Compilation
Maîtrise d'Informatique 1999-2000

Examen Final
17 janvier 1999

L'énoncé est composé de **??** 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 (2 points)

Quels sont les différents rôles des types dans un langage de programmation lors de la compilation ?

2 Exercice: Analyse ascendante (5 points)

Soit la grammaire de symboles terminaux : $\{*, a, b, +\}$, non-terminaux :$\{E, T, F\}$ et de règles :

E	≔	E + T	F	≔	F *
E	≔	T	F	≔	a
T	≔	T F	F	≔	b
T	≔	F			

1. Le mot `b**+ab*+a` appartient-il au langage reconnu par cette grammaire ?
2. Donner sous forme d'expression régulière le langage reconnu par cette grammaire.
3. Cette grammaire est-elle LR(0), SLR(1), LALR(1) ?
4. Donner les différentes étapes de l'analyse ascendante du mot de la question **??**.

3 Problème : Compilation d'un langage orienté objet (13 points)

Ce problème étudie la compilation d'un mini-langage orienté objet. Un programme de ce langage est composé de déclarations de classes, suivies de déclarations de variables, suivies d'une suite d'instructions.

Préambule L'énoncé du problème est long, lisez-le soigneusement. Les questions sont largement indépendantes. Lorsqu'il vous est demandé de décrire une méthode, vous devez indiquer précisément les structures de données manipulées et les algorithmes utilisés. Vous pouvez utiliser les notations d'un pseudo-langage algorithmique. Il n'est pas nécessaire de détailler le code des opérations élémentaires mais vous devez par contre indiquer ce qu'elles font.

Grammaire La grammaire du langage a pour symboles terminaux :

`Cid`	identificateurs de classe
`id`	identificateurs de variable ou méthode
`num`	constantes entières
`op`	opérateurs arithmétiques $\{+, -, *, /, <, \leq\}$
`class extends var new if then else fi`	mots clés
`= . := , ( )`	symboles

Les symboles non-terminaux sont : $\{P, C, X, V, M, X, L, E, F, I, J\}$. P représente un programme, C une suite de déclarations de classes, V une suite de déclarations de variables, M une suite de déclarations de méthodes, X une suite de variables associées à des identificateurs de classe (leur *type*), L une valeur gauche (une variable déclarée globalement, le paramètre d'une méthode ou bien une variable d'un objet lui-même désigné par une valeur gauche), E une expression (un entier, une opération arithmétique appliquée à des entiers, un nouvel objet, une variable ou le champ d'un objet), F une suite d'expressions, I une instruction (affectation, conditionnelle, application d'une méthode) et J une suite d'instructions. Les règles sont :

```
P  ≔  C V J
C  ≔  ϵ
C  ≔  C class Cid extends Cid V M
V  ≔  ϵ
V  ≔  V var id := E
M  ≔  ϵ
M  ≔  M method id(X) = E
```

```
L  ≔  id
L  ≔  L.id
E  ≔  num
E  ≔  new Cid
E  ≔  L
E  ≔  E op E
F  ≔  ϵ
F  ≔  E
F  ≔  F, E
```

```
I  ≔  L:= E
I  ≔  L.id(F)
I  ≔  if E then J else J fi
J  ≔  ϵ
J  ≔  J I
X  ≔  ϵ
X  ≔  id : Cid
X  ≔  X, id : Cid
```

Classes prédéfinies On suppose prédéfinis deux identificateurs de classe : `Object` (classe universelle de tous les objets) et `Int` la classe des entiers. On suppose qu'il n'y a pas de variables ni de méthodes définies dans ces classes.

Exemple On appelle P le programme suivant :

```
class Vehicule extends Object
      var pos := 0
      method move(x:Int) = pos:=pos+x
class Voiture extends Vehicule
      var places := 1
      method rejoint(v:Vehicule) =
          if v.pos < pos then v.move(pos-v.pos) else self.move(v.pos-pos)
class Camion extends Vehicule
      var charge := 10
      method move(x:Int) = if x<10 then pos:=pos+x else pos:=pos+10 fi
var c:= new Camion
var v:= new Voiture
v.places:= 4
v.move(30)
c.move(10)
v.rejoint(c)
```

Visibilité Soit un programme P formé des déclarations de classes C, de variables V et d'instructions J.

Les variables visibles dans les instructions J sont exactement les variables déclarées par V. Les variables d'une classe *Cid* sont composées des variables de la classe qu'elle étend, plus celles qu'elle déclare en propre. Les variables visibles dans une méthode de la classe *Cid* sont toutes les variables de la classe *Cid*.

