

Université de Mons  
Faculté des sciences  
Département d'Informatique

---

# Compilation : interpréteur Dumbo

## Rapport de projet

---

*Professeur :*

Véronique BRUYÈRE  
Alexandre DECAN

*Auteur :*

Thomas LAVEND'HOMME  
Guillaume PROOT



Année académique 2019-2020

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Mode d'emploi</b>	<b>2</b>
<b>3</b>	<b>Grammaire</b>	<b>2</b>
<b>4</b>	<b>Gestion d'expression</b>	<b>3</b>
4.1	Gestion des variables locales et globales . . . . .	3
4.2	Gestion du for . . . . .	3
4.3	Gestion du if . . . . .	3
<b>5</b>	<b>choix personnel</b>	<b>3</b>
<b>6</b>	<b>Problème survenus</b>	<b>4</b>
<b>7</b>	<b>Conclusion</b>	<b>4</b>

# 1 Introduction

L'objectif de ce projet est de nous familiariser avec le fonctionnement d'un interpréteur et pour cela il nous a été demandé d'en réaliser un en python interprétant du code *dumbo*.

## 2 Mode d'emploi

Pour faire fonctionner notre interpréteur, il vous est demandé de vous placer dans le répertoire où se trouve le fichier *dumbo\_interpreter.py* ainsi que vos fichiers à interpréter. Ensuite, dans l'invite de commande, rentrez la commande suivante : *python3 dumbbo\_interpreter.py data template output*.

## 3 Grammaire

Pour ce projet, nous avons utilisé la grammaire de base du langage dumbbo à laquelle nous avons apporté quelques modifications et ajouts :

- Tout d'abord nous avons ajouté la règle : *dumbo\_block* : ("{{" "}}") pour pouvoir gérer les cas où nous aurions affaire à un dumbbo\_block vide.
- Nous avons géré les différentes expressions individuellement afin de limiter les erreurs et simplifier la clarté du code. Pour ce qui est des expressions "print" nous l'avons séparée en deux cas distinct :
  - **print** affichera ce qui est attendu sans check supplémentaire. (Exemple : `print 4`  $\Rightarrow$  4)
  - **print!** affichera ce qui est attendu excepté que les entiers deviendront des booléens. ( `0`  $\Rightarrow$  False et le reste  $\Rightarrow$  True). (Exemple : `print! 4`  $\Rightarrow$  True)
- Afin de gérer l'expression "if < boolean > do < expressions list > endif", nous avons ajouté l'assignation d'un booléen à une variable.
- En plus de la grammaire de base de dumbbo nous avons ajouté la possibilité de faire des opérations logiques (AND, OR, NOT, TRUE, FALSE) ainsi que des comparaisons logiques et arithmétiques (=, <, >, <=, >=, !=).
- Dans la grammaire de base de dumbbo, il n'y a pas de distinctions entre l'instanciation de variable et l'appel de celle-ci. C'est pourquoi, nous avons décidé de séparer cela en en fonctions différentes *variable\_set* et *variable\_get*. Tel les set et get dans d'autres langages de programmation *variable\_get* ne fonctionnera que si la variable sur laquelle la

fonction est appliquée existe déjà alors que *variable\_set* n'impose pas cette contrainte.

- Enfin, nous demandons pour instancier une variable d'y ajouter son type (exemple : `int x = 4`)

## 4 Gestion d'expression

### 4.1 Gestion des variables locales et globales

Pour gérer les variables, nous avons décidé d'utiliser une liste chaînée de dictionnaire où chaque maillon est un scope. Quand on détecte un nouveau dumbo bloc, nous créons un nouveau scope, on visite ensuite le bloc en insérant toutes les variables de ce bloc dans le scope nouvellement créé et lorsque le parcours du dumbo bloc est finis nous refusionnons le scope avec les variables globales.

### 4.2 Gestion du for

Nous gérons les boucles for de la manière suivantes :

1. Tout d'abord nous créons une variable locale (*loop\_variable*). Nous l'assignons à la valeur correspondante dans la *string\_list* ou par rapport à la variable donnée dans un nouveau scope.
2. On exécute le dumbo bloc à l'intérieur de la boucle.
3. On efface la variable locale et on recommence l'opération tant qu'il y a des éléments dans la *string\_list*.

### 4.3 Gestion du if

Nous avons décidé de gérer les if de la manière suivante :

1. On teste si la condition est vraie.
2. Si celle-ci est vérifiée, on crée un nouveau scope.
3. On exécute le dumbo bloc.
4. On supprime les variables locales.

## 5 choix personnel

Nous avons décidé d'utiliser la librairie LARK à la place de PLY car malgré le fait que nous ne l'ayons pas abordé en tp nous la trouvions plus

facile à prendre en main et elle respectait plus pour nous les conventions python.

Il n'y a pas de booléen à la base avec notre interpréteur. Quand nous gérons une condition logique, nous considérons  $0 \Rightarrow False$  et  $(int \neq 0) \Rightarrow True$ .

Nous avons aussi choisi d'imposer un typage aux variables du fichier d'entrée. Cela nous permet de gérer plus facilement les erreurs de type lors de la compilation.

Nous avons pris le parti de faire deux print différents comme expliqué dans la section grammaire ci-dessus. En effet, vu que nous n'utilisons pas de booléen, il nous fallait trouver un moyen de pouvoir tout de même en afficher au besoin. C'est pourquoi nous avons ajouté le print ! qui transformera les entiers affichés en leur booléen correspondant ( $False \Rightarrow 0$  et  $True \Rightarrow$  pour le reste) à l'inverse du print qui lui laissera tout entier sous sa forme classique.

## 6 Problème survenus

Un problème que nous avons rencontré est la gestion de la boucle for. En effet, nous utilisons un "Transformer" de Lark qui fonctionne en bottom-up ce qui entraînait que l'on parcourait d'abord l'intérieur de la boucle avant l'expression en elle-même. Pour résoudre cela, nous avons revu notre implémentation en utilisant la classe "Interpreter" de Lark à la place de "Transformer" qui elle utilise une approche top-down.

Nous n'utilisons pas de booléens dans notre projet juste des entiers (  $0 = False$ ,  $1 = True$ ) ce qui fait qu'il n'y a pas de distinctions entre les deux. Par exemple tous les 1 de notre programme, s'ils sont mis dans des conditions seront considérés comme le booléen True.

## 7 Conclusion

Ce projet nous auras permis de mieux nous familiariser avec le processus de compilation et différents outils facilitant celui-ci. Si nous devons émettre une critique concernant notre projet, ça serait sur notre gestion des types. En effet, celle-ci est loin d'être vraiment optimisée.