Traduction des langages

Arbre syntaxique abstrait, table des symboles et résolution des identifiants

Objectif :
- Définir l'AST (Abstract Syntax Tree) pour le langage RAT
- Comprendre le traitement réalisé par la passe de résolution des identifiants
- Comprendre l'intérêt de la table des symboles
- Définir la nouvelle structure d'arbre obtenue après la passe de résolution des identifiants
- Définir les actions à réaliser par la passe de résolution des identifiants

1 Le langage Rat

Cours : transparents de présentation de Rat pages 80-82

1.1 Grammaire du langage RAT

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

1.2 Exemple

```
bool less ( rat a rat b ){
  return ((num a * denom b )<( num b * denom a ));
}

prog{
```

```
  rat a = [3/4];
  rat b = [4/5];
  const n = 5;
  int i = 0;
  while(i<n){
    a=(a+a);
    b=(b*b);
    i=(i+1);
  }
  if(call less(a b)){print a;} else {print b;}
}
```

2 Structure de l'AST pour RAT

Cours : partie V sur l'AST.
Les transparents 83 - 85 montrent, pour un exemple de code RAT, l'arbre de dérivation et l'AST retenu.

▷ **Exercice 1** *Définir la structure de l'AST pour RAT.*

Voici le type utilisé dans les TP :

```
module AstSyntax =
struct

(* Opérateurs binaires de Rat *)
type binaire = Plus | Mult | Equ | Inf

(* Expressions de Rat *)
type expression =
  (* Appel de fonction représenté par le nom de la fonction et la liste des paramètres réels *)
  | AppelFonction of string * expression list
  (* Rationnel représenté par le numérateur et le dénominateur *)
  | Rationnel of expression * expression
  (* Accès au numérateur d'un rationnel *)
  | Numerateur of expression
  (* Accès au dénominateur d'un rationnel *)
  | Denominateur of expression
  (* Accès à un identifiant représenté par son nom *)
  | Ident of string
  (* Booléen vrai *)
  | True
  (* Booléen faux *)
  | False
  (* Entier *)
  | Entier of int
  (* Opération binaire représentée par l'opérateur, l'opérande gauche et l'opérande droite *)
  | Binaire of binaire * expression * expression

(* Instructions de Rat *)
type bloc = instruction list
and instruction =
```

```
  (* Déclaration de variable représentée par son type, son nom et l'expression d' initialisation *)
  | Declaration of typ * string * expression
  (* Affectation d'une variable représentée par son nom et la nouvelle valeur affectée *)
  | Affectation of string * expression
  (* Déclaration d'une constante représentée par son nom et sa valeur (entier) *)
  | Constante of string * int
  (* Affichage d'une expression *)
  | Affichage of expression
  (* Conditionnelle représentée par la condition, le bloc then et le bloc else *)
  | Conditionnelle of expression * bloc * bloc
  (*Boucle TantQue représentée par la conditin d'arrêt de la boucle et le bloc d'instructions *)
  | TantQue of expression * bloc

(* Structure des fonctions de Rat *)
(* type de retour - nom - liste des paramètres (association type et nom) - corps de la fonction - expression de retour *)
type fonction = Fonction of typ * string * (typ * string) list * instruction list * expression

(* Structure d'un programme Rat *)
(* liste de fonction - programme principal *)
type programme = Programme of fonction list * bloc

end
```

3 Passe de résolution des identifiants

Nous rappelons qu'un compilateur fonctionne par passes, chacune d'elle réalisant un traitement particulier (gestion des identifiants, typage, placement mémoire, génération de code,...). Chaque passe parcourt, et potentiellement modifie, l'AST.

La première passe est une passe de résolution des identifiants. C'est elle qui vérifie la bonne utilisation des identifiants. En particulier, dans le cas du langage RAT, c'est elle qui vérifie que :

- déclaration : vérifier l'absence de double déclaration dans le même bloc ;
- utilisation d'un symbole : vérifier l'existence de ce symbole dans ce bloc ou un bloc englobant ;
- utilisation d'un symbole : vérifier que sa catégorie (variable, constante, fonction) est correcte.

