## Différents modes de programmation

* Des modes où les variables ont le même sens qu’en mathématiques dans lesquels les programmes sont parfois appelés "spécifications exécutables". Ces modes sont fondés sur des théories mathématiques éprouvées. (il s’agit de la programmation fonctionnelle et la programmation logique)
* Le mode de programmation impérative fondé sur une autre notion de variable et qui par des problèmes plus conséquent lorsque l’on veut "raisonner" sur les programmes (les prouver, les transformer, etc.)

## La notion de variable

* En mathématiques : une inconnue dans une équation.

Résoudre l’équation en donnant une définition explicite

Ex : 2x – 6 = 0 ⬄ x = 3

* En informatique : expression abstraite d’une adresse en mémoire. Une définition est appelée "affectation" et exprime le rangement d’une valeur à cette adresse en mémoire.

Ex : a = 0 ; a = 1 x = x + 1

La notion de variable au sens informatique est évidemment très dangereuse pour raisonner sur les programmes.

Elle interdit par exemple toute substitution d’une variable par "une" définition, alors que c’est la base même des mathématiques.

On peut revenir aux mathématiques. Il suffit de formaliser la mémoire ou plus exactement un état mémoire.

Un état mémoire est une fonction qui à tout nom de variable (identificateur) associe une valeur. On dit que la programmation impérative est une programmation par effet sur un état mémoire. On peut définir un programme par son effet sur les états mémoires. La sémantique d’un programme impératif est alors une fonction entre état mémoire.

f : ensemble des états mémoire 🡪ensemble des état mémoire

s |-> s’ = f(s)

Où s (respectivement s’) est un état mémoire initial (respectivement final)

NB : une telle fonction entre états mémoires est parfois appelée fonction de transition

Références bibliographiques :

* Programming from specifications – Carroll Morgan 1990
* Construction et vérification de programmes – Roland C. Backhouse 1989
* Logique – méthodes formelle par l’étude des programmes
* Algebric specifications
* [www.cs.tcd.ie/Matthew\_Hennessy/](http://www.cs.tcd.ie/Matthew_Hennessy/)

# II) Sémantique des langages de programmation

NB : On ne peut donner un sens à un énoncé que s’il est bien typé. Ces types sont des éléments essentiels dans la construction des programmes. Ces éléments sont liés à des notions importantes de spécification et de preuve (spécification algébrique des types). Ces notions permettent de définir formellement les opérations appliquées sur les données d’un programme. Dans ce chapitre, on s’intéresse plutôt à la sémantique globale du programme à partir de ces éléments.

## Sémantique dénotationnelle d’un langage

Définir la sémantique d’un langage, c’est donner la signification précise de tout énoncé syntaxiquement correct.

Dans ce point de vue, la sémantique est donnée par une fonction de l’ensemble des énoncés syntaxiquement corrects dans une autre ensemble supposé connu. Cet ensemble est généralement appelé le domaine sémantique (les "domaines d’interprétation") du langage considère ce domaine sémantique qui est choisi est un domaine mathématique qui permet de donner une description précise et non ambigüe.

Donc, une fonction classiquement notée [[ . ]] :

[[ . ]] ensemble des énoncés syntaxiquement corrects 🡪 domaine sémantique (mathématique)

P |-> [[ P ]]

NB : On dit parfois qu’un énoncé du langage sert de "notations" à une certaine réalité mathématique et que cette réalité est la "dénotation" de l’énoncé.

NB : Un aspect important de la sémantique dénotationnelle est la propriété de compositionnalité : la dénotation d’un énoncé est obtenue par composition de dénotations de ses constituants.

Exemple : sémantique du langage des seules binaires

Considérons d’abord le langage du simple\\des seuls mots "0" et "1". Sa grammaire est la suivante

bin :: 0 | 1

Son domaine sémantique est l’ensemble des entrées {0,1} c N et la fonction sémantique [[ . ]] peut être définie par

0 🡪 0

1 🡪 1

On note plutôt

[[ 0 ]]bin = 0

[[ 1 ]]bin = 1

Considérons maintenant le langage des suites binaires. Sa grammaire est la suivante

nat :: bin | nat bin

Son domaine sémantique est N et la fonction sémantique [[ . ]] est définir par

[[ b ]]nat = [[ b ]]bin

[[ nb ]]nat = 2 x [[ n ]]nat + [[ b ]] bin

Exemple : la suite binaire 101 aura comme sémantique l’entier 5 (dont la représentation en base 2 est 101)

