Compilation
Master Informatique M1 2009-2010

Examen Final
16 décembre 2009

Les exercices sont indépendants. L'énoncé est composé de 5 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. Lorsqu'il vous est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ses détails (vous pouvez par exemple 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.

1 Question de cours (3 points)

Les langages ont des politiques différentes quant à l'utilisation des variables dans les fonctions:
- en Java ou C, une fonction ne peut utiliser que des variables locales (déclarées dans le corps de la fonction) ou des variables globales;
- en Pascal, les fonctions peuvent être imbriquées et le corps d'une fonction f peut utiliser les variables d'une fonction g dans laquelle f est définie;
- en CAML, une fonction f peut avoir pour valeur de retour une autre fonction g dont le corps utilise les paramètres ou les variables locales de f.

Dans chacun de ces trois cas, expliquer (brièvement) comment on peut organiser les variables utilisées dans le corps de la fonction dans la mémoire de la machine à pile.

2 Fonctions à la Pascal (17 pts)

On étudie un langage (appelé P) analogue à celui traité dans le projet, sans module, mais avec des définitions de fonctions en plus des définitions de procédure. Les questions sont largement indépendantes et couvrent l'ensemble du programme.
La grammaire est la suivante:

⟨*fichier*⟩ ::= `program` ⟨*ident*⟩ **;** ⟨*définition*⟩* ⟨*bloc*⟩ **.** `EOF`
⟨*définition*⟩ ::= `var` ⟨*decl_var*⟩$^+$ | `procedure` ⟨*ident*⟩ ⟨*params*⟩? **;** ⟨*vars*⟩? ⟨*bloc*⟩ **;**
| `function` ⟨*ident*⟩ ⟨*params*⟩? **:** ⟨*type*⟩ **;** ⟨*vars*⟩? ⟨*bloc*⟩ **;**
⟨*decl_var*⟩ ::= ⟨*ident*⟩$^+_{,}$ **:** ⟨*type*⟩ **;**
⟨*params*⟩ ::= **(** ⟨*param*⟩$^+_{;}$ **)**
⟨*param*⟩ ::= `var`? ⟨*ident*⟩$^+_{,}$ **:** ⟨*type*⟩
⟨*type*⟩ ::= `boolean` | `integer` | `array` **[** ⟨*entier*⟩ **..** ⟨*entier*⟩ **]** `of` ⟨*type*⟩
⟨*expr*⟩ ::= `true` | `false` | ⟨*entier*⟩ | ⟨*lvalue*⟩ | **(** ⟨*expr*⟩ **)** | ⟨*ident*⟩ **(** ⟨*expr*⟩$^\star_{,}$ **)**
| `not` ⟨*expr*⟩ | **-** ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*opérateur*⟩ ⟨*expr*⟩
⟨*lvalue*⟩ ::= ⟨*ident*⟩ | ⟨*lvalue*⟩ **[** ⟨*expr*⟩ **]**
⟨*opérateur*⟩ ::= **=** | **<>** | **<** | **<=** | **>** | **>=** | **+** | **-** | ***** | `div` | `mod` | `and` | `or`
⟨*instruction*⟩ ::= ⟨*lvalue*⟩ **:=** ⟨*expr*⟩ | `if` ⟨*expr*⟩ `then` ⟨*instruction*⟩
| `if` ⟨*expr*⟩ `then` ⟨*instruction*⟩ `else` ⟨*instruction*⟩
| `while` ⟨*expr*⟩ `do` ⟨*instruction*⟩
| `for` ⟨*ident*⟩ **:=** ⟨*expr*⟩ `to` ⟨*expr*⟩ `do` ⟨*instruction*⟩
| ⟨*bloc*⟩ | ⟨*ident*⟩ **(** ⟨*expr*⟩$^+_{,}$ **)** | ⟨*ident*⟩
⟨*bloc*⟩ ::= `begin` ⟨*instruction*⟩$^\star_{;}$ **(;** ?**)** `end`

1. Il est possible de déclarer une procédure p ou une fonction f sans paramètre. Il y a par contre dans notre grammaire une différence entre l'appel de procédure qui s'écrira simplement p; alors que l'appel de fonction utilise des parenthèses $f()$. Expliquer pourquoi cette convention simplifie l'**analyse syntaxique**. Que faudrait-il faire si on voulait ne pas avoir à mettre de parenthèse?

