# MV6 2015 – Projet de Programmation

Benoît Valiron

## 1 Introduction

Le but de ce projet est de développer une machine virtuelle pour le langage Tyrme (défini plus loin) sur le principe de ce qui a été vu en cours. Plus spécifiquement, étant donné

- la structure de la MV de Tyrme,
- son jeu d'instructions et la forme du code-octet,
- la grammaire de terme de Tyrme,
- un lexeur et un parseur pour Tyrme,

l'objectif de ce projet est de concevoir

- la machine virtuelle,
- un assembleur/désassembleur pour les instructions,
- un compilateur de Tyrme vers les instructions.

Dans un deuxième temps, vous pourrez travailler sur les extensions proposées en Section 3.3.

## 2 Les données du problème

Dans cette section, on décrit le langage, la machine, le jeu d'instructions et le code-octet.

### 2.1 Le langage Tyrme

Le langage Tyrme est une extension du langage Myrte vu en classe. Il est décrit par la grammaire de termes suivante.

```
type value =
    | Int of int
    | Bool of bool
    | String of string
    | Unit

type binop = Add | Eq | And | Cat | App

type var = string
```

```
type expr =
    | Const of value
    | Binop of binop * expr * expr
    | If of expr * expr * expr
    | Let of var * expr * expr
    | Letf of var * var * expr * expr
    | Print of expr * expr
    | Pair of expr * expr
    | Fst of expr
    | Snd of expr
```

Par rapport à Myrte, vous noterez les extensions suivantes :

1. deux nouveaux types de valeurs :
   - les chaines de caractères (`string`), munies d'un opérateur binaire de concaténation de chaines (`Cat`),
   - une valeur « vide » : `Unit` ;
2. un opérateur de test booléen ;
3. un constructeur de variable locale (`Let`) ;
4. un constructeur de fonctions *unaire* (`Letf`) : chaque fonction ne peut prendre qu'un seul argument, et un opérateur binaire `App` pour appliquer une fonction à son argument ;
5. un opérateur d'écriture de `string` sur la sortie standard ;
6. un ensemble de constructeurs pour gérer les paires : `Pair` pour en construire, `Fst` pour projeter la première composante, et `Snd` pour projeter la deuxième composante.

La sémantique de l'opérateur `Print` est la suivante : `Print("abc", e)` imprime la chaine de caractères `"abc"` sur la sortie standard, puis execute `e`. Donc par exemple, `Print ("abc", 2+3)` s'évalue en `5`, après avoir imprimer la chaine de caractères `"abc"`

Pour faciliter l'écriture d'expressions, un lexeur et un parseur sont fournis.

## 2.2 La machine virtuelle

La machine à implémenter est une machine à a-pile, plus un tas. Chaque mot mémoire est, comme en OCaml, soit un entier, soit un pointeur vers une zone du tas. Cette zone, un bloc, est soit une chaîne de caractères, soit la paire d'un tag et d'un tableau contiguë de mots mémoire.

Vous pourriez, par exemple, représenter
- **la notion de bloc** par un type `bloc` qui contient les différentes possibilités : `string`, ou paire ⟨ tag, tableau ⟩ ;
- **le tas** par un tableau d'éléments de type `bloc` ;
- **un pointeur** par un entier, indice de l'un des éléments du tableau représentant le tas.

Cette technique fonctionnerait pour l'implémentation de la machine avec n'importe quelle langage (C,java,...). L'inconvénient est que vous seriez alors responsable de gérer la mémoire.

Une autre option consiste à utiliser le fait que Ocaml a un ramasse-miettes : on va donc simplement utiliser le tas Ocaml, et donc déléguer à Ocaml la gestion du tas de Tyrme. C'est cette option qui vous est proposée d'utiliser dans le projet.

Dans cette méthode, le tas n'est pas représenté explicitement. En lieu et place de pointeur, on va simplement mettre la structure qui nous intéresse (par exemple une chaine de caractère, ou une paire $\langle$ tag, tableau $\rangle$).

