Compilation

Rapport de projet

Activité d'Apprentissage S-INFO-012

FLORIANI Laurence

Année Académique 2020-2021
Troisième année de Bachelier en Sciences Informatiques

Faculté des Sciences, Université de Mons

1er mai 2021

Table des matières

1 Introduction

Dans le cadre du cours de Compilation, j'ai implémenté un programme nommé "dumbo.py". Cette application doit permettre la génération de texte à partir de deux fichiers passés en entrée :

- Le premier fichier, `Data`, contient le code `Dumbo` qui initie les variables ou ne fait rien.
- Le second fichier, `Template`, comporte du texte ainsi que du code `Dumbo` permettant d'injecter les variables précédemment initiées dans le texte. D'autres opérations y sont également possibles.

J'ai choisi d'utiliser le `parser Lark` car je le trouvais particulièrement intuitif à employer. Le fichier `dumbo.lark` contient la grammaire finale tandis que le fichier `TreeToDumbo.py` effectue le travail pour passer de l'arbre de syntaxe au résultat final.

Afin d'implémenter ce programme, j'avais à ma disposition une grammaire de base décrivant le langage `dumbo`. Celle-ci doit être complétée puis modifiée afin de lever les ambiguïtés. En dernier lieu, les opérations arithmétiques et booléennes de base y sont également ajoutées.

2 Mode d'emploi

Il faut se placer dans le dossier contenant le fichier `dumbo.py` puis entrer dans la console la commande suivante :

python3 dumbo.py data template

Celle-ci permet de lancer l'application avec un fichier `data.dumbo` et un fichier `template.dumbo`, le résultat du programme sera affiché sur la sortie courante.

Certains tests unitaires ont été également implémenter afin d'évaluer des fonctionnalités qui ne l'étaient pas toujours dans les exemples. La commande ci-après permet de les exécuter. *python -m unittest dumbo_ tests.py*

3 Grammaire

La grammaire complétée et modifiée est disponible en annexe.

La première modification est l'ajout d'une production de départ à la grammaire. En effet, le `parser Lark` en a besoin pour fonctionner correctement.

<*start*>	⟶	<*programme*>

La seconde modification consiste à faire en sorte qu'un bloc d'instructions `Dumbo` vide puisse être accepté par la grammaire.

<*dumo-bloc*>	⟶	{{ <*expression-list*> }}
	|	{{ }}

La troisième modification consiste à distinguer les différents types d'expression qui existent. Ceci permettra à l'étape suivante de les traiter de manière distincte. Ci-après se trouve les expressions possibles triées selon leur rôle :

- Assigner une variable : `expression_var`
- Afficher du contenu : `expression_print`
- Appliquer une condition : `expression_if`
- Effectuer une boucle sur une liste de `string` : `expression_for_lis`
- Effectuer une boucle avec un nom de variable : `expression_for_var`

<*expression*>	⟶	<*expression-var*>
	|	<*expression-print*>
	|	<*expression-for-lis*>
	|	<*expression-for-var*>
	|	<*expression-if*>

<*expression-var*>	⟶	<*variable-set*> : = <*string-expression*>
	|	<*variable-set*> : = <*string-list*>
	|	<*variable-set*> : = <*arith-expression*>
<*expression-print*>	⟶	print <*string-expression*>
<*expression-for-lis*>	⟶	for <*variable-set-for*> in <*string-list*>
		do <*expression-list*> endfor
<*expression-for-var*>	⟶	for <*variable-set-for*> in <*variable-get-str*>
		do <*expression-list*> endfor
<*expression-if*>	⟶	if <*test*> do <*expression-list*> endif

La quatrième modification consiste à distinguer les différentes utilisations de l'axiome <*variable*>. En effet, l'instanciation d'une variable ne doit pas être implémentée de la même manière que la récupération. En tout quatre types sont distingués, deux pour l'instanciation et deux pour la récupération. Remarque : `CNAME` est implémenté de base dans `Lark` et correspond à la définition d'un nom de variable.

