LES TROPHÉES NSI

Édition 2022

DOSSIER DE CANDIDATURE

PRÉSENTATION DU PROJET

The µ lang, le langage de programmation en balises

> PRÉSENTATION GÉNÉRALE

Le langage µ (à prononcer *langage mu* ou *mu lang*) est un langage de programmation procédural markup interprété en python (et bientôt en C) à typage dynamique, sans gestion de mémoire pour l'instant. Il peut être assimilé à la famille des langages Lisp, même si nous ne connaissions pas ce type de langages lors de la définition des principes du langage µ.

Notre but était de mieux comprendre comment l'ordinateur interprète un script, nous avons donc essayé de créer un langage de programmation qui, bien que simple, puisse fonctionner et permettre de créer des algorithmes complets. Ce langage a été pensé pour être original au niveau de la syntaxe : il est sous forme de balises, à la manière d'un langage de Markup.

> ORGANISATION DU TRAVAIL :

L'équipe est composée de :

- Ryan Bartolomeo, celui qui a eu les idées bizarres pour rendre le langage unique.
- Thomas Bernhard, celui qui a assez d'expérience pour les réaliser.

Nous avons utilisé GitHub pour collaborer car, le travail a d'abord commencé en dehors de la classe autour d'une question: "Mais comment un ordinateur *comprend*-il un script? ".

Puisque nous étions 2, il n'a pas été nécessaire de répartir le travail : lorsqu' un membre du groupe trouve une idée d'amélioration, il l'explique à l'autre et celui qui pense à la meilleure manière de l'implémenter le fait, ce qui fait que nous avons peu ou prou tous deux autant participé à la création du langage.

La communication autour de ce projet s'est majoritairement faite entre les cours au lycée (nous sommes dans la même classe) sous forme de débats quand nous pensions à deux façons différentes d'aborder un problème. Et lorsqu'un des deux ne comprenait pas, le registre didactique accompagné de papier et de crayon était le principal moyen de communication.

LES ÉTAPES DU PROJET :

La première version du langage, dite µlang 1.0, est complètement différente des autre versions car sa fabrication été motivée par la simplification du processus de création du langage : le but n'était pas de créer un vrai langage, mais plutôt de comprendre comment *décortiquer* les instructions afin que l'ordinateur puisse s'approprier les ordres et les exécuter. Malheureusement, au fil de l'évolution de l'interpréteur, le langage n'avait plus une structure très claire. Il a alors été délaissé et est rentré dans les archives. Le langage µ a ainsi dormi un mois avant de renaître lorsque Ryan a appris le JavaScript et s'est posé une question de ce type:

“ Pourquoi le JavaScript, un langage généralement inclus au milieu d’un fichier HTML, n’est pas aussi un MarkUp Language ? Serait-il possible de coder avec des balises ?“

Une courte réflexion a alors fait éclore la première ligne du nouveau µ lang :

```
<while True><print>"Hello world"</print></while>
```

Après une plus longue concertation avec Thomas sur la façon dont un langage de programmation à balise puisse ressembler, le langage µ naquit de nouveau.

Le µ lang 2.0 a été plus facile à implémenter au début, car les réflexions sur les méthodes à adopter pour réaliser le projet ont déjà été adoptées lors de la version 1.0.

Il a aussi été décidé pour cette deuxième version d’utiliser le latin pour le nom de chaque balise et d’entourer le code de balises µ en conséquence du contexte HTML. Le latin (dum, inferioris, indo, loq ...) a été adopté pour éviter que les balises du programme soient confondues avec des balises du document HTML et rendre le langage plus instinctif : le latin étant la racine du français, l’apprentissage du vocabulaire µ était plus facile à apprendre. Pour ce qui est de l’obligation d’entourer le langage de balises µ, nous avons posé cette contrainte pour que la machine comprenne qu’il s’agit d’une suite d’instructions.

Plus concrètement, le projet s’est construit dans le même ordre dont le fichier script est traité: d’abord la découpe du fichier en liste de mots (tokenizer.py), puis en arbre de syntaxe abstraite (parser.py) et enfin l'exécution des instructions (nodes.py). Ces parties sont expliquées plus en détail à la fin de ce document.

Lorsque un simple programme était compréhensible, nous avons ajouté de nouvelles balises étape par étape, en commençant par celles basiques comme les opérations mathématiques (l'addition, la multiplication, la division : Addere, Multiplicare, Partium). Ensuite, nous avons ajouté la gestion des variables (avec la balise Indo). La prochaine étape a été la création des conditions (Inferioris, Aequalis, Verum, Falsum, Et, Ubi). S’en est suivi la création de la gestion listes (Ordinata) et grâce à l’implémentation des conditions, nous avons pu faire des blocs qui s'exécutent sous une condition (balise Si). Il a ensuite été facile d’implémenter les boucles (Dum).

