Projet Traduction des Langages

Émile De Vos
Thomas Berbessou

Département Sciences du Numérique - Deuxième année - Parcours HPC et Big Data
2023-2024

Table des matières

Table des figures

1 Introduction

Le but de ce projet a été d'implémenter des fonctionnalités supplémentaires au langage RAT. Ces modifications sont l'ajout de pointeurs, de l'instruction "goto", de boucles "for" et de tableaux. L'implémentation de ces nouvelles fonctionnalités implique des modifications à toutes les strates de notre compilateur en partant du Lexer jusqu'à l'écriture du code TAM. Nous verrons au cours de ce projet tous les ajouts et modifications nécessaires afin de mettre en place ces nouvelles fonctionnalités.

2 Pointeur

2.1 Introduction

Dans cette partie, nous allons traiter de l'ajout de pointeur dans le langage Rat. Un pointeur est une variable qui amène à une adresse mémoire au lieu d'amener à une valeur.

2.2 Lexer

Pour le lexer, nous avons dû ajouter les mots clefs suivants : "*" qui permet d'accéder à la variable d'un pointeur, "&" qui permet d'accéder à l'adresse d'une variable, "new" qui permet de créer un nouveau pointeur, "null" qui permet de faire pointer un pointeur sur rien.

```
rule token = parse
  (* ignore les sauts de lignes mais les compte quand même *)
| '\n'          { new_line lexbuf; token lexbuf }
  (* ignore les espaces et tabulations *)
| [' ' '\t']    { token lexbuf }
  (* ignore les commentaires *)
| "//"[^'\n']* { token lexbuf }

(* caractères spéciaux de RAT *)
| ","           { VIRG }
| ";"           { PV }
| "{"           { AO }
| "}"           { AF }
| "("           { PO }
| ")"           { PF }
| "="           { EQUAL }
| "["           { CO }
| "]"           { CF }
| "/"           { SLASH }
| "+"           { PLUS }
| "*"           { MULT }
| "<"           { INF }
|"&"            {ESP}
|"new"          {NEW}
|":"            {DP}
|"null"         {NULL}
(* constantes entières *)
| ("-")?['0'-'9']+ as i
                { ENTIER (int_of_string i) }
(* identifiants et mots-clefs *)
| ['a'-'z'](['A'-'Z''a'-'z''0'-'9']|"-"|"_")* as n
                { ident n }

(* fin de lecture *)
| eof           { EOF }
(* entrée non reconnue *)
| _             { error lexbuf }
```

FIGURE 1 – nouveau lexer

2.3 Parser

Dans le parser, la première chose que nous avons faite a était de créer un nouvel élément qui est affectable (A). Cet élément regroupe alors les variables de base qui sont les Ident et les Pointeurs qui sont nommés Deref. Le type Affectable est alors :

$$\begin{array}{l} \text{type affectable} \\ \mid \text{Ident of string} \\ \mid \text{Deref of affectable} \end{array} \tag{1}$$

Nous pouvons voir que nous avons également modifié le parser, le nouveau parser est adapté pour accepter les nouvelles règles suivantes :

$$\begin{array}{ll} TYPE \rightarrow TYPE* & \text{Pour que les pointeurs puissent être des pointeurs} \\ A \rightarrow (*A) & \text{Pour déréférencer une variable} \\ A \rightarrow id & \\ I \rightarrow A \ = \ E & \\ E \rightarrow (new\ TYPE) & \text{Pour créer un nouveau pointeur de type TYPE} \\ E \rightarrow \&id & \text{Pour avoir l'accés à une variable} \\ E \rightarrow null & \text{Le pointeur null} \\ E \rightarrow A & \end{array} \tag{2}$$

