Compilation
Maîtrise d'Informatique 2001-2002

Examen Final
1er février 2002

L'énoncé est composé de 2 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscriptes ainsi que les énoncés et supports de cours distribués cette année sont les seuls documents autorisés.
Les réponses aux questions devront être clairement justifiées.

1 Question de cours (3 points)

1. Que signifie le fait qu'un langage comporte une surcharge de types ?
2. Citer des avantages et des inconvénients d'un tel langage pour l'utilisateur.
3. Comment la surcharge est-elle traitée par le compilateur ?

2 Analyse ascendante (5 points)

On se donne la grammaire suivante, l'ensemble des terminaux est {a,b,c}, la grammaire possède deux non-terminaux E (symbole initial) et F, enfin les règles de production sont les suivantes :

E ::= a E b E
E ::= F E
E ::= F
F ::= c

1. Cette grammaire est-elle LL(1) ?
2. Cette grammaire est-elle SLR(1) ?
3. On modifie la grammaire en ajoutant le non-terminal D qui devient le symbole initial et les deux règles de production :

   D ::= ϵ
   D ::= a E D

   (a) Montrer que la grammaire obtenue n'est pas SLR(1).
   (b) Expliquer la nature du conflit et trouver un mot pour lequel ce conflit se produit lors de l'analyse ascendante.

3 Génération de code (12 points)

Ce problème étudie l'optimisation de l'appel de fonction par expansion du corps de la fonction.

Description du langage Un programme du langage est composé d'une suite de déclarations de fonctions :

$$\textbf{let } f\,(x_1,\ldots,x_n) = exp$$

avec f le nom de la fonction, x_i le nom du i-ème paramètre, *exp* une expression représentant le corps de la fonction et pouvant faire référence récursivement à f.

Une *expression* peut être :
- une variable, déclarée dans un **let in** ou bien comme paramètre d'une fonction.
- une valeur entière ou booléenne (`true` ou `false`).
- une expression exp_1 `op` exp_2 formée d'un opérateur binaire prédéfini `op` appliqué à deux expressions exp_1 et exp_2 avec `op` $\in \{+,-,*,=\}$
- un appel de fonction $id(exp_1,\ldots,exp_n)$
- une expression conditionnelle : **if** *exp* **then** exp_1 **else** exp_2
- une déclaration de variable locale suivie d'une expression : **let** $x = exp$ **in** *exp*.

Le langage ne manipule que des objets de type `bool` (booléens) ou `int` (entiers).

Questions Soit le programme :

let `double`(j) = j + j
let `f`(i,j) = **let** n = `double`(i) **in** j * n

1. Donner la suite d'instructions de la machine à pile vue en cours (les principales commandes sont rappelées à la fin de ce document) permettant d'implanter les fonctions `double` et `f` (on précisera la forme des tableaux d'activation de ces fonctions).
2. On souhaite éviter le coût de l'appel de la fonction `double` dans le corps de `f`. Pour cela on décide **d'expanser** `double`, c'est-à-dire de transformer l'appel à `double`(i) en le code du corps de la fonction `double` : $i + i$.
   Donner la suite d'instructions pour compiler `f` en utilisant cette stratégie.
   Comparer la taille de ce code avec celui obtenu à la question précédente.
3. On ajoute au programme les déclarations suivantes :
   let `c`(i) = **let** n = i*i **in** n+i
   let `h`(n,m) = `c`(n)+m
   let `fact`(n) = **if** n=0 **then** 1 **else** n*fact(n-1)
   let `g`(i) = `double`(`fact`(i))
   (a) Redéfinir la fonction `h` en expansant l'appel à `c`. Quel problème se pose et comment peut-on le résoudre ?
   (b) Peut-on expanser l'appel de `fact` dans la fonction `g` ?
   (c) Que se passe-t-il si on expanse l'appel de `double` dans la fonction `g` ? quelle solution proposez-vous ?
4. Décrire un type CAML pour représenter les arbres de syntaxe abstraite du langage.
5. Écrire une fonction qui étant donné une expression e et une liste l de fonctions à expanser, calcule une expression équivalente à e dans laquelle tous les appels à une fonction de l ont été expansés. On pourra supposer que pour chaque nom de fonction déclarée dans le programme, on peut accéder à sa définition (paramètres formels et corps). On pourra préciser sans donner le codage, les prétraitements à réaliser lors de l'analyse sémantique.
6. On décide d'expanser une fonction si elle n'est pas récursive et si son corps ne nécessite pas plus de 10 instructions machine. Décrire une méthode permettant de compiler le code d'un programme en identifiant les fonctions qui répondent au critère et en expansant les appels correspondants.
   Préciser la manière de stocker les paramètres et les variables locales.
   Il est recommandé de simplement spécifier (ie donner le nom, le profil et décrire en langage courant le comportement) des fonctions élémentaires nécessaires à la génération de code, telles que : accès à la table des symboles, génération de nouveaux labels, test si une fonction est récursive...

Machine à pile

Les instructions de cette machine à pile sont analogues à celles décrites dans le cours. Le registre *gp* indique le bas de la pile, le registre *fp* indique le pointeur de référence des variables locales, le registre *sp* indique la première case libre de la pile. Des instructions `LABEL` peuvent être introduites dans le code, elles ne modifient pas l'état de la pile. "Empiler x" signifie ajouter x au sommet de la pile (*sp* est augmenté de 1). "Dépiler x" signifie retirer la valeur au sommet de la pile qui sera notée x (*sp* est diminué de 1).

`PUSHI` n	empile la constante entière n.
`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)
`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*.
`LOAD` n	dépile une adresse a et empile la valeur située n cases au dessus de a.
`STORE` n	dépile une valeur x, une adresse a et stocke x à l'adresse située n cases au dessus de a.
`PUSHN` n	empile n valeurs nulles sur la pile.
`POPN` n	dépile n valeurs de la pile.
`JZ` l	dépile x, si $x = 0$ le pointeur de code saute à l'instruction `LABEL` l.
`JUMP` l	saute à l'instruction `LABEL` l.
`CALL` f	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 `LABEL` f.
`RETURN`	dépile toutes les valeurs au-dessus de *fp*, restaure l'ancienne valeur de *fp*, reprend l'exécution à l'instruction suivante la dernière instruction `CALL`.