> FONCTIONNEMENT ET OPÉRATIONNALITÉ :

Aujourd’hui, µ-lang 1.0 est presque complet mais n’est plus entretenu.

Le µ-lang 2.0 est opérationnel car il est possible de faire des boucles, des variables et des fonctions. Il reste à améliorer la compréhension des fonctions, car pour l’instant leur utilisation est encore très contraignante: par exemple, l’instruction de renvoi de valeur lors de l'exécution de la fonction doit forcément être à la fin de la fonction, et sinon ignorée. Les listes existent, mais restent peu manipulables : seulement la consultation et l’assignement d’une valeur à un certain index est possible.

De nombreux programmes en µ ont été testés afin de s’assurer de la fiabilité de l’interpréteur, notamment fib.µ, qui doit calculer la suite de fibonacci.

> OUVERTURE :

Nous espérons améliorer l'implémentation des fonctions qui présentent pour le moment de nombreux problèmes.

Il serait aussi intéressant de le rendre disponible en ligne, comme alternative au JavaScript sur le fichier script pour qu'un fichier HTML, même s' il a une algorithmie, reste entièrement composé de balises.

Nous sommes en train d'implémenter un interpréteur µ en C, afin de rendre l'exécution du script beaucoup plus rapide.

Nous avons créé un dictionnaire de mots clefs afin que l'éditeur de texte Atom puisse comprendre la grammaire du langage, en par exemple colorant les balises ou bien complète les noms automatiquement.

Ce que nous regrettons, c'est de ne pas avoir fixé la syntaxe intégrale du langage avant de commencer à concevoir l'interpréteur.

DOCUMENTATION

> Spécifications fonctionnelles

Afin d'exécuter le code, il faut lancer :

```
python3 main.py votre_script.µ
```

Lorsque vous lancez cette instruction, le script va être ouvert puis interprété. Cette interprétation passe par 3 étapes: la mise en jetons, la conversion de ces jetons en arbres puis l'exécution des noeuds de l'arbres

i Lexer, aka mise en tokens

Qu'est ce qu'un fichier script ? Du point de vue d'un ordinateur, un script n'est qu'un fichier contenant du texte. La première étape de notre interprétation de ce script doit donc passer par une compréhension et un découpage des ordres écrits par le programmeur. Le but du lexer est donc de relever les différents mots clefs et expressions, afin de transformer ce texte en une succession d'ordres et d'indications compréhensibles par la machine. Prenons l'exemple de ce script :

```
<µ>
  <indo> v  || Hello world ! || </indo>
  <loq>
      v
      <mul> 23 3 </mul>
  </loq>
</µ>
```

