

Universidade da Beira Interior

# Desenho de Linguagens de Programação e de Compiladores

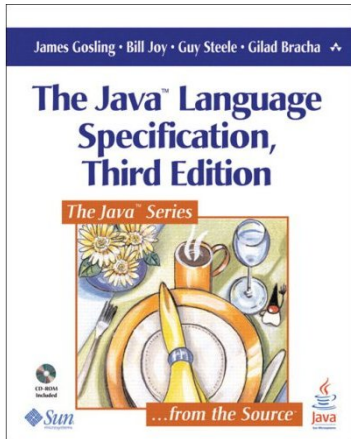
Simão Melo de Sousa

Aula 2 - Sintaxe abstracta, Semântica, Intrepretadores

como definir o significado dos programas escritos numa determinada linguagem de programação?

no caso geral, aceitamos uma descrição informal, em lingua natural (norma ISO, *reference manual*, etc.)

em toda a verdade, é pouco satisfatório, porque frequentemente impreciso ou até mesmo ambíguo



*The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.*

*It is recommended that code not rely crucially on this specification.*

a **semântica formal** caracteriza matematicamente os cálculos descritos num programa

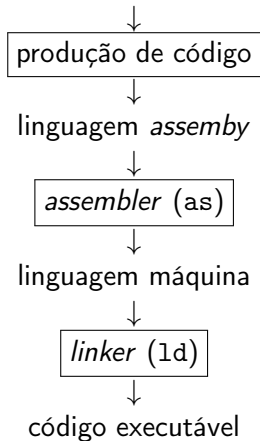
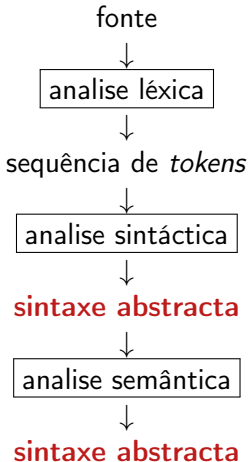
útil para o desenho e implementação de ferramentas programáticas (interpretadores, compiladores, etc.) : **Desenho de Linguagens de Programação e de compiladores!**

indispensável para poder raciocinar sobre programas : **Computação Fiável!**

mas então, o que é um programa ?

visto como objecto sintáctico (sequência de caracteres),  
é de manipulação demasiado pouco prático e difícil

preferimos utilizar uma representação abstracta : a **sintaxe abstracta**



os textos

`2*(x+1)`

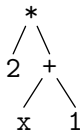
e

`(2 * ((x) + 1))`

e

`2 * (* multiplico por dois *) ( x + 1 )`

representam todos a mesmo **árvore de sintaxe abstracta**



define-se uma sintaxe abstracta por uma gramática, da forma seguinte

$e ::=$	$c$	<i>constante</i>
	$  \ x$	<i>variável</i>
	$  \ e + e$	<i>adição</i>
	$  \ e \times e$	<i>multiplicação</i>
	$  \ \dots$	

que se lê: « uma expressão  $e$  é

- ou uma constante,
- ou uma variável,
- ou uma soma de duas expressões,
- etc. »



a notação  $e_1 + e_2$  da sintaxe abstracta reutiliza por comodidade o simbolo  $+$  da sintaxe concreta. trata-se de uma representação.

mas poderíamos ter escolhido tanto  $Add(e_1, e_2)$ , como  $+(e_1, e_2)$ , etc.

em OCaml, representamos as árvores de sintaxe abstractas por tipos soma/indutivos

```
type binop = Add | Mul | ...

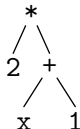
type expression =
  | Cte of int
  | Var of string
  | Bin of binop * expression * expression
  | ...
```

a expressão  $2 * (x + 1)$  é representada por

```
Bin (Mul, Cte 2, Bin (Add, Var "x", Cte 1))
```

não há constructores na sintaxe abstracta para os parênteses

na sintaxe **concreta**  $2 * (x + 1)$ ,  
os parênteses servem para reconhecer a árvore



em detrimento de outro

chamamos **açúcar sintáctico** a uma construção da sintaxe concreta que é uma facilidade (que pode ser expressa com outra construção da sintaxe concreta, de forma menos cómoda) da sintaxe concreta, ou então que não existe na sintaxe abstracta

esta é então traduzida a custa de outras construções da sintaxe abstracta (geralmente na fase da análise sintáctica).

exemples :

- em OCaml, a expressão  $[e_1; e_2; \dots; e_n]$  é açúcar sintáctico para
$$e_1 :: e_2 :: \dots :: e_n :: []$$
- em C, a expressão  $a[i]$  é açúcar para  $*(a+i)$

