



Université
de Rennes

TLC-S7: Rapport Compilateur while

A rendre le 24/01/2024

Responsable: Fabrice Lamarche

LE DILAVREC Titouan, MOREAU de LIZOREUX Nicolas, CHAUVEL Tom, JOSSO Célia

Sommaire

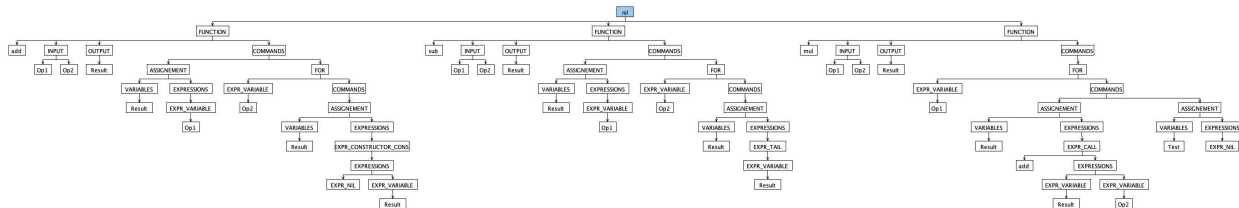
1. Description technique	3
1.1. AST	3
1.2. Architecture	4
1.2.1. Design Pattern Visiteur	4
1.3. Analyse syntaxique et sémantique	4
1.3.1. Table des symboles	4
1.3.2. Vérification des assignements et des paramètres	5
1.4. Génération de code 3 adresses à partir de l'AST	5
1.5. Génération de code à partir du code 3 adresses	6
1.6. Bibliothèque runtime de WHILE écrite dans le langage cible	6
2. Description de la validation du compilateur	7
2.1. Méthodologie utilisée	7
2.2. Code coverage	7
3. Bilan	7
3.1. Ce qui fonctionne... Ou non	7
3.2. Fonctionnalités restantes à implémenter	7
4. Description de la méthodologie de gestion de projet	8
4.1. Outils utilisés pour la gestion du projet	8
4.2. Etapes de développement et découpage des tâches	8
4.3. Rapport de travail individuel	8
5. Post mortem : Organisation du projet	9
5.1. Ce qui a bien fonctionné	9
5.2. Ce qui a moins bien fonctionné	9
5.3. Avec plus de recul, que ferions-vous ?	9

1. Description technique

Dans cette partie, il s'agira de montrer une vue d'ensemble de l'architecture du compilateur et ainsi que de la chaîne de compilation

1.1. AST

Voici notre AST. Nous avons essayé de le rendre le plus propre possible. Nous l'avons construit à partir du fichier `integers.while` (répertoire `test/lang`) :

