École Normale Supérieure

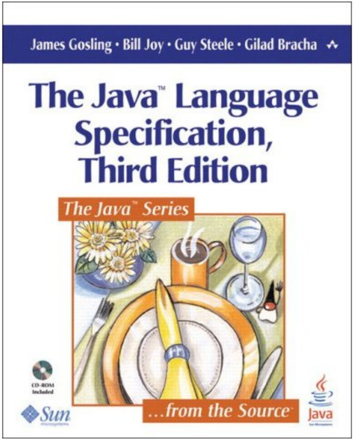
Langages de programmation et compilation

Jean-Christophe Filliâtre

syntaxe abstraite, sémantique, interprètes comment définir la signification des programmes écrits dans un langage ?

la plupart du temps, on se contente d'une description informelle, en langue naturelle (norme ISO, standard, ouvrage de référence, etc.)

s'avère peu satisfaisant, car souvent imprécis, voire ambigu



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.

la **sémantique formelle** caractérise mathématiquement les calculs décrits par un programme

utile pour la réalisation d'outils (interprètes, compilateurs, etc.)

indispensable pour raisonner sur les programmes mais qu'est-ce qu'un programme ?

en tant qu'objet syntaxique (suite de caractères),
il est trop difficile à manipuler

on préfère utiliser la **syntaxe abstraite**

syntaxe abstraite

source

↓

analyse lexicale

↓

suite de lexèmes (*tokens*)

↓

analyse syntaxique

↓

syntaxe abstraite

↓

analyse sémantique

↓

syntaxe abstraite

↓

production de code

↓

langage assembleur

↓

assembleur (`as`)

↓

langage machine

↓

éditeur de liens (`ld`)

↓

code exécutable

les textes

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

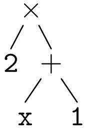
et

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

et

```
2 * (* je multiplie par deux *) ( x + 1 )
```

représentent tous le même **arbre de syntaxe abstraite**



on définit une syntaxe abstraite par une grammaire, de la manière suivante

$$
\begin{array}{rcll}
e & ::= & c & \textit{constante} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \dots &
\end{array}
$$

se lit « une expression e est

- soit une constante,
- soit une variable,
- soit l'addition de deux expressions,
- etc. »

la notation $e_1 + e_2$ de la syntaxe abstraite emprunte le symbole de la syntaxe concrète

mais on aurait pu tout aussi bien choisir $add(e_1, e_2)$, $+(e_1, e_2)$, etc.