Pour ce faire, le type TYPE devient : $type\ typ = Int|Rat|Bool|Undefined|Pointeur\ of\ typ$

```
e :
| n=ID PO lp=separated_list(VIRG,e) PF   {AppelFonction (n,lp)}
| CO e1=e SLASH e2=e CF                  {Binaire(Fraction,e1,e2)}
| TRUE                                   {Booleen true}
| FALSE                                  {Booleen false}
| e=ENTIER                               {Entier e}
| NUM e1=e                               {Unaire(Numerateur,e1)}
| DENOM e1=e                             {Unaire(Denominateur,e1)}
| PO e1=e PLUS e2=e PF                   {Binaire (Plus,e1,e2)}
| PO e1=e MULT e2=e PF                   {Binaire (Mult,e1,e2)}
| PO e1=e EQUAL e2=e PF                  {Binaire (Equ,e1,e2)}
| PO e1=e INF e2=e PF                    {Binaire (Inf,e1,e2)}
| PO exp=e PF                            {exp}
| NULL                                   {Null}
|a=affec                                 {Affectable a}
|PO NEW t=typ PF                         {New t}
|ESP n=ID                                {Adresse n}
```

FIGURE 2 – nouveau parser d'expressions

Après cette étape, l'Ast d'un affectable est : type affectable = Deref of affectable | Ident of string. Un affectable n'est finalement défini que par son nom.

```
type affectable = Deref of affectable
                | Ident of string
```

FIGURE 3 – Ast d'un affectable après le parser

2.4 passeTdsRat

Pour la passe TDS, il fallait d'abord ajouter toutes les nouvelles expressions que nous avons ajoutées précédemment, à savoir Null, Adresse et New.

```
let rec analyse_tds_expression tds e =
  match e with
  | AstSyntax.Null -> AstTds.Null
  | AstSyntax.New t -> AstTds.New t
  | AstSyntax.Affectable a -> AstTds.Affectable(analyse_tds_affectable tds false a)
  | AstSyntax.Adresse n -> (match chercherGlobalement tds n with
                            |None -> raise (IdentifiantNonDeclare n)
                            |Some i -> (match info_ast_to_info i with
                                        |InfoVar _ -> AstTds.Adresse i
                                        |_ -> raise (MauvaiseUtilisationIdentifiant n)))
  | AstSyntax.Entier(n) -> AstTds.Entier n
  | AstSyntax.Booleen (b) -> AstTds.Booleen b
  | AstSyntax.Binaire (b, e1, e2) -> AstTds.Binaire(b,analyse_tds_expression tds e1,analyse_tds_expression tds e2)
  | AstSyntax.Unaire (b, e) -> AstTds.Unaire(b, analyse_tds_expression tds e)
  | AstSyntax.AppelFonction(n, l) ->
    match chercherGlobalement tds n with
    | None -> raise (IdentifiantNonDeclare n)
    | Some i -> (match info_ast_to_info i with
                 | InfoFun (s, t, tl) -> AstTds.AppelFonction(i,(List.map (analyse_tds_expression tds) l))   (* afficher_locale tdsbloc ; *) (* décommenter pour afficher la table locale *)
                 |_ -> raise (MauvaiseUtilisationIdentifiant n))
```