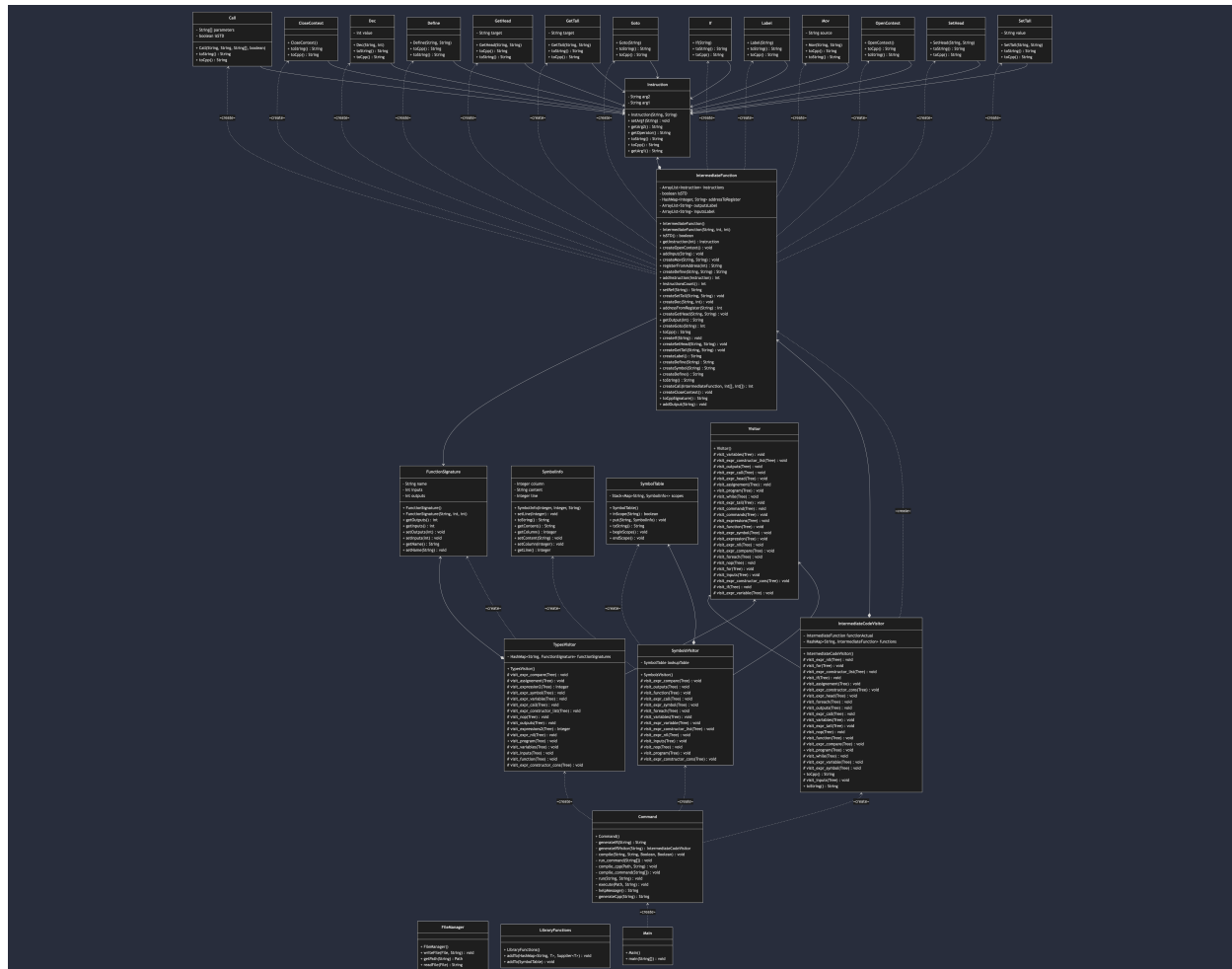


Sur cet AST, nous remarquons que notre programme contient 3 fonctions :

- Une fonction `add`
 - 2 paramètres d'entrée : `Op1` et `Op2`
 - 1 paramètre de sortie : `Result`
 - Une suite de commandes :
 - Une assignation stockée dans la variable `Result` prenant la valeur de `Op1`
 - Une boucle `for` itérant sur `Op2`. Elle stocke dans `Result` la construction d'un arbre ayant pour fils gauche `nil` et pour fils droit `Result`
- Une fonction `sub`
 - 2 paramètres d'entrée : `Op1` et `Op2`
 - 1 paramètre de sortie : `Result`
 - Une suite de commandes :
 - Une assignation stockée dans la variable `Result` prenant la valeur de `Op1`
 - Une boucle `for` itérant sur `Op2`. Elle stocke dans `Result` la `tail` de `Result`
- Une fonction `mul`
 - 2 paramètres d'entrée : `Op1` et `Op2`
 - 1 paramètre de sortie : `Result`
 - Une boucle `for` itérant sur `Op1`. Elle stocke dans `Result` le résultat de la fonction `add` qui est appelée sur les paramètres `Result` et `Op2`

1.2. Architecture

Ci-dessous notre diagramme de classe de notre compilateur.



