Rapport de projet - TLC

Théo Le Goc

Robin Gaignoux

Kilian Marcelin

Rémi Cazoulat

Sommaire

Code while

Images

Autre code

1. Documentation utilisateur du compilateur et du langage

La version de While utilisé par notre compilateur est une variante du langage While créé par Olivier Ridoux. Dans cette documentation, vous allez pouvoir retrouver tout le nécessaire à l'écriture d'un programme en While, ainsi que l'utilisation de notre compilateur.

1.1. Bugs et problèmes connus

- L'utilisation d'une variable de retour avant qu'elle soit initialisée fait crash le programme

1.2. Comment écrire en While ?

Dans cette partie, vous apprendrez tout le nécessaire pour écrire en While. Pour cela, il suffit d'ouvrir n'importe quel éditeur de texte et de sauvegarder le fichier en utilisant l'extension `.while`. Le langage permet de manipuler seulement les arbres binaires, et aucun autre type de donnée. On peut cependant simuler d'autres types (voir partie Simuler un autre type).

1.3. Arbres binaires

Toutes les valeurs en while sont représentées par des arbres binaires. Chaque nœud de l'arbre peut prendre les valeurs suivantes :

- `nil` : un nœud/arbre nul.
- `Symb` : une suite de caractères, commençant par une lettre minuscule, suivie d'un nombre quelconque de lettres, de chiffres, et optionnellement de `!` ou de `?`.
- `Noeud` : un nœud, possédant un fils droit et un fils gauche.

1.4. Les expressions While

Le nommage des variables suit la règle suivante : le nom commence par une lettre majuscule, peut ensuite être suivi de n'importe quel caractère alphabétique, numérique, et peut finir par `!` ou `?`. Une assignation de variable est effectuée avec l'opérateur `:=`.

```
Variable := VALEUR_ASSIGNATION
```

Code while 1: Exemple de création de variable

Il est possible d'assigner des valeurs nulles (un nœud nul) à une variable, ou la valeur d'une autre variable.

```
Variable := nil;
Variable := MonAutreVariable
```

Code while 2: Exemple d'assignation de variable

Il est aussi possible de créer des arbres à l'aide des fonctions `cons` ou `list` :

- `(cons)` construit un arbre vide, = `nil`.
- `(cons T)` retourne l'arbre `T`.
- `(cons A B)` construit un arbre binaire ayant `A` pour fils gauche et `B` pour fils droit.
- `(cons T1 T2 ... Tn)` = `(cons T1(cons T2 ... (cons Tn-1 Tn)...)`
- `(list)` construit une liste vide, e.g `nil`
- `(list T)` = `(cons T nil)`
- `(list T1 T2 ... Tn)` = `(cons T1(cons T2 ... (cons Tn nil)...)`

```
Variable1 := nil;
Variable2 := (cons Variable1 cons nil);
Variable3 := (list Variable1 cons) // Construit le même arbre que Variable2
```

Code while 3: Exemple de création d'arbre et de liste

Pour récupérer le fils gauche d'un arbre, on écrira `(hd T)`.

Pour récupérer le fils droit, on écrira `(tl T)`.

```
Variable1 := (cons nil (cons nil));
Variable2 := (hd Variable1); // Renvoie nil
Variable3 := (tl Variable1)  // Renvoie (cons nil)
```

Code while 4: Exemple création d'arbre et de liste

Attention, ces expressions renvoient `nil` si `T` = `Symb` ou si `T` = `nil`.

Il est possible d'appeler une fonction avec la syntaxe suivante : `(f V1 V2 ... Vn)`, où `f` est une fonction définie par l'utilisateur et `V1` à `Vn` les paramètres. Une fonction peut retourner plus d'une valeur.

```
Variable1 := (maFonction Param1 Param2 (cons))
```

Code while 5: Exemple d'appel de fonctions à 3 paramètres

1.5. Simuler un autre type

1.5.1. Simuler un entier

Les entiers sont représentés par le nombre de branches à droite d'un arbre, c'est-à-dire le nombre de branches à droite visitées jusqu'à rencontrer un arbre `nil`.

1.5.2. Simuler un booléen

Un booléen `false` est représenté par un arbre vide ou par une feuille (`Symb`). Un booléen `true` est représenté par n'importe quel autre arbre.

1.5.3. Simuler une chaîne de caractères

Une chaîne de caractères est représentée par la concaténation de tous les symboles trouvés sur les feuilles, en partant des branches à gauche. Si une branche de droite se termine sur un symbole, alors le caractère est concaténé à la suite de caractères.

