是左结合的，这意味着 `a b c` 与 `(a b) c` 的含义相同。与任何语言一样，这种结合性很重要，因为它影响源代码如何被解析成抽象语法树（AST）。

2.3.2 Variable Substitution

F♭ 的主要特性是高阶函数，这也引入了变量。回忆一下，程序是通过重写它们来计算的：

```
(Fun x -> x + 2)(3 + 2 + 5) ⇒ 12
    因为 3 + 2 + 5 ⇒ 10 因
    为 3 + 2 ⇒ 5 和
                和
    5 + 5 ⇒ 10
    10 + 2 ⇒ 12
```

注意在这个例子中，参数被替换为变量在主体中——这给了我们一个重写解释器。换句话说，**F♭** 函数通过将实际参数替换为形式参数来计算；例如，

```
(Fun x -> x + 1) 2
```

将2代入函数体中的$x+1$，即计算$2+1$。这不是一种非常高效的计算方法，但它是一种非常简单和准确的方法，这正是操作语义学所涉及的内容——清楚地、明确地描述程序应该如何计算。

Bound and Free Occurrences of Variables 我们需要小心定义变量替换的方式。例如，

```
(Fun x -> Fun x -> x) 3
```

不应评估为 `Fun x -> 3`，因为内部 `x` 受内部参数的约束。为了正确形式化这个概念，我们需要做出以下定义。

Definition 2.6 (变量出现)**.** *A variable use x* ***occurs*** *in e if x appears somewhere in e. Note we refer only to variable*使用*, not definitions.*

Definition 2.7 (边界出现)**.** *Any occurrences of variable x in the expression*

`Fun` x `->` e

are ***bound****, that is, any free occurrences of x in e are bound occurrences in this expression. Similarly, in the expression*

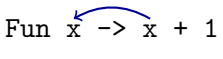
`Let Rec` f x `=` e_1 `In` e_2

occurrences of f and x are bound in e_1 and occurrences of f are bound in e_2. Note that x is not bound in e_2, but only in e_1, the body of the function.

Definition 2.8 (免费出现)**.** *A variable x occurs* ***free*** *in e if it has an occurrence in e which is not a bound occurrence.*

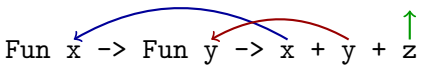
让我们看看一些关于约束变量与自由变量的例子发生 参考文献。

Example 2.6.



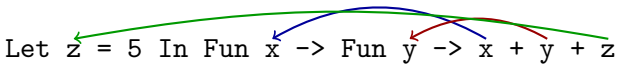
`x` is bound in the body of this function.

Example 2.7.



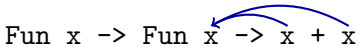
`x` and `y` are bound in the body of this function. `z` is free.

Example 2.8.



`x`, `y`, and `z` are all bound in the body of this function. `x` and `y` are bound by their respective function declarations, and `z` is bound by the `Let` statement. Note that, while **F♭** contains `Let` as syntax, it can be defined as a macro (see Section 2.3.4 below). Binding rules work similarly for `Fun`s and `Let` statements.

Example 2.9.



`x` is bound in the body of this function. Note that both `x` usages are bound to the inner variable `x`.

Definition 2.9 (闭式表达式)**.** *An expression e is closed if it contains no free variable occurrences. All programs we execute are closed (no link-time errors) – non-closed programs don't diverge, we can't even contemplate executing them because they are not in the domain of the evaluation relation.*

在上述例子中，例子2.6、2.8和2.9是封闭表达式。例子2.7不是一个封闭表达式。

现在我们已经理解了约束变量和自由变量，我们可以给出变量替换的正式定义。

Definition 2.10 (变量替换)**.** *Given a closed expression e', the variable substitution of x for e' in e, denoted $e[e'/x]$, is the expression resulting from the operation of replacing all free occurrences of x in e with e'.*

这里是一个等价的归纳定义的替换:

$$
\begin{array}{rcll}
x[v/x] & = & v & \\
x'[v/x] & = & x' & x \neq x' \\
(\texttt{Fun}\ x \to e)[v/x] & = & (\texttt{Fun}\ x \to e) & \\
(\texttt{Fun}\ x' \to e)[v/x] & = & (\texttt{Fun}\ x' \to e[v/x]) & x \neq x' \\
(\texttt{Let}\ x\ \texttt{=}\ e_1\ \texttt{In}\ e_2)[v/x] & = & \texttt{Let}\ x\ \texttt{=}\ e_1[v/x]\ \texttt{In}\ e_2 & \\
(\texttt{Let}\ x'\ \texttt{=}\ e_1\ \texttt{In}\ e_2)[v/x] & = & \texttt{Let}\ x'\ \texttt{=}\ e_1[v/x]\ \texttt{In}\ e_2[v/x] & x \neq x' \\
n[v/x] & = & n \text{ for } n \in \mathbb{Z} & \\
\texttt{True}[v/x] & = & \texttt{True} & \\
\texttt{False}[v/x] & = & \texttt{False} & \\
(e_1 + e_2)[v/x] & = & e_1[v/x] + e_2[v/x] & \\
(e_1\ \texttt{And}\ e_2)[v/x] & = & e_1[v/x]\ \texttt{And}\ e_2[v/x] & \\
 & \vdots & &
\end{array}
$$

例如，让我们考虑一个函数的简单应用：`(Fun x -> x + 1) 2`。我们知道，为了评估这个函数，我们只需将函数体中所有的`x`实例替换为`2`。这可以写成`(x + 1)[2/x]`。这将导致我们评估`2 + 1`，进而得到结果`3`。

虽然这看起来可能不是一个令人启迪的认识，但这一事实在数学上是可辨别的，这为我们进行更复杂的替换提供了一个起点。考虑以下例子。

Example 2.10.

Expression:

```
(Fun x -> Fun y -> (x + x + y)) 5
```

Substitution:

```
(Fun y -> (x + x + y))[5/x]
= (Fun y -> (x + x + y)[5/x])
= Fun y -> (x[5/x] + x[5/x] + y[5/x])
= Fun y -> (5 + 5 + y)
```

α-conversion

在示例2.9中，我们看到了两个变量可以共享同一个名字的情况。当然，变量本身是不同的，它们遵循与OCaml中相同的范围规则**F**♭。但是，阅读频繁使用相同变量名表示不同变量的表达式可能会非常令人困惑。例如，考虑以下表达式。

```
Let Rec f x =
    If x = 1 Then
        (Fun f -> f (x - 1)) (Fun x -> x)
    Else
        f (x - 1)
In f 100
```

这个表达式是如何评估的？由于复杂的绑定，仅通过观察它有点难以理解。我们可以通过使用不同的名称使其更容易理解。α-转换是将一个变量定义及其所有绑定到的出现替换为不同名称的变量的过程。

Example 2.11.

```
Fun x -> x + 1
```

变为

```
Fun z -> z + 1
```

示例2.11展示了x替代z的简单情况。对于多次使用相同变量名的情况，我们可以采用相同的方法。考虑示例2.12，其中内部变量x被α转换为z。

Example 2.12.

```
Fun x -> Fun x -> x
```

becomes

```
Fun x -> Fun z -> z
```

同样，我们可以将外部变量重命名为 z，如示例 2.13 所示。请注意，在这种情况下，x 的出现被 *not* 改变了，因为它是由内部变量而不是外部变量所绑定。

Example 2.13.
Fun (x) -> Fun x -> x
变为
Fun (z) -> Fun x -> x

让我们弄清楚在先前的混乱函数中哪些变量出现被绑定到哪个函数，并通过使用 α-转换以更清晰的方式重写该函数。一个可能的结果如下：

```
Let Rec f x =
   If x = 1 Then
      (Fun z -> z (x - 1)) (Fun y -> y)
   Else
      f (x - 1)
In f 100
```

现在更容易理解正在发生的事情。如果将函数 `f` 应用到一个不是 `1` 的整数上，它只是再次将其应用于比它接收到的整数小一的参数。由于我们正在评估 `f 100`，这导致 `f 99`，然后又评估 `f 98`，依此类推。最终，评估 `f 1`。

当 `f 1` 被评估时，我们探索 `If` 表达式的另一分支。我们知道在这个点上 `x` 是 1，因此我们可以看到被评估的表达式是 `(Fun z -> z 0)(Fun y -> y)`。使用替换给出 `(Fun y -> y) 0`，这又反过来给出 `0`。因此，我们可以得出结论，上述表达式将被评估为 `0`。

观察，然而，我们没有正式证明这一点；迄今为止，我们一直以轻率的方式处理替换和其他操作。为了创建一个正式的证明，我们需要一套操作语义，它规定了如何在 **F♭** 中进行评估。第2.3.3节介绍了为 **F♭** 语言创建操作语义的过程，并为我们提供了证明上述结论所需的工具。

2.3.3 Operational Semantics for F♭

我们现在准备开始定义**F♭**的操作语义。由于与我们的布尔语言相同的原因，我们需要一条将值与$\Rightarrow$中的值相关联的规则：

(Value Rule) $\dfrac{}{v \Rightarrow v}$

我们也可以像上面定义布尔语言一样，为 **F♭** 定义布尔运算。请注意，然而，**F♭** 中的并非所有值都是布尔值。幸运的是，我们定义的规则已经为我们解决了这个问题，因为例如，没有 `5` 和 `3` 的逻辑与运算。也就是说，我们知道这些规则仅适用于 **F♭** 布尔值，因为它们使用了仅定义于 **F♭** 布尔值的运算。

(Not Rule) $$\frac{e \Rightarrow v}{\texttt{Not } e \Rightarrow \text{the negation of } v}$$

(And Rule) $$\frac{e_1 \Rightarrow v_1 \; e_2 \Rightarrow v_2 \; e_1 \texttt{ And } e_2 \Rightarrow}{}$$

v_1 和 v_2 的逻辑与 ...

我们也可以以类似的方式定义整数上的运算。为了清晰起见，我们将明确限制这些规则，使它们仅对求值结果为整数的表达式进行操作。

(+ Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ + } e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2}$$

(- Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ - } e_2 \Rightarrow \text{the integer difference of } v_1 \text{ and } v_2}$$

与布尔规则一样，观察这些规则允许递归地应用 $\Rightarrow$ 关系：`5 + (4 - 3)` 可以使用 `+` 规则进行评估，因为 `4 - 3` 可以首先使用 `-` 规则进行评估。

这些规则允许我们编写包含布尔表达式的 **F♭** 程序或包含整数表达式的 **F♭** 程序，但我们目前没有方法将两者结合。我们使用两种机制在程序中将两者混合：条件表达式和比较运算符。**F♭** 中唯一的比较运算符是 `=` 运算符，它比较整数的值。我们定义 `=` 规则为如下。

(= Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ = } e_2 \Rightarrow \texttt{True} \text{ if } v_1 \text{ and } v_2 \text{ are identical, else } \texttt{False}}$$

注意，仅在 v_1 和 v_2 是整数的地方定义了 `=` 规则。由于这个限制，表达式 `True = True` 在 **F♭** 中不可评估。这当然是一个选择问题；作为语言设计者，可以选择移除这个限制并允许直接比较布尔值。然而，为了正式化这一点，需要更改规则。使用上述 `=` 规则忠实实现 **F♭** 的方法是 *required* 拒绝表达式 `True = True`。

一个条件表达式的直观定义是，如果布尔值为真，则评估为第一个表达式的值；如果布尔值为假，则评估为另一个表达式的值。虽然这是真的，但在规则中如何表达这一点是至关重要的。让我们考虑以下对条件表达式规则的错误尝试：

(Flawed If Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad e_3 \Rightarrow v_3}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_2 \text{ if } v_1 \text{ is } \texttt{True}, v_3 \text{ otherwise}}$$

似乎这条规则允许我们评估我们想要评估的许多条件表达式。但让我们考虑这个表达式：

```
If True Then 0 Else (True + True)
```

如果我们尝试将上述规则应用于前面的表达式，我们会发现先决条件 $e_3 \Rightarrow v_3$ 不会成立；没有规则可以将 `True + True` $\Rightarrow v$ 与任何 v 相关联，因为 `+` 规则仅适用于整数。尽管如此，我们希望表达式计算结果为 `0`。那么我们如何实现这个结果呢?

在这种情况下，我们别无选择，只能编写两个具有不同前提条件的不同规则。我们可以在前一条规则中捕获所有关系，同时使用以下两个规则允许像前一条规则那样的表达式进行评估:

(`If True` Rule)
$$\frac{e_1 \Rightarrow \texttt{True}, \quad e_2 \Rightarrow v_2}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_2}$$

(`If False` Rule)
$$\frac{e_1 \Rightarrow \texttt{False}, \quad e_3 \Rightarrow v_3}{\texttt{If } e_1 \texttt{ Then } e_2 \texttt{ Else } e_3 \Rightarrow v_3}$$

再次，这两条规则之间的关键区别在于它们有不同的前提条件集。请注意，`If True` 规则不评估e_3，`If False`规则也不评估e_2。这允许未走过的逻辑路径包含不可评估的表达式，而不会必然阻止包含这些表达式的表达式进行评估。

Application

我们现在准备接近最难的一条规则：应用。我们如何形式化评估像 `(Fun x -> x + 1) (5 + 2)` 这样的表达式？在2.3.2节中，我们看到了我们可以通过变量替换来评估函数应用。由于我们有一个替换操作的数学定义，我们可以围绕它建立我们的函数应用规则。

假设我们希望评估上述表达式。我们可以将其应用分为两部分：被应用的功能和函数的参数。我们想知道表达式评估为何值；因此，我们试图证明对于某些 v，有 $e_1 e_2 \Rightarrow v$。

(Application Rule (Part 1))
$$\frac{?}{e_1\ e_2 \Rightarrow v}$$

在我们的布尔运算中，我们需要在尝试对这些参数进行操作之前评估它们（以便允许递归表达式）。我们的应用规则也是如此；在 `(Fun x -> x + 1) (5 + 2)` 中，我们必须在将其用作参数之前评估 `5 + 2`。[4] 我们必须对我们要应用的功能做同样的事情。

(Application Rule (Part 2))
$$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2 \quad ?}{e_1\ e_2 \Rightarrow v}$$

[4] Actually, some languages would perform substitution before evaluating the expression, but **F♭** and most traditional languages do not. Discussion of this approach is handled in Section 2.3.6.

我们显然还没有完成，因为我们还没有任何先决条件，使我们能够将 v 与某物联系起来。此外，我们知道我们需要使用变量替换，但我们没有在上面的规则中表示 **F♭** 变量的元变量。我们可以通过重新考虑如何评估第一个参数来解决这个问题；我们知道应用规则仅在应用 *functions* 时才有效。在将我们的规则限制为应用函数时，我们可以命名一些元变量来描述函数的内容。

(Application Rule (Part 3)) $$\frac{e_1 \Rightarrow \texttt{Fun}\ x\ \texttt{->}\ e \quad e_2 \Rightarrow v_2 \quad ?}{e_1\ e_2 \Rightarrow v}$$

在上述规则中，x 是代表函数变量的元变量，而 e 代表函数体。现在我们有了这些信息，我们可以用变量替换来定义函数应用。当我们将 `Fun x -> x + 1` 应用到一个值，例如 `7`，我们希望将函数体中所有 `x`（函数的变量）的实例替换为 `7`（提供的参数）。形式上，

(Application Rule) $$\frac{e_1 \Rightarrow \texttt{Fun}\ x\ \texttt{->}\ e \quad e_2 \Rightarrow v_2 \quad e[v_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

F♭ Recursion

我们现在有一套非常完整的**F♭**语言的规则。然而，我们并没有`Let Rec`的规则。正如我们将看到的，`Let Rec`实际上在基本**F♭**中并不是必要的；我们可以从我们已有的规则中构建递归。然而，稍后我们将创建具有类型系统的**F♭**变体，其中那种递归方法将无法工作。出于这个原因以及我们的便利，我们现在将定义`Let Rec`规则。

再次，我们从一个迭代方法开始。我们知道我们想要能够评估 `Let Rec` 表达式，因此我们提供元变量来表示表达式的组成部分。

(Recursive Application Rule (Part 1))
$$\frac{?}{\texttt{Let Rec}\ f\ x\ \texttt{=}\ e_1\ \texttt{In}\ e_2 \Rightarrow v}$$

让我们考虑我们想要完成的事情。暂时考虑一种递归方法来计算1到5之间数字的和：

```
Let Rec f x =
   If x = 1 Then
      1
   Else
      f (x - 1) + x
In f 5
```

如果我们关注最后一行（`In f 5`），我们可以看到我们希望递归函数的主体应用于值 `5`。我们可以相应地编写我们的规则，通过将 `f` 替换为函数的主体。我们必须确保使用相同的元变量来表示函数的参数，以便允许新的函数主体中的变量被捕获。我们得到以下规则。

(Recursive Application Rule (Part 2))

$$\frac{e_2[\texttt{(Fun } x \texttt{ -> } e_1\texttt{)}/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

我们可以通过将其应用于上述递归求和来测试我们的新规则。

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{Fun x ->}\cdots \Rightarrow \texttt{Fun x ->}\cdots} \quad \dfrac{}{\texttt{5} \Rightarrow \texttt{5}} \quad \dfrac{\dfrac{}{\texttt{5 = 1} \Rightarrow \texttt{False}} \quad \dfrac{\dfrac{\overset{???}{\overline{\texttt{f (5-1)} \Rightarrow v'}} \quad \dfrac{}{\texttt{5} \Rightarrow \texttt{5}}}{\texttt{f (5-1) + 5} \Rightarrow v}}{\texttt{If 5 = 1 Then 1 Else f (5-1) + 5} \Rightarrow v}}{\texttt{(Fun x -> If x = 1 Then 1 Else f (x-1) + x) 5} \Rightarrow v}}{\texttt{Let Rec f x = If x = 1 Then 1 Else f (x-1) + x In f 5} \Rightarrow v}$$

如上标签所示，我们的递归规则尚未完成。当我们到达对 `f (5-1)` 的评估时，我们陷入了困境；`f` 未绑定。没有对 `f` 的绑定，我们无法重复递归。

除了将 f 的调用替换为 e_2 中的函数应用之外，我们还需要确保函数体本身中出现的任何 f 都被绑定。我们将它们绑定到什么？我们可以尝试将它们也替换为函数应用，但这会让我们陷入一个兔子洞；每个函数应用都需要另一个替换。然而，我们可以通过重新引入 `Let Rec` 语法，将 f 在 e_1 中的应用替换为 f 的 *recursive* 应用。这导致以下应用规则：

(Recursive Application Rule (Part 3))

$$\frac{e_2[\texttt{Fun } x \texttt{ -> } e_1[(\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } f)/f]/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

虽然这在某种程度上是有道理的，但并不完全正确。在第2.3.2节中，我们看到了替换必须用一个 *value* 来替换一个变量，而上面的 `Let Rec` 项是一个表达式。幸运的是，我们有函数作为值；因此，我们可以将表达式放入一个函数中，并确保我们用适当的参数调用它。

(Recursive Application Rule)

$$\frac{e_2[\texttt{Fun } x \texttt{ -> } e_1[(\texttt{Fun } x \texttt{ -> Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } f\ x)/f]/f] \Rightarrow v}{\texttt{Let Rec } f\ x \texttt{ = } e_1 \texttt{ In } e_2 \Rightarrow v}$$

现在，当我们评估求和示例时，我们遇到的不是 `f (5-1)`，而是 `Let Rec f x = If x = 1 Then 1 Else f (x-1) + x In f (5-1)`。这使得我们可以递归地回到 `Let Rec` 规则。最终，我们可能到达一个分支，

不评估条件表达式的 Else 边，在这种情况下，该 Let Rec 不会被展开（允许我们终止）。规则的每次应用实际上“展开”一层递归。

总结来说，我们可以将 **F♭** 的操作语义定义为如下：

Definition 2.11 (**F♭** 操作语义). *The **F♭** operational semantics relation $e \Rightarrow v$ holds if e is closed **F♭** expression and a proof exists using the rules of Figure 2.1.*

让我们考虑一些使用 **F♭** 操作语义的证明树示例。

Example 2.14.

Expression:

```
If 3 = 4 Then 5 Else 4 + 2
```

Proof:

$$\dfrac{\dfrac{\overline{3 \Rightarrow 3} \quad \overline{4 \Rightarrow 4}}{\texttt{3 = 4} \Rightarrow \texttt{False}} \quad \dfrac{\overline{4 \Rightarrow 4} \quad \overline{2 \Rightarrow 2}}{\texttt{4 + 2} \Rightarrow 6}}{\texttt{If 3 = 4 Then 5 Else 4 + 2} \Rightarrow 6}$$

Example 2.15.

Expression:

```
(Fun x -> If 3 = x Then 5 Else x + 2) 4
```

Proof:

$$\dfrac{\overline{\texttt{Fun x -> If 3 = x Then 5 Else x + 2} \Rightarrow \texttt{Fun x -> If 3 = x Then 5 Else x + 2}} \quad \overline{4 \Rightarrow 4} \quad \overbrace{\texttt{If 3 = 4 Then 5 Else 4 + 2} \Rightarrow 6}^{\text{通过示例2.14}}}{\texttt{(Fun x -> If 3 = x Then 5 Else x + 2) 4} \Rightarrow 6}$$

Example 2.16.

Expression:

```
(Fun f -> Fun x -> f(f x))(Fun y -> y - 1) 4
```

Proof:

Due to the size of the proof, it is broken into multiple parts. We use $v \Rightarrow \bigstar$ as an abbreviation for $v \Rightarrow v$ (when v is lengthy) for brevity.

Part 1:

$$\dfrac{\overline{\texttt{Fun f -> Fun x -> f(f x)} \Rightarrow \bigstar} \quad \overline{\texttt{Fun y -> y - 1} \Rightarrow \bigstar} \quad \overline{\texttt{(Fun x -> (Fun y -> y - 1) ((Fun y -> y - 1) x))} \Rightarrow \bigstar}}{\texttt{(Fun f -> Fun x -> f(f x))(Fun y -> y - 1)} \Rightarrow \texttt{(Fun x -> (Fun y -> y - 1) ((Fun y -> y - 1) x))}}$$

Part 2:

$$\dfrac{\overline{\textit{(by part 1)}} \quad \overline{4 \Rightarrow 4} \quad \dfrac{\overline{\texttt{Fun y -> y - 1} \Rightarrow \bigstar} \quad \dfrac{\overline{\texttt{Fun y -> y - 1} \Rightarrow \bigstar} \quad \overline{4 \Rightarrow 4} \quad \dfrac{\overline{4 \Rightarrow 4} \quad \overline{1 \Rightarrow 1}}{\texttt{4 - 1} \Rightarrow 3}}{\texttt{((Fun y -> y - 1) 4)} \Rightarrow 3} \quad \dfrac{\overline{3 \Rightarrow 3} \quad \overline{1 \Rightarrow 1}}{\texttt{3 - 1} \Rightarrow 2}}{\texttt{(Fun y -> y - 1) ((Fun y -> y - 1) 4)} \Rightarrow 2}}{\texttt{(Fun f -> Fun x -> f(f(x)))(Fun y -> y - 1) 4} \Rightarrow 2}$$

Interact with F♭. 递归评估的追踪很困难，因此读者应该花些时间探索 Let Rec 的语义。

一个好的方法是使用 **F♭** 解释器。尝试评估我们上面看过的表达式：

(Value Rule) $$\frac{}{v \Rightarrow v}$$

(Not Rule) $$\frac{e \Rightarrow v}{\texttt{Not}\ e \Rightarrow \text{the negation of } v}$$

(And Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1\ \texttt{And}\ e_2 \Rightarrow \text{the logical and of } v_1 \text{ and } v_2}$$

(Or Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2 \Rightarrow v_2}{e_1\ \texttt{Or}\ e_2 \Rightarrow \text{the logical or of } v_1 \text{ and } v_2}$$

(+ Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1\ \texttt{+}\ e_2 \Rightarrow \text{the integer sum of } v_1 \text{ and } v_2}$$

(- Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1\ \texttt{-}\ e_2 \Rightarrow \text{the integer difference of } v_1 \text{ and } v_2}$$

(= Rule) $$\frac{e_1 \Rightarrow v_1, \quad e_2 \Rightarrow v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1\ \texttt{=}\ e_2 \Rightarrow \texttt{True} \text{ if } v_1 \text{ and } v_2 \text{ are identical, else } \texttt{False}}$$

(`If True` Rule) $$\frac{e_1 \Rightarrow \texttt{True}, \quad e_2 \Rightarrow v_2}{\texttt{If}\ e_1\ \texttt{Then}\ e_2\ \texttt{Else}\ e_3 \Rightarrow v_2}$$

(`If False` Rule) $$\frac{e_1 \Rightarrow \texttt{False}, \quad e_3 \Rightarrow v_3}{\texttt{If}\ e_1\ \texttt{Then}\ e_2\ \texttt{Else}\ e_3 \Rightarrow v_3}$$

(Application Rule) $$\frac{e_1 \Rightarrow \texttt{Fun}\ x\ \texttt{->}\ e, \quad e_2 \Rightarrow v_2, \quad e[v_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

(Let Rule) $$\frac{e_1 \Rightarrow v_1 \quad e_2[v_1/x] \Rightarrow v_2}{\texttt{Let}\ x\ \texttt{=}\ e_1\ \texttt{In}\ e_2 \Rightarrow v_2}$$

(Let Rec)

$$\frac{e_2[\texttt{Fun}\ x\ \texttt{->}\ e_1[(\texttt{Fun}\ x\ \texttt{->}\ \texttt{Let Rec}\ f\ x\ \texttt{=}\ e_1\ \texttt{In}\ f\ \ x)/f\,]/f\,] \Rightarrow v \quad \texttt{Let Rec}\ f\ x\ \texttt{=}\ e_1\ \texttt{In}\ e_2 \Rightarrow v}{}$$

图2.1: **F♭**操作语义

```
# Let Rec f x =
    If x = 1 Then 1 Else x + f (x - 1)
  In f 3;;
==> 6
```

另一个有趣的实验是评估一个递归函数而不应用它。注意，结果是等价于应用 `Let Rec` 规则的一次。这是了解“展开”实际上是如何发生的一个好方法：

```
# Let Rec f x =
    If x = 1 Then 1 Else x + f (x - 1)
  In f;;
==> Fun x ->
      If x = 1 Then
        1
      Else
        x + (Let Rec f x =
          If x = 1 Then
            1
          Else
            x + (f) (x - 1)
        In
          f) (x - 1)
```

如我们之前提到的，定义一种语言的操作语义的主要好处之一是我们可以严格验证关于该语言的断言。现在我们已经定义了**F♭**的操作语义，我们可以证明一些关于它的事情。

Lemma 2.3. ***F♭*** *is deterministic.*

Proof. 通过检查规则，任何时刻最多只能应用一条规则。 □

Lemma 2.4. ***F♭*** *is* 未 *normalizing.*

Proof. 要证明一种语言不是归一化的，我们只需证明存在某个 e，使得不存在 v 满足 $e \Rightarrow v$。令 e 为 `(Fun x -> x x)(Fun x -> x x)`。对于任何 v，有 $e \not\Rightarrow v$。因此，**F♭** 不是归一化的。 □

这个证明中的表达式非常有趣，我们将在第2.3.5节中更详细地研究它。它不等于一个值，因为其评估的每一步都产生自身作为先决条件。这大致相当于试图通过使用A作为已知条件来证明命题A。

在这种情况下，表达式无法评估，因为它永远不会耗尽要完成的工作。这并非**F♭**中出现的唯一非归一化表达式；另一种类型包括那些没有适用评估规则的表达式。例如，`(4 3)`是一个更简单的非归一化表达式。当e_1不是一个函数表达式时，不存在评估$(e_1\ e_2)$的规则。

这两个情况在操作语义上看起来像发散。然而，在解释器的情况下，这两种表达式表现不同。通常，永远做不完工作的表达式会导致解释器无限循环（因为大多数解释器不够聪明，无法意识到它们永远不会完成）。另一方面，尝试将表达式应用于非函数的表达式会导致解释器以异常的形式提供清晰的错误。这是因为这里的错误很容易检测到；在尝试评估这些表达式时，解释器可以迅速发现其可用的规则中没有适用于该表达式的，并通过引发异常来响应。尽管如此，它们在理论上都是等价的。

2.3.4 The Expressiveness of F♭

F♭ 没有很多功能，但我们可以用它做更多的事情，比看起来要多。**F♭**是图灵完备的，这意味着（在其他事情中）任何在OCaml中可能进行的计算都在**F♭**中可能。我们使用*encodings*这样做。编码是一种在系统不支持直接表示行为的方式中表示行为的方法。

我们定义这些编码为源代码上的转换：在代码被评估之前，我们将一个表达式的所有实例转换为另一个。这种转换不是评估：它是对AST作为数据结构的修改。因此，我们必须像在定义变量替换时一样小心地保留变量绑定。

在继续到 **F♭** 中编码的示例之前，重要的是要注意编码允许一种语言中的特性被*expressed*；这并不能保证这将是 *efficient*。**F♭** 没有乘法，但我们可以通过重复加法来编码乘法。这样做将使我们能够表达乘法，但它的运行速度比我们的语言直接支持乘法并且我们的解释器可以使用我们的硬件中的乘法电路来高效计算要慢得多。

Multi-Parameter Functions 对于我们的第一个编码示例，我们将考虑多参数函数。在OCaml中，我们可以写`fun x y -> x + y`来表示一个有两个参数的函数：`x`和`y`。在**F♭**中，写`Fun x y -> x + y`是一个语法错误：2.3.1节中**F♭**的语法只允许在`Fun`关键字和`->`符号之间有一个变量。但我们可以使用以下方式通过柯里化来编码多参数函数：

$$\texttt{Fun } x_1\, x_2 \ldots x_n \texttt{ -> } e \overset{\text{定义}}{=} \texttt{Fun } x_1 \texttt{ -> } \ldots \texttt{ -> Fun } x_n \texttt{ -> } e$$

此编码表示，类似于 `Fun x y -> x + y` 的表达式可以先解码为 `Fun x -> Fun y -> x + y`，然后再进行评估。在 **F♭** 解释器中这不会起作用，因为该解释器没有支持此语法的解码器。但我们可以想象一个工具，它可以解析这种扩展语法并将其解码为适当的 **F♭**；然后这样的工具可以使用未经修改的 **F♭** 解释器来运行程序。

Let Expressions 尽管 **F♭** 引入了 `Let` 的语法，但编码起来相当简单：

$$\texttt{Let } x \texttt{ = } e_1 \texttt{ In } e_2 \overset{\text{def}}{=} \texttt{(Fun } x \texttt{ -> } e_2\texttt{) } e_1$$

我们不需要这样做，但观察到一个 **F♭** 的存在并不会使其表达更丰富；这仅仅是我们的一种便利。一个类似 **F♭** 的语言如果没有 `Let`，其表达也会同样丰富。

例如，该程序

```
Let x = 3 + 2 In x + x
```

可以改写为

```
(Fun x -> x + x) (3 + 2)
```

并且这两个程序都会评估为 `10`。

Pairs 存在许多可能的对编码。我们在这里选择的一种编码利用了函数和布尔值。考虑在对上存在三个重要操作：创建对、检索其第一个元素和检索其第二个元素。

让我们首先考虑构建对。我们将使用语法 `pr` e_1 e_2 来构建一对；例如，`pr 3 5` 表示在 `3` 和 `5` 之间创建一对。由于我们在 **F♭** 中没有对，我们将使用等待描述要检索哪个元素的参数的函数来表示它们。

$$\texttt{pr} \stackrel{\text{def}}{=} \texttt{Fun x -> Fun y -> Fun b -> If b Then x Else y}$$

这样，我们的 `pr 3 5` 示例变为

```
(Fun x -> Fun y -> Fun b -> If b Then x Else y) 3 5
```

which evaluates towhich 评估为

```
(Fun b -> If b Then 3 Else 5)
```

基于这个 `pr` 的定义，我们可以定义投影这对的左右元素的编码：

$$\texttt{left} \stackrel{\text{def}}{=} \texttt{Fun p -> p True}$$
$$\texttt{right} \stackrel{\text{def}}{=} \texttt{Fun p -> p False}$$

第一个函数接受一对（我们期望它是由 `pr` 构造的），并将值 `True` 传递给它。当评估时，当函数被调用时，将用 `True` 替换 `b`，执行编码对函数体的 `Then` 分支，从而返回对的左元素。例如，我们可以用 `left (pr 3 5)` 来表示

```
(Fun p -> p True)
   ((Fun x -> Fun y -> Fun b -> If b Then x Else y) 3 5)
```

评估为 `3`。

元组——即三元组、四元组等——可以以类似于对的形式进行编码。例如，我们可以使用一个整数而不是布尔值来选择要检索的元素，并嵌套 `If` 表达式以返回正确的值。

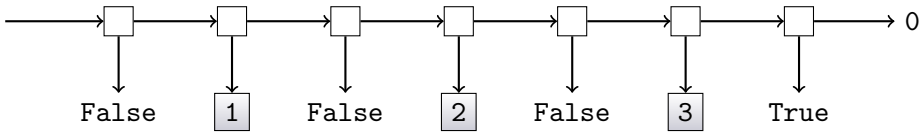


图2.2：使用成对实现的列表

Lists 与对一样，列表也可以以多种方式编码。一种方式是依赖于我们上面对对的编码，并创建 *nested* 对来表示列表的不同情况。与对一样，也有一些有趣的定义：

- `empty`, 表示空列表
- `cons`, (OCaml `::`)
- `isEmpty`, 判断列表是否为空
- `head`,
- `tail`,

为了支持这些操作，我们想象每个列表都是一个表示是否为空的布尔值和描述列表内容的值的配对。对于空列表，这个“内容”值没有意义；我们只是有它，以便空列表和非空列表都可以被当作配对处理。对于非空列表，“内容”值本身是一个包含头元素和列表其余部分的配对。

我们可以将上述策略编码如下：

```
empty  def=  pr True 0 def
       = Fun h -> Fun t -> pr False (pr h t)
 cons  def=  Fun lst -> left lst                  def
isEmpty = Fun lst -> left (right lst)      head
       = Fun lst -> right (right lst)      tail
```

例如，我们可能将三个元素的列表写成 `cons 1 (cons 2 (cons 3 empty))` . 这创建了一个对，其中左元素是 `True`（以表示非空列表），而右元素本身也是一个对。在这个内部对中，左元素是 `1`；右元素是包含下一个两个元素的编码列表。图2.2展示了这个列表。在该图中，阴影矩形表示列表中的数据；图中的其他部分都是编码。每个空方框是一个对；其他所有数据要么是标签（指示列表是否为空），要么是占位符值。

Sequencing 在OCaml中，表达式可以通过分号运算符分隔来*sequenced*；这允许我们从第一个表达式运行副作用，例如打印

在评估第二个表达式之前表达。因为 **F♭** 是一种纯函数式语言，所以对表达式进行排序几乎没有意义；编写 $e_1;e_2$ 等同于只编写 e_2，因为 e_1 的结果从未被使用，且 e_1 不能有任何副作用。尽管如此，我们可以在 **F♭** 中以下方式定义排序：

e_1 ; e_2 定义= (Fun x -> e_2) e_1

在 e_2 中，x 必须是一个不在 e_2 中自由出现的变量。评估此表达式时，首先评估 e_1，然后丢弃结果（因为我们用 x 替换它，但在 x 中 e_2 并不是自由出现的）然后评估 e_2。我们将在第 4 章讨论在何种情况下顺序变得重要。

Freezing and Thawing 我们可以随意通过冻结和解冻来停止和重新启动计算。

Freeze e def= Fun x -> e def= e 0
Thaw e

在 e 中，x 是一个非自由变量。Freeze e 冻结了 e，使其无法被计算。Thaw e 启动了一个冻结的计算。例如，考虑：

```
Let x = Freeze (2 + 3) In (Thaw x) + (Thaw x)
```

这个表达式的值与没有冻结和解冻的等效表达式相同，但 `2 + 3` 被评估了两次。再次强调，在纯函数式语言中，唯一的区别是冻结和解冻效率较低。在具有副作用的语言中，如果冻结的表达式引起副作用，那么函数的冻结/解冻版本可能产生与原始函数不同的结果，因为冻结的副作用将根据解冻次数应用多次。

Logical Combinators 经典的 **logical combinators**，用于重新组合数据的简单函数，可以在 **F♭** 中定义，就像一个函数集合一样。

```
I def= Fun x -> x def= Fun x -> Fun y -> x def= K
def= Fun x -> Fun y -> Fun z -> (x z) (y z) def= S
def= Fun x -> x x                                 D
```

2.3.5 Russell's Paradox and Encoding Recursion

尽管 **F♭** 内置了 `Let Rec` 操作以帮助编写递归函数，但我们甚至没有这个规则也可以编写递归函数！这是一个基本且不明显的编码，因此在我们探索它之前，我们必须介绍一些背景信息。正如我们将看到的，递归的编码与由伯特兰·罗素引起的著名集合论悖论密切相关。

让我们从提出以下问题开始。*How can programs compute forever in* ***F♭*** *without recursion?* 这个问题的答案以一个看似简单的表达式形式出现：

```
(Fun x -> x x)(Fun x -> x x)
```

从引理2.2中回忆起，这个表达式的存在的一个推论是 **F**♭ 不是归一化的。这种计算在某种意义上是奇怪的。`(x x)` 是一个应用于自身的函数。这种非归一化计算的核心存在一个逻辑悖论，即罗素悖论。

Russell's Paradox

在弗雷格的集合论（约1900年），集合被写成谓词 $P(x)$。我们可以将谓词视为单参数函数，它返回一个布尔值：如果参数在谓词表示的集合中，则为真；如果不在此集合中，则为假。测试集合成员资格是通过应用谓词来完成的。例如，考虑以下谓词

```
Fun x -> (x = 2 Or x = 3 Or x = 5)
```

这个谓词表示整数集 $\{2, 3, 5\}$，因为它将为该集中的任何元素返回 `True`，并为所有其他参数返回 `False`。如果我们扩展 **F**♭ 以包括一个本地整数小于运算符，则谓词

```
Fun x -> x < 2
```

将表示一个包含所有小于 2 (的整数值的无限大集合 **F**♭，因为) 仍然没有实数的概念。一般来说，给定一个表示集合 S 的谓词 P,

$e \in S$ 当且仅当 $P\ e \Rightarrow$ `True`

罗素在弗雷格的集合理论中发现了悖论，它可以以下述方式表达。

Definition 2.12 (罗素悖论)**.** *Let P be the set of all sets that do not contain themselves as members. Is P a member of P?*

询问一个集合是否是其自身的成员似乎是一个奇怪的问题，但实际上有许多集合是其自身的成员。无限退化的集合 $\{\{\{\{\ldots\}\}\}\}$ 将其自身作为成员。不是苹果的东西的集合也是其自身的成员（显然，非苹果的集合不是苹果）。这类集合仅在“非基础”集合理论中产生。

为了探索罗素悖论的实质，让我们尝试回答它提出的问题：P 是否包含自身作为一个成员？假设答案是肯定的，并且 P 包含自身作为一个成员。如果是这样的话，那么 P 就不应该在 P 中，P 是包含所有作为自身成员的集合的集合。然后，假设答案是否定的，并且 P 不包含自身作为一个成员。那么 P 应该已经被包括在

P, 因为其不包含自身。换句话说，P 是 P 的成员当且仅当它不是。因此，罗素悖论确实是一个悖论。

这也可以通过使用 **F♭** 函数作为谓词来展示。具体来说，我们将为 P 上面的谓词编写一个。我们必须定义 P 以接受一个参数（我们知道它是一个集合 - 我们模型中的一个谓词）并确定它是否包含自身（将其作为参数传递给自己）。因此，我们表示 P 的方式是 `Fun x -> Not(x x)`。我们只需将 P 应用到自身以获得答案。

Definition 2.13 (计算罗素悖论)**.** *Let*

$P \stackrel{def}{=}$ *`Fun x -> Not(x x)`.*

What is the result of P P? Namely, what is

`(Fun x -> Not(x x)) (Fun x -> Not(x x))`?

