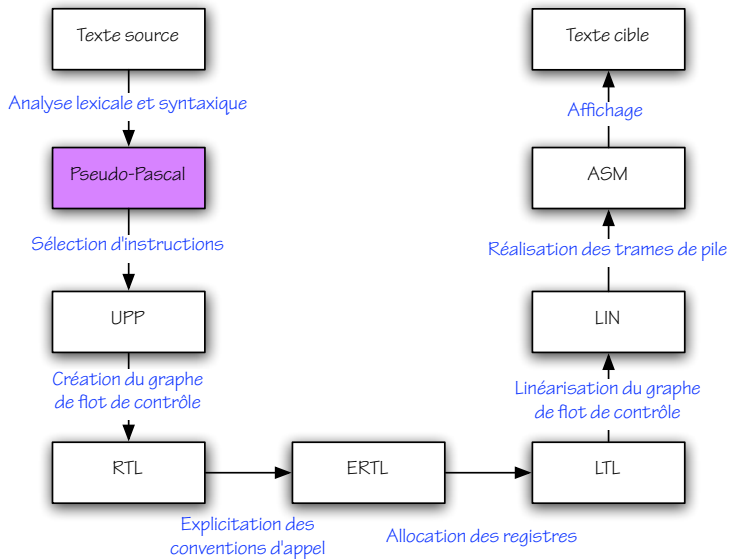


Compilation (INF 564)

Syntaxe, sémantique et interprétation de Pseudo-Pascal

François Pottier

9 décembre 2015



Syntaxes concrète et abstraite

Syntaxe abstraite de Pseudo-Pascal

Sémantique opérationnelle

Interprétation

Syntaxes concrète et abstraite

Un *langage* de programmation est un ensemble de programmes.

En termes de *syntaxe concrète*, un programme est une *suite de caractères*. La syntaxe concrète spécifie comment les programmes s'écrivent "sur papier".

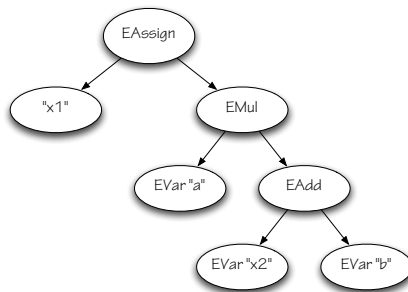
En termes de *syntaxe abstraite*, un programme est un *arbre*. La syntaxe abstraite définit la structure idéale (mathématique) des programmes.

Syntaxes concrète et abstraite

Voici un fragment de syntaxe *concrète* :

```
x1 := a * (x2 + b);
```

Celui-ci correspond à un fragment de syntaxe *abstraite* :



Syntaxe concrète sur machine

Un programme en syntaxe concrète se présente sous forme de *suite de caractères* :

```
x 1 _ : = _ a _ * _ ( x 2 _ + _ b ) ;
```

ou bien, après *analyse lexicale*, sous forme de *flux de lexèmes* :

```
ID("x1") COLONEQ ID("a") TIMES LPAREN  
ID("x2") PLUS ID("b") RPAREN SEMICOLON
```

Ce flux de lexèmes sera transformé par *analyse syntaxique* en un arbre de syntaxe abstraite (cf. prochain cours).

Syntaxe abstraite sur machine

Les arbres de syntaxe abstraite sont définis en OCaml sous forme d'un *type algébrique* :

```
type expr =  
| EVar of string  
| EAdd of expr * expr  
| EMul of expr * expr  
| EAssign of string * expr  
| ...
```

L'arbre correspondant au fragment précédent s'écrit alors :

```
EAssign ("x1", EMul (EVar "a", EAdd (EVar "x2", EVar "b"))))
```

Syntaxe abstraite sur papier

Lorsqu'on raisonne *sur papier* à propos d'arbres de syntaxe *abstraite*, sous quelle forme les écrit-on? Dessiner de petits arbres serait trop lourd. La notation OCaml est bien lourde également.

De ce fait, on emploie informellement la syntaxe *concrète* du langage pour dénoter des arbres de syntaxe *abstraite*! C'est ce que nous allons faire dans ce qui suit...

