Rapport du projet de programmation fonctionnelle et de traduction des langages

Roméo Roques, Arthur Picard

1 Introduction

Le projet de programmation fonctionnelle et de traduction des langages vise à étendre le compilateur du langage RAT, réalisé lors des travaux pratiques de traduction des langages. L'objectif principal est d'intégrer de nouvelles constructions au compilateur, notamment les pointeurs, les tableaux, les boucles "for", et les instructions "goto". Ces extensions augmenteront la capacité du compilateur à traiter des structures plus complexes, élargissant ainsi son domaine d'application.

La mise en œuvre du compilateur se fera en utilisant le langage OCaml, en respectant les principes de la programmation fonctionnelle étudiés tout au long du semestre.

Dans la suite de ce rapport, nous examinerons en détail les différentes étapes de l'évolution du compilateur pour intégrer les nouvelles fonctionnalités, notamment les justifications de choix de conception, la modification des types, le traitement des pointeurs, des tableaux, des boucles "for", des instructions "goto", et enfin, nous évaluerons ces choix à la lumière des principes de la programmation fonctionnelle.

Table des matières

2 Types

2.1 Affectables de AstSyntax

Dans le cadre du code **OCaml** de l'**AstSyntax**, une évolution significative a été réalisée par l'ajout du type **affectable**. Cette modification vise à intégrer la gestion des affectations de pointeurs, de tableaux et d'identifiants au sein de notre **AstSyntax**. La structure élaborée est explicitée dans le code ci-dessous :

```
(* Affectable apres ajout pointeurs, tableau et identifiants *)
type affectable =
  (* Pointeur vers ID et expression de l'indice *)
  | TabInd of affectable * expression
  (* Identifiant represente par son nom *)
  | Ident of string
  (* De-eferencement d'un identifiant *)
  | Deref of affectable
```

Listing 1 – Ajout dans le typage de l'Ast

Cette structure **affectable** est composée de trois variantes principales :

1. **TabInd** : Représente un affectable avec une expression d'indice associée, permettant la manipulation de tableaux.
2. **Ident** : Représente un identifiant par son nom, préservant la gestion établie lors des travaux pratiques et travaux dirigés à l'aide d'un **string**. Cela offre la possibilité de vérifier si une variable a été préalablement définie à travers la fonction **chercherGlobalement**.
3. **Deref** : Représente le dé-référencement d'un identifiant, permettant ainsi la gestion de pointeurs multiples, notamment des pointeurs vers d'autres pointeurs.

.

2.2 Expressions de AstSyntax

La liste ci-dessous présente les nouveaux ajouts dans le typage de l'AST pour les expressions, comme illustré dans le code suivant :

```
(* Expression de pointeur null *)
| Null
(* Expression de reservation memoire pour la valeur d'un pointeur *)
| New of typ
(* Expression de reservation memoire de tableau *)
| NewTab of typ * expression (* Type du tableau et expression de taille*)
(* Adresse d'une variable *)
| Adresse of string
```

Listing 2 – Ajout dans le typage de l'Ast dans les expressions

— `Null` : Représente une expression de pointeur nul. Cette expression est utilisée pour indiquer l'absence de référence vers un objet.

— `New of typ` : Indique une expression de réservation en mémoire pour la valeur d'un pointeur. L'opérateur `New` est associé à un type (`typ`) spécifiant le type du pointeur à allouer en mémoire.

— `NewTab of typ * expression` : Représente une expression de réservation de mémoire pour un tableau. L'opérateur `NewTab` est associé à un type de tableau (`typ`) et une expression (`expression`) indiquant la taille du tableau à allouer en mémoire.

— `Adresse of string` : Correspond à l'adresse d'une variable. Cette expression permet d'obtenir l'adresse mémoire d'une variable spécifiée par son nom (`string`).

2.3 Instructions de AstSyntax

Les instructions de `AstSyntax` ont été étendues pour inclure de nouveaux éléments, comme illustré dans le code ci dessous :

```
(* Boucle For *)
| For of instruction * expression * affectable * expression * bloc
(* Goto *)
| Goto of string
(* Label *)
| Label of string
```

Listing 3 – Ajout dans le typage de l'Ast dans les instructions

— `For of instruction * expression * affectable * expression * bloc` : Représente une boucle `For` dans le langage. Cette instruction prend en compte une initialisation (`instruction`), une condition d'arrêt (`expression`), une affectation après chaque itération (`affectable`), une expression de modification à chaque itération (`expression`), et le bloc d'instructions (`bloc`) à exécuter à chaque itération de la boucle.

— `Goto of string` : Représente une instruction de saut inconditionnel. L'instruction `Goto` permet de transférer le contrôle vers un label spécifié par son nom (`string`).

