# Traduction des Langages

Quentin Fraty
Nathan Maillet

## 1 Introduction

## 2 Mutation de *tds.ml* en *mtds*

**La première étape avant de se lancer sur le projet était de modifier *tds.ml* afin de ne plus le modifier, même si par la suite nous voulions modifier notre raisonnement, comme par exemple sur les pointeurs.**

***mtds.ml* a donc pour but de généraliser *tds.ml*. Nous sommes donc passer d'Identifiants *string* en *'a*, ce qui explicite l'apect monadique de la tds sans pour autant la contraindre, ce qui aurait été un apport négligeable en OCaml 4.**

## 3 Pointeurs

## 4 Bloc else optionnel

**Grâce à la façon choisie initialement pour traiter les blocs conditionnels, il fut assez simple d'implémenter cette fonctionnalité. En effet, lors du parsing, il suffit de créer un *AstSyntax.Conditionnelle* avec un bloc else vide, et de le traiter comme un bloc conditionnel classique. Le typage est inchangé, et le code produit est le même que si le bloc else était présent.**

**Cette solution permet de minimiser la redondance de code, et de ne pas avoir à traiter le bloc else comme un cas particulier, avec comme seul bémol que la passe de génération de code rajoute dans tous les cas un label pour le else, qu'il soit vide ou non. Cela n'a aucun impact sur l'exécution du programme.**

**Au niveau du jugement de typage, dans la mesure où le bloc else est vide, celui-ci peut être résumé comme suit :**

$$\frac{\sigma \vdash E : \text{bool} \quad \sigma, \tau_r \vdash \text{BLOC} : \text{void}}{\sigma, \tau_r \vdash \text{if } E \text{ BLOC} : \text{void}, []}$$

## 5 Conditionnelle ternaire

**Cette fonctionnalité ajoutée au langage Rat a nécessité de créer un nouveau type d'expression, *AstSyntax.Ternaire*. Tout d'abord, il a fallu modifier la grammaire pour accepter cette nouvelle expression :**

**Au niveau du lexer, il a fallu ajouter le symbole ? et : dans la liste des symboles à accepter. Au niveau du parser, il a fallu ajouter une règle pour accepter cette nouvelle expression :**

```
| PO e1=e QMARK e2=e COLON e3=e PF  {Ternaire (e1,e2,e3)}
```

**Ensuite, les différentes passes du compilateur ont successivement vérifié les choses suivantes :**

- **La conformité des identifiants utilisés dans les trois expressions de la conditionnelle ternaire.**
- **Le fait que la premiere expression soit bien booléenne, et que les deux autres soient de même type. Cela peut être résumé par le jugement de typage suivant :**

$$\frac{\sigma \vdash E : \text{bool} \quad \sigma \vdash E_1 : \tau \quad \sigma \vdash E_2 : \tau}{\sigma \vdash (E \ ? \ E_1 : E_2) : \tau}$$

**Enfin, la génération du code associé à un ternaire se rapproche à celui d'une instruction conditionnelle, dans la mesure où la valeur mise en sommet de pile est celle de l'une des deux expressions : l'utilisation de jump est alors nécessaire.**

```
| AstType.Ternaire(e1, e2, e3) ->
      let labElse = getEtiquette ()
      and labEndIF = getEtiquette () in
      ast_to_tam_expression e1
      ^ jumpif 0 labElse
      ^ ast_to_tam_expression e2
      ^ jump labEndIF
      ^ label labElse
      ^ ast_to_tam_expression e3
      ^ label labEndIF
```

# 6 Loop à la Rust

**Les loop à la rust sont une extension du langage Rat qui permet de créer des boucles infinies, avec la possibilité de les casser à l'aide du mot clé `break` et de terminer prématurément une itération à l'aide du mot clé `continue`.**

**TODO : grammaire**

**Comme nous allons le voir, toute la complexité des boucles à la rust est liée à la gestion des labels, qui est exclusivement gérée par la passe de gestion d'identifiants. Tout d'abord, il a été choisi de créer un nouveau type de données insérables dans la table des symboles.**

**En effet, il est nécessaire de stocker les informations liées à une boucle pour pouvoir la terminer, ou la recommencer lorsqu'un break ou continue est utilisé.**
**Cependant, et ce comme énoncé dans le sujet, il fallait donner la possibilité à l'utilisateur d'imbriquer des fonctions de même label.**
**Pour ce faire, ce nouvel élément de la TDS associe à un label une InfoBoucle, qui contient en réalité une liste d'informations associées à ce label.**
**Cela permet de prendre en compte le cas où deux boucles de même label sont imbriquées, et de pouvoir les distinguer.**