Syntaxes concrète et abstraite

Syntaxe abstraite de Pseudo-Pascal

Sémantique opérationnelle

Interprétation

Catégories

Pour définir Pseudo-Pascal, nous allons définir toute une série de *catégories* syntaxiques : *types*, *constantes*, *opérateurs* unaires et binaires, *opérations primitives*, *cibles* d'appel, *expressions*, *conditions*, *instructions*, *définitions* de procédures ou fonctions, et enfin *programmes*.

La définition complète est résumée dans une *fiche*.

Types

Les *valeurs* que Pseudo-Pascal permet de manipuler sont les entiers, les booléens, et les tableaux de valeurs. Les *types* reflètent cette classification des valeurs.

τ	::=	types
		integer entiers
		boolean booléens
		array of τ tableaux

Constantes

Les *constantes* sont booléennes ou entières. Il n'y a pas de constantes de type tableau : les tableaux sont alloués dynamiquement.

```
k ::=      constantes
    | b  constante booléenne
    | n  constante entière
```

Opérateurs

Des *opérateurs* unaires et binaires sont utilisés dans la construction des expressions.

uop ::=		opérateurs unaires
	-	négation
bop ::=		opérateurs binaires
	+	addition
	-	soustraction
	x	multiplication
	/	division
	< ≤ > ≥ = ≠	comparaison

Opérations primitives

Le langage propose quelques opérations *primitives* (prédéfinies).

```
 $\pi$  ::= opérations primitives
    | write  affichage d'un entier
    | writeln affichage d'un entier et retour à la ligne
    | readln lecture d'un entier
```

La *cible* d'un appel de procédure ou fonction est soit primitive, soit définie par l'utilisateur.

```
 $\varphi$  ::= cible d'un appel
    |  $\pi$  opération primitive
    |  $f$  procédure ou fonction définie par l'utilisateur
```

Expressions, conditions et instructions

Pseudo-Pascal distingue *expressions*, *conditions*, et *instructions*.

Cette distinction est *arbitraire*. On pourrait n'effectuer aucune distinction au niveau de la syntaxe abstraite et se reposer sur le *typage* pour effectuer les vérifications nécessaires.

Expressions

Les *expressions* sont définies ainsi :

$e ::=$	expressions
k	constante
x	variable
$uop\ e$	application d'un opérateur unaire
$e\ bop\ e$	application d'un opérateur binaire
$\varphi(e \dots e)$	appel de fonction
$e[e]$	lecture dans un tableau
$new\ array\ of\ \tau\ [e]$	allocation d'un tableau

Conditions

Les *conditions* sont des combinaisons booléennes d'expressions :

c	::=	conditions
		e expression (à valeur booléenne)
		$\text{not } c$ négation
		$c \text{ and } c$ conjonction
		$c \text{ or } c$ disjonction

Instructions

Les *instructions* sont définies au-dessus des expressions et conditions :

$i ::=$	instructions
$\varphi(e \dots e)$	appel de procédure
$x := e$	affectation
$e[e] := e$	écriture dans un tableau
$i \dots i$	séquence
if c then i else i	conditionnelle
while c do i	boucle

Procédures et fonctions

Une *définition de fonction* nomme la fonction puis déclare ses *paramètres formels*, le type de son résultat, et ses *variables locales*, avant de donner le *corps* de la fonction.

$d ::=$	définitions de procédures/fonctions
$f(x:\tau \dots x:\tau) : \tau^?$	en-tête
$\text{var } x:\tau \dots x:\tau$	variables locales
i	corps

Une définition de *procédure* est identique, excepté qu'une procédure n'a pas de résultat.

Programmes

Un *programme* est composé de déclarations de *variables globales*, de déclarations de procédures ou fonctions, et d'un corps.

$p ::=$	programme
var $x:\tau \dots x:\tau$	variables globales
$d \dots d$	définitions de procédures/fonctions
i	corps

Expressivité et complétude

Même si la définition de sa syntaxe peut paraître longue, Pseudo-Pascal est un langage *très réduit*. Il ne propose ni structures de données, ni fonctions de première classe, ni objets, ni exceptions...

Informellement, un langage est dit *expressif* s'il permet une écriture concise et élégante des algorithmes. En ce sens, Pseudo-Pascal est *plus expressif* que l'assembleur MIPS, mais reste tout de même *peu expressif* comparé à Java ou OCaml.

