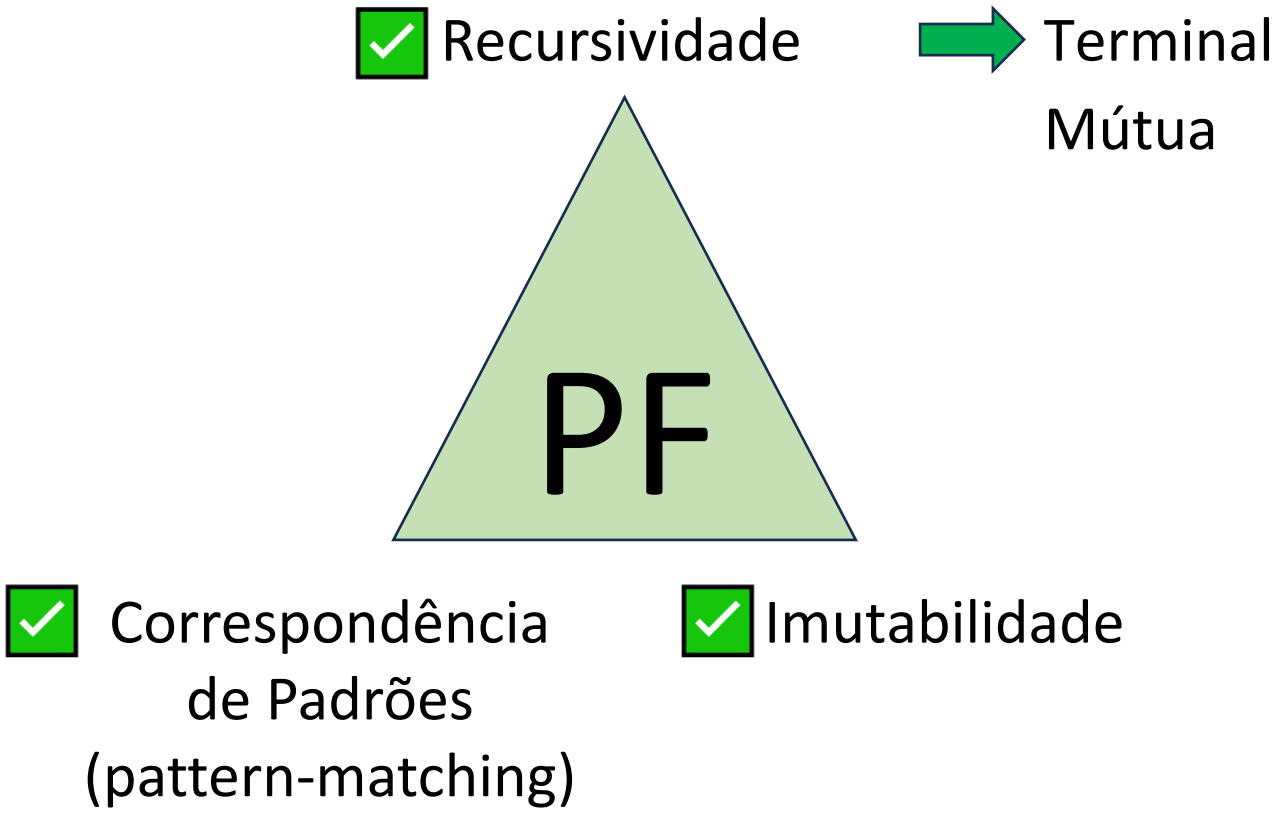
Aula 6: Recursividade e Funções em OCaml

UC: Programação Funcional

2023-2024

Até agora vimos



Recursividade terminal

- Agora já temos alguma prática com recursividade
  - "O código segue os dados": dados recursivos tratados recursivamente!
- Não há necessidade de ciclos e declarações de atribuição. A recursão é muitas vezes mais fácil:
  - Exemplo: Processamento de árvores
  - Exemplo: Anexar elementos a listas
- Mas cada chamada recursiva precisa de obter um "espaço novo" para os argumentos e as variáveis locais, sempre separado do "espaço" de quem chama

A seguir: vamos aperfeiçoar o nosso conhecimento de como as chamadas são executadas (call stacks)

- E veremos como a recursão pode ser tão *eficiente* como um ciclo!

