Chapitre 5

Sémantique dénotationnelle

Dans les trois chapitres précédents, nous nous sommes essentiellement intéressés aux aspects syntaxiques des langages de programmation. Dans ce chapitre et dans les suivants, nous nous intéresserons plutôt au *sens* des programmes, et tiendrons pour résolues leurs difficultés d'ordre syntaxique. En particulier, lorsque nous parlerons des constructions syntaxiques des programmes, nous aurons en tête les *arbres syntaxiques* correspondants plutôt que les suites de caractères avec lesquels ils sont écrits.

Le sens d'un programme est avant tout son comportement, ce qu'il calcule et comment il le calcule. Lorsque nous écrivons des programmes, nous avons une idée (plus ou moins) précise de ce qu'il va faire, du sens qu'il a. Toute description de l'action d'un programme repose nécessairement sur le sens précis des différentes constructions syntaxiques du langage dans lequel est écrit ce programme.

Écrire un interprète pour un langage de programmation, c'est exprimer le sens de ce langage par un programme particulier (écrit dans un autre langage). Écrire un compilateur pour un langage, c'est exprimer le sens du langage source en fonction de celui du langage cible (assembleur ou autre langage de haut niveau). La fidélité de la traduction repose aussi bien sûr sur le compilateur, et donc sur le langage dans lequel celui-ci est écrit.

Ces considérations n'ont pas d'autre but que de montrer que la précision de la description d'un langage repose bien sûr sur ce qu'on décrit, mais aussi sur le langage que l'on utilise pour exprimer cette description. Nous utiliserons dans la suite des descriptions formelles du sens des langages de programmation. Lorsque nous voudrons effectuer des implémentations de ces descriptions, nous passerons alors d'un langage mathématique à un langage informatique.

5.1 Le sens du sens

Une sémantique cohérente, précise et non-ambiguë, a de nombreuses utilités. Elle peut être utilisée comme une définition du langage dans un but de standardisation, par exemple. Elle peut aussi bien sûr servir de référence aux programmeurs, qui ne souhaitent pas avoir à exécuter un programme pour le comprendre. La plupart des actions qui impliquent un raisonnement mathématique

à propos des programmes nécessite une sémantique formelle du langage. Les implémenteurs d'interprètes ou de compilateurs utilisent la sémantique du langage comme référence dans le but d'éviter la construction d'implémentations divergentes du même langage.

Certains langages peuvent être implantés directement à partir d'une sémantique formelle, qui est alors dite *exécutable*. Enfin, la possibilité d'une sémantique formelle pour un langage est probablement un gage de qualités telles la simplicité ou l'orthogonalité des concepts.

5.2 Sémantiques

Nous allons, dans ce chapitre et le suivant, distinguer deux façons assez différentes d'exprimer le sens d'un langage. Dans ce chapitre, nous verrons comment exprimer *ce que représente un programme* : c'est ce qu'on appelle sa *dénotation* : un objet mathématique. Au chapitre suivant, nous nous attacherons à décrire le *comportement des programmes*, plutôt que leur essence.

Nous n'aborderons pas ici une façon de décrire le sens des programmes : la *sémantique axiomatique*, où le sens d'un programme est donné en fonction de son impact sur certaines propriétés. Typiquement, pour un programme S, on écrira $\{p\}S\{q\}$ pour indiquer que si la propriété p était vraie avant l'exécution de S, alors q est vraie ensuite. La propriété p est appelée *pré-condition*, alors que q est appelée *post-condition*. Ce type de description est utile lorsqu'on veut prouver des propriétés de correction partielle ou totale (c'est-à-dire incluant la terminaison) de programmes. Elle semble moins utilisable pour produire une définition de langage de programmation.

5.3 Le langage PCF

Afin que nos descriptions sémantiques ne restent pas abstraites, nous allons utiliser un mini langage de programmation, qui n'est composé que de quelques constructions syntaxiques, mais qui a assez de puissance d'expression pour exprimer des algorithmes arbitraires. Nous appelons PCF[1] ce langage. On le retrouve quelquefois dans la littérature sous le nom de « Mini-ML ».