— `Label of string` : Représente un label dans le langage. Les labels sont utilisés en conjonction avec les instructions `Goto` pour spécifier des points de saut dans le code. L'instruction `Label` indique le début d'une section de code identifiée par son nom (`string`).

2.4 Affectables de AstTds

Des modifications de l'affectable dans l'**AstTds** visent à étendre la prise en charge des pointeurs dans le contexte des tables des symboles (**AstTds**). La structure modifiée est présentée dans le code ci-dessous (24) :

```
type affectable =
  | TabInd of affectable * expression  (*pointeur vers ID et expression de
     l'indice*)
  (* Identifiant represente par son nom *)
  | Ident of Tds.info_ast
  (* Dereferencement d'un identifiant *)
  | Deref of affectable
```

Listing 4 – Ajout dans le typage de l'Ast d'affectable

Seule la gestion des identifiants est modifiée. En effet, **Ident** représente un identifiant par une structure **info_ast**, préservant la gestion des informations associées à l'identifiant dans la table des symboles.

2.5 Expressions de AstTds

Les modifications des expressions dans **AstTds** visent à enrichir la représentation interne des expressions dans le contexte de la table des symboles. Les éléments ajoutés sont présentés dans le code ci-dessous (24) :

```
  | Null
  | New of typ
  | NewTab of typ * expression  (* Type du tableau et expression de taille *)
  | Adresse of Tds.info_ast  (* On recupere l'adresse via le deplacement *)
```

Listing 5 – Ajout dans le typage de l'Ast dans les expressions

Seule l'expression concernant les adresse a été modifié. On représente une expression d'adresse via **Tds.info_ast**, permettant de récupérer l'adresse via le déplacement dans la mémoire.

2.6 Instructions de AstTds

Les modifications dans les instructions visent à enrichir la représentation interne des instructions dans le contexte de la table des symboles (**AstTds**). Les éléments ajoutés sont présentés dans le code ci-dessous (24) :

```
  | For of instruction * expression * affectable * expression * bloc
  | Goto of Tds.info_ast
  | Label of Tds.info_ast
```

Listing 6 – Ajout dans le typage de l'Ast dans les instructions

Les instructions ont été étendues avec les variantes suivantes :

— **For** : La boucle **For** reste inchangée.

— **Goto** : Représente une instruction `goto` avec une référence à une information dans la table des symboles (**Tds.info_ast**).

— **Label** : Représente une instruction d'étiquette (`label`) avec une référence à une information dans la table des symboles (**Tds.info_ast**).

2.7 Affectable de AstType

```
(* Affectable apres ajout pointeurs *)
  type affectable =
    | TabInd of affectable * expression * typ  (*pointeur vers ID, expression
       de l'indice et type du tableau*)
    (* Identifiant repr sente par son nom *)
    | Ident of Tds.info_ast
    (* Dereferencement d'un identifiant *)
    | Deref of affectable * typ
```

Listing 7 – Ajout dans le typage de l'Ast d'affectable

Ici, on rajoute le type des tableaux et des pointeurs afin de vérifier la présence d'anomalies de typage dans un tableau ou un pointeur. Par exemple, la présence d'un **rat** dans un tableau de tableau de **int**.

2.8 Expressionset instructions de AstType

Les expressions rajoutées New, NewTab et Adresse de AstType sont les mêmes que AstTds car le type avait déjà été rajouté dans l'AstTds. De même, les instructions restent elles aussi inchangées.

3 Jugement de typage

3.1 Jugement de typage pour les Pointeurs

$$\frac{\Gamma \vdash A : \text{TYPE*}}{\Gamma \vdash (*A) : \text{TYPE}}$$

$$\frac{\Gamma \vdash \text{TYPE}}{\Gamma \vdash \text{TYPE*} : \text{type des pointeurs sur TYPE}}$$

$$\Gamma \vdash \text{null} : \text{pointeur}$$

$$\frac{\Gamma \vdash \text{TYPE}}{\Gamma \vdash (\text{new TYPE}) : \text{TYPE*}}$$

$$\frac{\Gamma \vdash \text{id} : \text{TYPE}}{\Gamma \vdash \&\text{id} : \text{TYPE*}}$$

