Compilation
Master Informatique M1 2010-2011

Examen Partiel
27 octobre 2010

Les questions sont largement indépendantes. L'énoncé est composé de 5 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites et les supports de cours distribués cette année sont les seuls documents autorisés.

Consigne Lorsqu'il est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ses détails (vous pouvez utiliser des fonctions auxiliaires dont l'implantation est directe en vous contentant de spécifier leur comportement). Vous devez par contre décrire les données manipulées et les algorithmes utilisés de manière précise.

Le langage considéré: HA

Dans tout l'énoncé, on s'intéresse à un langage de programmation HA qui s'inspire du langage Haskell.

Conventions lexicales

Identifiants Les identifiants sont formés de caractères alphabétiques minuscules ou majuscules, de caractères numériques, ils commencent par une minuscule.

Mots-clés Les mots-clés (mots réservés) sont **if**, **then**, **else**, **let**, **in**

Caractères spéciaux Les caractères spéciaux sont: `( ) ,` et `;` (parenthèse gauche, droite, virgule et point-virgule).

Symboles infixes Les symboles infixes représentent des fonctions binaires `op` pour lesquelles on utilise la syntaxe e_1 `op` e_2 pour représenter l'application de `op` aux expressions e_1 et e_2. Dans HA, les symboles infixes sont formés d'une suite de caractères appartenant à l'ensemble: {`!`,`&`,`+`,`*`,`/`,`<`,`=`,`>`,`|`, `-`,`:`}. Parmi ces suites de caractères on distingue les *symboles réservés* qui sont: `::`, =, `->` et les suites d'au moins deux symboles - qui servent pour les commentaires.

Commentaires Les commentaires sont de deux sortes:
- les *commentaires simples* commencent par une suite `--` de deux ou plus symboles - et finissent au premier retour à la ligne.
- les *commentaires imbriqués* sont délimités par les suites de symboles `{-` et `-}`.

Grammaire

Types Les types de ce langage sont soit des types de base: **int** ou **bool**, soit le type (noté τ_1`->`τ_2) des fonctions qui prennent un argument de type τ_1 et renvoient un résultat de type τ_2, soit le type (noté `(`$\tau_1, \ldots, \tau_n$`)`) des n-uplets $(v_1, \ldots, v_n)$ où chaque v_i a pour type τ_i.
Ainsi une fonction qui prend en argument un booléen et un couple d'entiers et calcule un entier aura pour type: `bool -> (int,int) -> int`.
La flèche `->` associe à droite et des parenthèses peuvent être utilisées pour lever les ambiguités: ainsi une fonction de type `(int -> int) -> int` prend en argument une fonction de type `int->int` et calcule un entier.

Expressions simples Un expression simple est soit une constante entière, soit une constante booléenne (`true` ou `false`), soit un identifiant, soit un symbole infixe utilisé de manière préfixe en le parenthésant : `(`*infix*`)` , soit une expression parenthésée `(`e`)`, soit un n-uplet formé d'expressions séparées par des virgules: `(`e_1`, ... ,`e_n`)`.

Expressions générales les expressions générales sont soit des expressions simples, soit l'application notée $e_1\, e_2$ d'une expression e_1 à une expression simple e_2, soit une expression infixe e_1 `op` e_2 avec `op` une suite de symboles représentant un symbole infixe, soit une déclaration locale de la forme **let** *ident* $=$ e_1 **in** e_2, soit une expression conditionnelle **if** e_1 **then** e_2 **else** e_3.

Déclarations et Programme Les déclarations peuvent être de trois formes:
- déclaration du type d'une fonction: `ident :: typ`
- définition de valeur de la forme: `ident` = *expr*
- définition de fonction de la forme: `ident` $x_1 \ldots x_n$ = *expr*

Les déclarations sont séparées par des points virgules. Un programme est une suite de déclarations.

