Compilation
Master Informatique M1 2008-2009

Examen Partiel
30 octobre 2008

Les exercices sont indépendants. L'énoncé est composé de 10 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 (7 points)

Soit les grammaire G_1, G_2, G_3 suivantes définies sur l'ensemble des terminaux $\{$`;` , `a`$\}$.

G_1		
S	::=	L
L	::=	L; a | a

G_2		
S	::=	L
L	::=	a;L | a

G_3		
S	::=	L
L	::=	L;L | a

1. Donner sous forme d'expression régulière le langage reconnu par ces 3 grammaires.
2. (a) Donner les étapes de l'analyse ascendante du mot $a; a; a$ par les deux premières grammaires (on indiquera l'état de la pile, le mot d'entrée et les actions).
   (b) Que constate-t-on sur la taille de la pile en fonction de la taille du mot d'entrée ? Laquelle des deux grammaires vous paraît-elle la plus efficace pour l'analyse ascendante ?
3. Montrer que la grammaire G_3 est ambiguë.
4. On entre la grammaire suivante dans `ocamlyacc` correspondant à la grammaire G_3 dans laquelle le terminal `;` est noté `pv` et où on a ajouté un terminal `eof` pour la fin de fichier.

```
%token a pv eof
%%
S: L eof
;
L: L pv L
 | a
;
```

`ocamlyacc` indique un conflit shift/reduce et le fichier d'information contient les éléments suivants sur la grammaire :

```
1  S : L eof

2  L : L pv L
3    | a
```

et sur l'état de conflit:

```
8: shift/reduce conflict (shift 6, reduce 2) on pv
state 8
        L : L . pv L  (2)
        L : L pv L .  (2)

        pv  shift 6
        eof  reduce 2
```

(a) Donner un exemple d'entrée où le conflit se produit et donner les dérivations dans le cas où on choisit la lecture et dans le cas où on choisit la réduction.

(b) Par défaut, que fait l'analyseur engendré par `ocamlyacc` ?

(c) Comment modifier la grammaire Yacc avec des précédences pour que ce soit l'étape de réduction qui soit choisie.

2 Analyse Lexicale (7 pts)

Dans cet exercice, on se propose d'écrire avec `ocamllex` un analyseur qui transforme un programme `ocaml` en une page HTML présentant le code avec les mots clés en vert, les commentaires en rouge et des numéros à chaque début de ligne.

Hypothèses On suppose données les fonctions `ocaml` suivantes :

fonction	type	commentaire
lexbuf	Lexing.lexbuf	le buffer d'analyse lexicale correspondant au fichier d'entrée
emet	`string` → `unit`	écrit la chaîne de caractères dans le fichier html
debut_html	`unit` → `unit`	écrit l'entête de fichier html en utilisant la balise `<pre>` qui respecte les indentations et sauts de ligne du texte qui suit
fin_html	`unit` → `unit`	écrit la fin de fichier html en fermant la balise `</pre>`
debut_vert	`unit` → `unit`	écrit les commandes html pour passer en vert
debut_rouge	`unit` → `unit`	écrit les commandes html pour passer en rouge
fin_couleur	`unit` → `unit`	écrit les commandes html pour revenir à la couleur de base
numero	`int` → `unit`	écrit les commandes html pour passer à la ligne et imprimer l'entier passé en argument en début de ligne
mot_cles	`string list`	la liste des mots clés de `ocaml`

Rappels des conventions lexicales d'`ocaml`

- Les identificateurs sont composés de caractères alpha-numériques et du caractère de soulignement _ et ne commencent pas par un chiffre.
- Les commentaires commencent par `(*` et se terminent par `*)` et peuvent être imbriqués.

Rappels des conventions lexicales de `HTML` Les caractères < et `&` ont un sens spécial en HTML. Pour qu'ils apparaissent dans le fichier produit il faut les écrire respectivement `<` et `&`.

1. Compléter le programme `ocamllex` suivant pour qu'il réalise les fonctionnalités demandées. On suppose que le programme donné en entrée est syntaxiquement bien formé.

```
{
let is_keyword s = ...
let newline = let r = ref 0 in fun () -> ...
```

```
}

let ident = ...

rule scan = parse
  | ident as s      { ... }
  | '\n'            { ... }
  | "(*"            { ... }
  | ...             { ... }
  | eof             { ... }

and comment = parse
  | "(*"        { ... }
  | "*)"        { ... }
  | ...         { ... }
{
  let () =
    ...
    scan lexbuf;
    ...
}
```

2. Dans `ocaml`, une chaîne de caractères peut contenir la sous-chaîne `(*` ou `*)` sans que ce soit interprété comme un début ou fin de commentaire. Transformer votre analyseur lexical pour prendre en compte correctement les chaînes de caractères. Celles-ci débutent et terminent par le caractère `"` (attention `'"'` représente une valeur `ocaml` de type `char` correspondant au guillemet et pas le début d'une chaîne). Les chaînes peuvent contenir des caractères échappés qui commencent par un \ comme \" ou \\.