Pseudo-Pascal est *Turing-complet* : tout algorithme peut en principe être exprimé en Pseudo-Pascal.

De la syntaxe à la sémantique

Nous avons défini la *structure* des programmes Pseudo-Pascal, mais non leur *signification*. Que se passe-t-il lorsqu'un programme est exécuté? Comment en prédire avec certitude le résultat?

Il nous faut une sémantique *formelle*. Une simple description textuelle est imprécise et ne permet pas le raisonnement mathématique — comment *prouverons-nous* que notre compilateur est correct?

Syntaxes concrète et abstraite

Syntaxe abstraite de Pseudo-Pascal

Sémantique opérationnelle

Interprétation

Jugement principal

La sémantique de Pseudo-Pascal est définie principalement par un *jugement* dont voici la forme :

$$p \rightarrow$$

Ce jugement se lit : “le programme p s'exécute sans erreur et termine”.

Jugements auxiliaires

Ce jugement principal est défini à l'aide de trois *jugements auxiliaires* dont voici la forme :

$$\begin{aligned}G, H, E / e &\rightarrow G', H', E' / v \\G, H, E / c &\rightarrow G', H', E' / b \\G, H, E / i &\rightarrow G', H', E'\end{aligned}$$

Un triplet G, H, E représente *l'état* de la machine. On le note aussi S .

Les *environnements* globaux et locaux G et E associent aux variables des valeurs.

Le *tas* H associe aux adresses des suites finies de valeurs.

Valeurs

Les *valeurs* manipulées au cours de l'exécution sont définies ainsi :

$v ::=$	valeurs
b	constante booléenne
n	constante entière
ℓ	adresse de tableau
nil	adresse invalide

Les tableaux sont alloués dans le *tas*; une variable de type tableau contient en fait une *adresse*.

Dérivation de jugements

Un jugement est considéré comme vrai si et seulement s'il est *dérivable* de façon *finie* à partir d'un jeu fixé de *règles de déduction*. Il s'agit là d'un mécanisme de *définition inductive*.

Voici un exemple simpliste. Les jugements Pair n et Impair n pourraient être définis par les règles suivantes :

	Successesseur pair	Successesseur impair
Zéro	$\frac{\text{Impair } n}{\text{Pair } n + 1}$	$\frac{\text{Pair } n}{\text{Impair } n + 1}$
Pair 0		

La barre horizontale se lit de haut en bas comme une *implication*. La méta-variable n est implicitement *universellement quantifiée*.

Règles

L'intégralité des règles qui définissent la sémantique de Pseudo-Pascal est donnée par une *fiche*.

Deux pages de règles quelques peu cryptiques peuvent sembler une définition bien lourde, et pourtant Pseudo-Pascal est *un langage très réduit* et très simple!

Voici quelques-unes de ces règles...

Constante

L'évaluation d'une *constante* est immédiate :

Constante
 $S/k \rightarrow S/k$

(Un *état* S est un triplet G, H, E .)

Accès aux variables

L'accès aux variables se fait à travers l'environnement approprié :

Variable locale

$$x \in \text{dom}(E)$$

$$\frac{}{G, H, E/x \rightarrow G, H, E/E(x)}$$

Variable globale

$$x \in \text{dom}(G) \setminus \text{dom}(E)$$

$$\frac{}{G, H, E/x \rightarrow G, H, E/G(x)}$$

Si une variable globale et une variable locale portent le même nom, cette dernière *éclipse* la précédente.

Affectation

L'affectation se fait en modifiant l'environnement approprié :

Affectation: variable locale

$$\frac{S/e \rightarrow G', H', E' / v \quad x \in \text{dom}(E')}{S/x := e \rightarrow G', H', E'[x \mapsto v]}$$

Affectation: variable globale

$$\frac{S/e \rightarrow G', H', E' / v \quad x \in \text{dom}(G') \setminus \text{dom}(E')}{S/x := e \rightarrow G'[x \mapsto v], H', E'}$$

Opérateurs

La sémantique des *opérateurs* est donnée par une fonction $\llbracket \cdot \rrbracket$ qui à chaque opérateur associe son interprétation en tant que fonction *partielle* des valeurs dans les valeurs.

