Projeto de funções

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Vamos voltar para o problema da Márcia.

Depois que você fez o programa para o Alan, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Como proceder para projetar este programa?

Projeto de funções

Vamos usar um processo de projeto funções

- Análise
- Definição dos tipos de dados
- Especificação
- Implementação
- Verificação
- Revisão

Esse processo é inspirado no livro How to Design Programs.

Objetivos

Cada etapa tem um objetivo e depende das etapas anteriores

- Análise: identificar o problema a ser resolvido
- Definição dos tipos de dados: definir como as informações serão representadas
- Especificação: especificar com precisão o que a função deve fazer
- Implementação: implementar a função de acordo com a especificação
- Verificação: verificar se a implementação está de acordo com a especificação
- Revisão: identificar e fazer melhorias na especificação e implementação

Note que as vezes precisamos alterar a ordem das etapas, por exemplo, às vezes estamos na implementação e encontramos uma nova condição e devemos voltar e alterar a especificação.

Mas devemos evitar fazer a implementação diretamente!

Mas esse processe serve para projetar funções, como projetamos programas?

Um programa é composto de várias funções, então temos que decompor o programa em funções e aplicar o processo para projetar cada função.

Vamos treinar com problemas simples, de uma função, depois vamos utilizar o processo em problemas mais elaborados.

Depois que você fez o programa para o Alan, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Análise

- Quais informações são relevantes e quais podem ser descartadas?
- Existe alguma omissão?
- Existe alguma ambiguidade?
- Quais conhecimentos do domínio do problema são necessários?
- O que deve ser feito?

Resultado

Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar o álcool, senão a gasolina.

Análise

Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar o álcool, senão a gasolina.

Definição dos tipos de dados

- Quais são as informações envolvidas no problema?
- Como as informações serão representadas?

Resultado

Informações: preço do litro do combustível e o tipo do combustível.

Representações:

```
;; Preco é um número positivo

;; Combustivel é um dos valores
;; - "Alcool"
;; - "Gasolina"
```

Mesmo Racket sendo uma linguagem com vinculação dinâmica de tipos e não tendo apelidos de tipos e tipos enumerados, nós descrevemos em comentários os “tipos” que vamos utilizar. Nesse caso, os tipos servem como documentação.

Análise

Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar o álcool, senão a gasolina.

Tipos

```
;; Preco é um número positivo

;; Combustivel é um dos valores
;; - "Alcool"
;; - "Gasolina"
```

Especificação

- Assinatura da função
- Propósito (o que a função faz)
- Exemplos de entrada e saída

Resultado

```
;; Preco Preco -> Combustivel
;; Encontra o combustivel que deve ser
;; utilizado no abastecimento. Produz
;; "Alcool" se preco-alcool for até 70%
;; do preco-gasolina, produz "Gasolina"
;; caso contrário.
(define (seleciona-combustivel
          preco-alcool
          preco-gasolina) ...)
```

Exemplos

- Álcool 3.00, Gasolina 4.00, produz “Gasolina” ($3.00 < 0.7 \times 4.00$ é falso)
- Álcool 2.90, Gasolina 4.20, produz “Alcool” ($2.90 < 0.7 \times 4.20$ é verdadeiro)
- Álcool 3.50, Gasolina 5.00, não está claro na especificação o que fazer quando o preço do álcool é exatamente 70% ($3.50 = 0.7 \times 5.00$)!

Precisamos tomar uma decisão e modificar o propósito para ficar mais preciso. Vamos assumir que exatamente 70% também implica no uso do álcool (quais são as outras possibilidades?). O propósito modificado fica

```
;; Preco Preco -> Combustivel
;; Encontra o combustivel que deve ser utilizado no abastecimento.
;; Produz "Alcool" se preco-alcool for menor ou igual a 70% do preco-gasolina,
;; produz "Gasolina" caso contrário.
```

Propósito

No propósito da função descrevemos **o quê** a função faz, e não **como** ela faz (que é a implementação - as vezes precisamos dizer como ela faz, mas isso é raro).

No propósito também informamos as garantias da saída e as restrições sobre os parâmetros.

Diferença de “o quê” e “como”

Número par

- O quê: verifica se um número é par
- Como: faz o resto da divisão do número por 2 e compara com 0; ou; faz a divisão inteira do número e multiplica por 2 e compara com o número

Ordenação

- O quê: ordena os elementos de uma lista em ordem não decrescente
- Como: ordenação por seleção, por inserção, por intercalação, etc

Para saber se a especificação está boa, faça a segunda pergunta:

Um outro desenvolvedor, que não tem acesso ao problema original e nem a análise, tem as informações necessárias na especificação para fazer uma implementação e verificação inicial?

