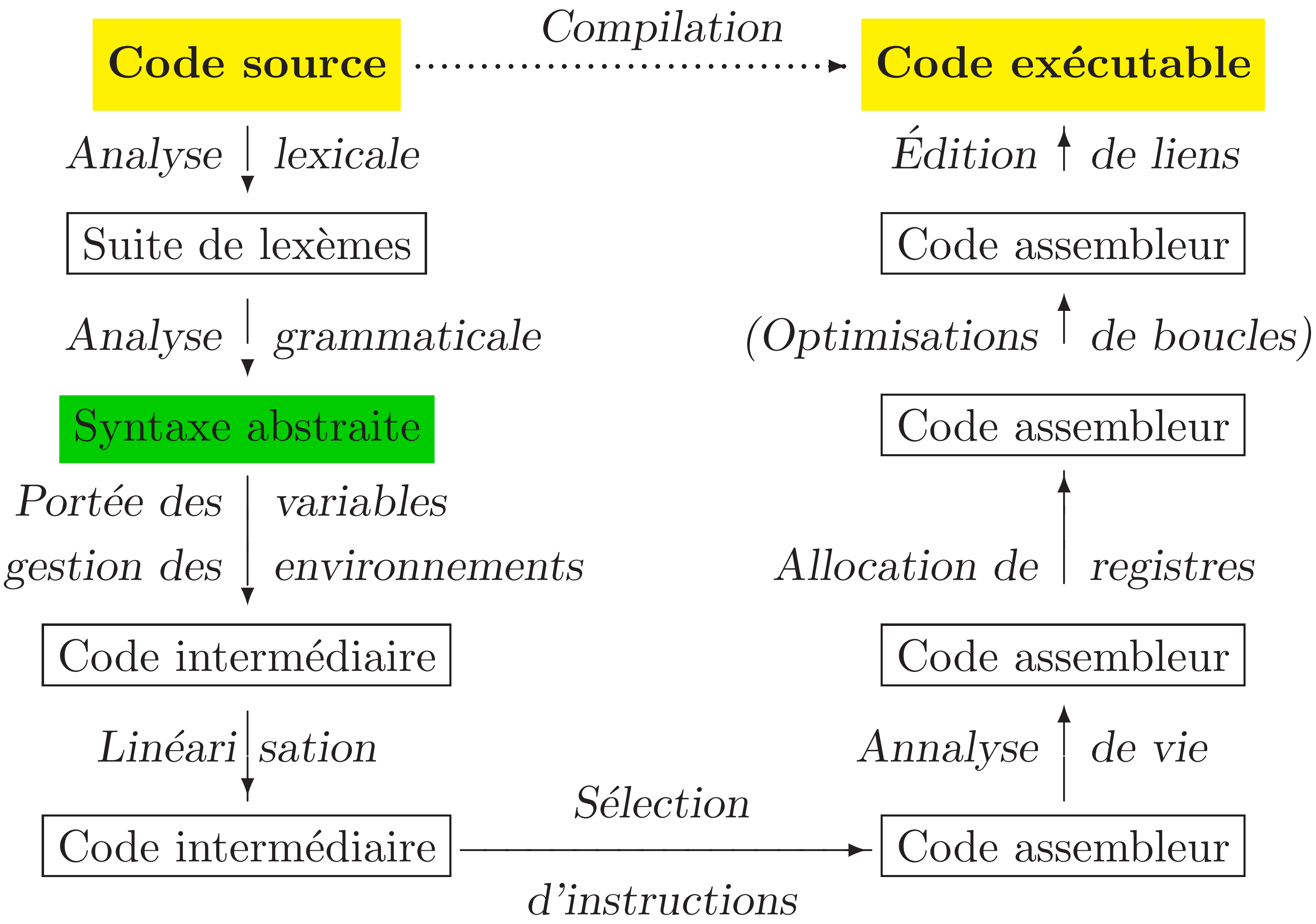
Langages de programmation
Sémantiques opérationnelles
Interpréteurs
Le langage Pseudo-Pascal

Luc Maranget

`http://pauillac.inria.fr/~maranget`

2005


Code source
Compilation
Code exécutable
Analyse lexicale
Suite de lexèmes
Analyse grammaticale
Syntaxe abstraite
Portée des variables
gestion des environnements
Code intermédiaire
Linéarisation
Code intermédiaire
Sélection
d'instructions
Code assembleur
Annalyse de vie
Code assembleur
Allocation de registres
Code assembleur
(Optimisations de boucles)
Code assembleur
Édition de liens

Langages de programmation

Langages généraux Ils doivent être complets, *i.e.* permettre d'exprimer tous les algorithmes calculables. Au minimum, il faut une mémoire infinie (grande en pratique) et la possibilité d'exprimer la récursion (construction primitive ou boucle `while`). Ex : Fortran, Pascal, C, Ocaml, etc.

