Compilation
Maîtrise d'Informatique 2001-2002

Examen Partiel
7 novembre 2001

L'énoncé est composé de 4 pages. L'examen dure 2 heures. Les notes de cours et de TD manuscriptes ainsi que les supports de cours distribués cette année sont les seuls documents autorisés.

1 Analyse descendante (12 pts)

On modélise par une grammaire G un sous-ensemble du français. Les classes des unités lexicales sont : $\{\texttt{verbe}, \texttt{nom}, \texttt{qui}, \texttt{et}, \texttt{.}\}$. Ces classes sont aussi les symboles terminaux de la grammaire G. L'unité lexicale `verbe` peut prendre les valeurs : `dérange`, `suit`, `aime`, `frappe` et `écoute`. L'unité lexicale `nom` peut prendre les valeurs : `Alphonse`, `Bastien`, `Yannick` et `Hélène`. Les unités lexicales `qui`, `et` et `.` ont une seule valeur (eux-mêmes).

Les symboles non-terminaux de la grammaire G sont : $\{P, S, C, R\}$. Le symbole P représente les phrases, S les sujets, C les compléments directs et R les subordonnées relatives.

Les productions de la grammaire sont :

$$
\boxed{
\begin{array}{ll}
P & ::= S\ \texttt{verbe}\ C\ \texttt{.} \\
S & ::= \texttt{nom} \\
C & ::= \texttt{nom} \\
C & ::= \texttt{nom}\ R \\
R & ::= \texttt{qui}\ \texttt{verbe}\ C \\
R & ::= R\ \texttt{et}\ R
\end{array}
}
$$

1. On veut analyser la phrase :

   Alphonse suit Bastien qui écoute Hélène qui dérange Yannick et qui aime Alphonse.

   (a) Donner la suite de terminaux de la grammaire correspondant à cette phrase après la phase d'analyse lexicale.

   (b) Construire un arbre de dérivation syntaxique pour cette phrase.

2. (a) La grammaire proposée est-elle LL(1) ? justifier la réponse.

   (b) La grammaire proposée est-elle ambiguë ? justifier la réponse.

3. (a) Modifier les productions

$$
\boxed{
\begin{array}{ll}
R & ::= \texttt{qui}\ \texttt{verbe}\ C \\
R & ::= R\ \texttt{et}\ R
\end{array}
}
$$

   afin de supprimer la récursion gauche.

   (b) Soit G_1 la grammaire ainsi obtenue. Calculer la table de transitions pour l'analyse LL(1) de G_1 uniquement pour les lignes correspondant au non-terminal R et au nouveau non-terminal R' introduit pour supprimer la récursion gauche.

(c) Quelle constatation peut-on faire ? Quelle explication peut-on donner ?

4. On considère la grammaire G_2 obtenue à partir de la grammaire initiale G en supprimant la production $R ::= R$ `et` R. Transformer la grammaire G_2 pour la rendre LL(1), calculer la table de transitions LL(1) de G_2.
5. En suivant la table de transitions calculée à la question précédente, donner les étapes de l'analyse descendante de la phrase :

Hélène frappe Bastien qui dérange Yannick.

2 Analyse lexicale (10 pts)

Les outils d'analyse lexicale tels que `ocamllex` permettent de repérer automatiquement pour une unité lexicale sa position par rapport au début du fichier analysé. Cette position est repérée par le nombre de caractères depuis le début du fichier.

Plus précisément, la fonction `Lexing.lexeme_start` (resp. `Lexing.lexeme_end`) de type `lexbuf` $\rightarrow$ `int` permet dans une action de retrouver le numéro du caractère du début (resp. du caractère qui suit la fin) de l'unité lexicale reconnue par l'expression régulière associée.

Cette information est particulièrement utile pour rapporter une erreur. Cependant, il est usuel d'afficher la position de l'erreur en indiquant le *numéro de la ligne* où s'est produite l'erreur ainsi que le *numéro de colonne* c'est-à-dire le nombre de caractères depuis le début de la ligne.

Le but de l'exercice est d'étudier deux méthodes différentes de calcul des numéros de ligne et de colonne à partir de l'information de la position d'un caractère.

On se donne un analyseur simplifié dans lequel on ne reconnaît que des identificateurs, des commentaires non emboités (délimités par `(*` et `*)`) et des chaînes de caractères (délimitées par `"` et qui ne contiennent pas le caractère `"`).

Cet analyseur utilise des fonctions pour manipuler un buffer afin de construire une chaîne de caractères de taille quelconque. La spécification de ces fonctions est donnée dans la table suivante :

Nom	Type	Spécification
`contents_buffer`	`unit` $\rightarrow$ `string`	`contents_buffer ()` calcule la chaîne stockée dans le buffer
`add_buffer`	`string` $\rightarrow$ `unit`	`add_buffer` s ajoute la chaîne s à la fin du buffer
`reset_buffer`	`unit` $\rightarrow$ `unit`	`reset_buffer ()` remet le buffer à zéro

