Dumbo interpreter

Rapport de projet de compilation

Année académique 2016-2017

Auteurs :
MAAZOUZ Mehdi
LECOCQ Alexis

Directeurs :
BRUYÈRE Véronique
DECAN Alexandre

15 mai 2017

Table des matières

1 Introduction **2**

2 Langage Dumbo **3**

2.1 Grammaire utilisée 3

2.2 Différences avec la grammaire fournie 4

2.3 Précédence des opérateurs 4

3 Interpréteur dumbo **5**

3.1 Note sur les instructions "if" et "for" 5

3.2 Structure du programme 5

3.2.1 Analyse lexicale 5

3.2.2 Analyse syntaxique 5

3.2.3 Analyse sémantique/exécution 5

4 Manuel d'utilisation **6**

5 Conclusion **7**

1 Introduction

Dans le cadre du cours de compilation, nous devons réaliser un projet afin de mettre en pratique la théorie vue au cours. Le projet doit être écrit en Python 3 et utiliser la librairie ply.

L'objectif du projet est de réaliser un moteur de template à l'aide d'un langage créé pour l'occasion : le dumbo. Nous en décrirons la grammaire dans un prochain chapitre.

Un moteur de template est principalement utilisé pour séparer les données de la manière de les représenter. Notre script dumbo_interpreter.py doit recevoir trois arguments :

- data_file : fichier dumbo contenant les données ;
- template_file : fichier dumbo contenant la présentation des données ;
- output_file : fichier de sortie contenant le fichier template dans lequel les données ont été insérées.

2 Langage Dumbo

2.1 Grammaire utilisée

Afin de simplifier la lecture du tableau, nous avons utilisé des expressions régulières dans certaines règles. Ces dernières sont situées dans la partie basse du tableau pour éviter toute confusion avec les autres règles. Voici la grammaire du langage :

<program>	⟶	<program> <subprogram> | <subprogram>
<subprogram>	⟶	<text> | {{ <codeblock> }} | {{ }}
<codeblock>	⟶	<codeblock> <codeline> | <codeline>
<codeline>	⟶	<instruction> ;
<instruction>	⟶	print <value>
<instruction>	⟶	<variable> := <value>
<instruction>	⟶	for <variable> in <variable> do <codeblock> endfor
<instruction>	⟶	for <variable> in <stringlist> do <codeblock> endfor
<instruction>	⟶	if <boolop> do <codeblock> endif
<value>	⟶	<variable> | <boolop> | <intop> | <stringop> | <stringlist>
<boolop>	⟶	<boolop> and <bool> | <boolop> or <bool> | <bool>
<bool>	⟶	true | false | <variable> | <intop> <comparator> <intop>
<comparator>	⟶	< | > | = | !=
<stringop>	⟶	<stringop> . <string> | <string>
<string>	⟶	<variable> | <string_regex>
<stringlist>	⟶	() | (<stringseq>)
<stringseq>	⟶	<stringseq> , <string> | <string>
<intop>	⟶	<intop> + <term> | <intop> - <term> | <term>
<term>	⟶	<term> * <factor> | <term> / <factor> | <factor>
<factor>	⟶	<integer_regex> | <variable>
Expressions régulières :		
<text>	⟶	([^{]|{[^{])+
<variable>	⟶	[A-Za-z_][A-Za-z0-9_]+
<integer_regex>	⟶	[0-9]+
<string_regex>	⟶	'[^']*'

Dans cette grammaire, <value> ⟶ <variable> est optionnel car cette réduction peut être donnée à trois reprises par <value> ⟶ <variable> | <boolop> | <intop>. Cependant, définir cette règle en premier dans yacc permet de "court-circuiter" les autres réductions et cela est plus clair pour le lecteur.

2.2 Différences avec la grammaire fournie

Cette grammaire correspond à la grammaire fournie dans l'énoncé, excepté quelques modifications :

- <codeblock> peut être vide afin de traiter l'exemple 3 fourni en annexe sans erreur ;
- les booléens true et false ainsi que les opérateurs booléens "and" et "or" ont été ajoutés avec une précédence gauche ;
- le if permet d'exécuter une série d'instructions seulement si une condition booléenne est vraie ;
- les entiers, ainsi que les opérations entières "+", "-", "*" et "/", ont été ajoutés avec la même précédence qu'en mathématique ;
- print peut afficher, en plus d'une chaîne de caractères, un booléen, un entier ou une liste ;
- une variable peut contenir, en plus d'une chaînes de caractères, un booléen, un entier ou une liste ;
- <text> peut contenir tous les caractères qui ne correspondent pas à l'ouverture d'un bloc de code, ce qui rend le moteur de template plus flexible (pas de caractère interdit). En effet, <text> contient une suite de longueur minimale 1 de :
  - soit une lettre qui n'est pas l'accolade ouvrante ;
  - soit une accolade ouvrante suivie d'une lettre qui n'est pas l'accolade ouvrante ;
- <variable> ne peut pas commencer par un chiffre afin de différencier une variable d'un entier ;
- <string_regex> peut désormais contenir n'importe quel caractère qui n'est pas un simple guillemet (pas de caractère interdit).

