

# Le $\delta$ code

Valentin Novo

December 2023

## 1 Introduction

## 2 Table des codes, des fonctions et des commandes

Code	Symbole compilé	Fonction
<code>\exists</code>	$\exists$	Déclarer une variable
<code>\forall</code>	$\forall$	Faire une boucle
<code>:=</code>	$:=$	Affectation
<code>\in</code>	$\in$	Appartenance (pour les tests et les déclarations)
<code>\empty</code>	$\emptyset$	Ensemble vide
<code>\include</code>	$\subseteq$	Inclusion dans un ensemble (pour les déclarations)
<code>\False</code>	$\perp$	Valeur booléenne "faux"
<code>\True</code>	$\top$	Valeur booléenne "vrai"
<code>\i</code>	$\iota$	Unité imaginaire
<code>\div</code>	$\div$	Opérateur de division
<code>\mul</code>	$\times$	Multiplication numérique et produit cartésien
<code>\P</code>	$\wp$	Ensemble des parités d'un ensemble
<code>\land</code>	$\wedge$	ET logique
<code>\lor</code>	$\vee$	OU logique
<code>\lnot</code>	$\neg$	NON logique
<code>\impl</code>	$\Rightarrow$	IMPLICATION logique
<code>\equiv</code>	$\Leftrightarrow$	EQUIVALENCE logique
<code>\xor</code>	$\oplus$	OU exclusif
<code>\nes</code>	$\square$	Assertion
<code>\neq</code>	$\neq$	différent de
<code>\ge</code>	$\geq$	Supérieur ou égal
<code>\le</code>	$\leq$	Inférieur ou égal
Alt Gr +6		Divise
	⌈	intervalle d'entier
	⌋	intervalle d'entier

<code>\S</code>	$\mathbb{S}$	Ensemble des str
<code>\N</code>	$\mathbb{N}$	Ensemble des entiers naturels
<code>\Z</code>	$\mathbb{Z}$	Ensemble des entiers relatifs
<code>\R</code>	$\mathbb{R}$	Ensemble des réels
<code>\C</code>	$\mathbb{C}$	Ensembles des complexes
<code>\B</code>	$\mathbb{B}$	Ensemble des booléens
<code>\case</code>	$\triangleright$	Tests pour structures conditionnelles
<code>\do</code>	$\rightsquigarrow$	Code à exécuter si...
<code>\app</code>	$-o->$ (flèche avec un cercle)	Déclarer un dictionnaire
<code>_n</code>	$_n$	Indice (n est un chiffre de 0 à 9)
<code>\mapsto</code>	$\mapsto$	associer une valeur (applications)
<code>\to</code>	$\longrightarrow$	Déclaration du type d'une application/fonction
<code>\nexists</code>	$\nexists$	Détruire une variable
<code>\inter</code>	$\cap$	Opérateur "intersection" sur des ensembles
<code>\union</code>	$\cup$	Opérateur "union" sur des ensembles
<code>\alias</code>	$\triangleq$	Créer un alias
<code>\(</code>	$\langle$	Délimite l(es) argument(s) d'une application
<code>\)</code>	$\rangle$	Referme un $\langle$
<code>\midpoint</code>	$\cdot$	Différentes utilités, voir par la suite

Nom de la fonction	Utilité	Nombre d'arguments
<code>echo</code>	Afficher du texte	1
<code>ask</code>	Demander une saisie à l'utilisateur d'un certain type	2 (type puis texte)
<code>convert</code>	convertir une variable dans un autre type	2 (variable et type)
<code>dim</code>	nombre de composantes d'un tuple ou d'un produit cartésien	1

Commande	Effet
<code>@HIDE</code>	Arrête la diffusion des messages d'informations sur la déclaration, affectation, ...
<code>@SHOW</code>	Autorise la diffusion des messages d'informations sur la déclaration, affectation, ...
<code>@HALT</code>	Casse une boucle ou arrête totalement l'exécution du programme
<code>@CONTINUE</code>	Saute une itération dans une boucle
<code>@BELL</code>	Son du clavier
<code>@USE</code>	Permet d'utiliser un module
<code>@GLOBAL</code>	Dans une fonction, signale que les modifications apportées aux variables globales doit être conservée

### 3 Comment coder ?

Avant toute chose, il est fondamental de se rappeler qu'une ligne de  $\delta$  code se termine toujours par un ".".

#### 3.1 Les variables

Le  $\delta$  code a, comme tout langage, des variables, cependant, il faut les déclarer avant de leur affecter une valeur.

Une ligne de déclaration commence par le symbole " $\exists$ ". On peut distinguer 3 cas :

1. Pour déclarer une variable qui a un type "classique" (booléen, entier, réel, string, complexe), il suffit d'écrire  $\exists \text{ nomvariable} \in \text{Ensemble}$ .
2. Pour déclarer une variable qui est un  $n$  – *uplet*, il faut préciser le type de chaque objet. Le type est donc le produit cartésien du type de chaque objet, on écrit :  $\exists \text{ var} \in \text{Ens}_1 \times \text{Ens}_2 \times \dots \times \text{Ens}_n$ .
3. Pour déclarer une variable qui sera un ensemble, on a deux possibilités :
  - (a) Soit on crée une variable en la déclarant comme inclus dans le type :  $\exists \text{ var} \subseteq \text{Ens}$ .
  - (b) Soit en la déclarant comme un objet appartenant à l'ensemble des parties du type :  $\exists \text{ var} \in \wp(\text{Ens})$ .

