



Rapport Projet Compilateur d'un langage RAT

ARNOLD Cyprien

DUONG Tom

Département Sciences du Numérique - 2ème année - 2021-2022

Table des matières

1	Introduction	3
2	Les Pointeurs	4
2.1	Rajout ast	4
2.2	Passe Tds	5
2.3	Passe Typage	5
2.4	Passe génération de code	5
3	L'opérateur d'assignation d'addition	7
3.1	Rajout ast	7
3.2	Passe Tds	7
4	Les types nommés	8
4.1	Rajout ast	8
4.2	Passe Tds	9
5	Les enregistrements	10
5.1	Rajout ast	10
6	Conclusion	11

1 Introduction

Le but du projet de programmation fonctionnelle et de traduction des langages était d'étendre le compilateur du langage RAT réalisé en TP de traduction des langages pour traiter de nouvelles constructions : les pointeurs, l'opérateur d'assignation d'addition, les enregistrements et les types nommées.

Le compilateur est écrit en OCaml et respecte les principes de programmation fonctionnelle étudiés lors des cours, TD et TP de programmation fonctionnelle.

Nous nous sommes appuyés sur la grammaire donnée dans le sujet pour compléter le *Lexer* et *Parser* puis nous avons complété les 4 passes faites en TP. Nous avons rajouté des tests dans les 4 fichiers tests associés au passe et avons complété le fichier *ast*.

Nous n'avons pas eu le temps de faire les passes associées aux *Enregistrements*. Nous allons maintenant vous présenter la conception des 3 nouvelles extensions du langage.

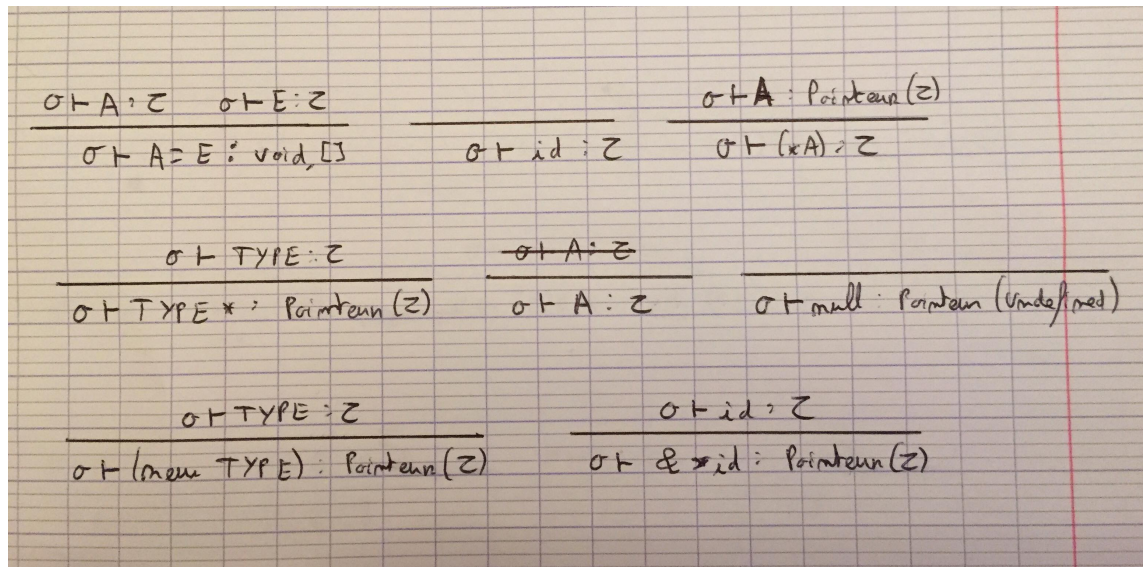
2 Les Pointeurs

2.1 Rajout ast

La grammaire correspondant au Pointeur est :

- $A \rightarrow (* A)$: 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.

Voici les jugements de typage associés :



Nous avons rajouté le type **Pointeur** qui prend un type en constructeur.

Dans l'*ast*, nous avons rajouté un nouveau type nommé **affectable** qui prend en constructeur soit :

- un **Ident** qui prend en constructeur un string qui correspond au nom de la variable. Ce constructeur était déjà présent précédemment.
- un **Dref** qui prend en constructeur un affectable qui correspond à (* aff) et qui donc représente l'accès à la valeur pointée par le pointeur.

Dans expression nous avons rajouté :

- un **Affectable** qui prend en constructeur un affectable et qui correspond à une variable du type qu'on a définie précédemment.
- un **Null** qui correspond au pointeur null
- un **NouveauTyp** qui prend en constructeur un type et qui correspond à l'initialisation d'un pointeur d'un type TYPE.

