Compilation
Maîtrise d'Informatique 2000-2001

Examen Final
3 septembre 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 Génération de code pour langages objets (4 pts)

1. Expliquer en quelques lignes la différence entre appel de méthode statique et appel de méthode dynamique dans les langages à objets. Donner un exemple.
2. Expliquer les grandes lignes du schéma de compilation des appels de méthodes dynamique et statique. Quel type d'appel de méthode est le plus efficace ?

2 Analyses ascendante et descendante (8 pts)

Soit la grammaire G définie sur l'ensemble des terminaux $\{\texttt{id}, (,), \texttt{;}\}$ et des noms terminaux $\{S, E\}$ par les règles suivantes :

```
S ::= E ♯
E ::= id
E ::= id(E)
E ::= (E)
E ::= (E;E)
```

1. La grammaire G est-elle LR(0), SLR(1) ?
2. Cette grammaire n'est pas LL(1), expliquer pourquoi.
3. Construire une grammaire G' équivalente à G dans laquelle les règles de G ayant un préfixe commun sont factorisées.
4. Calculer pour la grammaire G' l'ensemble $\textsc{prem}(X)$ pour chaque non-terminal X.
5. Calculer pour la grammaire G' l'ensemble $\textsc{suiv}(X)$ pour chaque non-terminal X différent de S.
6. Construire la table des expansions de G' pour l'analyse descendante.
7. La grammaire G' est-elle LL(1) ?

3 Langage avec valeurs de paramètres par défaut (8 pts)

Consigne Lorsqu'il vous est demandé de décrire une méthode, vous n'êtes pas obligés de donner du code dans tous ces 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, afin que la traduction en un programme ne soit plus qu'un travail de routine.

La grammaire du langage a pour symboles terminaux :

$$\{\texttt{let}, \texttt{type}, \texttt{id}, \texttt{num}, \texttt{true}, \texttt{false}, \texttt{op}, \texttt{:=}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{,}, \texttt{(}, \texttt{)}\}$$

`id` représente un identificateur correspondant à une variable ou une fonction, `num` représente une constante entière, `op` est un des opérateurs arithmétiques $\{+, -, *, /, =, <, >\}$, `type` représente le type d'une expression qui peut être l'un des deux mots clés `int` ou `bool`.

Les expressions représentent des entiers ou des booléens. Elles sont engendrées par la grammaire suivante dans laquelle F représente une suite d'expressions (éventuellement vide) séparées par des virgules :

E	:= `num`		E	:= E `op` E		F	:= ϵ
E	:= `true`		E	:= (E)		F	:= F_1
E	:= `false`		E	:= `id`(F)		F_1	:= E
E	:= `id`		E	:= `if` E `then` E `else` E		F_1	:= F_1 `,` E

Un programme dans ce langage est composé d'une suite de déclarations qui peuvent être soit des déclarations de fonctions soit l'association d'un nom à une expression. Par exemple :

`let` square (x:`int`) : `int` = x * x
`let` f (x:`int`,y:`int`) : `int` := square(x) + square(y)
`let` x := 3
`let` y := 4
`let` z := f (x,y)

La particularité de ce langage est de permettre de donner des valeurs par défaut à certains paramètres des fonctions. Une valeur par défaut peut être n'importe quelle expression. Au moment de l'appel, il n'est pas nécessaire de fournir une valeur pour un tel paramètre, à défaut, la valeur associée au paramètre au moment de la déclaration de la fonction sera utilisée. Ainsi on peut écrire :

`let` g (x:`int`,y:`int`,k=1) : `int` := k * square(x) + square(y)
`let` t := g(x,y)
`let` u := g(x,y,2)

L'identificateur t aura pour valeur $1 * \texttt{square}(3) + \texttt{square}(4)$ c'est-à-dire 25 tandis que u aura pour valeur $2 * \texttt{square}(3) + \texttt{square}(4)$ c'est-à-dire 50 .

On choisit dans la déclaration des paramètres de regrouper tous les paramètres sans valeur par défaut d'abord puis de mettre les paramètres avec valeurs par défaut ensuite.

1. Compléter la grammaire des expressions pour donner la grammaire complète des programmes de ce langage.
2. Compléter les déclarations de type suivantes (`expr` et `prog`) pour représenter les arbres de syntaxe abstraite de ce langage.
   type oper = Div | Plus | Mult | Eq | Minus | Lt | Gt
   type typ = Int | Bool
   type expr = Ident of string | IntCte of int | BoolCte of bool | Oper of oper * expr * expr
   | ...
   type prog = ...
3. Proposer une structure pour la table des symboles des fonctions et des variables qui permette de vérifier qu'un programme est bien formé, c'est-à-dire qu'il respecte les règles usuelles de portée et de typage :
   - toute variable utilisée dans une expression doit au préalable avoir été déclarée.
   - Les paramètres d'une fonction ne sont visibles que dans le corps de la fonction, les fonctions peuvent être récursives

- Les opérateurs s'appliquent à des entiers, sauf `=` qui peut s'appliquer à deux booléens.
- Dans une expression conditionnelle, la condition a pour type un booléen et les deux branches ont le même type.

4. Décrire un programme permettant de vérifier qu'un programme est correctement formé. Il n'est pas nécessaire de détailler les opérations de manipulation de la table des symboles, il suffit de donner leur spécification.
5. On souhaite engendrer du code assembleur pour calculer les valeurs de chaque variable introduite dans le programme. Expliquer de quelle manière on peut traiter les arguments donnés par défaut : détailler la déclaration d'une fonction avec paramètres ayant une valeur par défaut ainsi que l'appel d'une telle fonction.
6. Que faut-il changer au mode de compilation si on autorise les valeurs par défaut à faire référence aux paramètres de la fonction comme dans l'exemple :

   ```
   let h (x:int,y:int,k=x) = k * square(x) + square(y)
   ```