

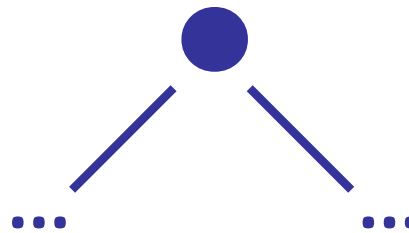
# Projet de compilation spécification du langage WHILE

Olivier Ridoux

# Domaine de calcul (1)

- Arbres binaires = `binTree`

1.



2.



3. `symb` ( $\approx$  chaîne)

# Domaine de calcul (2)

- Un seul type = binTree
- **Pas de int, bool, string, ...**

# Domaine de calcul (3)

- Simuler les autres types

– bool

$$\llbracket \otimes \rrbracket_{\text{bool}} = \text{False} \text{ et } \llbracket \text{autre} \rrbracket_{\text{bool}} = \text{True}$$

– int

$$\begin{array}{c} \llbracket \bullet \rrbracket_{\text{int}} = \llbracket t \rrbracket_{\text{int}} + 1 \text{ et } \llbracket \text{autre} \rrbracket_{\text{int}} = 0 \\ \swarrow \quad \searrow \\ \dots \quad t \end{array}$$

– ...

spécification

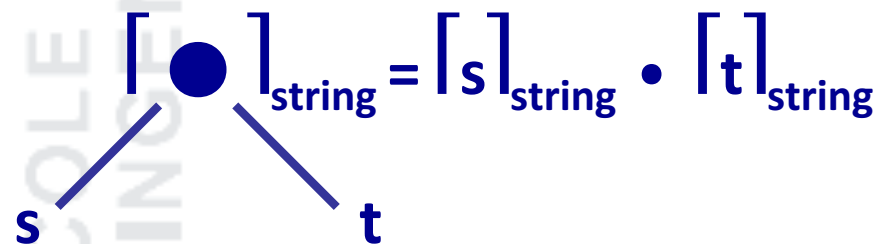
# Domaine de calcul (4)

- Simuler les autres types

– string

$$[\text{syml}]_{\text{string}} = \text{syml.str}$$

$$[\otimes]_{\text{string}} = \varepsilon$$

$$[\bullet]_{\text{string}} = [s]_{\text{string}} \cdot [t]_{\text{string}}$$


spécification

# Expressions (0)

- Le rôle fondamental des expressions  
est de  
**dénoter des arbres binaires**

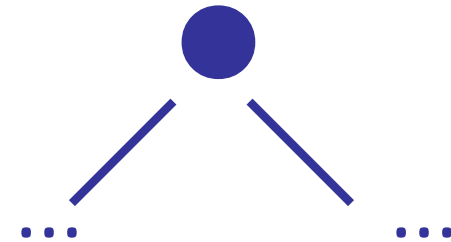
# Expressions (1)

- **(cons ... ...)**  $\rightarrow$

- **nil**  $\rightarrow \otimes$

- **Var**  $\rightarrow$  **valeur de** Var = binTree

- **symb**  $\rightarrow$  **symbole de** symb.str

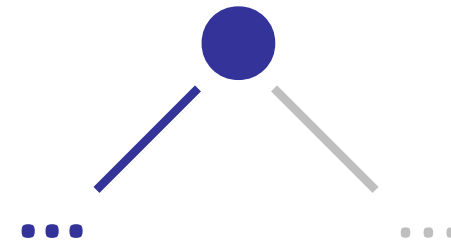


# Expressions (1)

- **(hd t)** →

si t →

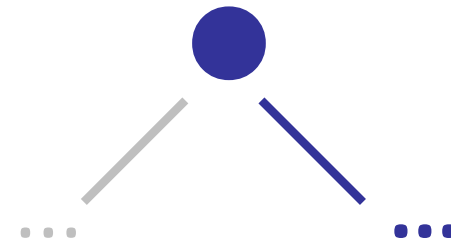
alors (hd t) →



sinon (hd t) →



- Pareil pour **(tl t)**





# Pas d'erreurs à l'exécution (1)

- Gérer proprement les erreurs à l'exécution est **complexe**...