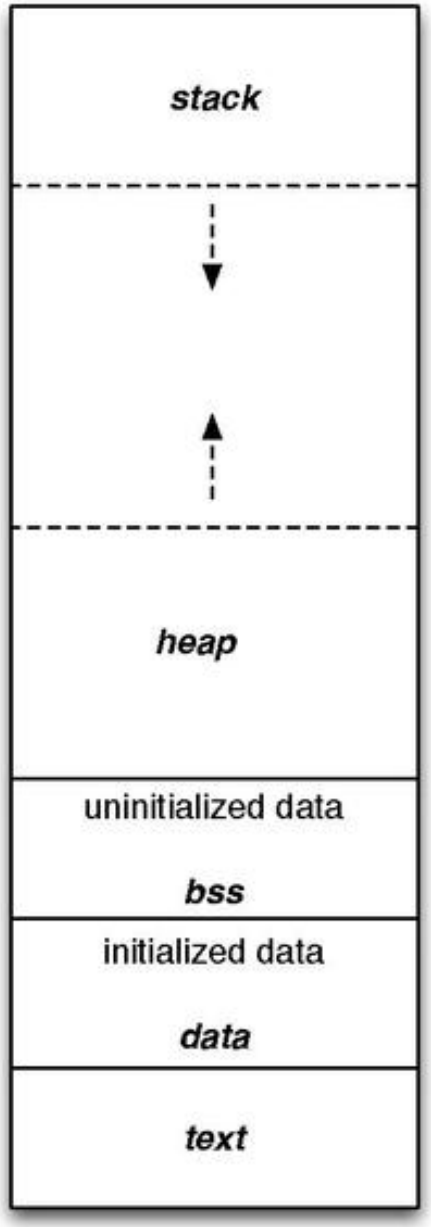
Pilha de Chamadas

As chamadas de função terminam na ordem inversa à do seu início

Enquanto um programa é executado, existe uma pilha de chamadas (call stack) para todas as chamadas de função que foram iniciadas mas ainda não foram concluídas

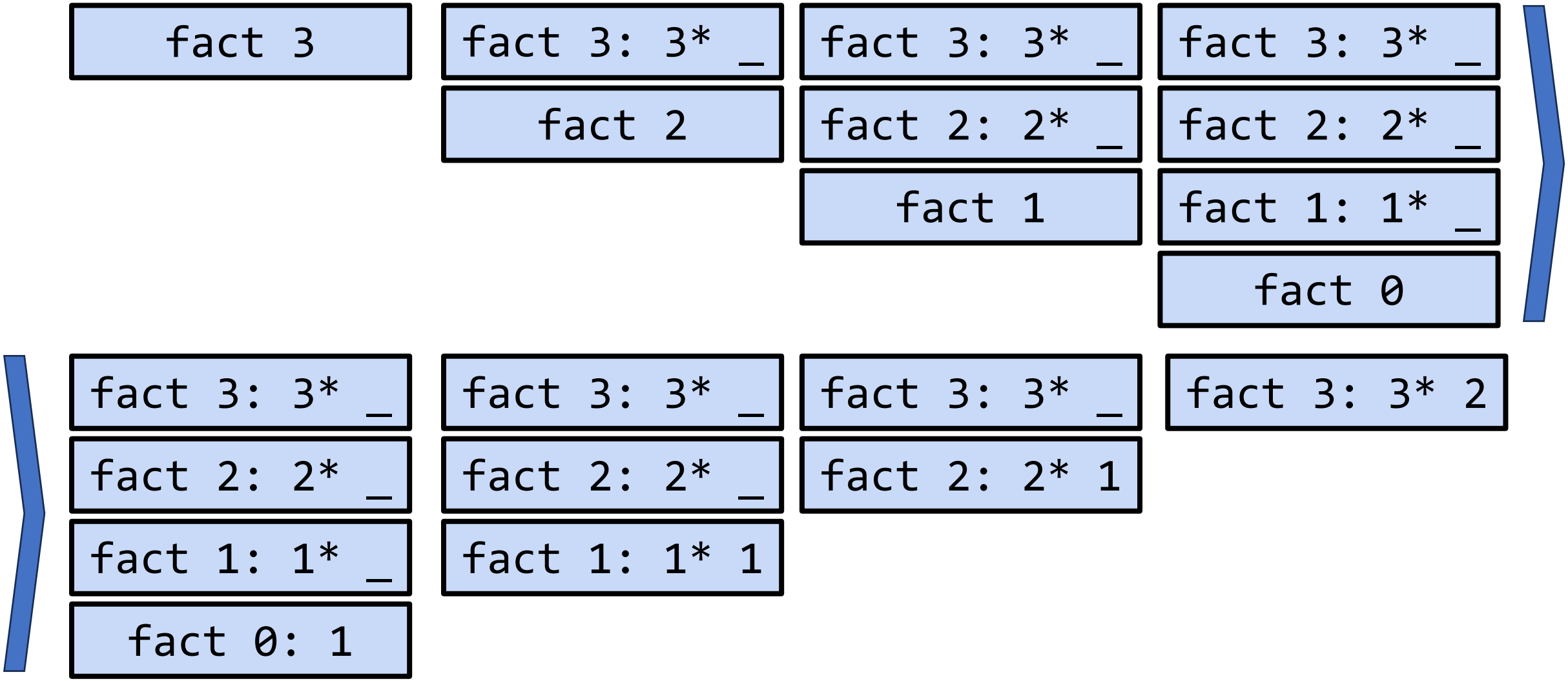
- Chamar **f** empurra a stack frame de **f** na pilha
- Quando a chamada termina, a sua stack frame é retirada da pilha
- Recursividade: Podemos ter > 1 stack frames para chamadas da mesma função

Uma stack frame armazena informação: valores de parâmetros, valores de variáveis locais e "o que resta a fazer" para o corpo da função