Les deux manières de faire sont équivalentes.

Notez bien que les variables références des **valeurs**, contrairement à d'autres langages, après exécution de :  $\exists \text{ var1} \in \text{Typ1}.\text{var1} := \text{val1}.\exists \text{ var2} \in \text{Typ1}.\text{var2} := \text{var1}$ , les deux variables  $\text{var1}$  et  $\text{var2}$  contiennent la même valeur mais modifier  $\text{var1}$  n'impactera pas la valeur de  $\text{var2}$  et réciproquement.

#### 3.2 Les alias

Il est possible de créer des alias, c'est à dire une sorte de variable qui ne contiendra pas de valeur mais pointerait vers une autre variable.

Pour définir un alias on écrit :  $\text{var\_alias} \triangleq \text{variable}$

On peut alors modifier la variable contenant l'alias et la variable originale. Modifier l'un des deux revient à modifier les deux à la fois contrairement à une variable.

#### 3.3 Utiliser *echo* et *ask*

Les fonctions *echo* et *ask* permettent d'afficher un texte et de demander une saisie à l'utilisateur.

Pour afficher quelque chose avec *echo*, on utilise la syntaxe : *echo\$var*. Où *var* est le nom de la variable contenant le texte à afficher. **Attention**, il est interdit de passer autre chose qu'une variable en paramètre d'*echo* !

Pour demander une information à l'utilisateur avec *ask*, on utilise la syntaxe : *var1 := ask\$Ens\$var2*. Avec *var1* la variable qui va recevoir la réponse, *Ens*, le type voulu et *var2* la variable contenant le texte à afficher.

### 3.4 Les conditions

Pour créer des structures conditionnelles, la syntaxe est la suivante :  $\succ cond_1 \rightsquigarrow \backslash CODE_1 / \succ cond_2 \rightsquigarrow \backslash CODE_2 / \dots \succ cond_n \rightsquigarrow \backslash CODE_n /$ .

Analysons ce code.

Ici, le programme commence par tester si *cond<sub>1</sub>* est vrai, si c'est le cas, il exécute *CODE<sub>1</sub>* et les autres conditions ne sont pas examinées. Sinon le programme teste si *cond<sub>2</sub>* est vraie, si c'est le cas il exécute *CODE<sub>2</sub>*, sinon il passe à la condition suivante,... Et il continue ainsi jusqu'à ce qu'il n'y ait plus de conditions à tester ou qu'il y en ait une qui soit vraie. Notez bien que la place du . final est importante. En effet, regardons ce deuxième exemple :  $\succ cond_1 \rightsquigarrow \backslash CODE_1 / . \succ cond_2 \rightsquigarrow \backslash CODE_2 /$ .

Ici les deux tests ne sont pas dans la même phrase, ainsi, si *cond<sub>1</sub>* et *cond<sub>2</sub>* sont vraies, *CODE<sub>1</sub>* et *CODE<sub>2</sub>* s'exécuteront. Si on enlève le premier ., les deux conditions sont dans la même phrase, ainsi si *cond<sub>1</sub>* et *cond<sub>2</sub>* sont vraies, seul *CODE<sub>1</sub>* sera exécuté.

### 3.5 Les boucles

Il est bien entendu possible d'écrire des boucles, cependant seules les boucles "for" peuvent être faites simplement.

La syntaxe est :  $\forall var \in iterable : \backslash CODE /$ .

Quelques remarques :

- *var* est le nom de la variable qui changera à chaque tour de boucle, avant d'entrer dans la boucle, *var* ne doit être le nom d'aucune variable !
- *iterable* est l'objet sur lequel on va itérer, on peut itérer sur les strings, les ensembles, les intervalles d'entiers et l'ensemble  $\mathbb{N}$ . Notez bien que les chaînes de caractères seront parcourues dans l'ordre, de même pour les intervalles d'entiers et  $\mathbb{N}$ , par contre, l'ordre dans lequel les éléments d'un ensemble sont parcourus est aléatoire et ne dépend pas de l'ordre d'ajout.
- *CODE* est le code à exécuter à chaque itération. Notons que dans ce code, si vous placez un "@HALT", la boucle s'arrêtera lorsque cette instruction sera rencontrée, et le programme continuera son exécution après la boucle. De même, si vous placez un "@CONTINUE", l'itération s'arrêtera lorsque le programme rencontrera cette instruction, et commencera l'itération suivante.

Si vous souhaitez faire des boucles conditionnelles, cela sera un peu plus compliqué. Les boucles conditionnelles qu'on peut écrire ne sont pas des boucles "Tant que" mais plutôt des boucles "Jusqu'à ce que". Pour les faire, il faut utiliser une conditionnelle qui cassera la boucle lorsqu'une certaine condition est vraie et ceci à l'intérieur d'une boucle infinie. Les boucles conditionnelles se présentent ainsi :

$\forall i \in \mathbb{N} : \backslash CODE_1 /$ .

