Chapitre 8 - Interprétation d'un langage fonctionnel

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251

Compilation, interprétation

Ceci n'est pas un cours de compilation ou de création de langage, mais :

- le concept d'interpréteur est utile au delà de ce cours (DSLs, ...)
- écrire un interpréteur pour un langage permet de mieux maîtriser le langage

On va donc définir des interpréteurs pour des mini langages inspirés d'OCaml.

Compilation, interprétation

Un interpréteur :

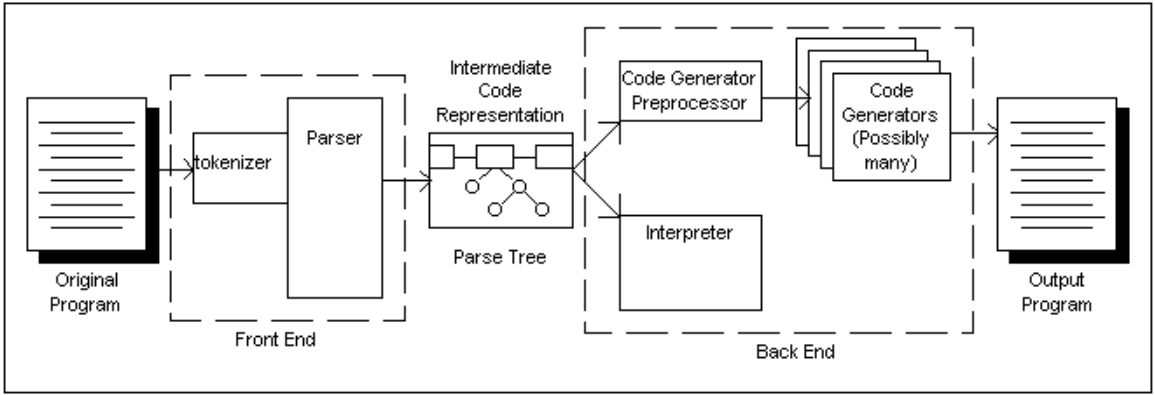
- lit un programme
- l'exécute

Un compilateur :

- lit un programme
- le transforme dans une autre représentation
  - généralement, du code machine

Les interpréteurs modernes (JavaScript, Java, ...) incluent également de la compilation.

Phases de compilation



Source

Phases de compilation

Plusieurs phases dans un compilateur :

- *lexing* : transformer l'entrée en *tokens*
- *parsing* : transformer les *tokens* en *Abstract Syntax Tree* (AST)
- analyse sémantique : vérifications faites sur le programme
- transformation en représentation intermédiaire (IR)
- optimisations sur l'IR
- génération de code

Pour plus de détails : INF600E, INF7641, INF889A, INF889J

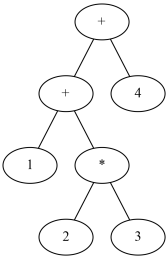
Abstract Syntax Tree (AST)

La représentation que l'on utilise pour notre langage est un *abstract syntax tree* (AST).

Une expression mathématique est représentée par un arbre, par exemple

```
(1 + (2 * 3)) + 4
```

est représenté par l'arbre :



Abstract Syntax Tree (AST)

On peut représenter ces arbres sous formes de types algébriques :

```
type expr =
  | Int of int
  | Plus of expr * expr
  | Times of expr * expr
```

Et donc, notre arbre est :

```
Plus (Plus (Int 1, Times (Int 2, Int 3)),
      Int 4)
```

Dans ce chapitre

On s'intéressera ici à l'interprétation uniquement :

- on prend un AST en entrée
- on l'exécute

Évaluation d'expressions mathématiques

Évaluation d'expressions mathématiques

Langage d'expressions mathématique :

```
type expr =
  | Int of int
  | Plus of expr * expr
  | Minus of expr * expr
  | Times of expr * expr
  | Divide of expr * expr
  | Negate of expr
```

Valeurs du langage

On introduit un synonyme de type pour dénoter les valeurs du langage

```
type value = int
```

Évaluer les expressions mathématiques

La fonction `eval` évalue une expression en une valeur de résultat :

```
let rec eval (e : expr) : value =
  match e with
  | Int n -> n
  | Plus (e1, e2) -> (eval e1) + (eval e2)
  | Minus (e1, e2) -> (eval e1) - (eval e2)
  | Times (e1, e2) -> (eval e1) * (eval e2)
  | Divide (e1, e2) -> (eval e1) / (eval e2)
  | Negate e -> 0 - (eval e)
```