en OCaml, on réalise la syntaxe abstraite par des types construits

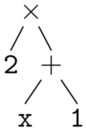
```
type binop = Add | Mul | ...

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

l'expression `2 * (x + 1)` est représentée par

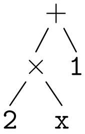
```
  Bin (Mul, Cte 2, Bin (Add, Var "x", Cte 1))
```

il n'y a pas de constructeur dans la syntaxe abstraite pour les parenthèses

dans la syntaxe **concrète** `2 * (x + 1)`,
les parenthèses servent à reconnaître cet arbre



plutôt que



(on expliquera comment dans un autre cours) on appelle **sucre syntaxique** une construction de la syntaxe concrète qui n'existe pas dans la syntaxe abstraite

elle est donc traduite à l'aide d'autres constructions de la syntaxe abstraite (généralement à l'analyse syntaxique)

exemples :

- en OCaml, l'expression $\texttt{[}e_1\texttt{;}\ e_2\texttt{;}\ \dots\texttt{;}\ e_n\texttt{]}$ est du sucre pour

$$e_1\ \texttt{::}\ e_2\ \texttt{::}\ \dots\ \texttt{::}\ e_n\ \texttt{::}\ \texttt{[]}$$

- en C, l'expression `a[i]` est du sucre pour `*(a+i)` c'est sur la syntaxe abstraite que l'on va définir la sémantique

il existe de nombreuses approches

- sémantique axiomatique
- sémantique dénotationnelle
- sémantique par traduction
- sémantique opérationnelle

encore appelée **logique de Floyd–Hoare**
(Robert Floyd, *Assigning meanings to programs*, 1967
Tony Hoare, *An axiomatic basis for computer programming*, 1969)

caractérise les programmes par l'intermédiaire des propriétés satisfaites par les variables ; on introduit le triplet

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

signifiant « si la formule P est vraie avant l'exécution de l'instruction i, alors la formule Q sera vraie après »

exemple :

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

exemple de règle :

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

la **sémantique dénotationnelle** associe à chaque expression e sa dénotation $[\![e]\!]$, qui est un objet mathématique représentant le calcul désigné par e

exemple : expressions arithmétiques avec une seule variable `x`

$$e ::= \mathtt{x} \mid n \mid e + e \mid e * e \mid \dots$$

la dénotation peut être une fonction qui associe à la valeur de `x` la valeur de l'expression

$$\begin{aligned}
[\![\mathtt{x}]\!] &\stackrel{\text{def}}{=} x \mapsto x \\
[\![n]\!] &\stackrel{\text{def}}{=} x \mapsto n \\
[\![e_1 + e_2]\!] &\stackrel{\text{def}}{=} x \mapsto [\![e_1]\!](x) + [\![e_2]\!](x) \\
[\![e_1 * e_2]\!] &\stackrel{\text{def}}{=} x \mapsto [\![e_1]\!](x) \times [\![e_2]\!](x)
\end{aligned}$$

(encore appelée sémantique dénotationnelle à la Strachey)

on peut définir la sémantique d'un langage en le traduisant vers un langage dont la sémantique est déjà connue

exemple

un langage ésotérique dont la syntaxe est composée de huit caractères et dont la sémantique peut être définie par traduction vers le langage C

commande	traduction en C
(prélude)	`char array[30000] = {0};` `char *ptr = array;`
`>`	`++ptr;`
`<`	`--ptr;`
`+`	`++*ptr;`
`-`	`--*ptr;`
`.`	`putchar(*ptr);`
`,`	`*ptr = getchar();`
`[`	`while (*ptr) {`
`]`	`}`

la **sémantique opérationnelle** décrit l'enchaînement des calculs élémentaires qui mènent de l'expression à son résultat (sa valeur)

deux formes de sémantique opérationnelle

- « sémantique naturelle » ou « à grands pas » (*big-steps semantics*)

$$e \twoheadrightarrow v$$

- « sémantique à réductions » ou « à petits pas » (*small-steps semantics*)

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

illustrons la sémantique opérationnelle sur le langage **mini-ML**

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & c & \text{constante } (1,\ 2,\ \dots,\ \mathit{true},\ \dots) \\
 & | & \mathit{op} & \text{primitive } (+,\ \times,\ \mathit{fst},\ \dots) \\
 & | & \texttt{fun}\ x \rightarrow e & \text{fonction} \\
 & | & e\ e & \text{application} \\
 & | & (e, e) & \text{paire} \\
 & | & \texttt{let}\ x = e\ \texttt{in}\ e & \text{liaison locale}
\end{array}
$$

$$\texttt{let}\ \mathit{compose} = \texttt{fun}\ f \rightarrow \texttt{fun}\ g \rightarrow \texttt{fun}\ x \rightarrow f\ (g\ x)\ \texttt{in}$$
