Compilation
Maîtrise d'Informatique 2002-2003

Examen Final

16 janvier 2003

L'énoncé est composé de 4 pages. L'examen dure 3 heures. Les notes de cours et de TD **manuscriptes** 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 Analyse syntaxique (6 points)

Soit G la grammaire dont l'ensemble des terminaux est {`id`,`op`,`:=`,`@`}, qui possède trois non-terminaux I (symbole initial), E et L, et dont les règles de production sont les suivantes :

(1)	I	≔	L `:=` E
(2)	L	≔	`id`
(3)	L	≔	L `@` `id`
(4)	E	≔	L
(5)	E	≔	E `op` L

1. Calculer l'ensemble des suivants de E, I et L.
2. La grammaire G est-elle SLR(1) ? Si oui, construire la table de transitions si non justifier votre réponse.
3. La grammaire G n'est pas LL(1), pourquoi ? Proposer une grammaire équivalente qui soit LL(1) et donner la table de transitions correspondante.

2 Analyse sémantique (4 points)

On se donne un langage qui permet de manipuler des structures d'enregistrement (records).

Description du langage Les types de ce langage sont soit des types de base (`bool` et `int`) soit des identificateurs correspondant à un type enregistrement défini par l'utilisateur.

Un type enregistrement est déclaré par la commande : `record` $id = \{lab_1 : \tau_1; \dots lab_n : \tau_n\}$ avec id le nom du type introduit, lab_i $(i = 1 \dots n)$ des identificateurs correspondant aux étiquettes de l'enregistrement et τ_i $(i = 1 \dots n)$ des types.

La syntaxe de ce langage est donnée par la grammaire suivante dans laquelle `INT`, `BOOL` et `ID` sont des terminaux représentant respectivement des constantes entières, booléennes et des identificateurs. Les autres terminaux de la grammaire sont les mots clés `int`, `bool`, `var`, `fun`, `record` et les symboles : `,`, `;`, `.`, `:`, `=`, `{`, `}`, `(` et `)`.

La grammaire des types (*typ*) et des suites d'identificateurs associés à des types (*idtypl*) est :

typ	≔	`bool` | `int` | `ID`
idtypl	≔	`ID` `:` *typ* | `ID` `:` *typ* `;` *idtypl*

Un programme est formé d'une suite de déclarations (*decl*) de records, fonctions et variables :

decl	≔	`record ID={`*idtypl*`}` | `fun ID(`*idtypl*`):`*typ*`=`*expr* | `var ID=`*expr*

Les expressions sont données par la grammaire :

expr	≔	`ID` | `INT` | `BOOL` | *expr*`.ID` | `ID{` *lablist* `}` | `ID(`*exprlist*`)`
lablist	≔	`ID = ` *expr* | `ID = ` *expr* `;` *lablist*
exprlist	≔	*expr* | *expr* `,` *exprlist*

- L'expression $f(e_1, \ldots, e_n)$ représente l'application de la fonction f aux arguments $e_1, \ldots, e_n$;
- L'expression $e.l$ représente la valeur du champ l de l'expression e.
- L'expression $t\{lab_1 = e_1; \ldots lab_n = e_n\}$ représente un enregistrement associant la valeur e_i à chaque étiquette lab_i. L'identificateur t est le nom du type record auquel appartient l'enregistrement; t est déclaré au préalable par : `record` $t = \{lab_1 : \tau_1; \ldots lab_n : \tau_n\}$.

Exemple : Le programme suivant définit le type des points en dimension 2 ainsi qu'un point p_2 particulier :

```
record point2 = {x:int;y:int}
var p2 = point2{x=0;y=1}
```

Notations : Si t est un type record défini par l'utilisateur, on appelle $labels(t)$ l'ensemble des labels apparaissant dans l'enregistrement et si l est un de ces labels, alors $labtyp(t, l)$ représente le type associé au label l dans l'enregistrement t.

