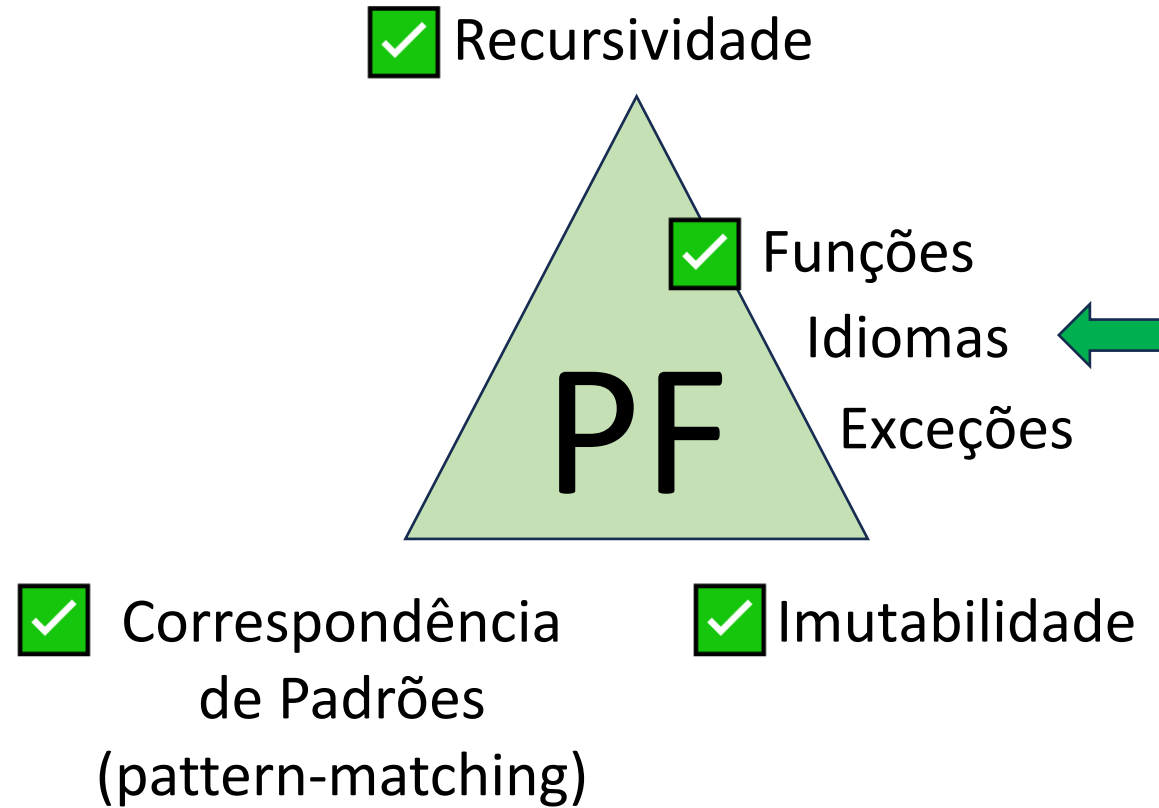


# Aula 7: Idiomas e Exceções em OCaml

UC: Programação Funcional  
2023-2024


# Até agora vimos



# Mais Idiomas

- Agora conhecemos a regra do âmbito lexical e do fecho de funções
  - Mas para que é que serve?

Lista parcial mas abrangente:

- Passar funções com dados privados para iteradores (já foi visto na aula passada)
- Currying (funções multi-arg e aplicação parcial) 
- Combinar funções (e.g., composição)
- Callbacks (e.g., em programação reactiva)
- Implementação de um ADT com um registo de funções (opcional)

# Currying

- Lembrem-se que cada função OCaml recebe exatamente um argumento
- Temos codificado  $n$  argumentos através de um  $n$ -tuplo
- Outra forma: Uma função recebe um argumento e retorna uma função que recebe outro argumento e ...
  - Chamado de “currying” em homenagem a Haskell Curry

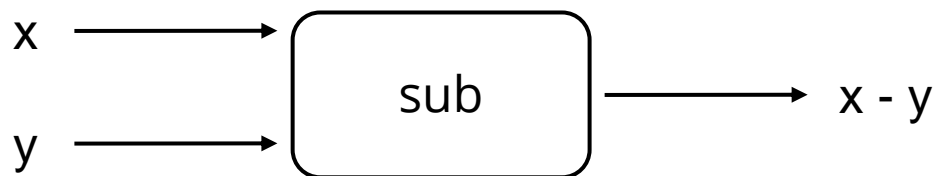


# Currying

- Fazer com que uma função receba o primeiro argumento (conceitualmente) e devolva outra função que receba o segundo argumento (conceitualmente) e assim por diante.