La présentation que nous en donnons ci-dessous reste assez abstraite, au sens où nous ne précisons pas quelles sont les constantes et fonctions primitives du langage, et nous restons assez vagues quant à sa syntaxe concrète.

Syntaxe de PCF La syntaxe du langage est donnée sous la forme d'une grammaire. Les catégories lexicales Const et Id correspondent aux constantes (entières, booléennes, *etc.*), mais incluent aussi les primitives telles les opérations arithmétiques ou la concaténation de chaînes de caractères. Le langage PCF est un langage d'expressions (tout comme le langage OCaml) et on notare Exp la catégorie syntaxique (le non terminal) correspondante. On notera par la suite les éléments de ces différentes catégories par des méta-variables particulières (e, e_1, e_2, *etc.* pour les expressions, par exemple).

[1] *Programming language for Computable Functions* : un langage introduit par Gordon Plotkin en 1977 pour étudier les relations entre les sémantiques que nous présentons dans ce chapitre et le suivant.

$c \in \text{Const}$ Constantes, incluant les fonctions primitives
$x \in \text{Id}$ Identificateurs
$e \in \text{Exp}$ Expressions

Les expressions de PCF sont composées de constantes, primitives, de liaisons locales et de fonctions.

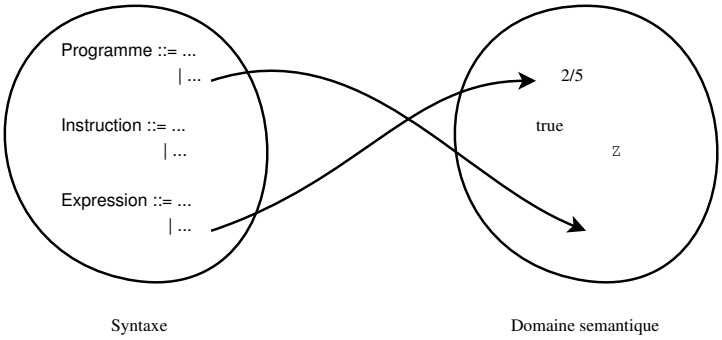
$$\begin{array}{rcl} e & ::= & c \mid x \mid e_1 e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ & \mid & e_1 + e_2 \mid e_1 = e_2 \\ & \mid & \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid & \mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2 \\ & \mid & \mathsf{fun}\ x\ \rightarrow\ e \end{array}$$

On reconnaît là le noyau des langages fonctionnels comme OCaml. Un exemple de programme dans ce langage pourrait être :

$\mathsf{letrec}\ \text{fact} = \mathsf{fun}\ n\ \rightarrow\ \mathsf{if}\ n \leq 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * \text{fact}(n-1)\ \mathsf{in}$
$\text{fact}(5)$

5.4 Sémantique dénotationnelle de PCF

La sémantique dénotationnelle d'un langage de programmation consiste à définir ce qu'est la *dénotation* des programmes de ce langage. La dénotation d'un programme est un objet mathématique, donc abstrait, mais qui est précisément défini. La sémantique dénotationnelle voit généralement un programme comme une fonction (au sens mathématique du terme) des entrées vers les sorties, par exemple, ou, plus généralement, d'un *domaine* dans un autre. Les définitions de sémantique des langages sont en général *compositionnelles*, c'est-à-dire qu'elles définissent la sémantique d'un programme en fonction des sémantiques des éléments qui le composent. En général, la sémantique est définie inductivement, en suivant la structure du langage (ses arbres symtaxiques).



