# Racket Básico: Tipos e QuickCheck

**Profs. Diogo S. Martins e Emilio Francesquini**

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

12 de junho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição



Crédito de parte das imagens, a menos se especificado: Wikipedia

# Roteiro

- Tipos e operações básicas em:
  - Booleanos
  - Números
  - Pares e Listas
- Testando o seu código com QuickCheck

# Expressões booleanas e numéricas

# Tipo Boolean

- Variáveis do tipo Boolean em Racket são:
  - `#t` - Verdadeiro
  - `#f` - Falso
- Na prática, contudo, qualquer valor diferente de `#f` é considerado como verdadeiro

```
$ racket
Welcome to Racket v6.12.
> (if #t 1 2)
1
> (if #t "Verdadeiro" "Falso")
"Verdadeiro"
> (if #f "Verdadeiro" "Falso")
"Falso"
> (if 2 "Verdadeiro" "Falso")
"Verdadeiro"
> (if "False" "Verdadeiro" "Falso")
"Verdadeiro"
```

# Operações com Booleans

- `and`, `or`, `xor`, `not`
- Diferentemente de outras linguagens, as funções recebem uma lista de Booleans

```
> (and #t #f)
#f
> (or #f #f #t)
#t
> (not (xor #t (or #f #f #t)))
#t
> (not "lixo")
#f
```

# Operações Numéricas

- Em Racket, a conversão entre os tipos numéricos é feita automaticamente (tipos dinâmicos)
- Os inteiros são de precisão arbitrária (i.e. limitados apenas pela quantidade de memória)
- Oferece suporte a números complexos e frações automaticamente

```
> (+ 1 2)
3
> (/ 5 (* 4 (+ 2 1)))
5/12
> (/ 5.0 12)
0.4166666666666667
> (sqrt -1)
0+1i
> (/ (sqrt -1) 4)
0+1/4i
> (+ 1.0 2+3i 5)
8.0+3.0i
> (complex? (sqrt -1))
#t
> (complex? (sqrt 16))
#t
> (real? (sqrt -1))
#f
> (real? (sqrt 16))
#t
```

# Operações mais comuns com números

- Comparação: `= < > <= >=`
- Aritmética: `+ - * / remainder modulo quotient`
- Outros: `sqrt expt exp abs max min …`
- Uma lista mais completa pode ser vista em: `https://docs.racket-lang.org/reference/number-types.html`

# QuickCheck: testes baseados em propriedades

# QuickCheck: introdução

- O QuickCheck é uma ferramenta para teste de código baseado em propriedades
- Princípios:
  1. Define-se uma propriedade (invariante) do algoritmo
  2. O mecanismo de teste gera automática e aleatoriamente casos de teste
- Método complementar aos testes baseados em exemplos (i.e. testes de unidade)
- **Nota**: nas nossas atividades avaliativas, é obrigatório entregar testes automáticos junto com o código

# Instalando o QuickCheck

- Para instalar o QuickCheck:

```
raco pkg install quickcheck
```

- Após algumas perguntas sobre a instalação de dependências o QuickCheck terá sido instalado.
- Para incluir a biblioteca:

```
(require quickcheck)
```

# QuickCheck - Primeiro exemplo: números pares

- Queremos escrever uma função que devolva `#t` caso o número recebido como parâmetro seja par, e `#f` caso contrário.

```
1 (define (par? n)
2   (= (modulo n 2) 0))
```

- Para testar a função, defino a propriedade: “o sucessor de todo par é ímpar”.

```
1 (define prop-alternancia-par-impar
2   (property ([n arbitrary-integer])
3             (xor (par? n) (par? (add1 n)))))
```

- A propriedade é composta de variáveis, geradores e a propriedade, em si, a ser verificada
- Para rodar os testes:

```
> (quickcheck prop-alternancia-par-impar)
OK, passed 100 tests.
```

# Se par não ímpar

- Como já temos a função `par`, é um tanto desnecessário definir a função ímpar como abaixo. Mas o exemplo é interessante para mostrar a função ==> (implicação)
- Quero testar com a propriedade: “Se n é par logo não é ímpar.”

```
1 (define (impar? n)
2   (= (remainder n 2) 1))
3
4 (define prop-se-impar-nao-par
5   (property ([n arbitrary-integer])
6             (==> (par? n) (not (impar? n)))))
```

# Exercício 1: Fatorial

1. Defina uma função que calcule o fatorial de um número
2. Teste o correto funcionamento da função usando o QuickCheck

# Fatorial - Tentativa 1

- Propriedade: $(n+1)! = (n+1) \cdot n!$

```
1 (define (fatorial n)
2   (if (= n 1)
3       1
4       (* n (fatorial (sub1 n)))))
5
6 (define fatorial-n-fatorial-n+1
7   (property ([n arbitrary-integer])
8             (= (* (add1 n) (fatorial n)) (fatorial (add1 n)))))
```

Não funciona pois não contempla `0`!

