

*“É melhor fazer cem funções operarem sobre uma estrutura de dados
do que dez funções operarem sobre dez estruturas de dados.”*

Atribuído a Alan Perlis

VISÃO GERAL DO CAPÍTULO

14.1	FUNÇÕES E O CÁLCULO LAMBDA	362
14.2	SCHEME	366
14.3	HASKELL	388
14.4	RESUMO	408
	EXERCÍCIOS	408

A programação funcional emergiu como um paradigma distinto no início da década de 1960. Sua criação foi motivada pela necessidade dos pesquisadores no desenvolvimento de inteligência artificial e em seus subcampos – computação simbólica, prova de teoremas, sistemas baseados em regras e processamento de linguagem natural. Essas necessidades não eram particularmente bem atendidas pelas linguagens imperativas da época.

A linguagem funcional original era a Lisp, desenvolvida por John McCarthy (McCarthy, 1960) e descrita no *LISP 1.5 Programmer's Manual* (McCarthy et al., 1965). A descrição é notável tanto pela sua clareza quanto pela sua brevidade; o manual tem apenas 106 páginas!

Ele contém não apenas uma descrição do sistema Lisp, mas também uma definição formal da própria Lisp. Aqui está uma citação dos autores (McCarthy et al., 1965, p. 1):

A linguagem Lisp serve primariamente para processamento simbólico de dados. Ela tem sido usada para cálculos simbólicos em cálculo diferencial e integral, projeto de circuitos elétricos, lógica matemática, jogos e outros campos da inteligência artificial.

Recorde-se do Capítulo 1, no qual foi dito que a característica essencial da *programação funcional* é que a computação é vista como uma função matemática mapeando entradas a saídas. Diferentemente da programação imperativa, não há uma noção de estado e, portanto, não há necessidade de uma instrução de atribuição. Assim, o efeito de um laço é obtido via repetição, pois não há uma maneira de incrementar ou decrementar o valor de uma variável no estado, já que não há variáveis. Em termos práticos, porém, muitas linguagens funcionais suportam as noções de variável, atribuição e laço. O importante aqui é que esses elementos não fazem parte do modelo “puro” de programação funcional, e, por isso, não vamos destacá-los neste capítulo.

Devido à sua impureza relativa, que explicaremos em breve, a programação funcional é vista por alguns como um paradigma mais confiável para projetos de software do que a programação imperativa. Porém, essa visão é difícil de documentar, pois a maioria das aplicações de inteligência artificial para as quais a programação funcional é usada não é facilmente acessível a soluções no paradigma imperativo, e reciprocamente. Para uma discussão mais cuidadosa dos méritos da programação funcional *versus* programação imperativa, veja Hughes, 1989.

14.1 FUNÇÕES E O CÁLCULO LAMBDA

Uma função matemática típica, como o quadrado de um número, freqüentemente é definida por:

$$\text{Square}(n) = n * n$$

Essa definição dá o nome da função, seguido de seus argumentos entre parênteses, seguido por uma expressão que define o significado da função. *Square* é entendida como sendo uma função que mapeia do conjunto de números reais **R** (seu *domínio*) para o conjunto de números reais **R** (seu *intervalo*), ou mais formalmente:

$$\text{Square}: \mathbf{R} \rightarrow \mathbf{R}$$

Definição: Dizemos que uma função é *total* se ela é definida para todos os elementos em seu domínio, e *parcial* em caso contrário.

A função *Square* é total sobre todo o conjunto de números reais.

Em linguagens de programação imperativa, uma variável como *x* representa uma localização na memória. Assim, a instrução:

$$x = x + 1$$

significa literalmente “atualizar o estado do programa somando 1 ao valor armazenado na célula de memória denominada *x* e depois armazenar aquela soma novamente naquela célula

de memória”. Assim, o nome x é usado para representar um valor (como em $x + 1$), muitas vezes chamado *valor-r* (*r-value*), e um endereço de memória é chamado *valor-l* (*l-value*).¹ (Veja a Seção 7.4.)

Na matemática, as variáveis são, de certa forma, diferentes em suas semânticas: elas sempre significam expressões reais e são imutáveis. Na matemática não há um conceito de “célula de memória” ou de atualização de um valor da memória ou do valor de uma célula de memória. Linguagens de programação funcional denominadas *puras* eliminam a noção de célula de memória de uma variável em favor da noção matemática; isto é, uma variável dá nome a uma expressão imutável, que também elimina o operador de atribuição. Uma linguagem funcional é *pura* se não houver nenhum conceito de um operador de atribuição ou de uma célula de memória; caso contrário, dizemos que ela é *impura*. No entanto, muitas linguagens de programação funcionais retêm alguma forma de operador de atribuição e são, portanto, *impuras*.

Uma consequência da falta de variáveis e atribuições baseadas em memória é que a programação funcional não tem nenhuma noção do estado, como foi feito na definição do significado da linguagem imperativa Clite. O valor de uma função como *Square* depende somente dos valores de seus argumentos, e não de qualquer computação prévia ou mesmo da ordem de avaliação de seus argumentos. Essa propriedade de uma linguagem funcional é conhecida como *transparência referencial*. Uma função tem *transparência referencial* se o seu valor depende somente dos valores de seus argumentos.

A base da programação funcional é o *cálculo lambda*, desenvolvido por Church (1941). Uma expressão lambda especifica os parâmetros e a definição de uma função, mas não seu nome. Por exemplo, veja a seguir uma expressão lambda que define a função *square* discutida acima.

$$(\lambda x \cdot x * x)$$

O identificador x é um parâmetro usado no corpo (sem nome) da função $x*x$. A aplicação de uma expressão lambda a um valor é representada por:

$$((\lambda x \cdot x * x)2)$$

que dá como resultado 4.

Este exemplo é uma ilustração de um cálculo lambda aplicado. O que Church realmente definiu foi um cálculo lambda *puro* ou *não-interpretado*, da seguinte maneira:

- 1 Qualquer identificador é uma expressão lambda.
- 2 Se M e N forem expressões lambda, então a *aplicação* de M a N , escrito como $(M N)$, é uma expressão lambda.
- 3 Uma *abstração*, escrita como $(\lambda x \cdot M)$, na qual x é um identificador e M é uma expressão lambda, é também uma expressão lambda.

Um conjunto simples de regras gramaticais BNF para a sintaxe desse cálculo lambda puro pode ser escrito como:

$$\text{ExpressãoLambda} \rightarrow \text{variable} \mid (M N) \mid (\lambda \text{variable} \cdot M)$$

$$M \rightarrow \text{ExpressãoLambda}$$

$$N \rightarrow \text{ExpressãoLambda}$$

1. Os termos *valor-r* (*r-value*) e *valor-l* (*l-value*) foram inventados originalmente porque eles se referiam aos valores retornados pelo lado direito e lado esquerdo (*right* e *left*) de uma atribuição, respectivamente.

Alguns exemplos de expressões lambda:

x
 $(\lambda x \cdot x)$
 $((\lambda x \cdot x)(\lambda y \cdot y))$

Na expressão lambda $(\lambda x \cdot M)$, dizemos que a variável x está *ligada* à subexpressão M . Uma *variável ligada* é uma variável cujo nome é igual ao nome do parâmetro; caso contrário, dizemos que a variável é *livre* (*free*). Dizemos que qualquer variável não-ligada em M é *livre*. Variáveis ligadas são simplesmente marcadores de lugar, assim como parâmetros de função nos paradigmas imperativo e orientado a objetos. Qualquer variável assim pode ser renomeada consistentemente com qualquer variável livre em M sem mudar o sentido da expressão lambda. Formalmente, as variáveis livres em uma expressão lambda podem ser definidas como:

$$\begin{aligned} \text{free}(x) &= x \\ \text{free}(MN) &= \text{free}(M) \cup \text{free}(N) \\ \text{free}(\lambda x \cdot M) &= \text{free}(M) - \{x\} \end{aligned}$$

Uma substituição de uma expressão N por uma variável x em M , escrita como $M[x \leftarrow N]$, é definida da seguinte forma:

- 1 Se as variáveis livres de N não possuem ocorrências ligadas em M , então o termo $M[x \leftarrow N]$ é formado pela substituição de todas as ocorrências livres de x em M por N .
- 2 Caso contrário, assuma que a variável y é livre em N e ligada em M . Depois substitua consistentemente as ocorrências de ligações de y em M por uma nova variável, digamos u . Repita essa operação de renomear variáveis ligadas em M até que se aplique a condição do passo 1, depois proceda como no passo 1.

Os exemplos a seguir ilustram o processo de substituição:

$$\begin{aligned} x[x \leftarrow y] &= y \\ (xx)[x \leftarrow y] &= (yy) \\ (zw)[x \leftarrow y] &= (zw) \\ (zx)[x \leftarrow y] &= (zy) \\ (\lambda x \cdot (zx))[x \leftarrow y] &= (\lambda u \cdot (zu))[x \leftarrow y] = (\lambda u \cdot (zu)) \\ (\lambda x \cdot (zx))[y \leftarrow x] &= (\lambda u \cdot (zu))[y \leftarrow x] = (\lambda u \cdot (zu)) \end{aligned}$$

O sentido de uma expressão lambda é definido pela seguinte regra de redução:

$$((\lambda x \cdot M)N) \Rightarrow M[x \leftarrow N]$$

Isso é chamado *redução-beta* e pode ser lido como “sempre que tivermos uma expressão lambda da forma $((\lambda x \cdot M)N)$, podemos simplificá-la pela substituição $M[x \leftarrow N]$ ”. Uma redução-beta, portanto, representa uma aplicação singular de função.

Uma *avaliação* de uma expressão lambda é uma seqüência $P \Rightarrow Q \Rightarrow R \Rightarrow \dots$ na qual cada expressão na seqüência é obtida pela aplicação de uma redução-beta à expressão anterior. Se P é uma expressão lambda, então uma *redux* de P é qualquer subexpressão obtida por uma redução-beta. Uma expressão lambda que não contém uma função de aplicação é chamada *forma normal*.

Um exemplo de avaliação é:

$$((\lambda y \cdot ((\lambda x \cdot xyz)a))b) \Rightarrow ((\lambda y \cdot ayz)b) \Rightarrow abz$$

Neste exemplo, nós avaliamos a expressão mais interna λ primeiro; poderíamos ter feito com a mesma facilidade a redução mais externa primeiro:

$$((\lambda y \cdot ((\lambda x \cdot xyz)a))b) = ((\lambda x \cdot xbz)a) = abz$$

A igualdade das expressões lambda é chamada *igualdade-beta* por razões históricas e também porque o termo sugere a redutibilidade beta de uma expressão para outra. Informalmente, se duas expressões lambda M e N forem iguais, escritas como $M = N$, então M e N podem ser reduzidas à mesma expressão até renomear suas variáveis. Igualdade-beta trata da aplicação de uma abstração $(\lambda x \cdot M)$ para um argumento N , e assim proporciona um modelo fundamental para as noções de chamada de função e passagem de parâmetro em linguagens de programação.

Uma linguagem de programação funcional é essencialmente um cálculo lambda aplicado com valores constantes e funções embutidas. A expressão lambda pura (xyx) pode facilmente ser escrita também assim: $(x \times x)$ ou $(x * x)$. Além do mais, essa última forma pode ser escrita em um estilo prefixo $(* x x)$. Quando somamos constantes como números com sua interpretação usual e suas definições para funções, como $*$, obtemos um cálculo lambda aplicado. Por exemplo, $(* 2 x)$ é uma expressão em um cálculo lambda aplicado. Conforme veremos, Lisp/Scheme e Haskell puras são exemplos de cálculos lambda aplicados.

Uma distinção importante em linguagens funcionais é feita usualmente na forma como elas definem avaliação de função. Em linguagens como Scheme, todos os argumentos para uma função são normalmente avaliados no instante da chamada. Isso usualmente é chamado *avaliação rápida*, ou *chamada por valor* (conforme discutimos no Capítulo 9). *Avaliação rápida* em linguagens funcionais refere-se à estratégia de avaliar todos os argumentos para uma função no instante da chamada. Com a avaliação rápida, funções como `if` e `and` não podem ser definidas sem erros potenciais em tempo de execução, como na função Scheme

```
(if (= x 0) 1 (/ 1 x))
```

que define o valor da função como sendo 1 quando x é zero e $1/x$ em caso contrário. Se todos os argumentos para a função `if` forem avaliados no instante da chamada, a divisão por zero não pode ser evitada.

Definição: Uma alternativa para a estratégia da avaliação rápida é chamada *avaliação lenta*, na qual um argumento para uma função não é avaliado (ela é adiada) até que ele seja necessário.

Como um mecanismo de passagem de argumento, a avaliação lenta é similar (mas não idêntica) à chamada por nome, e é o mecanismo-padrão na linguagem Haskell. Scheme também tem mecanismos que permitem ao programador especificar o uso de avaliação lenta, mas não exploraremos esses mecanismos neste capítulo.

Uma vantagem da avaliação rápida é a eficiência, em que cada argumento passado a uma função é avaliado apenas uma vez, enquanto na avaliação lenta um argumento para uma função é reavaliado cada vez que ele é usado, e isso pode ocorrer mais de uma vez. Uma vantagem da avaliação lenta é que ela permite certas funções interessantes a serem definidas, de modo que não podem ser implementadas em linguagens rápidas.

Mesmo a definição anterior da função `if` torna-se isenta de erro com uma estratégia de avaliação lenta, já que a divisão $(/ \ 1 \ x)$ só ocorrerá quando $x \neq 0$.

Em cálculo lambda puro, a aplicação da função:

$$((\lambda x \cdot *x \ x)5) = (* \ 5 \ 5)$$

não dá qualquer interpretação para o símbolo `5` ou o símbolo `*`. Somente em um cálculo lambda aplicado conseguiríamos uma redução maior para o número 25.