Sous-typage Le langage autorise du sous typage entre les enregistrements. Chaque type est un sous-type de lui-même. Le type enregistrement t est un sous-type du type enregistrement u si l'ensemble $labels(u)$ est inclus dans $labels(t)$ et si pour tout l de $labels(u)$ on a $labtyp(u, l)$ est un sous-type de $labtyp(t, l)$. On note $t \leq u$ la propriété t est un sous-type de u. Si t est un sous-type de u alors toute expression e de type t est aussi une expression de type u.

Exemple : Ainsi si un programme définit les points en dimension 3 de la manière suivante :

```
record point3 = {x:int;y:int;z:int}
var p3 = point3{x=0;y=1;z=1}
```

alors `point3` est un sous-type de `point2` et `p3` est de type `point3` mais aussi de type `point2`.

Accès : L'expression $e.l$ est bien typée si e a pour type un type record t tel que l appartient à $labels(t)$; l'expression $e.l$ a alors pour type $labtyp(t, l)$.

Enregistrement : L'enregistrement $t\{lab_1 = e_1; \ldots lab_n = e_n\}$ a pour type t si t est un type record tel que $labels(t) = \{lab_1, \ldots, lab_n\}$ et chaque expression e_i a pour type $labtyp(t, lab_i)$.

Autres constructions : Les autres constructions satisfont les règles usuelles de typage.

Questions

1. (a) Les déclarations suivantes sont-elles bien typées ? On suppose que les types `point2` et `point3` sont définis comme précédemment :

   ```
   fun f (p:point2) : point2 = point3 {z=0;x=p.x;y=0}
   var t = f (point2 {x=1;y=1;z=1})
   ```

   (b) Écrire les règles de typage de ce langage sous forme d'un système d'inférence. On définira en particulier la relation $\Gamma \vdash expr : typ$ qui dit que l'expression *expr* admet pour type *typ* dans l'environnement Γ, ainsi que le jugement $\Gamma \vdash prog$ `ok` qui dit que le programme *prog* est valide dans l'environnement Γ. On prendra soin de préciser la forme de l'environnement Γ, on pourra introduire des notations et fonctions intermédiaires pour manipuler les profils, la table des symboles ...

2. Les types CAML suivant permettent de représenter les arbres de syntaxe abstraite de ce langage :

   ```
   type ident = string
   type typ = Bool | Int | Rec of ident
   type idtyp = ident * typ
   type expr =  Var of ident | Cint of int | Cbool of bool
             | Call of ident * expr list | Acc of expr * ident
             | Rec of ident * (ident * expr) list
   type decl = DFun of idtyp * (idtyp list) * expr | DVar of ident * expr
             | Drec of ident * idtyp list
   ```

   (a) Ecrire une fonction qui vérifie que les programmes sont bien formés.

   (b) Quel problème se poserait si l'information de type n'apparaissait pas dans les enregistrements ? Que pourrait-on faire pour y remédier ?

3 Génération de code (10 points)

Dans cet exercice, on utilisera les instructions de la machine à pile vue en cours et qui sont rappelées à la fin de ce document.

On considère un langage qui permet de définir des fonctions et des variables et ne manipule que des entiers. Un programme est formé d'une suite de déclarations. La syntaxe de ce langage est donnée par la grammaire suivante dans laquelle `INT`, `ID` sont des terminaux représentant respectivement des constantes entières et des identificateurs. Les autres terminaux de la grammaire sont les mots clés `if`, `then`, `else`, `var`, `fun` et les symboles : `,`, `=`, `(` et `)`.

decl	≔	`fun ID (` *idlist* `) =` *expr* | `var ID =` *expr*
expr	≔	`ID` | `INT` | `ID (` *exprlist* `)` | `if` *expr* `then` *expr* `else` *expr*
idlist	≔	`ID` | `ID ,` *idlist*
exprlist	≔	*expr* | *expr* `,` *exprlist*

Les arbres de syntaxe abstraite de ce langage sont représentés à l'aide des types CAML suivants :

```
type ident = string
type expr =  Var of ident | Cte of int | Call of ident * expr list
           | Cond of expr * expr * expr
type decl = DFun of ident * (ident list) * expr | DVar of ident * expr
type prog = decl list
```

On suppose que l'analyse sémantique a été faite et que chaque identificateur est unique et est stocké dans une table des symboles. On introduit un type `info` pour représenter les informations relatives aux variables. On suppose que l'on dispose de deux fonctions :

`update` : `ident` → `info` → `unit` et `access` : `ident` → `info`

qui permettent respectivement de mettre à jour l'information d'une variable dans la table des symboles et d'accéder à l'information associée à une variable dans cette table. On pourra également utiliser une fonction `new_label` : `unit` → `unit` qui engendre un label unique.