Cours : audio du transparent 86

Pour réaliser cette passe, nous avons besoin d'une table des symboles : structure de données associant aux identifiants leurs informations.

3.1 Table des symboles

Cours : transparents 86 à 91 (table des symboles).

L'interface du module Tds utilisé en TP est la suivante :

```
(* Définition du type des informations associées aux identifiants *)
type info =
  (* Information associée à une constante : son nom (non indispensable mais aide au test et debbugage) et sa valeur *)
  | InfoConst of string * int
  (* Information associée à une variable : son nom (non indispensable mais aide au test et debbugage),
  son type, et son adresse ie son déplacement (int) par rapport à un registre (string) *)
  | InfoVar of string * typ * int * string
```

```
  (* Information associée à une fonction : son nom (utile pour l'appel), son type de retour
     et la liste des types des paramètres *)
  | InfoFun of string * typ * typ list

(* Table des symboles *)
type tds

(* Données stockées dans la tds et dans les AST : pointeur sur une information *)
type info_ast

(* Création d'une table des symboles à la racine *)
val creerTDSMere : unit -> tds

(* Création d'une table des symboles fille *)
(* Le paramètre est la table mère *)
val creerTDSFille : tds -> tds

(* Ajoute une information dans la table des symboles locale *)
(* tds : la tds courante *)
(* string : le nom de l'identificateur *)
(* info : l'information à associer à l'identificateur *)
(* Si l'identificateur est déjà présent dans TDS, l'information est écrasée *)
(* retour : unit *)
val ajouter : tds -> string -> info_ast -> unit

(* Recherche les informations d'un identificateur dans la tds locale *)
(* Ne cherche que dans la tds de plus bas niveau *)
val chercherLocalement : tds -> string -> info_ast option

(* Recherche les informations d'un identificateur dans la tds globale *)
(* Si l'identificateur n'est pas présent dans la tds de plus bas niveau la recherche est effectuée *)
(* dans sa table mère et ainsi de suite jusqu'à trouver (ou pas) l'identificateur *)
val chercherGlobalement : tds -> string -> info_ast option

(* Affiche la tds locale *)
val afficher_locale : tds -> unit

(* Affiche la tds locale et récursivement *)
val afficher_globale : tds -> unit

(* Créer une information à associer à l'AST àpartir d'une info *)
val info_to_info_ast : info -> info_ast

(* Récupère l'information associée à un noeud *)
val info_ast_to_info : info_ast -> info

(* Modifie le type si c'est une InfoVar, ne fait rien sinon *)
val modifier_type_info : typ -> info_ast -> unit

(* Modifie les types de retour et des paramètres si c'est une InfoFun, ne fait rien sinon *)
val modifier_type_fonction_info : typ -> typ list -> info_ast -> unit

(* Modifie l'emplacement (dépl, registre) si c'est une InfoVar, ne fait rien sinon *)
```

```
val modifier_adresse_info : int -> string -> info_ast -> unit
```

- **`info_ast` est un pointeur sur une `info`. On utilisera à bon escient `info_ast_to_info` pour obtenir la valeur (déréférencement du pointeur) et `info_to_info_ast` exclusivement pour créer le pointeur pour l'insérer dans la table des symboles.**
- **Cette partie de l'UE fait exceptionnellement une entorse à la programmation sans effet de bord : on autorise les effets de bord sur la tds et les infos. En effet, les passes suivantes (typage, placement mémoire) vont mettre à jour les informations associées aux symboles. Il est plus simple de modifier l'information associée que de reconstruire une nouvelle tds.**

3.2 Structure de l'AST post passe de résolution des identifiants

La passe de résolution des identifiants réalise des vérifications mais prépare également les passes suivantes. Elles auront besoin d'avoir accès aux informations des identifiants, il faut donc modifier l'AST de façon à rendre ces informations accessibles.