5.4.1 Domaines

Nous serons dans ce chapitre assez approximatifs dans la construction des domaines mentionnés ci-dessus. En particulier, nous construisons ci-dessous un domaine qui contient l'espace des fonctions du domaine vers lui-même. Si nous voulions être précis, nous indiquerions que ces domaines sont composés d'éléments dont certains représentent l'indéfini (pour représenter par exemple le

« résultat » d'un calcul qui ne termine pas). Les éléments de nos domaines sont donc partiellement indéfinis, ce qui permet de doter chacun de ces domaines d'un ordre partiel (« moins défini que »), et les domaines sont construits de telle sorte que chacun contienne un élément indéfini (noté $\perp$, inférieur à tous les autres) et que toute chaîne strictement croissante ait une borne supérieure. Cela dote les domaines considérés d'une structure dite d'*ordre partiel complet*, ou CPO (pour *Complete partial Order*. Les fonctions que l'on considèrera ici sont des fonctions monotones f telles que pour toute chaîne C, on ait $\mathsf{sup}(f(C)) = f(\mathsf{sup}(C))$. Par souci de simplicité, nous ignorerons par la suite cet aspect des choses.

Pour le langage PCF, nous construisons un domaine D, récursivement, à partir des domaines pour les constantes, et contenant les fonctions du domaine vers lui-même.

- INT : le domaine des entiers
- BOOL = { *true*, *false* }
- D = INT + BOOL + FUN : le domaine des valeurs
- FUN = D $\rightarrow$ D : les fonctions sur D

Nous aurons besoin, pour la construction des fonctions sémantiques, d'un domaine noté ENV, qui contiendra des fonctions associant des valeurs de D à des identificateurs de Id :

- ENV = Id $\rightarrow$ D : le domaine des environnements

Si ρ est un environnement, $\rho' = \rho \oplus [x \mapsto d]$ est l'environnement défini par :

$\rho'[\![x]\!] = d$
$\rho'[\![y]\!] = \rho[\![y]\!]$, for $y \neq x$

Ci-dessus, et par la suite, lorsque qu'une fonction reçoit un argument de nature syntaxique (un identificateur ou une expression, par exemple), on utilise des parenthèses spéciales $[\![\_]\!]$ pour bien marquer ce fait.

5.4.2 Fonctions sémantiques

Le langage PCF tel qu'il est décrit ici nécessite essentiellement une seule fonction sémantique, qui donne le sens dans D d'une expression e. Une expression e contenant des identificateurs *libres*, c'est-à-dire sans le let ou le fun qui les introduit, la sémantique d'une expression est, en toute généralité, paramétrée par celle de ses variables libres. La fonction sémantique des expressions, notée $\mathcal{E}$, a la forme suivante :

$$\mathcal{E} : \text{Exp} \rightarrow \text{ENV} \rightarrow \text{D}$$

La définition de $\mathcal{E}$ s'écrit par cas selon la structure de l'expression considérée :

$\mathcal{E}[\![c]\!]\rho = d_c$ pour chaque $c \in$ Const, où d_c est la valeur de D dénotée par c
$\mathcal{E}[\![x]\!]\rho = \rho[\![x]\!]$, pour $x \in$ Id
$\mathcal{E}[\![e_1 + e_2]\!]\rho = \mathcal{E}[\![e_1]\!]\rho + \mathcal{E}[\![e_2]\!]\rho$
$\mathcal{E}[\![e_1\ e_2]\!]\rho = (\mathcal{E}[\![e_1]\!]\rho)(\mathcal{E}[\![e_2]\!]\rho)$
$\mathcal{E}[\![e_1 = e_2]\!]\rho = \mathit{true}$ si $\mathcal{E}[\![e_1]\!]\rho = \mathcal{E}[\![e_2]\!]\rho$
$= \mathit{false}$ dans le cas contraire
$\mathcal{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\rho =$

si $\mathcal{E}[\![e_1]\!]\rho = \mathit{true}$ alors $\mathcal{E}[\![e_2]\!]\rho$ sinon $\mathcal{E}[\![e_3]\!]\rho$
$\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho = \mathcal{E}[\![e_2]\!](\rho \oplus [x \mapsto \mathcal{E}[\![e_1]\!]\rho])$
$\mathcal{E}[\![\mathsf{letrec}\ f = e_1\ \mathsf{in}\ e_2]\!]\rho = \mathcal{E}[\![e_2]\!]\rho'$
où $\rho' = \rho \oplus [f \mapsto \mathcal{E}[\![e_1]\!]\rho']$ *(Noter que ρ' est ici défini récursivement)*
$\mathcal{E}[\![\mathsf{fun}\ x\ \rightarrow\ e]\!]\rho = f$ telle que $f(v) = \mathcal{E}[\![e]\!](\rho \oplus [x \mapsto v])$