**Un deuxième choix de conception réside dans le choix des informations stockées dans la TDS pour les boucles.**
**Tout d'abord, à toute boucle est associé un identifiant pour la TDS :soit celui donné par l'utilisateur, soit un identifiant uniqué généré par le compilateur de manière analoque à la génération de labels lors de la passe de génération de code.**

```
let giveID =
  let num = ref 0 in
  fun () ->
    num := (!num)+1 ;
    "id"^((string_of_int (!num)))
```

**Ensuite, l'information associée à une boucle est le couple (label de début, label de fin), qui sont générés par le compilateur à partir du label précédent.**
**Ainsi, lorsque deux boucles de même label sont utilisées, l'infoboucle associée au label est une liste de couples (label de début, label de fin), dont la tête est le couple associé à la plus imbriquée. Grâce à la récursivité du langage fonctionnel, il est possible de récupérer le label de début/fin de la boucle la plus imbriquée de manière efficace, ce qui est la logique choisie pour le break et le continue.**

**De plus, il fallait donner la possibilité à l'utilisateur de ne pas utiliser de labels pour les loops, break et continue. Le comportement choisi devait alors être que tout break / continue aurait un effet sur la boucle courante, et dans le cas d'imbrication de boucles, celle la plus imbriquée.**
**Pour pouvoir implémenter cela, il a été choisi de *décorer l'arbre* avec l'infoboucle associée à la boucle courante pour pouvoir associer au break / continue la bonne boucle.**

**Suite à cette passe, celle de typage est comparativement triviale, dans la mesure où celle-ci revient à une analyse du typage d'un bloc. Voici les jugements de typage associé :**

$$\frac{\sigma, \tau_r \vdash \text{BLOC} : \text{void}}{\sigma, \tau_r \vdash \text{loop BLOC} : \text{void}, []} \text{ et } \frac{id :: \sigma, \tau_r \vdash \text{BLOC} : \text{void}}{\sigma, \tau_r \vdash \text{define id : loop BLOC} : \text{void}, []}$$

**La passe de placement mémoire, de manière analogue, revient à analyser le placement mémoire des variables du bloc de la boucle.**

**En revanche, la passe de génération de code s'avéra poser des difficultés que nous n'avions pas anticipé. En effet, chaque itération de boucle doit écrire ses variables locales au même endroit, ce qui pose des problèmes lorsqu'un break ou un continue sont utilisés : alors que le TantQue implémenté en Rat assurait l'exécution entière de son bloc courant avant de se finir, ce n'est pas le cas de la boucle à la Rust.**
**Ainsi, en imaginant qu'un break prématuré puisse faire 'éviter' une déclaration de variable (et donc un push de la pile), il s'avérait nécessaire d'avoir le nombre de déclarations dans la boucle actuelle en tant que paramètre hérité (décoration d'arbre). Cependant même avec une telle chose, le cas d'imbrication de boucle ne semblait pas traitable, dans la mesure où un break peut aussi faire sortir d'une boucle moins imbriqué, comme dans l'exemple suivant :**

```
main {
        int x = 3;
        define l : loop {
                x++;
                loop {
```

```
                int y = 5;
                break l;
                bool z = true;
            };
            rat r = [3/2];
        };
    }
```

**Ici, il faudrait *se rappeler* au niveau du break l que cela revient à pop 1, et pas 2 ou 3 qui serait obtenu si on prenait toutes les déclarations réalisées.**
**N'ayant pas de solution simple à ce problème et par manque de temps, il a été choisi de ne pas pop le contexte d'une boucle en cas de break ou continue. Ce choix, bien que semblant fonctionner, peut poser des problèmes de pile si un utilisateur venait à réaliser un *continue inconditionnel* dans une boucle, ce qui dupliquerait le contexte à chaque itération.**

# 7 Surcharge de fonctions

# 8 Incréments

**Il a été choisi d'implémenter dans le langage Rat de nouvelles fonctionnalités qui puissent êtres traitées au niveau du parseur. C'est pourquoi l'incrémentation comme en C (variable++ et ++variable) a été partiellement ajoutée : contrairement à C, ces deux éléments sont considérés comme des instructions à part entière, ce qui limite leur intérét à celui d'un compteur. Voici les lignes associées dans le fichier Parser :**

```
| PLUS PLUS n=r PV
{Affectation (n,Binaire (Plus,Identifiant n,Entier 1))}
| n=r PLUS PLUS PV
{Affectation (n,Binaire (Plus,Identifiant n,Entier 1))}
```

De manière similaire, l'opérateur '+=' a été ajouté au langage Rat, et ce grâce à une transformation directement dans le parseur :

```
| n=r PLUS EQUAL e1=e PV
{Affectation (n,Binaire (Plus,Identifiant n,e1))}
```

**Ainsi, dans la mesure où ces instructions sont 'converties' en combinaisons d'éléments simples compris par notre compilateur, aucune passe n'a à être adaptée pour prendre en compte ces opérations.**

# 9 Affichage des erreurs

# 10 Conclusion