Avec une instruction " $\succ_{cond_1} \rightsquigarrow \backslash @HALT /$ ." dans  $CODE_1$

## 3.6 Fonctions et Applications

Ce langage possède 2 types de fonctions différents : Les applications et les fonctions.

### 3.6.1 Les Applications

Les applications sont des fonctions "plus faibles" : elles n'ont accès qu'à deux choses : les arguments et elle-même.

De plus, il est impossible de faire des boucles ou de longs codes avec. Cette sorte de fonction peut, sous certains angles, être considérée comme des fonctions mathématiques.

Les applications doivent être déclarées par une phrase avec la syntaxe suivante :  $func : typ_1 \longrightarrow typ_2$ . ou  $func$  est le nom de la fonction que vous voulez définir,  $typ_1$  est le type des arguments et  $typ_2$  est le type de la valeur retournée.

Ensuite, on doit lier les arguments avec la sortie. Pour cela on utilise une phrase avec la syntaxe :  $func : x_1; \dots; x_n \longmapsto f(x_1, \dots, x_n)$ ., ici,  $x_1; \dots; x_n$  sont les noms des arguments de l'application (on peut mettre ce qu'on veut comme nom d'argument). Ce qui a été appelé  $f$  dans l'exemple précédent désigne réellement ce que vous voulez que  $func$  renvoie.

Il est possible que vous souhaitiez faire des applications constantes, qui ne prendrons pas de paramètre, pour cela, il faut définir votre application de  $\wp(\emptyset)$  dans l'ensemble d'arrivée, et que son paramètre est l'ensemble vide.

En somme, on fait:

$func : \wp(\emptyset) \longrightarrow F$ .

$func : \emptyset \longmapsto c$ .

Il est également possible que vous souhaitiez faire que votre application puisse renvoyer des choses différentes selon les arguments. Par exemple, la factorielle de  $n$  est définie comme valant 1 si  $n = 0$  sinon c'est  $n \times (n - 1)!$ . Ceci est possible, et se traduirait par :

$factorielle : \mathbb{N} \longrightarrow \mathbb{N}$ .

$factorielle : n \longmapsto \succ n = 0 : 1 \succ n \neq 0 : n \times factorielle \langle n - 1 \rangle$ .

Notez bien que dans une application on utilise les ":" au lieu du " $\rightsquigarrow$ ".

Maintenant qu'on sait créer une application, il serait bien de pouvoir l'utiliser. On a déjà effleuré ce sujet dans l'exemple précédent. Pour appeler une application, on utilise la syntaxe:

$app\langle x_1; \dots; x_n \rangle$ . Notez que contrairement aux fonctions, il est possible de passer autre chose qu'une variable en paramètre d'une application. Les phrases :

$f\langle 1 \rangle$

$f\langle x - 1 \rangle$

Sont tout à fait correctes alors que si  $f$  est une fonctions les phrases :

$f\$1$

$f\$x - 1$

Sont incorrectes

### 3.6.2 Les fonctions

Les applications sont bien utiles mais ont une puissance restreinte, pour contourner cela, il faut définir des fonctions.

Un bloc de définition d'une fonction commence et finit par un "#". On peut distinguer de partie dans ces blocs:

- La partie de création
- Le code de la fonction

La partie de création se construit comme suit :

$func : typ_1 \longrightarrow typ_2$ .

$\langle x_1 \dots x_n \rangle$ .

@GLOBAL.

"Docstring".

La première ligne est la même que pour définir une application, nous n'y reviendrons pas. La seconde permet de nommer les arguments qui vont être utilisés. Les deux dernières lignes sont optionnelles, mais si vous souhaitez les inclure ensemble, il faut respecter cet ordre. L'instruction @GLOBAL si elle est utilisée signale que la fonction peut modifier durablement les variables globales. La dernière ligne permet de renseigner une documentation de la fonction.

Après cette partie, vous pouvez placer les instructions à exécuter lors de l'appel de la fonction. Attention, le corps de la fonctions ne peut être vide.

Notez que dans une fonction, si vous placez un  $\mapsto$  sans expression, l'ensemble vide sera renvoyé de même, si vous ne placez aucun  $\mapsto$ , l'ensemble vide sera retourné.

Si vous souhaitez faire une fonction sans paramètre, il faut la définir dans  $\wp(\emptyset)$ , et mettre  $\langle \emptyset \rangle$  comme nom de paramètre. Pour appeler cette fonction  $f$ , il faudra écrire  $f\$$

Si vous souhaitez que votre fonction ne renvoie rien, définissez la dans  $\wp(\emptyset)$

Les fonctions ont la particularité de pouvoir accéder (mais pas nécessairement modifier) aux variables globales. Cependant, si dans votre code principal vous définissez une variable "x" et qu'un argument de votre fonction "f" s'appelle "x", la variable globale "x" sera renommée "GLOBAL.x" **dans** "f" et seulement dans "f".