é sobre a sintaxe abstracta que vamos definir a semântica dos programas

existam várias abordagens

- semântica axiomática
- semântica denotacional
- semântica orientada pela tradução
- semântica operacional

igualmente designada de **lógica de Hoare**

(*An axiomatic basis for computer programming*, 1969)

caracteriza os programas com o recurso às propriedades satisfeitas pelas variáveis /memória manipuladas por estes ; introduz-se para esse efeito a noção de triplo de *Hoare*

$$\{P\} \ i \ \{Q\}$$

que significa « se a fórmula  $P$  é satisfeita antes da execução da instrução  $i$ , então a fórmula  $Q$  será verdade após o fim desta execução »

exemplo :

$$\{x \geq 0\} \ x := x + 1 \ \{x > 0\}$$

exemplo de regra da semântica :

$$\{P[x \leftarrow E]\} \ x := E \ \{P(x)\}$$

a **semântica denotacional** associa a cada expressão e a sua denotação  $\llbracket e \rrbracket$ , isto é um objecto matemático que representa o calculo associado à e

exemplo : expressões aritméticas com uma só variável  $x$

$$e ::= x \mid n \mid e + e \mid e * e \mid \dots$$

a denotação por definir pode ser a função que associa a cada valor  $x$  o valor da expressão

$$\begin{aligned}\llbracket x \rrbracket &= x \mapsto x \\ \llbracket n \rrbracket &= x \mapsto n \\ \llbracket e_1 + e_2 \rrbracket &= x \mapsto \llbracket e_1 \rrbracket(x) + \llbracket e_2 \rrbracket(x) \\ \llbracket e_1 * e_2 \rrbracket &= x \mapsto \llbracket e_1 \rrbracket(x) \times \llbracket e_2 \rrbracket(x)\end{aligned}$$

Qual é a denotação de  $\llbracket 2 * x + (x + 1) \rrbracket$ ?

(também conhecida por semântica denotacional *à la* Strachey)

podemos definir a semântica duma linguagem traduzindo-a para uma linguagem cuja semântica está já estabelecida



a **semântica operacional** descreve o encadeamento dos cálculos elementares que a avaliação da expressão realiza até chegar ao resultado (o seu valor)

opera directamente sobre os objectos sintácticos do programa (a sintaxe abstracta)

duas formas de semântica operacional

- « semântica natural » (*big-steps semantics*)

$$e \xrightarrow{v} v$$

- « semântica por reduções » (*small-steps semantics*)

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow v$$

ilustremos a semântica operacional sobre a linguagem **mini-ML**

$e ::=$	$x$	identificador
	$c$	constante ( $1, 2, \dots, true, \dots$ )
	$op$	primitiva ( $+, \times, fst, \dots$ )
	$\text{fun } x \rightarrow e$	função
	$e \ e$	aplicação
	$(e, e)$	par
	$\text{let } x = e \text{ in } e$	ligações locais

```
let compose = fun f → fun g → fun x → f (g x) in  
let plus = fun x → fun y → + (x, y) in  
compose (plus 2) (plus 4) 36
```

```
let distr_pair = fun f → fun p → (f (fst p), f (snd p)) in  
let p = distr_pair (fun x → x) (40, 2) in  
+ (fst p, snd p)
```

a condicional pode ser definida como

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \stackrel{\text{def}}{=} \text{opif } (e_1, ((\text{fun } \_ \rightarrow e_2), (\text{fun } \_ \rightarrow e_3)))$$

onde *opif* é uma primitiva

os ramos ficam **gelados** a custa de funções

de forma semelhante, a recursividade pode ser definida como

$$\text{rec } f \ x = e \stackrel{\text{def}}{=} \text{opfix } (\text{fun } f \rightarrow \text{fun } x \rightarrow e)$$

onde *opfix* é um operador de ponto fixo, satisfazendo

$$\text{opfix } f = f \ (\text{opfix } f)$$

exemplo

$$\text{opfix } (\text{fun } \text{fact} \rightarrow \text{fun } n \rightarrow \text{if } n = 0 \text{ then } 1 \text{ else } \times (n, \text{fact } (-(n, 1))))$$

procuramos definir uma relação entre uma expressão  $e$  e um **valor**  $v$

$$e \xrightarrow{v} v$$

os valores são definidos desta forma

$v ::=$	$c$	constante
	$  \quad op$	primitiva não aplicada
	$  \quad \text{fun } x \rightarrow e$	função
	$  \quad (v, v)$	par

para definir  $e \xrightarrow{v} v$ , precisamos das noções como **regras de inferência** e de **substituição**

uma relação pode ser definida como a **menor relação** que satisfaça um conjunto de axioma da forma

$$\overline{P}$$

e de um conjunto de implicações da forma

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

exemplo : podemos definir a relação  $\text{Even}(n)$  pelas duas regras

$$\overline{\text{Even}(0)} \quad \text{e} \quad \frac{\text{Even}(n)}{\text{Even}(n+2)}$$

que devem ser lidas como

em primeiro lugar  $\text{Even}(0)$

em segundo lugar  $\forall n. \text{Even}(n) \Rightarrow \text{Even}(n+2)$

a menor relação satisfazendo estas duas propriedades coincide com a propriedade «  $n$  é um inteiro par » :