$$\texttt{let}\ \mathit{plus} = \texttt{fun}\ x \rightarrow \texttt{fun}\ y \rightarrow +\ (x, y)\ \texttt{in}$$
$$\mathit{compose}\ (\mathit{plus}\ 2)\ (\mathit{plus}\ 4)\ 36$$

$$\texttt{let}\ \mathit{distr\_pair} = \texttt{fun}\ f \rightarrow \texttt{fun}\ p \rightarrow (f\ (\mathit{fst}\ p), f\ (\mathit{snd}\ p))\ \texttt{in}$$
$$\texttt{let}\ p = \mathit{distr\_pair}\ (\texttt{fun}\ x \rightarrow x)\ (40, 2)\ \texttt{in}$$
$$+\ (\mathit{fst}\ p, \mathit{snd}\ p)$$

la conditionnelle peut être définie comme

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

où *opif* est une primitive

les branches sont **gelées** à l'aide de fonctions de même, la récursivité peut être définie comme

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

où *opfix* est un opérateur de point fixe, satisfaisant

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

exemple d'utilisation :

$$\begin{array}{ll}\mathit{opfix}\ (\texttt{fun}\ \mathit{fact} \rightarrow \texttt{fun}\ n \rightarrow & \texttt{if}\ n = 0\ \texttt{then}\ 1 \\ & \texttt{else}\ \ \times(n, \mathit{fact}\ (-(n, 1))))\end{array}$$

on cherche à définir une relation entre une expression e et une **valeur** v

$$e \twoheadrightarrow v$$

les valeurs sont ainsi définies

$$\begin{array}{rcll} v & ::= & c & \text{constante} \\ & | & op & \text{primitive non appliquée} \\ & | & \mathtt{fun}\ x \rightarrow e & \text{fonction} \\ & | & (v, v) & \text{paire} \end{array}$$

pour définir $e \twoheadrightarrow v$, on a besoin des notions de **règles d'inférence** et de **substitution** une relation peut être définie comme la **plus petite relation** satisfaisant un ensemble d'axiomes de la forme

$$\overline{\quad P \quad}$$

et un ensemble d'implications de la forme

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

exemple : on peut définir la relation $\mathsf{Pair}(n)$ par les deux règles

$$\frac{}{\mathsf{Pair}(0)} \qquad \text{et} \qquad \frac{\mathsf{Pair}(n)}{\mathsf{Pair}(n+2)}$$

qui doivent se lire comme

$$\begin{array}{rl} \text{d'une part} & \mathsf{Pair}(0) \\ \text{et d'autre part} & \forall n.\ \mathsf{Pair}(n) \Rightarrow \mathsf{Pair}(n+2) \end{array}$$

la plus petite relation satisfaisant ces deux propriétés coïncide avec la propriété « n est un entier naturel pair » :

- les entiers pairs sont clairement dedans, par récurrence
- s'il y avait un entier impair, on pourrait enlever le plus petit

une **dérivation** est un arbre dont les nœuds correspondent aux règles et les feuilles aux axiomes ; exemple

$$\dfrac{\dfrac{\overline{\mathsf{Pair}(0)}}{\mathsf{Pair}(2)}}{\mathsf{Pair}(4)}$$

l'ensemble des dérivations possibles caractérise exactement la plus petite relation satisfaisant les règles d'inférence

Définition (variables libres)

L'ensemble des **variables libres** *d'une expression* e*, noté* $fv(e)$*, est défini par récurrence sur* e *de la manière suivante :*

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

Une expression sans variable libre est dite **close**.

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

$$\mathit{fv}\,(\texttt{let}\ x = \texttt{z}\ \texttt{in}\ (\texttt{fun}\ y \rightarrow (x\ y)\ \texttt{t})) \quad = \quad \{z, t\}$$

Définition (substitution)

Si e est une expression, x une variable et v une valeur, on note $e[x \leftarrow v]$ la **substitution** *de toute occurrence libre de x dans e par v, définie par*

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

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

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

$$(\texttt{fun}\ y \rightarrow y\ y)\,[y \leftarrow 17] \quad = \quad \texttt{fun}\ y \rightarrow y\ y$$

$$\frac{}{c \twoheadrightarrow c} \qquad \frac{}{op \twoheadrightarrow op} \qquad \frac{}{(\texttt{fun}\ x \rightarrow e) \twoheadrightarrow (\texttt{fun}\ x \rightarrow e)}$$

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

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

note : on a fait le choix d'une stratégie d'**appel par valeur**
i.e. l'argument est complètement évalué avant l'appel (cf cours 7)

il faut ajouter des règles pour les primitives ; par exemple

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

