Master Informatique, M1 — 2013–2014
Introduction à la compilation — Durée : 2h
2 pages

Examen du 24 octobre 2013

Les notes de TD manuscrites ainsi que les transparents de cours et le polycopié de cette année sont les seuls documents autorisés. Veuillez lire attentivement les questions. Veuillez rédiger proprement, clairement et de manière concise et rigoureuse.

1 Questions de cours

Question 1 Donner l'automate correspondant à l'expression régulière $a(a|b)^+b(ab)^*$, où a et b sont les seules lettres de l'alphabet

Question 2 Écrire un programme en assembleur MIPS qui évalue les trois instructions suivantes, où les variables globales `x` et `y` seront allouées dans le segment de données, et les variables locales sur la pile.

```
let x = if 1<>2 then 10 else 12
let y = let z = x + 3 * x in z + 1
print (y)
```

2 Analyse lexicale

Nous nous intéressons dans cet exercice à des sapins de noël dessinés à l'aide de suites de longueur *paire* de symboles `*` comme par exemple :

```
     **
    ****
   ******
  ********
    ****
  ********
 **********
************
     **
     **
```

Un sapin sera représenté en machine par une liste d'entiers (*pairs*), c'est-à-dire une valeur de type `int list`. Chaque entier correspondra au nombre de symboles `*` contenu sur une ligne du sapin. Par convention, le premier élément de la liste sera associé à la première ligne du sapin, le deuxième entier à la deuxième ligne etc. Ainsi, le sapin précédent sera représenté par la liste `[2;4;6;8;4;8;10;12;2;2]`.

On souhaite réaliser un analyseur lexical qui construise une liste d'entiers pairs à partir d'un fichier contenant la description d'un sapin. Les fichiers d'entrée acceptés par notre analyseur (c'est-à-dire ceux reconnus syntaxiquement corrects) devront respecter la syntaxe suivante :

1. Chaque ligne ne devra être composée que de symboles `*` et d'espaces (`'\t'` ou `' '`) et sera terminée par un retour chariot (`'\n'`) ;
2. Une seule suite de symboles `*` sera autorisée par ligne ;
3. Ces suites de symboles `*` devront être de longueur paire.

Aucune contrainte n'est imposée sur l'indentation des suites de symboles `*` dans le fichier. Par exemple, le fichier composé des lignes suivantes devra être reconnu comme syntaxiquement correct par notre analyseur lexical, et la liste d'entiers `[2;4;6;8;4;0;8;10;12;2;2]` devra être renvoyée.

```
**
  ****
     ******
********
  ****

    ********
**********
              ************
**
     **
```

Question 3 Compléter l'analyseur lexical donné ci-dessous aux emplacements indiquez par les commentaires `(* A compléter *)` afin qu'il retourne la liste d'entiers associée à un fichier. L'analyseur devra lever l'exception `Erreur_de_syntaxe` en cas de problème.

```
{
  exception Erreur_de_syntaxe
}

(* A completer *)

rule sapin = parse

 (* A completer *)

and (* A completer *)

{
  let sapin file =
    let f = open_in file in
    let s = sapin (from_channel f) in close_in f; s
}
```

3 Analyse descendante

On considère la grammaire suivante, où l'ensemble des caractères terminaux est $\{\texttt{=>}, \vee, \wedge, \texttt{(}, \texttt{)}, \texttt{p}\}$ et l'ensemble des caractères non-terminaux est $\{I, I', D, D', C, C', E\}$.

$$\begin{array}{lcl@{\qquad}lcl} I & \to & D\,I' & I' & \to & \epsilon \mid \texttt{=>}\ I \\ D & \to & C\,D' & D' & \to & \epsilon \mid \vee\ D \\ C & \to & E\,C' & C' & \to & \epsilon \mid \wedge\ C \\ E & \to & \texttt{p} \mid \texttt{(}\ I\ \texttt{)} & & & \end{array}$$

Question 4 Quel est le langage engendré par cette grammaire ?

Question 5 Calculer les ensembles Null, First et Follow pour les non-terminaux. Penser à ajouter le symbole `#` dans les suivants de I.

Question 6 En utilisant les ensembles précédents, construire la table d'expansion de l'analyse descendante pour cette grammaire.

Question 7 Cette grammaire est-elle LL(1) ? Si non, illustrer avec un contre-exemple.

Question 8 À l'aide de la table précédente, écrire un analyseur descendant en OCaml en introduisant une fonction de type `token list -> token list` pour chaque non-terminal. On pourra supposer que le type `token` est défini par

```
type token = Fl | Ou | Et | Lp | Rp | Id | EOF
```

où les constructeurs `Fl`, `Ou`, `Et`, `Lp`, `Rp` et `Id` représentent respectivement les terminaux `=>`, $\vee$, $\wedge$, `(`, `)` et `p`. Le constructeur `EOF` représente `#`.