Machine Virtuelle

Le langage Tyrme

Eric Uzenat & Mohamed Yennek

Sommaire

Introduction

L'objectif de ce projet est de réaliser une machine virtuelle pour le langage Tyrme, qui est un mini langage de programmation reprenant la syntaxe de base du langage Ocaml.

Un lexer et un parser du langage sera fournit qui transcrit le code en un arbre syntaxique. Nous implémenterons un compilateur. Qui transcrira un arbre syntaxique en liste d'instruction. Puis la machine à pile (contenant un accumualeur et une pile) s'occupant de l'execution de chaque instruction de la liste. Et enfin le programme qui s'occupe d'interpréter les instructions que la machine à pile traitera. A cela, nous implémenterons aussi un assembleur qui génére du code-octet à partir d'une liste d'insruction, puis un désassembleur qui lui lira le code octet pour le retranscrire en liste d'instruction. Tout cela sera mis en forme dans un programme interactif en ligne de commande qui s'occupera de lire le code dans un fichier et pour l'interpreter, le compiler en code-octet, et executer ce code-octet. Un certain nombre de fonctionnalitées supplémentaire sera aussi implémenté.

Dans une premiere partie nous dresserons un mode d'emploi détaillé pour la compilation et l'utilisation du programme. Puis, nous expliqueront ensuite plus en detail la manière dont nous avons pensé chaque partie du projet.

1. Mode d'emploi

1.1. La compilation

La compilation du programme est très simple. Un makefile est en effet fournit pour simplifier grandement les choses. Ainsi, pour compilet le programme, il suffira de taper :

```
>> make tyrme
```

Il sera possible aussi de nettoyer tout les fichiers generé lors de la compilation en tapant :

```
>> make clean
```

Cette commande gardera l'executable, si en plus de sa on veut le supprimer, alors on tapera :

```
>> make cleanall
```

Ainsi, pour recompiler le programme si celui la à deja été compilé, il est impératif d'executer ces deux commandes dans cet ordre :

```
>> make cleanall
>> make tyrme
```

A la suite d ela compilation, un fichier executable tyrme sera générer

1.2. L'execution du programme

Venons en maintenant à l'execution du programme. Pour l'éxecuter, il faudra taper, dans le terminal, la ligne suivante :

```
>> ./tyrme [-c/-i/-e/-dinstr] [-d] fichier.[ty/bty]
```

Les options `-c`, `-i`, `-e`, `-dinstr` sont obligatoire, il faudra toujours les renseigner (mais ils ne s'utilisent jamais ensemble). L'option `-d` est elle optionnelle. Le fichier devra être d'extansion `.ty` ou `.bty` selon qu'il s'agisse de code source ou de byte-code. Il faut savoir que l'option `-c` et `-i` s'utilise avec un fichier d'extansion `.ty`, et `-e` avec un fichier d'extansion `.bty`.

Expliquons maintenant chacune de ces options. L'option `-c` compile le fichier `fichier.ty` et génère un fichier `fichier.bty` contenant le byte-code. L'option `-i` interprete du code source, c'est à dire quelle va éxécuter le fichier `fichier.ty` sans avoir besoin de le compiler en code octet.

L'option `-e` lui executera le byte-code du fichier `fichier.bty`. L'option `-dinstr` affichichera la liste d'instruction d'un fichier `.ty` ou `.bty`. Et enfin, l'option `-d` sera un debuggueur, il affichera chaque étape de l'éxécution dela machine virtuelle.

Par exemple, si l'on veut executer le byte-code contenu dans un fichier `prog1.bty`, on tapera :

```
>> ./tyrme -e prog1.bty
```

Si l'on veut interpreter un fichier source appeler `prog2.ty` en mode debuggage, on tapera :

```
>> ./tyrme -i -d prog2.ty
```

Si l'on veut afficher la liste d'instruction du fichier `prog3.bty`, on tapera :

```
>> ./tyrme -dinstr prog3.bty
```

2. Implémentation du projet

2.1. Arbre syntaxique et instruction

2.1.1. L'arbre syntaxique

L'arbre syntaxique correspond aux expressions du programme. C'est le `lexer` et le `parser` qui va s'occuper de générer cette arbre à partir d'un fichier écrit en langage Tyrme. Les expressions de cette arbre se trouve dans le fichier `ast.mli`. Nous allons brievement expliquer chacune de ces expressions.