$$\frac{e_1 \twoheadrightarrow \mathit{opif} \quad e_2 \twoheadrightarrow (\mathit{true}, ((\mathtt{fun}\ \_ \to e_3), (\mathtt{fun}\ \_ \to e_4))) \quad e_3 \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

$$\frac{e_1 \twoheadrightarrow \mathit{opfix} \quad e_2 \twoheadrightarrow (\mathtt{fun}\ f \to e) \quad e[f \leftarrow \mathit{opfix}\ (\mathtt{fun}\ f \to e)] \twoheadrightarrow v}{e_1\ e_2 \twoheadrightarrow v}$$

$$\frac{e_1 \twoheadrightarrow \mathit{fst} \quad e_2 \twoheadrightarrow (v_1, v_2)}{e_1\ e_2 \twoheadrightarrow v_1}$$

$$\dfrac{\dfrac{\dfrac{}{+ \twoheadrightarrow +} \qquad \dfrac{\dfrac{}{20 \twoheadrightarrow 20} \quad \dfrac{}{1 \twoheadrightarrow 1}}{(20,1) \twoheadrightarrow (20,1)}}{+(20,1) \twoheadrightarrow 21} \qquad \dfrac{\dfrac{}{\texttt{fun}\ldots \twoheadrightarrow \texttt{fun}\ldots} \quad \dfrac{}{21 \twoheadrightarrow 21} \quad \dfrac{\vdots}{+(21,21) \twoheadrightarrow 42}}{(\texttt{fun}\ y \to +(y,y))\ 21 \twoheadrightarrow 42}}{\texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \to +(y,y))\ x \twoheadrightarrow 42}$$

donner la dérivation de

$$(\mathit{opfix}\ F)\ 2$$

avec F défini comme

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

il existe des expressions e pour lesquelles il n'y a **pas de valeur** v telle que $e \twoheadrightarrow v$

exemple : $e = 1\ 2$

exemple : $e = (\texttt{fun}\ x \to x\ x)\ (\texttt{fun}\ x \to x\ x)$

pour établir une propriété d'une relation définie par un ensemble de règles d'inférence, on peut raisonner par **récurrence** sur la dérivation

cela signifie par récurrence structurelle *i.e.* on peut appliquer l'hypothèse de récurrence à toute sous-dérivation
(de manière équivalente, on peut dire que l'on raisonne par récurrence sur la hauteur de la dérivation)

en pratique, on raisonne par récurrence sur la dérivation et par cas sur la dernière règle utilisée

propriétés de la sémantique naturelle de mini-ML

Proposition (l'évaluation produit des valeurs closes)

Si $e \twoheadrightarrow v$ alors v est une valeur.
De plus, si e est close, alors v l'est également.

preuve par récurrence sur la dérivation $e \twoheadrightarrow v$
cas d'une application

$$\frac{\begin{array}{c}(D_1)\\ \vdots\\ e_1 \twoheadrightarrow (\texttt{fun}\ x \rightarrow e)\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots\\ e_2 \twoheadrightarrow v_2\end{array} \qquad \begin{array}{c}(D_3)\\ \vdots\\ e[x \leftarrow v_2] \twoheadrightarrow v\end{array}}{e_1\ e_2 \twoheadrightarrow v}$$

par HR v est une valeur
si e est close alors e_1 et e_2 aussi, et par HR $\texttt{fun}\ x \rightarrow e$ et v_2 sont closes, donc $e[x \leftarrow v_2]$ est close, et par HR v aussi

(exercice : traiter les autres cas)

propriétés de la sémantique naturelle de mini-ML

Proposition (déterminisme de l'évaluation)

Si $e \twoheadrightarrow v$ et $e \twoheadrightarrow v'$ alors $v = v'$.

par récurrence sur les dérivations de $e \twoheadrightarrow v$ et de $e \twoheadrightarrow v'$
cas d'une paire $e = (e_1, e_2)$

$$
\frac{\begin{array}{c}(D_1)\\ \vdots\\ e_1 \twoheadrightarrow v_1\end{array} \qquad \begin{array}{c}(D_2)\\ \vdots\\ e_2 \twoheadrightarrow v_2\end{array}}{(e_1, e_2) \twoheadrightarrow (v_1, v_2)}
\qquad
\frac{\begin{array}{c}(D_1')\\ \vdots\\ e_1 \twoheadrightarrow v_1'\end{array} \qquad \begin{array}{c}(D_2')\\ \vdots\\ e_2 \twoheadrightarrow v_2'\end{array}}{(e_1, e_2) \twoheadrightarrow (v_1', v_2')}
$$

par HR on a $v_1 = v_1'$ et $v_2 = v_2'$ donc $v = (v_1, v_2) = (v_1', v_2') = v'$

(exercice : traiter les autres cas)

remarque : la relation d'évaluation n'est pas nécessairement déterministe

exemple : on ajoute une primitive *random* et la règle

$$\frac{e_1 \twoheadrightarrow \mathit{random} \quad e_2 \twoheadrightarrow n_1 \quad 0 \leq n < n_1}{e_1\ e_2 \twoheadrightarrow n}$$

on a alors $\mathit{random}\ 2 \twoheadrightarrow 0$ aussi bien que $\mathit{random}\ 2 \twoheadrightarrow 1$

on peut programmer un **interprète** en suivant les règles de la sémantique naturelle

on se donne un type pour la syntaxe abstraite des expressions

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

et on définit une fonction