FIGURE 4 – analyse_Tds_Expression

```
let rec analyse_tds_instruction tdse tds oia i =
  match i with
  |AstSyntax.Goto(n) -> (match chercherGlobalement tdse n with
                         |None -> raise (MauvaiseUtilisationIdentifiant n) (*SI ethiquetet non déclarer -> erreur*)
                         |Some i -> AstTds.Goto(i))

  |AstSyntax.Ethiq(n) -> (match chercherGlobalement tdse n with
                          |None -> raise (MauvaiseUtilisationIdentifiant n) (*Impossible car l'erreur serait déjà lever dans get_ethiquette*)
                          |Some i -> AstTds.Ethiq(i))

  | AstSyntax.Declaration (t, n, e) ->
      begin
        match chercherLocalement tds n with
        | None ->
            (* L'identifiant n'est pas trouvé dans la tds locale,
            il n'a donc pas été déclaré dans le bloc courant *)
            (* Vérification de la bonne utilisation des identifiants dans l'expression *)
            (* et obtention de l'expression transformée *)
            let ne = analyse_tds_expression tds e in
            (* Création de l'information associée à l'identfiant *)
            let info = InfoVar (n,Undefined, 0, "") in
            (* Création du pointeur sur l'information *)
            let ia = info_to_info_ast info in
            (* Ajout de l'information (pointeur) dans la tds *)
            ajouter tds n ia;
            (* Renvoie de la nouvelle déclaration où le nom a été remplacé par l'information
            et l'expression remplacée par l'expression issue de l'analyse *)
            AstTds.Declaration (t, ia, ne)
        | Some _ ->
            (* L'identifiant est trouvé dans la tds locale,
            il a donc déjà été déclaré dans le bloc courant *)
            raise (DoubleDeclaration n)
      end
  | AstSyntax.Affectation (a,e) ->
    let na = analyse_tds_affectable tds true a in
    let ne = analyse_tds_expression tds e in AstTds.Affectation(na, ne)
```

FIGURE 5 – analyse_Tds_Instruction

Nous avons ensuite traité le cas où nous avions un affectable. Pour le traitement des affectables, il faut différencier deux cas. Soit affectable est une expression, dans ce cas, il est à droite d'un signe "=" et on fait alors appel à la fonction "analyse_tds_affectable" avec un booleen a false pour dire que l'affectable ne doit pas être modifié.

Dans le cas où l'affectable est dans une affectation, appelle dans ce cas la fonction analyse affectable avec le booleen à true pour dire que l'affectable peut-être modifié. Ceci est important afin de renvoyer des erreurs dans le cas par exemple où on essayerait de modifier les valeurs d'une constante.

Les opérations faites dans la fonction "analyse_tds_affectable" sont similaires à ce que nous faisions avant cette modification, on vérifie que la variable est bien définie avant de s'en servir

en regardant si elle est dans la TDS.

Après cette passe, l'Ast d'un affectable n'est plus le nom de l'expression en string, mais ses infos associées

```
(*type affecable après la passe des identifiant, après avoir remplacer les string par des infos*)
type affectable =
    | Ident of Tds.info_ast
    | Deref of affectable
```

FIGURE 6 – Ast d'un affectable après la passe Tds

2.5 PasseTypeRat

Les jugements de typage pour les pointeurs sont les suivants :

$$\frac{\sigma \longrightarrow a : Pointeur}{\sigma \vdash *a : \tau} \qquad \frac{\sigma \vdash t : \tau}{\sigma \vdash t* : Pointeur(\tau)}$$
$$\frac{\sigma \vdash a : \tau}{\sigma \vdash \&a : Pointeur(\tau)} \qquad \sigma \vdash null : Pointeur(Undefined) \tag{3}$$

En pratique, on a donc fait une fonction "analyse_type_affectable". On a également ajouté les traitements des expressions null, adresse et new.

```
(*analyse du type des affectable, globalement il ajoute le type à l'info et vérifie la cohérence des types*)
let rec analyse_type_affectable a =
  match a with
    |AstTds.Ident(i) -> (match info_ast_to_info i with
                        | InfoVar(_, t, _, _ )-> (AstType.Ident(i), t)
                        |InfoConst _ -> (AstType.Ident(i), Int)
                        |_-> failwith "probleme info type")
    |AstTds.Deref(b)-> (match analyse_type_affectable b with
                        |(nom, Pointeur t) ->  (AstType.Deref(nom,t), t)
                        |_ -> failwith("IMpossible, ni pointeur ni variable"))
```

FIGURE 7 – analyse_type_affectable

Après cette passe, tout affectable se voit attribuer un type, que ce soit un identifiant ou un Deref.

```
type affectable =
    | Ident of Tds.info_ast
    | Deref of affectable*typ
```

FIGURE 8 – Ast d'un affectable après la passe de typage

2.6 PassePlacementRat

Pour la passe de placement, nous n'avons fait aucune modifications.