- os inteiros pares estão evidentemente contemplados pela relação, basta usar um argumento indutivo para se convencer.
- se há pelo menos um ímpar na relação, poderíamos obter uma relação ainda menor retirando dela o menor dos ímpares presentes.



uma **derivação** é uma árvore cujos nodos correspondem às regras e as folhas aos axiomas ; exemplo

$$\frac{\frac{\text{Even}(0)}{\text{Even}(2)}}{\text{Even}(4)}$$

o conjunto das derivações possíveis caracteriza exactamente a menor relação que satisfaz as regras de inferência

## Definição (variáveis livres)

O conjunto das **variáveis livres** de uma expressão  $e$ , denotada por  $fv(e)$ , é definido recursivamente sobre  $e$  da forma seguinte :

$$\begin{aligned}
 fv(x) &= \{x\} \\
 fv(c) &= \emptyset \\
 fv(op) &= \emptyset \\
 fv(\text{fun } x \rightarrow e) &= fv(e) \setminus \{x\} \\
 fv(e_1 \ e_2) &= fv(e_1) \cup fv(e_2) \\
 fv((e_1, e_2)) &= fv(e_1) \cup fv(e_2) \\
 fv(\text{let } x = e_1 \text{ in } e_2) &= fv(e_1) \cup (fv(e_2) \setminus \{x\})
 \end{aligned}$$

Uma expressão sem variáveis livres é dita **fechada**.

$$fv(\text{let } x = +(20, 1) \text{ in } (\text{fun } y \rightarrow +(y, y)) \ x) = \emptyset$$

$$fv(\text{let } x = \textcolor{red}{z} \text{ in } (\text{fun } y \rightarrow (x \ y) \ \textcolor{red}{t})) = \{z, t\}$$

## Definição (substituição)

Se  $e$  é uma expressão,  $x$  uma variável livre de  $e$  e  $v$  um valor, nota-se por  $e[x \leftarrow v]$  a **substituição** de  $x$  por  $v$  em  $e$ , definida por

$$\begin{aligned}
 x[x \leftarrow v] &= v \\
 y[x \leftarrow v] &= y \quad \text{se } y \neq x \\
 c[x \leftarrow v] &= c \\
 op[x \leftarrow v] &= op \\
 (\text{fun } x \rightarrow e)[x \leftarrow v] &= \text{fun } x \rightarrow e \\
 (\text{fun } y \rightarrow e)[x \leftarrow v] &= \text{fun } y \rightarrow e[x \leftarrow v] \quad \text{se } y \neq x \\
 (e_1 \ e_2)[x \leftarrow v] &= (e_1[x \leftarrow v] \ e_2[x \leftarrow v]) \\
 (e_1, e_2)[x \leftarrow v] &= (e_1[x \leftarrow v], e_2[x \leftarrow v]) \\
 (\text{let } x = e_1 \text{ in } e_2)[x \leftarrow v] &= \text{let } x = e_1[x \leftarrow v] \text{ in } e_2 \\
 (\text{let } y = e_1 \text{ in } e_2)[x \leftarrow v] &= \text{let } y = e_1[x \leftarrow v] \text{ in } e_2[x \leftarrow v] \\
 &\quad \text{se } y \neq x
 \end{aligned}$$

$$((\text{fun } x \rightarrow +(x, x)) \text{ x}) [x \leftarrow 21] = (\text{fun } x \rightarrow +(x, x)) 21$$

$$+(\text{x}, \text{let } x = 17 \text{ in } x) [x \leftarrow 3] = +(3, \text{let } x = 17 \text{ in } x)$$

$$(\text{fun } y \rightarrow y \ y) [y \leftarrow 17] = \text{fun } y \rightarrow y \ y$$

$$\overline{c \xrightarrow{v} c} \quad \overline{op \xrightarrow{v} op} \quad \overline{(\text{fun } x \rightarrow e) \xrightarrow{v} (\text{fun } x \rightarrow e)}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} v_2}{(e_1, e_2) \xrightarrow{v} (v_1, v_2)} \quad \frac{e_1 \xrightarrow{v} v_1 \quad e_2[x \leftarrow v_1] \xrightarrow{v} v}{\text{let } x = e_1 \text{ in } e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} (\text{fun } x \rightarrow e) \quad e_2 \xrightarrow{v} v_2 \quad e[x \leftarrow v_2] \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

nota : fizemos a escolha estratégica da **passagem por valor**  
i.e. o argumento é avaliado antes da chamada/passagem

devemos juntar as regras para as primitivas ; por exemplo

$$\frac{e_1 \xrightarrow{v} + \quad e_2 \xrightarrow{v} (n_1, n_2) \quad n = n_1 + n_2}{e_1 \ e_2 \xrightarrow{v} n}$$

