Université de Mons
Faculté des Sciences
Département des mathématiques

Rapport du projet de Compilation

Dumbo code

Professeurs :
Véronique Bruyère
Alexandre Decamps

Auteurs :
Loïc Dupont
Clément Durieux

Année académique 2020-2021

Table des matières

1 Introduction.

Le projet que nous avons réalisé veuille à accomplir l'objectif décrit ci-après. On se donne deux fichiers : un fichier dit *template* et un autre, appelé *data*, qui contient des variables faisant référence à des informations dont on a besoin pour compléter le fichier *template*. On veut obtenir un résultat qui est le fichier *template* que l'on a complété là où cela était nécessaire (dans des blocs appelés *dumbo_bloc* qui sont délimités par des "{{}}") grâce aux informations présentes dans le fichier *data*. Ce qui est en dehors des *dumbo_bloc* n'est pas modifié. Nous avons fait le choix d'opter pour LARK dans la réalisation de ce projet.

2 Mode d'emploi.

Vous trouverez dans le fichier compressé plusieurs dossiers ainsi qu'un $README.md$ et l'énoncé du projet. Dans le dossier *exemples*, se trouve l'ensemble des templates fournis lors du projet et qui ont été utilisés lors de la réalisation de celui-ci. Dans le dossier *src*, se trouve notre grammaire (fichier $lark\_grammar.lark$), notre code pour la réalisation du compilateur (fichier *dumbo.py*), une structure de donnée utilisée dans l'élaboration du projet (fichiers $Node.py$ et $liste\_chainee.py$) ainsi que les tests unitaires du projet (fichiers *dumbotester.py* et *dumbotester*2.*py*). Pour exécuter le compilateur, nous utiliserons le fichier *dumbo.py*. Pour ce faire, on insère dans le dossier *exemples* un $fichier\_data.dumbo$ correspondant à des assignements de variables (pouvant être vides) et un $fichier\_template.dumbo$ qui correspond à la structure globale du code où notre compilateur devra afficher certains éléments. Après avoir placé ces deux fichiers dans le dossier *exemples*, mettez-vous dans le dossier globale (*Dumbo*) et lancez la commande

$$python3\ src/dumbo.py\ exemples/fichier\_data.dumbo\ exemples/fichier\_template.dumbo$$

Si aucune erreur de syntaxe ou de sémantique a été trouvée, notre compilateur va exécuter le code présent dans les fichier $fichier\_data.dumbo$ et $fichier\_template.dumbo$. Une réponse vous sera fourni dans la console qui est votre code compilé avec la syntaxe dumbo. Nous avons aussi mis aussi à la disposition 2 fichiers de tests unitaires. Pour les exécuter, il suffit de lancer la commande

*python*3 *dumbotester.py* ou *python*3 *dumbotester*2.*py*

La première commande exécute un très grand nombre de cas possibles tandis que la deuxième commande exécute seulement ceux qui sont essentiels pour le projet.

3 La grammmaire.

La grammaire utilisée dans le cadre de ce projet est la grammaire de base du *dumbo* que l'on a complétée et modifiée. Nous allons passer en revue les différentes modifications et les rajouts que nous avons effectué à la grammaire de base.

Tout d'abord, nous avons rajouté la transition allant de *txt* vers (/[^{]/)+ qui nous assure que *txt* ne contienne pas d'accolade ouvrante. Par la même occasion, cela empêche *txt* d'accepter le marqueur de début d'un *dumbo_bloc* (une double accolade ouvrante).

Nous avons ensuite rajouté une transition partant de *dumbo_bloc* qui permet de gérer le cas où le *dumbo_bloc* est uniquement composé de doubles accolades ouvrantes puis fermantes.

Nous avons également effectué des modifications concernant les transitions partant de *expression*. En effet, on a rajouté des transitions allant de *expression* vers $print\_expr$, for_expr, $assign\_expr$ et if_expr. Ces transitions serviront à traiter les cas où on rencontre respectivement un *print*, un for, une assignation et un if.

Détaillons les transitions partant de ces quatre derniers. Pour $print\_expr$, on a la transition allant vers "*print*"($string\_expression \mid INT$) qui permet d'effectuer des affichages (*print*) de chaînes de caractères et des entiers. Concernant for_expr, on a la transition allant vers "for" $VARIABLE$ "*in*" ($string\_list \mid VARIABLE$) "*do*" $expression\_list$ "*endfor*" qui permet de gérer le cas où l'on rencontre un for (selon si l'on traite une liste de *string* ou une variable).

Dans le cas de $assign\_expr$, on a une transition allant vers $assign\_expr\_arith \mid assign\_expr\_var$ afin de gérer les cas où on effectue une assignation avec un entier ($assign\_expr\_arith$) ou avec autre chose ($assign\_expr\_var$). Pour if_expr, on a la transition vers "if" $bool\_expr$ "*do*" $expression\_list$ "$endif$" ce qui traite le cas où on rencontre un if.