Se a resposta for sim, então a especificação está boa, senão ela está incompleta.

```
;; Preco Preco -> Combustivel
;; Encontra o combustivel que deve ser
;; utilizado no abastecimento. Produz
;; "Alcool" se preco-alcool for menor ou
;; igual a 70% do preco-gasolina, produz
;; "Gasolina" caso contrário.
(define (seleciona-combustivel
          preco-alcool
          preco-gasolina)
  ...)
```

3.00, 4.00, “Gasolina” ($3.00 \leq 0.7 \times 4.00$ é false)

2.90, 4.20, “Alcool” ($2.90 \leq 0.7 \times 4.20$ é true)

3.50, 5.00, “Alcool” ($3.50 \leq 0.7 \times 5.00$ é true)

Implementação

- É necessário conhecimento específico do domínio do problema? Então enumere o que será utilizado.
- Existem casos distintos? Então enumere os casos.
- É uma composição de funções? Então use pensamento desejoso e faça a composição das funções supondo que elas existam.
- Os dados de entradas tem autorreferência? Então faça a análise dos casos base e com autorreferência e chame a função recursivamente nos casos apropriados.

Temos duas formas de reposta, `"Alcool"` e `"Gasolina"`, portanto, precisamos de uma condição para distinguir quando utilizar cada resposta. No caso, a reposta é `"Alcool"` se `preco-alcool` é menor ou igual a 70% do preço de `preco-gasolina`; e `"Gasolina"` caso contrário.

```
;; Preco Preco -> Combustivel
;; Encontra o combustivel que deve ser utilizado no abastecimento. Produz
;; "Alcool" se preco-alcool for menor ou igual a 70% do preco-gasolina,
;; produz "Gasolina" caso contrário.
(define (seleciona-combustivel preco-alcool preco-gasolina)
  (if (<= preco-alcool (* 0.7 preco-gasolina))
      "Alcool"
      "Gasolina"))
```

```
;; Preco Preco -> Combustivel
;; Encontra o combustivel que deve ser
;; utilizado no abastecimento. Produz
;; "Alcool" se preco-alcool menor ou
;; igual a 70% do preco-gasolina,
;; produz "Gasolina" caso contrário.
(define (seleciona-combustivel
          preco-alcool
          preco-gasolina)
  (if (<= preco-alcool
          (* 0.7 preco-gasolina))
      "Alcool"
      "Gasolina"))
```

3.00, 4.00, então “Gasolina”. 2.90, 4.20, então “Alcool”. 3.50, 5.00, então “Alcool”.

Verificação

- A implementação está de acordo com a especificação?

Resultado

Vamos utilizar os exemplos que criamos na especificação para verificar se a resposta é a esperada.

```
> (seleciona-combustivel 3.00 4.00)
"Gasolina"
> (seleciona-combustivel 2.90 4.20)
"Alcool"
> (seleciona-combustivel 3.50 5.00)
"Alcool"
```

Preparem-se, agora vem uma sequência de muitas perguntas!

De forma geral, o fato de uma função produzir a resposta correta para alguns exemplos, implica que a função está correta? Não!

Então porque “perder tempo” fazendo os exemplos? O primeiro objetivo dos exemplos é ajudar o projetista a entender como a função funciona e o segundo ilustrar o seu funcionamento para que a especificação fique mais clara. Depois esses exemplos podem ser usados como uma forma inicial de verificação, que mesmo não mostrando que a função funciona corretamente, aumenta a confiança do desenvolvedor que o código está correto.

Já que os exemplos são uma verificação inicial, então temos que ampliar a verificação? Sim! De que forma? Testes de propriedades, fuzzing, etc. Para esta disciplina, vamos utilizar apenas os exemplos para fazer a verificação.

Nós fizemos os exemplos em linguagem natural e no momento de verificar os exemplos nós "traduzimos" para o Racket e fizemos as chamadas da funções de forma manual na janela de interações. Podemos melhorar esse processo? Sim!

Vamos escrever os exemplos diretamente em forma de código de maneira que eles possam ser executados automaticamente quando necessário. Para isso vamos usar um biblioteca, feita especialmente para essa disciplina.

