

Linguagens de Programação

Fabio Mascarenhas - 2017.2

<http://www.dcc.ufrj.br/~fabiom/lp>

Recursão mútua

- Não podemos expressar recursão mútua com letrec:

```
letrec par = fun (x)
    if x < 1 then true
    else impar(x-1) end
end
in
letrec impar = fun(x)
    if x < 1 then false
    else par(x-1) end
end
in par(4) end end
```

- Podemos usar rec para definir funções mutuamente recursivas, mas não é trivial: usamos rec para definir um par de funções, e dentro delas elas desconstroem o par

Pares

- Podemos definir um par como uma função que retorna o primeiro elemento se for passada true e o segundo se for passada false:

```
fun cons(a, b)
  fun (c)
    if c then a else b end
  end
end
```

```
fun fst(p)
  p(true)
end
```

```
fun snd(p)
  p(false)
end
```

- Seria até possível eliminar números e booleanos da linguagem, e fazer representar tudo com funções! Aí teríamos o *cálculo lambda*.

Recursão mútua com pares

- Agora podemos definir o par de funções mutuamente recursivas:

```
letrec pi = cons(fun (x)
                  if x < 1 then true
                  else snd(pi)(x-1) end
                end,
                fun(x)
                  if x < 1 then false
                  else fst(pi)(x-1) end
                end)
in let par = fst(pi), impar = snd(pi) in (par)(4) end end
```

- Toda essa volta pode parecer um exercício tolo quando já tínhamos funções recursivas no top-level, mas isso é uma prova de que o top-level não é parte essencial da linguagem, e poderia ser compilado para lets e recs!
- De fato, o cálculo lambda só tem três termos: variáveis, funções e aplicações

Tipos algébricos

- Podemos representar estruturas de dados mais complicadas usando funções
- A ideia é um elemento do tipo ser uma função que recebe uma função para cada construtor
- Por exemplo, para listas:

```
fun Cons(h, t)
  fun (v, c)
    c(h, t)
  end
end

fun Vazia()
  fun (v, c)
    v()
  end
end

fun tamanho(l)
  l(fun () 0 end,
    fun (h, t) 1 + tamanho(t) end)
end
```

rec na própria linguagem

- Se o cálculo lambda tem apenas variáveis e funções, como conseguimos fazer funções recursivas?
- Existe uma maneira de definir rec como uma função!
- Vamos voltar ao exemplo do fatorial:

```
let fat = rec fat = fun (x)  
  if x < 2 then 1  
  else x * fat(x-1) end  
end  
in fat(5) end
```

Duplicação

- Primeiro vamos extrair o núcleo de *rec*, o termo $T(x)$:

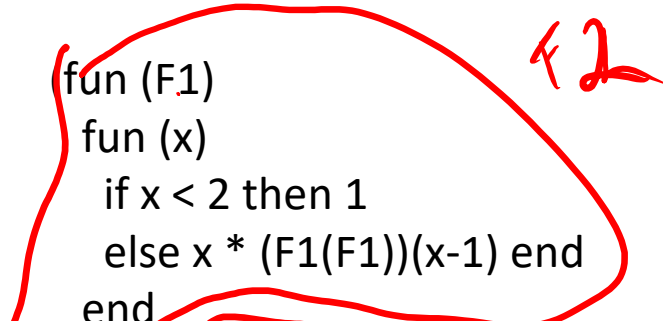
```
fun (fat)
  fun (x)
    if x < 2 then 1
    else x * fat(x-1) end
  end
end
```

- Mas isso ainda não é a função fatorial! Podemos chegar na fatorial usando um truque:

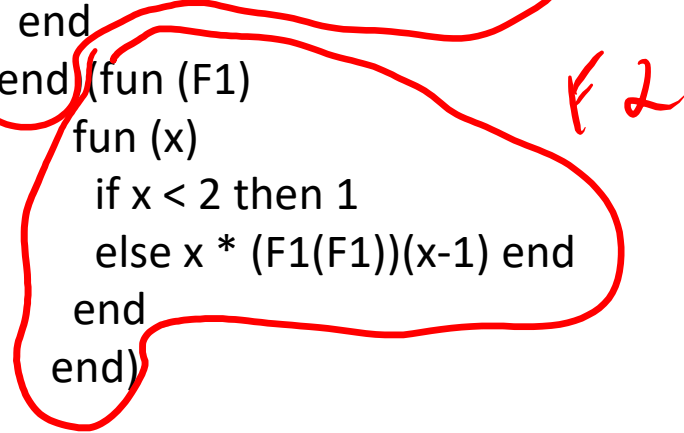
```
let F2 = fun (F1)
  fun (x)
    if x < 2 then 1
    else x * (F1(F1))(x-1) end
  end
end
in F2(F2) end
```

