Rapport d'activité 1 (Semaine 46 15/11/2021)

1. Gestion de projet

Ce projet est réalisé par les membres présentés dans le tableau ci-dessous. Une méthode de travail classique avec des réunions hebdomadaires (chaque mercredi de 14h à 16h) a été adoptée, il peut y avoir des stand-up-meeting pour régler les soucis ou échanger de nos avancements. Des compte-rendus de réunions sont réalisés régulièrement sur Overleaf et sont déposés dans le dépôt `gitlab`.

Membres de l'équipe	Rôles ou charges	Temps de travail
Chaima TOUNSI-OMEZZINE	Chef de projet	12h
Céline ZHANG	Secrétaire	14h
Gisel RODRIGUEZ-BAIDE	Reviewer	12h

Chaima TOUNSI-OMEZZINE s'occupera de vérifier les tâches, les objectifs, de planifier les réunions, et produira les tests exhausives.

Céline ZHANG se chargera d'écrire les compte-rendus de réunions d'équipe, de gérer les ressources et de concevoir la grammaire.

Gisel RODRIGUEZ-BAIDE se chargera de revoir nos lignes de code (*code review*), de vérifier la cohérence et vérifiera les tests.

Organisation

Le tableau de bord Trello [1] a été choisi pour découper et gérer nos tâches issues du cahier des charges nécessaires à la réalisation du livrable final. Ceci permet également de surveiller l'avancement de chaque membre sur ses tâches.

2. Travail effectué

2.1. Grammaire

Une première version de la grammaire CIR-C a été réalisée, elle se trouve dans le fichier `grammaire.g4` dans notre dépôt gitlab. Elle est inspirée de la grammaire proposée dans le sujet, mais plusieurs modifications ont été apportées pour mettre des priorités aux opération, pour supprimer des règles inutiles, et enlever les récursivités gauches. Un aperçu de la grammaire est proposé dans l'image 1.

2.2. AST

L'analyseur lexical et syntaxique ont été produits, l'objectif actuel est de créer nos visiteurs et de produire l'AST. Donc, l'écriture des visiteurs sont en cours, les tests n'ont pas encore été réalisées.

1. `https://trello.com/b/E3EJnI0p/projetpcl-1`

2.3. Tests

Plusieurs fichiers de tests exhausifs ont été rédigés afin de tester la grammaire actuelle, ils se trouvent pour l'instant dans le répertoire `examples`. Les déclarations de variables, de fonctions, de structures, les blocs d'instructions, la priorité des opérations et la gestion des commentaires ont été traités dans ces tests. Dans l'ensemble, la grammaire actuelle respecte les priorités des opérations, elle produit correctement les affectations, les déclarations de types (`int`, `struct`), de variables et de fonctions, elle reconnaît les boucles while, les `if`, les blocs, les instructions, les `return`, les `sizeof`, l'utilisation des pointeurs et les commentaires.
Un exemple de fichier test d'un cas simple de code se trouve ci-dessous. Un arbre syntaxique est produit ci-après, figure 3.

```
/* Une d claration de struct qui utilise autres struct comme attributs */
struct testStruct {
    int a ;
    struct struct1 *one, *two ;
};

/*  --> d claration d'une fonction avec :
        param tres : un pointeur sur une structure de type test_struct
        retourne : un pointeur sur une structure de type test_struct
*/
struct testStruct * structF(struct testStruct * ab) {
    while (a < b > c*5) {   // test d'une boucle while
        ab->a = a - b*10 +'c' ;
        if (a == b != c)  // test de if then else
          return NULL;
        else
          return ab;
    }
    return ab ;
}
```

Listing 1 – Test

3. Problèmes rencontrés et solutions

3.1. Tokens

Grâce aux tests, des problèmes dans les règles de la grammaire ont été relevés. En effet, au début, des terminaux ont été écrit pour produire les mots clés (par exemple les *tokens* `'int'`, `'struct'`, `';'`, `','` etc.) plusieurs erreurs ont été générées lors de la construction de l'arbre syntaxique (l'arbre était plat) donc l'idée de remplacer ces mots par des terminaux a été abandonnée, en conséquence, la majorité des soucis a été résolue.

3.2. Regex

Il y a une ambiguïté sur le sujet concernant les caractères ASCII à prendre en compte (' " \), notamment pour le droit d'affecter à un entier un `char`, ex : `int e = '\'`. C'est une question qui devra être posée. Par ailleurs, le code de la grammaire peut-être simplifié par la simplification de la regex. En effet, l'écriture de `(param,)*` peut-être simplifié dans nos règles, notamment `params` car Antlr 4 serait capable de comprendre `','` comme une séparation.

FIGURE 1 – Aperçu de la grammaire actuelle partie 1

3.3. Opérations

Par ailleurs, concernant les opérations, la priorité a été traitée, cependant, les multiples enchaînements des opérations de même priorité peut poser problèmes comme `a = b = c;` `a - b + c;` `a == b != c;` `a < b > c;` etc. Le problème d'associativité est soulevé, notre grammaire ne traite pas forcément ce problème. Les opérations sont générées sur la même couche de l'arbre (le traitement par défaut est de gauche à droite ? c'est une question qui est à poser). Puis, il y a un soucis de cohérence sur l'enchaînement de `==, !=` ou `<, >`, est-ce qu'il faut considérer une erreur sémantique ou syntaxique (dans ce cas là, intégrer le traitement à la grammaire). Il faudra consulter l'avis d'un encadrant pour choisir les solutions les plus adaptées. Un arbre syntaxique, figure 3, est généré ci-dessous à partir du code `Test` proposé ci-dessus.