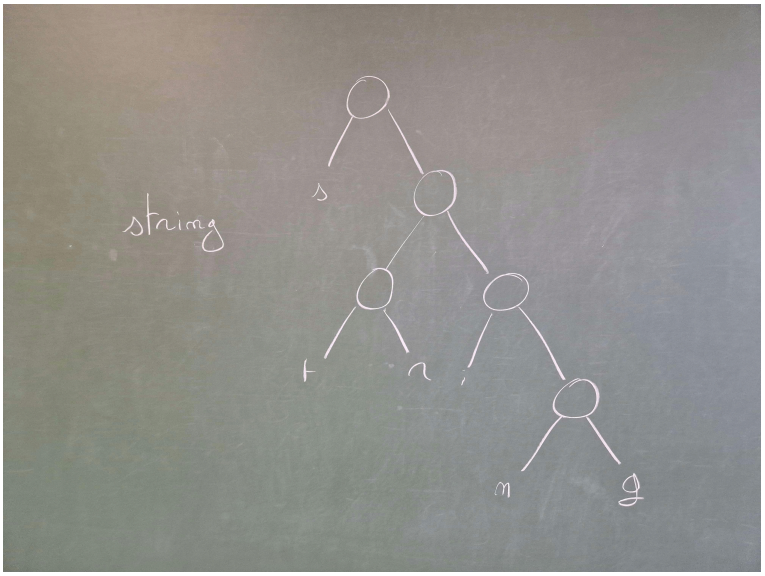

Figure 1: Représentation en arbre du mot "string"

1.6. Les commandes et structures de contrôle

Dans cette partie, nous définissons $[E]_{\text{type}}$ comme étant la simulation d'un type avec l'arbre E. L'arbre E peut être une variable ou une expression cons/list, mais pas une fonction. Les structures de contrôle suivantes sont présentes dans le langage While :

`if E then C1 fi` : si $[E]_{\text{bool}}$, exécute le bloc de code `C1`, sinon ne fait rien.

`if E then C1 else C2 fi` : si $[E]_{\text{bool}}$, exécute le bloc de code `C1`, sinon exécute le bloc de code `C2`.

`while E do C od` : si $[E]_{\text{bool}}$, exécute `C`. Répète l'opération tant que $[E]_{\text{bool}}$. Attention ! Cette structure de contrôle peut boucler indéfiniment.

`for E do C od` : répète $[E]_{\text{int}}$ fois la commande `C`. Attention ! L'exécution de `C` ne doit pas perturber le décompte du `for`. Cette structure ne peut pas boucler indéfiniment.

`foreach X in E do C od` : pour chaque élément `X` de `E`, répéter `C`. Exécute $[E]_{\text{int}}$ fois la commande `C`.

`V1, V2, …, Vn = E1, E2, …, En` : évalue toutes les expressions `E1`, `E1`, …, `En`, puis stocke les résultats dans les variables `V1`, `V2`, …, `Vn` respectivement.

`V1, V2, …, Vn = (f E1 E2 … Em)` : évalue la fonction `(f E1 E2 … Em)` et stocke les valeurs de retour dans `V1`, `V2`, …, `Vn`. Attention ! La fonction `f` doit prendre `m` paramètres et retourner `n` valeurs

Chaque instruction d'un bloc de code doit être suivie d'un `;` si ce n'est pas la dernière instruction du bloc.

1.7. Comment écrire une fonction en While ?

```
function symbol :
read I1, ..., In
%
    C
%
write O1, ... Om
```

Code while 6: Exemple de définition de fonction

Dans cette définition, `symbol` est le nom de la fonction, `I1, …, In` sont les paramètres de la fonction, et `O1, …, Om` sont les valeurs de retour de la fonction. `C` est un bloc de code contenant une succession de commandes dans le format présenté dans la section précédente.

Il faut noter que dans le langage While, le nombre de paramètres d'une fonction peut être nul (ici, il s'agit d'une fonction renvoyant une valeur constante). Cependant, le nombre de valeur de retour est toujours supérieur ou égal à 1. Il ne peut pas y avoir de fonctions sans valeur(s) de retour.

1.8. Comment utiliser notre compilateur ?

Une fois votre fichier en langage While écrit (fichier `.while`), il vous faudra utiliser notre compilateur. Celui-ci va traduire le programme en C++ puis va le compiler en exécutable. Pour accéder au code source de notre compilateur vous pouvez vous rendre sur le GitHub du projet : https://github.com/tlegoc/ESIR_TLCProjet. Le compilateur précompilé dans sa version la plus récente est disponible dans la section releases.

Note : la version 3, la dernière disponible, est probablement moins stable que la version 2. La V3 à été réécrite pour faire fonctionner le code 3 adresses via des goto, et contient aussi quelques corrections de bugs présents dans la V2. Cependant, le code C++ généré par la V3 n'a pas été autant testé que pour la V2.

Pour la suite, nous vous recommandons d'utiliser l'environnement de développement Intellij IDEA.

1.8.1. Utiliser depuis la source

1.8.1.1. Lancement via IDE

Dans le dossier compiler se trouve le code source du compilateur. Ouvrez le avec Intellij IDEA, ou n'importe quel IDE supportant le format de build d'Intellij.

1. Dans le dossier `compiler` se trouve le code source du compilateur. Ouvrez le avec Intellij ou votre IDE.
2. Vérifiez la présence de `lib_while.h` et `while.lib` pour Windows, `libwhile.a` pour Linux. S'ils ne sont pas présents, compilez la librairie runtime via `compile_windows.bat` ou `compile_linux.sh` présent dans `lib/`.
3. Créez une configuration de lancement :
   - Vérifiez que votre répertoire de travail se trouve a la racine du compiler (dossier `compiler`).
   - La classe principale est `Main`.
   - En argument doit se trouver le chemin vers votre fichier `.while`.
4. Vous pouvez maintenant lancer le programme.
5. Un fichier éxécutable sera généré à côté du fichier source. Il vous suffira de le lancer avec le nombre d'arguments nécessaire.

