Rapport du projet de Programmation Fonctionnelle et Traduction des Langages

Titouan BROCHARD et Yige YANG

Département des Sciences Numériques, deuxième année parcours B

Table des matières

1 Introduction

L'objectif premier de ce projet est d'ajouter le traitement de nouvelles constructions dans un compilateur du langage RAT réalisé en TP. Les constructions à ajouter sont : les pointeurs, les variables globales, les variables statiques locales et les paramètres par défaut. Vous trouverez ci-dessous la grammaire complète de notre compilateur.

Ces ajouts nous permettent d'avoir une compréhension plus profonde du fonctionnement d'un compilateur, puisque chaque construction nous amène à réfléchir sur les effets que l'on souhaite avoir pour chacune des passes et ainsi pouvoir réaliser une structure similaire dans le futur - qui pourrait ne pas être un compilateur - dans le but d'avoir un outil simple et fonctionnel.

Enfin, la réalisation d'un code sans "warning", bien écrit, correctement commenté, et testé, nous amène à réaliser un code pérenne qui pourra être ajusté au besoin.

Ce rapport a pour but de synthétiser les modifications apportées au compilateur afin d'ajouter les différentes constructions. Il sera donc divisé par construction.

1. $PROG' \rightarrow PROG\ \$$
2. $PROG \rightarrow VAR\star\ FUN\star\ id\ \ BLOC$
3. $VAR \rightarrow static\ TYPE\ \ id\ \ =\ \ E\ ;$
4. $FUN \rightarrow TYPE\ id\ \ (\ DP\ \ )\ BLOC$
5. $BLOC \rightarrow\ \{\ I\star\ \}$
6. $I \rightarrow\ TYPE\ \ id\ \ =\ \ E\ ;$
7. $\quad |static\ TYPE\ id\ \ =\ \ E\ ;$
   $\quad |\ \sout{id\ \ =\ \ E\ ;}$
8. $\quad |\ A\ \ =\ \ E\ ;$
9. $\quad |\ const\ \ id\ \ =\ \ entier\ ;$
10. $\quad |\ print\ \ E\ ;$
11. $\quad |\ if\ \ E\ \ BLOC\ \ else\ \ BLOC$
12. $\quad |\ while\ \ E\ \ BLOC$
13. $\quad |\ return\ \ E\ ;$
14. $A \rightarrow\ id$
15. $\quad |\ (\ *\ A\ )$
16. $DP \rightarrow\ \Lambda$
17. $|\ \ TYPE\ id\ \langle\ D\ \rangle ?\langle, TYPE\ id\ \langle\ D\ \rangle ?\rangle\star$
18. $D \rightarrow = E$
19. $TYPE \rightarrow\ bool$
20. $\quad |\ int$
21. $\quad |\ rat$
22. $\quad |\ TYPE\ *$
23. $E \rightarrow\ id\ \ (\ CP\ )$
24. $\quad |\ [\ E\ /\ E\ ]$
25. $\quad |\ num\ \ E$
26. $\quad |\ denom\ \ E$
    $\quad |\ \sout{id}$
27. $\quad |\ A$
28. $\quad |\ true$
29. $\quad |\ false$
30. $\quad |\ entier$
31. $\quad |\ (\ E\ \ +\ \ E\ )$
32. $\quad |\ (\ E\ \ *\ \ E\ )$
33. $\quad |\ (\ E\ \ =\ \ E\ )$
34. $\quad |\ (\ E\ \ <\ \ E\ )$
35. $\quad |\ (\ E\ )$
36. $\quad |\ null$
37. $\quad |\ (\ new\ TYPE\ )$
38. $\quad |\ \&\ id$
39. $CP \rightarrow\ \Lambda$
40. $\quad |\ E\ \langle,\ E\rangle\star$

FIGURE 1 – Grammaire étendue de notre compilateur

2 Les pointeurs

La première construction à ajouter concerne les pointeurs. L'ajout des pointeurs fut fastidieux puisque nous avons mis du temps à comprendre les erreurs que nous avions : en effet, nous avons fait face à plusieurs difficultés comme notamment l'écriture des tests sur Windows qui n'est pas acceptée par le Lexer (erreur qui est par ailleurs très difficile à détecter) et la compréhension que de nombreux fichiers devaient être modifiés et non uniquement les passes.