2.7 PasseRataTam

Lors de la génération de code TAM, nous regardons de quel type est l'affectable, si l'affectable est de type Ident, on regarde s'il s'agit d'une constante ou non. Si c'est une constante, on la lit. S'il s'agit d'une variable, alors si écriture est à vrai, l'affectable est à gauche de l'égalité, on enregistre donc la nouvelle valeur de la variable. Au contraire, si écriture est à false, alors la variable est à droite de l'égalité et on lit sa valeur. Enfin, si l'affectable est de type Deref, on applique "analyse_affectable" écriture sur ce que pointe l'affectable.

```
let rec analyse_affectable a ecriture =
  match a with
  |AstType.Ident(i) -> (match info_ast_to_info i with
                        |InfoConst(_,v) -> loadl_int v
                        |InfoVar(_, t, dep, reg) -> if ecriture then store (getTaille t) dep reg else load (getTaille t) dep reg
                        |_ -> failwith "OCAML CRIE")
  |AstType.Deref(ab, t) -> if ecriture then (analyse_affectable ab false) ^ (storei (getTaille t)) else (analyse_affectable ab false) ^ loadi (getTaille t)
```

FIGURE 9 – analyse_affectable

3 GoTo

3.1 Introduction

L'instruction Goto est une instruction qui permet de passer directement à une autre partie du code sans faire les instructions entre ou d'en répéter un morceau. Pour cela, on définit une étiquette quelque part dans le code, qu'on appelle par un string et, pour se rendre à cette étiquette, ou fait l'instruction goto étiquette. L'implémentation de Goto était compliqué lors de la passe TDS, car il y a de nombreuse contrainte, notamment le fait qu'une étiquette puisse avoir le même nom qu'une variable ou que deux étiquettes puissent avoir le même nom si elles sont dans le programme principal est dans une fonction.

3.2 Lexer

Pour le lexer, nous avons dû ajouter le mot clé "goto" qui permet de définir un saut.

```
let ident =
  let kws = Hashtbl.create 16 in
  List.iter (fun (kw, token) -> Hashtbl.add kws kw token)
    [
      "const",    CONST;
      "print",    PRINT;
      "if",       IF;
      "else",     ELSE;
      "while",    WHILE;
      "bool",     BOOL;
      "int",      INT;
      "rat",      RAT;
      "num",      NUM;
      "denom",    DENOM;
      "true",     TRUE;
      "false",    FALSE;
      "return",   RETURN;
      "for",      FOR;
      "goto",     GOTO
    ];
```

FIGURE 10 – lexer pour les nouvelles instructions

3.3 Parser

Dans le parser, il a fallu rajouter deux règles de production aux instructions :

$$\begin{array}{ll} - & I \to goto\ id; \\ - & I \to id : \end{array} \tag{4}$$

Après cette étape, l'Ast d'un goto est "Goto of string". Un goto n'est finalement défini que par le nom de l'étiquette vers laquelle il amène.

```
i :
| t=typ n=ID EQUAL e1=e PV          {Declaration (t,n,e1)}
| a=affec EQUAL e1=e PV             {Affectation (a,e1)}
| CONST n=ID EQUAL e=ENTIER PV      {Constante (n,e)}
| PRINT e1=e PV                     {Affichage (e1)}
| IF exp=e li1=bloc ELSE li2=bloc   {Conditionnelle (exp,li1,li2)}
| WHILE exp=e li=bloc               {TantQue (exp,li)}
| RETURN exp=e PV                   {Retour (exp)}
| FOR PO t=typ n=ID EQUAL e1=e PV e2=e PV  n2=ID EQUAL e3=e PF lb=bloc {Pour (t,n,n2,e1,e2,e3,lb)}
| GOTO n=ID PV                       {Goto(n)}
| n=ID DP                            {Ethiq(n)}
```

