Rapport d'activité 3 (Semaine 2 10/01/2022)

1. Gestion de projet

Ce projet est réalisé par les membres présentés dans le tableau ci-dessous. La méthode de travail [1] décrite lors des précédents rapport d'activité a été conservée. Le suivi des tâches se fait avec le Trello [2] et les compte-rendus de réunions sont déposés sur Overleaf et sur le dépôt gitlab. La répartition des tâches au sein du groupe durant cette période est détaillée dans les paragraphes suivants.

Membres de l'équipe	Rôles ou charges	Temps de travail
Chaima TOUNSI-OMEZZINE	Chef de projet	19h
Céline ZHANG	Secrétaire	22h
Gisel RODRIGUEZ-BAIDE	Reviewer	19h

Chaima TOUNSI-OMEZZINE a implémenté les classes du package `Symboles` et `SymbolTable` pour créer la structure de la table de symboles, elle a produit des tests exhaustifs pour la vérification de la TDS et les contrôles sémantiques.

Céline ZHANG a finalisé l'implémentation de l'AST. De même, elle a travaillé sur `TdsVisitor` pour la construction de la TDS, l'implémentation des contrôles sémantiques et la gestion des erreurs.

Gisel RODRIGUEZ BAIDE a travaillé sur `TdsVisitor` pour la construction de la TDS et l'implémentation des contrôles sémantiques et la gestion des erreurs. Elle a aussi généré le fichier TDS.csv pour visualiser la TDS sur excel.

Tous les membres du groupe ont participé à la relecture du code et au debug les problèmes rencontrés.

2. Travail effectué

2.1. AST

Une deuxième version de l'AST a été réalisée en se basant sur les remarques et corrections des encadrants du projet. Des modifications au niveau de l'affichage de l'AST a été faite de sorte qu'on peut par exemple mettre en évidence les champs de la définition d'une structure, différencier les paramètres d'une fonction et afficher leurs types, montrer le type de retour d'une fonction, les types des variables, etc.), la figure 1 illustre ceci. Des différents tests ont été faites sur l'AST pour vérifier sa bonne cohérence.

2.2. Table de symboles

Pour la génération de la table de symboles on a commencé par la création de sa structure. On a implémenté alors, la classe `SymbolTable` qui représente une `TDS` caractérisée par son `numero`, son `niveau` d'imbrication, son titre, une référence vers sa TDS mère, `parent`, une liste de ses TDS filles, `children` et la liste des lignes qui la constituent, `lines`. Le format du `titre` du bloc est le suivant : `I-G.P.N name`, `I` correspond au niveau d'imbrication du bloc, `G` est le numéro de bloc du grand-père, `P` le numéro du père,

1. une méthode de travail classique avec des réunions hebdomadaires et stand-up meetings si besoin
2. `https://trello.com/b/E3EJnI0p/projetpcl-1`

`N` le numéro du bloc actuel (vaut 1 s'il est le premier fils, 2 le deuxième, ainsi de suite) et `name` est le nom du bloc, en général c'est le nom d'une fonction ou `root` pour la `TDS` racine.

Ensuite, une classe `LineElement` a été mise en place pour ajouter des lignes dans la TDS. En effet, en instanciant la classe `LineElement` on peut créer une nouvelle ligne. Chaque ligne représente chaque nouveau élément à ajouter à la TDS : son identifiant `idf`, sa `nature` (`FUNCTION`, `PARAM_FUNC`, `VARIABLE`, `STRUCT`) et le `Symbole` qui le caractérise.

En effet, Symbole est une classe abstraite dont héritent quatre autres classes. Il y a alors quatre types de symboles :

- `FctSymbole` pour la déclaration des fonctions.
- `IntSymbole` pour la déclaration de variables de type `int`.
- `StructDefSymbole` pour la définition de nouveaux types de structure.
- `StructSymbole` pour la déclaration de variables de type struct.

On les a répartis ainsi en fonction des différentes informations pertinentes à ajouter dans la TDS pour chaque type, par exemple : `IntSymbole` n'ajoute que le type et l'identificateur de la variable alors que `FctSymbole` sauvegarde le type de retour, le nombre de paramètres et la liste des paramètres de la fonction.

