

# Aula 4: Expressões `let` Locais, Opções e Imutabilidade em OCaml

UC: Programação Funcional  
2023-2024

# Até agora vimos

- Tipos

Podemos *aninhar*

`int bool t1 -> t2 t1 * t2 * ... * tn t list`

- Expressões

Podemos *aninhar*

`34 true (e1,e2,...,en) [] e1::e2 x f(e1,e2  
,...,en) e1+e2 if e1 then e2 else  
e3 fst snd e=[] List.hd List.tl`

- Bindings

???

`let x = e`

`let [rec] f ((x1:t1), ... (xN:tN)) = e`

let Local

# Expressões `let`

Expressões podem ter as suas próprias **variáveis** (privadas) e **funções**

- Para estilo e comodidade: vamos dar nomes ao resultados intermédios
- Para consistência: tudo o resto pode aninhar-se
- Para eficiência: evitar a computação desnecessária
  - *Não é apenas “um pouco mais rápido”*
- Para segurança (“safety”) e manutenção: ocultar pormenores de implementação

# Expressões **let** locais

Sintaxe: `let  $x$  =  $e1$  in  $e2$`

## Verificação do Tipo

- Se  $e1$  tem tipo  $t1$  no atual ambiente estático  $E$
- Se  $e2$  tem tipo  $t2$  no ambiente estático **estendido com  $x \mapsto t1$**
- Então `let  $x$  =  $e1$  in  $e2$`  tem tipo  $t2$
- Senão, reportar erro e falhar

# Expressões **let** locais

Sintaxe: `let  $x$  =  $e1$  in  $e2$`

## Avaliação

- Se  $e1$  avalia em  $v1$  no atual ambiente dinâmico  $E$
- Se  $e2$  avalia em  $v2$  no ambiente dinâmico **estendido com**  $x \mapsto v1$
- Então `let  $x$  =  $e1$  in  $e2$`  avalia em  $v2$

# Expressões **let** “são apenas expressões”

*As expressões **let** podem ser usadas em **qualquer lado** onde uma expressão possa ser usada*

*Porque uma expressão **let** é uma expressão*

Ainda dizemos: “Múltiplos bindings” é apenas um aninhamento de expressões **let**

- Mas como estilo, não os indentamos dessa forma ( ... )

# Expressões **let** “são apenas expressões”

- Âmbito
  - Mais controlo sobre as partes de um programa que podem aceder a uma ligação (binding)
  - Para expressões **let** : apenas o *corpo do let* !
- Nada mais é realmente novo
  - **expressões let** funciona de forma semelhante como os **let bindings** normais (top-level)



# Expressões **let**

Sintaxe: **let** [**rec**] **f** ((**x1:t1**), ..., (**xN:tN**)) = **e** **in** **e2**

Verificação de tipo: O mesmo que o binding de funções, usando o ambiente estático onde a expressão **let** ocorre

Regras de Avaliação: O mesmo que o binding de funções (top-level), utilizando o ambiente dinâmico onde ocorre a expressão **let**

Um binding local: **f** usado apenas em **e** (se **rec** estiver presente) e **e2** (usamos sempre)

# Exemplos de expressões `let`

Não é um estilo muito bom:

```
let nats_upto (x : int) =  
  let rec range ((lo : int), (hi : int)) =  
    if lo < hi then  
      lo :: range (lo + 1, hi)  
    else  
      []  
  in  
  range (0, x)
```

```
nats_upto 5;;  
- : int list = [0; 1; 2; 3;  
4]
```

O `range` está escondido...  
Mais ninguém o pode utilizar!

Além disso, o `range`  
necessita do parâmetro `hi` ?

# Exemplos de expressões `let`

**Melhor estilo:**

```
let nats_upto (x : int) =  
  let rec loop (lo : int) =  
    if lo < x then  
      lo :: loop (lo + 1)  
    else  
      []  
  in  
  loop 0
```

```
nats_upto 5;;  
- : int list = [0; 1; 2; 3; 4]
```

- As funções podem utilizar bindings do ambiente onde são definidas, incluindo:
  - ambientes “exteriores”
  - parâmetros da função externa
  - etc.
- Parâmetros desnecessários são um mau estilo

# Funções aninhadas

- É de bom-tom definir funções auxiliares no seu interior, se
  - Não é útil noutro local (evita a desorganização)
  - Possibilidade de utilização abusiva noutro local (melhora a fiabilidade)
  - Suscetível de ser alterado ou suprimido (ocultar pormenores da aplicação)
- “Tradeoff” fundamental:
  - A reutilização de código poupa esforços e (normalmente) evita bugs
  - No entanto, torna-o mais difícil de alterar mais tarde (dependemos dos seus pormenores)

