Projet de programmation fonctionnelle et de traduction des langages

Mohammed ZOUICHA
Reda ZHANI - B2

Département Sciences du Numérique - Deuxième année
2023-2024

Table des matières

Table des figures

1 Introduction

Ce projet constitue une application concrète des connaissances acquises dans le cadre du cours de programmation fonctionnelle et de traduction des langages.
L'objectif de ce projet de programmation fonctionnelle et de traduction des langages est d'améliorer le compilateur du langage RAT réalisé en TP de traduction des langages . L'extension prévue englobe des fonctionnalités clés telles que les pointeurs, les tableaux,goto, et les boucles for.

Dans ce rapport, on détaillera les modifications apportées à l'AST et les stratégies d'implémentation pour chacune de ces fonctionnalités. De plus, des tests additionnels ont été intégrés pour garantir la validation des nouvelles fonctionnalités implémentées. La grammaire du langage RAT est également spécifiée ci-dessous :

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

FIGURE 1 – Grammaire (EBNF) du langage RAT tetendu

2 Pointeurs

Dans le langage RAT , les pointeurs sont manipulés à l'aide d'une syntaxe similaire à celle du langage C. Voici les règles ajoutées :

$A \rightarrow (*A)$
$TYPE \rightarrow TYPE*$: Permet de gérer les pointeurs de pointeurs
$E \rightarrow$ null
$E \rightarrow$ (new $TYPE$)
$E \rightarrow$ &id

2.1 Evolution de l'AST et Tds :

On a rajouté un nouveau **type** affectable contenant :

1. *Ident of string* : Identifiant
2. *Valeur of affectable* : accès en lecture ou ecriture à la valeur pointée par l'affectable

Ainsi, on a rajouté de nouvelles **expressions** :

1. *Null* : Un pointeur null
2. *New of typ* : Allouer un nouveau pointeur de type typ
3. *Adresse of string* : L'adresse d'une variable
4. *Affectable of affectable* : Un affectable

Concernant le lexer.ml, On a ajouté trois tokens null, new et &.

2.2 Jugement de typage :

- **E→ nul** :
$$\sigma \vdash \text{null} : \text{Pointeur(Undefined)}$$
- **A→(* A)** :
$$\frac{\sigma \vdash A : \text{Pointeur } (\tau)}{\sigma \vdash (*A) : \tau}$$
- **T Y P E⟶T Y P E *** :
$$\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash TYPE* : \text{Pointeur } (\tau)}$$
- **E⟶(new T Y P E)** :
$$\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash (newTYPE) : \text{Pointeur}(\tau)}$$
- **E⟶ &id** :
$$\frac{\sigma \vdash \text{id} : \tau}{\sigma \vdash \& \text{ id} : \text{Pointeur } (\tau)}$$

2.3 Modification des passes :

La passe qui traite **la gestion des identifiants** a été étendue pour traiter les nouvelles informations liées aux pointeurs. Cela concerne l'analyse récursive de l'affectable contenu dans la valeur (i.e : *x)
Dans **la passe de typage**, On a mis à jour la fonction qui gère l'analyse des affectables pour gérer la compatibilité du type. Dans **la passe de placement de mémoire**, la taille associée à

un pointeur est 1. Cela reflète la convention courante où les pointeurs occupent généralement un seul octet en mémoire.
Ensuite, dans **la passe de la génération du code**, concernant la fonction qui gère l'analyse d'un pointeur, on a défini un booleen pour differencier les affectables q'un souhaite modifier. Ensuite, à l'aide de load on copie au sommet de la pile le bloc souhaité, et on effectue un **storei** ou un **loadi** selon la nature de l'affectable.

3 Les tableaux

Dans le langage RAT , les pointeurs sont manipulés à l'aide d'une syntaxe similaire à celle du langage C. Voici les règles ajoutées :
$A \rightarrow (A \mid E])$
TYPE $\rightarrow$ TYPE[]
$E \rightarrow$ (new $TYPE[E]$)
$E \rightarrow \{CP\}$: initialisation
Concernant le lexer, on a rien modifier pour ajouter les tableaux à notre compilateur.