2.3 Précédence des opérateurs

Nous avons établi une grammaire non ambiguë qui respecte les précédences annoncées (gauche pour les booléens et mathématique pour les entiers). Nous n'avons donc pas eu besoin d'utiliser la variable "precedence" fournie par yacc.

3 Interpréteur dumbo

3.1 Note sur les instructions "if" et "for"

Avant l'introduction des instructions "if" et "for", l'exécution du code se faisait en même temps que l'analyse syntaxique. Cependant, cette dernière attribue une valeur aux composants d'une règle avant d'entrer dans la fonction de cette règle. Il était alors impossible d'empêcher l'exécution d'une instruction "if" car le code était exécuté avant d'entrer dans la fonction. De manière analogue, la variable de boucle déclarée dans la fonction for ne pouvait être connue des instructions du "for", étant donné que ces instructions étaient exécutées avant la fonction "for". Nous avons donc décidé d'opter pour une structure plus modulaire avec compilation avant exécution.

3.2 Structure du programme

L'exécution d'un script dumbo se déroule en 3 étapes :
- Analyse lexicale
- Analyse syntaxique
- Analyse sémantique/exécution

3.2.1 Analyse lexicale

La première étape consiste à analyser l'ensemble des lexèmes présents dans le fichier source. Cette étape est réalisée dans le fichier lex.py. Pour de plus amples informations sur les lexèmes utilisés, nous vous invitons à consulter ce fichier.
Entrée : un ensemble ordonné de caractères, le contenu du fichier source.
Sortie : un ensemble ordonné de lexèmes.

3.2.2 Analyse syntaxique

La seconde étape consiste à analyser la syntaxe du fichier et compiler les fonctions/opérations. Une fonction/opération compilée est en fait un tuple python dont :
- le premier élément est une chaîne de caractères représentant la fonction/opération ;
- le second élément est un tuple contenant des informations sur l'emplacement de la fonction/opération afin de faciliter le débogage :
  1. le nom du fichier ou input ;
  2. le numéro de ligne ;
  3. la position sur la ligne ;
- les éléments suivants sont les arguments de la fonction/opération (leur nombre est variable).

Entrée : un ensemble ordonné de lexèmes.
Sortie : un ensemble ordonné de fonctions/opérations.

3.2.3 Analyse sémantique/exécution

La troisième et dernière étape consiste à exécuter les fonctions et évaluer les opérations. Étant donné que les variables sont dynamiquement typées, l'analyse sémantique est également réalisée à cette étape.
Entrée : un ensemble ordonné de fonctions/opérations.
Sortie : un ensemble ordonné de caractères, la sortie de l'exécution des instructions.

4 Manuel d'utilisation

Le programme se lance en ligne de commande grâce à python3 avec l'ensemble des paramètres définis dans l'introduction. Exemple d'utilisation du moteur de template :

```
$ python3 dumbo_interpreter.py data_file template_file output_file [-d|--debug]
```

Cette commande va initialiser les variables présentes dans le fichier "data_file" et les conserver pour exécuter le fichier "template_file". La sortie de ce dernier sera écrite dans le fichier "output_file". Si une erreur survient, les détails seront affichés dans la console. L'argument -d ou --debug est optionnel. S'il est présent, lex et yacc seront exécutés en mode debug (des informations supplémentaires seront affichées dans la console).

À des fins de débogage, vous pouvez utiliser les commandes suivantes :

```
$ python3 lex.py dumbo_file [-d|--debug]
```

Cette commande va afficher les lexèmes détectés dans le ficher "dumbo_file" ainsi que leur position. L'argument -d ou --debug a le même effet que pour le script "dumbo_interpreter.py".

```
$ python3 yacc.py dumbo_file [-d|--debug]
```

Cette commande va afficher la liste d'instructions contenues dans le fichier "dumbo_file". L'argument -d ou --debug a le même effet que pour le script "dumbo_interpreter.py".

```
$ python3 execute.py dumbo_file [-d|--debug]
```

Cette commande va exécuter les instructions contenues du fichier "dumbo_file" et afficher la sortie du programme dans la console. L'argument -d ou --debug a le même effet que pour le script "dumbo_interpreter.py".

Parce qu'une ligne de code vaut mieux qu'un long discours, nous vous invitons à lire le code pour avoir des exemples d'utilisation si vous souhaitez utiliser les modules lex, yacc et execute.

5 Conclusion

Nous avons bien réalisé l'objectif fixé dans l'introduction, à savoir créer un moteur de template à l'aide d'un nouveau langage. Nous avons ainsi eu l'occasion de mettre en pratique et d'approfondir les concepts vus au cours théorique notamment les grammaires non ambiguë, les analyses lexicale, syntaxique et sémantique. Nous tenons à remercier la titulaire BRUYÈRE Véronique ainsi que l'assistant DECAN Alexandre pour la patience (notamment envers les arrivées tardives) et le dévouement dont ils ont fait preuve cette année.