Objectifs principaux

- Modification des passes de l'AST en ajoutant unquement les affectations sans la possibilité d'utiliser les pointeurs (étape 0)
- Ajout de toutes les règles de grammaire concernant les pointeurs.

— $A \rightarrow (* A)$: déréférencement : accès en lecture ou écriture à la valeur pointée par A ;
— $TYPE \rightarrow TYPE *$: type des pointeurs sur un type TYPE ;
— $E \rightarrow null$: pointeur null ;
— $E \rightarrow (new\ TYPE)$: initialisation d'un pointeur de type TYPE ;
— $E \rightarrow \& id$: accès à l'adresse d'une variable.

FIGURE 2 – Règles de grammaire ajoutées pour les pointeurs

Évolution de la grammaire

Plusieurs règles ont été ajoutées : Ceci a donc été ajouté dans le parser, sans oublier l'ajout dans le lexer des élément ,new,null,* ainsi que des token associés dans le parser. Cependant, l'astérisque étant déjà utilisée pour la multiplication, nous n'avons pas eu à l'ajouter.

Modifications de l'AST

Il y a eu plusieurs modifications dans l'AST :Dans le module AstSyntax, on a ajouté les différents éléments suivants :

— — L'ajout du type affectable :
type affectable = |Ident of string (*A->id*) |Dereferencement of affectable (*A->(A)*)
— La transformation de l'expression Ident en affectable : ceci sert à pouvoir traiter dans le type affectable si nous sommes dans le cas d'un pointeur ou non.
| Affectable of affectable (*E->A*)
— L'ajout des nouvelles expressions.
| Null | New of typ | Adresse of string
— La transformation de l'instruction d'affectation qui ne prend plus le nom de la variable mais un affectable. Toujours dans le but de différencier les deux cas.
| Affectation of affectable * expression
— Dans le module AstType, nous avons simplement remplacé les noms par des info$_a stcommeaavaitputrefaitdanslesTP.Eneffet, partirdumomentoulaTDSatcre, nousn'avon$
— Dans les autres modules, ce sont des changements minimes.

Modifications de la passeTDS

La fonction analyse_tds_affectable est utilisée pour analyser les expressions affectables (variables ou pointeurs).

— is_left : détermine si l'identifiant est analysé dans une position d'affectation ou de lecture.
— true : Le contexte est une affectation (gauche de l'opérateur =).
— false : Le contexte est une lecture (droite de l'opérateur =).

Lorsqu'un identifiant est constant (InfoConst), il ne peut pas être utilisé à gauche d'une affectation. Si is_left = true et l'identifiant est une constante, une erreur est levée.

— Construction assez similaire aux constructions déjà faite en TP.
— Le déréférencement se fait de façon récursive à l'aide de la fonction analyse_tds_affectable

— $PROG \rightarrow VAR\star\ FUN\star\ id\ \ BLOC$
— $VAR \rightarrow static\ TYPE\ \ id\ \ =\ \ E\ ;$

FIGURE 3 – Modifications grammaticales pour les variables statiques globales

Modifications de la passe de typage

La fonction analyse_type_affectable analyse les affectables pour :
— Vérifier leur type.
— Retourner l'affectable avec son type associé dans l'AST.

Modifications de la passe de génération de code

Nous avons fait le choix de représenter null par un 0. Nous pouvons critiquer cela en indiquant que si nous faisons appel à cette variable, nous aurons tout de même une valeur. Cependant, il est à noter que grâce aux passes précédentes, il n'est pas possible d'obtenir la valeur d'un pointeur nul.

3 Les variables statiques globales

Les variables statiques globales peuvent être définies avant les fonctions et le bloc principal et sont accessibles partout. Nous avons longuement hésité sur la recherche locale ou globale dans la TDS lors d'une déclaration statique mais nous reviendront un peu plus en détails là-dessus juste après.

