Compilation
Maîtrise d'Informatique 2002-2003

Examen Final

1 septembre 2003

L'énoncé est composé de 6 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.

Exercice A Passage paramètres (3 points)

On rappelle que les paramètres des méthodes Java sont passés dans une pile par valeur, ce sont soit des valeurs numériques soit des objets. La valeur d'un objet est une adresse dans le tas à laquelle sont stockées les valeurs des variables d'instances de l'objet.

```
class A {
  int x;
  A (int v) { x=v; }
  static void m1 (A y) {y = new A(1); }
}
class B {
  A a;
  B (int v) { a=new A(2); }
  static void m2 (B b) {b.a = new A(1); }
  public static void main (String argv[]) {
    A b = new A(0); A.m1(b);
    System.out.println(b.x);
    B c = new B(0);
    System.out.println(c.a.x);
    B.m2(c);
    System.out.println(c.a.x); }
}
```

Fig. 1 – Programme Java

1. Dire quel est le résultat de l'exécution du programme Java donné dans la figure 1.
2. Donner l'état de la pile et du tas aux 6 points de contrôle suivants : à l'entrée dans les procédures $m1$ et $m2$ ainsi que juste avant et juste après le retour de ces procédures.

Exercice B Grammaire (6 points)

Soit la grammaire G définie sur l'ensemble des terminaux $\{\texttt{new}, [,], \texttt{atom}, \sharp\}$ et des non-terminaux $\{S, E, T\}$ avec S comme symbole de départ et les règles suivantes :

```
S ::= E♯
E ::= atom
E ::= T
E ::= new T
T ::= E [E]
```

1. Calculer l'ensemble des suivants de E et de T.
2. Construire l'automate des items simples pour cette grammaire.
3. Cette grammaire n'est pas SLR(1).
   (a) Indiquer dans quel état se produit le conflit et la nature de celui-ci;
   (b) Donner une expression reconnue par la grammaire pour laquelle le conflit se produit;
   (c) La grammaire proposée est-elle ambiguë ? LALR(1) ?

Exercice C Compilation des expressions `switch` (11 points)

En C ou en Java, la structure de controle `switch` permet d'effectuer un branchement à choix multiples. La syntaxe d'une instruction `switch` est

switch (*expr*) {*opt-label-instr-list*}

où *expr* est une expression valide et le corps *opt-label-instr-list* du `switch` est une suite d'instructions chacune pouvant être préfixée par un label de choix. Un label de choix est soit `default`: soit un ou plusieurs labels de la forme `case` *cste*: avec *cste* une constante du langage.

Un exemple de programme utilisant cette instruction est donné dans la figure 2.

```
#define EOF -1
main()
{int c, nwhite, nother;
  nwhite = nother = 0;
  while ((c=getchar()) != EOF)
   switch (c) {
    case ’ ’:
    case ’\n’:
    case ’\t’:
      nwhite++; break;
    default:
     nother++; break;
    }
printf("white space = %d, other = %d\n",nwhite,nother);
}
```

FIG. 2 – Exemple de programme utilisant l'instruction **switch**

L'instruction **switch** doit vérifier que toutes les constantes apparaissant dans les labels **case** sont distinctes et qu'il y a au plus un label **default**. Dans la suite on supposera que ces conditions sont toujours vérifiées.

La sémantique de l'instruction **switch** (e) {li} est d'évaluer l'expression e en une valeur v, de chercher dans la suite d'instructions avec labels li s'il y a une étiquette **case** c telle que la valeur de la constante c est aussi v. Trois cas peuvent se produire :

- Si un label **case** c de valeur v existe, les instructions de li sont exécutées à partir de l'instruction comportant ce label,
- s'il n'existe pas de label de valeur v mais qu'il y a un label **default** alors les instructions de li sont exécutées à partir de l'instruction comportant le label **default**,
- s'il n'y a pas de label de valeur v ni de label **default** alors aucune instruction n'est exécutée.

Dans tous les cas l'instruction **break** apparaissant dans li provoque la sortie de l'instruction `switch`.