$$\frac{e_1 \xrightarrow{v} fst \quad e_2 \xrightarrow{v} (v_1, v_2)}{e_1 \ e_2 \xrightarrow{v} v_1}$$

$$\frac{e_1 \xrightarrow{v} opif \quad e_2 \xrightarrow{v} (true, ((fun \_ \rightarrow e_3), (fun \_ \rightarrow e_4))) \quad e_3 \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} opfix \quad e_2 \xrightarrow{v} (fun \ f \rightarrow e) \quad e[f \leftarrow opfix \ (fun \ f \rightarrow e)] \xrightarrow{v} v}{e_1 \ e_2 \xrightarrow{v} v}$$

$$\begin{array}{c}
 \begin{array}{c}
 + \xrightarrow{v} + \quad \frac{20 \xrightarrow{v} 20 \quad 1 \xrightarrow{v} 1}{(20, 1) \xrightarrow{v} (20, 1)} \\
 \hline
 +(20, 1) \xrightarrow{v} 21
 \end{array}
 \quad
 \frac{\text{fun } \dots \xrightarrow{v} \quad 21 \xrightarrow{v} 21 \quad \frac{\vdots}{+(21, 21) \xrightarrow{v} 42}}{(\text{fun } y \rightarrow +(y, y)) \ 21 \xrightarrow{v} 42} \\
 \hline
 \text{let } x = +(20, 1) \text{ in } (\text{fun } y \rightarrow +(y, y)) \ x \xrightarrow{v} 42
 \end{array}$$



dar a derivação de

$(\text{opfix } F) 2$

com  $F$  definido como

```
fun fact → fun n → if n = 0 then 1 else × (n, fact (-(n, 1)))
```

existem expressões  $e$  para as quais não há valor  $v$  tal que  $e \xrightarrow{v}$

exemplo :  $e = 1\ 2$

exemplo :  $e = (\text{fun } x \rightarrow x\ x) (\text{fun } x \rightarrow x\ x)$

para estabelecer uma propriedade de uma relação definida por um conjunto de regras de inferência, podemos raciocinar por **recorrência** sobre a derivação

isto significa por **indução estrutural** i.e. podemos aplicar a hipótese de indução (*HI*) a todas as sub-derivações  
(de forma equivalente, podemos raciocinar por recorrência sobre a altura da derivação)

em prática, raciocinamos por recorrência sobre a derivação e *por caso* sobre a última regra utilizada

## Proposição (a avaliação produz valores fechados)

*Se  $e \xrightarrow{v} v$  então  $v$  é um valor.*

*Mais, se  $e$  é fechado, então  $v$  também.*

demonstração por indução sobre a derivação  $e \xrightarrow{v} v$   
caso da aplicação

$$\begin{array}{ccc}
 (D_1) & (D_2) & (D_3) \\
 \vdots & \vdots & \vdots \\
 e_1 \xrightarrow{v} (\text{fun } x \rightarrow e) & e_2 \xrightarrow{v} v_2 & e[x \leftarrow v_2] \xrightarrow{v} v \\
 \hline
 e_1 \ e_2 \xrightarrow{v} v
 \end{array}$$

por HI  $v$  é um valor

se  $e$  é fechada então  $e_1$  e  $e_2$  também, e por HI  $\text{fun } x \rightarrow e$  e  $v_2$  são fechadas, logo  $e[x \leftarrow v_2]$  é fechada, e por HI  $v$  também

(exercício : tratar dos outros casos)

## Proposição (determinismo da avaliação)

Se  $e \xrightarrow{v} v$  e  $e \xrightarrow{v'} v'$  então  $v = v'$ .

por indução sobre as derivações de  $e \xrightarrow{v} v$  e de  $e \xrightarrow{v'} v'$   
 caso de um par  $e = (e_1, e_2)$

$$\begin{array}{ccc}
 \begin{array}{c} (D_1) \\ \vdots \\ e_1 \xrightarrow{v} v_1 \end{array} & \begin{array}{c} (D_2) \\ \vdots \\ e_2 \xrightarrow{v} v_2 \end{array} & \begin{array}{c} (D'_1) \\ \vdots \\ e_1 \xrightarrow{v} v'_1 \end{array} \quad \begin{array}{c} (D'_2) \\ \vdots \\ e_2 \xrightarrow{v} v'_2 \end{array} \\
 \hline
 (e_1, e_2) \xrightarrow{v} (v_1, v_2) & & (e_1, e_2) \xrightarrow{v} (v'_1, v'_2)
 \end{array}$$

por HI temos  $v_1 = v'_1$  e  $v_2 = v'_2$ , logo  $v = (v_1, v_2) = (v'_1, v'_2) = v'$