...ne pas les gérer proprement  
n'est **pas intéressant**,  
voire **irresponsable**

→ s'arranger pour qu'il n'y ait  
**pas d'erreur** à l'exécution

# Pas d'erreurs à l'exécution (2)

- Valeur par défaut =  $\otimes$  pour...

...les variables **pas initialisées**

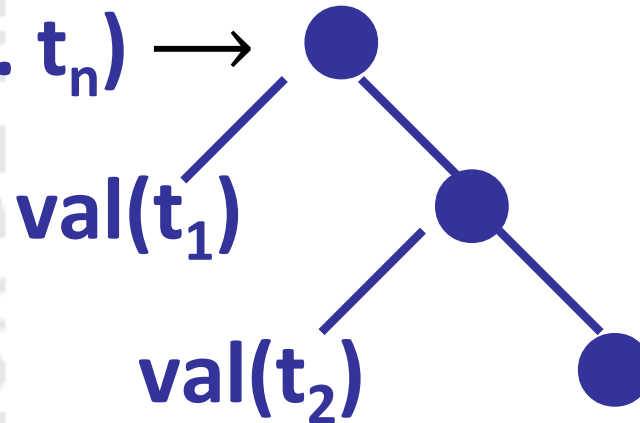
...les opérations **pas définies**

$$(\text{hd } \otimes) = (\text{tl } \otimes) = \otimes$$

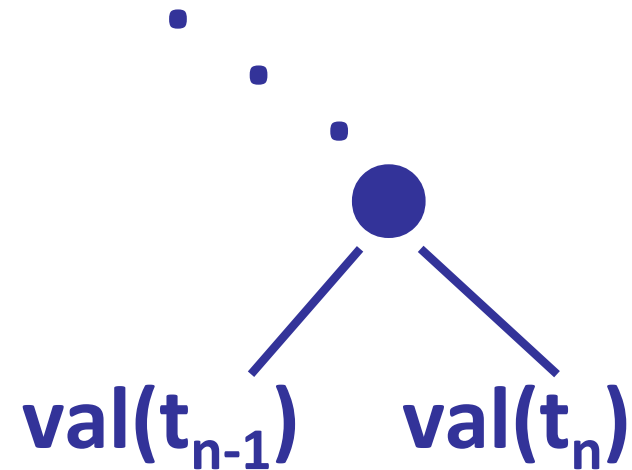
$$(\text{hd toto}) = (\text{tl toto}) = \otimes$$

# Expressions (2)

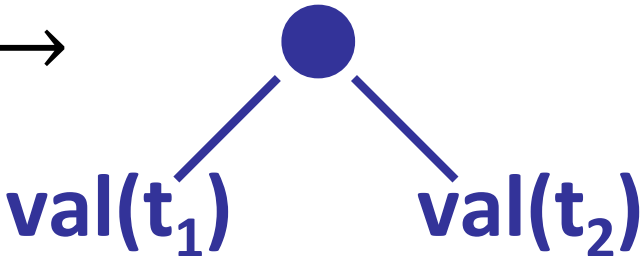
- $(\text{cons } t_1 \ t_2 \ \dots \ t_n) \rightarrow$



- $(\text{cons } t_1 \ t_2 \ \dots \ t_n) =$   
 $(\text{cons } t_1$   
 $(\text{cons } t_2$   
 $\dots (\text{cons } t_{n-1} \ t_n) \dots))$