1.8.1.2. Compilation du `.jar`

Suivez les étapes d'utilisation jusqu'au point 2. Ensuite, compilez les artéfacts avec Intellij via **Build > Build Artifacts**. Un fichier `.jar` sera écrit dans le dossier `out/artifacts/while_jar/`.

1.8.2. Utilisation depuis le `.jar`

Nous fournissons le compilateur sous forme de fichier `.jar` dans les releases du projet. Une fois le compilateur déposé dans le dossier de votre choix, vous pouvez compiler un fichier While comme suit :

```
java -jar whilec.jar <CHEMIN_VERS_VOTRE_FICHIER_SOURCE>
```

Code 1: Utilisation du compilateur `whilec`

Remplacez `whilec.jar` par le nom du fichier .jar que vous aurez téléchargé.

2. Rapport de projet

Dans ce rapport, nous allons aborder la partie technique de la réalisation du compilateur ainsi que la partie organisationnelle du projet.

Lien du git : https://github.com/tlegoc/ESIR_TLCProjet

2.1. Description technique

2.1.1. La chaîne de compilation

Le compilateur va tout d'abord effectuer une analyse du programme While afin de vérifier s'il est syntaxiquement correct. On utilisera pour ça la librairie d'ANTLRv3.

Cette première analyse nous donne un arbre de syntaxe abstraite (AST). Celui-ci nous permet de créer la table des symboles et le code 3 adresses, permettant la validation du programme et la génération du code C++. Ce code C++ est ensuite compilé avec G++ sur Linux ou MSVC sur Windows.

2.1.2. Description de l'AST

L'AST permet d'exprimer la structure d'un programme de manière plus concise et plus significative que l'arbre de dérivation syntaxique. Tout comme la grammaire, l'AST a été réalisé

avec ANTLR. L'arbre est composé de nœuds qui peuvent avoir de 0 à n fils. Les nœuds ont des étiquettes associées qui sont des lexèmes de la grammaire.

Certains nœuds de notre AST ont des étiquettes qui sont des lexèmes imaginaires, qui nous simplifient la tâche par la suite. Voici quelques exemples des lexèmes imaginaires que nous avons créés :

- FUNC pour la déclaration d'une fonction.
- BODY pour le contenu d'une fonction.
- ASSIGN pour les assignations, ASSIGN prend lui même 2 lexèmes imaginaires ASSIGN_VARS et ASSIGN_EXPR pour pouvoir assigner plusieurs expressions à plusieurs variables en une ligne.
- FOR/WHILE/FOREACH/IF pour la déclaration de boucle ou de if.

Voici une visualisation de l'AST généré pour la déclaration d'une fonction nommée add :