(exercício : tratar dos outros casos)

nota : a relação de avaliação pode não ser necessariamente uma relação determinista

exemplo : juntamos uma primitiva *random* e a regra

$$\frac{e_1 \xrightarrow{v} \text{random} \quad e_2 \xrightarrow{v} n_1 \quad 0 \leq n < n_1}{e_1 \ e_2 \xrightarrow{v} n}$$

temos então  $\text{random } 2 \xrightarrow{v} 0$  como igualmente  $\text{random } 2 \xrightarrow{v} 1$

podemos programar um **interpretador** seguindo as regras da semântica natural

damo-nos um tipo para a sintaxe abstracta das expressões

```
type expression = ...
```

e definimos uma função

```
val value : expression -> expression
```

que corresponde a relação  $\xrightarrow{v}$  (já que se trata de uma função)

```
type expression =  
  | Var    of string  
  | Const  of int  
  | Op     of string  
  | Fun    of string * expression  
  | App    of expression * expression  
  | Pair   of expression * expression  
  | Let    of string * expression * expression
```



precisamos de implementar a substituição  $e[x \leftarrow v]$

```
val subst : expression -> string -> expression -> expression
```

supomos  $v$  fechado (logo, não há risco de captura de variável)

```
let rec subst e x v = match e with
| Var y ->
    if y = x then v else e
| Const _ | Op _ ->
    e
| Fun (y, e1) ->
    if y = x then e else Fun (y, subst e1 x v)
| App (e1, e2) ->
    App (subst e1 x v, subst e2 x v)
| Pair (e1, e2) ->
    Pair (subst e1 x v, subst e2 x v)
| Let (y, e1, e2) ->
    Let (y, subst e1 x v, if y = x then e2 else subst e2 x v)
```

a semântica natural é realizada pela função

```
val value : expression -> expression
```

```
let rec value = function
  | Const _ | Op _ | Fun _ as v ->
    v
  | Pair (e1, e2) ->
    Pair (value e1, value e2)
  | Let(x, e1, e2) ->
    value (subst e2 x (value e1))
  ...
```

```
...
| App (e1, e2) ->
  begin match value e1 with
  | Fun (x, e) ->
    value (subst e x (value e2))
  | Op "+" ->
    let (Pair (Const n1, Const n2)) = value e2 in
    Const (n1 + n2)
  | Op "fst" ->
    let (Pair(v1, v2)) = value e2 in v1
  | Op "snd" ->
    let (Pair(v1, v2)) = value e2 in v2
  end
```

```
# value
(Let
  ("x",
    App (Op "+", Pair (Const 1, Const 20)),
    App (Fun ("y", App (Op "+", Pair (Var "y", Var "y"))),
      Var "x"))));;
```

- : expression = Const 42

o filtro é propositadamente não-exaustivo

```
# value (Var "x");;
```

```
# value (App (Const 1, Const 2));;
```

```
Exception: Match_failure (" ", 87, 6).
```

(poderíamos preferir um tipo *option*, uma exceção definida para este propósito, etc.)

a avaliação pode não conseguir terminar

por exemplo sobre

$$(\text{fun } x \rightarrow x \ x) (\text{fun } x \rightarrow x \ x)$$

```
# let b = Fun ("x", App (Var "x", Var "x")) in  
  value (App (b, b));;
```

Interrupted.

acrescentar os operadores *opif* e *opfix* ao interpretador

podemos evitar a operação de substituição ?

ideia : interpretar a expressão  $e$  com recurso a um ambiente que dá o valor actual de cada variável (um dicionário)

```
val value : environment -> expression -> value
```

problema : o resultado de

$$\text{let } x = 1 \text{ in fun } y \rightarrow +(x, y)$$

é uma função que deve « memorizar » que  $x = 1$

resposta : é preciso usar o mecanismo designado de **fecho**



utilizamos o módulo Map para os ambientes

```
module Smap = Map.Make(String)
```

definimos um novo tipo para os valores

```
type value =  
  | Vconst of int  
  | Vop     of string  
  | Vpar    of value * value  
  | Vfun    of string * environment * expression  
  
and environment = value Smap.t
```

```
val value : environment -> expression -> value
```

```
let rec value env = function
  | Const n ->
      Vconst n
  | Op op ->
      Vop op
  | Pair (e1, e2) ->
      Vpar (value env e1, value env e2)
  | Var x ->
      Smap.find x env
  | Let (x, e1, e2) ->
      value (Smap.add x (value env e1) env) e2
  | ...
```

```
| Fun (x, e) ->
    Vfun (x, env, e)
| App (e1, e2) ->
    begin match value env e1 with
    | Vfun (x, clos, e) ->
        value (Smap.add x (value env e2) clos) e
    | Vop "+" ->
        let Vpar(Vconst n1,Vconst n2) = value env e2 in
        Vconst (n1 + n2)
    | Vop "fst" ->
        let Vpar (v1, _) = value env e2 in v1
    | Vop "snd" ->
        let Vpar (_, v2) = value env e2 in v2
    end
```

nota : é o processo usado quando se compila uma linguagem funcional (como ML, ver aula 5)

juntar os operadores *opif* e *opfix* neste interpretador

(nota: juntar o operador *opfix* não é imediato, falaremos mais sobre isso na aula sobre a compilação de linguagens funcionais)

a semântica natural não permite distinguir as expressões cuja avaliação «  
falha », como