3.1 Evolution de l'AST et Tds

On a rajouté un nouveau type Tab typ, qui permet d'avoir un table d'un type quelconque mais les elements sont de meme type. Ainsi, on a ajouté de nouvelles expressions :

1. creation of typ * expression : permet de créer un tableau
2. Initialisation of expression list : Initialiser un tableau

3.2 Jugement de typage

- **A$\rightarrow$(A [E])** :

$$\frac{\sigma \vdash A : tab(\tau) \quad \sigma \vdash E : \mathcal{T}}{\sigma \vdash (A[E]) : \tau}$$

- **E$\longrightarrow$(new TYPE[E])** :

$$\frac{\sigma \vdash TYPE : \tau \quad \sigma \vdash E : \mathrm{int}}{\sigma \vdash TYPE[E] : tab(\tau)}$$

- **E$\longrightarrow${C P}** :

$$\frac{\sigma \vdash CP : \tau * ... \tau \quad \sigma \vdash E : \mathrm{int}}{\sigma \vdash CP : tab(\tau)}$$

3.3 Modification des passes :

Dans **la passe de gestion des identifiants**, nous avons modifié la fonction chargée de l'analyse des affectables pour inclure une analyse récursive des affectables contenus dans chaque case. Cette modification permet d'accéder en mode lecture ou écriture à une case spécifique d'un tableau. Dans **la passe de typage**, on vérifie la compatibilité des types y compris le type des indices et le type des listes pour l'initialisation des tableaux.
Ensuite, dans **la passe de génération du code**, lors de la création d'un tableau, le processus se déroule comme suit : nous commençons par générer le code de l'indice, puis nous empilons l'adresse où se trouve l'élément. Ensuite, nous effectuons un (**LOADI**) de la taille de l'élément à lire. En ce qui concerne l'initialisation d'un tableau, nous déterminons d'abord la taille du tableau en fonction de la taille de la liste multipliée par la taille du type des éléments de la liste.

Ensuite, nous définissons un pointeur pour le tableau. Nous générons ensuite le code pour chaque expression. Avec l'instruction **loada (-taille-1) "ST"**, nous empilons l'adresse du tableau, et **loadi 1** prend l'adresse en sommet de pile, copiant ainsi le bloc de taille 1 à cette adresse. Enfin, avec **storei (taille tableau)**, nous prenons l'adresse en sommet de pile et déplaçons le bloc de taille du tableau (les éléments que nous souhaitons utiliser pour remplir le tableau) à cette adresse.

4 Boucle For :

La notation étendue de RAT permet d'écrire des boucles "for" en utilisant une syntaxe similaire à celle du langage C. La structure générale est la suivante :

I⟶for (int id = E ; E ; id = E) BLOC

Concernant le lexer, on ajouter un seul token For. Ainsi, cette structure permet de définir et d'itérer une boucle "for" en spécifiant clairement l'indice de départ, la condition d'arrêt et la façon dont l'indice évolue à chaque itération.

4.1 Evolution de l'AST et Tds

On a ajouté dans l'ast l'instruction for

1. **For of typ * string * expression * expression * string * expression * bloc**
   Le premier paramètre représente la définition et l'initialisation de l'indice de boucle. Le deuxième paramètre correspond à la condition d'arrêt de la boucle. Enfin, le dernier paramètre indique l'évolution de l'indice de boucle, généralement une incrémentation ou une décrémentation de 1, bien que toute expression soit autorisée.

4.2 Jugements de typage

- **I⟶for (int id = E ; E ; id = E) BLOC** :

$$\frac{\sigma \vdash E_1 : int \quad \sigma \vdash id : int \quad (id, int) :: \sigma \vdash BLOC : \ void, [] \quad (id, int) :: \sigma \vdash E_2 : bool \quad (id, int) :: \sigma \vdash E_3 : int}{\sigma \vdash \text{ for } (int \quad id = E_1 \quad ; \quad E_2 \quad ; \quad id = E_3) : void, []}$$

4.3 Modification des passes

Dans la phase de gestion des identifiants, des modifications sont apportées à la fonction responsable de l'analyse des instructions. Tout d'abord, une recherche globale est effectuée dans la table des symboles (TDS) pour vérifier que l'identifiant de la boucle "for" n'a pas déjà été déclaré.