Langages spécialisés (DSL pour *Domain Specific Languages*) Langages pour le graphisme, pour commander des robots, la calculette. Ils peuvent ne pas être complets.

Expressivité Les langages généraux ne sont pas tous équivalents. L'expressivité est la capacité d'exprimer des algorithmes succinctement (et directement).

Exemples de constructions expressives

Les structures de données (et filtrage)

Les modules, les objets

Le typage Restreint l'expressivité au profit de la sécurité. (Comparer Scheme et ML).

L'expressivité du système de typage est aussi importante (comparer ML et Pascal ou Java).

Les fonctions

- Est-ce que les fonctions peuvent être locales ?
- Est-ce que les fonctions sont des valeurs ?

Syntaxe concrète – syntaxe abstraite

La syntaxe concrète Support concret du discours : une suite de caractères. Une grammaire dit comment les lettres forment des mots et des phrases.

- Plusieurs notations sont possibles pour une même construction. Exemple : les caractères `'a'` et `'\097'`, ou `let x = ... in x+1` et `x+1 where x=...`.
- La syntaxe concrète est linéaire et utilise des parenthèses.

La syntaxe abstraite est une représentation arborescente qui fait abstraction de la notation (on parle d'arbre de syntaxe abstraite). Elle définit les constructions du langage.

L'analyse *lexicale* traduit une suite de caractères en suite de mots. L'analyse *grammaticale* lit une suite de mots, reconnaît les phrases du langages et retourne un arbre de syntaxe abstraite.

Exemple simple : La calculette

Syntaxe concrète (dans le style BNF)

$$
\begin{array}{lcl}
\textbf{expression} & ::= & \mathrm{ENTIER} \\
 & | & \mathrm{IDENTIFICATEUR} \\
 & | & \textbf{expression binop expression} \\
 & | & \text{"("}\ \textbf{expression}\ \text{")"} \\
\textbf{binop} & ::= & \text{"+"} \mid \text{"}-\text{"} \mid \text{"}*\text{"} \mid \text{"/"}
\end{array}
$$

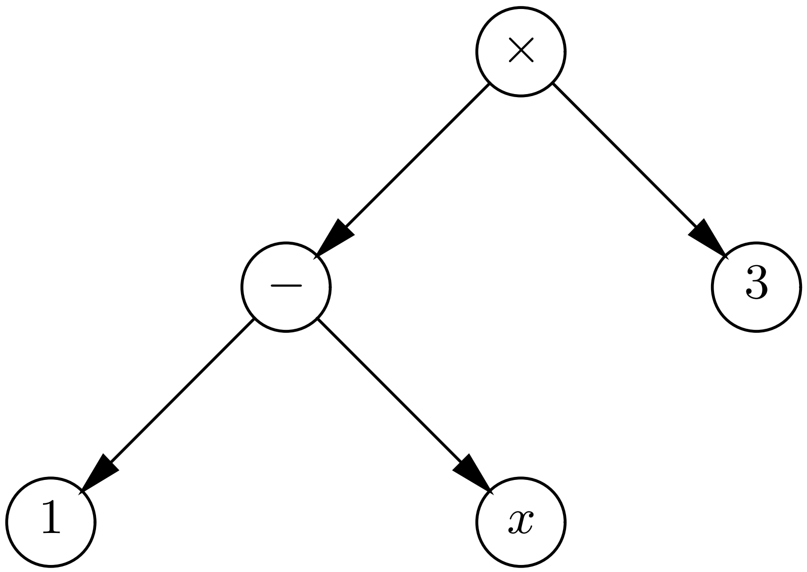
Syntaxe abstraite

```
type expression =
  | Const of int
  | Variable of string
  | Bin of binop * expression * expression
and binop = Plus | Moins | Mult | Div
```

Exemple les expressions « `(1 - x) * 3` » et « `(1-x)*(3)` » ont pour représentation

```
Bin (Mult,
  Bin (Moins, Const 1, Variable "x"), Const 3)
```

Ou plutôt :



Essentiellement : les parenthèses disparaissent.

