Compilation
Maîtrise d'Informatique 1999-2000

Examen Final
23 janvier 1999

L'énoncé est composé de 4 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 : Grammaire (3 points)

On se donne les deux grammaires suivantes dans lesquelles I est un symbole non-terminal associé à des productions quelconques :

$$G_1 : \boxed{\begin{array}{ll} \mathrm{L} & \coloneqq \mathrm{I\ L} \\ \mathrm{L} & \coloneqq \epsilon \\ \mathrm{I} & \coloneqq \ldots \end{array}} \qquad G_2 : \boxed{\begin{array}{ll} \mathrm{L} & \coloneqq \mathrm{L\ I} \\ \mathrm{L} & \coloneqq \epsilon \\ \mathrm{I} & \coloneqq \ldots \end{array}}$$

Indiquer les différences en terme de faisabilité et d'efficacité entre l'analyse descendante et l'analyse ascendante pour ces deux grammaires.

2 Exercice: Passage de paramètres (3 points)

Soit le programme :

```
program main;
procedure p(x,y,z);
 begin
   y:=y+1;
   z:=z+x;
 end
begin a:=2; b:=3; p(a+b,a,a); print(a) end
```

Quel est le résultat affiché par ce programme dans le cas où les paramètres x, y et z sont passés par valeur ? par référence ? par copy-restore ? par nom ?

3 Exercice :Élimination des labels symboliques (3 points)

On étudie un langage assembleur composé d'une suite d'instructions. Les instructions de saut mentionnent un étiquette symbolique e qui référence une instruction du code de la forme `LABEL` e.

Le langage assembleur est décrit par la grammaire suivante dans laquelle I' est un symbole non-terminal représentant les instructions différentes de `LABEL` :

$$\boxed{\begin{array}{ll} \mathrm{L} & \coloneqq \mathrm{L\ I} \\ \mathrm{L} & \coloneqq \epsilon \\ \mathrm{I} & \coloneqq \texttt{LABEL}\ e \\ \mathrm{I} & \coloneqq \mathrm{I'} \\ \mathrm{I'} & \coloneqq \ldots \end{array}}$$

Le but est de traduire tout programme P en un programme P' qui est exactement le programme P dans lequel toutes les instructions `LABEL` ont disparu, et de calculer la correspondance entre étiquette et numéro d'instruction : à toute étiquette e apparaissant dans une instruction `LABEL` e on associera le numéro dans P' de la première instruction (différente de `LABEL`) qui suit `LABEL` e dans P.

Par exemple :

$P =$	I_1	donne	$P' =$	I_1	et associe l'etiquette e à 2 et f à 4
	`LABEL` e			I_2	
	I_2			I_3	
	I_3			I_4	
	`LABEL` f				
	I_4				

Donner un systèmes d'attributs qui permette d'associer à chaque programme P le programme correspondant sans les instructions `LABEL` et qui associe chaque étiquette au bon numéro d'instruction. On précisera le rôle de chaque attribut, et s'il s'agit d'un attribut synthétisé ou hérité.

4 Problème : Machine à pile et attributs (11 points)

On se donne un mini-langage fonctionnel pour la manipulation d'expressions arithmétiques. Un programme de ce langage est composé de la déclaration de fonctions ou de variables suivi d'une expression à évaluer.

La grammaire est la suivante :

P	≔B E	E	≔`num`
B	≔ϵ	E	≔`id`
B	≔B D	E	≔`id`(F)
D	≔`val id` = E	E	≔`if` E `relop` E `then` E `else` E
D	≔`fun id` (L) = E	E	≔E `op` E
L	≔`id`	F	≔E
L	≔L, `id`	F	≔F, E

`op` (resp. `relop`) est un symbole terminal dont la valeur est l'un des opérateurs arithmétiques $\{+, -, *, /\}$ (resp. $\{<, \leq\}$). E représente une expression, D une déclaration de variable globale ou de fonction, L représente une suite d'identificateurs, F une suite d'expressions, B une suite de déclarations.

La valeur d'un programme formé d'une suite de déclarations B et d'une expression E est égal à la valeur de l'expression E évaluée dans l'environnement correspondant à B. Ce langage est compilé vers la machine à pile décrite en cours.

