

# Traduction des langages

## Les pointeurs

### Objectif :

- Etre capable d'ajouter les pointeurs au langage RAT
- Modifier le compilateur en conséquence

## 1 Les pointeurs

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.

Un **pointeur** est une variable qui contient l'**adresse** d'un autre objet informatique (une "variable de variable" en somme).

### 1.1 Déclaration

La déclaration d'un pointeur se fait selon la syntaxe suivante :

```
type* id;
```

Cette instruction déclare une variable de nom `id` et de type `pointeur(type)` (pointeur sur une valeur de type `type`). Par exemple :

```
int* x;
```

déclare une variable `x` qui pointe sur une valeur de type `int`.

### 1.2 Adresse et valeur pointée

Les deux opérateurs particuliers en relation avec les pointeurs sont : `&` et `*`.

- `&` est l'opérateur qui **retourne l'adresse mémoire d'une variable**

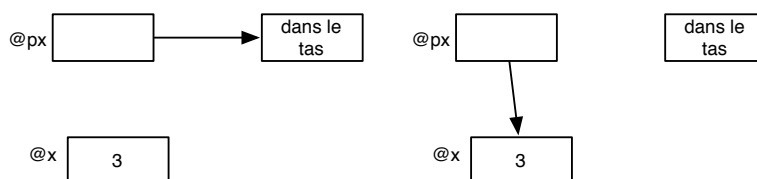
Si `x` est de type `t` alors `&x` est de type `Pointeur(t)`

Exemple :

```
int * px = (new int);
```

```
int x = 3;
```

```
px = &x;
```



- `*` est l'opérateur qui **retourne la valeur pointée par une variable pointeur**.

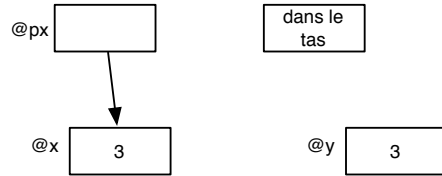
Si `x` est de type `Pointeur (t )` alors `*x` est de type `t`

Exemple :

```

int * px = (new int);
int x = 3;
px = &x;
int y = *px;

```



### 1.3 Exercice

▷ **Exercice 1** Donner le type des variables dans le programme suivant :

```

int * x = (new int);
int * * y = & x;
int z = 3;
*x = z;
*y = &z;

```

```

int * x = (new int);    // x.type = Pointeur(int)
int * * y = & x;        // y.type = Pointeur(Pointeur(int))
int z = 3;              // z.type=int
*x = z;                // (*x).type = z.type = int
*y = &z;                // (&z).type = Pointeur(int) = *y

```

▷ **Exercice 2** Donner le type des variables dans le programme suivant :

```

pointeur{
    int * x = (new int);
    * x = 3;
    int z= 18;
    int *y = & z;
    * y = *x;
}

```

```

pointeur{
    int * x = (new int); // x.type = Pointeur(int)
    * x = 3;             // (*x).type = int
    int z= 18;           // z.type=int
    int *y = & z;         // y.type = Pointeur(int)
    * y = *x;
}

```

## 2 Modification de TAM et du compilateur

### ▷ Exercice 3

1. Quelle(s) règle(s) faut-il modifier ou ajouter pour compléter l'introduction des pointeurs dans le langage ?

On simplifie les choses en limitant ce que l'on peut écrire :

- \* uniquement devant un identifiant mais en nombre quelconque ( $***x$ )
- & unique et uniquement devant un identifiant (sens de  $\&\&z$  ?)

- (a)  $TYPE \rightarrow TYPE *$
- (b)  $I \rightarrow A = E$ ; (remplace  $I \rightarrow id = E$ ;) )
- (c)  $E \rightarrow A$  (remplace  $E \rightarrow id$ )
- (d)  $A \rightarrow *A$
- (e)  $A \rightarrow id$
- (f)  $E \rightarrow null$
- (g)  $E \rightarrow (new\ TYPE)$  (on récupère une nouvelle adresse dans le tas)
- (h)  $E \rightarrow \&\ id$

Les parenthèses sont obligatoires autour de `new TYPE` pour que la grammaire reste analysable par Menhir. Cette forme de grammaire peut sembler compliquée, mais cela permet de préparer le projet.