3.2 Jugement de Typage pour les Tableaux

$$\frac{\Gamma \vdash A : \text{TYPE[]}}{\Gamma \vdash (A[E]) : \text{TYPE}}$$

$$\frac{\Gamma \vdash \text{TYPE}}{\Gamma \vdash \text{TYPE[]} : \text{type des tableaux de TYPE}}$$

$$\frac{\Gamma \vdash \text{TYPE} \quad \Gamma \vdash E}{\Gamma \vdash (\text{new TYPE}[E]) : \text{TYPE[]}}$$

$$\frac{\Gamma \vdash \text{CP}}{\Gamma \vdash \{CP\} : \text{TYPE[]}}$$

3.3 Jugement de typage pour la Boucle for

$$\frac{\Gamma \vdash \text{int id} = \text{E} \quad \Gamma \vdash E \quad \Gamma \vdash \text{id} = \text{E} \quad \Gamma \vdash \text{BLOC}}{\Gamma \vdash \text{for (int id} = \text{E; E; id} = \text{E) BLOC}}$$

3.4 Jugement de typage pour Goto

$$\frac{\Gamma \vdash \text{id}}{\Gamma \vdash \text{goto id;}}$$

$$\frac{\Gamma \vdash \text{id}}{\Gamma \vdash \text{id :}}$$

4 Pointeurs

4.1 Grammaire

Pour les pointeurs, cette grammaire a été choisie :

— $A \rightarrow (*A)$: 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.

De cette grammaire découle nos modifications dans le parser :

```
a :
    ...
    | MULT n=a          {Deref n}
    | n=ID              {Ident n}

...

typ :
    | PO t=typ PF             {t}
    | t=typ MULT     {Pointer t}

...

e :
    ...
    | NULL                 {Null}
    | NEW t=typ           {New t}
    | REF n=ID       {Adresse n}
...
```

Listing 8 – Modification du parser pour les pointeurs

4.2 PasseTdsRat

La fonction `analyse_tds_affectable` parcourt l'arbre de syntaxe abstraite (AST) des affectables, en analysant et enrichissant les informations liées à la table des symboles (`tds`). Elle prend en compte la nature de la modification (`modif`) qui peut être en lecture (`"l"`) ou en écriture (`"e"`).

```
let rec analyse_tds_affectable tds a modif =
  match a with
  | AstSyntax.Ident n -> (match chercherGlobalement tds n with
                          | Some ia -> (match info_ast_to_info ia with
                                        | InfoVar(_,_,_,_) -> AstTds.Ident(ia)
                                        | InfoConst(_,_) -> if (modif == "l")
                                            then
                                          Ident(ia) else raise
                                              (MauvaiseUtilisationIdentifiant n)
                                        | _ -> raise
                                          (MauvaiseUtilisationIdentifiant n))
                          | None -> raise (IdentifiantNonDeclare n))
  | AstSyntax.Deref a -> AstTds.Deref (analyse_tds_affectable tds a modif)
...
```

Listing 9 – Modification de la passe de gestion des identifiants pour les pointeurs

La fonction `analyse_tds_affectable` prend un affectable (`a`), une table des symboles (`tds`), et une indication de modification (`modif`). Elle analyse le type de l'affectable et enrichit l'AST des affectables (`AstTds`) en conséquence. Dans cet extrait, deux cas principaux sont traités : `AstSyntax.Ident` pour les identifiants et `AstSyntax.Deref` pour le déréférencement.

- Pour `AstSyntax.Ident n` : La fonction cherche l'identifiant (`n`) dans la table des symboles globale (`tds`). Si l'identifiant est trouvé, le type d'information associé (`ia`) est extrait. Selon la nature de cet identifiant

(`InfoVar` pour une variable, `InfoConst` pour une constante, etc.), l'AST des affectables (`AstTds`) est enrichi avec l'information correspondante. Si l'identifiant n'est pas déclaré, une exception `IdentifiantNonDeclare` est levée.

- Pour `AstSyntax.Deref a` : La fonction poursuit l'analyse en utilisant la récursivité sur l'affectable (`a`) pour obtenir l'AST correspondant. L'AST résultant est ensuite encapsulé dans un `AstTds.Deref`, indiquant le déréférencement.

La fonction `analyse_tds_expression` suit une approche similaire, analysant l'expression (`e`) et enrichissant l'AST des expressions (`AstTds`) en conséquence.

```
1   and analyse_tds_expression tds e =
2   match e with
3   ...
4   | AstSyntax.Affectable a -> let res = analyse_tds_affectable tds a "l" in
        AstTds.Affectable res
```

Listing 10 – Modification de la passe de gestion des identifiants pour les pointeurs

Dans cet extrait, la fonction `analyse_tds_expression` prend une expression (`e`) et la table des symboles (`tds`). Si l'expression est un affectable, la fonction appelle `analyse_tds_affectable` avec l'indication de modification en lecture (`"l"`), et l'AST résultant est encapsulé dans un `AstTds.Affectable`.

