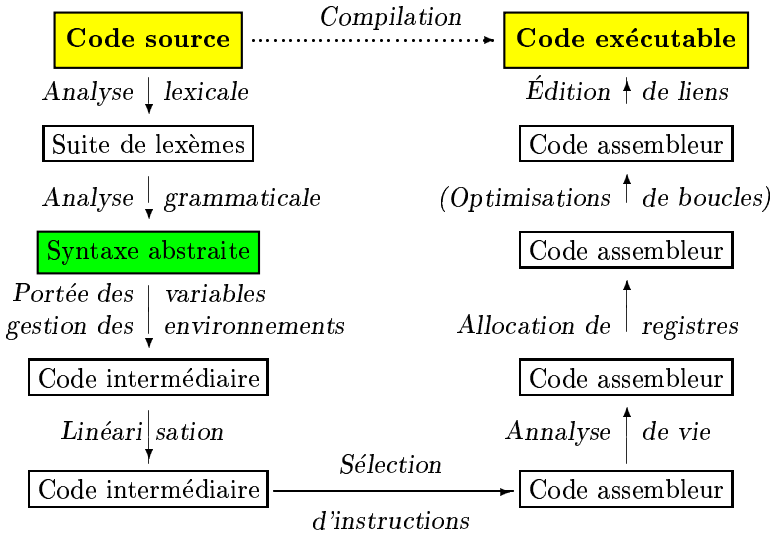
Langages de programmation
Sémantiques opérationnelles
Interprètes
Le langage pseudo-pascal.

Didier Rémy
Octobre 2000

http://cristal.inria.fr/~remy/poly/compil/2/
http://w3.edu.polytechnique.fr/profs/informatique//Didier.Remy/compil/2/

Slide 1



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 (ou grande) et la possibilité d'exprimer la récursion (construction primitive ou boucle `while`). Ex : Fortran, Pascal, C, Ocaml, etc.

Slide 2

Langages spécialisés

Langages pour le graphisme, pour commander des robots, pour faire des annimations graphiques ; 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 fonctions

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

Les structures de données (et filtrage)

Slide 3

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 donc aussi importante.

Note : Pour comparer l'expressivité formellement, il faut en général le faire point à point en exhibant un codage d'un langage dans un autre qui respecte certaines règles (compositionalité, localité, *etc.*)

Syntaxe concrète – syntaxe abstraite

La syntaxe concrète décrit les mots du langage et la façon d'assembler les mots en phrases pour constituer des programmes.

Elle ne donne aucun sens aux phrases :

Slide 4

- Plusieurs notations sont possibles pour une même construction. Exemple : les caractères `'a'` et `'\097'` ou en la boucle `for` qui s'exprime en fonction de la boucle `while`
- 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)

Slide 5

```
expression ::= ENTIER
             | IDENTIFICATEUR
             | expression binop expression
             | '(' expression ')'
binop      ::= '+' | '-' | '*' | '/'
```

Syntaxe abstraite

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

Exemple l'expression `(1 - x) * 3` a pour représentation

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

Slide 6

Sémantique des programmes

Il s'agit de donner un sens aux programmes.

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.

Slide 7

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.

Cette sémantique est plus proche de la syntaxe (moins abstraite), mais elle est plus facile à manipuler.

C'est aussi celle qui nous intéresse le plus souvent (pour calculer).

Sémantique opérationnelle de la calculette

Les valeurs sont les entiers.

```
type valeur = int
```

On peut définir l'évaluation par un programme Ocaml qui prend un environnement initial associant des valeurs à certaines

Slide 8

variables, une expression à évaluer et retourne un entier :

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

Slide 9

Une présentation plus neutre (S.O.S.)

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

Slide 10

Une autre présentation équivalente, mais plus mathématique et plus modulaire modulaire 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, c'est-à-dire comme le plus petit ensemble vérifiant une certains nombre de règles (implications).

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

$$\frac{P_1 \wedge \ldots \wedge P_k}{C}$$

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, i.e. c'est un ensemble de paires notées $x \mapsto v$.
- $e \in$ *expressions* et $v \in \mathbb{N}$. On note $\bar{n}$ l'entier naturel associé à sa représentation n.