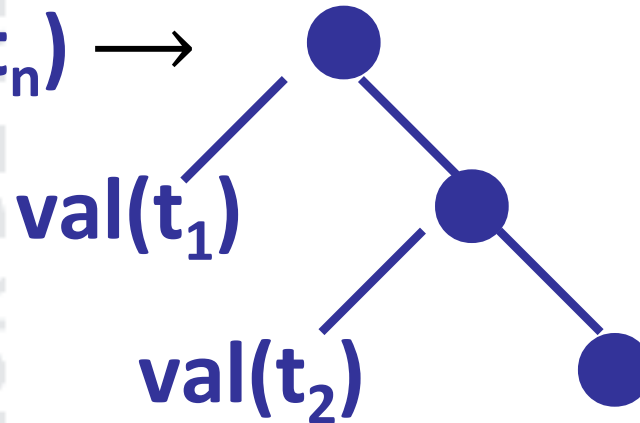


# Expressions (3)

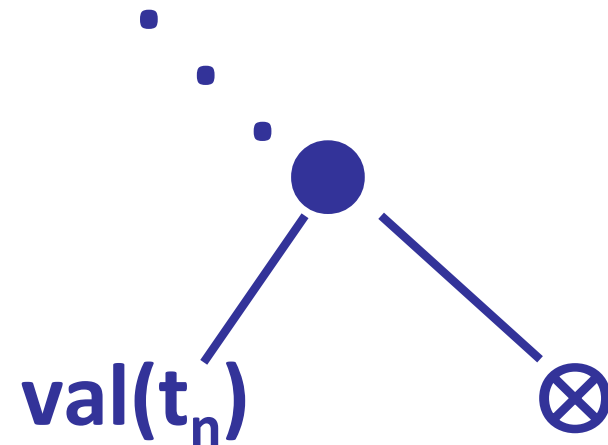
- $(\text{cons } t_1 \ t_2) \rightarrow$  
- $(\text{cons } t_1) \rightarrow$   $\text{val}(t_1)$
- $(\text{cons}) \rightarrow$   $\otimes$

# Expressions (3)

- $(\text{list } t_1 t_2 \dots t_n) \rightarrow$

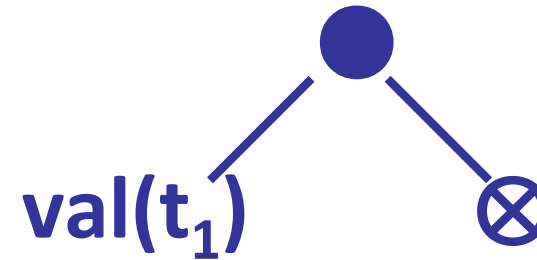


- $(\text{list } t_1 t_2 \dots t_n) =$   
 $(\text{cons } t_1$   
 $\text{ (cons } t_2$   
 $\dots (\text{cons } t_n \text{ nil}) \dots))$



# Expressions (4)

- $(\text{list } t_1) \rightarrow$



- $(\text{list}) \rightarrow$



spécification

# Expressions (7)

- $(f\ t_1\ t_2\ \dots\ t_n) \rightarrow$

si **f** est une fonction définie

function **f**: ...