# Fatorial - Tentativa 2

```
1 (define (fatorial n)
2   (if (= n 0)
3       1
4       (* n (fatorial (sub1 n)))))
5
6 (define fatorial-n-fatorial-n+1
7   (property ([n arbitrary-integer])
8             (= (* (add1 n) (fatorial n)) (fatorial (add1 n)))))
```

Também não funciona: desta vez o problema está na propriedade. Não especificamos que só faz sentido para números positivos.

# Fatorial - Solução

```
1 (define (fatorial n)
2   (if (= n 0)
3       1
4       (* n (fatorial (sub1 n)))))
5
6 (define fatorial-n-fatorial-n+1
7   (property ([n arbitrary-natural])
8             (= (* (add1 n) (fatorial n)) (fatorial (add1 n)))))
```

- Veja `https://docs.racket-lang.org/quickcheck` para a lista completa de geradores disponíveis
- Também é possível criar o seu próprio gerador

# QuickCheck Enfrenta a Conjectura de Collatz

- A conjectura de Collatz é bem simples de formular de maneira informal:
  - Dado um número natural, se o número é par então divida por 2; senão multiplique por 3 e adicione 1.
  - A conjectura afirma que, após um número finito de iterações, alcança-se o número 1 para qualquer número inicial.
- Vamos utilizar o QuickCheck para verificar a conjectura
- Lembre-se o QuickCheck dizer que algo passou em X testes apenas quer dizer que ele **não encontrou** nenhum contra-exemplo para a propriedade sendo testada, não que a propriedade tenha sido provada.
  - Isso é análogo a Unit Tests, onde os testes passarem não provam a ausência de bugs

# Collatz

```
1  (define (collatz-f n)
2    (if (par? n)
3        (/ n 2)
4        (add1 (* 3 n))))
5
6  (define (collatz n)
7    (if (= 1 n)
8        true
9        (collatz (collatz-f n))))
10
11 (define prop-collatz
12   (property ([n arbitrary-natural])
13             (collatz n)))
```

**Falha para** $n = 0$

# Collatz - Versão 1

```
1  (define (collatz-f n)
2    (if (par? n)
3        (/ n 2)
4        (add1 (* 3 n))))
5
6  (define (collatz n)
7    (if (= 1 n)
8        true
9        (collatz (collatz-f n))))
10
11 (define prop-collatz
12   (property ([n arbitrary-natural])
13             (collatz (add1 n))))
```

# Collatz - Alternativa

- Especifica o nosso próprio gerador
- Por que 100000? Por que não 1000000, INT_MAX, ...?

```
1  (define (collatz-f n)
2    (if (par? n)
3        (/ n 2)
4        (add1 (* 3 n))))
5
6  (define (collatz n)
7    (or (= 1 n) (collatz (collatz-f n))))
8
9  (define prop-collatz2
10   (property ([n (choose-integer 1 100000)])
11             (collatz (add1 n))))
```

# Pares e listas

# Pares e Listas

- Um par combina (surpreendentemente) 2 valores
  - O primeiro pode ser extraído com a função `car`
  - O segundo com a função `cdr`
- Uma lista é uma estrutura definida recursivamente
  - Pode ser vazia, definida por `null`
  - Pode ser um par, onde o segundo elemento é uma lista

# Pares

- Para construir um par utiliza-se a função `cons` que recebe 2 parâmetros, os dois valores a serem utilizados no par
- Não confundir com a função `const` que devolve uma função constante

```
> (cons 1 2)
'(1 . 2)
> (cons 1 '())
'(1)
```

- As funções `car` e `cdr` dão acesso aos elementos do par

```
> (car '(1 2))
1
> (car (cons 2 3))
2
> (cdr '(1 2))
'(2)
> (cdr '(1))
'()
```