FIGURE 11 – nouveau parser d'instructions

```
type bloc = instruction list
 and instruction =
   | Goto of Tds.info_ast
```

FIGURE 12 – Ast d'un goto après le parser

3.4 passeTdsRat

Il s'agit de la passe la plus dure pour l'implémentation du "goto". Le problème vient du fait que l'étiquette d'un "goto" peut avoir le même nom qu'une variable ou qu'une fonction. De plus, l'étiquette peut se trouver à la fois avant ou après le "goto" dans le code.

Pour régler ce problème, nous avons créé une nouvelle Tds spécialement pour les étiquettes (tdsEtiquette). Au début de la passe, nous parcourons le code afin d'enregistrer toutes les étiquettes dans tdsEtiquette. De plus, getEtiquette permet de contrôler l'unicité des étiquettes. ELle enregistre également le contexte, ce qui nous permettra lors de la passe de tam de différencier les étiquettes, en fonction de si elles sont dans une fonction ou dans le main.

```
let analyser (AstSyntax.Programme (fonctions,prog)) =
  let tds = creerTDSMere () in
  let tdse= creerTDSMere () in let tdseEthiquette = get_ethiquette prog "main" tdse in
  let nf = List.map (analyse_tds_fonction tds) fonctions in
 (* List.iter print_f nf;*)
  let nb = analyse_tds_bloc tdseEthiquette tds None prog  in
  AstTds.Programme (nf,nb)
```

FIGURE 13 – Ast d'un goto après le parser

```
let rec get_ethiquette li contexte tdse=
  match li with
  |[] -> tdse
  |AstSyntax.Ethiq(n)::q -> (match (chercherGlobalement tdse n) with
                            |None ->  let i = InfoEthiq(n, contexte) in ajouter tdse n (info_to_info_ast i); get_ethiquette q contexte tdse
                            |Some _ -> raise (MauvaiseUtilisationIdentifiant n))  (*ON doit visiter chaque bloc, dont ceux des conditionelle/BOucle*)
  |AstSyntax.Conditionnelle(_,b1,b2)::q -> get_ethiquette (b1@b2@q) contexte tdse
  |AstSyntax.TantQue(_,b)::q -> get_ethiquette(b@q) contexte tdse
  |_::q -> get_ethiquette q contexte tdse
```

FIGURE 14 – Ast d'un goto après le parserb

Une fois tdsEtiquette créé et contenant toutes les étiquettes, on parcourt le code comme avant

```
let rec analyse_tds_instruction tdse tds oia i =
  match i with
  |AstSyntax.Goto(n) -> (match chercherGlobalement tdse n with
                         |None -> raise (MauvaiseUtilisationIdentifiant n) (*SI ethiquetet non déclarer -> erreur*)
                         |Some i -> AstTds.Goto(i))

  |AstSyntax.Ethiq(n) -> (match chercherGlobalement tdse n with
                          |None -> raise (MauvaiseUtilisationIdentifiant n) (*Impossible car l'erreur serait déjà lever dans get_ethiquette*)
                          |Some i -> AstTds.Ethiq(i))

  | AstSyntax.Declaration (t, n, e) ->
      begin
        match chercherLocalement tds n with
        | None ->
            (* L'identifiant n'est pas trouvé dans la tds locale,
            il n'a donc pas été déclaré dans le bloc courant *)
            (* Vérification de la bonne utilisation des identifiants dans l'expression *)
            (* et obtention de l'expression transformée *)
            let ne = analyse_tds_expression tds e in
            (* Création de l'information associée à l'identfiant *)
            let info = InfoVar (n,Undefined, 0, "") in
            (* Création du pointeur sur l'information *)
            let ia = info_to_info_ast info in
            (* Ajout de l'information (pointeur) dans la tds *)
            ajouter tds n ia;
            (* Renvoie de la nouvelle déclaration où le nom a été remplacé par l'information
            et l'expression remplacée par l'expression issue de l'analyse *)
            AstTds.Declaration (t, ia, ne)
        | Some _ ->
            (* L'identifiant est trouvé dans la tds locale,
            il a donc déjà été déclaré dans le bloc courant *)
            raise (DoubleDeclaration n)
      end
  | AstSyntax.Affectation (a,e) ->
    let na = analyse_tds_affectable tds true a in
    let ne = analyse_tds_expression tds e in AstTds.Affectation(na, ne)
```

