Sequências e streams

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Sequências

Sequências

- Uma sequência encapsula uma coleção ordenada de valores
- Sequências são geralmente utilizadas com as formas sintáticas `for`

Sequências

- Tipos que são sequências
  - Listas
  - Strings
  - Streams
  - etc

Sequências

- Exemplos

```
> (sequence? (list 5 2 10))
#t
> (sequence? "casa")
#t
> (sequence? (in-range 10 20))
#t
> (sequence? 1.2)
#f
```

List comprehension

List comprehension

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

List comprehension

- 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
```

List comprehension

- 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 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-afer`
  - Veja a referência sobre sequências

List comprehension

- O Racket oferece ainda uma coleção de formas especiais para fazer iteração em sequências, veja a referência sobre iterações

Streams

Streams

- Um stream é uma sequência potencialmente infinita
- Em geral, os elementos do stream são produzidos quando são necessários, neste caso, um **stream** é uma sequência preguiçosa
- Streams têm várias utilidades, mas vamos usá-los principalmente para definir "sequências infinitas" (como a função `in-naturals`)

Streams

- As operações primitivas de streams são semelhantes as das listas
  - `stream-cons`
  - `stream-first`
  - `stream-rest`

Streams

- Outros 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.

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)
```

Streams implícitos

```
> (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

- Streams são criados utilizando as primitivas `delay` e `force`

Promessas

- `delay` cria uma promessa de avaliar uma expressão

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

- `(stream-cons <a> <b>)` é uma forma especial equivalente a

```
(cons <a> (delay <b>))
```

Promessas

- `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>
```

Implementação

- Uma implementação simples seria fazer a expressão `(delay <expr>)` ser equivalente a `(λ () <expr>)`, e `(force p)` simplesmente executaria `(p)`
- Mas neste caso o resultado da promessa deve ser calculado a cada chamada
- Para armazenar o resultado da promessa vamos usar variáveis!

Implementação

- Neste caso, `(delay <expr>)` é equivalente a `(memoriza (λ () <expr>))`

```
(define (memoriza proc)
  (let ([was-run? #f]
        [result (void)])
    (lambda ()
      (if (not was-run?)
          (begin
            (set! result (proc))
            (set! was-run? #t)
            result)
          result))))
```

Racket lazy

Racket lazy

- No passado os streams não estavam bem integrados com a linguagem Racket
- A linguagem `lazy` melhorou esta integração

Racket lazy

```
#lang lazy

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

> (naturais 10)
'(10 . #<promise>)
> (take 6 (naturais 10))
'(10 . #<promise:...llects/lazy/lazy.rkt:672:43>)
> (!! (take 6 (naturais 10)))
'(10 11 12 13 14 15 16)
```

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.