Compilation
Master Informatique M1 2007-2008

Examen Final

18 décembre 2007

Les deux exercices sont indépendants. L'énoncé est composé de 6 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.

On attachera une grande importance à la clarté et la précision. Les programmes seront écrits en Objective Caml de manière concise et en choisissant convenablement les noms des identificateurs. **Les programmes mal rédigés ne seront pas corrigés.**

1 Analyse Syntaxique (7 pts)

1. Soit la grammaire G_1 définie sur l'ensemble des terminaux $\{$`*`, `id`, `(`, `)`$\}$.

E' ::= E♯	S ::= (E)
E ::= E `id`	S ::= `id`
E ::= (E ∗ E)	
E ::= S	

2. Cette grammaire est-elle LR(0), SLR(1)? Vous justifierez votre réponse.
3. Cette grammaire est-elle LL(1) ? Le justifier.
4. Construire une grammaire G'_1 LL(1) équivalente à G en factorisant les préfixes communs de G_1 et en éliminant les récursions gauches.
5. Calculer pour la grammaire G'_1 les ensembles $\textsc{prem}(X)$ et $\textsc{suiv}(X)$ pour chaque non-terminal.
6. On considère la grammaire G_2 issue de G_1.

E' ::= E♯	S ::= (E)
E ::= E `id`	S ::= `id`
E ::= E ∗ E	
E ::= S	

On supposera que la construction de produit `*` est associative à gauche et que la notation E `id` est plus prioritaire que la construction de produit. Ecrire une grammaire Yacc en utilisant les précédences et permettant de reconnaître le langage reconnu par G_2. On ne cherchera pas ici à construire un arbre de syntaxe abstraite et les règles produiront une valeur de type `unit`.

2 Traitement des exceptions (13 points)

On se donne un mini-langage de programmation impératif appelé TRAP étendu avec un mécanisme d'exceptions. On supposera que les analyses lexicales et syntaxiques sont définies pour ce langage. L'objectif du problème est de définir la phase d'analyse sémantique (typage et vérifications statiques) et de génération de code pour la machine à pile vue en cours.

La syntaxe concrète du langage est donnée ci-dessous.

$$
\begin{array}{lcl}
exp & ::= & exp\ op\ exp \mid i \mid x \\[1ex]
op & ::= & + \mid - \mid * \mid \ \mid > \mid = \\[1ex]
inst & ::= & inst; inst \mid x := exp \mid \texttt{if}\ exp\ \texttt{then}\ inst\ \texttt{else}\ inst \\
 & & \mid \texttt{while}\ exp\ \texttt{do}\ inst\ \texttt{done} \mid \texttt{try}\ inst\ \texttt{with}\ handlers \\
 & & \mid \texttt{exit}\ T \mid \texttt{skip} \\
 & & \mid \texttt{read}(x) \mid \texttt{write}(exp) \\
 & & \mid \texttt{message}(chaine) \mid \texttt{begin}\ inst\ \texttt{end} \\[1ex]
handlers & ::= & T\ \texttt{do}\ inst \mid T\ \texttt{do}\ inst\ handlers \\[1ex]
prog & ::= & \texttt{exception}\ T_1; ...; \texttt{exception}\ T_n\ bloc \\[1ex]
bloc & ::= & \texttt{var}\ x_1, ..., x_n : typ\ \texttt{in}\ inst \\
ty & ::= & \texttt{int}
\end{array}
$$

exp designe la classe syntaxique des expressions. $op(e_1, e_2)$ désigne l'application d'un opérateur binaire à ses arguments; i désigne une constante entière et x désigne une variable. *inst* est la classe des instructions. Une instruction peut être une séquence, une affectation, une conditionnelle, une boucle, un traitement de levée d'exception (`try` *inst* `with` T_1 `do` $inst_1 ... T_n$ `do` $inst_n$), une instruction de levée d'exception (`exit` T), une instruction qui ne fait rien (`skip`), une instruction de lecture d'une valeur entière sur l'entrée standard (`read`(x)) et dont le résultat est stocké dans la variable x, une instruction d'affichage d'une valeur entière (`write`(e)) et, enfin, l'affichage d'une chaîne de caractère sur la sortie standard (`message`$(chaine)$). Un programme *prog* est formé d'une liste de déclarations globales d'exceptions $T_1, ..., T_n$ suivie d'un bloc. Un bloc est une liste de déclarations de variables suivie d'une instruction (`var` $x_1, ..., x_n$ `in` *inst*). Le seul type manipulé dans un programme est le type entier (les comparaisons seront ainsi interprétées comme produisant un résultat entier).