Um aspecto importante da programação funcional é que as funções são tratadas como valores de primeira classe. Um nome de uma função pode ser passado como um parâmetro, e uma função pode retornar outra função como valor. Uma função dessas às vezes é chamada *forma funcional*. Um exemplo de uma forma funcional seria uma função `g` que toma como parâmetro uma função e uma lista (ou uma sequência de valores) e aplica a função dada a cada elemento na lista, retornando uma lista. Usando `Square` como exemplo, então:

```
g(f, [x1, x2, ...]) = [f(x1), f(x2), ...]
```

torna-se

```
g(Square, [2, 3, 5]) = [4, 9, 25]
```

Nas Seções 14.2 e 14.3, exploramos o uso dessas e de muitas outras formas funcionais úteis.

14.2 SCHEME

Como linguagem original de programação funcional, Lisp tem muitas características que foram transportadas para linguagens posteriores, portanto, Lisp proporciona uma boa base para estudar outras linguagens funcionais. Com o passar dos anos, foram desenvolvidas muitas variantes de Lisp; hoje, somente duas variantes principais permanecem em uso difundido: Common Lisp (Steele, 1990) e Scheme (Kelsey et al., 1998) (Dybvig, 1996). Como linguagens que tentam unificar um conjunto de variantes, tanto Scheme quanto Common Lisp contêm um conjunto de funções equivalentes. Este capítulo apresenta um subconjunto puramente funcional de Scheme.

Quando vista como linguagem funcional pura, nosso subconjunto² Scheme não tem nenhuma instrução de atribuição. Em vez disso, os programas são escritos como funções (repetitivas) sobre valores de entrada que produzem valores de saída; os próprios valores de entrada não são alterados. Nesse sentido, a notação Scheme está muito mais próxima da matemática do que estão as linguagens de programação imperativa e orientada a objetos como C e Java.

Sem uma instrução de atribuição, nosso subconjunto Scheme faz uso intensivo das funções repetitivas para repetição, em lugar da instrução *while* que encontramos nas linguagens imperativas. Apesar da ausência das instruções *while*, pode-se provar que esse subconjunto é *Turing completo*, o que significa que qualquer função computável pode ser implementada naquele subconjunto. Isto é, uma linguagem funcional é Turing completo porque ela tem valores inteiros e operações, uma maneira de definir novas funções usando funções existentes, condicionais (instruções *if*) e repetição.

2. A linguagem Scheme completa tem uma instrução de atribuição chamada `set!`, que evitaremos usar nessa discussão.

Uma prova de que essa definição de Turing completo é equivalente àquela do Capítulo 12 para linguagens imperativas está além do escopo deste livro. No entanto, o uso de Scheme para implementar a semântica denotacional de Clite dada neste capítulo deve proporcionar uma evidência convincente, embora informal, de que uma linguagem puramente funcional é pelo menos tão poderosa quanto uma linguagem imperativa. O inverso também é verdadeiro, pois interpretadores Scheme e Lisp são implementados em máquinas von Neumann, que são a base para o paradigma imperativo.

14.2.1 Expressões

Expressões em Scheme são construídas em notação de prefixo de Cambridge, na qual as expressões ficam entre parênteses e o operador ou a função precede seus operandos, como no exemplo:

```
(+ 2 2)
```

Se essa expressão for apresentada a um interpretador Scheme, ele retorna o valor 4.

Uma vantagem dessa notação é que ela permite que operadores aritméticos como `+` e `*` tomem um número arbitrário de operandos:

```
(+) ; evaluates to 0
(+ 5) ; evaluates to 5
(+ 5 4 3 2 1) ; evaluates to 15
(*) ; evaluates to 1
(* 5) ; evaluates to 5
(* 1 2 3 4 5) ; evaluates to 120
```

Observe que ponto-e-vírgula em Scheme inicia um comentário, que continua até o fim da linha. Essas expressões aritméticas são exemplos de listas Scheme; dados e programas (funções) são representados por listas. Quando uma lista Scheme é interpretada por uma função, o operador ou o nome da função vem após o parêntese esquerdo e os demais números são seus operandos. Expressões mais complicadas podem ser construídas usando aninhamento:

```
(+ (* 5 4) (- 6 2))
```

que é equivalente a $5 * 4 + (6 - 2)$ em notação interfixada, e é avaliado como 24.

Variáveis globais são definidas em Scheme pelo uso da função `define`. Para definir uma variável `f` igual a 120 colocaríamos a expressão:

```
(define f 120)
```

A função `define` é a única das que examinaremos que muda seu ambiente, em vez de simplesmente retornar um valor. No entanto, não trataremos `define` como uma instrução de atribuição em nosso subconjunto Scheme;³ nós só a usamos para introduzir um nome global para um valor, como ocorre na matemática.

3. A função `set!` é uma verdadeira atribuição em Scheme porque ela pode ser usada para mudar o valor de uma variável existente. Muitos textos Scheme usam a função `set!` em um nível global como equivalente para `define`.

14.2.2 Avaliação de Expressões

Para entender como Scheme avalia as expressões, são aplicadas três regras principais.

Primeira, nomes ou símbolos são substituídos por suas ligações atuais. Supondo a definição da variável `f` da seção anterior:

```
f                ; evaluates to 120
( + f 5)         ; evaluates to 125
                ; using the bindings for +, f
```

Esse uso de `f` é um exemplo da primeira regra.

A segunda regra é que as listas são avaliadas como chamadas de função escritas em notação de prefixo de Cambridge:

```
(+)              ; calls + with no arguments
(+ 5)            ; calls + with 1 argument
(+ 5 4 3 2 1)    ; calls + with 5 arguments
(+ (5 4 3 2 1))  ; error, tries to evaluate 5 as
                  ; a function
(f)              ; error; f evaluates to 120, not
                  ; a function
```

A terceira regra é que constantes avaliam a si próprias:

```
5                ; evaluates to 5
#f               ; is false, predefined
#t               ; is true, predefined
```

Pode-se impedir que um símbolo ou uma lista seja avaliado usando a função `quote` (bloqueador de avaliação) ou o apóstrofo (`'`), como em:

```
(define colors (quote (red yellow green)))
(define colors '(red yellow green))
```

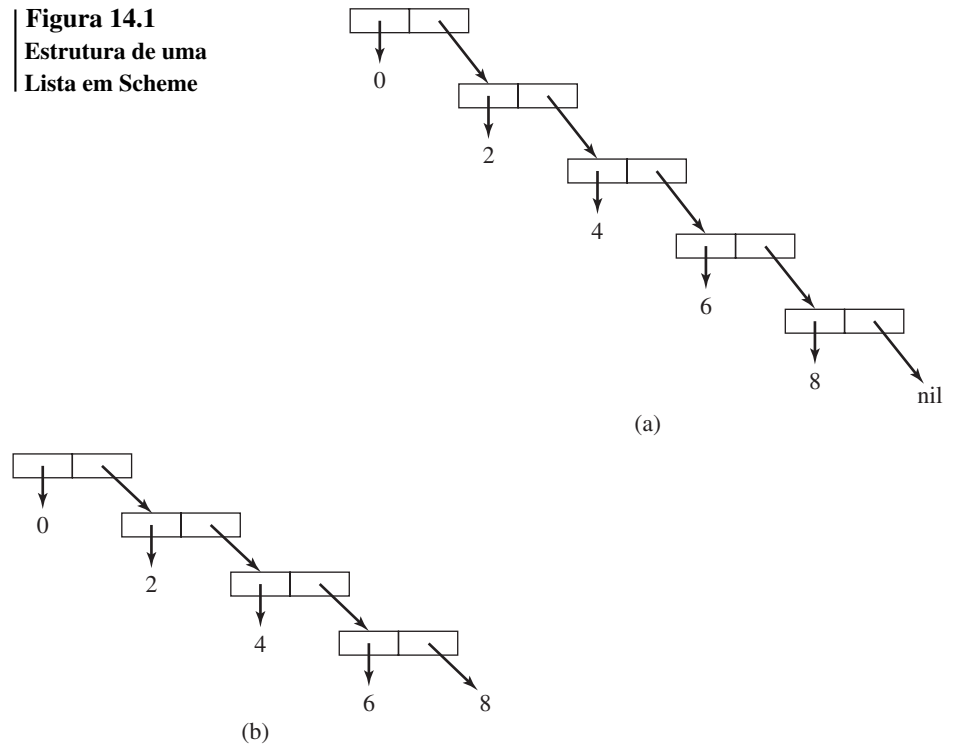
Observe que não há um apóstrofo de fechamento; o apóstrofo bloqueia o símbolo ou a lista vem imediatamente após ele. Você pode também bloquear símbolos:

```
(define x f)      ; defines x to be 120 (value of f)
(define x 'f)     ; defines x to be the symbol f
(define acolor 'red) ; defines acolor to be red
(define acolor red) ; an error, symbol red not defined
```

14.2.3 Listas

Conforme já vimos, a estrutura de dados fundamental de Scheme é a lista; ela é usada para comandos e dados. Já vimos muitos exemplos de listas de constantes. Nesta seção vamos ver como colocamos coisas nas listas e como as recuperamos depois.

Figura 14.1
Estrutura de uma
Lista em Scheme



Primeiro, vamos definir algumas listas de números:

```
(define evens '(0 2 4 6 8))
(define odds '(1 3 5 7 9))
```

Graficamente, a lista `evens` seria representada conforme mostra a Figura 14.1a.

O símbolo `()` representa uma lista vazia; por convenção, as listas Scheme usualmente terminam com `()`.⁴ Em termos de lista “linkada” de forma imperativa, o símbolo `()` pode ser visto como um ponteiro `null`. Se o símbolo `()` estiver faltando no fim da lista, Scheme o mostraria como:

```
(0 2 4 6 . 8)
```

Esse tipo de estrutura pode ser gerado por algumas das funções Scheme discutidas a seguir. A Figura 14.1 mostra a diferença em representação de memória entre uma lista que tenha `()` como seu último elemento (Figura 14.1a) e outra que não tenha (Figura 14.1b).

4. O valor `()` é implementado pela `null` reference, conforme discutido anteriormente no Capítulo 5.

A função básica usada para construir uma lista é `cons`, que toma dois argumentos, o segundo dos quais deverá ser uma lista:

```
(cons 8 ()) ; gives (8)
(cons 6 (cons 8 ())) ; gives (6 8)
(cons 4 (cons 6 (cons 8 ()))) ; gives (4 6 8)
(cons 4 (cons 6 (cons 8 9))) ; gives (4 6 8 . 9)
```

Observe que o último exemplo cria uma lista irregular, já que o segundo argumento do `cons` mais interno não é uma lista.

Um nó em uma lista Scheme tem duas partes, o primeiro elemento ou o início da lista, e os demais elementos da lista ou final. A função `car` retorna o início da lista, enquanto a função `cdr` retorna o final.⁵ Referindo-nos novamente à lista `evens` representada na Figura 14.1, os exemplos a seguir ilustram essas funções:

```
(car evens) ; gives 0
(cdr evens) ; gives (2 4 6 8)
(car (cdr evens)) ; gives 2
(cadr evens) ; gives 2
(cdr (cdr evens)) ; gives (4 6 8)
(cddr evens) ; gives (4 6 8)
(car '(6 8)) ; gives 6
(car (cons 6 8)) ; gives 6
(cdr '(6 8)) ; gives (8)
(cdr (cons 6 8)) ; gives 8, not (8)
(car '(8)) ; gives 8
(cdr '(8)) ; gives ()
```

Observe que Scheme permite que seqüências de `car`'s e `cdr`'s (até cinco) sejam abreviadas incluindo somente a letra do meio; assim, `cadr` é um `car` de um `cdr`, `cddr` é um `cdr` de um `cdr`, e assim por diante.

Funções de nível mais alto para juntar listas incluem as funções `list` e `append`. A função `list` toma um número variável de argumentos e constrói uma lista que consiste naqueles argumentos:

```
(list 1 2 3 4) ; gives (1 2 3 4)
(list '(1 2) '(3 4) 5) ; gives ((1 2) (3 4) 5)
(list evens odds) ; gives ((0 2 4 6 8) (1 3 5 7 9))
(list 'evens 'odds) ; gives (evens odds)
```

5. Os termos `car` e `cdr` são vestígios dos primeiros tempos, quando Lisp foi implementada no IBM 704. Os registradores de endereço daquela máquina tinham duas partes, a parte do *address* e a parte do *decrement*. Assim, os termos originais Lisp `car` e `cdr` eram abreviaturas para “conteúdo da parte endereço do registrador” e “conteúdo da parte decremento do registrador”, respectivamente. Embora a máquina 704 tenha caído na obscuridade, essas duas abreviaturas permanecem.

Ao contrário, a função `append` toma dois argumentos – e ambos os argumentos serão listas –, e ocorre um encadeamento da segunda lista ao fim da primeira lista:

```
(append '(1 2) '(3 4))      ; gives (1 2 3 4)
(append evens odds)         ; gives (0 2 4 6 8 1 3 5 7 9)
(append '(1 2) '())         ; gives (1 2)
(append '(1 2) (list 3))    ; gives (1 2 3)
```

Para acrescentar um único número ao início da lista de números pares, usaríamos a função `cons`:

```
(cons 10 evens)             ; gives (10 0 2 4 6 8)
```

A lista vazia `()` no fim da lista é importante; ao percorrer repetidamente uma lista, verificamos sempre se há uma lista vazia no fim.