On à dans un premier temps le type `var` qui n'est en fait rien d'autre qu'une chaine de caractere.

```
type var = string
```

Viens ensuite les type `value`, qui corresponde au type des donnée, autrement dit, les `int`, les `boolean` les `string` et les types vide (`unit`)

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

Nous avons ensuite les types `binop` qui correspondent au opération binaire que l'on peut effectuer sur les types `value`. On aura l'addition, la soustraction, la multiplication, la division, l'égalité, l'opérateur boolean inferieur ou égal, la concaténation et `App`, qui correspond à l'application d'une variable à une fonction.

```
type binop =
| Add
| Sub
| Mult
| Div
| Leq
```

```
| Eq
| And
| Cat
| App
```

Pour finir, nous avons donc les types expressions qui corresponde au noeud de notre arbre syntaxique. Chacune de ces expression correspond à un mot de notre programme. Nous Allons expliquer brievement à quoi correspond chacune de ces expressions.

```
type expr =
| Var of var
| 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
| Liste of expr list
| Proj of int * expr
```

Le constructeur `Var` correspond à la déclaration d'une variable, il prend en argument un type `var` qui correspond donc à une chaine de caractere.

Le constructeur `Const` correspond à la declaration d'une constante. Il prend en argument un type value (`Int`, `Bool`, `String`, ...).

Le constructeur `Binop` correspond à la déclaration d'une opération binaire. Il prend en argument trois , arguments qui sont dans l'ordre l'opérateur binop qui correspond à l'opération, une expression, puis une seconde expression.

Le constructeur `If` correspond à la déclaration des branche if … then … else, il prend lui aussi trois argument, qui sont trois expressions . La premiere correpond à l'expression de test entre le if et le then, la seonde à l'expression qu'on execute si le test est vrai, et la derniere à l'expression qu'on execute si le teste est faux.

Le constructeur `Let` correspond à la déclaration d'une variable, il prend trois argument. Le premier correspond au nom de la variable (type var), le second correspond à l'expression que l'on veut stocker dans la variable, et le dernier correspond à l'expression qui s'executera après que l'on ait réalisé l'affectation.

Le constructeur `Letf` correspond à la déclaration d'une fonction à un parametre. Il prend en argument le nom de la fonction, le nom du parametre, une premiere expression qui corespond au corp de la fonction, puis une seconde correspondant à ce que l'on veut éxécuter après la fonction.

Le constructeur `Print` correspond à la déclaration d'une fonction d'affichage. Il prend en argument une premiere expression qui est la donnée que l'on affichera sur la sortie standard, puis une seconde expression qui corespond à la suite du code que l'on veut éxécuter.

Le constructeur `Pair` correspond à une structure de donnée pouvant contenir deux expressions.

Les constructeurs `Fst` et `Snd` s'utilise en général avec `Pair`, elle prenne donc une expression de type Pair en argument. `Fst` permet d'obtenir la premiere expression et `Snd` la seconde.

`Liste` prend en argument une liste d'expression. Elle correspond à la definition d'un tableau. Pour récupérer les élément de la liste, on fera appel au constructeur `Proj`, qui prend en argument un entier et une expression (de type `Liste`).

2.1.2. Les instructions

La compilation transforme notre arbre syntaxique en liste d'instruction que devra interpréter notre machine virtuelle. Il est donc interessent de fournir la liste exhaustive des instructions et de donnée un court descriptif de chacune d'elle, en precisant son opcode.

```
type instr =
| Halt
| Push
| Print
| Acc of int
| Const of int
| Pop of int
| Str of string
| Binop of int
| BranchIf of int
| Branch of int
| MakeBlock of int * int
| GetBlock of int
| GetBlock2 of int
| Closure of int * int
```

```
| Apply
| Return of int
| AppTerm of int * int
```

L'instruction `Halt` met fin à l'execution de la machine virtuelle. Son opcode est `0`.

L'instruction `Push` empile le contenue de l'accumulateur sur la pile. Son opcode est `1`.

L'instruction `Print` affiche sur la sortie standard le contenue de l'accumuateur. Son opcode est `2`.