Donc l'accumulateur et les éléments de la pile de la machine ne seront plus simplement des entier (`int`), mais des éléments du type

```
type mot =
 | PointBloc of tag * quelque_chose
 | PointStr of string
 | MotInt of int
```

Dans le cas classique où par exemple l'accumulateur contient une valeur entière, on utiliserait `MotInt`. Dans le cas ou l'accumulateur devrait contenir un pointeur, on utiliserait l'un des deux autres constructeurs.

## 2.3 Le jeu d'instructions

Le jeu d'instructions de la machine est le suivant.

- `halt` (opcode 0) arrête l'exécution de la machine ;
- `push` (opcode 1) empile le contenu de l'accumulateur sur la pile ;
- `print` (opcode 2) affiche le contenu de la chaîne de caractère pointée par l'accumulateur ;
- `acc n` (opcode 4) copie la valeur de la n-ème case du sommet de la pile dans l'accumulateur (`acc 0` rends la tête de la liste) ;
- `const n` (opcode 5) met l'entier `n` dans l'accumulateur ;
- `pop n` (opcode 7) dépile et jette les n premières cases de la pile ;
- `str s` (opcode 14) met dans l'accumulateur un pointeur vers la chaîne de caractères s.
- `binop b` (opcode 13) effectue l'opération binaire dont l'opcode est l'entier b : si l'accumulateur vaut `x`, le sommet de la pile vaut `y`, et si `b` correspond à `op`, alors `binop b` mets la valeur `(y op x)` dans l'accumulateur. **À la différence de Myrte, le sommet de la pile lu est dépilé**.
  Les opération binaires sont les suivantes :
  - `add` addition entière (opcode 15)
  - `sub` soustraction entière (opcode 16)
  - `mul` multiplication entière (opcode 17)
  - `div` division entière (opcode 18)
  - `eqi` égalité d'entiers (opcode 19)
  - `cat` concaténation de chaînes (opcode 20)

En outre, ces instructions augmentent le PC de une unité.

On a aussi les instructions de branchement :

- `branchif o` (opcode 8) saute o instructions en avant si l'accumulateur est entier et différent de 0 (si o est négatif, on saute en arrière) ;
- `branch o` (opcode 9) saute o instructions en avant (ou en arrière si o est négatif) ;

Les instructions suivantes gèrent les blocs :

- `makeblock (t, n)` (opcode 11) met dans l'accumulateur un bloc de tag t et contenant les n valeurs au sommet de la pile, qui sont dépilées :

| A | S | | | | | | |
|---|---|---|---|---|---|---|---|
| `x` | `a1` | `a2` | `...` | `an` | `b1` | `b2` | `...` |
| `(t,[a1;a2;...;an])` | `b1` | `b2` | `...` | | | | |

- `getblock n` (opcode 10) met dans l'accumulateur la n-ème case du bloc pointé par l'accumulateur. Par exemple, `getblock 3` effectue la transformation suivante :

| A | S | | |
|---|---|---|---|
| `(t,[a1;a2;...;an])` | `b1` | `b2` | `...` |
| `a4` | `b1` | `b2` | `...` |

**Notez que le bloc est indicé à partir de zéro !**

- `closure (n, o)` (opcode 12) alloue un bloc de taille n + 1, de tag 88, contenant l'indice de l'instruction située o instructions en avant, suivi des n premières cases de la pile, qui **ne sont pas** dépilées :

| A | S | | | | | | |
|---|---|---|---|---|---|---|---|
| `x` | `a1` | `a2` | `...` | `an` | `b1` | `b2` | `...` |
| `(88,[pc+o;a1;a2;...;an])` | `a1` | `a2` | `...` | `an` | `b1` | `b2` | `...` |

On dit que le bloc forme l'environnement de la fermeture.

Toutes ces instructions augmentent aussi le PC de une unité.