2. On se donne les types suivants pour représenter les **arbres de syntaxe abstraite** de ce langage:

```
type typ = Tboolean  | Tinteger  | Tarray of int * int * typ
type ident = string
type mode = Value | Reference
type constant = Cboolean of bool | Cinteger of int
type unop = Unot | Uminus
type binop = Beq | Blt | Ble | Badd | Bsub | Bmul | Bdiv | Bmod
           | Band | Bor
type array_info = int (* premier indice *) * typ (* type des éléments *)
type expr = Econst of constant | Elvalue of lvalue
 | Eunop of unop * expr | Ebinop of binop * expr * expr
 | Efun_call of ... (* appel de fonction *)
and lvalue = Lvar of ident | Larrget of lvalue * expr * array_info
type statement = Sskip | Sassign of lvalue * expr
 | Sprocedure_call of ident * (mode * expr) list
 | Sif of expr * statement * statement
 | Swhile of expr * statement
 | Sfor of ident * expr * expr * statement
 | Sbloc of statement list
type body = {b_vars:ident list;b_stmt:statement}
type var = {v_name:string;v_type:typ;v_local:bool;v_mode:mode;}
type definition = DFvar of var
 | DFprocedure of ident * var list * body
 | DFfunction of ... (* declaration de fonction *)
type file = {name:string; defs:definition list; main:statement;}
```

   Compléter les définitions des types `expr` et `definition` avec les types des cas `Efun_call` pour l'appel de fonction et `DFfunction` pour les définitions de fonction.

3. On suppose que l'analyse de portée été réalisée et que chaque identificateur apparaissant dans le programme est unique et que les informations correspondant à ces identifiants est stockée dans une table des symboles. On se propose de réaliser l'**analyse sémantique**. On suppose définies les fonctions suivantes:
   - `get_var : ident →var`. L'expression `get_var` x fournit les informations sur la variable x;
   - `get_proc : ident →var list` . L'expression `get_proc` p fournit la liste des variables paramètres de la procédure p;
   - `get_fun:ident →var list * typ`. L'expression `get_var` f fournit la liste des variables paramètres de la procédure f ainsi que le type du retour de la function.

   Une particularité du langage Pascal est la manière d'indiquer le calcul de la valeur de retour de la fonction. Dans le corps de la définition d'une fonction f, la valeur de retour sera donnée par l'ajout d'une ou plusieurs instructions $f := e$. Il faudra donc s'assurer que le type de e correspond au type de retour attendu pour la fonction, pour cela il suffit de rendre visible dans le corps de la fonction f une variable de même nom avec le type attendu.

   (a) Ecrire une fonction `type_expr` qui vérifie qu'une expression est bien formée et calcule son type.

   (b) Ecrire une fonction `check_statement` qui vérifie qu'une instruction est bien formée.

(c) Ecrire une fonction `check_return` qui vérifie que dans le corps d'une fonction f, tous les chemins d'exécution contiennent au moins une instruction de retour f`:=`e.

4. On s'intéresse maintenant à la **génération de code**. On rappelle les principales conventions du langage.
   - Toutes les opérandes ou arguments de fonctions sont évalués. La seule exception est l'évaluation *paresseuse* des expressions e_1 `and` e_2 et e_1 `or` e_2 : pour l'opérateur `and` (resp. `or`), e_2 n'est évaluée que si e_1 est vrai (resp. faux).
   - Passage des arguments : quel que soit leur type, les arguments sont passés par référence lorsque le mot clé `var` est présent, par valeur sinon.
   - La valeur retournée par une fonction est la *dernière* valeur affectée à la variable portant le nom de la fonction lorsque le flot de contrôle atteint la fin de la fonction.
   - Dans une boucle `for` x `:=` e_1 `to` e_2 `do` i, les expressions e_1 et e_2 ne sont évaluées qu'une seule fois. Aucune évaluation de i n'est faite si la valeur de e_2 est strictement inférieure à celle de e_1.