si **f** a le bon nombre de paramètres

read  $X_1, X_2, \dots, X_N$

si **f** a le bon nombre de résultats

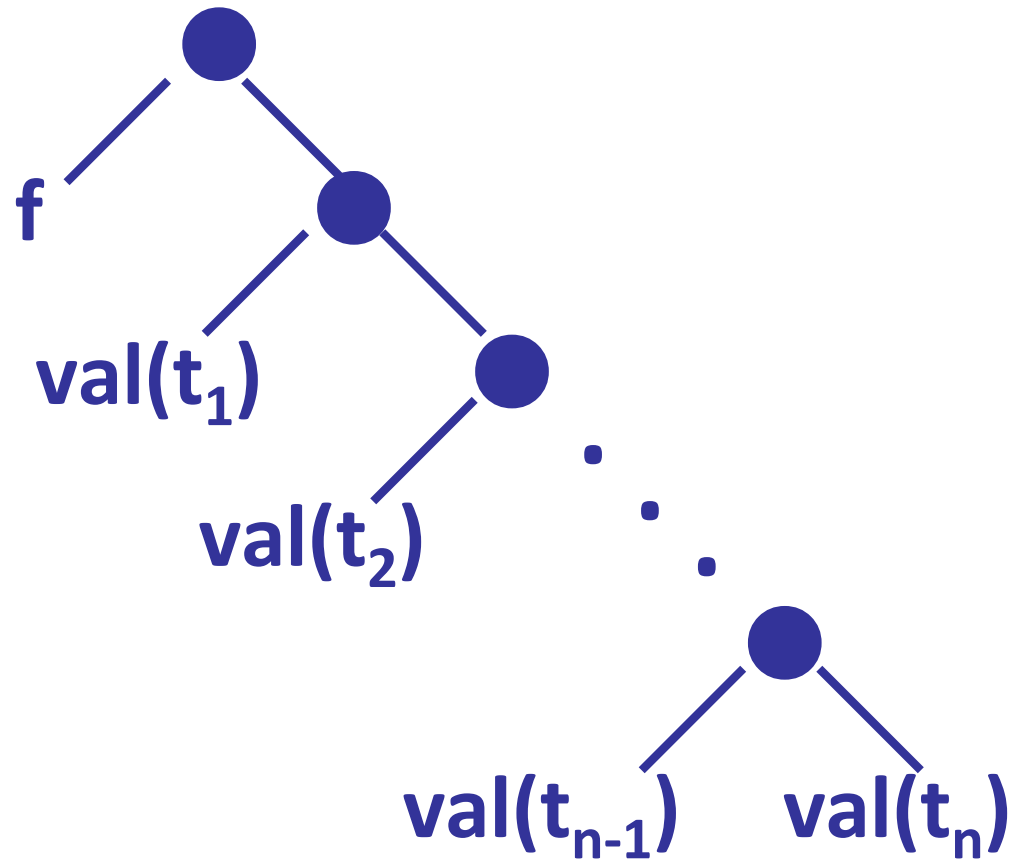
write **Y**

alors **valeur de Y**

- **Vérification de type par le compilateur**

# Expressions (8)

•  $(f\ t_1\ t_2\ \dots\ t_n) \rightarrow$



si  $f$  n'est pas une fonction définie

spécification



# La mémoire

- La mémoire du processus d'exécution stocke :
  1. des **arbres binaires**
  2. des **relations entre variables et arbres binaires**
- **Il n'y a pas de variable dans les arbres binaires**
- **On ne peut pas modifier un arbre binaire**

# Variables (1)

- $v$   
...s'évalue dans la mémoire courante  
...valeur par défaut =  $\otimes$
- Mémoire : **variable**  $\rightarrow$  **binTree**

# Variables (2)

- Les variables sont locales à la fonction où elles apparaissent
- **Pas de variable globale !**

**...les programmes sont purement fonctionnels**

# Variables (3)

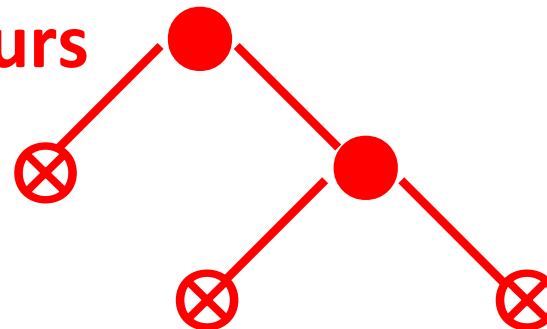
- Pas d'effet de bord !