# Listas

- Listas são estruturas recursivas construidas utilizando pares
- Pode-se pensar nelas de maneira análoga à listas encadeadas em linguagens de programação imperativas
- Listas podem ser criadas utilizando a função `list` ou utilizando a sintaxe especial com o caracter `'` (apóstrofo).

```
> (list 1 2 3 4)
'(1 2 3 4)
> (list (list 1 2) (list 3 4))
'((1 2) (3 4))
```

- `null` é a lista vazia

```
> '()
'()
> null
'()
> (null? null)
#t
> (null? '(1))
#f
> (null? '())
#t
```

# Listas - Operações Mais Comuns

- Linguagens funcionais tem como base para o processamento o uso eficiente de listas
- Algumas das operações mais comuns:
  - `(build-list n proc)` - Constrói uma lista de `n` elementos pela aplicação de `proc` para os inteiros de 0 a `(sub1 n)`, em ordem.
  - `(length lst)` - Devolve o número de elementos da lista.
  - `(append lst ...)` - Concatena 2 ou mais listas
  - `(reverse lst)` - Devolve uma lista com todos os elementos de `lst`, contudo com a ordem inversa
- Atenção: A maior parte das funções apresentadas só funciona com *listas limpas* (*proper lists*)!

# Listas - Iterando/Filtrando

- `(map proc lst ...)` - Devolve uma lista que contém o resultado da aplicação de proc em cada um dos elementos da(s) listas fornecidas como parâmetro
  - `ormap` e `andmap` são muito utilizadas e cobinam o map seguido de um or/and entre os resultados das aplicações de `proc` aos elementos de `lst`
- `(for-each proc lst)...` - Aplica a função proc para cada um dos elementos das listas recebidas como parâmetro. Não tem um valor de retorno e é útil pelos efeitos colaterais da função proc.
- `(filter pred lst)` - Devolve uma lista com o subconjunto dos elementos de `lst` para os quais a aplicação de `pred` é `#T`
- `(member v lst)` - localiza a primeira ocorrência de `v` em `lst` (retorna a sublista restante)

Uma lista mais completa das operações em listas pode ser vista em `https://docs.racket-lang.org/reference/pairs.html`

# Lambdas ou Funções Anônimas

- Frequentemente é conveniente criar funções para utilizar localmente, por exemplo, como parâmetro para as funções `map`, `filter`, ...
- Criar uma função utilizando `define` seria exagero nestes casos

```
1
2  (define (par? n)
3    ((= (modulo n 2) 0)))
4
5  (filter par? '(1 2 3 4 5 6))
6
7  ;; Que é equivalente a
8
9  (define par?
10   (lambda (n) (= (modulo n 2) 0)))
11
12 ;; Usando diretamente no filter
13 (filter (lambda (n) (= (modulo n 2) 0)) '(1 2 3 4 5 6))
```

Em tempo, o Racket já possui funções `even?` e `odd?` disponíveis na biblioteca padrão.

# Números Primos

- Queremos uma função `(primos-ate n)`, que devolva uma lista com todos os números primos até `n`
- Nossa função deve ter testada utilizando o QuickCheck

# Primos - Solução Força Bruta - Parte 1

```
1  (define (divisivel? a b)
2    (= (modulo a b) 0))
3
4  (define (possiveis-divisores a)
5    (cons 2
6          (cdr (build-list (floor (/ a 2))
7                           (lambda(x) (add1 (* x 2)))))))
8
9  (define (primo? n)
10   (or (= n 2) (and
11    (> n 2)
12    (not (ormap
13          (lambda (x) (divisivel? n x))
14          (possiveis-divisores n))))))
15
16 (define prop-primo-infatoravel
17   (property
18    ([n (choose-integer 2 (random 100000))])
19    (==> (primo? n) (not (ormap
20                          (lambda (x) (divisivel? n x))
21                          (possiveis-divisores n))))))
```

# Primos - Solução Força Bruta - Parte 2

```
1 (define (primos-ate n)
2   (filter primo? (build-list (sub1 n) (lambda (x) (+ 2 x)))))
3
4 (define prop-primos-ate
5   (property
6    ([n (choose-integer 2 (random 10000))])
7     (andmap primo? (primos-ate n))))
```

# Exercícios

# Exercícios

Para todos os exercícios, inclua testes baseados em exemplos ou testes baseados em propriedades, a depender do problema.