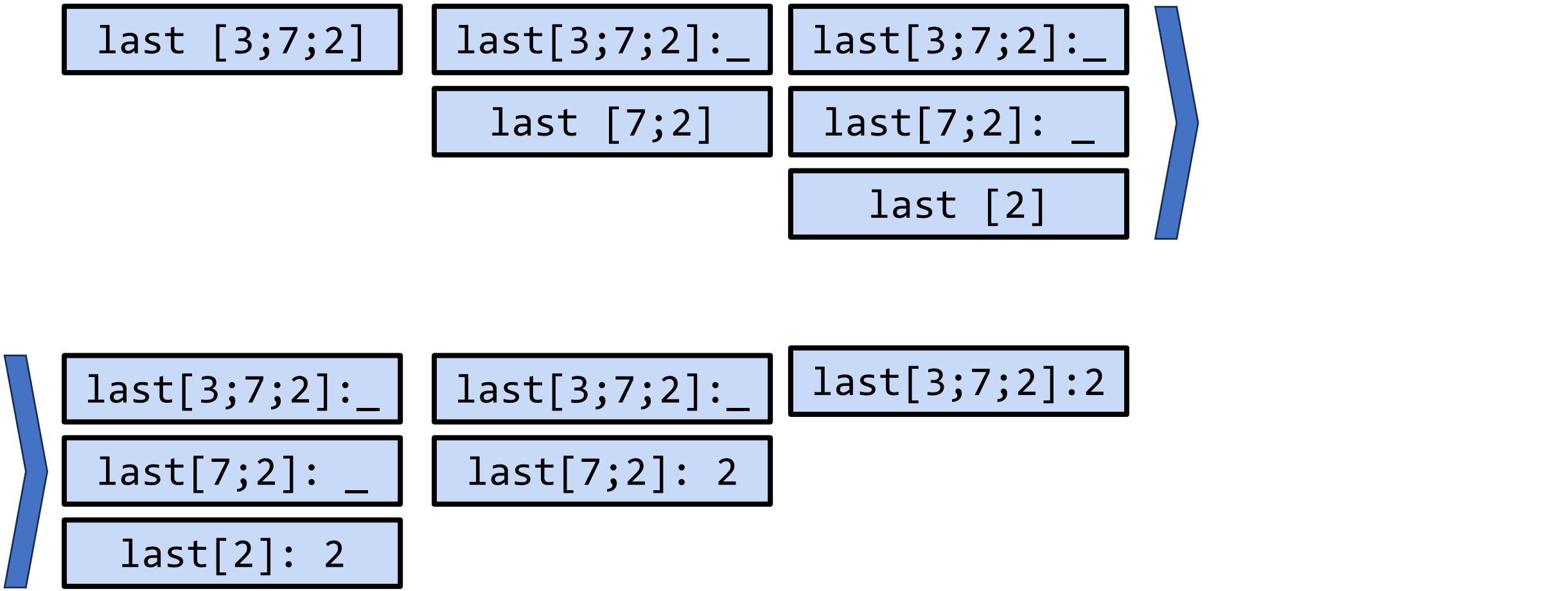
Exemplo

```
let rec fact n = if n=0 then 1 else n*fact(n-1)

let x = fact 3
```



Exemplo

```
let rec last xs =
  match xs with x::[] -> x | _::xs’ -> last xs’

let x = last [3;7;2]
```

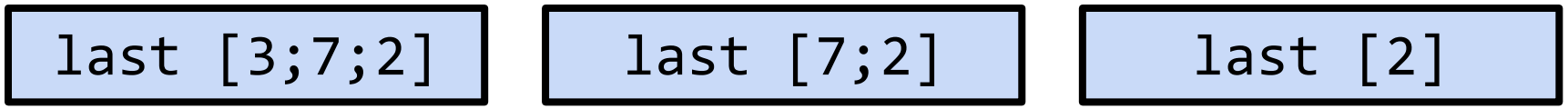


Precisamos SEMPRE de uma stack frame?

- Não é necessário manter uma stack frame apenas para obter o resultado de uma chamada de função e devolvê-lo imediatamente "tal como está"
  - Uma chamada que será a resposta da chamada atual "tal como está" é designada por chamada terminal
  - Uma função recursiva em que todas as chamadas recursivas são chamadas terminais é designada por "recursiva terminal"
- O OCaml lida com as chamadas terminais de forma especial otimizando-as
  - Retira a stack frame da função de chamada ANTES da chamada
  - Isto pode reduzir a utilização O(N) para O(1) 
  - "Não existe stack overflow"
  - Juntamente com outras otimizações é tão eficiente como um ciclo

A função `last` é recursiva terminal!

```
let last xs =
  match xs with x::[] -> x | _::xs’ -> last xs’

let x = last [3;7;2]
```



Tornando as funções em recursivas terminais

Por vezes, podemos reescrever o nosso algoritmo para ser recursivo terminal

```
let fact n =
  let rec loop (n,acc) =
    if n=0 then acc else loop (n-1,acc*n) in
  loop(n,1)
```

```
let x = fact 3
```

Por vezes, não o podemos fazer de forma fácil. Alguns exemplos:

- Processamento de uma árvore (duas chamadas recursivas não podem ser ambas chamadas terminais)
- Se a ordem for importante (e.g., concatenar uma lista de strings)

Metodologia

Existe uma *metodologia* para guiar a transformação para uma função recursiva terminal:

- Criar uma função auxiliar que recebe um *acumulador*
- O caso de base anterior torna-se no acumulador inicial
- O novo caso de base torna-se no acumulador final

