INP TOULOUSE ENSEEIHT

RAPPORT
Projet de programmation fonctionnelle et de traduction des langages

Gabriel ROCHAIX–YAMAMOTO
Jonas LAVAUR

Département Sciences du Numérique - Deuxième année
2022-2023

Sommaire

1 Introduction

Le but de ce projet était de compléter le compilateur créé lors des séances de TP afin d'y intégrer de nouvelles constructions : les pointeurs, le bloc else optionnel (c'est-à-dire le if sans else), la conditionnelle ternaire et la loop à la Rust.

De plus, nous avons également ajouté une construction non demandée : l'opérateur += qui additionne deux valeurs (expression à droite et valeur de la variable à gauche) et range le résultat dans la variable (à gauche). On a donc ajouté à la grammaire l'instruction $I \rightarrow id \mathrel{+}= E$.

Enfin, l'archive rendue contient un fichier texte : "testsEnPlus.txt" (dans le dossier "tests") qui répertorie tous les tests que l'on a ajouté concernant les nouvelles constructions. Dans le dossier "tests", on a également créé un dossier supplémentaire nommé "ast_syntaxe" qui permet de tester uniquement le lexer et le parser (compilateur avec les 4 passes "Nop").

2 Explications sur les types - Jugements de typages

Dans cette partie, nous allons présenter l'évolution de la structure des AST et donner les jugements de typages associés aux nouvelles constructions du langage.

2.1 Pointeurs

Pour *AstSyntax* :
- Nous avons ajouté un type $affectable$ qui est soit un $Ident\ of\ string$, soit un $Deref\ of\ affectable$.
- Nous avons supprimé dans expression $Ident\ of\ string$.
- Nous avons ajouté dans expression $New\ of\ typ$, $Adresse\ of\ string$, $Affectable\ of\ affectable$ et $Null$.
- Nous avons ajouté dans instruction $Affectation\ of\ affectable * expression$

Pour les 3 autres AST (*AstTds*, *AstType* et *AstPlacement*), nous avons fait les mêmes modifications mais les *string* deviennent des $info\_ast$.

Voici les jugements de typages concernant les nouvelles règles de grammaire associées aux pointeurs :

- $\dfrac{\sigma \vdash A : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash A\ =\ E : void, []}$

- $\sigma \vdash id : \tau$

- $\dfrac{\sigma \vdash A : Pointeur(\tau)}{\sigma \vdash (*\ A) : \tau}$

- $\dfrac{\sigma \vdash TYPE : \tau}{\sigma \vdash TYPE * : Pointeur(\tau)}$

- $\dfrac{\sigma \vdash TYPE : \tau}{\sigma \vdash (TYPE) : \tau}$

- $\sigma \vdash A : \tau$

- $\sigma \vdash null : Pointeur(Undefined)$

- $\dfrac{\sigma \vdash TYPE : \tau}{\sigma \vdash (new\ TYPE) : Pointeur(\tau)}$

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

2.2 Bloc else optionnel

Les AST n'ont pas à être modifiés (cf. partie 3.2).

Voici le jugement de typage concernant le bloc else optionnel :

- $$\frac{\sigma \vdash E : bool \qquad \sigma \vdash BLOC : void}{\sigma \vdash if\ E\ BLOC : void, []}$$

2.3 Conditionnelle ternaire

Pour les 4 AST (*AstSyntax*, *AstTds*, *AstType* et *AstPlacement*) nous avons ajouté dans expression $Ternaire\ of\ expression * expression * expression$.

Voici le jugement de typage concernant la conditionnelle ternaire :

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

2.4 Loop à la Rust

Pour les 4 AST (*AstSyntax*, *AstTds*, *AstType* et *AstPlacement*) nous avons ajouté dans instruction $Loop\ of\ string * bloc$, $Break\ of\ string$ et $Continue\ of\ string$. Les *strings* correspondant au nom de la loop associé, puis à l'étiquette de la loop après la passe des identifiants.

Lorsque l'on définit une loop, un break ou un continue sans identifiant, le string reste vide jusqu'à la passe des identifiants où l'on génère un string "autocreated@". Le "@" permet de s'assurer que le programmeur ne définira pas une loop de même identifiant car le "@" n'est pas dans le regex des identifiants.