# Evitar a recursão repetida

- Qual é o volume de trabalho desta função?

```
let rec bad_max (xs : int list) =  
  if xs = [] then  
    0 (* bad style, will fix later *)  
  else if List.tl xs = [] then  
    List.hd xs  
  else if List.hd xs > bad_max (List.tl xs) then  
    List.hd xs  
  else  
    bad_max (List.tl xs)
```

# Evitar a recursão repetida

- Qual é o volume de trabalho desta função?

```
bad_max (List.rev (nats_upto 50));;  
- : int = 49 (* retorna em microsegundos  
*)
```

```
bad_max (nats_upto 50);;  
- : int = ... (* ainda em funcionamento *)
```

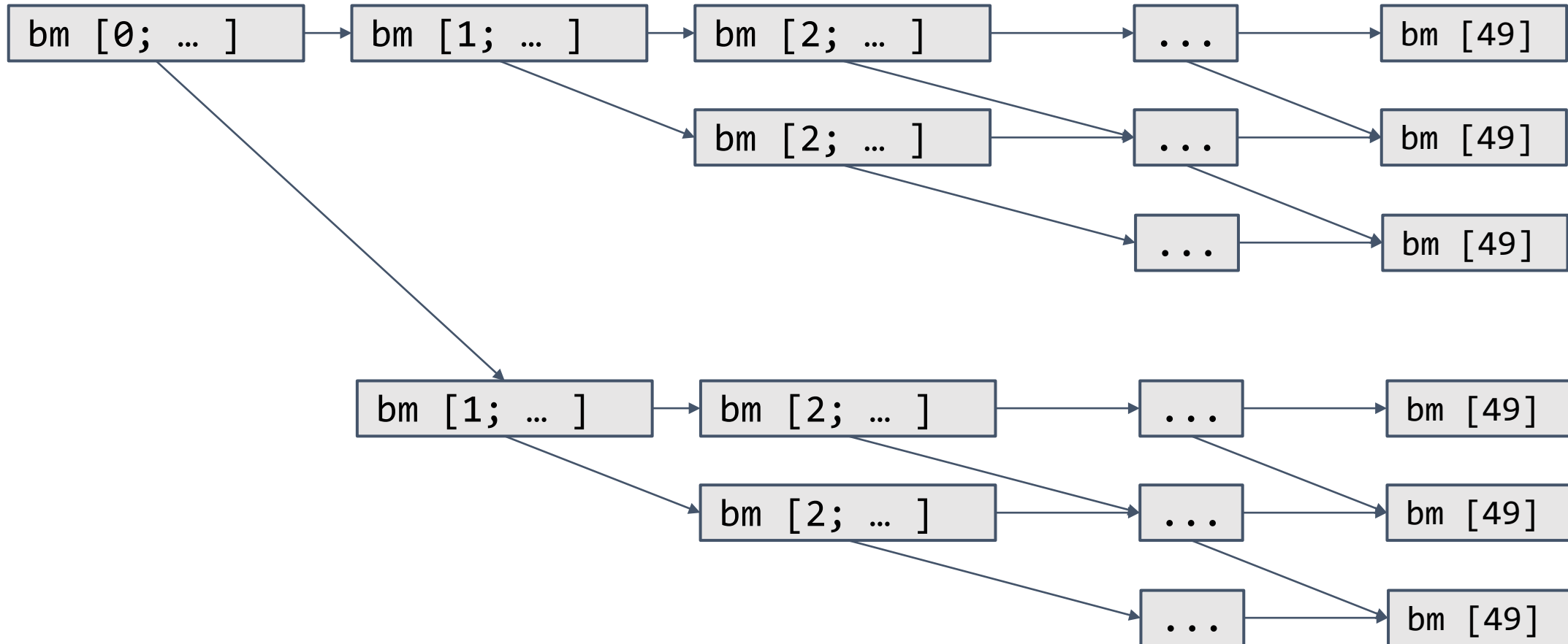
# Rápido Vs. inutilizável

```
if List.hd xs > bad_max (List.tl xs)
then List.hd xs
else bad_max (List.tl xs)
```