L'instruction `Acc n` place dans l'accumulateur la nème valeur de la pile. Il faut savoir que `Acc 0` correspond à la tete de la pile. Son opcode est `4`.

L'insctruction `Const v` place dans l'accumulateur la constante `v`. Son opcode est `5`.

L'instruction `Pop n` dépile les `n` premiers éléments de la pile. Son opcode est `7`.

L'insctruction `Str s` place dans l'accumulateur la chaine de caractere `s`. Son opcode est `14`.

L'instruction `Binop b` applique l'opération `b` entre l'accumulateur et la tete de la pile, et place le resultat dans l'accumulateur. Son opcode est `13`. L'operation `b` correspond enfait lui aussi à un opcode qui définit le type de l'opération. On aura `15` pour l'addition, `16` pour la soustraction, `17` pour le produit, `18` pour la division, `19` pour l'egalité, `20` pour le concatenation et `21` pour l'operateur and.

L'instruction `BranchIf n` saute de `n` instruction si l'accumulateur vaut 1, et de une instruction si il vaut 0. Son opcade est `8`

L'instruction `Branch n` saute de `n` instruction. Son opcode est `9`

L'instruction `MakeBlock t, n` met dans l'accumulateur un bloc de tag `t` et contenant les `n` valeurs au sommet de la pile, qui sont dépilé. Son opcode est `11`

L'instruction `GetBlock n` met dans l'accumulateur la n-ème case du bloc pointé par l'accumulateur. Son opcode est `10`

L'instruction `GetBlock2 n` recupere la n-ème valeur k de la pile (en supposant qu'il s'agisse d'un entier) et met dans l'accumulateur la k-èm e case du bloc pointé par l'accumulateur. Son opcode est `31`.

L'instruction `Closure n, o` alloue un bloc de taille `n + 1`, de tag `88`, contenant l'indice de l'instruction située `o` instruction en avant, suivi des `n` premières cases de la pile qui ne sont pas dépilées. Son opcode est `12`

L'instruction `Apply` appelle la fonction dont la fermeture est contenue dans l'accumulateur, avec comme argument le contenu du sommet de la pile. Son opcode est `3`

L'instruction `Return n` dépile et jette les `n` premieres cases de la pile et retourne à la fonction appelante. Son opcode est `6`.

L'instruction `AppTerm 1, n` effectue les operation `Apply` et `Return n` en même temps. Son opcode est `30`.

2.2. Assembleur et desassembleur

2.2.1. L'assemblage

L'assemblage est l'opération qui permet de transformer une liste d'instruction en code-octet. Les instructions sont ecrite en binaire l'une à la suite des autres. Chaque instruction est composée d'un mot de huit bit correspondant à l'opcode de l'instruction, puis pour certaines instructions est suivie d'un ou plusieur autre bloc d'octet :

- pour `Str s` : un mot de 4 octet contenant la longueur n de la chaine, suivie de n mots de 8 bits, un par caractere de la liste

- pour `Binop b` : un mot de 8 bits contenant l'opcode de l'opération binaire

- pour `MakeBlock t, o` : un bloc de 8 bits pour le tag et un bloc de 32 bits pour la valeur n

- pour toute les autres instructions n-aires avec n >= 1, n mots de 4 octet

Pour ce faire, nous procédons d'une manière très simple. Dans un premier temps on dispose d'une fonction qui à partir d'une instruction ecrit dans un fichier son opcode, et les renseignements nécéssaire propre à l'instruction. Cette fonction est ensuite appelé dans une autre fonction qui appelle la premiere sur l'ensemble de la liste d'instruction.

2.2.2. Le desassemblage

Le desassemblage est l'opération inverse de l'assemblage. C'est à dire qu'a partir d'un fichier contenant le code octet, on va pouvoir reconstruire toute la liste d'instruction. Pour ce faire on utilise les convention cité au paragraphe précédent. On lit donc 8 bit pour l'opcode, puis des bloc de 8 ou 32 bits selon l'opcode lu (si nécéssaire), et on recommence l'opération jusqu'a la fin du fichier.

2.3. La compilation

La compilation est l'opération qui va permettre de transformer notre arbre syntaxique généré par le parser en une liste d'instruction. Dans cette partie nous essayerons d'expliquer au mieux la compilation de chaque expression de l'arbre syntaxique tyrme.