Voici les jugements de typages concernant la loop à la Rust :

- $$\frac{\sigma \vdash BLOC : void}{\sigma \vdash loop\ BLOC : void, []}$$

- $$\frac{\sigma \vdash id : void \qquad \sigma \vdash BLOC : void}{\sigma \vdash id : loop\ BLOC : void, []}$$

- $\sigma \vdash break : void, []$

- $$\frac{\sigma \vdash id : void}{\sigma \vdash break\ id : void, []}$$

- $\sigma \vdash continue : void, []$

- $$\frac{\sigma \vdash id : void}{\sigma \vdash continue\ id : void, []}$$

2.5 Construction bonus : opérateur +=

Les AST n'ont pas à être modifiés (cf. partie 3.5).

Voici le jugement de typage concernant l'opérateur += :

- $\dfrac{\sigma \vdash id : int \qquad \sigma \vdash E : int}{\sigma \vdash id \mathrel{+}= E : void, []}$

- $\dfrac{\sigma \vdash id : rat \qquad \sigma \vdash E : rat}{\sigma \vdash id \mathrel{+}= E : void, []}$

3 Explications concernant les nouvelles constructions

Dans cette partie, nous allons expliquer nos choix de conception pour les différentes constructions implémentées.

3.1 Pointeurs

Nous n'avons pas utilisé le même caractère "*" pour la multiplication et pour les pointeurs dans le lexer car les tests fournies avec le code initial et les tests d'exemple fournies dans le sujet n'utilisent pas le même caractère. Pour les pointeurs nous avons utilisé le "∗" (U+2217).

3.1.1 Passe de placement

Dans l'analyseur des affectables, $Ident$ et $Affectable$ n'ont pas de mémoire associée, donc on peut renvoyer leur $AstPlacement$ associé, couplé à 0. De même pour $Affectation$ dans l'analyseur des instructions.

3.1.2 Passe génération de code

Dans l'analyseur des affectables :
- Pour $Deref\ a$, on analyse d'abord a à l'aide de l'analyseur des affectables. Ensuite si le paramètre modif est true, alors on souhaite écrire, donc on utilise (STOREI 1), sinon on souhaite lire donc on utilise (LOADI 1). 1 car l'adresse que pointe un pointeur est un entier, donc de taille 1.
- Pour $Ident\ info\_ast$, si l'on a une $InfoConst$ on lit un entier donc on utilise LOADL, si l'on a une InfoVar on utilise STORE ou LOAD de même selon si modif est Vrai ou Faux.

Dans l'analyseur des expressions :
- Pour New, on crée un nouveau pointeur. Il faut donc allouer une case mémoire à l'aide de MALLOC.
- Pour $Adresse\ info\_ast$, on souhaite empiler une adresse donc on utilise LOADA et on récupérère les paramètres de déplacement et registre dans $info\_ast$.
- Pour $Affectable$, on applique juste l'analyseur des affectables avec modif à Faux car si l'on rentre dans cet analyseur et que l'on match avec un $Affectable$ alors on veut accéder en lecture à ce dernier.
- Pour $Null$, on ne fait rien.

Dans l'analyseur des instructions :
- Pour $Declaration$, on ne modifie rien.
- Pour $Affectation$, on appliquer d'abord l'analyseur des expressions auquel on concatène le résultat avec ce que renvoie l'analyseur des affectables avec modif à Vrai car dans ce cas, si l'on match avec une $Affectation$ alors on veut accéder en écriture à l'affectable.

3.2 Bloc else optionnel

On a rajouté une règle de grammaire de type instruction permettant d'écrire un bloc if sans else, puis nous l'avons rattaché à l'AST conditionnel avec la liste d'instructions, du bloc else, vide. Cela permet de se rattacher à du code déjà existant et fonctionnel, de faciliter le travail et de rendre le code plus facile à maintenir. Ainsi, nous n'avons pas à modifier les AST, ni les passes.

Un autre choix de conception aurait été de changer les AST et créer une conditionnelle sans bloc else, mais on aurait écrit du code redondant.

3.3 Conditionnelle ternaire

Nous avons dû ici mettre à jour les AST, car il ne s'agit pas d'une instruction mais d'une expression donc on ne pouvait pas se rattacher à un code existant. Nous avons donc rajouté un type *Ternaire* dans le type expression de chaque AST.

3.4 Loop à la Rust

3.4.1 Lexer et Parser

On a ajouté les tokens "loop", "break" et "continue" dans le lexer, ainsi que les règles de grammaire de type instruction dans le parser, pour créer une loop, un break et un continnue. Nous avons associé un Bloc au contenu de la loop car cela nous permet de nous rattacher à du code qui existe déjà.

3.4.2 Passe des identifiants

Lors de la passe des identifiants, nous ne pouvons pas utiliser la TDS qui existe déjà car nous devons pouvoir créer des loops de même identifiant que des variables. Nous avons donc choisi de créer une nouvelle TDS sous la forme d'une pile d'étiquettes seulement pour gérer les loops. Lorsque l'on rentre dans une loop, on ajoute à la pile l'étiquette associée, et lorsque l'on sort d'une loop on retire une étiquette.