Sémantique des programmes

Il s'agit de donner un sens aux programmes.

Sémantique informelle L'instruction `while` est de la forme :

`while (`*expression*`)` *instruction*

la sous-instruction est exécutée de manière répété tant que la valeur de l'expression *reste* non nulle. On teste l'expression avant d'exécuter l'instruction.

Sémantique dénotationnelle On associe un objet mathématique (abstrait) à chaque expression. Par exemple, on construit un modèle mathématique des entiers, muni des opérations sur les entiers. C'est beaucoup plus dur pour les fonctions (calculables).

Sémantique opérationnelle On définit un ensemble de valeurs (ou résultats) puis une relation d'évaluation qui relie des programmes avec des résultats.

Avantages et inconvénients

Sémantique informelle Facile à comprendre, synthétique, mais parfois imprécise. Impossible de faire des preuves.

Sémantique dénotationnelle La vérité selon les maths. Très complexe, difficile à exploiter.

Sémantique opérationnelle Plus facile à manipuler, c'est aussi celle qui nous intéresse le plus souvent (pour calculer). Ce n'est pas tout à fait la réalité mathématique.

Note :

- Les racines de la sémantique formelle viennent souvent de la logique formelle.
- On peut exprimer la sémantique opérationnelle en écrivant un interpréteur (on construit alors sur la sémantique du langage d'implémentation).

Interpréteur pour la calculette

```
type valeur = int

let cherche x env = List.assoc x env

let rec evalue env = function
  | Const n -> n
  | Variable x -> cherche x env
  | Bin (op, e1, e2) ->
      let v1 = evalue env e1 and v2 = evalue env e2 in
      begin match op with
       | Plus -> v1 + v2 | Moins -> v1 - v2
       | Mult -> v1 * v2 | Div -> v1 / v2
      end
```

Une présentation plus neutre (S.O.S.)

(**S**émantique **O**pérationnelle **S**tructurelle)

Une autre présentation plus mathématique et régulière consiste à définir une relation $\rho \vdash e \Rightarrow v$ qui se lit *"Dans l'environnement ρ, l'expression e s'évalue en la valeur v"* par des règles d'inférence.

Une règle d'inférence est une implication $P_1 \wedge P_2 \wedge \ldots \wedge P_k \Longrightarrow C$ présentée sous la forme

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_k}{C}$$

que l'on peut lire pour réaliser (évaluer) C il faut réaliser à la fois P_1 et $\ldots$ P_k.

Sémantique des expressions arithmétiques

Les jugements de la forme $\rho \vdash e \Rightarrow v$

- ρ lie des variables x à des valeurs v. Autrement dit, c'est un ensemble de paires notées $x \mapsto v$.
- $e \in \mathit{expressions}$ et $v \in \mathbb{Z}$. On note $\bar{n}$ l'entier relatif associé à sa représentation n.

$$\rho \vdash \texttt{Const}\ n \Rightarrow \bar{n} \qquad\qquad \frac{x \in \mathit{dom}\,(\rho)}{\rho \vdash \texttt{Variable}\ x \Rightarrow \rho(x)}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Plus}\,, e_1, e_2) \Rightarrow v_1 + v_2}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Times}\,, e_1, e_2) \Rightarrow v_1 * v_2}$$

Une construction de liaison

On étend la calculette avec des liaisons « `let` ».

| Let of string * expression * expression

avec, par exemple, la syntaxe concrète

"let" VARIABLE *"="* **expression** *"in"* **expression**

L'expression `Let` (x, e_1, e_2) lie la variable x à l'expression e_1 dans l'évaluation de l'expression e_2.

Formellement :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho, x \mapsto v_1 \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2}$$

où $\rho, x \mapsto v$ ajoute la liaison de x à v dans l'environnement ρ en cachant une ancienne liaison éventuelle de x.