3 Analyse sémantique (7 points)

L'objet de cet exercice est d'étudier l'analyse de portée et le typage d'un mini-langage avec modules. Le langage considéré est un mini-langage fonctionnel typé.

– Les types (notés τ) sont soit des types de base (dont `bool`) soit des types fonctionnels $\tau_1 \to \tau_2$ qui est le type des fonctions qui prennent un argument de type τ_1 et renvoient un résultat de type τ_2.
On introduit le type `ocaml` noté `asaType` pour représenter les arbres de syntaxe abstraite des types.

```
type asaType = Base of string | Bool
             | Arrow of (asaType * asaType)
```

– Les expressions (notées e) sont soit des constantes (`cte`), soit des identificateurs (`ident`), soit des fonctions simples **fun** (`ident` : τ) $\to e$, soit des objets récursifs **rec** (`ident` : τ).e, soit des conditionnelles **if** e_1 **then** e_2 **else** e_3, soit des applications $e_1\, e_2$. Les expressions peuvent être parenthésées pour lever les ambiguïtés.

Si l'environnement γ associe un type à chaque identificateur et la fonction δ donne le type de chaque constante, les règles de typage des expressions sont :

$$\frac{c \in \texttt{cte}}{\gamma \vdash c : \delta(c)} \qquad \frac{x \in \texttt{ident} \ \ (x : \tau) \in \gamma}{\gamma \vdash x : \tau}$$

$$\frac{\gamma, x : \tau_1 \vdash e : \tau_2}{\gamma \vdash (\textbf{fun}\ (x : \tau_1) \to e) : \tau_1 \to \tau_2} \qquad \frac{\gamma \vdash e : \tau_1 \to \tau_2 \quad \gamma \vdash e_1 : \tau_1}{\gamma \vdash (e\, e_1) : \tau_2}$$

$$\frac{\gamma, x : \tau \vdash e : \tau}{\gamma \vdash (\textbf{rec}\ (x : \tau).e) : \tau} \qquad \frac{\gamma \vdash e_1 : \texttt{bool} \quad \gamma \vdash e_2 : \tau \quad \gamma \vdash e_3 : \tau}{\gamma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}$$

On suppose donnés des types `ident` et `cte` pour représenter les identificateurs et les constantes, et un type fonctionnel `env` pour représenter les environnements avec les fonctions suivantes :

Nom	Type	Commentaire
`empty`	`env`	environnement vide
`mem`	`ident` → `env` → `bool`	teste si l'identifiant est déclaré dans l'environnement
`find`	`ident` → `env` → `asaType`	renvoie le type associé à l'identifiant dans l'environnement
`add`	`ident` → `asaType` → `env` → `env`	renvoie un nouvel environnement dans lequel l'identifiant est associé au type passé en argument
`type_cte`	`cte` → `asaType`	renvoie le type d'une constante

1. Proposer un type `ocaml` (noté `asaExpr`) pour représenter les arbres de syntaxe abstraite des expressions de ce langage.
2. Écrire une fonction `ocaml` (notée `type_expr`) qui étant donnés un environnement et une expression vérifie que l'expression est bien formée dans l'environnement et calcule son type.
3. On suppose maintenant que dans notre langage, les déclarations sont organisées en modules. Plus précisément une déclaration peut être
   – une déclaration simple de variable : **let** $x\ = e$
   – une déclaration de module : **module** $m\ =$ **struct** *ldecl* **end**, avec m un identificateur et *ldecl* une liste de déclarations (soit des variables introduites par **let** soit d'autres modules). Une déclaration de module a pour effet d'introduire dans l'environnement des noms *qualifiés* (*nom_de_module*.*nom_de_la_variable*) pour les variables du module. Un nom de module peut lui-même être formé d'un simple identificateur ou bien d'une suite de noms de modules séparés par des points dans le cas de modules imbriqués.
   Par exemple si on suppose que le langage comporte des types de base `real` et `real2` pour les réels et les paires de réels avec des fonctions de base `pair:real→real→real2` pour constituer une paire, `fst` et `snd` de type `real2→real` pour récupérer la première et la seconde composante et `plus`, `mult`, `mean`, `norm`, `angle` de type `real→real→real` pour additionner, multiplier, calculer la moyenne arithmétique ($\frac{x+y}{2}$), la moyenne géométrique ($\sqrt{x^2+y^2}$), le rapport angulaire de deux réels, une constante `0` réelle, des fonctions `cos`, `sin` de type `real→real` pour calculer les cosinus, sinus, on peut construire un module pour les complexes :

```
module complexe = struct
  let reel = fun (r:real)→pair r 0
  module cartesien = struct
    let add = fun (c1:real2)→fun (c2:real2)→
            pair (plus (fst c1) (fst c2)) (plus (snd c1) (snd c2))
    let fromPol = fun (p:real2)→
            pair (mult (fst p) (cos (snd p)))
                 (mult (fst p) (sin (snd p)))
  end
  module polaire = struct
```

```
    let add = fun (p1:real2)→fun (p2:real2)→
          pair (norm (fst p1) (fst p2)) (mean (snd p1) (snd p2))
    let fromCart = fun (c:real2)→
          pair (norm (fst c) (snd c)) (angle (fst c) (snd c))
  end
end
```

