



**Rapport Projet :
Programmation Fonctionnelle et Traduction Des
Langages**

DELMAIRE-SIZES Clément et PEYREMORTE Romain

Département Sciences du Numérique - Seconde année - Programmation
Fonctionnelle - Traduction Des Langages
2022-2023

Table des matières

1	Introduction du rapport	2
2	Base Travaux Personnels	3
3	Pointeurs	3
3.1	Types	4
3.2	Lexer et parser	4
3.2.1	Lexer	4
3.2.2	Parser	4
3.3	Analyses	4
4	Bloc else optionnel	4
4.1	Jugement de typage	5
4.2	Lexer et parser	5
4.3	Analyses	5
5	Conditionnelle ternaire	5
5.1	Jugement de typage	5
5.2	Lexer et parser	5
5.2.1	Analyses	5
6	Loop à la rust	6
6.1	Jugement de typage	6
6.2	Lexer et parser	6
6.3	TDS	6
6.4	Analyses	6
7	Conclusion	7

Table des figures

1 Introduction du rapport

L'objectif de ce projet est de réaliser un compilateur pour le langage RAT. Les séances de TPs nous ont d'abord amenées à construire le compilateur pour un langage RAT avec des fonctionnalités basiques. Ce compilateur s'appuie dans un premier temps sur une analyse syntaxique du code RAT, puis par une analyse sémantique réalisée à l'aide de différentes passes (passe de résolution des identifiants, passe de typage, passe de placement mémoire et enfin passe de génération de code). Par la suite, le compilateur a été étendu pour prendre en compte de nouvelles constructions : les pointeurs, le bloc else optionnel dans la conditionnelle, la conditionnelle sous la forme d'un opérateur ternaire et les boucles "loop" à la Rust. L'implantation du traitement de ces nouvelles fonctionnalités est détaillée dans la suite du rapport.

2 Base Travaux Personnels

À l'issue des travaux pratiques, nous avons alors la grammaire suivante :

- | | |
|---|------------------------------------|
| 1. $MAIN \rightarrow PROG$ | 18. $TYPE \rightarrow int$ |
| 2. $PROG \rightarrow FUN\ PROG$ | 19. $TYPE \rightarrow rat$ |
| 3. $FUN \rightarrow TYPE\ id\ (DP)\ BLOC$ | 20. $E \rightarrow call\ id\ (CP)$ |
| 4. $PROG \rightarrow id\ BLOC$ | 21. $CP \rightarrow$ |
| 5. $BLOCK \rightarrow \{IS\}$ | 22. $CP \rightarrow E\ CP$ |
| 6. $IS \rightarrow I\ IS_1$ | 23. $E \rightarrow [E / E]$ |
| 7. $IS \rightarrow \Lambda$ | 24. $E \rightarrow num\ E$ |
| 8. $I \rightarrow TYPE\ id = E ;$ | 25. $E \rightarrow denom\ E$ |
| 9. $I \rightarrow id = E ;$ | 26. $E \rightarrow id$ |
| 10. $I \rightarrow const\ id = entier ;$ | 27. $E \rightarrow true$ |
| 11. $I \rightarrow print\ E ;$ | 28. $E \rightarrow false$ |
| 12. $I \rightarrow if\ E\ BLOC_1\ else\ BLOC_2$ | 29. $E \rightarrow entier$ |
| 13. $I \rightarrow while\ E\ BLOC$ | 30. $E \rightarrow (E + E)$ |
| 14. $I \rightarrow return\ E$ | 31. $E \rightarrow (E * E)$ |
| 15. $DP \rightarrow \Lambda$ | 32. $E \rightarrow (E = E)$ |
| 16. $DP \rightarrow TYPE\ id\ DP$ | 33. $E \rightarrow (E < E)$ |
| 17. $TYPE \rightarrow bool$ | 34. $E \rightarrow (E)$ |

Nous avons à ce moment là un compilateur fonctionnel, composé des 4 passes mentionnées précédemment et permettant la compilation d'un langage RAT "basique".