如果这个 **F♭** 程序被评估，它将永远运行。我们可以通过查看评估证明的几个步骤来非正式地检测这个模式：

$$\dfrac{\dfrac{\dfrac{\vdots}{\texttt{Not (Not ((Fun x -> Not (x x))(Fun x -> Not (x x))))}}}{\texttt{Not ((Fun x -> Not (x x)) (Fun x -> Not (x x)))}}}{\texttt{(Fun x -> Not (x x)) (Fun x -> Not (x x))}}$$

我们知道 `Not (Not (`e`))` 的值与 e 相同。[5] 我们可以看到我们在兜圈子。再次，这个陈述告诉我们 $P\ P \Rightarrow$ `True` 当且仅当 $P\ P \Rightarrow$ `False`。

这不是罗素看待他的悖论的方式，但它具有相同的核心结构；它只是用计算术语重新表述，而不是集合论。悖论的计算机实现是谓词无法计算为真或假，因此它不是一个合理的逻辑陈述。罗素在弗雷格的集合论中发现这个悖论震撼了数学的基础。为了解决这个问题，罗素发展了他的分叉类型理论，这是编程语言中类型的祖先。程序

```
(function x -> not(x x)) (function x -> not(x x))
```

在OCaml中无法输入，原因与相应的谓词在Russell的类型分支理论中不可输入的原因相同。尝试将上述代码输入到OCaml顶层，看看会发生什么。

更多关于罗素悖论的信息可以在[14]中找到。

[5] In this case, anyway, but not in general. General assertions about the equivalence of expressions are hard to prove. In Section 2.4, we will explore a formal means of determining if two expressions are equivalent.

Constructing the Y-combinator

在逻辑视图中，将一个函数作为参数传递给自己是一个不好的做法。然而，从编程的角度来看，这可以是一个非常强大的工具：将函数传递给自己允许递归函数的定义而不使用 `Let Rec`。我们现在展示如何修改悖论组合器以完成有用的工作。特别是，我们展示了如何构建所谓的 *fixed point combinator*、`Y`，这允许轻松编写递归函数。

我们根据Python和C++等面向对象语言如何实现消息传递到自身来构建`Y`：每次调用对象方法时，对象本身都会作为额外的参数传递。

我们通过再次编写一个 *summate* 函数来展示这种方法。这次我们遵循 C++ 的想法，编写一个接受 *two* 参数的函数：`arg`，这是我们想要求和的数字，以及 `this`，这是我们要求传入的函数的副本，以便发生递归。我们将我们的函数命名为 `summate0` 以反映它并不是我们想要的 summate 的事实。它可以定义为

```
summate0 def= Fun this -> Fun arg ->
   If arg = 0 Then 0 Else arg + this this (arg - 1)
```

注意使用 `this this (arg - 1)`。`this` 的第一次使用命名了要应用的功能；`this` 的第二次使用是该函数的一个参数。参数 `this` 允许递归调用再次调用该函数，从而允许我们递归到所需的程度。

我们现在求和整数 $\{0, 1, \ldots, 7\}$ 与表达式 会话

```
summate0 summate0 7
```

`summate0` 始终期望其第一个参数this是其自身。然后它可以使用一个副本进行递归调用（第一个`this`）并将另一个副本传递给未来的复制。因此，`summate0 summate0`“启动泵”，换句话说，通过给进程一个额外的初始副本。

更好的是，回想一下柯里化允许我们在不应用它的情况下获得内部函数。本质上，一个具有多个参数的函数可以被部分评估，一些参数被固定，而其他参数等待输入。我们可以利用这一点来定义我们想要的求和函数：

```
summate def= summate0 summate0
```

这使我们能够通过我们的 `summate` 函数隐藏个体自传递，从而在很大程度上清理了事情。我们可以将整个过程总结如下，记住即使是 `Let` 本身也可能是一个函数调用的宏：

```
summate def= Let summ = Fun this -> Fun arg ->
   If arg = 0 Then 0 Else arg + this this (arg - 1)

In summ summ
```

我们现在有一个定义递归函数的模型，而不使用Let Rec运算符。这意味着没有内置递归的未类型化语言仍然可以是图灵完备的。虽然这是一个成就，我们还能做得更好；我们可以几乎完全将自我传递的想法从函数体本身抽象出来。

The Y-Combinator Y-组合子是自传递的进一步抽象。想法是 Y-组合子使用作为参数传递的抽象代码体进行自我应用。我们首先定义一个名为 almostY 的函数，然后修改该定义以得到真正的 Y-组合子。

```
almostY def= Fun body -> body body
```

使用 almostY，我们可以如下定义 summate。

```
summate def= almostY (Fun this -> Fun arg ->
   If arg = 0 Then 0 Else arg + this this (arg - 1))
```

真实 Y-组合器实际上更进一步，允许我们以更自然的风格仅使用“this (arg - 1)”来编写递归调用，避免了额外的 this 参数。为此，我们假设 body 参数已经处于这种简单形式。然后我们定义一个新的形式，wrapper，在 body 中用 (this this) 替换 this：

Definition 2.14 (Y组合)**.**

```
combY def= Fun body ->
   Let wrapper = Fun this -> Fun arg -> body (this this) arg
   In wrapper wrapper
```

这是与上一小节中定义的相同的 Y。

这些示例中执行的转换步骤也是高阶函数编程力量的好例子：“代码手术”通过对 body 进行函数抽象和应用来产生 wrapper。然后可以使用 Y-组合子来定义 summate 为

```
summate def= combY (Fun this -> Fun arg ->
   If arg = 0 Then 0 Else arg + this (arg - 1))
```

Another route to Y: Encoding Recursion by Passing Self

这里是基于上述逻辑padarox构造的另一种构建Y的方法。

首先，让我们从原始的Bad先生开始：

```
(Fun x -> Not (x x)) (Fun x -> Not (x x))
```

第一步让Bad先生做一些好事，而不是制作无限数量的Not、
Not (Not (Not(...))), 让我们制作无限数量的某个预定义函数F:

```
(Fun x -> F (x x)) (Fun x -> F (x x))
```

嗯，这会导致无界 F (F (F(...))), 但这个序列永远不会终止，所以它对我们没有任何好处。下一步是在某些点 *freeze* 计算以直接停止其继续；回想一下我们上面的冻结宏，我们只需要将某些表达式包裹在一个 Fun _ -> ... [6] 中来冻结它:

```
makeFroFs def=
(Fun x -> F (Fun _ -> x x)) (Fun x -> F (Fun _ -> x x))
```

现在，我们已推迟无限执行；假设 F 是说

```
Fun froF -> True
```

以上将计算为

```
F (Fun _ -> makeFroFs)
```

并且，由于我们定义的 F 抛弃了其参数，这个计算将反过来以值 True 终止。

这个特定示例很好地终止了 *too*，它丢弃了我们辛苦制作的 F 的所有副本。我们可以通过创建一个 F 来获得递归，它有时会使用其参数 Fun _ -> makeFroFs 通过 *thawing* 它来进行递归调用。考虑以下修订的 F，目的是最终成为一个对参数 n 的整数 $\{0, 1, \ldots, n\}$ 求和的函数:

```
F def= Fun froFs -> Fun n ->
    If n = 0 Then 0 Else n + froFs 0 (n - 1)
```

如果 makeFroFs 使用这个 F，它将再次计算为 F (Fun _ -> makeFroFs)，但这个新的 F 并没有将参数 Fun _ -> makeFroFs 丢弃。考虑 makeFroFs 5 的计算，根据上述内容，它等同于计算

```
(F (Fun _ -> makeFroFs)) 5
```

因此，对于上述 F 的定义，我们看到参数 froFs 将被实例化为 Fun -> makeFroFs，参数 n 则为 5。因为 5 = 0 是 False，所以我们计算 else 子句，该子句在上述实例化之后是

```
5 + (Fun _ -> makeFroFs) 0 (5-1)
```

[6] We use "_" to represent a wildcard pattern, just like in OCaml.

并且，为了计算加法右侧，首先将虚拟参数 0 应用到解冻 makeFroFs，因此我们现在正在设置计算 makeFroFs (5-1)。向上看——这几乎是我们开始的地方，除了参数小了一个！所以，makeFroFs 现在是一个递归求和函数，这个例子最终将计算到 15。我们已经成功地将悖论组合器重新用于编写递归函数。

有一些小的清理步骤我们可以对上面的内容进行。首先，因为我们关心的 F 是两个参数的柯里化函数（例如，我们需要额外的参数 n 来进行不同值的递归调用），我们可以创建一个修订版的冻结器 Fun n -> F n，它不需要使用 0 作为解冻器，而是可以“惩罚”并使用参数本身来进行解冻。因此，我们可以重新定义上面的内容如下

```
makeFs ≝ (Fun x -> F' (Fun n -> (x x) n))
(Fun x -> F' (Fun n -> (x x) n))
```

```
F' ≝ Fun fs -> Fun n ->
     If n = 0 Then 0 Else n + fs (n - 1)
```

– 我们可以在这里从递归调用中移除 0 参数，因为 n-1 参数通过我们的 pun 进行解冻工作。

最后我们可以进行的一次重构，以清理这个代码，是创建一个通用的递归函数生成器，通过将上面的 makeFs 中的 F' 提取为一个显式参数 f。这给我们

```
Y ≝ Fun f ->
(Fun x -> f (Fun n -> (x x) n))
(Fun x -> f (Fun n -> (x x) n))
```

我们可以应用到一些具体的 F，例如 Y F'，来创建我们的递归求和函数。我们称上述表达式为 Y，因为这种递归函数创建者多年前被逻辑学家发现并赋予了这个名字。

2.3.6 Call-By-Name Parameter Passing

在 **call-by-name** 参数传递中，函数调用时不会评估函数的参数，而是仅在需要时才进行评估。这种参数传递方式现在主要具有历史意义；Algol 使用它，但现代语言默认不使用按名调用（Digital Mars D 语言允许通过使用 lazy 修饰符将参数视为按名调用）。原因是如果使用按名调用参数传递，编写高效的编译器会更加困难。尽管如此，简要了解按名调用参数传递还是有价值的。

让我们定义按名调用的操作语义。

(Call-By-Name Application) $$\frac{e_1 \Rightarrow \texttt{Fun}\ x\ \texttt{->}\ e, \quad e[e_2/x] \Rightarrow v}{e_1\ e_2 \Rightarrow v}$$

冻结和解冻，在2.3.4节中定义，是一种在按值调用语言中获得按名调用行为的方法。因此，考虑以下计算：

```
(Fun x -> Thaw x + Thaw x) (Freeze (3 - 2))
```

`(3 - 2)` 直到我们进入函数体内部并解冻它时才进行评估，然后它被评估两次。这正是按名调用参数传递的行为，因此 `Freeze` 和 `Thaw` 可以通过这种方式进行编码。`(3 - 2)` 被执行两次表明按名调用的主要弱点，即函数参数的重复评估。

懒或 **call-by-need** 评估是按名调用的一种版本，它将第一次评估的函数参数缓存起来，以便在后续使用中不必重新评估。Haskell [13, 7] 是一种具有懒计算的纯函数式语言。

2.3.7 F♭ Abstract Syntax

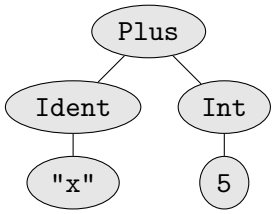
前几节详细描述了**F♭**的操作语义，从其具体语法角度出发。然而，回顾我们的布尔语言，解释器是在一个程序表示上操作的，这种表示更有利于程序性操作；这种表示被称为其**abstract syntax**。为了定义OCaml解释器的**F♭**的抽象语法，我们需要定义一个变体类型，以捕捉**F♭**的表达能力。我们将使用的变体类型如下。

```
type ident = Ident of string

type expr =
 Var of ident | Function of ident * expr | Appl of expr * expr |
 Let of ident * expr * expr | LetRec of ident * ident * expr * expr |
 Plus of expr * expr | Minus of expr * expr | Equal of expr * expr |
 And of expr * expr| Or of expr * expr | Not of expr |
 If of expr * expr * expr | Int of int | Bool of bool

type fbtype = TInt | TBool | TArrow of fbtype * fbtype | TVar of string;;
```

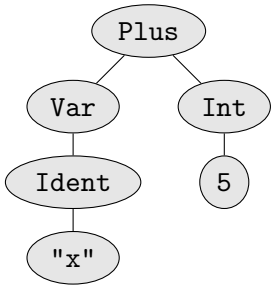
一个重要的点是存在 `ident` 类型。注意 `ident` 在 `expr` 类型中的使用：作为变量标识符，以及作为 `Fun` 和 `Let Rec` 的函数参数。`ident` 类型将额外的语义信息附加到字符串上，表示该字符串专门表示一个标识符。

注意，尽管`ident`以两种不同的方式使用：表示变量的*declaration*（例如在`Function (Ident "x",...)`中）和表示变量的*use*（例如在`Var(Ident "x")`中）。观察到一个变量的使用是一个表达式，因此可以出现在AST中任何表达式可以出现的地方。另一方面，变量的声明不是一个表达式；变量仅在函数中声明。这样，我们能够使用OCaml类型系统来帮助我们保持AST节点的属性清晰。

例如，考虑以下AST：



首先看，这似乎表示了 **F♭** 表达式 `x + 5`。然而，上述 AST 使用我们定义的变体实际上无法存在。`Plus` 变体在构造时接受两个表达式，而 `Ident` 不是一个变体；因此，等效的 OCaml 代码 `Plus(Ident "x",Int 5)` 甚至无法通过类型检查。相反，AST 表示的是 **F♭** 表达式 `x + 5`。



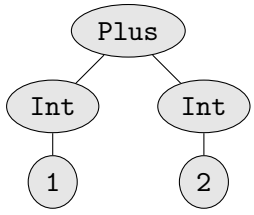
这由OCaml代码 `Plus(Var(Ident "x"),Int 5)` 表示。

能够从抽象语法转换为具体语法，反之亦然，这是一项重要的技能，但要熟练掌握这种转换需要一些时间。让我们看看一些例子 **F♭**，以追求提高这项技能。

Example 2.17.

Concrete:
`1 + 2`

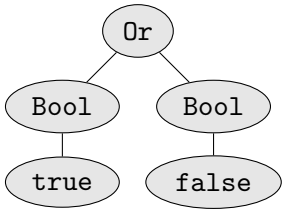
Abstract:
`Plus(Int 1,Int 2)`



Example 2.18.

Concrete:
`True Or False`

Abstract:
`Or(Bool true,Bool false)`



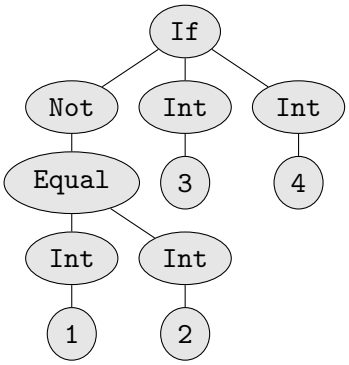
Example 2.19.

Concrete:

```
If Not(1 = 2) Then 3 Else 4
```

Abstract:

```
If(Not(Equal(Int 1, Int 2)),
   Int 3, Int 4)
```



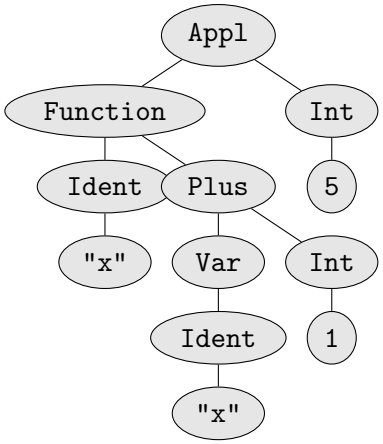
Example 2.20.

Concrete:

```
(Fun x -> x + 1) 5
```

Abstract:

```
Appl(
  Function(
    Ident "x",
    Plus(Var(Ident "x"),
      Int 1)),
  Int 5)
```



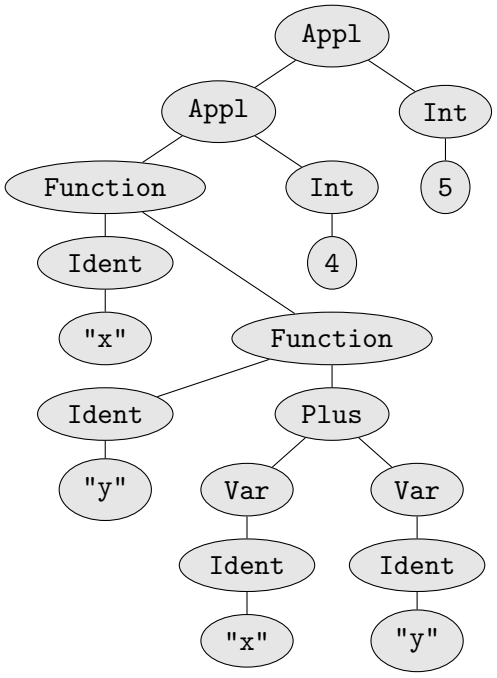
Example 2.21.

Concrete:

```
(Fun x -> Fun y ->
  x + y) 4 5
```

Abstract:

```
Appl(
  Appl(
    Function(
      Ident "x",
      Function(
        Ident "y",
        Plus(
          Var(Ident "x"),
          Var(Ident "y")
      ))),
    Int 4),
  Int 5)
```



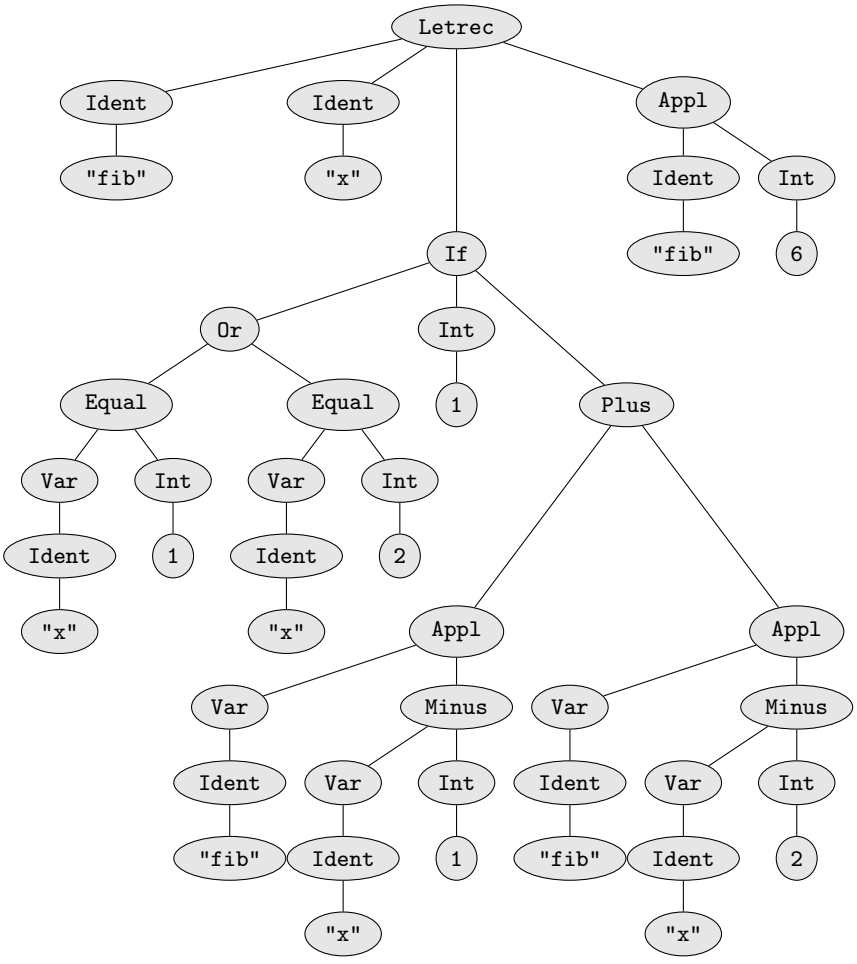
Example 2.22.

Concrete:

```
Let Rec fib x =
   If x = 1 Or x = 2 Then 1 Else fib (x - 1) + fib (x - 2)
In fib 6
```

Abstract:

```
Letrec(Ident "fib", Ident "x", If(Or(Equal(Var(Ident "x"), Int 1),
Equal(Var(Ident "x"), Int 2)), Int 1, Plus(Appl(Var(Ident "fib"),
Minus(Var(Ident "x"), Int 1)), Appl(Var(Ident "fib"), Minus(Var(Ident
"x"), Int 2)))), Appl(Var(Ident "fib"), Int 6))
```



注意，即使在抽象语法中，即使是简单的表达式也可能变得很长。仔细审查上述示例，并尝试一些你自己的额外示例。在编写编译器和解释器时，能够轻松地在抽象和具体语法之间切换是很重要的。

2.4 Operational Equivalence

数学对象空间中定义的最基本操作之一是等价关系（≅）。我们使用操作语义将程序定义为数学实体。因此，等价关系对程序也适用。

首先，我们被迫考虑我们直觉上所说的“等价”程序的含义。两个等价的事物可以互换；因此，两个等价的程序必须做同样的事情。然而，它们不必以完全相同的方式进行。例如，我们可以很容易地看出程序 `(1 + 1 + 1 - 1)` 和 `(2)` 是等价的，但前者需要更多的计算来评估。

因为两个等效程序可以互相替换，但可能具有不同的执行时间属性，所以我们可以在优化编译器和其他此类工具中使用操作等价性。操作等价性为这项工作提供了一个严格可靠的基石；我们不必担心优化会改变应用程序的行为，因为我们能从数学上证明这种改变是不可能的。操作等价性还定义了重构的过程，这是一个开发者可以改变程序结构以实现可维护性或增强而不改变程序行为的过程。

定义程序等价关系实际上并不像人们预期的那样简单。最初的思路是定义这种关系，使得两个程序在执行时总是产生相同的结果。然而，正如我们将看到的，这个定义是不够的，我们需要做一些工作才能得到一个令人满意的定义。

让我们先看看一些示例等价关系，以了解它们是什么。η-转换（或eta-转换）是其中一个有趣的等价关系的例子。它被定义为以下内容。

`Fun` x `->` $e \cong$
`Fun` z `->` `(Fun` x `->` e`)` z, 对于 z 在 e 中不是免费的

η- 转换类似于面向对象编程中的代理模式[12]。与我们的冻结/解冻语法密切相关的一项法律是

`Thaw (Freeze` e`)` $\cong e$

在两个例子中，一个表达式可以被另一个替换而不会产生不良影响（可能除了改变执行时间），所以我们说它们是等价的。然而，为了编写形式化的证明，我们需要发展一个更严格的等价定义。

2.4.1 Defining Operational Equivalence

让我们首先非正式地加强我们对操作等价性的定义。我们以莱布尼茨[18]的方式定义等价性：

Definition 2.15 (操作等价（非正式））**.** *Two program expressions are*等价于 *if and only if one can be replaced with the other at any place, and no external change in behavior will be noticed.*

我们希望研究可能开放程序的可比性，因为存在良好的可比性，例如 x `+ 1 - 1` $\cong x +$ 0。我们通过一个 **program context** 的概念来定义“在任何地方”，非正式地说，这是一个有某些空白的 **F♭** 程序（`(` `•` `)`）在其中。使用这个非正式定义，测试 $e_1 \cong e_2$ 是否大致相当于执行以下步骤（当然，对于所有可能的程序和所有可能的漏洞，当然）。

1. 选择任何程序上下文（即包含孔的程序）。2. 将 e_1 放在 `•` 位置并运行程序。3. 对 e_2 进行相同的操作。4. 如果可观察结果不同，则 e_1 不等同于 e_2。5. 对 *every possible context* 重复步骤 1-4。如果这些无限多个上下文中没有产生不同的结果，那么 e_1 等价于 e_2。

现在让我们详细阐述程序上下文的概念。考虑一个具有一些“空洞”的 **F♭** 程序（`(` `•` `)`）在其中打孔：将任何表达式中的一些子项替换为 `•`. 然后在此程序上下文中“孔填充” C，表示为$C[e]$，意味着替换 `•` 在 C 中包含 e。孔填充类似于替换，但没有关于约束或自由变量的担忧。它是无条件的直接替换。

让我们看看使用上面定义的 η-转换作为上下文和空缺填充的一个例子。让

$C \stackrel{\text{def}}{=}$ `(Fun z -> Fun x -> •) z`

现在，用 x `+ 2` 填充洞口很简单

```
((Fun z -> Fun x -> •) z)[x + 2] =
      (Fun z -> Fun x -> x + 2) z
```

最后，我们准备严格定义操作等价性。

Definition 2.16 (操作等价)**.** *$e \cong e'$ if and only if for all contexts C such that $C[e]$ and $C[e']$ are both closed expressions, $C[e] \Rightarrow v$ for some v if and only if $C[e'] \Rightarrow v'$ for some v'.*

另一种表述这个定义的方式是，如果在任何可能的环境中，C，一个表达式终止则另一个也终止，那么这两个表达式是等价的。我们称这种等价为*operational*等价，因为它基于语言的解释器，或者更确切地说，它基于操作语义。这个定义最有趣，也许也是最不直观的部分是，它没有提及v和v'之间的关系。事实上，在理论上它们可能不同。然而，直觉告诉我们，v和v'必须非常相似，因为等价性适用于任何可能的环境。

这个等价定义的唯一问题是其“近亲繁殖”的性质——没有从语言中分离出来的绝对等价标准。**Domain theory** 是一个数学学科，它通过现有程序定义了一个程序代数。

数学对象（完备和连续偏序）。我们在此处不讨论域理论，主要是因为它不能很好地推广到具有副作用的语言。[16] 探讨了操作语义与域理论之间的关系。

2.4.2 Properties of Operational Equivalence

在这一节中，我们提出了一些关于 $\mathbf{F}\flat$ 的一般等价原理。

Definition 2.17 (自反性)**.**

$e \cong e$

Definition 2.18 (对称)**.**

$e \cong e'$ *if* $e' \cong e$

Definition 2.19 (传递性)**.**

$e \cong e''$ *if* $e \cong e'$ *and* $e' \cong e''$

Definition 2.20 (同构)**.**

$C[e] \cong C[e']$ *if* $e \cong e'$

Definition 2.21 (β-等价)**.**

$(\texttt{(Fun}\ x\ \texttt{->}\ e\texttt{)}\ v) \cong (e[v/x])$

provided no free variables in v *become bound variables in* $e[v/x]$ *(i.e. no free variables are* 捕获*). Note that* v *being closed is one condition that ensures that no free variables will be captured.*

Definition 2.22 (η-等价)**.**

$(\texttt{Fun}\ x\ \texttt{->}\ e) \cong (\texttt{Fun}\ z\ \texttt{->}\ (\texttt{Fun}\ x\ \texttt{->}\ e)\ z)$

Definition 2.23 (α-等价)**.**

$(\texttt{Fun}\ x\ \texttt{->}\ e) \cong (\texttt{Fun}\ y\ \texttt{->}\ (e[y/x]))$

Definition 2.24.

$(n\ \texttt{+}\ n') \cong$ *the sum of* n *and* n'

Similar equations hold for `-`, `And`, `Or`, `Not`, *and* `=` *applied to concrete integer or boolean values.*

Definition 2.25 (法律)**. return** $\texttt{Let}\ x\ \texttt{=}\ e\ \texttt{In}\ x \cong e$

pure functional law $\texttt{Let}\ x_1\texttt{=}\ e_1\ \texttt{In}\ \texttt{Let}\ x_2\texttt{=}\ e_2\ \texttt{In}\ e_3 \cong \texttt{Let}\ x_2\texttt{=}\ e_2\ \texttt{In}\ \texttt{Let}\ x_1\texttt{=}\ e_1\ \texttt{In}\ e_3$
provided x_1/x_2 *do not occur freely in* e_1 *or* e_2*.*

operation execution ordering $e_1 \oplus e_2 \cong$ *Let* x_1 = e_1 *In Let* x_2 = e_2 *In* $x_1 \oplus x_2$
provided x_1/x_2 *do not occur freely in* e_1 *or* e_2. $\oplus$ *here is any binary operation, including function application.*

associativity *Let* x_1 = *(Let* $x_2 = e_1$ *In* e_2*)* *In* $e_3 \cong$ *Let* x_2 = e_1 *In Let* x_1 = e_2 *In* e_3
provided x_2 *does not occur freely in* e_3.

Let-β *Let* x = v *In* $e \cong e$[v29]
provided no free variables in v *become bound variables in* $e[v/x]$ *(i.e. no free variables are* 捕获*).*

Let-α *Let* x = e_1 *In* $e_2 \cong$ *Let* y = e_1 *In* $e2[y/x]$

Definition 2.26 (如果法律{v*}

(*If True Then* e_1 *Else* $e2) \cong e1$ (*If False Then* e_1 *Else* e_2) $\cong e_2$

Definition 2.27.

If $e \Rightarrow v$ *then* $e \cong v$

程序上的等价变换可用于证明计算结果，而不是直接使用解释器进行计算；这通常更容易。编译器优化的重要部分是应用保持等价的变换，如上述变换。

2.4.3 Examples of Operational Equivalence

为了巩固操作等价的概念（这个概念常常让编程语言理论的新手感到困惑），我们提供了许多等价和非等价表达式的例子。我们从两个不等价的表达式的非常简单的例子开始。

Lemma 2.5. *2* $\ncong$ *3*

Proof. 通过示例。令 $C \stackrel{\text{def}}{=}$ `If •= 2 Then 0 Else (0 0)`. C[2] $\Rightarrow$ 0 当 C[3] $\not\Rightarrow v$ 对任何 v。因此，根据定义，2 $\ncong$ 3。 □

注意，在上面的证明中，我们使用了表达式 `(0 0)`。这个表达式无法评估；应用规则仅适用于函数。因此，这个表达式在构建关于操作等价的证明时，是一个有意使代码陷入停滞的绝佳工具。其他类似的陷入停滞的表达式也存在，例如 `True + True`、`Not 5` 和永远受欢迎的 `(Fun x -> x x)(Fun x -> x x)`。

應該很明顯，我們可以使用第2.5引理中的方法來證明任何不相等的兩個整數在運算上不相等。但對於一個非值表達式，我們能夠做出什麼聲明呢?

Lemma 2.6. *x* $\ncong$ *x + 1 - 1.*

首先一看，这种不等价性可能看起来反直觉。但证明相当简单：

Proof. 通过例子。设 $C \stackrel{\text{def}}{=}$ `(Fun x ->` $\bullet$`) True`. 然后是 C[`x`] $\Rightarrow$ `True`。C[`x + 1 - 1`] $\equiv$ `(Fun x -> x + 1 - 1) True`，无法评估，因为没有规则允许我们评估 `True + 1`。因此，根据定义，`x` $\not\cong$ `x + 1 - 1`。 □

类似证明可以用来证明不等价性，例如 `Not(Not(`e`))` $\not\cong e$。关键在于，$\mathbf{F}\flat$ 中的一些规则区分了整数值和布尔值。因此，如果一方对其将评估的值的类型做出假设，而另一方没有，则等价性无法成立。

我们已经证明了不等价性；我们能否证明等价性？事实证明，某些等价性可以证明，但这个过程要复杂得多。迄今为止，我们证明不等价性的依据是，它们只需要展示一个例子，其中表达式产生不同的结果。然而，等价性的证明必须证明在无限多个上下文中不存在这样的例子。例如，考虑以下：

Lemma 2.7. *If $e \not\Rightarrow v$ for any v, $e' \not\Rightarrow v'$ for any v', and both e and e' are closed expressions, then $e \cong e'$. For example,* ***`(0 0)`*** $\cong$ ***`(Fun x -> x x)(Fun x -> x x)`****.*

首先，我们需要一个定义：

Definition 2.28 (触摸)**.** *An expression is said to* ***touch*** *one of its subexpressions if the evaluation of the expression causes the evaluation of that subexpression.*

这是，`If True Then 0 Else 1` *touches*子表达式`0`因为它是当整个表达式评估时评估的。`1`没有被触及，因为表达式的评估从未导致`1`被评估。

我们现在可以证明引理2.7。

Proof. 通过反证法。不失一般性，假设存在某个上下文 C，使得 $C[e] \Rightarrow v$，而 $C[e'] \not\Rightarrow v'$ 对任何 v' 都成立。

因为 e 是一个封闭表达式，对于所有 v 和所有接触 e 的上下文 C'，有 $C'[e] \not\Rightarrow v$。因为 $C[e] \Rightarrow v$，我们知道 C 不接触空洞。

因为 C 没有触及孔，孔的评估方式不会影响 C 的评估；也就是说，对于某些 v^*，$C[e^*] \Rightarrow v^*$ 必须对所有 e^* 或对没有任何 e^* 为真。

因为 $C[e] \Rightarrow v$，$C[e^*] \Rightarrow v^*$ 对于某些 v^* 对于所有 e^*。但是 $C[e'] \not\Rightarrow v'$ 对于任何 v'。因此，通过反证法，C 不存在。因此，根据定义，$e \cong e'$。 □

上述证明比我们遇到的任何不等价证明都要长且复杂得多，而且它甚至不是很稳健。例如，它可以受益于一个证明，即一个封闭表达式不能通过替换规则来改变（这在上面是理所当然的）。此外，上述证明甚至没有证明一个非常有用的事实！更有趣的是，当其中一个表达式以比另一个更理想的方式（更快、更少的资源等）执行时，证明两个表达式的操作等价性，以激励编译器优化。

引理2.7虽然有效，但在证明操作等价性所涉及的复杂性方面。证明等价性竟然如此困难；甚至证明 `1 + 1` $\cong$ `2` 也相当具有挑战性。参见[16]了解更多关于这个主题的信息。

2.4.4 The λ-Calculus

我们简要考虑**λ-calculus**。在第2.3节中，我们看到了如何编码元组、列表、Let语句、冻结和解冻，甚至递归在**F♭**中。编码方法非常强大，同时也为我们提供了一种基于对简单语言的理解来理解复杂语言的方法。偶数、布尔值和if-then-else语句都是可编码的，尽管我们将跳过这些主题。因此，所需的一切就是函数和应用来制作一个图灵完备的编程语言。这种语言被称为**pure λ calculus**，因为函数通常写成$\lambda x.e$而不是`Fun` x `->` e。

执行在 λ 计算中极其简单明了。主要要点如下。

- 即使具有自由变量的程序也可以执行（或在 *reduce* 在 λ-计算术语中）。
- 减少可以发生在任何地方，例如在尚未调用的函数体内。
- $(\lambda x.e)e' \Rightarrow e$[v15] 是唯一的约简规则，称为 β-约简。（它有一个特殊的条件，即它必须是 *capture-free*，即 e' 中的自由变量在结果中不成为约束变量。允许在任何地方进行约简会带来一些复杂性，捕获就是其中之一。）

这种计算形式在概念上非常有趣，但与实际计算机语言的执行方式相去甚远，所以我们在这里并没有给予它过多的关注。

Exercises

Exercise 2.1. 您会如何修改Sheep语言以允许术语$bah, baah, \cdots$而不排除任何已允许的术语?