Le seul cas un peu complexe de cette définition est le cas des définitions récursives, que l'on fait correspondre à des valeurs qui utilisent un environnement défini récursivement. Sans entrer dans le détail, c'est la structure de CPO des domaines que nous considérons qui permet d'écrire une telle définition.

5.5 Sémantique dénotationnelle des affectations

Si maintenant nous enrichissons notre langage avec une mémoire et des instructions modifiant cette mémoire, nous obtenons de nouveaux domaines et devons définir les fonctions sémantiques correspondantes.

Syntaxe

$e \in$ Exp, défini par $e\ ::=\ \ldots \mid s; e$
$s \in$ Stm, défini par $s\ ::=\ x := e \mid s_1; s_2$

La catégorie Stm est celle des instructions : affectation et séquence d'instructions. Les expressions de Exp peuvent maintenant être précédées d'une instruction (ou d'une séquence d'instructions).

Domaines sémantiques Les domaines sémantiques doivent, dès lors, représenter un peu plus finement les variables. En effet, puisque celles-ci sont devenues modifiables, il est nécesssaire de distinguer leur addresse, à laquelle il est fait référence dans la partie gauche d'une affectation, de leur valeur, qui est la valeur stockée à cette adresse. À un nom de variable, on associe donc d'abord une référence ou adresse (*location* en anglais). La valeur d'une variable pourra ensuite être obtenue depuis cette référence.

Deux nouveaux domaines apparaissentdonc : LOC, pour les référence, et STORE qui associe une valeur à une référence. Le domaine D, quant à lui, ne change pas en apparance, mais le sous-domaine FUN change de nature : puisque l'application d'une fonction produit un résultat et une modification de la mémoire, les fonction reçoivent en argument supplémentaire une mémire, et produisent un couple composé d'un résultat et d'une nouvelle mémoire.

D = INT + BOOL + FUN
FUN = (D × STORE) → (D × STORE)
LOC, le domaine des références (*locations*)
STORE = LOC → D le domaine des références initialisées (on parlera de « nouvelles » références pour mentionner des références inutilisées)
ENV = ID → LOC

On voit clairement que l'accès à la valeur d'une variable par ENV procède en deux temps.

Fonctions sémantiques À la fonction sémantique $\mathcal{E}$ des expressions, s'ajoute maintenant une fonction sémantique $\mathcal{S}$ pour les instructions :

$\mathcal{E}$: Exp $\to$ ENV $\to$ STORE $\to$ (D $\times$ STORE)

$\mathcal{S}$: Stm $\to$ ENV $\to$ STORE $\to$ STORE

Les instructions, et donc aussi les expressions, peuvent modifier la mémoire : c'est pourquoi les deux fonctions sémantiques correspondantes reçoivent un argument de type STORE. Puisque les instructions ne produisent pas de valeur, seule une mémoire est produite en résultat. Les expressions, quant à elles, produisent à la fois une valeur (de D) et une mémoire : c'est pourquoi la fonction sémantique $\mathcal{E}$ produit un couple de (D $\times$ STORE) en résultat. Les catégories syntaxiques Exp et Stm étant définies de façon mutuellement récursive, il est naturel que les fonctions sémantiques $\mathcal{E}$ et $\mathcal{S}$ le soient aussi.