Un *exemple* de programme (bien formé) est:

```
x = 2+3;
p = f 2 4;
f :: int -> int -> (int,int,int);
f x y = (1,x+y,x-y);
g :: ((int,int,int) -> bool) -> bool;
g h = h p || h (0,1,0)
```

Arbres de syntaxe abstraite

Dans la suite, pour représenter les arbres de syntaxe abstraite des programmes on introduit les types Ocaml suivants:

```
type cst = Int of int | Bool of bool
type typ = TInt | TBool | Prod of typ list | Arr of typ * typ
```

qui représentent respectivement les arbres de syntaxe abstraite pour les constantes (entières ou booléenes) et les expressions de types. Le type des n-uplets $(\tau_1, \ldots, \tau_n)$ est représenté en Ocaml par `Prod`$([\tau_1; \ldots; \tau_n])$ et le type des fonctions τ_1->τ_2 est représenté par l'expression Ocaml `Arr`(τ_1, τ_2).

1 Analyse lexicale (6 points)

On se donne le fichier ocamllex (incomplet) suivant.

```
{
 let kwd_tbl =
  ["int",INT;"bool",BOOL;"true",CST(Bool true);"false",CST(Bool false);
   "let",LET; "in",IN; "if",IF;"then",THEN;"else",ELSE]
 let id_or_kwd s = try List.assoc s kwd_tbl with _ -> IDENT s
}
let low = ['a'-'z']
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let integer = ['0'-'9']+
let space = [' ' '\t' '\n']
let symb = ['!' '&' '*' '+' '/' '-' ':' '>' '<' '=' '|']
let ident = low (letter | digit)*
let dashes = "--"('-')*

rule token = parse
  | space+          { token lexbuf }
  | ident as id     { id_or_kwd id }
  | integer as s    { CST (Int(int_of_string s)) }
  | '('             { LP }
  | ')'             { RP }
  | ','             { VIRG }
  | ';'             { PV }
  | eof             { EOF }
  | _ as c          { raise (Lexing_error c) }
```

1. Compléter les règles d'analyse lexicale pour traiter les commentaires simples et les commentaires imbriqués.
2. Le langage HA autorise la définition de fonctions binaires qui seront notées de manière infixe. Si `op` est une telle fonction, et e_1 et e_2 sont deux expressions, on écrira e_1 `op` e_2 pour représenter l'expression (`op` e_1 e_2). D'un point de vue lexical, `op` est une suite de *symboles ordinaires* (voir liste ci-dessus). Les symboles arithmétiques et logiques comme `+`, `*`, `>=`, `||` sont des cas particuliers de cette convention.
   (a) Ajouter à l'analyseur lexical, une règle permettant de reconnaître de tels opérateurs.
   Le lexème associé sera noté `INFIX` et sa valeur sera la chaîne de caractères correspondant à l'opérateur. Ainsi le symbole d'addition `+` devra produire le léxème `INFIX("+")` tandis que le symbole logique de disjonction `||` produira le lexème `INFIX("||")`.
   (b) Certaines suites de symboles jouent un rôle particulier dans la grammaire, on leur associe des léxèmes spécifiques. Pour cela, on ajoute dans l'analyseur lexical (avant la règle générique pour les symboles infixes) les règles suivantes:

```
| "::"              { DDP }
| '='               { DEF }
| '!'               { NEG }
| "->"              { ARR }
| '-'               { MINUS }
```

Donner la suite de lexèmes produite dans l'analyse de l'entrée suivante (justifier brièvement le résultat):

```
3 + x -- foo
3 || x - - foo
3 ||+ x --- foo
3 -> x --+ foo
```

2 Analyse Syntaxique (6 points)

Le prélude du fichier ocamlyacc pour la syntaxe de HA déclare les tokens suivants:

```
%token <cst> CST
%token <string> IDENT
%token <string> INFIX
%token NEG MINUS
%token LET IN IF THEN ELSE
%token LP RP VIRG PV DDP ARR DEF
%token INT BOOL
%token EOF
```