2. Quelles modifications faut-il apporter à l'AST ?

- (a) Ajouter un cas de type : Pointeur of typ
- (b) Modifier un cas d'instruction : Affectation of affectable \* expression
- (c) Ajouter le nouveau type affectable = Ident of string | Valeur of affectable
- (d) Ajouter des cas aux expressions :
  - Affectable of affectable
  - Null
  - New of typ
  - Adresse of string

3. Modifier la passe de gestion des identifiants.

Lors de la passe de gestion des identifiants, l'AST est modifié pour remplacer les identifiants par un pointeur sur leur information.

Il faut ajouter à AstTds :

- (a) Modifier un cas d'instruction : Affectation of affectable \* expression

- (b) 

```
(* les affectables *)
type affectable =
  | Ident of Tds.info_ast
  | Valeur of affectable
```

- (c) Ajouter des cas aux expressions :

- Affectable of affectable
- Null
- New of typ
- Adresse of Tds.info\_ast

Actions à réaliser :

(a) Ajout des méthodes d'analyse des affectables

On notera que selon que l'affectable est en partie droite ou gauche d'une affectation, le traitement n'est pas même : par exemple une constante est autorisée en partie droite mais pas en partie gauche. Il y a plusieurs possibilités pour résoudre ce problème, ici nous en proposons une avec un booléen (*modif*) qui indique si l'affectable est modifié (partie gauche d'une affectation) ou pas (les autres cas).

```
let rec analyse_tds_affectable tds (a:AstSyntax.affectable) modif =  
  match a with  
  | AstSyntax.Ident n ->  
  begin  
    match chercherGlobalement tds n with  
    | None -> raise (IdentifiantNonDeclare n)  
    | Some info ->  
    begin  
      match info_ast_to_info info with  
      | InfoFun _ -> raise (MauvaiseUtilisationIdentifiant n)  
      | InfoVar _ -> Ident info  
      | InfoConst (_,v) -> if modif then raise (MauvaiseUtilisationIdentifiant n) else Ident info  
    end  
  end  
  | AstSyntax.Valeur v -> Valeur (analyse_tds_affectable tds v)
```

(b) Instruction :

```
AstSyntax.Affectation (n,e) ->  
  (* Dans affectation donc affectable en écriture *)  
  Affectation ( analyse_tds_affectable tds n true, analyse_tds_expression tds e)
```

(c) Expression :

```
(* Dans expression donc affectable en lecture *)  
|AstSyntax.Affectable aff -> Affectable (analyse_tds_affectable tds aff false)  
|AstSyntax.Null -> Null  
|AstSyntax.New t -> New t  
|AstSyntax.Adresse n -> begin  
  match chercherGlobalement tds n with  
  | None -> raise (IdentifiantNonDeclare n)  
  | Some info ->  
  begin  
    match info_ast_to_info info with  
    | InfoFun _ -> raise (MauvaiseUtilisationIdentifiant n)  
    | InfoVar _ -> Adresse info  
    | InfoConst _ -> raise (MauvaiseUtilisationIdentifiant n)  
  end  
end
```

4. Modifier la passe de typage.

Lors de la passe de typage les types sont supprimés de l'AST car ajoutés aux informations (pas de changement particulier pour les pointeurs) et l'AST est

modifié pour la résolution de surcharge (là encore pas de traitement particulier).