1. On suppose que la grammaire du langage est partiellement écrite avec des non-terminaux *cste*, *expr* et *instr* qui reconnaissent respectivement les expressions constantes, les expressions du langage, et les instructions et qui comportent au moins les règles :

expr	::=	`ident`	*cste*	::=	`'char'`
expr	::=	*cste*	*instr*	::=	**break**;
expr	::=	`ident++`	*instr*	::=	*expr* ;

   avec `char` et `ident` des terminaux correspondant aux unités lexicales des caractères et des identificateurs.

   Compléter la grammaire pour accepter les instructions **switch**.

2. Donner l'arbre de dérivation syntaxique de l'instruction :

   ```
   switch (c) {
    case ' ':
    case '\n':
    case '\t':
      nwhite++; break;
    default:
      nother++; break;
    }
   ```

3. On suppose définis des type de données `asa_cste`, `asa_expr` et `asa_instr` pour représenter respectivement les arbres de syntaxe abstraite des constantes, expressions et instructions. La figure 3 donne un sous-ensemble des constructeurs des arbres de syntaxe abstraite avec leurs types.

   Char : `char` → `asa_cste`
   Var : `string` → `asa_expr`
   Cste : `asa_cste` → `asa_expr`
   Incr : `string` → `asa_expr`
   Expr : `asa_expr` → `asa_instr`
   Break : `asa_instr`

   FIG. 3 – Constructeurs pour les arbres de syntaxe abstraite

   (a) Proposer un type pour le constructeur de l'arbre de syntaxe abstraite `asa_instr` correspondant à l'instruction `switch`.

   (b) Donner l'expression correspondant à l'arbre de syntaxe abstraite de l'instruction de la question 2.

4. On suppose donné une fonction *val_cste* de type `asa_cste` → `int` qui calcule la valeur (entière) d'une expression constante. Pour un caractère, il s'agit de son code ASCII. On rappelle que les codes ASCII de '\t', '\n' et ' ' sont respectivement 9, 10 et 32.

   On se propose de compiler notre langage vers les instructions de la machine à pile du cours dont un rappel est donné en Annexe D. On construit donc des fonctions *compil_expr* et *compil_instr* qui produisent à partir des arbres de syntaxe abstraite de notre langage initial, les instructions correspondantes de la machine à pile. Dans le cas d'une expression, sa valeur à la fin de l'exécution se trouve au sommet de la pile.

   Donner le schéma de compilation général pour une instruction `switch` en supposant que les fonctions *compil_expr* et *compil_instr* sont déjà définies pour les autres cas.

5. Donner la suite d'instructions de la machine à pile pour l'instruction de la question 2.

6. On ajoute maintenant à notre machine une instruction de saut indéxé `JUMPI` qui prend en argument un label l et effectue un saut du pointeur de code pc à l'adresse $l+i$ si i est la valeur du sommet de la pile (qui est dépilé).

   L'idée est de construire pour chaque instruction `switch` un tableau correspondant à l'éventail des cas possibles du `switch`.

   Dans le cas de l'exemple on a trois cas à tester : les caractères ' ', '\t' et '\n' correspondant aux valeurs numériques de $32, 9$ et 10. Il suffit donc de garder une table de choix pour les valeurs numériques entre 9 et 32. Dans cette table de taille $32-9+1=24$, les cases 0, 1 et 23 correspondant aux valeurs 9, 10 et 32 contiendront l'instruction `JUMP` l si l est le label de l'instruction qui suit ces cas; les autres cases contiendront l'instruction `JUMP` *def* si *def* est le label de l'instruction par défaut.

   On suppose que chaque instruction **switch** est prétraitée, et qu'on lui associe un label *choix* qui indique la place du tableau de choix, deux entiers *min*, *max* qui donnent la valeur minimum et la valeur maximum apparaissant dans les **case** ainsi qu'une étiquette *fin* pour marquer la fin du **switch** et une étiquette *def* pour marquer les instructions de la clause par défaut (cette étiquette est la même que l'étiquette *fin* s'il n'y a pas de clause par défaut) et finalement une liste d'étiquettes *lcases* pour chaque instruction préfixée d'un **case**.