Cette structure présente plusieurs avantages :

- Lorsque l'on rencontre un break ou un continue sans nom, on peut les associer à la loop la plus interne en accédant à la dernière étiquette ajoutée dans la pile.

- Lorsque l'on rencontre un break ou un continue avec nom, on peut vérifier si l'on est bien imbriqué dans une loop de même nom en vérifiant dans la pile d'étiquettes si elle existe.

- On peut gérer le fait d'avoir deux loops de même identifiant non imbriquées, car l'étiquette de la première loop aura été dépilée lorsque l'on rencontre la deuxième loop.

Pour pouvoir gérer le fait d'avoir deux loops de même identifiant et imbriquées (warning en Rust), on a choisi de numéroter nos loops selon le rang d'apparition de leur nom. Ainsi les étiquettes qu'on associe aux loops sont composées de leurs noms et d'un numéro, leurs labels qu'on utilisera sera donc leurs noms et numéros concaténés (string qu'on envoie à la passe suivante).

La pile des étiquettes est donc une pile $(string * int)\ list$. La TDS des loops est un couple de piles $((string*int)\ list)*((string*int)\ list)$, avec la première pile étant la pile des étiquettes et la deuxième qui compte combien de fois on a crée une loop d'un même identifiant. Ainsi :

- Dans la première pile (la pile des étiquettes), lorsqu'on a 2 loops de même nom *name* imbriquées, on aura 2 éléments de string *name*, mais d'int différents, qui correspondent à leurs rangs d'apparition. Dans la deuxième pile, il n'y aura qu'une seule fois *name*, avec l'entier associé égal à 2 (nombre d'occurences de *name*).

- Lorsque l'on crée une loop de nom *name*, on obtient son rang d'apparition en incrémentant son entier associé dans la deuxième pile, et on peut ensuite ajouter son étiquette (*name*, rang d'apparition de *name*) dans la pile des étiquettes.

- Lorsque l'on rencontre un break/continue de nom *name*, on peut obtenir l'étiquette de la loop de nom *name* la plus interne en cherchant dans la pile des étiquettes le premier élément de string *name*.

On utilise une référence pour la TDS de loop, ainsi on n'a pas besoin de renvoyer la TDS à chaque analyse d'instruction.

Enfin, un autre choix de conception aurait été d'ajouter les loops à la TDS déjà existante et donc de créer une $InfoLoop$ dans le type $info$ du module Tds. Mais, on aurait des problèmes de conflits avec les variables ou les fonctions de même nom, et on aurait dû trouver une autre solution pour savoir si l'on est imbriqué dans une loop.

3.4.3 Passe génération de code

Lorsque l'on rencontre une loop d'étiquette *label*, on commence par écrire un label $label\_start$, puis on analyse son bloc, et enfin on écrit un label $label\_end$.

Lorsque l'on rencontre un break d'étiquette *label*, on écrit $jump\ label\_end$.

Lorsque l'on rencontre un continue d'étiquette *label*, on écrit $jump\ label\_start$.

3.5 Construction bonus : opérateur +=

Nous avons ajouté le token "+=" au lexer et la règle de grammaire associée à cette construction dans le parser. De plus, $id \mathrel{+}= expression$ revient à faire l'affectation suivante : $id = id + expression$. Ainsi, nous n'avons pas à modifier les AST, ni les passes car on se sert du code déjà existant.

Un autre choix de conception aurait été de changer les AST et créer un $PlusEq\ of\ string * expression$ mais cela n'aurait pas été pertinent.

4 Conclusion

Pour conclure, les plus grandes difficultés que nous avons rencontré résident dans les changements à effectuer lors de l'ajout des pointeurs et dans la gestion des identifiants pour les loops à la Rust.

De plus, concernant le bloc else optionnel et l'opérateur +=, il était très intéressant de remarquer que l'on n'était pas obligé de modifier les AST et de se servir de code déjà existant afin de ne pas faire de la redondance de code. Egalement, concernant la passe de génération de code pour les pointeurs, la commande permettant d'obtenir le fichier "out.tam", pour voir le code TAM généré, nous a été très utile afin de comprendre nos erreurs.

Enfin, au niveau des améliorations éventuelles, on se rend compte qu'on a développé de nombreuses fonctions auxiliaires pour la passe de gestion des identifiants concernant les loops et peut-être qu'il y a une solution plus efficace.