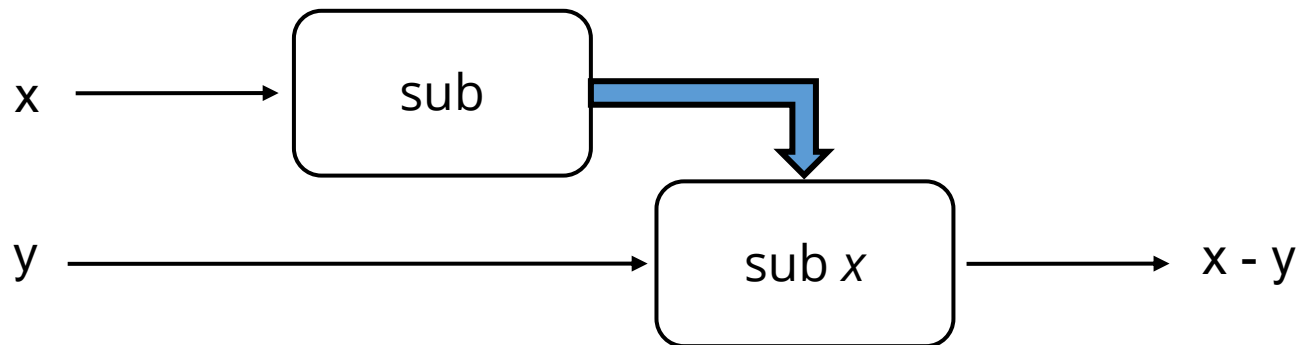
**Sem** Currying:

`let sub (x,y) = x - y`



**Com** Currying:

`let sub x y = x - y`

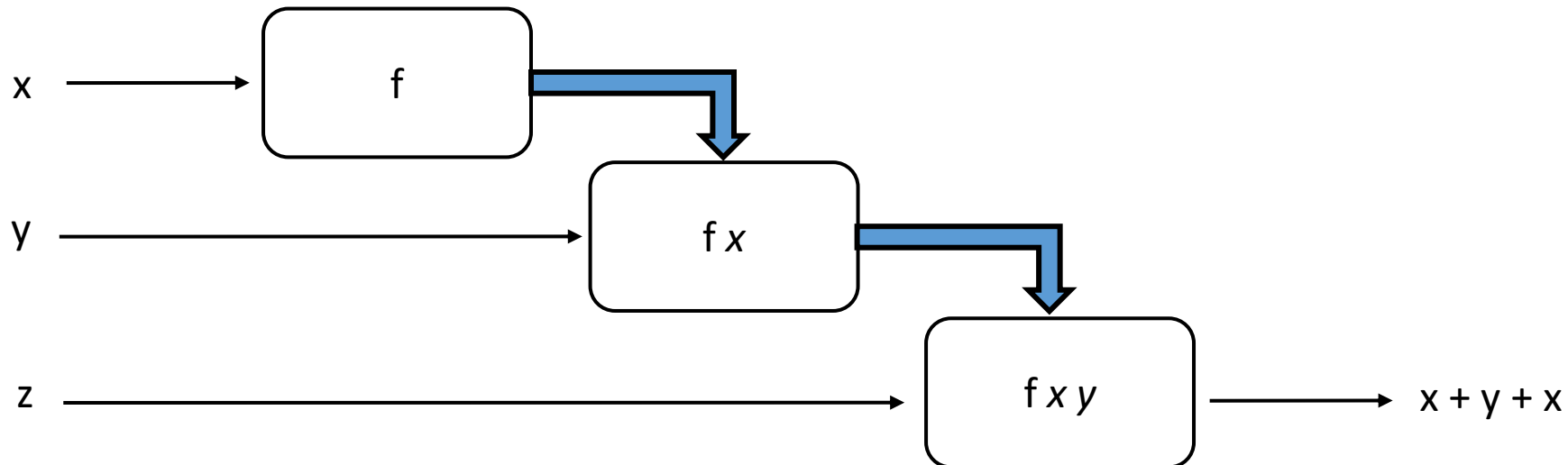


# Tipos das funções com Currying

```
let f x y z = x + y + z
```

Tipo:  $\text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}$

Tipo:  $\text{int} \rightarrow (\text{int} \rightarrow (\text{int} \rightarrow \text{int}))$  os parêntesis são opcionais