Definitions de $\mathcal{E}$ et $\mathcal{S}$

$\mathcal{E}[\![c]\!]\rho\sigma = (d_c, \sigma)$ pour chaque $c \in$ Const, où d_c est la valeur représentant c dans D

$\mathcal{E}[\![x]\!]\rho\sigma = (\sigma(\rho[\![x]\!]), \sigma)$, pour $x \in$ Id

$\mathcal{E}[\![e_1 + e_2]\!]\rho\sigma =$
soit $(n_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
soit $(n_2, \sigma_2) = \mathcal{E}[\![e_2]\!]\rho\sigma_1$
$(n_1 + n_2, \sigma_2)$

$\mathcal{E}[\![e_1\ e_2]\!]\rho\sigma =$
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
$v_1(\mathcal{E}[\![e_2]\!]\rho\sigma_1)$

$\mathcal{E}[\![e_1 = e_2]\!]\rho\sigma =$
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
soit $(v_2, \sigma_2) = \mathcal{E}[\![e_2]\!]\rho\sigma_1$
si $v_1 = v_2$ alors (true, σ_2) sinon (false,σ_2)

$\mathcal{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\rho\sigma =$
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
if $v_1 = \mathit{true}$ then $\mathcal{E}[\![e_2]\!]\rho\sigma_1$ else $\mathcal{E}[\![e_3]\!]\rho\sigma_1$

$\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma =$
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!]\rho\sigma$
soit l_1 une nouvelle référence
$\mathcal{E}[\![e_2]\!](\rho \oplus [x \mapsto l_1])(\sigma_1 \oplus [l_1 \mapsto v_1])$

$\mathcal{E}[\![\mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma =$
soit l_1 une nouvelle référence
soit $(v_1, \sigma_1) = \mathcal{E}[\![e_1]\!](\rho \oplus [x \mapsto l_1])(\sigma)$

$\mathcal{E}[\![e_2]\!](\rho \oplus [x \mapsto l_1])(\sigma_1 \oplus [l_1 \mapsto v_1])$

Notons que $\sigma(l_1)$ est indéfinie : la raison en est que le calcul de e_1 ne doit pas utiliser x.

$\mathcal{E}[\![\mathsf{fun}\ x\ \rightarrow\ e]\!]\rho\sigma = f$
telle que $f(v, \sigma') =$
soit l_x une nouvelle référence
$\mathcal{E}[\![e]\!](\rho \oplus [x \mapsto l_x])(\sigma' \oplus [l_x \mapsto v])$

$\mathcal{E}[\![s; e]\!]\rho\sigma =$
soit $\sigma' = \mathcal{S}[\![s]\!]\rho\sigma$
$\mathcal{E}[\![e]\!]\rho\sigma'$

$\mathcal{S}[\![x := e]\!]\rho\sigma =$
soit $l_x = \rho[\![x]\!]$
soit $(v, \sigma') = \mathcal{E}[\![e]\!]\rho\sigma$
$\sigma' \oplus [l_x \mapsto v]$

$\mathcal{S}[\![s_1; s_2]\!]\rho\sigma =$
soit $\sigma_1 = \mathcal{S}[\![s_1]\!]\rho\sigma$
$\mathcal{S}[\![e_2]\!]\rho\sigma_1$

Les seuls cas intéressants des définitions ci-dessus sont ceux où on utlise des variables, ou bien pour les déclarer, ou alors pour les modifier. Toutes les variables sont traitées de la même manière : elles représentent toutes des références, et sont, à ce titre, modifiables. Pour avoir des variables qui soient similaires à celles d'OCaml, c'est-à-dire des constantes dont la valeur peut être une structure de données modifiable, il faut d'abord avoir des données structurées dans le langage, représentées par des blocs dans la mémoire (par exemple, en faisant de STORE une collection d'enregistrements) et – à défaut d'un système de types – de marquer certains champs de ces blocs comme modifiables. Cette modification ne pose aucun problème technique.