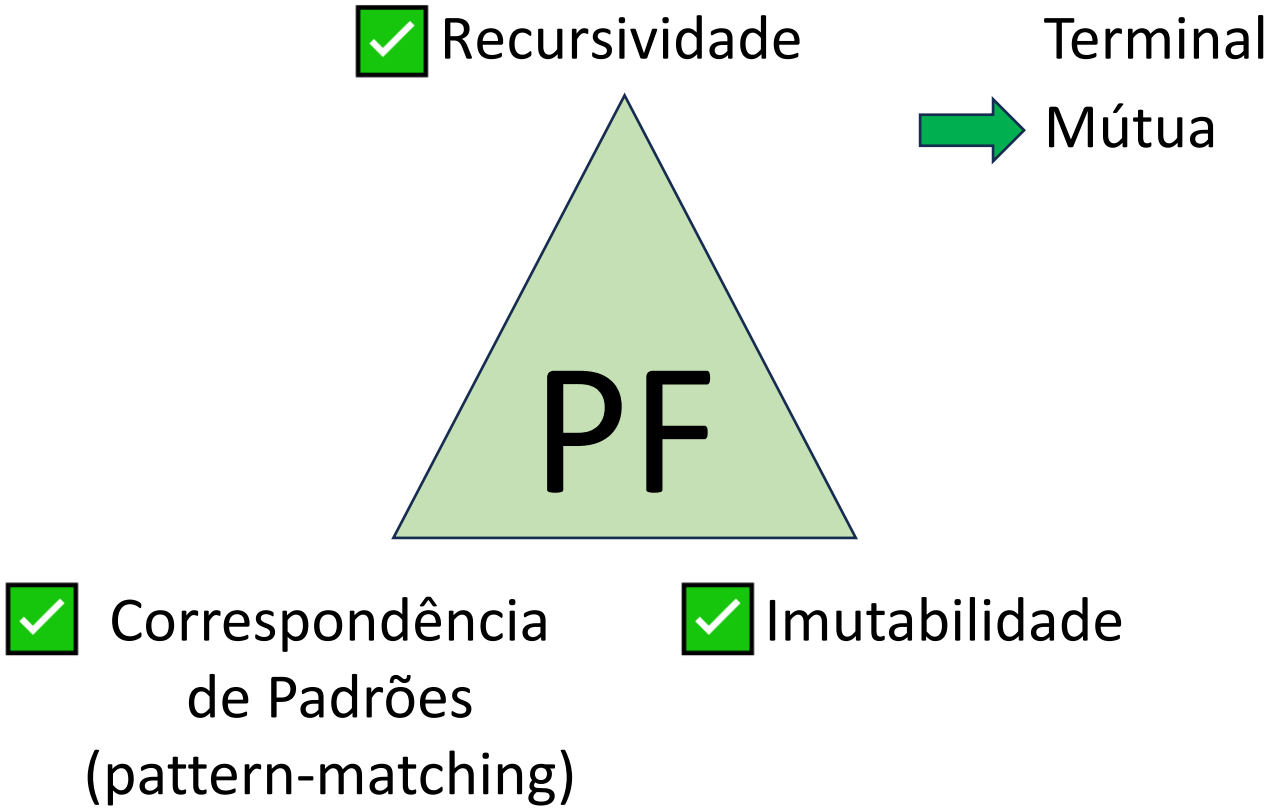
Cabe-nos a nós decidir “se isto funciona”

- Exemplo: para o fatorial, é fundamental que “a multiplicação seja associativa”

Moral / Não exagerar!

- A recursividade terminal nem sempre é viável ou elegante
- A recursividade terminal nem sempre é necessária: tenham cuidado com a otimização prematura e privilegiem a legibilidade
- Mas quando é viável e a eficiência é importante, é uma ferramenta fundamental para os programadores funcionais obterem a eficiência dos ciclos
    - Se conseguirem escrever como um ciclo, podem escrevê-lo como uma função recursiva terminal - os “valores atualizados” são argumentos para a chamada terminal!
- A maioria das linguagens funcionais, incluindo o OCaml, prometem a otimização da chamada terminal

