Le δ code

Valentin Novo

December 2023

1 Introduction

2 Table des codes, des fonctions et des commandes

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

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

Nom de la fonction	Utilité	Nombre d'arguments	Type de retour
echo	Afficher du texte	1	$\wp(\emptyset)$
ask	Demander une saisie à l'utilisateur d'un certain type	2 (type puis texte)	Type voulu
convert	convertir une variable dans un autre type	2 (variable et type)	Type voulu
dim	nombre de composantes d'un tuple ou d'un produit cartésien	1	$\mathbb{N}$
card	nombre d'élément d'un ensemble	1	$\mathbb{N}$
help	Affiche la documentation	1	$\wp(\emptyset)$

Commande	Effet
@HIDE	Arrête la diffusion des messages d'informations sur la déclaration, affectation, ...
@SHOW	Autorise la diffusion des messages d'informations sur la déclaration, affectation, ...
@HALT	Casse une boucle ou arrête totalement l'execution du programme
@CONTINUE	Saute une itération dans une boucle
@BELL	Son du clavier
@USE	Permet d'utiliser un module
@GLOBAL	Dans une fonction, signale que les modifications apportées aux variables globales doit être conservée
@USE	Importe un module voir 3.9

3 Comment coder ?

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

3.1 Les variables

Le δ code a, comme tout langages, 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 $\exists$ *nomvariable* $\in$ *Ensemble.*
2. Pour déclarer une variable qui est un $n-uplet$, il faut préciser le type de chaques objet. Le typle est donc le produit cartésien du type de chaque objet, on écrit : $\exists\ var \in Ens_1 \times Ens_2 \times \cdots \times Ens_n$.
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 : $\exists\ var \subseteq Ens$.
   (b) Soit en la déclarant comme un objet appartenant à l'ensemble des parties du type : $\exists\ var \in \wp(Ens)$.

   Les deux manières de faire sont équivalentes.

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

De plus, lorsqu'une variable a été créée, son type a été définit et ne peut être modifié ! Si vous souhaitez que votre variable contienne une valeur avec un autre type, il faudra *détruire* cette variable avec une phrase : $\nexists var.$ pour ensuite la recréer.

3.2 Les alias

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

Pour définir un alias on écrit : *var_alias* $\triangleq$ *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 les fonctions pré-définies

3.3.1 *echo*

La fonction *echo* permet d'afficher un texte dans la console.

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* (voir plus loin 3.6.2) !

3.3.2 *ask*

La fonction *ask* permet de demander de manière sécurisée une information à l'utilisateur.

Pour demander une information à l'utilisateur avec *ask*, on utilise la syntaxe : *askEnsvar1.* Avec *Ens*, le type voulu et *var1* la variable contenant le texte à afficher. On peut ensuite récupérer cette information pour une affectation par exemple.

3.3.3 *card*

La fonction *card* renvoie le nombre d'élément d'un ensemble.

On utilise la syntaxe suivante : *card$var1.* ou *var1* est une variable qui contient un ensemble. *card* retourne un entier de $\mathbb{N}$

3.4 *dim*

La fonction *dim* renvoie le nombre de composante d'un $n-uplet$.

On utilise la syntaxe suivante : *dim$var1.* ou *var1* est une variable qui contient un tuple. *dim* retourne un entier de $\mathbb{N}$

3.5 *help*

La fonction *help* affiche la documentation du paramètre.

On utilise la syntaxe : *help$var.* ou *var* est un objet, ça peut être une variable, une application, une fonction,...

3.6 Les conditions

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

Analysons ce code.

Ici, le programme commence par tester si $cond_1$ est vrai, si c'est le cas, il execute $CODE_1$ et les autres conditions ne sont pas examinées. Sinon le programme teste si $cond_2$ est vraie, si c'est le cas il execute $CODE_2$, 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 impoe¡rtante. En effet, regrdons ce deuxième exemple : $\succ cond_1 \rightsquigarrow \backslash CODE_1/.\succ cond_2 \rightsquigarrow \backslash CODE_2/.$

Ici les deux tests ne sont pasdans la même phrase, ainsi, si $cond_1$ et $cond_2$ sont vraies, $CODE_1$ et $CODE_2$ s'executeront. Si on enlève le premier ., les deux conditions sont dans la même phrase, ainsi si $cond_1$ et $cond_2$ sont vraies, seul $CODE_1$ sera executé.

3.7 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 ! De plus, la variable *var* sera détruite à la fin de la boucle.

- *iterable* est l'objet sur lequel on va itérer, on peut itérer sur les strings, les ensembles, les intervalles d'entiers et l'ensmble $\mathbb{N}$. Notez bien que les chaines de caractères seront parcourus 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 conditionnellles, cela sera un peu plus compliqué. Les boucles conditionnelles qu'on peut écrire ne sont pas des boucle "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 bocle 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 $CODE_1$

3.8 Fonctions et Applications

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

3.8.1 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 considérées 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 liéer les arguments avec la sortie. Pour cela on utilise une phrase avec la syntaxe : $func : x_1; \cdots ; x_n \longmapsto f(x_1, \cdots , x_n)$., ici, $x_1; \cdots ; x_n$ 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 réélement ce que vouvs 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:
$app\langle x_1; \cdots ; x_n \rangle$. 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

Notez que si vous appelez *help* sur une application, cela affichera la signature de l'application:
$f : T_1 \longrightarrow T_2$
$x_1 \cdots x_n \longmapsto f(x_1 \cdots x_n)$

3.8.2 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 :
$func : typ_1 \longrightarrow typ_2$.
$\langle x_1 \cdots x_n \rangle$.
$@GLOBAL$.
"*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, c'est ce qui sera affiché lors de l'appel de *help* dessus.

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 $\longmapsto$ sans expression, l'ensemble vide sera renvoyé de même, si vous ne placez aucun $\longmapsto$, 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 à valeur dans $\wp(\emptyset)$

Les fonctions ont la particularité de pouvoir accèder (mais pas nessecairement 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 renomée "GLOBAL·x" **dans** "f" et seulement dans "f".

3.9 Les modules

Dans cette section, nous supposerons que l'on utilise le fichier *test.df* contenant la variable *var*, le dictionnaire *dict*, la fonction *f*, l'application *app* et l'alias *var2*

Il est possible de charger un autre programme en temps que module, cela peut s'effectuer de différentes manières :

3.9.1 Avec *@USE*

Dans notre exemple, on utilisera *@USE test* pour importer totalement *test.df*, c'est à dire qu'après, le programme connaitra *var*, *dict*, *f*, *app* et *var2*, mais toutes les autres instructions de *test.df* seront également exécutées.

En somme, en utilisant *@USE test*, *test.df* sera totalement executé.

3.9.2 Avec *@USE**

Si vous utilisez *@USE* test*, le programme importera *var*, *dict*, *f*, *app* et *var2*, mais les autres instructions ne seront pas executées.

Attention cependant, pour pouvoir utiliser *@USE* test* il faut : que vous ayez déjà executé une fois *test.df* seul, qu'il n'y ait ni boucle infinie, ni interaction avec l'utilisateur (*ask*) dans le programme.

3.9.3 Avec *@USE ... FROM ...*

Cette instruction est un cas particulier en quelque sorte de *@USE**, elle vous permet d'importer seulement certains objets du fichier désigné, par exemple:
@USE var,f FROM test. permetd'importer uniquement *var* et *f*, notez bien que si vous importez plusieurs choses à la fois, ces dernières devront être séparées par des virgules, mais collées à la virgule.

Notez également que les conditions pour pouvoir utiliser cette syntaxe sont les mêmes que pour utiliser *@USE**.