<*variable-set*>	⟶	CNAME
<*variable-set-for*>	⟶	CNAME
<*variable-get-str*>	⟶	VARIABLE-STR
<*variable-get-int*>	⟶	VARIABLE-INT

VARIABLE-STR	$\longrightarrow$	CNAME
VARIABLE-INT	$\longrightarrow$	CNAME

La cinquième modification est d'ajouter les opérations arithmétiques. La grammaire de celles-ci est inspirée de celle vue en cours. Remarque : `SIGNED-INT` est implémenté dans `Lark`, il représente tout entier positif ou négatif.

<*arith-expression*>	$\longrightarrow$	<*add-expr*>
	|	<*sub-expr*>
	|	<*term*>
<*add-expr*>	$\longrightarrow$	<*arith-expression*> + <*term*>
<*sub-expr*>	$\longrightarrow$	<*arith-expression*> - <*term*>
<*term*>	$\longrightarrow$	<*multi-expr*>
	|	<*div-expr*>
	|	<*factor*>
<*multi-expr*>	$\longrightarrow$	<*term*> * <*factor*>
<*div-expr*>	$\longrightarrow$	<*term*> / <*factor*>
<*factor*>	$\longrightarrow$	<*integer*>
	|	<*variable-get-int*>
<integer>	$\longrightarrow$	SIGNED-INT

La sixième et dernière modification consiste à ajouter les opérations booléennes et les opérateurs logiques. Les opérateurs logiques s'appliquent entre deux expression arithmétiques, précédemment décrites. Et les opérateurs logiques `AND` et `OR` sont utilisés entre deux opérations booléennes.

<test>	$\longrightarrow$	<test> or <and-test>
	|	<and-test>
<and-test>	$\longrightarrow$	<and-test> and <comparison>
	|	<comparison>
<comparison>	$\longrightarrow$	<lower>
	|	<upper>
	|	<equal>
	|	<not-equal>
	|	<boolean>
<lower>	$\longrightarrow$	<arith-expression> < <arith-expression>
<upper>	$\longrightarrow$	<arith-expression> > <arith-expression>
<equal>	$\longrightarrow$	<arith-expression> = <arith-expression>
<not-equal>	$\longrightarrow$	<arith-expression> ! = <arith-expression>
<boolean>	$\longrightarrow$	<true>
	|	<false>
<true>	$\longrightarrow$	true

<false>	⟶	false

4 Analyse sémantique

La classe `TreeToDumbo` s'occupe de transformer l'arbre construit par `Lark` pour obtenir le résultat final. Elle a deux attributs : le résultat, un `string`, et les variables stockées dans un dictionnaire avec leur valeur.

L'analyse sémantique s'effectue de manière descendante. Pour chaque production dans la grammaire, il faut définir leur(s) effet(s) c'est-à-dire ce qu'ils font.

4.1 Gestion des variables

La gestion des variables se fait de manière globale au sein de la classe principale. En effet, un dictionnaire contient les noms des variables adjoints à leur valeur. Ce dictionnaire est construit lors de la lecture de l'arbre associé au fichier `data`. Ensuite, il est passé en argument lors de la lecture de l'arbre associé au fichier `template` afin d'avoir l'accès aux variables définies préalablement

4.2 Implémentation des boucles

D'abord, deux types de boucles ont été distingués, le premier est utilisé avec un nom de variable donc avec une valeur récupérée dans le dictionnaire précédemment décrit. Le second concerne directement une liste de `string` à parcourir.

Dans le cas d'une variable au sein de la commande `for`, son nom sera agrémenté d'un nombre afin d'être différencié des noms de variables utilisés à l'extérieur de la boucle. Ce nombre est incrémenté à chaque `for` afin que cette technique puisse être employé plusieurs fois.

5 Problèmes rencontrés et solutions apportées

5.1 Analyse ascendante

Ma première idée était d'utiliser une technique d'analyse ascendante, or, j'ai vite eu un problème avec les boucles `for`. En effet, l'intérieur de la boucle se trouvait analyser avant de savoir qu'il s'agissait effectivement d'une boucle. Donc les variables associées à cette boucle étaient appelées avant d'être instanciées.