FIGURE 15 – analyse_Tds_Instruction

Nous pouvons voir sur le code ci-dessus que lorsqu'une instruction "Goto" est détectée, on regarde dans tdsEtiquette si cette étiquette est présente. Si elle l'est, on crée une AstTds. Goto. Sinon, on lève l'exception "MauvaiseUtilisationIdentifiant". De même pour une instruction étiquette, on remplace l'étiquette par son info.Etiq qui contient ses informations.

3.5 PasseTypeRat

Le jugement de typage pour le go to est :

$$\frac{\sigma \vdash id : \text{string}}{\sigma \vdash \texttt{goto id;} : \text{void}, []} \tag{5}$$

Il n'y a rien de particulier à faire dans cette passe pour le 'Goto" et les étiquettes, cette instruction n'ayant ni type n'y appel.

3.6 PassePlacementRat

Les intructions 'Goto" et les étiquettes ne sont pas sauvegardées sur l'ordinateur autrement que pou le code lui-même. Ainsi une étiquette et l'instruction "Goto" aura un placement mémoir de zéro. Pour cette raison, la passe "passePlacementRat" n'a pas de changement majeure.

3.7 PasseRataTam

Pour traduire une instruction "Goto" et une étiquette du langage Rat au langage TAM, il suffit d'effectuer les transformations suivantes :

$$\begin{array}{ll} - & goto\ etiq;\ \rightarrow\ JUMP\ etiq \\ - & etiq :\ \rightarrow\ etiq \end{array} \tag{6}$$

Il est important de souligner que ici, comme on a stocké le contexte, c'est-à-dire ou a été trouvé l'étiquette, on peut créer des étiquettes avec des noms différent, en appelant par exemple l'étiquette etiqmain si elle était trouvée dans le main ou étiqf si elle se trouvait dans f.

4 Boucle for

4.1 Introduction

Une boucle for est une instruction permettant de répéter un certain nombre de fois un bloc du code. Cette instruction utilise un indice qui va varier au cours des différentes répétitions du bloc.

4.2 Lexer

Pour le lexer, nous avons dû rajouter le mot "for" à notre langage. Ce mot sert à définir une boucle "for" et de la différencier d'une autre boucle par exemple.

```
let ident =
  let kws = Hashtbl.create 16 in
  List.iter (fun (kw, token) -> Hashtbl.add kws kw token)
    [
      "const",    CONST;
      "print",    PRINT;
      "if",       IF;
      "else",     ELSE;
      "while",    WHILE;
      "bool",     BOOL;
      "int",      INT;
      "rat",      RAT;
      "num",      NUM;
      "denom",    DENOM;
      "true",     TRUE;
      "false",    FALSE;
      "return",   RETURN;
      "for",      FOR;
      "goto",     GOTO
    ];
```