Função fatorial

- Por que F2(F2) é a função fatorial? Vamos expandir o let:



```
fun (F1)
  fun (x)
    if x < 2 then 1
    else x * (F1(F1))(x-1) end
  end
end
```



```
(fun (F1)
  fun (x)
    if x < 2 then 1
    else x * (F1(F1))(x-1) end
  end
end)
```


Função fatorial

- Agora fazemos a aplicação:

```
fun (x)
  if x < 2 then 1
  else x * (fun (F1)
    fun (x)
      if x < 2 then 1
      else x * (F1(F1))(x-1) end
    end
  end) (fun (F1)
    fun (x)
      if x < 2 then 1
      else x * (F1(F1))(x-1) end
    end
  end) (x-1) end
end
```

Handwritten red annotations illustrating the recursive application of the factorial function. A large red loop encircles the inner function definition, with "F2" written next to it. Another red loop encircles the inner function definition again, also with "F2" written next to it. To the right, "F2(F2)" is written in red, indicating the recursive call.

- Agora está se parecendo mais com uma função fatorial!

Função fatorial

- Dentro do corpo da função fatorial temos uma cópia de F2(F2), ou seja, fatorial:

```
fun (x)
  if x < 2 then 1
  else x * ((fun (F1)
    fun (x)
      if x < 2 then 1
      else x * (F1(F1))(x-1) end
    end
  end)(fun (F1)
    fun (x)
      if x < 2 then 1
      else x * (F1(F1))(x-1) end
    end
  end))(x-1) end
end
```

Extraindo *fix*

- Podemos extrair a transformação acima para uma função:

```
fun fix(f)
  let F2 = fun (F1)
    f(F1(F1))
  end
  in F2(F2) end
end
```

- E a função fatorial vira (note o uso de um parâmetro CBN!):

```
let fat = fix(fun (_fat)
  fun (x)
    if x < 2 then 1
    else x * _fat(x-1) end
  end
end) in fat(5) end
```

fix em ação

- Para entender como *fix* funciona, primeiro expandimos o let dentro dela:

```
fun fix(f)
  (fun (F1)
    f(F1(F1))
  end)(fun (F1)
    f(F1(F1))
  end)
end
```

The diagram illustrates the expansion of a `let` expression within the `fix` function. Red circles and arrows highlight the recursive call structure. The first circle encloses the `(fun (F1) f(F1(F1)) end)` block, with an arrow pointing to the `f(F1(F1))` call. The second circle encloses the `(fun (F1) f(F1(F1)) end)` block, with an arrow pointing to the `f(F1(F1))` call. A double red slash `//` is placed to the right of the first circle, indicating a recursive call.

- Agora podemos aplicar *fix* à função do slide anterior

Fatorial com *fix*

- Aplicando *fix* temos:

```
let fat = (fun (F1)
  (fun (_fat)
    fun (x)
      if x < 2 then 1
      else x * _fat(x-1) end
    end
  end)(F1(F1))
end)(fun (F1)
  (fun (_fat)
    fun (x)
      if x < 2 then 1
      else x * _fat(x-1) end
    end
  end)(F1(F1))
end)
in fat(5)
```

Fatorial com *fix*

- Fazendo a aplicação do lado direito do *let*:

```
let fat = fun (x)
```

```
  if x < 2 then 1
```

```
  else x *
```

```
    (fun (F1)
```

```
      (fun (_fat)
```

```
        fun (x)
```

```
          if x < 2 then 1
```

```
          else x * _fat(x-1) end
```

```
        end
```

```
      end)(F1(F1))
```

```
    end)(fun (F1)
```

```
      (fun (_fat)
```

```
        fun (x)
```

```
          if x < 2 then 1
```

```
          else x * _fat(x-1) end
```

```
        end
```

```
      end)(F1(F1))
```

```
    end))(x-1) end
```

```
  end
```

```
in fat(5)
```

fat

Por que um parâmetro CBN na função pra *fix*

- A função que passamos para *fix* precisa de um parâmetro CBN, ou *fix* entra em loop infinito!
- Mesmo se a linguagem não tem parâmetros call-by-name podemos evitar o loop, a custo de uma maior carga sintática

```
fun fix(f)  
  let F2 = fun (F1)  
    f(fun () F1(F1) end)  
  end  
  in F2(F2)  
end  
let fat = fix(fun (fat)  
  fun (x)  
    if x < 2 then 1  
    else x * (fat())(x-1) end  
  end  
end) in fat(5) end
```

(fun (x) x(x))
x(x)
end
(fun (x) x(x))
x(x)
end