2.3.1 L'environnement

Tout d'abord, il convient de presenter le concept d'environnement. A la compilation, notre fonction utilise un environnement pour stocker la position des variables et des fonctions dans la pile. Cette environnement est définit comme suit

```
type env = (var * int) list

let empty_env = []

let envsucc e =
  let succ (x,y) = (x, y+1) in
  List.map succ e
```

env correspond à l'envirronement, on définit en plus de cela l'envirronement vide et la fonction qui permet de générerer l'envirronnement successeur nécéssaire à l'ajout d'une valeur dans cette envirronement.

3.1.2. Compilation des expressions

- ***Var***

Pour `Var s`, où s correspond donc au nom d'une variable, on retournera s'implement une liste à un élément contenant `Acc n` où `n` sera la position de la valeur de la variable dans l'envirronement.

```
let rec compil : env * expr -> instr list = function
  | env, Var s -> [ Acc (List.assoc s env) ]
     ...
```

- ***Const***

Pour `Const v`, où v correspond à la valeur de la constante (type `value`), on retournera de la même maniere une liste à un seul élément. Cet élément sera soit Const i où i est un entier, soit Str s où s est une chaine de caractère, selon la valeur de `v`.

```
let rec compil : env * expr -> instr list = function
  ...
  | env, Const v ->
    begin
      match v with
      | Int i -> [ Const i ]
      | Bool i ->
        begin
          match i with
          | true -> [ Const 1 ]
          | false -> [ Const 0 ]
        end
      | String s -> [ Str s ]
      | Unit -> []
      | _ -> failwith "erreur compilation"
    end
   ...
```

On match donc la valeur v pour determiner l'instruction que l'on voudra retourner.

- ***Binop***

Pour `Binop(o, e1, e2)`, on à deux cas possible, soit l'opération `o` est de type `App`, cest à dire que l'on applique l'argument `e2` à la fonction `e1`. Soit o est une opération de type `Add`, `Eq`, `And`, …

Pour le premier cas, il s'agit de l'application d'une fonction, ainsi on devra compiler `e2` (qui correspond à l'argument de la fonction) l'empiler avec `Push`, compiler `e1` (correspond au nom de la fonction) puis on ajoute `Apply` pour appliquer la fonction.

Pour le second cas, on compile d'abord `e1`, on l'empile. Puis on compile `e2`, Et on donne l'operation `Binop b`, ou `b` est l'opcode qui sera retourner par la fonction `op`.

```
let rec compil : env * expr -> instr list = function
  ...
  | env, Binop (o, e1, e2) ->
    begin
      match o with
      | App ->
     compil (env, e2) @
```

```
      [ Push ] @
      compil (envsucc env, e1) @
      [ Apply ]
     | _ ->
    compil (env, e1) @
      [ Push ] @
      compil (envsucc env, e2) @
      [ Binop (op o) ]
   end
 ...
```

- ***If***

Pour If(e1, e2, e3), on compile dans un premier temps e2 et e3 que l'on stock dans des variable (on aura besoin de recupérer leur tailler par la suite. Puis notre Liste d'instruction se compose de la compilation de e1, du BranchIf n, où n correspond à la longueur de la liste générer par la compilation de e3 plus deux, puis la liste en question, suivie de Branch m, où m correspond à la longueur de liste générer par la compilation de e2 plus un, et enfin cette même liste. Enfait, e1 correspond au test, e2 au if et e3 au else. Le BranchIf n saute de n opération pour arriver aux instruction généré par e2 si le teste est valide. Sinon il saute de une instruction.

```
let rec compil : env * expr -> instr list = function
  ...
 | env, If (e1, e2, e3) ->
    let i2 = compil (env, e2) in
    let i3 = compil (env, e3) in
    compil (env, e1) @
      [ BranchIf ( 2 + (List.length i3) ) ] @
      i3 @
      [ Branch ( 1 + (List.length i2) ) ] @
      i2
  ...
```