Ensuite, une table des symboles fille (TDSFille) est créée pour gérer les identifiants déclarés à l'intérieur de la boucle "for".

Après avoir analysé les expressions, une recherche locale est effectuée dans la TDSFille de la boucle pour récupérer l'identifiant spécifique à cette boucle. Cela permet d'utiliser le même identifiant dans la condition d'arrêt. À ce stade, la TDSFille de la boucle "for" ne contient que l'identifiant (la tds de la boucle a dans ce moment que n).

Dans **la passe de typage**, on vérifie la compatibilité des types, par exemple, que l'identifiant de la boucle est bien un entier, et la condition d'arret est bien un booleen. Dans la passe de placement de mémoire, la pile reste la meme apres la terminaison d'une boucle.

Ensuite, dans **la passe de génération du code**, on génére le code de l'intialisation d'identifiant de la boucle for comme etant une declaration, ainsi, on génére le code du bloc de la boucle et l'incrémentation de l'identifiant , comme etant une affectation. En outre, des étiquettes de début et de fin sont générées pour faciliter l'itération du bloc de la boucle (via l'instruction "jump" vers l'étiquette de début) et pour permettre la terminaison de la boucle (via l'instruction "jumpif 0" vers l'étiquette de fin).

5 Goto

RAT étendu autorise l'utilisation d'instructions "goto" pour effectuer des sauts vers des points spécifiques du programme, marqués par des étiquettes. Une étiquette est un nom suivi du caractère ':'.
En effet, dans le lexer, on a ajouté deux tokens : goto et ' : '

5.1 Evolution de l'AST et Tds

Dans l'Ast, on a ajouté de nouvelles instruction :

- Goto of string : La syntaxe pour le saut, où l'instruction suivante sera celle marquée par l'identifiant.
- DeclGoto of string : Pour marquer une instruction comme destination possible d'un "goto"

Ainsi, pour permetter à une autre variable/constant ou fonction d'avoir le meme identifiant q'une etiquette, on a ajouter dans le type info un nouveau type : InfoEtiquette similaire à l'InfoVar et on a cree un nouveau TdsEtiquette pour stocker ces InfoEtiquette.

5.2 Modification des passes

Dans **la passe de gestion des identifiants**, une table des symboles spécifique aux étiquettes est ajoutée.

Lors de la déclaration d'une étiquette, la recherche se fait dans la table des symboles dédiée aux étiquettes. Si l'étiquette est déjà déclarée, une erreur est déclenchée pour éviter les déclarations redondantes. Cette approche garantit la gestion appropriée des étiquettes utilisées avec l'instruction "goto".
Dans les phases de **typage** et de **placement en mémoire**, les instructions "goto" demandent peu de modifications.

Lors de **la passe de génération de code**, l'instruction "jump" avec une étiquette spécifiée indique qu'à l'exécution, l'instruction suivante sera celle identifiée par l'étiquette en question. En parallèle, l'utilisation de l'instruction "label n" permet de marquer cette étiquette comme une destination pour une instruction "goto".

6 Conclusion

En conclusion, ce projet a été une occasion d'appliquer et d'approfondir les connaissances acquises en TP. Il a permis d'étendre de manière significative le compilateur du langage RAT, tout en mettant en pratique les bonnes pratiques de programmation. Ce travail constitue une étape importante dans le développement d'un compilateur capable de traiter des constructions avancées, tout en restant fidèle aux principes de la programmation fonctionnelle. Le compilateur peut être améliorer de différente sorte, par exemple, introduire des fonctionnalités avancées dans le système de types, telles que le polymorphisme ou les génériques, pour accroître l'expressivité du langage.