Como Scheme foi projetada para processar listas, ela contém um conjunto de funções especiais de processamento de lista. Devido à necessidade freqüente, há uma função especial denominada `null?` para testar quanto à existência de uma lista vazia:

```
(null? '())                 ; returns #t
(null? evens)               ; returns #f
(null? '(1 2 3))           ; returns #f
(null? 5)                   ; returns #f
```

Scheme contém um conjunto de funções para testar se um objeto é igual ou equivalente a outro. Essas funções são `equal?`, `=` e `eqv?`. Em lugar de relacionar suas diferenças, vamos nos basear na função `equal?`, que é razoavelmente geral. Essa função retorna *true* (`#t`) se os dois objetos tiverem a mesma estrutura e mesmo conteúdo; caso contrário, ela retorna *false*:

```
(equal? 5 5)                ; returns #t
(equal? 5 1)                ; returns #f
(equal? '(1 2) '(1 2))      ; returns #t
(equal? 5 '(5))             ; returns #f
(equal? '(1 2 3) '(1 (2 3))) ; returns #f
(equal? '(1 2) '(2 1))      ; returns #f
(equal? '() '())            ; returns #t
```

14.2.4 Valores Elementares

Até aqui, todos os valores elementares que vimos em Scheme são números ou símbolos (nomes). Há realmente alguns tipos diferentes de números, incluindo inteiros, racionais e de ponto flutuante. Outros tipos elementares de valores Scheme incluem caracteres, funções, símbolos e *strings*. Cada um desses tipos pode ser caracterizado usando um qualificativo apropriado; por exemplo, todos os tipos a seguir retornam `#t`:

```
(pair? evens)
(list? evens)
(symbol? 'evens)
(number? 3)
```

Além dos tipos de valores listados, há também valores booleanos : `#t` que significam *true* e valores `#f` que significam *false*. Todos os valores exceto `#f` e a lista vazia `()` são interpretados como `#t` quando usados como um atributo.

14.2.5 Fluxo de Controle

Os dois construtores de controle de fluxo que usaremos são o controle `if` e o controle `case`.

A função `if` vem em duas versões usuais: a versão `if-then` e a versão `if-then-else`. Abstratamente, isso aparece como:

```
(if test then-part)
(if test then-part else-part)
```

Veja um exemplo de cada:

```
(if (< x 0) (- 0 x))
(if (< x y) x y)
```

A primeira função `if` retorna o negativo de `x`, se `x` for menor que 0. A segunda função retorna o menor dentre os dois valores de `x` e `y`.

A função `case` é muito semelhante à `case` em Ada e à `switch` em Java; a função `case` tem um opcional `else` que, caso esteja presente, deve ser o último `case`. Um `case` simples para calcular o número de dias em um mês (ignorando os anos bissextos) é:

```
(case month
  ((sep apr jun nov) 30)
  ((feb) 28)
  (else 31)
)
```

Note que cada `case` específico tem uma lista de constantes, exceto `else`.

14.2.6 Definindo Funções

As funções Scheme são definidas pelo uso de `define`, que tem a seguinte forma geral:

```
(define name (lambda (arguments) function-body))
```

Assim, uma função mínima pode ser definida por:

```
(define min (lambda (x y) (if (< x y) x y)))
```

Em outras palavras, Scheme (assim como sua antecessora Lisp) é um cálculo lambda aplicado com a habilidade de dar nomes a definições lambda específicas.

Como os programadores logo se cansam de colocar sempre a palavra `lambda` e o conjunto extra de parênteses, Scheme fornece uma maneira alternativa de escrever a função `define`:

```
(define (name arguments) function-body)
```

Usando essa alternativa, a função mínima pode ser escrita de forma mais simples:

```
(define (min x y) (if (< x y) x y))
```

A função a seguir calcula o valor absoluto de um número:

```
(define (abs x) (if (< x 0) (- 0 x) x))
```

Ocorrem funções mais interessantes quando usamos a repetição para definir uma função em termos dela própria. Um exemplo da matemática é a venerável função fatorial, que pode ser definida em Scheme da seguinte maneira:

```
(define (factorial n)
  (if (< n 1) 1 (* n (factorial (- n 1)))))
```

Por exemplo, a aplicação da função `(factorial 4)` resulta em 24, se usarmos a definição acima da seguinte maneira:

```
(factorial 4) = (* 4 (factorial 3))
              = (* 4 (* 3 (factorial 2)))
              = (* 4 (* 3 (* 2 (factorial 1))))
              = (* 4 (* 3 (* 2 (* 1 (factorial 0)))))
              = (* 4 (* 3 (* 2 (* 1 1))))
              = (* 4 (* 3 (* 2 1)))
              = (* 4 (* 3 2))
              = (* 4 6)
              = 24
```

Mesmo tarefas iterativas simples como somar uma lista de números são executadas de forma recursiva em Scheme. Aqui, assumimos que os números formam uma lista `(a1 a2...an)` cujos elementos devem ser somados:

```
(define (sum alist)
  (if (null? alist) 0
      (+ (car alist) (sum (cdr alist)))))
```

Observe que essa função de soma é similar em efeito à função soma interna de Scheme (+).

A função `sum` exibe um padrão comum a uma variedade de funções Scheme. A função é invocada por meio da lista após verificar primeiro se a lista está vazia. Esse teste geralmente é chamado *base case* em definições recursivas. O *passo recursivo* prossegue somando o primeiro elemento da lista à soma do restante da lista (a aplicação recursiva da função). É garantido que a função termina porque, em cada passo repetitivo do cálculo, o argumento da lista se torna menor tomando o `cdr` da lista.

Outras funções Scheme interessantes manipulam listas de símbolos, em lugar de números. Todas as funções apresentadas a seguir já estão definidas em Scheme; nós as mostramos aqui porque elas representam padrões recorrentes em programação Scheme.

O primeiro exemplo calcula o tamanho de uma lista, isto é, quantos elementos ela contém, contando sublistas como elementos simples. Veja alguns exemplos do uso da função `length`:

```
(length '(1 2 3 4))           ; returns 4
(length '((1 2) 3 (4 5 6)))  ; returns 3
(length '())                 ; returns 0
(length 5)                   ; error
```

A definição da função `length` segue de perto o padrão definido pela função `sum` com apenas algumas pequenas diferenças:

```
(define (length alist)
  (if (null? alist) 0 (+ 1 (length (cdr alist)))
      ))
```

A aplicação da função `(length '((1 2) 3 (4 5)))` resulta em:

```
(length '((1 2) 3 (4 5)))
= (+ 1 (length '(3 (4 5))))
= (+ 1 (+ 1 (length '((4 5)))))
= (+ 1 (+ 1 (+ 1 (length ())))))
= (+ 1 (+ 1 (+ 1 0)))
= (+ 1 (+ 1 1))
= (+ 1 2)
= 3
```

Outra função comum interna é a função `member`, que testa para saber se um elemento `elt` (que pode ser uma lista) ocorre como um membro de determinada lista. Se não, a função retorna `#f`. Caso contrário, ela retorna o restante da lista, começando no elemento encontrado, que pode ser interpretado como `#t`. Alguns exemplos do uso da função `member`:

```
(member 4 evens)              ; returns (4 6 8)
(member 1 evens)              ; returns #f
(member 2 '((1 2) 3 (4 5)))  ; returns #f
(member '(3 4) '(1 2 (3 4) 5)) ; returns ((3 4) 5)
```

A função `member` é mais comumente usada como um atributo. Uma vez mais, a definição começa com o *base case*, ou seja, uma lista vazia, no qual o *case* `member` retorna a lista vazia. Caso contrário, ela testa para saber se o cabeçalho da lista é igual ao elemento procurado; se for, ela retorna a lista e, em caso contrário, ela se repete no final da lista.

```
(define (member elt alist)
  (if (null? alist) '()
      (if (equal? elt (car alist)) alist
          (member elt (cdr alist))
          )))
```

Nossa última função simples é `subst`, que é usada para substituir seu primeiro argumento para todas as ocorrências de seu segundo argumento em uma lista (seu terceiro argumento). Assim como a função `membro`, a verificação quanto a objetos iguais só é feita no nível mais alto da lista:

```
(subst 'x 2 '(1 2 3 2 1))      ; returns (1 x 3 x 1)
(subst 'x 2 '(1 (2 3) 2 1))    ; returns (1 (2 3) x 1)
(subst 'x 2 '(1 (2 3) (2)))    ; returns (1 (2 3) (2))
(subst 'x '(2 3) '(1 (2 3) 2 3)) ; returns (1 x 2 3)
(subst '(2 3) 'x '(x o x o))   ; returns ((2 3) o (2 3) o)
```

Essa função é interessante porque ela deve construir a lista de saída como resultado da função, em vez de simplesmente retornar um de seus argumentos. Caso contrário, essa função é similar em estrutura à função `member`:

```
(define (subst y x alist)
  (if (null? alist) '()
      (if (equal? x (car alist))
          (cons y (subst y x (cdr alist)))
          (cons (car alist) (subst y x (cdr alist))))
      )))
```

14.2.7 Expressões Let

Muitas vezes, ao se definir uma função, pode ocorrer uma subexpressão várias vezes. Scheme segue a convenção matemática de permitir a introdução de um nome para uma subexpressão. Isso é feito por meio da função `let`, que tem a seguinte forma geral:

```
(let (( var1 expr1 ) ( var2 expr2 ) ... ) body )
```

O uso criterioso de uma função `let` pode melhorar a legibilidade de uma definição de função. Um efeito colateral do uso de `let` é que Scheme só avalia a expressão uma vez, em vez de fazê-lo todas as vezes em que ela ocorre na definição. Veja a seguir um exemplo do uso de uma função `let`:

```
(let ((x 2) (y 3)) (+ x y))      ; returns 5
(let ((plus +) (x 2)) (plus x 3)) ; returns 5
```

Um uso mais interessante da função `let` ocorre quando ela aparece dentro da definição de uma função:

```
(define (subst y x alist)
  (if (null? alist) '()
      (let ((head (car alist)) (tail (cdr alist)))
        (if (equal? x head)
            (cons y (subst y x tail))
            (cons head (subst y x tail)))
        )))
```

Como na matemática, uma função `let` meramente introduz um conjunto de nomes para expressões comuns. Os nomes são ligados aos seus valores somente no corpo da função `let`. Um exemplo mais amplo da função `let` ocorre na Seção 14.2.7.⁶

Em todos os exemplos anteriores, os argumentos para uma função são átomos ou listas. Scheme também permite que uma função seja um parâmetro formal para outra função que está sendo definida:

```
(define (mapcar fun alist)
  (if (null? alist) '()
      (cons (fun (car alist)) (mapcar fun (cdr alist)))
  ))
```

A função `mapcar` tem dois parâmetros, uma função `fun` e uma lista `alist`; ela aplica a função `fun` a cada elemento de uma lista, criando uma lista a partir dos resultados. Como exemplo, considere uma função de um argumento que eleva o argumento ao quadrado:

```
(define (square x) (* x x))
```

A função `square` pode ser usada com `mapcar` em qualquer uma das duas variações a seguir para elevar ao quadrado todos os elementos de uma lista:

```
(mapcar square '(2 3 5 7 9))
(mapcar (lambda (x) (* x x)) '(2 3 5 7 9))
```

Na primeira variação, o primeiro parâmetro é uma função definida. Na segunda variação, é passada uma função sem nome com o uso da *lambda notation* para defini-la. Ambas as variações produzem o resultado `(4 9 25 49 81)`.⁷

Essa facilidade para definir formas funcionais permite às linguagens funcionais como Scheme, Lisp e Haskell uma facilidade de extensibilidade. Com essa facilidade, os usuários podem facilmente acrescentar muitas formas funcionais do tipo “aplicável a tudo”. Muitas funções Scheme simples são definidas dessa maneira.

As Seções 14.3.8 a 14.3.10 desenvolvem vários exemplos interessantes que se combinam para ilustrar o valor exclusivo da programação funcional em Scheme. O primeiro exemplo reinterpreta as semânticas de Clite, que originalmente foram discutidas e implementadas em Java (veja o Capítulo 8). O segundo exemplo, diferenciação simbólica, é um pouco mais clássico, enquanto o terceiro exemplo reconsidera o problema das oito rainhas, que originalmente foi implementado em Java (veja o Capítulo 13).

6. Em Scheme, a ordem de avaliação dos valores `expr1`, `expr2`, e assim por diante, em uma `let` não está implícita. Em outras palavras, cada valor é avaliado independentemente das ligações dos nomes `var1`, `var2`, e assim por diante. Se for necessário se referir a um nome anterior em uma expressão posterior dentro de uma `let`, a função `let*` deverá ser usada.

7. A função interna `map` em Scheme é equivalente a `mapcar`.

Observação

Tracing

Muitos interpretadores Scheme proporcionam um recurso *tracing*, ou passo-a-passo, para ajudar a entender os detalhes do comportamento de uma função. Isso é particularmente útil na depuração. Infelizmente, nenhuma função *tracing* particular faz parte da Standard Scheme (Linguagem Scheme padrão), e assim seu uso varia ligeiramente de uma implementação para outra. A seguir, damos um exemplo do uso de uma função *tracing* para executar passo a passo a ativação e a desativação de chamadas na função *factorial* discutida anteriormente:

```
> (trace factorial)           Trace: Value = 1
> (factorial 4)               Trace: Value = 1
Trace: (factorial 4)          Trace: Value = 2
Trace: (factorial 3)          Trace: Value = 6
Trace: (factorial 2)          Trace: Value = 24
Trace: (factorial 1)          24
Trace: (factorial 0)          > (untrace factorial)
                              > (factorial 4)
                              24
```

Para algumas implementações Scheme, o efeito do *tracing* pode ser conseguido por meio da técnica imperativa padrão de encaixar comandos de impressão de depuração. Uma função conveniente para ser usada é `printf`, que é similar à função do mesmo nome em C. A função `printf` toma como seu primeiro argumento uma *string* que especifica como a saída deve ser mostrada; o código `~a` é usado para mostrar um valor, enquanto `~n` é usado para representar um código de fim de linha. Usando isso, podemos reescrever a função *fatorial* para obter um resultado similar ao que vimos acima, usando:

```
(define (factorial n)
  (printf "(factorial ~a ~n)" n)
  (if (<= n 0) 1
      ; else
      (let ((x (* n (factorial (- n 1)))))
        (printf "(factorial ~a) = ~a ~n" n x)
        x
      )
  ))
```

A seguir está um exemplo do resultado para essa função *factorial* modificada:

```
> (factorial 3)              (factorial 1) = 1
(factorial 3)                (factorial 2) = 2
(factorial 2)                (factorial 3) = 6
(factorial 1)                6
(factorial 0)
```

Isso funciona facilmente porque as funções `define` e `let` possibilitam uma sequência de funções como um corpo e retornam o valor da última função computada.