Exemple la classe `Vehicule` n'a qu'une variable `pos` (elle étend `Object` qui ne contient pas de variable) alors que la classe `Voiture` a deux variables: la variable `pos` héritée de `Vehicule` et la variable `places` déclarée dans `Voiture`. De même la classe `Camion` a deux variables: `pos` héritée de `Vehicule` et `charge` déclarée dans `Camion`.

Table des symboles On suppose que l'analyse de visibilité a été faite, c'est-à-dire que chaque identificateur (de classe, variable, paramètre ou méthode) a une entrée dans une table des symboles. Cette table est accessible à partir de la déclaration de l'identificateur. Chaque utilisation d'un identificateur de classe est reliée à cette entrée de la table de symbole. Une valeur gauche l est formée d'une suite de noms de variables $x_1.x_2 \dots x_n$, les conditions de visibilité demandent que x_1 soit une variable ou un paramètre de méthode préalablement déclaré. L'utilisation de x_1 peut donc être reliée à sa déclaration via la table des symboles. Par contre, l'interprétation des identificateurs $x_2, \dots, x_n$ dépend de la classe des objets manipulés, la liaison entre l'utilisation de ces identificateurs et leur déclaration sera donc faite plus tard au moment du typage ou de la compilation. Il en est de même pour les identificateurs de méthode.

C'est à vous de préciser quelles informations doivent être stockées dans la table des symboles pour réaliser les fonctions demandées.

Exemple Pour ajouter une information sur la nature de l'identificateur *id*, on décide que la table des symboles comportera un champ *genre* dont la valeur peut-être `paramètre`, `variable`, `method` ou `classe`. Vous pouvez utiliser la notation *id.genre*:=`method` pour enregistrer que l'identificateur *id* est une méthode et tester directement la valeur de *id.genre*=`method` pour traiter le cas où *id* est une méthode.

Valeur d'un objet Un objet dans une classe *Cid* est représenté en mémoire en juxtaposant les valeurs des variables de la classe de cet objet dans des cases contiguës de la mémoire. A sa création par la fonction `new` *Cid*, un objet de la classe *Cid* est créé avec les valeurs initiales données aux variables dans la définition de la classe.

Exemple un nouvel objet dans la classe `Vehicule` est représenté par la variable `pos` qui est initialisée à 0; un nouvel objet de la classe `Voiture` est représenté par deux variables : `pos` héritée de la classe `Véhicule` initialisée à 0 et `places` initialisée à 1; un nouvel objet de la classe `Camion` est également composé de deux variables : `pos` initialisée à 0 ainsi que `charge` initialisée à 10.

Valeur d'un programme Un programme P est formé d'une suite de déclarations de classes C, de variables V et d'instructions J. La *valeur* de P est l'ensemble des valeurs des variables V à l'issue de l'exécution des instructions J.

Compilation La compilation des méthodes est analogue à celle des procédures. Une méthode m de paramètres X déclarée dans une classe *Cid* se compile comme une procédure admettant un paramètre supplémentaire `self` : *Cid* que l'on peut choisir de mettre en premier. Ainsi l'invocation $L.m(F)$ est traduite comme l'invocation de la procédure m sur l'objet représenté par la valeur gauche L et les paramètres F. Le passage des paramètres se fait par référence.

Questions

1. Donner un type pour représenter les arbres de syntaxe abstraite de ce langage.
2. Donner l'arbre de dérivation syntaxique et l'arbre de syntaxe abstraite pour le programme P.
3. **Typage** Les classes permettent d'associer un type à chaque objet. Ainsi `new` *Cid* est un nouvel objet dont le type est la classe *Cid*.
   Dans les langages objet, les types sont ordonnés par une relation d'héritage. Si C_1 et C_2 sont deux identificateurs de classe, alors on dira que $C_1 \leq C_2$ si l'une des conditions suivantes est vérifiée :
   – $C_1 = C_2$;
   – C_1 est une classe déclarée avec la mention `extends` C_2;
   – il existe C_3 un identificateur de classe tel que $C_1 \leq C_3$ et $C_3 \leq C_2$.
   Si une expression E a pour type la classe C_1 et que $C_1 \leq C_2$ alors E a également pour type C_2.
   (a) Étant donné un programme P et deux identificateurs de classe C_1 et C_2, on veut savoir si $C_1 \leq C_2$. Proposer une méthode pour calculer cette information, indiquer la complexité en espace et en temps de votre méthode.
   (b) Une variable est soit un paramètre de méthode qui est déclaré avec son type, soit une variable x déclarée par `var` x `:=` E. Le type de x est alors le type de E. Dans le corps d'une méthode m de la classe *Cid*, `self` est un paramètre implicite, déclaré avec le type *Cid*.
   Une valeur gauche L représente un objet. Si L est de la forme *id* alors c'est une variable ou un paramètre déclaré, relié à sa déclaration dans la table des symboles. Si L est de la forme *L'*`.id` alors L' est une valeur gauche qui représente un objet, soit C_1 la classe de cet objet. *L'*`.id` est correctement typé si *id* est une variable de la classe C_1, auquel cas le type de *L'.id* est égal au type de cette variable.
   E_1 `op` E_2 est une expression bien typée de type `Int` si E_1 et E_2 sont bien typés de type `Int`.
   Question : Donner une méthode pour vérifier qu'une expression est correctement typée et calculer le type de chaque variable déclarée et de chaque expression.
   (c) Donner un système d'attributs de la grammaire afin de calculer, pour chaque classe, la taille occupée en mémoire par un objet dans cette classe. Dire pour chaque attribut utilisé s'il est synthétisé ou hérité et ce qu'il représente.
   (d) Les règles suivantes déterminent lorsqu'une instruction I est correcte :
   – L`:=`E est correcte si E est de type C_1, L de type C_2 et $C_1 \leq C_2$;
   – `if` E `then` J_1 `else` J_2 `fi` est correcte si E est une expression de type `Int` (0 représente `faux` et un entier non nul représente `vrai`), et J_1 et J_2 sont des instructions correctes;
   – L`.id`(F) est correcte si :
   L est une valeur gauche qui représente un objet de la classe *Cid*,
   id est une méthode de la classe *Cid* qui attend k arguments de type $C_1, \ldots, C_k$; F représente une liste de k arguments $f_1, \ldots, f_k$ et soit C'_i le type de f_i on a $C'_i \leq C_i$.
   Question : Donner une méthode pour vérifier qu'une séquence d'instructions est bien formée.