3 Pointeurs

Par rapport aux programmes déjà réalisés, nous commençons ce projet par l'ajout de pointeurs. Les pointeurs sont manipulés en RAT étendu avec une notation proche de celle de C, ce qui ajoute les nouvelles grammaires suivantes à celles déjà traitées :

1. $A \rightarrow id$
2. $A \rightarrow (*\ A)$
3. $TYPE \rightarrow TYPE\ *$
4. $TYPE \rightarrow (TYPE)$
5. $E \rightarrow A$
6. $E \rightarrow null$
7. $E \rightarrow (new\ TYPE)$
8. $E \rightarrow \&\ id$

Nous avons donc un nouvel élément à traiter, A l'affectable, qui fait que nous n'avons plus la grammaire $E \rightarrow id$, et nous remplaçons $I \rightarrow id = E$ par $I \rightarrow A = E$. Nous avons alors un nouveau AST à analyser à chaque passe, l'affectable, qui sera en plus une analyse récursive à cause du déréférencement de la grammaire 2 décrite ci-dessus.

3.1 Types

L'ajout de pointeurs dans l'analyse crée un nouveau type à mettre en place, le **POINTEUR** **of typ** qui permet de traiter la récursivité du pointeur.

Avec l'ajout de nouvelles éléments, nous avons les jugements de typage suivant :

$$(*A) : \frac{\sigma \vdash a : \text{Pointeur}(t)}{\sigma \vdash (*a) : t}$$

$$\text{null} : \frac{}{\sigma \vdash \text{null} : \text{Pointeur}(\text{Undefined})}$$

$$\text{new} : \frac{\sigma \vdash t : \tau}{\sigma \vdash (\text{new } t) : \text{Pointeur}(t)}$$

$$\text{ADRESSE} : \frac{\sigma \vdash \text{id} : t}{\sigma \vdash \&\text{id} : \text{Pointeur}(t)}$$

3.2 Lexer et parser

3.2.1 Lexer

Ces nouvelles grammaires apportent de nouveaux caractères et chaînes de caractères à décoder lors de la lecture du code RAT :

1. *null*
2. *new*
3. & que l'on nommera ADRESSE

3.2.2 Parser

Avec l'ajout d'un nouvel élément dans la grammaire, le parser s'est vu aussi ajouter un nouvel élément analysable, l'affectable, noté *af* dans *parser.mly*. Nous avons alors modifié les instructions pour que l'affectation prenne en compte un affectable et non un identifiant. Nous avons ajouté le pointeur au décodage des types, comme le traitement des parenthèses encadrant un type. Pour l'expression, nous avons mis en place le décodage des nouvelles grammaires pour renvoyer les bonnes **AstSyntax**.

3.3 Analyses

Nous avons commencé les modifications des différents AST au niveau de leur définition (ajout de l'affectable). Nous avons aussi repris les règles pour remplacer des *info_ast * expression* en *affectable * expression* (dans le cas de l'affectation par exemple).

Lors du passage d'identifiant et du codage, pour différencier l'utilisation des affectables dans le cadre d'une affectation ou d'une expression, nous avons utilisé des variables booléennes (*modifier* pour l'identification et *stockage* pour le codage) permettant un contrôle plus rigoureux ou bien appliquer le bon code. En effet, des constantes peuvent être considérées comme des affectables, mais ces dernières ne peuvent être modifiées, comme lorsque l'on lit l'information d'un affectable dans une expression tandis que l'affectation doit la mémoriser.

La gestion des identifiants est sinon comme lors des TP's tandis que le déréférencement est une analyse récursive jusqu'à obtenir l'identifiant de bases. Dans le déréférencement, on a dû faire attention au typage, qu'il soit bien un pointeur.

4 Bloc else optionnel

Le deuxième ajout du projet consiste à pouvoir réaliser la conditionnelle sans le bloc *ELSE*, c'est à dire ajouter la grammaire $I \rightarrow \text{if } E \text{ BLOC}$.

