Linguagens de Programação

Fabio Mascarenhas - 2015.2

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

Recursão

- Funções anônimas parecem não poder ser recursivas

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

variável livre

- Erro, `fat` está livre dentro da função anônima!

- Precisamos de um novo let:

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

Letrec em duas partes

- A definição de letrec fica mais simples se ele for açúcar sintático para um let e outro termo, rec:

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

- O que rec faz? Ele resolve uma equação *x = T(x)*, onde T(x) é um termo usando x (no caso acima, uma `fun`), e retorna essa solução

- Ou seja, rec acha o *ponto fixo* de T(x)!

rec x = E(x)

Avaliando rec

- Não se assuste em aparecer o ponto fixo, não vamos ver as implicações matemáticas disso

- “Resolver” o ponto fixo, ou seja, dar uma implementação para rec, é fácil!

  - Basta substituir *x* por *rec x = T(x)* em *T(x)*

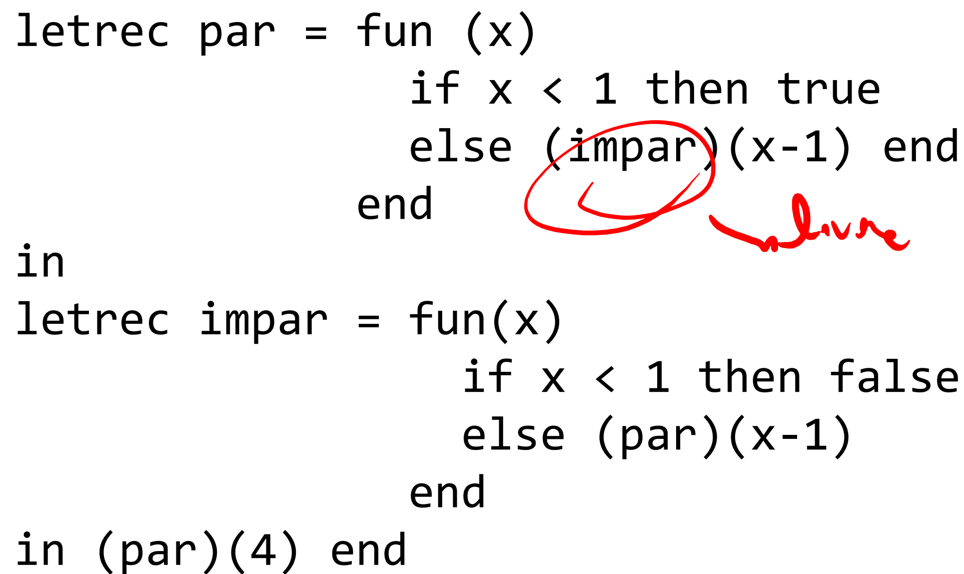
  - Ou seja, *desenrolamos* T(x)

  - Depois avaliamos T(x)

  - Isso quer dizer que *rec loop = loop* entra em loop infinito

Recursão mútua

- Não podemos expressar recursão mútua com letrec:



- 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