Les deux dernières instructions de la machine sont utiles pour traiter la compilation des fonctions :

- `apply` (opcode 3) appelle la fonction dont la fermeture est contenue dans l'accumulateur, avec comme argument le contenu du sommet de la pile :

| PC | A | S | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `pc1` | `(88,[pc2;a1;a2;...;an])` | `b1` | `b2` | `...` | | | | | |
| `pc2` | `(88,[pc2;a1;a2;...;an])` | `b1` | `a1` | `a2` | `...` | `an` | `pc1+1` | `b2` | `...` |

- `return n` (opcode 6) dépile et jette les n premières cases de la pile et retourne à la fonction appelante :

| PC | A | S | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| `pc1` | `c` | `a1` | `a2` | `...` | `an` | `pc2` | `b1` | `b2` | `...` |
| `pc2` | `c` | `b1` | `b2` | `...` | | | | | |

## 2.4 Code-octet

Le bytecode (code-octet) contient deux sortes de mots :

- des mots de 8 bits (1 octet), représentant soit des entiers non-signés (entre 0 et 255), soit des représentations ASCII de caractères. Pour lire et écrire la représentation binaire ASCII sur 8 bits d'un caractère, on peut utiliser les fonctions Ocaml standards :
  `val input_char : in_channel -> char`
  `val output_char : out_channel -> char -> unit`
  Pour passer de la représentation entière 8 bits d'un entier (par ex. 97) au caractère ocaml de type char (par ex. 'a'), on peut utiliser
  `val int_of_char : char -> int`
  `val char_of_int : int -> char`
- des mots de 32 bits (4 octets), représentant des entiers signés. L'encodage est celui de Ocaml, et on peut lire et écrire des entiers signés en binaires sur 32 bits [1] en utilisant les fonctions Ocaml :
  `val input_binary_int : in_channel -> int`
  `val output_binary_int : out_channel -> int -> unit`
  Afin de vous simplifier la vie, des macros ont été définis pour vous dans le fichier `tyrme.ml` : pour écrire sur un buffer, `out_i8` et `out_i32`, et pour lire d'un buffer, `in_i8` et `in_i32`.

Les instructions sont écrites en binaire, à la suite les unes des autres, dans le sens de la lecture. Chaque instruction est composée de

- un mot de 8 bits (1 octet) contenant l'op-code de l'instruction concernée, puis un bloc constitué comme suit :
  - pour `str` : un mot de 32 bits (4 octets) contenant la longueur `n` de la liste, suivi de `n` mots de 8 bits, un par caractère de la liste (obtenus avec `output_channel`).
  - pour `binop` : un mot de 8 bits contenant l'opcode de l'opération binaire
  - pour `makeblock` : un bloc de 8 bits pour le tag et un bloc de 32 bits pour la valeur `n`.
  - pour chacune des autres instructions $n$-aires, $n$ mots de 32 bits contenant l'entier en question (quand $n = 0$, il y n'y a donc rien à mettre).

Comme les mots sont de deux tailles, on vous propose d'écrire les fonctions `assemble` et `desassemble` comme des fonctions récursives (donc sans boucle), et de lire les instructions une à une. Mais vous êtes libres de choisir un encodage impératif si vous préferez.

# 3 Travail à effectuer

Le travail à effectuer consiste en une section obligatoire (rapportant 12 points) et une série de parties optionelles permettant d'augmenter votre note.

## 3.1 Modalités

Vous pouvez vous mettre par groupe de 2.