Structure d'une fonction d'évaluation

La majorité des fonctions d'évaluation auront une structure similaire :

- On filtre par motif de l'expression
  - `match e with`
- Si c'est une valeur directe, on la renvoie
  - `| Int n -> n`
- Si il y a des sous-expression, il faut les évaluer (appels récursifs), et faire quelque chose avec le résultat
  - `| Negate e -> 0 - (eval e)`

Utilisation de la fonction d'évaluation

```
# eval (Plus (Plus (Int 1, Times (Int 2, Int 3)), Int 4));;
- : int = 11
```

On a donc un **interpréteur** d'expressions mathématiques

Version formelle

$$
\begin{aligned}
\mathit{eval}(n) &= n \text{ si } n \text{ est un nombre} \\
\mathit{eval}(e_1 + e_2) &= \mathit{eval}(e_1) + \mathit{eval}(e_2) \\
\mathit{eval}(e_1 - e_2) &= \mathit{eval}(e_1) - \mathit{eval}(e_2) \\
\mathit{eval}(e_1 * e_2) &= \mathit{eval}(e_1) * \mathit{eval}(e_2) \\
\mathit{eval}(e_1 / e_2) &= \mathit{eval}(e_1) / \mathit{eval}(e_2) \\
\mathit{eval}(-e) &= -\mathit{eval}(e)
\end{aligned}
$$

Liaisons

Liaisons

On souhaite ajouter des noms au langage, par exemple `x` évalue à 42

Liaisons : langage

```
type expr =
  ...
  | Var of string
```

Liaisons : environnement

On va utiliser un **environnement** qui associe des *noms* à des *valeurs* :

```
type env = (string * value) list

let lookup (env : env) (var : string) : value =
  List.assoc var env
```

Note : il existe des types de donnée plus adaptés (e.g., un *map*).

Liaisons : eval

`eval` doit alors prendre un environnement :

```
let rec eval (env : env) (e : expr) : value =
  match e with
  | Int n -> n
  | Plus (e1, e2) -> (eval env e1) + (eval env e2)
  | Minus (e1, e2) -> (eval env e1) - (eval env e2)
  | Times (e1, e2) -> (eval env e1) * (eval env e2)
  | Divide (e1, e2) -> (eval env e1) / (eval env e2)
  | Negate e -> 0 - (eval env e)
  | Var v -> lookup env v
```

Évaluation avec une liaison

```
# eval [("x", 42)] (Divide ((Var "x"), (Int 2)));;
- : value = 21
```

Version formelle

Avec un environnement $\rho : \mathit{Var} \to \mathbb{N}$:

$$
\begin{aligned}
\mathit{eval}(n, \rho) &= n \text{ si } n \text{ est un nombre} \\
\mathit{eval}(v, \rho) &= \rho(v) \text{ si } v \text{ est une variable} \\
\mathit{eval}(e_1 + e_2, \rho) &= \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho) \\
\mathit{eval}(e_1 - e_2, \rho) &= \mathit{eval}(e_1, \rho) - \mathit{eval}(e_2, \rho) \\
\mathit{eval}(e_1 * e_2, \rho) &= \mathit{eval}(e_1, \rho) * \mathit{eval}(e_2, \rho) \\
\mathit{eval}(e_1 / e_2, \rho) &= \mathit{eval}(e_1, \rho) / \mathit{eval}(e_2, \rho) \\
\mathit{eval}(-e, \rho) &= -\mathit{eval}(e, \rho)
\end{aligned}
$$

Introduction de nouvelles liaisons

On souhaite désormais pouvoir définir de nouvelles liaisons, par exemple :

```
let x = 1+2 in x+3
```

- associe la valeur (le résultat de l'évaluation) de `1+2` à `x`
- évalue `x+3`

Nouvelles liaisons : langage

```
type expr =
  ...
  | Let of string * expr * expr
```

Nouvelles liaisons : environnement

Il va falloir *étendre* l'environnement :

```
let extend (env : env) (var : string) (value : value) : env =
  (var, value) :: env
```

Note : définir plusieurs fois la même variable va *cacher* l'ancienne valeur

C'est la notion de *shadowing* vue au chapitre 2

Nouvelles liaisons : `eval`

```
let rec eval (env : env) (e : expr) : float =
  match e with
  ...
  | Let (x, e1, e2) ->
    let v = eval env e1 in
    eval (extend env x v) e2
```