# Rápido Vs. inutilizável

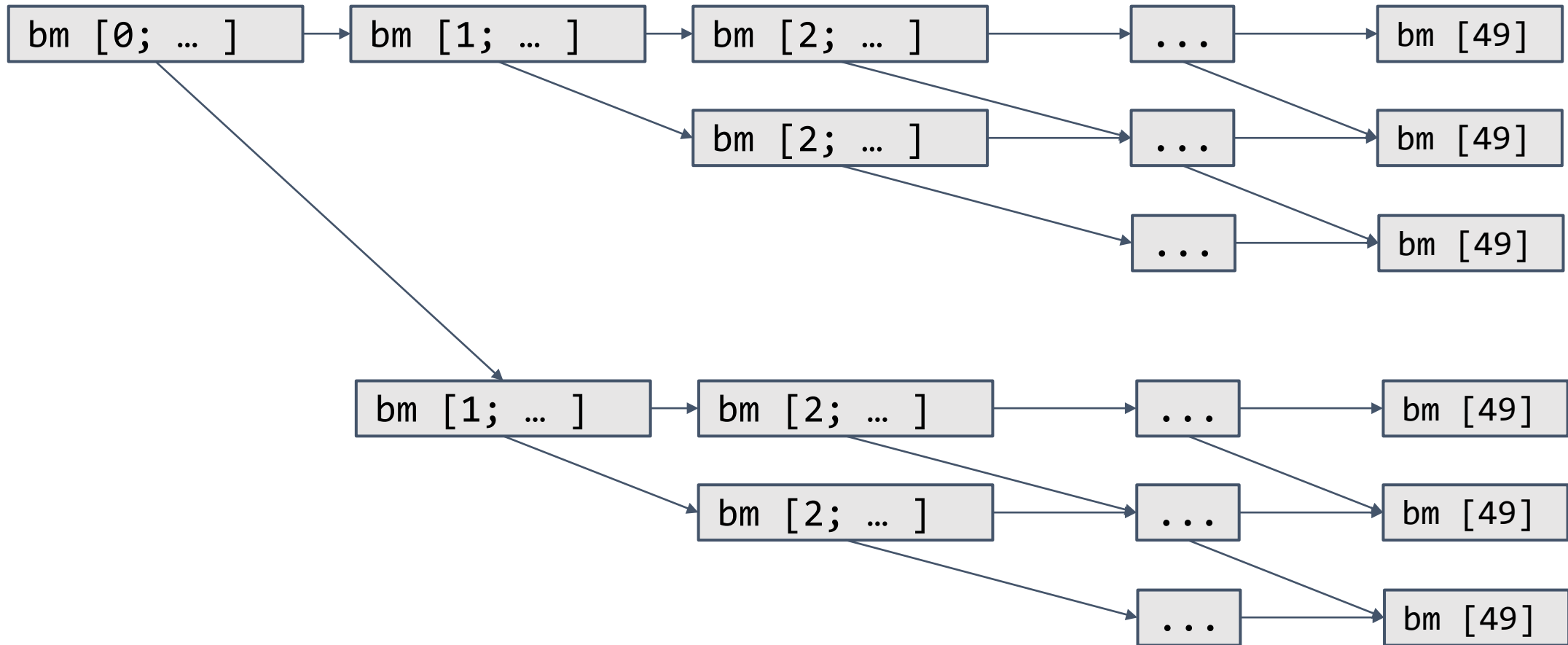
```
if List.hd xs > bad_max (List.tl xs)
then List.hd xs
else bad_max (List.tl xs)
```





# Rápido Vs. inutilizável

```
if List.hd xs > bad_max (List.tl xs)
then List.hd xs
else bad_max (List.tl xs)
```



$2^{50}$   
Calls



# Explosão da computação

- Suponhamos que a lógica de uma chamada `bad_max` , sem contar *com a recursão demora* 1 microsegundo
  - Não importa se este palpite está 1000x errado
- Então `bad_max [49; ... ; 0]` demora 50 microsegundos
- E `bad_max [0; ... ; 49]` demora  $2^{50}$  microsegundos
  - Isto é 35.7 anos
  - E `bad_max [0; ... ; 51]` demora 4x mais