▷ **Exercice 2** *Définir la structure de l'AST post passe de résolution des identifiants.*

- **On associe les informations des identifiants à tous les nœuds où il y a des identifiants.**
- **Puisque les identifiants ont été résolus, il n'est plus nécessaire de les garder dans l'arbre.**
- **Comme les valeurs des constantes sont stockées dans l'info associée à la constante, il est inutile de garder leur déclaration dans l'AST.**

Voici un type possible pour l'arbre après la passe de résolution des identifiants :

```
module AstTds =
struct
 (* Expressions existantes dans notre langage *)
  (* ~ expression de l'AST + informations associées aux identificateurs *)
  type expression =
    | AppelFonction of Tds.info_ast * expression list
    | Rationnel of expression * expression
    | Numerateur of expression
    | Denominateur of expression
    | Ident of Tds.info_ast
    | True
    | False
    | Entier of int
    | Binaire of AstSyntax.binaire * expression * expression

  (* instructions existantes dans notre langage *)
  (* = instruction de l'AST + infos associées aux identificateurs + suppression de nœuds *)
  type bloc = instruction list
  and instruction =
    | Declaration of typ * expression * Tds.info_ast
    | Affectation of  expression * Tds.info_ast
    | Affichage of expression
    | Conditionnelle of expression * bloc * bloc
    | TantQue of expression * bloc
```

```
  | Empty (* les nœuds ayant disparus: Const *)


  (* Structure des fonctions dans notre langage *)
  (* type de retour, liste des paramètres, corps,
     expression de retour, informations associées à l'identificateur (dont son nom) *)
  type fonction = Fonction of typ * (typ * Tds.info_ast ) list *
                              instruction list * expression * Tds.info_ast

  (* Structure d'un programme dans notre langage *)
  type programme = Programme of fonction list * bloc
end
```

On ne peut pas se contenter d'associer l'information de l'identifiant, car on a besoin d'avoir des modifications par effet de bord ⇒ c'est au niveau de la déclaration que les informations sont connues, chaque passe va mettre à jour petit à petit ces informations ; mais c'est au niveau des utilisations qu'on en a besoin. On a donc un pointeur sur les informations.

3.3 Actions à réaliser lors de la passe de résolution des identifiants

La passe de résolution des identifiants est donc une transformation d'un arbre vers un autre.

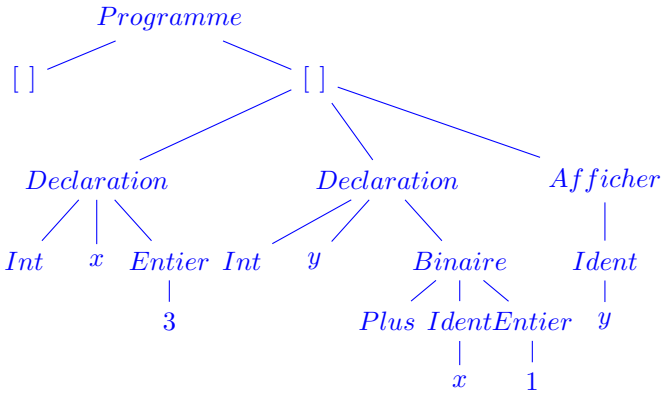
▷ **Exercice 3** *Définir les actions à réaliser lors de la passe de résolution des identifiants.*

Considérons l'exemple simple du cours et voyons ce qu'on veut obtenir.

