Rapport Projet Traduction des Langages

Assala ASSELLALOU
Beya HACHICHA

Département Sciences du Numérique - HPC et Big Data
2024-2025

Table des matières

1 Introduction **3**

2 Grammaire du langage RAT étendu **3**

3 Évolution de la structure des AST **4**
3.1 Les pointeurs : 4
3.2 Les variables globales : 4
3.3 Les variables statiques locales : 5

4 Jugement de typage **5**

5 Les pointeurs **6**
5.1 Implémentation 6
5.1.1 Passe de gestion des identifiants 6
5.1.2 Passe de typage 6
5.1.3 Passe de placement mémoire 7
5.1.4 Passe de génèration de code 7
5.2 Remarques générales 8

6 Les variables globales **8**
6.1 Implémentation 8
6.1.1 Passe de gestion des identifiants 8
6.1.2 La passe de typage 8
6.1.3 La passe de placement mémoire 9
6.1.4 La passe de génération de code 9
6.2 Remarques générales 9

7 Les variables statiques locales **9**
7.1 Implémentation 9
7.1.1 Passe de gestion d'identifiants 9
7.1.2 Passe de typage 9
7.1.3 Passe de placement mémoire 10
7.1.4 Passe de génèration de code 10
7.2 Remarques générales 10

8 Conclusion **11**

Table des figures

1 Introduction

Ce projet s'inscrit dans la continuité du travil réalisé en séance de TP.

Durant les séances de TP, nous avons développé un compilateur du langage RAT qui gère les 4 passes fondamentales à la compilation (gestion des identifiants, typage, placement mémoire et génération de code).

Maintenant, le but de ce projet est d'étendre le compilateur du langage RAT réaliser en rajoutant de nouvelles fonctionnalités telles que les pointeurs, les variables globales, les variables statiques locales et les paramètres par défaut.

Dans ce rapport, nous allons tout d'abord présenter la nouvelle grammaire du langage RAT étendu ainsi que l'évolution de la structure des AST. Ensuite, nous donnerons les jugement de typage pour les règles rajoutées ou modifiées dans la nouvelle grammaire pour enfin arriver aux explications des traitements des nouvelles constructions à citer les pointeurs, les variables globales, les variables statiques locales et les paramètres par défaut.

2 Grammaire du langage RAT étendu

1. PROG' → PROG $
2. PROG → VAR⋆ FUN⋆ id BLOC
3. VAR → *static TYPE id = E ;*
4. FUN → TYPE id (DP) BLOC
5. BLOC → { I ⋆ }
6. I → TYPE id = E ;
7. |static TYPE id = E ;
... ~~|id = E ;~~
8. |A = E ;
9. |const id = entier ;
10. |print E ;
11. |if E BLOC else BLOC
12. |while E BLOC
13. |return E ;
14. A → id
15. |(⋆ A)
16. DP → Λ
17. |TYPE id <D>?<,TYPE id <D>?> ⋆
18. D → = E
19. TYPE → bool
20. | int
21. | rat
22. | TYPE ⋆
23. E → id (CP)
24. | [E / E]
25. | num E
26. | denom E
.. ~~|id~~
27. | A
28. | true
29. | false
30. | entier
31. | (E + E)
32. | (E ⋆ E)
33. | (E = E)
34. | (E < E)
35. | (E)
36. | null
37. | (new TYPE)
38. | & id
39. CP → Λ
40. | E (, E) ⋆

Les nouvelles constructions que le langage RAT étendu permet de manipuler :

1. pointeurs
2. variables globales
3. variables statiques locales
4. paramètres par défaut

3 Évolution de la structure des AST

Pour que notre compilateur soit en mesure de traiter toutes les nouvelles modifications portées sur le langae RAT étendu, nous avons du appliquer certain changements à l'AST.