$X := (\text{cons nil nil}) ;$

$Y := (\text{cons nil } X) ;$

$X := \text{nil}$

...Y vaut toujours

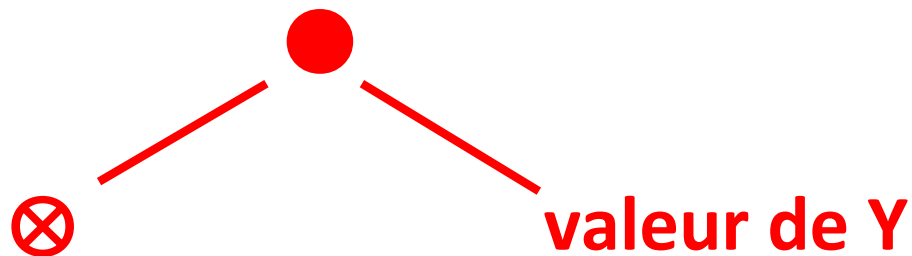


...le seul moyen de modifier Y est  $Y := \dots$

# Variables (1.4)

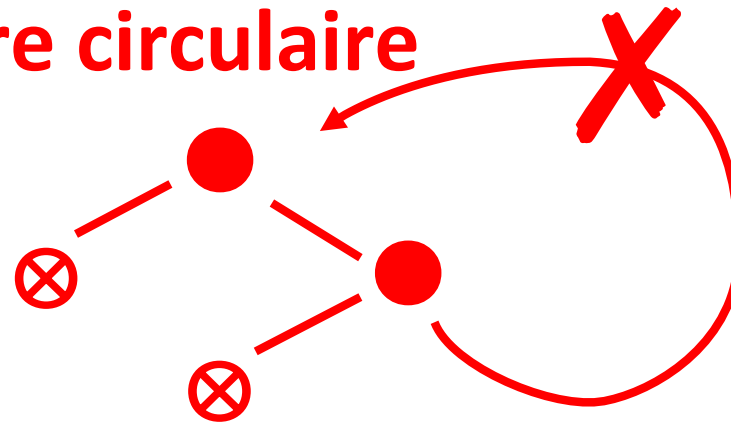
- Pas de variable dans les termes

$X := (\text{cons nil } Y)$

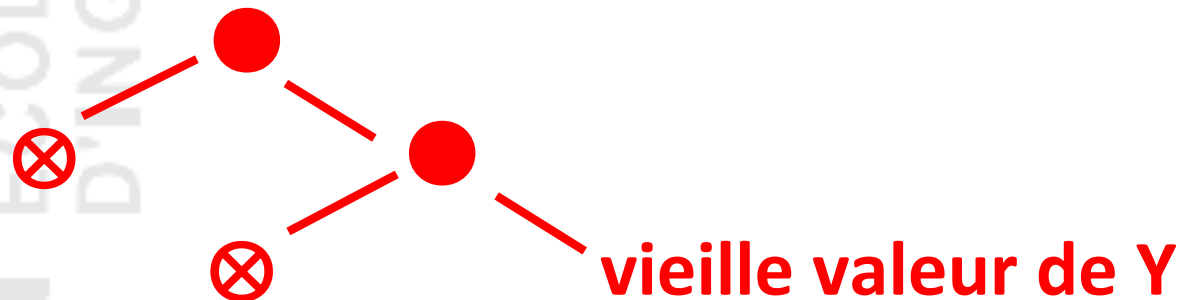


# Variables (4)

- Pas de structure circulaire



$X ::= (\text{cons nil} (\text{cons nil } Y)) ; Y := X$



spécification

# Commandes (0)

- Le rôle **fondamental** des commandes  
est de

**modifier le contenu de la mémoire**

- **On ne peut rien modifier d'autre**

# Commandes (1)

- **nop**

ne fait rien

- Utile pour le principe, et pour le test



# Commandes (2)

- $C_1 ; C_2$

exécute  $C_1$  puis  $C_2$

- L'exécution de  $C_1$  modifie l'état de la mémoire...
- ... $C_2$  est exécuté dans le nouvel état, et modifie à nouveau l'état de la mémoire

# Commandes (3)

- **if E then  $C_1$  else  $C_2$  fi**  
si  $\llbracket E \rrbracket_{\text{bool}}$  alors exécuter  $C_1$ ,  
sinon exécuter  $C_2$
- **if E then  $C_1$  fi**  
si  $\llbracket E \rrbracket_{\text{bool}}$  alors exécuter  $C_1$ ,  
sinon ne rien faire

# Commandes (4)

- **while E do C od**

si  $[E]_{bool}$  alors exécuter C et recommencer  
sinon ne rien faire

...boucle while standard où  $\otimes$  joue le rôle de **faux**

...C est exécuté 0, 1, ..., ou n fois

**C\***

- **while E do C od peut boucler indéfiniment !**

# Commandes (5.1)

- **for E do C od**

exécuter  $\lceil E \rceil_{\text{int}}$  fois la commande **C**

$$C^{\lceil E \rceil_{\text{int}}}$$

- 

soit  $v$  la valeur de  $E$

Ⓐ si  $v \neq \bullet$  ne rien faire sinon exécuter  $C$   
et recommencer en Ⓐ avec  $v = (tl\ v)$

- for E do C od **ne peut pas boucler indéfiniment**

## Commandes (5.2)

- L'exécution de C ne doit pas perturber le décompte du for  
**for X do X := (cons nil X) od**