4. Travail à venir

Les prochaines étapes pour le mois à venir sont la modification de la grammaire si besoin, selon les retours des encadrants, la continuation de la génération de l'arbre abstrait, c'est-à-dire l'écriture des visiteurs en Java, et la réflexion sur l'analyse sémantique ainsi que la construction de la table des symboles.

TO-DO LIST

- Se renseigner sur les questionnements
- Modifier la grammaire si nécessaire
- Réaliser l'arbre abstrait AST
- Débuter l'analyse sémantique
- Commencer la table des symboles TDS

```
78  somme :
79      produit (('+'|'-') produit)*;
80
81  produit :
82      oppose (('*'|'/') oppose)*;
83
84  oppose :
85      ('!'|'-')?value;
86
87  fleche :
88      IDF '->' IDF;
89
90  value :
91      INTEGER                                          #Integer
92    | IDF                                              #Identifier
93    | fleche                                           #Arrow
94    | IDF '('((expr ',')* expr)?')'                    #Function
95    | 'sizeof' '('('struct' IDF')'                     #Sizeof
96    | '('expr')'                                       #Parenthesis
97    ;
98
99
100 //lexer ruler
101 INTEGER :
102     '0'
103   | '1'..'9' ('0'..'9')*
104   | '\'' [\u0020-\u007E] '\''
105   ;
106
107 IDF :
108     ('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
109
110
111 // commenters
112 COMMENTERS :
113     ('/*' .*? '*/'|'//' ~[\u000A\u000D]*) -> skip;
114
115
116 // skip
117 WS :
118     ('\t' | ' ' | '\r' | '\n')+ -> skip;
```

FIGURE 2 – Aperçu de la grammaire actuelle partie 2

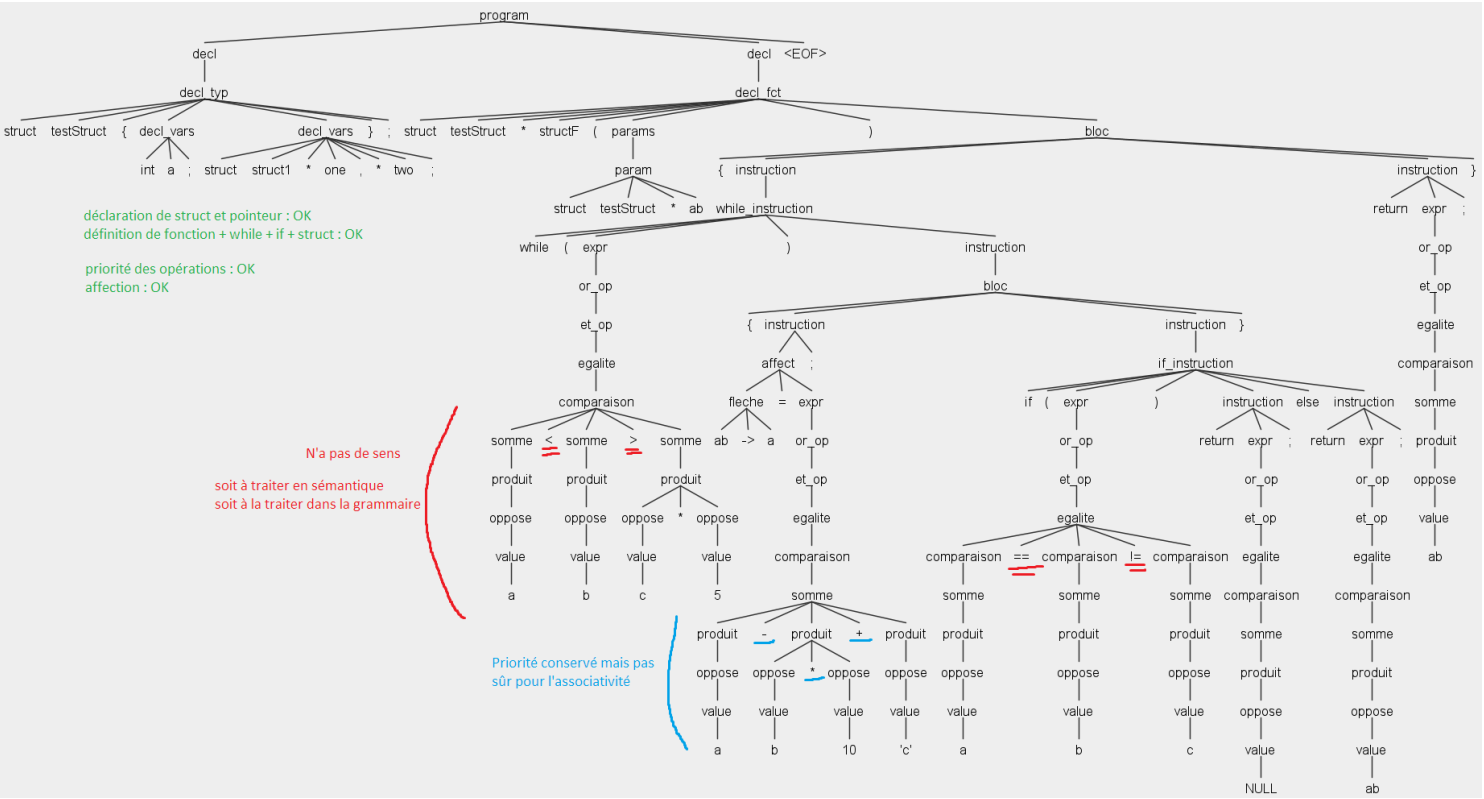

FIGURE 3 – Arbre syntaxique du code Test