1.2.1. Design Pattern Visiteur

Nous avons mis en place une classe abstraite `Visitor.java` se basant sur le Design Pattern visitor. Elle permet de visiter n'importe quel label présent dans l'AST (fonctions, inputs, outputs, expressions, variables etc.)

Grâce à cette classe abstraite, nous avons pu faire un visiteur pour la table des symboles (`SymbolsVisitor.java`). Le visiteur est classe permettant de naviguer dans l'arbre de navigation syntaxique depuis la racine de l'arbre jusqu'aux feuilles.

1.3. Analyse syntaxique et sémantique

Dans cette partie et les suivantes, nous traiterons de notre méthodologie pour effectuer les analyses syntaxique et sémantiques

1.3.1. Table des symboles

Nous avons besoin d'une table des symboles pour vérifier que les variables et les fonctions sont bien initialisées avant d'être appelées, et pour vérifier types. Pour cela :

- Nous avons implémenté une classe `SymbolInfo` qui a pour attributs `line` (numéro de ligne), `column` (numéro de colonne) et `content` (contenu du symbole). Elle permet d'énumérer les informations concernant le symbole.
- Ensuite, nous avons implémenté `SymbolTable`, la table des symboles. Nous l'avons représenté en `Stack<Map<String, SymbolInfo>>`. Nous y avons implémenté plusieurs méthodes pour ajouter des symboles à un contexte, ajouter un contexte à la table des symboles, vérifier si le symbole est dans un contexte etc.

Puis, avons créé un visiteur qui parcourt l'AST effectue ces vérifications à l'aide de la table des symboles.

1.3.2. Vérification des assignements et des paramètres

Nous avons également un visiteur qui vérifie lors d'un assignement que le nombre de variables à gauche est égal au nombre d'expressions à droites, ou au nombre de paramètres renvoyés par une fonction si il y a un appel de fonction à droite.

1.4. Génération de code 3 adresses à partir de l'AST

Un visiteur se charge de générer le code 3 adresses (`IntermediateCodeVisitor.java`).

Voici les instructions du code intermédiaire que nous avons décidé de faire :

Operation	arg1	arg2	arg3	explanation
define	new register's label			create a new nil tree
define	new register's label	value		create a new nil tree with a string value
mov	R1	R2		copy the R2's value into R1
setHead	R1	R2		set a copy of R2 as the head of R1
setTail	R1	R2		set a copy of R2 as the tail of R1
getHead	R1	R2		set a copy of R1's Head into R2
getTail	R1	R2		set a copy of R1's Tail into R2
call	function's label	R1	R[]	call the function with parameters stored in R, and store the return in R1
if	R1			check if R1 is true and if it is the case execute the next context
goto	label			jump to the label
dec	R1	value		decrement R1 by the value
closecontext				close a code context
opencontext				open a code context
label				create a label

- TODO ADD CODE TROIS ADRESSE D'UN PRGM

1.5. Génération de code à partir du code 3 adresses

Nous avons choisit C++ comme langage cible. Pour ce faire, on itère sur toutes nos instructions en code 3 adresses stockées dans une ArrayList. Comme chacune de ces instructions possèdent une méthode `toCpp()`, il suffit d'appeler cette méthode pour obtenir l'instruction 3 adresses en C++. Ensuite, il nous reste juste à concaténer toutes nos instructions générées en C++ et à les enregistrer dans un fichier.

1.6. Bibliothèque runtime de WHILE écrite dans le langage cible

Cette bibliothèque fournit la gestion des arbres ainsi que l'apport de certaines fonctions et méthodes standards.

Par exemple, on a ajouté un opérateur de cast vers des chaines de caractères ou vers des entiers pour éviter de faire trop souvent la conversion à la main.

On a également ajouté une fonction de pretty printing `pp` afin de mieux déboguer lors de l'exécution.