Até agora vimos



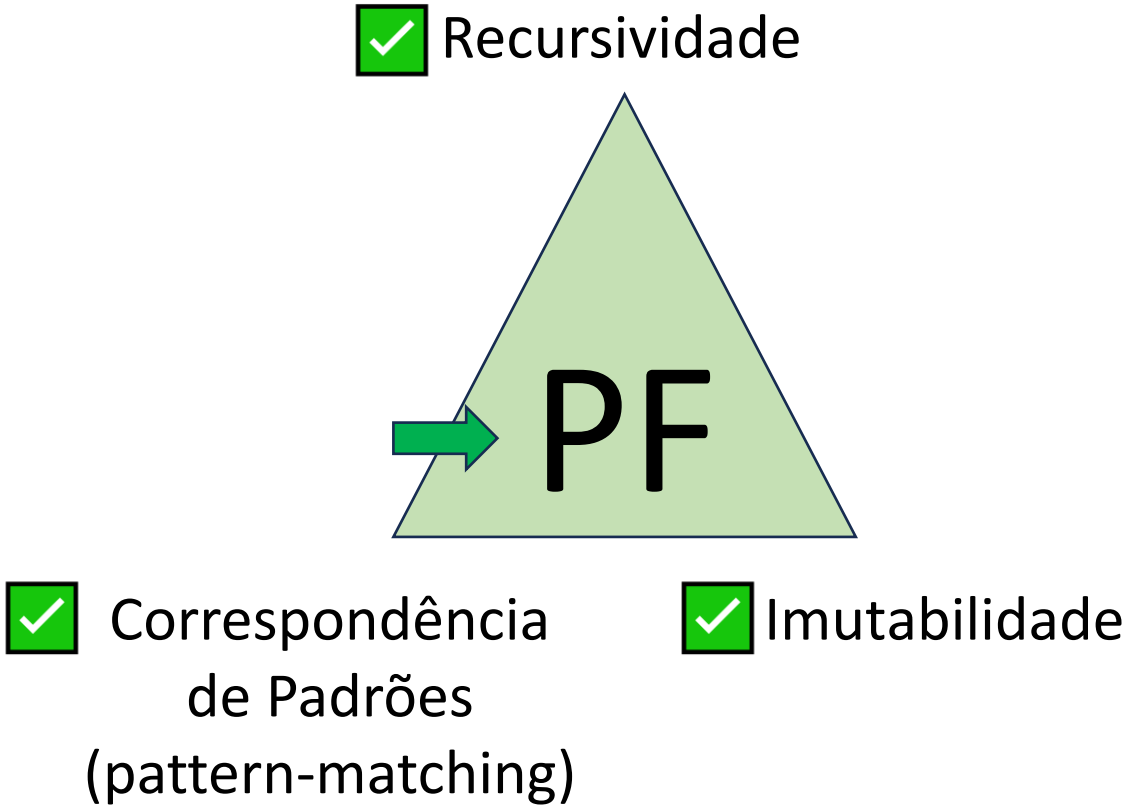
Recursividade Mútua

```
let rec start xs =
  match xs with
  | [] -> true
  | i::xs' -> if i mod 2 = 0 then saw_even xs' else saw_odd xs'
and saw_even xs =
  match xs with
  | [] -> true
  | i::xs' -> i mod 2 <> 0 && saw_odd xs'
and saw_odd xs =
  match xs with
  | [] -> true
  | i::xs' -> i mod 2 = 0 && saw_even xs'
```

Recursividade Mútua

```
let saw_even2 f xs =
  match xs with
  | [] -> true
  | i::xs' -> i mod 2 <> 0 && f xs'
let rec saw_odd2 xs =
  match xs with
  | [] -> true
  | i::xs' -> i mod 2 = 0 && saw_even2 saw_odd2 xs'
let start2 xs =
  match xs with
  | [] -> true
  | i::xs' -> if i mod 2 = 0 then saw_even2 saw_odd2 xs' else
saw_odd2 xs'
```

Até agora vimos



O que é a programação funcional?

- Não existe uma definição clara (“reconhece-se quando se vê”), mas normalmente inclui...
  - Imutabilidade na maioria/todos dos casos
  - Utilização de funções como valores

  ... e pode incluir ...
  - Utilização de tipos recursivos e de variantes recursivos
  - Um estilo mais próximo das definições matemáticas
  - Expressões idiomáticas que utilizam “laziness” (veremos adiante)
  - (Uma má definição) ~~Qualquer coisa que seja diferente de C …~~
- Uma *linguagem funcional* incentiva a programação funcional
  - Nem sempre acontece (em OCaml também temos referências, ciclos e outras construções imperativas – mas não as usaremos e não precisaremos delas)

Funções de primeira classe

- Primeira-classe significa que algo está “na linguagem” das expressões
  - Podemos computá-las, transmiti-las, colocá-las em estruturas de dados, etc.
- As funções em muitas linguagens, incluindo OCaml, são de primeira classe
  - As funções OCaml são [quase] valores
- A utilização mais comum é como argumento de outra função
  - Permite-nos abstrair sobre “o que computar” em determinadas situações: quem chama apenas passa a função a executar!
  - Uma função que recebe ou retorna funções é chamada de função de ordem superior

Fechos (Closures)

- As funções podem utilizar ligações do ambiente envolvente onde foram definidas
  - Mesmo que a função seja transmitida e chamada noutro local
  - Torna as funções de primeira classe *muito* mais poderosas e úteis
- Estudaremos esta questão cuidadosamente após alguns exemplos mais simples
  - Será necessário que os *valores* das funções não sejam apenas funções, mas também os ambientes em que foram definidas
  - "Função + ambiente" chamamos *fecho de função*
- Em teoria, uma linguagem pode ter funções de 1ª classe sem fechos ou vice-versa, mas normalmente uma linguagem com uma tem a outra

Exemplos

```
(* Uma função *)
let double x = x * 2
(* Outra função *)
let incr   x = x + 1

(* Uma lista de funções *)
let funcs = [double; incr]
```

```
(* Uma função que aplica
funções *)
let rec apply_funcs (fs, x) =
 match fs with
  | [] -> x
  | f :: fs' ->
    apply_funcs (fs', f x)
```

```
let foo = apply_funcs (funcs, 100) (* devolve 201 *)
let bar = apply_funcs (List.rev funcs, 100) (* 202 *)
```