14.2.8 Exemplo: Semânticas de Clite

Nesta seção, implementamos muitas das semânticas de Clite usando Scheme. Recorde-se do Capítulo 8, no qual foi dito que, para a linguagem elementar Clite, o ambiente é estático; assim o estado pode ser simplesmente representado como uma coleção de pares variável-valor. Isso é expresso da seguinte forma:

$$state = \{\langle var_1, val_1 \rangle, \langle var_2, val_2 \rangle, \dots, \langle var_m, val_m \rangle\}$$

Aqui, cada var_i representa uma variável, e cada val_i representa seu valor atual atribuído.

O estado é como uma *janela de observação* em um ambiente de desenvolvimento integrado IDE (Integrated Development Environment). Ele está sempre ligado a uma instrução particular no programa-fonte, e para cada variável do programa mostra seu valor atual. Em nossa implementação Java, o estado foi implementado como uma tabela *hash* na qual o identificador de variável era a chave e o valor associado era o valor corrente da variável (veja os detalhes no Capítulo 8).

Uma idéia fundamental na implementação Scheme é que um estado é naturalmente representado como uma lista, e cada elemento da lista é um par que representa a ligação de uma variável ao seu valor. Assim, o estado Clite:

$$\{\langle x, 1 \rangle, \langle y, 5 \rangle\}$$

pode ser representado como a lista Scheme:

```
((x 1) (y 5))
```

Primeiro, implementamos as funções de acesso de estado `get` e `union` (união de substituição) da implementação Java (veja o Capítulo 8). Lembre-se de que `get` é usada para obter o valor de uma variável a partir do estado atual. Em Scheme, o código necessário é similar à função `member`:

```
(define (get id state)
  (if (equal? id (caar state)) (cadar state)
      (get id (cdr state))
  ))
```

Como o sistema de tipos Clite requer que todas as variáveis usadas em um programa sejam declaradas, e nós assumimos que a sintaxe abstrata foi estaticamente verificada conforme descrito no Capítulo 6, não pode haver uma referência a uma variável que não esteja no estado. Assim, a função `get` é mais simples do que a função `member` nesse aspecto, já que ela não precisa ser verificada para uma lista nula. Um exemplo da função `get` é:

```
(get 'y '((x 5) (y 3) (z 1)))
= (get 'y '((y 3) (z 1)))
= 3
```

Um bom modelo para a função `onion` é a função `subst` já definida anteriormente, sendo que uma diferença é a estrutura das listas e a outra diferença é que deve haver exatamente uma ocorrência de cada variável no estado:

```
(define (onion id val state)
  (if (equal? id (caar state))
      (cons (list id val) (cdr state))
      (cons (car state) (onion id val (cdr state)))
  ))
```

A função `onion` pode então adotar a hipótese simplificadora de que a variável que estamos buscando ocorre dentro do estado, já que supomos verificação semântica estática conforme foi descrito no Capítulo 6. Portanto, não há necessidade de verificar a existência de uma lista nula como *base case*. A função `onion` pode também interromper a repetição uma vez que tenha sido encontrada a variável. Veja um exemplo da função `onion`:

```
(onion 'y 4 '((x 5) (y 3) (z 1)))
= (cons '(x 5) (onion 'y '((y 3) (z 1)))
= (cons '(x 5) (cons '(y 4) '((z 1))))
= '((x 5) (y 4) (z 1))
```

Ao desenvolver funções semânticas para Clite, assumimos que as instruções em sintaxe abstrata Clite (veja a Figura 2.14) são representadas como listas Scheme da seguinte forma:

```
(skip)
(assignment target source)
(block s1 ... sn)
(loop test body)
(conditional test thenbranch elsebranch)
```

Aqui, cada campo individual para um tipo de instrução abstrata não é nomeado, como era em Java, mas sim identificado por sua posição na lista. Assim, em uma instrução `loop` o `test` é o segundo elemento (isso é, o `cadr`) da lista, enquanto o `body` é o terceiro elemento (o `caddr`).

A função de significado para uma instrução Clite pode ser escrita como uma simples instrução *case* em Scheme:

```
(define (m-statement statement state)
  (case (car statement)
    ((skip) (m-skip statement state))
    ((assignment) (m-assignment statement state))
    ((block) (m-block (cdr statement) state))
    ((loop) (m-loop statement state))
    ((conditional) (m-conditional statement state))
    (else ()))
  ))
```

O propósito de uma *Instrução* abstrata é uma função de transformação de estado da forma que toma um *Estado* como entrada e produz um *Estado* como saída. A implementação

dessas funções de significado resulta diretamente das regras dadas no Capítulo 8 (e resumidas aqui). Assumimos também que foi executada uma verificação semântica estática, conforme descrito no Capítulo 6.

Uma instrução *Skip* corresponde a uma instrução vazia. Como tal, ela deixa o estado inalterado; o estado de saída é uma cópia do estado de entrada.

```
(define (m-skip statement state) state)
```

O propósito de um *Block* é exatamente aquele de suas instruções aplicadas ao estado corrente na ordem em que elas aparecem. Isto é, se um *Block* não tiver nenhuma instrução, o estado não é alterado. Caso contrário, o estado resultante do sentido da primeira *Statement* no *Block* torna-se a base para definir o sentido do resto do bloco. Essa é uma definição repetitiva.

Isso é implementado em Scheme, que interpreta primeiro o significado da primeira instrução na lista, e então aplica repetidamente essa função ao restante da lista. Nós já eliminamos o *block tag* do cabeçalho da lista na função *m-statement*.

```
(define (m-block alist state)
  (if (null? alist) state
      (m-block (cdr alist) (m-statement (car alist) state))
  ))
```

Uma instrução *Loop* tem um teste booleano, que é uma *Expression*, e um corpo *Statement*. Se o teste booleano não resultar *true* (*verdadeiro*), o significado (estado de saída) de um *Loop* será o mesmo que o estado de entrada; caso contrário, o significado será o estado resultante da primeira execução do corpo da instrução uma vez, depois passando o estado resultante para a reexecução do *Loop*.

A implementação Scheme segue quase diretamente dessa definição:

```
(define (m-loop statement state)
  (if (m-expression (car statement) state)
      (m-loop statement (m-statement (cdr statement) state))
      state
  ))
```

Finalmente, considere a função de significado Scheme para avaliação de expressão Clite apenas para inteiros. Para facilitar, usamos a seguinte representação de lista para cada tipo de expressão Clite abstrata:

```
(value val), where val is an integer
(variable id), where id is a variable name
(operator term1 term2), where operator is one of:
  plus, minus, times, div -- arithmetic
  lt, le, eq, ne, gt, ge -- relational
```

A função de significado para uma expressão abstrata Clite é implementada usando um *case* no tipo de expressão. O significado do valor da expressão é exatamente o próprio valor (isto é, o *cadr*). O significado de uma variável é o valor associado com o identificador da variável (o *the cadr*) no estado corrente.

O significado de uma expressão binária é obtido aplicando-se o operador aos significados dos operandos (o `cadr` e o `caddr`):

```
(define (m-expression expr state)
  (case (car expr)
    ((value) (cadr expr))
    ((variable) (get (cadr expr) state))
    (else (applyBinary (car expr) (cadr expr)
                        (caddr expr) state))
  ))
```

A função `applyBinary` definida no Capítulo 8 limitada aos inteiros é facilmente implementada como um *case* no operador. Aqui mostramos apenas os operadores aritméticos, deixando a implementação dos operadores relacionais como exercício:

```
(define (applyBinary op left right state)
  (let ((leftval (m-expression left state))
        (rightval (m-expression right state)))
    (case op
      ((plus)      (+ leftval rightval))
      ((minus)     (- leftval rightval))
      ((times)     (* leftval rightval))
      ((div)       (/ leftval rightval))
      (else #f))
  ))
```

A implementação dos operadores relacionais, bem como a própria instrução de atribuição, fica como exercício.

Como exemplo da aplicação da função `m-expression` à expressão `y+2` no estado $\{ \langle x, 5 \rangle, \langle y, 3 \rangle, \langle z, 1 \rangle \}$, considere o seguinte:

```
(m-expression '(plus (variable y) (value 2)) '((x 5) (y 3) (z 1)))
= (applyBinary '(plus (variable y) (value 2)) '((x 5) (y 3) (z 1)))
= (+ (m-expression '(variable y) '((x 5) (y 3) (z 1)))
     (m-expression '(value 2) '((x 5) (y 3) (z 1))))
= (+ (get 'y '((x 5) (y 3) (z 1)))
     (m-expression '(value 2) '((x 5) (y 3) (z 1))))
= (+ 3
     (m-expression '(value 2) '((x 5) (y 3) (z 1))))
= (+ 3 2)
= 5
```

Esse desenvolvimento, mesmo sendo de uma pequena fração da semântica de Clite, deverá ser suficiente para convencê-lo de que um modelo semântico completo para uma linguagem imperativa pode ser implementado em Scheme.

Assim, por meio da interpretação, Scheme é capaz de calcular qualquer função que possa ser programada em uma linguagem imperativa. O inverso também é verdade, pois os computadores modernos são fundamentalmente imperativos em sua natureza. Como os interpretadores Scheme são implementados nessas máquinas, qualquer função programada em Scheme pode ser computada por um programa imperativo. Portanto, de fato, as linguagens imperativas e as linguagens funcionais são equivalentes quanto ao poder computacional.

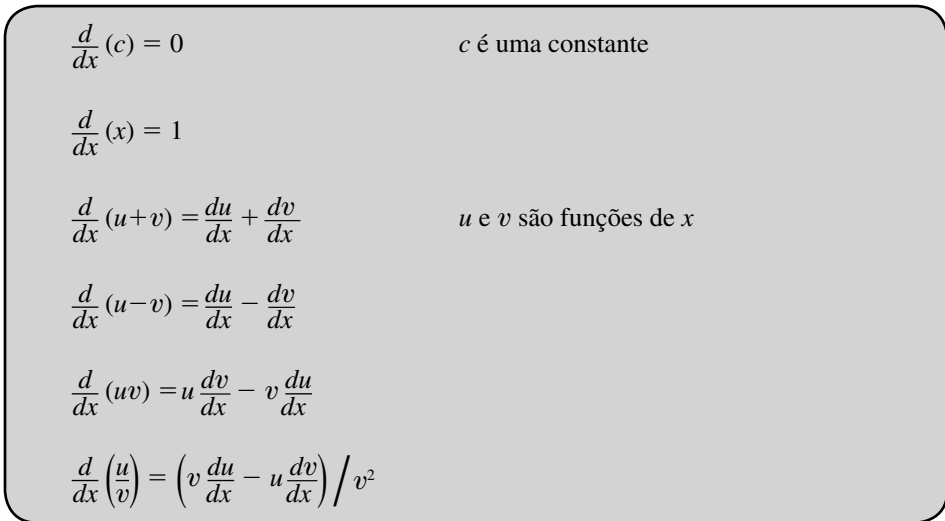
14.2.9 Exemplo: Diferenciação Simbólica

A utilidade da linguagem Scheme para manipulação simbólica é ampla, conforme sugere o exemplo anterior. O exemplo seguinte ilustra ainda mais parte do poder de Scheme fazendo diferenciação simbólica e simplificação de fórmulas simples de cálculo. Algumas regras familiares para diferenciação simbólica são dadas na Figura 14.2.

Por exemplo, diferenciando-se a função $2 \cdot x + 1$ em relação ao x usando essas regras, obtemos:

$$\begin{aligned}\frac{d(2 \cdot x + 1)}{dx} &= \frac{d(2 \cdot x)}{dx} + \frac{d1}{dx} \\ &= 2 \cdot \frac{dx}{dx} + x \cdot \frac{d2}{dx} + 0 \\ &= 2 \cdot 1 + x \cdot 0 + 0\end{aligned}$$

que comumente se simplifica resultando no valor 2.



$$\begin{aligned}\frac{d}{dx}(c) &= 0 && c \text{ é uma constante} \\ \frac{d}{dx}(x) &= 1 \\ \frac{d}{dx}(u+v) &= \frac{du}{dx} + \frac{dv}{dx} && u \text{ e } v \text{ são funções de } x \\ \frac{d}{dx}(u-v) &= \frac{du}{dx} - \frac{dv}{dx} \\ \frac{d}{dx}(uv) &= u \frac{dv}{dx} + v \frac{du}{dx} \\ \frac{d}{dx}\left(\frac{u}{v}\right) &= \left(v \frac{du}{dx} - u \frac{dv}{dx}\right) / v^2\end{aligned}$$

| Figura 14.2 Regras de Diferenciação Simbólica

Em Scheme, é conveniente representar expressões usando a notação Polish prefix:

```
(+ term1 term2)
(- term1 term2)
(* term1 term2)
(/ term1 term2)
```

A função requerida para fazer a diferenciação simbólica primeiro testa, para determinar se a expressão é uma constante ou a variável que está sendo diferenciada, como nas duas primeiras regras descritas. Caso contrário, a expressão é uma lista que começa com um operador, e o código aplica uma das quatro regras restantes usando um *case* no operador. Foi usada uma função `let` para fazer o código Scheme ficar o mais semelhante possível às regras da Figura 14.2.

```
(define (diff x expr)
  (if (not (list? expr))
      (if (equal? x expr) 1 0)
      (let ((u (cadr expr)) (v (caddr expr)))
        (case (car expr)
          ((+) (list '+ (diff x u) (diff x v)))
          ((-) (list '- (diff x u) (diff x v)))
          ((* ) (list '*
                     (list '* u (diff x v))
                     (list '* v (diff x u))))
          ((/) (list 'div (list '-
                               (list '* v (diff x u))
                               (list '* u (diff x v)))
                     (list '* u v))))
      )))
```

Uma aplicação da função `diff` à expressão $2 \cdot x + 1$ resulta em:

```
(diff 'x '(+ (* 2 x) 1))
= (list '+ (diff 'x '(* 2 x)) (diff 'x 1))
= (list '+ (list '+ (list '* 2 (diff 'x 'x))
                  (list '* x (diff 'x 2))))
      (diff 'x 1))
= (list '+ (list '+ (list '* 2 1) (list '* x (diff 'x 2)))
      (diff 'x 1))
= (list '+ (list '+ '(* 2 1) (list '* x (diff 'x 2)))
      (diff 'x 1))
= (list '+ (list '+ '(* 2 1) (list '* x 0)) (diff 'x 1))
= (list '+ (list '+ '(* 2 1) '(* x 0)) (diff 'x 1))
= (list '+ '(+ '(* 2 1) '(* x 0)) (diff 'x 1))
= (list '+ '(+ '(* 2 1) '(* x 0)) 0)
```

que, na forma interfixada, é $2 * 1 + 0 * x + 0$. O resultado normal, 2, apareceria após simplificar essa expressão. Fica como exercício escrever um simplificador de expressão.