1. **Code :**

```
main{
 int x = 3;
 int y = x+1;
 print y;
}
```

2. **AstSyntax**



3. **Actions à réaliser**

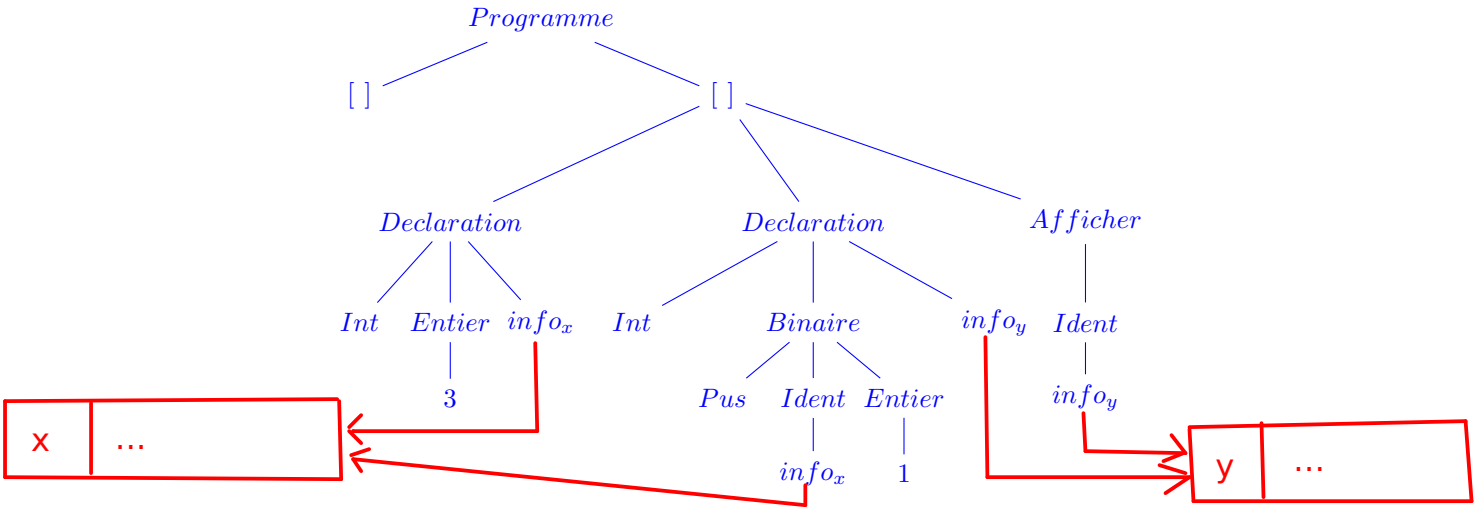
(a) **Instruction 1 : vérifier que x n'a pas déjà été déclaré**

(b) **Instruction 2 : vérifier que y n'a pas déjà été déclaré**

(c) **Instruction 2 : vérifier que x a déjà été déclaré comme variable ou constante**

(d) **Instruction 3 : vérifier que y a déjà été déclaré comme variable ou constante**

4. **AstTds**



$info_x$ et $info_y$ sont des pointeurs, donc quand l'information sera modifiée au niveau de la déclaration, cette modification sera visible au niveau de l'utilisation.

Comment obtenir ce résultat ?

- **On propage une tds dans l'arbre. Elle sera modifiée par effet de bord (pas besoin de la faire remonter) ;**
- **Au niveau des déclarations, on crée une information qu'on ajoute à la tds ;**
- **Au niveau des utilisations, on va chercher l'information dans la tds, on contrôle la bonne utilisation, et on l'attache au nœud ;**
- **Pour faire circuler la tds, elle est passée en paramètre de la fonction d'analyse.**