Vamos ver …

1. Como utilizar funções de primeira classe e fechos de funções

2. A semântica exacta dos fechos de funções e das chamadas de funções

3. Múltiplas expressões (idiomas) que esta semântica permite (veremos em detalhe na próxima aula)

Funções como argumentos

- Podemso passar uma função como argumento para outra função
  - Mais uma vez, não se trata de uma “nova funcionalidade”, apenas não o fizemos antes
- Forma elegante de eliminar o código comum
  - Substituir N funções semelhantes com chamadas a 1 função com N argumentos de função diferentes

```
let rec n_times (f,n,x) =
 if n = 0 then x
 else f (n_times (f, n - 1, x))
```

Tipo da função `n_times`

- Qual é o tipo de **`n_times`**?

```
let rec n_times (f,n,x) =
 if n = 0 then x
 else f (n_times (f, n - 1, x))
```

- `val n_times : ('a -> 'a) * int * 'a -> 'a`
  - Os tipos são *inferidos* com base na forma como os argumentos são utilizados!
  - Mais útil do que `(int -> int) * int * int -> int`

Relação com os tipos

- As funções de ordem superior são frequentemente “tão reutilizáveis” que têm tipos polimórficos, ou seja, tipos com variáveis de tipo (e.g., `'a`)
- Existem funções de ordem superior que não são polimórficas
  - `val times_until_zero : (int -> int) * int -> int`
- E há funções polimórficas que não são de ordem superior
  - `val len : 'a list -> int`
- É sempre uma boa ideia compreender o tipo de uma função, especialmente para funções de ordem superior

Map

```
(* ('a -> 'b) * 'a list -> 'b list *)
let rec map (f, xs) =
  match xs with
   | [] -> []
   | x :: xs' -> f x :: map (f, xs')
```

Função de ordem superior da lista de honra

- O nome é normalizado (pode ser utilizado para qualquer tipo de dados)
  - Geralmente fornecidos nas bibliotecas para tipos de dados *standard*
- Utilizado a toda a hora em código funcional, portanto idiomático
  - Utilizá-lo quando apropriado não é apenas “menos código”, mas indica-nos o que estamos a fazer (um mapeamento)
  - Ao invés, não o utilizar quando “fazemos um mapeamento” confunde o leitor

Filter

```
(* ('a -> bool) * 'a list -> 'a list *)
let rec filter f xs =
  match xs with
   | [] -> []
   | x :: xs' -> if f x
                 then x :: filter f xs'
                 else filter f xs'
```

- Outra função de ordem superior da lista de honra
  - Mantem apenas alguns elementos de uma coleção
  - De novo uma expressão idiomática, por isso deve ser usada sempre que “estiver a fazer um filtro ”

Funções anónimas

- Sintaxe: `fun p -> e`
  - Em que `p` é um padrão e `e` é uma expressão
- Uma função como uma expressão!
  - A verificação de tipos e as regras de avaliação são “iguais” às funções
  - Note-se que se utiliza `->` em vez de `=` para separar parâmetros do corpo da função
  - A função não tem nome!
  - Notem que é “apenas mais uma expressão” - pode aparecer em qualquer sítio onde sejam permitidas expressões!

Utilização de funções anónimas

- Normalmente, as funções anónimas são utilizadas como argumentos para outras funções
  - Não é necessário dar um nome a uma função que só se utiliza uma vez!
- Uma limitação: as funções anónimas não se podem chamar a si próprias
  - Não têm nome para poderem fazer uma chamada recursiva
- As funções não recursivas são apenas açúcar sintático!!
  - `let f p = e` significa apenas `let f = fun p -> e`
  - Mais uma vez, não funciona para funções recursivas
  - E, como é habitual, o melhor estilo é utilizar o açúcar sintático disponível (simplifica e aumenta a legibilidade)

Uma questão de estilo

- **Mau:** `if e then true else false`
- **Bom:** `e`

- **Mau:** `fun x -> f x`
- **Bom:** `f`

- **Mau:** `n_times ((fun x -> List.tl x), 3, xs)`
- **Bom:** `n_times (List.tl, 3, xs)`

Generalizando

- Os nossos exemplos até agora têm sido muito semelhantes
  - Experimentem receber uma função como argumento e processem um número ou uma lista
- Podemos fazer muito mais!
  - Passar várias funções como argumentos
  - Colocar funções em estruturas de dados
  - Devolver funções como resultados
  - Escrever funções de ordem superior sobre tipos variantes
- Útil quando se pretende abstrair sobre “o que computar com”

Devolvendo funções

As funções são de primeira classe, pelo que podem ser devolvidas a partir de (outras) funções. Vejamos um exemplo, embora disparatado:

```
let double_or_triple f =
 if f 7
 then fun x -> 2 * x
 else fun x -> 3 * x
```

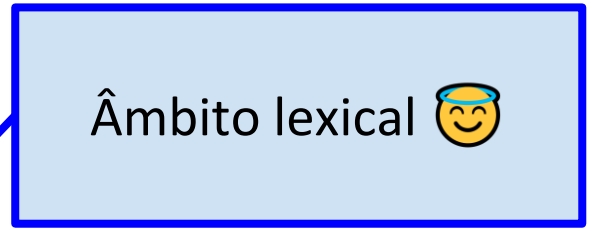
`double_or_triple` tem tipo **`(int -> bool) -> (int -> int)`**
Mas o OCaml Toplevel diz-nos que é **`(int -> bool) -> int -> int`**