Exercise 2.2. 该术语 it 在青蛙语中吗？为什么或为什么不?

Exercise 2.3. 是否有可能在Frog中构造一个不使用终端t的项？如果是，请给出一个例子。如果不是，为什么?

Exercise 2.4. 完成第2.2.1节中描述的布尔语言的运算语义定义，通过编写`Or`和`Implies`的规则。

Exercise 2.5. 为什么不直接使用解释器而忘记操作语义方法呢?

Chapter 3

Tuples, Records, and Variants

在第二章中，我们看到了使用只有函数和应用的语言，我们可以表示诸如元组和列表等高级编程结构。然而，我们指出，这些编码存在基本问题，例如效率低下，以及它们必然将细节暴露给程序员，这使得在实际操作中难以处理且危险。回想一下，我们如何将第二章中对一对的编码作为函数应用；显然这是错误的行为。在本章中，我们探讨如何将这些高级特性构建到语言中，即元组和记录，并通过考察变体来结束本章。

3.1 Tuples

编程中最基本的数据聚合形式之一是 **pairing** 的概念。使用对或 2-元组，几乎可以表示任何数据结构。三重可以表示为（$1, (2, 3)$），并且一般 n-元组可以用类似的方式用对表示。记录和 C 风格的结构可以用（标签，值）对的集合（n-元组）表示。甚至对象也可以从对构建，但这有点过分（就像将对编码为函数一样过分）。

第二章中，我们展示了基于函数的成对编码。这种成对表示方法有两个问题。首先，这种表示方法效率低下。更重要的是，成对的行为略有错误，因为我们可以将它们像函数一样应用。要真正正确处理成对，我们需要直接将它们添加到语言中。我们可以相当直接地将成对添加到 **F**♭ 中。我们展示了如何将成对功能添加到解释器中，并将成对的操作语义留给读者作为练习。

首先，我们在解释器中扩展了 `expr` 类型，包括以下内容。

```
type expr =
  ...
| Pr of expr * expr | Left of expr | Right of expr
```

接下来，我们将以下条款添加到我们的 `eval` 函数中。

```
let rec eval e =
  match e with
    ...
  | Pr(e1, e2) -> Pr(eval e1, eval e2)
  | Left(expr) -> (match eval expr with
      Pr(e1,e2) -> e1
    | _ -> raise TypeMismatch)
  | Right(expr) -> (match eval expr with
      Pr(e1, e2) -> e2
    | _ -> raise TypeMismatch)
```

注意，我们的对是 *eager*，也就是说，对的两部分被评估，并且必须是对本身的值，这样对本身才被认为是值。例如，`(2, 3+4)` ⇒ `(2, 7)`。OCaml元组表现出这种行为。还要注意，我们现在的值空间更大了。它包括:

- 数字 `0`，`1`，`-1`，`2`，`-2`，⋯
- 布尔值 `True`，`False`
- 函数 `Fun x ->` ...
- 对 (v_1, v_2)

Exercise 3.1. 我们如何编写我们的解释器以非急切的方式处理成对元素？换句话说，解释器中需要包含什么，以便将 (e_1, e_2) 视为值（而不是只有 (v_1, v_2) 被视为值）？

现在我们有了2元组，编码3元组、4元组以及n-元组变得容易。我们只需将它们编码为 $(1, (2, (3, (\ldots, n))))$。正如我们之前看到的，列表可以编码为$n$-元组。

3.2 Records

记录是元组的变体，其中字段具有名称。与元组相比，记录具有多个优点。主要优点是命名字段。从软件工程的角度来看，命名字段“邮编”远优于“元组的第三个元素”。记录字段的顺序是任意的，与元组不同。

记录与对象比元组更接近。我们可以通过记录多态来编码对象多态。记录多态在第3.2.1节中讨论。使用记录来编码对象的动机是子类由其父类的字段超集组成，并且两个类的实例都可以在超类上下文中使用。同样，记录多态允许在字段子集的上下文中使用记录，因此映射相当自然。我们将在第5章中使用记录来模拟对象。

我们的 **F♭** 记录将与 OCaml 记录具有相同的语法。也就是说，记录以 $\{l_1{=}e_1;\ l_2{=}e_2;\ \ldots;\ l_n{=}e_n\}$ 的形式编写，选择以 $e.l_k$ 的形式编写，它选择

l_k的值来自记录e。我们使用l作为遍历标签的元变量，就像我们使用e作为表示表达式的元变量一样；一个实际的记录例如{x=5; y=7; z=6}，所以这里的x是一个实际的标签。

如果记录始终是静态已知的固定大小，即，如果我们编写代码时已知它们是固定大小的，那么我们可以简单地映射标签到整数，并将记录编码为元组。例如，

{x=5; y=7; z=6} = (5, (7, 6))
e.x = Left e
e.y = Left (Right e)
e.z = Right (Right e)

显然，这会导致代码难看、难以阅读，但对于C风格的struct来说，它是可行的。但是，在记录可以收缩和增长的情况下，这种编码在本质上太弱了。C++ struct可以是彼此的子类型，因此未声明的字段实际上可能在运行时存在。

另一方面，对可以很好地编码为记录。对 (3, 4) 可以简单地编码为记录 {l=3; r=4}。更复杂的对，如用于表示列表的对，也可以编码为记录。例如，表示列表 [3; 4; 5; 6] 的对 (3, (4, (5, 6))) 可以编码为记录 {l=3; r={l=4; r={l=5; r=6}}}。

此列表编码的变体在4.3.2节中的归并排序示例中使用。这种变体将上述列表编码为 {l=3; r={l=4; r={l=5; r={l=6; r=emptylist}}}}。这种编码具有一个很好的特性，即值始终包含在l字段中，其余列表始终包含在r字段中。这更接近于OCaml、Scheme和Lisp等真实语言表示列表的方式（回想一下我们在OCaml中如何编写类似 let (first::rest) = mylist的语句）。

3.2.1 Record Polymorphism

记录不仅仅使程序更具可读性。例如，如果您有 {size=10; weight=100} 和 {weight=10; name="Mike"}，这两个记录中的任何一个都可以传递给如下函数

Fun x -> x.weight.

在脚本语言中，这种灵活性非正式地被称为 *duck typing* – 如果 Fun d -> d.quack () 在调用时不会产生运行时错误，我们将假设传递给它的 d 是一只鸭子，因为它像鸭子一样嘎嘎叫。

这种灵活性称为在类型语言中的 **subtype polymorphism**。在上面的函数中，x 可以是具有 weight 字段的任何记录。记录上的子类型多态称为 **record polymorphism**。OCaml 不允许记录多态，因此上述代码的 OCaml 版本将无法通过类型检查。

在面向对象的编程语言中，子类型多态被称为 **object poly- morphism**，或者更常见地，简单地称为 *polymorphism*。遗憾的是，后者与 OCaml 的参数多态容易混淆。

3.2.2 The F♭R Language

我们现在将定义 **F♭R** 语言：具有记录的 **F♭**。再次，我们将集中讨论解释器，并将操作语义留给读者作为练习。

首先我们需要考虑的是如何表示记录标签。记录标签是符号，因此我们可以使用我们的标识符（`Ident "x"`）作为标签，但最好将记录标签视为一种不同的类型。例如，标签永远不会被绑定或替换。因此，在我们的解释器中，我们将定义一个新的类型。

```
type label = Lab of string
```

接下来，我们需要一种表示记录本身的方法。记录可以是任意长度，因此需要一个 (*label*, *expression*)-对的列表。此外，我们还需要一种表示选择的方法。**F♭R** `expr` 类型现在看起来如下。

```
type expr = ...
| Record of (label * expr) list | Select of expr * label
```

Let's look at some concrete to abstract syntax examples for our new language.

Example 3.1.

```
{size=7; weight=255}

Record [(Lab "size", Int 7); (Lab "weight", Int 255)]
```

Example 3.2.

```
e.size

Select(Var(Ident "e"), Lab "size")
```

此外，我们的价值观定义现在必须扩展到包括记录。具体来说，
$\{l_1{=}v_1;\ l_2{=}v_2;\ \ldots;\ l_n{=}v_n\}$ 是一个值，前提是 $v_1, v_2, \ldots, v_n$ 也是值。

最后，我们将必要的规则添加到我们的解释器中。因为记录可以是任意长度，所以在评估它们时我们需要做更多的工作。OCaml中的`let-rec-and`语法用于声明相互递归的函数，我们下面将使用它。

```
(* A function to project a given field *)

let rec lookupRecord body (Lab l) = match body with
  [] -> raise FieldNotFound
| (Lab l', v)::t -> if l = l' then v else lookupRecord t (Lab l)

(* The eval function, with an evalRecord helper *)

let rec eval e = match e with
  ...
| Record(body) -> Record(evalRecord body)
```

```
| Select(e, l) -> match eval e with
    Record(body) -> lookupRecord body l
  | _ -> raise TypeMismatch

and evalRecord body = match body with
  [] -> []
| (Lab l, e)::t -> (Lab l, eval e)::evalRecord t
```

注意，我们的解释器正确处理了空记录 {}，因为它自身计算为自身，根据定义，它是一个值。

Interact with F♭SR. 我们可以使用我们的 **F♭SR** 解释器来探索记录（**F♭SR** 是包含记录和状态的 **F♭**，并在第 4 章中介绍）。首先，让我们尝试一个简单的例子来演示记录的急切求值。

```
# {one = 1; two = 2;
   three = 2 + 1; four = (Fun x -> x + x) 2};;
==> {one=1; two=2; three=3; four=4}
```

接下来，让我们尝试一个更有趣的例子，其中我们使用记录来编码列表。注意，我们将 `emptylist` 定义为 `-1`。下面的函数用于计算列表中所有值的总和（假设它有一个整数列表）。

```
# Let emptylist = 0 - 1 In
  Let Rec sumlist list =
    If list = emptylist Then
      0
    Else
      (list.l) + sumlist (list.r) In
  sumlist {l=1; r={l=2; r={l=3; r={l=4; r=emptylist}}}};;
==> 10
```

3.3 Variants

我们一直在OCaml中使用变体，作为表达式类型 `expr`。现在我们更深入地研究无类型变体。OCaml实际上有两种（不兼容）的变体形式，常规变体和多态变体。在我们工作的无类型上下文中，OCaml的多态变体更合适，我们将使用这种形式。

我们简要对比OCaml中两种变体形式，供不熟悉多态变体的读者参考。回忆一下，在OCaml中，常规变体首先被声明为类型

```
type feeling =
    Vaguely of feeling | Mixed of feeling * feeling |
    Love of string | Hate of string | Happy | Depressed
```

允许 `Vaguely(Happy)`、`Mixed(Vaguely(Happy),Hate("Fred"))` 等。多态变体不需要类型声明；因此，对于上述内容，我们可以直接写出 ‘`Vaguely(‘Happy)`、‘`Mixed(‘Vaguely(‘Happy),‘Hate("Fred"))` 等。每个变体名称前必须加上 ‘ 前缀，表示它是一个多态变体名称。

3.3.1 Variant Polymorphism

与记录一样，变体是多态的。在记录中，许多不同形式的记录可以通过特定的选择（任何具有所选字段的记录）通过。在变体中，多态性是双重的，因为许多不同形式的 `match` 语句可以处理给定的变体。

3.3.2 The F♭V Language

我们现在将定义 **F♭V** 语言，**F♭** 以及 . . . **V** 变体。

新的语法需要变体语法和匹配语法。就像我们限制函数只能有一个参数一样，我们也限制变体构造函数只能有一个参数；多参数或零参数的变体必须进行编码。在具体语法中，我们通过 $n(e)$ 构建 n 命名变体和 e 其参数，例如 `‘Positive(3)`。然后通过匹配使用变体：`Match` e `With` $n_1(x_1)$ `->` e_1 `|` . . . `|` $n_m(x_m)$ `->` e_m。我们不定义一个通用的模式 `match`，就像在 OCaml 中找到的那样——我们的 `Match` 将一次匹配一个变体字段，并且不会在变体之外工作。

抽象语法对于 **F♭V** 如下。首先，每个变体需要一个名称。

```
type name = Name of string
```

F♭V 抽象语法 `expr` 类型现在看起来像

```
type expr = ...
  | Variant of (name * expr)
  | Match of expr * (name * ident * expr) list
```

让我们看看我们新语言的一些具体到抽象语法的示例。

Example 3.3.

```
‘Positive(4)

Variant(Name "Positive", Int 4)
```

Example 3.4.

```
Match e With
   ‘Positive(x) -> 1 | ‘Negative(y) -> -1 | ‘Zero(p) -> 0

Match(Var(Ident("e")),[(Name "Positive",Ident "x",Int 1);
                       (Name "Negative",Ident "y",Int -1);
                       (Name "Zero",Ident "p",Int 0)])
```

注意，在这个例子中，我们不能只有一种变体 `Zero`，因为不允许0元变体，必须提供一个虚拟参数。多参数变体可以通过一个单参数变体编码，该变体在成对或记录上（由于我们在 **F♭V** 中既没有成对也没有记录，唯一的选择是将第2.3.4节中使用的成对编码用于 **F♭**）。

此外，我们对于 **F♭V** 的定义也必须从 **F♭** 的定义扩展，包括变体：$n(v)$ 是一个值，前提是 v 是。为了定义 **F♭V** 执行的含义，我们通过以下两条规则扩展了 **F♭** 的操作语义：

(Variant Rule)
$$\frac{e \Rightarrow v}{n(e) \Rightarrow n(v)}$$

(`Match` Rule)
$$\frac{e \Rightarrow n_j(v_j), \quad e_j[v_j/x_j] \Rightarrow v}{\begin{array}{l}\texttt{Match } e \texttt{ With} \\ \quad n_1(x_1) \texttt{ -> } e_1 \texttt{ | } \ldots \\ \texttt{ | } n_j(x_j) \texttt{ -> } e_j \texttt{ | } \ldots \\ \texttt{ | } n_m(x_m) \texttt{ -> } e_m\end{array} \Rightarrow \quad v}$$

变体规则构建一个新的标签为 n 的变体；其参数被贪婪地评估为一个值，就像在 OCaml 中一样：`‘Positive(3+2)` $\Rightarrow$ `‘Positive(5)`。`Match` 规则首先计算要匹配到变体的表达式 e，然后在匹配中查找该变体，找到 $n_j(x_j)$ `->` e_j，然后根据变体参数的值评估 e_j，其中变量为 x_j。

Example 3.5.

```
Match ‘Grilled(3+1) With
  ‘Stewed(x) -> 4 + x |
  ‘Grilled(y) -> 2 + y   ⇒ 6, 因为
‘Grilled(3+1) ⇒ ‘Grilled(4) 和 (2 + y)[4/y] ⇒ 6
```

Exercise 3.2. 扩展 **F♭V** 语法和操作语义，使得 `Match` 表达式始终具有以下形式的最终匹配：“`| _ -> e`”。这是否比旧的一个严格更具表现力，还是不是?

Variants and Records are Duals 这里我们看到记录的定义是如何被建模为一个变体的使用，而记录的使用则是变体的定义。

变体是记录的对偶：一个记录是这个字段 *and* 那个字段 *and* 那个字段；一个变体是这个字段 *or* 那个字段 *or* 那个字段。由于它们是对偶，*defining* 一个记录看起来像 *using* 一个变体，定义一个变体看起来像使用一个记录。

变体可以直接编码记录，反之亦然，在编程类比中，这与德摩根定律允许逻辑与被编码为或，反之亦然的方式相同：p `Or` $q = $ `Not(Not` p `And Not` q`)`；
p `And` $q = $ `Not(Not` p `Or Not` q`)`。

变体可以使用以下记录进行编码。

`Match` s `With` ‘$n_1(x_1)$ `->` e_1 `|` $\ldots$ `|` ‘$n_m(x_m)$ `->` $e_m = s\{n_1$`=Fun` x_1 `->` e_1`;` $\ldots$`;` n_m`=Fun` x_m `->` $e_m\}$
‘$n(e) =$ `(Fun` x `->` `(Fun` r `->` $r.n\ x$ `))` e

编码的难点在于定义必须转换为使用，反之亦然。这是通过函数实现的：一个注入被建模为一个函数，该函数是 *given* 一个记录，并将选择指定的字段。

这里是如何使用变体对记录进行编码的。

$\{l_1$=e_1; ...; l_n=$e_n\}$ = (Fun k_1 ->...Fun k_n ->Fun s ->
Match s With ‘$l_1(x)$ -> k_1 | ... | ‘$l_n(x)$ -> k_n)(e_1)...(e_n)
$e.l_k = e$ ‘l_k(0)

在上方，x 是任何新变量。

另一个关于记录和变体之间双重性的有趣方面是，*both*记录和变体可以编码对象。变体是一种消息，对象是消息上的一个案例。在对象的变体编码中，很容易将消息作为一等实体传递。然而，使用变体来编码对象会使对象难以进行类型检查，这就是为什么我们认为对象更像记录。

Chapter 4

Side Effects: State and Exceptions

我们现在将离开纯函数式编程的世界，开始考虑具有副作用的语言。目前，我们将仅关注两种特定的副作用：状态和异常。然而，还有许多其他类型的副作用，包括以下内容。

- Goto-语句或循环中断，类似于异常。注意，循环中断需要循环，而循环需要状态。
- 输入和输出。
- 分布式消息传递。

4.1 State

像 **F♭**、**F♭R** 和 **F♭V** 这样的语言是纯函数式语言。一旦我们向一种语言添加任何类型的副作用，它就不再是纯函数式语言了。副作用是非局部的，意味着它们可以影响程序的其它部分。例如，考虑以下 OCaml 代码。

```
let x = ref 9 in
  let f z = x := !x + z in
    x := 5; f 5; !x
```

这个表达式计算结果为 `10`。尽管在声明 `f` 时，`x` 被定义为对 `9` 的引用，但最后一行将 `x` 设置为对 `5` 的引用，并在应用 `f` 期间，`x` 被重新分配为对 `(5 + 5)` 的引用，最终在最后一行解引用时变为 `10`。显然，副作用的使用使得程序分析变得非常困难，因为它不如函数式程序那样声明式。在查看具有副作用的程序时，必须检查代码的 *entire* 部分，以了解哪些副作用影响了哪些表达式的结果。因此，只在强烈需要时使用副作用是一种良好的编程道德。

让我们首先非正式地讨论引用的语义。本质上，当程序中创建一个引用时，会创建一个包含指定值的单元格，而引用本身是该单元格的指针。这些引用的一个好比喻是

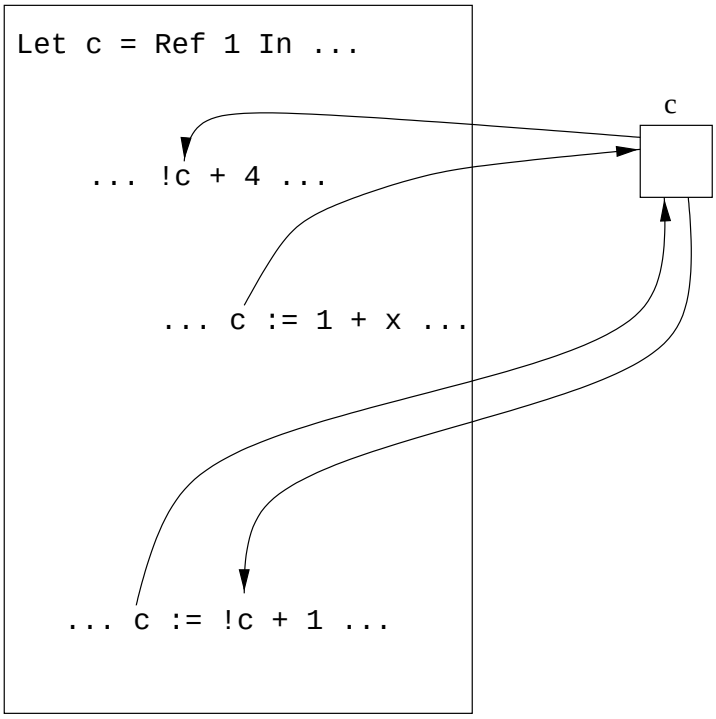


图4.1：参考细胞的“接线盒”视图。

单元可以想象成每个都是一个接线盒。然后，每个赋值都是一个进入接线盒的路径，每个读取或解引用都是一个离开接线盒的路径。引用单元或接线盒位于程序的一侧，允许程序的不同部分相互通信。这种“远程通信”可以是一种有用的编程范式，但同样应该谨慎使用。图4.1说明了这个隐喻。

在C++和Java的世界中，非const（或非final）的全局变量是最臭名昭著的引用形式。虽然全局变量使得执行某些任务变得容易，但它们通常使得程序难以调试，因为它们可以在代码的任何地方被修改。

4.1.1 The F♭S Language

添加状态到我们的语言需要对迄今为止我们考虑过的纯函数语言进行重大修改。因此，在查看解释器之前，我们将花一些时间为我们基于状态的语言开发一个坚实的操作语义。我们将定义一种语言 **F♭S**：**F♭** 具有状态。

最显著的差异是，我们迄今为止所考察的纯函数操作语义中，评估关系 $e \Rightarrow v$ 并不足以捕捉基于状态的语言的语义。现在，表达式的评估可以产生副作用，我们的 $\Rightarrow$ 规则需要以某种方式包含这一点。具体来说，我们需要一个地方来记录所有这些副作用：一个 **store**。在 C 语言中，存储是栈和堆，内存位置通过它们的地址进行引用。当我们编写我们的

解释器，我们只能访问堆，因此我们需要创建一个抽象存储来记录副作用。

Definition 4.1 (存储). *A **store** is a finite map $c \mapsto v$ of cell names to values.*

单元名称是内存位置的抽象表示。因此，存储是一个运行时堆的抽象。C堆是存储的低级实现，其中单元名称仅仅是数值内存地址。从数据结构的角度来看，存储也称为*dictionary*。我们写Dom(S)来指代这个有限映射的域，即它映射的所有单元的集合。

让我们首先将 **F♭** 的具体语法扩展到包括 **F♭S** 表达式。增加的内容如下。

- 引用，`Ref` e。
- 作业，e `:=` e'。
- 解引用，`!`e。
- 单元格名称，c。

我们需要单元格名称，因为 `Ref 5` 需要评估为存储中的位置，c。因为单元格名称指的是堆中的位置，而堆最初为空，所以程序在执行开始时没有单元格名称。

尽管不是 **F♭S** 语法的一部分，我们仍需要一个表示对存储操作的表达方式。我们用 $S\{c \mapsto v\}$ 来表示扩展或修改以包含映射 $c \mapsto v$ 的存储 S。我们用 $S(c)$ 来表示存储 S 中单元格 c 的值。

现在我们已经发展了存储的概念，我们可以为 **F♭S** 定义一个令人满意的评估关系。评估如下所示。

$$\langle e, S_0 \rangle \Rightarrow \langle v, S \rangle,$$

在计算开始时，S_0最初为空，而S是计算结束时最终的存储。在评估过程中，单元格，c，将开始出现在程序语法中，作为内存位置的引用。由于单元格不需要评估，因此**F♭S**的值空间也包括单元格，c。

Evaluation Rules for F♭S

最后，我们准备好为 **F♭S** 编写评估规则。我们需要考虑存储来修改所有的 **F♭** 规则（记住，我们的评估规则现在是 $\langle e, S_0 \rangle \Rightarrow \langle v, S \rangle$，而不仅仅是 $e \Rightarrow v$）。我们通过 **threading** 沿着控制流修改存储来实现这一点。这样做会在规则之间引入更多的依赖性，甚至包括那些不直接操作存储的规则。我们将重写 **F♭S** 的函数应用规则，以说明其他规则需要进行的更改类型。

(Function Application)

$$\frac{\langle e_1, S_1 \rangle \Rightarrow \langle \texttt{Fun } x \texttt{ -> } e, S_2 \rangle, \quad \langle e_2, S_2 \rangle \Rightarrow \langle v_2, S_3 \rangle, \quad \langle e[v_2/x], S_3 \rangle \Rightarrow \langle v, S_4 \rangle}{\langle e_1\ e_2, S_1 \rangle \Rightarrow \langle v, S_4 \rangle}$$

注意这里存储的变化 *threaded* 通过不同的评估，展示了在一个地方的存储变化如何传播到其他地方的存储，以及反映缩进评估顺序的固定顺序。我们新的内存操作规则如下。

$$\textit{(Reference Creation)} \qquad \frac{\langle e, S_1\rangle \Rightarrow \langle v, S_2\rangle}{\langle \texttt{Ref}\ \ e, S_1\rangle \Rightarrow \langle c, S_2\{c \mapsto v\}\rangle,\ \text{for } c \notin \mathrm{Dom}(S_2)}$$

$$\textit{(Dereference)} \qquad \frac{\langle e, S_1\rangle \Rightarrow \langle c, S_2\rangle}{\langle \texttt{!}e, S_1\rangle \Rightarrow \langle v, S_2\rangle,\ \text{where } S_2(c) = v}$$

$$\textit{(Assignment)} \qquad \frac{\langle e_1, S_1\rangle \Rightarrow \langle c, S_2\rangle, \quad \langle e_2, S_2\rangle \Rightarrow \langle v, S_3\rangle}{\langle e_1\ \texttt{:=}\ e_2, S_1\rangle \Rightarrow \langle v, S_3\{c \mapsto v\}\rangle}$$

这些规则可能难以评估，因为商店需要在评估的每个点上保持最新。让我们看看一些示例表达式，以了解这是如何工作的。

Example 4.1.

```
!(!(Ref Ref 5)) + 4
```

$\langle$`!(!(Ref Ref 5)) + 4`$\rangle, \{\} \Rightarrow \langle$`9`$, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
  $\langle$`!(!(Ref Ref 5))`$, \{\}\rangle \Rightarrow \langle$`5`$, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
    $\langle$`!(Ref Ref 5)`$, \{\}\rangle \Rightarrow \langle c_1, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
      $\langle$`Ref Ref 5`$, \{\}\rangle \Rightarrow \langle c_2, \{c_1 \mapsto$ `5`$, c_2 \mapsto c_1\}\rangle$, because
        $\langle$`Ref 5`$, \{\}\rangle \Rightarrow \langle c_1, \{c_1 \mapsto$ `5`$\}\rangle$

Example 4.2.

```
(Fun y -> If !y = 0 Then y Else 0) Ref 7
```

$\langle$`(Fun y -> ...) Ref 7`$, \{\}\rangle \Rightarrow \langle$`0`$, \{c_1 \mapsto$ `7`$\}\rangle$, because
  $\langle$`Ref 7`$, \{\}\rangle \Rightarrow \langle c_1, \{c_1 \mapsto$ `7`$\}\rangle$, and
  $\langle$`(If !y = 0 Then y Else 0)`$[c_1/$`y`$], \{c_1 \mapsto$ `7`$\}\rangle \Rightarrow$
    $\langle$`0`$, \{c_1 \mapsto$ `7`$\}\rangle$, because
    $\langle$`!`c_1 `= 0`$, \{c_1 \mapsto$ `7`$\}\rangle \Rightarrow \langle$`False`$, \{c_1 \mapsto$ `7`$\}\rangle$, because
      $\langle$`!`$c_1, \{c_1 \mapsto$ `7`$\}\rangle \Rightarrow \langle$`7`$, \{c_1 \mapsto$ `7`$\}\rangle$

F♭S Interpreters

正如我们不得不修改我们的 **F♭S** 操作语义中的评估关系以支持状态一样，编写一个 **F♭S** 解释器也需要一些额外的工作。有两种明显的方法可以采取，我们将在下面讨论这两种方法。第一种方法涉及模仿操作语义，并在表达式和存储一起定义评估。这种方法产生了一个 *functional* 解释器，其中 `eval(`e, S_0`)`

对于表达式 e 和初始状态 S_0 返回元组 (v, S)，其中 v 是结果值，S 是最终状态。

第二和更高效的设计涉及在一个全局、可变的字典结构中跟踪状态。这通常是实际实现的方式。这种方法导致更熟悉的评估语义，即 `eval` e 返回 v。显然，这样的解释器不再具有功能性，而是 *imperative*。我们希望这样的解释器能够忠实实现 **F♭S** 的操作语义，因此我们理想情况下希望有一个定理表明这种方法与第一种方法等价。然而，证明这样的定理将是困难的，主要是因为我们的证明将依赖于 OCaml 的操作语义或我们选择的任何实现语言。因此，我们将基于良好的信念认为这两种方法确实是等价的。

The Functional Interpreter 功能解释器以 *functional* 方式实现了一种 *stateful* 语言。它在评估过程中传递状态，就像我们在定义操作语义时做的那样。通过沿线程传递状态，命令式风格的编程可以被“黑客”成函数式风格，并且有常规方法通过任何函数程序传递状态，即 *monads*。在20世纪60年代末，Strachey首次采用了线程方法来以函数方式建模状态。请注意，操作语义是 *always* 纯函数式的，因为数学始终是纯函数式的。

为了实现函数解释器，我们使用整数键到值的有限映射来模拟存储。下面展示了实现的框架。

```
(* Declare all the expr, etc types globally for convenience. *)

(* The store functionality is a separate module.  *)

module type STORE =
  sig
   (* ... *)
  end

(* The Store structure implements a (functional) store.  A simple
 * implementation could be via a list of pairs such as
 *
 *   [((Cell 2),(Int 4)); ((Cell 3),Plus((Int 5),(Int 4))); ... ]
 *)

module Store : STORE =

type store = (* ... *)

 struct
  let empty = (* initial empty store *)
  let fresh = (* returns a fresh Cell name *)
    let count = ref 0 in
```

```
    function () -> ( count := !count + 1; Cell(!count) )
  (* Note: this is not purely functional!  It is quite
   * difficult to make fresh purely functional.
   *)

(* Look up value of cell c in store s *)
   let lookup (s,c) = (* ... *)

(* Add or modify cell c to point to value v in the store s.
 * Return the new store.
 *)
   let modify(s,c,v) =  (* ... *)
  end

(* The evaluator is then a functor taking a store module *)

module FbSEvalFunctor =
  functor (Store : STORE) ->
  struct

    (* ... *)


    let eval (e,s) = match e with
      (Int n) -> ((Int n),s) (* values don't modify store *)
    | Plus(e,e') ->
        let (Int n,s') = eval(e,s) in
        let (Int n',s'') = eval(e',s') in
        (Int (n+n'),s'')

(* Other cases such as application use a similar store
 * threading technique.
 *)
    | Ref(e) -> let (v,s') = eval(e,s) in
                let c = Store.fresh() in
                   (c,Store.modify(s',c,v))
    | Get(e) -> let (Cell(n),s') = eval(e,s) in
      (Store.lookup(Cell(n)),s')
    | Set(e,e') ->  (* exercise *)

  end

module FbSEval = FbSEvalFunctor(Store)
```

The Imperative Interpreter 状态性的命令式 **F♭S** 解释器比其函数式对应物更高效，因为不需要线程。在命令式

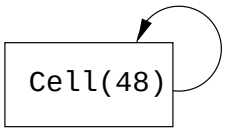


图4.2：一个简单循环。

解释器中，我们将存储表示为字典结构（类似于Java的`HashMap`类或C++ STL `map`模板）。`eval`函数不需要额外的存储参数，前提是它有全局字典的引用。与存储无关的规则，如`Plus`和`Minus`，对存储一无所知。只有直接存储评估规则，`Ref`、`Set`和`Get`实际上扩展、更新或查询存储。一个好的评估器还会定期进行垃圾回收，或删除不需要的存储元素。垃圾回收在4.1.4节中简要讨论。状态解释器的实现留给读者作为练习。

Side-Effecting Operators

现在我们有一个可变的存储，我们的代码具有返回值之外的性质，即副作用。如序列（`;`）、`While`-和`For`循环之类的运算符现在变得相关。这些句法概念可以轻松地定义为宏，因此我们不会将它们添加到官方的**F♭S**语法中。宏定义如下。

e_1`;` e_2 = `(Fun x ->` e_2`)` e_1
`While` e `Do` e' = `Let Rec f x = If` e `Then f` e' `Else 0 In f 0`

Exercise 4.1. 为什么在像 **F♭** 这样的纯函数式语言中，序列化和循环操作是不相关的？

4.1.2 Cyclical Stores

一个有趣的现象出现在有状态的语言中。可以创建一个 **cyclical store**，即内容是指向自身的单元格。创建循环存储相当容易：

```
Let x = Ref 0 in x := !x
```

这是最简单的存储周期，其中单元格直接指向自身。这种存储方式如图4.2所示。

Exercise 4.2. 在上述示例中，`!!!!!!!!x` 返回什么？能否用 OCaml 编写类似上面的存储周期？为什么或为什么不呢？（**Hint**：这种表达式的类型是什么？）

更微妙的一种存储周期形式是当函数被放置在单元格中，函数体引用了该单元格。考虑以下示例。

```
Let c = Ref 0 In
   c := (Fun x -> If x = 0 Then 0 Else 1 + !c(x-1));
!c(10)
```

单元 c 包含一个函数，该函数反过来又引用 c。换句话说，该函数具有对自己的引用，并可以应用自己，从而产生递归。这种递归形式被称为 **tying the knot**，并且是大多数编译器实现递归的方法。类似的技巧用于使对象具有自我意识，尽管 C++ 明确传递 self，更像是 Y-组合子。

Exercise 4.3. 绑定结点可以用OCaml编写，但不能直接像上面那样写。为了使其工作，必须如何声明引用？为什么我们可以创建这种循环存储，但不能创建第4.2节中描述的那种？

Interact with F♭SR. 让我们在 **F♭SR** 中编写一个递归乘法函数，首先使用 Let Rec，然后通过打结来实现。

```
# Let Rec mult x = Fun y ->
    If x = 0 Then
      0
    Else
      y + mult (x - 1) y In
  mult 8 9;;
==> 72
```

现在我们将使用绑定操作。因为 **F♭SR** 不包含排序操作，我们使用第4.1.1节中提出的编码。

```
# Let mult = Ref 0 In
  (Fun dummy -> (!mult) 9 8)
    (mult := (Fun x -> Fun y ->
       If x = 0 Then
         0
       Else
         y + (!mult) (x - 1) y));;
==> 72
```

4.1.3 The "Normal" Kind of State

C++、Java 和 Scheme 等语言在表达变异时有不同的形式。获取单元格值时不需要显式解引用(!)运算符。这种变异形式在解释器中更难捕捉，因为变量根据它们是在等式的左边还是右边而具有不同的含义。

赋值运算符。一个 **l-value** 出现在赋值号的左侧，表示一个内存位置。一个 **r-value** 出现在赋值号的右侧，或在代码的其他地方，表示一个实际值。

考虑C/C++/Java赋值语句`x = x + 1`。赋值语句左侧的`x`是一个左值，而`x`在`x + 1`中是一个右值。在OCaml中，我们会写出类似的表达式`x := !x + 1`。因此，OCaml在指代单元格或值方面是明确的。对于Java的左值，需要单元格来执行存储操作，而对于Java的右值，则需要单元格的*value*，这就是为什么在OCaml中我们必须写出`!x`。

l-values 和 r-values 是不同的。一些表达式可以是 l-values 和 r-values，例如 `x` 和 `x[3]`。其他值可能只能是 r-values，例如 `5`、`(0 == 1)` 和 `sin(3.14159)`。因此，在语言语法中，l-values 和 r-values 的表达方式不同，具体来说，l-values 是 r-values 的子集。这种语言存在一些不足。例如，在 OCaml 中我们可以说 `f(3) := 1`，将值 `1` 赋给函数 `f` 返回的单元。这种类型的表达式在 Java 中是无效的，因为 `f(3)` 不是一个 l-value。我们可以修改 **F♭S** 语法，以使用这种更严格的概念，即 l-values 必须只能是变量，如下所示:

```
type expr =
  ...
| Get of expr | Set of ident * expr | Ref of expr
```

