

# Compilateur pour Petit Julia

Hector BUFFIÈRE et Thomas LAURE

1er semestre 2020-2021

## 1 Introduction

Projet réalisé dans le cadre du cours Langages de Programmation et Compilation du 1er semestre de L3 à l'Ecole Normale Supérieure.

Sujet : <https://www.lri.fr/~filliatr/ens/compil/projet/sujet-v2.pdf>

Ce projet a été réalisé dans le langage Ocaml.

## 2 Analyse lexicale et syntaxique

### 2.1 Analyse lexicale

L'analyse lexicale a été réalisée avec Ocamllex.

Le point-virgule automatique en fin de ligne est traité à l'aide d'une référence booléenne globale. Quand on rencontre un lexème, on met celle-ci à jour selon si ce lexème serait suivi d'un point-virgule automatique si l'on rencontrait un retour chariot ensuite. Quand on rencontre un retour chariot, on renvoie le lexème ";" si la valeur de la référence était Vrai.

Petit Julia accepte que des entiers soient suivis directement d'une parenthèse ou d'une variable, on renvoie donc des lexèmes dédiés dans ce cas, et de même pour les variables suivies ou suivant directement une parenthèse.

Enfin, Ocaml n'acceptant pas les entiers au delà de  $2^{62}$ , on utilise le module *Int64* d'Ocaml pour représenter les entiers 64 bits de Petit Julia.

Nous avons également ajouté les lexèmes / et \ qui n'étaient pas demandés par le sujet.

### 2.2 Arbre de syntaxe abstraite

Les expressions sont représentées par un type construit contenant leur localisation dans le fichier, et la nature du nœud de l'arbre qu'elles représentent.

Les déclarations de fonctions et de structures sont des types construits constitués de leurs noms, de leurs arguments, de leurs localisations dans le fichier, de leurs instructions pour les premières et de leur caractère mutable ou non pour les secondes.

Les paramètres sont aussi un type construits donnant comme information leur nom, leur type et leur localisation dans le fichier.

## 2.3 Analyse syntaxique

L'analyse syntaxique a été réalisée avec Menhir.

L'analyseur tient compte du sucre syntaxique proposé. La fonction `div` est remplacée par l'opérateur binaire `/`, la fonction `println` par la fonction `print` à laquelle est ajoutée l'expression `\n` après les arguments. Les types omis dans les déclarations de paramètres sont bien évalués à *Any*.

Les éléments des blocs (listes d'expressions) et des listes de paramètres peuvent être séparés par n'importe quel nombre de points-virgules, et la grammaire en tient compte.

Les précédences et associativités sont celles indiquées dans le sujet. Un conflit lié au `return` a été résolu en déclarant les lexèmes centraux des expressions comme de précedence plus grande que celle du `return`. Les conflits liés à la succession d'une expression et d'un bloc sans séparateur dans les `if`, `while` et `for` ont été résolus en séparant la production (mot-clé expression) de la production ((mot-clé expression) instructions fin) et en associant une précedence plus grande à la première.

## 3 Typage

Le typage se fait en deux étapes, comme suggéré dans le sujet.

La première produit un environnement primaire dans lequel on vérifie que les déclarations de types sont valides (types bien formés, pas de doublon dans les arguments d'une fonction, dans les déclarations de fonctions ou de structures). On initialise également les variables globales à ce moment. On garde en mémoire plusieurs dictionnaires contenant les noms et types déclarés des variables globales, tous les types de fonctions associées à un même identifiant, les structures et leurs champs, et réciproquement la structure associée à un champ pour pouvoir typer les accès à ce champ.

Dans un deuxième temps, on construit les environnements locaux associés à chaque fonction puis seulement après on type le corps desdites fonctions (Julia étant fait de telle sorte qu'une déclaration en fin de fonction peut influencer sur un comportement en début de fonction, ce que nous avons mis un certain temps à bien comprendre). Enfin, on peut vérifier que toutes les expressions du programme sont bien typées dans l'environnement global et l'environnement local dans lequel elles sont plongées le cas échéant.

Le typage reconstruit un arbre de syntaxe abstraite sur le même modèle que le précédent en remplaçant la localisation d'une expression dans le fichier par son type. Les fonctions et les structures sont retenues dans un dictionnaire, on détruit donc les instructions des fonctions dans la nouvelle liste de déclarations.

Les blocs possèdent également l'information de l'environnement local spécifique dans lequel ils sont éventuellement plongés (s'il s'agit d'instructions d'un `for`, d'un `while` ou du corps d'une fonction). A un appel de fonction, on associe la liste des identifiants des fonctions de même nom compatibles avec le nombre d'arguments et leur type. Si aucune fonction n'est compatible, on échoue, et si plusieurs fonctions sont compatibles et que l'on est sûrs de ne pas avoir d'informations plus précises à l'exécution, c'est à dire que tous les types des arguments sont connus précisément et que parmi les fonctions compatibles aucune n'est

plus spécifique que les autres, on échoue aussi.