Compilation
Master Informatique M1 2005-2006

Examen Partiel
3 novembre 2005

Les quatre exercices sont indépendants. L'énoncé est composé de 4 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.

1 Analyse Syntaxique (4 pts)

1. Soit la grammaire G_1 définie sur l'ensemble des terminaux $\{$`+`, `(`, `)`, `id`$\}$.

S' := S♯	A := (S)
S := S + A	A := id
S := A	

2. Cette grammaire est-elle LR(0) ? Le cas échéant, vous construirez la table de transitions de l'automate d'analyse.
3. Construire la dérivation droite de `id+(id+id)+id#` en faisant apparaître les états successifs de la pile et de la chaîne d'entrée.
4. Cette grammaire est-elle LL(1) ? Le justifier.
5. Construire une grammaire G'_1 LL(1) équivalente à G en factorisant les préfixes communs de G_1.
6. Calculer pour la grammaire G'_1 les ensembles $\textsc{prem}(X)$ et $\textsc{suiv}(X)$ pour chaque non-terminal. Construire la table des expansions de G'_1 pour l'analyse descendante.

2 Analyse Syntaxique (3 pts)

1. Soit la grammaire G_2 suivante définie sur l'ensemble des terminaux $\{$`+`, `*`, `(`, `)`, `id`$\}$.

S' := E♯	
E := E + T	E := T
T := T ∗ F	T := F
F := (E)	F := id

2. Cette grammaire est-elle LR(0), SLR(1), LR(1) ?
3. Écrire une grammaire équivalente en Yacc en utilisant des précédences (en simplifiant la grammaire).
4. Le type abstrait des expressions reconnues par la grammaire G_2 est défini ainsi :

```
type expr = Plus of expr * expr
          | Mult of expr * expr
          | Id of string
```

Modifier la grammaire Yacc pour produire des valeurs de type `expr`.

5. Écrire une fonction `afficher: expr -> unit` qui affiche à l'écran des valeurs de type `expr`. Cette fonction d'affichage fera en sorte que les expressions affichées puissent être reconnues par la grammaire Yacc en générant le même arbre de syntaxe abstraite. De plus, le nombre de parenthèses affichées devra être minimal. Ainsi :
   `afficher (Plus(Plus(Id("x1"), Mult(Id("x2"),Id("x3"))), Id("x4")))` devra afficher la chaîne : `x1 + x2 * x3 + x4`.
   Indication : On s'inspirera pour cela de la grammaire de départ G_2 en définissant un ensemble de fonctions mutuellement récursives. À chaque non terminal X de G_2, on fera correspondre une fonction `print_X: expr -> unit`.

3 Analyse lexicale : Automates (7 pts)

L'objectif de cet exercice est d'écrire des fonctions de reconnaissance de mots à partir d'un automate fini.

1. On se donne le type Caml suivant pour représenter les automates finis déterministes, où `etat` désigne le type (abstrait) des états.

```
type dautomate =
    { depart: etat;
      trans: etat -> char -> etat;
      final: etat -> bool }
```

   On suppose que la fonction `trans` lève l'exception `Not_found` lorsqu'il n'y a pas de transition.
   Écrire une fonction `reconnait: dautomate -> char list -> bool` telle que `reconnait a l` détermine si le mot `l` est reconnu par l'automate `a`.

2. On se donne maintenant le type suivant pour représenter les automates finis avec ϵ-transitions.

```
type eautomate =
    { depart: etat;
      trans: etat -> char -> etat;
      epsilon: etat -> etat list;
      final: etat -> bool }
```

   On remarquera que, pour tout état, il y a toujours au plus une transition étiquetée par un caractère donné.
   Écrire une fonction `reconnait: eautomate -> char list -> bool` telle que `reconnait a l` détermine si le mot `l` est reconnu par l'automate `a`.
   Indication : On écrira une fonction procédant par "backtracking" sans souci d'efficacité.

3. L'objectif est maintenant d'améliorer l'efficacité de la reconnaissance par un automate avec ϵ-transitions. On se donne un module `Etats` représentant des ensembles d'états.

```
module Etats: sig
  type t
  val empty : t
  val add : etat -> t -> t
  val fold : (etat -> 'a -> 'a) -> t -> 'a -> 'a
  val equal : t -> t -> bool
  val exists : (etat -> bool) -> t -> bool
  val singleton : etat -> t
end
```

(a) Écrire une fonction `suivants: eautomate -> Etats.t -> char -> Etats.t` telle que `suivants a s c` calcule l'ensemble des états reliés par `c` à des états de l'ensemble `s`.