对于赋值左侧的变量，我们需要的是变量的地址，而不是其内容。

C和C++中关于标准状态概念的一个最终问题是未初始化变量的问题。因为变量不需要初始化，所以可能存在由于这些未初始化变量导致的运行时错误。请注意，**F♭S**和OCaml的`Ref`语法要求变量在声明时必须显式初始化。

4.1.4 Automatic Garbage Collection

内存分配后，也可能在某个时刻被释放。在C和C++中，这分别通过`free()`和`delete()`显式完成。然而，Java、OCaml、Smalltalk、Scheme和Lisp等语言，以几个为例，支持通过**garbage collector**自动释放未使用的内存。

垃圾回收是一个广泛的研究领域（参见[23]以获取概述），但我们将简要描述一个实现的样子。

首先，某些因素触发了垃圾回收器。通常，触发因素是存储空间变得过于满载。在Java中，可以通过方法 `System.gc()` 显式调用垃圾回收器。当垃圾回收器被调用时，评估将暂停，直到垃圾回收器完成。垃圾回收器将遍历当前计算以查找直接使用的单元。这些单元的集合被称为 *root set*。寻找根的好地方是在评估堆栈和全局变量位置。

一旦建立根集合，垃圾收集器将不在根集合中的所有单元标记为初始 *free*。然后，它从根开始递归遍历内存图

集合，并将从根集合可到达的单元格标记为 *not free*，因此遍历结束时，所有与根集合中 *any* 的单元格属于不同连通分量的单元格都被标记为 *free*，并且这些单元格可以被重用。重用内存的方法有很多，但这里我们不会详细介绍。

4.2 Environment-Based Interpreters

现在我们已经讨论了有状态的语言，我们将简要地触及一下我们定义解释器的方式中的一些效率问题。我们将专注于消除函数应用中的显式替换。

一个“低级”解释器永远不会为每个变量使用重复函数参数。这样做的问题在于，它可能会极大地增加解释器必须维护的数据大小。考虑以下形式的表达式。

`(Fun x -> x x x) (`庞大的表达式`{v*}`

这个表达式通过计算评估

(庞大的表达式)(庞大的表达式)(庞大的表达式),

数据规模的三倍。

为了解决这个问题，我们定义了一个 **explicit environment interpreter**。在一个显式环境解释器中，我们不是直接进行变量替换，而是跟踪每个变量的值在一个 **runtime environment** 中。运行时环境只是变量到值的映射。我们用 $\{x_1 \mapsto v_1, x_2 \mapsto v_2\}$ 来表示环境，意味着变量 x_1 映射到值 v_1，变量 x_2 映射到值 v_2。

现在，为了计算

`(Fun x -> x x x)(`庞大的表达式`{v*}`

解释器在环境 {x $\mapsto$ 中计算 `(x x x)`，惊人的表达式 expr}。

技术上，即使是我们在前面看到的简单替换解释器也不会真正复制三份数据。相反，它们维护一个数据副本和三个指向它的指针。这是因为不可变数据始终可以通过引用传递，因为它永远不会改变。然而，在编译器中，代码不能像这样复制，因此需要像上面展示的那种方案。

存在使用此方法出现某些异常的可能性。具体来说，当另一个函数应用的结果是一个函数，并且该返回的函数包含局部变量时，会出现问题。考虑以下示例。

$f =$
```
Fun x -> If x = 0 Then
   Fun y -> y
Else Fun y -> x + y
```

当计算 f (3) 时，环境在计算体时将 x 绑定到 3，因此返回的结果是

Fun y -> x + y,

但是简单地返回这个值是错误的，因为我们就会失去 x 被绑定到 3 的这个事实。解决方案是，当返回一个函数值时，返回该函数的 *closure*。

Definition 4.2 (闭包)**.** *A **closure** is a function along with an environment, such that all free values in the function body are bound to values in the environment.*

对于上述情况，正确的返回值是闭包

(Fun y -> x + y, $\{\texttt{x} \mapsto 3\}$)

Theorem 4.1. *A substitution-based interpreter and an explicit environment interpreter for **F♭** are equivalent: all **F♭** programs either terminate on both interpreters or compute forever on both interpreters.*

此函数值的封闭视图在编写编译器时至关重要，因为编译器不应在运行时进行代码替换。编译器在第8章中讨论。

4.3 The F♭SR Language

我们现在可以定义 **F♭SR** 语言。**F♭SR** 是一种按值调用的语言，它包括了 **F♭** 的基本特性，并扩展以包括记录 (**F♭R**) 和状态 (**F♭S**)。

在4.4节和第5章、第6章中，我们将研究**F♭SR**中缺失的语言特性，即对象、异常和类型。在第8章中，我们将讨论**F♭SR**的翻译，但不会包括这些其他语言特性。**F♭SR**的抽象语法类型如下。

```
type label = Lab of string

type ident = Ident of string

type expr =
    Var of ident | Function of ident * expr | Appl of expr * expr |
    LetRec of ident * ident * expr * expr | Plus of expr * expr |
    Minus of expr * expr | Equal of expr * expr |
    And of expr * expr | Or of expr * expr | Not of expr |
    If of expr * expr * expr | Int of int | Bool of bool |
    Ref of expr | Set of expr * expr | Get of expr | Cell of int |
    Record of (label * expr) list | Select of  label * expr |
    Let of ident * expr * expr
```

```
type fbtype =
    TInt | TBool | TArrow of fbtype * fbtype | TVar of string |
    TRecord of (label * fbtype) list | TCell of fbtype;;
```

在接下来的两节中，我们将探讨一些非平凡的“真实世界”**F♭SR**程序，以展示从这个简单语言中可以获得的强大功能。我们首先考虑一个计算阶乘函数的函数，并以考察归并排序算法的实现来结束本章。

4.3.1 Multiplication and Factorial

阶乘函数使用 **F♭SR** 表示相当简单。这里的主要关注点是使用柯里化 `Let Rec` 定义进行乘法编码。此示例假设输入为正整数。

```
(*
 * First we encode multiplication for positive nonnegative
 * integers.  Notice the currying in the Let Rec construct.
 * Multiplication is encoded in the obvious way: repeated
 * addition.
 *)
Let Rec mult x = Fun y ->
  If y = 0 Then
    0
  Else
    x + (mult x (y - 1)) In

(*
 * Now that we have multiplication, factorial is easy to
 * define.
 *)
Let Rec fact x =
  If x = 0 Then
    1
  Else
    mult x (fact (x - 1)) In

fact 7
```

4.3.2 Merge Sort

编写一个在 **F♭SR** 中的归并排序是一个相当全面的例子。其中最大的挑战之一是编码整数比较，即 $<$、$>$ 等。在查看代码之前，让我们讨论一下这是如何实现的。

首先，鉴于我们已经有了一个等式测试 =，编码 <= 操作基本上免费为我们提供了其他标准比较操作。假设我们有一个 lesseq 函数，其他操作可以简单地编码如下。

```
Let lesseq = (* real definition *) In

Let lessthan = (Fun x -> Fun y ->
  (lesseq x y) And Not (x = y)) In

Let greaterthan = (Fun x -> Fun y ->
  Not (lesseq x y)) In

Let greatereq = (Fun x -> Fun y ->
  (greaterthan x y) Or (x = y)) In ...
```

因此，只需对 lesseq 进行编码。但如何仅使用常规 **F♭SR** 操作符来完成此操作呢？基本思路如下。为了测试 x 是否小于或等于 y，我们计算一个 z，使得 $x + z = y$。如果 z 非负，我们知道 x 小于或等于 y。如果 z 为负，我们知道 x 大于 y。

首先，我们似乎遇到了一个“鸡生蛋还是蛋生鸡”的问题。如果没有比较运算符，我们怎么判断 z 是否为负？实际上，我们如何开始计算 z 呢？一个想法是从 $z = 0$ 开始，循环，每一步增加 z 1，并测试 $x + z = y$。如果我们找到一个 z，我们就停止。我们知道 z 是正的，我们得出结论 x 小于或等于 y。如果我们找不到 z，我们从 $z = -1$ 开始，执行类似的循环，每一步减少 z 1。如果我们以这种方式找到一个合适的 z 值，我们得出结论 x 大于 y。

上述方案中的缺陷应该是明显的：如果 $x > y$，第一个循环将永远不会终止。事实上，为了让这个想法工作，我们需要并行运行两个循环！显然 **F♭SR** 不允许并行计算，但存在一个沿着这些线的解决方案。我们可以交错两个循环测试的 z 的值。也就是说，我们可以尝试 $\{0, -1, 1, -2, 2, -3, \ldots\}$。

关于这种方法的好处是，z的每个其他值都是负数，我们可以简单地传递一个布尔值来表示z的符号。如果z是非负的，下一次循环迭代会将其反转并减去1。如果z是负数，下一次循环迭代只需将其反转。注意，我们可以通过简单地写出0 - x来反转**F♭SR**中数字x的符号。现在，带着我们对lesseq的想法，让我们开始编写归并排序。

```
(* We need to represent the empty list somehow.  Since we
 * will need to test for it, and since equality is only
 * defined on integers, emptylist will need to be an
 * integer.  We define it as -1.
 *)
Let emptylist = (0 - 1) In
```

```
(* Next, let's define some list operations, head, tail,
 * cons, and length.  There encodings are straightforward,
 * and were covered in the text.  Notice that we are
 * encoding lists as records, using {l,r} records like
 * pairs.
 *)
Let head = Fun seq -> seq.l In

Let tail = Fun seq -> seq.r In

Let cons = Fun elt -> Fun seq -> {l=elt; r=seq} In

Let Rec length seq =
   If seq = emptylist Then
     0
   Else
     1 + length (seq.r) In


(* Now, we're ready to define lesseq.  Notice how lesseq is
 * a wrapper function that uses le.  le passes along the
 * sign of v as an argument, as discussed in the text.
 *)
Let lesseq = Fun a -> Fun b ->
  Let Rec le x =
    Fun y -> Fun v ->
      Fun v_is_non_neg ->
        If (x + v) = y Then
          v_is_non_neg
        Else
          If v_is_non_neg Then
            le x y (0 - v - 1) (Not v_is_non_neg)
          Else
            le x y (0 - v) (Not v_is_non_neg) In
  le a b 0 True In

(* This function takes a list and splits it into nearly
 * equal halves.  These halves are returned as a pair
 * (encoded as an {left, right} record).
 *)
Let split = Fun seq ->
  Let Rec splt seq1 = Fun seq2 ->
    If lesseq (length seq1) (length seq2) Then
      {left=seq1; right=seq2}
    Else
      splt (tail seq1) (cons (head seq1) seq2) In
```

```
  splt seq emptylist In

(* Here is where we merge two sorted lists.  We scan through
 * each list in parallel, chopping off the smaller of the
 * two list heads and appending it to the result.  This is
 * where we make use of our lesseq function.
 *)
Let Rec merge seq1 = Fun seq2 ->
  If seq1 = emptylist Then
    seq2
  Else If seq2 = emptylist Then
    seq1
  Else
    If lesseq (head seq1) (head seq2) Then
      cons (head seq1) (merge (tail seq1) seq2)
    Else
      cons (head seq2) (merge seq1 (tail seq2)) In

(* mergesort sorts a single list by breaking it into two
 * smaller lists, recursively sorting those lists, and
 * merging the two back into a single list.  Recall that
 * the base cases of the recursion are a single element list
 * and an empty list, both of which are necessarily in
 * sorted order by definition.
 *)
Let Rec mergesort seq =
  If lesseq (length seq) 1 Then
    seq
  Else
    Let halves = split seq In
    merge (mergesort (halves.left))
          (mergesort (halves.right)) In

(* Finally we call mergesort on an actual list.  Notice the
 * record encoding.
 *)
mergesort {l=5;  r=
          {l=6;  r=
          {l=2;  r=
          {l=1;  r=
          {l=4;  r=
          {l=7;  r=
          {l=8;  r=
          {l=10; r=
          {l=9;  r=
          {l=3;  r=emptylist}}}}}}}}}}
```

4.4 Exceptions and Other Control Operations

直到现在，表达式都是通过逐个遍历评估规则来评估的。为了执行加法，先评估左右操作数得到值，然后评估加法表达式本身得到一个值。这个加法表达式可能是一个更大表达式的组成部分，然后这个更大表达式本身也可能被评估为一个值，等等。

在这个部分，我们将讨论显式 **control operations**，重点关注异常。显式控制操作是明确改变评估过程控制的操作。即使是简单的语言也有控制操作。一个常见的例子是 C++ 和 Java 中的 `return` 语句。

在 **F♭** 中，函数的值等于其整个主体评估的结果。如果在某个复杂的条件循环表达式的中间，我们得到了计算的最终结果，仍然需要完成函数的执行。一个返回语句通过立即从函数返回并 *aborting* 其余的函数计算来解决这个问题。

另一个常见的控制操作是循环退出操作，或C++或Java中的`break`。`break`与`return`类似，不同之处在于它退出的是当前循环而不是整个函数。

这些类型的控制操作很有趣，但在这个部分，我们将更专注于两种更强大的控制操作，即**exceptions**和**exception handlers**。读者应该已经熟悉了从使用OCaml异常机制中了解到的异常基础知识。

存在一些其他控制操作，我们不会讨论，例如 call/cc、shift/reset 和 control/prompt 操作符。

我们将在本节中避免使用 `goto` 操作符，除了以下简要讨论。`goto`基本上会跳转到函数中的一个标记位置。`goto`的主要问题是它过于原始。在函数的上下文中，在标签之间跳转的模式并不那么有用。它本身也具有固有的危险性，因为可能会意外地跳入甚至未执行的函数的中间部分，或者跳过变量初始化。此外，如果其他控制操作符足够丰富，`goto`实际上并没有提供更多的表现力，至少不是有意义的表达力。

事实是，控制操作符根本不需要。回想一下 **F♭**，以及相关的λ演算，它们已经图灵完备了。因此，控制操作符只是使编程更方便的便利工具。将控制操作符视为“元操作符”是有用的，即作用于评估过程本身的操作符。

4.4.1 Interpreting Return

如何解释异常？在我们回答这个问题之前，我们将考虑向 **F♭** 添加一个 `Return` 操作符，因为它是一个比异常更简单的控制操作符。

问题在于 Return 以及其他控制运算符，它们不适合正常的评估方案。例如，考虑以下表达式

```
(Fun x ->
   (If x = 0 Then 5 Else Return (4 + x)) - 8) 4
```

由于在应用函数时 x 不会成为 0，因此 Return 语句将被评估，并且应立即停止执行，不评估“- 8。”问题在于评估上述语句意味着评估

```
(If 4 = 0 Then 5 Else Return (4 + 4)) - 8,
```

这反过来意味着评估

```
(Return (4 + 4)) - 8.
```

但我们知道减法规则是通过评估这个表达式的左右两边来得到值，并对它们进行整数减法。显然，这种情况不适用，因此我们需要为其中一个子表达式中包含 Return 的减法制定特殊规则。

首先，我们需要将 Return 添加到 **F**♭ 的值域中，并提供一个适当的评估规则:

$$\textit{(Return)} \qquad \frac{e \Rightarrow v}{\texttt{Return}\ e \Rightarrow \texttt{Return}\ v}$$

接下来，我们需要特殊的减法规则，一个用于当 Return 在左侧时，另一个用于当 Return 在右侧时。

$$\textit{(- Return Left)} \qquad \frac{e \Rightarrow \texttt{Return}\ v}{e\ \texttt{-}\ e' \Rightarrow \texttt{Return}\ v}$$

$$\textit{(- Return Right)} \qquad \frac{e \Rightarrow v, \quad e' \Rightarrow \texttt{Return}\ v'}{e\ \texttt{-}\ e' \Rightarrow \texttt{Return}\ v'} v \text{ 不是形式 } \texttt{Return}\ v''$$

注意，这些减法规则计算结果为 Return v，而不是简单地 v。这意味着 Return 通过减法运算符“冒泡”上来。我们需要为 *every* **F**♭ 运算符定义类似的返回规则。使用这些新规则，可以清楚地看出

```
Return (4 + 4) - 8 ⇒ Return 8.
```

当然，我们不希望 Return 无限上浮。当 Return 从函数应用中弹出时，我们只得到与之关联的值。换句话说，我们的原始表达式，

```
(Fun x ->
   (If x = 0 Then 5 Else Return (4 + x)) - 8) 4
```

应评估为 8，而不是 Return 8。为了实现这一点，我们需要一个特殊的函数应用规则。

(Appl. Return) $$\frac{e_1 \Rightarrow \texttt{Fun}\ x\ \texttt{->}\ e, \quad e_2 \Rightarrow v_2, \quad e[v_2/x] \Rightarrow \texttt{Return}\ v}{e_1\ e_2 \Rightarrow v}$$

一些其他特殊应用规则适用于函数或自变量本身是 Return 的情况。

(Appl. Return Fun) $$\frac{e_1 \Rightarrow \texttt{Return}\ v}{e_1\ e_2 \Rightarrow \texttt{Return}\ v}$$

(Appl. Return Arg.) $$\frac{e_1 \Rightarrow v_1,\ e_2 \Rightarrow \texttt{Return}\ v}{e_1\ e_2 \Rightarrow \texttt{Return}\ v}$$

当然，我们仍然需要原始函数应用规则（见第2.3.3节）来处理函数执行通过执行结束处的隐式返回值的情况。

让我们通过考虑 Return Return e 的影响来结束对 Return 的讨论。这样的表达式有两种可能的解释。根据上述规则，此表达式从两级函数调用中返回。另一种解释可能是添加以下规则：

(Double Return) $$\frac{e \Rightarrow \texttt{Return}\ v}{\texttt{Return}\ e \Rightarrow \texttt{Return}\ v}$$

当然，我们需要将原始的 Return 规则限制在 v 不以 Return v 形式出现的情况下。根据这个规则，我们不再从两级函数调用中返回，而是第二级 Return 实际上中断并通过第一级。当然，双重返回不是一个常见的结构，这些规则在实践中的使用并不频繁。

4.4.2 The F♭X Language

Return的上文翻译可以轻松扩展以处理一般异常。我们将定义一种语言**F♭X**，它是通过OCaml风格的异常机制扩展的**F♭**。**F♭X**没有Return (，OCaml也没有)，但Return可以很容易地用异常编码。例如，“伪OCaml”表达式

```
(function x -> (if x = 0 then 5 else return (4 + x)) - 8) 4
```

可以按照以下方式编码。

```
exception Return of int;;

(function x ->
  try
    (if x = 0 then 5 else raise (Return (4 + x))) - 8
  with
    Return n -> n) 4;;
```

Return 可以以类似的方式编码在其他函数中。

现在，让我们定义我们的 **F♭X** 语言。基本思想与上面讨论的 Return 语义非常相似。我们定义一种新的值类型，

Raise #xn v,

哪个冒泡出异常 #xn。这是从上面推广的值类 Return v。#xn 是一个代表异常名称的元变量。异常包含两份数据：一个名称和一个参数。参数可以是一个任意表达式。尽管我们只允许单参数异常，但可以通过，例如，向异常提供一个具有零、一个或多个字段的记录参数，轻松地编码零值或多值版本的异常。我们还向 **F♭X** 添加了表达式

Try e With #xn x -> e'

注意 Try 绑定 e' 中 x 的自由出现。另外，请注意 **F♭X** Try 语法与 OCaml 语法略有不同，因为 OCaml 允许在 With 子句中使用任意模式匹配表达式。我们只允许一个匹配 #x 所有值的子句。

F♭X 未类型化，因此不需要声明异常。我们使用“#”符号来指定具体语法中的异常，例如，#MyExn。下面是一个**F♭X**表达式的示例。

```
Fun x -> Try
   (If x = 0 Then 5 Else Raise #Return (4 + x)) - 8
With #Return n -> n) 4
```

异常是副作用，可能会引起“远程作用”。因此，就像任何其他副作用一样，它们应该谨慎使用。

4.4.3 Implementing the F♭X Interpreter

抽象语法类型对于 **F♭X** 如下所示。

```
type exnlab = string
type expr =
  ...
| Raise of exnlab * expr
| Try of expr * exnlab * ident * expr
```

规则来自返回规则和应用返回规则。抛出“冒泡”异常，就像 Return 自己冒泡一样。Try 是冒泡停止的点，就像函数应用是停止冒泡的 Return 点。异常的操作语义如下。

$$\textit{(Raise)} \quad \frac{e \Rightarrow v, \text{ for } v \text{ not of the form } \texttt{Raise } \ldots}{\texttt{Raise \#}xn\ e \Rightarrow \texttt{Raise \#}xn\ v}$$

$$\textit{(Try)} \quad \frac{e \Rightarrow v \text{ for } v \text{ not of the form } \texttt{Raise \#}xn\ v}{\texttt{Try } e \texttt{ With \#}xn\ x \texttt{ -> } e' \Rightarrow v}$$

$$\textit{(Try Catch)} \quad \frac{e \Rightarrow \texttt{Raise \#}xn\ v, \quad e'[v/x] \Rightarrow v'}{\texttt{Try } e \texttt{ With \#}xn\ x \texttt{ -> } e' \Rightarrow v'}$$

此外，我们还需要添加所有其他 **F♭** 规则的特殊版本，以便 Raise 能够像 Return 一样在计算中向上传递。例如

$$\textit{(- Raise Left)} \quad \frac{e \Rightarrow \texttt{Raise \#}xn\ v}{e \texttt{ - } e' \Rightarrow \texttt{Raise \#}xn\ v}$$

请注意，我们必须处理一个不寻常的情况，即当 Raise 通过 Raise 冒泡时，这种情况在实践中很少发生。规则与上面的“- 提升左”规则非常相似。

$$\textit{(Raise Raise)} \quad \frac{e \Rightarrow \texttt{Raise \#}xn\ v}{\texttt{Raise \#}xn'\ e \Rightarrow \texttt{Raise \#}xn\ v}$$

现在，让我们追踪执行过程，其中 {v*} 保持不变。

```
Fun x -> Try
      (If x = 0 Then 5 Else Raise #Return (4 + x)) - 8
   With #Return n -> n) 4
```

在函数应用和对 If 语句的评估之后，我们剩下

```
Try (Raise #Return (4 + 4)) - 8
   With #Return n -> n
```

这是

```
Try (Raise #Return 8) - 8
   With #Return n -> n
```

通过在减法中冒泡计算得到

```
Try Raise #Return 8
   With #Return n -> n
```

根据 Try Catch 规则返回的值 8，如预期所示。

Chapter 5

Object-Oriented Language Features

面向对象编程已成为行业标准。然而，面向对象特性并没有从根本上为语言增加很多。它们肯定不会导致程序变得更短。面向对象编程的成功主要归因于它是一种适合人类心理学的风格。人类作为其基本功能的一部分，在识别和与日常对象互动方面非常擅长。而且，编程中的对象与日常世界中的对象足够相似，我们的丰富直觉可以应用于它们。

在讨论面向对象编程中对象的属性之前，让我们简要回顾一下 *everyday* 对象的一些更重要属性，这些属性会使它们对编程很有用。

- 日常物品是 *active*，也就是说，它们不完全受我们控制。物体具有内部和不断演变的 *state*。例如，一辆正在行驶的汽车是一个活动物体，它具有复杂的内部状态，包括剩余汽油量、发动机冷却液和变速器油液位、电池电量、油位、温度、发动机部件的磨损程度等。

- 日常物品是 *communicative*。我们可以向它们发送消息，并且我们可以通过发送消息从物体那里获得反馈。例如，启动汽车和检查油表都可以被视为向汽车发送消息.[1]

- 日常物品是 *encapsulated*。它们具有我们看不见的内部属性，尽管我们可以通过发送消息来了解其中的一些。以汽车为例，检查油表可以告诉我们剩余多少油，即使我们无法直接看到油箱，也不知道里面是普通汽油还是高级汽油。

[1] It may, at first, seem unnatural to view checking the gas gauge as sending a message. After all, it is really the car sending a message to us. We view ourselves as the "sender," however, because we *initiated* the check, that is, in a sense, we *asked* the car how much gas was remaining.

- 日常物品可能是 {v*}，也就是说，物品可能由几个更小的物体组件组成，而这些组件本身也可能有更小的组件。再次强调，汽车是一个完美的嵌套物体例子，由包括发动机和变速箱在内的更小物体组成。变速箱本身也是一个嵌套物体，包括输入轴、输出轴、扭矩转换器和一套齿轮。

- 日常物品大部分都有独特的名称。汽车通过车牌或车辆登记号码来唯一命名。

- 日常物品可能是 *self-aware*，在意义上，它们可能有意与自己互动，例如狗舔自己的爪子。

- 日常物体交互可能是 *polymorphic*，因为一组不同的物体可能共享一个通用的消息协议，例如所有车型和卡车的加速器/制动/转向轮消息系统。

面向对象编程中的对象也具有这些属性。因此，面向对象编程对大多数人来说感觉自然且熟悉。现在让我们考虑编程中的对象。

- 对象具有以实例变量或字段形式存在的内部状态。对象通常是 *active*，并且它们的状态不完全受调用者的控制。

- 对象支持消息，这些消息是有名称的代码片段，与特定对象相关联。通过这种方式，对象是 *communicative*，而对象组通过相互发送消息来完成任务。

- 对象由封装的代码（方法或操作）以及可变状态（实例变量或字段）组成。立即可以清楚地看出，对象本质上是非函数性的。这反映了日常对象的状态性。

- 对象通常以 *nested* 的方式组织。例如，考虑一个表示图形网页浏览器的对象。该对象本身是框架，但该框架由工具栏和查看区域组成。工具栏由按钮、地址栏和菜单组成，而查看区域由渲染面板和滚动条组成。这一概念在图5.1中得到了说明。对象嵌套通常是通过外部对象将其内部对象作为字段或实例变量存储来实现的。

- 对象具有唯一的名称，或 *object references*，以引用它们。这与现实世界中对象的命名类似，优点是对象引用始终唯一，而现实世界中的对象名称可能存在歧义。

- 对象是 *self aware*，即对象包含对自己引用的引用。`this` 在 Java 中和 `self` 在 Smalltalk 中是自引用。

- 对象是 *polymorphic*，这意味着一个“更胖”的对象总是可以被传递给一个接受“更瘦”对象的方法，也就是说，一个具有更少方法和公共字段的对象。

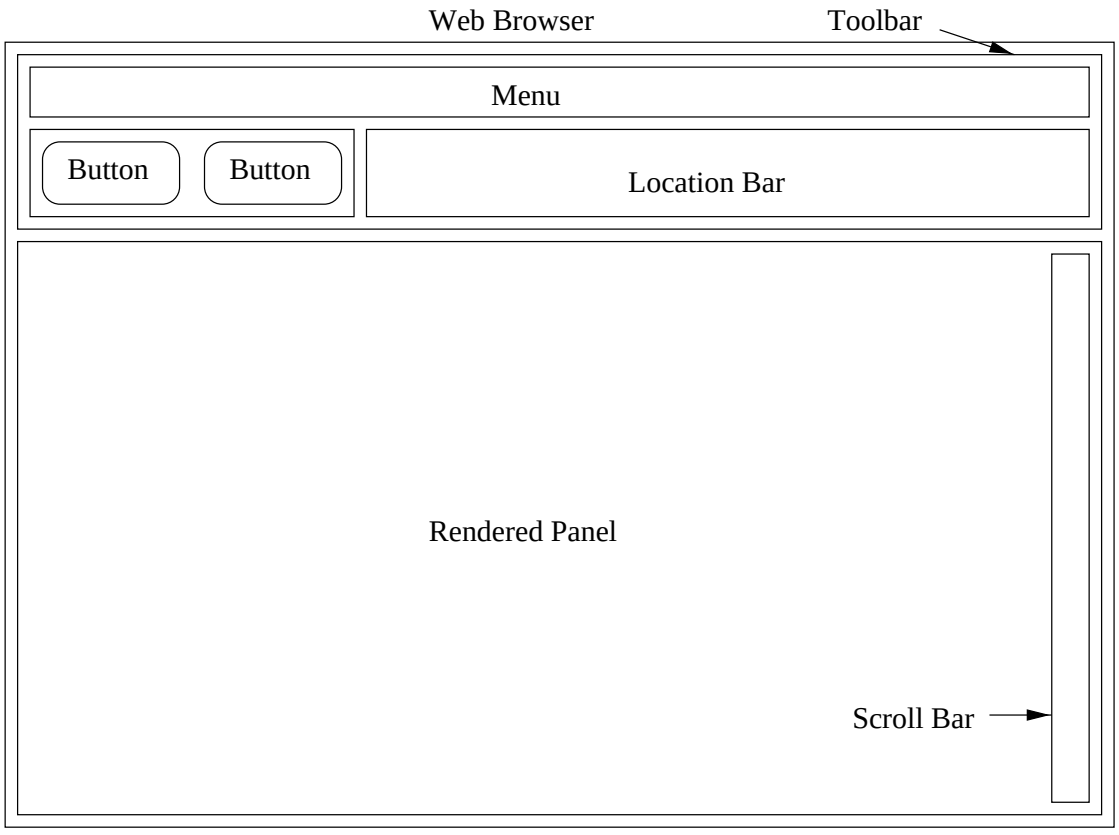


图5.1：表示网页浏览器的嵌套对象。

存在几个对象通常具有的附加功能。*Classes* 在具有对象的编程语言中几乎总是存在。类不是必需的：没有类也有对象是完全有效的。语言Self [19, 20, 2] 没有类，而是有*prototype*对象，这些对象被复制以执行类的工作。JavaScript 从Self中借鉴了这个想法，因此现在许多程序员正在使用基于原型的对象。类的重要概念包括创建、继承、方法重写、超类访问和动态调度。我们将在后面讨论这些概念。

字段和方法的信息隐藏是大多数面向对象语言具有的另一个特性，通常以 `public`、`private` 和 `protected` 关键字的形式存在。

其他对象的方面包括对象类型和模块。类型被简要讨论，但模块超出了本书的范围。我们还忽略了方法 *over- loading*，因为它只是语法糖，只增加了可读性，并没有增加功能。回想一下，方法重载意味着有两个具有相同名称但不同类型签名的两个方法。重写意味着在子类中重新定义超类方法的操作。重写将在第5.1.5节中讨论。

5.1 Encoding Objects in F♭SR

一个对象最重要的方面之一是它们与我们已讨论过的现有概念的接近程度。在本节中，我们展示了如何轻松地将对象编码为 **F♭SR**。我们将对象建模为函数和引用的记录。记录标签和值可以被视为方法或字段的槽位。带有函数的槽位是方法，带有引用的槽位是字段。

5.1.1 Simple Objects

考虑图5.2[2]中的对象，它代表二维空间中的一个点。该对象具有字段`x`和`y`，以及两个方法：`magnitude`和`iszero`，具有明显的功能。要将此对象编码为记录，我们需要一个具有以下结构的记录。

```
Let point = {
   x = 4;
   y = 3;
   magnitude = Fun _ -> ...;
   iszero = Fun _ -> ...
} In ...
```

我们尚不能编写 `magnitude` 方法，因为我们需要用 `x` 和 `y` 来定义它，但在函数体中无法引用它们。我们将使用的解决方案与 C++ 在幕后使用的相同：我们将对象本身作为参数传递给函数。让我们将我们的第一个编码修订为以下版本（假设我们已经定义了一个 `sqrt` 函数）。

[2]We use UML to diagram objects. UML is fully described in [11]. Although the diagram in Figure 5.2 is technically a class diagram, for our purposes we will view it simply as the induced object.