Pour évaluer `let x = e1 in e2` :

- on évalue `e1` à une valeur `v`
- on ajoute la liaison `x`, associée à la valeur `v`
- on évalue `e2`

Nouvelles liaisons : utilisation

Pour évaluer l'expression suivante :

let `x = 1 + 2` **in** `x + 3`

On fait :

```
# eval [] (Let ("x", Plus (Int 1, Int 2),
                      Plus (Var "x", Int 3)));;
- : value = 6
```

Version formelle

On note l'extension d'un environnement par $\rho[x \mapsto v]$, où x est la variable et v sa valeur.

$$
\begin{aligned}
\mathit{eval}(n, \rho) &= n \text{ si } n \text{ est un nombre} \\
\mathit{eval}(x, \rho) &= \rho(v) \text{ si } x \text{ est une variable} \\
\mathit{eval}(e_1 + e_2, \rho) &= \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho) \\
\mathit{eval}(e_1 - e_2, \rho) &= \mathit{eval}(e_1, \rho) - \mathit{eval}(e_2, \rho) \\
\mathit{eval}(e_1 * e_2, \rho) &= \mathit{eval}(e_1, \rho) * \mathit{eval}(e_2, \rho) \\
\mathit{eval}(e_1 / e_2, \rho) &= \mathit{eval}(e_1, \rho) / \mathit{eval}(e_2, \rho) \\
\mathit{eval}(-e, \rho) &= -\mathit{eval}(e, \rho) \\
\mathit{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) &= \mathit{eval}(e_2, \rho[x \mapsto \mathit{eval}(e_1, \rho)])
\end{aligned}
$$

Plusieurs types de données

Plusieurs types de données

Jusqu'ici, on a uniquement manipulé des `int`, car :

```
type value = int
```

Un langage pratique se devra de supporter plusieurs types de données. On va rajouter ici des booléens.

Booléens : valeurs

On change le type des valeurs :

```
type value =
  | VBool of bool
  | VInt of int
```

On utilise les valeurs d'OCaml pour représenter les valeurs de notre langage, mais on pourrait choisir une représentation différente.

Booléens : langage

On ajoute `Bool` dans nos expressions

```
type expr =
  | Bool of bool
  ...
```

Booléens : évaluation

Il faut adapter `eval`, car on n'est plus sûr d'avoir des nombres partout.

On introduit une exception pour gérer les erreurs de types.

```
exception TypeError

let rec eval (env : env) (e : expr) : value =
  match e with
  | Int n -> VInt n
  | Bool b -> VBool n
  | Plus (e1, e2) ->
    begin match (eval env e1, eval env e2) with
      | Int v1, Int v2 -> Int (v1 + v2)
      | _ -> raise TypeError
    end
  ...
```

Plusieurs types de données

C'est une stratégie, mais il en existe d'autres :

- en C, les booléens sont des `int`
- en Java, + a un comportement différent en fonction des arguments : concaténation ou addition
  - à quoi évalue le programme suivant ?

```
System.out.println(2 + 3 + ">=" + 1 + 1);
```

Plusieurs types de données

C'est une stratégie, mais il en existe d'autres :

- en C, les booléens sont des `int`
- en Java, + a un comportement différent en fonction des arguments : concaténation ou addition
  - à quoi évalue le programme suivant ?

```
System.out.println(2 + 3 + ">=" + 1 + 1);
```

$\rightarrow$ `5 >= 11`

- en JavaScript, + est aussi compliqué (conversions implicites selon `toString` ou `valueOf`)
  - à quoi évalue le programme suivant ?

```
[] + 1
```

Plusieurs types de données

C'est une stratégie, mais il en existe d'autres :

- en C, les booléens sont des `int`
- en Java, + a un comportement différent en fonction des arguments : concaténation ou addition
  - à quoi évalue le programme suivant ?

```
System.out.println(2 + 3 + ">=" + 1 + 1);
```

→ `5 >= 11`

- en JavaScript, + est aussi compliqué (conversions implicites selon `toString` ou `valueOf`)
  - à quoi évalue le programme suivant ?

```
[] + 1
```

