Architecture, Systèmes et Réseaux

Traduction des langages

Rapport de projet Mini-Compilateur

Antoine Rey

Quentin Pointeau

16 janvier 2025

Introduction

Si aujourd'hui vous pouvez lire votre livre préféré dans votre langue maternelle alors que l'auteur est italien, allemand, anglais ou portugais, c'est bien grâce à la traduction. Que vous soyez historien, juriste, politicien ou encore chercheur, il est souvent utile de traduire vos écrits dans une langue compréhensible par vos interlocuteurs. Et ce besoin de traduction existe aussi dans le domaine de la programmation informatique. Le problème reste le même : comment donner des instructions claires à une machine alors que vous ne parlez pas sa langue ? C'est dans cette optique que sont nés les programmes de compilation, où l'intérêt est très souvent de traduire un langage de haut niveau dans un langage de bas niveau, mieux compréhensible par la machine. En outre, cette pratique de compilation est omniprésente dans le domaine de la programmation informatique et c'est pourquoi nous avons été dispensés de cours de traduction des langages. Le fil rouge de ce cours : arriver à traduire le langage RAT (haut niveau) en langage de la machine TAM (bas niveau). Nous avons alors programmé un compilateur simple de RAT vers TAM lors des séances de travaux pratiques. Ensuite, le sujet d'un projet visant à améliorer notre compilateur nous a été donné. Quatre nouveautés se devaient d'être implantées dans notre compilateur : la gestion des pointeurs, des variables globales, des variables statiques locales ainsi que celle des paramètres par défaut des fonctions. Mais avant d'attaquer l'implantation de ces nouveautés, nous vous proposons de faire un récapitulatif du principe d'un arbre abstrait, qui vous le verrez, nous sera bien fort utile tout au long de ce projet.

Abstract Syntax Tree (AST)

Tout d'abord, la phase d'analyse lexicale nous permet de vérifier que notre code source n'utilise uniquement que des mots clés du langage. Cette analyse est réalisée grâce au lexer dont le code se trouve dans le fichier `lexer.mll`. Suite à cette analyse lexicale, nous possédons une suite de tokens qui va nous être utile lors de la phase d'analyse syntaxique. Les phases d'analyse syntaxique est réalisée grâce au parser dont le code se trouve dans le fichier `parser.mly`. Cependant, au lieu de construire un arbre de dérivation qui serait bien trop indigeste lors de l'analyse syntaxique, on se propose de créer un arbre abstrait qui permettra de ne véhiculer que les informations nécessaires à l'analyse sémantique. De plus, l'AST permet d'être indépendant de l'analyseur syntaxique.
On se retrouve donc à la fin de l'analyse sémantique avec un `AstSyntax` dont l'interface est la suivante :

Interface de l'AST

```
module type Ast = sig
  type affectable (* Ajouté au début du projet pour mettre en place les pointeurs *)
  type expression
  type instruction
  type fonction
  type programme
end
```

Dans cet AST, plusieurs constructeurs représentent des entités possédant un nom comme les variables ou les fonctions par exemple. Le but de la passe de gestion des identifiants va être de créer des zones mémoires pour chacune des entités qui ont besoin d'un nom et de remplacer le nom par un pointeur vers cette zone mémoire. Cela simplifie notre AST et nous donne donc un `AstTds` où tous les identifiants sont remplacés par des pointeurs vers leur zone mémoire. Par exemple, une déclaration aura subit la modification suivante :

AstSyntax → AstTds

```
(* La déclaration dans l'AstSyntax *)
Declaration of typ * string * expression

(* La déclaration dans l'AstTds *)
Declaration of typ * Tds.info_ast * expression
```

Nous entrons ensuite dans la passe de typage où l'AST va subir plusieurs modifications. En effet, les types des fonctions et variables vont être supprimés de l'AST et seront intégrés dans les zones mémoire. De plus, cette passe de typage nous permet de gérer la résolution de surcharge afin de différencier les procédures d'affichage ou les opérateurs binaires.

