

# Linguagens de Programação

---

Fabio Mascarenhas - 2015.2

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

# Ambientes

---

- Substituição é uma semântica inspirada na forma como calculamos expressões algébricas, mas vimos que ela tem pontos sutis em sua relação com variáveis mutáveis
- Existe uma semântica alternativa que é mais próxima de como linguagens são implementadas na prática: a semântica de *ambientes*
- O tratamento das questões de nomes, captura e variáveis livres é mais simples com ambientes, e também podemos fazer experimentos com escopo

# Ambiente

---

- Um *ambiente* é um mapeamento de variáveis em valores (ou endereços)
- A avaliação de uma expressão agora se dá em termos de um *ambiente de avaliação*
- Em geral, subexpressões são avaliadas no mesmo ambiente de avaliação da expressão pai, exceto quando a expressão pai introduz nomes (na semântica de substituição, exceto quando a expressão pai efetua alguma substituição)
- Ao invés de fazermos substituições, deixamos as expressões como estão e criamos um novo ambiente com o efeito que a substituição teria

# Eval com ambientes

---

- O início de uma função eval para a semântica de ambientes de *MicroC*:

```
eval(env: Env, e: Exp) = e match {  
  case Soma(c1, e2) => for {  
    n1 ← eval(env, e1)  
    n2 ← eval(env, e2)  
  } yield n1 + n2  
  !  
}
```

- A maioria dos casos é uma recursão simples como a de Soma

# Eval com ambientes – variáveis e let

---

- Variáveis não são mais um erro automático; elas só estão livres se não há nenhuma entrada no ambiente para elas:

case Var(nome)  $\Rightarrow$  le(env(nome))

- Uma expressão let adiciona uma entrada no ambiente para avaliar seu corpo:

case let (n, e, c)  $\Rightarrow$  fun t  
ve  $\leftarrow$  eval(env, e)  
le  $\leftarrow$  push(ve)  
vc  $\leftarrow$  eval(env + (n  $\rightarrow$  le), c)  
} yield vc

# Eval com ambientes – funções de primeira ordem

---

- Cada argumento da função naturalmente é avaliado no mesmo ambiente da aplicação
- Mas e o corpo da função? Podemos criar um novo ambiente associando os parâmetros aos resultados dos argumentos, e avaliar o corpo nesse ambiente
- E se usarmos o ambiente da aplicação como base?

# Escopo estático vs escopo dinâmico

---

- Se quisermos ter o mesmo comportamento que tínhamos com a substituição, o ambiente base para o corpo das funções de primeira ordem tem que ser o *ambiente vazio*!
- Usar um ambiente vazio implementa a regra de *escopo estático*, ou *escopo léxico* – a associação entre *usos* e *definições* das variáveis é *textual*
- Usar o ambiente da chamada de função como base implementa a regra do *escopo dinâmico* – a associação entre usos e definições das variáveis depende da *pilha de chamadas* no momento do uso da variável

# Exemplo e aplicações

---

- Um exemplo simples de escopo dinâmico:

```
fun mul(x)
  x * y
end

let y = 2 in mul(5) end
```

- O escopo das variáveis locais (e parâmetros de função) em linguagens é quase sempre estático
- Mas formas de escopo dinâmico aparecem em outras partes

# Aplicações de escopo dinâmico

---

- *Variáveis especiais* ou *parâmetros* de Common Lisp: variáveis com escopo dinâmico como as que acabamos de ver; conceitualmente, é como usar dois ambientes, ou como usar substituição para a semântica das variáveis normais e um ambiente para as variáveis especiais
- Variáveis *locais à thread* do sistema operacional, pois quando implementamos a linguagem em uma máquina real o ambiente é mapeado na pilha de execução
- *Injeção de dependências*, onde módulos externos que o programa usa são resolvidos pelo seu contexto de execução
- Tratadores de exceção: o tratador corrente é uma variável com escopo dinâmico, com uma ligação como tratador no momento de sua definição

# Eval com ambientes em *fun*

---

- O problema de qual ambiente usar como base para aplica uma função de primeira classe é mais complicado que o de uma função de primeira ordem

```
fun curry_mul(x)
  fun (y)
    x * y
  end
end
```

```
let x = 2 in (curry_mul(5))(3) end
```

- O ambiente em que aplicamos a função anônima a 3 obviamente tem que incluir  $x \rightarrow 5$  se quisermos respeitar o escopo léxico
- Mas esse é o ambiente no momento em que criamos (avaliamos) essa função anônima! No ponto da aplicação não temos mais acesso a ele

# Closures

---

- Para resolver isso uma FunV não pode ser apenas os parâmetros e o corpo da função, mas sim uma tripla com os parâmetros, o corpo e o *ambiente no qual ela foi avaliada*
- Essa tripla tem o nome de *closure* ou *fecho*

```
case class FunV(env: Env,  
               params: List[String],  
               corpo: Exp) extends Valor
```

- Usamos o ambiente armazenado no closure como base para a aplicação a função

# Conversão de closures

---

- Só com funções de primeira classe conseguimos expressar qualquer computação, inclusive funções [mutuamente] recursivas
- Mas com a ideia de closures também conseguimos simular funções de primeira classe em linguagens com funções de primeira ordem e estruturas de dados
- Isso é feito com uma transformação chamada *conversão de closures*, e usada em compiladores de linguagens que têm funções de primeira classe
- Uma versão mais simples também é usada na compilação de funções de primeira ordem aninhadas, para manter o escopo léxico nesse caso