4.3 PasseTypeRat

La fonction `analyse_type_affectable` effectue une analyse de type sur l'AST des affectables (`AstTds`). Elle prend un affectable (`a`) en entrée et renvoie un couple composé du type de l'affectable et de la nouvelle représentation dans l'AST des types (`AstType`).

```
1  let rec analyse_type_affectable a =
2    match a with
3    | AstTds.Ident(ia) ->
4      (*Renvoie d'un couple compose du type de l'identifiant et du nouvel Ident*)
5      (get_type_info (info_ast_to_info ia), AstType.Ident(ia))
6    | AstTds.Deref da ->
7      (* Analyse de l'affectable *)
8      let (ta, nda) = analyse_type_affectable da in
9      (match ta with
10       | Pointer(typ) -> (typ, AstType.Deref(nda, typ))
11       | _ -> raise (TypeInattendu(ta, Pointer(Undefined))))
```

Listing 11 – Modification de la passe de typage pour les pointeurs

La fonction `analyse_type_affectable` distingue deux cas principaux :

- Pour `AstTds.Ident(ia)` : La fonction extrait l'information de type associée à l'identifiant (`ia`). Elle renvoie ensuite un couple composé du type de l'identifiant et d'une nouvelle représentation dans l'AST des types (`AstType.Ident(ia)`).

- Pour `AstTds.Deref da` : La fonction analyse récursivement l'affectable (`da`) et obtient le type (`ta`) ainsi que la nouvelle représentation dans l'AST des types (`nda`). Si le type est un pointeur (`Pointer(typ)`), la fonction renvoie un couple composé du type du pointeur et de la nouvelle représentation dans l'AST des types (`AstType.Deref(nda, typ)`). Sinon, une exception `TypeInattendu` est levée, indiquant un type inattendu pour une opération de déréférencement.

4.4 PasseCodeRatToTam

La fonction `analyser_code_affectable` génère du code Tam (`TAM`) en analysant l'AST des affectables (`AstType`). Elle prend un affectable (`a`) et un mode (`mode`) en entrée (mode lecture (`"l"`) ou écriture (`"e"`)).

```
let rec analyser_code_affectable a mode  =
  (* Analyse de l'affectable *)
  match a with
  | AstType.Ident info ->
    (match info_ast_to_info info with
     | InfoVar(_, t, d, reg) ->
        if mode = "l" then
          load (getTaille t) d reg
        else
          store (getTaille t) d reg
     | InfoConst(_, i) -> loadl_int i
     | _ -> failwith "Impossible Ident")
  | AstType.Deref (da, t) -> analyser_code_affectable da "l" ^
                             (if mode = "e" then
                               storei (getTaille t)
                             else
                               loadi (getTaille t))
```

Listing 12 – Modification de la passe de génération de code pour les pointeurs

La fonction `analyser_code_affectable` distingue deux cas principaux dans le cas des pointeurs :

- Pour `AstType.Ident info` : La fonction extrait l'information associée à l'identifiant (`info`). Selon le type de cette information, elle génère du code Tam approprié. Si le mode est en lecture (`"l"`), elle charge la valeur de la variable en mémoire à l'aide de `load`, sinon elle conserve la valeur à l'adresse spécifiée par `store`. Pour une constante, elle utilise `loadl_int` pour charger la valeur constante en mémoire.

- Pour `AstType.Deref (da, t)` : La fonction analyse récursivement l'affectable (`da`) en mode lecture (`"l"`) et génère du code Tam pour le déréférencement. Si le mode est en écriture (`"e"`), elle utilise `storei` pour stocker la valeur à l'adresse spécifiée, sinon elle utilise `loadi` pour charger la valeur à l'adresse spécifiée.

5 Tableaux

5.1 Grammaire

Pour les tableaux, cette grammaire a été choisie :

— $A \rightarrow (A[E])$: accès en lecture ou écriture à la case d'indice E du tableau A ;

— TYPE $\rightarrow$ TYPE[] : type des tableaux de TYPE (la syntaxe est différente de celle de C pour simplifier l'écriture de la grammaire) ;

— $E \rightarrow$ (new TYPE$[E]$) : création d'un tableau de TYPE et de taille E ;

— $E \rightarrow \{CP\}$: initialisation d'un tableau avec un ensemble de valeurs.

