

Sequências e streams

Paradigma de Programação Funcional

Marco A L Barbosa



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhual 4.0 Internacional.

Conteúdo

Sequências

List comprehension

Streams

Streams implícitos

Promessas

Racket lazy

Referências

O estudo utilizando apenas este material **não é suficiente** para o entendimento do conteúdo. Recomendamos a leitura das referências no final deste material e a resolução (por parte do aluno) de todos os exercícios indicados.

Sequências

Sequências

- ▶ Uma sequência encapsula uma coleção ordenada de valores
- ▶ Sequências são geralmente utilizadas com as formas sintáticas `for`
- ▶ Tipos que são sequências
 - ▶ Listas
 - ▶ Streams
 - ▶ etc
- ▶ Exemplos

```
> (sequence? (list 5 2 10))  
#t  
> (sequence? 1.2)  
#f
```

List comprehension

List comprehension

- ▶ Utilização da notação de conjunto para definir uma lista
- ▶ Combina `map` e `filter`
- ▶ Exemplo
 - ▶ $S = \{2x \mid x \in 1..10\}$
 - ▶ $T = \{2x \mid x \in 1..10, n^2 > 8\}$

List comprehension

- ▶ Utilização da notação de conjunto para definir uma lista
- ▶ Combina `map` e `filter`
- ▶ Exemplo
 - ▶ $S = \{2x \mid x \in 1..10\}$
 - ▶ $T = \{2x \mid x \in 1..10, n^2 > 8\}$
- ▶ Em Racket temos a forma sintática especial `for/list`

```
(define S (for/list ([x (in-range 1 11)])  
            (* 2 x)))
```

```
(define T (for/list ([x (in-range 1 11)]  
                    #:when (> (sqr x) 8))  
            (* 2 x)))
```

```
> S  
'(2 4 6 8 10 12 14 16 18 20)  
> T  
'(6 8 10 12 14 16 18 20)
```


List comprehension

- ▶ Uma aproximação da sintaxe do for/list é

```
(for/list (clause ...)
  body ...+)
```

```
clause = [id sequence-expr]
         | #:when boolean-expr
         | #:unless boolean-expr
```

- ▶ É possível fazer uma iteração em paralelo em duas ou mais sequências

```
> (for/list ([i (in-naturals)]
            [x (list 3 5 2 4)])
  (- x i))
'(3 4 0 1)
```

- ▶ A função `in-naturals` devolve uma sequência (stream) com os números naturais
- ▶ Como as sequências tem tamanhos diferentes, a iteração é interrompida quando alguma sequência termina

List comprehension

- ▶ Existem muitas funções pré-definidas que são úteis neste contexto
 - ▶ `in-range`
 - ▶ `in-naturals`
 - ▶ `in-cycle`
 - ▶ `in-value`
 - ▶ `stop-before`
 - ▶ `stop-after`
 - ▶ Veja a referência sobre sequências
- ▶ O Racket oferece ainda uma coleção de formas especiais para fazer iteração em sequências, veja a referência sobre iterações

Streams

Streams

- ▶ Um stream é semelhante a uma lista, mas os elementos só são avaliados quando são necessários, em outras palavras, um **stream** é uma lista atrasada
- ▶ Streams tem muitas utilidades, mas vamos usá-los principalmente para definir “listas infinitas” (como a função `in-naturals`)

Streams

- ▶ As operações primitivas de streams são semelhantes as das listas
 - ▶ `stream-cons`
 - ▶ `stream-first`
 - ▶ `stream-rest`
- ▶ Outras funções pré-definidas
 - ▶ `stream-ref`
 - ▶ `stream->list`
 - ▶ `stream-fold`
 - ▶ `stream-map`
 - ▶ `stream-filter`
 - ▶ Veja a referência de streams

Streams

- ▶ Escrita de testes
 - ▶ Podemos utilizar as funções pré-definidas `stream-ref` e `stream->list`
 - ▶ Função `stream` que cria um stream com os elementos especificados (semelhante a função `list`)

Exemplo 8.1

Defina uma função que crie um stream de números inteiros a partir de um valor inicial n .

Exemplo 8.2

Defina uma função que crie um stream com os n primeiros elementos de um outro stream. (Semelhante a função `take`)

Exemplo 8.3

Defina uma função que receba dois streams como parâmetro e crie um stream em que cada elemento é a soma dos dois elementos na mesma posição dos streams de entrada.

Exemplo 8.4

Problema 1 do Projeto Euler. Defina uma função que some todos os números naturais menores que um dado n que sejam múltiplos de 3 ou 5.

Streams implícitos

Streams implícitos

```
> (define uns (stream-cons 1 uns))  
> (stream->list (stream-take uns 10))  
'(1 1 1 1 1 1 1 1 1 1)
```

```
> (define naturais (stream-cons 0 (stream-soma naturais uns)))  
> (stream->list (stream-take naturais 10))  
'(0 1 2 3 4 5 6 7 8 9)
```

```
> (define fibs (stream-cons  
  0  
  (stream-cons  
    1  
    (stream-soma (stream-rest fibs)  
                  fibs))))  
> (stream->list (stream-take fibs 10))  
'(0 1 1 2 3 5 8 13 21 34)
```

Promessas

Promessas

- ▶ Stream são criados utilizando as primitivas `delay` e `force`
- ▶ `delay` cria uma promessa de avaliar uma expressão

```
> (define p (delay (+ 4 5)))  
> p  
#<promise:p>
```

- ▶ `force` faz com que uma promessa seja avaliada, se a promessa não foi forçada antes, o resultado é armazenado na promessa de maneira que quando `force` for utilizado novamente a promessa produza o mesmo valor

```
> (force p)  
9  
> p  
#<promise!9>  
> (force p)  
9
```

Racket lazy

Racket lazy

- ▶ Os streams não são bem integrados com a linguagem Racket
- ▶ A linguagem experimental lazy está tentando melhorar esta integração

```
#lang lazy
```

```
(define (naturais n)  
  (cons n  
        (naturais (add1 n))))
```

```
> (naturais 10)
```

```
'(10 . #<promise>)
```

```
> (take 20 (naturais 10))
```

```
'(10 . #<promise:...llects/lazy/lazy.rkt:672:43>)
```

```
> (!! (take 20 (naturais 10)))
```

```
'(10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29)
```


Referências

Referências

- ▶ Seção 3.5 (3.5.1 e 3.5.2) do livro SICP
- ▶ Seção 4.14 e 2.18 da Referência Racket.
- ▶ Referência da linguagem lazy.