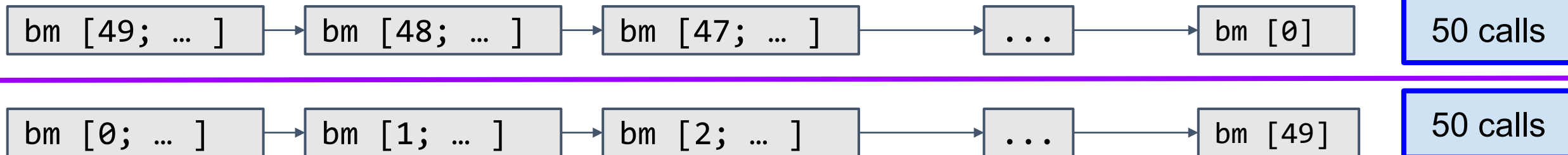
# Usar **let** local para evitar a recursão repetida

```
let rec better_max (xs : int list) =  
  if xs = [] then  
    0 (* bad style *)  
  else if List.tl xs = [] then  
    List.hd xs  
  else  
    let tl_max = better_max (List.tl xs) in  
    if List.hd xs > tl_max then  
      List.hd xs  
    else  
      tl_max
```

- Recursar a cauda da lista apenas uma vez
- Guardar o *resultado* no ambiente, vinculado a **tl\_max**

# Rápido Vs. inutilizável

```
let tl_max = better_max (List.tl xs) in
if List.hd xs > tl_max then
  List.hd xs
else
  tl_max
```



Opções

# Dados: o que fazer para funções parciais?

- Muitas computações não estão definidas ou “comportam-se mal” para algumas entradas
  - `List.hd []`, `List.tl []`, elemento máximo de `[]`, `1 / 0`, etc.
- OCaml providencia exceções para esses casos...
  - Veremos como usar exceções mais tarde
  - Mas as exceções deixam que nos esqueçamos da “coisa” excepcional
  - E isso pode levar a um fluxo de controlo surpreendente e indesejáveis
- OCaml providencia um tipo `option` para manipulação de funções parciais
  - Também para quando a estrutura de dados “pode ou não ter alguns dados”

# Opções

- Basicamente: “listas” de tamanho zero ou um, mas com um construtor de tipo *diferente*
- Um valor do tipo `t option` pode ser:
  - `None`
    - Valor válido do tipo `t option` para qualquer tipo `t`
      - Como `[]` para `t list`
  - `Some v`
    - Onde o valor `v` tem o tipo `t`
      - Como `[v]` para `t list`

**option** é outro  
construtor de tipo

Tal como os que vimos:

`t1 * t2`

`t1 -> t2`

`t list`

# Opções: None

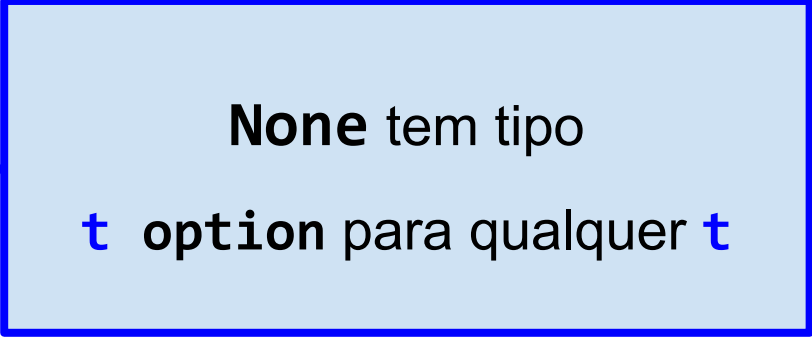
Sintaxe: `None`

Verificação do Tipo

- `None` tem tipo `'a option`

Avaliação

- `None` é um valor



**None** tem tipo  
`t option` para qualquer `t`



# Opções: Some

Sintaxe: Some *e*

- Onde *e* é uma expressão

Verificação do Tipo

- Se *e* tem tipo *t*
- Então Some *e* tem tipo *t* option
- Senão, reportar erro e falhar

Avaliação

- Se *e* avalia no valor *v*
- Então Some *e* avalia em Some *v*

# Opções: Teste

Sintaxe: `e = None`

- Onde `e` é uma expressão

Validação do Tipo

- Se `e` tem tipo `t option`
- Então `e = None` tem tipo `bool`
- Senão, reportar erro e falhar

Avaliação

- Se `e` avalia em `None`,
- Então `e = None` avalia no valor `true`
- Senão `e = None` avalia no valor `false`

Alternativamente, podemos  
usar as funções da biblioteca

**`Option.is_some` ou**

**`Option.is_none`**

# Opções: Obter elemento

Sintaxe: `Option.get e`

- Onde `e` é uma expressão

Verificação do Tipo

- Se `e` tem tipo `t` `option`
- Então `Option.get e` tem tipo `t`
- Senão, reportar erro e falhar