AstTds → AstType

```
(* L'affichage dans l'AstTds *)
| Affichage of expression

(* L'affichage dans l'AstType *)
| AffichageInt of expression
| AffichageRat of expression
| AffichageBool of expression
```

Enfin, la dernière passe qui vient apporter des modifications à l'AST est la passe de placement mémoire. Les changements apportés concernent le placement des variables dans la mémoire, le retour des fonctions ainsi que les blocs d'instructions. En effet, il est nécessaire de connaître l'emplacement des variables pour les utiliser facilement mais aussi de retirer de la pile les variables déclarées pendant un bloc étant donné leur portée locale. Ainsi, afin de connaître la taille totale à libérer à la fin d'un bloc d'instructions ou à la fin d'une fonction et la taille à stocker pour le résultat d'une fonction, on choisit d'ajouter des entiers au type `bloc` et au constructeur `Retour`.

AstType → AstPlacement

```
(* AstTds *)
type bloc = instruction list
| Retour of expression * Tds.info_ast

(* AstType *)
type bloc = instruction list * int (* taille du bloc *)
| Retour of expression * int * int (* taille de retour et taille des paramètres *)
```

La dernière passe quant à elle permet de générer des instructions compréhensibles par la machine TAM à partir de toutes les informations que nous avons pu réunir jusque-là : c'est la passe de génération de code.

Affectables

Le but de la première étape du projet est de rediriger les identifiants vers un nouveau type `affectable` tout en conservant le même fonctionnement du compilateur. Les affectables permettront ensuite d'accueillir les règles de grammaire concernant les pointeurs. Nous avons donc dû modifier et ajouter les règles de grammaire suivantes :

- $I \rightarrow$
- ~~$\mid id = E$~~
- $\mid A = E$
- $A \rightarrow id$
- $E \rightarrow A$

Les jugements de typage associés aux règles de production ci-dessus sont les suivants :

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

$$\frac{id \in \sigma \qquad \sigma(id) = \tau}{\sigma \vdash id : \tau} \qquad \frac{\sigma \vdash A : \tau}{\sigma \vdash E : \tau}$$

Étant donné qu'il n'y a pas de nouveau mot-clé, le lexer reste identique. Ainsi, la première étape a été d'ajouter un nouveau type pour les affectables dans le parser et de modifier les règles de production pour qu'elles respectent celles citées ci-dessus.

Modification du parser

```
(* Ajout du nouveau type affectable *)
%type <affectable> a

(* Ajout des nouvelles règles de production *)
i:
| a1=a EQUAL e1=e PV    {Affectation (a1,e1)}
e:
| a1=a                  {Affectable a1}
a:
| n=ID                  {Ident n}
```

Vous aurez sans doute remarqué que nous synthétisons un nouveau type `Affectable` pour la règle de production $E \to A$. Ceci constitue en fait l'une des modifications apportées à l'`AstSyntax`. Dans cet AST, nous créons le type `affectable` afin d'accueillir les identifiants qui sont donc délogés du type `expression`, remplacés par le nouveau constructeur `Affectable`, ce qui constitue donc la redirection des identifiants de `expression` vers `affectable`. Ainsi, l'affectation se doit d'être modifiée afin de prendre dans son membre de gauche un affectable et non plus un identifiant.

Modification de l'AstSyntax

```
(* Affectables *)
type affectable =
  | Ident of string

(* Expressions *)
type expression =
  (* ... *)
  | Affectable of affectable

(* Instructions *)
and instruction =
  (* ... *)
  | Affectation of affectable * expression
```

Pour la passe de gestion des identifiants, une nouvelle fonction d'analyse des affectables est nécessaire. Cette fonction reprend dans les grandes lignes l'ancien traitement des identifiants de `analyse_tds_expression` mais la particularité réside dans l'introduction d'un nouvel argument `en_ecriture` à la fonction afin de gérer le cas d'erreur où l'utilisateur essaierait d'affecter une nouvelle valeur à une constante. Pour ce qui est de la gestion des affectables dans les expressions, nous nous contentons seulement de rediriger le traitement vers `analyse_tds_affectable` en mettant le booléen `en_ecriture` à `false` en faisant attention de vérifier si l'identifiant correspond à une constante afin de renvoyer un entier et donc de se débarrasser d'une zone mémoire dans l'AST. Enfin, pour ce qui est de la gestion de l'affectation, nous analysons l'affectable en mettant cette fois-ci `en_ecriture` à `false`.