→ `'1'`

Plusieurs types de données

Pour éviter trop de duplication, on peut refactoriser :

```
let int_unary_operator (op : int -> int) (v : value) : value =
  match v with
  | VInt x -> VInt (op x)
  | _ -> raise TypeError

let int_binary_operator
  (op : int -> int -> int) (v1 : value) (v2 : value) : value =
  match v1, v2 with
  | VInt x, VInt y -> VInt (op x y)
  | _ -> raise TypeError
```

Plusieurs types de données

```
let rec eval (env : env) (e : expr) : value =
  match e with
  ...
  | Plus (e1, e2) ->
    int_binary_operator (+) (eval env e1) (eval env e2)
  | Minus (e1, e2) ->
    int_binary_operator (-) (eval env e1) (eval env e2)
  | Times (e1, e2) ->
    int_binary_operator ( * ) (eval env e1) (eval env e2)
  | Divide (e1, e2) ->
    int_binary_operator (/) (eval env e1) (eval env e2)
  | Negate e ->
    int_unary_operator (fun x -> 0 - x) (eval env e)
  ...
```

Booléens : langage

On rajoute deux opérations : `and` et `or` :

```
type expr =
  ...
  | And of expr * expr
  | Or of expr * expr
```

Booléens : évaluation de `and` et `or`

Comment doivent se comporter `and` et `or` ?

Version “typage fort” :

- deux booléens en arguments
- renvoie un booléen
- une erreur de type sinon

Booléens : évaluation de and et or

```
let rec eval (env : env) (e: expr) : value =
  match e with
  ...
  | And (e1, e2) ->
    begin match (eval env e1, eval env e2) with
      | VBool b1, VBool b2 -> VBool (b1 && b2)
      | _ -> raise TypeError
    end
```

Booléens paresseux

Pour `e1 and e2`, si `e1` est faux, ce n'est pas nécessaire d'évaluer `e2` pour connaître le résultat.

C'est la version *paresseuse* (choix fait par la plupart des langages).

Cela n'influence pas le résultat dans un langage pur.

Booléens paresseux

```
let rec eval (env : env) (e: expr) : value =
  match e with
  ...
  | And (e1, e2) ->
    begin match eval env e1 with
      | VBool true ->
        begin match eval env e2 with
        | VBool b -> VBool b
        | _ -> raise TypeError
        end
      | VBool false -> VBool false
      | _ -> raise TypeError
    end
```

Booléens paresseux

```
let rec eval (env : env) (e: expr) : value =
  match e with
  ...
  | Or (e1, e2) -> begin match eval env e1 with
      | VBool true -> VBool true
      | VBool false -> begin match eval env e2 with
          | VBool b -> VBool b
          | _ -> raise TypeError
        end
      | _ -> raise TypeError
    end
```

Version formelle

$$
\begin{aligned}
\mathit{eval}(n, \rho) &= \mathbf{int}(n) \text{ si } n \text{ est un nombre} \\
\mathit{eval}(b, \rho) &= \mathbf{bool}(b) \text{ si } b \text{ est un booléen} \\
\mathit{eval}(x, \rho) &= \rho(v) \text{ si } x \text{ est une variable} \\
\mathit{eval}(e_1 + e_2, \rho) &= \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho) \\
&\ldots \\
\mathit{eval}(e_1 \&\& e_2, \rho) &= \mathit{eval}(e_2, \rho) \text{ si } \mathit{eval}(e_1, \rho) = \mathbf{bool}(\texttt{true}) \\
\mathit{eval}(e_1 \&\& e_2, \rho) &= \mathbf{bool}(\texttt{false}) \text{ si } \mathit{eval}(e_1, \rho) = \mathbf{bool}(\texttt{false}) \\
\mathit{eval}(e_1 || e_2, \rho) &= \mathit{eval}(e_2, \rho) \text{ si } \mathit{eval}(e_1, \rho) = \mathbf{bool}(\texttt{false}) \\
\mathit{eval}(e_1 || e_2, \rho) &= \mathbf{bool}(\texttt{true}) \text{ si } \mathit{eval}(e_1, \rho) = \mathbf{bool}(\texttt{true})
\end{aligned}
$$

Conditions

Conditions

On souhaite maintenant ajouter des conditions

if **true** **then** 1 **else** 2

Conditions : langage

```
type expr =
  ...
  | If of expr * expr * expr
```

Conditions : sémantique

Pour évaluer :

if `e1` **then** `e2` **else** `e3`

- on évalue `e1`
- si `e1` est considéré comme vrai, on évalue `e2`
- sinon, on évalue `e3`

Que signifie *vrai* ?

