Rapport du projet: Traduction des langages-Language Rat

Auteurs: Haliloua Othmane et Oussama Karmaoui, 2SN-B.

Janvier 2023

Table des matières

1 Introduction :

Le compilateur du langage RAT a été construit dans une version basique lors des séances des TPs où on a travaillé sur les quatre passes fondamentaux : La gestion d'identifiants, le typage, le placement mémoire et la génération du code. Cepandant, cette version ne traite pas les d'autres outils de language qu'un programmeur va sans doute vouloir intégrer dans son code : les pointeurs, les boucles infinies, les instructions d'échappement de boucles, les bloc conditionnels sans "else" obligatoire et les expressions conditionnelles sous forme d'un opérateur ternaire. Ce projet a pour but donc d'étendre le compilateur du langage RAT déja réalisé afin de prendre en compte toutes les nouvelles constructions qu'on vient de citer.

2 Types-Evolution de l'AST :

2.1 Ast-syntaxique :

Après l'ajout des terminaux et règles de grammaires au lexer et parser correspondants aux nouvelles instructions et expressions ajoutés , on a modifié AstSyntaxe en ajoutant :
un type affectable = Ident of string | Valeur of affectable qui est un type correspond à un identifiant ou à la valeur adressé par un pointeur identifié .

* On a ajouté au type expression :

Affectable of affectable : à la place de Ident of string pour manipuler le cas des pointeurs.

Ternaire of expression*expression*expression : qui represente la conditionnellle avec opérateur ternaire .

New of typ : décrit un pointeur sur le type typ qui est nouvellement déclaré (utilisé en déclaration) .

Addresse of string : adresse de la variable d'identifiant en paramètre .

Null : le pointeur null .

* On a modifié le type instruction en ajoutant les nouvelles instructions de meme :

Affectation of affectable*expression : on a remplacé Ident en Affectation par affectable pour qu'on puisse affecter des valeurs aux valeurs adressées par les pointeurs .

ConditionnelleSansElse of expression*bloc : qui exprime le conditionnelle sans else (else optionnelle) .

Loop of bloc : pour la loop sans identifiant , paramétré par son bloc seulement.

LoopId of string*bloc : pour la loop avec identifiant string , qui le caractérise avec son bloc .

Pour le break , break avec identifiant , continue et continue avec identifiant on a ajouté au type instruction : **Break , BreakId of string , Continue , Continue of string**

* On a ajouté de meme le type Pointeur of typ dans le type typ en Type.ml et l'info InfoLoop of string dans Tds.ml qui une info sur les loop identifiés .

2.2 Ast-Tds :

A coté des changements faits dans les Tps 2-6 : on remplace le type string dans les paramètres des types expression , instruction , et fonction par le type Tds.info_ast , on fait la meme chose aussi pour affectable : affectable = Ident of Tds.info_ast | Valeur of affectable . Pour les expressions et les instructions en relation avec les loop identifiées ($LoopId, BreakId, ContinueId$) , la Tds.info_ast dans leur paramètres représente info_ast_to_info de l'info de la boucle loop correspondante qui est sous forme de InfoLoop(identifiant de loop) .

2.3 Ast-Type :

on garde la meme syntaxe de AstType faite dans les tps 2-6 (Affichage devient AffichageInt | AffichageRat | AffichageBool et Fonction devient : Fonction of Tds.info_ast * Tds.info_ast list * bloc (on a inséré typ des param dans la Tds.info_ast)) , et on réécrit les meme types d'expressions et instruction récemment ajoutés en AstTds.

2.4 Ast-Placement :

le type expression est celui de AstType , de meme pour affectable . Meme changements pour les instructions précèdentes et bloc (bloc = instruction list * int et Retour of expression*Tds.info_ast devient Retour of expression*int*int) fonction reste de meme syntaxe que AstType et de meme pour les nouvelles instructions ajoutés .

3 Jugements de typage :

(les dash représentent des espaces ici)

Jugement 1 :

$$\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash TYPE^* : Pointeur(\tau)}$$

Jugement 2 :

$$\frac{\sigma \vdash TYPE : \tau}{\sigma \vdash New - TYPE : Pointeur(\tau)}$$

Jugement 3 :

$$\frac{\sigma \vdash id : Pointeur(\tau)}{\sigma \vdash (*id) : \tau}$$

Jugement 4 :

$$\sigma \vdash Null : Pointeur(Undefined)$$

Jugement 5 :

$$\frac{\sigma \vdash id : Pointeur(\tau) - -\sigma \vdash E : \tau}{\sigma \vdash (*id) = E : void, []}$$

Jugement 6 :

$$\frac{\sigma \vdash id_1 : Pointeur(\tau) - -\sigma \vdash id_2 : \tau}{\sigma \vdash (*id_1) = \&id_2 : void, []}$$