Pour la passe de typage, on se contente seulement de rediriger la gestion des affectables. Il n'y a aucune subtilité.

Les affectables ne nécessitent pas une gestion de la mémoire particulière, la passe de placement mémoire restant ainsi inchangée.

Enfin, la passe de création de code Tam se voit modifiée avec l'ajout de `analyse_code_affectable` et le retour de l'argument `en_ecriture` afin d'effectuer un stockage en mémoire si on est en écriture ou un chargement en mémoire si on est en lecture.

Pointeurs

Maintenant que les affectables sont en place, il est temps d'implanter les pointeurs, dans une version néanmoins simplifiée par rapport à des langages *"conventionnels"* comme le C. Les règles de production pour l'implantation des pointeurs sont les suivantes :

— $A \rightarrow (* A)$
— $TYPE \rightarrow TYPE *$
— $E \rightarrow$
— $\quad | \, null$
— $\quad | \, (new \, TYPE)$
— $\quad | \, \& \, id$

Les jugements de typage associés aux règles de production ci-dessus sont les suivants :

$$\frac{\sigma \vdash a : \text{Pointeur}(\tau)}{\sigma \vdash *a : \tau} \qquad \frac{\sigma \vdash id : \tau}{\sigma \vdash \&id : \text{Pointeur}(\tau)}$$

$$\frac{\sigma \vdash \text{T} : \tau}{\sigma \vdash new \, \text{T} : \text{Pointeur}(\tau)} \qquad \frac{}{\sigma \vdash null : \text{Pointeur}(Undefined)}$$

Contrairement à l'introduction des affectables, l'introduction des pointeurs s'accompagne de trois nouveaux tokens. En effet, le token associé à '`*`' (`MULT`) est déjà présent vu qu'il est utilisé pour annoncer une multiplication. Nous devons donc modifier le lexer comme suit.

Modification du lexer

```
(* table des mots-clés *)
[
  (* ... *)
  "new",  NEW;
  "null", NULL;
]

(* caractères spéciaux de RAT *)
(* ... *)
| "&"   { AMPERSAND }
```

Ensuite, les règles de production doivent être ajoutées dans le parseur.

Modification du parser

```
(* Ajout des nouvelles règles de production *)
typ:
| t=typ MULT            {Pointeur t}
e:
| NULL                  {Null}
| PO NEW t=typ PF       {New t}
| AMPERSAND n=ID        {Adresse n}
a:
| PO MULT a1=a PF       {Deref a1}
```

Vous remarquerez la synthèse d'un type `Pointeur` que l'on ajoute aux types disponibles dans RAT. De cet ajout du type découle la modification des fonctions `est_compatible` et `getTaille`.

Ajout du type Pointeur

```
type typ = Bool | Int | Rat | Undefined | Pointeur of typ

let rec est_compatible t1 t2 =
  match (t1, t2) with
  (* ... *)
  | Pointeur _, Pointeur Undefined -> true          (* null -> Pointeur Undefined *)
  | Pointeur x, Pointeur y -> est_compatible x y

let getTaille t =
  match t with
  (* ... *)
  | Pointeur _ -> 1
```

Les attributs synthétisés lors de la phase d'analyse sémantique apparaissent donc dans l'`AstSyntax`.

Modification de l'AstSyntax

```
(* Affectables *)
type affectable =
  (* ... *)
  | Deref of affectable

(* Expressions *)
type expression =
  (* ... *)
  | Adresse of string
  | New of typ
  | Null
```

Lors de la passe de gestion des identifiants, on gère le cas `Deref` en analysant l'affectable et en propageant le booléen `en_ecriture` car le déréférencement peut aussi bien concerner l'affectation que l'utilisation de l'affectable déréférencé. Pour le cas `Adresse`, on cherche globalement si l'identifiant a déjà été déclaré et on vérifie que l'`info` associé à l'identifiant est bien une `InfoVar`. En ce qui concerne les cas `New` et `Null`, on se contente juste de propager, aucune modification ou effet de bord n'est effectué.