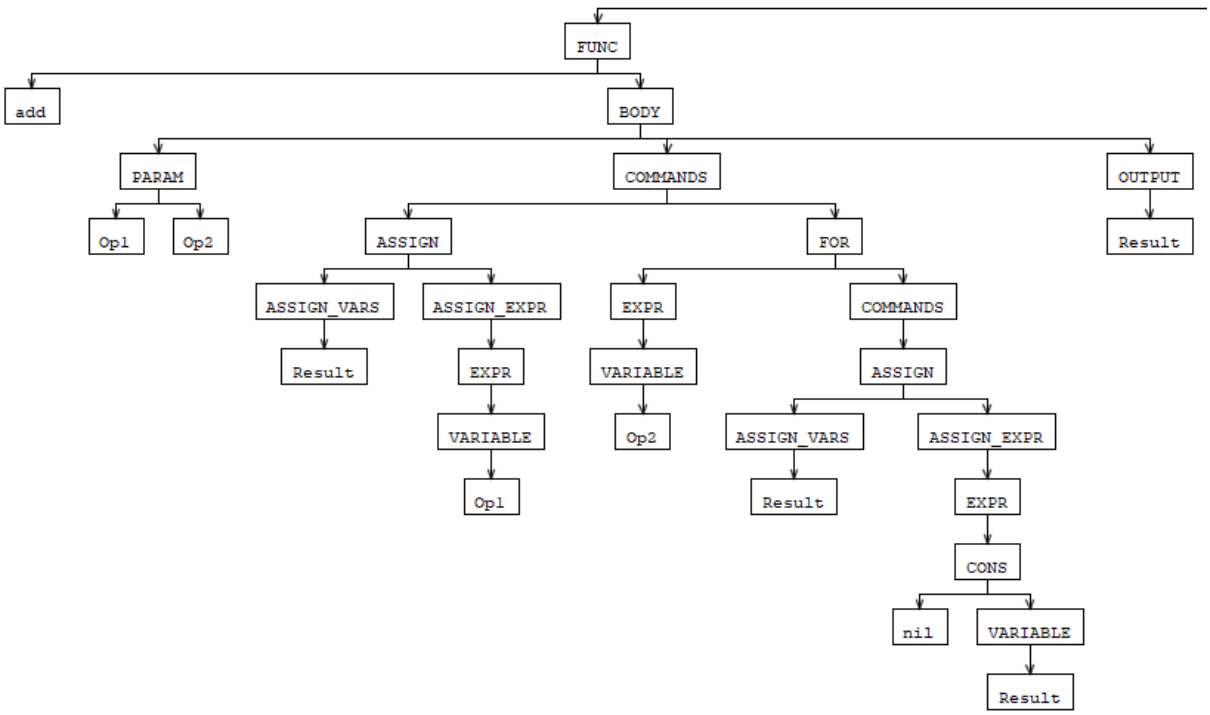

Figure 2: AST de la fonction add

Les nœuds ayant des étiquettes en majuscules sont les lexèmes imaginaires que nous avons créés. Ils nous permettent d'accéder facilement à certaines informations : sous PARAM, on accède aux paramètres de la fonctions. Sous COMMANDS, on accède aux instructions réalisées dans la fonction ou dans une boucle. Sous OUTPUT, on peut obtenir le retour de la fonction.

ANTLR nous génère 2 classes, un lexer et un parser, correspondant respectivement à l'analyseur lexical et à l'analyseur syntaxique. On peut accéder à l'AST depuis le parser avec la fonction `getTree()`. En utilisant cet arbre, nous pouvons désormais passer au code 3 adresses.

2.1.3. Code trois adresses

Le code 3 adresses est représenté par des lignes de quadruplet de la forme :

```
OPERATEUR RES ARG1 ARG2
```

- `OPERATEUR` représente l'action de la ligne. C'est un `enum`, pouvant prendre des valeurs comme `CALL`, `ASSIGN`…
- `RES` de type `Argument` contient la variable ou le registre dans lequel le résultat de la ligne est stocké. Cela peut aussi être le symbole utilisé pour appeller la fonction (dans le cas où `OPERATEUR` est défini à `CALL`).
- `ARG1` de type `Argument` contient le premier argument envoyé à l'appel, utilisé dans le cas où `OPERATEUR` est défini à `CONS`, `ASSIGN`, `HD`, `TL`…
- `ARG2` de type `Argument` contient le second argument (si l'appel a besoin d'un second argument, dans le cas de `CONS` par exemple).

Dans le cas où `RES`, `ARG1` ou `ARG2` est non défini, il est remplacé par un `EmptyArgument`, qui est affiché `EMPTY` en code 3 adresses.

Nous avons eu plusieurs versions de notre code 3 adresses. Une première étant plutôt un prototype qui ne prenait pas en compte toutes les fonctionnalités, une deuxième ne permettant pas l'utilisation de goto dans notre langage cible (il y avait des opérateurs FORBEGIN, FOREND, WHILEBEGIN, etc…) et une troisième fonctionnant avec des blocs (avec l'opérateur `BLOCK`) et des jumps (`JGREATER` et `JEQUALS`, qui comparent `ARG1` avec `ARG2` et qui ira au bloc nommé en `RES` si le résultat est égal ou plus grand. Faire un `JLESS` n'était pas pertinent, car il n'y a pas de cas où il pourrait être utile, car nous comparons en fait toujours une variable avec un arbre vide).

Dans la version 3 de notre compilateur, le programme qui suit est traduit par le code 3 adresse suivant :

Code While	Code 3 adresses

function add :
read Op1, Op2
%
    Result := Op1 ;
    for Op2 do
        Result := (cons nil Result)
    od
%
write Result
 	
function main :
read Param1, Param2
%
    Result := (add Param1 Param2);
    Out := (cons int Result)
%
write Out
	FUNCBEGIN add
	INIT reg_0
	ASSIGN reg_0 Op1
	ASSIGN Result reg_0
	INIT reg_1
	ASSIGN reg_1 Op2
	JEQUALS for_exit_0 reg_1 Nil
	BLOCK for_enter_0
	SCOPEBEGIN EMPTY
	INIT reg_2
	INIT reg_3
	ASSIGN reg_3 Nil
	INIT reg_4
	ASSIGN reg_4 Result
	INIT reg_6
	CONS reg_6 reg_3 reg_4
	ASSIGN reg_2 reg_6
	ASSIGN Result reg_2
	INIT reg_7
	TL reg_7 reg_1
	ASSIGN reg_1 reg_7
	JGREATER for_enter_0 reg_1 Nil
	SCOPEEND EMPTY
	BLOCK for_exit_0
	FUNCEND add
	FUNCBEGIN main
	INIT reg_8
	ASSIGN reg_8 Param1
	PARAMSET reg_8
	INIT reg_9
	ASSIGN reg_9 Param2
	PARAMSET reg_9
	INIT reg_10
	OUTPUTSET reg_10
	CALL add
	INIT Result
	ASSIGN Result reg_10
	INIT reg_11
	INIT reg_12
	ASSIGN reg_12 int
	INIT reg_13
	ASSIGN reg_13 Result
	INIT reg_15
	CONS reg_15 reg_12 reg_13
	ASSIGN reg_11 reg_15
	ASSIGN Out reg_11
	FUNCEND main

Code while 7: Code du test `add` et son code 3 adresses en V3

Un appel de fonction sera effectué en appellant `CALL NOM_DE_FONCTION` et en définissant les variables ou registres d'entrées et les variables ou registres de sorties avant le `CALL`, en utilisant les opérateurs `PARAMSET` et `OUTPUTSET`. Nous avons mis en place des opérateurs `SCOPEBEGIN` et `SCOPEEND` qui nous permettent de créer un contexte local à une boucle ou à un if. Nous avons fait le choix de faire en sorte que si une variable est déclarée dans une boucle ou un if, et qu'après

la fin de cette boucle ou de ce if une variable du même nom est déclarée, alors ce n'est pas la même (ce qui se fait en Java).