...double la longueur de X

**$X \rightarrow (\text{cons nil } (\text{cons nil } \dots X \dots))$**   
(X fois)

# Commandes (5.3)

**for E do C od**

**$\neq$**

**while E do C ; E := (tl E) od**

# Commandes (6)

- **foreach X in E do C od**

exécuter  $\lceil E \rceil_{int}$  fois la commande **C**  
 $C^{\lceil E \rceil_{int}}$

en remplaçant **X** par les éléments de **E**

soit  $v$  la valeur de  $E$

Ⓐ si  $v = \text{nil}$  ne rien faire

sinon exécuter  $C[X \leftarrow (\text{hd } v)]$  et

recommencer en Ⓐ avec  $v = (\text{tl } v)$

# Commandes (7.1)

- $V_1, \dots, V_n := E_1, \dots, E_n$

évaluer les  $E_i$ ,

et affecter les valeurs aux  $V_i$  correspondants

$X, Y := Y, X$

...permuter les valeurs de X et Y



# Commandes (7.2)

$V_1, \dots, V_n := E_1, \dots, E_n$

$\neq$

$V_1 := E_1 ; V_2 := E_2 ; \dots ; V_n := E_n$

$=$

$r_1 := E_1 ; r_2 := E_2 ; \dots ; r_n := E_n$

$V_1 := r_1 ; V_2 := r_2 ; \dots ; V_n := r_n$

## Commandes (7.3)

- $V_1, \dots, V_m := (f E_1 \dots E_n)$

...évalue l'appel,

et affecte les résultats aux  $V_i$  correspondants

**Quotient, Reste := (div X Y)**

- **Vérification de type par le compilateur**

fonction f: ... read  $X_1, \dots, X_n$  % ... % write  $Y_1, \dots, Y_m$

# Lancement (1)

- Soit le programme f.wh

**function p: read X % ... % write Y**

**function main: read A % ... (p E) ... % write B**

- **main** appelle **p** ; l'affectation du **paramètre effectif E** au **paramètre formel X** est faite par la séquence d'appel de procédure

- **Qui appelle main ? D'où vient le paramètre effectif de main ? Qui l'affecte à A ?**

## Lancement (2)

- Soit la ligne de commande

**[unix] % f "(cons nil (cons nil nil))"**

- le lancement du programme **f** appelle **main**
- le lancement du programme **f** affecte au **paramètre formel de main** le **paramètre effectif** dénoté dans la ligne de commande
- analyse syntaxique de **(cons nil (cons nil nil))** !

## Lancement (2)

- Soit la ligne de commande

**[unix] % f 2**

- le lancement du programme **f** appelle **main**
- le lancement du programme **f** affecte au **paramètre formel de main** le **paramètre effectif** dénoté dans la ligne de commande
- Traduction de **2** en **(cons nil (cons nil nil))** !
- **Conversion du monde shell au monde WHILE**

# Retour (1)

- Soit le programme f.wh

**function p: read X % ... % write Y**

**function main: read A % ... (p E) ... % write B**

- **main** appelle **p** ; la captation du **paramètre de retour Y** est faite par la séquence de retour de procédure
- **Qui capte le résultat de main ? Où va la valeur de B ?**

# Retour (2)

- Soit la ligne de commande

**[unix] % f ...**

- le lancement du programme **f** appelle **main** qui appelle **p**
  - le résultat de **main** est affiché sur la console de l'environnement d'appel
  - *pretty-printing* de la valeur de **B** !
- **Conversion du monde WHILE au monde shell**

# Retour (3)

- Conventions de *pretty-printing*



→ **(cons** pp(**t**) pp\*(**...**)**)**

– ⊗

→ **nil**

– Var

→ pp(**valeur de** Var)

– symb

→ **représentation de** symb

– (int ...)

→ **[...]<sub>int</sub>**

– (bool ...)

→ **[...]<sub>bool</sub>**

– (string ...)

→ **[...]<sub>string</sub>**

**(int (cons nil (cons nil nil))) → 2 !**

spécification



# En résumé

environnement d'appel ...

(shell, HTML/DOM/JS, C/LUA, ...)

environnement d'exécution WHILE

function main: read  $X_1, \dots, X_n$

%

..

%

write  $Y_1, \dots, Y_n$

mémoire  
de binTree

... et de retour

spécification