Pour la passe de typage, on va s'assurer que le déréférencement d'un affectable n'est pas illégal, c'est-à-dire qu'un déréférencement ne s'effectue que sur un pointeur. De même, on s'assure que la récupération d'une adresse mémoire s'effectue uniquement sur un pointeur. En effet, le choix a été fait ici de ne pas implanter la récupération d'adresse de constantes ou de fonctions car dans l'état actuel du langage RAT, nous n'avons rien pour les exploiter. Pour les cas `New t` et `Null`, on renvoie logiquement un `Pointeur t` et un `Pointeur Undefined`.

En ce qui concerne la passe de placement mémoire, rien ne change car les nouvelles règles de production ajoutées ne concernent pas les instructions et aucun placement mémoire dans la pile n'est effectué. On pourrait se poser la question de la gestion de la mémoire dans le tas, mais celle-ci sera gérée dans la passe de génération de code.

La passe de génération de code apporte justement un grand lot de nouveautés avec la gestion des pointeurs. La gestion des déréférencements apporte une nouvelle problématique : comment gérer les déréférencements successifs ? Le simple `load` ou `store` dans le cas d'un identifiant n'est plus suffisant. Nous devons être capables de pouvoir déréférencer un nombre quelconque de fois un pointeur, dans la limite de sa profondeur. Prenons un exemple : une variable de type `int****` doit pouvoir être déréférencée par l'utilisateur une, deux, trois, ou quatre fois. Afin de pallier ces nouvelles contraintes, le choix a été fait d'ajouter deux paramètres `acc` et `prof` à notre fonction `analyse_code_affectable` qui nous permettent respectivement de stocker les `loadi` successifs à effectuer après le `loada` avant de faire le `loadi` ou le `storei` final en fonction du paramètre `en_ecriture` et de connaître la profondeur d'appels des déréférencements. La connaissance de la profondeur d'appels nous permet de connaître le type obtenu après les n déréférencements successifs, ce qui est essentiel pour connaître la taille de ce type et donc effectuer le `loadi n` ou `storei n` final. À ce sujet, l'implantation d'une nouvelle fonction `type_prof` a été nécessaire pour récupérer le type obtenu après les déréférencements successifs.

Fonction type_prof

```
(* type_prof : typ -> int -> typ *)
(* Renvoie le type à la profondeur indiquée d'une variable *)
let rec type_prof t i =
  if i = 0 then t
  else
    match t with
    | Pointeur tt -> type_prof tt (i - 1)
    | _ -> failwith "Profondeur donnée trop importante"
```

Pour le cas de l'adresse, c'est bien plus simple car il nous suffit de vérifier si l'`info` est bien une `InfoVar` et dans ce cas effectuer un `loada` avec le déplacement et le registre récupérés dans l'`info`. Le `New` est géré en chargeant dans la pile un entier correspondant à la taille que l'on souhaite allouer dans le tas puis en appelant la fonction `MAlloc` de la machine TAM. L'expression `Null` quant à elle ne correspond à rien dans la machine TAM donc on renvoie une chaîne de caractères vide.

Variables globales

Après l'implantation des pointeurs, nous nous sommes attaqués à celle des variables globales. La contrainte que nous nous sommes imposée pour éviter de trop complexifier cette implantation est celle suggérée dans le sujet, c'est-à-dire qu'on impose aux variables globales d'être déclarées au tout début du fichier, avant même la déclaration des fonctions. Nous nous devons alors de modifier la règle de production du programme et d'ajouter une nouvelle règle de production comme suit :

— $PROG \rightarrow VAR* \ FUN* \ id \ BLOC$
— $VAR \rightarrow static \ TYPE \ id = E$

Les jugements de typage associés aux règles de production ci-dessus sont les suivants :

$$\frac{\sigma \vdash VAR : void, \sigma' \qquad \sigma@\sigma' \vdash FUN : void, \sigma'' \qquad \sigma''@\sigma'@\sigma \vdash BLOC : void, \sigma'''}{\sigma \vdash VAR \ FUN \ id \ BLOC : void, \sigma'''@\sigma''@\sigma'}$$

$$\frac{\sigma \vdash TYPE : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash static \ TYPE \ id = E : void, [id, \tau]}$$

Cela ne vous aura pas échappé, un nouveau mot-clé fait son apparition dans le langage RAT : `static`. Nous devons donc modifier le lexer pour faire apparaître ce mot-clé dans nos tokens.