Modification de l'interprète

```
let ajoute x v env = (x,v)::env

let rec evalue env = function
   ...
 | Let (x, e1, e2) ->
     let v1 = evalue env e1 in
     evalue (ajoute x v1 env) e2
```

Noter le codage de $\rho, x \mapsto v$ par $(x, v) :: \rho$.

Exercise 1 *Soit* **let** $x = 1$ **in** ($\textbf{let}\ x = 2\ \textbf{in}\ x) + x$*,*

- *Donner la syntaxe abstraite,*
- *le résultat de l'évaluation,*
- *la dérivation de l'évaluation de l'expression la plus interne.*

□

Arbre de dérivation

L'ensemble des réponses (de l'exercice précédent) est contenu dans l'arbre de dérivation, qui constitue une preuve de l'évaluation de l'expression en la valeur 3

$$\dfrac{\emptyset \vdash \texttt{Const } 1 \Rightarrow 1 \qquad \dfrac{\dfrac{x \mapsto 1 \vdash \texttt{Const } 2 \Rightarrow 2 \qquad x \mapsto 1, x \mapsto 2 \vdash x \Rightarrow 2}{x \mapsto 1 \vdash \texttt{Let}\,(x, \texttt{Const } 2, x) \Rightarrow 2} \qquad x \mapsto 1 \vdash x \Rightarrow 1}{x \mapsto 1 \vdash \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Let}\,(x, 2, x), x) \Rightarrow 3}}{\boxed{\emptyset \vdash \texttt{Let}\,(x, \texttt{Const } 1, \texttt{Bin}\,(\texttt{Plus}\,, \texttt{Let}\,(x, 2, x), x)) \Rightarrow 3}}$$

Formalisation des erreurs

L'évaluation peut mal se passer, même dans la calculette, division par 0, accès à une variable inconnue, etc.

La sémantique opérationnelle peut formaliser les erreurs ou pas.

On remplace la relation $\rho \vdash e \Rightarrow v$ par une relation $\rho \vdash e \Rightarrow r$ où r est une réponse. Les réponses sont l'union des valeurs v ou des erreurs z.

$$\frac{\begin{array}{c}\rho \vdash e_1 \Rightarrow v_1 \\ \rho \vdash e_2 \Rightarrow v_2 \qquad v_2 \neq 0\end{array}}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow v_1/v_2} \qquad \frac{\rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow \texttt{Division}}$$

Le type des erreurs (par exemple) :

type erreur = Division_par_zéro | Variable_libre **of** string

Implémentation des erreurs

Formellement, on devrait utiliser un type somme

```
type résultat = Valeur of valeur | Erreur of erreur

let rec evalue env = function
    ...
  | Bin (Div, e1, e2) -> begin match  evalue env e2 with
      | Erreur _ as e -> e
      | Valeur 0  -> Erreur Division_par_zéro
      | Valeur v2 ->
        begin match evalue env e1 with
        | Erreur _ as e  -> e
        | Valeur v1      -> Valeur (v1 / v2)
        end end
```

En pratique, on utilise les exceptions du langage hôte

```
exception Erreur of erreur
let erreur x = raise (Erreur x)
let cherche x l =
  try List.assoc x l
  with Not_found -> erreur (Variable_libre x)

let rec evalue env = function
    ...
  | Bin (Div, e1, e2) ->
      let v2 = evalue env e2 in
      if v2 = 0 then
        erreur (Division_par_zéro)
      else
        let v1 = evalue env e1 and v2 = evalue env e2 in
        v1 / v2
```

Terminaison

L'évaluation peut ne pas terminer.

La S.O.S (à grands pas) ne permet ne modélise pas la terminaison, un programme qui ne termine pas ne peut pas être mis en relation avec une réponse.

Sémantique à réduction à petits pas

Dans le cours *Langage et programmation* on définit une sémantique opérationnelle à réduction "à petits pas" qui permet quand même de décrire le calcul des programmes qui ne terminent pas : le calcul est modélisé par une relation de réduction interne, i.e. les programmes se réduisent sur eux mêmes (chaque étape élémentaire du calcul est modélisé par un petit pas de réduction) jusqu'à plus soif ou indéfiniment.

Constructions impératives

Les constructions impératives (variables « *mutables* », entrées sorties, etc.) sont exécutées pour leur effet.

Les expressions sont évaluées pour leur valeur.