Pour faciliter la déclaration de variables et de registres, nous avons mis en place une liste recensant toutes les variables et tous les registres déclarés dans le contexte actuel (évoluant si on rencontre une nouvelle variable ou si on quitte un contexte. Dans ce cas toutes les variables et tous les registres déclarées dans ce contexte sont enlevées de la liste). Si la variable ou le registre n'est pas encore déclaré, on ajoute une ligne qui a comme opérateur `INIT` avec comme `RES` le nom de la variable ou registre à déclarer, ce qui nous facilite la vie dans la traduction en C++. Au début nous déclarions les variables et les registres à l'aide de la table des symboles, mais une fois que nous avions changé le code 3 adresses dans sa version avec des blocs et des jumps, notre table des symboles n'était plus faite de la bonne façon et par faute de temps nous avons préféré faire la déclaration de variables et de registres directement dans le code 3 adresses au lieu de refaire la table des symboles. Elle nous est cependant utile pour l'étape de vérification de type.

2.1.4. Optimisations

La dernière version de notre compilateur ne possède actuellement aucune optimisation faite. Le code 3 adresse est conséquent (généralement 3 à 4 fois plus long que le programme original).

Nous travaillons en parallèle sur une V4 possédant de réelles optimisations comme la suppression du code mort, la propagation de copies et la propagation de sous expressions communes. Un vrai frein à l'implémentation de l'optimisation a été la mise en place du graphe de flot de controle, servant à définir les valeurs d'entrées et de sortie de chaque blocs. Scinder le programme 3 adresses en blocs nous a aussi posé quelques problèmes, notamment à cause du fait qu'il fallait des contextes locaux aux boucles et au if. Il y a dans nos fichiers sources deux classes contenant un début de tentative d'optimisation. Elles ont été écrites quand nous avions une version du code 3 adresses plus ancienne et elles ne sont plus à jour. Notre code 3 adresses actuel est dans un format qui facilite l'optimisation, mais nous n'aurons pas le temps de l'avancer avant le rendu. C'est cependant un sujet que nous aurions aimé creuser.

2.1.5. Vérification et Validation

La validation du programme se fait pendant le parcours du code pour générer la table des symboles. A ce moment là, le programme s'assure de plusieurs choses :

- Vérification de la validité des signatures de fonction
- Vérification de la redéfinition des fonctions
- Vérification du type
- Vérification de la bonne utilisation des variables (sont-elles déjà déclarées, sont-elles accessibles…)
- Vérification des appels de fonctions inexistantes
- Vérification des paramètres d'appel d'une fonction

Ces 6 tests sont éxécutés en des points specifiques de l'AST, et permettent de s'assurer que le programme est valide. En plus de ces 6 tests, un dernier test, éxécuté après génération de la table des symboles viens vérifier qu'une fonction main existe.

Ces tests sont très simples et ne méritent pas d'explication (une simple vérification de l'arbre allant de paire avec la table des symboles en cours de construction. Cela permet aussi d'éviter un problème dans le code C++, car une fonction appelée doit toujours être déclarée AVANT utilisation). Par contre, la vérification du type est un peu particulière.

Dans le While, il n'existe pas réellement de type : tout est un arbre, et par conséquent il n'y a pas à réellement vérifier le typage tel qu'on l'entends normalement. Dans notre cas, la vérification de type consiste à s'assurer lors de l'utilisation de données qu'il y a le bon nombre d'élements dans une assignation.

```
Variable1, Variable2 := (cons nil nil), (maFonction)
```

Code while 8: Exemple de vérification de typage

Dans ce cas par exemple, il faut vérifier qu'il y a 2 élements à droite et à gauche de l'opérande `:=`. Si `maFonction` renvoie 1 élément, ce code est valide. Sinon il ne l'est pas.

```
Compiling ../tests/test5_fail.while
Error: redefinition of returnvardoesnotexist at (11:9)
Error: use of undeclared variable AhBahNon at (21:20)
Error: use of undeclared variable AhBahNonToujoursPas at (24:8)
Error: use of undeclared variable AhBahNonToujoursPasLERETOUR at (28:10)
Error: use of undeclared variable Bahahah at (32:20)
Error: call to unknown function idonotexist at (41:15)
Error: right part of assignment is of incorrect size 0, expected 1 at (41:4)
Error: call to function returnvardoesnotexist with wrong numbers of parameters, got 3, expected 2 at (48:15)
Error: call to function returnvardoesnotexist with wrong numbers of parameters, got 1, expected 2 at (49:9)
Error: could not validate program
Error: could not compile program
```

Figure 3: Exemples d'erreurs renvoyées par le compilateur

2.1.6. Génération de code

À partir de la table des symboles et du code 3 adresses, on peut générer un code C++ en lisant ligne par ligne le code 3 adresses. Il n'y a pas besoin de revenir en arrière, le code 3 adresses et la table des symboles suffisent à l'écriture du C++.