(e) Une déclaration de classe est bien formée, si les déclarations de variables dans la classe sont bien formées et si pour chaque méthode, les déclarations de variables et les instructions de la méthode sont bien formées.
De plus si $C_1 \leq C_2$, une méthode déclarée dans C_2 peut être redéfinie dans C_1 auquel cas les types des paramètres doivent être les mêmes.
Question : Donner une méthode pour vérifier qu'une déclaration de classe est bien formée.

4. **Méthodes statiques/ méthodes dynamiques** Une méthode m définie dans une classe C_1 peut être redéfinie dans une classe C_2 telle que $C_2 \leq C_1$. Si on invoque $c.m$ il faut déterminer quel code appliquer, pour cela on s'appuie sur la classe de c. On distingue, la *classe statique* de c qui est la classe déterminée statiquement par le programme (celle que vous avez calculée à la question **??**) de la *classe dynamique* qui est la classe de l'objet correspondant à la valeur de la variable. Par exemple si x est une variable de type C_1 et v un objet de type C_2 alors on peut exécuter l'instruction $x := v$. À la suite de cette instruction, le type dynamique de x est devenu C_2.
Dans le cas statique, l'invocation de $x.m$ correspondant au type déclaré de x soit C_1, dans le cas dynamique, il s'agit du type à l'exécution de x soit C_2.
Question Donner le résultat (valeurs des variables `v` et `c`) de l'évaluation du programme P dans les cas où les méthodes sont compilées de manière statique puis de manière dynamique.
5. **Compilation statique**
   (a) Réécrire le code du programme P dans un langage à la Pascal ne comportant pas de fonctionnalités objet (transformer les classes en record, les méthodes en procédures, introduire des fonctions correspondant à `new`).
   (b) Donner le code compilé correspondant au programme P dans le cas où les méthodes sont toutes statiques. On utilisera le code intermédiaire de la machine à pile donné à la fin de ce document.
6. **Compilation dynamique** Dans le cas de la compilation dynamique, le label du code à exécuter ne peut être résolu qu'au moment de l'exécution. On choisit donc de stocker avec chaque objet en plus des valeurs des variables, les labels des différentes méthodes que cet objet peut invoquer.
   (a) Est-il en général possible de trouver un programme Pascal équivalent au programme objet avec méthodes dynamiques ?
   (b) Quelles instructions faut-il ajouter à la machine pour compiler le code correspondant ?
   (c) Donner le code compilé correspondant au programme P.

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` (`SUB`,`INF`)	dépile y puis x et empile $x+y$ ($x-y$, 1 si $x<y$ et 0 sinon)
`PUSHG` n	empile la valeur située n cases au dessus de *gp*.
`STOREG` n	dépile x et le stocke à l'emplacement situé n cases au dessus de *gp*.
`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` n	empile l'adresse de la n-ème case au dessus de *gp*.
`PUSHFP` n	empile l'adresse de la n-ème case au dessus de *fp*.
`LOAD` n	dépile une adresse x et empile la valeur située n cases au dessus de x.
`STORE` n	dépile une valeur a, une adresse x et stocke a à l'adresse située n cases au dessus de x.
`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`.