Biblioteca de testes

Para instalar a biblioteca selecione o menu “File -> Install Package...”, digite o endereço “https://github.com/malbarbo/racket-test-examples.git” e clique em “Install”.

```
#lang racket
(require examples)

;; Preco Preco -> Combustivel
;;
;; Encontra o combustivel que deve ser utilizado no abastecimento.
;; Produz "Alcool" se preco-alcool menor ou igual a 70% do preco-gasolina,
;; produz "Gasolina" caso contrário.
(examples
 (check-equal? (seleciona-combustivel 3.00 4.00) "Gasolina")
 (check-equal? (seleciona-combustivel 2.90 4.20) "Alcool")
 (check-equal? (seleciona-combustivel 3.50 5.00) "Alcool"))

(define (seleciona-combustivel preco-alcool preco-gasolina)
  (if (<= preco-alcool (* 0.7 preco-gasolina))
      "Alcool"
      "Gasolina"))
```

Ao executarmos o programa obtemos algo como

```
3 success(es) 0 failure(s) 0 error(s) 3 test(s) run
```

Porque um teste pode falhar?

- O teste está errado
- A implementação está errada
- O teste e a implementação estão errados

Revisão

- Podemos melhorar a especificação e o código?
- Podemos fazer simplificações eliminando casos especiais (generalizando)?
- Podemos criar abstrações (definição de constantes e funções)?
- Podemos renomear os objetos?

Exemplo - aumento de salário

O governo deu uma aumento de salário para os funcionários públicos. O percentual de aumento depende do valor do salário atual. Para funcionários que ganham até R$ 1200 o aumento é de 10%, para funcionários que ganham entre R$ 1200 e R$ 3000 o aumento é de 7%, para funcionários que ganham entre R$ 3000 e R$ 8000, o aumento é de 3%, e finalmente, para os funcionários que ganham mais que R$ 8000 não haverá aumento. Projete uma função para calcular o novo salário de um funcionário qualquer.

Qual o primeiro passo?

Análise

Calcular o novo salário a partir do salário atual. Se o salário for $\leq$ R$ 1200, aumento de 10%, se $>$ R$ 1200 e $\leq$ R$ 3000, aumento de 7%, se $>$ R$ 3000 e $\leq$ R$ 8000, aumento de 3%, $>$ R$ 8000, sem aumento.

Exemplo - aumento de salário

Definição de tipos de dados

```
;; Salário é um número positivo com duas casas decimais
```

Especificação

```
;; Salario -> Salario

;; Calcula o novo salário a partir de um percetual de aumento determinado
;; a partir de salario-atual da seguinte forma:
;; - salario-atual <= 1200, aumento de 10%
;; - 1200 < salario-atual <= 3000, aumento de 7%
;; - 3000 < salario-atual <= 8000, aumento de 3%
;; - 8000 < salario-atual, sem aumento
(define (novo-salario salario-atual) ...)
```

```
(examples

; salario-atual <= 1200

(check-equal? (novo-salario 1000.00) 1100.00)
```

Falta alguma coisa nesse exemplo específico? Sim! Além do resultado esperado, é interessante fazer um comentário sobre como resultado foi obtido. Esse comentário irá nos auxiliar na etapa de implementação.

```
 ; salario-atual <= 1200
 (check-equal? (novo-salario 1000.00) 1100.00) ; (* 1000.00 1.10)
 (check-equal? (novo-salario 1200.00) 1320.00)
```

Note que também adicionamos um exemplo para o caso limite.

Exemplo - aumento de salário

```
; salario-atual <= 3000
(check-equal? (novo-salario 2000.00) 2140.00) ; (* 2000.00 1.07)
(check-equal? (novo-salario 3000.00) 3210.00)

; salario-atual <= 8000
(check-equal? (novo-salario 5000.00) 5150.00) ; (* 5000.00 1.03)
(check-equal? (novo-salario 8000.00) 8240.00)

; salario-atual > 8000
(check-equal? (novo-salario 8000.01) 8000.01)); 8000.00
```

Implementação

Quantas formas distintas de produzir o resultado da função identificamos nos exemplos? Quatro formas (veja os comentários dos exemplos).

