

# Linguagens de Programação

---

Fabio Mascarenhas - 2013.1

<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 e parâmetros
- 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 podemos fazer experimentos com escopo

# Ambiente

---

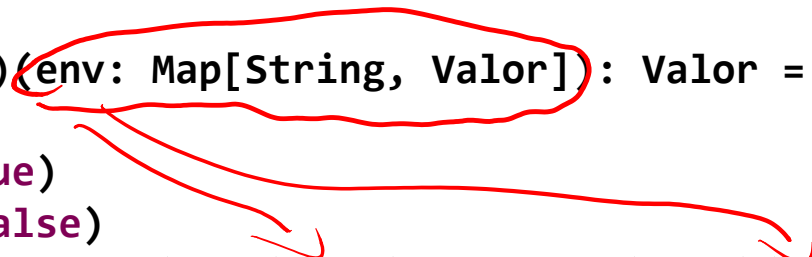
- Um *ambiente* é um mapeamento de variáveis em valores
- A avaliação de uma expressão agora se dá em termos de um *ambiente de avaliação*
- Em geral, subexpressões são avaliadas no mesmo ambiente de avaliação da expressão pai, exceto quando a expressão pai introduz nomes (na semântica de substituição, 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 *fun*:

```
def eval(funs: Set[Fun1])(env: Map[String, Valor]): Valor = this match {  
  case Num(v) => NumV(v)  
  case True() => Bool(true)  
  case False() => Bool(false)  
  case Soma(e1, e2) => (e1.eval(funs)(env), e2.eval(funs)(env)) match {  
    case (NumV(v1), NumV(v2)) => NumV(v1 + v2)  
    case _ => sys.error("soma precisa de dois números")  
  }  
  ...  
}
```



- 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:

```
def eval(funs: Set[Fun1])(env: Map[String, Valor]): Valor = this match {  
  ...  
  case Var(nome) => env.get(nome) match {  
    case Some(v) => v  
    case None => sys.error("variável livre: " + nome)  
  }  
  ...  
}
```

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

```
def eval(funs: Set[Fun1])(env: Map[String, Valor]): Valor = this match {  
  ...  
  case Let(nome, exp, corpo) =>  
    corpo.eval(funs)(env + (nome -> exp.eval(funs)(env)))  
  ...  
}
```

# Eval com ambientes – funções de primeira ordem

---

- Podemos passar o ambiente para a função apply, e cuidar dele lá
- Naturalmente cada argumento vai ser avaliado nesse mesmo ambiente
- Mas e o corpo da função? Podemos criar um novo ambiente associando os parâmetros aos resultados dos argumentos, e avaliar o corpo nesse ambiente
  - Usamos o ambiente da aplicação como base?
- Ainda temos um problema com o tratamento de parâmetros call-by-name; voltaremos a isso mais tarde

# 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 locais é quase sempre estático
- Mas formas de escopo dinâmico aparecem em outras partes



# Aplicações de escopo dinâmico

---

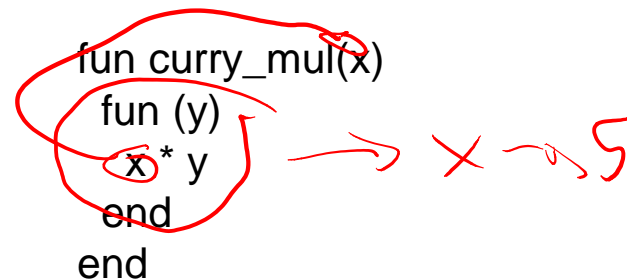
- *Variáveis especiais* ou *parâmetros* de Common Lisp: variáveis com escopo dinâmico como as que acabamos de ver; conceitualmente, é como usar dois ambientes, ou como usar substituição para a semântica das variáveis normais e um ambiente para as variáveis especiais
- *Variáveis locais à thread* do sistema 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 – funções de primeira classe


---

- 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: Map[String, Valor],  
               params: List[String],  
               corpo: Exp) extends Valor
```

*A ambiente no  
armazenado da  
função*

- 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 usada na compilação de funções de primeira ordem aninhadas, para manter o escopo léxico nesse caso

# Call by name com ambientes

---

- Na passagem para a semântica e ambientes perdemos a capacidade de chamar parâmetros call-by-name, pois o nosso ambiente é um mapa de nomes para valores
- Poderíamos fazer o ambiente ser um mapa de nomes para *expressões*, e transformar valores atômicos em suas expressões equivalentes, do modo que fizemos na substituição
- Mas vamos aproveitar essa reintrodução de CBN para usar outra abordagem (que também poderíamos ter usado lá na substituição)

# Criando e “forçando” thunks

---

- A ideia é criar um valor que encapsula uma expressão que foi passada como parâmetro call by value, junto com o ambiente dessa expressão (por quê o ambiente também?)
- Quando acessamos uma variável call-by-name desempacotamos esse valor e avaliamos a expressão
- Como quase tudo que já vimos, existe um jargão para esses valores, e sua avaliação: eles são *thunks*, e quando avaliamos eles dizemos que estamos *forçando* o thunk
- Já temos até a estrutura perfeita para representar nossos thunks: FunV

# Eval com ambientes - rec

---

- O ambiente no qual avaliamos o lado direito de um rec deve ter uma associação da variável do rec com ele mesmo
- Para isso precisamos usar uma definição recursiva de Scala para definir esse novo ambiente:

```
val renv: Map[String, Valor] = env + (nome -> e.eval(funs)(renv))  
e.eval(funs)(renv)
```

*env + (nome -> () => e.eval(funs)(renv))*

- Ops, mas essa definição recursiva não é bem formada!
- Uma maneira de resolver é mudar os ambientes para um Map[String, () => Valor], ou seja, mapeando um nome para um *thunk*, e fazendo Var forçar esse thunk para ter o valor

# Thunks recursivos

---

- Uma forma de implementar rec com ambientes é usar um *thunk recursivo*, um thunk que, quando forçado, vai “desenrolar” a recursão do mesmo modo que fazíamos com a substituição
- Para simplificar o interpretador vamos também criar um novo valor para os thunks CBN

```
case class Thunk(env: Map[String, Valor], c: Exp) extends Valor
case class RecV(env: Map[String, Valor], nome: String, c: Exp) extends Valor
```

- Agora podemos forçar todos os acessos a variáveis, usando a função force:

```
def force(funs: Set[Fun1]): Valor = this match {
  case Thunk(envc, corpo) =>
    corpo.eval(funs)(envc)
  case RecV(envc, nome, corpo) =>
    corpo.eval(funs)(envc + (nome -> RecV(envc, nome, corpo)))
  case v => v
}
```



# Efeitos colaterais: referências e atribuição

---

- A partir de agora vamos começar a sair do mundo funcional e explorar outros paradigmas de programação
- Vamos começar revendo o paradigma *imperativo*, onde o programa não é apenas uma expressão algébrica pura, mas executa *ações* que influenciam um *estado* externo ao programa
- Outro nome para programação imperativa é a programação com *efeitos colaterais*
- Primeiro vamos adicionar uma forma bem simples de efeito colateral a *fun*, referências e atribuição, e ver como isso muda radicalmente nosso interpretador