# Exemplo

```
let sorted3 =  
    fun x -> fun y -> fun z ->  
        z >= y && y >= x  
let t = ((sorted3 7) 9) 11
```

- Chamar `(sorted3 7)` devolve um fecho com:
  - O código `fun y -> fun z -> z >= y && y >= x`
  - O ambiente mapeia `x` para 7
- Chamar *essa* função de fecho com 9 devolve um fecho com:
  - O código `fun z -> z >= y && y >= x`
  - O ambiente mapeia `x` para 7, `y` para 9
- Chamar *essa* função com 11 devolve `true`

# Açúcar Sintático

```
let sorted3 =  
    fun x -> fun y -> fun z ->  
        z >= y && y >= x
```

```
let t = ((sorted3 7) 9) 11  
let t = sorted3 7 9 11
```

- De forma geral, **e1 e2 e3 e4 ...**, significa **(...((e1 e2) e3) e4)**
- Assim, em vez de **((sorted3 7) 9) 11**, podemos escrever **sorted3 7 9 11**
- Quem chama pode simplesmente pensar “função multi-argumento com espaços em vez de uma expressão com tuplos”
  - Difere dos tuplos; quem chama e quem recebe o valor de retorno deve utilizar a mesma técnica



# Açúcar Sintático

```
let sorted3 =  
  fun x -> fun y -> fun z ->  
    z >= y && y >= x
```

```
let sorted3 x y z =  
  z >= y && y >= x  
let t = sorted3 7 9 11
```

- Em geral, **let [rec] f p1 p2 p3 ... = e**,  
significa **let [rec] f p1 = fn p2 => fn p3 => ... => e**
- Podemos apenas escrever **let sorted3 x y z = z >= y && y >= x**
- Quem chama a função pode simplesmente pensar "função multi-argumento com espaços em vez de um padrão de tuplos"
  - Difere dos tuplos; quem chama e quem recebe o valor de retorno deve utilizar a mesma técnica

## Exemplo: Versão final

```
let sorted3 x y z =  
    z >= y && y >= x  
let t = sorted3 7 9 11
```

Tão elegante quanto o açúcar sintático (ainda com menos caracteres do que quando usamos tuplos):

```
let sorted3 =  
    fun x -> fun y -> fun z ->  
        z >= y && y >= x  
let t = ((sorted3 7) 9) 11
```

# Currying fold

O currying é particularmente conveniente para criar funções semelhantes com iteradores.

```
let rec fold f acc xs =  
  match xs with  
  [] -> acc  
  | h::t -> f (fold f acc t) h
```

Agora podemos usar esta função para definir uma função que soma elementos de uma lista:

```
let sum1 xs = fold (fun x y -> x+y) 0 xs
```

Mas isso é desnecessariamente complicado em comparação com a simples utilização de uma **aplicação parcial**:

```
let sum2 = fold (fun x y -> x+y) 0
```

# “Muito poucos argumentos”

- Anteriormente usávamos currying para simular múltiplos argumentos
- Mas se quem chama fornecer “muito poucos” argumentos, recebemos um fecho “à espera dos restantes argumentos”
  - Chamado de *aplicação parcial*
  - Conveniente e útil
  - Pode ser feito com *qualquer* função na forma currying
- Não se trata de uma nova semântica: ma de um idioma agradável e geral

# Iteradores Redux

A aplicação parcial é particularmente interessante para iteradores

- Implementações de iteradores fornecidos “colocar a função argumento primeiro”
- Isto é o que as bibliotecas fazem

```
let remove_negs = List.filter (fun x -> x >= 0)
let remove_all n = List.filter (fun x -> x <> n)
let remove_zeros = remove_all 0
```

# De facto

- O estilo geral em OCaml é currying para múltiplos argumentos
  - Não é usar tuplos
  - Esperem ver argumentos de modo currying nas bibliotecas, mesmo para funções de primeira ordem

```
let rec append xs ys = (* 'a list -> 'a list -> 'a list *)
  match xs with
  | [] -> ys
  | x::xs' -> x :: append xs' ys
```

- É “estranho” que tenhamos usado tuplos durante algumas aulas, mas quisemos mostrar currying depois de percebermos a sua semântica

# Eficiência / Conveniência

Então, o que é mais rápido/melhor: tuplos ou currying de múltiplos argumentos?