De cette grammaire découle nos modifications dans le parser :

```
a :
| PO a1=a CO e1=e CF PF {TabInd(a1, e1)}

...

typ :
...
| t=typ CO CF {Tab t}

...

e :
...
| NEW t=typ CO e1=e CF {NewTab (t,e1)}
```

Listing 13 – Modification du parser pour les tableaux

5.2 PasseTdsRat

```
let rec analyse_tds_affectable tds a modif =
  match a with
  | AstSyntax.TabInd (a, e) -> AstTds.TabInd (analyse_tds_affectable tds a
      modif, analyse_tds_expression tds e)
...
```

Listing 14 – Modification de la passe de gestion des identifiants pour les tableaux

La fonction utilise une correspondance de motifs (`match`) pour distinguer les différentes variantes de l'AST des affectables. Dans cet extrait, elle traite le cas spécifique de `AstSyntax.TabInd`, où un affectable est associé à une expression. Elle appelle récursivement `analyse_tds_affectable` sur l'affectable (`a`) et `analyse_tds_expression` sur l'expression (`e`). Ensuite, elle crée l'AST correspondant (`AstTds.TabInd`) avec les résultats de ces analyses.

Cette extension permet de gérer les affectations à des indices de tableaux dans la table des symboles (`tds`) et enrichit l'AST des affectables (`AstTds`) avec cette nouvelle construction.

5.3 PasseTypeRat

```
let rec analyse_type_affectable a =
  match a with
  ...
  | AstTds.TabInd (a, e) -> let (ta, na) = analyse_type_affectable a in
                            let ne = analyse_type_expression e in
                            let te = get_typ_expression ne in
                            if te != Int then raise (TypeInattendu(te, Int))
                            else (match ta with
                                  | Tab(tt) -> (tt, AstType.TabInd(na, ne, tt))
                                  | _ -> raise (TypeInattendu(ta,
                                     Tab(Undefined))))
```

Listing 15 – Modification de la passe de typage pour les tableaux

La fonction utilise une correspondance de motifs (`match`) pour distinguer les différentes variantes de l'AST des affectables. Dans cet extrait, elle traite le cas spécifique de `AstTds.TabInd`, où un affectable est associé à une expression.

Elle appelle récursivement `analyse_type_affectable` sur l'affectable (`a`) et `analyse_type_expression` sur l'expression (`e`). Ensuite, elle extrait les types résultants (`ta` et `te`). La fonction effectue des vérifications de type pour s'assurer que l'expression associée à l'indice du tableau est de type `Int`.

Enfin, elle construit l'AST correspondant (`AstType.TabInd`) avec les résultats de ces analyses, et elle renvoie le type du tableau (`tt`) associé à l'indice. Cette extension permet de gérer les types des affectations à des indices de tableaux dans le cadre de la passe de l'analyseur sémantique (`PasseTypeRat`).

5.4 PasseCodeRatToTam

```
let rec analyser_code_affectable a mode  =
  (* Analyse de l'affectable *)
  match a with
  ...
  | AstType.TabInd (a, e, t) ->
    let res = analyser_code_affectable a "l" in
    let res2 = analyser_code_expression e in
    res ^ (* Chargement @ du tableau *)
    res2 ^ loadl_int (getTaille (t)) ^ subr "IMul" ^ (* Chargement indice :
        (i*tailleElem) *)
    subr "IAdd" ^ (* t[i] = @t-(i*tailleElem) *)
    (if mode = "e" then
       storei (getTaille (t))
     else
       loadi (getTaille (t)))
```

Listing 16 – Modification de la passe de génération de code pour les tableaux

La fonction utilise une correspondance de motifs (`match`) pour distinguer les différentes variantes de l'AST des affectables. Dans cet extrait, elle traite le cas spécifique de `AstType.TabInd`, où un affectable est associé à une expression et un type.

Elle appelle récursivement `analyser_code_affectable` sur l'affectable (`a`) en mode lecture (`"l"`). Ensuite, elle génère du code Tam pour charger l'adresse du tableau en mémoire (`@t`) et pour calculer l'indice du tableau (`i*tailleElem`). Elle effectue une multiplication (`IMul`) et une addition (`IAdd`) pour obtenir l'adresse de l'élément du tableau correspondant à l'indice. Enfin, elle génère du code Tam pour charger ou stocker la valeur à l'adresse calculée, en fonction du mode spécifié (`mode`). Cette extension permet de gérer la génération de code Tam pour les affectations à des indices de tableaux dans le cadre de la passe de la génération de code Tam (`PasseCodeRatToTam`).

