LOG3210 – Éléments de langages et compilateurs

Hiver 2021

Examen Intra

Groupe 01
1947497 – Yuhan Li

Soumis à : Esther Guerrier

15 mars 2021

Grammaire

D'abord, pour l'importation j'ai créé une fonction (ImportStmt) que j'ai appelé une seule fois, soit au début de la fonction Program puisqu'on effectue les importations uniquement au début du code. ImportStmt() comprend deux jetons (début et fin d'importation) et une fonction Identifier() qui est une des fonctions de base du langage utilisée pour les expressions quelconques (ex. : nom de fonction, de variable, d'import).

Outre Identifier(), ma grammaire comporte 4 autres expressions qui sont à la base de la syntaxe. Ces fonctions sont dédiées aux String, Bool, Real et Int. Chacune de ces fonctions possèdent un jeton que je sauvegarde au niveau du nœud. Pour le String, j'ai ajouté le jeton <STR> qui suit un format inspiré de : https://stackoverflow.com/questions/46990699/define-token-to-match-any-string. Le contenu entre guillemet permet de reconnaître n'importe quelle chaîne de String, vide ou pas. J'ai regroupé les fonctions dans PrimitiveExpr() pour simplifier l'appel à ces valeurs de base. De plus, sachant qu'on peut assigner des valeurs directement par l'appel de fonction, j'ai ajouté CallFunction() dans les options de PrimitiveExpr().

CallFunction() est aussi appelé dans Stmt() puisqu'un appel de fonction peut être une ligne de code à lui seul. En effet, j'ai assumé que chaque fonction appelée dans Stmt() était des lignes ou des blocs d'expressions qui ont une exécution à part entière. Donc, en plus de CallFunction(), j'ai inclus les fonctions liées au cœur, à la déclaration de variable, à l'implémentation de fonction, ainsi qu'aux If/While.

Pour la déclaration, j'ai divisé ça en deux fonctions (*DeclareStmt* et *AssignStmt*), simplement pour avoir une meilleure visualisation de ce qui se produit. Pour DeclareStmt(), j'ai ajouté un jeton ("compi") pour l'annonce d'une déclaration. J'ai également ajouté d'autres terminaux au niveau du jeton TYPE afin de représenter chaque type nécessaire à EstheRust. Pour AssignStmt(), j'ai implémenté Expr() qui est appelée à la droite du terminal "=". Cette fonction représente des valeurs concrètes qui peuvent provenir d'une opération logique ou arithmétique. C'est pourquoi, Expr() est implémentée de manière récursive afin d'offrir une possibilité d'accès à chaque type d'opération et aux valeurs de base de PrimitiveExpr().

Concernant les diverses opérations, je les ai regroupés en 4 fonctions, où la récursivité suit un ordre d'importance inspiré du tp1 et tp2. La moins prioritaire est CompExpr() (comparaison : and/or ou </>/==/etc), donc elle est appelée directement à partir de Expr(). Pour établir un comportement récursif, on retrouve, de part et d'autre de l'opérateur de comparaison, un appel à l'opération plus prioritaire ainsi qu'à CompExpr(). Ce parsage hiérarchique se poursuit donc jusqu'à l'opération la plus importante, soit les parenthèses (*ParExpr*). C'est dans ParExpr() aussi qu'on accède les valeurs de base (*PrimitiveExpr*).

Pour l'implémentation de fonction, j'ai créé deux fonctions pour mieux visualiser les évènements. La première fonction concerne la signature complète d'une fonction, qui débute en appelant la deuxième fonction. Cette seconde fonction concerne la signature de la première ligne des déclarations. J'ai donc ajouté deux jetons, soit pour l'annonce d'une déclaration et pour la flèche pointant vers le type de retour.

Pour le If et le While, j'ai implémenté une fonction pour chacun. Chacune des fonctions prend en compte le If/While avec leur condition, ainsi que leur "body". La fonction IfStmt() contient des crochets en plus, car ce dernier contient le "Else" qui apparaît seulement une fois lorsque nécessaire.