2. Description de la validation du compilateur

2.1. Méthodologie utilisée

Pour valider le compilateur, nous avons écrit plusieurs tests en langage while. Ils se situent dans le chemin `/test/lang/`.

2.2. Code coverage

Pour le code coverage, nous avons utilisé le plugin Maven JaCoCo (Java Code Coverage). Avec ce plugin, nous avons obtenu un code coverage global de 79%.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
<code>while_compiler</code>		26%		8%	38	56	141	208	12	29	1	5
<code>while_compiler.Visitor.Symbols</code>		80%		79%	11	41	18	119	8	29	0	3
<code>while_compiler.Visitor.Types</code>		80%		83%	5	23	7	51	3	17	0	1
<code>while_compiler.Visitor.IntermediateCode</code>		97%		90%	7	92	8	307	2	56	0	2
<code>while_compiler.Visitor.IntermediateCode.Instructions</code>		99%		83%	1	47	0	68	0	44	0	14
<code>while_compiler.Visitor</code>		100%		94%	2	30	0	69	0	12	0	1
Total	783 of 3,897	79%	63 of 198	68%	64	289	174	822	25	187	1	26

3. Bilan

3.1. Ce qui fonctionne... Ou non

D'après les tests, la transpilation fonctionne pour tout. Nous n'avons pas détecté d'erreur à partir des tests que nous avons fait.

3.2. Fonctionnalités restantes à implémenter

Lorsque l'on crée l'exécutable d'un fichier while, les arguments de ligne de commande ne sont pas supportés par l'exécutable, excepté les entiers, qui fonctionnent.

4. Description de la méthodologie de gestion de projet

4.1. Outils utilisés pour la gestion du projet

Pour la gestion du projet, nous avons utilisé Gitlab pour le versionning et un groupe Discord pour communiquer entre nous, communiquer nos problèmes et s'appeler pour travailler en dehors des séances de TP.

4.2. Etapes de développement et découpage des tâches

4.3. Rapport de travail individuel

Voici un rapport des activités de chacun sur le projet :

Etape	Membre(s)
Décrire la grammaire de While en ANTLR	Tout le monde
Création de l'AST (et simplification)	Tout le monde
Analyse sémantique	<ul style="list-style-type: none"> • Table des symboles : Célia • Visiteur de base : Titouan • Autres visiteurs : Tom, Nicolas et Célia • Validation du programme : Tom + tests écrits par tout le monde
Traduction en 3 adresses	Tout le monde, mais Tom plus que les autres
Bibliothèque run time	Tom
Backend	Tom
Script permettant d'enchaîner le compilateur While avec le compilateur du langage cible de manière à générer un exécutable	Titouan
Documentation (rapport + documentation utilisateur)	Célia et Nicolas, avec relecture des autres
Test Junit et code coverage	Nicolas

5. Post mortem : Organisation du projet

5.1. Ce qui a bien fonctionné

Travailler ensemble plutôt que séparés, en faisant du peer (ou plus) programming, a bien aidé, surtout au début du projet, pour s'entraider et mieux comprendre les attendus et les enjeux du projet. C'était plus pratique pour bien avoir une vision d'ensemble du projet.

5.2. Ce qui a moins bien fonctionné

Le fait de souvent travailler ensemble nous a sûrement fait perdre du temps et nous a rendu moins efficace sur le projet.

Aussi, puisque pendant les vacances de Noël et au retour des vacances la moitié du groupe était malade, cela a compliqué les communications et donc l'avancée sur le projet.

5.3. Avec plus de recul, que ferions-vous ?

Maintenant que nous comprenons bien ce qui nous est demandé et que nous serions capable de le refaire seuls, nous pourrions travailler en autonomie sur les étapes du projet (tout en continuant de communiquer pour s'entraider et faire part de nos avancements). Cela nous permettrait de gagner du temps et de vraiment travailler en mode « projet ».