4.1 Jugement de typage

Cet ajout est similaire au conditionnel déjà étudié et implanté. Son jugement de typage est donc similaire :

$$if\ E\ BLOC : \frac{\sigma \vdash E : bool \quad \sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash if\ E\ BLOC : void}$$

4.2 Lexer et parser

Pour faciliter l'implémentation, nous avons juste traduit la conditionnelle sans bloc else en **AstSyntax.instruction.conditionnelle** composé de l'expression, du bloc et d'une liste vide. En effet, le bloc consiste en une liste d'instructions. Donc avoir une liste vide dans le bloc else est comme avoir aucun bloc else car il n'y a aucune instruction.

Nous aurions pu créer une nouvelle instruction mais cela nous aurait obligé à implanter l'analyse de cette instruction et donc à recopier du code.

4.3 Analyses

Par construction de la conditionnelle sans bloc else à partir de la conditionnelle à bloc else fait que l'implémentation est déjà réalisée dans la base des TPs. Le fait d'avoir une liste vide n'impacte en rien la compilation des analyses car l'analyse d'un bloc revient en bref à appliquer l'analyse d'une instruction sur toutes les instructions de la liste.

5 Conditionnelle ternaire

En continuant les conditionnelles, un ajout réalisé durant ce projet est la conditionnelle ternaire qui est une expression conditionnelle. On a donc la grammaire suivant à ajouter :

$$E \rightarrow (E ? E : E)$$

En fonction de la première expression, on obtient la seconde ou la troisième.

5.1 Jugement de typage

Comme pour le conditionnel, nous devons vérifier le type de la première expression :

$$(E ? E : E) : \frac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : t_1 \quad \sigma \vdash E_3 : t_2 \quad t_1 = t_2}{\sigma, \tau_r \vdash (E_1 ? E_2 : E_3) : t_1}$$

5.2 Lexer et parser

Avec cette nouvelle grammaire, viennent de nouveaux caractères : le ? et le : que nous avons donc implémenté en token. L'analyse avec le lexer et le parser permet d'obtenir un **AstSyntax.expression.Ternaire** composé de *expression * expression * expression*.

5.2.1 Analyses

L'analyse de cette expression passe par l'analyse des expressions qui la composent puis la vérification du jugement de typage. En tant qu'expression, aucune analyse n'a été réalisé lors de la passe de placement. Pour ce qui est du codage, nous avons repris la forme utilisée pour coder l'instruction conditionnelle (utilisation de *jumpif*).

6 Loop à la rust

Une dernière structure ajoutée lors de ce projet est la Loop à la rust. Le mot "*loop*" introduit un bloc réalisé en boucle jusqu'à interruption du processus, ou instruction demandant d'arrêter. Avec la boucle vient le *continue* relançant directement la boucle et le *break* arrêtant la boucle. Il est possible d'associer un identifiant à une boucle et ainsi faire en sorte que les *continues* et *breaks* s'effectuent sur une boucle visée.

Cela ajoute donc les grammaires suivantes :

1. $I \rightarrow \text{loop } BLOC$
2. $I \rightarrow id : \text{loop } BLOC$
3. $I \rightarrow \text{beak};$
4. $I \rightarrow \text{beak } id;$
5. $I \rightarrow \text{continue};$
6. $I \rightarrow \text{continue } id;$

6.1 Jugement de typage

Les boucles sont juste composés de blocs, ce qui facilite l'analyse de typage.

$$\text{loop } BLOC : \frac{\sigma, \tau_r \vdash BLOC : \text{void}}{\sigma, \tau_r \vdash \text{loop } BLOC : \text{void}}$$

6.2 Lexer et parser

Les nouvelles grammaires apportent donc de nouveaux tokens *loop*, *continue* et *break*.

6.3 TDS

Pour mieux gérer les boucles et les boucles identifiés, nous avons défini une nouvelle *info* : l'*InfoLoop* de *string* * *string*. Cette dernière information permet à chaque instruction *Loop*, *Break* et *Continue* de savoir quelle boucle ils manipulent. Cette *info* contient l'étiquette de la boucle déterminée lors de la passe d'identification, ainsi que l'identifiant s'il est déterminé (" dans le cas contraire).


