Compilation
Master Informatique M1 2009-2010

Examen Session 2
3 juin 2010

L'énoncé est composé de 6 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 Les questions sont largement indépendantes, il est toujours possible de supposer résolue une question pour traiter les questions suivantes. Lorsqu'il est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ces 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.

Contexte

On se propose dans cet exercice de traiter la compilation 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 $\{+, -, *, /, <, \leq, ==\}$
int bool class extends true false this new return if else	mots clés
`= == . , ; ( ) { }`	symboles

Les symboles non-terminaux sont : $\{C, T, V, M, L, E, I\}$.

- C représente une déclaration de classe ;
- T un type (soit une classe, soit un type de base **int** ou **bool**) ;
- 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 rappelle que $\langle M \rangle^+_{\mathtt{t}}$ 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 et $\langle M \rangle^*_{\mathtt{t}}$ représente le même langage que $\langle M \rangle^+_{\mathtt{t}}$ auquel on ajoute le mot vide) :

C	≔	`class Cid extends Cid {` $\langle V \rangle^*$ $\langle M \rangle^*$ `}`
T	≔	`Cid` | **int** | **bool**
V	≔	T $\langle$ `id` $\rangle^+_{,}$ `;`
M	≔	T `id(` $\langle T$ `id` $\rangle^*_{,}$ `)` $\langle V \rangle^*$ I **return** E
L	≔	`id` | `E.id`
E	≔	**true** | **false** | **this** | `num` | **new** `Cid` | L | E `op` E | `E.id(` $\langle E \rangle^*_{,}$ `)` | L = E | `(`E`)`
I	≔	E `;` | **if** `(`E`)` I **else** I | **if** `(`E`)` I | `{` $\langle I \rangle^*$ `}`

Exemple On donne ci-dessous un exemple de programme dans ce langage:

```
class Point extends Object {
int x,y;
Point move (Point v)
    Point z;
    { z = new Point; z.x = this.x + v.x; z.y = this.y + v.y; }
    return z
bool  eq (Point v)
    bool res;
    if (this.x == v.x) res = (this.y == v.y); else res = false;
    return res
}
class PointScreen extends Point {
int width,height;
Point move (Point v)
    Point z;
    { z = new Point; z.x = this.x + v.x; z.y = this.y + v.y;
      if (z.x < 0) z.x = 0;
      if (this.width < z.x) z.x = this.width;
      if (z.y < 0) z.y = 0;
      if (this.height < z.y) z.y = this.height;
    }
    return z
}
```

1 Analyse syntaxique (6 points)

Un sous-ensemble de la grammaire de ce langage exprimé en ocamlyacc est le suivant. Les tokens `pg`, `pd`, `ag`, `ad`, `eq`, `pv` et `pt` correspondent respectivement aux symboles: `(`, `)`, `{`, `}`, `=`, `;` et `.` du langage.

```
%token if else id pg pd op ag ad eq true false pv pt
%left op
%start I
%type <unit> I
%%
L : id          {}
  | E pt id     {}
;
E : L           {}
  | L eq E      {}
```

```
  | true       {}
  | false      {}
  | E op E     {}
;
I : E pv       {}
  | ag IS ad   {}
  | if pg E pd I          {}
  | if pg E pd I else I {}
;
IS :           {}
  | IS I       {}
;
```

La compilation ocamlyacc indique 4 conflits shift/reduce correspondant aux 3 états suivants.

```
20: shift/reduce conflict (shift 14, reduce 4) on op
20: shift/reduce conflict (shift 16, reduce 4) on pt
state 20
        L : E . pt id  (2)
        E : L eq E .  (4)
        E : E . op E  (7)
        op  shift 14
        pt  shift 16
        pd  reduce 4
        pv  reduce 4

21: shift/reduce conflict (shift 16, reduce 7) on pt
state 21
        L : E . pt id  (2)
        E : E . op E  (7)
        E : E op E .  (7)
        pt  shift 16
        pd  reduce 7
        op  reduce 7
        pv  reduce 7

24: shift/reduce conflict (shift 25, reduce 10) on else
state 24
        I : if pg E pd I .  (10)
        I : if pg E pd I . else I  (11)
        else  shift 25
        $end  reduce 10
        if  reduce 10
        id  reduce 10
        ag  reduce 10
        ad  reduce 10
        true  reduce 10
        false  reduce 10
```