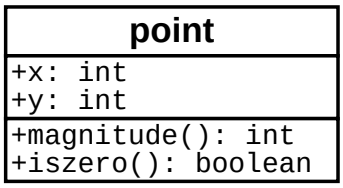


图5.2: “点”对象。

```
Let point = {
   x = 4;
   y = 3;
   magnitude = Fun this -> Fun _ ->
      sqrt(sqr this.x + sqr this.y);
   iszero = Fun this -> Fun _ ->
      ((this.magnitude) this {}) = 0
} In ...
```

关于上述示例有一些有趣之处。首先，当我们发送消息时，对象本身需要被明确地作为参数传递。例如，要向 point 发送大小消息，我们会写成

```
point.magnitude point {}
```

为了方便，我们可以使用缩写 `obj <- method` 来表示消息 (`obj.method obj`)。

甚至在对对象的函数进行调用时，我们仍然需要将 this 传递给另一个方法。iszero 通过调用 magnitude 来说明这一点。这种自引用的编码称为 **self-application encoding**。还有许多其他的编码，我们将在下面讨论一些。

存在此编码的问题，尽管如此；我们的对象仍然是不可变的。具有不可变字段的对象在很多情况下都是好事。例如，无法调整大小的窗口和具有固定成员的集合可以使用不可变字段实现。然而，通常我们需要我们的对象支持可变字段。幸运的是，这个问题有一个简单的解决方案。考虑以下我们point编码的修订版，它使用Ref来表示字段。

```
Let point = {
   x = Ref 4;
   y = Ref 3;
   magnitude = Fun this -> Fun _ ->
      sqrt(sqr !(this.x) + sqr !(this.y));
   iszero = Fun this -> Fun _ ->
      (this.magnitude this {}) = 0;
   setx = Fun this -> Fun newx -> this.x := newx;
   sety = Fun this -> Fun newy -> this.y := newy
} In ...
```

为了将 x 设置为 12，我们可以写 `point <- setx 12`，或者我们可以直接使用 `(point.x) := 12` 修改字段。要访问字段，我们现在写 `!(point.x)`，或者可以定义一个 getx 方法来抽象解引用。这种策略为我们提供了简单对象的忠实编码。在接下来的几节中，我们将讨论如何编码对象的一些更高级功能。

5.1.2 Object Polymorphism

假设我们定义以下函数。

```
Let tallerThan = Fun person1 -> Fun person2 ->
   greatereq (person1 <- height) (person2 <- height)
```

如果我们定义支持 height 消息的 person 对象，我们可以将它们作为参数传递给此函数。然而，我们也可以创建专门的 person 对象，例如 mother、father 和 child。只要这些对象仍然支持 height 消息，它们都是 tallerThan 函数的有效候选者。甚至像 dinosaur 和 building 这样的对象也可以作为参数。唯一的要求是传递给 tallerThan 的任何对象都支持消息 height。这被称为 **object polymorphism**。

我们已经遇到了一个类似的概念，当我们讨论记录多态性时。回想一下，一个函数

```
Let getheight = Fun r -> r.height ...
```

可以接受任何具有 height 字段的记录：

```
... In getheight {radius = 4; height = 4; weight = 44}
```

对象多态实际上与记录多态是相同的，这可以从我们将对象视为记录的方式来证明。因此，通过使用对象的记录编码，我们可以轻松地获得对象多态。

```
Let eqpoint = {
   (* 所有来自 point 之上的代码: x, y, magnitude ... *)
equal = Fun this -> Fun apoint ->
!(this.x) = !(apoint.x) And !(this.y) = !(apoint.y)
} In eqpoint <- equal({ ... *)
   x = Ref 3;   })
   y = Ref 7;
   (*
```

对象传递给 equal 只需定义 x 和 y。在我们的嵌入式语言中，对象多态性比 C++ 或 Java 更强大：C++ 和 Java 在决定允许传递给方法的内容时，会查看参数的 *type*。子类总是可以被传递给接受超类的方法，但除此之外不允许其他操作。在我们的编码中，这更接近 Smalltalk，只要对象支持函数需要的消息，就可以将其传递给函数或方法。

一个对象多态可能遇到的潜在困难是 **dispatch**：由于我们不知道对象的形式直到运行时，我们不知道正确的方法将确切地在哪里在内存中布局。因此，在编译面向对象的编程语言中查找方法可能需要哈希。这正是我们在 **F♭SR** 编译器中编写记录处理代码时出现的问题（见第8章），再次说明了对象和记录之间的相似性。

5.1.3 Information Hiding

大多数面向对象的语言允许使用 **information hiding** 来保护方法和实例变量免受直接外部使用。信息隐藏是封装对象数据的关键工具之一。在 C++ 和 Java 中，使用 public 、private 和 protected 修饰符来控制字段和方法的信息隐藏程度。

目前我们只需将隐藏编码为 **F♭SR**。请注意，只有公共和私有数据在 **F♭SR**（ 中才有意义，受保护的数据只有在类和继承的上下文中才有意义，我们尚未定义这些内容)。在我们的编码中，我们通过简单地使数据不可访问来实现隐藏。在现实中的类型化语言中，是类型系统本身（以及在 Java 的情况下字节码验证器）强制执行数据的隐私。

让我们从信息隐藏的部分编码开始。

```
Let pointImpl = (* point 从之前 *) In
Let pointInterface = {
setx = pointImpl.setx pointImpl;   magnitude = pointImpl.magnitude pointImpl;
sety = pointImpl.sety pointImpl } In ...
```

在此编码中，每种方法都“预先应用”到完整点对象 pointImpl 上，而 pointInterface 只包含公共方法和实例。现在方法调用简单如下

```
pointInterface.setx 5
```

这个解决方案有缺陷。返回 this 的方法会重新暴露完整对象的隐藏字段和方法。考虑以下示例。

```
Let pointImpl = {
   (* ... *)
   sneaky = Fun this -> Fun _ -> this
} In Let pointInterface = {
   magnitude = pointImpl.magnitude pointImpl;
   setx = pointImpl.setx pointImpl;
   sety = pointImpl.sety pointImpl;
   sneaky = pointImpl.sneaky pointImpl
} In pointInterface.sneaky {}
```

sneaky 方法返回完整的 pointImpl 对象，而不是 pointInterface 版本。为了解决这个问题，我们需要更改编码，这样我们就不必每次调用方法时都传递 this。相反，我们将从开始就给对象一个指向自身的指针。

```
Let prePoint = Let privateThis = {
   x = Ref 4;
   y = Ref 3
} In Fun this -> {
   magnitude = Fun _ ->
      sqrt !(privateThis.x) + !(privateThis.y);
   setx = Fun newx -> privateThis.x := newx;
   sety = Fun newy -> privateThis.y := newy;
   getThis = Fun _ -> this
} In Let point = prePoint prePoint In ...
```

现在消息发送操作只是

```
point.magnitude {}
```

该方法 getThis 仍然返回 this，但 this 仅包含公共部分。请注意，为了使此编码工作，privateThis 只能作为消息的目标，不能返回。

这个编码的缺点是每次在对象体内使用时，都需要将 this 应用到自身。例如，而不是编写

```
(point.getThis {}).magnitude {}
```

我们反而需要写成

```
((point.getThis {}) (point.getThis {})).magnitude {}
```

这些编码相对简单。类和继承的编码更困难，下面将进行讨论。对象类型也特别困难，在第6章中介绍。

5.1.4 Classes

类是创建对象的模板。本质上，类是一个对象工厂。从类创建的每个对象都必须有其自己的唯一实例变量集（静态字段除外，我们将在后面处理）。

相对容易生成类的简单编码。只需冻结创建对象的代码，解冻以创建新对象。忽略信息隐藏，编码如下。

```
Let pointClass = Fun _ -> {
   x = Ref 4;
   y = Ref 3;
   magnitude = Fun this -> Fun _ ->
      sqrt !(this.x) + !(this.y);
   setx = Fun this -> Fun newx -> this.x := newx
   sety = Fun this -> Fun newy -> this.y := newy
} In ...
```

我们可以定义 `new pointClass` 为 `pointClass {}`。一些创建和使用实例的典型代码可能如下所示。

```
Let point1 = pointClass {} In
Let point2 = pointClass {} In
point1 <- setx 5 ...
```

`point1` 并且 `point2` 将拥有自己的 `x` 和 `y` 值，因为每次 `Ref` 解冻时都会创建新的存储单元。同样的冻结和解冻技巧可以应用于我们对信息隐藏的编码，以实现类中的隐藏。编码类的难点在于编码继承，我们将在下一节中讨论。

一个更有用的类概念不是仅仅冻结一个对象，而是创建一个返回该对象的函数；该函数的参数与类的构造函数值类似。

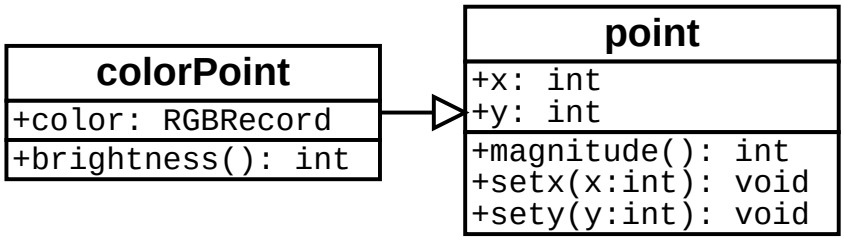


图5.3：point、colorPoint继承层次结构。

```
Let pointClass' = Fun ix -> Fun iy -> {
   x = Ref ix;
   y = Ref iy;
   magnitude = Fun this -> Fun _ ->
      sqrt !(this.x) + !(this.y);
   setx = Fun this -> Fun newx -> this.x := newx
   sety = Fun this -> Fun newy -> this.y := newy
} In ...
```

然后 pointClass' 0 0 创建一个初始化为 $(0, 0)$ 的点对象。

5.1.5 Inheritance

通常，大约80%的物体效用体现在我们上面讨论的概念中，即

- 对象将数据和代码封装在单个名称下以实现特定功能。
- 多态对象。
- 模板用于生成对象（类的主要功能）

其他20%的效用来自**inheritance**。继承允许定义相关对象，这些对象共享一些公共代码。可以通过以下方案在**F♭SR**中编码继承。在子类中，我们创建一个超类对象的实例，并将其作为“奴隶”保留。我们使用奴隶来访问超类对象的方法和字段。因此，子类对象只包含新的和重写的方法。真正的面向对象语言往往不这样实现继承，因为这样做效率低下（想象一下，在一个继承链很长的情况下，方法调用必须一路回溯到层次结构的顶部才能被调用）。尽管如此，这种编码足以说明继承的主要点。例如，考虑以下ColorPoint的编码，它是Point的子类，如图5.3所示。（在这个编码中，我们将假设没有类构造函数参数，即我们将从pointClass而不是pointClass'以上开始工作。）

```
Let pointClass = ... In
Let colorPointClass = Fun _ ->
   Let super = pointClass {} In {
   x = super.x; y = super.y;
   color = Ref {red = 45; green = 20; blue = 20};
   magnitude = Fun this -> Fun _ ->
      mult(super.magnitude this {})(this.brightness this {});
   brightness = Fun this -> Fun _ ->
      (* 计算亮度... *)
setx = super.setx; sety = super.sety
} In ...
```

此编码中有几个有趣点。首先，请注意，为了从超类继承方法和字段，我们明确地将它们链接在一起（即 x、y、setx 和 sety）。要覆盖超类中的方法，我们只需在子类中重新定义它，而不是链接到超类；magnitude 是一个例子。此外，请注意，我们仍然可以从子类中调用超类方法。例如，magnitude 在其主体中调用 super.magnitude。注意 super.magnitude 是用 this 而不是 super 作为其参数传递的。这与动态分发有关，我们现在将讨论这个问题。

5.1.6 Dynamic Dispatch

我们说一个方法是动态分派的，如果在查看一个消息发送 v<-m 时，我们不确定在运行时将执行哪个方法 m。动态分派与对象多态性相关：一个变量 v 可能包含许多不同种类的对象，因此 v<-m 可以向这些不同种类的对象中的任何一个发送 m。除非我们在运行时知道 v 是什么类型的对象，否则我们无法知道将调用哪个方法 m。

术语 **dynamic dispatch** 指的是通过消息发送调用的方法在编译时不是固定的。如果我们在一个 pointClass 中声明了一个方法 isNull 并由 colorPointClass 继承，其代码为

```
Fun this -> Fun _ -> (this.magnitude this {}) = 0,
```

这里的 magnitude 方法不是在编译时固定的。具体来说，假设 pointClass 有这个 isNull，我们执行

```
Let p = pointClass {} In
Let cp = colorPointClass {} In
p <- isNull {}; cp <- isNull {};
```

在 p 的调用中 isNull，this 是一个 point，因此 isNull 将内部调用 point 的 magnitude 方法。另一方面对于 cp，this 是一个 colorPoint，因此 colorPoint’s

magnitude 方法将被调用。总之，从 isNull 内部调用的 magnitude 方法是动态分派的。

上级调度现在也可以解释了。如果 isNull 在 colorPointClass 中被额外覆盖为

```
Fun this -> Fun _ -> (super.isNull this {}) = 0 And (* etc *),
```

然后通过 cp 调用 isNull，它将正确调用 colorPointClass 的幅度，而如果我们改为编写

```
Fun this -> Fun _ -> (super.isNull super {}) = 0 And (* etc *),
```

(注意粗体)的变化，pointClass的大小本应从isNull内部调用，亮度在大小计算中会错误地没有被考虑到一个colorPoint。

Another example 为了对此问题进行进一步说明，让我们再举一个例子。考虑以下类，rectClass及其子类squareClass。

```
Let rectClass = Fun w -> Fun l -> {
  getWidth = Fun this -> Fun _ -> w;
  getLength = Fun this -> Fun _ -> l;
  area = Fun this -> Fun _ ->
    mult ((this.getLength) this {}) ((this.getWidth) this {})
} In

Let squareClass = Fun e ->
  Let super = rectClass 1 e In {

  getLength = (super.getLength);

  (* We override width to be the same as length *)
  getWidth = (super.getLength);

  areaStatic = Fun this -> Fun _ ->
    (super.area) super {};
  areaDynamic = Fun this -> Fun _ ->
    (super.area) this {}
} In
Let mySquare = squareClass 10 In
mySquare <-areaDynamic {} (* returns 100 *);
mySquare <-areaStatic {} (* returns 10 *)
```

注意，在 squareClass 中，getLength 已被覆盖以与 getWidth 表现相同。在 squareClass 中计算面积有两种方法。areaStatic 调用

(super.area) super {}这意味着当调用 rectClass 的 area 方法时，它还以 rectClass 的 getLength 和 getWidth 为参数被调用。结果是 $1\times10 = 10$，因为矩形被初始化为宽度为 1。

相反，动态分派区域方法areaDynamic被写成(super.area) this {}。这次，调用了rectClass的area方法，但this是squareClass的实例，而不是rectClass，并且使用了squareClass重写的getLength和getWidth。结果是1，这是正方形的正确面积。动态分派的行为几乎总是我们想要的行为。动态分派的关键是，当方法被继承时，继承的方法获得对this的修订概念。我们的编码促进了动态分派，因为我们明确地将this传递到我们的方法中。

Java（以及几乎每种其他面向对象的语言）默认使用动态分派来调用方法。然而，在C++中，除非方法被声明为virtual，否则它将*not*动态分派。

5.1.7 Static Fields and Methods

静态字段和方法几乎存在于所有面向对象的语言中，它们仅仅是与特定对象实例无关，而是与类本身相关的字段和方法。

我们可以简单地通过将我们的类定义改为记录而不是函数来编码静态字段和方法。对象的创建本身可以是类的一个静态方法。实际上，这就是Smalltalk实现构造函数的方式。考虑以下对pointClass的重新实现，其中我们使用了静态字段。

```
Let pointClass = {

  newWithXY = Fun class ->
    Fun newx ->
    Fun newy -> {

    x = Ref newx;
    y = Ref newy;
    magnitude = Fun this -> Fun _ ->
      sqrt ((!(this.x)) + (!(this.y)))
  };

  new = Fun class -> Fun _ ->
    (class.newWithXY) class (class.xdefault)
                            (class.ydefault);

  xdefault = 4;
  ydefault = 3
} In

Let point = (pointClass.new) pointClass {} In
```

```
(point.magnitude) point {}
```

注意类方法 newWithXY 实际上负责构建 point 对象。new 简单地使用存储为类字段的默认值调用 newWithXY。这是一种非常干净的编码多个构造函数的方式。

也许编码最有趣的事情是具有静态字段的类开始看起来像我们原始的简单对象编码。仔细观察——注意类方法将 class 作为它们的第一个参数，这与常规方法将 this 作为参数的原因完全相同。因此，实际上，pointClass 只是一个恰好能够创建对象的另一个原始对象。

将类视为对象是Smalltalk中的主导范式。Java也通过反射API提供了一些将类视为对象的支持。即使在像C++这样的语言中，它不将类视为对象，设计模式如*Factory*模式[12]也捕捉了对象创建对象的这一概念。

这种编码真正体现了面向对象编程的精神，它是一种干净且特别令人满意的方式来思考类和静态成员。

5.2 The F♭OB Language

现在我们已经从 **F♭SR** 的已知语法角度研究了对象的编码，我们现在可以研究如何将这些特性直接添加到一种语言中。我们将称这种语言为 **F♭OB**，**F♭**，具有对象的语言。我们的对象编码在操作上是正确的，但没有添加对对象的语法支持，使得它们在实际操作中过于困难。例如，colorPoint 编码相当难以阅读。当类型被引入到语言中时，这种可读性问题只会变得更糟。

F♭OB 包括我们上面讨论的大部分功能：类、消息发送、方法、字段、super和this。我们支持与Smalltalk相同方式的信息隐藏：所有实例变量都是隐藏的（受保护的）且所有方法都是公开的。

F♭OB 也支持 **primitive objects**。原始对象是“内联”定义的对象，即不是从类创建的对象。它们是更轻量级的一种对象形式，类似于 Smalltalk 的块和 Java 的匿名类。在实际应用中，原始对象并不常见，但我们将其包含在 **F♭OB** 中，因为它几乎不需要做任何工作。表达式 new aClass 返回的值是一个对象，这意味着对象是一种一等表达式。只要我们的具体语法允许我们直接定义原始对象，就不需要在解释器中做额外的工作。

原始对象的一个有趣后果是，我们可以完全去除函数（不是方法）。函数可以简单地编码为原始对象的方法。因此，支持原始对象的对象导向语言具有许多高阶函数的优点。

5.2.1 Concrete Syntax

让我们通过一个示例介绍 **F♭OB** 具体语法。以下代码是之前 `pointClass` / `colorPointClass` 层次的实现（图5.3）。为了简单起见，此编码不包括类构造函数参数。

```
Let pointClass =
  Class Extends EmptyClass
    Inst
      x = 0;
      y = 0
    Meth
      magnitude = Fun _ -> sqrt(x + y);
      setx = Fun newx -> x := newx;
      sety = Fun newy -> y := newy

In Let colorPointClass =
  Class Extends pointClass
    Inst
      x = 0;
      y = 0;
      (* A use of a primitive object: *)
      color = Object
        Inst
        Meth red = 0; green = 0; blue = 0
    Meth
      magnitude =
        Fun _ ->
          mult(Super <- magnitude {})(This <- brightness {})
      (* An unnormalized brightness metric *)
      brightness = Fun _ ->
        color <- red + color <- green + color <- blue;
      setx = Super <- setx (* explicitly inherit *)
      sety = ...; setcolor = ...

In Let point = New pointClass
In Let colorPoint = New colorPointClass In
  (* Some sample expressions *)
  point <- setx 4;  point <- sety 7;
  point <- magnitude{};
  colorpoint <- magnitude {}
```

有很多关于这个语法的细节，所以让我们花些时间指出一些主要元素。首先，请注意 `This` 和 `Super` 是特殊的“保留变量”。在我们的 **F♭** 编码中，我们必须编写“`Fun this ->`”并将 `this` 作为显式参数传递。现在，自我意识是隐式发生的，并且 `This` 是对自我的引用。

注意使用原始对象来定义 color 字段。red、green 和 blue 值作为方法实现，因为字段始终是“私有”的。

在我们之前的编码中，我们定义基类时使用了一种风格，而在定义子类时使用了一种不同的风格。在 **F♭OB** 中，我们通过始终指定超类来使用相同的语法。为了允许不继承自任何其他类的基类，我们允许一个类扩展 EmptyClass，它只是一个不定义任何内容的特殊类。

F♭OB 实例变量使用第4.1.3节中讨论的状态的l/r值形式。不需要显式使用!运算符来获取实例变量的值。因此，**F♭OB**实例变量是可变的，不遵循所有变量都是不可变的Caml约定。请注意，方法参数仍然是不可变的。因此有两种变量：不可变的方法参数和可变的实例。由于很清楚哪些是实例变量，哪些不是，因此混淆的可能性不大。区分实例变量和常规变量的工作是解析器的任务。这种方法的优点是防止实例变量被外部人员直接操作。

方法体通常是函数，但不必是；它们可以是任何不可变、公开可用的值。例如，不可变实例可以被视为方法（参见上面示例中的 color 原始对象）。

注意我们仍然必须显式地继承方法。这不是最干净的语法，但它简化了解释器，并有助于将第5.2.3节中讨论的**F♭SR**翻译。

同样，没有构造函数。new用于创建新实例，并且，遵循Smalltalk，初始化是通过编写initialize方法来显式完成的。

为了简单起见，**F♭OB** 不支持静态字段和方法，也不采用前一小节中讨论的“类作为对象”的观点。

5.2.2 A Direct Interpreter

我们首先考虑一个针对 **F♭OB** 的直接解释器。抽象语法可以用以下 Caml 类型表示。

```
type ide = Ide of string | This | Super

type label = Lab of string

type expr = (* F♭ 表达式，包括 Let *)
(* Object holds the instance list and the method list *)
| Object of ((label * expr) list) * ((label * expr) list)
| Class of expr * ((label * expr) list) * ((label * expr) list)
| EmptyClass
| New of expr
| InstVar of label
| Send of  expr * label
  (* parser has to decide if a var. is InstVar or just a Var *)
```

```
| InstSet of label * expr
```

这里是对解释器的初步草图。这个解释器还不完整，但它给出了一般应该是什么样子的概念。

```
(* Substitute "sinst" for Super in all method bodies *)
let rec subst_super sinst meth =
  match meth with
    [] -> []
  | (l, body)::rest ->
      (l, subst(body, InstVar(sinst), Super))::
        (subst_super sinst rest)

let rec eval e =
  match e with
    ...
  | Object(inst, meth) -> Object(eval_insts inst, meth)
  | Send(term1, label) ->
      (match (eval term1) with
         Object(inst, meth) ->
           subst(selectMeth(meth, label),
                 Object(inst, meth), This)
         _ -> raise TypeMismatch)
  | Class(super, inst, meth) -> Class(super, inst, meth)
  | New(Class(super, inst, meth)) ->
      (match super with
         EmptyClass -> eval (Object(inst, meth))
       | s -> let sobj = eval(New super) in
              let sinst = (* A fresh instance variable label *) in
              let newinst = (sinst, sobj)::inst in
              let newmeth = subst_super sinst meth in
              eval (Object(newinst, newmeth))
  ...

and eval_insts inst =
  match inst with
    [] -> []
  | (l, body)::rest -> (l, eval(body))::(eval_insts rest)
```

这段代码草图很好地说明了This和Super的作用。我们只替换this *when we send a message*。这是因为This是一个动态构造，我们不知道它在运行时会是什么（参见5.1.6节中关于动态分发的讨论）。另一方面，Super是一个静态构造，在我们编写代码时就已经知道了，这允许我们在Class表达式评估后立即替换Super。

5.2.3 Translating F♭OB to F♭SR

另一种赋予 **F♭OB** 程序意义的方法是通过定义将 **F♭OB** 程序映射到 **F♭SR** 程序的翻译映射。这是对如何在 **F♭SR** 中编码对象（第5.1节的主题）的完整描述。

在以下第8章中，我们开发了一个针对 **F♭SR** 的编译器。为了获得针对 **F♭OB** 的编译器，我们只需添加一个翻译步骤，将 **F♭OB** 翻译为 **F♭SR**，然后使用此编译器简单地编译 **F♭SR**。因此，当我们完成所有工作后，我们将“免费”获得一个 **F♭OB** 编译器。本节仅讨论 **F♭OB** 到 **F♭SR** 的翻译。

翻译是第5.1节中给出的编码的正式化。尽管实际的面向对象的编译器要复杂得多，但本节至少应该提供一个可理解的关于面向对象编译器做什么的视图。具体的语法翻译以下以分块方式归纳定义。

$toFbSR($`Object Inst` x_1`=`e_1`;` $\ldots$`;` x_n`=`e_n `Meth` m_1`=`e'_1`;` $\ldots$`;` m_k`=`$e'_k$$) =$
{`inst = {`x_1`=Ref(`$toFbSR(e_1)$`);` $\ldots$`;` x_n`=Ref(`$toFbSR(e_n)$`)};`
`meth = {`m_1`= Fun this ->` $toFbSR(e'_1)$`;` $\ldots$`;`
m_k`= Fun this ->` $toFbSR(e'_k)$`} }`

$toFbSR($`Class Extends` e `Inst` x_1`=`e_1
`;` $\ldots$`;` x_n`=`e_{n1}`=`e'_1`;` $\ldots$`;` m_k`=`$e'_k$$) =$

`Meth` m `Fun _ -> Let super =` $(toFbSR(e))$ `{} In {inst = {`x_1`=Ref(`$toFbSR(e_1$
`));` $\ldots$`;` x_n`=Ref(`$toFbSR(e_n)$`)}; meth = {`m_1`= Fun this ->` $toFbSR(e'$
$_1$`);` $\cdots$`;` m_k`= Fun this ->` $toFbSR(e'_k)$`} }`

$toFbSR($`New` $e$$) = (toFbSR(e))$ {}

$toFbSR($`Super` m $args$$) =$ `super.meth.`m `this` $args$

$toFbSR($`EmptyClass`$) =$ `Fun _ -> {}`

$toFbSR(e$ `<-` m $args) =$

`Let ob =` $toFbSR(e)$ `In ob.meth.`m `ob` $args$, 对于 e 非 `Super`

$toFbSR(x$ `:=` $e) =$ `this.inst.`x `:=` $toFbSR(e)$

$toFbSR(x) =$ `!(this.inst.`x`)`，对于 x 一个实例变量

$toFbSR(y) = y$，对于 y 一个函数变量 $toFbSR($ 其他任何事物$) =$ 同态

翻译相当干净，除了消息到 `Super` 必须与其他消息略有不同处理，以便正确实现动态分派。再次注意，实例变量与函数变量处理不同，因为它们是可变的。空函数应用，即 `f ()`，可以写成空记录应用：`f {}`。在 `Fun _ -> `e 中的“ ”变量是 e 中未出现的任何变量。然而，在 FbDK 实现的 **F♭SR** 中，“ ”本身是一个有效的变量标识符。

作为一个如何进行此翻译的示例，让我们在5.2.1节中的`point` / `colorPoint`类的 **F♭OB**版本上执行它。结果是以下内容：

```
Let pointClass =
  Fun _ -> Let super = (Fun _ -> {}) {} In {
```

```
    inst = {
      x = Ref 3;
      y = Ref 4
    };
    meth = {
      magnitude = Fun this -> Fun _ ->
        sqrt ((!(this.inst.x)) + (!(this.inst.y)));
      setx = Fun this -> Fun newx ->
        (this.inst.x) := newx;
      sety = Fun this -> Fun newy ->
        (this.inst.y) := newy
    }
}

In Let colorPointClass =
  Fun _ -> Let super = pointClass {} In {
    inst = {
      x = Ref 3;
      y = Ref 4;
      color = Ref ({inst = {}; meth = {
        red = Fun this -> 45;
        green = Fun this -> 20;
        blue = Fun this -> 20
      }})
    };
    meth = {
      magnitude = Fun this -> Fun _ ->
        mult ((super.meth.magnitude) this {})
             ((this.meth.brightness) this {});
      brightness = Fun this -> Fun _ ->
        ((((!(this.inst.color)).meth.red) this) +
        ((((!(this.inst.color)).meth.green) this) +
        ((((!(this.inst.color)).meth.blue) this);
      setx = Fun this -> Fun newy ->
        (super.meth.setx) this newy;
      sety = Fun this -> Fun newy ->
        (super.meth.setx) this newy;
      setcolor = Fun this -> Fun c ->
        (this.inst.color) := c
    }
} In

(* Let colorPoint = New colorPointClass In
 *   colorPoint <- magnitude {}
 *)
Let colorPoint = colorPointClass {} In
```

```
  (colorPoint.meth.magnitude) colorPoint {};;
```

Interact with F♭SR. 上面的翻译代码应在 **F♭SR** 中运行良好，前提是你首先定义了 `mult` 和 `sqrt` 函数。`mult` 可以轻松定义为

```
Let Rec mult x = Fun y ->
  If y = 0 Then
    0
  Else
    x + (mult x (y - 1)) In ...
```

`sqrt` 不易定义。由于我们更关心对象的行为，而不是数值精度，只需编写一个返回其参数的虚拟 `sqrt` 函数:

```
Let sqrt = Fun x -> x In ...
```

现在，尝试使用基于文件的 **F♭SR** 解释器运行它。我们的虚拟 `sqrt` 函数为 7 的 `point` 版本的大小返回值，并且 `colorPoint` 大小将此结果乘以亮度之和（在本例中为 85）。结果是

```
$FbSR fbobFBsr.fbsr$
==> 595
```

在为抽象语法编写Caml版本*toFbSR*之后，通过将*toFbSR*与第8章中定义的函数组合，可以轻易地获得一个**F♭OB**编译器:

```
let FbOBcompile e = toC(hoist(atrans(clconv(toFbSR e))))
```

最后，还有几种处理这类编码的方法。有关编码对象的更多信息，请参阅[10]。

Chapter 6

Type Systems

在 **F♭** 中，如果我们评估表达式

```
3 + (If False Then 3 Else False),
```

我们将在运行时遇到某种解释器特定的错误。如果 **F♭** 有一个类型系统，这样的表达式将不允许进行评估。在 Lisp 中，如果我们定义一个函数

```
(defun f (x) (+ x 1)),
```

然后调用 `(f "abc")`，结果是一个运行时类型错误。同样，Smalltalk 表达式

```
String new myMessage
```

当运行时会产生“不支持的消息”异常。这两个运行时错误如果在支持静态类型系统的语言中运行前就能检测到，将会更好。C++代码

```
int a[10]; a[123] = 5;
```

执行未知且可能有害的效果，但等效的Java代码将在运行时抛出 `ArrayIndexOutOfBoundsException`，因为数组访问由动态类型系统进行检查。

这些只是类型系统旨在解决的问题类型的一些例子。在本章中，我们讨论了这类类型系统，以及用于推断和检查类型的算法。

6.1 An Overview of Types

一个 **type** 简单来说是一个在程序运行前隐式或显式注解的属性。类型声明是程序 *all* 执行时保持不变的不变量，可以表示为“这个变量始终持有 `String` 对象”或“这个函数始终返回 `tree` 表达式”之类的语句。

类型除了简单地减少运行时错误之外，还有许多其他优点。由于类型（以及模块签名）指定了程序的固有属性，因此它们充当了关于代码功能的精确且描述性的注释。以这种方式，类型有助于大型软件开发。

使用一种类型语言，编译时已知的信息更多，这有助于编译器生成更快的代码。例如，我们将在我们的 **F♭SR** 编译器中使用的记录实现（通过哈希，见第8章）在 C++ 中不需要，因为它具有静态类型系统。我们在编译时知道所有记录的大小，因此知道它们应该在内存中的布局位置。相比之下，Smalltalk 非常慢，缺乏静态类型系统是其中很大一部分原因。Strongtalk 系统 [8] 尝试将静态类型系统引入 Smalltalk。

最后，在无类型语言中，进行非常丑陋的“黑客”操作更容易。例如，一个列表 `[1;true;2;false;3;true]` 可以用无类型语言编写，但这很危险，更愿意将其表示为 `[(1,true);(2,false);(3,true)]`，在 OCaml 中，它具有类型 `(int * bool) list`。

然而，无类型语言相对于有类型语言有一个显著的优势：它们更具表现力。例如，考虑第5章中 **F♭SR** 对 **F♭OB** 的编码。这种编码在OCaml作为目标语言时将不起作用，因为OCaml的类型系统禁止记录多态。Y-组合子是另一个无类型语言真正发光的例子。**F♭** 可以通过 Y-组合子支持递归，但 **F♭** 的简单有类型版本不能，因为它无法被类型化。回想第2.3.5节，OCaml也无法对 Y-组合子进行类型化。

我们所有人都见过以多种方式使用的类型。以下是一些类型更常见维度的列表。

- 原子类型：`int`，`float`，……
- 类型构造函数，从类型生成类型：`'a -> 'b`，`'a * 'b`
- OCaml样式的类型构造函数定义通过 `type`
- C样式类型定义通过 `struct` 和 `typedef`
- 面向对象类型：类类型，对象类型
- 模块类型，或签名
- Java风格的接口
- 异常类型：⟨*method*⟩ `throws` ⟨*exception*⟩
- 参数多态性：`'a -> 'a`

- 记录/对象多态：将一个 `ColorPoint` 传递给期望一个 `Point` 的函数。

也存在几个较新的类型维度，目前是活跃的研究领域。

- 效果类型：类型“`int -x,y-> int`”表示在此函数中将分配变量 `x` 和 `y`。Java 的 `throws` 子句是方法的效果类型的一种形式。

- 具体类分析：对于变量 `x:Point`，具体类分析产生如 {`Point`、`ColorPoint`、`DataPoint`} 这样的集合。这意味着在运行时 `x` 可能是 `Point`、`ColorPoint` 或 `DataPoint` (以及，没有其他)。这在优化中很有用。

- 汇编语言类型 [3, 17]: 在汇编级别代码上放置类型，并拥有一个保证没有不安全指针操作的类型系统。

- 类型中的逻辑断言：`int -> { x:int | odd(x) }` 用于返回奇数的函数。

存在一个重要的区分需要在 *static* 和 *dynamic* 类型系统之间进行。
Static type systems 是我们通常所说的类型系统。静态类型系统是 C、C++、Java 和 OCaml 中找到的标准类型概念。类型由编译器检查，类型不安全的程序无法编译。

Dynamic type systems，另一方面，在运行时检查类型信息。Lisp、Scheme和Smalltalk实际上是动态类型的。实际上，**F♭**和**F♭SR**在技术上也是动态类型的，因为当表达式的类型不是预期的类型时，它们会引发一个`typeMismatch`。每次你使用一个函数时，运行时环境都会确保它是一个函数。如果你使用一个整数，它会确保它是一个整数，等等。这些运行时类型检查给运行时环境增加了很多开销，因此导致程序运行缓慢。

在某些静态类型语言中也会发生一些动态类型检查。例如，在Java中，向下转型会在运行时进行验证并可能引发异常。在Java和OCaml中，越界数组访问也会在运行时进行检查，因此是动态类型。在C或C{v*}中，数组访问没有类型，因为根本不进行任何检查。请注意，数组的*type*（即`int`，`float`）是静态检查的，但*size*是动态检查的。

最后，语言可以被 **untyped**。FbSR *compiler* 生成无类型代码，因为运行时错误会导致核心转储。理解无类型语言和有类型语言之间的区别很重要。在无类型语言中，根本没有任何检查，运行时可能会出现异常行为。在第8章中，我们使用类型转换“禁用”类型系统，将 **F♭SR** 编译为无类型C代码。机器语言是无类型语言的另一个例子。

为了真正看到无类型和动态类型语言之间的区别，考虑以下两个程序片段。第一个是使用类型系统“禁用”通过类型转换的C++代码。

```
#include <iostream>

class Calculation {
 public: virtual int f(int x) { return x; }
};

class Person {
 public: virtual char *getName() { return "Mike"; }
};

int main(int argc, char **argv) {
  void *o = new Calculation();
  cout << ((Person *)o)->getName() << endl;

  return 0;
}
```

代码编译无错误，但运行时输出为“ã¿。”但如果进行优化编译，结果为“Ãà。”在不同的计算机上运行，可能会导致段错误。使用不同的编译器，结果可能完全奇特且不可预测。关键是，因为我们使用的是无类型语言，没有动态检查，这种无意义的代码会导致未定义的行为。

与等效的Java代码的行为进行对比。回想一下，Java的动态类型系统在执行类型转换时会检查对象的类型。我们将使用大约相同的代码:

```
class Calculation {
  public int f(int x) { return x; }
}

class Person {
  public String getName() { return "Mike"; }
}

class Main {
  public static void main(String[] args) {
    Object o = new Calculation();
    System.out.println(((Person) o).getName());
  }
}
```

当我们运行这段Java代码时，其行为相当可预测。

```
Exception in thread "main" java.lang.ClassCastException: Calculation
at Main.main(example.java:12)
```

`ClassCastException` 是Java动态类型系统抛出的异常。不安全的代码在Java中永远不会执行，因此程序的行为是一致的且定义良好的。在C++版本中，没有动态检查，不安全的代码 *is* 执行，导致出现混乱和意外的行为。

6.2 TF♭: A Typed F♭ Variation

我们将使用前缀“**T**”来表示我们之前研究过的语言的类型版本。因此，我们有几种可能的语言：**TF♭**、**TF♭R**、**TF♭S**、**TF♭SR**、**TF♭OB**、**TF♭X**、**TF♭SRX**等。语言太多，无法逐一考虑，所以我们首先将查看**TF♭**作为热身，然后考虑完整的**TF♭SRX**。

6.2.1 Design Issues

在开始研究 **TF♭** 或任何类型语言之前，有一些设计问题需要解决。在给出 **TF♭** 的规范之前，我们将花一点时间来讨论这些问题。

第一个问题是要问我们的语言必须包含多少显式类型信息？程序需要装饰多少类型信息，以及编译器可以推断出多少？存在一系列的可能性。

在光谱的一端，我们甚至可以完全不使用装饰。我们只需坚持我们的无类型语言语法，并让编译器 *infer* 所有类型信息。

或者，我们可以使用有限的装饰，让编译器进行部分推断。可能性有很多。例如，C语言要求指定函数的参数和返回类型，以及声明变量的类型。然而，在函数体内，单个表达式不需要指定类型，因为这些类型可以推断出来。有些语言只要求声明函数参数的类型，然后函数的返回值会自动推断。

在光谱的另一端，每个子表达式及其标识符都必须用其类型进行装饰。然而，这太过极端，使得语言变得不可用。而不是编写

```
Fun x -> x + 1,
```

我们需要一些粗略的语法，例如

```
(Fun x -> (x:int + 1:int):int):(int -> int).
```

对于 **TF♭** 和 **TF♭SRX**，我们将集中讨论 C 和 Pascal 视角下的显式类型信息。我们指定函数参数和返回类型，以及声明的变量类型，并允许其余部分进行推断。

我们应该也说明类型检查和类型推断之间的区别。在任何类型语言中，编译器在生成代码之前应该 **typecheck** 程序。**Type inference algorithms** 推断类型 *and* 检查程序体没有类型错误。OCaml 是一个例子。

一个 **type checker** 通常只是检查给定在声明上的类型，确保身体类型正确。这是 C 、C++ 和 Java 的工作方式，尽管技术上它们也在推断一些类型，例如 3+4 的类型。

在任何情况下，都必须能够快速运行类型推断或类型检查算法。OCaml在理论上可以花费指数级时间来推断类型，但在实践中是线性的。

6.2.2 The TF♭ Language

最后，我们准备好查看 **TF♭**，一种类型化的 **F♭** 语言。为了简化问题，我们不会包括 `Let Rec` 的 **F♭** 语法，而只包括非递归、匿名函数。我们将在第 6.4 节中讨论类型递归。

与我们的操作语义和解释器开发类似，在讨论类型时，我们将定义两件事。首先，我们定义 **type systems**，这是为程序分配类型的语言无关的表示法。类型系统与操作语义类似。其次，我们定义 **type checkers**，这是类似于解释器的类型系统的 OCaml 实现。我们从类型系统开始。

Type Systems

类型系统是类似于操作语义的基于规则的正式系统。类型系统严格且正式地指定程序具有什么类型，并且与形式逻辑有强烈而深刻的平行关系（回忆我们在第2.3.5节中关于罗素悖论的讨论）。类型系统通常是一组关于类型断言的规则。

Definition 6.1. *(Type Environment) A **type environment**, Γ, is a set $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ of bindings of free variables types. If a variable x is listed twice in Γ, the rightmost (innermost) binding is the proper type. We write $\Gamma(x) = \tau$ to indicate that τ is the innermost type for x in Γ.*

Definition 6.2. *(Type Assertion) A **type assertion**, $\Gamma \vdash e\ \tau$, indicates that in type environment Γ, e is of type τ.*

TF♭ 类型在具体语法中是

τ `::= Int | Bool |` τ `->` τ.