6 Boucles "for"

6.1 Grammaire

Pour la boucle for, cette grammaire a été choisie :

$$\text{I} \rightarrow \text{for(int id} = E; E; \text{id} = E)\text{BLOC}$$

Cependant, afin de simplifier le code à écrire dans les autres passe, on a choisi de remplacer int id = E ; par une instruction qui sera une déclaration dans le bloc de la boucle for. A noter que pour la deuxième affectation, nous aurions bien voulu utiliser une instruction mais le point virgule était problématique pour le parser. On obtient donc :

$$\text{I} \rightarrow \text{for(Instruction} = I; E; \text{id} = E)\text{BLOC}$$

De cette grammaire découle nos modifications dans le parser :

```
i :
...
| FOR PO i1=i e2=e PV n2=a EQUAL e3=e PF li=bloc  {For (i1,e2,n2,e3,li)}
...
```

Listing 17 – Modification du parser pour la boucle for

6.2 PasseTdsRat

```
let rec analyse_tds_instruction tds maintds oia i =
  match i with
...
| AstSyntax.For (i1, e2, a, e3, b) -> let tdsfor = creerTDSFille tds in

        let ni1 = (match i1 with
        | AstSyntax.Declaration _ -> analyse_tds_instruction tdsfor tdsGOTO
            maintds oia i1
        | _ -> failwith "Mauvaise declaration") in


        let ne2 = (match e2 with
        | AstSyntax.Booleen _ -> analyse_tds_expression tdsfor e2
        | AstSyntax.Binaire (op, _, _) -> if op == AstSyntax.Equ || op ==
            AstSyntax.Inf then
            analyse_tds_expression tdsfor e2 else raise
                (MauvaiseUtilisationIdentifiant "Operation condition for")
        | AstSyntax.Affectable _ -> analyse_tds_expression tdsfor e2
        | _ -> failwith "Mauvaise condition d'arret") in

        AstTds.For (ni1, ne2, analyse_tds_affectable tdsfor a "e",
            analyse_tds_expression tdsfor e3, analyse_tds_bloc tdsfor None b)
```

Listing 18 – Modification de la passe de gestion des identifiants pour la boucle for

Il est important de remarquer qu'une nouvelle tds à été rajoutée dans **analyse_tds_instruction**. Cette nouvelle tds permet de supporter la liste des instructions présentes dans la boucle for sans qu'elles se retrouve dans la tds principales et n'empeche la nouvelle déclaration de variables définies dans la boucle for.

La fonction utilise le (`match`) pour distinguer les différentes variantes d'instructions. Dans cet extrait, elle traite le cas spécifique de `AstSyntax.For`, où une boucle `for` est déclarée. Elle crée une nouvelle table des symboles (`tdsfor`) en utilisant la fonction auxiliaire `creerTDSFille`. Cette nouvelle tds permet de tout déclarer dans le bloc et ainsi pouvoir déclarer à nouveau en dehors du bloc des variables déclarées dans le bloc de la boucle for.

Ensuite, elle analyse récursivement les éléments de la boucle `for`, à savoir la déclaration de la variable de boucle (`ni1`), la condition d'arrêt (`ne2`), l'affectation (`analyse_tds_affectable`), l'expression de fin de boucle (`analyse_tds_expression`), et le bloc d'instructions (`analyse_tds_bloc`). Enfin, elle construit l'AST correspondant (`AstTds.For`) avec les résultats de ces analyses.

6.3 PasseTypeRat

La fonction **analyse_type_instruction** va être modifiée afin de permettre repérer et gérer l'erreur de typage dans les paramètres de la boucle ainsi que dans son bloc.

```
let rec analyse_type_instruction i =
  match i with
...
  | AstTds.For (i1, e2, a, e3, b) ->
      let ne1 = analyse_type_instruction i1 in
      let te1 = (match ne1 with
                | AstType.Declaration (info, _) ->
                (match info_ast_to_info info with
                     | InfoVar (_, t, _, _) -> if t=Int then Int
                                               else raise (TypeInattendu (t, Int))
                     | _ -> failwith "Erreur dans le for")
                | _ -> failwith "Impossible for"  ) in
       let ne2 = analyse_type_expression e2 in
       let ne3 = analyse_type_expression e3 in
       let te2 = get_typ_expression ne2 in
       let ta, na = analyse_type_affectable a in
       let te3 = get_typ_expression ne3 in
       (match te1, te2, ta, te3 with
       | Int, Bool, Int, Int ->
           let nb = analyse_type_bloc b in
           AstType.For (ne1, ne2, na, ne3, nb)
       | _ -> raise (TypeInattendu (te1, Int)))
```

Listing 19 – Modification de la passe de typage pour la boucle for

La fonction utilise une correspondance de motifs (`match`) pour distinguer les différentes variantes de l'AST des instructions. Dans cet extrait, elle traite le cas spécifique de `AstTds.For`, où une boucle `for` est déclarée.