Nous ajoutons aussi des accolades en début et en fin de boucles ou de if (identifiable avec `SCOPEBEGIN` et `SCOPEEND`) permettant de créer un contexte local au bloc. Cela nous évite que la déclaration d'une variable dans une boucle ou dans un if n'empiète sur la déclaration de cette même variable plus loin dans le code, où la variable définie dans la boucle ou dans le if n'est en fait pas la même. Nous traduisons notre code en C++, donc nous avons seulement besoin de rajouter une accolade ouvrante ou fermante si nous rencontrons l'opérateur `SCOPEBEGIN` ou `SCOPEEND` pour créer un contexte local.

2.1.7. Bibliothèque runtime

Notre bibliothèque runtime utilise 4 classes permettant de définir les arbres, à savoir `INode`, interface commune aux arbres, `CTree` représentant un noeud, `CNil` représentant un nil et `CSymb` représentant un symbole. Les fonctions d'assignation sont redéfinies, évitant ainsi d'avoir des problèmes de modification de variable : si on modifie une variable, on ne modifie que celle-ci et uniquement celle-ci.

`INode` possède une fonction `clone()`, retournant un pointeur vers une copie de la variable.

Les variables sont stockées sous forme de smart pointers, évitant aussi les problèmes de mémoire. Nous avons créé des macros permettant de rendre le programme plus concis.

```
#define NODE shared_ptr<INode>

#define TREE shared_ptr<CTree>
#define MSTREE make_shared<CTree>

#define SYMB shared_ptr<CSymb>
#define MSSYMB make_shared<CSymb>

#define NIL shared_ptr<CNil>
#define MSNIL make_shared<CNil>

// Helpers
#define MSTRUE make_shared<CTree>
#define MSFALSE make_shared<CNil>

// Pas utilisé, mais sert à dupliquer une variable pour
// éviter les erreurs de modification
#define COPY(x) x->clone()
```

Code 2: Defines utilisés pour faciliter la conversion

Nous avons aussi créé plusieurs fonctions afin de simplifier la conversion du code 3 adresses vers C++.

```
// Conversion node->valeur
bool toBool(NODE node);
int toInt(NODE &node);
std::string toString(NODE &node);

// Conversion valeur->node
NODE fromInt(int n);

// Renvoie un noeud vide
void Nil(NODE &res);
NIL Nil();

void Symbol(NODE &res, string val);

// Aide pour le parcours d'arbre
bool isLeaf(NODE &res);
bool isNil(NODE &node);

// Fonctions propre au while
void Cons(NODE &res);
void Cons(NODE &res, NODE T);
void Cons(NODE &res, NODE A, NODE B);
void hd(NODE &res, NODE &T);
void tl(NODE &res, NODE &T);

// Comparaison de noeud, évite de devoir
// redéfinir operator==
NODE equals(NODE &A, NODE &B);

// Pretty printer
void pp(NODE &node);
void ppln(NODE &node);

// Parsing d'un string représentant un arbre
NODE parseString(const string &s, bool &valid);

// Parsing des paramètres d'argument pour les passer au
// main
bool parseParameters(vector<NODE > &nodes, int argc, char **argv);
```

Code 3: Fonctions de la librairie

2.2. Description de la validation du compilateur

Nous avons testé notre compilateur à différents niveaux avec la même méthodologie : nous effectuons plusieurs tests en écrivant des petits programmes While, certains censés être corrects et d'autres non. Plus précisément, nous avons testé au niveau de :

- la grammaire et l'AST, en écrivant des tests syntaxiquement incorrects.
- la création de la table des symboles, que nous avons vérifié à la main sur nos 4 programmes de test valides.
- la création du code 3 adresses, vérifié à la main de même.
- la génération du code C++.

À chaque étape, ces tests nous ont permis de comprendre les dysfonctionnements et de les corriger. Cela était le cas principalement pour l'AST et le code 3 adresses que nous avons dû refaire à de nombreuses reprises, tout comme le reste du programme. Nous avons fait des programmes tests qui couvraient toutes les cas de figures possibles auxquels nous pouvions penser. Il est assez improbable que ces tests soient exhaustifs, car il est complexe de penser à tout ce qu'il est possible de faire avec le langage While.

Au moment où nous rendons le projet, nous avons réussi à couvrir tous les points que nous avons identifié (sauf ceux décrits dans la section problèmes et bugs connus), et les fonctionnalités restantes à implémenter sont toutes celles touchant à l'optimisation.

2.3. Description de la méthodologie de gestion de projet

2.3.1. Outils

Pour mener à bien ce projet, l'usage de plusieurs outils a été nécessaire.

Premièrement et principalement GitHub, qui nous a permis la gestion des versions, le travail simultané sur différentes branches ainsi que la gestion des conflits lors des fusions de versions.

