0 Programação Relacional em Scheme

Uma pessoa só tem certeza
daquilo que constrói

Giambattista Vico -
Historiador italiano, século XVIII

Neste capítulo, veremos uma linguagem de programação lógica com um sabor diferente do Prolog que vimos no início, chamada *miniKanren*, e veremos como implementá-la, por meio da linguagem *Scheme*.

Usaremos *Scheme* porque é uma linguagem que é pequena, o que significa que não será gasto muito esforço para apresentar o seu funcionamento. E é, ao mesmo tempo, poderosa, o que significa que não precisaremos de muito código para fazer o que nos propomos.

Esse capítulo tem um sabor diferente dos demais. Essa diferença pode ser visto rapidamente, pela cara do código, mas também de várias outras formas. Redefiniremos alguns termos usados anteriormente. Essas redefinições terão semelhanças e diferenças às definições originais, mas escolhemos não notar essas diferenças aqui, por acreditarmos serem claras o suficiente. Outro ponto que vale nota é que, aqui, buscamos apenas um maior entendimento e, para tanto, tentamos deixar o código como "implementado em primeiros princípios", evitando detalhes que poderiam deixá-lo mais eficiente, mas que fariam uso de construções que não queremos introduzir aqui.

0.1 Introdução ao Scheme

Scheme é um *Lisp*[*]. O termo *Lisp* é às vezes usado para ser referir a uma linguagem de programação, mas o mais correto seria ser referida como a uma família de linguagens (de fato, dezenas de linguagens), todas com algumas características em comum, notadamente:

- São linguagens multi-paradigma, mas como um foco no paradigma de programação funcional, isto é, funções são "cidadãos de primeira classe";
- "Todo"[†] código *Lisp* (que não tenha erro de sintaxe) é avaliado para algum valor;
- Programas são expressos em notação polonêsa, em formato de listas[‡](listas em *Lisps* são delimitadas por parênteses)[§]:

[*] De *LISt Processing*

[†] Com uma excessão bem natural, que será mencionada mais adiante.

[‡] Também conhecidas como *S-Expressions*, ou *Sexps*

[§] Usaremos daqui em diante a notação **codigo** ⇒**valor** para denotar que o código **codigo** avalia para o valor **valor**.

`(+ 1 2)` $\Rightarrow$3;

- O que nos leva a outro ponto: não existe diferença sintática entre a forma como programas *Lisp* são escritos e a forma como suas estruturas de dados são representadas. Diz-se, assim, que *Lisps* são homoicônicas (vale dizer, Prolog também é uma língua homoicônica), o que significa que a diferença entre dados e programa é "borrada", e programas podem, e frequentemente são, manipulados livremente.

Agora, vamos rapidamente introduzir a sintaxe principal de *Scheme*, com alguns exemplos*:

- Listas são representadas como (el_1 el_2 ... el_n), em que el_i é o iésimo elemento da lista. No entanto, se escrevêssemos uma lista simplesmente assim, a lista seria confundida com uma aplicação de função (aplicação da função el_1 aos argumentos el_2 a el_n), então, para fins de desambiguação, é usada uma aspa simples, e a lista é escrita como '($el_1 el_2 ... el_n$), que é equivalente a `(list` $'el_1$ ... $'el_n$`)` . Essa aspa simples também pode ser usada para "evitar que um objeto seja avaliado"†:
  A linha
  `a` $\Rightarrow$erro
  resulta em erro se `a` não for uma variável, já que tentará ser avaliada, sendo que `a` não tem valor associado, mas a linha
  $'a \Rightarrow 'a$
  é avaliada como ela mesma.

- Uma estrutura de dados mais geral do que lista em *Scheme* é o que é chamado *cons pair* (que nós chamaremos daqui para frente simplesmente de "par").. A lista `'(a b c d)` é equivalente a `(cons a (cons b (cons c (cons d '()))))`, em que `'()` é a lista vazia Assim *cons* constrói uma estrutura de dados formada por um par. Para obter o primeiro elemento do par, usa-se *car* e, para obter o segundo, *cdr*‡. Temos, por exemplo,
  `(car (cons 1 (cons 2 3)))` $\Rightarrow$1
  `(cdr (cons 1 (cons 2 3)))` $\Rightarrow$`(cons 2 3)` = `'(2 . 3)`
  `(car '(1 2 3 4))` $\Rightarrow$1
  `(cdr '(1 2 3 4))` $\Rightarrow$`'(2 3 4)`
  Com isso, podemos definir um lista indutivamente como sendo ou a lista vazia, `'()`, ou um par, cujo *cdr* é uma lista§.