Évolution de la grammaire

La première règle de grammaire (modifiée d'une existante) permet de définir des variables statiques (avec une multiplicité de 0..*) avant la définition des fonctions et du bloc principal. Aussi, il a donc été nécessaire d'ajouter une seconde règle grammaticale permettant d'indiquer la déclaration d'une variable statique globale. On remarque l'apparition du mot clé *static* qu'il a donc fallut ajouter dans le lexer (ainsi que son token associé).

Modifications de l'AST

Ici, les modifications ont été relativement assez simple, puisque qu'il a suffit d'ajouter dans tous les modules l'apparition de *var* dans la définition d'un programme et il a également fallut créer l'arbre abstrait *var* qui suit globalement la syntaxe d'une déclaration de variable classique.

Modifications de la passeTDS

Comme nous avons pu l'indiquer dans l'introduction, pour l'analyse des variables globales, il a fallut réfléchir sur la recherche de celle-ci pour éviter toute double déclaration. Nous en avons conclut que les variables globales étant dans la TDS initiale, une

$$I \rightarrow static\ TYPE\ id\ =\ E\ ;$$

FIGURE 4 – Règle grammaticale pour les variables statiques locales

recherche locale était de mise, tandis que la recherche globale lors de la déclaration d'une variable classique ou des paramètres d'une fonction garantirait qu'il n'y ait pas de double déclaration.

Modifications de la passeCode

Ici, nous ne sommes vraiment pas sûrs de ce que nous avons fait. En effet, derrière les variables statiques se cache "à la compilation", donc nous avons décidé de mettre les variables dans le tas. Cependant, ne sachant si le registre initial était bon, nous ne sommes pas certains d'avoir réussi à les mettre dans le tas.

4 Les variables statiques locales

À l'inverse des variables statiques globales, les variables statiques globales ont une portée limitée à la fonction et n'est pas initialisée à chacun des appels. Ainsi, entre deux appels, si la variables a changé de valeur, celle-ci restera sauvegardée.

Évolution de la grammaire

Ici, il faut ajouter une instruction avec le mot clef *static* devant afin d'indiquer qu'une variable est statique et locale. Étant une instruction, elle ne pourra pas être confondu avec les variables globales qui sont des *var*.

Modifications

Globalement, cela s'apparente à une Déclaration donc il n'y a pas eu beaucoup de changements. En revanche, il est intéressant de noter que nosu avons fait une recherche locale pour la PasseTDS puisqu'elles ont une portée locale et qu'une variable locale à une fonction ne doit pas interférer avec les variables d'autres fonctions ou bien avec les variables globales.

5 Les paramètres par défaut

Cet ajout permet, lors de l'écriture d'une fonction, de définir une valeur par défaut pour un ou plusieurs des paramètres de celle-ci. On assigne cette valeur lors de la déclaration d'une fonction via l'opérateur (=). Ainsi, si une fonction est invoquée sans l'un des paramètres et que celui-ci est un paramètre par défaut, le code pourra compiler et utilisera donc cette valeur par défaut. Ce qui a compliqué la tâche ici, a été de trouver une solution pour la Table Des Symboles (TDS). Nous avons mis beaucoup de temps à déterminer comment faire et nous avons finalement fait le choix d'ajouter une Table Des Paramètres (TDP) en complément de la TDS.

$$DP \rightarrow \Lambda$$
$$\quad | \ TYPE\ id\ \langle\ D\ \rangle ?\langle, TYPE\ id\ \langle\ D\ \rangle ?\rangle\star$$
$$D \rightarrow = E$$

FIGURE 5 – Modifications dans la grammaire

Objectifs principaux

- Extension de la Table des Symboles (TDS) pour inclure les paramètres avec leurs valeurs par défaut.
- Création d'une Table des Paramètres par Défaut (TDP), une structure dédiée pour gérer ces valeurs.