- une **Adresse** qui prend en constructeur un string et qui correspond à l'adresse du pointeur du string donné en paramètre.

Dans instruction nous avons rajouté **AffectationPointeur** qui vient remplacer l'instruction **Affectation** (le généralise) car maintenant on peut affecter un identifiant mais aussi un pointeur.

Dans les passes suivantes, nous ajoutons au type **affectable** un constructeur qui est **EntierCons** et qui prend en paramètre un entier. Ce dernier sert lors de la passe d'identifiant à remplacer les variables de type InfoCons en Entier et donc transmettre cette valeur jusqu'à la passe de code .

2.2 Passe Tds

Ce que nous avons rajouté à la phase de code est de vérifier que le pointeur représenté par **Dref** pointe bien vers un identifiant qui a été défini avant, si ce n'est pas le cas nous levons l'exception **IdentifiantNonDeclare**.

De même, pour l'expression **Adresse**, nous vérifions que le string associé n'est pas une constante, sinon nous levons l'exception **MauvaiseUtilisationIdentifiant**

2.3 Passe Typage

Nous avons implémentés les jugements de typage dans cette passe à savoir :

- **Dref(Pointeur)** a pour type le type dont le pointeur pointe
- un **Null** a pour type **Pointeur(Undefined)**
- un **NouveauTyp(type)** a pour type **Pointeur(type)**
- un **Adresse(variable)** a pour type **Pointeur(type_de_la_variable)**.

Une fois fait, nous avons complété la fonction *est_compatible* :

- **Pointeur(Undefined)** est compatible avec n'importe quelle autre pointeur
- **Pointeur(Type1)** et **Pointeur(Type2)** sont compatible ssi Type1 et Type2 sont compatible.

Une fois fait, nous sommes directement passé à la passe de génération de code car nous n'avons pas besoin de modifier la passe de placement mémoire.

2.4 Passe génération de code

Pour la gestion d'une expression d'un pointeur, il y avait 2 cas : si le pointeur était à gauche de l'instruction, il fallait affecter la nouvelle valeur à l'adresse donc faire un **STOREI**, si l'expression était à droite de l'expression, il fallait charger la valeur de l'adresse donc faire un **LOADI**.

Pour la gestion de **NouveauType(type)**, nous faisons un **SUBR MAlloc** avec comme instruction juste au dessus la valeur de la taille du type.

Pour la gestion de **Null**, nous avons utilisé **SUBR Mvoid** qui renvoie "la valeur adresse non initialisée" (doc TAM).
Sinon pour gérer **Adresse(info)**, nous utilisons **LOADA** qui renvoie l'adresse de la variable se trouvant au registre indiqué dans info.

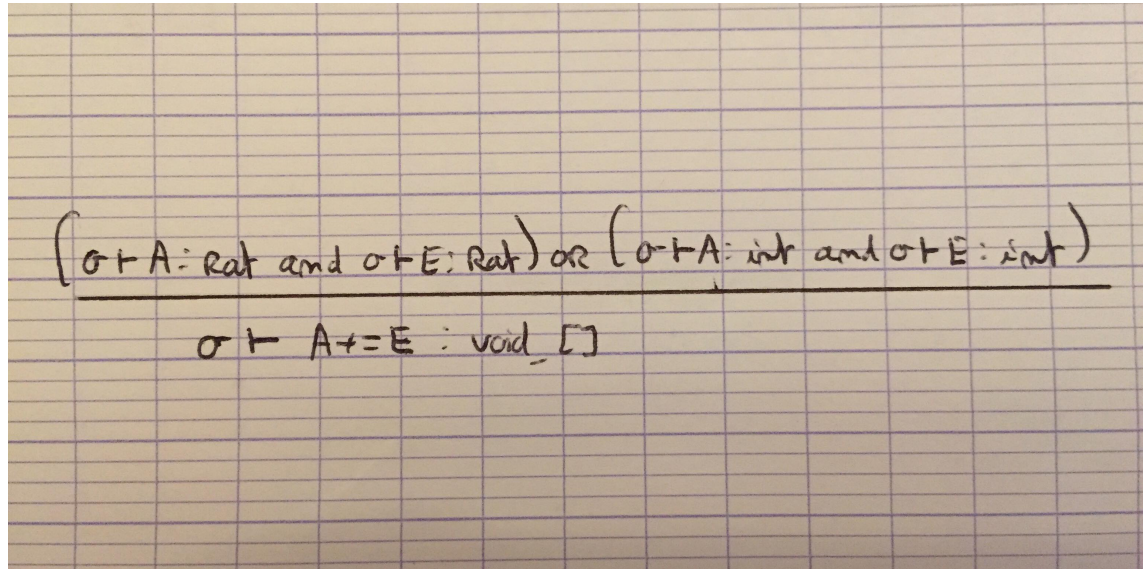
3 L'opérateur d'assignation d'addition

3.1 Rajout ast

La grammaire correspondant à l'assignation d'addition est :