- Isto é a mesma coisa
- O Toplevel do OCaml nunca imprime parêntesis desnecessários
- **`t1 -> t2 -> t3 -> t4`** significa o mesmo que
  `t1 ->(t2 ->(t3 -> t4))`
- Em breve saberemos porque é que esta precedência é conveniente

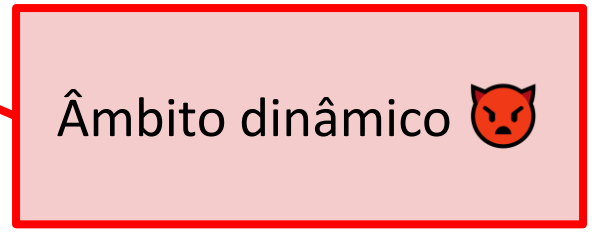
Outros tipos de dados

- As funções de ordem superior não são apenas para listas
- Funcionam muito bem para travessias comuns nos vossos próprios tipos de dados

Âmbito lexical



- Já sabemos que os corpos das funções podem utilizar qualquer coisa que esteja no seu âmbito...
  - Mas agora as funções estão a ser passadas / devolvidas! “Em que âmbito?”
- **Os corpos das funções são avaliados no ambiente em que foram definidos!**
  - **NÃO** no ambiente em que a função é chamada! ⚠
- Veremos a razão pelo qual o âmbito lexical é “a coisa certa”



Exemplo

```
(* linha 1 *) let x = 1
(* linha 2 *) let f y = x + y
(* linha 3 *) let x = 2
(* linha 4 *) let y = 3
(* linha 5 *) let z = f (x + y)
```

- A linha 2 define a função `f` que, quando chamada, avalia `x + y` dentro do ambiente onde `x ↦ 1` e `y ↦ arg`
- Na chamada da linha 5:
  - Procuramos por `f` para obter a definição na linha 2
  - Avaliamos a expressão do argumento `(x + y)` no ambiente atual
  - Chamar a função com o argumento `5`, resulta em `6 (+ 1)`

Fechos

- Como é que as funções podem ser avaliadas em ambientes antigos?
  - OCaml armazena ambientes antigos conforme necessário!
- "Semântica oficial" das funções (refinada):
  - Um valor de função tem duas partes:
    - O código
    - O ambiente a partir do momento em que a função foi definida
  - Isto é um "par" escondido do programador (sem `fst` / `snd`)
  - Tudo o que podemos fazer é "chamar a função com um argumento"
  - O "par" é chamado de fecho
  - As chamadas são avaliadas no ambiente do fecho alargado pelo mapeamento $\mapsto$ do parâmetro argumento

Exemplo

```
(* linha 1 *) let x = 1
(* linha 2 *) let f y = x + y
(* linha 3 *) let x = 2
(* linha 4 *) let y = 3
(* linha 5 *) let z = f (x + y)
```

- A linha 2 constrói a fecho e liga-o a **f**
  - Código: passamos **y** e temos o corpo **x + y**
  - Ambiente: **x ↦ 1** (mais o que tivermos no ambiente)
- A linha 5 chama o fecho com o argumento **5**:
  - Assim, o corpo avalia no ambiente com **x ↦ 1**, **y ↦ 5**

Âmbito lexical através de fechos

- Conhecemos a regra: a chamada avalia o corpo da função no ambiente onde a função foi definida
  - E o modelo para implementar o âmbito lexical é através de fechos
- Então, o que é que falta? Já temos tudo!
  - Vejamos alguns exemplos para demonstrar o âmbito lexical com funções de ordem superior
  - Discutirmos qual é a razão do **âmbito dinâmico** ser normalmente uma má ideia
  - Veremos algumas expressões idiomáticas poderosas com funções de ordem superior
    - Exemplo: passar funções para iteradores como o `filter`

A regra mantém-se inalterada

- Com funções de ordem superior, a nossa semântica permanece a mesma
  - O corpo da função avaliado no ambiente em que foi definido, estendido com/ parametro ↦ arg
- Nada muda quando passamos / devolvemos funções
  - Mas “o ambiente ” pode não ser o ambiente atual do toplevel
- Pode não ser intuitivo à primeira vista, mas torna as funções de primeira classe muito mais poderosas!

Devolvendo uma função

```
(* linha 1   *) let x = 1
(* linha 2   *) let f y =
(* linha 2 a *)   let x = y + 1 in
(* linha 2 b *)   fun q -> x + y + q
(* linha 3   *) let x = 3
(* linha 4   *) let g = f 4
(* linha 5   *) let y = 5
(* linha 6   *) let z = g 6
```

"Confiar na regra": a linha 4 liga **g** ao fecho:

- Código: passamos **q** e temos o corpo **x + y + q**
- Ambiente: **y ↦ 4, x ↦ 1,** ... (o que tivermos no ambiente)
- Este fecho adicionará sempre **9** ao seu argumento
- Portanto, a linha 6 liga **z** ao **15**

Passando uma função

```
(* linha 1   *) let f g =
(* linha 1 a *)   let x = 3 in
(* linha 1 b *)   g 2
(* linha 2   *) let x = 4
(* linha 3   *) let h y = x + y
(* linha 4   *) let z = f h
```

- "Confiar na regra": a linha 3 liga **h** num fecho:
  - Código: passamos **y** e temos o corpo **x + y**
  - Ambiente: **x ↦ 4, f ↦ <fecho>,** ... (o que tivermos no ambiente)
  - Este fecho adicionará sempre **4** ao seu argumento
- Portanto, a linha 4 liga **z** ao **6**
  - Nota: A linha 1a não pode afetar nada! Pode/deve ser eliminada.