Como existe mais de uma forma de resposta, então precisamos usar seleção. Quantos casos vamos precisar? Como são quatro formas de resposta, então precisamos de quatro casos, um para cada forma.Com essas informações já conseguimos esboçar o corpo da função

```
(define (novo-salario salario-atual)
  (cond
    []
    []
    []
    []))
```

Exemplo - aumento de salário

Agora olhamos para os exemplos, identificamos as condições que caracterizam cada caso, e fazemos a implementação.

Qual a condição que caracteriza o primeiro caso? `(<= salario-atual 1200)`

Qual é a forma da resposta para esse caso? `(* salario-atual 1.1)`

Agora podemos preencher o primeiro caso

```
(define (novo-salario salario-atual)
  (cond
    [(<= salario-atual 1200) (* salario-atual 1.1)]
    []
    []
    []))
```

Repetindo esse processo para os demais casos e chegamos na seguinte implementação

```
(define (novo-salario salario-atual)
  (cond
    [(<= salario-atual 1200) (* salario-atual 1.1)]
    [(and (< 1200 salario-atual) (<= salario-atual 3000)) (* salario-atual 1.07)]
    [(and (< 3000 salario-atual) (<= salario-atual 8000)) (* salario-atual 1.03)]
    [(> salario-atual 8000) salario-atual]))
```

Verificação

```
7 success(es) 0 failure(s) 0 error(s) 7 test(s) run
```

Revisão

```
(define (novo-salario salario-atual)
  (cond
    [(<= salario-atual 1200) (* salario-atual 1.1)]
    [(and (< 1200 salario-atual) (<= salario-atual 3000)) (* salario-atual 1.07)]
    [(and (< 3000 salario-atual) (<= salario-atual 8000)) (* salario-atual 1.03)]
    [(> salario-atual 8000) salario-atual]))
```

Como podemos melhorar o código?

- Eliminando condições redundantes
- Adicionando comentários sobre os número “mágicos”

Exemplo - aumento de salário

```
(define (novo-salario salario-atual)
  (cond
    [(<= salario-atual 1200) (* salario-atual 1.10)] ; 10% de aumento
    [(<= salario-atual 3000) (* salario-atual 1.07)] ;  7% de aumento
    [(<= salario-atual 8000) (* salario-atual 1.03)] ;  3% de aumento
    [else salario-atual]))                           ; sem aumento
```

Não podemos esquecer de fazer a verificação novamente!

Exemplo - massa tudo de ferro

O Jorge precisa saber a massa de diversos pequenos tubos de ferro mas está sem uma balança. No entanto, ele possui um paquímetro e pode medir com precisão o diâmetro interno e externo e a altura dos tubos, agora ele só precisa de um programa para fazer os cálculos. Algum voluntário?

Alguma coisa parece complicada nesse exercício?

Exemplo - massa tudo de ferro

Nesse exercício precisamos de conhecimento de um domínio (área), que talvez ainda não tenhamos, isso pode fazer o problema parecer mais difícil do que realmente é. Mas então, como proceder nesses casos?

Precisamos de uma pessoa (ou livros) que possam nos instruir sobre o conhecimento do domínio, geralmente os interessados no software podem indicar tais pessoas.

O importante é entender que o desenvolvedor de software geralmente resolve o problema de outras pessoas, e esses problemas podem envolver conhecimentos que não temos e por isso precisamos estar dispostos a estudar e aprender o conhecimento de outras áreas.

Vamos resolver esse problema, por onde começamos?

Análise

- Calcular a massa de um tubo de ferro a partir das suas dimensões. Como as dimensões de um tubo de ferro está relacionada com a massa do tubo?
- Dimensões $\rightarrow$ Volume $\rightarrow$ Massa
- Como determinamos o volume de um tubo de ferro a partir das suas dimensões? O volume de um tubo é dado por $\pi((diametro\_externo - diametro\_interno)/2)^2 \times altura$
- Como obtemos a massa a partir do volume? A massa é dado por $volume \times densidade$.
- Qual a densidade do ferro? A densidade do ferro é $7874\ kg/m^3$.

Definição de tipos de dados

```
;; Comprimento é um número positivo dado em metros.
;; Massa é um número positivo dado em quilogramas.
```

Especificação

```
;; Comprimento Comprimento Comprimento -> Massa
;; Calcula a massa de um tubo de ferro a partir das suas dimensões.
;; Requer que (> diametro-externo diametro-interno)
(define (massa-tubo-ferro diametro-externo diametro-interno altura) ...)

(examples
 ; (* pi (sqr (/ (- 0.05 0.03) 2)) 0.1 7874)
 (check-equal? (massa-tubo-ferro 0.05 0.03 0.1) 0.2472436))
```

Implementação

Precisamos utilizar seleção na implementação? Não! Por quê? Porque só existe uma forma de resposta, ou seja, a resposta é sempre calcula com a mesma expressão.

