Compilation
Master Informatique M1 2009-2010

Examen Partiel
28 octobre 2009

1 Analyse Syntaxique (7 points)

Certaines notations facilitent l'écriture de la grammaire des langages de programmation. Soit une grammaire G, `t` un terminal et M un mot formé de terminaux et de non terminaux :

- $\langle M \rangle_{\mathtt{t}}^{+}$ représente le langage des mots de la forme $m_1 \mathtt{t} \ldots \mathtt{t} m_n$ tels que m_i est dérivable à partir de M.
- $\langle M \rangle_{\mathtt{t}}^{*}$ représente le même langage que $\langle M \rangle_{\mathtt{t}}^{+}$ auquel on ajoute le mot vide.

1. Compléter la grammaire G avec deux nouveaux non terminaux: M_0 à partir duquel on pourra dériver le langage $\langle M \rangle_{\mathtt{t}}^{*}$ et M_1 à partir duquel on dérive le langage $\langle M \rangle_{\mathtt{t}}^{+}$.
2. On utilise cette notation pour décrire un langage d'expressions fonctionnelles avec des déclarations locales multiples. Les terminaux de cette grammaire sont formés des mots-clés **fun**, **let**, **in**, **and**, des symboles `(`, `)`, `+`, `=` et $\rightarrow$, et des classes `id` et `cte` pour représenter les identificateurs et les constantes.

   $$\textit{expr} ::= \mathtt{id} \mid \mathtt{cte} \mid \mathtt{(}\textit{expr}\mathtt{)} \mid \textit{expr}\mathtt{+}\textit{expr} \mid \textbf{fun}\ \mathtt{id} \rightarrow \textit{expr} \mid \textit{expr}\ \textit{expr} \mid \textbf{let}\ \langle \mathtt{id}\mathtt{=}\textit{expr} \rangle_{\textbf{and}}^{+}\ \textbf{in}\ \textit{expr}$$

   (a) Ecrire une grammaire `ocamlyacc` pour ce langage.

   (b) Cette grammaire a plusieurs sources d'ambiguité: nommez-en au moins deux.
3. On se donne la grammaire `ocamlyacc` suivante:

```
%token id op pg pd fun arr
%%
E : pg E pd       {}
  | E op E        {}
  | fun id arr E  {}
  | E E           {}
  | id            {}
;
```

   `ocamlyacc` indique des conflits shift/reduce dans trois états. Le fichier d'information contient les éléments suivants sur la grammaire :

```
1  E : pg E pd
2    | E op E
3    | fun id arr E
4    | E E
5    | id
```

   et sur un des états de conflit:

```
14: shift/reduce conflict (shift 3, reduce 3) on id
14: shift/reduce conflict (shift 9, reduce 3) on op
14: shift/reduce conflict (shift 4, reduce 3) on pg
14: shift/reduce conflict (shift 5, reduce 3) on fun
state 14
E : E . op E  (2)
E : fun id arr E .  (3)
E : E . E  (4)
```

(a) Donner un exemple d'entrée pour chacun des conflits de l'état précédent.

(b) Donner pour l'exemple correspondant au premier conflit sur `id` les dérivations dans le cas où on choisit la lecture et dans le cas où on choisit la réduction.

(c) Par défaut, que fait l'analyseur engendré par `ocamlyacc` ? Dans un conflit entre réduction et lecture, `ocamlyacc` choisit la lecture.

(d) On souhaite que l'application $E\,E$ ait une précédence plus forte que les opérations binaires $E\,\mathtt{op}\,E$ qui auront elles-mêmes une précédence plus forte que la construction de fonction **fun** *id* $\rightarrow$ E. On demande également que les opérations binaires et l'application associent à gauche. Ainsi le programme **fun** $f \rightarrow f\,1\,3 + 2$ sera compris comme **fun** $f \rightarrow (((f\,1)\,3) + 2)$.
Comment modifier la grammaire `ocamlyacc` avec des précédences afin de supprimer les conflits en respectant les règles de précédence proposées.

2 Analyse Lexicale (7 pts)

Dans cet exercice, on se propose de traiter l'analyse syntaxique d'un programme en langage assembleur à l'aide de `ocamllex`. La syntaxe est inspirée de celle de TIGCC, un assembleur pour les calculatrices TI89. Les principaux éléments de ce langage sont:

Commentaire commence par `/*` et se termine à la première occurrence de `*/` ou bien commence par une barre (`|`) et se termine au premier retour à la ligne. Un commentaire est considéré comme un espace.

