Compilation
Maîtrise d'Informatique 2001-2002

Examen Final
3 septembre 2002

L'énoncé est composé de 4 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites ainsi que les énoncés et supports de cours distribués cette année sont les seuls documents autorisés.
Les réponses aux questions devront être clairement justifiées.

1 Grammaires (6 points)

Soit la grammaire G définie sur l'ensemble des terminaux $\{\mathtt{x}, (,), \mathtt{;}\}$ et des non-terminaux $\{S, E\}$ avec S comme symbole de départ et les règles suivantes :

```
S ::= ( S E
S ::= x
S ::= ε
E ::= ; S E
E ::= )
```

1. Les mots `((x;x))` et `((x);x))` sont-ils reconnus par cette grammaire ?
2. Calculer les ensembles $\text{PREM}(S)$, $\text{PREM}(E)$, $\text{SUIV}(S)$ et $\text{SUIV}(E)$.
3. La grammaire G est-elle LL(1) ? LR(0) ? SLR(1) ? Donner les tables d'analyse correspondantes.

2 Analyse sémantique : arguments optionnels (7 points)

On se propose de modéliser un langage avec des arguments optionnels. Lors d'une déclaration de fonction, il est possible de donner une valeur par défaut à certains arguments. Lors de l'appel d'une fonction avec arguments optionnels, il est possible d'omettre les arguments optionnels (auquel cas la valeur par défaut est utilisée), on peut également forcer la valeur d'un paramètre optionnel en l'entourant de crochets carrés: $[e]$.

Par exemple on peut définir une fonction f de la manière suivante :

$$\textbf{let } f(x : \mathtt{int})[y = 2 : \mathtt{int}] : \mathtt{int} = x + 2 * y$$

Cela signifie que la fonction f admet deux arguments x et y mais que la valeur de y peut être omise au moment de l'appel et aura alors la valeur 2. On peut donc écrire $f\ 5$ qui vaudra $5 + 2 * 2 = 9$; si on veut forcer la valeur de y à 3, on écrira $f\ 5\ [3]$.

1. On suppose qu'un programme de notre langage est formé d'une succession de déclarations de fonctions suivant la syntaxe :

   $$\textbf{let } \mathtt{ident}\ \mathit{args}\ : \mathtt{typ} = \mathit{val}$$

   On se propose de construire une grammaire pour reconnaître notre langage. On suppose que `ident` et `typ` sont des symboles terminaux qui reconnaissent respectivement les identificateurs (variables et noms de fonctions) et les types primitifs du langage et que **let** est un mot clé. *args* et *val* sont des non-terminaux qui représentent respectivement les arguments formels des fonctions et les expressions du langage.

En généralisant l'exemple, donner des règles de grammaire pour le non-terminal *args* ainsi que les règles pour le non-terminal *val* permettant de reconnaître une valeur qui est l'appel d'une fonction à ses arguments.

2. À quelle phase de la compilation est-il judicieux de traiter les arguments optionnels ?
3. On suppose défini un type `asa_val` pour représenter les arbres de syntaxe abstraite des expressions du langage.
   (a) Proposer un type CAML pour représenter les profils des fonctions tels qu'ils seront stockés dans la table des symboles au moment de l'analyse sémantique.
   (b) Préciser le constructeur du type `asa_val` qui sera utilisé pour représenter l'appel de fonctions.
4. Un conflit dans la résolution des arguments peut apparaître si on déclare

$$\textbf{let } f(x:\texttt{int})[y=2:\texttt{int}][z=4:\texttt{int}]:\texttt{int} = x+z*y$$

   et que l'on écrit f 5 [3]. La valeur optionnelle [3] peut être prise comme valeur de l'argument y ou de l'argument z.
   Proposer une méthodologie pour résoudre ce conflit.

5. On décide d'écrire une fonction `explicite` qui, étant donné un arbre de syntaxe abstraite représentant une expression, vérifie qu'elle est bien typée et, si c'est le cas, renvoie une expression équivalente dans laquelle toute les valeurs optionnelles sont explicitées.
   Donner le code de cette fonction dans le cas de l'appel d'une fonction du langage à ses arguments.
6. Si on déclare :

$$\textbf{let } f(x:\texttt{int})[y=2:\texttt{int}][z=\texttt{true}:\texttt{bool}]:\texttt{int} = \texttt{if } z \texttt{ then } x \texttt{ else } y$$

   On peut choisir que l'expression f 5 [`false`] soit acceptée ou refusée par le système de type du langage. Elle est refusée si on considère que le premier argument optionnel requis par la fonction est de type attendu `int`; elle est acceptée si [`false`] est interprété comme le premier argument optionnel de type `bool` a' savoir z, la valeur par défaut est alors prise pour y.
   Quel est le comportement de la fonction `explicite` définie précédemment sur cet exemple ?
   Que faut-il modifier dans la fonction `explicite` pour avoir le comportement opposé ?
7. Peut-on utiliser les arguments optionnels pour améliorer les performances du code exécutable ? Dans quelles circonstances ?