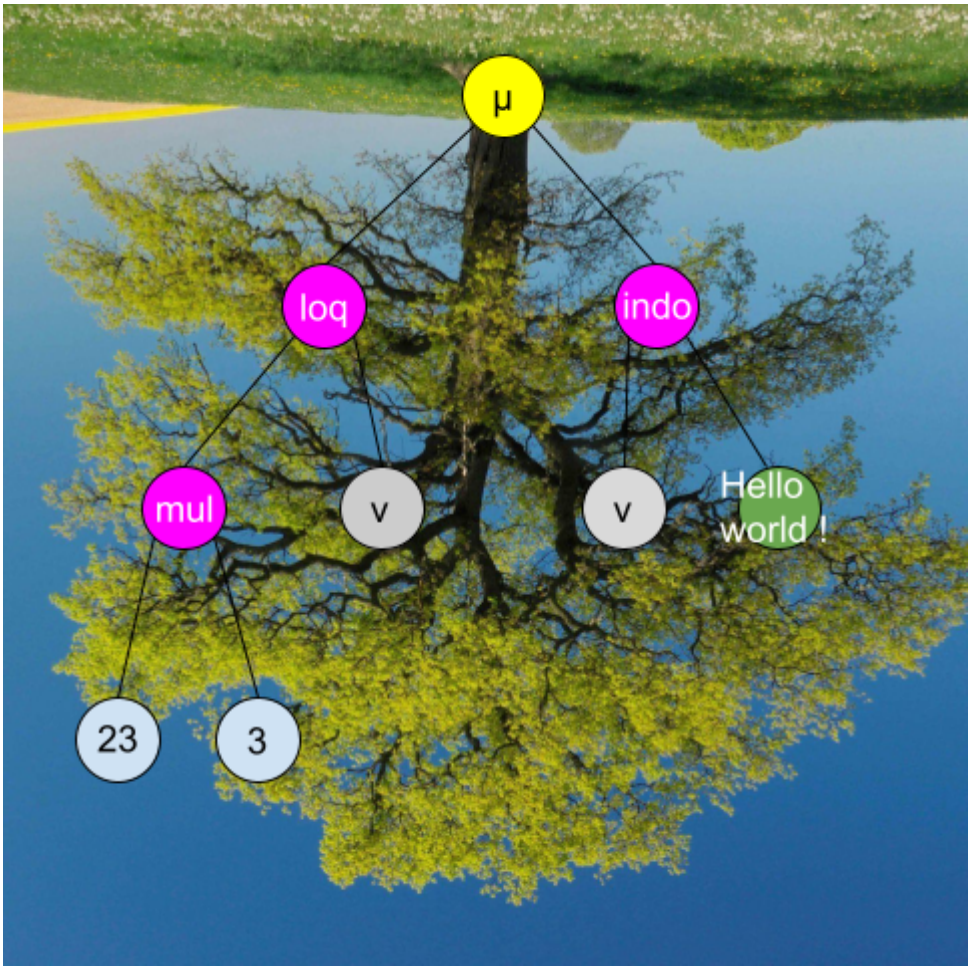
Un ordinateur ne peut lire qu'un seul caractère à la fois. On va ainsi *scanner* le document afin de trouver un caractère clef à la compréhension du script. Ici, on trouve des "<", qui vont indiquer un début de balise. On va donc considérer tout ce qui se situe entre "<" et ">" comme une balise d'action. Le même principe est suivi pour les "||", qui indiquent le début et la fin d'une chaîne de caractère (Filum). Lorsque le programme ne reconnaît ni une balise d'action, ni une chaîne de caractères, il vérifie si le *mot* qu'il traite est composé de chiffres et considère donc cette partie comme un entier (Numerus) ou une variable (comme "var" dans ce script). Chacun de ces objets est considéré comme un jeton ou **token** et va être stocké dans une liste, qui serait pour ce programme :

```
Balise("µ"), Balise("indo"), Variable("v"), Filum("hello world!"),
Balise("/indo"), Balise("loq"), ...
```

ii Parser, mise en arbre

Maintenant que nous avons une liste de tokens, nous devons comprendre leur hiérarchie, car un peu comme un point marque la fin d'une phrase, chaque balise marque la fin et le début d'une partie.

Chaque *partie* contient des *sous-parties*, qu'il faut organiser dans la mémoire. La plupart des langages de programmation peuvent être transformés en arbre. En informatique, un arbre est une structure de données qui organise les informations en nœuds, où chaque nœud a une liste d'enfants (pouvant être vide). Nous organisons donc le script en arbre, où chaque jeton est un nœud. L'arbre de ce script ressemblerait à ceci :



Où chaque couleur représente le *type* du noeud : Balise µ, Balise, Identifieur, Numerus, Filum

iii Interpréter

Une fois que nous avons cet arbre de valeur, il suffit de faire un parcours en profondeur de l'arbre et d'exécuter ce à quoi chaque nœud correspond en fonction de son type et de sa valeur.

Pour ce qui est des variables, nous utilisons un dictionnaire où la clef correspond au nom de la variable et la valeur à la valeur de la variable. A l'exécution de ce script, le dictionnaire contient :

Nom	Valeur
v	Filum(“Hello world ! ”)

car il n'y a malheureusement pas encore de *garbage collector*

> Spécifications techniques

Malgré le fait que ce langage a été conçu depuis un système Linux, l'interpréteur peut fonctionner sur n'importe quelle machine à condition d'y installer au préalable un interpréteur python.

Il en va ainsi que l'exécution d'un programme µ peut être assez longue, car on opère à une interprétation par un langage interprété. Le premier choix d'allègement de la RAM à été de n'importer aucune bibliothèque, afin que l'on puisse alléger au maximum le script µ, mais ce n'était pas assez. C'est pourquoi nous travaillons sur une version en langage C, dite **c-mu**, qui permettra de rendre l'interprétation du µ aussi rapide que celle d'un script python. Elle est disponible en tant que branche sur la page GitHub du projet (https://github.com/Mdrs-Corp/the-mu-lang/tree/c-mu).