E que expressão é essa? A que identificamos na análise do problema e utilizamos para calcular a resposta do exemplo.

```
(define (massa-tubo-ferro diametro-externo diametro-interno altura)
  (* 3.14
     (sqr (/ (- diametro-externo diametro-interno) 2))
     altura
     7874)) ; densidade do ferro
```

Verificação

```
FAILURE
name:       check-equal?
location:   exercicios-resolvidos.rkt:48:1
actual:     0.24724360000000015
expected:   0.2472436
```

Comparação de igualdade de números de ponto flutuante quase não dá certo! Nesses casos, usamos `check-=` que permite especificar uma margem de erro.

```
(examples
 (check-= (massa-tubo-ferro 0.05 0.03 0.1) 0.2472436 0.00000001))
```

Revisão

```
(define (massa-tubo-ferro diametro-externo diametro-interno altura)
  (* 3.14
     (sqr (/ (- diametro-externo diametro-interno) 2))
     altura
     7874)) ; densidade do ferro
```

O que podemos melhorar?

- Definir constantes para os número “mágicos”
- Separar o cálculo do volume em etapas

```
(define PI 3.14)              ; Na prática precisamos de mais casas decimais!
(define DENSIDADE-FERRO 7874) ; Em kg/m^2

(define (massa-tubo-ferro diametro-externo diametro-interno altura)
  (define area-da-base (* PI (sqr (/ (- diametro-externo diametro-interno) 2))))
  (define volume (* area-da-base altura))
  (* volume DENSIDADE-FERRO))
```

Não podemos esquecer de fazer a verificação novamente!

Exemplo - ajuste texto

Em um determinado programa é preciso exibir textos em uma quantidade máxima de espaço (número de caracteres). Se o texto não cabe no espaço, apenas a parte inicial do texto que cabe no espaço junto de três pontos deve ser exibida. Além disso, o texto pode ser alinhado a direita, a esquerda ou centralizado. Projete uma função que transforme um texto para que possa ser exibido no espaço desejado.

Análise

Ajustar um texto a um tamanho específico, usando ..., se necessário, para sinalizar que o texto foi abreviado, e alinhar o texto a direita, a esquerda ou no centro.

Definição de tipos de dados

```
;; Alinhamento é um dos valores
;; - "direita"
;; - "esquerda"
;; - "centro"
```

Especificação

```
;; String Number Alinhamento -> String
;;
;; Produz uma nova string a partir de s que tem exatamente num-chars
;; caracteres e é alinhada de acordo com o alinhamento.
(define (ajusta-string s num-chars alinhamento) ...)
```

Essa especificação é precisa o bastante para fazermos uma implementação ou para usarmos essa função? Não.

Exemplo - ajuste texto

```
(examples
 (check-equal? (ajusta-string "casa" 4 "direita") "casa") ; "casa"
 (check-equal? (ajusta-string "casa" 4 "esquerda") "casa")
 (check-equal? (ajusta-string "casa" 4 "centro") "casa")
 (check-equal? (ajusta-string "casa verde" 7 "direita") "casa...")
 (check-equal? (ajusta-string "casa verde" 7 "esquerda") "casa...")
 (check-equal? (ajusta-string "casa verde" 7 "centro") "casa...")
 (check-equal? (ajusta-string "casa verde" 9 "direita") "casa v...")
 (check-equal? (ajusta-string "casa" 9 "direita") "     casa")
 (check-equal? (ajusta-string "casa" 9 "esquerda") "casa     ")
 (check-equal? (ajusta-string "casa" 9 "centro") "  casa   ")
 (check-equal? (ajusta-string "casa" 10 "centro") "   casa   "))
```

Exemplo - ajuste texto

O que está faltando nos exemplos?

A forma como as saídas foram computadas e as respectivas condições!

Lembrem-se, o objetivo inicial dos exemplos é ajudar o projetista a entender como a função deve funcionar.