Jugement 7 :

$$\frac{\sigma \vdash id : Pointeur(\tau)}{\sigma \vdash id = Null : void, []}$$

Jugement 8 :

$$\frac{\sigma \vdash Bloc : void, []}{\sigma \vdash loop\{Bloc\}\text{: } void, []}$$

Jugement 9 :

$$\frac{\sigma \vdash id : etiquette --(id, etiquette) :: \sigma \vdash Bloc : void, []}{\sigma \vdash id : loop\{Bloc\}\text{: } void, []}$$

Jugement 10 :

$$\frac{\sigma \vdash E : bool --\sigma \vdash Bloc : void, []}{\sigma \vdash if(E)\{Bloc\}\text{: } void, []}$$

Jugement 11 :

$$\frac{\sigma \vdash E_1 : bool --\sigma \vdash E_2 : \tau --\sigma \vdash E_3 : \tau}{\sigma \vdash (E_1 ? E_2 : E_3) : \tau}$$

4 Pointeurs :

4.1 Analyse lexicale et syntaxique :

Nous avons tout d'abord ajouté des nouveaux tokens au lexer pour prendre en compte les modifica- tions apportées sur la grammaire : notre compilateur doit reconnaître les terminaux : new, et null. Ensuite, nous avons ajouté au parser les nouveaux règles de productions permettant la définition et la manipulation des pointeurs.

4.2 Passe de gestion d'identifiants :

Pour la passe de gestion des identifiants, les strings deviennent des Tds.Infoast pour les affectables, on a aussi ajouté une fonction d'analyse vis à vis de la table des symboles pour le type "affectable" de l'AST syntaxique, pour qu'on en se sert pour analyse des expression sous la forme de : Affectable of affectable. En plus de ceci, on a ajouté des analyses pour les autres nouvelles expressions.

4.3 Passe de typage :

En gros, le pointeur est vu comme une variable de type Pointeur(τ), ce qui fait qu'il n'y a pas de traitement strictement particulier pour les pointeurs, quoique comme expressions ou comme intervenants dans une instruction à part l'ajout de la fonction d'annalyse de typage de ce qui est de type affectable.

4.4 Passe de placement memoire :

Un pointeur est une adresse, on ne s'intéresse qu'à cette dernière, qu'on peut stocker facilement dans une seule case, par conséquent, la taille d'un pointeur a été fixée à 1.

4.5 Passe de génération de code :

Pour la génération de code, il nous fallait prendre en compte si un affectable est simplement une expression employée à droite d'une affectation ou à gauche. l'annalyse d'un Affectable of affectable dans l'analyse de génération de code d'une expression traite le premier cas, pourtant, lors de l'analyse de l'instruction d'affectation on se sert de la fonction d'analyse d'affectable ajoutée qui traite le cas à gauche, que ce soit un simple identifiant ou un déréférencement. Et pour ce faire, on s'est servi des opération : Empiler les adresses avec LOADA d[r], réaliser une allocation de mémoire avec SUBR MAlloc et empiler/écrire les mots à partir de leurs adresses au sommet de la pile avec LOADI(n) et STOREI (n).

5 Bloc else optionnel :

5.1 Analyse lexicale et syntaxique :

En utilisant les meme tokens utilisés par l'instruction if..else et en ajoutant la règle if BLOC end au parser, permettant la définition du bloc else optionnel .

5.2 Passe de gestion d'identifiants :

On fait une analyse identifiants sur l'expression et le bloc qui constituent le bloc else optionnelle .

5.3 Passe de typage :

On analyse le type de l'expression de condition et on vérifie si son type est compatible au type Bool , dans ce cas on analyse le type du bloc du else optionnelle .

5.4 Passe de placement memoire :

On s'interesse seulement au bloc du else optionnelle sur lequel on fait une analyse_dep_bloc à l'aide des parametre d'adressage déplacement d et registre reg .

5.5 Passe de génération de code :

Contrairement au code de if..else , on ajoutera pas une étiquette etiqSinon , mais on utilise plutot jumpif 0 etiqFin juste après l'analyse_code de l'expression de la condition de else optionnel.

6 Conditionnelle Ternaire :

6.1 Analyse lexicale et syntaxique :

la syntaxe de la conditionnelle sous forme d'opérateur ternaire nécessite l'introduction de 2 nouveaux tokens dans le lexer : le " ?" à droite de l'expression de la condition et le " :" entre la valeur_si_vrai et la valeur_si_faux selon la condition , et sans oublier d'ajouter la règle correspondante au parser.

6.2 Passe de gestion d'identifiants :

On fait une analyse_tds_expression pour chaqun des 3 exprerssions de l'opérateur ternaire .

6.3 Passe de typage :

On fait une analyse de type des 3 expressions d'opérateurs ternaire , tout en vérifiant si le type de la première expression est booleen ou non , et que les 2 dernières expressions ont un type compatible .

6.4 Passe de placement memoire :

pas d'analyse de placement mémoire vu que la conditionnelle ternaire est une expression .