Avaliação

- Se `e` avalia no valor `Some v`
- Então `Option.get e` avalia no valor `v`
- Senão `e` avalia em `None` e assim `Option.get e` lança uma exceção

# Um melhor Máximo

```
let rec good_max (xs : int list) =  
  if xs = [] then  
    None  
  else  
    let tl_ans = good_max1 (List.tl xs) in  
    if Option.is_some tl_ans && Option.get tl_ans > List.hd xs then  
      tl_ans  
    else  
      Some (List.hd xs)
```

retorna um `int option`, e **Não** um `int`, pelo que pode finalmente dar uma resposta razoável mesmo para entradas “más” (i.e., `[]`)

Os utilizadores da função são forçados a considerar os casos de quando existem dados e quando não existem

# Aliasing e Mutação

# Conseguimos diferenciar? Um utilizador da função consegue?

```
let sort_pair (pr : int*int)=  
  if fst pr < snd pr then  
    pr  
  else  
    (snd pr, fst pr)
```

```
let sort_pair (pr : int*int)=  
  if fst pr < snd pr then  
    (fst pr, snd pr)  
  else  
    (snd pr, fst pr)
```

- Se o par estiver ordenado, retornamos (um alias para) o mesmo par, ou um novo par com os mesmos conteúdos ?
- Em OCaml, *não conseguimos dizer* de que forma o `sort_pair` é implementado (!!)
  - Esta “*expressividade negativa*” é muito importante!
  - Quando os dados são imutáveis, o aliasing não importa

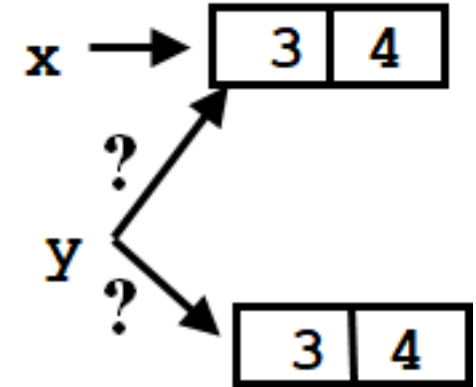
# Porque é que a falta de mutação importa ?

```
let x = (3,4)
```

```
let y = sort_pair x
```

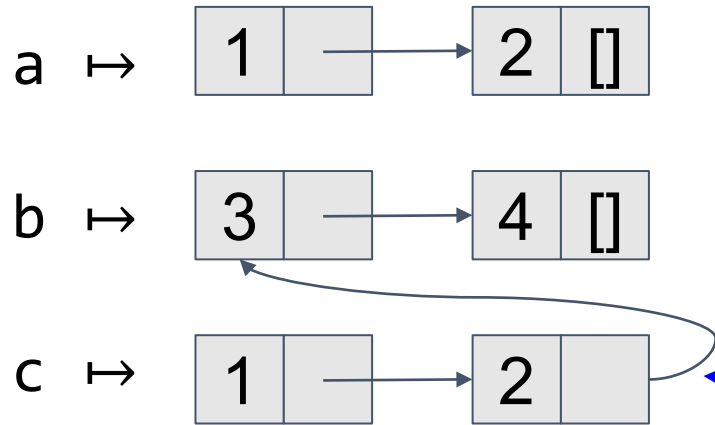
*De alguma forma mutamos fst x para 5*

```
let z = fst y
```