On ajoute des transitions partant de add_expr et mul_expr qui nous permettent de gérer les additions,

soustractions, divisions et multiplications d'entiers dans le cas d'une assignation (sauf si il n'y que des multiplications et des divisions car cela nous produirait un conflit donc on gère ce cas avec une autre transition).

Nous définissons également des transitions partant de inf_expr, sup_expr, eq_expr et dif_expr qui nous permettent de gérer les comparaisons d'entier.

Les transitions partant de add_int, $ADD\_OPERATION$, mul_int et $MUL\_OPERATION$ servent à gérer les additions, soustractions, divisions et multiplications d'entiers dans le cas d'une comparaison d'entier.

Il y a également une transition partant de $bool\_expr$ qui vise à traiter des booléens sous toutes les formes possibles ($true$, $false$, comparaison d'entier, etc) ainsi que les opérations *or* et *and*.

On a une transition partant de $string\_expression$ permettant de gérer ce qui est fait dans la grammaire de base et les multiplications et divisions d'entiers lors d'une assignation.

Les transitions partant de *string* et $STRING\_INTERIOR$ permettent de gérer les *strings* c'est-à-dire les chaînes de caractères composées de n'importe quelle suite de caractères entourées de guillemets simples. Celle partant de $VARIABLE$ permet de gérer les variables. Les noms de variables contiennent uniquement des caractères alphanumériques ou underscore et ne peuvent débuter par un chiffre.

Les autres transitions sont triviales ou font partie de la grammaire de base du dumbo.

4 Analyse sémantique.

4.1 Structure de données utilisée.

Dans nos différentes méthodes, nous utilisons un arbre $Tree$ qui est défini grâce à notre grammaire et LARK. Il est composé d'un attribut *data* (la racine de l'arbre) qui peut contenir divers objets et d'un attribut *children* qui est une liste contenant l'ensemble de ses sous-arbres qui sont soit des arbres $Tree$ soit des $Tokens$, les $Tokens$ ont également deux attributs : *type* qui nous donne le type du $Token$ (c'est forcément un symbole terminal) et *value* qui est la valeur du $Token$. Par exemple, dans le cas où l'on cherche à gérer un if (qui est un arbre $Tree$), l'attribut *data* contiendra le if_expr et l'attribut *children* contiendra une liste ayant pour premier élément, toute la séquence $bool\_expr$ (qui est ce qui doit être évalué comme étant vrai afin d'effectuer l'action dans le if) et comme deuxième élément, toute la séquence $expression\_list$ qui est l'ensemble des actions mentionnées à l'intérieur du if. La présence des différents éléments dans *children* est donnée par la grammaire.

4.2 La classe $OurInterpreter$ et les fonctions dont elle hérite.

Notre classe $OurInterpreter$ hérite de la classe $Interpreter$ étant définie dans LARK et qui nous donne accès à plusieurs choses utiles notamment les fonctions *visit* et $visit\_children$ qui seront très importantes lors de notre analyse sémantique. En ce qui concerne *visit*, nous l'utilisons la plupart du temps lors d'appels du type : $self.visit(tree.children[0])$. Un tel appel nous permet d'explorer entièrement le $Tree$ qui est le premier fils de notre arbre (il suffit de remplacer 0 par le chiffre n pour effectuer la même opération avec le $n+1$ ème fils de notre arbre). Pour ce qui est de $visit\_children$, elle nous permet de visiter tous les fils de notre arbre.

4.3 Gestion des scopes.

Afin de gérer les scopes, nous utilisons une liste chainée sous le format LIFO qui contient l'ensemble des dictionnaires utilisés dans les différentes scopes. Les nouveaux dictionnaires sont donc insérés au début de la liste chainée. Chaque dictionnaire correspond à une scope particulière. Lorsque nous entrons dans une nouvelle scope, un nouveau dictionnaire est créé. Ce dernier contient les mêmes données que le dictionnaire le plus récent si ce n'est que les variables définies à la fois dans la scope actuelle et la précédente sont associées à la valeur qu'elles ont dans la scope actuelle. On insère ensuite le dictionnaire que l'on considérait dans la scope précédente dans la liste chainée. Lorsque nous sortons d'une scope, le premier dictionnaire de la liste chainée est supprimé. On le récupère ensuite car il s'agit du dictionnaire à utiliser dans la scope dans laquelle on vient de rentrer. Ce raisonnement nous assure à la fois d'avoir toujours le dictionnaire qui se rapporte à la scope dans laquelle on se trouve mais aussi d'avoir le dictionnaire lié à l'éventuelle prochaine scope en première position de la liste chainée.