Évolution de la grammaire

Afin de supporter les paramètres par défaut, les règles de grammaire ont été modifiées comme suit : - Ajout de la possibilité de définir une valeur par défaut avec '= E' : valdef : EQUAL exp=e {exp} - Les paramètres sont désormais représentés par un triplet (type, nom, valeur_optionnelle) param : t=typ n=ID a=option(valdef) { (t, n, a) }

Modifications de l'AST

Il y a eu deux modifications dans l'AST :Premièrement,, dans le module AstSyntax, il a fallut ajouter les paramètres par défauts dans la liste des paramètres et nous l'avons fait de la manière suivante : type fonction = Fonction of typ * string * (typ * string * expression option) list * bloc Deuxièmement, dans le module AstTds, il a fallut ajouter les paramètres par défaut pour la même raison : type fonction = Fonction of typ * Tds.info$_a st$ * $(typ*Tds.info_a st*expressionoption)list*blocCesmodifications, quienfaitn'ensontqu'uneseulepermet$

Fonctionnement de la TDP dans la passeTDS

La TDP est une structure basée sur une table de hachage qui associe à chaque fonction une liste de valeurs par défaut pour ses paramètres. **Modifications principales :**Analyse des paramètres dans analyse_liste_param :

- — Chaque paramètre est vérifié et ajouté à la TDS fille.
  - Si une valeur par défaut est spécifiée, elle est collectée et ajoutée à une liste temporaire.
- Stockage dans la TDP via analyse_tds_fonction :
  - Après analyse des paramètres, la liste des valeurs par défaut est stockée dans la TDP en associant le nom de la fonction à cette liste.
- Création de TDS fille :
  - Une nouvelle TDS est créée pour chaque fonction afin de gérer ses paramètres et ses variables locales. Cela assure une séparation claire des contextes.
    Voici un exemple :
    function f(a : int, b : int = 2, c : int = 3) { let x = 5 ; return a + b + c + x ; }
- TDS construite :

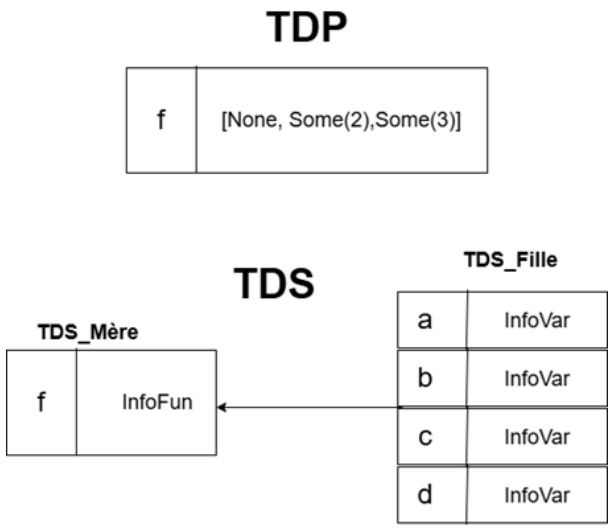

FIGURE 6 – Schéma récapitulant l'utilisant de la TDP

  - Une entrée InfoFun est ajoutée à la TDS mère pour f‘.
  - Une TDS fille est créée pour enregistrer a, b, c et les variables locales (x).
- TDP construite :
  - f -> [None, Some(2), Some(3)]

Ainsi, les paramètres manquants lors de l'appel seront complétés à l'aide de ces valeurs.

Modifications dans passeTypeRat

- Complétion des arguments avec TDP lors des appels de fonction :
  - Lorsqu'une fonction est appelée, les arguments manquants sont automatiquement complétés à l'aide des valeurs par défaut stockées dans la TDP. Cette logique a été intégrée dans la fonction analyse_type_appel_fonction.
- Fonctionnement général :
  - Les arguments fournis sont analysés normalement.
  - La TDP est consultée pour récupérer les valeurs par défaut des paramètres manquants.
  - Une liste complète d'arguments est construite en combinant les arguments fournis et les valeurs par défaut.
- Fonctions ajoutées :
  - zip_params : *Cette fonction combine deux listes : une liste d'arguments fournis et une liste de valeurs par défaut. Elle permet de compléter les paramètres manquants tout en préservant l'ordre des arguments.*
    - Si un argument est fourni, il est utilisé.
    - Sinon, la valeur par défaut correspondante est utilisée.