   (a) On se donne le programme suivant:

```
program test;
var a : array[1..5] of integer;
var k : integer;
function exp (x,n: integer) : integer;
var i,y,z : integer;
begin
  i:=n;y:=1;z:=x;
  while i > 0 do
  begin
     if i mod 2 = 1 then y := y * z;
     z := z * z; i := i div 2;
  end;
  exp:=y
end;
function aexp (m : array[1..5] of integer; n:integer)
 : array[1..5] of integer;
var j   : integer;
var res : array[1..5] of integer;
begin
  for j := 1 to 5 do res[j]:=exp(m[j],n);
  aexp:=res
end;
begin
  for k:=1 to 5 do a[k]:=k;
  a:= aexp(a,3);
end.
```

   - Expliquer comment les variables de ce programme seront stockées dans la machine à pile (on donnera les tableaux d'activation de chaque fonction).
   - Donner le code de la machine à pile correspondant à ce programme particulier.

   (b) Ecrire la fonction `gen_code` de génération de code d'une expression dans les cas suivants:
   - conjonction de deux expressions booléennes e_1 **and** e_2 (cas `Ebinop(Band,`e_1, e_2`)`);
   - appel de fonction $f(e_1, \ldots, e_n)$ (cas `Efun_call(`f, l`)`).

(c) Expliquer comment traiter de manière générale la génération de code pour calculer la valeur de retour des fonctions.

5. On considère maintenant un sous-ensemble du langage sans fonctions ni procédures. On veut transformer les programmes vers un **langage intermédiaire** (code à trois adresses) dont les instructions sont de la forme suivante avec a qui est soit un identificateur soit une constante entière.

`id = a`	`id[a] = a`	**goto** L	**label** L
`id = a op a`	`id = id[a]`	**ifzero** `id` **goto** L	

Le type CAML suivant permet de représenter les arbres de syntaxe abstraite de ce langage intermédiaire S.

```
type reg = int
type atom = Var of reg | Int of int
type ssa_expr = At of atom | Bop of binop * atom * atom
    | GetArray of reg * atom (* acces dans un tableau *)
type ssa_instr = Aff of reg * ssa_expr
   | SetArray of reg * atom * atom (* mise à jour tableau *)
   | Lab of string | Goto of string | Ifzero of ident * string
```

(a) Ecrire une fonction `transform_expr` de type `reg -> expr -> ssa_instr list` qui étant donnés un numéro de registre r et une expression e (sans appel de fonction) renvoie une liste d'instructions du langage S dont l'effet sera de calculer la valeur de e dans le registre r. On supposera que chaque variable x du programme est associée à un registre r_x et que l'on dispose d'une function `new_reg` qui donne le nom d'un nouveau registre non utilisé.

(b) Soit le programme du langage P:

```
program test;
var a : array[1..5] of integer;
var k,i,y,z : integer;
begin
  for k:=1 to 5 do
  begin
  i:=n;y:=1;z:=k;
  while i > 0 do
  begin
     if i mod 2 = 1 then y := y * z;
     z := z * z; i := i div 2;
  end;
  a[k]:=y
  end;
end.
```

Donner le code de S obtenu par transformation de ce programme.

(c) Ecrire une fonction `transform_statement` qui donne le code dans le langage S équivalent à une instruction du langage P (on ne considère pas le cas de l'appel de procédure).

6. On souhaite réaliser l'**allocation de registres**. On se donne le programme suivant du langage S introduit à la question 5 :

```
a = 40
b = 12
t = b <> a
ifzero t goto E1
u = a
v = b
label L
r = u mod v
ifzero r goto E2
t = v
v = r
u = t
goto L
label E1
v = b
label E2
o = v
```

   (a) Construire le graphe de flot de contrôle de ce programme.

   (b) Indiquer les variables vivantes en chaque point de programme.

   (c) Construire le graphe d'interférence.

   (d) Proposer un coloriage de ce graphe avec 3 couleurs.

   (e) Peut-on faire mieux sans changer le programme ?

3 Instructions de la machine à pile

`PUSHI` n	empile la constante entière n.
`ADD`/`SUB`/`MUL`/`DIV`/`MOD`	dépile y puis x et empile $(x+y)$/$(x-y)$/$(x*y)$/(x/y)/x **mod** 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*.
`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, saute à l'instruction d'adresse c.
`RETURN`	restaure les valeurs de *fp*,*sp* et *pc*.