Slide 11

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

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

Une construction de liaison

On étend la calculette avec des liaisons. On ajoute un noeud de syntaxe abstraite

```
| Let of string * expression * expression
```

avec, par exemple, la syntaxe concrète

```
"let" IDENTIFICATEUR "=" expression "in" expression
```

Slide 12

L'expression $\texttt{Let}\,(x, e_1, e_2)$ lie la variable x à l'expression e_1 dans l'évaluation de l'expression e_2.

Formellement :

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \qquad \rho, x_1 \mapsto v_1 \vdash e_2 \Rightarrow v_2}{\rho \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2}$$

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
```

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

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

Slide 13

Exercice

```
let x = 1 in (let x = 2 in x) + x
```

Donner

- la syntaxe abstraitre de cette expression et le résultat de son
- Le résultat de son évaluation.
- La dérivation de l'évaluation de l'expression la plus interne.

Arbre de dérivation

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

Slide 14

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

Formalisation des erreurs

L'évaluation peut mal se passer, même dans la calculette, par exemple lors d'une division par 0 ou de l'accès à une variable non liée.

La sémantique opérationnelle doit aussi formaliser les erreurs.

Slide 15

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.

$$\dfrac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 \qquad v_2 \neq 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow v_1/v_2} \qquad \dfrac{\rho \vdash e_2 \Rightarrow 0}{\rho \vdash \texttt{Bin}\,(\texttt{Div}\,, e_1, e_2) \Rightarrow \texttt{Division}}$$

Le type des erreurs (par exemple) :

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

Implémentation des erreurs

Formellement, on utilise un type somme

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

En pratique, on utilise les exceptions du langage hôte

Slide 16

```
exception Erreur of erreur
let cherche x l =
  try List.assoc x l
  with Not_found -> raise (Erreur (Variable_libre x))
let rec évalue environnement = function
    ...
  | Bin (Div, e1, e2) ->
      let v1 = évalue env e1 and v2 = évalue env e2 in
      if v2 = 0 then raise (Erreur (Division_par_zéro ))
      else v1 / v2 ;;
```

Terminaison

L'évaluation peut ne pas terminer.

La terminaison n'est pas modéliser par la S.O.S. Un programme qui ne termine pas ne peut pas être mis en relation avec une valeur.

Slide 17

Dans le cours *Langage et programmation* on définit une sémantique opérationnelle à réduction qui permet quand même de décrire le calcul des programmes qui ne terminent pas.

La calculette fonctionnelle

Ajout des fonctions

- `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.

Difficultés engendrées

Slide 18

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

   La *liaison statique* 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 :

Slide 19

```
let x = 3          (* env1 : x est lié à une valeur v *)
let rec mem =
  function [] -> false
| h:: t -> x = h && mem t
let x = w          (* env2 : x est lié à la valeur w *)
mem l              (* il faut évaluer l'application dans env1 *)
```

Valeur fonctionnelles

```
let incr x = fun y -> x + y
                   (* incr x retourne une fonction *)
let f = incr 3     (* f mémorise la valeur de l'argument *)
f 4                (* et le 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.

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

Slide 20

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 passé 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

Les variables libres d'une expression sont les variables utilisées dans un contexte où elles ne sont pas liée dans cette expression :

$$vl\,(\texttt{Const}\ n) = \emptyset \qquad vl\,(\texttt{App}\,(e_1, e_2)) = vl\ e_1 \cup vl\ e_2$$
$$vl\,(\texttt{Variable}\ x) = \{x\} \qquad vl\,(\texttt{Plus}\,(e_1, e_2) = vl\ e_1 \cup vl\ e_2$$
$$vl\,(\texttt{Fun}\,(x, e)) = vl\ e_1 \setminus \{x\} \qquad \ldots$$

Slide 21

Exemple La seul variable libre de `(fun y -> x + (fun x x) y)` sont `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 $\langle e, \rho \restriction vl\,(e) \rangle$ car ρ ne sera utiliser que pour évaluer l'expression e.

Évaluation paresseuse

La sémantique précédente est dite *en appel par valeur* parce que les arguments sont évalués avec d'être passés aux fonctions.

Pour donner une sémantique *en appel par nom* (utilisée dans les langages Algol, Miranda, Haskell, Gaml), on gèle les arguments des fonctions, jusqu'à ce que leur évaluation soit rendue nécessaire.

Slide 22

Comme pour les fonctions, il faut sauver le contexte d'évaluation dans des fermetures, aussi appelées glaçons (expressions gelées).