Modification du lexer

```
(* table des mots-clés *)
[
  (* ... *)
  "static",  STATIC;
]
```

Il aurait été complexe et fastidieux d'intégrer la déclaration des variables globales à celles des variables *"classiques"*. Nous avons donc fait le choix d'intégrer un nouveau type dans notre parser.

Modification du parser

```
%type <variableG> var

var : STATIC t=typ n=ID EQUAL e1=e PV   {DeclarationG (t,n,e1)}
```

De même, le choix a été fait de séparer les différentes déclarations dans l'`AstSyntax` dû à la création du nouveau type `variableG`. De plus, afin d'intégrer les variables globales au programme, il fallait, tout comme pour les fonctions, ajouter une liste de variables globales au type `programme`.

Modification de l'AstSyntax

```
(* Déclaration de variable globale représentée par son type, son nom et l'expression
 ↪  d'initialisation *)
type variableG = DeclarationG of typ * string * expression

(* Modification du type programme *)
type programme = Programme of variableG list * fonction list * bloc
```

Nous avons donc créé une nouvelle fonction d'analyse des variables globales pour la passe de gestion des identifiants. Celle-ci reprend presque trait pour trait l'analyse d'une déclaration classique à la différence qu'on ajoute cette fois-ci l'`info` dans la TDS mère et non dans une de ses sous TDS, afin que l'identifiant soit accessible pour n'importe quel bloc dans la suite du programme. En outre, la modification du type `programme` entraîne aussi la modification de la fonction d'analyse du programme. Ainsi, on ajoute un `List.map` pour analyser toutes les variables globales.

Maintenant que la table des symboles est prête, nous avons estimé qu'il serait judicieux de fusionner le traitement des variables globales avec celui des variables classiques car le traitement sur les types, le placement mémoire et la génération de code seront identiques. Ainsi, le type `variableG` disparaît de l'`AstTds` et le type `programme` est modifié en conséquence.

Modification de l'AstTds

```
type programme = Programme of bloc * fonction list * bloc
```

L'analyse d'un `AstSyntax.DeclarationG` renvoie donc un `AstTds.Declaration`.

Pour la passe de typage, on se contente d'ajouter un `List.map` pour analyser le type des variables globales, donc vérifier si les déclarations ne sont pas illégales.

Pour la passe de placement mémoire, on analyse le bloc des variables globales avant le bloc principal afin de s'assurer que les variables globales soient stockées à la base de la pile.

De même pour la passe de génération de code où on analyse le bloc des variables globales avant le bloc du programme principal. La seule différence ici est que nous traitons les variables globales comme un bloc de taille 0 car à la fin de son analyse nous ne voulons pas les `POP` car nous souhaitons garder leurs valeurs pour la suite. Nous verrons par la suite que la fonction `analyser` a bien changé lors de l'implantation des variables statiques locales.

Variables statiques locales

Justement, nous voilà arrivés à l'implantation des variables statiques locales. Cette fois-ci, une seule règle de production doit être ajoutée.

— $I \rightarrow static\ TYPE\ id = E$

On rappelle le jugement de typage de cette règle de production.

$$\frac{\sigma \vdash TYPE : \tau \qquad \sigma \vdash E : \tau}{\sigma \vdash static\ TYPE\ id = E : void, [id, \tau]}$$

Les variables statiques locales utilisent exactement les mêmes mots-clés que pour la déclaration des variables globales, le lexer reste donc identique. Cependant, nous devons ajouter la règle de production des variables statiques locales au parser.

Modification du parser

```
(* Ajout de la nouvelle règle de production *)
i:
| STATIC t=typ n=ID EQUAL e1=e PV   {StatiqueL (t,n,e1)}
```

Comme à l'accoutumée, nous modifions l'`AstSyntax` pour y faire apparaître le nouveau constructeur permettant de traiter le cas des variables statiques locales.

