

Compilation

Cours n°2: Syntaxe, sémantique et interprétation de Pseudo-C

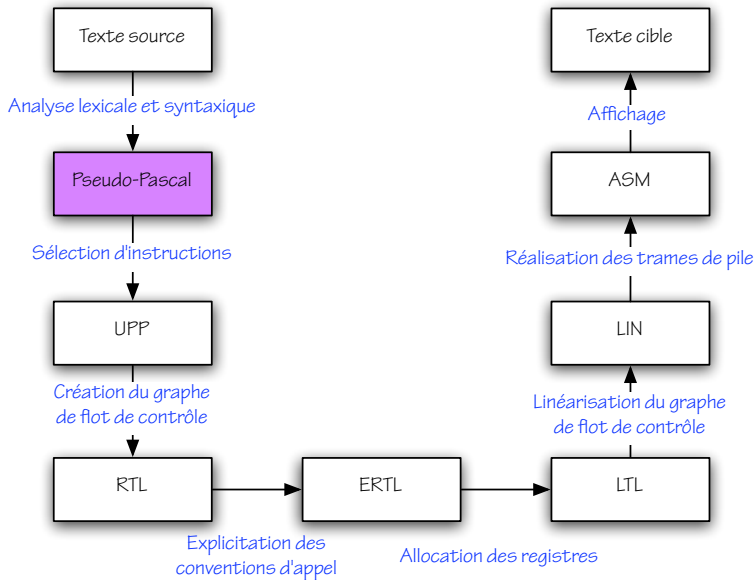
Sandrine Blazy
(d'après le cours de François Pottier)

ensiie - 2^e année

3 novembre 2008

**école nationale supérieure d'informatique
pour l'industrie et l'entreprise**

ensiie



1 Syntaxes concrète et abstraite

Syntaxe abstraite de Pseudo-C

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 ».

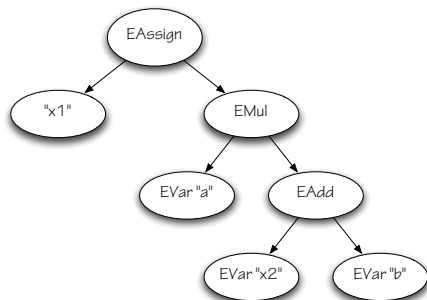
En termes de *syntaxe abstraite*, un programme est un *arbre*. La syntaxe abstraite définit la structure idéale (mathématique) des programmes.

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 est transformé par *analyse syntaxique* en un arbre de syntaxe abstraite (cf. cours de Lex & Yacc).

Syntaxe abstraite sur machine

Les arbres de syntaxe abstraite sont définis en Objective Caml 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 Objective Caml 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

2 Syntaxe abstraite de Pseudo-C

Sémantique opérationnelle

Interprétation

Catégories

Pour définir Pseudo-C, 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-C 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.

$$\begin{array}{l} k ::= \text{constantes} \\ | b \text{ constante booléenne} \\ | n \text{ constante entière} \end{array}$$

Opérateurs

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

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

Opérations primitives

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

π	::=	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.

φ	::=	cible d'un appel
		π opération primitive
		f procédure ou fonction définie par l'utilisateur

Expressions, conditions et instructions

Pseudo-C 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
$\text{uop } e$	application d'un opérateur unaire
$e \text{ bop } e$	application d'un opérateur binaire
$\varphi(e \dots e)$	appel de fonction
$e[e]$	lecture dans un tableau
$\text{new array of } \tau [e]$	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
$\text{var } x : \tau \dots x : \tau$	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-C 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-C est *plus expressif* que l'assembleur MIPS, mais reste tout de même *peu expressif* comparé à Java ou Objective Caml.

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

De la syntaxe à la sémantique

Nous avons défini la *structure* des programmes Pseudo-C, 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-C

3 Sémantique opérationnelle

Interprétation

Jugement principal

La sémantique de Pseudo-C 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 ».