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

$$\frac{\rho' \vdash e \Rightarrow v}{\rho, x \mapsto \langle e, \rho' \rangle \vdash x \Rightarrow v}$$

Évaluation paresseuse

L'appel par nom conduit à réévaluer plusieurs fois la même expression (chaque fois qu'un glaçon est dégelé).

L'*évaluation paresseuse* est une évaluation en appel par nom, avec en plus la propriété qu'un glaçon n'est évalué qu'un seule fois. On utilise l'affectation (voir plus loin) pour transformer un glaçon en une valeur après sa première évaluation.

Slide 23

L'évaluation paresseuse parce qu'elle ne force pas l'évaluation des arguments à de meilleures propriétés ($\texttt{App}\,(\texttt{Fun}\,(x, e_1), e_2)$) est équivalent à $e_1\{x \leftarrow e_2\}$, *i.e.* e_1 où x est remplacé par e_2.

En pratique toutefois :

+ L'évaluation paresseuse ne se combine pas bien avec des effets de bords (affectation, exception).
- L'évaluation paresseuse peut être simulée (manuellement) en appel par valeur en codant les glaçons comme des fonctions.

La conditionnelle

La construction $\texttt{Ifz}\,(e_1, e_2, e_3)$ est toujours une construction paresseuse (l'un des deux derniers arguments seulement est évalué).

$$\frac{\rho \vdash e_1 \Rightarrow 0 \qquad \rho \vdash e_2 \Rightarrow v}{\rho \vdash \texttt{Ifz}\,(e_1, e_2, e_3) \Rightarrow v} \qquad \frac{\rho \vdash e_1 \Rightarrow n \qquad n \neq 0 \qquad \rho \vdash e_3 \Rightarrow v}{\rho \vdash \texttt{Ifz}\,(e_1, e_2, e_3) \Rightarrow v}$$

Slide 24

Le AND et le OR

En Pascal ces opérateurs sont stricts : ils évaluent leurs arguments. En C et dans la plupart des langages, ils sont paresseux comme la construction "ifz". Le plus simple est de lire e_1 `and` e_2 comme `if` e_1 `then` e_2 `else false`.

Fonctions globales non retournées (en appel par valeur)

Les fermetures peut être évitées lorsque les fonctions sont globales et jamais retournées en résultat.

Les seules variables libres d'une fonction sont alors les variables globales. L'environnement de définition est l'environnement global, que l'on peut réinstaller à toute application.

Slide 25

Pour cela, on sépare l'environnement global et de l'environnement local et on écrit : $\rho_0; \rho \vdash e \Rightarrow v$.

$$\rho_0; \emptyset \vdash \texttt{Fun}\,(x, e) \Rightarrow \texttt{Fun}\,(x, e)$$

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

Fonctions locales non retournées

Les fonctions locales non retournées sont un cas intermédiaire.

On peut éviter les fermetures par remontée des variables.

Slide 26

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 $\texttt{App}\,(f, e_i)$ sont transformés en $\texttt{App}\,(f, e, y)$.

La portée des variables libres de f est en effet plus large que celle de f (renommer les liaisons intermédiaires qui cacheraient un y_i).

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.

Slide 27

Variables remontées

```
let member x l =
  let rec mem = function
    | [] -> false
    | h:: t ->
        x = h || mem t
  in mem l
```

Variables descendues

```
let rec member x =
  function
  | [] -> false
  | h:: t ->
      x = h || member x t
```

La transformation de gauche à droite est plutôt utilisée dans un compilateur, et celle de droite à gauche par le programmeur.

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

L'affectation des variables

Slide 28

On peut affecter les variables en Pascal ou C.

Il faut introduire une notion de mémoire σ qui lie des adresses-mémoire ℓ à des valeurs. L'environnement lie maintenant les variables modifiables (ou les références) à des positions mémoire (*left-value*) dont la vraie valeur (*right-value*) est définie dans σ.