Il est commode (mais pas obligatoire, cf. Caml) de distinguer *instructions* et *expressions*.

```
type instruction =
| Affecte of string * expression
| Sequence of instruction * instruction
```

Avec pour syntaxe concrète :

```
instruction ::= VARIABLE ":=" expression
              | instruction ";" instruction
```

Sémantique des instructions

La mémoire σ associe les adresses ℓ aux valeurs v,
l'environnement ρ associe les variables x aux adresses.

L'exécution d'une instruction i est rendue par un jugement $\rho/\sigma \vdash i \Rightarrow /\sigma'$ qui se lit *dans l'état-mémoire σ et l'environnement ρ, l'exécution de l'instruction i produit un nouvel état mémoire σ'.*

$$\frac{\rho/\sigma \vdash i_1 \Rightarrow /\sigma_1 \qquad \rho/\sigma_1 \vdash i_2 \Rightarrow /\sigma_2}{\rho/\sigma \vdash \texttt{Seq}\,(i_1, i_2) \Rightarrow /\sigma_2}$$

$$\frac{x \in dom\,(\rho) \qquad \rho(x) \in dom\,(\sigma) \qquad \rho/\sigma \vdash e \Rightarrow v}{\rho/\sigma \vdash \texttt{Affecte}\,(x, e) \Rightarrow /\sigma, \rho(x) \mapsto v}$$

$$\frac{x \in dom\,(\rho) \qquad \rho(x) \in dom\,(\sigma)}{\rho/\sigma \vdash \texttt{Variable}\ x \Rightarrow \sigma(\rho(x))}$$

Interprétation des instructions

```
(* Les adresses sont les adresses de Caml *)
type environnement = (string * valeur ref) list

let rec evalue env = function
    ...
  | Variable x -> !(cherche x env)

and execute env = function
| Sequence (i1, i2) -> execute env i1 ; execute env i2
| Affecte (x, e)    ->
    let v = evalue env e in
    let cell = cherche env x in
    cell := v
```

Les booléens, la conditionnelle

Les booléens sont *true* et *false*. Le domaine des valeurs est alors la réunion de l'ensemble des entiers et de celui des booléens. En toute rigueur les types doivent apparaître dans les règles de SOS :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Bin}\,(\texttt{Plus}\,, e_1, e_2) \Rightarrow \leftarrow_{\text{int}}(\rightarrow_{\text{int}}(v_1) + \rightarrow_{\text{int}}(v_2))}$$

C'est assez lourd, on s'en passe.

$$\frac{\rho \vdash e_1 \Rightarrow true \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \texttt{If}\,(e_1, e_2, e_3) \Rightarrow v} \qquad \frac{\rho \vdash e_1 \Rightarrow false \qquad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \texttt{If}\,(e_1, e_2, e_3) \Rightarrow v}$$

Interpréteur

Mais en Caml, c'est l'inverse : on distingue facilement valeurs entières et booléennes.

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

type valeur = Int of int | Bool of bool

let rec evalue env = function
| Const n -> Int n
        ⋮
| If (e1, e2, e3) ->
    match evalue env e1 with
    | Bool true  -> evalue env e2
    | Bool false -> evalue env e3
```

Les opérations sur les booléens

Deux sémantiques

- Paresseuse (je préfère séquentielle) :

$$\frac{\rho \vdash e_1 \Rightarrow \mathit{false}}{\rho \vdash \mathtt{Bin}\,(\mathtt{And}\,, e_1, e_2) \Rightarrow \mathit{false}} \qquad \frac{\rho \vdash e_1 \Rightarrow \mathit{true} \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \mathtt{Bin}\,(\mathtt{And}\,, e_1, e_2) \Rightarrow v_2}$$

- Stricte (en Pascal seulement) :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow v_2}{\rho \vdash \mathtt{Bin}\,(\mathtt{And}\,, e_1, e_2) \Rightarrow v_1 \wedge v_2}$$

Laquelle semble préférable ?

Sur l'ordre d'évaluation

La SOS semble parfois ne rien en dire. Mais…

- Dépendances.

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho, x \mapsto v_1 \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2}$$

$$\frac{\rho/\sigma \vdash i_1 \Rightarrow /\sigma_1 \qquad \rho/\sigma_1 \vdash i_2 \Rightarrow /\sigma_2}{\rho/\sigma \vdash \texttt{Seq}\,(i_1, i_2) \Rightarrow /\sigma_2}$$

- Erreurs.

$$\frac{\rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow \texttt{Division}}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow \texttt{Division}}$$

Doit-on spécifier l'ordre d'évaluation ?

Oui (Java, Pseudo-Pascal) La sémantique est non-ambigüe.

Non (C, Caml) L'auteur du compilateur peut optimiser.

En pratique de programmation

Éviter :

