Compilation
Maîtrise d'Informatique 2000-2001

Examen Final
25 janvier 2001

L'énoncé est composé de 3 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 Passage de paramètres (4 points)

1. Soit le programme:

```
program param;
var x : int;
procedure proc(y:int);
   begin x:=x+y; y:=y+1 end;
begin x:=2; proc(x); print(x) end;
```

   Quel est le résultat de ce programme lorsque le paramètre `y` de `proc` est passé :

   (a) par valeur,
   (b) par référence,
   (c) par copie/restauration,
   (d) par nom.

2. Quels sont les avantages et les inconvénients de ces différentes modes de passage de paramètres ?

2 Analyse ascendante-descendante (6 points)

On se donne la grammaire suivante pour reconnaître des expressions arithmétiques écrites de manière post-fixée. L'ensemble des terminaux est {`num`,`op`} et la grammaire possède un seul non-terminal E et les règles suivantes :

E ::=`num`
E ::=E E `op`

1. Les expressions suivantes appartiennent-elles au langage engendré par cette grammaire ? Justifier votre réponse.
   (a) `num num num op op`
   (b) `num num op num op op`
2. La grammaire est-elle LL(1) ? LR(0) ? SLR(1) ? Dans le cas de réponse positive, construire la table de transitions correspondante.
3. On suppose que l'on veut traiter également des opérateurs unaires, pour cela on ajoute la règle : E::= E `op`. La nouvelle grammaire ainsi obtenue est-elle LL(1) ? SLR(1) ? LR(1) ?

3 Compilation d'un langage avec tableaux dynamiques (10 points)

Ce problème étudie la compilation d'un mini-langage fonctionnel permettant de manipuler des tableaux de taille arbitraire contenant des données qui peuvent être elles-mêmes des tableaux.

3.1 Syntaxe du langage

Un programme du langage est composé d'une suite de déclarations.

Déclarations Une *déclaration de variable* s'écrit : **let** $id = exp$ avec id le nom de la variable et exp sa valeur.

Une *déclaration de fonction* s'écrit : **let** $id\ (id_1, \ldots, id_n) = exp$ avec id le nom de la procédure, id_i le nom du i-ème paramètre, exp une expression représentant le corps de la fonction et pouvant faire référence à id.

Expressions Une *expression* peut être :
- une variable, déclarée dans un **let** 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` $\in \{+, -, =, <, \leq\}$
- un appel de fonction $id(exp_1,\ldots,exp_n)$
- une expression conditionnelle : **if** *exp* **then** exp_1 **else** exp_2.
- une expression avec déclaration locale *decl* **in** *exp* où *decl* est une déclaration de variable ou de fonction éventuellement récursive avec la même syntaxe que les déclarations globales.

Fonctions prédéfinies On suppose que les fonctions suivantes permettant la manipulation des tableaux sont prédéfinies :

`create` qui étant donné un entier n et une expression x, crée un tableau de taille n initialisé par la valeur de x;

`length` qui étant donné un tableau renvoie sa longueur;

`get` qui étant donné un tableau t et un entier i compris entre 1 et la longueur de t, renvoie la valeur $t[i]$ du tableau t à l'indice i;

`set` qui étant donné un tableau t, un entier i compris entre 1 et la longueur de t, et une expression x, affecte la valeur de x à l'emplacement d'indice i du tableau et renvoie le tableau t ainsi modifié.

3.2 Typage

Les fonctions peuvent prendre des tableaux en arguments et renvoyer des tableaux. Les données manipulées sont des entiers ou bien des tableaux formés d'objets quelconques. Les fonctions peuvent être polymorphes, c'est-à-dire s'appliquer à des tableaux contenant des objets arbitraires.

Les *types* du langage sont soit le type de base des entiers `int`, soit le type des booléens `bool`, soit un type tableau contenant des objets de type τ, noté τ `array`.

Pour traiter le polymorphisme, on introduit les *schémas de type* qui peuvent contenir des variables de type qui représentent des types arbitraires. Un schéma de type est donc soit `int`, soit `bool`, soit une variable de type (notée $\alpha, \beta, \ldots$), soit un schéma de type tableau noté σ `array` avec σ un schéma de type.