- un booléen égal à `true` (choix typé fort)
- ou une valeur considéré *truthy*, par exemple en JavaScript

```
if ("") { 1 } else { 2 }
```

Conditions : sémantique

Pour évaluer :

if `e1` **then** `e2` **else** `e3`

- on évalue `e1`
- si `e1` est considéré comme vrai, on évalue `e2`
- sinon, on évalue `e3`

Que signifie *vrai* ?

- un booléen égal à `true` (choix typé fort)
- ou une valeur considéré *truthy*, par exemple en JavaScript

  if `("")` `{ 1 }` **else** `{ 2 }`

  $\rightarrow$ 2

  if `([])` `{ 1 }` **else** `{ 2 }`

Conditions : sémantique

Pour évaluer :

if `e1` **then** `e2` **else** `e3`

- on évalue `e1`
- si `e1` est considéré comme vrai, on évalue `e2`
- sinon, on évalue `e3`

Que signifie *vrai* ?

- un booléen égal à `true` (choix typé fort)
- ou une valeur considéré *truthy*, par exemple en JavaScript

  if `("")` `{ 1 }` **else** `{ 2 }`

  $\rightarrow 2$

  if `([])` `{ 1 }` **else** `{ 2 }`

  $\rightarrow 1$

Conditions : eval

```
let rec eval (env : env) (e : expr) : value =
  match e with
  ...
  | If (condition, consequent, alternative) ->
    begin match eval env condition with
    | VBool true -> eval env consequent
    | VBool false -> eval env alternative
    | _ -> raise TypeError
    end
```

Version formelle

$$
\begin{aligned}
\mathit{eval}(n, \rho) &= \mathbf{int}(n) \text{ si } n \text{ est un nombre} \\
\mathit{eval}(b, \rho) &= \mathbf{bool}(b) \text{ si } b \text{ est un booléen} \\
\mathit{eval}(x, \rho) &= \rho(v) \text{ si } x \text{ est une variable} \\
\mathit{eval}(e_1 + e_2, \rho) &= \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho) \\
&\ldots \\
\mathit{eval}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \rho) &= \mathit{eval}(e_2, \rho) \text{ si } \mathit{eval}(e_1, \rho) = \mathbf{bool}(\texttt{true}) \\
\mathit{eval}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, \rho) &= \mathit{eval}(e_3, \rho) \text{ si } \mathit{eval}(e_1, \rho) = \mathbf{bool}(\texttt{false})
\end{aligned}
$$

Type produit (paires)

Type produit (paires)

On peut également ajouter des paires, comme représentant des *types produits* :

```
type value =
  ...
  | VPair of value * value
```

Paires : langage

Il nous faut :

- une expression pour construire une paire, `(e1, e2)`
- une expression pour extraire la première valeur d'une paire, `fst e`
- une expression pour extraire la seconde valeur d'une paire, `snd e`

```
type expr =
  ...
  | Pair of expr * expr
  | First of expr
  | Second of expr
```

Paires : évaluation

```
let rec eval (env : env) (e : expr) : value =
  ...
  | Pair (e1, e2) ->
    VPair (eval env e1, eval env e2)
  | First e ->
    begin match eval env e with
      | VPair (first, _) -> first
      | _ -> raise TypeError
    end
  | Second e ->
    begin match eval env e with
      | VPair (_, second) -> second
      | _ -> raise TypeError
    end
```

Version formelle

$$
\begin{aligned}
\mathit{eval}(n, \rho) &= \mathbf{int}(n) \text{ si } n \text{ est un nombre} \\
\mathit{eval}(b, \rho) &= \mathbf{bool}(b) \text{ si } b \text{ est un booléen} \\
\mathit{eval}(x, \rho) &= \rho(v) \text{ si } x \text{ est une variable} \\
\mathit{eval}(e_1 + e_2, \rho) &= \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho) \\
&\ldots \\
\mathit{eval}((e_1, e_2), \rho) &= \mathbf{pair}(\mathit{eval}(e_1, \rho), \mathit{eval}(e_2, \rho)) \\
\mathit{eval}(\texttt{fst}\ \ e, \rho) &= v_1 \text{ si } \mathit{eval}(e, \rho) = \mathbf{pair}(v_1, v_2) \\
\mathit{eval}(\texttt{snd}\ \ e, \rho) &= v_2 \text{ si } \mathit{eval}(e, \rho) = \mathbf{pair}(v_1, v_2)
\end{aligned}
$$

n-uplets

On pourrait rajouter les *n-uplets* `(e1, ..., en)`, mais on ne gagnerait pas en expressivité

Comment représenter `(1, 2, 3, 4)` seulement avec des paires ?

n-uplets

On pourrait rajouter les *n-uplets* `(e1, ..., en)`, mais on ne gagnerait pas en expressivité

Comment représenter `(1, 2, 3, 4)` seulement avec des paires ?