```
x = 10 * (getchar() - '0') + getchar() - '0' ;
```

Adopter :

```
c1 = getchar() - '0' ;
c2 = getchar() - '0' ;
x = 10 * c1 + c2 ;
```

Les tableaux

Pour simplifier : tableaux alloués explicitement.

On ajoute trois constructions à la syntaxe :

- L'expression `Alloc` (e_1) alloue un nouveau tableau de taille la valeur de e_1 les cases du nouveau tableau sont indéfinies
- L'expression `Lire` (e_1, e_2) lit la case (correspondant à la valeur de) e_2 du tableau (la valeur de) e_1.
- L'instruction `Ecrire` (e_1, e_2, e_3) écrit la case e_2 du tableau e_1 avec la valeur de e_3.

Construction proche des tableaux de Java (et Caml), quelle différence ?

Sémantique des tableaux

Allocation

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow k \qquad k \geq 0 \qquad\qquad t = (0 \mapsto \ell_0, \ldots, k-1 \mapsto \ell_{k-1}) \qquad \ell_0 \notin dom\,(\sigma) \ldots\ \ell_{k-1} \notin dom\,(\sigma)}{\rho/\sigma \vdash \texttt{Affecte}\,(x, \texttt{Alloc}\,(e_1)) \Rightarrow /\sigma, l_0 \mapsto \bot, \ldots l_{k-1} \mapsto \bot, \rho(x) \mapsto t}$$

Lecture

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow t \qquad \rho/\sigma \vdash e_2 \Rightarrow k \qquad k \in dom\,(t)}{\rho/\sigma \vdash \texttt{Lire}\,(e_1, e_2) \Rightarrow \sigma(t(k))}$$

Écriture

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow t \qquad \rho/\sigma \vdash e_2 \Rightarrow k \qquad k \in dom\,(t) \qquad \rho/\sigma \vdash e_3 \Rightarrow v}{\rho/\sigma \vdash \texttt{Ecrire}\,(e_1, e_2, e_3) \Rightarrow /\sigma, t(k) \mapsto v}$$

Interprétation des tableaux, syntaxe et valeurs

```
type expression = ...
| Alloc of expression
| Lire of expression * expression
and instruction = ...
| Ecrire expression * expresssion * expression
```

```
type valeur = Int of int | Array of valeur array | Undefined
type erreur = ... | Type | Index

let array_of_valeur = function
| Array t -> t
| _         -> erreur Type
```

Code des expressions

```
let rec evalue env = function
    ...
  | Alloc (e1) ->
      let k = int_of_valeur (evalue env e1) in
      if k >= 0 then
        Array (Array.create k Undefined)
      else
        erreur Index
  | Lire (e1, e2) ->
      let t = array_of_valeur (evalue env e1)
      and k = int_of_valeur (evalue env e2) in
      if 0 <= k && k < Array.length t then
        t.(k)
      else
        erreur Index
```

Code des instructions

```
and execute env = function
  ...
| Ecrire (e1, e2, e3) ->
    let t = array_of_valeur (evalue env e1)
    and k = int_of_valeur (evalue env e2)
    and v = evalue env e3 in
    if 0 <= k && k < Array.length t then
        t.(k) <- v
      else
        erreur Index
```

Fonctions globales

Dans la calculette : un programme est une suite de définitions de fonctions, plus une expression.

```
fact(x) = if x=0 then 1 else x * fact (x-1) ;;
fact(10)
```

Important Les seules variables du corps d'une fonction sont :

- le paramètre ;
- ou une fonction (appliquée).