1. Indiquer pour chaque état et chaque conflit à quelle situation concrète cela correspond (donner un exemple d'entrée sur laquelle le conflit se produit) et dire dans chaque cas s'il faut choisir une réduction ou une lecture.
2. Compléter la grammaire par des précédences sur les tokens permettant de résoudre les conflits et

d'obtenir le comportement attendu.

2 Analyse sémantique (8 points)

Classe prédéfinie On suppose prédéfini un identificateur de classe : `Object` (classe universelle de tous les objets) qui ne contient ni variables ni méthodes. On suppose que lorsqu'une classe D est définie qui étend la classe C alors la classe C est soit `Object` soit une classe prálablement définie. Ainsi il n'y a pas de circularité dans les définitions de classe et toute classe a `Object` comme ancêtre.

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. Une variable de même nom ne peut est redéclarée dans une sous-classe. 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 et les variables locales. Dans le corps d'une méthode, le mot clé **this** désigne l'objet lui-même auquel s'applique la méthode.

Types Les types de ce langage sont soit les *types de base* (`int` et `bool`, soit les identificateurs de classe. Quand le type d'une expression est une classe on dira que l'expression représente un *objet*. La notion de sous-classe s'étend en une notion de *sous-typage*. Chaque type est un sous-type de lui-même et une classe C est un sous-type d'une classe D si et seulement si C est une sous-classe de D.

1. On donne les types OCaml suivants pour représenter les arbres de syntaxe abstraite correspondant aux expressions :

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

   Compléter les types suivants pour représenter les arbres de syntaxe abstraite correspondant aux instructions, aux types, aux déclarations de méthodes et de classes.

```
type instr = | Expr of expr | If of ... | Block of ...
type typ = | Int | Bool | ...
type var = typ * ident
type meth = ...
type cl = ...
type prog = cl list
```

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;
   - `visible: class_ident -> ident -> class_ident`,
     `visible` $C\,x$ vérifie que la variable ou la méthode x est bien visible à partir de C et renvoie la classe dans laquelle x est définie, échoue sinon.

– `subclass: class_ident -> class_ident -> bool`,
`subclass` C D, renvoie `true` si C est une sous-classe de D.

(a) Définir une fonction `subtyp` de type `typ -> typ -> bool` qui teste si un type est un sous-type d'un autre.

(b) A chaque méthode est associée une signature qui décrit la liste des types des arguments de la méthode ainsi que le type de la valeur de retour. Si une méthode m a été définie dans une classe C alors on autorise la redéfinition de m dans une sous-classe D de C seulement si la signature est identique.

i. Définir une fonction `check_method` qui étant donné une déclaration de méthode m vérifie que m n'a pas déjà été définie dans une sur-classe avec une signature différente et enregistre sa signature dans une table sinon.

ii. Définir une fonction `check` qui étant donné un programme, vérifie qu'il satisfait la condition de non redéfinition.

3. Les règles de typage de ce langage sont classiques pour un langage objet.
   – dans la classe C, une déclaration de méthode m de la forme $T\ m(T_1\ x_1, \ldots, T_n\ x_n)\ I$ **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 T_i et si le type de l'expression E est un sous-type de T.
   – Une application de méthode $e.m(e_1, \ldots, e_n)$ est bien formée si l'expression e est bien formée de type une classe D, si la méthode m est visible à partir de la classe D avec une signature $([T_1; \ldots; T_n], T)$ et si chaque expression e_i a pour type un sous-type de T_i. Le type de l'application sera alors T.
   – Une affectation $L = E$ est bien formée si L est bien formé et de type T, si l'expression E est bien formée de type T' et si T' est un sous-type de T. Le type de l'affectation est le type de E.
   – Le type de **new** C est la classe 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 et sont soit dans le même type de base, soit deux objets.