- ***Let***

Venons en maintenent à la compilation du Let, qui correspond à la declaration et à l'affectation d'une variable. La donnée s correspond au nom de notre variable, il convient dans un premier temps de l'ajouter à notre environnement à la position 0 (on oublie pas d'incrémenter de 1 toute les autres

position. La liste d'instruction est donc constitué de la compilation de e1 (correspond à la donnée stocké)), suivie d'un Push, puis la compilation de e2 avec le nouvel envirronement suivie d'un Pop.

```
let rec compil : env * expr -> instr list = function
  ...
  | env, Let (s, e1, e2) ->
    let new_env = (s, 0) :: (envsucc env) in
    compil (env, e1) @
      [ Push ] @
      compil (new_env, e2) @
      [ Pop 1 ]
  ...
```

- ***Letf***

La compilation de fonction est un processus assez complexe. Il est important de préciser que l'on à introduit le principe de recursion terminal avec `AppTerm`. Ainsi, pour l'implémenter correctement, La liste d'instruction devra subir trois phase d'otpimisation que nous expliquerons après.

Dans un premier temps, il convient de mettre a jour l'environnement de compiation. On disposera de deux environnements de compilation que l'on mettra a jour comme suit

```
 let new_env0 = (s2, 0) :: (envsucc env) in
 let new_env1 = (s1, 0) :: (envsucc new_env0) in
 let new_env2 = (s1, 0) :: (envsucc env) in
```

`new_env1` contient l'environnement courant auquel on ajoutera le nom de l'argument et le nom de la fonction. On l'utilisera pour la compilation du corp de la fonction.

`new_env2` contient l'environnement courant auquel on ajoutera simplement le nom de la fonction. On l'utilisera pour la compilation de ce qu'il y à apres le corp de la fonction.

Presentons maintenant les etapes de compilation. On compilera donc dans un premier temps `e1` avec `new_env1`, puis on devra lui appliquer les trois phase d'optimisation nécéssaire à l'implémentation de la recursion terminale appelons cet liste d'instruction `op3`. Puis on compilera `e2` dans une liste que l'on appelera `i3`. La liste d'instruction finale sera donc L'instruction `Closure` avec comme argument la taille de l'envirennement de base et la taille de `i3` (qui correspond dont à la reprise de la onction principale), suivie de `Push` (pour empiler la signature sur la pile), ensuite la liste `i3`, suivie d'un `Pop`. Enfin on ajoutera la liste optimisé `op3` correspondant au corps de la fonction, sans oublier le `Return n` à la fin où n aura pour valeur la taille de l'environnement plus deux.

```
let rec compil : env * expr -> instr list = function
   ...
   | env, Letf (s1, s2, e1, e2) ->
       let i2= compil (new_env1, e1) in
       let i3= compil (new_env2, e2) in
       let op1 = optimise i2 in
       let op2 = optimise2 op1 in
       let op3 = optimise3 op2 in
       [ Closure ( List.length env, (((List.length i3)+4))) ] @
         [ Push ] @
         i3 @
         [ Pop 1 ] @
         [ Branch ((List.length i2) + 3) ] @
         [ Push ] @
         op3 @
         [ Return (2 + List.length env) ]
   ...
```

Venons en maintenant à l'explication des trois phase d'optimisation, il faut savoir qu'elle ne s'applique que sur la liste du corp de la fonction. La premiere consiste à supprimer tout les `Branch` par des `Return`, en effet, dans un `if … then … else`, si on entre dans le if on entrera jamais dans le else alors le `Branch` ne sert a rien car il n'y a rien apres celui ci, on peut mettre directement un `Return`. La seconde consiste à remplacer tout les `[ Apply ; Return n ]` par des `[ AppTerm (1, n) ]` afin de pour à la fois appliquer la signature de la fonction et depiler toute les valeurs. Dire qu'un `Apply` se trouve a coté d'un `Return` signifie que la fonction est en récursion terminale. Enfin, on aura un petit soucie au niveau des `BranchIf`, il faudra les décrémenter de `1` à chaque `AppTerm` trouver, ce qui conclut notre troisieme phase d'optimisation.