Âmbito lexical, porquê?

- Âmbito Lexical: utilizar o ambiente onde a função foi definida
- Âmbito dinâmico: utilizar o ambiente onde a função é chamada
- Nos primórdios da PL, as pessoas reflectiam: *Que regra é melhor*?
  - A experiência tem mostrado que o âmbito lexical é *quase sempre* o correto
- Vejamos 3 razões precisas e técnicas para o fazer
  - Não se trata de uma questão de opinião!

Âmbito lexical, porquê?

1. O significado das funções não depende dos nomes das variáveis, apenas da "forma"

```
let f y =
  let x = y + 1 in
  fun q -> x + y + q
```

Exemplo: podemos mudar **f** para usar **w** em vez de **x**

- Âmbito Lexical: não pode alterar o comportamento da função
- Âmbito dinâmico: depende do ambiente de quem chama

```
let f g =
  let x = 3 in
  g 2
```

Exemplo: podemos remover variáveis não utilizadas

- Âmbito Lexical : variável não utilizada, não pode alterar o comportamento
- Âmbito dinâmico : algum g podem utilizar x e depender do facto de ser 3
  - Estranho 

Âmbito lexical, porquê?

2. As funções podem ser verificadas quanto ao tipo e fundamentadas quanto ao local onde são definidas

```
let x = 1
let f y =
    let x = y + 1 in
    fun q -> x + y + q
let x = "hi"
let g = f 4
let z = g 6
```

- Exemplo: o âmbito dinâmico tenta adicionar uma cadeia de caracteres e tem uma variável não ligada **y**

Âmbito lexical, porquê?

3. Os fechos podem armazenar facilmente os dados de que necessitam

```
let rec filter f =
  fun xs ->
   match xs with
   | [] -> []
   | x :: xs' ->
       if f x then
         x :: filter f xs'
       else
         filter f xs'

let greater_than x =
  fun y -> x < y
```

```
let is_positive =
  greater_than 0

let only_positives =
  filter is_positive

let all_greater (xs, n) =
  filter (greater_than n) xs
```

O âmbito dinâmico existe mesmo?

- O âmbito lexical é definitivamente o padrão presente na maioria das linguagens
- O âmbito dinâmico é ocasionalmente conveniente em algumas situações
  - Permite que o código “associe apenas variáveis utilizadas noutra função” para alterar o comportamento sem passar parâmetros
    - Pode ser conveniente, mas também um pesadelo 😱
- Se repararmos bem, o tratamento de exceções é semelhante ao âmbito dinâmico (veremos mais detalhes na próxima aula)
  - `raise e` salta para o handler “mais recente” (registado dinamicamente)
  - Não precisa de estar sintaticamente dentro do handler!

Quando as coisas avaliam

- Sabemos que:
  - O corpo da função não é avaliado até à chamada
  - Corpo da função avaliado sempre que a função é chamada
    - No ambiente a partir do momento em que a função foi definida
- Com fechos, é possível evitar a repetição da computação que não dependem dos argumentos da função
  - Pode tornar o código mais rápido, mas também tem uma semântica interessante

Recomputação

Ambos funcionam e são equivalentes

```
let all_shorter (xs,s) =
  filter (fun x -> String.length x < String.length s) xs

let all_shorter' xs s =
  let n = String.length s in
  filter (fun x -> String.length x < n) xs
```

all_shorter utiliza `String.length s` repetidamente (por cada x em xs)

all_shorter' utiliza `String.length s` apenas uma vez antes do **filter**

- Não há novas funcionalidades! Apenas uma nova utilização dos fechos.

Fold

```
let rec fold_left (f, acc, xs) =
 match xs with
  | [] -> acc
  | x :: xs' -> fold_left (f, f (acc, x), xs')
```

- Outra "função de ordem superior da lista de honra" conhecida por **`fold`** ou **`reduce`**
  - Acumula a resposta aplicando repetidamente **`f`** :
    - **`fold_left (f, acc, [v1; v2; v3]) é o mesmo que f (f (f (acc, v1), v2), v3)`**
  - **`fold_left : ('a * 'b -> 'a) * 'a * 'b list -> 'a`**
  - Por outro lado, **`fold_right`** percorre a lista "no sentido inverso"

Iteradores

- **`map`**, **`filter`**, **`fold`** são semelhantes aos padrões de iteração que vemos em linguagens imperativas
  - Mas não são “embebidas ” no OCaml - podemos defini-los nós mesmos!
  - “Apenas um idioma”! O poder da combinação de funcionalidades elegantes.
- Separa “travessia” de “computação”
  - Permite a reutilização do código de transvesia
  - Permite a reutilização do código de computação
  - Permite uma forma comum de falar sobre a travessia de estruturas de dados
    - A maioria das estruturas fornecem funções de **`map`**, **`filter`**, **`fold`** que “funcionam da mesma forma”

Créditos para Dan Grossman.