```
val eval: expression -> expression
```

correspondant à la relation $\twoheadrightarrow$ (puisqu'il s'agit d'une fonction)

```ocaml
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
```

il faut coder l'opération de substitution $e[x \leftarrow v]$

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

on suppose v close (donc pas de problème de capture de variable)

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

la sémantique naturelle est réalisée par la fonction

```
val eval: expression -> expression
```

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

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

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

```
- : expression = Const 42
```

le filtrage est volontairement non-exhaustif

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

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

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

(on pourrait préférer un type option, une exception explicite, etc.)

l'évaluation peut ne pas terminer

par exemple sur

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

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

```
Interrupted.
```

ajouter les opérateurs *opif* et *opfix* à cet interprète peut-on éviter l'opération de substitution ?

idée : on interprète l'expression e à l'aide d'un **environnement** donnant la valeur courante de chaque variable (un dictionnaire)

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

difficulté : le résultat de

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

est une fonction qui doit « mémoriser » que $x = 1$

réponse : il faut utiliser une **fermeture** on utilise le module `Map` pour les environnements

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

on définit un nouveau type pour les valeurs

```
type value =
  | Vconst of int
  | Vop    of string
  | Vpair  of value * value
  | Vfun   of string * environment * expression

and environment = value Smap.t
```

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

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

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

note : c'est ce que l'on fait quand on compile ML (cf cours 8) ajouter l'opérateur *opif* à cet interprète

note : ajouter l'opérateur *opfix* est plus complexe
(on en reparlera dans le cours 8)

la sémantique naturelle ne permet pas de distinguer les expressions dont le calcul « plante », comme

$$1\ 2$$

des expressions dont l'évaluation ne termine pas, comme

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

la sémantique opérationnelle **à petits pas** y remédie en introduisant une notion d'étape élémentaire de calcul $e_1 \rightarrow e_2$, que l'on va itérer

on peut alors distinguer trois situations

1. l'itération aboutit à une valeur

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

2. l'itération bloque sur e_n irréductible qui n'est pas une valeur

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$$

3. l'itération ne termine pas

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots$$

sémantique opérationnelle à petits pas pour mini-ML

on commence par définir une relation $\xrightarrow{\epsilon}$ correspondant à une réduction « en tête », c'est-à-dire au sommet de l'expression

deux règles :

$$\begin{array}{rcl} (\texttt{fun}\ x \rightarrow e)\ v & \xrightarrow{\epsilon} & e[x \leftarrow v] \\ \texttt{let}\ x = v\ \texttt{in}\ e & \xrightarrow{\epsilon} & e[x \leftarrow v] \end{array}$$

note : là encore, on a fait le choix d'une stratégie d'**appel par valeur** on se donne également des règles pour les primitives

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

$$\begin{array}{rcl} \mathit{fst}\ (v_1, v_2) & \xrightarrow{\epsilon} & v_1 \\ \mathit{snd}\ (v_1, v_2) & \xrightarrow{\epsilon} & v_2 \end{array}$$

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

$$\begin{array}{rcl} \mathit{opif}\ (\mathit{true}, ((\texttt{fun}\ \_ \rightarrow e_1), (\texttt{fun}\ \_ \rightarrow e_2))) & \xrightarrow{\epsilon} & e_1 \\ \mathit{opif}\ (\mathit{false}, ((\texttt{fun}\ \_ \rightarrow e_1), (\texttt{fun}\ \_ \rightarrow e_2))) & \xrightarrow{\epsilon} & e_2 \end{array}$$

pour réduire en profondeur, on introduit la règle d'inférence

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

où E est un **contexte**, défini par la grammaire suivante :

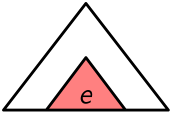
$$\begin{array}{rcl} E & ::= & \square \\ & | & E\ e \\ & | & v\ E \\ & | & \mathtt{let}\ x = E\ \mathtt{in}\ e \\ & | & (E, e) \\ & | & (v, E) \end{array}$$

un contexte est donc un « terme à trou », où $\square$ représente le trou

exemple :

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

$E(e)$ dénote le contexte E dans lequel $\square$ a été remplacé par e



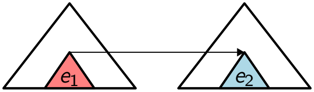
exemple :

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

la règle

$$\frac{e_1 \stackrel{\epsilon}{\to} e_2}{E(e_1) \to E(e_2)}$$

permet donc d'évaluer une sous-expression



exemple : on a la réduction

$$\frac{+(1,2) \stackrel{\epsilon}{\to} 3}{\texttt{let } x = +(1,2) \texttt{ in } +(x,x) \to \texttt{let } x = 3 \texttt{ in } +(x,x)}$$

grâce au contexte $E \stackrel{\text{def}}{=} \texttt{let } x = \square \texttt{ in } +(x,x)$

tels qu'on les a choisis, les contextes impliquent ici une évaluation en appel par valeur et ≪ de gauche à droite ≫

ainsi, $(+(1,2),\square)$ n'est pas un contexte d'évaluation

on aurait pu également choisir une évaluation de droite à gauche

on note $\stackrel{\star}{\rightarrow}$ la clôture réflexive et transitive de $\rightarrow$
(*i.e.* $e_1 \stackrel{\star}{\rightarrow} e_2$ ssi e_1 se réduit en e_2 en zéro, une ou plusieurs étapes)

on appelle **forme normale** toute expression e telle qu'il n'existe pas d'expression e' telle que $e \rightarrow e'$

les valeurs sont des formes normales ; les formes normales qui ne sont pas des valeurs sont les expressions erronées (comme 1 2) on va écrire les fonctions suivantes :

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

correspond à $\stackrel{\epsilon}{\rightarrow}$

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

décompose une expression sous la forme $E(e)$
avec e réductible en tête