Finalement, une fois que la structure de la TDS a été mise en place, Une classe `TdsVisitor` a été implémenté pour la construction et le remplissage de la TDS. On visite alors les mêmes classes qui décrivent les règles de notre grammaire lors de la génération de l'AST. Pour chaque programme, une TDS `root` est créée en première et ensuite pour chaque nouveau `bloc`, par exemple, pour une boucle `while` ou un `if`, une TDS fille lui est associée et ainsi de suite. De plus, comme notre programme comporte des fonctions prédéfinies (`print` et `malloc`), des lignes correspondantes sont ajoutées systématiquement dans la TDS `root` lors de sa création.

Voici un exemple de code en Listing 1 et la `TDS` de la figure 2 qui lui correspond :

```
1  int func(int x, struct one * st) {
2    int n;                                          // simple_test.exp
3    n = a->b->c = !x + ('c' + y) * -8;              /* if (test==20) happy else sade;
4                                                    'c'=99 ; '\''=39 ; '\"'=34 ; '\\'=92 */
5    return n;
6  }
7                                                    /*/ // //* hello world / // /*/
8  struct one {
9    struct another *root;
10 };
```

Listing 1 – Test : `simple_test.exp`

```
1 Error in #1-0.1 func: struct one is not defined
2 Error in #1-0.1 func: [idf] a not found
3 Error in #1-0.1 func: arrow problem: struct not defined
4 Error in #1-0.1 func: arrow problem: struct not defined
5 Error in #1-0.1 func: [idf] y not found
6 Error in #1-0.1 func: var c doesn't exist
7 Error in #1-0.1 func: assignment types (int and null) don't match
8 No int main() found.
9 Total: 8
```

Listing 2 – Erreurs sémantiques : `simple_test.exp`!h

2.3. Contrôles sémantiques

En ce qui concerne les côntroles sémantiques, on les gère en même temps que la création de la TDS. Ainsi, la majorité des contrôles sont réalisés dans `TdsVisitor`.