FIGURE 16 – lexer pour la boucle for

4.3 Parser

Dans le parser, nous avons rajouté une règle de production aux instructions :

$$- \quad I \rightarrow for(int\ id\ =\ E;\ E;\ id\ =\ E)BLOC \tag{7}$$

```
i :
| t=typ n=ID EQUAL e1=e PV          {Declaration (t,n,e1)}
| a=affec EQUAL e1=e PV             {Affectation (a,e1)}
| CONST n=ID EQUAL e=ENTIER PV      {Constante (n,e)}
| PRINT e1=e PV                     {Affichage (e1)}
| IF exp=e li1=bloc ELSE li2=bloc   {Conditionnelle (exp,li1,li2)}
| WHILE exp=e li=bloc               {TantQue (exp,li)}
| RETURN exp=e PV                   {Retour (exp)}
| FOR PO t=typ n=ID EQUAL e1=e PV e2=e PV  n2=ID EQUAL e3=e PF lb=bloc {Pour (t,n,n2,e1,e2,e3,lb)}
| GOTO n=ID PV                       {Goto(n)}
| n=ID DP                           {Ethiq(n)}
```

FIGURE 17 – parser pour la boucle for

4.4 passeTdsRat

Da ns le cas de l'analyse d'une instruction de boucle for, on remarque qu'une boucle "for" est composée de plusieurs éléments :

$$\begin{aligned} &- \quad \text{une déclaration : int id = E1} \\ &- \quad \text{une condition : E2} \\ &- \quad \text{une afféctation : id = E3} \end{aligned} \tag{8}$$

On analyse alors chaque élément de la boucle "for"

```
|AstSyntax.Pour(t, n1, n2, e1, e2, e3, b) -> if not(n1=n2) then raise (MauvaiseUtilisationIdentifiant n2) (* Dans ce cas, on a utiliser une variable différente dans la condition d'arret de la boucle que celle avec la quelle on à définit la boucle*)
                                            else match chercherGlobalement tds n1 with (* il serait surement necessaire de verifier que la condition d'arret comprend bien le paramètre n1, à faire éventuellement plus tard *)
                                            |None -> let info = InfoVar (n1,Undefined, 0, "") in
                                            let infop = info_to_info_ast info in
                                            (* Ajout de l'information (pointeur) dans la tds *)
                                            ajouter tds n1 infop;
                                            AstTds.Pour(t,infop, analyse_tds_expression tds e1,analyse_tds_expression tds e2,analyse_tds_expression tds e3, analyse_tds_bloc tdse tds oia b)
                                            |Some _ -> raise (DoubleDeclaration n1)
```

FIGURE 18 – analyse de la tds pour la boucle for

4.5 PasseTypeRat

Les jugements de typages de la boucle for sont :

$$- \quad \frac{\sigma \vdash id{:}int \quad \sigma \vdash E1{:}int \quad \sigma \vdash E2{:}bool \quad \sigma \vdash E3{:}int \quad (id,int){::}\sigma,\tau_r \vdash BLOC{:}void,\ []}{\sigma,\tau_r \vdash for(int\ id\ =\ E1;\ E2;\ id\ =\ E3)\ BLOC\ :\ void,\ []} \tag{9}$$

Pendant la passe de typage, on rajoute le cas où une instruction est une boucle "for". Dans ce cas, on analyse chaque élément de la boucle : On vérifie que l'expression E1 et E3 son bien du type int et que l'expression E2 est bien du type bool.

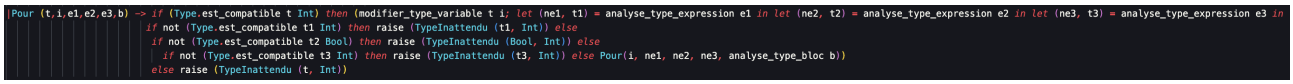

FIGURE 19 – analyse du typage pour la boucle for

Après cette passe, l'Ast de la boucle "for" contient les infos de chaque expression et indice de la boucle.

4.6 PassePlacementRat

Lors de la passe, nous avons rajouté l'analyse d'une boucle dans "analyse_placement_instruction". Lors de cette passe, on enregistre l'indice de la boucle et on analyse le bloc.

Après cette passe, on enregistre l'Ast de l'indice après le placement dans l'Ast de la boucle.