Un schéma de type représente un ensemble de types obtenus en remplaçant les variables de type par des types quelconques. Ainsi le schéma de type α représente tous les types, le schéma de type α `array` représente tous les types de la forme τ `array` avec τ un type (ie `int array`, `bool array`, (`int array`) `array`, ((`int array`) `array`) `array` ...)

À toute fonction est associée un *profil* de la forme $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ où σ_i sont des schémas de type.

On peut associer aux opérations primitives les profils suivants :
- `create` : `int` $\times\ \alpha \to \alpha$ `array`
- `length` : α `array` $\to$ `int`
- `get` : α `array` $\times$ `int` $\to \alpha$
- `set` : α `array` $\times$ `int` $\times\ \alpha \to \alpha$ `array`

1. Dire ce que font les fonctions suivantes et donner leur profil.

```
let f(x,y)=x
let get_matrix(m,i,j) = get(get(m,i),j)
let set_matrix(m,i,j,v) = set(get(m,i),j,v)
```

2. Écrire une fonction `add_array` qui prend en argument deux tableaux d'entiers x et y de même taille n et renvoie un nouveau tableau z de taille n tel que pour tout i compris entre 1 et n on ait $z[i] = x[i] + y[i]$.

3.3 Génération de code

Le langage est compilé vers le code d'une machine à pile dont les instructions sont rappelées à la fin de ce document.

Les tableaux sont alloués dans la zone du tas. La valeur d'un tableau sera l'adresse du bloc dans lequel le tableau est stocké. Un tableau n'est jamais recopié lors d'un appel de fonction, seule l'adresse est passée en argument. La seule fonction qui alloue de la place pour un tableau dans le tas est la fonction `create`.

Par rapport à la machine vue en cours, on suppose que l'on dispose de la commande suivante :

`ALLOC` dépile un entier n, réserve un bloc de $n+1$ mots dans le tas et renvoie l'adresse *addr* du bas du bloc au sommet de la pile (les élements du bloc seront donc placés aux adresses *addr* à *addr* $+ n$))

1. Peut-on connaître statiquement (à la compilation) la taille du tableau correspondant à la valeur d'une expression quelconque du langage, en particulier pour l'expression correspondant au corps d'une fonction ? .
2. Proposer une manière de représenter les tableaux qui permette que tous les objets manipulés dans la pile (entiers ou tableaux) aient la même taille et d'implanter la fonction `length`.
3. On veut coder les opérations primitives (`length`, `get`, `set` et `create`). Expliquer l'organisation du tableau d'activation créé par l'appel de ces fonctions et donner le code de la machine à pile correspondant au corps de chacune de ces procédures
4. Soit le programme :
   ```
   let x = create(2,0)
   let y = x
   let x' = set(x,1,2)
   let v = get(y,1)
   ```
   Indiquer l'état de la pile et du tas après l'exécution de chaque déclaration.
5. Donner le code machine correspondant à la fonction `add_array` définie précédemment.
6. L'étudiant Arthur doit implanter une fonction d'addition de trois tableaux de même taille, il décide de réutiliser la fonction `add_array` définie précédemment et procède de la manière suivante :
   ```
   let add3_array(x,y,z) = let t = add_array(x,y) in add_array(t,z)
   ```
   Quel défaut présente la solution choisie ? Quelles solutions pouvez-vous proposer pour y remédier ?
7. On veut étendre le langage pour pouvoir spécifier dans une déclaration de fonction qu'un paramètre (de type tableau) est passé par valeur, c'est-à-dire qu'un nouveau tableau est créé au moment de chaque appel. On introduit pour cela la déclaration **let** f (**val** x) = *exp*. Lors de l'appel de $f(a)$, les modifications apportées aux éléments de x dans l'expression *exp* n'auront pas d'incidence sur les éléments de a.

   Montrer que l'on peut effectuer cette extension du langage sans modifier le compilateur vers la machine à pile, en transformant simplement une déclaration de fonction avec appel par valeur en une déclaration de fonction avec appel par référence ayant le même comportement.

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` (resp. `SUB`)	dépile y puis x et empile $x+y$ (resp. $x-y$)
`EQ` (resp. `INF`,`INFEQ`)	dépile y puis x et empile 1 si $x=y$ (resp. $x<y$, $x \leq y$) 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.
`LOADN`	dépile un entier n et une adresse a, empile la valeur située n cases au dessus de a.
`STOREN`	dépile un entier n, une valeur x et une adresse a 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`.