1. On définit le type `info` suivant :

   ```
   type info = {glob : bool ; dec : int}
   ```

   Le champ `glob` de l'information associée à une variable est vrai si cette variable est globale, `dec` représente alors le décalage par rapport à la base de la pile (*gp*) auquel cette variable sera stockée. Le champ `glob` est faux lorsque la variable est un paramètre de fonction auquel cas `dec` représente le décalage par rapport au frame-pointer (*fp*).

   Écrire une fonction qui étant donné un programme introduit dans la table des symboles les informations pour chaque variable.

2. Ecrire une fonction `codeProg` qui, étant donné un programme *p*, construit une suite d'instructions dont le résultat est de stocker sur la pile la valeur de chaque variable globale déclarée dans *p*.

3. On décide d'enrichir le langage pour autoriser des déclarations de variables locales. Pour cela, on ajoute à la grammaire le mot clé `in`, ainsi que la production *expr* ≔`var ID =` *expr* `in` *expr*. On ajoute au type CAML `expr` le constructeur `VarIn of ident * expr * expr`.

   (a) Écrire une fonction qui calcule le nombre maximal de variables simultanément visibles dans une expression.

   (b) Donner le code correspondant au programme suivant après avoir précisé à quel endroit dans la pile chaque variable est stockée. Les fonctions `inf`, `mult` et `prim` sont supposées prédéfinies et ne seront pas codées.

   ```
   fun f (x,y,z) = if inf(x,y) then var z0 = mult(2,x) in prim(z,z0)
                   else var z1 = mult(x,y) in prim(z1,x)
   var t = var z2 = 3 in f (z2,var z3=4 in mult(z2,z3),z2)
   ```

(c) Que faut-il changer aux fonctions de calcul d'information et de génération de code précédentes pour traiter les variables locales.

4. On suppose maintenant que le langage permet de plus la définition locale de fonctions. On remplace la règle de déclaration de variables locales par la production *expr*:=*decl* `in` *expr* et dans le type `expr` le constructeur `VarIn` par `Decl of decl * expr` (les types `expr` et `decl` sont alors définis de manière mutuellement récursive).

   (a) Quelle difficulté supplémentaire est introduite par cette possibilité ?

   (b) Donner l'organisation des variables, le niveau de chaque identificateur et le code correspondant au programme suivant :

```
fun f(x) = fun g(y) = mult(x,y) in var u = prim (x,x) in g(u)
var t = f(1)
```

   (c) On suppose définie une fonction `update_level` : `ident` → `int` → `unit` qui permet de stocker le niveau d'un identificateur de variable ou de fonction dans la table des symboles ainsi qu'une fonction `update_level` : `ident` → `int` qui permet d'accéder au niveau d'un identificateur dans la table des symboles.
   Écrire une fonction qui permet de mettre à jour la table des symboles avec le niveau de chaque identificateur d'un programme.

   (d) Modifier la fonction de génération de code pour prendre en compte les déclarations locales de variables et de fonctions.

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 *fp* indique le pointeur de référence des variables locales, 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 $x * y$
`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*.
`PUSHL` n	empile la valeur située n cases au dessus de *fp*.
`STOREL` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *fp*.
`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.
`CALL` f	sauve la valeur courante de *fp* et le pointeur d'instructions, affecte à *fp* la valeur de la première case libre de la pile, saute à l'instruction `LABEL` f.
`RETURN`	dépile toutes les valeurs au-dessus de *fp*, restaure l'ancienne valeur de *fp*, reprend l'exécution à l'instruction suivante la dernière instruction `CALL`.