Opérateur unaire

$$\frac{S/e \rightarrow S'/v}{S/\text{uop } e \rightarrow S'/\llbracket \text{uop} \rrbracket(v)}$$

Opérateur binaire

$$\frac{S/e_1 \rightarrow S'/v_1 \quad S'/e_2 \rightarrow S''/v_2}{S/e_1 \text{ bop } e_2 \rightarrow S''/\llbracket \text{bop} \rrbracket(v_1, v_2)}$$

L'évaluation des opérands se fait *de gauche à droite*.

Accès aux tableaux

L'accès à un tableau se fait en consultant le tas :

Lecture dans un tableau

$$\frac{\begin{array}{l} S/e_1 \rightarrow S'/\ell \quad S'/e_2 \rightarrow S''/n \\ S'' = G'', H'', E'' \quad H''(\ell) = v_0 \dots v_{p-1} \quad 0 \leq n < p \end{array}}{S/e_1[e_2] \rightarrow S''/v_n}$$

L'évaluation *échoue* si l'indice n est en dehors de l'intervalle $[0 \dots p[$.

Allocation d'un tableau

L'allocation d'un tableau se fait en ajoutant au tas une adresse fraîche ℓ , laquelle pointe vers un nouveau tableau de taille n :

$$\begin{array}{l}
 \text{Allocation d'un tableau} \\
 S/e \rightarrow G', H', E' / n \quad n \geq 0 \\
 \ell \# H' \quad H'' = H'[\ell \mapsto \text{default}(\tau)^n] \\
 \hline
 S/\text{new array of } \tau [e] \rightarrow G', H'', E' / \ell
 \end{array}$$

Chaque case du tableau contient initialement une *valeur par défaut* de type τ .

Valeurs par défaut

La *valeur par défaut* d'une variable de type τ est définie ainsi :

```
default(boolean) = false
default(integer) = 0
default(array of  $\tau$ ) = nil
```

Cette notion est rendue nécessaire par le fait que Pseudo-Pascal permet l'allocation de nouveaux emplacements mémoire sans exiger en même temps leur *initialisation*.

Appel de fonction

L'appel d'une fonction se fait en exécutant le corps de la fonction dans un *nouvel environnement* local, lequel *disparaît* lorsque la fonction rend la main :

Appel d'une fonction définie

$$\begin{array}{c}
 p \ni f(x_1 : \tau_1 \dots x_n : \tau_n) : \tau \quad \text{var } x'_1 : \tau'_1 \dots x'_q : \tau'_q \quad i \\
 E' = (x_j \mapsto v_j)_{1 \leq j \leq n} \cup (x'_j \mapsto \text{default}(\tau'_j))_{1 \leq j \leq q} \cup (f \mapsto \text{default}(\tau)) \\
 \frac{G, H, E' / i \rightarrow G', H', E'' \quad v = E''(f)}{G, H, E / f(v_1 \dots v_n) \rightarrow G', H', E / v}
 \end{array}$$

Le *résultat* de l'appel est lu dans la variable locale nommée f .

Cette règle ne dit pas tout...

Appel de fonction (suite)

Il faut préciser que l'appel se fait *par valeur*.

$$\begin{array}{l} \text{Évaluation des arguments de fonction} \\ \forall j \in \{1 \dots n\} \quad S_{j-1}/e_j \rightarrow S_j/v_j \\ S_n/\varphi(v_1 \dots v_n) \rightarrow S'/v \\ \hline S_0/\varphi(e_1 \dots e_n) \rightarrow S'/v \end{array}$$

Si les arguments effectifs sont des *expressions*, ils sont *évalués* (transformés en valeurs) *avant* l'appel.

Conjonction

En Pseudo-Pascal, l'évaluation de la conjonction et de la disjonction est "à court-circuit :"

Conjonction (si)

$$\frac{S/c_1 \rightarrow S'/\text{false}}{S/c_1 \text{ and } c_2 \rightarrow S'/\text{false}}$$

Conjonction (sinon)

$$\frac{\begin{array}{l} S/c_1 \rightarrow S'/\text{true} \\ S'/c_2 \rightarrow S''/b \end{array}}{S/c_1 \text{ and } c_2 \rightarrow S''/b}$$

Boucle

La sémantique des *boucles* est très simple :

Boucle (si)

$$\frac{S/c \rightarrow S'/\text{true} \quad S'/i; \text{while } c \text{ do } i \rightarrow S''}{S/\text{while } c \text{ do } i \rightarrow S''}$$

Boucle (sinon)

$$\frac{S/c \rightarrow S'/\text{false}}{S/\text{while } c \text{ do } i \rightarrow S'}$$

Erreurs, non-terminaison

Si on *ne peut pas* prouver $p \rightarrow$, cela signifie que l'exécution de p rencontre une erreur *ou* ne termine pas.