Le plus simple est de commencer par l'analyse des instructions :

```
(* Ast.instruction -> tds -> Asttds.instruction *)
let rec analyse_tds_instruction tds i =
  match i with
  | AstSyntax.Declaration (t, id, e) ->
      On cherche localement id dans la tds
      Si il est présent
      Alors Erreur "double déclaration"
      Sinon
          - on analyse l'expression et on récupère le nouvel arbre associé (ne)
          - on ajoute l' identifiant à la tds (InfoVar (id, Undefined, 0, ""))
          - on renvoie le nouvel arbre : Declaration (t, ne, info)
  | AstSyntax.Affectation (id,e) ->
      On cherche globalement id dans la tds
```

```
      Si il est présent
      Alors
        Si c'est un InfoVar
        Alors
          - on analyse l'expression et on récupère le nouvel arbre associé (ne)
          - on renvoie le nouvel arbre : Affectation (ne, info)
        Sinon Erreur "mauvaise utilisation d'un identifiant (modification de constante ou fonction)"
      Sinon Erreur " identifiant non déclaré"
  | AstSyntax.Constante (id,v) ->
      On cherche localement id dans la tds
      Si il est présent
      Alors Erreur "double déclaration"
      Sinon
        - on ajoute l' identifiant à la tds (InfoConst (id,v))
        - on renvoie le nouvel arbre : Empty
  | AstSyntax.Affichage e ->
      - on analyse l'expression et on récupère le nouvel arbre associé (ne)
      - on renvoie le nouvel arbre : Affichage (ne)
  | AstSyntax.Conditionnelle (c,t,e) ->
      (* Analyse de la condition *)
      let nc = analyse_tds_expression tds c in
      (* Analyse du bloc then *)
      let tast = analyse_tds_bloc tds t in
      (* Analyse du bloc else *)
      let east = analyse_tds_bloc tds e in
      (* les variables sont déclarées localement aux blocs *)
      Conditionnelle (nc, tast, east)
  | AstSyntax.TantQue (c,b) ->
      (* Analyse de la condition *)
      let nc = analyse_tds_expression tds c in
      (* Analyse du bloc *)
      let bast = analyse_tds_bloc tds b in
      TantQue (nc, bast)
```

On peut alors passer à l'analyse du bloc :

```
(* Ast.bloc -> Asttds.bloc *)
and analyse_tds_bloc tds li =
  - création de la tds locale au bloc
  - analyse de chacune des instructions du bloc avec la tds locale au bloc
     => map
```

Puis on passe à l'analyse des expressions :

```
let rec analyse_tds_expression tds e =
  match e with
  | AstSyntax.AppelFonction (id,le) ->
      On cherche globalement id dans la tds
      S'il est présent
      Alors
        Si c'est un InfoFun
        Alors
          - on analyse la liste des paramètres (map) et on récupère le nouvel arbre associé (ne)
          - on renvoie le nouvel arbre : AppelFonction (ide, ne, info)
                                         AppelFonction(info,ne)
```

```
        Sinon Erreur "mauvaise utilisation d'un identifiant (appel d'autre chose qu'une fonction)"
      Sinon Erreur " identifiant non déclaré"
  | AstSyntax.Ident n ->
      On cherche globalement id dans la tds
      S'il est présent
      Alors
        Si c'est un InfoVar alors on renvoie le nouveau noeud : Ident info
        Si c'est un InfoConst(_,v) alors on remplace la constante : Entier v
        Sinon Erreur "mauvaise utilisation d'un identifiant ( utilisation d'une fonction sans call )"
      Sinon Erreur " identifiant non déclaré"
  | AstSyntax.Rationnel (e1,e2) ->
      Rationnel (analyse_tds_expression tds e1 , analyse_tds_expression tds e2)
  | ...
```

La suite est sur le même schéma car il n'y a pas d'identifiant :

- **on propage la tds et la vérification dans tout l'arbre**
- **on reconstruit l'arbre avec ce qui remonte de l'analyse**

On peut alors faire l'analyse du programme :

```
let analyser (AstSyntax.Programme (fonctions,prog)) =
  - création de la TDS initiale : vide
  - analyse des fonctions et récupération du nouvel arbre associé aux fonctions (nf)
  - analyse du bloc principal et récupération du nouvel arbre associé au bloc principal (nb)
  - renvoi du nouvel arbre : Programme (nf, nb)
```

Enfin, il reste à faire l'analyse des fonctions : réfléchir et coder.