Dumbo interpreter

HUYLENBROECK Florent
BOSSART Laurent

Devoir pour le cours de compilation

UMONS
Annee academique 2019-2020

Contents

1 Introduction

Dans le cadre de notre cours de compilation, il nous a été demandé de créer le programme nommé *"dumbo_interpreter"*, permettant de générer facilement des fichiers contenant du texte. La génération se fait en fonction de données reçues en paramètres. Pour ce faire nous devions créer le langage *dumbo*.
Ce programme prend en entrée 3 paramètres :

- Un fichier data, contenant du code dumbo servant à initialiser des variables.
- Un fichier template, dans lequelle sera injecté les variables du fichier data, selon l'évaluation du code Dumbo contenu dans ce fichier template.
- Un fichier output, dans lequel sera écrit le résultat de la génération.

2 Grammaire Dumbo

En plus d'une grammaire de base donnée dans l'énoncé du devoir, certaines fonctionnalités additionnelles nous ont été demandées. Voici la grammaire complète qui a été implémentée :

$< programme >$	$\rightarrow$	$< txt >$
		$\mid < txt >< programme >$
$< programme >$	$\rightarrow$	$< dumbo\_block >$
		$\mid < dumbo\_block >< programme >$
$< txt >$	$\rightarrow$	$[a-zA-Z0-9;\&<>"\_=-.\setminus/\setminus n\setminus s:,]+$
$< dumbo\_block >$	$\rightarrow$	$\{\{< expression\_list >\}\}$
		$\mid \{\{\ \}\}$
$< expression\_list >$	$\rightarrow$	$< expression >; < expression\_list >$
		$\mid < expression >;$
$< expression >$	$\rightarrow$	$print < string\_expression >$
		$\mid for < variable > in < string\_list >$
		$do < expression\_list > endfor$
		$\mid for < variable > in < variable >$
		$do < expression\_list > endfor$
		$\mid < variable >:=< string\_expression >$
		$\mid < variable >:=< string\_list >$
		$\mid if < boolean\_expression > do < expression\_list > endif$
$< string\_expression >$	$\rightarrow$	$< string >$
		$\mid < comparable\_expression >$
		$\mid < string\_expression > . < string\_expression >$
$< string\_list >$	$\rightarrow$	$(< string\_list\_interior >)$
$< string\_list\_interior >$	$\rightarrow$	$< string >$
		$\mid < string >, < string\_list\_interior >$
$< variable >$	$\rightarrow$	$[a-zA-Z0-9\_]$
$< string >$	$\rightarrow$	$'[a-zA-Z0-9;\&<>"\_=-.\setminus/\setminus n\setminus s:,]+'$
$< comparable\_expression >$	$\rightarrow$	$< variable >$
		$\mid < number\_expression >$
$< number\_expression >$	$\rightarrow$	$[0-9]+(.[0-9]+)?$
		$\mid < number\_expression > [+-*/] < number\_expression >$
		$\mid - < number\_expression >$
$< boolean >$	$\rightarrow$	$true$
		$\mid false$
		$\mid < comparable\_expression > ([<>=]\mid!=) < comparable\_expression >$
$< boolean\_expression >$	$\rightarrow$	$< boolean >$
		$\mid < boolean > and < boolean\_expression >$
		$\mid < boolean > or < boolean\_expression >$

Certaines modifications ont été apportées à la grammaire de base :

- Une $< string\_expression >$ peut maintenant comprendre une $< comparable\_expression >$ à la place d'une $< variable >$. Cependant, une $< comparable\_expression >$ peut contenir une $< variable >$.
- Étant donné qu'il nous a été demandé de gérer la division, les nombres réels sont supportés, en extension aux nombres entiers.

3 Gestion des boucles *for* et *if*

Une fois les instructions booléennes correctement implementées, la gestion des boucles *if* ne nous a pas semblée compliquée. Nous l'avons gérée comme pour l'assignation, la concaténation, et le print : par la construction d'un tuple ("if", $boolean\_expression$, $instruction\_list$). Lors de l'interprétation du template, si un tel tuple est rencontré et que la condition est respectée, les instructions sont exécutées récursivement par notre fonction $apply\_function$ laquelle prend un tuple en entrée et applique l'opération définie par ce tuple.

Pour la boucle *for*, il a été nécessaire d'ajouter une fonctionnalité à notre interpréteur Dumbo : la gestion de la *profondeur*. Nous avons donc redéfini la structure de données permettant de stocker nos variables : d'un dictionnaire **variables**[nom] = *valeur*, nous sommes passés à un dictionnaire de dictionnaire **variables**[profondeur][nom] = *valeur*. Ainsi, à chaque entrée dans une boucle for, la profondeur est incrémentée d'un (valeur de départ = 0). Les valeurs de la $< string\_list >$ sur laquelle on itère sont copiées à cette profondeur-là. Le corps de la boucle est exécuté récursivement comme pour nos autres fonctions. Ensuite, à la sortie de cette boucle, le sous-dictionnaire à ce niveau de profondeur est effacé et la profondeur est décrémentée d'un.

Cette modification nous a amené à modifier plusieurs autres fonctions dans notre code. Par exemple, la fonction qui identifie une variable à partir d'une chaine de caractères a du être adaptée pour rechercher la variable en partant du niveau de profondeur actuel jusqu'au niveau 0, s'arrêtant à la première correspondance. L'assignation se fait maintenant de la même manière : si une variable correspondante est trouvée, une nouvelle valeur lui est assignée, sinon une variable est créée au niveau de profondeur actuel.

Cette modification nous permet de faire tourner plusieurs boucles *for* imbriquées, lesquelles peuvent utiliser le même nom de variable dans leurs corps respectifs.

4 Difficultés rencontrées

Une difficulté évidente a été de ne pas pouvoir se concerter en binôme dans la vraie vie, à cause du confinement. Mais ceci sort du cadre de cette section du rapport.
La première difficulté rencontrée a été de gérer l'appel de fonctions. Nous avons trouvé la solution dans la ressource qui nous a été fournie pendant les séances de travaux pratiques sur l'analyse lexicale : `http://www.dabeaz.com/ply/ply.html` (C'est d'ailleurs dans cette documentation que nous avons trouvé la majorité des réponses à nos problèmes). Nous avons donc suivi cette idée de créer un tuple dont le premier élement est le nom de la fonction et les suivants sont les paramètres nécessaires à l'exécution de cette fonction. Ainsi, l'exécution récursive des fonctions devenait possible afin de résoudre d'abord les arguments avant de calculer la valeur de la fonction.
La seconde difficulté rencontrée a été la gestion du *for*. Nous avons essayé plusieurs alternatives avant d'en arriver à utiliser notre système de profondeur. Nous avons d'abord commencé par séparer nos variables en deux listes : **variables** et **variables_loop** mais cette solution, bien qu'au final fonctionelle pour la gestion d'une seule boucle *for*, ne nous convenait pas.

5 Conclusion

Ce projet était intéressant dans le cadre de notre cours de compilation. En plus d'approfondir notre compréhension du fonctionnement d'un compilateur, il nous a permis de découvrir un type de programmation que nous n'avions pas rencontré avant. Nous sommes satisfaits des solutions que nous avons mises en place afin de résoudre les problèmes demandés.