Le squelette de notre analyseur lexical est :

```
{   type token = Ident of string | String of string | Eof
    exception LexError of string
}
let chiffre = ['0'-'9']
let lettre  = ['a'-'z' 'A'-'Z']
let ident = lettre (lettre | chiffre)*
rule token = parse
    [' ' '\t' '\n'] {token lexbuf}
  | ident           {Ident(Lexing.lexeme lexbuf)}
  | '"'             {chaine lexbuf}
  | "(*"            {comment lexbuf; token lexbuf}
  | eof             {Eof}
and chaine = parse
    '"'             {let s = contents_buffer() in reset_buffer(); String(s)}
  | _               {add_buffer (Lexing.lexeme lexbuf); chaine lexbuf}
  | eof             {raise (LexError "chaîne non fermée")}
and comment = parse
    "*)"            {()}
  | _               {comment lexbuf}
  | eof             {raise (LexError "commentaire non fermé")}
```

1. Modifier l'analyseur lexical afin que l'erreur levée lorsque l'on rencontre la fin du fichier dans une chaîne ou un commentaire contienne également le numéro de caractère du début de la chaîne ou du commentaire qui n'a pas été fermé.
   Pour cela on modifie le type de l'exception `LexError` qui devient :

   ```
   exception LexError of string * int
   ```

2. On suppose que l'analyse lexicale, syntaxique ou sémantique a détecté une erreur au n-ème caractère d'un fichier d'entrée. On veut construire une fonction `locate_error` qui étant donné un nom de fichier f et un numéro de caractère n permet de calculer les numéros de ligne et de colonne correspondant au n-ème caractère dans f. Par convention, le premier caractère du fichier a pour numéro 0; le premier caractère de chaque ligne est dans la colonne 0 et la première ligne du fichier a pour numéro 1.
   (a) Proposer une version de `locate_error` construite à l'aide d'un programme `ocamllex`.
   (b) Ecrire une version de `locate_error` qui n'utilise pas d'analyseur lexical mais simplement la fonction `input_char` de type `in_channel` $\rightarrow$ `char`.

3. On souhaite construire une solution qui évite d'analyser une seconde fois un fichier lorsqu'une erreur se produit. Pour cela, on stocke pour chaque nouvelle ligne rencontrée la position absolue dans le fichier du premier caractère de cette ligne associée au numéro de la ligne. Lorsqu'une erreur se produit au caractère n, il suffit alors de retrouver la position du dernier début de ligne avant n (c'est celui dont le numéro est le plus grand parmi les débuts de ligne inférieur à n). Par exemple, dans le fichier :

   ```
   ident
   "chaine"
   (* commentaire
   sur plusieurs lignes *)
   ```

les débuts de ligne sont placés aux caractères : $0, 6, 15$ et 30. La table des débuts de ligne comportera les éléments suivants : $(0,1)(6,2)(15,3)(30,4)$.
Pour retrouver la ligne et la colonne correspondant au caractère de position 25, il suffit de retrouver l'entrée qui a le numéro le plus grand parmi les entrées de numéro inférieur à 25. Dans l'exemple, il s'agit de l'entrée $(15,3)$. Le caractère est donc sur la troisième ligne et le numéro de colonne est $25 - 15 = 10$.

On suppose que l'on dispose d'une table permettant de stocker des couples (n, l) avec n un numéro de caractère et l le numéro de ligne correspondant. On dispose de deux opérations `add` et `max_inf` pour manipuler cette table. Les types et spécifications de ces opérations sont données dans le tableau suivant :

`table_add`	`int` $\rightarrow$ `int` $\rightarrow$ `unit`	`table_add` n l ajoute le couple (n, l) à la table
`max_inf`	`int` $\rightarrow$ `int` $\times$ `int`	`max_inf` n calcule le couple (m, l) tel que m est maximum parmi les couples (m', l') de la table tels que $m' \leq m$

(a) Modifier l'analyseur lexical du langage pour construire la table des débuts de ligne.

(b) Proposer une nouvelle implantation de la fonction `locate_error` utilisant cette table.

4. On se propose d'implanter la structure de la table des débuts de ligne en utilisant une référence sur un arbre binaire de recherche équilibré contenant les couples (n, l) et ordonné suivant la relation $(n, l) < (n', l') \Leftrightarrow n < n'$. On suppose que le type α `btree` implante un type polymorphe d'arbres binaires de recherche stockant aux nœuds des valeurs de type α et que l'on dispose des opérations de base sur ce type spécifiées dans la table suivante :

`bt_add`	$\alpha \rightarrow \alpha$ `btree` $\rightarrow \alpha$ `btree`	`bt_add` v t est un arbre binaire de recherche équilibré contenant toutes les valeurs de l'arbre t plus la valeur v
`bt_empty`	α `btree` $\rightarrow$ `bool`	`bt_empty` t renvoie vrai si t correspond à un arbre vide
`bt_root`	α `btree` $\rightarrow \alpha$	`bt_root` t renvoie la valeur stockée à la racine de t si t n'est pas vide
`bt_left`	α `btree` $\rightarrow \alpha$ `btree`	`bt_left` t renvoie le sous arbre gauche de l'arbre t si t n'est pas vide
`bt_right`	α `btree` $\rightarrow \alpha$ `btree`	`bt_right` t renvoie le sous arbre droit de l'arbre t si t n'est pas vide

À l'aide du type `btree` et des opérations primitives sur ce type, proposer une implantation de la table des débuts de ligne et un codage des opérations `table_add` et `max_inf`.

5. Estimer la complexité de l'opération `locate_error` en fonction du nombre de lignes du fichier d'entrée lorsque le table des débuts de ligne est implantée par un arbre binaire de recherche équilibré.