- ***Print***

La compilation du `Print` est très simple. La liste d'instruction se compose de la compilation de `e1`, suivie de l'instruction `Print`, puis la compilation de `e2`.

```
let rec compil : env * expr -> instr list = function
 ...
 | env, Print (e1, e2) ->
    compil (env, e1) @
      [ Print ] @
```

```
      compil (env, e2)
 ...
```

- ***Pairs***

Pour la compilation de `Pair`, on compile et on empile les resultats de `e1` et `e2`, puis on appel l'instruction `MakeBlock 0, 2` pour générer un block avec les deux valeurs empilé.

```
let rec compil : env * expr -> instr list = function
 ...
 | env, Pair (e1, e2) ->
    compil (env, e2) @
      [ Push ] @
      compil (env, e1) @
      [ Push ; MakeBlock (0, 2) ]
 ...
```

- ***Fst & Snd***

Pour `Fst` est `Snd`, on compile d'abord `e` on suppose qu'il s'agit d'une structure n-aire), puis on ajoute à la liste d'instruction généré `GetBlock n` avec `n` valant `0` pour `Fst` et `1` pour `Snd`

```
let rec compil : env * expr -> instr list = function
 ...
 | env, Fst e ->
    compil (env, e) @
      [ GetBlock 0 ]
  | env, Snd e ->
    compil (env, e) @
      [ GetBlock 1 ]
 ...
```

- ***Liste***

Pour la compilation de `Liste`, on procede de la même maniere que pour Pair, on compile chaque élément de la liste puis on empile le resultat. A la fin, on ajoute `MakeBlock 0, n` où n est la taille de la liste.

```
let rec compil : env * expr -> instr list = function
 ...
 | env, Liste l ->
    let rec aux x = match x with
      | [] -> []
      | h :: tl ->  aux tl @ compil (env, h) @ [ Push ]
    in (aux l) @ [ MakeBlock (0, List.length l) ]
 ...
```

- ***Proj***

Pour finir, pour la compilation de `Proj`, on a du rajouter une instruction `GetBlock2 n`, qui fonctionne de la même manière que `GetBlock n`, à la seul différence que `GetBlock2 n` renvoit la `k`-ème valeur du bloc pointé par l'accumulateur, où `k` correspond à la `n`-ème valeur de la pile. Ainsi on compile `e2`, on match `e1` pour savoir si il est de type `Int`, ou `Var`. Dans le premier cas, on concatene la liste d'instruction générer par la compilation de `e2` avec `GetBlock x` où `x` est la valeur de cet entier. Dans le second cas, on concatène la liste d'instruction généré par la compilation de `e2` avec `GetBlock2 x`, où `x` correspond à la position de la variable dans l'environnement.

```
let rec compil : env * expr -> instr list = function
 ...
 | env, Proj (e1, e2) ->
    let i = compil (env, e2) in
     match e1 with
      | Const( Int x ) -> i @ [ GetBlock x ]
      | Var s -> i @ [ GetBlock2 (List.assoc s env) ]
      | _ -> failwith "erreur proj"
```

2.4. La machine virtuelle

2.4.1. Présentation de la machine Tyrme

La machine virtuelle tyrme est une machine à pile (tout comme la machine d'ocaml). Elle dispose dont d'une pile et d'un accumulateur.

```
type mv_state ={
  mutable acc: mot;
  code: instr array;
  mutable pc: int;
  stack: mot array;
  mutable sp: int;
}
```

`acc` correspond à l'accumulateur, code au tableau contenant les instructions, `pc` à la position courante de la lecture du tableau, `stack` à la pile et `sp` à la position du sommet de la pile. Il est important de préciser que la machine ne contient pas de tas. Enfait on le simule grace à l'utilisation d'un type prédéfinie appelé `mot`. Ainsi, la pile et l'accumulateur contienne uniquement des élément de type `mot` que l'on définie comme suit :

```
type tag = int

type mot =
| MotInt of int
| PointString of string
| PointBloc of (tag * (mot list))
```

`MotInt` correspond donc à un entier, `PointString` correspont au pointeur vers une chaine de caractere, et `PointBloc` au pointeur vers un bloc, qui n'est enfait qu'une liste de mot.

2.4.2. Interprétation des instructions

Venons en maintenant à l'explication de la fonction qui s'occupe du traitement de notre machine virtuelle. Chaque Instruction est donc lus est interpréter par la machine, et modifie donc sa pile est sont tas.

