Compilation
Master Informatique M1 2004-2005

Examen Partiel

8 novembre 2004

L'énoncé est composé de 4 pages. L'examen dure 3 heures. Les notes de cours et de TD manuscrites ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.

1 Analyse lexicale (6 pts)

Dans cet exercice on se propose d'écrire un programme d'indentation automatique de code Java à l'aide d'ocamllex. L'indentation consiste à ajouter plus ou moins d'espace au début de chaque ligne, de manière à faire ressortir la structure du programme et donc à le rendre plus lisible.

Dans cet exercice on utilise un schéma d'indentation extrêmement simple qui fonctionne de la manière suivante. On appelle *marge* le nombre d'espaces que l'on va ajouter au début de la ligne courante. Initialement, la marge vaut 0. On procède alors ainsi :

- Les espaces présents au début de chaque ligne dans le code original sont ignorés et remplacés par exactement *marge* caractères « espace ».
- Si l'on rencontre un caractère `{` on augmente la marge de 4 et on l'affiche. Si ce caractère apparaît en début de ligne — i.e. est le premier caractère de la ligne qui ne soit pas un espace — on prendra soin d'augmenter la marge *après* l'avoir affiché.
- Si l'on rencontre un caractère `}` on diminue la marge de 4 et on l'affiche. Si ce caractère apparaît en début de ligne, on prendra soin de diminuer la marge *avant* de l'afficher.
- Tous les autres caractères sont affichés tels quels.

Voici respectivement à gauche et à droite un programme Java et le résultat de son indentation automatique :

```
class A
{
int x;
void f() { x=1; }
int g() {
if (x>0) {
x--;
}
return x;
}
}
```

```
class A
{
    int x;
    void f() { x=1; }
    int g() {
        if (x>0) {
            x--;
        }
        return x;
    }
}
```

On pourra supposer que les programmes Java considérés sont syntaxiquement bien formés.

1. Écrire un programme ocamllex réalisant l'indentation automatique de programmes Java. Le nom du fichier à indenter sera donné comme unique argument sur la ligne de commande. Le résultat sera affiché sur la sortie standard.

   Votre fichier ocamllex doit constituer un programme complet : une fois compilé avec ocamllex puis ocamlc il doit donner un exécutable répondant à la spécification.

2. Pour fonctionner correctement, l'indentation ne doit pas considérer les caractères `{` ou `}` présents dans des commentaires ou des chaînes de caractères. On rappelle les conventions lexicales de Java :

- un commentaire Java débute par `//` et s'étend jusqu'à la fin de la ligne, ou débute par `/*` et s'étend jusqu'à la première occurrence de `*/`
- une chaîne de caractères est encadrée par `"` ; elle peut contenir le caractère `"` seulement si celui-ci est précédé d'un caractère \ et ne peut pas contenir de retour-chariot.

Ajouter des règles à votre analyseur ocamllex pour prendre en compte les commentaires et les chaînes de caractères. (On donnera seulement les nouvelles règles et les actions associées.)

2 Analyse syntaxique (7 pts)

1. Soit la grammaire G_1 définie sur l'ensemble des terminaux $\{a, b\}$ et le non-terminal S par les règles suivantes :

S ::= ϵ	S ::= SS
S ::= aSb	S ::= bSa

   Réaliser l'analyse ascendante du mot *aabbba*.

   Le mot *aabba* est-il dérivable ? Justifier.

   Quel est le langage reconnu par cette grammaire ? (On ne demande pas de démonstration formelle).
2. Cette grammaire est-elle LL(1) ? LR(0) ? SLR(1) ? Justifier vos réponses (si nécessaire, calculer les tables de transitions de l'automate d'analyse associé).
3. Soit la grammaire G_2 définie sur le même ensemble de terminaux $\{a, b\}$ et l'ensemble des non-terminaux $\{S, A, B\}$, avec S le symbole initial, par les règles suivantes :

S ::= ϵ	
S ::= aAS	S ::= bBS
A ::= aAA	A ::= b
B ::= bBB	B ::= a

   Les mots reconnus par cette grammaire on autant de a que de b, expliquer pourquoi.
4. Cette grammaire est-elle LL(1) ? LR(0) ? SLR(1) ? Justifier vos réponses (si nécessaire, calculer les tables de transitions de l'automate d'analyse associé).