5.6 Sémantique dénotationnelle de goto

Tentons maintenant de donner un sens à la fameuse instruction **goto**, qui permet d'interrompre le déroulement d'un programme pour reprendre le calcul dans un autre contexte. La modélisation de ces sauts nécessite d'identifier et de représenter les contextes où on peut ainsi « sauter ». Les destinations de sauts sont identifiées par des étiquettes (*labels*, en anglais).

Une destination de saut est un calcul à reprendre. Nous considérerons ici une forme limitée de **goto**, qui ne permet que de sauter vers un label que depuis l'intérieur de la « portée » de ce label, c'est-à-dire dans la séquence d'instructions qui suit ce label. Un contexte de reprise est donc nécessairement situé entre deux instructions, et nous représenterons un tel contexte par une fonction de type STORE → STORE : on arrive dans ce contexte avec un certain état mémoire, et on reprend une exécution qui produira *in fine* un nouvel état.

Expliciter ainsi *un* contexte de calcul et autoriser de remplacer le calcul courant par un tel

nouveau contexte oblige à expliciter *tous* les contextes de calculs : en effet, pour pouvoir remplacer le reste du calcul par un autre que le reste « naturel », il faut que chaque calcul soit paramétré par ce qui reste à faire – la suite, ou la *continuation* –, pour pouvoir changer cette continuation par une autre. Nos fonctions sémantiques vont donc recevoir un paramètre supplémentaire : la continuation du calcul, ce qui nous amènera de façon naturelle à expliciter l'ordre d'évaluation des différentes sous-expressions de notre langage.

Les continuations vont prendre deux formes, selon qu'il s'agisse de continuations de calculs d'expressions ou de continuations de calculs d'instructions. Les continuations d'instructions appartiendront à Cc (pour *Command continuation*) :

$$\text{Cc} = \text{STORE} \to \text{STORE}$$

et représentent le reste du calcul après l'exécution d'une instruction. Les continuations d'expressions appartiendront à Ec :

$$\text{Ec} = (\text{D} \times \text{STORE}) \to (\text{D} \times \text{STORE})$$

et représenteront le reste du calcul après celui d'une expression.

Fonctions sémantiques

$\mathcal{E} : \text{Exp} \to \text{ENV} \to \text{STORE} \to \text{Ec} \to (\text{D} \times \text{STORE})$

$\mathcal{S} : \text{Stm} \to \text{ENV} \to \text{STORE} \to \text{Cc} \to \text{STORE}$

Les valeurs fonctionnelles reçoivent elles aussi un paramètre supplémentaire : une continuation. En effet, elles l'utilisent pour la passer à la sémantique de leur corps qui l'utilisera comme continuation.

$$\text{FUN} = (\text{D} \times \text{STORE}) \to \text{Ec} \to (\text{D} \times \text{STORE})$$

Les labels sont de simples identificateurs dont les valeurs (dans la mémoire) sont des continuations d'instructions. On aura donc :

$$\text{D} = \text{INT} + \text{BOOL} + \text{FUN} + \text{Cc}$$

Syntaxe On étend la syntaxe des instructions en :

$s \in \text{Stm}$, et $s ::= x := e \mid s_1; s_2 \mid lab : s \mid \texttt{goto}\ lab$

où

- $lab \in \text{Id}$
- dans "$lab : s$", s est dans la portée de lab, et peut donc contenir "goto lab".

Par souci de simplicité, nous utiliserons ici une version restreinte de **goto**. Des formes plus générales de **goto** nécessiteraient un traitement plus complexe, sans pour autant présenter de réelle nouveauté par rapport à ce que nous présentons ici.