3.1 Les pointeurs :

Lors de l'ajout des pointeurs, nous avons dû :

— Ajouter un nouveau type de variable (typ) : *Pointeur of typ*

— Ajouter un nouveau type *affectable* avec :

- **ASTSyntax** :

  affectable = | *Ident of string* | *Deref of affectable*

- **ASTTds, ASTType, ASTPlacement** :

  affectable = | *Ident of info_ast* | *Deref of affectable*

— Modifier l'instruction Affectation qui est devenue :

*Affectation of affectable * expression*

— Modifier l'expression *Ident of string* qui est devenue :
Affectable of affectable

— Ajouter les expressions suivantes :

- **ASTSyntax** :

  expression = |... | *Null* | *New of typ* | *Adresse of string*

- **ASTTds, ASTType, ASTPlacement** :

  expression = |... | *Null* | *New of typ* | *Adresse of info_ast*

3.2 Les variables globales :

Lors de l'ajout des variables globales, nous avons dû :

— Ajouter un nouveau type *variables* avec :

- **ASTSyntax** :

  variables = | *Static of typ * string * expression*

- **ASTTds** :

  variables = | *Static of typ * info_ast * expression*

- **ASTType, ASTPlacement** :

variables = | *Static of info_ast * expression*

— Modifier la structure générale du programme en rajoutant une liste de variables statiques avant la liste de fonctions.

3.3 Les variables statiques locales :

Lors de l'ajout des variables statiques locales, nous avons dû :

— Ajouter un nouvelle *instruction* **Staticlocal** avec :

• **ASTSyntax** :
instruction = | *...* | *typ * string * expression*

• **ASTTds** :
instruction = | *...* | *Staticlocal of typ * info_ast * expression*

• **ASTType, ASTPlacement** :
instruction = | *...* | *Staticlocal of info_ast * expression*

— Ajouter une liste d'instruction en paramètre de AstPlacement.Programme qui nous sera utile pour la passe de génération de code pour la génération du code TAM correspondant à la declaration et au stockage des variables statiques locales.

• **ASTPlacement** :
*type programme = Programme of variables list * fonction list** ***instruction list*** * bloc*

4 Jugement de typage

- **E → A**

$$\frac{\sigma \vdash \texttt{A} : \tau}{\sigma \vdash \texttt{E} : \tau}$$

- **E → null**

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

- **E →(new TYPE)**

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

- **E → & id**

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

- **A → id**

$$\frac{\texttt{id} : \tau \in \sigma}{\sigma \vdash \texttt{id} : \tau}$$

- **A → (* A)**

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

- **TYPE → TYPE ***

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

- **I →A = E**

$$\frac{\sigma \vdash \texttt{A} : \tau \quad \sigma \vdash \texttt{E} : \tau}{\sigma, \tau_r \vdash \texttt{A = E} : \text{void}, []}$$

5 Les pointeurs

5.1 Implémentation

5.1.1 Passe de gestion des identifiants

Dans la passe de gestion des identifiants on a rajouté :

- La fonction **analyse_tds_affectable** pour traiter :
  - **La *règle 14*** : $A \rightarrow id$:
    On renvoi une erreur si l'identifiant est utilisé mais non déclaré ou s'il s'agit d'un identifiant mal utilisée par exemple affectation sur une fonction.
  - **La *règle 15*** : $A \rightarrow (* A)$:
    On fait un appel récursif de la fonction *analyse_tds_affectable*.
- Dans la fonction **analyse_tds_expression** :
  - **plus de traitement pour :** $E \rightarrow id$:
    C'est un sous cas de la règle 27.
  - **Le traitement de la *règle 27*** : $E \rightarrow A$:
    Appel de la fonction *analyse_tds_affectable* avec son paramètre booléen *en_ecriture* qu'on met à false.
  - **L'ajout de la *règle 36*** : $E \rightarrow null$:
    Pas d'identifiant.
  - **ajout de la *règle 37*** : $E \rightarrow (new\ TYPE)$:
    Pas d'identifiant.
  - **Le traitement de la *règle 38*** : $E \rightarrow \& id$:
    On renvoi une erreur si l'identifiant est utilisé sans être déclaré ou si l'identifiant en question est mal utilisé (n'est pas une variable).
- Dans la fonction **analyse_tds_instruction** :
  - **plus de traitement pour :** $I \rightarrow id = E ;$:
    C'est un sous cas de la règle 8.
  - **traitement de la *règle 8*** : $I \rightarrow A = E ;$:
    Appel des fonctions *analyse_tds_affectable* et *analyse_tds_expression*.