La syntaxe abstraite de ce langage est définie par le type Caml suivant.

```
type prog =
   { exn : string list;
     bloc: bloc }

and bloc = { vars: string list; instructions: inst }

and exp =
    | Eop of op * exp * exp
    | Eint of int
    | Evar of string

and op = Eplus | Eminus | Etimes | Eeq | Esup

and inst =
    | Iseq of inst * inst
    | Iaffect of string * exp
    | Ifthenelse of exp * inst * inst
    | Iwhile of exp * inst
    | Itry of inst * (string * inst) list
    | Iexit of string
    | Iskip
    | Iread of string
    | Iwrite of exp
    | Imessage of string
```

Les structures de contrôle (conditionnelle et boucle) ont le sens habituel. Les contructions de déclaration d'exception, de levée et de traitement s'interprètent ainsi :

- `exception` T déclare une exception globale T.
- L'instruction de traitement d'exception `try` *inst* `with` T_1 `do` $inst_1 ... T_n$ `do` $inst_n$ (pour $1 \leq n$) exécute l'instruction *inst* puis se termine si aucune exception n'est levée dans *inst*. Si une exception T est levée, l'exécution de *inst* est interrompue. Si T est l'un des T_i (pris dans l'ordre des traitement), l'instruction $inst_i$ est exécutée. Si T est différent de tous les T_i, l'exception est alors levée à nouveau pou être traitée dans une construction `try/with` plus englobante.
- `exit` T est l'opération de levée d'exception.

Par exemple, le programme suivant attend un entier x puis calcule $x!$. L'exécution se termine lorsque x est négatif. Si x est nul, une exception locale est levée et le programme affiche directement la valeur 1.

```
exception Negatif
exception Nul
var x,res in
try
  while 1 do
    try
      read(x);
      if x = 0 then exit Nul;
      if x < 0 then exit Negatif;
      res := 1;
      while x > 0 do
        res := x * res;
        x := x - 1
      done;
      message("resultat = ");write(res)
    with
      Nul do message("resultat = 1")
  done
with Negatif do skip
```

Partie I: Vérifications statiques

1. (1pt) Les levées d'exception et les traitements d'exception ne peuvent que manipuler des exceptions qui ont déclarées.

   Ecrire une fonction `exceptions: inst -> string list` telle que `exceptions inst` retourne la liste des exceptions manipulées par `inst`. En déduire une fonction `verif_prog: prog -> unit` qui retourne `()` si toutes les exceptions manipulées d'un programme sont effectivement déclarées et lève une exception `Indefinies(l)` où l est la liste des variables non déclarées.
2. (2pt) On souhaite disposer d'une analyse permettant de distinguer les exceptions pouvant être levées des exception nécessairement rattrapées dans le corps d'une instruction. Cette analyse permet, notamment, d'indiquer les traitements d'exception inutiles. C'est le cas, par exemple, avec le programme suivant où le traitement `T1 do write(2)` est inutile :

   ```
   exception T1
   exception T2
   var x:int in
   try
     try
       read(x);
       if x > 0 then exit T1;
       exit T2
     with
       T1 do write(1)
   with
     T1 do write(2)
   ```

```
  T2 do write(3)
```

Ecrire une fonction `exceptions_levees: inst -> string list` telle que `exceptions_levees inst` retourne la liste des exceptions pouvant être levées mais non rattrapées dans `inst`.

Modifier cette fonction pour qu'elle indique également les traitements d'exceptions inutiles.

3. (1pt) On souhaite ajouter les instructions `break` et `continue` au langage. On rappelle que ces deux instructions permettent d'interrompre l'exécution du corps d'une boucle. Lorsque dans le corps d'une boucle, on rencontre l'instruction `break` alors l'exécution se poursuit à la fin de la boucle, lorsque l'on rencontre l'instruction `continue` alors l'exécution reprend au test de la boucle. Lorsque plusieurs boucles sont imbriquées, les instructions `break` et `continue` permettent de sortir de la première boucle englobante.

   Ces deux instructions peuvent être implémentées à l'aide du mécanisme d'exception de TRAP.

   Montrer la traduction dans TRAP du programme suivant.

```
var i, j : int in
read(i);
read(j);
while true do
  if i < 5 then break else skip;
  if j < 5 then continue else skip;
  write(i); write(j)
done
```

   Donner le principe de la traduction des programmes avec instructions `break` et `continue` vers des programmes de TRAP.

