Université de Mons
Faculté des sciences
Département d'Informatique

Compilation
Rapport de projet

Professeur :
Véronique Bruyère
Alexandre Decan

Auteur :
Pauline Joly
Arnaud Palgen

Année académique 2018-2019

Table des matières

1 Introduction

Pour le projet de compilation, il nous a été demandé de réaliser un compilateur d'un petit language nommé dumbo. Ce language permet d'injecter des données à partir d'un fichier data dans un un fichier template.

2 Description de la grammaire

<programme>	$\longrightarrow$	<txt> | <txt> <programme>
<programme>	$\longrightarrow$	<dumbo-bloc> | <dumbo-bloc> <programme>
<txt>	$\longrightarrow$	[a-zA-Z0-9 ;&<>"_-.\/\n:,]+
<dumbo-bloc>	$\longrightarrow$	{{ <expression-list> }}
<expression-list>	$\longrightarrow$	<expression> | <expression> ; <expression-list> | ϵ
<expression>	$\longrightarrow$	print <string-expression>
<expression>	$\longrightarrow$	print <variable>
<expression>	$\longrightarrow$	for <variable> in <string-list> do <expression-list> endfor
<expression>	$\longrightarrow$	for <variable> in <variable> do <expression-list> endfor
<expression>	$\longrightarrow$	<variable> : = <string-expression>
<expression>	$\longrightarrow$	<variable> : = <string-list>
<expression>	$\longrightarrow$	<variable> : = <integer-expression>
<expression>	$\longrightarrow$	<variable> : = <boolean>
<expression>	$\longrightarrow$	if <boolean-expression> do <expression-list> endif
<string-expression>	$\longrightarrow$	<string>
<string-expression>	$\longrightarrow$	<string-expression> . <string-expression>
<string-expression>	$\longrightarrow$	<variable> . <string-expression>
<string-expression>	$\longrightarrow$	<string-expression> . <variable>
<string-expression>	$\longrightarrow$	<variable> . <variable>
<string-list>	$\longrightarrow$	(<string-list-interior>)
<string-list-interior>	$\longrightarrow$	<string> | <string> , <string-list-interior>
<boolean-expression>	$\longrightarrow$	<boolean-expression> <logical-operator> <boolean-expression>
<boolean-expression>	$\longrightarrow$	<comparator-expression>
<boolean-expression>	$\longrightarrow$	<boolean>
<comparator-expression>	$\longrightarrow$	<integer-expression> <comparator> <integer-expression>
<integer-expression>	$\longrightarrow$	<integer>
<integer-expression>	$\longrightarrow$	<integer-expression> <op> <integer-expression>
<integer-expression>	$\longrightarrow$	<variable> <op> <integer-expression>
<integer-expression>	$\longrightarrow$	<integer-expression> <op> <variable>
<integer-expression>	$\longrightarrow$	<variable> <op> <variable>
<integer>	$\longrightarrow$	[0-9]+
<boolean>	$\longrightarrow$	true | false
<logical-operator>	$\longrightarrow$	and | or
<op>	$\longrightarrow$	+ | - | * | /
<comparator>	$\longrightarrow$	< | > | = | ! =
<variable>	$\longrightarrow$	[a-zA-Z0-9_]+
<string>	$\longrightarrow$	'[a-zA-Z0-9 ;&<>"_-.\/\n:,=]+'

3 Gestion du "for" et du "if"

3.1 Gestion du "for"

1. On commence par regarder si ce qui se trouve après le "in" est une <variable> contenant une <string-list> ou une <string-list>.
2. Si c'est une <string-list>, on place sa valeur dans *list_value* et on passe directement au point 4.. Sinon, on passe au point 3..
3. On va chercher dans le dictionnaire *variables* la valeur <variable> que l'on place dans *list_value*.
4. Ensuite, on regarde si la <variable> se trouvant juste après le "for" a déjà été utilisée. Si oui, on passe au point 5.. Sinon, on va au point 6..
5. On stocke alors la valeur initiale de cette <variable> dans *var_value*.
6. On instaure une boucle for de façon à ce qu'à chaque valeur de *item*, cette valeur est mise dans le dictionnaire *variables* au nom de la première <variable>. Ensuite, on effectue chaque <expression> de la <expression-list> pour chaque valeur de *item*.
7. Dans le cas où le nom de la variable avait déjà été utilisée, on remplace la valeur présente dans *variables* par *var_value*. Sinon, on retire la variable de *variables*.