Exemplo - ajuste texto

```
(examples
 ; (= (string-length s) num-chars)
 (check-equal? (ajusta-string "casa" 4 "direita") "casa") ; "casa"
 (check-equal? (ajusta-string "casa" 4 "esquerda") "casa")
 (check-equal? (ajusta-string "casa" 4 "centro") "casa")

 ; (> (string-length s) num-chars)
 ; (string-append (substring "casa verde" 0 (- 7 3)) "...")
 (check-equal? (ajusta-string "casa verde" 7 "direita") "casa...")
 (check-equal? (ajusta-string "casa verde" 7 "esquarda") "casa...")
 (check-equal? (ajusta-string "casa verde" 7 "centro") "casa...")
 (check-equal? (ajusta-string "casa verde" 9 "direita") "casa v...")
```

Exemplo - ajuste texto

```
; (and (< (string-length s) num-chars) (equal? alinhamento "direita"))
; (string-append (make-string (- 9 (string-length "casa")) #\space)
;                "casa")
(check-equal? (ajusta-string "casa" 9 "direita") "     casa")

; (and (< (string-length s) num-chars) (equal? alinhamento "esquerda"))
; (string-append "casa"
;                (make-string (- 9 (string-length "casa")) #\space))
(check-equal? (ajusta-string "casa" 9 "esquerda") "casa     ")
```

Exemplo - ajuste texto

```
; (and (< (string-length s) num-chars) (equal? alinhamento "centro"))
; (string-append
;   (make-string num-espacos-inicio #\space))
;   "centro"
;   (make-string num-espacos-fim #\space))
; onde
; num-espacos-inicio é (quotient (- 9 (string-length "casa")) 2)
; num-espacos-fim é (- 9 (string-length "casa") num-espacos-inicio)
(check-equal? (ajusta-string "casa" 9 "centro") "  casa   ")
(check-equal? (ajusta-string "casa" 10 "centro") "   casa   "))
```

Detalhamento do propósito da função a partir do aprimoramento do nosso entendimento obtido com os exemplos.

```
;; Se s tem exatamente num-chars caracteres, então produz s.
;;
;; Se s tem mais do que num-chars caracteres, então s é truncada e ...
;; é adicionado ao final para sinalizar que a string foi abreviada.
;;
;; Se s tem menos do que num-chars caracteres, então espaços são adicionados
;; no início se alinhamento é "esquerda", no fim se alinhamento é "direita",
;; ou no ínicio e fim se alinhamento e "centro". Nesse último caso, se a
;; quantidade de espaços adicionados for ímpar, então no fim será adicionado
;; 1 espaço a mais do que no início.
```

```scheme
(define (ajusta-string s num-chars alinhamento)
  (cond
    [(= (string-length s) num-chars) s]
    [(> (string-length s) num-chars) (string-append (substring s 0 (- num-chars 3)) "...")]
    [else
     (define num-espacos (- num-chars (string-length s)))
     (cond
       [(equal? alinhamento "direita")
        (string-append (make-string num-espacos #\space) s)]
       [(equal? alinhamento "esquerda")
        (string-append s (make-string num-espacos #\space))]
       [else
        (define num-espacos-inicio (quotient num-espacos 2))
        (define num-espacos-fim (- num-espacos num-espacos-inicio))
        (string-append
         (make-string num-espacos-inicio #\space)
         s
         (make-string num-espacos-fim #\space))])]))
```

Verificação

- Ok

Revisão

- Exercício para o leitor!

Implementação, versão alternativa

```
;; String Number Alinhamento -> String
;;
;; Produz uma nova string a partir de s que tem exatamente num-chars
;; caracteres e é alinhada de acordo com o alinhamento.
```

Na especificação podemos notar um “e”, indicando que a função faz duas coisas. Então, podemos implementar a função decompondo ela nessas “duas coisas”. Supomos que as funções existem e implementamos o corpo

```
(define (ajusta-string s num-chars alinhamento)
  (alinha (limita s num-chars)
          num-chars
          alinhamento))
```

Exemplo - ajuste texto

Agora colocamos essas duas funções em uma lista de trabalho, com um especificação inicial e depois procedemos para implementá-las seguindo as mesmas etapas

```
;; String Number -> String
;;
;; Produz uma nova string a partir de s com no máximo num-chars.
;; ...
(define (limita s num-chars) ...)

;; String Number Alinhamento -> String
;;
;; Produz uma nova string a partir de s alinhada de acordo com o alinhamento.
;; ...
(define (alinha s num-chars alinhamento)
```

Básicas

- Vídeos BSL
- Vídeos How to Design Functions