Compilation
Maîtrise d'Informatique 2003-2004

Examen Partiel
12 novembre 2003

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 lexicale (3 pts)

Le langage CAML permet de définir des fonctions qui au lieu d'être nommées par un identificateur, sont référencées par une suite de symboles. Ainsi la fonction d'addition en CAML peut s'écrire avec le symbole `+` utilisé de manière infixe :

```
#fun x y -> x + y;;
- : int -> int -> int = <fun>
```

ou plus simplement en utilisant l'opérateur entre deux parenthèses :

```
# (+);;
- : int -> int -> int = <fun>
```

L'expression `a+b` est équivalente à l'évaluation de `(+) a b`. L'opérateur `!` qui permet d'accéder à la valeur d'une référence s'utilise de manière préfixe te suit le même schéma :

```
#let x = ref 0 in !x;;
- : int = 0
# (!);;
- : 'a ref -> 'a = <fun>
```

L'utilisateur peut introduire ses propres notations infixes en définissant une fonction référencée par une suite de symboles entre parenthèses. On peut ainsi définir l'implication booléenne :

```
# let (=>) x y = if x then y else true;;
val ( => ) : bool -> bool -> bool = <fun>
```

et l'utiliser de manière infixe :

```
# false => true;;
- : bool = true
```

Pour traiter cette possibilité, le lexer de CAML interprète a priori les suites de symboles spéciaux comme des opérateurs (préfixes ou infixes en fonction du premier symbole) qu'ils soient ou non définis comme opérateur primitif (cas de `+`, `*` etc) ou par l'utilisateur.

Une version partielle simplifiée du parser de CAML est le suivant :

```
let blank = [' ' '\n' '\t']
let identchar = ['A'-'Z' 'a'-'z' '_' '\'' '0'-'9']
let startinfix = ['=' '<' '>' '|' '&' '$' '+' '-' '*' '/' '%']
let symbolchar =
  ['!' '$' '%' '&' '*' '+' '-' '.' '/' ':' '<' '=' '>' '?' '@' '^' '|' '~']
let lowercase = ['a'-'z' '_']
let decimal_literal = ['0'-'9'] ['0'-'9']*

rule token = parse
    blank +                     { token lexbuf }
  | lowercase identchar *       { LIDENT (Lexing.lexeme lexbuf) }
  | decimal_literal             { INT (int_of_string(Lexing.lexeme lexbuf)) }
  | "="                         { EQUAL }
  | ":="                        { COLONEQUAL }
  | "!="                        { INFIXOP "!=" }
  | ['!' '~' '?'] symbolchar * { PREFIXOP(Lexing.lexeme lexbuf) }
  | startinfix symbolchar *     { INFIXOP(Lexing.lexeme lexbuf) }
```

1. Quelle est la suite de tokens renvoyée par cet analyseur pour l' expression suivante :

   `x != y +. !z !=-t':==/`

2. Une erreur se produit lorsque l'on tape l'expression suivante :

   `let x = ref 0  in x:=1+!y`

   (a) Donner la suite de tokens correspondant à cette expression.

   (b) Expliquer la nature et l'origine de l'erreur.

   (c) Quelle déclaration préalable aurait pu rendre cette expression correcte ?

   (d) Comment doit-on transformer cette expression sur le plan lexical pour obtenir le comportement attendu (mettre dans la référence `x` la somme de 1 et `x`) ?

2 Analyse syntaxique (8 pts)

Soit la grammaire G définie sur l'ensemble des terminaux $\{$`id`$, (,), ;\}$ et des noms terminaux $\{S, E\}$ par les règles suivantes :

S := E ♯
E := `id`
E := `id`(E)
E := (E)
E := (E;E)

1. Montrer que la grammaire G est SLR(1), construire la table d'analyse ascendante correspondante.
2. En utilisant la table précédente, donner les étapes de l'analyse ascendante du mot `id((id;id))`♯ en faisant apparaître les états dans la pile.
3. La grammaire G est-elle LR(0) ? LR(1) ?
4. Cette grammaire n'est pas LL(1), expliquer pourquoi.

5. Construire une grammaire G' équivalente à G dans laquelle les règles de G ayant un préfixe commun sont factorisées.
6. Calculer pour la grammaire G' l'ensemble $\textsc{prem}(X)$ pour chaque non-terminal X et l'ensemble $\textsc{suiv}(X)$ pour chaque non-terminal X différent de S.
7. Construire la table des expansions de G' pour l'analyse descendante. La grammaire G' est-elle LL(1) ?

3 Construction d'un automate fini à partir d'une expression régulière (10 pts)

On remarque tout d'abord que si un automate fini reconnaît le langage correspondant à l'expression régulière r alors à chaque lettre lue sur l'entrée on peut faire correspondre une occurrence d'une lettre dans l'expression r. On indexe donc chaque occurrence d'une lettre dans l'expression régulière par un entier de manière à distinguer les différentes occurrences d'une même lettre.