Pour écrire la grammaire, nous avons utilisé ANTLRworks. Cela nous a permis par la suite de visualiser directement dans le logiciel l'arbre de dérivation syntaxique ainsi que l'AST.

L'outil de collaboration proposé par Intellij IDEA nous a permis de travailler en binôme simultanément sur le même projet à la manière de Google Docs. Cela nous a rendu plus efficaces, car nous pouvions nous partager le travail sur une même classe Java, en codant sur le même projet simultanément. Intellij permet aussi de gérer le projet java de façon simple.

Afin de nous coordonner en dehors des cours, nous avons utilisé Discord, pour communiquer nos avancées, ou pouvoir s'appeler pour reformer nos binômes de travail.

Pour rédiger ce rapport, nous avons utilisé Typst, permettant un travail en collaboration, ainsi qu'une production finale de qualité.

2.3.2. Etapes de développement et découpage des tâches

Le développement de ce projet s'est fait en 5 grandes étapes :

- La grammaire

Il s'agit de la toute première étape de ce projet, à laquelle nous avons tous participé. Celle-ci a consisté à retranscrire la grammaire, c'est-à-dire l'ensemble des règles qui définissent la structure syntaxique du While, dans ANTLR. Elle spécifie comment les différents éléments d'un langage peuvent être combinés pour former des constructions valides.

- L'AST

Nous avons tous participé à cette partie.

- Table des symboles et code 3 adresses

La table des symboles a été créée par Rémi et Kilian puis remodifiée par Théo et Robin. Le code 3 adresses a été réalisé par Robin et Théo, puis grandement remodifié par Théo et Rémi.

- Traduction en C++

Cette partie a été réalisée par Kilian et Rémi. Corrigée et complétée par Théo et Rémi

- Librairie runtime

Le code de la librairie runtime à été initialement écrit par Robin, puis mis à jour par Théo.

- Corrections finales et rédaction

Cette dernière étape nous permettait de s'assurer que toutes les contraintes du projet étaient implémentées et que nous n'avions pas d'erreur. Il nous a fallu modifier de nombreuses parties de notre code à cause d'oublis. Théo et Rémi se sont chargés de cela. Pendant ce temps Kilian et Robin se sont occupés de la rédaction du rapport.

2.3.3. Rapports individuels

Rémi Cazoulat - Annexe n°1
Théo Le Goc - Annexe n°2
Robin Gaignoux - Annexe n°3
Kilian Marcelin - Annexe n°4

2.4. Post mortem

Nous sommes satisfaits du fonctionnement de notre compilateur, même si nous aurions aimé avoir plus de temps pour faire quelques optimisations, comme celles présentées en cours. Notre travail par binôme nous a semblé efficace. Certaines étapes ont demandé plus de travail, suivant la compréhension de chacun du projet. Par exemple, nous avons perdu du temps à revenir sur la création de l'AST, du code 3 adresses et de la table des symboles, car les premières versions n'étaient pas utilisables. Cela était principalement dû au fait que nous ne savions pas comment nous allions utiliser l'arbre par la suite.

Avec plus de recul, nous aurions également dû faire plus attention à toutes les spécifications. Le langage While ne nous était pas familier, et nous sommes donc parti un peu trop vite et nous avons oublié certaines des contraintes imposées. Nous avons donc dû effectuer beaucoup de corrections au fur et à mesure du projet.

3. Annexes

3.1. Annexe 1 - Rapport individuel de Rémi Cazoulat

Au début de ce projet (pendant les heures de tp) j'ai travaillé en binome avec Kilian. Nous avons mis en place une première version de la table des symboles et une première version de la traduction du code 3 adresses en code C++. A la fin des séances de TP nous avions un prototype fonctionnel mais nous nous sommes rendu compte qu'il y avait pleins de cas auxquels nous n'avions pas pensé, comme l'appel d'une fonction en paramètre d'un for, d'un if ou d'une construction d'arbre. J'ai donc beaucoup modifié le visiteur de l'AST et la façon dont on construisait le code 3 adresses pour prendre en compte tous les cas de figures.

J'ai ensuite dû remodifier la traduction en code C++ après avoir changé quelque peu le code 3 adresses. Je me suis ensuite occupé de faire un code 3 adresses plus universel en utilisant des blocs et des jumps afin de de rendre ce code plus semblable à un code assembleur. J'ai du encore par la suite modifier la traduction en rajoutant des déclarations de blocs dans notre code C++ et des goto pour sauter d'un bloc à l'autre.

Je me suis ensuite beaucoup penché sur la question de l'optimisation, mais j'ai mis un certain moment à comprendre comment bien l'implémenter, et je n'ai malheureusement pas eu le temps de finaliser cette étape. Cela m'a tout de même permis de remarquer et de corriger certaines choses qui n'allaient pas dans mon code 3 adresses (comme les contextes des boucles ou des if) et j'aurai beaucoup approfondi ce que j'avais compris en CM. Je n'aurai pas pensé prendre

autant goût à ce projet, et j'espère avoir l'occasion de créer un langage dans le futur car c'est un exercice qui m'a beaucoup plu et motivé.

3.2. Annexe 2 - Rapport individuel de Théo Le Goc

Le but de ce projet était au final d'avoir un compilateur fonctionnel. À ce niveau l'objectif est atteint.