1 2

das expressões cuja avaliação não termina, como

$(\text{fun } x \rightarrow x \ x) \ (\text{fun } x \rightarrow x \ x)$

a semântica operacional **small steps** responde a esta situação introduzindo uma noção de passo intermédio na avaliação de  $e_1 \rightarrow e_2$ , que se irá iterar

podemos assim distinguir 3 situações

1. a iteração leva a um valor

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow v$$

2. a iteração bloqueia sobre uma expressão  $e_n$  irreduzível que não é um valor

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow e_n$$

3. a iteração não termina

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots$$

começamos por definir uma relação  $\xrightarrow{\epsilon}$  que corresponde a uma redução « de cabeça », isto é realizada no topo da expressão

duas regras :

$$(\text{fun } x \rightarrow e) \ v \ \xrightarrow{\epsilon} \ e[x \leftarrow v]$$

$$\text{let } x = v \text{ in } e \ \xrightarrow{\epsilon} \ e[x \leftarrow v]$$

nota : aqui também fizemos a escolha da estratégia da **passagem por valor**

damo-nos igualmente as regras para as primitivas

$$+ (n_1, n_2) \xrightarrow{\epsilon} n \quad \text{com } n = n_1 + n_2$$

$$fst (v_1, v_2) \xrightarrow{\epsilon} v_1$$

$$snd (v_1, v_2) \xrightarrow{\epsilon} v_2$$

$$opfix (\text{fun } f \rightarrow e) \xrightarrow{\epsilon} e[f \leftarrow opfix (\text{fun } f \rightarrow e)]$$

$$opif (true, ((\text{fun } \_ \rightarrow e_1), (\text{fun } \_ \rightarrow e_2))) \xrightarrow{\epsilon} e_1$$

$$opif (false, ((\text{fun } \_ \rightarrow e_1), (\text{fun } \_ \rightarrow e_2))) \xrightarrow{\epsilon} e_2$$



para reduzir em profundidade, introduzimos a regra de inferência

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

onde  $E$  é um **contexto**, definido para a gramática seguinte :

$$\begin{array}{lcl}
 E & ::= & \square \\
 & | & E \ e \\
 & | & v \ E \\
 & | & \text{let } x = E \text{ in } e \\
 & | & (E, e) \\
 & | & (v, E)
 \end{array}$$

um contexto é então um « termo com buracos », onde  $\square$  representa o buraco  
por exemplo

$$E \stackrel{\text{def}}{=} \text{let } x = +(2, \square) \text{ in let } y = +(x, x) \text{ in } y$$

$E(e)$  designa o contexto  $E$  no qual  $\square$  foi substituído por  $e$   
exemplo :

$$E(+(10, 9)) = \text{let } x = +(2, +(10, 9)) \text{ in let } y = +(x, x) \text{ in } y$$

a regra

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \rightarrow E(e_2)}$$

permite assim avaliar uma sub-expressão

assim, temos a redução

$$\frac{+(1, 2) \xrightarrow{\epsilon} 3}{\text{let } x = +(1, 2) \text{ in } +(x, x) \rightarrow \text{let } x = 3 \text{ in } +(x, x)}$$

a custa do contexto  $E \stackrel{\text{def}}{=} \text{let } x = \square \text{ in } +(x, x)$

tais como definidos, os contextos implicam aqui uma avaliação de passagem por valor e « da esquerda para a direita »

assim,  $(+(1, 2), \square)$  não é um contexto de avaliação

poderíamos ter escolhidos uma avaliação da direita para a esquerda

designamos por  $\rightarrow^*$  o fecho reflexivo transitivo de  $\rightarrow$   
(i.e.  $e_1 \rightarrow^* e_2$  se e só se  $e_1$  se reduz em  $e_2$  em zero, uma ou mais etapas)

designamos de **forma normal** toda a expressão  $e$  tal que não existe expressão  $e'$  tal que  $e \rightarrow e'$

os valores são formas normais ; as formas normais que não são valores são as expressões errôneas (como 1 2)

vamos escrever as funções seguintes :

```
val head_reduction : expression -> expression
```

corresponde a  $\xrightarrow{\epsilon}$

```
val decompose : expression -> context * expression
```

descompõe uma expressão na forma  $E(e)$

```
val reduction1 : expression -> expression option
```

corresponde a  $\rightarrow$

```
val reduction : expression -> expression
```

corresponde a  $\xrightarrow{*}$

começamos por caracterizar os valores

```
let rec is_value = function
  | Const _ | Op _ | Fun _ ->
    true
  | Var _ | App _ | Let _ ->
    false
  | Pair (e1, e2) ->
    is_value e1 && is_value e2
```