5.1.2 Passe de typage

Dans la passe de typage on a rajouté :

- La fonction **analyse_type_affectable** pour traiter :
  - **La *règle 14*** :
    On vérifie le type de l'identifiant, si c'est une constante le type est un entier, si c'est une variable le type est modifié en utilisant la fonction *modifier_type_variable* et par la suite est renvoyé .
  - **La *règle 15*** :
    On vérifie le type attendu qui doit être un pointeur pour accéder au type pointé sinon on lève une exception.
- Dans la fonction **analyse_type_expression** :
  - **Le traitement de la *règle 27*** :
    Appel de la fonction *analyse_type_affectable* pour obtenir pour obtenir la représentation typée de l'affectable (na) et son type (t), puis retourner l'affectable typé dans l'AST avec son type associé.
  - **Le traitement de la *règle 36*** :
    On retourne un pointeur de type *Undefined* dans l'AST typé qui correspond au cas d'un pointeur nul.

- **traitement de la *règle 37*** :
  Pour allouer dynamiquement un pointeur de type spécifié (t), on retourne le pointeur typé avec le type alloué (t) dans l'AST typé.
- **Le traitement de la *règle 38*** :
  Pour obtenir l'adresse d'une variable, on vérifie que l'identifiant correspond à une variable dans ce cas on met à jour son type avec *modifier_type_variable* et enfin retourne l'adresse typée dans l'AST avec le type de la variable.

— Dans la fonction **analyse_type_instruction** :

- **Le traitement de la *règle 8*** :
  On analyse l'expression avec *analyse_type_expression* qui retourne l'expression typée (ne) et son type (te), et on analyse l'affectable avec *analyse_type_affectable* qui retourne l'affectable typé (na) et son type (ta), ensuite on vérifie que le type de l'expression (te) est compatible avec le type de l'affectable (ta) en utilisant la fonction *est_compatible*.

5.1.3 Passe de placement mémoire

Dans la passe de placement mémoire on garde la même structure du code qu'on a eut en fin des TPs.

5.1.4 Passe de génèration de code

Dans la passe de génèration de code on a rajouté :

— La fonction **analyse_code_affectable_intermediaire** :

- **La *règle 14*** :
  On génère le code pour un identifiant (variable ou constante). Quand il s'agit d'une variable, on traite deux cas selon le paramètre booléen modif. Dans le cas ou la variable est en mode modification ($modif = true$) on utilise *store* pour écrire. Dans le cas contraire, on utilise *load* pour lire. Quand il s'agit d'une constante, on charge directement la valeur avec (*loadl_int*).Si l'identifiant n'est pas une variable ou une constante on lève une erreur.
- **La *règle 15*** :
  On génère le code pour accéder à la valeur pointée par un pointeur (déréférencement). On effectue une analyse récursive du pointeur avec ($modif = false$) pour ne pas modifier le pointeur lui-même. Si le type est un pointeur *(Pointeur t)*, on génère le code en fonction du mode : en modification *(modif = true)*, on produit l'instruction *storei* pour écrire à l'adresse pointée ; sinon, on génère *loadi* pour lire la valeur pointée. Si le type n'est pas un pointeur, on lève une erreur.

— La fonction **analyse_code_affectable** :
On ne conserve que la partie correspondant au code de l'affectable (na).

— Dans la fonction **analyse_code_expression** :

- **Le traitement de la *règle 27*** :
  On générer le code pour accéder à un affectable sans le modifier. Pour cela, on utilise la fonction *analyse_code_affectable* avec ($modif = false$) pour obtenir le code correspondant à la lecture de l'affectable.
- **Le traitement de la *règle 36*** :
  Afin de représenter un pointeur nul dans le code, on charge avec *loadl_int* la valeur -1 pour symboliser un pointeur nul.
- **Le traitement de la *règle 37*** :
  Pour allouer dynamiquement un espace mémoire pour un pointeur du type déterminé, on charge la taille mémoire nécessaire avec (*loadi (getTaille t)*) et on appelle (*Malloc*) pour allouer la mémoire.

- **Le traitement de la *règle 38*** :
  Pour générer le code pour accéder à l'adresse d'une variable. On commence par vérifier que l'identifiant correspond à une variable et on s'intéresse au déplacement (depl) et au registre (reg) associés. Ensuite, on Utilise pour charger l'adresse : (*loada*) en fonction du déplacement (*depl*) et du registre (*reg*) correspondants à la variable qu'on étudie.