3 Génération de code : `break`, `continue` (7 points)

On considère un mini-langage impératif *BC* comportant des conditionnelles et des boucles et seulement des variables globales. Ce langage comporte de plus deux instructions `break` et `continue` qui permettent d'interrompre l'exécution du corps d'une boucle. Lorsque dans le corps d'une boucle, on rencontre l'instruction `break` alors l'exécution se poursuit à la fin de la boucle, lorsque l'on rencontre l'instruction `continue` alors l'exécution reprend au test de la boucle. Lorsque plusieurs boucles sont imbriquées, les instructions `break` et `continue` permettent de sortir de la première boucle englobante.

On cherche à compiler le langage *BC* vers les instructions de la machine à pile vues en cours et rappelées à la fin de l'énoncé.

Les expressions du langage BC sont soit des constantes entières, soit des variables (globales) soit une opération binaire (arithmétique ou test d'égalité) appliquée à deux expressions, on convient que les booléens `true` et `false` sont représentés respectivement par les entiers 1 et 0.

Une instruction de BC peut être soit une affectation $x := e$, soit une conditionnelle `if` e `then` s_1 `else` s_2 `end` ou bien `if` e `then` s `end`, soit un bloc formé d'une liste d'instructions $\{s_1; \ldots; s_n\}$ soit une boucle `while` e `do` s `end`, soit une instruction d'échappement `break` ou `continue`.

Les types CAML suivants permettent de représenter les arbres de syntaxe abstraite de ce langage (les variables sont directement représentées par leur décalage par rapport à la base de la pile):

```
type op = Plus | Mult | Sub | Mod | Eq
type var = int
type expr = Const of int | Var of var | Op of op * expr * expr
type stat =   Aff of var * expr
            | If of expr * stat * stat
            | Block of stat list
            | While of expr * stat
            | Break | Continue
```

1. Écrire une fonction de compilation des instructions de BC vers la machine à pile dans le cas où il n'y a pas d'instruction `break` ou `continue`.
2. Soit le programme de BC :

   ```
   { p:=2;
     while true do
         if p=n then { r:=1; break } end;
         if p mod n = 0 then { r:=0; break }
         else p:=p+1 end
     end }
   ```

   Donner des instructions de la machines à pile correspondant au code compilé de cette fonction.
3. Écrire une fonction de compilation dans le cas général où les corps de boucle peuvent mentionner les instructions `break` et `continue`.
4. Dans un langage comme Java les instructions `break` ou `continue` peuvent mentionner un label qui a été introduit en tête d'une instruction `while`. Une instruction `break` ou `continue` sans label sort de la première boucle `while` englobante tandis qu'une instruction `break` (resp. `continue`) avec un label l sort de (resp. réitère) la boucle `while` portant le label l.

   Les constructeurs `While`, `Break` et `Continue` du type `stat` sont modifiés de la manière suivante :

   ```
   type label = None | Tag of string
   type stat = ...
               While of label * expr * stat
             | Break of label | Continue of label
   ```

   Modifier la fonction de compilation des instructions de BC pour traiter ce cas.

Machine à pile

Les instructions de cette machine à pile sont analogues à celles décrites dans le cours, on a juste ajouté une opération `MOD` qui calcule directement le reste de la division entière de deux nombres : On a $x \textbf{ mod } y = x - (x/y) * y$.

Le registre gp indique le bas de la pile, le registre sp indique la première case libre de la pile. Des instructions `LABEL` peuvent être introduites dans le code, elles ne modifient pas l'état de la pile. "Empiler x" signifie ajouter x au sommet de la pile (sp est augmenté de 1). "Dépiler x" signifie retirer la valeur au sommet de la pile qui sera notée x (sp est diminué de 1).

`PUSHI` n	empile la constante entière n.
`ADD`	dépile y puis x et empile $x + y$
`SUB`	dépile y puis x et empile $x - y$
`MUL`	dépile y puis x et empile $x * y$
`MOD`	dépile y puis x et empile x **mod** y
`EQ`	dépile y puis x et empile 1 si $x = y$ et 0 sinon
`PUSHG` n	empile la valeur située n cases au dessus de gp.
`STOREG` n	dépile x et le stocke à l'emplacement situé n cases au dessus de gp.
`PUSHN` n	empile n valeurs nulles sur la pile.
`POPN` n	dépile n valeurs de la pile.
`JZ` l	dépile x, si $x = 0$ le pointeur de code saute à l'instruction `LABEL` l.
`JUMP` l	saute à l'instruction `LABEL` l.