ENSEEIHT

Traduction des langages

2022-2023

Rapport du projet

Oussakel Rachida et Charifi Yasmine

HPC et big data

Contents

List of Figures

1 Introduction

Le compilateur du langage RAT a été initialement développé pour gérer les quatre passes fondamentales de compilation : la gestion des identifiants, le typage, la gestion de la mémoire et la génération de code. Cependant, cette version ne prend pas en compte certaines fonctionnalités couramment utilisées par les programmeurs, telles que les pointeurs, la conditionnelle sans un bloc else, la structures de l'opérateur ternaire et l'utilisation des boucles "loop" à la Rust. Le projet consiste donc à étendre le compilateur du langage RAT pour inclure ces fonctionnalités supplémentaires.

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

2 Modification de l'AST et Tds

Notre compilateur sera capable de traiter le langage RAT étendu en prenant en compte les modifications demandées dans le sujet. Pour se faire, nous avons rapporté les modifications ci-dessous sur l'Ast et Type :

2.1 Ajout des pointeurs

• Ajout d'un nouveau cas de type : Pointeur of type
• Ajout du nouveau type : affectable = Valeur of affectable | Ident of string
• Modifier l'instruction : Affectation of affectable * expression
• Ajout des cas aux expressions : Affectable of affectable | Null | New of typ | Adresse of string

2.2 Ajout du bloc else optionnel dans la conditionnelle

Pour prendre en compte cette structure conditionnelle, nous n'avons rien changé dans l'Ast ni dans type. Les modifications seront plutot dans ce qui suit.

2.3 Ajout de l'opérateur ternaire

• Ajout de l'expression : Ternaire of expression * expression * expression

2.4 Ajout des boucles "loop" à la Rust

• Ajout d'un nouveau type d'information associé aux boucles infinies : InfoLoop of string*string*string (Le premier paramètre refère à l'identifiant de la boucle, le 2 ème au début de celle-ci et le dernier à la fin de la boucle).
• Ajout des instructions : Loop of string * bloc , Continue of string, Break of string, qui permettent de gérer la boucle infinie.

3 modifications des passes

3.1 Les pointeurs

Dans un premier temps, nous avons modifier le compilateur pour prendre en charge de nouvelles fonctionnalités, notamment la gestion des pointeurs et de l'instruction "new".
Pour intégrer ces nouvelles fonctionnalités, nous avons dû mettre à jour plusieurs parties de notre compilateur, notamment le lexer, le parser, l'analyseur syntaxique abstrait (AST), le typage, la gestion de la mémoire et la génération de code.
Pour mettre à jour le lexer, nous avons dû ajouter de nouveaux tokens pour reconnaître les mots-clés "new", "" et "null". Pour mettre à jour le parser, nous avons ajouté de nouvelles règles de production pour permettre la définition et la manipulation des pointeurs.
Nous avons également apporté des modifications à l'AST pour prendre en compte la gestion des identifiants et des affectations. Pour le typage, nous avons ajouté de nouveaux types et mettre à jour les méthodes d'analyse de type pour prendre en compte les pointeurs.
Enfin, nous avons mis à jour la gestion de la mémoire en définissant la taille d'un pointeur comme étant 1, en ajoutant de nouvelles méthodes d'analyse de code pour générer le code approprié pour les nouvelles expressions et instructions tout en distinguant le cas d'évaluation et le cas d'affectation.

3.2 Le bloc else optionnel dans la conditionnelle

Tout d'abord, nous avons mis à jour le parser pour utiliser la nouvelle règle de production ajoutée à la grammaire : I $\rightarrow$ if E BLOC.
Ainsi, nous avons modifier la méthode "Analyse_code_instruction" de génération de code pour prendre en compte la présence ou l'absence d'un bloc "else" dans une instruction conditionnelle et générer le code approprié.
Tandis que les autres passes restent inchangées, vu qu'on traitait le bloc du "else" comme étant un bloc vide.

3.3 La conditionnelle sous la forme d'un opérateur ternaire

Pour ajouter l'opérateur ternaire dans notre compilateur, nous avons mis à jour la grammaire de notre langage de programmation pour inclure la syntaxe de l'opérateur ternaire. Nous avons aussi ajouté la règle de production : E → (E ? E : E).
Ce qui nous a conduit à mettre à jour le lexer pour ajouter de nouveaux tokens pour reconnaître les symboles '?' et ':'. Ainsi que la mise à jour du parser pour utiliser la nouvelle règle de production que nous avons ajouté à la grammaire. Nous avons aussi mis à jour l'AST pour prendre en compte l'opérateur ternaire comme cité ci-dessus. Quant à la passe de la gestion des identifiants, nous avons modifié la méthode "analyse_tds_expression" pour prendre en compte le nouveau cas d'expression. ternaire.
Nous avons également mis à jour la passe typage pour prendre en compte l'opérateur ternaire et vérifier que les expressions "expression1" et "expression2" ont des types compatibles et la "condition" est bien de type booléen.
Dans la passe de la génération de code, on s'est basé sur le code de l'instruction de la conditionnelle.

3.4 Les boucles "loop" à la Rust

Dans un premier temps, nous avons mis à jour la grammaire de notre langage de programmation pour inclure la syntaxe de la boucle "loop".Nous avons ainsi ajouté la règle :
1. I → loop BLOC
2. I → id : loop BLOC
3. I → break;
4. I → break id;
5. I → continue;
6. I → continue id;

Nous avons également mis à jour le lexer pour ajouter un nouveau token pour reconnaître le mot-clé "loop", "break" et "continue". Ainsi que le parser pour utiliser les nouvelles règles de production que nous avons ajouté à la grammaire. Pour la gestion des identifiants, nous avons modifier l'analyse des instructions où nous avons pris en compte le cas de la boucle infinie : tout en analysant l'identifiant de la boucle et en analysant l'ensemble des instructions de son bloc. Pendant chaque passage, nous avons ajouté les instructions loop, break et continue.

4 tests

En plus des tests fournis par les sources de Tps, nous avons ajouté des tests pour évaluer le fonctionnement des nouvelles instructions et expressions ajoutées en projet.

5 Conclusion

Ce projet a été passionnant car il nous a permis de découvrir en profondeur le fonctionnement d'un compilateur et de comprendre comment il est construit. Nous n'avions jamais imaginé que nous aurions à creuser aussi profondément dans les grammaires et les différentes passes de compilation pour créer, même de manière simplifiée, un compilateur pour un nouveau langage comme RAT.