14.2.10 Exemplo: O Problema das Oito Rainhas

No Capítulo 13, desenvolvemos uma versão orientada a objetos do algoritmo *backtracking* de Wirth (Wirth, 1976). A solução geral que Wirth apresenta é um procedimento repetitivo que executa uma iteração por meio de uma série de movimentos. Cada movimento é testado para ver se ele satisfaz a um critério apropriado de *validade*. Se ele satisfizer, o movimento será registrado, e se o problema ainda não estiver resolvido, o procedimento chamará a si próprio repetidamente para tentar o próximo nível. Se a chamada repetitiva falhar, então o movimento atual será desfeito e será tentado o próximo movimento. O processo de tentar movimentos continua até que seja encontrada uma solução completamente satisfatória ou todos os movimentos no nível atual tenham sido tentados sem sucesso.

A solução iterativa geral para esse problema, de acordo com Wirth (1976, p. 136), pode ser conseguida na seguinte notação semelhante a C:

```
boolean try(Solution) {
    boolean successful = false;
    initialize moves;
    while (more moves && !successful) {
        select next move;
        if (move is valid) {
            record move in Solution;
            if (Solution not done) {
                successful = try(Solution);
                if (!successful)
                    undo move in Solution;
            }
        }
    }
    return successful;
}
```

Esta seção desenvolve uma versão puramente funcional desse algoritmo e então especializa-o para resolver o problema das oito rainhas. Essa solução é interessante porque mostra alguns dos aspectos positivos e negativos da programação funcional, em contraste com a programação imperativa e orientada a objetos.

Há efetivamente dois problemas que devemos resolver na conversão desse modelo geral, imperativo, em outro puramente funcional. O primeiro problema é que a função `try` retorna dois resultados: se foi ou não encontrada uma solução de forma bem-sucedida e a própria solução.

Na versão imperativa descrita acima, a função `try` retorna `successful` como valor da função, e é retornada a solução (*Solution*) como parâmetro de referência: os comandos *record* e *undo* são atribuições generalizadas implementadas como chamadas de função. Uma linguagem puramente funcional não tem parâmetros e atribuições de referência, nem pode retornar mais do que um único resultado. Para resolver esse problema em Scheme, nós podemos `cons` o valor da variável `successful` destacar a própria solução.

O segundo problema é que o modelo de Wirth usa um laço *while* para iterar através de uma série de movimentos até que seja encontrada uma solução completamente bem-sucedida ou não haja mais movimentos nesse nível. Uma codificação Scheme direta desse problema poderia usar as características imperativas de Scheme para essencialmente duplicar

o algoritmo de Wirth. No entanto, para apresentar um equivalente puramente funcional à estrutura imperativa de laço, devemos substituir o laço por uma função repetitiva.

Desenvolvemos esse algoritmo um tanto às avessas, dependendo do seu ponto de vista. Isso significa que atacaremos as instruções *if* internas primeiro, depois o laço *while*, e finalmente a função global. Funções particulares ao problema atual que está sendo resolvido, como quando a solução está esgotada quando não há mais movimentos, e assim por diante, permanecerão não especificadas por enquanto.

A primeira função, chamada *tryone*, foi desenvolvida para resolver a instrução *se o movimento for válido* no modelo acima:

```
(define (tryone move soln)
  (let ((xsoln (cons move soln)))
    (if (valid move soln)
        (if (done xsoln) (cons #t xsoln)
            (try xsoln))
        (cons #f soln)
    )))
```

A função *tryone* só é chamada se a variável *successful* for *false*; assim, o parâmetro *soln* não tem o valor de *successful* no início da lista. No entanto, ela retorna uma solução com o valor de *successful* no início da lista.

Observe que usamos uma função *let* de uma forma restrita para evitar computar a solução estendida *xsoln* duas vezes. A função primeiro verifica se o movimento atual é válido, dada a solução parcial atual. Se ele não for válido, então ele retorna *false* para *successful* e a solução corrente via *(cons #f soln)*. Se o movimento for válido, então ele verifica se a solução estendida resolve o problema (função *done*). Nesse caso, ele retorna *true* (verdadeiro) para *successful* e a solução estendida. Caso contrário, ele chama repetidamente *try* com a solução estendida para tentar continuar estendendo a solução.

Em seguida, convertemos o laço *while* em uma função repetitiva, usando a estratégia a seguir. Isto é, qualquer laço *while* imperativo da forma:

```
while (test) {
  body
}
```

pode ser convertido em uma função repetitiva da forma:

```
(define (while test body state)
  (if (test state)
      (let ((onepass (body state)))
        (while test body onepass)
      )
      state
  ))
```

Aqui, a variável *onepass* fornece o estado que resulta ao executar o corpo do laço *while* uma vez. Assim, se o teste resulta em *true* (verdadeiro), a instrução *while* é executada novamente após fazer uma passagem pelo corpo do laço *while*. Caso contrário, é retornado o estado atual.

O estado do programa para as oito rainhas é um pouco mais complicado, mas uma estratégia geral de conversão de laço produz a seguinte função:

```
(define (trywh move soln)
  (if (and (hasmore move) (not (car soln)))
      (let ((atry (tryone move (cdr soln))))
        (if (car atry) atry (trywh (nextmove move) soln)))
      )
      soln
  ))
```

Observe que a função `let` aparece de forma restrita para evitar escrever a chamada à função `tryone` duas vezes. Observe também que a função `trywh` espera que o valor da variável `successful` esteja no início da lista, enquanto a função `tryone` não, já que `tryone` só é chamada quando `successful` é *false* (*falso*).

Finalmente, implementamos a função `try`. Ela é chamada com uma solução parcial sem a variável `successful`, retorna a `cons` da variável `successful` e assim é encontrada a solução. Ela é responsável por obter o primeiro movimento para inicializar o laço *while*:

```
(define (try soln) (trywh 1 (cons #f soln)))
```

Para especificar essa estratégia geral e resolver um problema em particular, devemos implementar várias funções.

- As funções `hasmore` e `nextmove` servem para gerar testes de movimentos.
- A função `valid` verifica se um teste de movimento estende de forma válida a solução parcial corrente.
- A função `done` testa para determinar se uma solução estendida resolve o problema.

Ilustramos implementações dessas funções desenvolvendo uma solução para o problema das oito rainhas.

Uma preocupação inicial no problema das oito rainhas é a de como armazenar a posição (linha e coluna) de cada uma das rainhas. Recorde-se da Seção 13.4.2, em que desenvolvemos a solução uma coluna de cada vez, armazenando a posição da linha para cada coluna usando uma matriz. Na solução desenvolvida aqui, armazenamos a posição da linha para cada coluna usando uma lista, mas com uma diferença fundamental. Armazenamos a lista em ordem inversa, de maneira que a linha acrescentada mais recentemente fica sempre no início da lista. Por exemplo, o tabuleiro com três rainhas nas posições (linha, coluna) mostradas na Figura 14.3 é representado como a lista a seguir:

```
(5 3 1)
```

Se a variável `N` representa o número de linhas e colunas do tabuleiro de xadrez, podemos definir as funções para gerar movimentos como:

```
(define (hasmore move) (<= move N))
(define (nextmove move) (+ move 1))
```

Figura 14.3
Três Rainhas em um
Tabuleiro de Xadrez 8×8

Q							
	Q						
		Q					

que gera números de linha na sequência de 1 a N . Semelhantemente, podemos definir uma solução a ser “feita” da seguinte maneira:

```
(define (done soln) (>= (length soln) N))
```

Agora, tudo o que resta é definir se um teste de fileira estende ou não a solução parcial atual de forma válida. Lembre-se do Capítulo 13, no qual é dito que devem ser satisfeitas três condições:

- 1 A fileira testada não deve estar ocupada. Isso significa que a fileira testada (ou o movimento) não deve ser um membro da solução atual.
- 2 A diagonal sudoeste formada pela fileira e coluna testada não deve estar ocupada. A diagonal sudoeste é a soma dos números de linha e coluna.
- 3 A diagonal sudeste formada pela fileira e coluna não deve estar ocupada. A diagonal sudeste é a diferença dos números de linha e coluna.

Dado um número de linha e coluna, as diagonais sudoeste e sudeste são facilmente calculadas como:

```
(define (swDiag row col) (+ row col))
(define (seDiag row col) (- row col))
```

Para testar uma solução, devemos primeiro converter uma lista de posições de linhas em uma lista de posições diagonais sudoeste e sudeste. Para um dado teste `soln` a posição da linha do teste de movimento é `(car soln)` e o número de coluna associado é `(length soln)`. As funções `selist` e `swlist` desenvolvem essas listas para qualquer teste de solução.

```
(define (selist alist)
  (if (null? alist)
      '()
      (cons (seDiag (car alist) (length alist))
            (selist (cdr alist)))))
(define (swlist alist)
  (if (null? alist)
      '()
      (cons (swDiag (car alist) (length alist))
            (swlist (cdr alist)))))
```

Finalmente, as três condições para o teste de solução podem ser testadas por meio da função `valid`. Essa função verifica se um teste de movimento corrente representando a posição de uma linha estende, de forma válida, a solução parcial atual. Isto é, o movimento (linha) não é um membro da solução, e o movimento (posição de linha e coluna associada) não é um membro da diagonal sudoeste nem da diagonal sudeste.

```
(define (valid move soln)
  (let ((col (length (cons move soln))))
    (and (not (member move soln))
         (not (member (seDiag move col) (selist soln)))
         (not (member (swDiag move col) (swlist soln)))
        )))
```

Esse programa pode ser testado usando a chamada `(try ())`, na qual a variável global `N` define o tamanho do problema. Por exemplo, a declaração:

```
(define N 8)
```

irá particularizar a solução para um tabuleiro 8×8 . Isso encerra nossa implementação funcional do *backtracking* e do problema das oito rainhas.

Esse exercício foi interessante por várias razões. Por um lado, ele mostra o poder da programação funcional. Por outro lado, nossa solução mostra algumas das fraquezas da programação funcional pura:

- 1 A conversão de um programa que tenha um laço iterativo, em uma função repetitiva, pode ser desnecessariamente tediosa.
- 2 O uso de uma lista para retornar múltiplos valores de uma função é inapropriado quando comparado ao uso de parâmetros de referência ou ao retorno de um objeto com variáveis de instância nomeadas.

Para compensar o primeiro ponto fraco (e também em favor da eficiência), Scheme estende a “Lisp pura” incluindo características imperativas tais como variáveis locais, instruções de atribuição e laços iterativos. O segundo ponto fraco é realmente uma fraqueza do sistema de tipo de Scheme, um problema que está substancialmente corrigido em linguagens funcionais posteriores como a Haskell. Discutiremos o sistema de tipo de Haskell na próxima seção.

14.3 HASKELL

Alguns desenvolvimentos recentes em programação funcional não são bem absorvidos pelas linguagens tradicionais, Common Lisp e Scheme. Nesta seção, introduzimos uma linguagem funcional mais moderna, Haskell (Thompson, 1999), cujas características assinalam mais claramente as direções presente e futura na pesquisa e nas aplicações de programação funcional. As características distintas e salientes de Haskell incluem sua estratégia de avaliação lenta e seu sistema de tipo. Embora Haskell seja uma linguagem fortemente tipada (todos os erros de tipo são identificados), às vezes um erro de tipo não é detectado até que o elemento do programa que contenha o erro seja realmente executado.

14.3.1 Introdução

Haskell tem uma sintaxe simples para escrever funções. Considere a função fatorial, que pode ser escrita em qualquer uma das maneiras a seguir:

```
-- equivalent definitions of factorial
fact1 0 = 1
fact1 n = n * fact1 (n - 1)

fact2 n = if n == 0 then 1 else n * fact2 (n - 1)

fact3 n
| n == 0 = 1
| otherwise = n * fact3 (n - 1)
```

Um hífen duplo (`--`) inicia um comentário Haskell, que continua até o fim da linha. A primeira versão, `fact1`, é escrita em um estilo recursivo, de modo que os casos especiais são definidos primeiro e seguidos pelo caso geral. A segunda versão, `fact2`, usa o estilo de definição mais tradicional `if-then-else`. A terceira versão, `fact3`, usa *guards* em cada direção; esse estilo é útil quando há mais de duas alternativas. Nos exemplos de aplicações, serão usados os três estilos.

Note a simplicidade da sintaxe. Diferentemente de Scheme, não há uma `define` introduzindo a definição da função, não há parênteses envolvendo os argumentos formais, não há vírgulas separando os argumentos. Além disso, não há um símbolo explícito de continuação (como em programação Unix shell) nem um terminador explícito (o ponto-e-vírgula em C/C++/Java). Em lugar disso, como em Python, Haskell acredita no recuo de construções continuadas. Em `fact3`, como ela é escrita sobre mais de uma linha, as *guards* ou a tecla pipe (`|`) devem ficar todas com o mesmo afastamento. Porém, os sinais de igual não precisam ficar alinhados. Uma definição extensa para uma *guard* convencionalmente começaria em uma nova linha e ficaria afastada do símbolo *guard*.