```
val reduce1: expression -> expression option
```

correspond à $\rightarrow$

```
val reduce: expression -> expression
```

correspond à $\stackrel{\star}{\rightarrow}$

on commence par caractériser les valeurs

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

on écrit ensuite la réduction en tête

```
let head_reduction = function
  | App (Fun (x, e1), e2) when is_a_value e2 ->
      subst e1 x e2
  | Let (x, e1, e2) when is_a_value e1 ->
      subst e2 x e1
  | App (Op "+", Pair (Const n1, Const n2)) ->
      Const (n1 + n2)
  | App (Op "fst", Pair (e1, e2))
    when is_a_value e1 && is_a_value e2 ->
      e1
  | App (Op "snd", Pair (e1, e2))
    when is_a_value e1 && is_a_value e2 ->
      e2
  | _ ->
      raise NoReduction
```

un contexte E peut être directement représenté par la fonction $e \mapsto E(e)$

```
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
  (* on ne peut décomposer *)
  | Var _ | Const _ | Op _ | Fun _ ->
     raise NoReduction
  (* cas d'une réduction en tête *)
  | App (Fun (x, e1), e2) when is_a_value e2 ->
     (hole, e)
  | Let (x, e1, e2) when is_a_value e1 ->
     (hole, e)
  | App (Op "+", Pair (Const n1, Const n2)) ->
     (hole, e)
  | App (Op ("fst" | "snd"), Pair (e1, e2))
    when is_a_value e1 && is_a_value e2 ->
     (hole, e)
  ...
```

```
...
(* cas d'une réduction en profondeur *)
| App (e1, e2) ->
    if is_a_value e1 then
      let (ctx, rd) = decompose e2 in
      (app_right e1 ctx, rd)
    else
      let (ctx, rd) = decompose e1 in
      (app_left ctx e2, rd)
| Let (x, e1, e2) ->
    let (ctx, rd) = decompose e1 in
    (let_left x ctx e2, rd)
| Pair (e1, e2) ->
    ...
```

```
let reduce1 e =
  try
    let ctx, e' = decompose e in
    Some (ctx (head_reduction e'))
  with NoReduction ->
    None
```

enfin