```
type expression = ...
| App of string * expression

type fonction = Fun of string * expression (* Pas une expression *)

type programme =
  {fonctions : (string * fonction) list ;
  expr : expression}
```

Sémantique des fonctions globales

Évaluation des programmes

On évalue $\{\texttt{fonctions=}\rho_f\ \texttt{;}\ \texttt{expr=}e\}$ en prouvant le jugement :

$$\rho_f;\ \ \emptyset \vdash e \Rightarrow v$$

Règle de l'application

$$\frac{\rho_f(f) = \texttt{Fun}\,(x, e_f) \qquad \rho_f;\ \ \rho \vdash e \Rightarrow v_a \qquad \rho_f;\ \ (x \mapsto v_a) \vdash e_f \Rightarrow v}{\rho_f;\ \ \rho \vdash \texttt{App}\,(f, e) \Rightarrow v}$$

Note L'environnement pour évaluer le corps e_f.

Re-Note En pratique, il peut y avoir plus d'un argument et les fonctions déclarent des variables locales.

Interpréteur

```
let rec evalue fenv env = function
        ...
| App (f, e) ->
    let va = evalue fenv env e in
    let Fun (x,ef) = cherche fenv f in
    evalue fenv [x, va] ef

let exécute_programme {fonctions=fenv ; expr=e} =
  evalue fenv [ ] e
```

Note Ou pourrait abstraire un peu l'environnement (un seul `env`, des fonctions `cherche_fun` et `cherche_val`).

Fonctions et construction impératives

(Au moins) deux règles possibles.

Appel par valeur

$$\frac{\rho_f(f) = (x, e_f) \qquad \rho_f;\ \rho/\sigma \vdash e \Rightarrow v_a \qquad \ell_a \notin dom(\sigma) \qquad \rho_f;\ (x \mapsto \ell_a)/\sigma, \ell_a \mapsto v_a \vdash e_f \Rightarrow v}{\rho_f;\ \rho/\sigma \vdash \texttt{App}\,(f, e) \Rightarrow v}$$

Appel par variable

$$\frac{\rho_f(f) = (x, e_f) \qquad \rho(y) = \ell_a \qquad \rho_f;\ (x \mapsto \ell_a)/\sigma \vdash e_f \Rightarrow v}{\rho_f;\ \rho/\sigma \vdash \texttt{App}\,(f, y) \Rightarrow v}$$

Vocabulaire La variable est le paramètre *formel* l'argument le paramètre *effectif*.

Paramètres tableaux

Appel par valeur de nos tableaux

$$\frac{\rho_f(f) = (x, e_f) \qquad \rho_f;\ \ \rho/\sigma \vdash e \Rightarrow t \qquad t = (0 \mapsto \ell_0, \ldots, k-1 \mapsto \ell_{k-1}) \qquad \ell_a \notin dom(\sigma) \qquad \rho_f;\ \ (x \mapsto \ell_a)/\sigma, \ell_a \mapsto t \vdash e_f \Rightarrow v}{\rho_f;\ \ \rho/\sigma \vdash \mathtt{App}\,(f, e) \Rightarrow v}$$

Connaissez vous le passage des tableaux, en Java, en C, en Pascal ?

Il y a une grosse différence : si on passe un tableau à une fonction Pascal, la fonction possède un nouveau tableau ! Alors, comment comprendre :

> *En C, en Java et en Pascal, la règle (par défaut) est le passage par valeur ?*

Bonus : la calculette fonctionnelle

- `Fun` (x, e) désigne une fonction qui à x associe l'expression e.
- `App` (e_1, e_2) désigne l'application d'une expression fonctionnelle e_1 à une expression e_2.
- Et surtout, d'autres variables que le(s) paramètre(s) peuvent apparaître dans le corps d'une fonction.

Difficultés engendrées

1. Les fonctions sont définies dans un environnement ρ mais elles sont appelées dans un environnement ρ' en général différent ;

   La *liaison lexicale* exige que la valeur des variables soit prise au moment de la définition.

2. Si les fonctions sont des valeurs (retournées en résultat), alors leur durée de vie peut excéder leur portée lexicale.

Exemple

Liaison statique

```
let x = 3                    (* env1 : x est lié à une valeur v  *)
let rec mem =                (* la valeur de x est celle de env1 *)
  function [ ] -> false | h::t -> x = h && mem t
let x = w                    (* env2 : x est lié à la valeur w   *)
mem l                        (* doit être évalue; dans env1      *)
```

Valeur fonctionnelles