Question. Préciser l'organisation de la table des symboles. 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.

3 Génération de code (8 points)

La compilation du langage se fait suivant les principes suivant:
– Les valeurs dans les types **int** et **bool** sont stockées directement sur la pile; les valeurs correspondant à des objets sont représentées sur la pile par une adresse correspondant à un bloc alloué dans le tas. Le premier champs de ce bloc est un pointeur sur le descripteur de classe, les autres éléments du bloc correspondent aux valeurs des variables visibles de la classe.
– Les descripteurs de classe sont stockés à des adresses globales sur la pile. Un descripteur d'une classe C est formé de plusieurs cases sur la pile qui contiennent un pointeur sur le descripteur de classe de l'ancètre de C ainsi que pour chaque méthode visible dans la classe C, l'adresse du code compilé de la méthode.
– Les méthodes sont interprétées comme des fonctions qui attendent en plus des arguments déclarées, un argument qui correspond à l'objet **this**.

1. Quelles informations sur les classes faut-il stocker pour pouvoir compiler l'instruction **new** C ? Donner le code compilé correspondant.

2. Indiquer quel est le tableau d'activation de la fonction correspondant à la méthode `move` de la classe `Point` de l'exemple.
3. Donner le code de la machine à pile correspondant au corps de cette fonction. On choisit de mettre le descripteur de la classe `Object` à l'adresse 0 de la pile et le descripteur de la classe `Point` à partir de l'adresse 1.
4. On complète le programme de l'exemple par la définition de classe suivante:

```
class Main extends Object {
bool main (bool b)
      Point p; PointScreen q;
      { p = new Point; p.x=400; p.y=300;
        q = new PointScreen; q.width=600; q.height=440; q.x=p.x; q.y=p.y;
        if (b) p=q;
        p.move(p);}
      return b
}
```

   (a) Indiquer l'organisation dans la pile des descripteurs de classes `Point`, `PointScreen`, `Main`
   (b) Donner le tableau d'activation de la fonction `main`.
   (c) On suppose que le code pour la méthode `move` de la classe `PointScreen` est à l'adresse `PointScreen_move`. Donner le code des instructions suivantes de `main`.

```
if (b) p=q; p.move(p);
```

4 Instructions de la machine à pile

`PUSHI` n	empile la constante entière n.
`ADD`/`SUB`/`MUL`/`DIV`	dépile y puis x et empile $(x+y)$/$(x-y)$/$(x*y)$/(x/y)
`INF`/`INFEQ`/`EQ`	dépile y puis x et empile 1 si $(x<y)$/$(x\leq y)$/$(x=y)$ et 0 sinon
`NOT` n	dépile n qui doit être un entier et empile le résultat de $n=0$
`PUSHG` n	empile la valeur située n cases au dessus de *gp*.
`STOREG` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *gp*.
`PUSHL` n	empile la valeur située n cases au dessus de *fp*.
`STOREL` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *fp*.
`PUSHA` l	empile l'adresse correspondant au label l.
`LOAD` n	dépile une adresse a et empile la valeur située à l'adresse $a+n$
`STORE` n	dépile une valeur v et une adresse a et stocke la valeur v à l'adresse $a+n$.
`PUSHN` n	empile n valeurs nulles sur la pile.
`SWAP`	échange les contenus des deux éléments consécutifs en sommet de pile.
`DUP`	duplique et empile la valeur qui est au sommet de la pile.
`POP` n	dépile n valeurs de la pile.
`JZ` l	dépile x, si $x=0$ le pointeur de code saute à l'instruction de label l.
`JUMP` l	saute à l'instruction de label l.
`CALL`	sauve la valeur courante de *fp* et le pointeur d'instructions, affecte à *fp* la valeur de la première case libre de la pile, dépile une valeur c correspondant à une adresse et saute à l'instruction correspondante.
`RETURN`	restaure les valeurs de *fp*,*sp* et *pc*.
`ALLOC` n	alloue sur le tas un bloc structuré de taille n et empile l'adresse correspondante.