On appelle P le programme suivant :

```
val a = 4
fun square(z) = z*z
fun f(x,y)=square(x)+square(y)
val b = 2
f(a,b)+a
```

1. Donner pour le programme P une manière de stocker les variables globales et les paramètres des fonctions dans la pile. Indiquer pour chaque variable les codes d'accès en lecture ou en écriture.
2. Donner le code compilé correspondant au programme P.
3. Donner un système d'attributs pour calculer pour chaque déclaration d'identificateur, le décalage par rapport au pointeur global *gp* ou local *fp* de l'objet représenté.
4. On se propose de traduire chaque programme de notre langage vers le code de la machine à pile. On pourra supposer que toutes les déclarations de fonctions ont des noms distincts. On

suppose également qu'une analyse a permis de relier chaque utilisation d'une variable à sa déclaration, ce qui nous permet de tester si la variable est locale ou globale et de connaître son décalage.

La traduction est effectuée à l'aide de fonctions de codage $code_B$, $code_E$, $code_F$ ...pour les principaux non-terminaux de la grammaire. Ces fonctions sont spécifiées par des équations. Le début de la spécification est le suivant :

$$\begin{array}{ll}
code_P(B\ E) & = code_B(B)\ code_E(E) \\
code_B(\epsilon) & = [] \\
code_B(B\ D) & = code_B(B)\ code_D(D) \\
code_E(num) & = \texttt{PUSHI}\ num \\
code_E(E_1\ \texttt{op}\ E_2) & = code_E(E_1) code_E(E_2) code_{\texttt{op}}(\texttt{op}) \\
code_{\texttt{op}}(+) & = \texttt{ADD} \\
code_{\texttt{op}}(-) & = \texttt{SUB} \\
code_{\texttt{op}}(*) & = \texttt{MUL} \\
code_{\texttt{op}}(/) & = \texttt{DIV} \\
code_{\texttt{relop}}(<) & = \texttt{INF} \\
code_{\texttt{relop}}(\leq) & = \texttt{INFEQ}
\end{array}$$

Compléter cette spécification pour l'ensemble de la grammaire.

5. On suppose dans un premier temps que les fonctions définies ne sont pas récursives. Décrire comment calculer pour chaque programme la taille maximale de la pile utilisée lors de l'exécution de ce programme (on pourra utiliser des attributs ou une définition récursive de fonction).
6. On suppose maintenant que les fonctions peuvent être récursives. Que peut-on dire sur la taille de la pile ?
7. La récursion terminale, est un cas particulier de récursion où on peut limiter la taille de la pile.

   On dira qu'un identificateur f est *terminal* dans une expression e si l'une des conditions suivantes est vérifiée :
   - f n'apparaît pas dans e,
   - $e = f(e_1, \ldots, e_n)$ et f n'apparaît dans aucun des e_i,
   - $e = \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ et f n'apparaît pas dans e_0 et est terminal dans e_1 et e_2.

   Une déclaration : `fun` $f(L) = E$ est récursive terminale si f est terminale dans E.

   On peut par exemple écrire une version récursive terminale de la factorielle :

   ```
   fun fact_it(n,m)=if n=0 them m else fact_it(n-1,n*m)
   fun fact(n)=fact_it(n,1)
   ```

   Donner un système d'attributs permettant de détecter qu'un identificateur f est terminal dans une expression e.
8. Soit f un identificateur terminal dans une expression e. Montrer que si le calcul de e fait un appel à $f(u_1, \ldots, u_n)$ alors c'est que la valeur de e est la même que la valeur de $f(u_1, \ldots, u_n)$.
9. Si le corps de la fonction récursive terminale $f(x_1, \ldots, x_n)$ fait un appel à $f(u_1, \ldots, u_n)$, on décide d'empiler les valeurs de $u_1, \ldots, u_n$ à la place des arguments $x_1, \ldots, x_n$ et de mettre la valeur de retour de $f(u_1, \ldots, u_n)$ à la place de la valeur de retour $f(x_1, \ldots, x_n)$.

   Comment faut-il modifier la génération de code correspondant à la production $E ::= id(F)$ lorsque l'on compile le corps de la fonction récursive terminale f et que $id = f$.
10. Étendre le calcul de la taille maximale de la pile au cas de déclarations de fonctions récursives terminales compilées de la manière précédente.
11. En utilisant cette stratégie, calculer la taille de la pile nécessaire à l'évaluation de `fact(10)`.