2. Converta as expressões infix para s-expression.
   - $1.2 \times (2 - 1 \div 3) + -8.7$
   - $(2 \div 3 + 4 \div 9) \div (5 \div 11 - 4 \div 3)$
   - $1 + 1 \div (2 + 1 \div (1 + 1 \div 2))$
   - $1 \times -2 + 3 \times -4 \times 5 \times -6 \times -7$
3. Defina um procedimento que tome três números como argumentos e retorne a soma dos quadrados dos dois maiores números
4. Defina um procedimento que calcule $\frac{1}{n}$. O procedimento deverá aceitar apenas números e deverá executar apenas divisões válidas
5. Um inteiro positivo é perfeito quanto a soma dos seus divisores (exceto o próprio número) é igual ao próprio número[1], e.g. $6 = 1 + 2 + 3$ e $28 = 1 + 2 + 4 + 7 + 14$. Construa o procedimento `perfeito?` (deve ser um predicado com esse nome) que determine se um um número $n$ é perfeito.
6. Defina um predicado `atom?` que verifica se um argumento é um *par* (do Lisp) ou não

[1] Alternativamente, é perfeito quando a metade da soma de todos os divisores é igual ao próprio número (e.g. 6 = (1+ 2 + 3 + 6)/2), mas nesse caso efetuamos uma soma e uma divisão desnecessariamente.

# Exercícios

Estude a solução de força bruta para listagem de primos e resolva os exercícios enunciados a seguir.

7. Crie testes utilizando o QuickCheck para verificar as funções `divisível?` e `possiveisDivisores`. Além disto, crie um teste adicional para a função `primos-ate` que verifica se a função obedece o parâmetro n, ou seja, que não são devolvidos primos maiores que n.
8. O n-ésimo número de Mersenne é dado pela fórmula $M_n = 2^n - 1$. Desde a antiguidade sabe-se que $M_2$, $M_3$, $M_5$ e $M_7$ são primos. Como 2, 3, 5 e 7 são primos, conjecturou-se que para todo $p$ primo, **então** $M_p$ também seria primo. Utilizando o QuickCheck mostre que este não é o caso.

# Convenções de indentação

# Convenções de indentação

## Motivação

- Em Lisp, instruções e dados são S-expressões
- S-expressões:
  - são prefixadas
  - balanceadas por parênteses
- É comum perder-se em “mar de parênteses”
- Recomendação: seguir as convenções de indentação

# Convenções de indentação

**Conv. 1**. Não deixar parênteses isolados

- Não-recomendado:

```
1 (define (fac n)
2    (if (zero? n)
3        1
4        (* n (fac (- n 1)))
5     )
6 )
```

- Recomendado:

```
1  (define (fac n)
2    (if (zero? n)
3        1
4        (* n (fac (- n 1)))))
```

# Apêndice: Convenções de indentação

**Conv. 2**. Subexpressões irmãs devem ter o mesmo alinhamento

- Não-recomendado

```
1 (list (foo)
2   (bar)
3   (baz))
```

- Recomendado (alt. 1):

```
1 (list (foo)
2       (bar)
3       (baz))
```

- Recomendado (alt. 2):

```
1 (list
2   (foo)
3   (bar)
4   (baz))
```

# Convenções de indentação

**Conv. 3**. Sempre que fechar um parênteses, tente quebrar a linha

Essa regra aplica-se mais a expressões longas, expressões curtas podem violá-la sem muito prejuízo

- Não recomendado:

```
1 (+ (* 2 3) (/ 17 5) (- 4 2))
```

- Recomendado:

```
1 (+ (* 2 3)
2    (/ 17 5)
3    (- 4 2))
```

# Convenções de indentação

**Conv. 4**. Sempre que quebrou linha para um operando, quebre para todos os outros

- Não recomendado:

```
1 (+ 1 foo
2    bar baz)
```

- Recomendado:

```
1 (+ 1
2    foo
3    bar
4    baz)
```

# Racket Básico: Tipos e QuickCheck

**Profs. Diogo S. Martins e Emilio Francesquini**

{santana.martins,e.francesquini}@ufabc.edu.br

MCTA016 - Paradigmas de Programação (Prática)

12 de junho de 2018

UFABC

CMCC
Centro de Matemática, Computação e Cognição



Crédito de parte das imagens, a menos se especificado: Wikipedia