— $I \rightarrow A += E ;$

Voici les jugements de typages associés :


$$\frac{(\sigma \vdash A : \text{Rat} \text{ and } \sigma \vdash E : \text{Rat}) \text{ or } (\sigma \vdash A : \text{int} \text{ and } \sigma \vdash E : \text{int})}{\sigma \vdash A += E : \text{void} []}$$

Nous avons simplement ajouté au type instruction le constructeur **AssignmentAdd** qui prend en argument un affectable (ce qu'on veut affecter) et l'expression (l'expression avec laquelle on va faire la somme).

3.2 Passe Tds

Dans cette passe, nous analysons que l'affectable a bien été défini et que l'expression est correcte puis nous transformons **AssignmentAdd(affectable,expression)** en **AffectationPointeur(affectable_analyse,Binaire(Plus,affectable_analyse,expression_analysée))** ce qui permet directement d'enlever le constructeur **AssignmentAdd** pour les phases suivantes et évitant ainsi toute modification dans les autres passes.

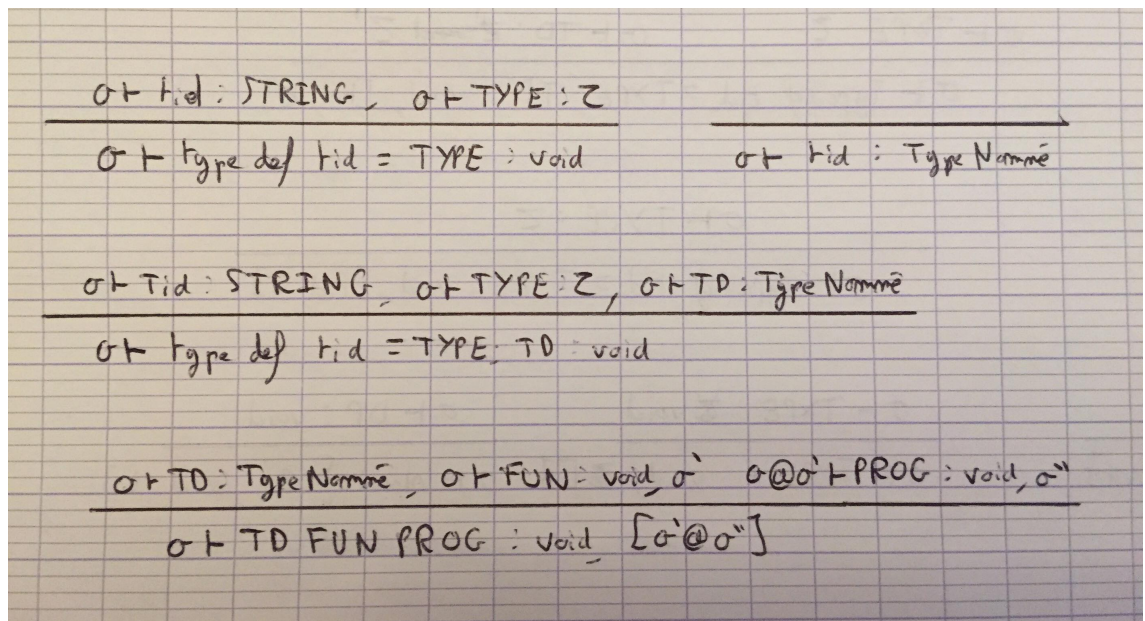
4 Les types nommés

4.1 Rajout ast

La grammaire correspondant aux types nommés est :

- ~~PROG~~ → ~~FUN PROG~~
- ~~PROG~~ → ~~TD FUN PROG~~ : définition globale (au programme) d'un type nommé ;
- ~~TD~~ →
- ~~TD~~ → ~~typedef tid = TYPE ; TD~~
- ~~I~~ → ~~typedef tid = TYPE ;~~ : définition locale (à un bloc) d'un type nommé ;
- ~~TYPE~~ → ~~tid~~ : utilisation d'un type nommé.

Voici les jugements de typage associés : Il y a donc ici 2 manières de déclarer



un type nommé : localement à un bloc ou globalement et tous les blocs y ont accès.

Dans l'*ast*, nous avons ajouté au type instruction le constructeur **Declaration-TypeNomme(nom,type)** qui prend un argument un string correspondant au nom du type nommé et son type.

