Examen de Compilation Avancée – M2 Informatique 2018-2019
Durée: 3 heures

Notes de cours autorisées.

Nota :
— Dans les réponses à écrire en syntaxe FOPIX ou KONTIX, vous pourrez utiliser la notation `[|a;...;c|]` inspirée d'OCAML pour la création et le remplissage d'un tableau.
— Pour les réponses à donner en JAVIX, vous pourrez ignorer les `Box`/`Unbox`. Vous pourrez aussi utiliser une pseudo-instruction `newCouple` qui déplace les deux éléments en sommet de pile dans un nouveau tableau de taille 2, et laisse le pointeur vers ce tableau sur le haut de la pile. De même, la pseudo-instruction `newSingleton` fabriquera et rempliera un tableau de taille 1.

Exercice 1 *(Un codage little-endian des entiers naturels) On considère le programme* OCAML *suivant, où les entiers naturels sont représentés par la liste de leurs chiffres binaires, avec en tête de liste le chiffre de poids faible.*

```
type num = int list (* listes contenant des 0 ou des 1 *)

let six = [0;1;1]

let rec to_int = function
| [] -> 0
| bit::bits -> bit + 2 * to_int bits

let test6 = to_int six

let rec num_compare bits bits' = match bits, bits' with
  | [], [] -> 0
  | [], _ -> num_compare [0] bits'
  | _, [] -> num_compare bits [0]
  | b::bits, b'::bits' ->
     let c = num_compare bits bits' in
     if c <> 0 then c
     else b - b'

let test66 = num_compare six six
```

1. *Traduisez ce programme en* FOPIX*, en expliquant quelle représentation des données est utilisée.*
2. *Traduisez en* JAVIX *le début de ce programme (jusqu'à* `test6` *inclus).*
3. *Donnez une version récursive terminale de* `to_int`*. Indice : il peut être intéressant d'avoir sous la main des puissances de deux successives.*
4. *Utiliser l'algorithme de mise en CPS vu en cours pour convertir le programme entier en* KONTIX*.*
5. *Donner les continuations et environnements successifs lors du calcul de* `test66` *en* KONTIX*.*
6. *Ecrire en* OCAML *une fonction* `normalize : num -> num` *qui supprime tous les 0 inutiles à droite de la liste. Ainsi* `normalize [0;1;0;0] = [0;1]` *et* `normalize [0;0;0] = []`*. Si ce n'est pas déjà le cas, donnez ensuite une version récursive terminale de cette fonction* `normalize`*.*
7. *(Bonus) Comment peut-on écrire en* OCAML *une version récursive terminale de* `num_compare` *qui soit plus simple que celle obtenue à la question 5 (c'est-à-dire sans continuation ni fonction passée en argument) ?*

□

Exercice 2 *(Arguments fonctionnels) On considère le code suivant :*

```
def compose (f,g,x) = ?(f) (?(g) (x))
def square (x) = x*x
def pow4(x) = compose (&square,&square,x)
eval pow4(3)
```

1. *Traduisez la fonction* `compose` *en* JAVIX.
2. *Traduisez ce programme entier en* KONTIX.
3. *Quel(s) optimisation(s) permettrai(en)t ici d'obtenir un code équivalent mais sans fonctions en arguments ?*

□

Exercice 3 *(Trampoline) On considère la fonction* `rev_map` *suivante :*

```
let rec rev_map f l acc = match l with
 | [] -> acc
 | x::l' -> rev_map f l' (f x :: acc)

let ex = rev_map (fun x -> 2*x) [1;2;3] []
```

1. *Utilisez la technique du trampoline pour transformer cette fonction récursive en une fonction* OCAML *non-récursive utilisant une boucle* `while`.
2. *Y avait-il ici une solution plus directe pour obtenir* `rev_map` *via une boucle* `while` *? Si oui laquelle ?*
3. *Dérécursifiez le code suivant à l'aide d'un trampoline :*

```
type tree = Node of int * tree list

let rec sum_tree s ll (Node (n,l)) =
  sum_treelist (n+s) ll l

and sum_treelist s ll = function
| [] -> sum_treelistlist s ll
| t::l -> sum_tree s (l::ll) t

and sum_treelistlist s = function
| [] -> s
| l::ll -> sum_treelist s ll l

let sum t = sum_tree 0 [] t

let ex = sum (Node (3, [Node (4,[]);
                        Node (5, [Node (7,[])])]))
```

4. *La technique du trampoline permettrait-elle d'obtenir du bytecode JVM acceptable par le validateur de la JVM ? Comparez cette approche avec la mise en forme CPS vue en cours.*

□

Annexe

```
type program = definition list

and definition =
  | DefVal of identifier * expression
  | DefFun of function_identifier * formals * expression

and expression =
  | Num of int
  | FunName of function_identifier
  | Var of identifier
  | Let of identifier * expression * expression
  | IfThenElse of expression * expression * expression
  | BinOp of binop * expression * expression
  | BlockNew of expression                          (* size *)
  | BlockGet of expression * expression             (* array, index *)
  | BlockSet of expression * expression * expression (* array, index, value *)
  | FunCall of expression * expression list

and identifier = string

and formals = identifier list

and function_identifier = string

and binop =
  | Add | Sub | Mul | Div | Mod (* arithmetic ops *)
  | Eq | Ne | Le | Lt | Ge | Gt (* comparisons *)
```

FIGURE 1 – FOPIX

```
type program = definition list * tailexpr

and definition =
  | DefFun of function_identifier * formals * tailexpr
  | DefCont of function_identifier * formal_env * identifier * tailexpr

and basicexpr =
  | Num of int
  | FunName of function_identifier
  | Var of identifier
  | Let of identifier * basicexpr * basicexpr
  | IfThenElse of comparison * basicexpr * basicexpr
  | BinOp of binop * basicexpr * basicexpr
  | BlockNew of basicexpr
  | BlockGet of basicexpr * basicexpr
  | BlockSet of basicexpr * basicexpr * basicexpr

and comparison = cmpop * basicexpr * basicexpr

and tailexpr =
  | TLet of identifier * basicexpr * tailexpr
  | TIfThenElse of comparison * tailexpr * tailexpr
  | TPushCont of function_identifier * identifier list * tailexpr
  | TFunCall of basicexpr * basicexpr list
  | TContCall of basicexpr

and identifier = string
and function_identifier = string
and formals = identifier list
and formal_env = identifier list
and binop = Add | Sub | Mul | Div | Mod
and cmpop = Eq | Ne | Le | Lt | Ge | Gt
```

FIGURE 2 – KONTIX

```
type t =
  { classname : string;
    code : instruction list;
    varsize : int;
    stacksize : int }

and instruction =
  | Comment of string
  | Label of label
  | Box
  | Unbox
  | Bipush of int
  | Pop
  | Swap
  | Dup
  | Binop of binop
  | Aload of var
  | Astore of var
  | Goto of label
  | If_icmp of cmpop * label
  | Anewarray
  | AAload
  | AAstore
  | Ireturn
  | Tableswitch of int * label list * label
  | Checkarray

and var = Var of int
and binop = Add | Sub | Mul | Div | Rem
and cmpop = Eq | Ne | Lt | Le | Gt | Ge
and label = string
```

FIGURE 3 – JAVIX