La *non-terminaison* et les *erreurs* ne sont pas distinguées dans cette sémantique. Il serait possible de le faire.

Le *typage* interdit une grande partie des erreurs, mais pas toutes (tableau nil, dépassement de bornes de tableau).

Les *effets de bord* ne sont pas modélisés ici. Il serait possible de le faire.

Jugements auxiliaires

Ce jugement principal est défini en termes de trois *jugements auxiliaires* dont voici la forme :

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

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* (cf. cours de 1A).

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

$$\begin{array}{ccc} \text{ZÉRO} & \text{SUCESSEUR PAIR} & \text{SUCESSEUR IMPAIR} \\ \text{Pair } 0 & \frac{\text{Impair } n}{\text{Pair } n + 1} & \frac{\text{Pair } n}{\text{Impair } n + 1} \end{array}$$

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

Règles

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

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

Voici quelques-unes de ces règles...

Constante

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

$$\begin{array}{c} \text{CONSTANTE} \\ S/k \longrightarrow S/k \end{array}$$

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

Accès aux variables

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

$$\frac{\text{VARIABLE LOCALE} \\ x \in \text{dom}(E)}{G, H, E/x \longrightarrow G, H, E/E(x)}$$

$$\frac{\text{VARIABLE GLOBALE} \\ x \in \text{dom}(G) \setminus \text{dom}(E)}{G, H, E/x \longrightarrow G, H, E/G(x)}$$

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

Exemple : Pour $E = \{x \mapsto 1; y \mapsto 6; z \mapsto 8\}$, alors $\text{dom}(E) = \{x; y; z\}$

Affectation

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

AFFECTATION: VARIABLE LOCALE

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

AFFECTATION: VARIABLE GLOBALE

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

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

$$S/e \longrightarrow S'/v$$

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

OPÉRATEUR BINAIRE

$$S/e_1 \longrightarrow S'/v_1 \quad S'/e_2 \longrightarrow S''/v_2$$

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

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

Accès aux tableaux

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

LECTURE DANS UN TABLEAU

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

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 :

$$\begin{array}{l} \text{ALLOCATION D'UN TABLEAU} \\ S/e \longrightarrow G', H', E'/n \quad n \geq 0 \\ \ell \# H' \quad H'' = H'[\ell \mapsto \text{default}(\tau)^n] \\ \hline S/\text{new array of } \tau [e] \longrightarrow G', H'', E'/\ell \end{array}$$

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-C 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

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

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

L'appel se fait *par valeur*.

Conjonction

En Pseudo-C, l'évaluation de la conjonction et de la disjonction est « à court-circuit » :

$$\begin{array}{l} \text{CONJONCTION (SI)} \\ S/c_1 \longrightarrow S'/\text{false} \\ \hline S/c_1 \text{ and } c_2 \longrightarrow S'/\text{false} \end{array}$$

$$\begin{array}{l} \text{CONJONCTION (SINON)} \\ S/c_1 \longrightarrow S'/\text{true} \\ S'/c_2 \longrightarrow S''/b \\ \hline S/c_1 \text{ and } c_2 \longrightarrow S''/b \end{array}$$

Boucle

La sémantique des *boucles* est très simple :

BOUCLE (SI)

$$\frac{S/c \longrightarrow S'/\text{true} \quad S'/e; \text{while } c \text{ do } e \longrightarrow S''}{S/\text{while } c \text{ do } e \longrightarrow S''}$$

BOUCLE (SINON)

$$\frac{S/c \longrightarrow S'/\text{false}}{S/\text{while } c \text{ do } e \longrightarrow S'}$$

Syntaxes concrète et abstraite

Syntaxe abstraite de Pseudo-C

Sémantique opérationnelle

4 Interprétation

Sémantiques et interprètes

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

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

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

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* ». Il y a « *staging* », d'où plus grande efficacité.

Écrire un interprète ne nous aide pas écrire à un compilateur, sauf pour :

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