La déclaration de ce module introduit dans l'environnement les identificateurs suivants :
– `complexe.reel : real→real2`
– `complexe.cartesien.add : real2→real2→real2`
– `complexe.cartesien.fromPol : real2→real2`
– `complexe.polaire.add : real2→real2→real2`
– `complexe.polaire.fromCart : real2→real2`

Tout identificateur déclaré est visible dans la suite des déclarations avec son nom qualifié. On suppose que le type `ident` utilisé dans les expressions correspond maintenant à des noms qualifiés c'est-à-dire qui sont formés d'une liste de noms de modules et d'un nom de variable. On introduit le type `simplid` pour représenter un nom simple (de module ou de variable). Un nom qualifié sera juste une liste (non vide) de noms simples.

```
type ident = simpleid list
```

On décide de la représenter en sens inverse de la notation concrète (par exemple le nom qualifié `complexe.cartesien.add` est représenté par la liste [`add`; `cartesien`; `complexe`]).

(a) Proposer un type `ocaml` pour représenter les arbres de syntaxe abstraite des déclarations.

(b) Écrire une fonction `type_decl` qui étant donnés un environnement et une liste de déclarations vérifie que ces déclarations sont bien typées et renvoie l'environnement complété par les nouvelles déclarations.

Dans cette question, on suppose que tout objet introduit est utilisé dans la suite avec son nom qualifié, c'est-à-dire qu'on écrira :

```
module M = struct
  let x = 3
  let y = M.x + 1
end
```

On pourra penser à passer en argument de `type_decl` une liste de noms de modules correspondant à la suite de modules dans lesquels les déclarations sont faites.

(c) Il est parfois possible de ne pas utiliser le nom qualifié d'une variable. Par exemple à l'intérieur d'un module, une déclaration peut être réutilisée avec son nom court, sans mentionner les modules dans lequel elle est définie. Par exemple dans le module `polaire`, on peut utiliser le nom court `reel` au lieu de `complexe.reel`, le nom court `add` au lieu de `complexe.polaire.add` et le nom (partiellement qualifié) `cartesien.add` au lieu de `complexe.cartesien.add` (car `cartesien` est une abréviation de `complexe.cartesien`)

On ajoute également une déclaration **open** m dans laquelle m est un nom (éventuellement qualifié) de module. L'effet de cette déclaration est que tout nom x (de variable ou de module) déclaré dans m peut être utilisé directement comme une abréviation pour le nom qualifié $m.x$. De fait, m peut lui-même être une abréviation pour un nom de module qualifié m', la variable x sera alors une abréviation pour le nom qualifié $m'.x$.

Pour traiter cette question on introduit une phase d'analyse de portée dans laquelle une suite de déclarations comportant des noms abrégés est transformée en des déclarations comportant des noms qualifiés. Pour cela on utilise une table d'abréviations (de type `abbrev`) qui associe à un identificateur simple visible (nom d'une variable ou d'un module de type

`simpleid`) la suite de modules dans lequel il est déclaré (un objet de type `simpleid list`).

Dans le cas de notre exemple, au moment de la déclaration de `complexe.polaire.fromCart`, on aura dans la table d'abréviations :

$$\texttt{reel} \mapsto [\texttt{complexe}], \texttt{cartesien} \mapsto [\texttt{complexe}], \texttt{add} \mapsto [\texttt{polaire}; \texttt{complexe}]$$

On pourra réutiliser pour manipuler cette table les noms d'opérations `empty`, `mem`, `find`, `add` avec des comportements analogues au cas des environnements.

Questions.

i. Écrire une fonction `transform_id` qui étant donnés un nom de type `ident` (qui peut être partiellement qualifié) et une table d'abréviations renvoie le nom qualifié correspondant.

ii. Écrire une fonction qui étant donnée une liste de déclarations (comportant également des déclarations **open**) utilisant des noms abrégés, reconstruit les déclarations avec des noms entièrement qualifiés. On suppose donnée la fonction `transform_expr` (de type `abbrev` $\rightarrow$`asaExpr` $\rightarrow$`asaExpr`, construite à partir de `transform_id`) qui étant donnée une table d'abréviations, renvoie l'expression correspondante ne comportant que des noms qualifiés. Afin de traiter les déclarations **open**, on pourra également utiliser une table qui à chaque module déclaré associe la liste des identificateurs définis dans ce module.