*Note que só introduziremos a parte da linguagem que nos será relevante, o que não é a linguagem toda.

†Uma colocação mais correta seria "tornar objetos auto-avaliantes", mas não precisamos entrar em muitos detalhes de como isso funciona. Para nós é suficiente dizer que 'a é um símbolo.

‡Esses nomes têm uma motivação histórica: eram nomes de registradores quando os primeiros *Lisps* estavam sendo criados.

§Note que `(cons 2 3)`, por exemplo, não é uma lista. Esse tipo de estrutura é chamada

- Para definir funções, use **lambda**, ou λ*:
  `((λ (a b) (/ a b)) 1 3)` ⇒1/3

- Para definir constantes, use **define**:
  `(define divide (λ (a b) (/ 1 3)))`†
  `(divide 1 3)` ⇒1/3
  `(define c (divide 1 3))`
  `(divide (divide 1 9) c)` ⇒1/3

- Um ponto importante, que usaremos muito logo mais é que, se quisermos criar listas com os valores das variáveis, no lugar de nomes simbólicos, podemos usar, no lugar da aspa simples a crase e preceder o nome da variável com uma vírgula:
  `(define x 10)`
  `` `(1 2 'x ,x) `` ⇒`'(1 2 'x 10)`

- Para realizar execuções condicionais, use **cond**:

```
(cond
  ((< 1 0) (+ 3 4))
  ((< 0 1) (- 3 4))
  (else 0))
```

  ⇒-1

  Podem ser adicionadas quantas cláusulas do tipo `((condicao)(efeito))` se quiser (vale notar que elas são avaliadas sequencialmente), sendo que a última pode opcionalmente ser como `(else (efeito))`, ou `(#t (efeito))`‡.

- Para adicionar variáveis locais, use *let*:

```
(let ((a (+ 3 4))
      (b (cons 1 2)))
  (+ a (car b)))
```

---

dotted list, porque, para distingui-la de uma lista, é costumeiramente impressa como `'(2 . 3)`, mas, assim como com Prolog, é uma estrutura de dados diferente, que tem o nome *dotted* ***list*** como que por uma aparência acidental.

*Editores de texto atuais aceitam os dois tipos de entrada, mas optamos por usar λ. Esse uso do símbolo tem a seguinte origem: Bertrand Russel e Alfred Whitehead buscaram, no início do século XX lançar as bases lógicas da matemática eu seu trabalho *Principia Mathematica*. Lá, para denotar que uma variável é livre, ela recebia chapéu, como em $\hat{a}(a + y)$. Mais tarde, Alonzo Church achou que seria mais conveniente ter fórmulas crescendo linearmente na horizontal, então decidiu mover o chapéu para o lado, obtendo $\hat{}a(a + y)$. Mas o chapéu flutuando parece engraçado, então Church o trocou pelo o símbolo não usado mais próximo que tinha, um Λ, como em $\Lambda a(a + y)$. Mas Λ tem uma grafia muito parecida com outra letra comum, então ele acabou eventualmente trocando para λ em sua teoria, que acabou se chamando *cálculo* λ [2].

†Um *açucar sintático* para essa construção é `(define (divide a b) (/ 1 3))`.

‡Preferiremos a segunda opção. O *#t* é de *true* e existe, analogamente, o *#f*, de *false*.

$\Rightarrow$8

O *let* tem duas partes, a de definições, da forma `((variavel valor)(variavel valor) ... (variavel valor))`*. em seguida, a parte de valor, que nos dá o valor que *let* assume.

Como exemplo, o seguinte programa calcula o comprimento de uma lista (lembre que, como

Dada essa introdução, faremos uso dessas e outras construções da linguagem sem maiores comentários (exceto quando uma construção especialmente difícil ou complexa o justificar). Para uma introdução mais compreensiva à linguagem, veja [1].