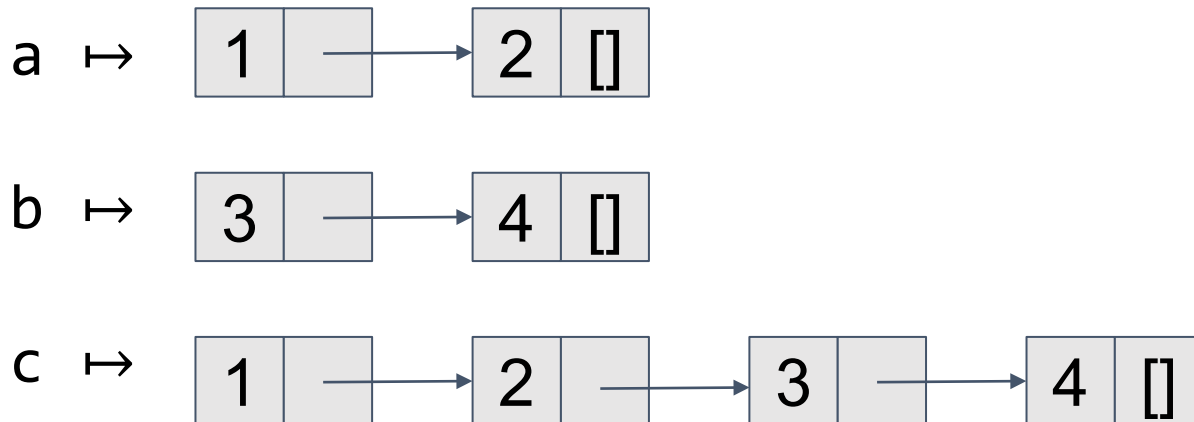
- Que valor temos em `z`?
  - Dependeria da forma como implementássemos `sort_pair`
- Teríamos de decidir cuidadosamente e documentar `sort_pair`
  - Sem mutação, podemos implementar “de qualquer maneira”
    - Nenhum código consegue distinguir entre aliasing e cópias idênticas
    - Não há necessidade de pensar em aliasing: concentrar-nos-emos noutras coisas
    - Podemos utilizar o aliasing, que poupa espaço, sem perigo

# Exemplo: Aliasing



```
let rec append ((xs:'a list), (ys:'a list)) =  
  if xs = []  
  then ys  
  else List.hd xs :: append (List.tl xs,  
ys)  
let a = [1; 2]  
let b = [3; 4]  
let c = append (a,b)
```

Ao utilizarmos a função nunca conseguimos dizer!  
(mas é na realidade o anterior)



Podemos de forma *segura* reusar e partilhar dados. Imutabilidade pode ser mais eficiente!



# A imutabilidade é ótima!

- Em OCaml, criamos aliasing a toda a hora sem pensar nisso, porque é *impossível* dizer onde existe um aliasing
  - Exemplo: `List.tl` é tempo constante; não copia nada
  - Não necessitamos de nos preocupar!
- Em linguagens com dados maioritariamente mutáveis (e.g., C), os programadores estão *obcecados* com o aliasing
  - Têm de ser (!) para que as atribuições subsequentes afetem as partes certas do programa
  - Muitas vezes é crucial fazer cópias nos sítios certos

# Igualdade em OCaml

- OCaml tem dois operadores de igualdade diferentes: `=` e `==`
  - `e1 <> e2` é o mesmo que **not** (`e1 = e2`)
  - `e1 != e2` é o mesmo que **not** (`e1 == e2`)
- Usem apenas `=` (ou `<>`)
  - Não são as sequências de caracteres a que estão habituados ⚠
  - No entanto, isso não é importante para ints e bools
- A diferença entre eles é interessante e relevante para o aliasing...

# Igualdade estrutural em OCaml

- = é definido recursivamente: os mesmos dados no mesmo formato
- $3=3$ ,  $(1,2)=(1,2)$ ,  $[3;4;5]=[3;4;5]$ ,  $\text{None}=\text{None}$ ,  $\text{Some } 7 = \text{Some } 7$ , etc.
- Duas sub-expressões devem ter o mesmo tipo
- É difícil ser igual de qualquer outra forma
- Nota:
  - Se  $x$  e  $y$  são *aliases*, então  $x=y$  deve avaliar em `true`
  - Se  $x$  and  $y$  não são *aliases*, então  $x=y$  deverá ser `true` ou `false`
  - Por isso, o aliasing não é importante

# Igualdade por **referência** em OCaml

- $e1 == e2$  é true se  $e1$  e  $e2$  avaliarem nos aliases (ou no mesmo int, bool, etc.)
  - Para estruturas de dados, é mais rápido que verificar a igualdade estrutural
- Mas esperem: os utilizadores *podem dizer* se duas coisas são aliases
  - Poderia ser complicado escrever código e implementar a linguagem
  - E não precisamos de nos preocupar com aliases para dados imutáveis
- Então OCaml chegou a um compromisso pragmático (hack ?)
  - A “definição oficial” de  $==$  não promete quase nada para dados imutáveis:
    - *tudo o que garante é que  $==$  implica  $=$*
  - Se assumirmos que significa mais, corremos o risco de sermos surpreendidos
  - Mantemos os “aliases” fora da definição da linguagem
    - uma das principais vantagens dos dados imutáveis!
- **Novamente: não usem  $==$  (nem  $!=$ ) durante a disciplina**

Créditos para Dan Grossman.