On prend comme exemple l'expression régulière $(a|b)^*a(a|b)$. Après avoir indexé les caractères, on obtient l'expression $(a_1|b_1)^*a_2(a_3|b_2)$. Si l'entrée est le mot $aabaab$, alors il correspond à l'expression régulière de la manière suivante:

$$a_1a_1b_1a_1a_2b_2$$

L'idée est de construire un automate dont les états sont des ensembles de lettres indexées correspondant aux occurrences qui peuvent être lues à un instant. Au départ on regarde quelles sont les premières lettres possibles. Dans notre exemple, il s'agit de a_1, b_1, a_2.

Pour construire les transitions, il suffit de pouvoir calculer pour chaque occurrence d'une lettre, les occurrences qui peuvent la suivre. Par exemple si on vient de lire a_1 alors les caractères suivants possibles sont a_1, b_1, a_2 ; si on vient de lire a_2 alors les caractères suivants possibles sont a_3, b_2. Si on est dans l'état initial $\{a_1, b_1, a_2\}$ et qu'on lit un a alors c'est soit a_1 soit a_2 et donc les caractères qui peuvent être lus ensuite sont a_1, b_1, a_2, a_3, b_2.

Les caractères indexés par des entiers et les expressions régulières seront définies par les types Caml suivants :

```
type char_i = char * int
type expreg =
  | Epsilon
  | Caractere of char_i
  | Union     of expreg * expreg
  | Produit   of expreg * expreg
  | Etoile    of expreg
```

1. Définir une fonction `null` : `expreg` $\rightarrow$ `bool` qui détermine si le mot vide ϵ appartient au langage défini par une expression régulière donnée.
2. On suppose donné un module `Ensc` fournissant un type abstrait `t` pour des ensembles de caractères indexés, ayant la signature suivante :

```
module Ensc : sig
  type t
```

```
  val vide : t
  val appartient : char_i -> t -> bool
  val singleton : char_i -> t
  val union : t -> t -> t
  val elements : t -> char_i list
end
```

Définir deux fonctions `debut` (resp. `fin`) : `expreg` $\rightarrow$ `Ensc.t` calculant l'ensemble des caractères qui peuvent apparaître au début (resp. à la fin) des mots reconnus par une expression régulière.

3. Les caractères suivant c dans une expression régulière r sont caractérisés par la propriété suivante :

   $d \in \mathit{suivant}(c, r)$ ssi il existe une sous-expression de r de la forme $r_1 r_2$ avec $d \in \mathit{d\acute{e}but}(r_2)$ et $c \in \mathit{fin}(r_1)$,
   ou bien
   il existe une sous-expression de r de la forme r_1^* avec $d \in \mathit{d\acute{e}but}(r_1)$ et $c \in \mathit{fin}(r_1)$

   (a) Calculer $\mathit{suivant}(c, r)$ pour $r = (a_1|b_1)^* a_2(a_3|b_2)$ et $c \in \{a_1, a_2, a_3, b_1, b_2\}$.

   (b) Écrire une fonction `suivant` : `char_i` $\rightarrow$ `expreg` $\rightarrow$ `Ensc.t` codant *suivant*.

4. Pour construire l'automate déterministe, on procède de la manière suivante sur l'expression régulière indexée r:

   (i) On ajoute un nouveau caractère $\sharp$ à la fin de l'expression.

   (ii) Les états de l'automate sont des ensembles de caractères indexés. L'état initial est $\mathit{d\acute{e}but}(r)$, les états d'acceptation sont tous ceux qui contiennent $\sharp$. On a une transition de q vers q' à la lecture du caractère c si

   $$q' = \bigcup_{c_i \in q} \mathit{suivant}(c_i, r)$$

   (a) Utiliser cet algorithme pour construire un automate déterministe reconnaissant le langage décrit par $r = (a_1|b_1)^* a_2(a_3|b_2)$. *Attention* : les calculs de *suivant* doivent prendre en compte l'adjonction du caractère $\sharp$.

   (b) On suppose donné un module `Automate` fournissant une structure de données pour un automate fini déterministe, ayant l'interface suivante :

   ```
   module Automate : sig
     val ajoute_etat : Ensc.t -> unit
       (* crée un nouvel état; lève l'exception DejaVu s'il existe déjà *)
     val ajoute_transition : Ensc.t -> char -> Ensc.t -> unit
       (* (ajoute_transition q c q') ajoute la transition q--c-->q' *)
     val transition : Ensc.t -> char -> Ensc.t
       (* (transition q c) retourne l'état associé à c depuis q s'il existe,
          et lève l'exception Not_found sinon *)
   end
   ```

   Écrire une fonction `automate` : `expreg` $\rightarrow$ `unit` construisant l'automate associé à une expression régulière donnée.

   (c) Écrire une fonction `reconnait` : `expreg` $\rightarrow$ `string` $\rightarrow$ `bool` testant l'appartenance d'un mot au langage décrit par une expression régulière.