Le lexer et le parser traduit les *loop*, les *break* et les *continue* en séparant ceux identifiés et ceux non identifiés. Cette différenciation n'est plus présente après l'analyse des identifiants.

6.4 Analyses

Pour savoir quelle est la dernière boucle instanciée et/ou quelles sont les boucles en cours, nous avons décidé le principe de pile mémorisant les *InfoLoop*. Pour pouvoir appliquer cette solution, nous avons utilisé le module *Stack* d'*Ocaml*. Les principales fonctions utilisées sont :

1. *create* : $unit \rightarrow 'a\ t$ qui renvoie une pile (stack) vide
2. *push* : $a \rightarrow 'a\ t \rightarrow unit$ qui met un élément au sommet de la pile
3. *pop_opt* : $'a\ t \rightarrow a\ option$ qui enlève le sommet de la pile sous forme d'option (*Some* ou *None* si rien dans la pile)
4. *copy* : $'a\ t \rightarrow 'a\ t$ qui crée une copie de la pile

Pour pouvoir utiliser ces fonctions de la façon dont on le voulait, nous avons ajouter des fonctions :

1. *remove_pile* : $'a\ t \rightarrow unit$ qui enlève le sommet sans retour
2. *ajouter_pile* : $info_ast\ t \rightarrow info_ast \rightarrow unit$ qui ajoute à la pile d'*info_ast* une *info_ast* composé d'une *InfoLoop*
3. *id_present_pile* : $info_ast\ t \rightarrow string \rightarrow info_ast\ option$ qui cherche dans la pile si une *info_ast* contenant une *InfoLoop* possède le string comme identifiant et renvoie *Some* de cette *info_ast* (ou *None* si pas présent). Cette fonction permet de trouver dans la pile des boucles présentes, la première *info_ast* définie par cet identifiant.  Le parcours de la pile entraîne la suppression des éléments jusqu'à trouver l'*info_ast* cherché. Il faut donc appliquer cette fonction sur une copie de la pile.

Une autre fonction définie en plus est l'analyse est *getNomLoop* : $unit \rightarrow string$. Cette fonction permet de générer des noms de boucle, "*loopX*" avec *X* généré de telle sorte que chaque appel donne un *X* différent. Cette fonction reprend le code de *getEtiquette* de *code.ml*.

Nous avons implémenté une variable *pileLoop* lors de l'analyse des instructions (et donc des blocs) lors de la passe d'identification. Cette dernière est la pile stockant les *info_ast* des *InfoLoop* en cours. Lorsqu'un nouveau bloc est créé, une copie de la pile actuelle est faite. Cette variable est comme le *tds* mais pour suivre l'évolution des boucles.

Après la passe d'identification, l'analyse s'applique sur des *Loops*, *Breaks* et *Continues* contenant les *InfoLoop* permettant ainsi de déjà connaître les étiquettes lors du codage. Il suffit juste que lors de cette passe, de mettre en place un label *debutetq* et un label *finetq* pour réaliser les sauts nécessaires (fin de boucle, break de boucle, reprise de boucle). Nous aurions pu garder la distinction entre les instructions définies avec identifiants, et celles sans durant toutes les analyses. Mais cela aurait amener à de la répétition de code. Cette solution permet de déjà donné une étiquette aux boucles dès la passe des identifiants et aussi de bien vérifier la bonne utilisation des *break* et *continue* (bien utilisé dans une boucle).

7 Conclusion

Durant ce projet, nous avons pu appliquer le raisonnement appris lors des Travaux Dirigés et lors des Travaux Pratiques de Traduction Des Langages sur de nouveaux éléments de codes à analyser. Nous avons pu aussi manipuler le Lexer et le Parser vu en Théorie des Automates et des Langages et ainsi bien voir leur utilité. Nous avons aussi utilisé le codage en Programmation Fonctionnelle et avons ainsi vu son utilité par rapport à la Programmation Impérative dans ce cas d'utilisation. Ce projet a été intéressant pour observer les différentes étapes de la traduction d'un langage à un autre, et qu'il est important de vérifier les identifiants et les types. Le travail en groupe a été faisable en répartissant le travail.