- ***Halt***

`Halt` met fin à l'execution de la machine virtuelle, ainsi pour le codé, il suffit simplement de terminer le programme avec `failwith`

```
let machine (s : mv_state) : mv_state =
```

```
  ...
  | Halt -> failwith "Terminaison de la machine"
  ...
```

- ***Push***

`Push` place le contenue de l'accumulateur au sommet de la pile. Pour se faire, il suffit d'incrémenter la valeur `sp` qui correspond au sommet de la pile, est de placé cette le contenue de l'accumulateur dedant.

```
let machine (s : mv_state) : mv_state =
   ...
   | Push ->
         s.sp <- s.sp +1;
         s.stack.(s.sp) <- s.acc
   ...
```

- ***Acc***

`Acc n` récupère la `n`-ème valeur de la pile en partant du sommet.

```
let machine (s : mv_state) : mv_state =
   ...
   | Acc n ->
        s.acc <- s.stack.(s.sp - n)
   ...
```

- ***Pop***

`Pop n` dépile les `n` premiers éléments de la pile. Il suffit de décrémenter le sommet `sp` de `n` unités.

```
let machine (s : mv_state) : mv_state =
   ...
   | Pop n ->
        s.sp <- s.sp - n
   ...
```

- ***Const***

`Const n` place `n` dans l'accumulateur avec le constructeur `MotInt`

```
let machine (s : mv_state) : mv_state =
   ...
   | Const n ->
        s.acc <- MotInt n
   ...
```

- ***Str***

`Str s` place un pointeur de chaine de caractere dans la pile à l'aide de `PointString`

```
let machine (s : mv_state) : mv_state =
   ...
   | Str s1 ->
        s.acc <- PointString s1
   ...
```

- ***Binop***

`Binop b` effectue une opération binaire entre la valeur contenue dans l'accumulateur, et le sommet de la pile. Pour ce faire, il a fallut matcher `b` pour recuperer l'opcode est realisé l'opération en fonction de celui ci. Si `#` est l'opération `MotInt a` est la valeur de l'accumulateur et `MotInt b` la valeur du sommet de la pile, alors on place dans l'accumulateur `MotInt(b # a)`. Puis on oublie pas de décrementer `sp` de `1` pour dépiler le sommet.

```
let machine (s : mv_state) : mv_state =
   ...
   | Binop b ->
        begin
           match b, s.acc, s.stack.(s.sp) with
             | 15, MotInt i, MotInt j ->
                  s.acc <- MotInt(j + i)
             | 16, MotInt i, MotInt j ->
           s.acc <- MotInt(j - i)
             | 17, MotInt i, MotInt j ->
           s.acc <- MotInt(j * i)
             | 18, MotInt i, MotInt j ->
```

```
          s.acc <- MotInt(j / i)
            | 19, MotInt i, MotInt j ->
          s.acc <- MotInt(if j = i then 1 else 0)
            | 20, PointString s1, PointString s2 ->
          s.acc <- PointString (s2 ^ s1)
            | 21, MotInt i, MotInt j ->
          let n =
          match i,j with
          | 0,0 -> 0
          | 0,1 -> 0
          | 1,0 -> 0
          | 1,1 -> 1
          | _ -> failwith "erreur compilation"
          in
          s.acc <- MotInt(n)
            | _ -> failwith "erreur compilation"
        end;
       s.sp <- s.sp - 1;
     ...
```

- ***BranchIf***

`BranchIf n` saute de `1` instruction si l'accumulateur vaut `0` et de `n` instruction sinon. Pour ce faire, on modifie simplement la valeur du `pc` en l'incrémentant de `1 (-1)` si l'accumulateur vaut `0` et de `n (-1)` sinon. On doit forcément rajouter `-1` car à chaque itération le `pc` est incrémenté de `1`.

```
let machine (s : mv_state) : mv_state =
   ...
   | BranchIf n ->
        s.pc <- s.pc + (if (s.acc = MotInt(0)) then 0 else n - 1);
   ...
```

- ***Branch***

`Branch n` saute simplement de `n` instruction donc incrémente le `pc` de `n (-1)`

```
let machine (s : mv_state) : mv_state =
```

```
   ...
   | Branch n ->
        s.pc <- s.pc + n - 1
   ...
```

- ***Return***

`Return n` dépile `n` valeur de la pile, et met à jour la valeur du `pc` (qui est aussi stocké dans la pile) afin de retourner au programme principale.

```
let machine (s : mv_state) : mv_state =
   ...
   | Return n ->
        s.sp <- s.sp - n;
        let pc0 = s.stack.(s.sp) in
        let pc =
         begin
          match pc0 with
            | MotInt i -> i
            | _ -> failwith "erreur apply"
        end
        in
        s.pc <- pc - 1;
        s.sp <- s.sp - 1
   ...
```