Fonctions sémantiques

$\mathcal{E}[\![c]\!]\rho\sigma k = k\ (d_c, \sigma)$ pour $c \in \text{Const}$, où d_c est la valeur de D correspondant à c

$\mathcal{E}[\![x]\!]\rho\sigma k = k\ (\sigma(\rho[\![x]\!]), \sigma)$, pour $x \in \text{Id}$

$\mathcal{E}[\![e_1 + e_2]\!]\rho\sigma k =$
$\quad \mathcal{E}[\![e_1]\!]\rho\sigma k_1$
$\quad$ où k_1 est la continuation définie par $k_1(n_1, \sigma_1) = \mathcal{E}[\![e_2]\!]\rho\sigma_1(k_2\ n_1)$
$\quad$ où $k_2\ n_1\ (n_2, \sigma_2) = k(n_1 + n_2, \sigma_2)$

$\mathcal{E}[\![e_1\ e_2]\!]\rho\sigma k =$
$\quad \mathcal{E}[\![e_1]\!]\rho\sigma k_1$
$\quad$ où $k_1(v_1, \sigma_1) = \mathcal{E}[\![e_2]\!]\rho\sigma_1(k_2\ v_1)$
$\quad$ où $k_2\ v_1\ (v_2, \sigma_2) = v_1\ (v_2, \sigma_2)\ k$

$\mathcal{E}[\![e_1 = e_2]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{letrec}\ x = e_1\ \mathsf{in}\ e_2]\!]\rho\sigma k =$ ... laissé en exercice

$\mathcal{E}[\![\mathsf{fun}\ x\ \rightarrow\ e]\!]\rho\sigma k =$
$\quad k(f, \sigma)$
$\quad$ où f est la fonction définie par $f\ (v, \sigma')\ k' = \mathcal{E}[\![e]\!](\rho \oplus [x \mapsto l_x])(\sigma' \oplus [l_x \mapsto v])k'$
$\quad$ où l_x est une nouvelle référence

$\mathcal{E}[\![s; e]\!]\rho\sigma k =$
$\quad$ soit k' définie par $k'(\sigma') = \mathcal{E}[\![e]\!]\rho\sigma' k$
$\quad \mathcal{S}[\![s]\!]\rho\sigma k'$

$\mathcal{S}[\![lab : s]\!]\rho\sigma k =$
$\quad$ soit l une nouvelle référence
$\quad$ soit k' définie récursivement par $k'(\sigma') = \mathcal{S}[\![s]\!](\rho \oplus [lab \mapsto l])(\sigma' \oplus [l \mapsto k'])k$
$\quad k'\sigma$

$\mathcal{S}[\![x := e]\!]\rho\sigma k =$
$\quad$ soit k' définie par $k'(v', \sigma') = k(\sigma' \oplus [\rho[\![x]\!] \mapsto v'])$
$\quad \mathcal{E}[\![e]\!]\rho\sigma k'$

$\mathcal{S}[\![s_1; s_2]\!]\rho\sigma k =$
$\quad \mathcal{S}[\![s_1]\!]\rho\sigma k'$
$\quad$ où k' est la continuation définie par $k'(\sigma') = \mathcal{S}[\![s_2]\!]\rho\sigma' k$

$\mathcal{S}[\![\mathsf{goto}\ lab]\!]\rho\sigma k =$
$\quad k'\sigma$
$\quad$ où $k' = \sigma(\rho[\![lab]\!])$

Ce qui conclut la description de la sémantique dénotationnelle de PCF avec affectation et **goto**. Les sémantiques que nous avons données précédemment, sans continuations, sont appelées *sémantiques directes*.

5.7 TP : mise en oeuvre de sémantique dénotationnelle en OCaml

L'objectif du TP est d'implémenter les sémantiques données ci-dessus : d'abord une sémantique directe, que vous mettrez en œuvre en deux temps : en ne traitant pas les instructions dans un premier temps (`raise (Failure "pas implémenté")`), puis en en ajoutant le traitement. Ensuite, la sémantique à continuation vous permettra (si le temps le permet) de traiter les **goto**.

Les éléments permettant de gérer les aspects syntaxiques de PCF et le programme principal vous sont donnés en début de TP. Il ne vous reste qu'à définir la ou les fonctions sémantiques pour obtenir un interprète complet du langage PCF.