Symbole suite de caractères formés des lettres de l'alphabet (majuscule ou minuscule) des chiffres et des trois caractères spéciaux `_`, `.` ou `$` et ne commençant pas par un chiffre.

Constante entière les constantes entières peuvent être données en plusieurs formats:
- binaire : $0b$ ou $0B$ suivi d'une suite de 0 ou de 1
- hexadecimal : $0x$ ou $0X$ suivi d'une suite de nombres hexadecimaux ($0-9\ A-F\ a-f$).
- decimal : suite de chiffres ne commençant pas par 0.

Déclaration formée de zéro ou plusieurs labels suivis d'une instruction et terminé par un retour à la ligne ou bien un point-virgule (;). Un label est un symbole immédiatement suivi de deux-points (:) sans espace.

Instruction – Le langage contient une instruction "vide".
- Une instruction non vide est donnée par son nom (formé uniquement de lettres) suivi de zéro ou plus arguments séparés par des virgules (`,`).
- Un argument d'instruction est soit un registre (de la forme `%`*nr* avec *nr* le nom de registre qui est une suite de lettres), soit le contenu de l'adresse stockée dans un registre (noté `%`*nr*`@`) soit un symbole, soit une constante entière.

1. Proposer un type d'arbre de syntaxe abstraite pour représenter les instructions de ce langage. Les constantes entières seront représentées par des valeurs Ocaml de type `int`.
2. Ecrire un programme `ocamllex` qui analyse un programme en assembleur et renvoie la séquence d'instructions. On pourra utiliser des variables globales pour stocker les labels, nom d'instructions et arguments au fur et à mesure de la reconnaissance d'une déclaration.
3. Modifier le programme précédent (on n'indiquera que les lignes qui changent) pour stocker dans une table les labels en leur associant le numéro d'instruction correspondant. Si le même nom est utilisé à plusieurs reprises, alors on retient la dernière occurrence.
4. Le langage est étendu pour inclure des labels locaux. Ceux-ci sont donnés par une constante entière *N* suivie du symbole deux-points (`:`). On peut utiliser comme argument d'une instruction assembleur les entrées *N*`b` ou bien *N*`f` qui font référence au label *N* introduit juste avant (backward) l'instruction dans le cas de *N*`b` ou juste après (forward) dans le cas de *N*`f`. Ces labels sont transformés immédiatement en des labels ordinaires avec un nom `L`$N\backslash 002i$ avec $\backslash 002$ un caractère spécial pour éviter que ce nom

coincide avec un nom de l'utilisateur et i un compteur qui repère le nombre d'utilisations du label local N avant cette instruction.
Expliquer comment étendre votre analyseur pour prendre en compte cette extension (on ne demande pas d'écrire le code mais de décrire précisément et clairement la méthode).

3 Analyse sémantique (7 points)

On se propose dans cet exercice de traiter le typage d'un mini-langage objet. Un programme de ce langage est composé d'une suite de déclarations de classes.

Grammaire La grammaire du langage a pour symboles terminaux :

`Cid`	identificateurs de classe
`id`	identificateurs de variable ou méthode
`num`	constantes entières
`op`	opérateurs binaires $\{+, -, *, /, <, \leq, ==\}$
class extends true false this new return if else	mots clés
`= . , ; ( ) { }`	symboles

Les symboles non-terminaux sont : $\{C, V, M, L, E, I\}$.
- C représente une suite de déclarations de classes ;
- V une déclaration de variables ;
- M une déclaration de méthode ;
- L une valeur gauche (une variable visible dans la classe, le paramètre d'une méthode ou bien une variable d'un objet résultat de l'évaluation d'une expression E) ;
- E une expression (un entier, l'objet lui-même noté **this**, une opération binaire appliquée à deux expressions, un nouvel objet, une variable ou le champ d'un objet, le résultat de l'application d'une méthode ou une affectation) ;
- I une instruction (expression, conditionnelle, séquence d'instructions).