- ***Apply***

Pour `Apply`, on suppose que l'accumulateur contient la signature d'une fonction. On commence par stocker la valeur du sommet de la pile (qui correspond à l'argument de la fonction), A la place, on placera le `pc + 1`. Puis on empile toutes les valeur de l'envirronnement avec la methode `aux`. Pour finir, on place le `pc` contenue dans la signature de la fonction dans le `pc` courant, et on empile l'argument que l'on à récupéré au début.

```
 let machine (s : mv_state) : mv_state =
  ...
   | Apply ->
```

```
        begin
          match s.acc with
             | PointBloc (88, t) ->
                  let n = s.stack.(s.sp) in
                  s.stack.(s.sp) <- MotInt (s.pc + 1);
                 let rec aux x = match x with
              | [] -> ()
              | t::q -> aux q; s.sp <- s.sp + 1; s.stack.(s.sp) <- t
               in
               aux t;
               let pc0 = s.stack.(s.sp) in
              let pc2 =
          begin
             match pc0 with
                | MotInt i -> i
                | _ -> failwith "erreur apply"
            end
                 in
           s.pc <- pc2 - 1;
          s.stack.(s.sp) <- n
         | _ -> failwith "erreur apply"
      end
   ...
```

- ***Closure***

`Closure n, o` place dans l'accumulateur une signature de fonction. Plus précisemment, il place dans l'accumulateur un `PointBloc`, de `tag 88`, et contenant la liste avec le `pc + o`, suivis de toute les valeur de la pile correspondant à l'environnement de la fonction.

```
let machine (s : mv_state) : mv_state =
   ...
   | Closure (n, o) ->
     let rec aux i  =
         if i = n then []
         else [ s.stack.(s.sp - i) ] @ (aux (i+1))
     in
```

```
    let t = [ MotInt (s.pc + o) ] @ (aux 0) in
    s.acc <- PointBloc (88, t)
  ...
```

- ***MakeBlock***

`MakeBlock t, n` place dans l'accumulateur un bloc de `tag t` contenant les `n` premiere valeur de la pile qui seront dépilé.

```
let machine (s : mv_state) : mv_state =
  ...
  | MakeBlock (t, n) ->
    let rec aux i =
      if i = n then []
      else [ s.stack.(s.sp - i) ] @ (aux (i+1))
    in
    let t = aux 0 in
    s.acc <- PointBloc (0, t);
    s.sp <- s.sp - n
  ...
```

- ***GetBlock***

`GetBlock n` parcours l'ensemble du bloc pointer par l'accumulateur et place dans l'accumulateur la `n`-ème valeur de ce bloc.

```
let machine (s : mv_state) : mv_state =
   ...
   | GetBlock n ->
     let x =
     match s.acc with
      | PointBloc (0, t) ->
           let rec aux x i = match x with
       | [] -> failwith "erreur getblock"
       | h::tl ->
            if i = n then h else aux tl (i+1)
            in
```

```
            aux t 0
         | _ -> failwith "erreur getblock"
        in
        s.acc <- x
      ...
```

- ***GetBlock2***

`GetBlock2 n` va dans un premier tans recuperer la n-ème valeur de la pile appelons la `k` (on suppose que celle ci correspond à un `MotInt`) puis parcours le bloc pointer par l'accumulateur et y place ensuite la `k`-ème valeur du bloc.

```
let machine (s : mv_state) : mv_state =
   ...
   | GetBlock2 n ->
     let n' =  match s.stack.(s.sp - n) with
      | MotInt i -> i
       | _ -> failwith "erreur getblock2"
     in
     let x =
       match s.acc with
       | PointBloc (0, t) ->
         let rec aux x i = match x with
     | [] -> failwith "erreur getblock"
     | h::tl ->
       if i = n' then h else aux tl (i+1)
         in
         aux t 0
       | _ -> failwith "erreur getblock"
     in
     s.acc <- x
   ...
```

- ***AppTerm***

Comme définit plus haut, AppTerm correspond donc à l'execution des instructions Apply et Return, réalisé en même temps.