Modification de l'AstSyntax

```
and instruction =
  (* ... *)
  (* Déclaration d'une variable statique locale *)
  | StatiqueL of typ * string * expression
```

Lors de la passe de gestion des identifiants, lorsqu'on tombe sur une déclaration de variable statique locale, on effectue presque trait pour trait le traitement d'une déclaration de variable classique à l'exception près qu'on vérifie qu'elle n'a pas été déclarée dans le bloc principal (`main`).

Pour ce qui est de la passe de typage, nous traitons les déclarations de variables statiques locales exactement de la même manière que les déclarations de variables statiques.

En ce qui concerne la passe de placement mémoire, de gros changements ont été effectués pour accueillir les variables statiques locales. En effet, nous avons choisi de stocker ces variables à la base de la pile, juste après les variables globales, afin que les variables statiques locales ne soient pas effacées de la pile à cause du `pop` de la taille du bloc. Ainsi, nous avons ajouté un argument aux fonctions `analyse_placement_fonction`, `analyse_placement_bloc` et `analyse_placement_instruction` afin de retenir la place dans la pile de la prochaine variable statique locale à stocker. Cette information est alors propagée par le traitement de toutes les

instructions, et notamment celles qui seraient susceptibles d'accueillir des déclarations de variables statiques locales (c'est-à-dire les instructions contenant des blocs, comme la conditionnelle). Lorsque nous traitons l'instruction de déclaration d'une variable statique locale, nous modifions l'adresse de la variable à l'emplacement `emplSL "SB"`, puis nous incrémentons emplSL de la taille de la variable mais aussi de la taille d'un booléen. La raison de cet ajout est la suivante : on veut garder un booléen en mémoire qui est associé à notre variable statique locale pour qu'il nous indique si la variable a déjà été initialisée ou non. En effet, l'initialisation d'une variable statique locale se fait lors que premier appel à la fonction donc nous avons besoin de connaître l'état de l'initialisation pour ne pas manquer l'initialisation ou au contraire effectuer plusieurs initialisations.

Enfin, lors de la passe de génération de code, nous chargeons en mémoire le booléen associé à la variable statique locale. Si ce booléen est à faux, c'est que la variable n'a pas déjà été initialisée. Donc nous analysons l'expression d'initialisation et nous stockons le résultat ainsi qu'un booléen à vrai (pour indiquer que l'initialisation a été effectuée) aux emplacement prévu à cet effet. Si le booléen est à vrai, on ne fait rien.

Paramètres par défaut

Nous arrivons enfin à l'implantation des paramètres par défaut dans les fonctions. Pour cette dernière partie, 3 nouvelles règles de productions sont ajoutées à la grammaire.

- $DP \rightarrow \Lambda$
- $\quad \mid TYPE\ \ id\ \ \langle\ D\ \rangle ?\ \langle , TYPE\ \ id\ \ \langle\ D\ \rangle ?\ \rangle \star$
- $D \rightarrow = E$

Les jugements de typages de ces nouvelles règles de production sont les suivantes :

$$\frac{\sigma \vdash TYPE_1 : \tau_1 \qquad \sigma \vdash D : \tau_1 \qquad \ldots \qquad \sigma \vdash TYPE_n : \tau_n \qquad \sigma \vdash D_n : \tau_n}{\sigma \vdash TYPE_1\ \ id_1\ \ D_1, \ldots, TYPE_n\ \ id_n\ \ D_n : \tau_1 \times \ldots \times \tau_n, [(id_1, \tau_1); \ldots; (id_n, \tau_n)]}$$

$$\frac{\sigma \vdash E : \tau}{\sigma \vdash = E : \tau} \qquad\qquad \frac{}{\sigma \vdash \Lambda : void}$$

Les paramètres par défaut n'utilisent aucun nouveau mot-clé, le lexer reste donc identique. Cependant, nous devons modifier la règle de production des paramètres et ajouter une nouvelle règle de production au parser.