1. en fait, comme ce qui est dit en séance 4, sur 31 bits... mais ici cela nous suffit

Vous me rendrez par mail à l'adresse `benoit.valiron@pps.univ-paris-diderot.fr` un fichier zip, avec le fichier `myrte.ml` rempli et le fichier `README.txt` complété, le tout avant

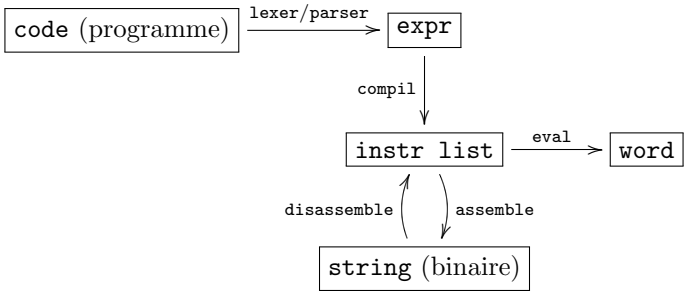
le dimanche 10 mai au soir.

## 3.2 Partie obligatoire

La partie obligatoire consiste essentiellement à remplir les fonctions laissées vides, dans lesquelles se trouvent

```
failwith "Student, this is your job"
```

Plus précisemment, l'objectif est de remplir un diagramme proche de celui vu en séance 2 :




La flèche `lexer/parser` est déjà donnée, il vous faut donc construire les autres.

Afin de vous guider, le dossier compressé fourni comprend le lexeur et le parseur, et un fichier `tyrme.ml` à remplir. Si vous souhaitez utiliser `ocaml` soit directement soit à travers le mode tuareg de emacs, après avoir exécuté

```
make
```

à la racine du répertoire, il faut lancer la commande `ocaml` comme suit :

```
ocaml lexer.cmo parser.cmo
```

quand emacs vous demande quel commande utiliser pour lancer ocaml/ Cela indique à ocaml de charger les librairies de lexing et de parsing. Vous pouvez ensuite soit utiliser `C-c C-e` à l'intérieur d'emacs, comme d'habiture.

Le dossier compressé comprend aussi le code-octet « secret » `projet.tm`. Si votre fonction de desassemblage et votre fonction `machine` fonctionnent correctement, exécuter ce code-octet devrait afficher une phrase (en principe sans fautes d'orthographe), suivie d'un saut de ligne.

## 3.3 Parties optionelles

Pour augmenter votre note, ou pour votre plaisir, vous pouvez aussi réaliser une ou plusieurs des extensions suivantes :

### 3.3.1 Un débogueur

Une extension très utile est simplement un. . . débogueur. Il s'agit d'une version de la machine virtuelle qui à chaque instruction exécutée, imprime l'état de la mémoire (accumulateur, PC, pile et tas) et attend un retour de l'utilisateur pour continuer.

### 3.3.2 Ajout de fonctions récursives

Les fonctions de Tyrme ne sont pas récursives : cette extension propose de les rendre récursive. Dans cette extension, une fonction

```
Letf of var * var * expr * expr
```

est maintenant récursive par défaut. Il vous faudra adapter les instructions pour supporter cette nouvelle donne.

### 3.3.3 Ajout de tableaux (produits $n$-aires)

Le langage Tyrme ne supporte pour le moment que des produits binaires. Étendez le langage pour supporter des produits $n$-aires. En particulier, ajoutez un constructeur de terme au type `expr`

```
Proj of int * expr
```

qui récupère la $i$-ème projection d'un produit $n$-aire.

### 3.3.4 Optimization des appels terminaux

Vous ajouterez une ou plusieurs instructions à la machine dédiées à optimiser les appels terminaux et les faire exécuter en taille de pile constante. Après la compilation, vous rajouterez une passe d'optimisation du bytecode produit utilisant cette instruction.

### 3.3.5 Autre implémentation du tas

En Section 2.2, vous avez fait un choix d'implémentation du tas de la machine virtuelle. Implémentez l'autre possibilité, et trouvez un ou deux termes du langage Tyrme qui montre comment les comportements des deux tas diffèrent.

### 3.3.6 Une machine virtuelle pour Myrte

Avec les tableaux et les fonctions récursives, vous avez assez de matériel pour implémenter... Une machine virtuelle pour Myrte. Note : il vous faudra ré-encoder les instructions en utilisant des paires d'entiers.