Cependant, nous avons pris beaucoup de raccourcis, afin de gagner le temps perdu par le manque de compréhension du sujet et la non connaissance de toutes les parties nécessaire au fonctionnement du compilateur :

- La génération du code 3 adresses à été faite 2 fois, et modifiée 3 fois. Nous avions mal géré le parcours de l'arbre, et le code n'était pas satisfaisant : les gotos, implémentés par Rémi vers la fin du projet, n'ont par exemple pas été assez testés. Une 4ème modification est en cours pour inclure l'optimisation.
- La génération de la table des symboles à été faite 2 fois. La première implémentation, en spaghetti stack, n'étais pas satisfaisante ni utilisable pour la génération du code C++.
- La validation à d'abord été faite dans une classe à part, avant d'être intégrée à la génération de la table des symboles, comme spécifié dans le sujet.
- La génération du code C++ à été refaite 1 fois, le premier jet posant plein de problèmes de compilation.
- La compilation du code C++ à été refaite de zéro suite à des problèmes de compatibilité windows/linux.

Même après ces refontes, le code n'est pas parfait. On ne répond pas correctement à tout les points du sujet par exemple. De plus, l'architecture du projet est brouillon voire chaotique, le code est réellement lisible que par Rémi et moi étant donné le nombre de modifications sans commentaires que nous avons effectuées. Certaines fonctions sont peu intuitives.

Avec le recul et au vu de toutes les connaissances acquises, il serait judicieux de recommencer le projet de zéro et de mieux prévoir toute l'architecture et le fonctionnement du code en amont, ce que nous n'avons pas assez bien fait.

On s'aperçoit aussi que le langage, bien que très peu pratique à utiliser, possède des propriétés cachées qui permettent de parser bien plus facilement le langage et d'avoir un AST plus simple.

Concernant la quantité de travaux réalisé, je me suis surtout occupé de faire des passes de réécriture/fix du code de Rémi, Robin et Kilian, surtout sur la table des symboles et librairie runtime. J'ai aussi complètement fait la partie compilation du code C++. Le projet était conséquent et, sur la fin, le nombre de réécritures et modifications à énormément fatigué toute l'équipe, d'ou l'investissement moindre de certains membres. Tout de même, l'entièreté du groupe était présente tout le long du projet.

3.3. Annexe 3 - Rapport individuel de Robin Gaignoux

Dans le cadre de ce projet, ma contribution s'est concentrée sur la création de la grammaire dans ANTLR, la première version de la génération du code 3 adresses, la refonte de la table des symboles et le début de la validation et de la librairie runtime. Ces étapes ont été réalisées en groupe tous les quatre, en binôme avec Théo, ou seul. Étant donné que je ne possède pas de PC portable personnel, j'ai dû réaliser ces tâches sur les PC de l'université. Cela a entraîné certains obstacles pratiques, comme l'absence de certains outils spécifiques (GitHub Desktop sur les PC Linux) ou des problèmes de stockage mémoire (pas assez de place sur les PC de l'université). L'utilisation de l'outil de développement collaboratif d'IntelliJ IDEA s'est donc

avérée très utile, car cela me permettait de coder “via” le PC de Théo, et cela m’a donc permis de maintenir une contribution active au projet pendant les séances de TP.

Tout au long du projet, j’ai donc pu appliquer et comprendre la théorie vue en cours. Cela n’était pas toujours simple, et le manque de compréhension m’a personnellement (et nous a en groupe) parfois conduit à des erreurs de jugement sur le fonctionnement global et individuel des différentes parties du compilateur, ce qui nous a pénalisé en temps et en énergie. Au-delà des aspects techniques du compilateur, cette expérience m’a également permis de renforcer ma capacité à résoudre des problèmes, à travailler en groupe et à s’organiser. En somme, ce projet fut enrichissant et je suis satisfait de celui-ci et du résultat obtenu.

3.4. Annexe 4 - Rapport individuel de Kilian Marcelin

Ce projet nous a permis de mettre en lien toutes les étapes de compilation d’un langage vues en cours. Désormais nous comprenons la complexité des compilateurs que nous utilisons au quotidien. Je suis globalement satisfait de notre compilateur même s’il mériterait quelques optimisations simples que nous n’avons pas eu le temps d’implémenter.

Nous avons travaillé en binôme, j’étais avec Rémi et nous avons principalement fait l’écriture de la grammaire, la création de l’AST, la première version de la table des symboles et la génération du code c++. Ces étapes n’ont pas toujours avancées comme nous l’auriont voulu. Notre structure de table des symbôles s’est retrouvée trop complexe à utiliser par la suite et a donc dû être refaite. La génération du code c++ a elle été beaucoup plus efficace.

Nous avons fait une première version du compilateur qui marchait mais qui n’avait pas toutes les fonctionnalités requises. Le langage while est assez différent des langages avec lesquels nous avons l’habitude de travailler, nous avons donc oublié des fonctionnalités avec lesquelles nous ne sommes pas familiés. Cela a demandé beaucoup de travail à Rémi et Théo durant les derniers jours pour les implémenter et je tenais à les remercier pour cela.