- Ambas são operações de tempo constante, por isso não importa na verdade na maioria da vezes - “é bastante rápido”
- Para a pequena parte em que a eficiência é importante:
  - Acontece que o OCaml compila currying de forma mais eficiente (otimiza a aplicação completa)
- A solução de compromisso mais interessante é a conveniência:
  - Aplicação parcial vs. “computar um tuplo para passar”

# A restrição de valor aparece 😞

Se utilizar a aplicação parcial para criar uma função polimórfica, esta pode não funcionar devido à **restrição de valor**


- Pode dar-lhe o tipo monomórfico com que o utilizou pela primeira vez ou um erro de tipo estranho
- Isto deve surpreender-vos; não fizeram nada de errado 😊 no entanto vão ter de alterar o código
- Veremos mais em detalhe quando discutirmos a inferência de tipos em OCaml



# Mais Idiomas

- Agora conhecemos a regra do âmbito lexical e do fecho de funções
  - Mas para que é que serve?

Lista parcial mas abrangente:

- Passar funções com dados privados para iteradores (já foi visto na aula passada)
- Currying (funções multi-arg e aplicação parcial)
- Combinar funções (e.g., composição) 
- Callbacks (e.g., em programação reactiva)
- Implementação de um ADT com um registo de funções (opcional)

# Composição de funções

O exemplo canónico é a composição de funções:

```
let compose f g = fun x -> f (g x)
```

- Cria um fecho que “lembra” a que **f** e **g** estão ligadas
- Tipo:  $('b \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \rightarrow ('a \rightarrow 'c)$
- Mas o Toplevel imprime algo *equivalente*
- Podemos fazer infix (apenas uma questão de estilo, não é específica do fecho):

```
let (%) = compose
```

# Exemplo

A terceira versão utilizando a nossa função infixada “é a melhor”

```
let sqrt_of_abs i = sqrt (float_of_int (abs i))
```

```
let sqrt_of_abs i = (sqrt % float_of_int % abs) i
```

```
let sqrt_of_abs = sqrt % float_of_int % abs
```

# Da esquerda para a direita ou vice-versa

```
let sqrt_of_abs = sqrt % float_of_int % abs
```

Como na matemática, a composição de funções é da “direita-para-a-esquerda”

- “obter o valor absoluto, converter para real e obter a raiz quadrada”
- “raiz quadrada da conversão para real do valor absoluto”

“Pipelines” de funções são comuns em programação funcional e muitos programadores preferem da esquerda-para-a-direita

- Predefinido exatamente assim em OCaml

```
let (|>) x f = f x
```

```
let sqrt_of_abs i = i |> abs |> float_of_int |> sqrt
```

## Outro exemplo

```
let pipeline_option f g =  
  fun x ->  
    match f x with  
    | None -> None  
    | Some y -> g y
```

Como é frequentemente no caso de funções de ordem superior, os tipos indicam o que a função faz:

```
('a -> 'b option) -> ('b -> 'c option) ->  
'a -> 'c option
```

# Mais funções de ordem superior

- E se pretendermos fazer o currying de uma função com tuplos ou vice-versa?
- E se os argumentos de uma função estiverem na ordem errada para a aplicação parcial pretendida?

Naturalmente, é fácil escrever funções *wrapper* de ordem superior


- E os seus tipos são fórmulas lógicas que nos dizem muito

```
let curried_of_paired f x y = f (x, y)
let paired_of_curried f (x, y) = f x y
let swap_tupled f (x, y) = f (y, x)
let swap_curried f x y = f y x
```


# Mais Idiomas

- Agora conhecemos a regra do âmbito lexical e do fecho de funções
  - Mas para que é que serve?