Em Haskell, os argumentos para uma função não ficam entre parênteses, tanto na definição da função quanto na sua invocação. Além disso, a invocação de função se liga mais fortemente do que operadores interfixados. Desse modo, a interpretação normal de `fact n - 1` é $\text{fact}(n) - 1$, que não é o que se deseja. Então, o valor `n - 1` deve ser colocado entre parênteses, já que é um único argumento para `fact` nas três variantes. A grandeza matemática $\text{fact}(n-1) * n$ seria escrita como:

```
fact (n - 1) * n
```

em que os parênteses são necessários, de maneira que o valor `n - 1` seja interpretado como o único argumento para `fact`.

Haskell é sensível a maiúsculas/minúsculas. Funções e variáveis devem começar com uma letra minúscula, enquanto tipos começam com uma letra maiúscula. Além disso, uma função não pode redefinir uma função padrão Haskell. Conforme veremos, as funções em Haskell são fortemente tipadas e polimórficas.

E, também, como a maioria das linguagens funcionais, Haskell usa, por padrão, inteiros de precisão infinita:

```
> fact2 30
265252859812191058636308480000000
```

uma resposta que claramente excede o maior valor `int` em um programa C/C++/Java.

Em Haskell, como em qualquer linguagem funcional, funções são objetos de primeira classe, em que funções não-avaliadas podem ser passadas como argumentos, construídas e retornadas como valores de funções. Além disso, funções podem ser *restringidas* quando um argumento n de função puder ter alguns de seus argumentos fixados. Uma *função restringida* é uma função de n argumentos, na qual alguns de seus argumentos são fixos. Como exemplo desse último caso, suponha que queremos definir uma função que dobra seu argumento:

```
double1 x = 2 * x
double2 = (2 *)
```

As funções `double1` e `double2` são equivalentes; a segunda é um exemplo de uma função restringida.

Com essa rápida introdução, vamos começar uma exploração mais sistemática de Haskell.

14.3.2 Expressões

As expressões em Haskell normalmente são escritas em notação interfixada, na qual o operador ou a função aparece entre seus operandos, como no exemplo a seguir:

```
2+2      -- compute the value 4
```

Quando essa expressão é apresentada a um interpretador Haskell, ele calcula o valor 4. Há os operadores usuais aritmético e relacional em Haskell, e podem ser criadas operações mais complicadas usando parênteses e as relações internas de precedência entre esses operadores. Veja, por exemplo, uma expressão Haskell que calcula o valor 48:

```
5*(4+6)-2
```

que seria equivalente à expressão Scheme `(- (* 5 (+ 4 6) 2))`. Além disso, podemos escrever expressões Haskell usando notação prefixada, desde que coloquemos parâmetros em todos os operadores e operandos disjuntos (*nonatomic*). Isso é ilustrado pela seguinte expressão (equivalente à expressão interfixada acima):

```
(-) ((*) 5 ((+) 4 6)) 2
```

Na Tabela 14.1 há um resumo mais completo dos operadores Haskell e das suas relações de precedência.

Os operadores *right-associative* (associativos à direita) são avaliados da direita para a esquerda quando eles são adjacentes em uma expressão no mesmo nível de parênteses; os operadores *left-associative* (associativos à esquerda) são avaliados da esquerda para a direita. Por exemplo, a expressão Haskell

```
2^3^4
```

representa 2 elevado à potência 3^4 (ou 2^{81} ou 2417851639229258349412352), e não 2^3 elevado à quarta potência (ou 2^{12} ou 4096). Os operadores não-associativos não podem aparecer adjacentes em uma expressão. Isso é, a expressão $a+b+c$ é permitida, mas $a<b<c$ não é.

| **Tabela 14.1** Resumo dos Operadores Haskell e suas Precedências

Precedência	Associativo à Esquerda	Não-Associativo	Associativo à Direita
9	!, !!, //		.
8			**, ^, ^ ^
7	*, /, 'div', 'mod', 'rem', 'quot'		
6	+, -	:+	
5		\\	:, ++
4		/=, <, <=, ==, >, >=, 'elem', 'notElem'	
3			&&
2			
1	>>, >=>	:=	
0			\$, 'seq'

O significado de muitos desses operadores deveria ser auto-explicativo. Muitos outros serão explicados na discussão seguinte.⁸

14.3.3 Listas e Extensões de Listas

Assim como Lisp e Scheme, a estrutura fundamental de dados de Haskell é a lista. Listas são coleções de elementos de certo tipo e podem ser definidas pela enumeração de seus elementos, como mostram as definições a seguir, para duas pequenas listas de números:

```
evens = [0, 2, 4, 6, 8]
odds  = [1, 3 .. 9]
```

A lista `odds` é definida pela convenção matemática familiar usando reticências (`..`) para omitir todos os elementos intermediários quando o padrão for óbvio. Graficamente, a lista `evens` é representada na Figura 14.1(a).

Alternativamente, uma lista pode ser definida por intermédio de algo chamado gerador, que toma a seguinte forma:

```
moreevens = [2*x | x <- [0..10]]
```

Isso significa, literalmente, “a lista de todos os valores $2 \times x$ tais que x é um elemento na lista `[0..10]`”. O operador `<-` representa o símbolo matemático \in , que representa a participação na lista.

8. Haskell também suporta a definição de operadores adicionais, desde que eles sejam formados a partir dos seguintes símbolos: `# $ % * + . / < = ? \ ^ | | : ~`.

Uma *extensão* de lista pode ser definida se usarmos um gerador, e a lista que ele define pode ser infinita. Por exemplo, a linha a seguir define a lista infinita que contém todos os inteiros pares não-negativos:

```
mostevens = [2*x | x <- [0,1 .. ]]
```

Aqui, o gerador é a expressão `x <- [0,1 ..]`. Alternativamente, essa lista infinita poderia ter sido definida por:

```
mostevens = [0,2 .. ]
```

Esse exemplo ilustra uma grande diferença entre a Haskell e as linguagens funcionais tradicionais. Listas infinitas e as funções que calculam valores a partir delas são comuns em Haskell. Elas são possíveis devido ao compromisso geral da Haskell com a *avaliação lenta*, que diz simplesmente para não avaliar nenhum argumento para uma função até o momento em que for absolutamente necessário.⁹ Para listas infinitas, isso significa que elas são armazenadas na forma não-avaliadas; no entanto, o *n*-ésimo elemento, não importa quão grande seja o valor de *n*, pode ser calculado sempre que for necessário.

Os geradores podem ter condições anexadas a eles, assim como na matemática. A função a seguir calcula os fatores de um número:

```
factors n = [ f | f <- [1..n], n `mod` f == 0 ]
```

Isso pode ser lido assim: os fatores de *n* são todos os números *f* no intervalo de um para *n* tal que *f* divide *n* exatamente. Observe que a expressão:

```
n `mod` f == 0
```

poderia ser escrita como:

```
mod n f == 0
```

Quando o nome de uma função é usado como um operador interfixado, o nome deve ser colocado entre caracteres delimitadores.

A função básica para construir uma lista é o operador interfixado `:`, que toma um elemento e uma lista como seus dois argumentos.¹⁰ Aqui estão alguns exemplos, nos quais `[]` representa a lista vazia:

```
8:[]      -- gives [8]
6:8:[]    -- gives 6:[8] or [6,8]
4:[6,8]   -- gives [4,6,8]
```

Uma lista Haskell tem duas partes: o primeiro elemento ou a *head* da lista, e a lista dos demais elementos ou seu final (*tail*). As funções `head` e `tail` retornam essas duas

9. Lembre-se da distinção entre avaliação “rápida” e “lenta” feita pela primeira vez no Capítulo 9, no qual foi discutida a passagem de parâmetros. Deve ficar claro que a avaliação rápida de argumentos para uma função proibiria a definição de listas infinitas ou funções que operam sobre elas.

10. Esse operador é semelhante à função `cons` de Scheme.

partes, respectivamente. Referindo-se à lista `evens` representada na Figura 14.1, os exemplos a seguir ilustram essas funções:

```
head evens           --gives 0
tail evens           --gives [2,4,6,8]
head (tail evens)    --gives 2
tail (tail evens)    --gives [4,6,8]
head [6,8]           --gives 6
head 6:[8]           --gives 6
tail [6,8]           --gives [8]
tail [8]             --gives []
```

Combinando geradores e encadeamento de lista, pode ser definida a série de números primos usando a função `sieve`:

```
primes = sieve [2..]
  where
    sieve (p:xs) = p : sieve [ a | a <- xs, a `mod` p /= 0 ]
```

Primeiro, observe o uso da cláusula `where`, que torna a definição de `sieve` local à definição de `primes` (análogo ao uso de `let` em Scheme ou na matemática). A definição diz que a lista de primos até `n` é retornada por `sieve` na lista de números de dois até `n`. A função `sieve`, dada uma lista que consiste de uma `head` `p` e um final `xs` (que é uma lista), é constituída da lista cuja `head` é `p` (que deve ser primo) e cujo final é o valor de `sieve` aplicado a `xs` com todos os múltiplos de `p` removidos. A segunda definição de `sieve` é um exemplo de reconhecimento de padrão e é explicada na Seção 14.3.6.

O principal operador para juntar listas é `++`.¹¹ Esse operador é ilustrado pelos exemplos a seguir:

```
[1,2]++[3,4]++[5]    -- gives [1,2,3,4,5]
evens ++ odds         -- gives [0,2,4,6,8,1,3,5,7,9]
[1,2]++[]             -- gives [1,2]
[1,2]++3:[]           -- gives [1,2,3]
1++2                  -- error; wrong type of arguments for ++
```

Como a Haskell foi projetada para processar listas, ela contém um conjunto de funções especiais de processamento de lista. Devido à necessidade que aparece frequentemente, há uma função especial `null` para testar quanto a uma lista vazia:

```
null []              -- gives True
null evens            -- gives False
null [1,2,3]         -- gives False
null 5                -- error; wrong type of argument for null
```

A Haskell contém funções para testar se um objeto é igual ou equivalente a outro. A função principal está incorporada no operador interfixado `==`, que é razoavelmente geral.

11. Esse operador é semelhante à função `append` de Scheme.

Essa função retorna `True` se os dois objetos tiverem a mesma estrutura e o mesmo conteúdo; caso contrário, ela retorna `False` ou um erro de tipo:

```
5==5           -- returns True
5==1           -- returns False
[1,2]==[1,2]   -- returns True
5==[5]         -- error; mismatched argument types
[1,2,3]==[1,[2,3]] -- error; mismatched argument types
[1,2]==[2,1]   -- returns False
[]==[]         -- returns True
```

Tipos lista podem ser definidos e, mais tarde, usados na construção de funções. Para definir um tipo lista `IntList` de valores `Int`, por exemplo, é usada a seguinte instrução:

```
type IntList = [Int]
```

Observe que o uso de sinais de parênteses assinala que o tipo que está sendo usado é uma espécie particular de lista – uma lista cujas entradas são do tipo `Int`.¹²

14.3.4 Tipos e Valores Elementares

Até aqui, todos os valores que vimos em Haskell são inteiros, símbolos predefinidos (nomes de função) e nomes de tipos. Haskell suporta vários tipos de valores elementares, incluindo booleanos (chamados de `Bool`), inteiros (`Int` e `Integer`), caracteres (`Char`), cadeias de caracteres (`String`) e números em ponto flutuante (`Float`).

Conforme já observamos, os valores booleanos são `True` e `False`. O tipo `Int` suporta um intervalo finito de valores (-2^{31} até $2^{31} - 1$, que é o intervalo usual para representação em 32 bits). No entanto, o tipo `Integer` suporta inteiros de qualquer tamanho, e, portanto, contém uma série infinita de valores.

Caracteres em Haskell são representados entre aspas simples, como `'a'`, e são usadas convenções familiares de escape para identificar caracteres especiais, como `'\n'` para nova linha, `'\t'` para tabulação, e assim por diante. As *strings* são representadas como uma série de caracteres entre aspas duplas (`"`) ou uma lista de valores `Char`. Isto é, o tipo `String` é equivalente ao tipo `[Char]`. Assim a lista `['h','e','l','l','o']` é equivalente à `String` `"hello"`. Ou seja, a seguinte definição de tipo está implícita em Haskell:

```
type String = [Char]
```

Devido a essa equivalência, muitos operadores `String` são o mesmo que operadores lista. Por exemplo, a expressão

```
"hello" ++ "world"
```

representa encadeamento de *string*, e resulta em `"helloworld"`.

12. Os nomes de tipos Haskell são diferentes de outros nomes pelo fato de começarem com letra maiúscula.

Valores em ponto flutuante são escritos em notação decimal ou notação científica. Cada um dos seguintes valores representa o número 3.14.

```
3.14
0.000314e4
```

Há várias funções disponíveis para transformar valores em ponto flutuante em Haskell, incluindo os seguintes, cujos significados são razoavelmente auto-explicativos (argumentos para funções trigonométricas são expressos em radianos):

```
abs acos atan ceiling floor cos sin
log logBase pi sqrt
```

14.3.5 Fluxo de Controle

Os principais construtores de controle de fluxo em Haskell são os comandos *guarded* e o *if-then-else*.¹³ O comando *guarded* é uma generalização de um *if-then-else* generalizado, e pode ser escrito mais abreviadamente. Por exemplo, suponha que queremos encontrar o máximo de três valores, *x*, *y* e *z*. Então podemos expressar isso como um *if-then-else* da seguinte maneira:

```
if x >= y && x >= z then x
else if y >= x && y >= z then y
else z
```

Alternativamente, podemos expressar isso como um comando *guarded* da seguinte maneira:

```
| x >= y && x >= z = x
| y >= x && y >= z = y
| otherwise = z
```

O comando *guarded* é muito usado quando se definem funções Haskell, conforme veremos a seguir.