我们可以用 OCaml 抽象语法来表示这个，如下所示

```
type fbtype = Int | Bool | Arrow of fbtype * fbtype
```

TF♭的表达式几乎与**F♭**相同，除了我们必须显式地用关于参数的类型信息装饰函数。例如，在具体语法中我们写

`Fun` x`:`τ `->` e

其中抽象语法表示是

```
Function of ide * fbtype * expr
```

The TF♭ Type Rules

我们现在准备定义 **TF♭** 类型规则。这些规则与之前我们查看的操作语义规则具有相同的结构；水平线表示“意味着”。以下三个规则是我们类型系统的公理（回想一下，公理是一条始终为真的规则，即一条线上没有任何内容的规则）。

$$\textit{(Hypothesis)} \quad \frac{}{\Gamma \vdash x : \tau \text{ for } \Gamma(x) = \tau}$$

$$\textit{(Int)} \quad \frac{}{\Gamma \vdash n : \texttt{Int} \text{ for } n \text{ an integer}}$$

$$\textit{(Bool)} \quad \frac{}{\Gamma \vdash b : \texttt{Bool} \text{ for } b \texttt{ True or False}}$$

Hypothesis 规则简单地说，如果一个变量 x 包含在类型环境 Γ 中，并且其类型为 τ，那么断言 $\Gamma \vdash x : \tau$ 是正确的。*Int* 和 *Bool* 规则简单地给字面表达式如 `7` 和 `False` 分配类型。这些规则构成了我们类型系统的基本案例。

接下来，我们有简单表达式的规则。

$$\textit{(+)} \quad \frac{\Gamma \vdash e : \texttt{Int}, \quad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e \texttt{ + } e' : \texttt{Int}}$$

$$\textit{(-)} \quad \frac{\Gamma \vdash e : \texttt{Int}, \quad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e \texttt{ - } e' : \texttt{Int}}$$

$$\textit{(=)} \quad \frac{\Gamma \vdash e : \texttt{Int}, \quad \Gamma \vdash e' : \texttt{Int}}{\Gamma \vdash e \texttt{ = } e' : \texttt{Bool}}$$

这些规则相当直接。对于加法和减法，操作数必须类型检查为 `Int`，表达式的结果是 `Int`。相等性类似，但类型检查为 `Bool`。请注意，相等性仅对整数操作数进行类型检查，而不是布尔操作数。*And*、*Or* 和 *Not* 规则类似，它们的定义应该是明显的。

If 规则稍微复杂一些。显然，表达式的条件部分必须类型检查为 `Bool`。但是，关于 `Then` 和 `Else` 子句呢？考虑以下表达式。

```
If e Then 3 Else False
```

这个表达式应该类型检查吗？如果 e 计算结果为 `True`，则结果为 `3`，一个 `Int`。如果 e 是 `False`，则结果为 `False`，一个 `Bool`。显然，这个表达式不应该类型检查，因为它并不总是计算为同一类型。这告诉我们，为了一个 `If` 语句能够类型检查，两个子句必须类型检查为同一类型，并且整个表达式的类型与两个子句相同。规则如下。

$$\textit{(If)} \quad \frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash e' : \tau \quad \Gamma \vdash e'' : \tau}{\Gamma \vdash \texttt{If } e \texttt{ Then } e' \texttt{ Else } e'' : \tau}$$

我们现在已经涵盖了所有重要的规则，除了函数和应用。*Function*规则与其他规则略有不同，因为函数引入了新的变量。函数体的类型取决于变量的类型本身，并且除非该变量在Γ，即类型环境中，否则不会进行类型检查。为了在我们的规则中表示这一点，我们需要对函数的变量附加到类型环境中进行类型断言。我们以以下方式执行此操作。

$$\textit{(Function)} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\texttt{Fun } x{:}\tau \texttt{ -> } e) : \tau \texttt{ -> } \tau'}$$

注意使用类型构造函数 -> 来表示整个函数表达式的类型。这个类型构造函数应该是熟悉的，因为 OCaml 类型系统使用相同的符号。此外，*Function* 规则包括向 Γ 添加一个假设。我们假设函数参数 x 的类型为 τ，并将这个假设添加到环境 Γ 中以推导出 e 的类型。*Application* 规则遵循 *Function* 规则：

$$\textit{(Application)} \quad \frac{\Gamma \vdash e : \tau \texttt{ -> } \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

就像在操作语义中一样，*derivation* of $\Gamma \vdash e : \tau$ 是一个规则应用的树，其中叶子是公理（*Hypothesis*、*Int* 或 *Bool* 规则），根是 $\Gamma \vdash e : \tau$。

让我们尝试一个求导示例。

⊢ `(Fun x:Int -> (Fun y:Bool ->`
`If y Then x Else x+1)) : Int -> Bool -> Int` 因为根据函数规则，只需证明 `x :`
`Int` ⊢ `(Fun y: Bool -> (If y Then x Else x+1)) :`
`Bool->Int` 因为再次根据函数规则，只需证明 `x : Int, y :`
`Bool` ⊢ `If y Then x Else x+1 : Int` 因为根据 *If* 规则，只需证明
`x : Int, y : Bool` ⊢ `y : Bool : Int, y : Bool` ⊢ `x : Int : Int, y :`
`Bool` ⊢ `x+1 : Int` 所有这些要么遵循 *Hypothesis* 规则，要么 `+` 和 *Hypothesis*.

考虑到上述内容，并令

```
f = (Fun x:Int -> (Fun y:Bool ->
   If y Then x Else x+1))
```

我们然后有

⊢ `f 5 True Int` 因为根据应用规则，⊢ `f : Int -> Bool -> Int`我们上面推导出的) : (`Int` 由*Int*规则得出。因此⊢ `f 5 : Bool -> Int` 由*Application*规则得出。鉴于这一点和⊢ `True : Bool` 由*Bool*规则得出，我们可以得到⊢ `f 5 True : Int` 由*Application*规则得出。

如我们之前提到的，**TF♭** 是一种非常弱的语言。无法定义递归函数。实际上，所有程序都保证会停止。因此，**TF♭** 是 *normalizing*。没有递归，**TF♭** 并不是非常有用。稍后我们将向 **TF♭SRX** 添加递归，并展示如何进行类型化。

Exercise 6.1. 尝试在 **TF♭** 中输入 Y-组合器。

现在我们已经为 **TF♭** 语言建立了一个类型系统，我们可以检测我们的程序是否 *well-typed*。但这意味着什么呢？答案以以下 **type soundness** 定理的形式出现。

Theorem 6.1. *If* $\vdash e\ \tau$*, then in the process of evaluating* e*, a "stuck state" is never reached.*

我们将不会精确地定义“卡住状态”的概念。它基本上是一个评估无法继续的点，例如 `0 (Fun x -> x)` 或 `(Fun x -> x) + 4`。在 **F♭** 解释器的术语中，卡住状态是引发异常的情况。这个定理断言类型系统可以防止运行时错误的发生。类似定理是大多数类型系统的目标。

6.3 Type Checking

为了编写一个 *interpreter* 用于 **TF♭**，我们只需修改 **F♭** 解释器以在运行时忽略类型信息。我们真正感兴趣的是编写一个 **type checker**。这是解释器的类型系统等价物；给定语言无关的类型规则，在特定语言中定义类型检查算法，即 OCaml。

一个类型检查算法 `typeCheck` 以类型环境 Γ 和表达式 e 作为输入。它要么返回 e 的类型 τ，要么抛出一个异常，指示 e 在环境 Γ 中类型不正确。

某些类型系统没有易于对应的类型检查算法。在 **TF♭** 中，我们很幸运，类型检查器几乎直接反映了类型规则。与解释器的情况一样，表达式的最外层结构决定了适用的规则。递归的流程是我们传递环境 Γ 和表达式 e *down*，并将结果类型 τ 返回 *up*。

这是对 **TF♭** 类型检查器的第一次尝试，`typecheck : envt -> expr -> fbtype`。Γ 可以实现为一个 `(ident * fbtype)` 列表，其中最新的项目位于列表的前端。

```
let rec typecheck gamma e =
  match e with
    (* lookup returns the first mapping of x in gamma *)
    Var x -> lookup gamma x
  | Function(Ide x,t,e1) ->
      let t' = typecheck (((Ide x),t)::gamma) e1 in
      Arrow(t,t')
  | Appl(e1,e2) ->
      let Arrow(t1,t2) = typecheck gamma e1 in
      if typecheck gamma e2 = t1 then
        t2
      else
        raise TypeError
  | Plus(e1,e2) ->
      if typecheck gamma e1 = Int and
         typecheck gamma e2 = Int then
        Int
      else
        raise TypeError
  | (* ... *)
```

Lemma 6.1. *`typecheck` faithfully implements the **TF♭** type system. That is,*

- $\vdash e\ \tau$ *if and only if* `typecheck [] e` *returns* τ*, and*
- `typecheck [] e` *raises a* `typeError` *exception if and only if* $\vdash e\ \tau$ *is not provable for any* τ*.*

Proof. 省略（通过案例分析）。 □

这个引理表明，`typecheck` 函数是 **TF♭** 类型系统的有效实现。

6.4 Types for an Advanced Language: TF♭SRX

现在我们已经研究了一个简单的类型系统和类型检查器，让我们继续研究一个更复杂语言的类型系统：**TF♭SRX**。我们包括了到目前为止使用的几乎所有语法，除了 **F♭OB** 的类和对象以及 **F♭V** 的变体。以下是使用 OCaml 类型定义的抽象语法。

```
type exnid = string
and expr =
    Var of ident
  | Function of ident * fbtype * expr
  | Letrec of ident * ident * fbtype * expr * fbtype * expr
  | Appl of expr * expr
  | Plus of expr * expr | Minus of expr * expr
  | Equal of expr * expr | And of expr * expr
  | Or of expr * expr | Not of expr
  | If of expr * expr * expr | Int of int | Bool of bool
  | Ref of expr | Set of expr * expr | Get of expr
  | Cell of int | Record of (label * expr) list
  | Select of  label * expr | Raise of expr * fbtype |
  | Try of expr * exnid * ident * expr
  | Exn of exnid * expr

and fbtype =
    Int | Bool | Arrow of fbtype * fbtype
  | Rec of label * fbtype list | Rf of fbtype
```

接下来，我们将定义 **TF♭SRX** 的类型规则。所有 **TF♭** 类型规则都适用，因此我们可以直接进入更有趣的规则。让我们从处理递归开始。我们真正编写的是 `In` 子句，但我们需要确保整个表达式的类型也是正确的。

$$\textit{(Let Rec)} \qquad \frac{\Gamma, f:\tau \texttt{ -> } \tau', x:\tau \vdash e:\tau', \qquad \Gamma, f:\tau \texttt{ -> } \tau' \vdash e':\tau''}{\Gamma \vdash (\texttt{Let Rec } f\ x{:}\tau \texttt{ = } e{:}\tau' \texttt{ In } e'):\tau''}$$

接下来，我们转向记录和投影。记录的类型简单地说是一张字段名称到与每个字段相关联的值类型的映射。投影的类型是投影的字段值的类型。规则是

$$\textit{(Record)} \qquad \frac{\Gamma \vdash e_1:\tau_1, \ldots, \Gamma \vdash e_n:\tau_n}{\Gamma \vdash \{l_1 \texttt{ = } e_1\texttt{; } \ldots\texttt{; } l_n \texttt{ = } e_n\} : \{l_1:\tau_1\texttt{; } \ldots\texttt{; } l_n:\tau_n\}}$$

$$\textit{(Projection)} \qquad \frac{\Gamma \vdash e : \{l_1:\tau_1\texttt{; } \ldots\texttt{; } l_n:\tau_n\}}{\Gamma \vdash e.l_i : \tau_i \text{ for } 1 \le i \le n}$$

我们还需要能够输入副作用。我们可以使用特殊类型 τ `Ref` 输入 `Ref` 表达式。`Set` 和 `Get` 表达式可以轻松跟随。

$$\textit{(Ref)} \qquad \frac{\Gamma \vdash e:\tau}{\Gamma \vdash \texttt{Ref } e:\tau \texttt{ Ref}}$$

$$\textit{(Set)} \qquad \frac{\Gamma \vdash e:\tau \texttt{ Ref}, \quad \Gamma \vdash e':\tau}{\Gamma \vdash e \texttt{ := } e':\tau}$$

$$\textit{(Get)} \qquad \frac{\Gamma \vdash e:\tau \texttt{ Ref}}{\Gamma \vdash \texttt{!}e:\tau}$$

最后，我们需要键入的其他类型的副作用是异常。要键入异常本身，我们将简单地使用类型 Exn。这允许所有异常以相同的方式进行键入。例如，以下代码

```
If b Then Raise (#IntExn 1) Else Raise (#BoolExn False)
```

将进行类型检查，并且具有类型 Exn。我们这样做是为了允许最大的灵活性。

因为它们改变了评估流程，只要参数类型检查到一个 Exn 类型，Raise 表达式应该始终进行类型检查。尽管如此，很难知道应该给提升表达式赋予什么类型。考虑以下示例。

```
If b Then Raise (#Exn True) Else 4
```

这个表达式应该类型检查为类型 Int。然而，根据我们的 If 规则，我们知道 Then 和 Else 子句必须具有相同的类型。因此，我们推断 Raise 表达式必须具有类型 Int 以便进行类型检查。在第 6.6.2 节中，我们看到了如何自动处理这种推断。现在，我们将简单地使用任意类型 τ 类型化 Raise 表达式。请注意，这是一项类型规则可以完美执行的操作，但在实际的类型检查器中很难实现。

接下来，注意 Try 表达式的 With 子句非常像一个函数。就像我们对函数所做的那样，我们还需要用类型信息来装饰标识符。然而，正如我们下面所看到的，这种装饰可以与我们现在要讨论的最终类型装饰相结合。

考虑以下表达式

```
Try Raise (#Ex 5)
With #Ex x:Int -> x + 1
```

这个表达式的类型显然是 Int。但假设这个例子稍微修改一下。

```
Try Raise (#Ex False)
With #Ex x:Int -> x + 1
```

这个表达式也会将类型转换为 Int。但假设我们使用我们的异常操作语义来评估这个表达式。当异常被抛出时，False 将替换 x，这可能导致运行时类型错误。问题是我们的操作语义对异常参数的类型一无所知。

我们可以不改变操作语义来解决这个问题。假设，我们不是写 #Ex False，而是写 #Ex@Bool False。类型规则将使用 @Bool 来验证参数确实是一个 Bool，而解释器将简单地看到字符串 “#Ex@Bool”，它将被用来与 With 子句匹配。这也消除了在 With 子句标识符上使用类型装饰的需要，因为它起到了相同的作用。从某种意义上说，这非常类似于 Java 或 C++ 中的方法重载。当一个方法被重载时，需要类型 *and* 和方法名来唯一地识别正确的方法。我们的最终异常语法如下：

```
Try Raise (#Ex@Bool False)
With #Ex@Int x -> x + 1
```

这个表达式类型检查通过，类型为 `Int`。当评估时，结果是 `Raise #Ex@Bool False`，即异常没有被 `With` 子句捕获。这是我们想要的行为。

现在我们已经解决了类型友好的语法，让我们继续讨论实际的类型规则。它们相当直接。

$$\textit{(Raise)} \quad \frac{\Gamma \vdash e : \tau'}{\Gamma \vdash \texttt{(Raise \#}xn\texttt{@}\tau'\ e\texttt{)} : \tau \text{ for arbitrary } \tau}$$

$$\textit{(Try)} \quad \frac{\Gamma \vdash e : \tau, \quad \Gamma, x : \tau' \vdash e' : \tau}{\Gamma \vdash (\texttt{Try}\ e\ \texttt{With \#}xn\texttt{@}\tau'\ x\ \texttt{->}\ e') : \tau}$$

使用这些规则，让我们输入上面的表达式：

$\vdash$ (`Try Raise (#Ex@Bool False) With #Ex@Int x -> x + 1`) : `Int`

因为根据 *Raise* 规则，$\vdash$ `Raise (#Ex@Bool False)` : τ 对于任意的 τ 因为根据 *Bool* 规则，$\vdash$ `False` : `Bool` 并且，根据 `+` 规则 `x` : `Int` $\vdash$ `x + 1` : `Int` 通过应用 *Int* 和 *Hypothesis* 规则

因此，根据 *Try* 规则，我们推断原始表达式的类型为 `Int`。

Exercise 6.2. 为什么没有针对细胞的类型规则?

Exercise 6.3. 如何在不使用 `Let Rec` 的情况下支持 **TF♭SRX** 中的递归函数，同时仍然要求递归函数正确类型检查？证明你的解决方案可以正确类型检查。

Exercise 6.4. 尝试输入我们迄今为止研究的一些未输入的程序，例如，Y-组合子，`Let`，序列缩写，递归阶乘函数以及列表的编码。有没有完全无法类型检查的?

Exercise 6.5. 提供一个非递归的 **F♭SR** 表达式示例，它在评估时正确地得到一个值，但在以 **TF♭SRX** 编写时无法通过类型检查。

6.5 Subtyping

类型系统我们在上面讨论的是相当充分的，但仍有许多程序即使没有运行时错误，也无法通过类型检查。我们希望考虑的第一个扩展到我们的标准类型系统是所谓的 **subtyping**。子类型的主要优势在于它允许记录和对象多态通过类型检查。

子类型应该已经是从Java和C++中熟悉的概念。子类是子类型，扩展或实现一个接口给出一个子类型。

6.5.1 Motivation

设 u 以一个例子来激励子类型。考虑一个fu 函数

```
Fun x:{l:Int} -> (x.l + 1):Int.
```

此函数将一个包含字段 `l` 的类型为 `Int` 的记录作为参数。在无类型 **F♭R** 语言中，传递给函数的记录除了 `l` 之外还可以包含其他字段，并且调用

```
(Fun x -> x.l + 1) {l = 4; m = 6}
```

不会产生运行时错误。然而，这不会通过我们的 **TF♭SRX** 规则进行类型检查：函数参数类型与传入值的类型不同。

解决方案是重新考虑记录类型，例如 {`m:Int; n:Int`}，表示至少包含 `m` 和 `n` 字段且字段类型为 `Int` 的记录，但也可能包含其他未知类型的字段。考虑之前的记录操作及其类型：在这种记录类型解释下，*Record* 和 *Projection* 规则仍然有意义。旧规则仍然合理，但我们需要一条新规则来反映对记录类型的新理解：

$$\textit{(Sub-Record}_0\textit{)} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n\}}{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_m : \tau_m\} \text{ for } m < n}$$

此规则有效，但不如我们能做到的好。要了解原因，考虑另一个例子，

$F =$ `Fun f -> f ({x=5; y=6; z=3}) + f({x=6; y=4}).`

这里函数 `f` 应该非正式地接受至少包含 `x` 和 `y` 字段的记录，但也应接受包含额外字段的记录。让我们尝试编写函数 F。

F : `({x:Int; y:Int} -> Int) -> Int`

考虑申请 $F\ G$ 的应用

$G =$ `Fun r -> r.x + r.x.`

如果我们对 G 进行类型检查，将得到 G : `{x:Int} -> Int`，这与 F 的参数 `{x:Int; y:Int} -> Int` 并不完全匹配，因此即使它不会导致运行时错误，类型检查 $F\ G$ 也会失败。

实际上我们 *could* 已经给 G 分配了类型 `{x:Int; y:Int} -> Int`，但现在才知道那应该是我们当时输入 G 时应该使用的类型。$Sub\text{-}Rec_0$ 规则

这里也没有帮助。我们需要一条规则，即具有记录类型参数的函数可以将其记录参数类型具有字段 *added*，因为这些字段将被忽略：

$$\textit{(Sub-Function}_0\textit{)} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n\}\ \texttt{->}\ \tau}{\Gamma \vdash e : \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n;\ \ldots;\ l_m : \tau_m\}\ \texttt{->}\ \tau}$$

使用此规则，$F\ G$ 确实会进行类型检查。问题是我们仍然需要其他规则。考虑记录内部的记录：

```
{pt = {x=4; y=5}; clr = 0} : {pt:{x:Int}; clr:Int}
```

仍然是一个有效的类型，因为将忽略 `y` 字段。然而，也没有类型规则允许这种类型。

6.5.2 The STF♭R Type System: TF♭ with Records and Subtyping

现在应该很清楚，我们上面试图使用的策略永远无法奏效。我们需要为每条记录和函数的每一种可能组合制定不同的类型规则！

解决方案是拥有一个单独的规则集 **subtyping**，专门用来确定一种类型是否可以在另一种类型的位子上使用。$\tau <: \tau'$ 读作“τ 是 τ' 的子类型”，意味着类型 τ 的对象也可以被视为类型 τ' 的对象。添加到 **TF♭** 类型系统中的规则（以及 **TF♭SRX** 的记录规则）是

$$\textit{(Sub)} \qquad \frac{\Gamma \vdash e : \tau, \quad \vdash \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

我们还需要使我们的子类型运算符具有自反性和传递性。这可以通过以下两条规则实现。

$$\textit{(Sub-Refl)} \qquad \frac{}{\vdash \tau <: \tau}$$

$$\textit{(Sub-Trans)} \qquad \frac{\vdash \tau <: \tau', \quad \vdash \tau' <: \tau''}{\vdash \tau <: \tau''}$$

我们的子类型记录规则需要做两件事。它需要确保如果一个记录 B 与具有一些额外字段的记录 A 相同，那么 B 应该是 A 的子类型。它还需要处理记录嵌套记录的情况。如果 B 的字段都是 A 的字段的子类型，那么 B 也应该成为 A 的子类型。我们可以用一个简单的规则来简洁地反映这一点，如下所示。

(Sub-Record)

$$\frac{\vdash \tau_1 <: \tau_1', \ldots, \tau_n <: \tau_n'}{\vdash \{l_1 : \tau_1;\ \ldots;\ l_n : \tau_n;\ \ldots;\ l_m : \tau_m\} <: \{l_1 : \tau_1';\ \ldots;\ l_n : \tau_n'\}}$$

函数规则还必须做两件事。如果函数 A 和 B 相等，除了 B 返回的是 A 返回的子类型，那么 B 是 A 的子类型。然而，如果 A 和 B 相同，除了 B 的参数是 A 的参数的子类型，那么 A *is a subtype of* B。简单来说，一个函数要成为另一个函数的子类型，它必须少取多给。规则应该使这一点清楚：

$$\textit{(Sub-Function)} \qquad \frac{\vdash \tau_0' <: \tau_0, \quad \vdash \tau_1 <: \tau_1'}{\vdash \tau_0 \texttt{ -> } \tau_1 <: \tau_0' \texttt{ -> } \tau_1'}$$

从前一节的讨论和例子中，应该很明显，这组更通用的规则将适用。

6.5.3 Implementing an STF♭R Type Checker

自动化类型检查**STF♭R**实际上相当困难。有两种方法可以使任务更容易。第一种是添加更多的显式类型修饰以帮助类型检查器。第二种是在*constraint*形式中完全推断类型，这是第6.7节讨论的主题。

这里，我们简要概述了如何编写 `typecheck` 函数用于 **STF♭R**。**TF♭** 类型检查器要求某些类型必须相同，例如，函数定义域类型必须与函数应用中的函数参数类型相同。

$$\textit{(Application)} \qquad \frac{\Gamma \vdash e : \tau \texttt{ -> } \tau', \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\ e' : \tau'}$$

在 **STF♭R** 的这一点，我们需要查看是否可以进行子类型化。`typecheck`(e') 返回 τ'' 然后是 $\tau'' <: \tau$ 通过一个函数 `areSubtypes`(τ'', τ) 进行检查。这通过 Sub 规则产生一个有效的证明。其他需要类型匹配的 **TF♭** 规则的规则被推广，以允许使用 Sub 规则。

Exercise 6.6. 实现 `areSubtypes` 函数 n.

6.5.4 Subtyping in Other Languages

有趣的是看到子类型在其他语言中的应用。例如，考虑Java和C++。在这些语言中，*subclassing*是子类型的主要形式。一个子类是其扩展的类的子类型。在Java中，一个类也是其实现的任何接口的子类型。

Java和C++因此更加严格。假设存在具有结构{`x:Int; y:Int; color:Int`}和{`x:Int; y:Int`}的类，它们不属于上述两种情况之一（即前者不是后者的子类，并且两者没有共享的接口）。因此，这两个类在Java或C++中不是子类型，但在**STF♭R**中是。然而，声明的子类型关系有优势，即子类型关系不必完全推断。

OCaml对象更灵活，因为没有对层次结构的限制，但同时也更不灵活，因为实际上没有对象多态性——需要显式地将`ColorPoint`强制转换为`Point`。有几种具有类型推断和子类型的研究语言，但类型通常很复杂或难以阅读[9]。

6.6 Type Inference and Polymorphism

类型推断最初是由ML的原始创造者Robin Milner以及逻辑学家J. Roger Hindley独立发现的。Milner的“算法W”的关键思想是最初给所有变量赋予任意类型`'a`，然后根据程序指示将类型*unify*或等同，**6.6.1 Type Inference and Polymorphism**。例如，如果应用程序`f x`具有类型`'a -> 'b`并且`x`具有类型`'c`，我们可能将`'a`和`'c`等同。

我们将探讨完整类型推断，即在没有任何显式类型信息的程序上的类型推断。

6.6.1 Type Inference and Polymorphism

类型推断与**parametric polymorphism**（紧密相连，通常称为“**generic types**”）。考虑函数`Fun x -> x`。如果没有多态性，这个函数可以推断出什么类型？`Int -> Int`是一个错误的答案，因为该函数可以在一个传递布尔值的环境中使用。使用类型推断，我们需要实现一个称为主类型的概念。

Definition 6.3. *(Principal Type) A* ***principal type*** τ *for expression* e *(where* $\vdash e : \tau$*) has the following property. For any other type* τ' *such that* $\vdash e : \tau'$*, for any context* C *for which* $\vdash C[e : \tau'] : \tau''$ *for any* τ''*, then* $\vdash C[e : \tau\,] : \tau'''$ *as well, for some* τ'''*.*

原意是，没有其他类型系统能让程序类型检查的使用更加广泛，因此主要类型系统总是最好的。我们类型推断算法的期望特性是它将始终推断主要类型。一个重要的推论是，在主要类型算法中，我们知道推断永远不会“妨碍”程序员，例如，当程序员想要在 `Int -> Int` 上使用它时，推断 `Bool -> Bool` 作为恒等函数。

OCaml 推断身份函数的类型 `'a -> 'a`，这可以证明它是该函数的主类型。事实上，OCaml 类型推断始终推断主类型。

6.6.2 An Equational Type System: EF♭

我们将以一种非标准的方式介绍类型推断。Milner的“算法W”*eagerly*统一了`'a`和`'c`：它在任何地方都用一个替换另一个。我们将介绍**equational inference**，在其中我们懒散地积累方程如`'a` $=$ `'c`，然后在算法的末尾解决方程组。

我们将研究 **EF♭**，它是 **F♭** 的一个简单、等式类型的版本。**EF♭** 使用与 **F♭** 语言相同的表达式语法，因为 **EF♭** 没有任何类型修饰。**EF♭** 类型是

$$\begin{array}{lcll} \tau & ::= & \texttt{Int} \mid \texttt{Bool} \mid \tau \texttt{ -> } \tau \mid \alpha & \textit{types} \\ \alpha & ::= & \texttt{'a} \mid \texttt{'b} \mid \ldots & \textit{type variables} \end{array}$$

EF♭ 推理期间的类型将包括一个额外的集合 *constraining equations*、E，这会约束类型变量的行为。因此，对于 **EF♭** 的类型判断形式为 $\Gamma \vdash e : \tau \backslash E$，与之前相同，只是在旁边附加了一组方程。$E$ 的每个成员都是一个类似于 `'a` $=$ `'c` 的方程。将使用方程类型来辅助推理。以下是整体方法的概述。

1. 推断整个程序的等式类型。
2. 如果方程不一致，则宣布存在类型错误。
3. 如果方程是一致的，简化它们以给出推断类型。

这是一个事实，如果方程式中没有不一致性，它们总是可以被简化为无方程式类型。

Definition 6.4. *(Equational Type) An* ***equational type*** *is a type of the form*

$\tau \backslash \{\tau_1 = \tau_1', \ldots, \tau_n = \tau_n'\}$

每个 $\tau = \tau'$ 是一个关于类型的方程，意味着 τ 和 τ' 与类型具有相同含义。我们将让 E 表示一些任意类型的方程集合。例如，

`Int -> 'a`$\backslash\{$`'a = Int -> 'a1, 'a1 = Bool`$\}$

这是一个等式类型。如果你这么想，这实际上与类型完全相同

```
Int -> Int -> Bool.
```

这是已知为 *equation simplification*，是我们类型推断算法中要执行的一个步骤。也可以编写无意义的类型，例如

`Int -> 'a`$\backslash\{$`'a = Int -> 'b, 'a = Bool`$\}$

这不能是一个类型，因为它意味着函数和布尔值是同一类型。这样的方程集被认为是 *inconsistent*，并将与类型推断过程的失败相对应。还存在一些看似不矛盾的自引用（循环）类型的可能性：

`Int -> 'a`$\backslash\{$`'a = Int -> 'a `$\}$

OCaml不允许此类类型，我们也将最初禁止它们。这些类型无法简化，这也是OCaml禁止它们的主要原因：语言的用户将不得不看到一些类型等式。

The EF♭ Type Rules

EF♭系统是一组以下规则。请注意，Γ在**TF**♭规则中扮演着相同的角色。它是一种类型环境，将变量绑定到简单（非等价）类型。我们的公理规则看起来与**TF**♭规则非常相似。

(Hypothesis) $$\frac{}{\Gamma \vdash x : \tau \backslash \emptyset \text{ for } \Gamma(x) = \tau}$$

(Int) $$\frac{}{\Gamma \vdash n : \texttt{Int} \backslash \emptyset \text{ for } n \text{ an integer}}$$

(Bool) $$\frac{}{\Gamma \vdash b : \texttt{Bool} \backslash \emptyset \text{ for } b \text{ a boolean}}$$

规则对于 `+`、`-` 和 `=` 也类似于它们的 **TF♭** 对应项，但现在我们必须将每个操作数的方程的并集作为整个表达式的类型方程集，并添加一个方程来反映操作数的类型。

(+) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e \texttt{ + } e' : \texttt{Int} \backslash E \cup E' \cup \{\tau = \texttt{Int}, \tau' = \texttt{Int}\}}$$

(-) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e \texttt{ - } e' : \texttt{Int} \backslash E \cup E' \cup \{\tau = \texttt{Int}, \tau' = \texttt{Int}\}}$$

(=) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e \texttt{ = } e' : \texttt{Bool} \backslash E \cup E' \cup \{\tau = \texttt{Int}, \tau' = \texttt{Int}\}}$$

And、*Or*和*Not*规则以类似的方式定义。If的规则也与**TF♭** *If*规则类似。注意，尽管如此，我们并没有像之前规则那样立即推断出某种类型。相反，我们推断出类型α，并将α与Then和Else子句的类型相等。

(If)

$$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E', \quad \Gamma \vdash e'' : \tau'' \backslash E''}{\Gamma \vdash (\texttt{If } e \texttt{ Then } e' \texttt{ Else } e'') : \alpha \backslash E \cup E' \cup E'' \cup \{\tau = \texttt{Bool}, \tau' = \tau'' = \alpha\}}$$

最后，我们准备好函数和应用规则。函数不再具有显式的类型信息，但我们可以简单地选择一个新的类型变量 `'a` 作为函数参数，并在稍后将其包含在方程中。应用规则也选择一个新的类型变量 `'a` 类型，并添加一个方程，其中 `'a` 作为函数类型的右侧。规则应该使这一点清楚。

(Function) $$\frac{\Gamma, x : \alpha \vdash e : \tau \backslash E}{\Gamma \vdash (\texttt{Fun } x \texttt{ -> } e) : \alpha \texttt{ -> } \tau \backslash E}$$

(Application) $$\frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma \vdash e' : \tau' \backslash E'}{\Gamma \vdash e\ e' : \alpha \backslash E \cup E' \cup \{\tau = \tau' \texttt{ -> } \alpha\}}$$

这些规则几乎直接定义了等式类型推理过程：证明基本上可以从底部（叶子）向上构建。每个添加的等式表示两种应该相等的数据类型。

Solving the Equations

从 **EF♭** 类型规则中应立即清楚的是，*any* 语法正确的程序可以通过这些规则进行类型化。一个程序是否 *well-typed* 并不是在我们实际解决整个程序等式类型的方程组之前确定的。**EF♭** 类型规则实际上只是在累积约束。

解决方程涉及两个步骤。首先，我们计算方程的 *closure*，产生通过传递性等保持的新方程。接下来，我们检查是否存在任何不一致的方程，例如表示类型错误的 `Int` $=$ `Bool`。

以下算法计算集合 E 的等价闭包。

- 对于形式为 τ_0 `->` $\tau_0' = \tau_1$ `->` τ_1' 的每个方程在 E 中，将 $\tau_0 = \tau_1$ 和 $\tau_0' = \tau_1'$ 添加到 E。
- 对于每个方程组 $\tau_0 = \tau_1$ 和 $\tau_1 = \tau_2$ 在 E 中，通过传递性) 将方程 $\tau_0 = \tau_2$ 添加到 E (。
- 重复（1）和（2），直到无法再向 E 添加更多方程。

请注意，我们将隐式地使用这些方程的对称性质，因此无需为每个方程 $\tau_1 = \tau_0$ 添加 $\tau_0 = \tau_1$。

闭包用于揭示不一致之处。例如，

闭包({`'a` $=$ `Int -> 'b`, `'a` $=$ `Int -> Bool`, `'b` $=$ `Int`}) $=$
{`'a` $=$ `Int -> 'b`, `'a` $=$ `Int -> Bool`,
`'b` $=$ `Int`, `Int -> 'b` $=$ `Int -> Bool`,
`Int` $=$ `Int`, `'b` $=$ `Bool`, `Int` $=$ `Bool`},

直接揭示不一致 `Int` $=$ `Bool`。

E的闭包可以在多项式时间内计算。计算闭包后，如果满足以下条件，则约束是一致的

1. 没有发现任何立即的不一致性，例如 `Int` $=$ `Bool`、`Bool` $=$ τ `->` τ' 或 `Int` $=$ τ `->` τ'。

2. 没有自引用方程（我们将很快解决这个问题）。

如果方程是一致的，下一步是解决方程约束。我们通过用实际类型替换类型变量来完成这项工作。算法如下。给定 $\tau \backslash E$,

1. 在 τ 中将某些类型变量 α 替换为 τ'，前提是 $\alpha = \tau'$ 或 $\tau' = \alpha$ 出现在 E 中，并且
   - τ' 不是类型变量，或
   - τ' 这是一个类型变量 α'，它在字典序上紧随 α。
2. 重复（1），直到不再可能进行此类替换。

注意，步骤（1）考虑了每个方程的对称等价形式，这就是为什么我们没有在闭包中包含它们。然而，该算法存在缺陷：替换可能永远继续。这发生在 E 包含循环类型时。回忆一下自引用类型的例子

```
Int -> 'a\{'a = Int -> 'a}.
```

尝试解决这些约束会导致非终止的链

```
Int -> Int -> 'a\{'a = Int -> 'a}
Int -> Int -> Int -> 'a\{'a = Int -> 'a}
Int -> Int -> Int -> Int -> 'a\{'a = Int -> 'a}
...
```

解决方案是在尝试解方程之前检查这样的循环。做到这一点最好的方法是使用图论语境来表述问题。具体来说，我们定义一个有向图 G，其中节点是 E 中的类型变量。如果 `'a` $= \tau$ 是 E 中的一个方程且 `'b` 出现在 τ 中，则从 `'a` 到 `'b` 存在一个有向边。

如果我们发现 G 中存在一个循环，并且至少有一条表示约束的边，而这个约束并非仅仅是类型变量之间的 (`'a = 'b`)，则我们提出一个 `typeError`。

总结来说，我们的整个 **EF**$\flat$ 类型推断算法如下。对于表达式 e，

1. 通过应用 **EF**$\flat$ 类型规则，生成 $\vdash e$ 的证明：$\tau \backslash E$。这样的证明总是存在的。

2. 通过计算其闭包扩展 E。3. 检查 E 是否立即不一致。如果是，则引发一个 `typeError`。

4. 使用上述算法检查 E 中的循环。如果存在循环，则引发一个 `typeError`。

5. 使用上述方程求解算法求解 E。如果 E 中没有循环，则此算法将始终终止。6. 输出：由求解算法产生的 e 的解类型 τ'。

Theorem 6.2. *The typings produced by the above algorithm are always principal.*

这个定理的证明超出了本书的范围。让我们用一个类型推断算法的实际例子来结束。假设我们想要推断以下表达式的类型{v*}

```
(Fun x -> If x Then 3 Else 4) False
```

首先，我们证明以下内容。

根据 *Application* 规则，
$\vdash$ ((Fun x -> If x Then 3 Else 4) False)
’c\{’a = Bool, Int = ’b, ’a -> ’b = Bool -> ’c} 因为，根据 *Fun* 规则，(Fun x -> If x Then 3 Else 4) :
’a -> ’b\{’a = Bool, Int = ’b} 因为，根据 *If* 规则，x :
’a $\vdash$ If x Then 3 Else 4 : ’b\{’a = Bool, Int = ’b} 因为
根据 *Int* 和 *Hypothesis* 规则，x : ’a $\vdash$ x : ’a\$\emptyset$，x : ’a $\vdash$ 3 :
Int\$\emptyset$，以及 x : ’a $\vdash$ 4 : Int\$\emptyset$ 并且，根据 *Bool* 规则，
$\vdash$ False : Bool\$\emptyset$

给定以下证明的

$\vdash$ ((Fun x -> If x Then 3 Else 4) False)
’c\{’a = Bool, Int = ’b, ’a -> ’b = Bool -> ’c}