Partie II: Génération de code

Le langage est compilé vers le code d'une machine à pile dont les instructions sont celles vues en cours et en TD. Elles sont brièvement rappelées à la fin de ce document.

4. (1,5pt) Ecrire une fonction de compilation des instructions de TRAP vers les instructions de la machine à pile dans le cas où il n'y a pas d'instruction de levée ni de traitement d'exception.

5. (1pt) Donner le code de la machine a pile correspondant au programme suivant :

```
exception Nul

var x,res:int in
try
  read(x);
  if x = 0 then exit Nul;
  res := 1;
  while x > 0 do
    res := x * res;
    x := x - 1
  done;
  write(res)
  with
    Nul do write(1)
```

6. (2pt) Etendre la fonction de compilation des instructions de TRAP pour traiter les instructions de déclaration, de levée et de traitement d'exception. On supposera l'existence de la fonction
   `gen_etiquette: unit -> string`
   permettant de générer un nom d'étiquette neuf.

Partie III: Exceptions et fonctions

On étend maintenant le langage TRAP avec la possibilité de définir des fonctions globales, c'est à dire non limitées au corps d'une fonction. On peut ainsi définir :

```
exception T1
exception T2

function f(x1,x2:int): int throws {T1, T2}
  var x3:int in
  if x1 > 1 then exit T1
  else if x2 > 1 then exit T2
  else return(x1+x2)

function g(y1,y2:int): int throws {}
  var r:int in
  try
    r = f(x1, x2+1);
    return(r)
  with
    T1 do return(0)
    T2 do return(1)
```

On suppose que la syntaxe abstraite est étendue (et modifiée) de la manière suivante :

```
type prog =
   { exn: string list;
     fonctions: (string * fonc) list;
     bloc: bloc }

and fonc = { params: (string * typ) list;
             resultat: typ;
             throws: string list;
             corps: bloc }

and inst = ...
         | Ireturn of exp
and exp = ...
         | Icall of string * exp list
```

Le type `prog` définit la liste des exceptions globales déclarées, une liste de fonctions globales et un bloc. Une fonction est caractérisée par la liste de ses paramètres, du type de son résultat et la liste des exceptions globale qu'elle peut lever.

7. (1pt) Ecrire une fonction `verif_fonction: fonc -> unit` telle que `verif_fonction f` vérifie que toutes les exceptions levées dans le corps de `f` ont bien été déclarées dans son interface. progressivement.
8. (2pt) Pour compiler les fonctions pouvant lever une exception, appliquons le schéma suivant [1] : une fonction f pouvant lever une exception est compilée en une fonction qui retourne un couple de valeurs. La première composante du couple représente la valeur de retour (au sens classique) et la seconde représente l'exception levée. Pour cela, on supposera l'existence d'une fonction `indice: string -> int` associant un indice entier non nul unique à chaque nom d'exception.

   Nous adoptons la convention que la valeur de retour contenant le code de l'exception est égal à 0 lorsqu'aucune exception n'a été levée par la fonction. C'est le cas dans le programme précédent lors de l'évaluation de `f(0,0)`.

   Donnez le schéma de compilation d'une instruction `exit T` lorsque `T` est une exception non rattrapée dans le corps de la fonction. En déduire le schéma de compilation des fonctions.
9. (1,5pt) Lors d'un appel de fonction, l'appelant doit réserver la place pour stocker le résultat ainsi que la valeur de retour d'exception. En fin d'appel, si une exception a été levée, l'exécution doit alors se poursuivre au traitement d'exception le plus proche.

   Donnez le schéma de compilation de l'appel de fonction.

[1] La méthode de compilation considérée ici est inefficace, en particulier en présence de fonctions récursives. Une technique plus efficace consiste à effectuer un saut direct vers le traitement d'exception le plus proche.

10. (bonus, 1,5pt) Proposer une autre méthode de compilation des exceptions et donner les schémas de compilation des instructions de levée et de traitement d'exception.

3 Instructions de la machine à pile

`PUSHI` n	empile la constante entière n.
`PUSHA` l	empile l'adreese correspondant au code de label l.
`ADD`	dépile y puis x et empile $x + y$
`SUB`	dépile y puis x et empile $x - y$
`MUL`	dépile y puis x et empile $x * y$
`EQ`	dépile y puis x et empile 1 si $x = y$ et 0 sinon
`NOT` n	dépile n qui doit être un entier et empile le résultat de $n = 0$
`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.
`PUSHGP`	empile l'adresse gp.
`PUSHFP`	empile l'adresse fp.
`PADD`	dépile un entier n et une adresse a et empile l'adresse $a + n$.
`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.
`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.