[[ 101 ]]nat = 2 x [[ 10 ]]nat + [[ 1 ]]bin (par déf. de [[ . ]] )

= 2 x (2 x [[ 1 ]]nat + [[ 0 ]]bin) + [[ 1 ]]bin ( " )

= 2 x (2 x (2 x 1 + 0 ) + 1 = 5

Cette définition de la sémantique est "structurelle" et compositionnelle" car elle est fondée sur les règles de grammaire définissant le langage et la sémantique d’une expression ne dépend que des sémantiques de ses sous-expressions. (exemple [[ nb ]]nat ne dépend que de [[ n ]]nat et [[ b ]]nat)

NB : une autre grammaire de la fonction sémantique [[ . ]] serait :

[[ b ]]nat = [[ b ]]bin

[[ bn ]]nat = …

La définition sémantique serait plus compliquée : elle suppose d’exprimer le "poids" du chiffre binaire b.

Donc, la façon de décrire la syntaxe influence la définition sémantique.

L’intérêt d’une sémantique dénotationnelle est de permettre de faire facilement la preuve de propriétés sur les expressions admises par la syntaxe.

* Les symboles terminaux
* Les termes construits par les règles de grammaire.

Ce type de démonstration est appelé "preuve par récurrence" ou "par induction" mathématiques.

Exemple :

Montrons que, pour toute expression n du langage "nat"

[[ 0n ]]nat = [[ n ]]nat

Il faut d’abord le démontrer pour les symboles terminaux de "nat" càd les expressions du langage "bin"

[[ ~~0~~0 ]]nat = 2 x [[ ~~0~~ ]]nat + [[0 ]]bin = 2 x 0 + 0 = [[ 0 ]]nat

[[ ~~0~~1 ]]nat = 2 x [[ ~~0~~ ]]nat + [[ 1]]bin = 2 x 0 + 1 = [[ 1 ]]nat

Puis on le démontre pour les termes construits par la règle de grammaire nat :: nat bin

Supp !osons la propriété vraie pour un certain n du langage "nat" et considérons la suite binaire nb, alors

## Sémantique dénotationnelle d’un langage de programmation

Considérons le cas d’un langage impératif simplifié que l’on appellera "L" comportant :

Comme instructeur de base :

* **L’affectation** x := exp
* **L’instruction vide** skip

Comme structure de contrôle :

* **La séquence** A ; B
* **La conditionnelle** if b then A
* **L’alternative** if b then A else B
* **La répétitive** do n times A
* **L’itérative** calcule b do A

NB : On s’intéresse uniquement ici à la sémantique dite dynamique : la sémantique des instructions et pas à la sémantique statique statique : le contrôle de type des expressions.

Nous avons déjà vu (au chapitre I) d’introduction) que la signification d’un programme impératif est une fonction mathématique que représente l’effet du programme sur un état mémoire. Cette fonction prend en argument l’état mémoire initial (avant exécution du programme) et a pour résultat l’état mémoire final (après exécution du programme).

Le domaine sémantique du langage "L" donc l’ensemble des fonctions de S dans S où S est l’ensemble des états mémoire sur tout programme P de "L"

[[ P ]] : S 🡪 S

s 🡪 s’ = [[ P ]](S)

Un état mémoire s est une fonction de l’ensemble d’identificateur dans l’ensemble des valeurs

Ex : s(x) note la valeur de la variable x dans l’état mémoire s.

On notera :

* s(x) ou [[ x ]]s , la valeur de la variable x dans l’état mémoire s
* plus généralement, s(exp) ou [[ exp ]], le résultat de l’évaluation de l’expression exp dans l’état mémoire s
* (S | x -> v) l’état mémoire identique à S excepté qu’il associe à la variable x la valeur v.

On peut alors établir les définitions suivantes :

Affectation : Pour tout s € S [[ x := exp ]](s) = (s | x-> [[exp]]s)

Instruction vide : Pour tout s € S [[ skip ]](s) = s

Séquence : pour tout s € S [[ A ; B ]](s) = [[ B ]]([[ A ]](s))