0.2 A linguagem miniKanren

Nosso objetivo aqui é implementar miniKanren, uma linguagem de programação relacional. No lugar de descrever toda a linguagem e depois implementá-la, seguimos pelo caminho de mostrar um pequeno exemplo do que esperamos conseguir e, então, implementamos a linguagem. A esperança é que essa abordagem ofereça maior entendimento dos conceitos explorados.

O tipo de coisa que queremos poder fazer com miniKanren é como o seguinte†:

```
(defrel (teacupᵒ t)
  (disj₂ (≡ 'tea t) (≡ 'cup t)))
(run* x
  (teacupᵒ x))
```

$\Rightarrow$`'(tea cup)`

Veremos mais exemplos enquanto o construímos. A construção a seguir é em grande parte baseada em [3]. Para conferir detalhes de implementações completas, veja [4].

Antes de outras coisas, será útil sabermos como definir uma variável. Lembre-se que uma variável relacional não é a a mesma coisa que uma variável em uma linguagem de programação tradicional (não relacional). Para uma variável única, usamos‡

*Podem ser adicionadas quantas variáveis se quiser. As atribuições são feitas "paralelamente" (o que significa que a atribuição de valor a uma variável não influi no da outra, o que pode ser feito de forma paralela, ou não).

†Usaremos as convenções da literatura de usar subescritos e sobrescritos e símbolos matemáticos, como $\equiv$, para representar as relações, na esperança de que isso clarifique a notação e deixe o texto menos pesado. Em particular, para diferenciar um objeto relacional de um funcional, o relacional terá um "o" sobrescrito (como em $relacao^o$).

‡Usamos *vector* para que a unicidade da variável seja definida por sua posição de memória. Outra opção, seria distingui-las por valor, se nos assegurássemos que seu valor é único.

```
(define (var name)(vector name))
```

usamos também*

```
(define (var? name)(vector? name))
```

Precisamos, agora, saber como associar um valor a uma variável. Diremos que o par `` `(,z . a) `` é uma associação de a à variável z. Mais em geral, um par é uma associação quando o *car* dele é uma variável.

A uma lista de associações chamaremos *substituição*. Na substituição `` `((,x . ,z)) ``, a variável x é "fundida" (ou, na nossa linguagem anterior, unificada) à variável z. A substituição vazia é simplesmente uma lista vazia: `(define empty-s '())`. Nem toda lista de associações é uma substituição, no entanto. Isso porque, não aceitamos, em nossas substituições, associações com o mesmo *car*. Então, o seguinte não é uma substituição: `` `((,z . a)(,x . c)(,z . b)) ``.

Precisamos de dois procedimentos importantes para lidar com substituições: um para extênde-las e um para obter o valor de uma variável presente nela.

Para obter o valor associado a x, usamos *walk*, que deve se comportar como o seguinte:

```
(walk y
  `((,z . a)(,x . ,w)(,y . ,z)))
```

$\Rightarrow a$

porque y está fundido a z, que está associado a a. O *walk* é como se segue†:

```
(define (walk v s)
  (let ((a (and (var? v)(assv v s))))
    (cond
      ((pair? a)(walk (cdr a) s))
      (else v))))
```

Esse código faz uso de *assv*, que ou produz *#f*, se não há associação cujo *car* seja v na substituição s, ou produz o *cdr* de tal associação. Perceba que, se *walk* produz uma variável como valor, ela é necessariamente fresca (isto é, que não foi associada).

Para extender uma substituição, usamos *ext-s*, que faz uso de *occurs?*:

```
(define (ext-s x v s)
  (cond
    ((occurs? x v s) #f)
    (else (cons '(,x . ,v) s))))
```

*Símbolos como "?" podem ser usados no meio do código asssim como outros, tais como "a" ou "b".

†Lembre-se que `(and a b)` $\Rightarrow b$, se $a \neq \#f$.

```scheme
(define (occurs? x v s)
  (let ((v (walk v s)))
    (cond
      ((var? v) (eqv? v x))
      ((pair? v)
        (or (occurs? x (car v) s)
            (occurs? x (cdr v) s)))
      (else #f))))
```

Esse *occurs?* realiza o "teste de ocorrência" (aquele que, como mencionamos anteriormenet, não é feito por padrão no Prolog por razões de eficiência, e que faz com que substituições do tipo `‘((,y . (,x))(,x . (a . ,y))` sejam inválidas).