Les règles sont les suivantes (on utilise les notations rappelées dans l'exercice 1) :

C ≔ ⟨`class Cid extends Cid` ⟨V⟩* ⟨M⟩*⟩+
V ≔ `Cid` ⟨`id`⟩+`,` `;`
M ≔ `Cid id(` ⟨`Cid id`⟩*`,` `)` I **return** E
L ≔ `id` | `E.id`
E ≔ **true** | **false** | **this** | `num` | **new** `Cid` | L | E `op` E | `E.id(` ⟨E⟩*`,` `)` | L = E | `(`E`)`
I ≔ E `;` | **if** `(`E`)` I **else** I | **if** `(`E`)` I | `{` ⟨I⟩+ `}`

Classes prédéfinies On suppose prédéfinis trois identificateurs de classe : `Object` (classe universelle de tous les objets), `Int` la classe des entiers et `Bool` la classe des booléens qui étendent la classe `Object`. On suppose qu'il n'y a pas de variables ni de méthodes définies dans ces classes.

Règles de portées Les variables d'une classe *Cid* sont composées des variables de la classe qu'elle étend, plus celles qu'elle déclare en propre. Si une variable de même nom est redéclarée dans une sous-classe, elle cache la définition dans la classe ancêtre. Les variables visibles dans une méthode de la classe *Cid* sont toutes les variables de la classe *Cid* ainsi que les paramètres de la méthode. Dans le corps d'une méthode, le mot clé **this** désigne l'objet lui-même auquel s'applique la méthode.

1. On donne les types OCaml suivants :

```
type ident = string
type class_ident = string
type op = Plus | Minus | Mult | Div | Lt | Le | Eq
type lexpr = | Id of ident | Dot of expr * ident
and expr = Bool of bool | Int of int | This | L of lexpr
         | Bin of expr * op * expr | ...
```

Compléter le type `expr` des arbres de syntaxe abstraite pour représenter les expressions. Proposer des types pour représenter les instructions (`instr`) et les classes (`class`).

2. On suppose que l'on dispose des fonctions suivantes:
   – `super: class_ident -> class_ident`,
   `super` C est l'identifiant de la classe que C étend;
   – `vars: class_ident -> ident list`,
   `vars` C est la liste des variables définies dans C;
   – `methods: class_ident -> ident list`,
   `methods` C est la liste des méthodes définies dans C.
   On suppose qu'il n'y a pas de circularité dans les définitions de classe (toute classe a `Object` comme ancêtre).
   (a) Définir une fonction `visible` qui, étant données une classe C et un identificateur x (variable ou méthode), vérifie que x est bien visible à partir de C et renvoie la classe dans laquelle la variable ou la méthode x est définie.
   (b) Définir une fonction `subclass` qui, étant données deux classes C et D, renvoie `true` si C est une sous-classe de D (c'est-à-dire si $C = D$ ou bien si C n'est pas `Object` et le père de C est une sous-classe de D).
3. Les règles de typage de ce langage sont classiques pour un langage objet. Les types sont les identificateurs de classe. On parle indifféremment du type ou de la classe d'une expression E.
   – dans la classe C, une déclaration de méthode m de la forme $D\ m(C_1\ x_1, \ldots, C_n\ x_n)\ I$ **return** E est bien formée si IE; est bien formé dans l'environnement de la classe C étendu avec **this** de type C et x_i de type C_i et si le type de l'expression E est une sous-classe de D.
   – Une application de méthode $E.m(e_1, \ldots, e_n)$ est bien formée si l'expression E est bien formée de type D, si la méthode m est visible à partir de la classe D avec une signature $([C_1; \ldots; C_n], C')$ et si chaque expression e_i a pour type une sous-classe de C_i. Le type de l'application sera alors C'.
   – Une affectation $L = E$ est bien formée si L est bien formé et de type D, si l'expression E est bien formée de type D' et si D' est une sous-classe de D. Le type de l'affectation est le type de E.
   – Le type de **new** C est C.
   – Le type des opérations binaires est classique, l'égalité $E_1 == E_2$ est bien formée de type `Bool` dès que E_1 et E_2 sont bien formés.
   (a) Proposer une organisation pour la table des symboles permettant de conserver les informations de type pour les variables et les signatures des méthodes.
   (b) Ecrire une fonction de typage pour les expressions qui suppose la table des symboles correctement complétée et prend comme argument le nom de la classe courante et un environnement qui associe aux noms de variables locales leur type.
   (c) Ecrire une fonction qui vérifie qu'un programme est bien typé en remplissant au fur et à mesure la table des symboles.