En solution, j'ai choisi de changer d'approche et de passer à une analyse descendante, Ceci à régler le problème au niveau des boucles `for` sans apporter d'autres problèmes. De plus, cela a permis d'éviter d'avoir un problème similaire lors l'implémentation des `if`.

5.2 Portée des variables

J'ai rencontré ce second problème au moment de tester le second exemple fourni. En effet, un même nom de variable est employé pour instancier une variable dans `data` puis il est utilisé pour parcourir une liste associée à une variable.

La solution que j'ai apportée consiste en l'utilisation d'un compteur et est décrite à la section 4.2.

6 Conclusion

Lors de ce projet, j'ai eu l'occasion de mieux me familiariser avec le processus d'interprétation ainsi qu'avec les différents outils associés. Toutefois, j'admets, en tant qu'autocritique, ne pas avoir correctement géré la portée des variables. La solution que j'ai apporté aux problèmes de boucle n'est pas fiable car ne permet pas de détecter une erreur lors de l'utilisation de boucles imbriquées avec un même nom de variable.

7 Annexe

La grammaire complète et modifiée :

<start>	⟶	<programme>
<programme>	⟶	<txt>
	|	<txt> <programme>
	|	<dumbo-bloc>
	|	<dumbo-bloc> <programme>
<txt>	⟶	Caractères alphanumériques, blancs ou spéciaux
<dumo-bloc>	⟶	{{ <expression-list> }}
	|	{{ }}
<expression-list>	⟶	<expression> ; <expression-list>
	|	<expression> ;
<expression>	⟶	<expression-var>
	|	<expression-print>
	|	<expression-for-lis>

	|	<expression-for-var>
	|	<expression-if>
<expression-var>	⟶	<variable-set> : = <string-expression>
	|	<variable-set> : = <string-list>
	|	<variable-set> : = <arith-expression>
<expression-print>	⟶	print <string-expression>
<expression-for-lis>	⟶	for <variable-set-for> in <string-list>
		do <expression-list> endfor
<expression-for-var>	⟶	for <variable-set-for> in <variable-get-str>
		do <expression-list> endfor
<expression-if>	⟶	if <test> do <expression-list> endif
<string-expresssion>	⟶	<string>
	|	<variable-get-str>
	|	<string-concat>
<string-concat>	⟶	<string-expression> . <string-expression>
<string-list>	⟶	(<string-list-interior>)
<string-list-interior>	⟶	<string>
	|	<string> , <string-list-interior>
<test>	⟶	<test> or <and-test>
	|	<and-test>
<and-test>	⟶	<and-test> and <comparison>
	|	<comparison>
<comparison>	⟶	<lower>
	|	<upper>
	|	<equal>
	|	<not-equal>
	|	<boolean>
<lower>	⟶	<arith-expression> < <arith-expression>
<upper>	⟶	<arith-expression> > <arith-expression>
<equal>	⟶	<arith-expression> = <arith-expression>
<not-equal>	⟶	<arith-expression> ! = <arith-expression>
<arith-expression>	⟶	<add-expr>
	|	<sub-expr>
	|	<term>
<add-expr>	⟶	<arith-expression> + <term>
<sub-expr>	⟶	<arith-expression> - <term>
<term>	⟶	<multi-expr>
	|	<div-expr>

	|	<factor>
<multi-expr>	⟶	<term> * <factor>
<div-expr>	⟶	<term> / <factor>
<factor>	⟶	<integer>
	|	<variable-get-int>
<integer>	⟶	SIGNED-INT
<variable-set>	⟶	CNAME
<variable-set-for>	⟶	CNAME
<variable-get-str>	⟶	VARIABLE-STR
<variable-get-int>	⟶	VARIABLE-INT
<string>	⟶	' Caractères alphanumériques '
<boolean>	⟶	<true>
	|	<false>
<true>	⟶	true
<false>	⟶	false
VARIABLE-STR	⟶	CNAME
VARIABLE-INT	⟶	CNAME