On écrit $\rho/\sigma \vdash e \Rightarrow v/\sigma'$ qui se lit *dans l'état-mémoire σ et l'environnement ρ, l'évaluation de l'expression e produit une valeur v et un nouvel état mémoire σ'.*

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow v_1/\sigma_1 \qquad \ell \notin \sigma_1 \qquad \rho, x_1 \mapsto \ell/\sigma_1, \ell \mapsto v \vdash e_2 \Rightarrow v_2/\sigma_2}{\rho/\sigma \vdash \texttt{Let}\,(x, e_1, e_2) \Rightarrow v_2/\sigma_2}$$

Lecture et écriture

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

Implémentation

Slide 29

On peut profiter des strutures mutables du langage hôte et se contenter d'un environnement qui lie les variables à des valeurs mutables (cela revient à utiliser la propre mémoire du langage hôte pour représenter σ).

```
type environnement = (string * valeur ref) list
...
let rec évalue env = function
    ...
  | Variable x -> !(chercher x env)
  | Affecte (x, e) ->
      let v = évalue env e in (trouve x env) := v
```

(Pour simplifier, on a considérer toutes les variables mutables.)

Tableaux

Les tableaux alloués dynamiquement sont des structures de données mutables un peu comme les variables. Mais à la différences des variables les tableaux peuvent être passés à d'autres fonctions. L'accès à un tableau est explicite (il est implicite pour une variable).

Slide 30

On ajoute trois constructions

- `Tableau` (e_1, e_2) alloue un nouveau tableau de taille la valeur de e_1 initialisé avec la valeur de e_2.
- `Lire` (e_1, e_2) lit la case (correspondant à la valeur de) e_2 du tableaux (la valeur de) e_1.
- `Ecrire` (e_1, e_2, e_3) écrit la case e_2 du tableaux e_1 avec la valeur de e_3.

(Les références de Ocaml sont des tableaux à une seule case.)

Sémantiqe des tableaux

On introduit de nouvelles valeures t qui sont des fonctions d'un ensemble d'indices $[0, k-1]$ vers des adresses-mémoire.

Création

Slide 31

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow k/\sigma_1 \qquad \rho/\sigma_1 \vdash e_2 \Rightarrow v/\sigma_2 \qquad \ell_i \notin dom\,(\sigma_2), i \in [0, k-1]}{\rho/\sigma \vdash \texttt{Tableau}\,(e_1, e_2) \Rightarrow t/\sigma'}$$

où $t \triangleq (i \mapsto \ell_i)_0^{k-1}$ et $\sigma' \triangleq \sigma, (\ell_i \mapsto v)_0^{k-1}$.

Lecture

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow t/\sigma_1 \qquad \rho/\sigma_1 \vdash e_2 \Rightarrow k/\sigma_2 \qquad k \in dom\,(t) \qquad t(k) \in dom\ \sigma_2}{\rho/\sigma \vdash \texttt{Lire}\,(e_1, e_2) \Rightarrow \sigma_2(t(k))/\sigma_2}$$

Écriture similaire

Interprétation des tableaux

- On profite des tableaux mutables du langage hôte.
- Il y a plusieurs formes de valeurs qu'il faut distinguer par un type somme.
- L'accès à une valeur du mauvais type produit une erreur (que le typage devra détecter).
- L'accès en dehors des bornes type produit une erreur.

Slide 32

Distinguer les valeurs (par un type somme)

```
(* les différents types de valeur *)
type valeur = Entier of int | Tableau of valeur array
type erreur = ... | Type | Indice
(* pour retirer une valeur de la bonne forme *)
let entier =
  function Entier n -> n | _ -> raise (Erreur Type)
let tableau =
  function Tableau t -> t | _ -> raise (Erreur Type)
```

Tableaux (suite)

Slide 33

```
let rec évalue env =
    ...
  | Tableau (e_1, e_2) ->
      let k = entier (évalue env e_1) in
      let v = entier (évalue env e_2) in
      Tableau (Array.create k v)
  | Lire (e_1, e_2) ->
      let t = tableau (évalue env e_1) in
      let k = entier (évalue env e_2) in
      if 0 <= k && k < Array.length t
      then Tableau (Array.create k v)
      else raise (Erreur Indice)
  | Ecrire (e_1, e_2) ->
      ...
```

Variables et tableaux non initialisées

En Pascal ou en C, les variables et les tableaux ne sont pas toujours initialisés.

Formellement, on introduit une valeur indéfinie.

L'accès à une variable ou à une case non définie est une erreur.