我们计算要闭合的方程集的闭包为

{’a = Bool, Int = ’b, ’a -> ’b = Bool -> ’c, ’b = ’c, Int = ’c}

该集合并非立即不一致，且不包含任何循环。因此，我们解方程。在这种情况下，τ仅包含’c，我们可以用Int替换’c。我们将Int输出为表达式的类型，这显然是正确的。

6.6.3 PEF♭: EF♭ with Let Polymorphism

所有我们在 **EF♭** 上的工作之后，我们仍然没有多态性，我们只有类型变量。为了说明这一点，考虑以下函数

```
Let x = Fun y -> y In (x True); (x 0)
```

回忆我们将 Let 编码为函数，这个表达式等价于

```
(Fun x -> (x True); (x 0)) (Fun y -> y)
```

在OCaml中，这样的程序类型检查良好。Fun y -> y的不同使用可以有不同类型。尽管如此，考虑一下在类型化这个表达式时**EF♭**会做什么。

$\vdash$ (Fun x -> (x True); (x 0)) :
’a -> ’c\{’a = Bool -> ’b, ’a = Int -> ’c, ...}

但是当我们计算这个等价类型的闭包时，我们得到方程 `Int` = `Bool`! 发生了什么错误? 在这种情况下的问题是，在主体中对 `x` 的每次使用都使用了相同的类型变量 `'a`。实际上，当我们类型化 `Fun y -> y` 时，我们知道 `'a` 可以是任何东西，所以对于不同的使用，`'a` 可以是不同的事物。我们需要将这种直觉融入我们的类型系统，以正确处理此类情况。我们在 **PEF**♭ 中定义了这样的类型系统，它是具有 **EF**♭ 和 `Let` 以及 `Let`-多态性的 **EF**♭。

PEF♭ 具有特殊的 `Let` 类型规则，其中我们允许在 Γ 中使用一种新的类型: $\forall\alpha_1 \ldots \alpha_n.\, \tau$。这被称为 **type schema**，并且只能出现在 Γ 中。类型模式的一个例子是 $\forall\alpha.\ \alpha$ `->` α。请注意，类型变量 $\alpha_1 \ldots \alpha_n$ 被此类型表达式视为 *bound*。

新规则为 `Let` 是

$$\textit{(Let)} \qquad \frac{\Gamma \vdash e : \tau \backslash E, \quad \Gamma, x : \forall\alpha_1 \ldots \alpha_n.\, \tau' \vdash e' : \tau'' \backslash E'}{\Gamma \vdash (\texttt{Let}\ \ x\ \ \texttt{=}\ \ e\ \ \texttt{In}\ \ e') : \tau'' \backslash E'}$$

在 τ' 是使用上述算法求解 $\vdash\ e : \tau \backslash E$ 的解的情况下，且 τ' 具有自由类型变量 $\alpha_1 \ldots \alpha_n$，这些变量在 Γ 中未出现。

注意，由于我们在该规则中调用简化算法，这意味着完整的算法不是上面给出的干净的三遍推断闭包简化形式: 规则需要在一些子推导上调用close-simplify。

我们还需要添加一个公理以确保每个`Let`的使用都分配到一个新的类型变量。规则是

$$\textit{(Let Inst.)} \qquad \frac{}{\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\, \tau' \vdash x : R(\tau') \backslash \emptyset}$$

在 $R(\tau')$ 是将变量 $\alpha_1 \ldots \alpha_n$ 重命名为新名称的地方。由于每次使用 `x` 时这些名称都是新的，因此不同的使用不会像上面那样冲突。

将有助于看到此类类型系统在实际中的示例。让我们输入上面的示例程序:

```
Let x = Fun y -> y In (x True); (x 0)
```

我们有

$\vdash$ `Fun y -> y : 'a -> 'a`$\backslash\emptyset$

此约束集显然具有解类型 `'a -> 'a`。因此，我们然后在假设 `x` 具有类型 $\forall$`'a.'a -> 'a` 的情况下对 `Let` 体进行类型检查。

`x :` $\forall$`'a.'a -> 'a` $\vdash$ `x : 'b -> 'b`$\backslash\emptyset$

根据 *Let-Inst* 规则。然后

$$\texttt{x} : \forall\texttt{'a.'a -> 'a} \vdash \texttt{x True} : \texttt{'c} \backslash \{\texttt{'b -> 'b} = \texttt{Bool -> 'c}\}$$

同样地,

$$\texttt{x} : \forall\texttt{'a.'a -> 'a} \vdash \texttt{x 0} : \texttt{'e} \backslash \{\texttt{'d -> 'd} = \texttt{Int -> 'e}\}$$

这里的重要点是，通过 *Let-Inst* 规则，x 的使用得到了一个不同的类型变量，'d。将这两个结合起来，类型就像这样

$$\begin{array}{l}\texttt{x} : \forall\texttt{'a.'a -> 'a} \vdash \texttt{x True; x 0} : \\ \quad \texttt{'e} \backslash \{\texttt{'b -> 'b} = \texttt{Bool -> 'c}, \texttt{'d -> 'd} = \texttt{Int -> 'e}\}\end{array}$$

根据 *Let* 规则，然后产生

$$\begin{array}{l}\vdash (\texttt{Let x = Fun y -> y In (Fun x -> x True; x 0)}) : \\ \quad \texttt{'e} \backslash \{\texttt{'b -> 'b} = \texttt{Bool -> 'c}, \texttt{'d -> 'd} = \texttt{Int -> 'e}\}\end{array}$$

由于 'b 和 'd 是不同的变量，我们没有之前遇到的冲突。

6.7 Constrained Type Inference

存在一个原因，我们以上述形式呈现Hindley-Milner类型推断：如果我们用子类型约束替换等价约束，$<:$ ，我们可以执行约束类型推断。要理解进行这种泛化的好处，最简单的方法就是看看规则。

F♭ 这不是展示用子类型替换等价性的强大功能的最佳系统。由于该语言没有记录，因此没有任何有趣的子类型可以发生。为了展示子类型的有用性，我们因此在一个有记录的环境**F**♭**R**中定义约束。**F**♭**R**加上约束是**CF**♭**R**。我们可以将**CF**♭**R**与未研究的**EF**♭**R**语言进行对比，该语言只是**EF**♭支持记录。对于一组方程E的类型$\tau \backslash E$，**CF**♭**R**有类型

$$\tau \backslash \{\tau_1 <: \tau_1', \ldots, \tau_n <: \tau_n'\}$$

CF♭**R** 具有以下一组类型规则。这些是 **EF**♭ 规则的直接推广，用 $<$ 替换 $=:$. $<:$ 始终在信息流的方向上。我们让 C 代表一组子类型约束。

(Hypothesis) $$\frac{}{\Gamma \vdash x : \tau \backslash \emptyset \text{ for } \Gamma(x) = \tau}$$

(Int) $$\frac{}{\Gamma \vdash n : \texttt{Int} \backslash \emptyset \text{ for } n \text{ an integer}}$$

(Bool) $$\frac{}{\Gamma \vdash b : \texttt{Bool} \backslash \emptyset \text{ for } b \text{ a boolean}}$$

(+) $$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e \ \texttt{+}\ e' : \texttt{Int} \backslash C \cup C' \cup \{\tau <: \texttt{Int}, \tau' <: \texttt{Int}\}}$$

(-) $$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e \ \texttt{-}\ e' : \texttt{Int} \backslash C \cup C' \cup \{\tau <: \texttt{Int}, \tau' <: \texttt{Int}\}}$$

(=) $$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e \ \texttt{=}\ e' : \texttt{Bool} \backslash C \cup C' \cup \{\tau <: \texttt{Int}, \tau' <: \texttt{Int}\}}$$

(If)

$$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C', \quad \Gamma \vdash e'' : \tau'' \backslash C'',}{\Gamma \vdash (\texttt{If}\ \ e\ \ \texttt{Then}\ \ e'\ \ \texttt{Else}\ \ e'') : \alpha \backslash C \cup C' \cup C'' \cup \{\tau <: \texttt{Bool}, \tau' <: \alpha, \tau'' <: \alpha\}}$$

(Function) $$\frac{\Gamma, x : \alpha \vdash e : \tau \backslash C}{\Gamma \vdash (\texttt{Fun}\ \ x\ \ \texttt{->}\ \ e) : \alpha\ \ \texttt{->}\ \ \tau \backslash C}$$

(Application) $$\frac{\Gamma \vdash e : \tau \backslash C, \quad \Gamma \vdash e' : \tau' \backslash C'}{\Gamma \vdash e\ e' : \alpha \backslash C \cup C' \cup \{\tau <: \tau'\ \texttt{->}\ \alpha\}}$$

我们未在 **EF**♭ 中看到的两个规则是 *Record* 规则和 *Projection* 规则。然而，这些规则并没有什么特别之处。

(Record) $$\frac{\Gamma \vdash e_1 : \tau_1 \backslash C_1, \ldots, \Gamma \vdash e_n : \tau_n \backslash C_n}{\Gamma \vdash \{l_1\texttt{=}e_1\texttt{;}\ \ \ldots\texttt{;}\ \ l_n\texttt{=}e_n\} : \{l_1 : \tau_1\texttt{;}\ \ \ldots\texttt{;}\ \ l_n : \tau_n\} \backslash C_1 \cup \ldots \cup C_n}$$

(Projection) $$\frac{\Gamma \vdash e : \tau \backslash C}{\Gamma \vdash e.l : \alpha \backslash \{\tau <: \{l : \alpha\}\} \cup C}$$

与 **EF**♭ 一样，这些规则几乎直接定义了类型推断过程，证明可以基本上从下往上构建。

完整的类型推断算法如下。给定一个表达式 e，

1. 使用上述类型规则证明 $\vdash\ e : \tau \backslash C$，这样的证明总是存在的。

2. 通过以下描述的方式计算闭包来扩展 C。

3. 如果 C 立即不一致，则引发一个 `typeError`。

4. 检查 C 是否存在循环，如下所述。如果 C 包含循环，则引发一个 `typeError`。

5. 推断出的类型是 $e : \tau \backslash C$。

算法计算 C 的闭包和进行环检测是 **EF**♭ 算法的明显推广。闭包的计算如下。

1. 对于 $\{l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots, l_m : \tau_m\} <: \{l_1 : \tau'_1, \ldots, l_n : \tau'_n\}$ 中的每个约束，向 C 中添加 $\tau_1 <: \tau'_1$，... ，$\tau_n <: \tau'_n$。2. 对于 C 中的每个约束 τ_0 `->` $\tau'_0 <: \tau_1$ `->` τ'_1，向 C 中添加 $\tau_1 <: \tau_0$ 和 $\tau'_0 <: \tau'_1$。3. 对于约束 $\tau_0 <: \tau_1$ 和 $\tau_1 <: \tau_2$，通过传递性向 C（中添加 $\tau_0 <: \tau_2$。4. 重复，直到无法再添加约束。

约束集是 {v*}，如果 $\tau <:\ \tau'$ 和 τ 和 τ' 是不同类型的类型（函数和记录，`Int` 和函数等），或者两个记录按 $<$ 排序：右边的记录有一个左边的记录没有的字段。

要执行 C 中的循环检测，我们使用以下算法。定义一个有向图 G，其中节点是 C 中的类型变量。如果存在一个方程 `'a` $<:\ \tau'$ 在 C 中，并且 `'b` 出现在 τ' 中，则从 `'a` 节点到一个 `'b` 节点存在一条边。此外，如果 $\tau' <:$ `'a` 出现在 C 中并且 `'b` 出现在 τ' 中，则从 `'b` 到 `'a` 存在一条边。当且仅当 G 有循环时，C 才有循环。

似乎我们的约束类型推理算法存在一个重大遗漏：我们从未解决过约束！然而，该算法是正确的。我们不希望解决约束的原因是任何替换都可能失去一般性。例如，考虑一个约束 `'a` $<:\ \tau$，以及用 `'a` 替换 τ 的可能性。这排除了 `'a` 位置是 τ 的子类型的可能性，因为替换实际上断言了 `'a` 和 τ 的相等性。用更简单的话说，我们需要将约束作为类型的一部分保留下来。这是约束类型系统的主要弱点；类型包括约束，因此难以阅读和理解。

我们在此点与等式情况有相同的缺点：目前还没有多态性。在等式情况下使用的解决方案在这里不起作用，因为它需要解决约束。

T解决方案是创建约束多态类型 esTranslated Text: es

$\forall \alpha_1, \ldots, \alpha_n.\ \tau \backslash C$

在假设 Γ 中，我们用多态类型（类型模式）替换了等价版本中的类型。这个过程相当复杂，我们不会深入探讨。约束多态类型是很好的对象类型，因为多态性对于类型继承是必需的。

Chapter 7

Concurrency

并发计算一次处理多个任务。我们假设您对并发性有一定了解，但我们将提供一个简要概述以开始。维基百科上关于并行计算的文章提供了更详细的概述。{v*}

7.1 Overview

并发执行可以大致分为三类实现类别，从松散耦合到紧密耦合。
*Distributed computation*是在没有共享内存的多台计算机上进行的计算，它们之间通过发送消息来交换数据。这些计算机之间可以解耦的程度有很大的差异。*Grid computing*是分布式计算，其中互联网是通信媒介。*Cluster computing*是通过快速局域网网络。大规模并行处理（MPP）涉及用于非常高的通信带宽的专用通信硬件。
*Distributed shared memory*是不同进程在不同的处理器上运行但通过内存总线共享一些特殊内存的情况。最后，*multithreaded computation*是多个执行线程共享单个本地内存的情况。多线程计算可以在单个（核心）CPU上运行，这意味着并发执行是通过交错两个线程的执行步骤来实现的假象，或者如果计算机有多个核心，则可以真正并发执行多线程程序。线程维基百科文章解释了线程和进程之间的区别。

所有上述模型仍然支持独立的控制焦点。这是本章研究的主要焦点——它涵盖了广泛的各种模型，它们是并发架构最优雅的形式。还有一些其他模型我们没有涉及。
*Vector processors*是可以在一步中对一个整个数组进行操作的计算机。这些架构过去被称为SIMD（单指令多数据）。*Stream processors*是向量处理器的现代版本，可以并行处理不仅仅是数组，例如稀疏数组。*GPGPU*(通用图形处理单元)是流处理器的最新版本，它作为专用图形处理器的推广而出现。最后，*FPGA's*是现场可编程电路：您可以在运行时创建自己的（并行）逻辑电路。

历史上，并发编程作为现有语言的库扩展而产生。早期模型包括UNIX套接字用于IPC和C中的进程创建。虽然一些并发可以通过对底层顺序语言的库扩展轻松编程，但并发的一些方面足够基本，因此将并发集成到编程语言中很重要。

Java、C等语言的并发标准模型如今是通过多线程实现的。多线程编程正在大规模兴起——新的CPU具有多个核心，可以同时运行多个线程。不幸的是，多线程编程也是一个即将发生的灾难——程序太难调试。问题的根源在于，在访问共享内存时，不同线程之间可能出现的交错情况太多。其中一些可能在测试中显现，但许多情况只有在部署后才会出现。

主“坏事情”在多线程编程中可能发生，包括以下内容。一个 **race condition** 是指两个操作交错并使数据处于不一致的状态。例如，一个同时更新变量的操作，其中一个线程设置了一个双精度浮点数的最高位，而另一个线程由于几乎同时访问而设置了最低位——双精度浮点数的值对任一线程来说都不是一个合理的值。**Deadlock** 是指线程可能需要等待资源释放（最初它们被锁定以防止竞态条件），并且可能陷入一个 **cycle** 的等待状态：A 等待 B，B 等待 C，C 等待 A。

竞态条件可以通过使用各种类型的锁来避免。**Monitors**是源程序中的互斥区域，在任何时刻只有一个线程可以执行监视器块。它们类似于Java中的`synchronized`关键字。一个**Semaphore**是一种低级锁定机制：在关键操作之前获取锁；如果其他人持有锁则阻塞；一旦你获得了锁，你就知道你是唯一访问的人；完成时释放它。通用信号量允许n个线程同时访问关键区域，而不仅仅是其中一个。

Atomicity 并发编程语言中的另一个关键设计概念。一个 **atomic region** 是可能与其他线程执行交织的代码区域，但你无法判断——它
always appears to have run atomically，一步之内。在你的语言设计中能获得越多的原子性，在调试中需要考虑的交织就越少，错误也就越少。Sun Java 并发教程提供了更多关于该模型的信息。

7.1.1 The Java Concurrency Model

让我们简要回顾Java的并发方法。Java实现了线程的标准概念：控制线程之间共享内存（即，每个线程都有自己的运行时栈，但只有一个堆）。使用`synchronized`关键字来声明互斥区域；它们是一种监视器。有几个变体：`synchronized`方法或代码块都可以定义。当一个同步方法/代码块正在运行时，该对象的任何其他同步方法/代码块都不能从另一个线程开始 - 任何这样的线程都必须等待当前运行的方法/代码块完成。Java的`synchronized`的优点在于它是如何*object-based*的：锁是针对对象的，这可以说是锁定的一个很好的抽象级别。然而，Java的缺点是监视器只提供互斥，

非原子性。

Java并发模型有一些其他不错的特性，我们在这里简要回顾一下。其中大部分可以在`java.util.concurrent`包中找到。

- 原子整数、浮点数等 - 在设置或从 `AtomicInteger` 等获取值时永远不会出现任何竞态条件。
- 锁 - `java.util.concurrent.locks.Lock`
- 并发集合 - 例如 `ConcurrentHashMap`，它支持并发原子性地进行添加/查找。

7.2 The Actor Model and AF♭V

我们将在此更详细地研究一种并发编程模型，即 **Actor Model**。演员是一个简单、优雅的并发编程模型。它在一定程度上是并发编程世界的“函数式编程”类比，而演员实际上非常适合函数式编程风格。Erlang编程语言[1]是一种具有这种结构的现实世界语言：一个函数式基础，上面有一个演员并发层。

演员模型最初由卡尔·休伊特在20世纪70年代的麻省理工学院开发。演员模型维基百科条目有很好的历史概述和描述，我们现在简要总结关键点。每个演员是一个自主的、分布式的代理，并且有一个不可伪造的名字。所有消息都是*asynchronous* - 发送演员从不等待接收演员的回复。消息到达的顺序是非确定性的，它们可能以与发送顺序不同的顺序到达。演员可以被视为相对于我们之前的分类的非共享内存模型：演员之间的所有通信都是通过显式消息进行的，而不是通过共享变量在公共内存中的隐式通信。除了它们局部的计算动作和发送消息之外，演员还可以创建其他演员。如果一个演员在消息到达时正忙，它将被放入消息队列 - 演员一次只处理一条消息。并发是通过许多演员可以并行处理他们的消息这一事实实现的，而不是通过单个演员的并发行为。在我们的理想化模型中，我们假设没有故障：所有消息最终都会到达（但，它们可能需要任意长的时间才能到达）。每个演员都是一个*reactive system*：通常演员是空闲的，它醒来并在有限的时间内处理一条消息，然后再次进入睡眠状态，直到另一条消息到来。这就是演员的生活，永远永远对外部事件做出反应。

演员模型最大的优点之一是内置的 *Atomicity*：多个演员可以并行运行，但任何交错运行都等同于每个演员在一个大步骤中运行所有步骤的运行。这是因为演员在处理原始消息的过程中永远不会在中间接受消息。

在这一节中，我们定义了 **AF♭V**，这是在 **F♭V** 语言之上的一个演员层。我们需要“**V**”，因为我们将会使用变体来定义消息。回想一下 **F♭V**变体类似于 OCaml 的推断变体 - `‘foo(4)` 是带有参数 4 的变体 `foo`。注意我们也可以将其视为带有参数 4 的 **message** `foo`。为了简单起见，**F♭V**变体总是恰好只有一个参数。

7.2.1 Syntax of AF♭V

AF♭V 表达式如下。

$$
\begin{array}{rcll}
v & ::= & \ldots \text{ the } \mathbf{F\flat V} \text{ values} \ldots \mid a & \textit{values} \\
e & ::= & \ldots \text{ the } \mathbf{F\flat V} \text{ expressions} \ldots \mid e \leftarrow e \mid \texttt{Create}(e, e) \mid a & \textit{expressions}
\end{array}
$$

在 a 来自一个无界的演员名称集合中。它们类似于 **F♭S** 中的 c 单元，它们不能出现在源程序中，但可以在运行时出现，并且有无限多个独特的；它们只是名称（非密码）。

AF♭V 语法 $e \leftarrow e'$ 表示消息发送；它期望 e 评估为演员名称，并将该名称作为消息发送给 e' 的值。$\texttt{Create}(e, e')$ 创建一个具有行为 e 的演员，并带有初始局部数据 e'。e 应该评估为一个函数，该函数是演员的（全部）代码。Create 返回这个新演员的（新）名称作为其结果。

某些**AF♭V**的功能包括以下内容。演员的代码只是一个函数——在演员中没有以字段形式存在的状态。在处理完每条消息后，演员进入空闲状态；它将在收到下一条消息时采取的行为是*the value at the end of the previous message send*。后一点是演员如何随时间发生变化的关键：每条消息的处理都是纯函数式的，但在最后，演员可以选择在处理下一条消息之前想要突变到哪个状态。这是将一点命令式编程混合到纯函数模型中的有趣方法。

在 **AF♭V** 中，您必须使用 **F♭V** 的 Match 语法编写自己的显式消息分发代码。与我们在 **F♭OB** 中使用的对象作为记录的方法相比，这里我们采用双重编码方法，将对象视为函数，将消息视为变体。最后，为了使演员知道自己的名称，在创建时将其传递给他们。

7.2.2 An Example

在进入操作语义之前，让我们先做一个简单的例子。这里有一个演员，它接收到一个启动消息，然后从其初始值倒数到0。在这里，我们使用术语Y来表示Y-组合子，而术语表示一个无重要性的值（就像之前的空记录{}一样）。

```
Fun myaddr ->
  Y (Fun this -> Fun localdata -> Fun msg ->
      Match msg With
         ‘main(n) -> myaddr <- ‘count(n); this(_)
       | ‘count(n) -> If n = 0
           Then this(_)
           Else
              myaddr <- ‘count(n-1);
              this(_) /* set to respond to next message */
     )
```

这里是一个代码片段，另一个演员可以使用它来启动具有上述行为的新演员并使其开始。假设我们缩写的上述代码为 CountTenBeh。

```
Let x = Create(CountTenBeh,_)  /* _ is the localdata - unused */
  In x <- ‘main(10)
```

这里是一种不同的倒计时方法，其中 localdata 字段持有值，并且它不在消息中。

```
Fun myaddr ->
  Y (Fun this -> Fun localdata -> Fun msg ->
      Match msg With
        ‘count(_) ->
           If localdata = 0
             Then _
             Else
                myaddr <- ‘count(_);
                this(localdata - 1) /* set to respond to next msg */
     )
```

假设上述代码被缩写为 CountTenBeh2；使用它则是

```
Let x = create(CountTenBeh2,10)  /* 10 is  the localdata */
  In x <- ‘count(_)
```

后者是给演员提供本地数据的正确方式 - 在前者中，必须在每条消息中转发计数器值。

这是第一个示例的另一个用法片段：

```
Let x = create(CountTenBeh,_)
  In x <- ‘main(10); x <- ‘main(5)
```

在这种情况下，演员 x 将并行且独立地从 10..0 和 5..0 进行倒计时 - 这些计数也可能以随机方式交织。对于第二个示例，一个类似的情况可能是：

```
Let x = create(CountTenBeh2,10)
  In x <- ‘count(_); x <- ‘count(_)
```

这只是让一条计数消息排队，因为演员每次得到一条就会发送一条新的，直到0，所以最终会留下一条剩余的消息。

7.2.3 Operational Semantics of Actors

操作语义分为两层：演员内部的局部计算，这与 $\mathbf{F}\flat\mathbf{V}$ 并无太大差异，以及所有演员的并发全局步进。让我们从后者开始。

一个 *global state* G 是活动演员和发送消息的“汤”：

$$G :: \quad \{\langle a, v\rangle \mid a \text{ 是一个演员名称，} v \text{ 是其行为}\} \cup \{[a \leftarrow v] \mid a \text{ 是一个演员名称，} v \text{ 是发送给 } a\} \text{ 的消息}$$

演员系统可以永远运行，没有最终值的观念。因此，系统执行 *small steps* 的计算：

$$G_1 \to G_2 \to G_3 \to \ldots$$

– 这表示 *one step* 的计算；在每一步中，汤中的 *one* 个演员完全处理了汤中的 *one* 消息。我们将在下面定义这个 $\to$ 关系。$\to^*$ 是 $\to$ 的自反、传递闭包 – 演员计算的多个步骤。一般来说，这会无限继续，因为演员系统可能不会终止。演员系统运行的最后意义是这个无限的状态流。

7.2.4 The Local Rules

让我们从局部规则开始。它们在 **F♭V** 操作语义中通过类似的关系 $\Rightarrow$ 定义，但局部执行还包含它们创建的演员和发送的消息的 *side effects*。我们将使任何这样的副作用都成为此箭头关系上的 *labels*。因此，我们使局部计算关系成为 $\overset{S}{\Longrightarrow}$ – S 这里是效果之汤。S 实际上与一个 G 的句法形式相同：它包含两种元素，$[a \leftarrow v]$ 指示本地演员在其执行期间发送给 a 的消息 v，以及 $\langle a, v\rangle$ 指示它创建的新演员，名为 a，具有行为（主体）v。

现在让我们看看 $\overset{S}{\Longrightarrow}$ 的操作语义规则。大多数规则是对相应的 **F♭V** 规则的细微修改；我们只给出 `+` 规则，以展示如何调整现有的 **F♭V** 规则以添加潜在的副作用 S：

(`+` Rule)
$$\frac{e_1 \overset{S}{\Longrightarrow} v_1, \quad e_2 \overset{S'}{\Longrightarrow} v_2 \text{ where } v_1, v_2 \in \mathbb{Z}}{e_1 \texttt{ + } e_2 \overset{S \cup S'}{\Longrightarrow} \text{the integer sum of } v_1 \text{ and } v_2}$$

自上文的 e 或 e' 在理论上可以各自创建演员或发送消息，因此我们需要将它们的效果附加到最终结果中。这些效果类似于状态，它们在旁边。与 **F♭S** 的一项主要区别是这里的效果是“只写”的——它们以任何方式都不会改变局部计算的走向，它们只是被吐出来。从这个意义上讲，局部演员计算仍然有效。

这里是有发送规则：

(`Send` Rule)
$$\frac{e_1 \overset{S}{\Longrightarrow} a, \quad e_2 \overset{S'}{\Longrightarrow} v}{e_1 \leftarrow e_2 \overset{S \cup S' \cup \{[a \leftarrow v]\}}{\Longrightarrow} v}$$

主要后果是将消息 $[a \leftarrow v]$ 添加到汤中。（这里的返回结果 v 大部分无关紧要，消息发送的目标是添加副作用。）

最后，这是创建规则：

$$\textit{(Create Rule)} \qquad \frac{e_1 \stackrel{S}{\Longrightarrow} v_1, \quad e_2 \stackrel{S'}{\Longrightarrow} v_2, \quad v_1\ a\ v_2 \stackrel{S''}{\Longrightarrow} v_3}{\texttt{Create}(e_1, e_2) \stackrel{S \cup S' \cup S'' \cup \{\langle a, v_3 \rangle\}}{\Longrightarrow} a, \text{ for } a \text{ a fresh actor name}}$$

这次返回的结果很重要 - 它是新演员的名字。$v_1\ a\ v_2$的运行将演员的自身名称和初始值传递给它以初始化它，因此v_1需要是一个接受这些参数a和v_2的柯里化函数(实际上它需要一个接受三个参数的柯里化函数，因为稍后消息也将作为参数传递；很快就会详细介绍)。

7.2.5 The Global Rule

最后但同样重要的是，这里是一个关于整个汤处理中一个演员的全局单步规则的完整消息：

$$(G \cup \{[a \leftarrow v']\} \cup \{\langle a, v \rangle\}) \rightarrow (G \cup \{\langle a, v'' \rangle\} \cup S) \text{ if } (v\ v' \stackrel{S}{\Longrightarrow} v'')$$

这是唯一的全球规则。它将一个演员与全球汤中为其预定的消息相匹配，使用本地语义来运行该演员（在隔离状态下），并将所有S（包含由该演员运行发送的所有消息以及由该演员运行创建的任何新演员）重新投入汤中。全局演员运行只是此规则的重复应用。注意，演员行为从v变为v''，这是此运行的结果。

到 测试这些规则，您可以运行示例程序 bove.

7.2.6 The Atomicity of Actors

上述语义具有执行原子的演员：每个演员独立运行至完成。然而，可以创建一种不同的语义，其中多个演员并行运行，因为演员在其执行中是完全局部的，这两种语义应该可以证明是等价的。

Chapter 8

Compilation by Program Transformation

本章的目标是通过编写一个 **F♭SR** 编译器来理解编译背后的核心概念。编译器是一项重要的技术，因为编译器生成的代码比解释代码快几个数量级。至少95%的生产软件运行的是编译代码。今天的编译是一个非常复杂的过程：编译器对程序进行多次遍历，将源代码转换为目标代码，并执行许多复杂的优化转换。我们本章的目标是理解编译背后的最基本概念：如何将高级程序映射到机器代码。

我们将概述将 **F♭SR** 编译到非常有限的 C 子集（“*pseudo-assembly*”）的过程。读者应该能够通过填补我们留下的空白来实现这个编译器。编译器使用一系列 *program transformations* 来表达编译过程。这些程序转换将 **F♭SR** 程序映射到等效的 **F♭SR** 程序，逐个去除高级特性。特别是，我们的编译器将以下转换依次应用于一个 **F♭SR** 程序：

1. 封闭转换
2. A-翻译
3. 函数提升

在程序经过这些转换后，我们得到了一个接近机器语言结构的 **F♭SR** 程序。最后一步是将这个原始的 **F♭SR** 程序翻译成 C 语言。

真实的生产编译器，如 `gcc` 和 Sun 的 `javac`，不使用转换过程，主要是因为编译本身的速度太慢。实际上，通过转换可以生成非常好的代码。SML/NJ ML 编译器采用转换方法 [5]。此外，大多数生产编译器将程序转换为一个既不是源语言也不是目标语言的中间形式（“*intermediate lan- guage*”），并对这个中间代码进行大量的优化转换。几本教科书详细介绍了编译技术 [6, 4]。

我们的主要目标，与生产编译器不同，是 *understanding*：欣赏高级和低级代码之间的差距，以及如何弥合这些差距。我们定义的每个转换都弥合一个差距。程序转换本身就很有趣，因为它们可以让我们深入了解 **F♭SR** 语言。优化，尽管是编译中的核心主题，但超出了本书的范围。我们的重点是编译高阶语言，而不是 C/C++；一些问题是相同的，但其他问题是不同的。此外，我们的可执行文件将不会尝试捕获运行时类型错误或回收未使用的内存。

期望的 **soundness property** 对于每个 **F♭SR** 程序翻译是：翻译前后的程序具有相同的执行行为（在我们的情况下，终止和相同的数值输出，但在一般情况下具有相同的输入/输出行为）。请注意，翻译输出的程序不一定在操作上等同于原始程序。

F♭SR 变换现在按照它们在源程序中应用的顺序进行说明。

8.1 Closure Conversion

闭包转换是一种消除函数中非局部变量的转换。例如，`x` 在 `Fun y -> x * y` 中是一个 **nonlocal variable**：它不是参数，并在函数体中使用。通过闭包转换，可以移除所有这样的非局部变量，得到一个等效的程序，其中所有在函数中使用的变量都是函数的参数。C 和 C++ 有全局变量（Java 通过静态字段也是如此），但全局变量不是有问题的非局部变量。必须移除的是那些作为其他函数参数的变量，其中函数定义已经被 *nested*。C 和 C++ 没有这样的非局部变量问题，因为函数定义不能嵌套。在 Java 中，内部类是嵌套类定义，也存在必须解决的关于非局部变量的问题。

Consider 例如以下柯里化加法函数 动作。

```
add = Fun x -> Fun y -> x + y
```

在内部 `Fun y` 的体 `x + y` 中，`x` 是一个非局部变量，而 `y` 是该函数的局部变量。

现在，我们提出问题，`add 3` 应该返回什么？让我们考虑一些明显的选择：

- `Fun y -> x + y` 因为变量 `x` 将未定义，我们不知道它的值是 `3`，所以这没有意义。
- `Fun y -> 3 + y` 这似乎是正确的事情，但它相当于代码替换，编译器无法做到这一点，因为编译后的代码必须是不可变的。

由于这些想法都不起作用，需要新的东西。解决方案是返回一个 *closure*，一个由函数和一个 *environment* 组成的对，该 *environment* 记录任何非局部变量的值以供以后使用：

```
(Fun y -> x + y, { x |-> 3 })
```

函数定义现在是闭包定义；要调用此类函数需要一个新进程。**Closure conversion** 是一个全局程序转换，它在语言本身中显式执行此操作。Fun 值被定义为 *closures*，即函数和一个记住非局部变量值的环境的元组。当调用定义为闭包的函数时，我们必须显式传递闭包中的非局部环境，以便它可以用来查找非局部变量的值。

翻译通过示例引入。考虑上述 add 中的内 `Fun y -> x + y` 翻译为闭包

```
{ fn = Fun yy -> (yy.envt.x) + (yy.arg);
  envt = { x = xx.arg } };
```

让我们看看翻译的细节。闭包被定义为记录形式的元组

```
{ fn = Fun ...; envt = {x = ...; ...}}
```

由原始函数（fn字段）和非局部环境（envt字段）组成，后者本身是一个记录。在非局部环境`{ x = xx.arg }`中，x是原始函数中的非局部变量，其值通过同名*label*、x记录在此记录中。所有此类非局部变量都放置在环境中；在此示例中x是唯一的非局部变量。

函数曾经使用参数 y 的现在被修改为使用名为 yy (的参数（原始变量名，重复两次) ）。我们实际上不需要更改名称，但这有助于理解，因为变量的角色已改变：新的参数 yy 预期是一个形式为 `{ envt = ..; arg = ..}` 的记录，将环境和原始参数传递给函数。

如果 yy 确实是在函数调用时的这样一个记录，那么在函数体内，我们可以使用 `yy.envt.x` 来访问原本在原始函数体内是非局部变量 x 的内容，以及使用 `yy.arg` 来访问原本是函数参数 y 的内容。

整个 add 函数通过转换两个函数进行闭包转换：

```
add' = {
   fn = Fun xx -> {
      fn = Fun yy -> (yy.envt.x) + (yy.arg);
      envt = { x = xx.arg }
   };
   envt = {}
 }
```

外部 `Fun x -> ...` 可能不需要进行闭包转换，因为它没有非局部变量，但为了统一，最好将所有函数都进行闭包转换。

Translation of function application 在上述示例中，没有应用，因此我们没有定义如何应用闭包转换后的函数。应用必须改变，因为函数现在实际上表示为记录。闭包转换后的函数调用 `add 3` 然后 *pass in the environment*，因为调用者需要知道它：

```
(add'.fn)({ envt = add'.envt; arg = 3})
```

因此，我们首先从闭包中提取函数部分，`(add'.fn)`，然后传递一个由闭包中提取的环境 `add'.envt` *also* 和参数，`3` 组成的记录。`add 3 4` 的翻译采用上述结果，应该评估为一个函数闭包 `{ fn = ...; envt = ...}`，并对其执行相同的技巧应用 `4`：

```
Let add3' = (add'.fn){ envt = add'.envt; arg = 3 } In
    (add3'.fn){ envt = add3'.envt; arg = 4}
```

结果将是 `12`，与原始结果相同，证实了在这种情况下翻译的可靠性。在一般应用中，转换如下。在函数调用时，闭包中记住的环境被传递到闭包中的函数。因此，对于上面的 `add' 3` 闭包，`add3'`，当它后来应用于例如 `7` 时，`envt` 将知道它是要添加到 `7` 中的 `3`。

One more level of nesting 闭包转换在考虑函数定义的另一个嵌套级别时甚至稍微复杂一些，例如

```
triadd = Fun x -> Fun y -> Fun z -> x + y + z
```

`Fun z`需要获取`x`，由于那个`Fun z`是在`Fun y`内部定义的，`Fun y`必须作为中介从最外层函数`x`传递。这里是翻译。

```
triadd' = {
   fn = Fun xx -> {
      fn = Fun yy -> {
         fn = Fun zz ->
            (zz.envt.x) + (zz.envt.x) + (zz.arg);
         envt = { x = yy.envt.x; y = yy.arg }
      };
      envt = { x = xx.arg }
   };
   envt = {}
}
```

一些观察可以得出。内部 `z` 函数有非局部变量 `x` 和 `y`，因此它们都需要在其环境中；`y` 函数不直接使用非局部变量，但

它具有非局部 x，因为其中的函数 Fun z 需要 x。因此，其非局部 envt 中包含 x。Fun z 可以从 y 的环境中获取 x 到其环境中，如 yy.envt.x。因此，Fun y 作为中间人将 x 获取到 Fun z。