6.5 Passe de génération de code :

Son code Tam est un peu près de celui du Conditionnelle avec if...else , en effet on crée 2 étiquettes etiq2 et etiqfin , et après du code de l'expression booleene , on fait jumpif 0 etiq2 , après on analyse le code de l'expression si vrai puis jump etiqFin , et après on analyse le code de l'expression si faux et puis etiqFin.

7 Loop à la Rust :

7.1 Analyse lexicale et syntaxique :

On a commencé par l'ajout des tokens *loop*, :, *continue* et *break* pour la reconnaissance en terme de lexique, puis on a ajouté les règles de production dans l'annalyseur syntaxique. Les boucles sans nom ont été prises comme une instruction du type : Loop of Bloc, Bloc faisant référence à la liste des instructions qui se répéteront dans cette boucle. D'autre part, les boucles avec un nom ont été prises comme une instruction de type LoopID of string * bloc, la chîne de charactères faisant référence au nom. Sans oublier les instruction break et continue qui peuvent être accompagnées du nom de la boucle correspondante ce qu'a été traité similairement.

7.2 Passe de gestion d'identifiants :

Le nom représenté par le string quoique pour loopID ou countinue ou break, sera postérieurement remplacé par une information : $InfoLoop\ of\ string$ qui nous permettra d'ajouter nos informations de boucle dans la table des symboles via la clé : "nomdelaboucle_loop" qui nous permettra de déclarer des variables de même nom qu'une boucle, étant donné que les informations des variables sont enregistrées via le clé : "nomdelavariable_id".

7.3 Passe de typage :

Aucun traitement particulier que celui fait en TP, c'est une simple analyse de typage sur les instruction du bloc de la boucle.

7.4 Passe de placement memoire :

Aucun traitement particulier différent de celui fait en TP, l'info d'une boucle ne comporte pas un champs pour un placement mémoire.

7.5 Passe de génération de code :

La difficulté particulière pour la génération du code des boucles est la labélisation des ettiquettes, ainsi que la reconnaissance de quelle boucle on échappe par l'instruction *break* et dans quelle boucle on ignore les instructions restantes pour réitérer via l'instruction *continue*, pour ce faire, on a envisagé de définir un pointeur sur un couple d'entiers, la première projection initiée à 1 s'incrémente après l'occurence d'un bloc "loop", la deuxième projection initiée à 0 s'incrémente à chaque fois qu'on entre dans un nouveau niveau de boucle, la valeur pointée par ce pointeur en quelque sorte représente les cordonnée de du bloc "loop" "actuel", qui sera délimité par les labels LoopStart_1ereprojection_2emeprojection et LoopEnd_1ereprojection_2emeprojection. La gestion est maintenue par la remise à zero de la deuxième projection et l'incrémentation de la première lors de l'appel de *Break* Id_loop dans la boucle du plus bas niveau (càd 1), càd la plus englobante, alors qu'il n'y a qu'une décrémentation de la deuxième projection si on est pas dans la boucle la plus englobante.

N.B : Des fichiers de tests on été ajoutés dans le répertoire tests de la source, chacun dans sa rubriques spécifiée (tds, type, placement, tam) pour vérifier que le code marche correctement vis à vis de la compilation des nouvelles règles ajoutées.

8 Conclusion :

Ce projet était intéressant dans le sens où il permettait de découvrir le fonctionnement précis d'un compilateur et sa construction. L'utilisation des différentes règles de grammaire et les analyses faisant suite à une évolution de l'Abstract Syntax Tree sont des concepts bien riches sur lesquelles le projet s'est focalisé.

Pour la passe de gestion de symboles on a suffixé les noms des informations par "id" pour toute sorte d'identifiant de variable ou fonction ou constante, et "loop" pour les boucles loop, et ce pour avoir la possibilité de définir une loop et une variable de même nom.

Le projet nous a présenté un défi au long de son déroulement, les séances du TP nous ont permis de nous familiariser avec les notions applicatives de la traduction des languages, spécialement avec le grand nombre de fichiers employés. Mais après, on s'est bien sortis de la boue, pourtant la conception d'une solution pour la génération de code pour loop à la Rust a été la plus dûre, dans le sens où il falait différencier les différents labels des boucles loop, en prenant compte l'incidence et le niveau de chacune, ce qui est directement lié au instructions break et continue, il nous fallait un moyen pour savoir de quelle boucle on échappe et dans quelle boucle on continuait.

Pour les blocs if, il y a une contrainte de la même nature ça a été déjà traitée via la fonction getEtiquette() qui s'occupait de la gestion.

Au final, nous avons beaucoup appris sur la structuration et la construction des compilateurs et leur fonctionnement. Notre perception des compilateurs s'étend maintenant à plus qu'une petite ligne decommande qu'on passe à notre terminal pour compiler sans savoir ce qui passe derrière les coulisses de ce compilateur.