— Dans la fonction **analyse_type_instruction** :

- **Le traitement de la *règle 8*** :
  On utilise la fonction *analyse_code_expression* pour générer le code de l'expression (ne) , et *analyse_code_affectable* pour générer le code de l'affectable (na) et ensuite on concatène les deux pour obtenir le code de l'affectation.

5.2 Remarques générales

— **Gestion des constantes :**
Lors du rajout des fonctionnalités relatives aux pointeurs, dans la passe de gestion des identifiants, nous avons été confrontés aux cas où l'identifiant correspondait à une constante. Nous avons donc choisi de garder les constantes dans la TDS et non pas les remplacer directement par leurs valeurs lors du traitement.
Dans l'analyse des affectables, nous avons donc rajouter le traitement des cas où l'info_ast liée à l'identifiant correspondait InfoCons par rapport au traitement de l'ancienne expression.

— **Tests :**
Nous avons rajouté des fichiers tests avec et sans fonctions dans les dossier FichiersRat de gestion_id, type, placement et tam.
Finalement, après quelques modifications des codes des passes, tous les tests en rapport avec les fonctionnalités liées aux pointeurs sont validés.

6 Les variables globales

6.1 Implémentation

6.1.1 Passe de gestion des identifiants

On rajoute au code précédant Dans la passe de gestion des identifiants :

— **Le traitement de la *règle 2*** : ***(PROG → VAR⋆ FUN⋆ id BLOC)*** :
on rajoute l'analyse des variables : (*VAR⋆*) dans l'analyse du programme principale.

— **Le traitement de la *règle 3*** : (***VAR → static TYPE id = E ;***) :
on écrit la fonction **analyse_tds_variables** pour le traitement de l'identifiant.
*** Remarque** :
Ce traitement est équivalent à celui de l'instruction de déclaration qu'on a fait pour la *règle 6* : (*I → TYPE id = E ;*) .

6.1.2 La passe de typage

Pour le **traitement de la règle 3**, on écrit la fonction **analyse_type_variables** dont le traitement de typage est équivalent à celui de l'instruction de déclaration qu'on a fait pour la *règle 6* : on vérifie que le type de l'identifiant et celui de l'expression sont compatibles et si ce n'est pas le cas on lève une erreur.
Pour le **traitement de la règle 2**, on réutilise la fonction **analyse_type_variables** avec un (*List.map*).

6.1.3 La passe de placement mémoire

On place les variables globales dans le registre SB avec un déplacement positif à partir de zéro et donc on doit faire attention à bien placer les variables du main dans sb mais après les variables globales statiques on commence donc le placement des variables du main à partir de 0 + taille des variables globales.

6.1.4 La passe de génération de code

Pour le **traitement de la règle 3**, on écrit la fonction **analyse_code_variables** pour génèrer le code qui correspond à la déclaration de variables globales. On commence par vérifier qu'il s'agit bien d'une variable et dans ce cas on empile la taille de la variable avec (*push (getTaille t)*), puis on utilise analyse_code_expression e pour obtenir le code de l'expression associé à la variable et par la suite on génère le code pour stocker la valeur dans la variable avec (*store (getTaille t) depl reg*).
Pour le **traitement de la règle 2**, on réutilise la fonction **analyse_code_variables** avec un (*List.map*).

6.2 Remarques générales

— **Stockage des variables globales :**
L'analyse des variables globales ressemblait beaucoup aux traitements réalisés dans l'instruction Déclaration. La seule particularité avec les variables globales était lors de la passe placement. En effet, maintenant que les variables statiques globales sont stockées dans la pile par rapport au registre "SB" à partir de zéro, lors du placement des variables globales, il faut faire attention à ne pas "écraser" les variables globales. Pour cela, lors du placement des variables déclarées dans le main, il faut commencer à partir de *taille_variables_globales* et non pas de zéro.

— **Tests :**
Nous avons rajouté des fichiers tests avec et sans fonctions dans les dossier FichiersRat de gestion_id, type, placement et tam.
Finalement, après quelques modifications des codes des passes, tous les tests en rapport avec les fonctionnalités liées aux pointeurs sont validés.