Lista parcial mas abrangente:

- Passar funções com dados privados para iteradores (já foi visto na aula passada)
- Currying (funções multi-arg e aplicação parcial)
- Combinar funções (e.g., composição)
- Callbacks (e.g., em programação reativa) 
- Implementação de um ADT com um registo de funções (opcional)

# O OCaml tem mutação (mas separada )

- As estruturas de dados mutáveis são aceitáveis em algumas situações
  - Quando “atualizamos o estado do mundo” é um modelo apropriado
  - Mas queremos que a maioria das construções da linguagem sejam verdadeiramente imutáveis
- Em OCaml fazemos isso com um construtor separado: *referências*
- Vamos utilizar as referências no próximo idioma
- Não utilizem referências nos trabalhos práticos 
  - É necessário praticar a programação sem mutações
  - A utilização delas conduzirão a soluções menos elegantes

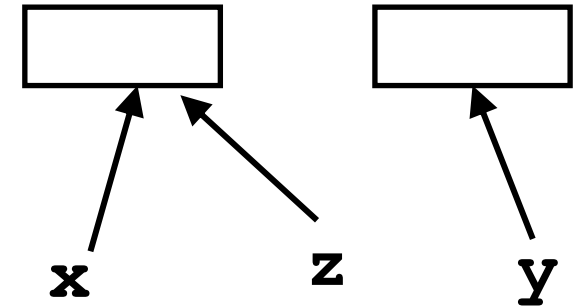


# Referências

- Novo tipo: **t ref** onde **t** é um tipo
  - Referências são de primeira classe
- Novas expressões:
  - **ref e** para criar uma referencia com o conteúdo inicial **e**
    - É apenas um registo com um *campo mutável* **contents**
  - **e1 := e2** para atualizar o conteúdo
  - **!e** para obter o seu conteúdo (não é a negação)
- O aliasing é importante para as referências devido à mutabilidade

# Exemplo com referências

```
let x = ref 42 (* : int ref *)
let y = ref 42
let z = x
let _ = x := 43
let w = (!y) + (!z) (* 85 *)
(* x + 1 não tipa *)
```



- Uma variável ligada a uma referência (e.g., **x**) continua a ser imutável: irá sempre referir-se à *mesma referência*
- Mas o *conteúdos* da referência podem mudar via `:=`
- E ainda podem existir **aliases** para a referência, que importa bastante
- As referências são valores de primeira classe

# Callbacks

Um idioma comum: A biblioteca pega em funções para chamar mais tarde, quando ocorre um evento – exemplos:

- Quando uma tecla é premida, o rato move-se, os dados chegam
- Quando o programa entra em algum estado (e.g., vira num jogo)

Uma biblioteca pode aceitar múltiplas chamadas de retorno

- Diferentes chamadas de retorno podem necessitar de diferentes dados privados com diferentes tipos
- Felizmente, o tipo de uma função não inclui os tipos de ligações no seu ambiente

# Estado mutável

Embora não seja absolutamente necessário, o estado mutável é razoavelmente apropriado aqui...

... Queremos mesmo que a “coleção de callbacks registradas” *mude* quando é chamada uma função para registrar uma callback

# Exemplo de uma biblioteca callback

A biblioteca mantém um estado mutável para “quais callbacks estão lá” e fornece uma função para aceitar novas callbacks

- Uma verdadeira biblioteca também permitiria a sua remoção, etc.
- No exemplo, as callbacks têm o tipo **`int->unit`**

Assim, toda a interface pública da biblioteca é a função para registo de novas callbacks:

```
val onKeyEvent : (int -> unit) -> unit
```

(Uma vez que as callbacks são executadas com side-effects, podem também necessitar de um estado mutável)

# Implementação da biblioteca

```
let callbacks : (int -> unit) list ref = ref []
```

```
let on_key_event f =  
    callbacks := f :: !callbacks
```

```
let do_key_event i =  
    List.iter (fun f -> f i) !callbacks
```

# Cientes

Podem apenas registrar um **int -> unit**, dado que se um outro dado é necessário, deverá estar no ambiente do fecho

- E se for necessário “lembrarmo-nos” de algumas “coisas”, podemos mutar o estado


Exemplos:

```
let times_pressed = ref 0
let _ = on_key_event (fun _ ->
    times_pressed := !times_pressed + 1
let print_if_pressed i =
    on_key_event (fun j ->
        if i = j
        then print_endline ("pressed " ^ string_of_int i)
        else ()))
```

# Mais Idiomas

- Agora conhecemos a regra do âmbito lexical e do fecho de funções
  - Mas para que é que serve?