escreve-se depois a *head reduction*

```
let head_reduction = function
| App (Fun(x, e1), e2) when is_value e2 ->
    subst e1 x e2
| Let (x, e1, e2) when is_value e1 ->
    subst e2 x e1
| App (Op "+", Pair (Const n1, Const n2)) ->
    Const(n1 + n2)
| App (Op "fst", Pair (e1, e2))
    when is_value e1 && is_value e2 ->
    e1
| App (Op "snd", Pair (e1, e2))
    when is_value e1 && is_value e2 ->
    e2
| _ ->
    raise NoReduction (* no possible reduction *)
```



um contexto  $E$  pode ser directamente representado por uma função  
(a função que a  $e$  associa  $E(e)$ )

```
type context = expression -> expression
```

```
let hole = fun e -> e
let app_left ctx e2 = fun e -> App (ctx e, e2)
let app_right v1 ctx = fun e -> App (v1, ctx e)
let pair_left ctx e2 = fun e -> Pair (ctx e, e2)
let pair_right v1 ctx = fun e -> Pair (v1, ctx e)
let let_left x ctx e2 = fun e -> Let (x, ctx e, e2)
```

```
let rec decompose e = match e with
  (* can be decomposed *)
  | Var _ | Const _ | Op _ | Fun _ ->
    raise NoReduction
  (* Head reduction case *)
  | App (Fun (x, e1), e2) when is_value e2 ->
    (hole, e)
  | Let (x, e1, e2) when is_value e1 ->
    (hole, e)
  | App (Op "+", Pair (Const n1, Const n2)) ->
    (hole, e)
  | App (Op ("fst" | "snd"), Pair (e1, e2))
    when is_value e1 && is_value e2 ->
    (hole, e)
  ...
```

```
...
(* deep reduction case *)
| App (e1, e2) ->
    if is_value e1 then begin
        let (ctx, rd) = decompose e2 in
            (app_right e1 ctx, rd)
    end else begin
        let (ctx, rd) = decompose e1 in
            (app_left ctx e2, rd)
        end
| Let (x, e1, e2) ->
    let (ctx, rd) = decompose e1 in
        (let_left x ctx e2, rd)
| Pair (e1, e2) ->
    ...
```

```
let reduction1 e =  
  try  
    let ctx, e' = decompose e in  
    Some (ctx (head_reduction e'))  
  with NoReduction ->  
    None
```

no final

```
let rec reduction e =  
  match reduction1 e with None -> e | Some e' -> reduction e'
```

um interpretador deste tipo não é eficaz

passa o seu tempo a recalcular o contexto e depois « esquece-o »

podemos fazer melhor, por exemplo, utilizando um *zipper* (estrutura de dados)

# equivalência das duas semânticas operacionais

vamos mostrar que as duas semânticas operacionais são equivalentes para as expressões cuja avaliação termina num valor, *i.e.*

$$e \xrightarrow{v} v \quad \text{se e só se} \quad e \xrightarrow{*} v$$

## Lema (passagem pelo contexto das reduções)

*Supomos que  $e \rightarrow e'$ . Então para toda a expressão  $e_2$  e para todo o valor  $v$*

1.  $e \ e_2 \rightarrow e' \ e_2$
2.  $v \ e \rightarrow v \ e'$
3.  $\text{let } x = e \text{ in } e_2 \rightarrow \text{let } x = e' \text{ in } e_2$

demonstração : de  $e \rightarrow e'$  sabemos que existe um contexto  $E$  tal que

$$e = E(r) \quad e' = E(r') \quad r \xrightarrow{\epsilon} r'$$

consideremos o contexto  $E_1 \stackrel{\text{def}}{=} E \ e_2$  ; então

$$\frac{r \xrightarrow{\epsilon} r'}{E_1(r) \rightarrow E_1(r')} \quad i.e. \quad \frac{r \xrightarrow{\epsilon} r'}{e \ e_2 \rightarrow e' \ e_2}$$

(idem para os casos 2 e 3)

## Proposição (*big-steps* implica *small-steps*)

Se  $e \xrightarrow{v} v$ , então  $e \xrightarrow{*} v$ .

demonstração : por indução sobre a derivação de  $e \xrightarrow{v} v$   
supomos que a última regra seja

$$\frac{e_1 \xrightarrow{v} (\text{fun } x \rightarrow e_3) \quad e_2 \xrightarrow{v} v_2 \quad e_3[x \leftarrow v_2] \xrightarrow{v} v}{e_1 \quad e_2 \xrightarrow{v} v}$$