Enfin, pour l'opérateur Cœur, ma syntaxe consiste uniquement d'une fonction comprenant le jeton <HEART> que j'ai ajouté au lexique.

Visiteur

D'abord, pour le visiteur de Program, Block, Stmt et PrimitiveExpr, le principe est le même; c'est-à-dire j'accepte tous leurs enfants afin de visiter tous les nœuds du code selon le parcours d'appel. Puis, pour les valeurs de bases possibles dans PrimitiveExpr, j'affiche leur valeur respective enregistré dans le nœud.

La section d'après, soit 4.1, concerne les importations. Le visiteur ne fait que parcourir tous les enfants (Identifier) du nœud en affichant juste avant l'annonce d'un import.

En 4.2 je traite les déclarations avec un visiteur pour la déclaration globale et un autre pour l'assignation d'une expression. Donc, le visiteur AssignStmt ne fait que visiter son enfant unique, qui est un Expr(), pour imprimer la valeur nécessaire. Le visiteur de DeclareStmt, lui, visite le premier nœud qui correspond au nom de la variable. Puis, il va afficher l'opérateur nécessaire pour ensuite visiter le nœud de l'assignation.

Bien sûr, au début de la déclaration, je parcours le niveau d'indentation auquel le code est rendu en affichant les "/t". J'effectue cela aussi pour les visiteur du If/While, du cœur, ainsi que pour les blocks de contenu vide ("pass"). L'incrémentation est gérée au début des blocks d'une fonction et d'un If/While, et la décrémentation aussi, mais à la fin de ces blocks.

Pour les opérations arithmétiques (4.3), le principe est similaire pour chacun des quatre nœuds. Je visite les nœuds nécessaires selon la priorité pour afficher les valeurs ou les opérations attendues. Toutefois, l'affichage de l'opérande est légèrement différente pour CompExpr() et UnaExpr(), car il faut convertir les opérandes selon la syntaxe Python (ex. : "&&" est "and" en Python). J'utilise donc getOp(string) pour ça.

Pour l'implémention des fonctions en 4.4, je visite d'abord le nœud FunctionParam() afin d'afficher "def", le nom de la fonction, et le contenu pour les paramètres. Puis, je passe à la visite du Block() afin d'afficher le "body" de la fonction. Je vérifie avant, comme expliqué plus haut, si ce block est vide ou non pour savoir si je dois afficher "pass". Enfin, pour savoir s'il faut retourner une variable, j'observe si mon nœud contient un troisième enfant qui correspondrait à la valeur de retour. De plus, en 4.4 se trouve également mon visiteur pour l'appel de fonction (*CallFunction*). Pour lui, le comportement du visiteur est semblable, d'où l'utilisation de la fonction *printParam* pour les deux. Toutefois, avant d'indenter, je dois m'assurer que le nœud est bel bien un Stmt() et non une fonction appelée lors d'une déclaration et assignation de variable.

En 4.5 et 4.6, j'implémente les visiteurs du If et du While qui sont assez semblables. Le principe est d'imprimer d'abord "if" ou "while", puis visiter le premier enfant afin d'afficher la condition du If ou du While. Ensuite, j'appelle une fonction commune (printBlock) que j'ai créé pour afficher le "body" en visitant le nœud Block(). Comme pour le visiteur 4.4 de fonction, je vérifie d'abord dans PrintBlock() si ce block est vide en vu d'afficher "pass". Cependant, suivant l'appel au PrintBlock(), le visiteur du If doit aussi prendre en compte l'affichage du "else" et de son block. Cela est fait en vérifiant si IfStmt() contient un troisième enfant.

Finalement la dernière section de mon fichier visiteur concerne l'opérateur Cœur. Ce visiteur affiche simplement la phrase équivalente au jeton cœur du lexique de EstheRust.