$\rightarrow$ `(1, (2, (3, 4)))`

Type somme

Type somme

On va définir un type de valeur spécifique pour représenter les valeurs de type *somme*.

Cela permet de représenter nos types algébriques.

Type somme à deux éléments

On ne va permettre que de distinguer entre deux éléments, mais c'est assez général pour tout représenter.

Par exemple, comment encoder le type suivant en utilisant que des sommes à deux éléments ?

```
type S = A | B | C | D
```

Type somme à deux éléments

On ne va permettre que de distinguer entre deux éléments, mais c'est assez général pour tout représenter.

Par exemple, comment encoder le type suivant en utilisant que des sommes à deux éléments ?

```
type S = A | B | C | D
```

```
type S = A | S2
type S2 = B | S3
type S3 = C | D
```

Type somme : valeurs

```
type value =
  | VLeft of value
  | VRight of value
```

Type somme : langage

On a besoin de plusieurs expressions :

- créer une valeur de type *gauche* : `left e`
- vérifier si on a une valeur de type *gauche* : `is-left e`
- extraire une valeur de type *gauche* : `extract-left e`
- les 3 même pour le type *droit* : `right e`, `is-right e`, `extract-right e`

Type somme : langage

```
type expr =
  ...
  | Left of expr
  | IsLeft of expr
  | ExtractLeft of expr
  | Right of expr
  | IsRight of expr
  | ExtractRight of expr
```

Type somme : évaluation

```
let rec eval (env : env) (e : expr) : value =
  ...
  | Left e ->
    VLeft (eval env e)
  | IsLeft e ->
    begin match eval env e with
      | VLeft v -> VBool true
      | _ -> VBool false
    end
  | ExtractLeft e ->
    begin match eval env e with
      | VLeft v -> v
      | _ -> raise TypeError
    end
```

Type somme : évaluation

```
let rec eval (env : env) (e : expr) : value =
  ...
  | Right e ->
    VRight (eval env e)
  | IsRight e ->
    begin match eval env e with
      | VRight v -> VBool true
      | _ -> VBool false
    end
  | ExtractRight e ->
    begin match eval env e with
      | VRight v -> v
      | _ -> raise TypeError
    end
```

Version formelle

$$
\begin{aligned}
\mathit{eval}(n, \rho) &= \mathbf{int}(n) \text{ si } n \text{ est un nombre} \\
\mathit{eval}(b, \rho) &= \mathbf{bool}(b) \text{ si } b \text{ est un booléen} \\
\mathit{eval}(x, \rho) &= \rho(v) \text{ si } x \text{ est une variable} \\
\mathit{eval}(e_1 + e_2, \rho) &= \mathit{eval}(e_1, \rho) + \mathit{eval}(e_2, \rho) \\
&\ldots \\
\mathit{eval}(\texttt{right}(e), \rho) &= \mathbf{right}(\mathit{eval}(e, \rho)) \\
\mathit{eval}(\texttt{left}(e), \rho) &= \mathbf{left}(\mathit{eval}(e, \rho)) \\
\mathit{eval}(\texttt{is-right}(e), \rho) &= \mathbf{bool}(\texttt{true}) \text{ si } \mathit{eval}(e, \rho) = \mathbf{right}(v) \\
\mathit{eval}(\texttt{is-left}(e)\rho) &= \mathbf{bool}(\texttt{true}) \text{ si } \mathit{eval}(e, \rho) = \mathbf{left}(v) \\
\mathit{eval}(\texttt{extract-left}(e), \rho) &= v \text{ si } \mathit{eval}(e, \rho) = \mathbf{left}(v) \\
\mathit{eval}(\texttt{extract-right}(e), \rho) &= v \text{ si } \mathit{eval}(e, \rho) = \mathbf{right}(v)
\end{aligned}
$$

Types algébriques

Types algébriques

Si on souhaite décrire des types algébriques de toute forme, on a déjà tout ce qu'il faut.