1. (a) Donner les règles de grammaires pour reconnaître les types du langage HA tels qu'ils sont décrits en préambule. On construira dans l'action l'arbre de syntaxe abstraite correspondant.
   (b) Votre grammaire est-elle ambigüe ? si oui indiquer comment régler la précédence des tokens pour obtenir le comportement attendu.
2. La grammaire (partielle) des expressions comporte les déclarations suivantes (les symboles infixes ont tous la même priorité et associent à gauche):

```
%left INFIX
%%
expr:
| aexpr                          { $1 }
| expr aexpr                     { App($1,$2) }
| NEG aexpr                      { App(Var "neg", $2) }
| MINUS aexpr                    { App(Var "uminus", $2) }
| expr INFIX expr                { App(App(Var $2, $1), $3) }
| LET IDENT DEF expr IN expr     { Letin ($2, $4, $6) }
| IF expr THEN expr ELSE expr    { If($2, $4,$6) }

aexpr:
| CST                            { Cst($1) }
| IDENT                          { Var($1) }
| LP expr RP                     { $2 }
| LP INFIX RP                    { Var($2) }
;
```

Un des états de cette grammaire indique les conflits suivants:

```
35: shift/reduce conflict (shift 16, reduce 14) on CST
35: shift/reduce conflict (shift 17, reduce 14) on IDENT
35: shift/reduce conflict (shift 21, reduce 14) on LP
state 35
expr : expr . aexpr  (11)
expr : expr . INFIX expr  (14)
expr : expr INFIX expr .  (14)
```

   (a) Donner un exemple d'une entrée pour laquelle cette situation de conflit se produit.
   (b) Indiquer comment ordonner les précédences pour que l'application $(f\ e)$ soit prioritaire par rapport aux opérateurs infixes.

(c) Expliquer pourquoi ocamlyacc n'indique pas dans cet état de conflit pour la lecture du token `INFIX`.

3. En pratique tous les symboles infixes n'ont pas la même priorité. Les symboles prédéfinis comme +, >=, . . .ont des priorités relatives fixées que l'on numérote de 1 à 9, la priorité 9 étant la plus forte. L'utilisateur peut fixer la priorité d'un nouvel opérateur qu'il introduit par une déclaration spéciale, de la forme **infixl** n `op` avec n un entier de 1 à 9. En l'absence de déclaration explicite la priorité d'un nouvel opérateur est fixée par défaut à 9. En pratique, on introduit dans la grammaire neuf tokens différents `INFIX1`, . . ., `INFIX9` correspondant aux différents niveaux de priorité. On ne considère ici que des opérateurs qui associent à gauche. Lorsqu'on rencontre une suite de symboles correspondant à un opérateur infixe, il faut produire le léxème `INFIX`i qui convient.
   Que faut-il modifier dans l'analyseur lexical et l'analyseur syntaxique pour prendre en compte ces priorités?
   Dans la phase d'analyse lexicale, on pourra penser à enregistrer dans une table les priorités déclarées à l'aide du mot clé **infixl**. Les opérateurs infixes pourront ensuite être traités en s'inspirant du traitement des mot-clés.

3 Analyse sémantique (6 points)

Les règles de portées autorisent l'ensemble des fonctions à être définies de manière mutuellement récursives. On demande que pour chaque fonction du programme définie par:

$f\ x_1\ \ldots\ x_n = e$

le programme contienne également une déclaration du type de cette fonction de la forme:

$f\ ::\ \tau$

Par ailleurs un certain nombre de fonctions sont prédéfinies, en particulier les opérateurs arithmétiques, booléens et les fonctions de comparaison. On suppose qu'une table de visibilité `prim` contient toutes les fonctions prédéfinies avec pour chacune son type. Par exemple + a pour type `int -> int -> int`, <= a pour type `int -> int -> bool`, `||` a pour type `bool -> bool -> bool` et `uminus` a pour type `int -> int`.

Contrairement aux fonctions, les définitions de variables:

$x = e$

ne sont pas forcément associées à des déclarations et ne peuvent être utilisés qu'après leur définition.