Elle analyse récursivement la déclaration de la variable de boucle (`ne1`), la condition d'arrêt (`ne2`), l'affectation (`analyse_type_affectable`), l'expression de fin de boucle (`analyse_type_expression`), et le bloc d'instructions (`analyse_type_bloc`). Ensuite, elle vérifie que les types sont conformes aux spécifications d'une boucle `for`, à savoir que la variable de boucle est de type `Int`, la condition d'arrêt est de type `Bool`, l'affectation est de type `Int`, et l'expression de fin de boucle est de type `Int`. Si ces conditions sont respectées, elle construit l'AST correspondant (`AstType.For`) avec les résultats de ces analyses.

6.4 PasseCodeRatToTam

Cette extension permet de générer du code TAM pour les boucles `for`.

```
and analyser_code_instruction i =
  match i with
...
  | AstPlacement.For (i1, e2, a, e3, b) ->
        let debut = getEtiquette() in
        let fin = getEtiquette() in
        analyser_code_instruction i1 ^
        label debut ^
        analyser_code_expression e2 ^
        jumpif 0 fin ^
        analyser_code_bloc b ^
        analyser_code_expression e3 ^
        analyser_code_affectable a "e" ^
        jump debut ^
        label fin
```

Listing 20 – Modification de la passe de génération de code pour la boucle for

La fonction utilise une correspondance de motifs (`match`) pour distinguer les différentes variantes de l'AST des instructions. Dans cet extrait, elle traite le cas spécifique de `AstPlacement.For`, où une boucle `for` est placée.

Elle génère du code TAM correspondant à une boucle `for` en utilisant des étiquettes (`debut` et `fin`) pour gérer la structure de la boucle. Le code généré commence par analyser l'instruction de l'initialisation de la boucle (`analyser_code_instruction i1`), suivi d'un saut vers l'étiquette de début (`jump debut`). Ensuite, elle génère le code pour l'expression de condition d'arrêt (`analyser_code_expression e2`) suivi d'un saut conditionnel (`jumpif 0 fin`). Le code du bloc de la boucle (`analyser_code_bloc b`) est généré, suivi du code pour l'expression d'incrémentation (`analyser_code_expression e3`) et de l'affectation (`analyser_code_affectable a "e"`). Enfin, un saut inconditionnel vers l'étiquette de début (`jump debut`) est ajouté, et l'étiquette de fin (`fin`) est déclarée.

7 Goto

7.1 Grammaire

Pour l'instruction **goto**, cette grammaire a été choisie :

$$I \rightarrow \text{goto id}\,;$$
$$I \rightarrow \text{id}\,:$$

De cette grammaire découle nos modifications dans le parser :

```
i :
...
| GOTO n=ID PV                          {Goto n}
| n=ID DP                               {Label n}
```

Listing 21 – Modification du parser pour goto

7.2 Nouvelle info

```
type info =
  | InfoConst of string * int
  | InfoVar of string * typ * int * string
  | InfoFun of string * typ * typ list
  | InfoEtiq of string * bool * string
```

Listing 22 – Ajout d'une nouvelle info pour l'étiquette dans l'Ast

La nouvelle information, **InfoEtiq**, conserve le nom de l'étiquette (**string**), un **booléen** permettant de savoir si une étiquette a déjà été déclarée en tant que **label** afin de vérifier la double déclaration des **label** et finalement un **string** qui conserve le nom de l'étiquette unique généré par la fonction **getEtiquette** dans la passe **PasseTdsRat**.

7.3 PasseTdsRat

```
let etiquettes_non_declarees = ref []

let supprimer_element x liste =
  List.filter (fun e -> e <> x) liste

let verifier_etiquettes etiquettes =
  if (List.length etiquettes) > 0 then
    raise (IdentifiantNonDeclare (List.hd etiquettes))
  else
    ()

let rec analyse_tds_instruction tds tdsGOTO maintds oia i =
  match i with
  ...
  | AstSyntax.Goto (n) ->
      (match chercherGlobalement tdsGOTO n with
      | Some ia ->
          (match info_ast_to_info ia with
          | InfoEtiq (_, false, nomGOTO) ->
              AstTds.Goto (info_to_info_ast (InfoEtiq(n, true, nomGOTO)))
          | _ -> raise (MauvaiseUtilisationIdentifiant n))
      | None ->
          let info = info_to_info_ast (InfoEtiq(n, false, getEtiquette())) in
          etiquettes_non_declarees := n :: !etiquettes_non_declarees;
          ajouter tdsGOTO n info;
          AstTds.Goto (info))

  | AstSyntax.Label (n) ->
      (match chercherGlobalement tdsGOTO n with
      | Some ia ->
          if (List.mem n !etiquettes_non_declarees) then
            etiquettes_non_declarees := supprimer_element n
                !etiquettes_non_declarees;
          (match info_ast_to_info ia with
          | InfoEtiq (_, false, nomGOTO) ->
              AstTds.Label (info_to_info_ast (InfoEtiq(n, false, nomGOTO)))
          | InfoEtiq (_, true, _) -> raise (DoubleDeclaration n)
          | _ -> raise (MauvaiseUtilisationIdentifiant n))
      | None ->
          let info = info_to_info_ast (InfoEtiq(n, true, getEtiquette())) in
          ajouter tdsGOTO n info;
          AstTds.Label (info))
```

Listing 23 – Extension goto dans la PasseTdsRat

On remarque la création d'une nouvelle table des symboles (`tdsGOTO`) spécifique pour les `Goto`. Cette table nous permet d'utiliser l'instruction `Goto` pour sauter vers une destination en dehors du bloc dans lequel on se trouve.