4.4 Gestion du if.

Dans notre implémentation, notre arbre $Tree$ a toujours deux fils lorque l'on gère un if. On appelle la fonction *visit* sur le premier fils qui correspond à la condition du if. Ensuite, on teste si cette condition est vraie et, dans ce cas, on appelle *visit* sur le deuxième fils ce qui a pour effet d'effectuer les actions reprises dans la boucle if. Remarquez que, de par notre grammaire, pour accéder à un if, on doit passer par une *expression* et les scopes sont gérées lors de cette étape.

4.5 Gestion du for.

On commence par définir trois variables comme étant les trois sous-arbres de $Tree$ qu'on est en train de considérer : *var* qui est une variable et un $Token$, *liste_elem* qui est soit une *string_list* et un $Tree$ soit une variable et un $Token$ et *liste_expr* qui est une *expression_list* et un $Tree$. Ces attributs sont les attributs utilisés dans le for, c'est-à-dire que le for est de la forme :

$$"for"\ var\ "in"\ liste\_elem\ "do"\ liste\_expr\ "endfor"$$

(la nature de ces trois attributs correspond bien à ce qui a été dit précédemment car cela est assuré par notre grammaire). On effectue ensuite un test afin de savoir si *liste_elem* est un $Token$. Dans ce cas, *liste_elem* est une variable et, pour chacun de ses éléments (il peut y en avoir plusieurs car dans notre énoncé on suppose qu'il s'agit d'une liste de *string*), on rajoute l'élément dans le dictionnaire en cours d'utilisation, on appelle ensuite la fonction *visit_children* sur *liste_expr* ce qui a pour effet d'effectuer les actions décrites dans *liste_expr* et on récupère finalement le dictionnaire précédant. Dans le cas où *liste_elem* n'était pas un $Token$, on effectue des opérations similaires, si ce n'est que les appels pour modifier les dictionnaires dont on dispose sont effectués sur des objets différents mais la logique reste la même.

5 Problèmes rencontrés et soluions.

Nous avons rencontré un premier souci lorsque nous avons dû compléter la grammaire. En effet, nous ne savions pas comment faire en sorte que *txt* accepte n'importe quel caractère à l'exception des doubles accolades ouvrantes. Lorsque nous avons appris qu'il était suffisant de considérer que *txt* n'accepte pas le caractère de l'accolade ouvrante, nous avons trouvé une solution en utilisant l'expression régulière (/[^{]/)+ qui permet de ne pas accepter le caractère "{", on l'utilise donc pour créer notre *txt*. Cela nous assure bien que *txt* accepte toute suite de caractères (non vide) qui ne contient pas d'accolade ouvrante. Plus généralement, nous avons rencontré plusieurs soucis avec la grammaire, ce qui a nécessité de faire de nombreux changements afin de gérer au mieux les différents cas possibles. Ces changements nous ont souvent poussés à devoir également modifier *dumbo.py*. Nous avons également rencontré des problèmes de compréhension concernant LARK de manière générale et de certaines fonctions qui y sont définies (notamment les fonctions *visit* et *visit_children*) ce qui nous a bloqué un certain temps et nous a demandé de faire de plus amples recherches afin de pouvoir poursuivre notre travail. Enfin, pour être sûrs de ne plus avoir de problèmes, nous avons décidé de faire un maximum de tests unitaires visant à traiter une grande partie des cas possibles, il y en a donc un très grand nombre. Ces tests se trouvent dans le fichier *dumbotester.py*.

6 Répartition du travail.

En ce qui concerne la répartition du travail, nous avons d'abord commencé par nous renseigner sur LARK. Nous avons ensuite commencé à définir la grammaire et une partie de notre algorithme *dumbo.py* ensemble. Cependant, vu que les problèmes rencontrés nous avaient fait prendre du retard, nous avons décidé de séparer le travail afin de rendre le projet dans les temps. Loïc s'est donc chargé de finir la partie programmation pendant que Clément s'est occupé du rapport. Bien évidemment, nous suivions chacun les progrès de l'autre de manière assez régulière afin de s'assurer que nous soyons bien d'accord sur la solution proposée et que nous comprenions bien le travail effectué par l'autre mais également afin de s'entraider lorsque l'un de nous bloquait sur sa partie.

7 Conclusion.

Au final, nous avons trouvé ce projet intéressant car, contrairement à d'autres, il est facile de voir qu'il pourrait être utilisé dans la vie de tous les jours et qu'il a donc une réelle utilité. En revanche, nous avons trouvé que la réussite du projet dépendait plus de notre compréhension de LARK que de la compréhension du cours de compilation. Ce projet a également été légèrement frustrant mais enrichissant de par la multitude de problèmes plus ou moins importants que nous avons rencontrée ainsi que l'énergie et le temps qu'il a fallu consacrer dans le but de les résoudre.