(b) Écrire une fonction `epsilon_fermeture: eautomate -> Etats.t -> Etats.t` telle que `epsilon_fermeture a s` calcule l'ϵ-clôture de l'ensemble `s`.

(c) Écrire une fonction `reconnait: eautomate -> char list -> bool` qui détermine si un mot est reconnu en procédant ainsi : à chaque étape, on manipule un ensemble d'états s. On lit le caractère courant c du mot et on calcule un nouvel ensemble d'états comme étant l'ϵ-clôture des suivants de s par c. Une fois le mot entièrement lu, on vérifie que l'un des états de s est final.

4. On se donne le type suivant pour représenter les expressions régulières.

```
type rexp =
    Repsilon
  | Rlettre of char
  | Runion of rexp * rexp
  | Rproduit of rexp * rexp
  | Retoile of rexp
```

ainsi qu'une fonction `nouvel_etat: unit -> etat` permettant de créer un nouvel état.

Écrire une fonction `construire: rexp -> eautomate` construisant un automate fini avec ϵ-transitions à partir d'une expression régulière.

4 Analyse sémantique : Flot de contrôle (7 pts)

Les compilateurs C ou Java avertissent l'utilisateur lorsque dans une fonction ou une méthode il manque une instruction `return` ou au contraire lorsqu'une instruction ne sera à coup sûr jamais exécutée. Le problème n'est pas décidable en toute généralité, mais devient simple dès lors que l'on fait abstraction de la valeur des tests. C'est cette situation que nous envisageons dans cet exercice.

On dit qu'une instruction *retourne toujours* si, quel que soit le flot de contrôle emprunté lors de l'évaluation de cette instruction, on finit toujours par atteindre une instruction `return`. On dit qu'une instruction est *inatteignable* si elle est située dans un flot de contrôle au delà d'une instruction qui retourne toujours.

1. Pour chacune des fonctions/méthodes suivantes, indiquer d'une part si son corps retourne toujours et d'autre part quelles sont les instructions inatteignables, le cas échéant.

   (a) `int m1() { if (b) { return 1; } x = 1; }`

   (b) `int m2() { if (b) { return 1; } else { return 2; } x = 2; }`

   (c) `int m3() { while (b) { return 1; } }`

2. On se donne le type Caml `instr` suivant pour représenter les arbres de syntaxe abstraite des instructions C ou Java :

```
type instr =
  | Expr of expr
  | If of expr * instr * instr
  | While of expr * instr
  | Bloc of instr list
  | Return of expr
```

Il n'est pas utile de préciser ce qu'est le type `expr` des expressions. Écrire une fonction Caml `retourne : instr -> bool` qui indique si une instruction retourne toujours.

3. Écrire une fonction Caml `nettoie : instr -> instr` qui supprime dans une instruction toutes les instructions inatteignables. (On réutilisera la fonction `retourne`, même s'il serait plus efficace de coder les deux fonctions en une seule.)
4. Plutôt que de supprimer les instructions inatteignables, on préfère maintenant échouer si l'on en rencontre. Modifier la fonction `retourne` de la question 2 pour qu'elle lève l'exception

   ```
   exception Inatteignable of instr
   ```

   dès lors qu'elle rencontre une instruction inatteignable (l'argument de l'exception étant précisément cette instruction).
5. On se pose maintenant la question de supprimer les fonctions (ou méthodes) inatteignables dans un programme. On se donne pour cela un module Caml `S` pour représenter des ensembles de chaînes de caractères :

   ```
   module S : sig
     type t (* type des ensembles de chaines de caracteres *)
     val singleton : string -> t
     val union : t -> t -> t
     val equal : t -> t -> bool
     val fold : (string -> 'a -> 'a) -> t -> 'a -> 'a
   end
   ```

   et une fonction `appels : string -> S.t` donnant pour chaque fonction du programme l'ensemble des fonctions qu'elle appelle. On suppose que le point d'entrée du programme est la fonction `"main"`. Écrire un programme Caml calculant l'ensemble des fonctions atteignables i.e. susceptibles d'être appelées dans une exécution du programme.