```
let incr x = fun y -> x + y
                       (* incr x retourne une fonction       *)
let f = incr 3         (* f doit mémoriser la valeur 3 de x  *)
f 4                    (* pour la restaurer lors de l'appel  *)
```

Fermetures

Une solution générale consiste à évaluer les fonctions en des fermetures.

Une fermeture est une paire $\langle \texttt{Fun}\,(x, e), \rho \rangle$ formée d'une expression fonctionnelle $\texttt{Fun}\,(x, e)$ et d'un environnement ρ.

$$\rho \vdash \texttt{Fun}\,(x, e) \Rightarrow \langle \texttt{Fun}\,(x, e), \rho \rangle$$

On peut éventuellement restreindre ρ à l'ensemble des variables libres dans $\texttt{Fun}\,(x, e)$. Elle permet d'*enfermer* une expression avec son environnement statique. Une fermeture est une valeur qui peut être passée en argument.

L'application restore l'environnement statique.

$$\frac{\rho \vdash e_1 \Rightarrow \langle \texttt{Fun}\,(x, e_0), \rho_0 \rangle \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho_0, x \mapsto v_2 \vdash e_0 \Rightarrow v}{\rho \vdash \texttt{App}\,(e_1, e_2) \Rightarrow v}$$

Variables libres

Un variables libre d'une expression est une variable utilisée dans cette expression dans un contexte où elles ne sont pas liée :

$$vl\,(\texttt{Const}\;n) = \emptyset \qquad vl\,(\texttt{App}\,(e_1, e_2)) = vl\;e_1 \cup vl\;e_2$$

$$vl\,(\texttt{Variable}\;x) = \{x\} \qquad vl\,(\texttt{Plus}\,(e_1, e_2)) = vl\;e_1 \cup vl\;e_2$$

$$vl\,(\texttt{Fun}\,(x, e)) = vl\;e_1 \setminus \{x\} \qquad \ldots$$

Exemple La seul variable libre de `(fun y -> x + (fun x x) y)` est `x`. En effet `y` apparaît dans une contexte dans lequel elle est liée. Par contre, `x` apparaît au moins une fois dans un contexte dans lequel elle n'est pas liée.

Fermetures

Il suffit de mettre dans une fermeture les variables libres. On peut remplacer $\langle e, \rho \rangle$ par $\langle e, \rho \upharpoonright vl\,(e) \rangle$.

Fonctions locales non retournées

Les fonctions locales non retournées sont un cas intermédiaire.

On peut éviter les fermetures par remontée des variables.

Principe : une fonction $f \triangleq \texttt{Fun}\,(x, e)$ avec une variable libre y locale à une fonction globale g peut être transformée en une fonction globale $f' \triangleq \texttt{Fun}\,((x, y), e)$. Les appels à f de la forme $\texttt{App}\,(f, e)$ sont transformés en $\texttt{App}\,(f, e, y)$.

Il faut si besoin renommer d'autres liaisons locale de la variable y qui cacheraient la liaison globale y.

Note : Comme les deux programmes ont la même sémantique, on peut utiliser la transformation inverse si le langage le permet, et écrire des programmes où les variables sont "descendues".

Exemple

Deux styles de programmation différents.

Variables remontées

```
let member (x,l) =
  let rec mem l =
    match l with
    | []    -> false
    | h::t ->
         x = h || mem t
  in mem l
```

Variables descendues

```
let member (x,l) =
  let rec mem (x,l) =
    match l with
    | []    -> false
    | h::t ->
         x = h || mem (x,t) in
  mem (x,l)
```

En pratique les deux écritures sont utiles (selon le contexte, et le nombre d'arguments, préciser).

Un compilateur peut effectuer l'une ou l'autre des transformations.

Le langage Pseudo-Pascal (PP)

- Le langage est impératif. Il distingue instructions et expressions.
- Les valeurs sont les entiers, les booléens et les tableaux. Sémantiquement les tableaux sont des références. Les tableaux sont alloués dynamiquement (durée de vie infinie).
- L'ordre d'évaluation est de gauche à droite.
- Les fonctions sont globales, mutuellement récursives et ne peuvent qu'être appelées.
- Les arguments des fonctions sont passés par valeur.
- Les variables sont toutes mutables.
- Les variables sont déclarées.

C'est à vous d'écrire un interpréteur, en grappillant dans ce cours et en utilisant votre culture. Voir aussi (et surtout) le poly.