La syntaxe concrète du langage est donnée informellement par la grammaire suivante :

Expr ::=
  FUN IDENT "→" *Expr*
| *LET IDENT* "=" *Expr IN Expr*
| *LETREC IDENT* "=" *Expr IN Expr*
| *IF Expr THEN Expr ELSE Expr*
| *Comparison*

Comparison ::=
  Addition ("=" *Addition*)*

Addition ::=
  Multiplication (("+" | "-") *Multiplication*)*

Multiplication ::=
  Application (("*" | "/") *Application*)*

Application :=
  Atomic Atomic*

Atomic ::=
  INT
| *IDENT*
| *TRUE* | *FALSE*
| "(" *Expr* ")"
| *DO* "{" *Statements RETURN Expr* "}"

Statements ::=
  IDENT ":=" *Expr* ";" *Statements*

```
| IDENT ":" Statements
| GOTO IDENT ";"
| IF Expr GOTO IDENT ";"
| (* rien *)
```

Un exemple de programme appartenant à ce langage :

```
let not = fun b →
  if b = true then false
  else true in
let factiter = fun n →
  if n = 0 then 1 else
  let res = 1 in
  do {
      loop:
        res := n * res;
        n := n − 1;
        if not(n = 0) goto loop;
        return res
     } in
letrec factrec =
  fun m → if m=0 then 1 else m * factrec(m − 1) in
factrec 10 = factiter 10 .
```

Les arbres de syntaxe abstraite sont donnés dans un fichier `ast.mli` par :

```
type const =
  Integer of int
| Boolean of bool
;;

type exp =
  Const of const
| Var of string
| App of exp * exp
| If of exp * exp * exp
| Binop of string * exp * exp
| Let of string * exp * exp
| Letrec of string * exp * exp
| Fun of string * exp
| Seqexp of statm * exp

and statm =
  Assign of string * exp
```

```
| Label of string * statm
| Goto of string
| IfGoto of exp * string
| Seq of statm * statm
| Skip
;;
```

On notera que l'interface `ast.mli` n'a pas d'implémentation correspondante : cette dernière n'est en effet pas nécessaire lorsque le module se limite à des définitions de types ou d'exceptions.

On donne aussi quelques fonctions de gestion des environnements. Les environnements sont implémentés exactement comme présentés dans la sémantique dénotationnelle : comme des fonctions qui associent une valeur à un identificateur (une chaîne de caractères). On implémente de façon similaire la mémoire. La fonction `extend` étend un environnement par une nouvelle liaison pour produire un nouvel environnement (`extend env x v`).

```
val init_env : string → α;; (* environnement initial *)
val new_loc : unit → int;;
val init_store : α → β;;
val extend : (α → β) → α → β → α → β;;
```

Enfin, le programme principal `main.ml` utilise l'interface `semant.mli` pour en utiliser le type `Semant.semval` des valeurs, ainsi que la fonction d'évaluation `Semant.eval`.

L'interface `semant.mli`, dans la version la plus complète (sémantique à continuations) est la suivante :

```
type location = int;;
type environment = string → location;;
type store = location → semval
and econt = semval * store → semval * store
and scont = store → store
and semval =
  IntVal of int
| BoolVal of bool
| FunVal of (semval * store → econt → semval * store)
| Cont of scont
;;

(* Pour information:
   val semexp : Ast.exp → environment → store → econt → semval * store;;
   val semstat : Ast.statm → environment → store → scont → store;;
*)

val eval : Ast.exp → semval;;
```

5.7.1 Sémantique dénotationnelle directe

Simplifiez l'interface `semant.mli` afin de l'adapter à une sémantique dénotationnelle directe. Puis créez le fichier `semant.ml` mettant en œuvre cette sémantique :

1. pour PCF sans instructions (ni **goto**) ;
2. pour PCF avec instuctions (vous ignorerez les étiquettes et **goto**).

5.7.2 Sémantique dénotationnelle à continuations

En revenant à la version initiale de `semant.mli`, implémentez progressivement une sémantique à continuations pour PCF :

1. pour le noyau fonctionnel du langage
2. en incluant les instructions
3. en incluant les **goto**.