Com isso em mãos, podemos definir nosso unificador:

```scheme
(define (unify u v s)
  (let ((u (walk u s))(v (walk v s)))
    (cond
      ((eqv? u v) s)
      ((var? u) (ext-s u v s))
      ((var? v) (ext-s v u s))
      ((and (pair? u) (pair? v))
       (let ((s (unify (car u)(car v) s)))
         (and s
           (unify (cdr u)(cdr v) s))))
      (else #f))))
```

0.3 Streams

Antes de continuarmos, precisamos tocar no modelo de avaliação de Scheme. Scheme é uma linguagem de "ordem aplicativa", o que significa que os argumentos de uma função são todos avaliados no momento em que a função é aplicada. Por esse motivo, os *and* e *or* usuais, por exemplo, não podem ser funções em Scheme. Uma alternativa à ordem aplicativa é a "ordem normal", que algumas outras linguagens de programação funcional adotaram. Linguagens de ordem normal só avaliam o argumento de uma função quando esse argumento for usada, atrasando a avaliação do mesmo (no que é chamado "avaliação tardia", ou "avaliação preguiçosa").

Avaliação preguiçosa é conveniente para diversas operações e pode ser emulada em linguagens de programação funcional de ordem aplicativa pelo uso de **streams**. *Streams* são definidos como sendo ou a lista vazia, ou um par cujo *cdr* é um *stream*, ou uma suspenção. **streams**

Uma **suspenção** é uma função formada por `(λ () corpo)`, em que `((λ ()` **suspenção** `corpo))` é uma *stream*. Agora, se definirmos

```
(define (≡ u v)
  (λ (s)
    (let ((s (unify u v s)))
      (if s `(,s) '()))))
```

≡ produz um goal. Dois outros goals, *sucesso* e *falha*, são denotamos #s e #u:

```
(define #s
  (λ (s)
    `(,s)))

(define #u
  (λ (s)
    '()))
```

Um goal é uma função que recebe uma substituição e que, se retorna, retorna uma *stream* de substituições.

Como um exemplo, temos que `((≡x y) empty-s)` $\Rightarrow$`` `(((,x . ,y))) ``, uma lista com uma substituição (com uma associação).

Ao lidar com *Streams*, precisamos de funções especiais, já que não são simples listas. *Streams* são uteis para a representação de estruturas de dados infinitas, então, por isso, funções e variáveis para lidar com elas terão um ∞ sobrescrito, para diferenciá-las das funções para listas comuns. Assim, podemos definir $append^{\infty}$:

```
(define (append∞ s∞ t∞)
  (cond
    ((null? s∞) t∞)
    ((pair? s∞)
      (cons (car s∞
        (append∞ (cdr s∞) t∞))))
    (else (λ ()
            (append∞ t∞ (s∞))))))
```

Note que, na suspensção, a ordem dos argumentos é trocada. Com essa função, podemos fazer

```
(define (disj₂ g₁ g₂)
  (λ (s)
    (append∞ (g₁ s) (g₂ s))))
```

em que $disj_2$ é uma disjunção (um *ou* lógico).

Veja agora a seguinte definição:

```
(define (neverᵒ)
  (λ (s)
    (λ ()
      ((neverᵒ) s))))
```

Esse é um goal que não sucede nem falha. Para entender porque a ordem dos argumentos é trocada na suspenção de *append*$^\infty$, compare o valor de s^∞ em

```
(let ((s∞ ((disj₂
              (neverᵒ)
              (≡ 'olive x))
             empty-s)))
  s∞)
```

com o valor de s^∞ em

```
(let ((s∞ ((disj₂
              (≡ 'olive x))
              (neverᵒ)
             empty-s)))
  s∞)
```

Leituras adicionais

[1] R. Kent Dybvyg, "The Scheme Programming Language - Fourth Edition", disponível em https://www.scheme.com/tspl4/, acesso em Setembro de 2018.

[2] Peter Norvig, "Paradigms of Artificial Inteligence Programming - Case Studies in Common Lisp", Morgan Kauffman Publishers, 1992.

[3] Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, Jason Hemann, "The Reasoned Schemer - Second Edition", The MIT Press, 2018.

[4] miniKanren http://minikanren.org/.