1. Donner un type Ocaml pour représenter les arbres de syntaxe abstraite pour les expressions (on complètera le type utilisé dans la grammaire précédente pour prendre en compte les n-uplets), les déclarations et les programmes.
2. Préciser les fonctions dont vous aurez besoin pour manipuler la table de visibilité.
3. Ecrire une fonction qui étant donné un programme p, complète la table de visibilité des opérations primitives `prim` avec toutes les fonctions déclarées.
4. Ecrire une fonction qui étant donnés une table de visibilité comportant les objets visibles (fonctions et variables), et une expression e, vérifie que l'expression e est bien typée et calcule son type.
5. Ecrire une fonction qui vérifie qu'une déclaration de fonction est bien formée et que son type correspond bien à la déclaration.
6. En déduire une fonction qui vérifie qu'un programme est bien formé.

```
let rec check_decls vis = function
   [] -> ()
| Fdecl(_,_)::p -> check_decls vis p (* type declarations are already in vis *)
| Fdef(id,[],e)::p ->
    let tau = type_expr vis e in
    let vis' = Env.add id tau vis in  check_decls vis' p
| Fdef(id,lx,e)::p -> let tau = Env.find id vis in
    check_fun vis lx e tau; check_decls vis p

let check_program p =  let vis = add_decls p in check_decls vis p
```

4 Filtrage (4 points)

On étend le langage HA pour inclure des *motifs* qui permettent d'associer des noms de variables aux composantes d'un n-uplet. Un motif est soit une variable x, soit un n-uplet de motifs $(p_1, \ldots, p_n)$ soit le motif universel noté _. Le type des arbres de syntaxe abstraite des motifs sera défini par:

```
let pattern = Pvar of string | Puniv | Ptuple of pattern list
```

En pratique on étend la construction **let** $x = e_1$ **in** e_2 au cas **let** $p = e_1$ **in** e_2 où p est un motif.

Par exemple, on pourra implanter l'addition de deux points ou encore l'accès à la première composante d'une paire d'entiers par les fonctions:

```
addp :: (int,int,int) -> (int,int,int) -> (int,int,int);
addp p1 p2 =
  let (x1,y1,z1) = p1 in let (x2,y2,z2) = p2 in
  (x1+x2,y1+y2,z1+z2);
fsti :: (int,int) -> int;
fsti p = let (x,_) = p in x
```

Les motifs et les expressions associées doivent se correspondre. Par exemple si e est une expression qui a pour type `((int,int),int)`, il peut lui correspondre le motif universel _, un motif formé d'une seule variable x qui sera de type `((int,int),int)`, un motif formé d'un couple de variables (x_1, x_2) avec x_1 de type `(int,int)` et x_2 de type `int` ou un motif $((x_1, y_1), x_2)$ avec x_1 de type `int`, y_1 de type `int` et x_2 de type `int`. Par contre le motif $(x_1, (x_2, \_))$ ne pourra pas être associé à l'expression e car le motif $(x_2, \_)$ de la seconde composante ne correspond pas au type attendu `int`.

On définit formellement une relation de bonne formation $(p : \tau) \rightsquigarrow l$ qui étant donnés un motif p et un type τ définit quand le motif p est bien formé par rapport au type τ et correspond à la liste d'associations l entre les variables du motif et leur type.

$$\frac{}{\langle \_ : \tau \rangle \rightsquigarrow []} \qquad \frac{}{\langle \mathtt{x} : \tau \rangle \rightsquigarrow [(x, \tau)]} \qquad \frac{\langle p_1 : \tau_1 \rangle \rightsquigarrow l_1 \ldots \langle p_n : \tau_n \rangle \rightsquigarrow l_n}{\langle (p_1, \ldots, p_n) : (\tau_1, \ldots, \tau_n) \rangle \rightsquigarrow l_1 @ \ldots @ l_n}$$

1. Dire si les motifs et les types suivants sont bien formés et si oui construire l'association des variables et des types correspondante.
   (a) motif: $(x, \_, (y, z))$ type: `(int -> int,(bool,bool),(int,bool))`
   (b) motif: (x, y) type: `(int -> int,bool,int)`.
2. Ecrire une fonction qui étant donnés un motif p et un type τ vérifie que le motif correspond au type et calcule la liste d'associations l telle que $\langle p : \tau \rangle \rightsquigarrow l$ et échoue sinon.
3. La règle de typage des expressions **let in** avec motif peut alors s'écrire:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad (p : \tau_1) \rightsquigarrow l \quad \Gamma + l \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let } p = e_1 \textbf{ in } e_2 : \tau_2}$$

   En déduire le cas correspondant de la définition de la fonction de typage d'une expression:

```
let rec type_expr vis = function
...
LetIn(p,e_1,e_2) ->
...
```