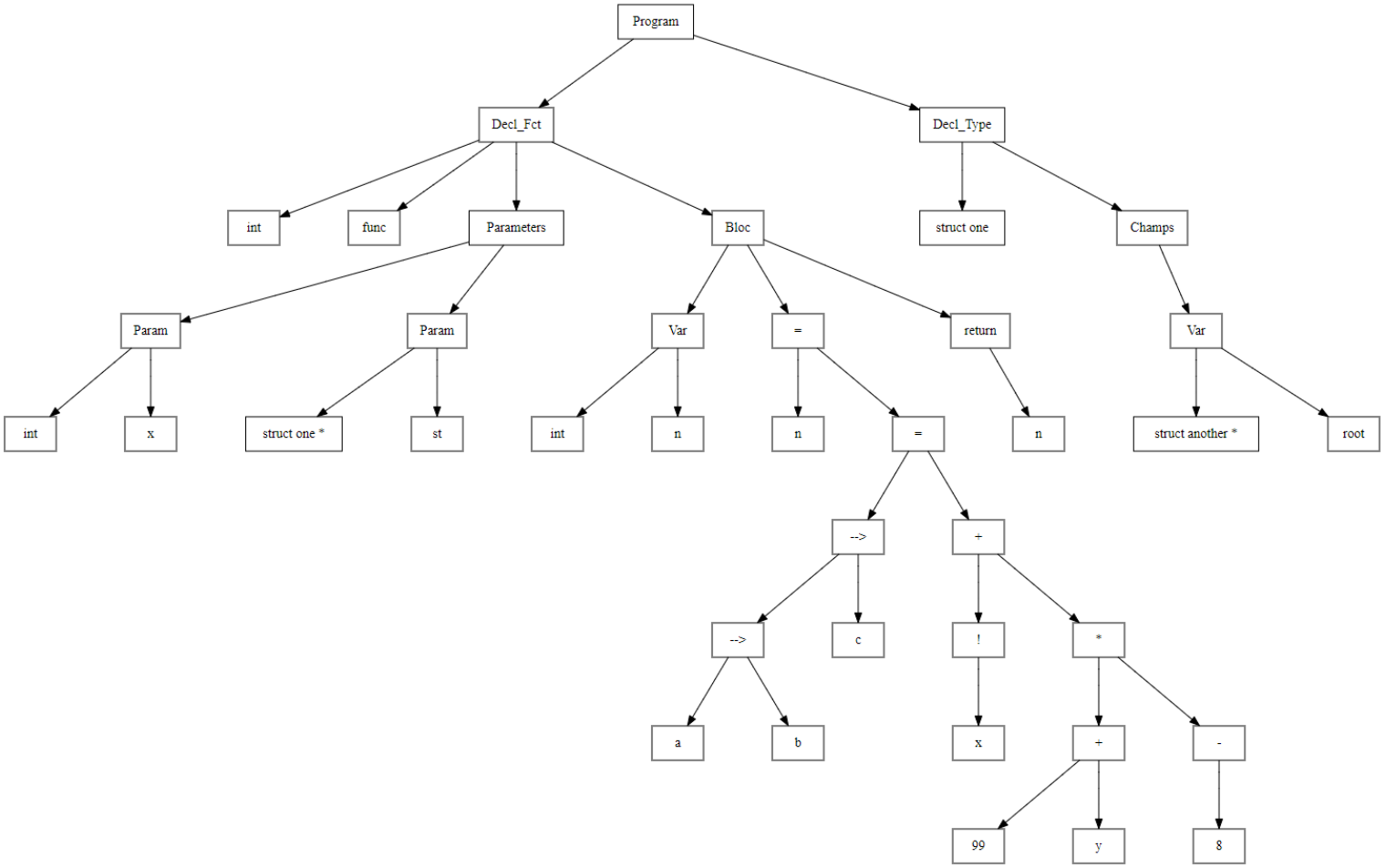

FIGURE 1 – AST : simple_test.exp

Table courante: #0-0 root	mon pere: Pas de parent			
IDF	NATURE	CARACTERISTIQUES SYMBOLE		
malloc	FUNCTION	Type de retour : void *	Nombre de params : 1	Params : (n __ int) |
print	FUNCTION	Type de retour : void	Nombre de params : 1	Params : (n __ int) |
func	FUNCTION	Type de retour : int	Nombre de params : 2	Params : (x __ int) | (st __ struct one *) |
one	STRUCT	Type: struct one	Champs : (struct another * : [root])	

Table courante: #1-0.1 func	mon pere: #0-0 root	
IDF	NATURE	CARACTERISTIQUES SYMBOLE
x	PARAM_FUNC	Type : int
n	VARIABLE	Type : int

FIGURE 2 – TDS : simple_test.exp

Les principales contrôles réalisés sont :

— La présence de la fonction `main` et le respect de sa signature : `int main()`.

— L'unicité des identifiants, donc pas de multiple définition d'un même identifiant dans un même bloc.

— La correspondance d'un appel de fonction avec la signature choisit lors de sa définition, le type de retour, le nombre de paramètres et leurs types).

— Lors des affectations, avoir des opérandes de même type.

— L'utilisation de fonctions, structures ou variables que si elles sont déjà définies ou déclarés.

— La correspondance entre le type de retour d'une fonction et la valeur qu'elle renvoie réellement.

— La bonne utilisation de `print` et `malloc` (respectant les signatures respectives : `void print(int n)` et `void * malloc(int n)`.

— N'accéder qu'aux champs définis lors de la déclaration des structures.

2.3. Gestion des erreurs

Pour la gestion des erreurs, notre classe `SymbolTable` possède une liste statique `Errors` qui contient tous les messages d'erreur ou de *warning* à afficher. A chaque fois qu'un contrôle sémantique détecte un problème, un message d'erreur est ajouté dans `Errors` en indiquant le plus d'information possible concernant l'erreur. Par exemple, le nom et le type de la fonction concernée, le bloc englobant.

Pour l'exemple précédent, après l'affichage de la TDS, nous affichons les messages d'erreur sémantique dans la listing 2.