4.7 PasseRataTam

Lors de la traduction de langage Rat à Tam, on évalue l'expression E2 puis le bloc. On rajoute deux JUMP pour rajouter ce phénomène de boucle.

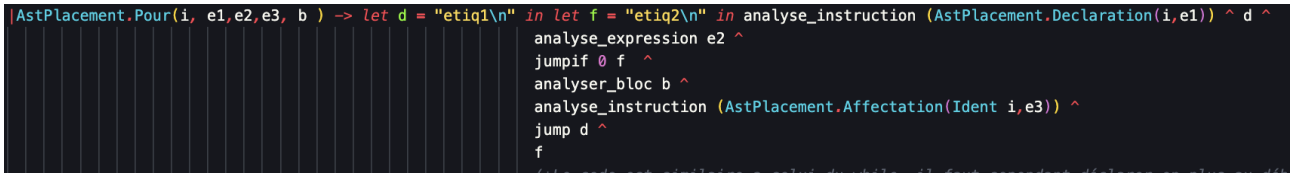

FIGURE 20 – traduction de la boucle for

5 Tableau

5.1 Introduction

Le langage étendu doit pouvoir manipuler des tableaux. Ces tableaux doivent pouvoir être initialisés, accédé en lecture ou en écriture. Nous n'avons pas réussi à mettre en place les tableau dans notre langage étendu, par complexité et par manque de temps.

5.2 Lexer

Il n'y a pas besoin de modifier le lexer pour ajouter les tableaux, Tous les mots et l'ajout du type Affectable ont été faits pour l'ajout des pointeurs.

5.3 Parser

Pour le parser, il faut rajouter les règles de production suivantes aux instructions :

$$\begin{aligned} &- \quad A \rightarrow (\ A[\ E\ ])\\ &- \quad TYPE \rightarrow TYPE\ []\\ &- \quad E \rightarrow (new\ TYPE\ [])\\ &- \quad E \rightarrow CP \end{aligned} \tag{10}$$

5.4 jugement de typage

Les jugements de typages des tableaux sont :

$$
\begin{array}{ll}
- & \dfrac{\sigma \vdash A : \mathrm{Tab}(\tau) \quad \sigma \vdash E : \mathrm{int}}{\sigma \vdash (A[E]) : \tau} \\
- & \dfrac{\sigma \vdash \mathrm{TYPE} : \tau}{\sigma \vdash \mathrm{TYPE}[] : \texttt{TAB}(\tau)} \\
- & \dfrac{\sigma \vdash E : \mathrm{int}}{\sigma \vdash \text{new TYPE}[E] : \mathrm{TYPE}[]} \\
- & \dfrac{\sigma \vdash CP : \mathrm{Liste}(\tau)}{\sigma \vdash \{CP\} : \mathrm{Tab}[\tau]}
\end{array}
\tag{11}
$$

6 Conclusion

En conclusion, nous pouvons dire que ce projet que les différents attendus de ce projet ont posé des problèmes différents lors de leur implémentation. En effet, des fonctionnalités telles que goto, n'ont pas posés de grands problèmes dans les passes une fois inscrites dans la tds grâce à la proximité entre le "goto" de notre langage avec le "jump" de Tam. Le plus gros enjeu du goto a été la création d'une nouvelle tds afin de gérer le nom des étiquettes. Les pointeurs ont nécessité plus de réflexion lors de l'analyse lexicale et syntaxique avec le besoin de créer le type Affectable notamment. De plus, les pointeurs ont nécessité des modifications dans toutes les passes de la traduction, avec le besoin de savoir si un pointeur était accédé en lecture ou en écriture via un booléen par exemple. Enfin, la boucle "for" n'a pas nécessité énormément de travail

Cependant, l'ajout de tableaux dans notre langage impose des modifications dans toutes les passes comme les pointeurs. La charge de travail et la complexité de la mise en œuvre de ses fonctionnalités nous ont empêché d'implémenter les tableaux.