Lista parcial mas abrangente:

- Passar funções com dados privados para iteradores (já foi visto na aula passada)
- Currying (funções multi-arg e aplicação parcial)
- Combinar funções (e.g., composição)
- Callbacks (e.g., em programação reactiva)
- Implementação de um ADT com um registo de funções (opcional) 



# Tipos de dados algébricos

Última expressão idiomática, closures podem implementar **tipos de dados abstratos**

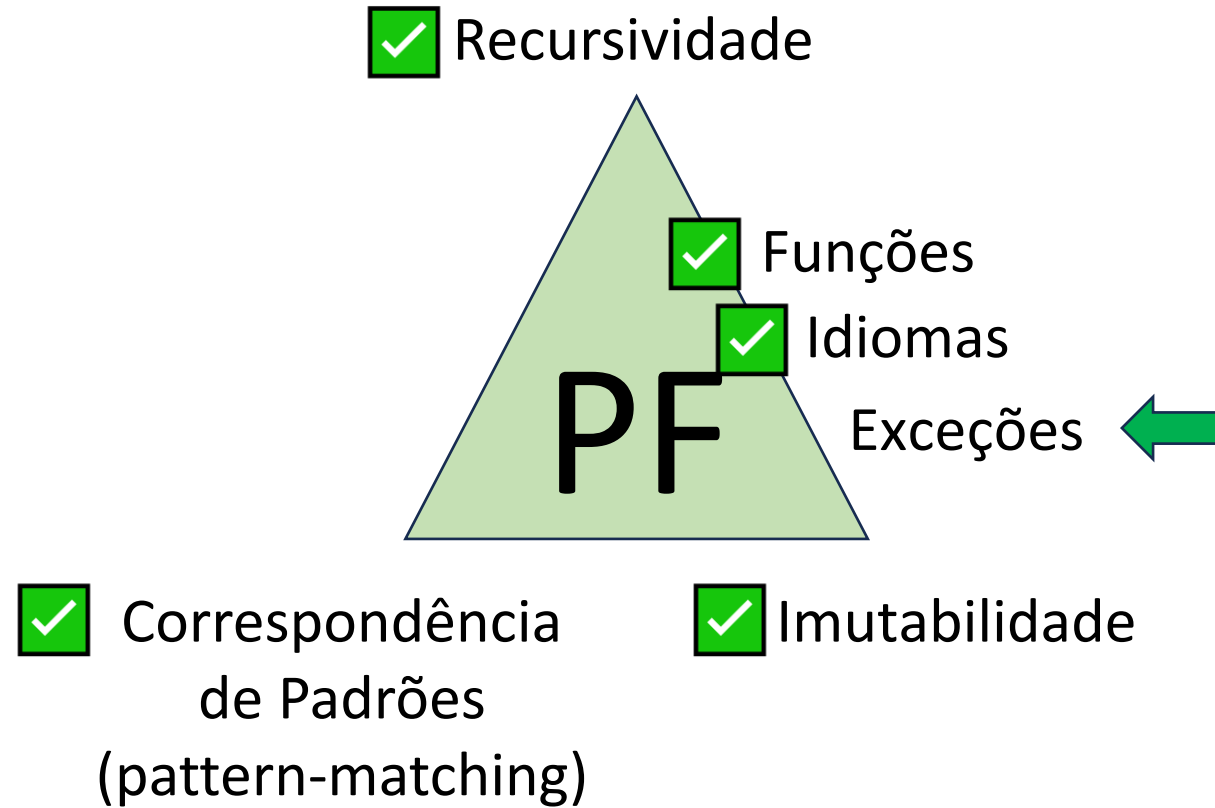
- Podem colocar várias funções num registo
- As funções podem partilhar os mesmos dados privados
- Os dados privados podem ser mutáveis ou imutáveis
- Parece-se muito com objectos, enfatizando que a POO e a programação funcional têm algumas semelhanças profundas

Tentem ver o código de uma implementação de conjuntos de números inteiros imutáveis com operações *insert*, *member* e *size*

O código é avançado, mas não acrescenta novas funcionalidades

- Combina o âmbito lexical, os tipos variante, os registos, os fechos, etc.
- A sua utilização é menos complicada

# Até agora vimos



# Exceções

As ligações de exceções (binding) declaram novos tipos de exceções:

```
exception Bogus  
exception BadNum of int
```

Expressões **raise** podem lançar uma exceção:

```
raise Bogus  
raise (BadNum 1)
```


Expressões **try ... with ...** podem apanhar a exceções

```
try e with Bogus -> 0  
try e with BadNum n -> n
```

# Levantamento de exceções

As ligações de exceção declaram novos tipos de exceção:

```
exception Bogus  
exception BadNum of int
```

Construímos exceções, que são apenas “valores” (  ) com estes construtores

```
Bogus  
(BadNum 1)
```