```
let rec reduce e =
  match reduce1 e with None -> e | Some e' -> reduce e'
```

un tel interprète n'est pas très efficace

il passe son temps à recalculer le contexte puis à l'« oublier »

on peut faire mieux, par exemple en utilisant un *zipper* [Huet, 1997] nous allons montrer que les deux sémantiques opérationnelles sont équivalentes pour les expressions dont l'évaluation termine sur une valeur, *i.e.*

$$e \twoheadrightarrow v \quad \text{si et seulement si} \quad e \xrightarrow{\star} v$$

Lemme (passage au contexte des réductions)

Supposons $e \to e'$. Alors pour toute expression e_2 et toute valeur v

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

preuve : de $e \to e'$ on sait qu'il existe un contexte E tel que

$$e = E(r) \qquad e' = E(r') \qquad r \stackrel{\epsilon}{\to} r'$$

considérons le contexte $E_1 \stackrel{\text{def}}{=} E\ e_2$; alors

$$\frac{r \stackrel{\epsilon}{\to} r'}{E_1(r) \to E_1(r')} \qquad i.e. \qquad \frac{r \stackrel{\epsilon}{\to} r'}{e\ e_2 \to e'\ e_2}$$

(de même pour les cas 2 et 3)

Proposition (“grands pas” implique “petits pas”)

Si $e \twoheadrightarrow v$, alors $e \xrightarrow{\star} v$.

preuve : par récurrence sur la dérivation de $e \twoheadrightarrow v$
supposons que la dernière règle soit

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

équivalence des deux sémantiques opérationnelles

par HR on a

$$\begin{array}{l} e_1 \rightarrow \cdots \rightarrow v_1 = (\texttt{fun}\ x \rightarrow e_3) \\ e_2 \rightarrow \cdots \rightarrow v_2 \\ e_3[x \leftarrow v_2] \rightarrow \cdots \rightarrow v \end{array}$$

par passage au contexte (lemme précédent) on a également

$$\begin{array}{l} e_1\ e_2 \rightarrow \cdots \rightarrow v_1\ e_2 \\ v_1\ e_2 \rightarrow \cdots \rightarrow v_1\ v_2 \\ e_3[x \leftarrow v_2] \rightarrow \cdots \rightarrow v \end{array}$$

en insérant la réduction

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

on obtient la réduction complète

$$e_1\ e_2 \rightarrow \cdots \rightarrow v$$

□

pour le sens inverse (“petits pas” implique “grands pas”) on a besoin de deux lemmes

Lemme (les valeurs sont déjà évaluées)

$v \twoheadrightarrow v$ pour toute valeur v.

preuve : immédiat

Lemme (réduction et évaluation)

Si $e \rightarrow e'$ et $e' \twoheadrightarrow v$, alors $e \twoheadrightarrow v$.

preuve : on commence par les réductions de tête *i.e.* $e \stackrel{\epsilon}{\rightarrow} e'$

supposons par exemple $e = (\texttt{fun}\ x \rightarrow e_1)\ v_2$ et $e' = e_1[x \leftarrow v_2]$
on construit la dérivation

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

en utilisant le lemme précédent ($v_2 \twoheadrightarrow v_2$) et l'hypothèse $e' \twoheadrightarrow v$

(traiter les autres cas)

montrons maintenant que si $e \xrightarrow{\epsilon} e'$ et $E(e') \twoheadrightarrow v$ alors $E(e) \twoheadrightarrow v$

par récurrence structurelle sur E ; on vient de faire le cas $E = \square$

considérons le cas $E = E'\ e_2$
on a $E(e') \twoheadrightarrow v$ c'est-à-dire $E'(e')\ e_2 \twoheadrightarrow v$, qui a par exemple la forme

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

par HR on a $E'(e) \twoheadrightarrow (\mathtt{fun}\ x \rightarrow e_3)$, et donc

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

c'est-à-dire $E(e) \twoheadrightarrow v$ (traiter les autres cas) $\square$

équivalence des deux sémantiques opérationnelles

Proposition (“petits pas” implique “grands pas”)

Si $e \xrightarrow{\star} v$, *alors* $e \twoheadrightarrow v$.

preuve : supposons $e \rightarrow e_1 \rightarrow \cdots \rightarrow e_n \rightarrow v$

on a $v \twoheadrightarrow v$ et donc par le lemme précédent $e_n \twoheadrightarrow v$

de même $e_{n-1} \twoheadrightarrow v$

et ainsi de suite jusqu'à $e \twoheadrightarrow v$

on peut définir une sémantique opérationnelle, à grands pas ou à petits pas, pour un langage avec des traits impératifs

on associe typiquement un **état** S à l'expression évaluée / réduite

$$S, e \twoheadrightarrow S', v \quad \text{ou encore} \quad S, e \rightarrow S', e'$$

exemple de règle :

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

l'état S peut être décomposé en plusieurs éléments, pour modéliser une pile (des variables locales), un tas, et plus encore...

- TD 2
  - écriture d'un interprète pour mini-Python

- prochain cours
  - analyse lexicale

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