Master Informatique, M1 — 2012–2013
Introduction à la compilation — Durée : 2h
2 pages

Examen du 24 octobre 2012

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 Analyse lexicale : la méthode des résidus

Dans cet exercice, on s'intéresse à une méthode pour déterminer si un mot appartient au langage $L(r)$ d'une expression régulière r. Il s'agit d'un algorithme efficace appelé *méthode des résidus*. Les expressions régulières que l'on considère sont définies par la grammaire suivante :

$$r := \emptyset \mid \epsilon \mid \mathtt{c} \mid r_1 r_2 \mid r_1 \mid r_2 \mid r^*$$

où `c` représente les caractères.

On définit le *résidu* d'une expression régulière r par rapport à un caractère `c` de la manière suivante :

$$Res(r, \mathtt{c}) = \{\ w \mid \mathtt{c}w \in L(r)\ \}$$

Le langage $Res(r, \mathtt{c})$ peut lui-même être décrit par une expression régulière.

Questions 1. Calculer les expressions régulières correspondant aux résidus suivants :

1. $Res(\mathtt{abc}, \mathtt{a})$
2. $Res(\mathtt{ab}|\mathtt{c}, \mathtt{a})$
3. $Res(\mathtt{ab}|\mathtt{c}, \mathtt{b})$
4. $Res(\mathtt{a}^*, \mathtt{a})$
5. $Res(\mathtt{a}^*\mathtt{b}, \mathtt{b})$

On considère le type OCaml suivant pour les expressions régulières.

```
type  expreg =
      | Vide                          (* Langage vide ∅ *)
      | Epsilon                       (* Mot vide ϵ *)
      | Caractere  of char            (* Caractère c *)
      | Union      of expreg * expreg (* r1|r2 *)
      | Produit    of expreg * expreg (* r1r2 *)
      | Etoile     of expreg          (* r* *)
```

Question 2. Écrire une fonction `contient_epsilon : expreg -> bool` qui détermine si une expression régulière r contient ϵ.

Question 3. Écrire une fonction `residu` : `expreg` → `char` → `expreg` calculant à partir de r et de c une expression régulière r' telle que $L(r') = Res(r, \mathtt{c})$.

Indication : le cas difficile est celui du produit $r_1 r_2$, où il faut distinguer deux cas selon le résultat de `contient_epsilon` r_1.

Question 4. On représente un mot par une liste de caractères de type `char list`. En utilisant les deux fonctions précédentes, écrire une fonction `reconnait : expreg -> char list -> bool` qui détermine si un mot appartient au langage d'une expression régulière. On procédera récursivement sur la liste.

2 Analyse syntaxique

On s'intéresse dans cet exercice à la construction `if-then-else` de certains langages de programmation où on peut écrire des expressions comme

```
if b then a
if b then a else a
if b then if b then a else a
```

où `a` représente une instruction (affectation par exemple) et `b` une expression booléenne.

Pour abstraire les expressions `if-then-else`, on se donne la grammaire formelle suivante avec comme ensemble de terminaux $\{\mathtt{i}, \mathtt{t}, \mathtt{a}, \mathtt{e}, \mathtt{b}\}$,

$$\begin{array}{lcl} S & \to & \mathtt{i}E\mathtt{t}S \mid \mathtt{i}E\mathtt{t}S\mathtt{e}S \mid \mathtt{a} \\ E & \to & \mathtt{b} \end{array}$$

où, `i`, `t` et `e` jouent le rôle de `if`, `then` et `else`, S celui des instructions et E celui des expressions booléennes.

Question 1. Donner les étapes de l'analyse ascendante du mot `ibtibtaea` (en utilisant la présentation à trois colonnes du cours : pile, entrée, action).

Afin de réaliser une analyse *descendante*, on se donne une version modifiée de cette grammaire.

$$\begin{array}{lcl} S & \to & \mathtt{i}E\mathtt{t}SS' \mid \mathtt{a} \\ S' & \to & \mathtt{e}S \mid \epsilon \\ E & \to & \mathtt{b} \end{array}$$

Question 2. Calculer les ensembles Null, First et Follow pour les non-terminaux S, S' et E. Penser à ajouter le symbole `#` dans les suivants de S.

Question 3. En utilisant les ensembles précédents, construire la table d'expansion de l'analyse descendante pour cette grammaire.

Question 4. Cette grammaire est-elle LL(1) ? Si non, illustrer avec un contre-exemple.

Question 5. 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 (cf. cours). Cet analyseur choisira d'associer le `else` avec le `then` le plus proche. On pourra supposer que le type `token` est défini par

```
type token = I | T | A | E | B | EOF
```

où `EOF` représente `#`.