Exemple : les listes

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

C'est un type somme contenant un type produit.

Listes

Une façon d'encoder les listes dans notre langage est la suivante :

- la liste vide est `left #f`
- la vérification de si une liste `l` est vide se fait avec `is-left l`
- la liste non vide est `right (head, tail)`
- la vérification de si une liste `l` est non vide se fait avec `is-right l`
- l'extraction de la tête d'une liste `l` se fait par `first (extract-right l)`
- l'extraction de la queue d'une liste `l` se fait par `second (extract-right l)`

Types algébriques

On peut généraliser le même raisonnement pour tout type algébrique :

- un type somme s'encode avec `left` et `right`
- un type produit s'encode avec une paire

Dans un langage réel, on souhaite donner des constructions plus pratiques à l'utilisateur.

Fonctions

Fonctions : langage

Deux constructions cruciales :

- construction de fonction anonyme : `function x -> x + 1`
  - une fonction a une un et un corps
    - une fonction à plusieurs arguments se construit avec plusieurs fonctions : `function x -> function y -> x + y`
  - si on souhaite la nommer, on pourra utiliser `let` : `let f = function x y -> x + y`
    - la construction `let f x y = x + y` en OCaml est du sucre syntaxique
- appel de fonction : `f 1`
  - une expression qui devra évaluer à la fonction a appeler
  - une expression pour l'argument

Fonctions : langage

```
type expr =
  ...
  | Function of string * expr
  | Call of expr * expr
```

Fonctions : valeurs

Quelle est la valeur d'une fonction ?

- on a besoin de son expression
- mais également de son *environnement*
  - ça nous permet de capturer les valeurs des liaisons utilisées

C'est ce qu'on appelle *fermeture* (*closure*) : une expression et un environnement

function `x -> (`**function** `y -> x + y)`

- `function y -> x + y` **capture** la liaison `x`

Fonctions : valeurs

```
type value =
  ...
  | VClosure of string list * expr * env
```

Problème :

- `value` dépend de `expr` et `env`
- `expr` dépend de `value`
- `env` dépend de value

Fonctions : valeurs

Il faut donc introduire des déclarations de types mutuellement récursives :

```
type value =
  ...
  | VClosure of string list * expr * env
and env = (string * value) list
```

Fonctions : évaluation

- Création de fonction : facile, on crée juste la valeur
- Application :
  - on évalue la fonction
    - si ce n'est pas une fonction, on lève une `TypeError`
  - on évalue les arguments
  - on étend l'environnement de définition avec les valeurs des arguments

Important :

- **paramètre** : ou paramètre formel, un nom, par exemple `x` dans `function x -> ...`
- **argument** : une expression utilisée lors de l'appel de fonction, par exemple `1` dans `f 1`

Fonctions : eval

```
let rec eval (env : env) (e : expr) : value =
  match e with
  ...
  | Function (param, body) ->
    VClosure (param, body, env)
  | Call (f, arg) ->
    begin match eval env f with
      | VClosure (param, body, def_env)  ->
        let arg_value = eval env arg in
        let new_env = extend def_env param arg_value in
        eval new_env body
      | _ -> raise TypeError
    end
```

Fonctions : exemple

```
# eval [] (Call (Function ("x", Plus (Var "x", Int 1)),
                 Int 2));;
- : value = Int 3
```

Fonctions récursives

Fonctions récursives

Comment exprimer une fonction récursive ?

```
let f = function x -> f x in f 2
```

Essayons :

```
# eval [] (Let ("f", Function (["x"], Call (Var "f", [Var "x"])),
                Call (Var "f",  [Int 2])));;
Exception: Not_found
```

Quel est le problème ?

→ Exception levée par `lookup` : `f` n'est pas lié dans `f`

Fonctions récursives : langage

On défini un `let` récursif spécifiquement pour les fonctions :

```
...
and expr =
  ...
  | LetRec of string * string * expr * expr
```

Fonctions récursives : valeurs

On ne sait pas représenter les fonctions récursives comme valeurs avec `VClosure`.