3.2 Gestion du "if"

1. Lorsque l'on arrive dans un noeud "if", on utilise une fonction **lambda** tel que si le premier fils est vrai alors on execute le second fils.
2. La gestion du premier fils est géré par la fonction *eval_boolean_expression*. Puis, si nécessaire, par *eval_comparator_expression*.

4 Présentation des problèmes et solutions envisagées ou appliquées

4.1 Analyse lexicale

1. Pour garder la mise en forme du template, nous avons dû prendre en compte les espaces et les retours à la ligne dans les textes (<txt>) mais pas dans les dumbo blocs. Pour cela, nous avons utilisé les états de l'analyseur lexicale. Ainsi, quand l'analyseur n'est pas dans le mode *inBloc*, il prend en compte les espace et les retours à la ligne. Cependant, lorsqu'il rencontre un retour à la ligne, il incrémente quand même son nombre de ligne pour faciliter la gestion des erreurs lexicales.
2. Un problème non résolu est que l'indication du numéro de ligne d'une erreur n'est pas précis. En effet, dans le cas où l'erreur survient après un " ;", toutes les lignes suivantes sont indiquées comme s'il y avait une erreur de syntaxe à chaque ligne.

4.2 Analyse syntaxique

1. L'execution directe du programme commme lors des séances de travaux pratiques n'etait pas possible car le language est plus complexe que les opérations des séances de travaux pratiques (ex : boucle for, if, etc). Pour cela nous avons construit un arbre syntaxique comme vu au cours.

2. Nous avons rencontrés plusieurs conflits shift/reduce ou reduce/reduce.Nous avons résolu les reduce/reduce en ajoutant un ordre de précédance par *precedence*. Nous avons résolu les shift/reduce en enlevant le shift dans l'état qui posait problème. Nous l'avons fait en créant directement le noeud à l'intèrieur des fils du noeud de l'expression. Par exemple, si l'on prend *def p_expression_string_expression*, dans le cas où "len(p) == 4", on peut voir que le noeud "string_expression" contient parmi ses fils le noeud "point".
3. L'ajout de nouveaux éléments dans le language a impliqué quelques modifications dans la grammaire.RAJOUT : Par exemple, nous ne pouvions plus laisser la production <string-expression> $\longrightarrow$ <variable>. En effet, lors de la modification de la grammaire, nous avons dû rajouter les variables entières ce qui nous donnait la production suivantes : <integer-expression> $\longrightarrow$ <variable>. Malheureusement, cela nous a donné un conflit reduce/reduce. Nous avons résolu ce problème en faisant intervenir <variable> un peu plus loin dans les productions.
4. Un problème non résolu est que l'indication du numéro de la ligne d'une erreur n'est à nouveau pas précise.

4.3 Analyse sémantique

Nous n'avons pas rencontré de problèmes lors de la réalisation de l'analyse sémantique.

4.4 Interpréteur

1. Si l'arbre syntaxique n'etait pas correct dû à des erreurs de syntaxiques ou lexicales, il y avait des erreurs lors de l'évaluation de l'arbre. Pour cela nous avons rajouté une variable globale booléenne ce qui nous permet de savoir quand une erreur s'est produite sans arrêter l'analyse lexicale et syntaxique. Ainsi, lorsqu'une erreur est rencontrée, l'évaluation de l'arbre ne se fait pas.
2. Nous avons rencontré un problème avec la variable globale *variables*. Qui, lors d'une nouvelle analyse syntaxique est réinitialisée. Pour cela, nous mémorisons sa valeur après la première analyse et concaténons cette dernière avec celle de la deuxième analyse pour avoir un dictionnaire de variables complet.

5 Conclusion

Ce projet nous a permis de mettre en pratique ce que nous avons vu au cours théorique ce qui était très intéressant. Cependant, nous avons trouvé que la création de l'analyse lexicale et syntaxique était un travail fort répétitif.