Modification du parser

```
(* Modification du type param *)
%type <typ*string*(expression option)> param

(* Modification sur la règle des paramètres *)
param : t=typ n=ID d1=d {(t,n,d1)}

(* Ajout des nouvelles règles de production *)
d:
| EQUAL e1=e                          {Some e1}
|                                     {None}
```

Nous avons fait le choix de représenter les paramètres par défaut par des expressions optionnelles, en synthétisant `Some` l'expression si celle-ci est donnée ou `None` si elle ne l'est pas. Cette liste d'`expression option` est tout d'abord enregistré dans la liste des informations de paramètres dans la déclaration de fonction.

Modification de l'AstSyntax

```
(* On ajoute la liste des paramètres par défaut à l'appel de fonction *)
type fonction =
  | Fonction of typ * string * (typ * string * expression option) list * bloc
```

Pour la passe de gestion des identifiants, on commence par vérifier que les paramètres de déclarations de fonction sont dans le bon ordre, c'est à dire que les paramètres par défaut sont bien situés après les paramètres classiques, sinon on lève l'erreur `ParamDefautPasALaFin`. Nous avons aussi fait le choix de faire passer les paramètres par défaut de chaque fonction aux appels de fonction correspondant. Toutefois, on ne peut pas faire passer cette `expression option list` grâce aux `InfoFun` car cela crée un cycle récursif d'importation à la compilation. Nous avons donc créé un nouveau module, `tdpd` pour Table Des Paramètres par Défaut, qui permet d'associer

une fonction à sa liste de paramètres par défaut et de la récupérer. Ainsi, lors de l'analyse d'une déclaration de fonction on enregistre la liste de paramètres par défaut dans une tdpd pour par la suite, lors de l'analyse d'un appel à cette fonction, donner à `AppelFonction` la liste de paramètres par défaut.

Modification de l'AstTds

```
(* On fait passer la liste des paramètres par défaut aux appels de fonction *)
type expression =
  | AppelFonction of Tds.info_ast * expression list * expression option list
```

Pour ce qui est de la passe typage, lors de l'analyse des fonctions, on vérifie que le type des expressions des paramètres par défaut correspond au type réel des paramètres. Lors de l'analyse des expressions, nous avons modifié le traitement d'`AppelFonction`. En effet, on commence par vérifier que le nombre d'arguments donné à la fonction couplés au nombre de paramètres par défaut suffit pour appeler la fonction, sinon on lève `MauvaisNombreArguments`. Ensuite, on analyse les expressions données en arguments et les expressions par défaut et on les concatène pour la suite.

Les passes Placement et Code ne changent donc pas.

Conclusion

En conclusion, l'implantation des nouvelles fonctionnalités aura permit au langage RAT d'être plus complet et plus agréable à utiliser. En effet, nous avons implanté les affectables dans l'optique d'implanter plus aisément les pointeurs. Cependant, nous avons rencontré quelques difficultés quant aux déréférencements successifs de pointeurs car nous avons dû implanter une nouvelle fonction `type_prof` afin d'obtenir la bonne taille à indiquer à l'instruction `storei` ou `loadi`. Ensuite, nous nous sommes attaqués aux variables globales qui ont suivi un traitement très semblable si ce n'est quasiment similaire aux déclarations des variables classiques. Par la suite, nous avons implanté les variables statiques locales qui nous auront donné du fil à retordre à cause de la façon dont nous avons dû les placer dans la pile. En effet, nous avons choisi de les placer juste après les variables globales et donc il nous a fallu un argument `emplSL` pour presque toutes les fonctions d'analyse de la place de placement mémoire afin d'indiquer le placement dans la pile de la prochaine variable statique locale à stocker. Enfin, nous avons implanté les paramètres par défaut des fonctions. Nous avons là aussi rencontré un problème : la boucle d'importation des modules `Ast` et `Tds` comme expliqué plus haut dans le rapport. Nous avons finalement réussi à surpasser ces problèmes et nous sommes donc arrivés à faire fonctionner un compilateur de RAT vers TAM qui prend en compte les pointeurs, les variables globales, les variables statiques locales et les paramètres par défaut.