Il faut ajouter à AstTyp :

- (a) **Modifier un cas d'instruction** : Affectation **of** affectable \* expression
- (b) **type** affectable = inchangé
- (c) **Ajouter des cas aux expressions (inchangé)** :
  - Affectable **of** affectable
  - Null
  - New **of** typ (là on garde le type car pas d'info associé)
  - Adresse **of** Tds.info\_ast

**Actions à réaliser :**

- (a) Dans type.mli (et type.ml) **modifier le type typ pour ajouter un cas Pointeur of typ, et dans type.ml modifier est\_compatible pour prendre en compte le cas pointeur.**
- (b) **Ajout des méthodes d'analyse des affectables**

```

let rec analyse_type_affectable (aff : AstTds.affectable) : (affectable * typ) =
  match aff with
  | Ident info ->
  begin
    match info_ast_to_info info with
    | InfoVar (t, -, -) -> (Ident info, t)
    | InfoConst _ -> (Ident info, Int)
    | _ -> failwith ("Internal error: symbol not found")
  end
  | Valeur aff ->
  begin
    match analyse_type_affectable aff with
    | (naff, Pointeur t) -> Valeur (naff), t
    | _ -> raise NotAPointer
  end
end

```

- (c) **Instruction :**

```

| AstTds.Affectation (n,e) ->
  let (naff, taff) = analyse_type_affectable aff in
  let (ne, texp) = analyse_type_expression e in
  if est_compatible texp taff
  then Affectation (naff, ne)
  else raise (TypeInattendu (texp,taff))

```

- (d) **Expression :**

```

| AstTds.Affectable a -> let (na,t) = analyse_type_affectable a in (Affectable na,t)
| AstTds.Null -> Null, Pointeur Undefined
| AstTds.New t -> New t, Pointeur t
| AstTds.Adresse info ->
  begin
    match info_ast_to_info info with
    | InfoVar (t, -, -) -> (Adresse info, Pointeur t)
    | _ -> failwith ("Internal error: symbol not found")
  end
end

```

5. Modifier la passe de placement mémoire.

**Rien à faire, il faut juste définir la taille du type pointeur (1).**

6. Proposer la traduction en TAM de l'exemple de l'exercice 2. On rappelle l'existence des instructions TAM suivantes, permettant de manipuler des adresses :

- Empiler une adresse : *LOADA d[r]*
- Empiler *n* mots à partir de l'adresse laissée en sommet de pile : *LOADI (n)*
- Écrire *n* mots de la pile à l'adresse laissée en sommet de pile : *STOREI (n)*
- Allocation de mémoire : *SUBR MAlloc* (réserve dans le tas une zone de la taille laissée en sommet de pile, l'adresse obtenue est laissée en sommet de pile).

<b>PUSH 1</b>	<b>On réserve pour x</b>
<b>LOADL 1</b>	<b>Taille de l'entier</b>
<b>SUBR MAlloc</b>	<b>On réserve pour x</b>
<b>STORE (1) 0[SB]</b>	<b>On enregistre au déplacement de x, l'adresse ou sera stockée sa valeur</b>
<b>LOADL 3</b>	<b>On charge la constante</b>
<b>LOAD (1) 0[SB]</b>	<b>On charge l'adresse ou sera stockée la valeur de x</b>
<b>STOREI (1)</b>	<b>Ecrit la valeur de la constante à l'adresse de x</b>
<b>PUSH 1</b>	
<b>LOADL 18</b>	
<b>STORE (1) 1[SB]</b>	
<b>PUSH 1</b>	<b>La taille de y</b>
<b>LOADA 1[SB]</b>	<b>Empile l'adresse de z</b>
<b>STORE (1) 2[SB]</b>	<b>L'adresse de y reçoit celle de z</b>
<b>LOAD (1) 0[SB]</b>	<b>x</b>
<b>LOADI (1)</b>	<b>*x</b>
<b>LOAD (1) 2[SB]</b>	<b>y</b>
<b>STOREI (1)</b>	<b>Range</b>

**On notera que**

**LOADL 18**  
**STORE (2) 0[SB]**

**est une action identique à**

**LOADL 18**  
**LOADA 0[SB]**  
**STOREI (2)**

**Et que :**

**LOAD (2) 0[SB]**

**est une action identique à**

**LOADA 0[SB]**  
**LOADI (2)**

7. Modifier la passe de génération de code.

**Réfléchir et coder !**