   Dans le cas de notre exemple, le programme compilé comporterait une table d'initialisation décrite dans la figure 4.

choix:
`JUMP` l — indice 0 correspondant au caractère '\t' de valeur 9
`JUMP` l — indice 1 correspondant au caractère '\n' de valeur 10
`JUMP` *def* — indice 2 correspondant au caractère de valeur 11
`JUMP` *def* — indice 3 correspondant au caractère de valeur 12
`JUMP` *def* — indice 4 correspondant au caractère de valeur 13
`JUMP` *def* — indice 5 ...
`JUMP` *def* — indice 6 ...
`JUMP` *def* — indice 7 ...
`JUMP` *def* — indice 8 ...
`JUMP` *def* — indice 9 ...
`JUMP` *def* —indice 10 ...
`JUMP` *def* —indice 11 ...
`JUMP` *def* —indice 12 ...
`JUMP` *def* —indice 13 ...
`JUMP` *def* —indice 14 ...
`JUMP` *def* —indice 15 ...
`JUMP` *def* —indice 16 ...
`JUMP` *def* —indice 17 ...
`JUMP` *def* —indice 18 ...
`JUMP` *def* —indice 19 ...
`JUMP` *def* —indice 20 ...
`JUMP` *def* —indice 21 ...
`JUMP` *def* —indice 22 ...
`JUMP` l —indice 23 correspondant au caractère ' ' de valeur 32

FIG. 4 – Table d'initialisation du **switch**

À l'endroit d'une instruction **switch** (e) { li }, on insère le code pour evaluer l'expression e, le code pour effectuer le branchement indexé aproprié en fonction de la valeur de e puis enfin les instructions du corps du **switch** avec les étiquettes correspondantes. Dans le cas de notre exemple, on obtient :

```
—code pour la partie expression du switch
  PUSHG (decalage c)
—code pour effectuer le branchement aproprié en fonction de la valeur de c en utilisant la table stockée en choix et l'instruction JUMPI
...À COMPLÉTER...
l:
  code pour les instructions nbwhite++;break;
def:
  code pour les instructions nbother++;break;
fin:
```

(a) Compléter le code pour effectuer le branchement dans le cas de l'exemple.

(b) Donner le schéma général pour cette partie du code en supposant données les valeurs *min* et *max* ainsi que les étiquettes *choix* et *def*.

(c) Quels sont les avantages et les inconvénients de cette seconde méthode de compilation par rapport à la méthode qui n'utilise que les instructions de branchement `JZ` et `JUMP` ?

7. L'étudiant Gus doit écrire un programme interactif qui comprend en particulier les instructions `Find`, `Load`, `Write` et `Quit`. Il choisit d'associer à chaque instruction un caractère (la première lettre) et écrit son programme sous la forme :

```
switch (c) {
  case 'f':...
  case 'l':...
  case 'w':...
  case 'q':...
  default:...
}
```

Son collègue, qui préfère utiliser CAML, commence par définir un type de données `command` puis utilise du filtrage :

```
type command = Find | Load | Save | Quit
match (c) with
  | Find → ...
  | Load → ...
  | Save → ...
  | Quit → ...
```

Sachant que CAML associe à chaque constructeur un numéro et compile le filtrage à l'aide d'une instruction analogue à **switch**, quels avantages le compilateur peut-il tirer de cette deuxième approche ?

Annexe D Machine à pile

Les instructions de cette machine à pile sont analogues à celles décrites dans le cours.

Le registre *gp* indique le bas de la pile, le registre *sp* indique la première case libre de la pile. Des labels *label :* peuvent être introduits dans le code, ils 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 $x - y$
`INF`	dépile y puis x et empile 1 si $x < y$ et 0 sinon
`INFEQ`	dépile y puis x et empile 1 si $x \leq y$ et 0 sinon
`SUP`	dépile y puis x et empile 1 si $x > y$ et 0 sinon
`SUPEQ`	dépile y puis x et empile 1 si $x \geq y$ et 0 sinon
`EQUAL`	dépile y puis x et empile 1 si $x = y$ et 0 sinon
`NOT`	dépile x et empile 1 si $x = 0$ 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.
`POP` 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.