14.3.6 Definindo Funções

As funções Haskell são definidas em duas partes. A primeira parte identifica o nome da função, do domínio e do intervalo, e a segunda parte descreve o significado da função. Assim, uma definição de função tem a seguinte forma:

```
name :: Domain -> Range
name x y z
    | g1 = e1
    | g2 = e2
    :
    | otherwise = e
```

13. Haskell tem também uma função *case*, que é similar à *case* em Ada e a *switch* em Java e C. No entanto, essa função parece ser relativamente sem importância em programação Haskell, já que seu significado é *subsumed* pelo comando *guarded*.

Aqui, o corpo da função é expresso como um comando *guarded*, e o domínio e o intervalo podem ser de quaisquer tipos. Por exemplo, uma função máxima para três inteiros pode ser definida como:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z = x
  | y >= x && y >= z = y
  | otherwise       = z
```

A primeira linha em uma definição de função pode ser omitida, caso em que Haskell deriva aquela linha automaticamente, dando à função a interpretação mais ampla possível. Quando omitimos a primeira linha da definição acima, Haskell deriva o seguinte para ela:

```
max3 :: Ord a => a -> a -> a -> a
```

Essa notação significa que se *a* é *qualquer* tipo ordenado (*Ord*), então o significado de *max3* é explicado pela definição acima. Um *tipo ordenado* em Haskell é qualquer tipo que aceita os operadores relacionais (*==*, *!=*, *>=*, *>*, *<=* e *<*), que permite ordenar seus valores individuais. Assim, nossa função *max3* está agora bem definida em argumentos que são *Int*, *Float*, *String* ou qualquer outro tipo cujos valores são ordenados, conforme está ilustrado nos exemplos a seguir:

```
> max3 6 4 1
6
> max3 "alpha" "beta" "gamma"
"gamma"
```

A função *max3* é um exemplo de uma função *polimórfica*.

Definição: Uma *função polimórfica* é aquela cuja definição se aplica igualmente bem a argumentos de vários tipos, dentro das restrições dadas pela função assinatura.

Por exemplo, uma tentativa de calcular o valor máximo entre uma *string* e dois inteiros dá o seguinte erro em tempo de execução:

```
> max3 "a" 2 3
ERROR: Illegal Haskell 98 class constraint
*** Expression : max3 "a" 2 3
*** Type       : Num [Char] => [Char]
```

Funções em Haskell podem ser definidas com o uso de recursão ou iteração. Aqui está uma definição de função fatorial recursiva:

```
fact :: Integer -> Integer
fact n
  | n == 0 = 1
  | n > 0  = n*fact(n-1)
```

Aqui está sua parte que calcula de modo iterativo (`product` é uma função interna):

```
fact n = product[1..n]
```

Tarefas iterativas simples, como somar uma lista de números, também podem ser escritas recursivamente ou iterativamente em Haskell. A função a seguir também ilustra como Haskell usa reconhecedor de padrão para distinguir diferentes estruturas e partes de lista.

```
mysum []      = 0
mysum (x:xs)  = x + mysum xs
```

A primeira linha define `mysum` para o caso especial (base) em que a lista está vazia. A segunda linha define `mysum` para os outros casos, nos quais a lista é não-vazia – lá, a expressão `x:xs` define um padrão que separa o head (`x`) do resto (`xs`) da lista, de maneira que eles podem ser distinguidos na definição da função. Então, o reconhecedor de padrão proporciona uma alternativa para escrever um comando geral ou uma instrução `if-then-else` para distinguir casos alternativos na definição da função.

Essa função soma é definida em qualquer lista de valores numéricos – `Int`, `Integer`, `Float`, `Double` ou `Long`. Aqui estão alguns exemplos, com seus resultados:

```
> mysum [3.5,5]
8.5
> mysum [3,3,4,2]
12
```

Haskell fornece um conjunto de funções transformadoras de lista em sua biblioteca-padrão chamada `Prelude`. A Tabela 14.2 fornece um resumo dessas funções. Nessas e em outras definições de função, a notação `a` ou `b` significa “qualquer tipo de valor”. Isto é, as várias funções na Tabela 14.2 são polimórficas até a extensão especificada em seu domínio e intervalo. Por exemplo, o tipo do elemento da lista passado para a função `head` não afeta seu significado.

Outra função simples é a função `member`, que testa para saber se um elemento ocorre como um membro de uma lista e retorna `True` ou `False` de forma correspondente. A definição começa com uma lista vazia como caso básico e retorna `False`. Caso contrário, ela testa para saber se o *head* da lista é igual ao elemento procurado; se for, ela retorna `True` ou, caso contrário, ela retorna o resultado da chamada repetitiva a si própria no final da lista.

```
member :: Eq a => [a] -> a -> Bool
member alist elt
  | alist == []      = False
  | elt == head alist = True
  | otherwise        = member (tail alist) elt
```

Uma maneira alternativa de definir funções em Haskell é explorar seus recursos de reconhecedor de padrão. Considere o seguinte:

```
member [] elt      = False
member (x:xs) elt  = elt == x || member xs elt
```

| **Tabela 14.2** Algumas Funções Lista Comuns em Haskell

Função	Domínio e Intervalo	Explicação
:	<code>a -> [a] -> [a]</code>	Acrescenta um elemento no início de uma lista
++	<code>[a] -> [a] -> [a]</code>	Junta (concatena) duas listas unidas
!!	<code>[a] -> Int -> a</code>	<code>x !! n</code> retorna o n-ésimo elemento da lista <code>x</code>
length	<code>[a] -> Int</code>	Número de elementos em uma lista
head	<code>[a] -> a</code>	Primeiro elemento em uma lista
tail	<code>[a] -> [a]</code>	Todos os elementos da lista, exceto o primeiro
take	<code>Int -> [a] -> [a]</code>	Toma <code>n</code> elementos do início de uma lista
drop	<code>Int -> [a] -> [a]</code>	Retira <code>n</code> elementos do início de uma lista
reverse	<code>[a] -> [a]</code>	Inverte a ordem dos elementos de uma lista
elem	<code>a -> [a] -> Bool</code>	Verifica se um elemento ocorre em uma lista
zip	<code>[a] -> [b] -> [(a,b)]</code>	Faz uma lista de pares *
unzip	<code>[(a,b)] -> ([a],[b])</code>	Faz um par de listas
sum	<code>[Int] -> Int</code> <code>[Float] -> Float</code>	Soma os elementos de uma lista
product	<code>[Int] -> Int</code> <code>[Float] -> Float</code>	Multiplica os elementos de uma lista

*Um “par” é um exemplo de uma *tupla* em Haskell (veja a Seção 14.3.7).

Essa definição alternativa é equivalente à primeira. Sua segunda linha combina as duas últimas linhas da primeira definição, aproveitando as vantagens da função disjunção (`||`) pela avaliação lenta de seus argumentos.

Nos exemplos anteriores, os argumentos para uma função são valores simples ou listas. Haskell também permite que uma função seja um argumento para outra função que está sendo definida:

```
flip f x y = f y x
```

Nesse caso, a função `flip` toma uma função `f` como argumento; `flip` então chama `f` com seu argumento invertido. Um dos usos de `flip` é definir `member` em termos de `elem`:

```
member xs x = elem x xs
member = elem . flip
```

em que o ponto representa composição de função. Isto é, $f.g(x)$ é definida para se tornar $f(g(x))$. Muitas vezes, é preferível a segunda forma, e não a primeira.

Outro exemplo de uma função como um argumento é:

```
maphead :: (a -> b) -> [a] -> [b]
maphead fun alist = [ fun x | x <- alist ]
```

A função `maphead` tem dois parâmetros, uma função `fun` e uma lista `alist`. Ela aplica a função `fun` a cada elemento `x` de `alist`, criando uma lista dos resultados. A função `maphead` é equivalente à função interna `map`.

Como exemplo, considere a função quadrado:

```
square x = x*x
```

Essa função pode ser combinada com `maphead` para elevar ao quadrado todos os elementos de uma lista:

```
maphead square [2,3,5,7,9]      -- returns [4,9,25,49,81]
maphead (\x -> x*x) [2,3,5,7,9] -- an alternative
```

No primeiro exemplo, o primeiro parâmetro é o nome de uma função predefinida. No segundo exemplo, é definida uma função sem nome, usando a notação *lambda*¹⁴ e passada para a função `maphead`.

14.3.7 Tuplas

Uma *tupla* em Haskell é uma coleção de valores de diferentes tipos,¹⁵ colocados entre parênteses e separados por vírgulas. Aqui está uma tupla que contém uma *String* e um *Integer*:

```
("Bob", 2771234)
```

Valores tupla são definidos de uma maneira similar a valores de lista, exceto pelo fato de eles serem colocados entre parênteses `()` e não entre colchetes `[]`, e seu tamanho ser inflexível. Por exemplo, aqui está uma definição para o tipo *Entry*, que pode representar uma entrada em um catálogo de telefone:

```
type Entry = (Person, Number)
type Person = String
type Number = Integer
```

Essa definição combina o nome de uma pessoa e seu número de telefone como uma tupla. O primeiro e o segundo membros de uma tupla podem ser selecionados com o uso das funções internas `fst` e `snd`, respectivamente. Por exemplo:

```
fst ("Bob", 2771234) = "Bob"
snd ("Bob", 2771234) = 2771234
```

Continuando esse exemplo, é natural definir um tipo *Phonebook* como uma lista de pares de nomes e números de pessoas:

```
type Phonebook = [(Person, Number)]
```

Agora podemos definir funções úteis nesse tipo de dado, como, por exemplo, a função `find` que retorna todos os números de telefone para determinada pessoa `p`:

```
find :: Phonebook -> Person -> [Number]
find pb p = [n | (person, n) <- pb, person == p]
```

14. O símbolo `\` é usado em Haskell para aproximar o símbolo grego λ na formação de uma expressão *lambda*. Em geral, a expressão *lambda* $(\lambda x. M)$ é escrita em Haskell como `(\x->M)`.

15. O análogo para uma tupla em C/C++ é a `struct`.

Essa função retorna a lista de todos os números n do catálogo telefônico para os quais há uma entrada $(person, n)$ e $person == p$ (a pessoa desejada). Por exemplo, se:

```
pb = [(“Bob”, 2771234), (“Allen”, 2772345), (“Bob”, 2770123)]
```

então, a chamada

```
find pb “Bob”
```

retorna a lista:

```
[2771234, 2770123]
```

Além disso, podemos definir funções que acrescentam e excluem entradas de um catálogo telefônico:

```
addEntry :: Phonebook -> Person -> Number -> Phonebook
addEntry pb p n = [(p,n)] ++ pb

deleteEntry :: Phonebook -> Person -> Number -> Phonebook
deleteEntry pb p n = [entry | entry <- pb, entry /= (p, n)]
```

O sistema de tipo Haskell é uma ferramenta mais poderosa do que esse exemplo mostra. Ela pode ser usada para definir novos tipos de dados repetidamente, conforme ilustraremos na próxima seção.

As demais seções desenvolvem exemplos interessantes que ilustram alguns dos valores especiais da programação funcional em Haskell. O primeiro exemplo revê as semânticas de Clite, que foram originalmente discutidas e implementadas em Java.

14.3.8 Exemplo: Semânticas de Clite

Nesta seção, implementamos grande parte das semânticas de Clite usando Scheme. Lembre-se do Capítulo 8, no qual foi citado que, para a linguagem elementar Clite, o ambiente é estático, assim o estado pode ser simplesmente representado como uma coleção de pares variável-valor. Isso é expresso da seguinte maneira:

$$state = \{\langle var_1, val_1 \rangle, \langle var_2, val_2 \rangle, \dots, \langle var_m, val_m \rangle\}$$

Aqui, cada var_i representa uma variável e cada val_i representa seu valor atribuído no momento.

O estado é como uma *janela de observação* em um ambiente de desenvolvimento integrado (*integrated development environment* – IDE). Ele está sempre ligado a uma instrução particular no programa e mostra para cada variável de programa seu valor atual. Em nossa implementação Java, o estado era implementado como uma tabela *hash* na qual o identificador da variável tem a chave e o valor associado tem o valor atual da variável.

Um ponto de partida para a implementação Haskell é representar um estado como uma lista de pares, com cada par representando a ligação de uma variável com seu valor. Isto é, aplicam-se as seguintes definições de tipo:

```
type State = [(Variable, Value)]
type Variable = String
data Value = Intval Integer | Boolval Bool
           deriving (Eq, Ord, Show)
```


A terceira linha nessa definição é um exemplo de uma definição *tipo algébrico* em Haskell, na qual o novo tipo `Value` é definido como um valor `Integer` ou `Bool`. A cláusula `deriving (Eq, Ord, Show)` declara que esse novo tipo herda a qualidade, as características de ordenação e a exibição de seus tipos componentes `Integer` e `Bool`, permitindo-nos assim usar as funções igualdade (`==`), ordem (`<`) e exibição (`show`) em todos os seus valores.

Assim, o estado `Clite`:

$$\{\langle x, 1 \rangle, \langle y, 5 \rangle\}$$

pode ser representado como uma lista Haskell:

```
[("x", (Intval 1)), ("y", (Intval 5))]
```

Em seguida, implementamos as funções de acesso de estado `get` e `union` (união de substituição) da implementação Java (veja o Capítulo 8).

Recorde-se de que a função `get` obtém o valor de uma variável a partir do estado atual. Em Haskell, o código necessário é similar à função-membro `member`:

```
get :: Variable -> State -> Value
get var (s:ss)
  | var == (fst s) = snd s
  | otherwise     = get var ss
```

na qual as funções `(fst s)` e `(snd s)` retornam o primeiro e o segundo membros de uma tupla `s`, respectivamente. Como o sistema de tipo `Clite` requer que todas as variáveis usadas em um programa sejam declaradas, não pode haver uma referência a uma variável que não esteja no estado. Assim, a função `get` é mais simples do que a função `member`, já que o `case` para a lista nula não precisa ser testado.

Uma aplicação da função `get` é:

```
get "y" [("x", (Intval 1)), ("y", (Intval 5)), ("z", (Intval 4))]
  = get "y" [("y", (Intval 5)), ("z", (Intval 4))]
  = (Intval, 5)
```

Um bom modelo para a função `union` (união de substituição) é a função `subst` definida anteriormente, sendo que uma diferença é a estrutura das listas e a outra é que deve haver exatamente uma ocorrência de cada variável no estado:

```
union :: Variable -> Value -> State -> State
union var val (s:ss)
  | var == (fst s) = (var, val) : ss
  | otherwise     = s : (union var val ss)
```

Uma vez mais a função `union` adota a hipótese simplificadora de que a variável pela qual estamos procurando ocorre dentro do estado; portanto, não há necessidade de verificar para uma lista nula como o *base case*. A outra hipótese simplificadora é que há

apenas uma única ocorrência da variável dentro do estado; portanto, a função `onion` não continua a se repetir, uma vez que ela encontra a primeira instância.

Uma aplicação da função `onion` é:

```
onion "y" (Intval 4) [(("x", (Intval 1)), ("y", (Intval 5)))]
= ("x", (Intval 1)) : onion "y" (Intval 4)
  [(("y", (Intval 5)))]
= [(("x", (Intval 1)), ("y", (Intval 4)))]
```

Em nossa discussão das funções semânticas para Clite, assumimos que instruções em sintaxe abstrata Clite (veja a Figura 2.14) são representadas como tipos de dados repetitivos Haskell da seguinte maneira:

```
data Statement = Skip |
                Assignment Target Source |
                Block [ Statement ] |
                Loop Test Body |
                Conditional Test Thenbranch Elsebranch
    deriving (Show)

type Target = Variable
type Source = Expression
type Test = Expression
type Body = Statement
type Thenbranch = Statement
type Elsebranch = Statement
```

Agora o significado para uma declaração abstrata Clite pode ser escrito como a seguinte função Haskell sobrecarregada¹⁶ `m`:

```
m :: Statement -> State -> State
```

O significado de uma declaração (*Statement*) abstrata é uma função de transformação de estado que toma um *State* como entrada e produz um *State* como saída. A implementação dessas funções de significado é consequência direta das regras dadas no Capítulo 8 (e resumidas aqui). Assumimos também que foi executada uma verificação de semânticas estáticas, conforme foi descrito no Capítulo 6.

Uma instrução *Skip* corresponde a uma instrução vazia. Como tal, ela deixa o estado inalterado; o estado de saída é uma cópia do estado de entrada.

```
m (Skip) state = state
```

Uma instrução *Loop* tem um teste booleano, que é uma *Expression* (expressão), e um corpo *Statement*. Se o teste booleano não resultar em *true* (*verdadeiro*), o sentido (estado de saída) de um *Loop* será o mesmo que o estado de entrada; caso contrário, o sentido será o estado

16. Uma função *sobrecarregada* é ligeiramente diferente de uma função polimórfica. A primeira se refere a uma função que tem diferentes definições, dependendo dos tipos de seus argumentos. A segunda, como já vimos, tem uma definição que se aplica a todos os tipos de argumentos.

resultante de se executar primeiro seu corpo uma vez, depois passando o estado resultante para a reexecução do *Loop* (laço).

A implementação Haskell segue quase diretamente dessa definição:

```
m (Loop t b) state
  | (eval t state) == (Boolval True) = m(Loop t b)(m b state)
  | otherwise                        = state
```

Uma instrução *Assignment* (atribuição) consiste de um destino *Variable* (variável) e uma origem *Expression*. O estado de saída é computado a partir do estado de entrada substituindo o *Value* (valor) do destino *Variable* pelo valor calculado da origem *Expression*, que é avaliada usando o estado de entrada. Todas as outras variáveis têm no estado de saída o mesmo valor que elas tinham no estado de entrada.

A implementação Haskell de uma instrução de atribuição Clite utiliza a função de união de substituição junto com o sentido de *Assignment*.

```
m (Assignment target source) state
  = union target (eval source state) state
```

O sentido de uma *Conditional* (condicional) depende da verdade ou da falsidade de seu teste booleano no estado corrente. Se o teste for verdadeiro, então o sentido da *Conditional* terá o sentido da *Statement* thenbranch; caso contrário, ele terá o sentido da *Statement* elsebranch.

Semelhantemente, a implementação do sentido de uma instrução *Conditional* segue diretamente dessa definição:

```
m (Conditional test thenbranch elsebranch) state
  | (eval test state) == (Boolval True)
    = m thenbranch state
  | otherwise
    = m elsebranch state
```

Finalmente, considere a função de significado Haskell para avaliação de expressão Clite apenas para inteiros. Para facilitar, escolhemos uma definição apropriada de tipo algébrico para expressões Clite abstratas:

```
data Expression = Var Variable |
                  Lit Value |
                  Binary Op Expression Expression
                  deriving (Eq, Ord, Show)

type Op = String
```

A função de significado para uma *Expression* Clite pode agora ser implementada da seguinte maneira:

```
eval :: Expression -> State -> Value
eval (Var v) state = get v state
eval (Lit v) state = v
```

$$\begin{aligned}\frac{d}{dx}(c) &= 0 && c \text{ é uma constante} \\ \frac{d}{dx}(x) &= 1 \\ \frac{d}{dx}(u+v) &= \frac{du}{dx} + \frac{dv}{dx} && u \text{ e } v \text{ são funções de } x \\ \frac{d}{dx}(u-v) &= \frac{du}{dx} - \frac{dv}{dx} \\ \frac{d}{dx}(uv) &= u \frac{dv}{dx} + v \frac{du}{dx} \\ \frac{d}{dx}\left(\frac{u}{v}\right) &= \left(v \frac{du}{dx} - u \frac{dv}{dx}\right) / v^2\end{aligned}$$

| **Figura 14.4** Regras de Diferenciação Simbólica

Deixamos como exercício a definição Haskell de `eval` para expressões com operadores aritméticos e relacionais.

Esse desenvolvimento de uma pequena fração das semânticas formais de Clite deverá convencê-lo de que um modelo semântico completo para uma linguagem imperativa pode ser facilmente definido em Haskell.

14.3.9 Exemplo: Diferenciação Simbólica

A utilidade da Haskell para manipulação de símbolos é ampla, como mostrou o exemplo anterior. Esse próximo exemplo ilustra ainda melhor alguns dos recursos da Haskell fazendo diferenciação simbólica e simplificação de fórmulas simples de cálculo. Algumas regras familiares para diferenciação simbólica são dadas na Figura 14.4.

Por exemplo, diferenciando a função $2 \cdot x + 1$ com relação a x usando essas regras resulta:

$$\begin{aligned}\frac{d(2 \cdot x + 1)}{dx} &= \frac{d(2 \cdot x)}{dx} + \frac{d1}{dx} \\ &= 2 \cdot \frac{dx}{dx} + x \cdot \frac{d2}{dx} + 0 \\ &= 2 \cdot 1 + x \cdot 0 + 0\end{aligned}$$

que, simplificando, resultaria em 2.

Em Haskell, é conveniente representar expressões usando tipos de dados repetitivos (análogo à notação Polish prefixa de Scheme):

```
Add expr1 expr2
Sub expr1 expr2
Mul expr1 expr2
Div expr1 expr2
```

A função necessária para fazer a diferenciação simbólica primeiro testa para determinar se a expressão é uma constante ou a variável que está sendo diferenciada, como nas duas primeiras regras na tabela acima.

Caso contrário, a expressão é uma lista que começa com um operador, e o código aplica uma das quatro regras restantes usando o reconhecedor de padrão no operador.

```
data Expr = Num Int | Var String | Add Expr Expr |
           Sub Expr Expr | Mul Expr Expr |
           Div Expr Expr
           deriving (Eq, Ord, Show)
diff :: String -> Expr -> Expr

diff x (Num c) = Num 0
diff x (Var y) = if x == y then Num 1 else Num 0
diff x (Add u v) = Add (diff x u) (diff x v)
diff x (Sub u v) = Sub (diff x u) (diff x v)
diff x (Mul u v) = Add (Mul u (diff x v))
                    (Mul v (diff x u))
```

Uma aplicação da função `diff` à expressão $2 \cdot x + 1$ resulta:

```
Add (Add (Mul (Num 2) (Num 1))
        (Mul (Var "x") (Num 0)) (Num 0))
```

Para tornar o resultado um pouco mais claro, representamos aqui um formatador que produz uma versão com muitos parênteses na expressão de saída:

```
formatExpr (Num n) = show n
formatExpr (Var x) = x
formatExpr (Add a b) =
    "(" ++ formatExpr a ++ " + " ++ formatExpr b ++ ")"
formatExpr (Sub a b) =
    "(" ++ formatExpr a ++ " - " ++ formatExpr b ++ ")"
formatExpr (Mul a b) =
    "(" ++ formatExpr a ++ " * " ++ formatExpr b ++ ")"
formatExpr (Div a b) =
    "(" ++ formatExpr a ++ " / " ++ formatExpr b ++ ")"
```

Para a mesma expressão dada acima, o formatador produz:

```
((2 * 1) + (x * 0)) + 0)
```

que é um pouco mais clara. O resultado normal, 2, ocorreria após simplificar essa expressão. Fica como exercício escrever um simplificador de expressão como esse.

14.3.10 Exemplo: O Programa das Oito Rainhas

Enfim, voltamos novamente nossa atenção para o problema de colocar N rainhas mutuamente antagônicas em um tabuleiro $N \times N$ de forma que nenhuma rainha possa capturar

Figura 14.5
Três Rainhas em um
Tabuleiro de Xadrez 8×8

Q							
	Q						
		Q					

outra rainha em um único movimento. Ao desenvolver a solução, usaremos as mesmas codificações das diagonais usadas na Seção 13.4.2. Porém, em vez de tentar converter o algoritmo *backtracking* de Wirth (Wirth, 1976) e depois adaptá-lo ao problema das oito rainhas, desenvolvemos uma versão puramente funcional do zero.

A primeira decisão é que a função desejada produza uma lista de todas as soluções possíveis, em que cada solução liste a posição da fila de cada rainha em ordem de coluna. Por exemplo, o tabuleiro com três rainhas nas posições (linha, coluna) mostradas na Figura 14.5 é representado pela seguinte lista:

[0, 2, 4]

No entanto, o programa é mais bem entendido quando se trabalha da direita para a esquerda. Tentando estender uma solução parcialmente segura, primeiro construímos novas listas com um teste de número de fileira (tomado da sequência $[0..n-1]$). Lembre-se da Seção 13.4.2, em que uma (row, col) é segura:

- Se a fileira de teste não é um elemento da solução existente.
- Se as diagonais sudoeste e sudeste estiverem desocupadas. Na Seção 13.4.2, a diagonal sudoeste foi computada como $row + col$ e a diagonal sudeste como $row - col$.

Essa verificação *segura* está englobada nas funções *safe* e *checks*. A função *safe* é passada para uma posição b e a um próximo teste de fileira q . Nesse programa, o tabuleiro é construído da direita para a esquerda; como as soluções são simétricas, a direção a partir da qual se quer trabalhar é puramente uma questão de preferência ou eficiência.

A linha q é segura em relação ao tabuleiro atual se as condições acima forem satisfeitas. Nesse caso, *checks* é chamada uma vez por cada valor de índice de b , isto é, de 0 a $length\ b - 1$. A verificação de linha para cada i é simplesmente: $q \neq b!!i$. A verificação da diagonal sudoeste para cada i deverá ser $q+n \neq b!!i - (n-i-1)$, que simplificando se torna $q - b!!i \neq -i - 1$. Por uma análise similar, a verificação da diagonal sudeste para cada i se simplifica tornando-se $q - b!!i \neq i+1$. A verificação no programa combina os dois casos tomando o valor absoluto.

Em um programa funcional, não há armazenamento global para armazenar e acessar as diagonais. Mesmo as linhas ocupadas são armazenadas como uma lista e passadas

como um argumento. Assim, nós preferimos computar dinamicamente as informações de diagonal conforme necessário, em vez de passá-las na lista de argumentos.

A solução para esse problema é:

```
queens n = solve n
  where
    solve 0 = [ [ ] ]
    solve (k+1) = [ q:b | b <- solve k,
                          q <- [0..(n - 1)], safe q b ]
    safe q b = and [not (checks q b i) |
                    i <- [0..(length b - 1)] ]
    checks q b i = q == b!!i || abs(q - b!!i) == i+1
```

Observe o uso da cláusula para ocultar as definições das funções helper `solve`, `safe` e `checks`. Veja também que o argumento formal para `queens`, ou seja, `n`, está referenciado em `solve`. Note a brevidade e a simplicidade dessa solução.

Repare que esse programa computa todas as soluções para um dado n como uma lista de listas de fileiras. Assim, `solve 0` retorna uma lista formada pela lista vazia, já que isso pode ser interpretado como uma solução válida. A função `solve` estende cada solução válida anterior (lista interna) filtrando cada número legal de linha com cada solução válida para saber se ela é segura. A função `safe` usa a função `checks` para saber se a linha ou as diagonais estão ocupadas, produzindo para cada tentativa de solução estendida uma lista de booleanos que primeiro é invertida e depois colocada junto. Nesse caso, uma linha ou diagonal ocupada produz *true* (verdadeiro), que é invertida como *false* (falso). Qualquer *false* (falso) na lista torna o `and` falso, resultando na rejeição da tentativa de solução estendida.

A execução desse programa para vários valores de n inclui:

```
> queens 0
[[]]
> queens 1
[[0]]
> queens 2
[]
> queens 3
[]
> queens 4
[[2,0,3,1],[1,3,0,2]]
```

que diz que:

- Para $n = 0$ uma solução consiste em não colocar nenhuma rainha.
- Para $n = 1$ uma solução consiste em colocar uma rainha em (0, 0).
- Para $n = 2, 3$ não há soluções.
- Para $n = 4$ há duas soluções, mas uma é a imagem espelhada da outra.