Ordre d'évaluation

Slide 34

La présence d'effets de bords (affectation) fixe l'ordre d'évaluation !

$$\frac{\rho/\sigma \vdash e_1 \Rightarrow \langle \texttt{Fun}\,(x, e_0), \rho_0\rangle/\sigma_1 \quad \rho/\sigma_1 \vdash e_2 \Rightarrow v_2/\sigma_2 \quad \rho_0, x \mapsto v_2/\sigma_2 \vdash e_0 \Rightarrow v/\sigma'}{\rho/\sigma \vdash \texttt{App}\,(e_1, e_2) \Rightarrow v/\sigma'}$$

Le langage Pseudo-Pascal (PP)

Description informelle

Slide 35

- Syntaxe Pascal.
- Valeur entières et booléens ; ni chaînes ni flottants
- Des tableaux dynamiques (durée de vie infinie). Pas d'enregistrements.
- Les fonctions sont globales, mutuellement récursives et jamais retournées.
  Le résultat d'une fonction est passé en affectant la variable de même nom.
- Le passage d'argument est en appel par valeur.
- Les variables sont mutables.

Arbre de syntaxe abstraite

On conserve les informations de types (pour permettre une analyse de type ultérieure)

```
type type_expr = Integer | Boolean | Array of type_expr;;
```

Un programme est composé d'un ensemble de déclarations de variables et de fonctions et d'un corps (une instruction).

Slide 36

```
type var_list = (string * type_expr) list
type program = {
    global_vars : var_list ;
     definitions : ( string * definition ) list ;
     main : instruction ;    (* corps du programme *) }
and definition = {
     arguments : var_list ; result : type_expr option;
     local_vars : var_list ;
     body : instruction ;    (* corps de la fonction *) }
```

Suite (expressions)

Slide 37

```
and expression =
     (* constantes *)
  | Int of int | Bool of bool
     (* opérations binaires *)
  | Bin of binop * expression * expression
     (* accès à une variable *)
  | Get of string
     (* appel de fonction *)
  | Function_call of string * expression list
     (* accès dans un tableau à une position *)
  | Geti of expression * expression
     (* Création d'un tableau d'une certaine taille *)
  | Alloc of expression * type_expr
and binop = Plus | Minus | Times | Div
  | Lt | Le | Gt | Ge | Eq | Ne
```

Suite (expressions)

Slide 38

```
and instruction =
    (* Affectation d'une variable *)
  | Set of string * expression
    (* Suite d'instructions *)
  | Sequence of instruction list
  | If of expression * instruction * instruction
  | While of expression * instruction
    (* Appel de procédure *)
  | Procedure_call of string * expression list
    (* Ecriture d'un entier *)
  | Write_int of expression
    (* Lecture d'un entier dans une variable *)
  | Read_int of string
    (* Affectation dans un tableau *)
  | Seti of expression * expression * expression
```

Exercices

Slide 39

Interprète pour le langage PP

La syntaxe abstraite est donnée par `pp.mli`.

Les interfaces sont imposées `interpret.mli`

Quelques programmes tests : `fact.ml`, `fib.ml`, et `memo_fib.ml`

Écrivez-en quelques autres.

Extensions du langage langage PP

Considérer un langage PP^+ plus évolué et le traduire dans le langage PP. On pourra

- Remplacer les variables locales par une construction de liaison Let (x, e_1, e_n). La traduction doit remonter les variables à la définition de fonction ou au niveau global si nécessaire.
- Autoriser les fonctions locales. Les rendre locales par remontée des variables.

Indications

Procéder pas à pas..

1. Reprendre la définition de la syntaxe abstraite.
2. Commencer par définir le type des valeurs ;
   Définir également le type des erreurs (que l'on complétera au fur et à mesure des besoins)

Slide 40

3. Définir la structure de l'environnement (éventuellement provisoire) et les fonctions qui le manipule.
4. Définir l'évaluation des expressions arithmétiques avec constantes et variables. Tester en évaluant sur des expressions arithmétiques simples

   Voir la solution si besoin

5. Ajouter l'évaluation des instructions simples (Affectation de variable, Séquence, Impression d'un entier). Tester.

   Voir la solution si besoin

6. Écrire la fonction d'évaluation d'un programme sans variable.
7. Écrire la fonction d'évaluation d'un programme avec variables globales.
8. Ajouter les instructions conditionnelles, puis les procédures, les fonctions et les tableaux.
9. Rapporter les erreurs d'exécution.

Slide 41