De plus, une complexité de cette instruction est qu'il peut y avoir plusieurs instructions `Goto` dans le code. Cependant, l'étiquette `Label` correspondante ne doit apparaître qu'une seule fois. C'est pourquoi l'info de l'étiquette comporte un booléen qui indique si une `Label` a déjà été déclarée.

Une autre complexité est la nécessité de lever une erreur si on exécute l'instruction `Goto` sur une étiquette qui n'a pas été déclarée en tant que `Label`. Nous utilisons donc une liste qui stocke les noms des étiquettes qui n'ont pas été déclarées en tant que `Label`. À la fin de l'analyse du code, on vérifie que la liste d'étiquettes non déclarées est vide. Si ce n'est pas le cas, une étiquette n'a pas été déclarée en tant que `Label`, et l'exception `IdentifiantNonDeclare` est levée.

7.4 PasseCodeRatToTam

Cette extension permet de générer du code TAM pour l'instruction `goto`.

```
| AstPlacement.Goto (is) -> (match info_ast_to_info is with
                              | InfoEtiq (_, _, etiq) -> jump etiq
                              | _ -> failwith "Impossible" )
 AstPlacement.Label (is) -> (match info_ast_to_info is with
                              | InfoEtiq (_, _, etiq) -> label etiq
                              | _ -> failwith "Impossible" )
```

Listing 24 – Modification dans la passe de génération du code pour goto

Ici, afin d'effectuer l'instruction **goto** vers une étiquette, on effectue **jumpif** vers l'identifiant unique de l'étiquette créé dans la **PasseTdsRat**.

8 Tests Unaires

Pour enrichir les tests unaires, nous avons ajouté des cas spécifiques couvrant les fonctionnalités étendues du langage RAT. Ces nouveaux tests portent sur les pointeurs, les tableaux, les boucles "for" et les instructions "goto". Nous avons minutieusement testé les erreurs de typage et d'identifiants, en les ajoutant progressivement pour garantir une couverture complète.

Afin d'évaluer la compatibilité des différentes améliorations, nous avons intégré ces tests de manière intégrée dans le programme général, tel que spécifié dans le sujet du projet. Cette approche nous a permis de vérifier la cohérence des fonctionnalités étendues et leur interaction au sein du langage.

Par ailleurs, nous avons délibérément conçu des scénarios de test impliquant des combinaisons improbables de pointeurs et de tableaux. Cette démarche nous a permis d'explorer des cas limites et de nous assurer de la robustesse de notre implémentation face à des situations peu conventionnelles.

9 Conclusion

Ce projet s'est avéré être une expérience enrichissante bien que complexe. L'écriture de la **PasseCodeToRat** a été particulièrement challenging, surtout au début, en raison de difficultés à déboguer le programme sans utiliser iTam. Nous avons finalement résolu ce problème en ajoutant des déclarations **print_endline** à la fin de la fonction `analyser`, ce qui nous a permis de mieux comprendre le flux d'exécution du programme.

Par ailleurs, la gestion des tableaux dans la **PasseCodeToRat** a été une autre source de difficulté que nous avons réussi à surmonter grâce à l'investissement en temps et à la persévérance.

L'un des défis majeurs a été la compréhension et la mise en œuvre de la gestion des identifiants liée à l'instruction **goto**. Nous avons particulièrement buté sur la manière de stocker les étiquettes non déclarées en tant que **label**. Nous avons envisagé des approches telles que l'utilisation de variables de classe à la manière de la programmation orientée objet, mais cela n'était pas applicable dans le contexte de la programmation fonctionnelle. De même, l'idée d'utiliser une variable globale nécessitait des modifications conséquentes dans la gestion des identifiants, ce qui aurait impacté d'autres parties du code.

En définitive, ce projet nous a offert une occasion d'approfondir notre compréhension du fonctionnement d'un compilateur. Cette expérience a été précieuse, contribuant significativement à notre capacité à comprendre et résoudre des erreurs de compilation dans d'autres projets parallèles.