    Exemple :
- zip_params [Some 1 ; None ; Some 3] [None ; Some 2 ; None] Retourne : [Some 1 ; Some 2 ; Some 3]
- analyse_type_appel_fonction(modifiée) :
  - Cette fonction vérifie si les arguments fournis sont compatibles avec les types attendus. Elle complète également les arguments manquants à l'aide de la TDP.

Bénéfices de l'approche

- La TDP est une structure dédiée et indépendante, ce qui améliore la clarté et la modularité du code.
- La séparation entre la gestion des paramètres et des valeurs par défaut rend le système plus flexible.

Conclusion

Premièrement, dans ce projet, nous avons remarqué que les tests des TP jouaient un rôle crucial. En effet, ils nous permettaient principalement de vérifier que notre code n'avait pas d'effet non désirés sur le travail qui avait été fait pendant les TP, ce qui était très pratique. En revanche, nosu avons observé des difficultés

Dans ce projet, les tests de ont joué un rôle crucial, nous permettant d'identifier et de résoudre efficacement les problèmes rencontrés. Cependant, nous avons également rencontré plusieurs difficultés dans la rédaction des nouveaux tests :

- Problèmes liés au système Windows : nous utilisons Linux sur notre PC windows via un système d'exploitation Ubuntu installé dans WSL (Windows Subsystem for Linux). Or cela a engendré de nombreux bugs imprévisibles notamment dans la rédaction des tests non accepté par le Lexer s'ils étaient écris sur windows.
- L'exercice était fastidieux, il nous a fallut déterminer tous les tests intéressant afin de tout tester. Par ailleurs, leur rédaction après l'écriture du code pour certains des ajouts fut une très mauvaise idée. Nous pensions gagner du temps en les faisant à la fin, mais nous avons observés deux choses : à la fin, nous n'avons plus le temps de tout faire ; et leur écriture au préalable nous aurait permis une meilleure appréhension du sujet et probablement bon nombre de fausses pistes en moins.

En outre, nous avons également rencontré d'autres difficultés :

- Gestion du temps : Avec la période d'examens juste après les vacances de Noël, le temps était particulièrement limité et nous avions tenté de l'anticiper, mais très insuffisamment. Malgré les efforts fournis pendant les vacances, le projet a été un défi en termes de temps.
- Gestion des exceptions : Une partie des échecs de tests était liée à une gestion incorrecte des exceptions, un aspect qui aurait pu être mieux anticipé.
- Compréhension : Nous n'avions pas non plus anticipé les difficultés de la langue. En effet, nous comprendre était une tâche difficile, engendrant des erreurs de communications, des quiproquos ainsi que des parties du projet réalisées en doublon... De plus, nous pensons qu'il pourrait être judicieux d'ajouter une close spéciale dans les rapports indiquant que la partie écrite par un élève étranger ne serait pas notée sur l'orthographe ainsi que la synthaxe. En effet, sinon l'étudiant aura tendance à aller trouver des outils de traductions sur internet et donne l'impression d'une moitié de rapport écrite avec une IA. Ainsi, l'unique solution trouvée a été de proposer une réécriture des parties du rapport écrite par l'élève étranger. Or ce temps n'a absolument pas été anticipé.

  Enfin, les améliorations possibles que l'on a noté sont les suivantes :

- Meilleure gestion du temps : Une planification plus rigoureuse via un outil comme Trello aurait permis de réduire la pression due aux délais et d'améliorer notre communication.
- Plus de tests unitaires : Bien que nous ayons écrit des tests, en ajouter davantage

aurait renforcé la robustesse du projet puisque nous n'avons pas réalisés l'ensemble des tests unitaires et d'intégration.