7 Les variables statiques locales

7.1 Implémentation

7.1.1 Passe de gestion d'identifiants

Dans la fonction **analyse_tds_instruction** on a rajoute :

— **Le traitement de la règle 7** : (***I*** $\rightarrow$ ***static TYPE id = E ;***) :
On utilise un *match* sur le paramètre **oia** pour s'assurer qu'on est dans le cadre d'une fonction :
* *None* si l'instruction i est dans le bloc principal : on lève une erreur.
* *Some ia* où ia est l'information associée à la fonction dans laquelle est l'instruction i sinon : on traite les identifiants pour les variables statiques locales comme pour la règle 6.

7.1.2 Passe de typage

Dans la fonction **analyse_type_instruction** le traitement du type des variables statiques locales *AstTds.Staticlocal* est comme pour la règle 6.

7.1.3 Passe de placement mémoire

Pour la passe de placement de mémoire, nous avons réalisés plusieurs fonctions intermédiaires pour gérer les variables statiques locales :

— **analyse_placement_fonction_instruction :** analyse une instruction d'une fonction en distinguant deux cas : -L'instruction est une déclaration de variable statique locale (Staticlocal) : on attribue à cette variable une adresse dans la pile par rapport à "SB" et on retourne aussi sa taille. -Toute autre instruction différente de Staticlocal : on traite l'instruction comme une instruction locale classique en mettant à jour le déplacement dans la pile locale "LB".

— **analyse_placement_fonction_linstruction :** analyse toutes les instructions d'une liste d'instruction donnée et trie les instructions en deux sous listes : les instructions locales et les déclarations de variables statiques locales. On analyse chaque instruction tout en mettant à jour les déplacement par rapport à "LB" et "SB". Une déclaration de variables statique locale modifie le déplacement par rapport à "SB" et une instruction locale modifie le déplacement par rapport à "LB". Cette fonction retourne la liste des instructions locales, la liste des déclarations statiques et les déplacements par rapport à "SB" et "LB".

La fonction analyser retourne un élément de type AstPlacement.Programme contenant, en plus des variables globales, des fonctions et du programme principal, une liste des instructions liées aux déclaration des variables statiques locales. Cette liste sera utilisée lors de la génération de code TAM pour gérer leur déclaration et leur stockage.

7.1.4 Passe de génèration de code

Pour la passe de génération de code, nous générons le code TAM des variables statiques locales à partir de la liste lis grâce à la fonction analyse_code_staticlocal (qui retourne un résultat équivalent à une déclaration).

7.2 Remarques générales

— **Stockage des variables statiques locales :**
Les variables statiques locales, bien qu'elles soient déclarées à l'intérieur d'une fonction, elles ont une durée de vie équivalente à celle du programme. Elles sont placées dans la pile par rapport au registre "SB" et conservent leurs valeurs du précédent appel lors d'appels successifs de la fonction.

— **Tests :**
Nous avons rajouté des fichiers tests avec et sans fonctions dans les dossier FichiersRat de gestion_id, type. Les tests relatifs à la fonctionnalité des variables statiques locales pour les passes de gestions d'identifiants et de typage sont validés. Pour la passe de placement, lorsque nous avons de tester notre code, nos tests renvoyaient une exception de type "Not_found"
Malgré nos multiples tentatives pour identifier et corriger ce problème, nous n'avons pas réussi à résoudre cette erreur.
Cela peut être dû à une erreur lors du traitement des variables statiques locales dans la pile statique "SB", sans parvenir à déterminer précisément la cause de cette erreur.

8 Conclusion

Lors des séances de TP et avec la réalisation de ce projet, nous avons réussi à réaliser un compilateur fonctionnel.
Ce projet nous a permis de développer notre compilateur pour qu'il soit capable de gérer les pointeurs, les variables globales et les variables statiques locales.
Malgré nos efforts, nous avons rencontré certaines difficultés, qui nous ont fait perdre un temps considérable.
Cela a limité notre capacité à intégrer des fonctionnalités supplémentaires, comme la gestion des paramètres par défaut. Grâce à ce projet, nous avons pu pleinement comprendre le fonctionnement d'un compilateur.
Nous avons également bien assimilé les différences entre l'analyse lexicale, syntaxique et sémantique.
Par ailleurs, l'implémentation des différentes passes nous a permis de suivre un cheminement clair des fondamentaux de la compilation, des notions qui nous semblaient auparavant d'une grande complexité.