3. Problèmes rencontrés et solutions

3.1. Grammaire

Lorsque nous sommes arrivés aux contrôles sémantiques, nous avons remarqué deux éléments qui ont pu être changés dans la grammaire pour enrichir les possibilités de notre compilateur et améliorer les messages d'erreur. Le premier est le traitement de l'affectation qui peut renvoyer la valeur affectée. Plus précisément, `if((a=b))` affecte `b` à `a` et renvoie `a` qui sera testé dans le `if`. Sans les doubles parenthèses, le compilateur gcc affiche un *warning*. Notre grammaire ne traite pas cette situation qui est un racourci pour faire une affection et exploiter la valeur directement, dans notre compilateur ceci nécessite deux étapes de code. Le deuxième concerne l'affichage des lignes, au niveau de la grammaire nous avons skip les sauts de ligne, nous aurions pu les garder pour compter le nombre de ligne du fichier source, ceci peut améliorer la compréhension des messages d'erreur.

3.2. AST

Nous avons eu des problèmes sur les instructions vides, la règle `instruction -> ;`, ceci a causé plusieurs erreurs null pointer exception, donc les visiteurs concernés (`ifThen`, `ifThenElse`, `while`, `Bloc`) ont dû être corrigés. Le problème aurait dû être signalé plus tôt, nous n'avons pas pensé à tester cet extrême en priorité.

3.3. Table de symboles

Au début, le choix d'une structure pour la représentation de la table de symboles n'était pas facile ce qui explique les différents changements dans la structure. Par exemple, la structure initiale traitait tous les symboles (détaillés au-dessus) de la même manière, et ainsi, que ce soit une définition de struct, une fonction ou même une déclaration de variable, une ligne identique est ajoutée à chaque fois dans la TDS ce

qui engendre des champs vides selon le type de symbole. Par conséquent, nous avons adopté la structure des symboles en ajoutant pour chaque symbole ses caractéristiques spécifiques dans la TDS. La construction de la TDS était basée sur des visiteurs ayant comme type de retour des `SymbolTable`. Mais, ensuite, cette configuration nous a posé quelques problèmes puisque dans certains cas on a besoin de retourner un type spécifique. Nous avons changé cette configuration par la suite alors pour que les visiteurs renvoient des `String`, ceci permet de contenir les informations, par exemple le type de retour des opérations. L'affichage de la TDS se faisait dans le terminal, pour une meilleur visibilité et une réutilisation, nous avons mis en place un système permettant d'exporter les données dans un fichier `.csv` ceci permet l'ouverture dans un classeur.

3.4. Contrôles sémantiques

La question initiale pendant cette phase était de choisir de traiter les contrôles sémantiques dans les visiteurs de la TDS ou dans d'autres fonctions indépendantes. Nous avons alors choisit de le faire simultanément lors de la construction de la TDS puisque nous visitons déjà le noeud concerné et le scan est du haut vers le bas. Ensuite, nous avons eu une petite difficulté sur la détermination du type d'une expression, par exemple : `a->b->c` ou `a+b*-c`. La solution adoptée, pour le premier, était alors de se baser sur les visiteurs pour vérifier que le champs `b` appartient réellement à la liste de champs de la structure de `a` et renvoyer à chaque fois le type de l'élément à droite, le type de `b` et ceci de manière récursive pour avoir à la fin le type générale de l'expression. Bien sûr, nous vérifions bien que chaque structure est déclarée dans la TDS. De la même manière avec les visiteurs, pour les expressions d'opérations arithmétiques, nous vérifions plus simplement si les deux éléments sont des entiers et nous renvoyons le type, à défaut des messages d'erreur sont préparés.

3.5. Gestion des erreurs

Initialement les erreurs ont été affichés automatiquement après chaque détection de problème, ceci n'était pas une bonne idée pour faire l'affichage, la TDS se trouvait coupée. C'est pour cela qu'une liste d'erreurs a été mise en place pour stocker tous les erreurs dans l'ordre chronologique et ensuite elles sont affichées ensemble dans l'ordre. Il y avait deux listes d'erreurs, l'une dans `SymbolTable` et l'autre dans `TdsVisitor` nous avons mis un attribut statique pour regrouper les erreurs. Une amélioration peut être apportée sur les messages d'erreur pour afficher les lignes concernées comme mentionné dans la partie 3.1. Grammaire ci-dessus.