3 Descriptions BNF (7 points)

La syntaxe des langages de programmation est souvent présentée en utilisant une description BNF (Backus-Naur Form). Cette notation utilise comme pour les grammaires la notion d'alphabet non-terminal et d'alphabet terminal. Les productions sont de la forme $X ::= w$ avec X un non-terminal. Dans les grammaires, w, la partie droite de la production, est un mot formé par concaténation de terminaux et de non-terminaux. Dans une description BNF, ce mot peut contenir de plus des constructions avancées (analogues à celles des expressions régulières). Ainsi pour répéter un nombre fini de fois un certain motif m, on utilisera l'opération d'étoile (notée $m*$); pour indiquer un motif optionnel m, on utilisera la notation $m?$; pour indiquer le choix entre deux motifs m_1 et m_2, on utilisera l'opération union notée $m_1|m_2$; finalement m_1m_2 représente la concaténation de deux expressions BNF quelconques et les crochets carrés [et] servent à parenthéser les expressions : ainsi $[m_1|m_2]*$ correspond à une succession de motifs m_1 et m_2.

La syntaxe des mots BNF est (`T` correspond aux terminaux et `NT` aux non-terminaux) :

m ::= `T`	m ::= `NT`	m ::= [m]
m ::= m m	m ::= m | m	
m ::= m ?	m ::= m *	

Exemple On pourra donner la description BNF suivante des expressions arithmétiques avec addition et division, en utilisant les non-terminaux E,T et F et les terminaux {+,/,(,),**id**,**int**} pour les distinguer des symboles BNF.

E ::= T [+ T]* T ::= F [/ F]* F ::= (E) | **id** | **int**

La notion de dérivation est étendue aux descriptions BNF. En plus du remplacement d'un non terminal par la partie droite de la production, on a des dérivations pour chacune des constructions étendues (union, étoile ou option)

$w_1 X w_2$	$\rightarrow w_1 w w_2$	si X::= w est la production associée à X
$w_1[v_1 \| v_2]w_2$	$\rightarrow w_1 v_1 w_2$	choix de la partie gauche de l'union
$w_1[v_1 \| v_2]w_2$	$\rightarrow w_1 v_2 w_2$	choix de la partie droite de l'union
$w_1[v] * w_2$	$\rightarrow w_1 w_2$	réduction de l'étoile vers le mot vide
$w_1[v] * w_2$	$\rightarrow w_1 v[v] * w_2$	dépliage de l'étoile
$w_1[v]?w_2$	$\rightarrow w_1 w_2$	réduction de l'option vers le mot vide
$w_1[v]?w_2$	$\rightarrow w_1 v w_2$	dépliage de l'option

1. Donner une dérivation du mot `id + id / int` à partir du non-terminal E.
2. Est-il possible de dériver le mot vide à partir du non-terminal E ? Quels sont les premiers caractères possibles des mots dérivés à partir de E ?
3. Pour représenter les mots des descriptions BNF, on introduit le type CAML `word`. On choisit dans ce type de représenter les N non-terminaux par des entiers (positifs) compris entre 0 et $N-1$ et les terminaux par des caractères.

```
type word = T of char | NT of int | Union of word * word
          | Star of word | Opt of word | Concat of word * word
```

   Dans une description BNF, du fait de l'union, il est possible de n'associer qu'une seule production à chaque non terminal. Une description BNF sera dans la suite représentée par un tableau `gram` de taille N contenant des mots BNF, avec la convention que la production associée au non-terminal `NT`(i) est `NT`(i)::= `gram.`(i).
4. On suppose donné un tableau de booléens `nulltab` de taille N tel que `nulltab.`(i) est vrai si et seulement si `NT`(i) permet de dériver le mot vide.
   Écrire une fonction CAML `nullword` qui prend en argument `nulltab` et un mot w et renvoie vrai si et seulement si le mot w permet de dériver le mot vide.
5. Le calcul de la table `nulltab` pour la grammaire `gram` se fait par itérations successives. À l'étape i, la table contient la possibilité de dériver le mot vide en moins de i pas, le processus s'arrête lorsqu'un point fixe est atteint.
   On suppose donnée en CAML une fonction `fix` qui prend en argument un tableau `tab` de type `'a array`, une fonction `transform` de type `'a array` → `int` → `'a` et une fonction `changed` de type `'a` → `'a` → `bool`. La fonction `fix` applique la fonction `transform` à tous les élements du tableau (`tab.(i)<-transform tab i`) jusqu'à ce que ces éléments soient tous inchangés par cette transformation (ie `changed` `tab.`(i) (`transform` `tab` i) = `false` pour tout i).
   Utiliser la fonction `fix` pour définir une fonction `null` qui calcule le tableau de booléens qui à chaque non-terminal de la description `gram` associe `true` si et seulement si ce non-terminal permet de dériver le mot vide.
6. On se donne maintenant un module `Ensc` pour manipuler des ensembles de caractères.

```
module Ensc : sig
  type t
```

```
  val vide : t
  val appartient : char -> t -> bool
  val singleton : char -> t
  val union : t -> t -> t
  val egal : t -> t -> bool
end
```