8.1.1 The Official Closure Conversion

考虑到前面的例子，我们可以写出官方的闭包转换翻译。我们将使用符号 clconv(e) 来表示闭包转换函数，如下定义（此代码非正式；它使用具体的 **F♭SR** 语法，例如在记录的情况下看起来像 Caml 语法）。

Definition 8.1 (闭包转换)**.**

1. clconv(x) = x (*变量*) 2.
clconv(n) = n (*数字*) 3.
clconv(b) = b (*布尔值*)

4. clconv(Fun x -> e) = 设 x，x1，...，xn 为 e 中的精确自由变量，结果是 **F♭SR** 表达式

```
{ fn = Fun xx -> SUB[clconv(e)];
   envt = { x1 = x1; ...; xn = xn } }
```

在 SUB[clconv(e)] 是对 clconv(e) 进行替换 (xx.envt.x1)/x1，...，(xx.envt.xn)/xn 和 (xx.arg)/x 后的结果，但在 clconv(e) (内部 Fun 的 *not* 替换后停止替换，直到遇到一个 Fun)。

5. clconv(e e') = Let f = clconv(e) In (f.fn){ envt = f.envt; arg = clconv(e')}

6. clconv(e op e') = clconv(e) op clconv(e') 对于语言中的所有其他运算符（在其他所有运算符中翻译为 *homomorphic*）。在每种情况下都很清楚，也许除了记录，我们只是为了确保。…

7. clconv({ l1 = e1; ...; ln = en }) = { l1 = clconv(e1);...; ln = clconv(en) }

对于上述示例，clconv(add) 是 add'。期望的稳定性结果是

Theorem 8.1. *Expression e computes to a value if and only if* clconv(e) *computes to a value. Additionally, if one returns numerical value n, the other returns the same numerical value n.*

闭包转换生成没有非局部变量的程序，因此所有函数都可以像C语言一样在“顶层”定义。事实上，在下面的第8.3节中，我们将明确将所有内部函数定义扩展到顶层。

8.2 A-Translation

机器语言程序是原子指令的线性序列；每条指令最多只能进行一次算术运算，因此需要许多指令来评估复杂的算术（和其他）表达式。A-转换通过将基于表达式的程序重新表述为一系列原子操作，弥合了基于表达式的程序与线性、原子指令之间的差距。我们将其表示为一系列 Let 语句，每个语句执行一个原子操作。

从算术表达式的例子中，这个想法应该是显而易见的。例如

```
4 + (2 * (3 + 2))
```

我们的 **F♭SR** 解释器在这样表达式上定义了一个评估顺序的树形概念。通过使用 Let 将解释器评估程序时按顺序提取的部分进行因式分解，可以使此程序上的评估顺序明确线性化。

```
Let v1 = 3 + 2 In
Let v2 = 2 * v1 In
Let v3 = 4 + v2 In
    v3
```

此程序应与原始程序给出相同的结果，因为我们所做的只是使计算顺序更加明显。注意这与3地址机器代码的相似之处：它是一系列直接应用于变量或常数的原子操作的线性序列。v1等变量是*temporaries*；在机器代码中，它们通常最终被分配到寄存器中。这些临时变量在此处没有被重用（重新分配）。在**F♭SR**中无法进行类似寄存器的编程，但这是真实3地址中间语言的工作方式。在最终机器代码生成时，临时变量被重用（通过*register allocation*策略）。

我们实际上将使用一种更朴素（但统一）的翻译，该翻译首先将常数和变量分配给其他变量：

```
Let v1 = 4 In
Let v2 = 2 In
Let v3 = 3 In
Let v4 = 2 In
Let v5 = v3 + v4 In
Let v6 = v2 * v5 In
Let v7 = v1 + v6 In
    v7
```

这个简单翻译与操作语义紧密对应——e ⇒ v的每个推导节点在上面的都是Let。

Exercise 8.1. 写出操作语义推导，并将其结构与上述内容进行比较。

这种翻译的优点是每个操作都在变量之间进行。在上面的前一个例子中，`4+v2` 对于某些机器语言可能不够底层，因为可能没有立即加法指令。一个简单的优化就是避免为常量创建新的变量。然而，我们在这个阶段更注重正确性而不是效率。`Let` 是 **F♭SR** 中的一个原始值——这并非绝对必要，但如果 `Let` 用应用来定义，A-翻译的结果将更难操作。

接下来考虑使用高阶函数的 **F♭SR** 代码。

```
((Fun x -> Fun y -> y)(4))(2)
```

函数首先需要计算应用于 2 的函数。我们也可以通过 `Let` 明确这一点：

```
Let v1 = (Fun x -> Fun y -> y)(4) In
Let v2 = v1(2) In
    v2
```

A-翻译将与算术示例一样，对所有操作进行完全线性化：

```
Let v1 =
   (Fun x ->
      Let v1' = (Fun y -> Let v1'' = y in v1'') In v1')
   In
Let v2 = 4 In
Let v3 = v1 v2 In
Let v4 = 2 In
Let v5 = v3 v4 In
    v5
```

所有形式的 **F♭SR** 表达式都可以以类似的方式线性化，*except* `If`：

```
If (3 = x + 2) Then 3 Else 2 * x
```

可以转换成类似以下的形式

```
Let v1 = x + 2 In
Let v2 = (3 = v1) In
If v2 Then 3 Else Let v1 = 2 * x In v1
```

但是 If 仍然有一个无法线性化的分支。机器代码中的分支可以通过标签和跳转进行线性化，这是 **F♭SR** 缺少的表达形式。上述转换后的示例仍然“足够接近”机器代码：我们可以将其实现为

```
v1 := x + 2
v2 := 3 = v1
BRANCH v2, L2
L1:  v3 := 3
GOTO L3
L2:  v4 := 4
L3:
```

8.2.1 The Official A-Translation

我们定义A-平移为一个Caml函数，`atrans(e) :  term -> term`。我们总是将对闭包转换的结果应用A-平移，但这个事实现在无关紧要。

A-翻译的中间结果是元组列表

```
[(v1,e1); ...; (vn,en)] :  (ide * term) list
```

这是用来表示的

```
Let v1 = e1 In ... In Let vn = en In vn ...
```

但是，在Caml中，由于声明列表将在翻译时一起附加，因此这种形式更容易操作。当编写编译器时，程序员可能希望或可能不希望使用这种中间形式。编写直接在Let表示上工作的函数并不困难。

我们现在绘制核心原语的翻译。假设以下辅助函数已被定义：

- `newid()` 每次调用都返回一个全新的 **F♭SR** 变量，
- 函数 `letize`，它将元组列表形式转换为实际的 Let 形式，并且
- `resultId`, 对于列表 `[(v1,e1); ...; (vn,en)]` 返回结果标识符 vn。

Definition 8.2 (翻译).

```
let atrans e = letize (atrans0 e)

and atrans0(e) = match e with
  (Var x) -> [(newid(),Var x)]
| (Int n) -> [(newid(),Int n)]
```

```
| (Bool b) -> [(newid(),Bool b)]
| Fun(x,e) -> [(newid(),Fun(x,atrans e)]
| Appl(e,e') -> let a = atrans0 e in let a' = atrans0 e' in
    a @ a' @ [(newid(),Appl(resultId a,resultId a')]

(* all other D binary operators + - = AND etc. of form
 * identical to Appl
 *)

| If(e1,e2,e3) -> let a1 = atrans0 e1 in
    a1 @ [(newid();If(resultId a1,atrans e2,atrans e3)]

(* ... *)
```

在A-翻译的末尾，代码在解释器中的运行方式都是“线性”的，而不是作为一棵树。机器码也是线性排序的；我们正越来越接近机器码。

Theorem 8.2. *A-translation is sound, i.e.* `e` *and* `atrans(e)` *both either compute to values or both diverge.*

尽管我们只部分定义了A-翻译，但**F♭SR** (记录、参考单元格)的额外语法并不会带来任何重大复杂性。

8.3 Function Hoisting

截至目前，我们已经定义了一个编译器的前端，它依次执行闭包转换和A-转换：

```
let atrans_clconv e = atrans(clconv(e))
```

在经过这两个阶段之后，函数将不再有非局部变量。因此，我们可以将程序体内的所有函数 *hoist* 移至程序的开头。这使得程序结构更符合 C（和机器）代码。由于我们的最终目标是 C，因此将所有函数提升后的剩余代码制成 `main` 函数。一个 `hoist` 函数执行这种转换。非正式地说，这个操作相当简单：例如，取

```
4 + (Fun x -> x + 1)(4)
```

将其替换为

```
Let f1 = Fun x -> x + 1 In 4 + f1(4)
```

通常，我们将所有函数提升到代码的前面，并通过 `Let` 给它们命名。如果没有自由变量在函数中，这种转换始终是合理的

body，闭包转换保证的属性。我们将以简单迭代（但效率低下）的方式定义此过程：

Definition 8.3 (函数提升)**.**

```
let hoist e =
   if e = e1[(Fun ea -> e')/f] 对于某些 e1 具有自由
   度 f，并且 e' 本身不包含函数（即 Fun ea -> e' 是最
   内层函数）then
   else e       Let f = (Fun ea -> e') In hoist(e1)
```

此函数首先提升最内层函数。如果函数不是按最内层优先提升，提升定义中仍将存在一些嵌套函数。因此，提升顺序很重要。

上述提升的定义简洁，但效率太低。可以使用一次遍历的实现，递归地将函数替换为变量并将它们累积在列表中。这种实现留作练习。生成的程序将具有以下形式

```
Let f1 = Fun x1 -> e1 In
...
Let fn = Fun xn -> en In
e
```

每个 $e, e_1, \ldots e_n$ 不包含函数常数。

Theorem 8.3. *If all functions occurring in expression* `e` *contain no nonlocal variables, then* `e` $\cong$ `hoist(e)`*.*

这个定理可以通过以下引理的迭代应用来证明：

Lemma 8.1.

$e1[(\textit{Fun}\ x\ \texttt{->}\ e')/f] \cong$
$(\textit{Let}\ f\ \texttt{=}\ (\textit{Fun}\ x\ \texttt{->}\ e')\ \textit{In}\ e_1$

provided e' *contains at most the variable* x *free.*

我们最后将程序转换为

```
Let f1 = Fun x1 -> e1 In
    ...
      fn = Fun -> Fun xn -> en In
      main = Fun dummy -> e In
      main(0)
```

所以，这个程序几乎就是一系列函数的集合，主体只是调用 `main`。这使得程序更接近 C 程序，C 程序不过是一系列函数的集合，`main` 在程序开始时隐式调用。

`Let Rec` 定义也需要提升到顶级；它们的处理方式类似，将留作练习。

8.4 Translation to C

我们现在准备好将其翻译成C语言。为了总结到目前为止的内容，我们有

```
let hoist_atrans_clconv e = hoist(atrans(clconv(e)))
```

我们已经完成了在 **F♭SR** 内可能的所有翻译。程序确实看起来更像机器码：所有函数都在顶层声明，每个函数体都由一系列原子指令的线性序列组成（除了 `If`，它是一个分支）。仍然有一些比机器码更复杂的事情：记录仍然是隐式分配的，函数调用是原子的，不需要推送参数。由于 C 语言内置了函数调用，记录是需要填补的唯一重大差距。

翻译涉及两个主要操作。

1. 将每个函数映射到C函数

2. 对于每个函数 离子体，将每个原子元组映射到原始C结构体 声明。

Atomic Tuples 在给出翻译之前，我们列举出所有由A-翻译产生的`Let`变量赋值的可能右侧（在以下`vi`，`vj`，`vk`和`f`是变量）。这些被称为**atomic tuples**。

Fact 8.1 (原子元组)**.** ***F♭SR*** *programs that have passed through the first three phases have function bodies consisting of tuple lists where each tuple is of one of the following forms only:*

1. `x` 用于变量 `x` 2. `n` 用
于数字 `n` 3. `b` 用于布尔
值 `b` 4. `vi vj` (应用) 5.
`vj + vk` 6. `vj - vk` 7.
`vj And vk` 8.
`vj Or vk`

9. `Not vj` 10. `vj = vk` 11. `Ref vj` 12. `vj := vk` 13. `!vj` 14. `{ l1 = v1; ...; ln = vn }` 15. `vi.l` 16. `If vi Then tuples1 Else tuples2` 其中 `tuples1` 和 `tuples2` 是 `Then` 和 `Else` 体变量的赋值列表。

函数应该都已经提升到顶部，所以在元组中不会有这些函数。观察发现，一些记录的使用来自原始程序，而其他的是在闭包转换过程中添加的。我们可以把它们都看作是常规记录。我们现在需要做的就是为上述每个元组生成代码。

8.4.1 Memory Layout

在编写任何编译器之前，需要为运行时对象提供一个固定的内存布局方案。由于对象可以从许多不同的程序点进行读写，如果给定对象的读写协议不统一，代码将无法正常工作！因此，事先仔细设计策略非常重要。像我们这样的简单编译器将使用简单的方案，但为了效率，最好使用更复杂的方案。

让我们简要地考虑一下C语言中内存的布局。值可以以几种不同的方式存储：

- 在寄存器中：这些必须是临时的，因为寄存器通常每个函数/方法都是局部的。此外，寄存器的大小只有一个字（或者几个字，对于浮点数）所以不能直接存储数组或结构体。
- 在函数的运行时栈中 *activation record*。然后该值被引用为从栈指针（它本身在寄存器中）某个固定偏移量的内存位置。
- 在固定内存位置（全局变量）。
- 在动态分配（`malloc`’ed）内存位置（在堆上）。

图8.1展示了我们正在处理的内存整体模型。为了更详细地阐述变量栈存储，这里有一些C伪代码，以向您展示栈指针`sp`的使用方法。

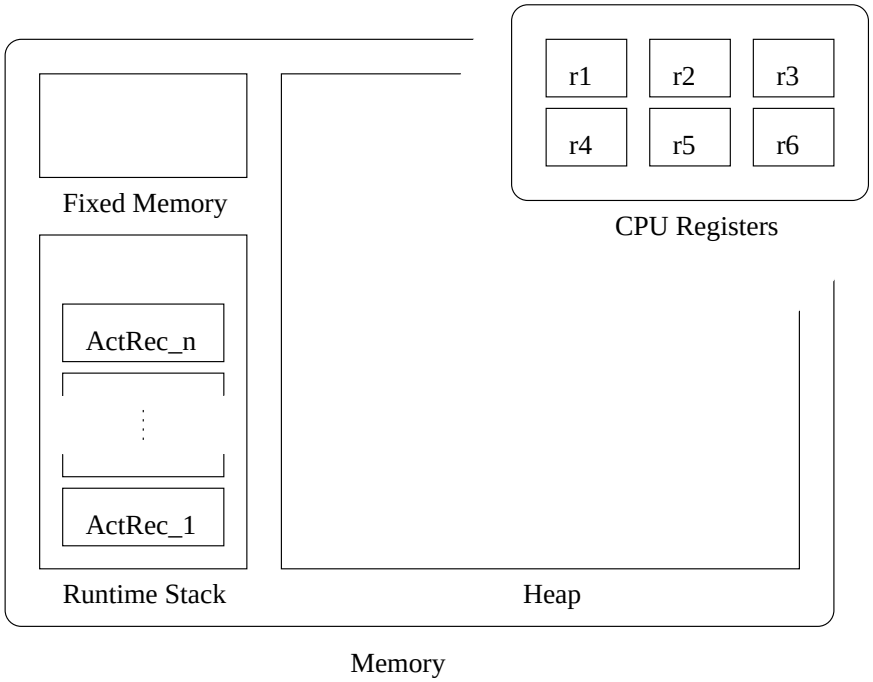

r1
r2
r3
r4
r5
r6
CPU Registers
Fixed Memory
ActRec_n
ActRec_1
Runtime Stack
Heap
Memory

图8.1：我们的记忆模型

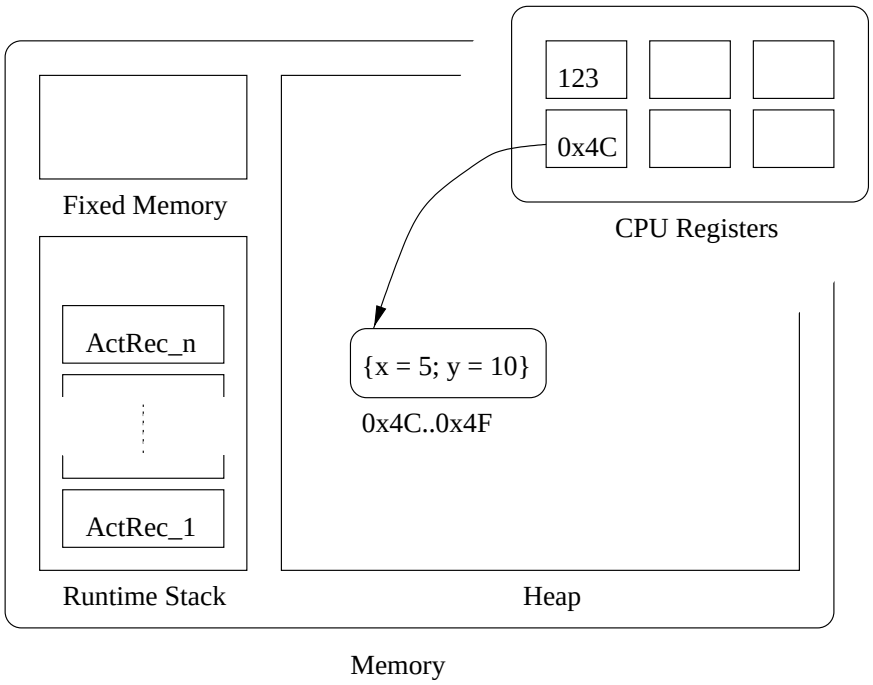


图8.2：装箱值与非装箱值。整数值 123 以非装箱值存储，而记录 {x=5; y=10} 以装箱值存储。

```
register int sp; /* compiler assigns sp to a register */
*(sp - 5) = 33;
printf("%d", *(sp - 5));
```

堆栈存储的实体也是临时的，因为当函数/方法返回时，它们将成为垃圾。

另一个重要问题是是否选择 **box** 或 **unbox** 值。

Definition 8.4. *A register or memory location v_i's value is stored* boxed *if v_i holds a pointer to a block of memory containing the actual value. A variable's value is* unboxed *if it is directly in the register or memory location v_1.*

图8.2说明了有框和无框值之间的差异。

对于多词实体，如数组，以未装箱的形式存储意味着变量直接持有指向序列中第一个单词的指针。为了阐明上述概念，我们回顾C的内存布局约定。变量可以声明为全局变量、寄存器（寄存器指令是仅将变量放入寄存器的请求）或调用栈上的变量；所有在函数内部声明的变量都保留在栈上。直接持有 ints、floats、structs 和数组的变量都是未装箱的。（例如：`int x; float x; int arr[10]; snork x` 对于 `snork` a `struct` 。）不存在直接持有函数的变量；C中的变量只能持有 *pointers* 到函数的引用。这是可能的

将“v = f”写入C语言中，其中f是一个先前声明的函数，而不是“v = &f”，但这实际上是因为前者是后者的语法糖。函数指针实际上是指向函数代码起始位置的指针。C语言中的装箱变量是显式声明的，作为指针变量。（例如：
int *x; float *x; int *arr[10]; snork *x 用于 snork 一个 struct。）所有malloc化结构都必须存储在指针变量中，因为它们是装箱的：变量不能直接是堆实体。变量是静态的，而堆是动态的。

这里是一个简单的C程序和Sun SPARC汇编输出的示例，它对这些概念给出了一些印象派的想法：

```
int glob;
main()
{
   int x;
   register int reg;
   int* mall;
   int arr[10];

   x = glob + 1;
   reg = x;
   mall = (int *) malloc(1);
   x = *mall;
   arr[2] = 4;
/* arr = arr2; --illegal:  arrays are not boxed */
}
```

在汇编语言中，%o1 是一个寄存器，[%o0] 表示间接引用，[%fp-24] 表示从帧指针寄存器 %fp 减去 24 并间接引用。上述 C 代码的汇编表示如下。

```
main:
     sethi   %hi(glob), %o1
     or      %o1, %lo(glob), %o0 /* load global address glob into %o0 */
     ld      [%o0], %o1    /* dereference */
     add     %o1, 1, %o0   /* increment */
     st      %o0, [%fp-20] /* store in [%fp-20], 20 back from fp -- x */
                           /* x directly contains a number,
                           /* not a pointer */
     ld      [%fp-20], %l0 /* %l0 IS reg (its in a register directly) */
     mov     1, %o0
     call    malloc, 0     /* call malloc.  resulting address to %o0 */
     nop
     st      %o0, [%fp-24] /* put newspace location in mall [%fp-24] */
     ld      [%fp-24], %o0 /* load mall into %o0 */
     ld      [%o0], %o1    /* this is a malloced structure -- unbox. */
     st      %o1, [%fp-20] /* store into x */
     mov     4, %o0
     st      %o0, [%fp-56] /* array is a sequence of memory on stack  */
```

```
.LL2:
     ret
     restore
```

我们的内存布局策略更类似于Java或ML这样的高级语言。Java JVM使用特定的、固定的内存布局方案：所有对象引用都是指向堆位置的包装指针；原始类型`bool`、`byte`、`char`、`short`、`int`、`long`、`float`和`double`保持未包装。由于Java数组是对象，它们也保持包装。Java中没有引用(`&`)或解引用(`*`)运算符。这些操作是隐式发生的。

Memory layout for our F♭SR compiler **F♭SR** 是（主要）Caml子集，因此其内存管理也比C内存更隐式。在我们的编译器中，我们将使用一个简单、统一的方案，在精神上与Java相似：Box `Refs`、记录和函数值，但保持布尔值和整数未装箱。此外，就像在C（和Java）中一样，所有局部函数变量都将分配在栈上，并通过栈指针的偏移来访问。我们将通过实现**F♭SR**局部变量作为C局部变量来实现这一点，这些变量将由C编译器在栈上分配。

由于一个 `Ref` 仅仅是一个可变位置，因此可能看起来没有将其装箱的任何理由。然而，如果一个函数返回一个 `Ref` 作为结果，并且它没有被装箱，那么它就会在栈上分配，从而被释放。以下是一个反映此问题的示例：

```
Let f = (Fun x -> Ref 5) In !f(_) + 1
```

如果 `Ref 5` 存储在栈上，在返回后可能会被清除。我们元组（`vi` 变量）中所有由 `Let` 定义的实体都可以在寄存器或调用栈上：由于词法作用域，这些变量都没有在函数外部直接使用，并且它们不直接包含函数返回后应该保留的值。为了效率，它们都可以声明为 `register Word` 变量：

```
register Word v1, v2, v3, v4, ...;
```

这个简单方案的另一个优点是每个变量只存储一个字节数据，因此我们不需要跟踪变量存储了多少数据。这个方案效率不高，真正的编译器会进行显著优化。一个例子是 `Ref`，已知它们不会逃离函数，可以取消装箱并在栈上分配。

所有剩下的就是制定一个方案来编译上述每个原子元组，然后我们就完成了。记录是最困难的，所以我们将在写出完整翻译之前考虑它们。

Compiling untyped records 回忆起我们讨论记录时的情况，如果没有类型系统，就无法预先知道记录中存在的字段。因此，我们不知道所需的字段确切位置。例如，考虑以下情况，

```
(Fun x -> x.l)(If y = 0 Then {l = 3} Else {a = 4; l = 3})
```

字段 `l` 将在这些记录中出现在两个不同的位置，因此选择将没有唯一的位置可以找到该字段。因此，我们需要使用散列表进行记录查找。在像Caml这样的类型语言中，可以避免这个问题：上面的代码在Caml中不是很好地类型化，因为if-then不能被类型化。请注意，记录的问题与对象的问题密切相关，因为对象只是具有 `Ref` 的记录。

此内存布局困难与记录说明了类型和编译之间的重要关系。类型系统对程序结构施加约束，可以使编译器更容易实现。此外，类型检查器将消除处理某些运行时错误的需要。我们的简单 **F♭SR** 编译器将在例如 `4 (5)` 上崩溃；在 Lisp、Smalltalk 或 Scheme 中，这些错误将在运行时被捕获但会减慢执行速度。在类型化语言中，编译器将拒绝程序，因为它将无法通过类型检查。因此，对于类型化语言，它们将既更快又更安全。

我们的记录编译方法如下进行。我们必须对记录进行重实现，作为哈希表（即一组键值对，其中键是标签名称）。为了使实现简单，记录被装箱，以便它们占用一个字节的内存，如我们在讨论装箱时所述。记录选择操作 $v_k.l$ 通过在运行时指向 v_k 的哈希表中的键 l 进行哈希实现。这基本上是如何实现 Smalltalk 消息发送的，因为记录类似于对象（并且 Smalltalk 是无类型的）。

以上不是最佳方案，因为需要为散列表留出空间，并且记录字段访问将比例如C中的`struct`访问慢*much*。由于闭包是记录，这也会显著减慢函数调用。一个简单的优化方法是对闭包记录进行特殊处理，因为字段位置始终固定，并使用`struct`闭包实现（为每个函数创建不同的`struct`类型）。

例如，考虑

```
(Fun x -> x.l)(If y = 0 Then {l = 3} Else {a = 4; l = 3})
```

代码 `x.l` 将调用近似形式 `hashlookup(x,"l")` 的调用。`{a = 4; l = 3}` 将创建一个新的哈希表并将 `"a"` 映射到 4 和 `"l"` 映射到 3。

8.4.2 The toC translation

我们现在准备通过函数将最终翻译写入C语言

- `toCTuple` 将原子元组映射到C语句字符串，
- `toCTuples` 将元组列表映射到C语句，
- `toCFunction` 将原始 **F♭SR** 函数映射到定义 C 函数的字符串，
- `toC` 将一组原始 **F♭SR** 函数映射到 C 函数字符串。

以下非正式书写的翻译为了简便起见做了一些变通。以下字符串 `"..."` 使用了缩写。例如，`"vi = x"` 是 `tostring(vi)` 的缩写

`^" = " ^tostring(x).` 函数和then/else体的元组
`Let x1 = e1 In Let ...In Let xn = en In xn`被假定为已转换为元组列表 `[(x1,e1),...,(xn,en)]`，对于顶级函数定义列表也是如此。在编写编译器时，简单地保持它们以`Let`形式可能更容易，尽管两种策略都可行。

```
toCTuple(vi = x) =           "vi = x;" (* x is a FbSR variable *)
toCTuple(vi = n) =           "vi = n;"
toCTuple(vi = b) =           "vi = b;"
toCTuple(vi = vj + vk) =     "vi = vj + vk;"
toCTuple(vi = vj - vk) =     "vi = vj - vk;"
toCTuple(vi = vj And vk ) =  "vi = vj && vk;"
toCTuple(vi = vj Or vk ) =   "vi = vj || vk;"
toCTuple(vi = Not vj ) =     "vi = !vj;"
toCTuple(vi = vj = vk) =     "vi = (vj == vk);"
toCTuple(vi = (vj vk) =      "vi = *vj(vk);"
toCTuple(vi = Ref vj) =      "vi = malloc(WORDSIZE); *vi = vj;"
toCTuple(vi = vj := vk) =    "vi = *vj = vk;"
toCTuple(vi = !vj) =         "vi = *vj;"
toCTuple(vi = { l1 = v1; ... ; ln = vn }) =
         /* 1. malloc a new hashtable at vi
            2. add mappings l1 -> v1 , ... , ln -> vn  */

toCTuple(vi = vj.l) =        "vi = hashlookup(vj,"l");"
toCTuple(vi = If vj Then tuples1 Else tuples2) =
  "if (vj) { toCTuples(tuples1) } else { toCTuples(tuples2) };"
toCtuples([]) = ""

toCtuples(tuple::tuples) = toCtuple(tuple) ^ toCtuples(tuples)

toCFunction(f = Fun xx -> tuples) =
  "Word f(Word xx) {" ^ ... declare temporaries ...
  toCtuples(tuples) ^
 "return(resultId tuples); };"

toCFunctions([]) = ""
toCFunctions(Functiontuple::Functiontuples) =
  toCFunction(Functiontuple) ^ toCFunctions(Functiontuples)

(* toC then invokes toCFunctions on its list of functions. *)
```

读者可能会想知道为什么为`Ref`分配了一个新的内存位置，而不是简单地存储被引用的对象的现有地址。这是一个微妙的问题，例如，代码`vi = &vj`肯定不适用于`Ref`的情况（`vj`可能会超出作用域）。

此上翻译草图省略了许多细节。以下是详细说明。

Typing issues 我们设计了内存布局，使得每个实体占用一个字的空间。因此，每个变量都是某种类型，该类型大小为一个字。将所有变量标记为 `Word`，其中 `Word` 是一个字大小的类型（例如定义为 `typedef void *Word;`）。许多

类型转换需要插入；我们基本上关闭了C的类型检查，但没有可以切换的“开关”。因此，例如 {v*} 实际上是 `vi = (Word (int vj) + (int vk))` – 将单词转换为 `int`s，进行加法，然后转换回单词。将类型转换为函数指针是一个绕口令：在C中你可以使用 `(*((Word (*)()) f))(arg)`。在编写编译器时避免混淆的最简单方法是将以下 `typedefs` 包含到生成的C代码中：

```
/*
 * Define the type ‘Word’ to be a generic one-word value.
 */
typedef void *Word;

/*
 * Define the type ‘FPtr’ to be a pointer to a function that
 * consumes Word and returns a Word.  All translated
 * functions should be of this form.
 */
typedef Word (*FPtr)(Word);

/*
 * Here is an example of how to use these typedefs in code.
 */
Word my_function(Word w) {
  return w;
}
int main(int argc, char **argv) {
  Word f1 = (Word) my_function;
  Word w1 = (Word) 123;
  Word w2 = ( *((FPtr) f1) )(w1); /* Computes f1(123) */
  printf("%d\n", (int) w2);       /* output is "123\n". */
  return 0;
}
```

Global Issues 一些全球性问题您需要处理，包括以下内容。您需要打印出由 `main` 函数返回的结果（因此，您可能希望将 FbSR 主函数命名为 `FbSRmain`，然后手动编写自己的 `main()`，该函数将调用 `FbSRmain`）；C 函数需要声明它们使用的所有临时变量。一种解决方案是在函数头中声明一个 C 数组

```
Word v[22]
```

在特定函数中，22 是所需的临时变量数量，并为临时变量使用名称 `v[0]`，`v[1]`，等。注意，只有当 `newid()` 函数被指示在编译每个新文件时再次从零开始编号临时变量时，此方法才有效。

函数。每个编译程序都必须在头文件中包含一个标准的C代码块，包括记录哈希实现、如上所述的`main()`等。

其他呈现问题的包括以下内容。记录创建仅作草图；但有许多C哈希集合库可用于此目的。`Then`和`Else`元组的最终结果(`resultId`)需要存储在`vi`变量*same*中，这也是元组结果存储的变量，以确保`If`代码正确。这最好在A翻译阶段处理。

存在其他一些在编写编译器时会出现的问题。完整的实现留作练习。

8.4.3 Compilation to Assembly code

现在我们已经涵盖了编译到C的过程，我们非正式地考虑编译器如何将代码编译成机器码。我们上面生成的C代码非常接近汇编代码。从概念上讲，将其翻译成汇编代码很容易，但我们跳过了这个主题，因为在这个过程中会产生大量情况（保存寄存器、在栈上为临时变量分配空间）。

8.5 Summary

```
let frontend e = hoist(atrans(clconv(e)));;
let translator e = toC(frontend(e));;
```

我们可以断言我们翻译器的正确性。

Theorem 8.4. *FbSR program* `e` *terminates in the* ***FbSR*** *operational semantics (or evaluator) just when the C program* `translator(e)` *terminates, provided the C program does not run out of memory. Core dump or other run-time errors are equated with nontermination. Furthermore, if* ***FbSR****'s* `eval(e)` *returns a number* `n`*, the compiled* `translator(e)` *will also produce numerical output* `n`*.*

8.6 Optimization

优化可以在翻译过程的各个阶段进行。上述翻译简单，但效率低下。在编译器设计中，总是存在简单性和效率之间的权衡，这包括编译本身的效率和生成的代码效率。在生成C代码之前的阶段，优化包括用操作上等效的块替换程序中的块。

一些简单的优化包括以下内容。特殊闭包记录 {`fn = .., envt = ..` } 可以实现为一个具有 `fn` 和 `envt` 字段的 C `struct` 指针，而不是使用非常慢的哈希方法，这将显著加快生成的代码速度。没有与其他记录字段名重叠的记录也可以以这种方式实现（可以有两条不同的记录具有

[1] Although hash lookups are $O(1)$, there is still a large amount of constant overhead, whereas `struct` access can be done in a single load operation.

相同字段，但不是两个不同的记录，其中一些字段相同，一些不同）。另一个优化是修改A-翻译以避免为变量和常量创建元组。例如，常量表达式`3 + 4`可以折叠为`7`。

更复杂的优化需要执行全局 **flow analysis**。简单来说，流分析找出特定定义的所有可能的 *uses*，以及与特定使用相对应的所有可能的 *definitions*。

定义是一个记录、一个 `Fun` 或一个数字或布尔值，而使用是一个记录字段选择、函数应用或数值或布尔运算符。

8.7 Garbage Collection

我们的编译代码 `malloc` 但从未释放。我们最终会耗尽内存。需要一个垃圾回收器。

Definition: 在运行时图像中，内存位置`n`是*garbage*，如果它将永远不会再次被读取或写入。

存在许多垃圾检测的概念。最常见的是相对保守一些，将垃圾视为没有任何已知（“根”）对象指向的内存位置。

Bibliography

[1] 开源Erlang。http://www.erlang.org/。

[2] 自编程语言。http://research.sun.com/self/language.html。 [3] 类型化汇编语言。http://www.cs.cornell.edu/talc/。

[4] A.V. Aho，R. Sethi，和J.D. Ullman. *Compilers: Principles, Techniques and Tools*。Addison-Wesley，1986。

[5] A. Appel. *Compiling with Continuations*. 剑桥大学出版社，1992年。

[6] 安德鲁·阿佩尔. *Modern Compiler Implementation in ML*. 剑桥大学出版社，1998年。

[7] 鸟 Richard. *Introduction to Functional Programming using Haskell*。普伦蒂斯·霍尔，第2版，1998年。

[8] 吉布拉德·布拉查和戴维·格里索尔德。在生产环境中对Smalltalk进行类型检查：Strongtalk。在*Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*，第215-230页，1993年。

[9] Kim Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT出版社，2002年。

[10] 约翰·艾弗里格，斯科特·史密斯，瓦列里·特里福诺夫，以及艾米·兹瓦里科。面向对象类型理论的运用：状态、可决性、集成。在 *OOPSLA '94*，第 16–30 页，1994 年。

[11] 马丁·福勒. *UML Distilled*. 奥德赛出版社，第2版，2000年。

[12] 艾瑞克·伽玛，理查德·赫尔姆，拉尔夫·约翰逊，约翰·弗利斯德斯。*Design Patterns: Elements of Reusable Object-Oriented Software*。Addison-Wesley专业计算系列。Addison-Wesley，1994。

[13] Paul Hudak, John Peterson, 和 Joseph Fasel. Haskell版本98的温和介绍，2000年6月。http://www.haskell.org/tutorial/。

[14] Andrew D. Irvine. 罗素悖论。斯坦福哲学百科全书，2001年6月。http://plato.stanford.edu/entries/russell-paradox/。

[15] Xavier Leroy. Objective Caml系统版本3.11发布，文档和用户手册，2008年11月。
`http://caml.inria.fr/pub/docs/manual-ocaml/ index.html`。

[16] Ian A. Mason, Scott F. Smith, 和 Carolyn L. Talcott. 从操作语义学到领域理论。
Information and Computation，128(1):26–47，1996。

[17] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith , David Walker, Stephanie Weirich 和 Steve Zdancewic. Talx86：一个现实的类型化汇编语言。在 *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*，第25–35 页，美国佐治亚州亚特兰大，1999 年 5 月。

[18] J J O’Connor 和 E F Robertson. 格奥尔格·威廉·冯·莱布尼茨。MacTutor 数学史档案，1998年10月。`http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Leibniz.html`。

[19] Randall B. Smith 和 David Ungar. 自我：简单之力量。在 *Conference proceedings on Object-oriented programming systems, languages and applications*，第227–242 页。ACM 压力，1987。

[20] Randall B. Smith 和 David Ungar. 作为体验的编程：Self 的灵感。
Lecture Notes in Computer Science，952:303–??，1995。[21] Scott Smith. 编程语言课程。`http://pl.cs.jhu.edu/pl`。[22] J. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory。麻省理工学院出版社，1977。

[23] Paul R. Wilson. 单处理器垃圾回收技术。*ACM Computing Sur- veys*，2002。
`ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps`。

Index