Rapport de projet

paul.patault@universite-paris-saclay.fr

2. Validation top-down non-déterministe

Question 1

Un run d'un automate d'arbre $A = (Q, \delta, I, F, \Sigma)$ pour un arbre $t \in \mathcal{T}(\Sigma)$ est une fonction $r : dom(t) \to Q$ telle que $\forall p \in dom(t), (t(p), r(p), r(p1), r(p2)) \in \delta$. Un run est dit acceptant si et seulement si $r(\epsilon) \in I$.

Question 2

```
let rec validate_td a t p q =
    if label (t p) = '#' then
        true
    else
        let transition = label (t p), q in
        let l = List.filter ((=) transition) a.delta in (* modifier ici *)
        List.fold (fun acc q' ->
            acc ||
            (validate_td a t (p@[first_child (t p)]) q' &&
             validate_td a t (p@[next_sibling (t p)]) q')
        ) false l
```

Question 3

La complexité de l'expression

$$\exists\ \texttt{q} \in I \text{ tel que } \texttt{validate\_td a t eps q}$$

est $O(|\texttt{a}|^{|\texttt{t}|})$, où $|\texttt{a}|$ est le nombre de transitions de l'automate `a`. En effet, l'algorithme nous fait prendre au pire $|\texttt{a}|$ fois chaque arête de l'arbre `t`.

3. Validation bottom-up

Question 1

```
let rec validate_bu a t p =
    let lab = label (t p) in
    if lab = '#' then
        List.filter ((=) ('#',[],[]) a.trans in
    else
        let left = validate_bu a t (p@[first_child (t p)]) in
        let right = validate_bu a t (p@[next_sibling (t p)]) in
        let res = ref [] in
        List.iter (fun r -> List.iter (fun l ->
            let trans = lab, l, r in
            let possible_states = List.filter ((=) trans) a.trans in
            res <- possible_states :: !res;
        ) left) right
        res
```

Question 2

La complexité de l'expression

$$(\texttt{validate\_bu a t eps}) \cap I$$

est $O(|\texttt{t}|)$.

Question 3

4. Compilation

Question 1

Question 2