Nous avons aussi modifié les arguments du constructeur **Programme** qui prend maintenant en plus d'une *fonction list* et *bloc*, une *instruction list* qui correspond à une liste de **DeclarationTypeNomme** enregistré dans le *parser*.

4.2 Passe Tds

Comme précédemment, l'idée ici est de tout faire dans la passe Tds. Elle va consister à remplacer tous les types nommés déclarés dans le programme par leur type réel.

Pour cela, nous vérifions tout d'abord que la déclaration des types nommés (globaux et locaux) est possible en regardant si un nom identique a déjà été déclaré dans l'Ast. Si ce n'est pas le cas, on rajoute le type nommé dans une **InfoVar** dans laquelle on met son nom et son type associé.

Une fois cela fait, il faut remplacer toutes les déclarations de types où les types sont des types nommés par le type effectif à savoir dans :

- l'instruction **Declaration(type,nom,expression)**
- la déclaration du type de retour d'une fonction dans *analyse_tds_fonction*
- la déclaration des types des paramètres d'une fonction dans *analyse_tds_fonction*

A partir de là, tous les types nommés sont remplacés par leur type réel et donc il n'y a pas à modifier les autres passes.

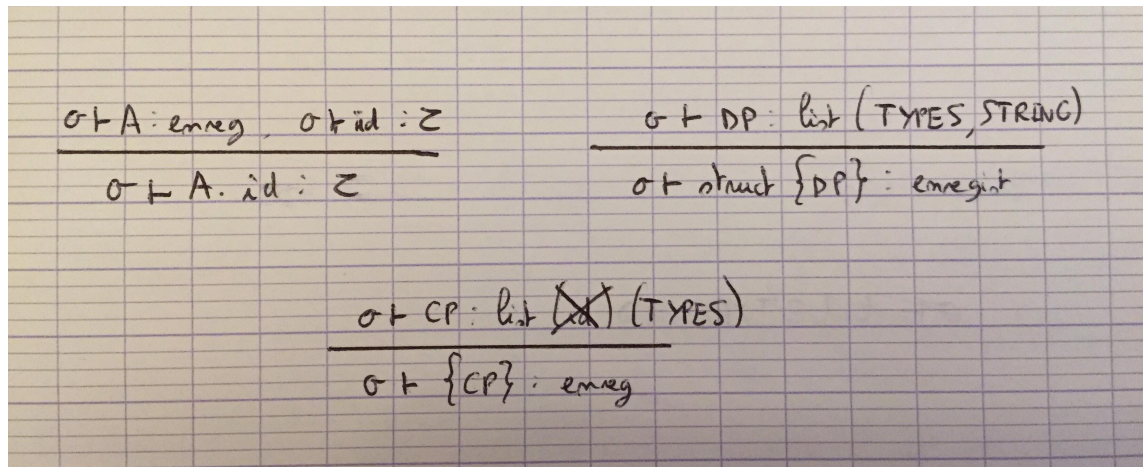
5 Les enregistrements

5.1 Rajout ast

La grammaire correspondant aux enregistrements est :

- $TYPE \rightarrow struct \{ DP \}$: définition d'un enregistrement (DP est une liste des types et noms des champs);
- $A \rightarrow (A.id)$: accès à un champ de l'enregistrement
- $E \rightarrow \{ CP \}$: création d'un enregistrement avec la liste des valeurs de ses champs

Voici les jugements de typage associés :



Nous avons rajouté un constructeur **Champ** au type affectable qui prend en argument un *affectable* correspondant à l'affectable dont on va utiliser l'attribut et un *string* correspondant à l'attribut.

Nous avons aussi rajouté le constructeur **ListChamp** au type expression qui prend une liste d'expression en argument qui va permettre de créer l'enregistrement.

Malheureusement, nous n'avons pas eu le temps de compléter les passes associées à cette extension du langage.

6 Conclusion

Ce projet nous a permis de bien mettre en application ce que nous avons vu en TD et cours de traduction des langages. Comprendre les différents aspects mis en jeu lors de la compilation est intéressant en tant que programmeur car on comprend mieux l’affichage des erreurs et pourquoi un programme de programmation est plus rapide que d’autre (par exemple en Python, on ne déclare le type des variables donc c’est au compilateur de le déterminer, rajoutant donc du temps).

Le démarrage était assez fastidieux car nous n’étions encore pas encore familiarisé avec l’utilisation de l’*ast* et *tds*. Mais une fois les TP finis, l’ajout des fonctionnalités a pris moins de temps car la structure du code était déjà faite.

Une des difficultés que nous avons rencontrés est la correction des erreurs dans le code, non pas syntaxiques mais sémantiques car il n’est pas évident d’afficher les valeurs des différentes variables.