On suppose donné le tableau de booléens `nulltab` (défini comme précédemment) ainsi qu'un tableau d'ensembles de caractères `firsttab` de taille N tel que `firsttab.`(i), qui a pour type `Ensc.t`, est l'ensemble des premiers caractères des mots dérivables à partir de `NT`(i).
Écrire une fonction CAML `firstword` qui prend en argument `nulltab` et `firsttab` et un mot w et renvoie l'ensemble des premiers caractères des mots dérivables à partir de w.

7. Utiliser la fonction `fix` pour définir une fonction `first` qui étant donné un tableau de booléens `nulltab` qui indique pour chaque non-terminal s'il permet de dériver le mot vide, calcule le tableau donnant pour chaque non-terminal l'ensemble des premiers caractères des mots dérivables à partir de ce non-terminal.

8. On veut construire un analyseur descendant. Pour cela on construit une fonction `analyse_rec` qui prend en argument une liste l de mots BNF $[w_1; \dots; w_k]$ correspondant à l'état courant de l'analyse et une entrée m sous la forme d'une chaîne de caractères. La fonction analyse l'entrée et échoue dans le cas où l'entrée n'est pas dérivable à partir du mot $w_1 \dots w_k$.
   - Initialement $l = [\texttt{NT}(0)]$ qui est le non-terminal initial
   - Dans le cas général, si la liste l n'est pas vide et l'entrée n'est pas complètement lue, on raisonne par cas suivant la forme de w_1 et le prochain caractère à lire en entrée.
     - si w_1 est un terminal $\texttt{T}(c)$ alors l'entrée doit également débuter par c, on avance le curseur de lecture de l'entrée m et on poursuit l'analyse avec $[w_2; \dots; w_k]$.
     - si w_1 est un non-terminal $\texttt{NT}(i)$ alors on regarde le mot w associé à ce non-terminal dans la grammaire, on poursuit avec la liste $[w; w_2; \dots; w_k]$
     - si w_1 est une concaténation $\texttt{Concat}(v_1, v_2)$ alors on poursuit l'analyse avec $[v_1; v_2; w_2; \dots; w_k]$
     - si w_1 est une union $\texttt{Union}(v_1, v_2)$ alors on teste si le prochain caractère de l'entrée m fait partie des premiers caractères dérivables à partir de v_1 (resp. v_2). On poursuit l'analyse avec $[v_1; w_2; \dots; w_k]$ (resp. $[v_2; w_2; \dots; w_k]$). On suppose qu'il n'y a pas d'ambiguité, c'est-à-dire que les ensembles de premiers caractères de mots issus de v_1 et v_2 ont une intersection vide.
     - si w_1 est une étoile $\texttt{Star}(v)$ alors on teste si le prochain caractère de l'entrée m fait partie des premiers caractères dérivables à partir de v (resp. $w_2 w_3 \dots w_k$). On poursuit l'analyse avec $[v; \texttt{Star}(v); w_2; \dots; w_k]$ (resp. $[w_2; \dots; w_k]$). On suppose qu'il n'y a pas d'ambiguité, c'est-à-dire que les ensembles de premiers caractères de mots issus de v et $w_2 w_3 \dots w_n$ ont une intersection vide.
     - si w_1 est une option $\texttt{Option}(v)$ alors on teste si le prochain caractère de l'entrée m fait partie des premiers caractères dérivables à partir de v (resp. $w_2 w_3 \dots w_k$). On poursuit l'analyse avec $[v; w_2; \dots; w_k]$ (resp. $[w_2; \dots; w_k]$). On suppose qu'il n'y a pas d'ambiguité, c'est-à-dire que les ensembles de premiers caractères de mots issus de v et $w_2 w_3 \dots w_n$ ont une intersection vide.
   - Si l'entrée est complètement lue alors l'analyse réussit si et seulement si chacun des mots w de la liste l permet de dériver le mot vide.
   - Si l'entrée n'est pas complètement lue mais que la liste l est vide alors cela correspond à un échec de l'analyse.

   Question Utiliser la description précédente et les fonctions définies précédemment pour proposer un codage de la fonction `analyse_rec` puis un codage de la fonction `analyse` qui prend en argument la description BNF et le mot d'entrée et échoue si le mot n'est pas dérivable à partir du non-terminal initial `NT`(0).