La fonction doit être liée dans sa définition, donc elle doit être dans l'environnement de la fermeture, donc :

- l'environnement doit être contenu dans la fermeture
- la fermeture doit être contenue dans l'environnement

On a donc une valeur cyclique qu'on ne peut pas exprimer facilement : il faudrait créer l'environnement *et* la fermeture en même temps

Fonctions récursives : valeurs

Plusieurs possibilités d'implémentation, ici on fait le choix suivant :

- la fermeture a un nom optionnel
- la présence d'un nom indique que c'est une fermeture récursive
- l'environnement est inchangé
- on ajoutera la liaison dans l'environnement lors de l'appel de fonction

```
type value =
  ...
  | VClosure of string option * string * expr * env
```

Fonctions récursives : évaluation

```
let rec eval (env : env) (e : expr) : value =
  match e with
  ...
  | LetRec (f, param, body, e2) ->
    let v = VClosure (Some f, param, body, env) in
    eval (extend env f v) e2
  | Function (param, body) ->
    VClosure (None, param, body, env)
```

Fonctions récursives : évaluation

```
let rec eval (env : env) (e : expr) : value =
  match e with
  ...
  | Call (f, arg) ->
    begin match eval env f with
      | VClosure (name, param, body, def_env) as f ->
        let rec_def_env = match name with
          | None -> def_env
          | Some n -> extend def_env n f in
        let arg_value = eval env arg in
        let new_env = extend rec_def_env param arg_value in
        eval new_env body
      | _ -> raise TypeError
    end
```

Fonctions récursives : exemple

```
# eval [] (LetRec ("f", "x", Call (Var "f", Constant (Int 1)),
                   Call (Var "f",  Constant (Int 2))));;
^CInterrupted.
```

Boucle infinie, comme attendu.

Fonctions récursives sans `let rec`

On aurait pu s'en sortir sans les fonctions récursives, avec le *combinateur Y* :

```
let y = function f -> (function x -> f (x x)) (function x -> f (x x))
```

Par exemple :

```
let g = function f -> function x -> f x
```

Notre boucle infinie se construirait avec `y g 1`

(Cela ne marche pas dans un langage statiquement typé.)

Éléments de notre interpréteur

On a donc un langage avec :

- nombres et expressions mathématiques
- liaisons
- booléens et expressions logiques
- type produit
- type somme
- conditions
- fonctions
- fonctions récursives

Encodage de Church

On peut en fait tout faire en utilisant les fonctions (récursives).

Par exemple,

```
let true_ = fun x y -> x
let false_ = fun x y -> y
let if_ = fun cond cons alt -> cond cons alt

if_ true_ 1 2 (* évalue à 1 *)
if_ false_ 1 2 (* évalue à 2 *)
```

C'est ce qu'on appelle l'encodage de Church pour les booléens. On peut faire de même pour les autres types qu'on a utilisé (nombres, sommes, produits).

λ calcul

Le **lambda calcul** est un système formel basé sur le concept de fonction.

Syntaxe :

$$
\begin{array}{rll}
M, N \in \mathit{Term} ::= & x & \text{variable} \\
& \mid \lambda x.M & \text{abstraction} \\
& \mid M\ N & \text{application}
\end{array}
$$

Sémantique sous forme de règles de réductions :

$$
\begin{array}{ll}
\lambda x.M[x] \rightarrow \lambda y.M[y] & \alpha\text{-conversion} \\
(\lambda x.M)N \rightarrow M[x := N] & \beta\text{-réduction}
\end{array}
$$

Il existe d'autres formulations, y compris des sémantiques opérationnelles plus proches de notre interpréteur.

Caractéristiques d'OCaml (rappel)

- **Fondation**: lambda calcul
- **Modèle d'éxécution**: interprété et compilé
- **Paradigme**: fonctionnel / multi-paradigme
- **Typage**: statique fort avec inférence
- **Modèle d'évaluation**: strict
  - Comme la majorité des langages fonctionnels (mais pas Haskell)

Concepts importants

- **Interpréteur** : outil qui lit un programme et l'exécute
- **Évaluation** : c'est l'interprétation d'une expression d'un langage
- **Typage fort** : on préfère lever une exception plutôt que de mélanger des types ou faire des conversions implicites
- **Environnement** : une association de noms à des valeurs
- **Fermeture** : une déclaration de fonction associée à son environnement de définition
- **Paramètre** : ou paramètre formel, un nom, par exemple `x` dans `fun x -> ...`
- **Argument** : une expression utilisée lors de l'appel de fonction, par exemple `1` dans `f 1`