# equivalência das duas semânticas operacionais

por HI temos

$$e_1 \rightarrow \dots \rightarrow v_1 = (\text{fun } x \rightarrow e_3)$$

$$e_2 \rightarrow \dots \rightarrow v_2$$

$$e_3[x \leftarrow v_2] \rightarrow \dots \rightarrow v$$

por passagem ao contexto (lema anterior) temos igualmente

$$e_1 \ e_2 \rightarrow \dots \rightarrow v_1 \ e_2$$

$$v_1 \ e_2 \rightarrow \dots \rightarrow v_1 \ v_2$$

$$e_3[x \leftarrow v_2] \rightarrow \dots \rightarrow v$$

integrando a redução

$$(\text{fun } x \rightarrow e_3) \ v_2 \xrightarrow{\epsilon} e_3[x \leftarrow v_2]$$

obtemos a redução completa

$$e_1 \ e_2 \rightarrow \dots \rightarrow v$$



para o sentido inverso (*small-steps* implica *big-steps*) precisamos de dois lemas intermédios

**Lema (os valores encontram-se avaliados)**

$v \xrightarrow{v} v$  para todos os valores  $v$ .

demonstração : trivial (segue imediatamente das definições)

## Lema (redução e avaliação)

Se  $e \rightarrow e'$  e  $e' \xrightarrow{v} v$ , então  $e \xrightarrow{v} v$ .

demonstração : comecemos pelas reduções de cabeça (*head-reduction*), i.e.  $e \xrightarrow{\epsilon} e'$

supomos por exemplo que  $e = (\text{fun } x \rightarrow e_1) \ v_2$  e  $e' = e_1[x \leftarrow v_2]$   
construimos a derivação

$$\frac{(\text{fun } x \rightarrow e_1) \xrightarrow{v} (\text{fun } x \rightarrow e_1) \quad v_2 \xrightarrow{v} v_2 \quad e_1[x \leftarrow v_2] \xrightarrow{v} v}{(\text{fun } x \rightarrow e_1) \ v_2 \xrightarrow{v} v}$$

utilizando o lema anterior ( $v_2 \xrightarrow{v} v_2$ ) e a hipótese  $e' \xrightarrow{v} v$

## equivalência das duas semânticas operacionais

mostremos agora que se  $e \xrightarrow{\epsilon} e'$  e  $E(e') \xrightarrow{v} v$  então  $E(e) \xrightarrow{v} v$

por indução estrutural sobre  $E$  ; fizemo-lo no caso de  $E = \square$

consideremos por exemplo  $E = E' e_2$

temos  $E(e') \xrightarrow{v} v$  isto é  $E'(e') e_2 \xrightarrow{v} v$ , que tem a forma

$$\frac{E'(e') \xrightarrow{v} (\text{fun } x \rightarrow e_3) \quad e_2 \xrightarrow{v} v_2 \quad e_3[x \leftarrow v_2] \xrightarrow{v} v}{E'(e') e_2 \xrightarrow{v} v}$$

por H1 temos  $E'(e) \xrightarrow{v} (\text{fun } x \rightarrow e_3)$ , logo

$$\frac{E'(e) \xrightarrow{v} (\text{fun } x \rightarrow e_3) \quad e_2 \xrightarrow{v} v_2 \quad e_3[x \leftarrow v_2] \xrightarrow{v} v}{E'(e) e_2 \xrightarrow{v} v}$$

isto é  $E(e) \xrightarrow{v} v$

□

## Proposição (*small-steps* implica *big-steps*)

Se  $e \xrightarrow{*} v$ , então  $e \xrightarrow{v} v$ .

demonstração : supomos  $e \rightarrow e_1 \rightarrow \dots \rightarrow e_n \rightarrow v$   
temos  $v \xrightarrow{v} v$  e logo, pelo lema anterior,  $e_n \xrightarrow{v} v$   
de igual forma  $e_{n-1} \xrightarrow{v} v$   
etc, até  $e \xrightarrow{v} v$

podemos definir uma semântica operacional *small-steps* ou *big-steps*, para uma linguagem imperativa

associa-se tipicamente um **estado**  $S$  à expressão avaliada/ reduzida

$$S, e \xrightarrow{v} S', v \quad \text{ou ainda} \quad S, e \rightarrow S', e'$$

exemplo de regra :

$$\frac{S, e \xrightarrow{v} S', v}{S, x := e \xrightarrow{v} S' \oplus \{x \mapsto v\}, \text{void}}$$

o estado  $S$  pode ser descomposto em vários elementos, para modelar uma pilha (variáveis locais), uma *heap*, e mais ainda...

- aula prática
  - escrever um o interpretador para mini-Python
- próxima aula teórica
  - tipagem

```
> ./mini-python tests/good/pascal.py
*
**
***
****
*****
*****
*****
*000000*
**00000**
***0000***
****000****
*****00*****
*****0*****
*****0*****
*000000*000000*
**00000**00000**
***0000***0000***
****000****000****
*****00*****00*****
*****0*****0*****
*****0*****0*****
```

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre ([link1](#), [link2](#))

adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)