Isto é diferente de levantar uma exceção com a expressão **raise**:

```
raise Bogus  
raise (BadNum 1)
```

# De forma mais precisa

- As novas ligações de exceção acrescentam novas variantes ao “único tipo da exceção” **exn**
- Construimos valores do tipo **exn** tal como construimos tipos variantes com construtores
  - Pode passá-las de função em função, embora isso não seja muito *comum*
- Levantamos (lançamos) exceções com **raise e**
- Verificação do tipo:
  - **e** deverá ter tipo **exn**
  - O tipo do resultado é qualquer tipo que se queira (⚠)
  - Avaliação: Não produz um resultado (⚠), “cancela tudo” e “passa a exceção produzida por **e**” para a expressão **try** mais próxima na pilha de chamadas (call stack)...

# Try/with

As expressões **try ... with ...** conseguem apanhar exceções

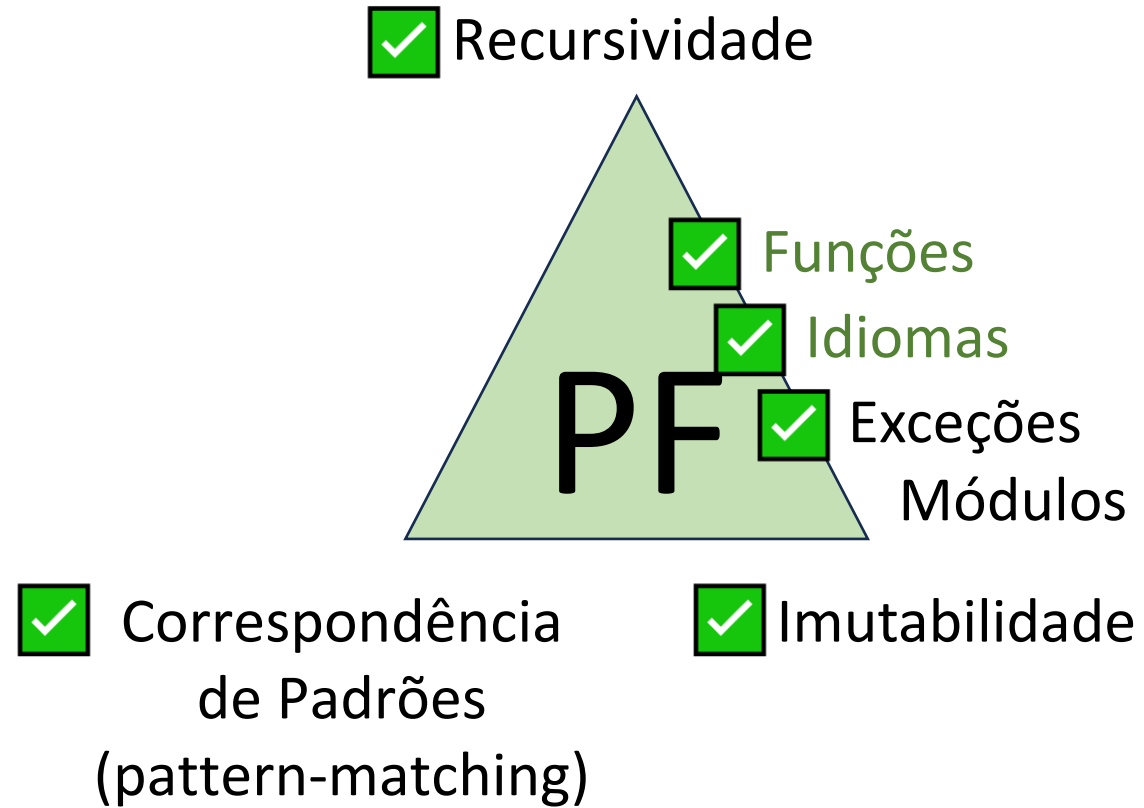
```
try e1 with Bogus -> e2  
try e1 with BadNum n -> e2
```

Regras de avaliação:

- Apenas avaliamos **e1** e isso dá-nos o resultado
- Mas se **e1** lançar uma exceção e essa exceção *coincidir* com o padrão (correspondência de padrões), então avaliamos **e2** e isso dá-nos o resultado

Verificação do tipo: **e1** e **e2** devem ter o mesmo tipo e isso é o tipo geral da expressão

# Até agora vimos



Créditos para Dan Grossman.