Pour distinguer ces deux situations, on pourrait définir un jugement $p \rightarrow \text{error}$ et/ou un jugement $p \rightarrow \infty$. (Exercice...)

Le typage *interdit certaines erreurs*, mais pas toutes (tableau nil, dépassement de bornes de tableau).

Habituellement, le typage *ne garantit pas la terminaison*. Toutefois, dans certains langages (p. ex. Coq), le typage garantit la terminaison. Dans ce cas, le langage n'est pas Turing-complet.

Interaction avec l'extérieur

Les jugements $p \rightarrow$ et $p \rightarrow \text{error}$ et $p \rightarrow \infty$ ne disent pas quelles valeurs sont lues et affichées par le programme (via `writeln` et `readln`).

On pourrait définir un jugement $p \xrightarrow{H}$ où l'historique H est une suite d'*événements* de lecture et d'écriture. (Exercice...)

Idem pour les jugements $p \xrightarrow{H} \text{error}$ et $p \xrightarrow{H} \infty$. Dans ce dernier, la suite H peut être infinie. (Exercice...)

Syntaxes concrète et abstraite

Syntaxe abstraite de Pseudo-Pascal

Sémantique opérationnelle

Interprétation

Sémantiques et interprètes

Cette sémantique constitue un interprète de Pseudo-Pascal exprimé dans un langage de programmation *logique*, où la construction élémentaire est la règle de déduction.

Un interprète de Pseudo-Pascal exprimé en OCaml constitue *également* une sémantique, quoique *moins élémentaire*, car s'appuyant sur la sémantique complexe d'OCaml.

En fait, écrire en OCaml un interprète de Pseudo-Pascal est facile précisément parce qu'OCaml *contient* tout Pseudo-Pascal : fonctions récursives, allocation dynamique de mémoire, tableaux, variables modifiables, etc.

Vers un interprète en OCaml

Les environnements G et E associent des valeurs aux noms de variables et sont *modifiables*. On définit donc :

```
type environment = value ref StringMap.t
```

Le tas H est *simulé* par le tas OCaml et n'est donc pas explicitement représenté.

Un environnement D *immuable* associe des définitions aux noms de procédures ou fonctions. On pose :

```
type definitions = PP.procedure StringMap.t
```

Vers un interprète en OCaml

Les fonctions centrales de l'interprète sont paramétrées par D , G , et E . On définit une abréviation :

```
type 'a interpreter =  
  definitions → environment → environment → 'a
```

Les fonctions qui interprètent expressions, conditions et instructions ont alors les types :

```
val interpret_expression : (PP.expression → value) interpreter  
val interpret_condition : (PP.condition → bool) interpreter  
val interpret_instruction : (PP.instruction → unit) interpreter
```

Quelques modules utiles

Voici les modules du “petit compilateur” dont vous aurez besoin en TD :

- ▶ **MIPSOps** définit entre autres les opérateurs binaires de Pseudo-Pascal;
- ▶ **Primitive** définit les opérations primitives de Pseudo-Pascal;
- ▶ **PP** définit la syntaxe abstraite de Pseudo-Pascal;
- ▶ **Integer** définit les opérations courantes sur les entiers 32 bits;
- ▶ **StringMap** définit des tables d'association dont les clefs sont des chaînes de caractères.

Pourquoi écrire un interprète?

Un interprète *examine* le programme source et le *simule* en même temps.

Un compilateur examine le programme source à “*compile time*” et engendre des instructions machine qui le simuleront à “*run time*”.

Cet *étagement* (ou “*staging*”) permet une plus grande efficacité.

Écrire un interprète n'aide pas écrire à un compilateur, sauf pour :

- ▶ mieux *comprendre* la sémantique du langage source;
- ▶ *tester* le compilateur par comparaison avec l'interprète.