TP 19

ANALYSE SYNTAXIQUE DESCENDANTE

1 Préliminaires de programmation

On définit le type suivant pour gérer une chaîne de caractères comme un flux de caractères :

```
type stream = {
  str : string;
  mutable index : int
}
```

Le champ `index` indique l'indice du prochain caractère qui sera lu, et vaut donc initialement zéro :

```
let new_state str =
  {str = str; index = 0}
```

On définit une exception `SyntaxError`, et l'on fournit les deux fonctions suivantes :

```
exception SyntaxError

let peek s =
  if s.index < String.length s.str then Some s.str.[s.index]
  else None

let error s =
  match peek s with
  | None ->
    printf "Unexpected end of input\n";
    raise SyntaxError
  | Some c ->
    printf "Unexpected token %c at position %d\n" c s.index;
    raise SyntaxError
```

▸ **Question 1** Écrire une fonction `expect` prenant en entrée un `stream` `s` et un `char` `c` et ayant le comportement suivant :

- si le prochain caractère du flux existe et vaut `c`, alors la fonction passe au caractère suivant et ne renvoie rien ;
- sinon, elle affiche un message d'erreur comme ci-dessous puis lève l'exception `SyntaxError`.

```
# let s = new_stream "a";;
val s : stream = {str = "a"; index =
↪  0}
# expect s 'b';;
Expected b
Unexpected token a at position 0
Exception: SyntaxError.
```

```
# s.index <- 1;;
- : unit = ()
# expect s 'a';;
Expected a
Unexpected end of input
Exception: SyntaxError
```

```
val expect : stream -> char -> unit
```

▸ **Question 2** Écrire une fonction `discard` qui prend en entrée un `stream` et passe au caractère suivant (en ignorant le caractère actuel). On signalera l'erreur s'il n'y a pas de caractère actuel.

```
val discard : stream -> unit
```

▸ **Question 3** Écrire une fonction `is_letter` prenant en entrée un caractère et renvoyant un booléen indiquant si ce caractère est dans l'ensemble $\{a, \dots, z, A, \dots, Z\}$.
Remarque
On utilisera `int_of_char` et l'on tirera parti du fait que les codes ASCII des lettres sont consécutifs.

```
val is_letter : char -> bool
```

2 Expressions régulières

On note Σ_0 l'ensemble $\{a, \dots, z, A, \dots, Z\}$ des lettres ASCII minuscules et majuscules, et $\Sigma = \Sigma_0 \cup \{(,), ., *, ?, +\}$. On considère la grammaire G_0 suivante pour les expressions régulières, où l'on note l'alternative $+$ au lieu de $|$ pour éviter la confusion.

$$E \to E + E \mid EE \mid (E) \mid E* \mid E? \mid . \mid a \in \Sigma_0$$

Figure 19.1 – La grammaire G_0

▸ **Question 4** Montrer que la grammaire G_0 est ambiguë.

2.1 Version avec parenthésage complet

Pour lever l'ambiguïté, on propose la grammaire G_1 suivante (où R est le symbole initial) :

$$R \to P + P \mid PP \mid P* \mid P? \mid . \mid a \in \Sigma_0$$
$$P \to (R)$$

Figure 19.2 – La gramamaire G_1

▸ **Question 5** Pour chacun des mots suivants, indiquer s'il appartient à $\mathcal{L}(G_1)$:

1. `(a)((b)*)`
2. `ab`
3. `(.)`
4. `a+b`
5. `(a+b)`
6. `(a)+(b)`
7. `((a)+(b))`

▸ **Question 6** Justifier rapidement que cette grammaire n'est pas ambiguë. On ne demande pas ici une preuve formelle mais juste l'intuition du raisonnement.
Remarque
La partie 2.4 fournit une preuve rigoureuse de cette propriété. Elle est **à traiter chez vous** (sauf si vous avez fini le reste du sujet), mais **à traiter absolument** : ce type de preuve a de fortes chances de vous être demandé à un moment ou à un autre.

On se dote du type suivant pour représenter une expression régulière :

```
type regex_ast =
  | Sum of regex_ast * regex_ast
  | Concat of regex_ast * regex_ast
  | Char of char
  | Star of regex_ast
  | Maybe of regex_ast
  | Any
```

▸ **Question 7** Comment passe-t-on d'un arbre de dérivation pour la grammaire G_1 à un arbre de type `regex_ast` (dit aussi *arbre de syntaxe abstraite*, ou AST) ?

Remarque

Dans la suite, on générera directement l'AST sans passer par l'arbre de dérivation.

▸ **Question 8** Écrire deux fonctions d'analyse syntaxique mutuellement récursives `regex` et `paren` correspondant respectivement aux variables R et P de la grammaire G_1.

```
val regex : stream -> regex_ast
```

▸ **Question 9** Écrire une fonction `parse_regex` prenant en entrée une expression régulière sous forme d'une chaîne de caractères et renvoyant l'arbre de syntaxe abstraite correspondant. On lèvera une exception si la chaîne n'est pas dans le langage de la grammaire G_1.

```
utop[241]> parse_regex "(.)+(((a)(b))*)";;
- : regex_ast = Sum (Any, Star (Concat (Char 'a', Char 'b')))
utop[242]> parse_regex "ab";;
Expected input to end
Unexpected token b at position 1
Exception: SyntaxError.
```

```
val parse_regex : string -> regex_ast
```

2.2 Version avec priorités et associativité

▸ **Question 10** Proposer une grammaire G_2, non ambiguë, engendrant exactement les mots de $\mathcal{L}(G_0)$ ne contenant ni * ni ?. On demande bien sûr que les ambiguïtés qui existaient dans G_0 soient résolues en accord avec les règles usuelles de priorité.

▸ **Question 11** Quel est l'arbre de dérivation de `a+b+c` dans votre grammaire ? Proposer une grammaire donnant l'autre arbre possible.

▸ **Question 12** Écrire un analyseur syntaxique en descente récursive pour votre grammaire. Cet analyseur devrait logiquement utiliser trois fonctions mutuellement récursives de type `state -> regex_ast` (plus une fonction externe).

```
val parse_regex_2 : string -> regex_ast
```

▸ **Question 13** Proposer une grammaire G_3, modification de la grammaire G_2, faiblement équivalente à G_0 mais toujours non ambiguë.

▸ **Question 14** Donner l'arbre de dérivation de `(ab)**+c` pour G_3.

▶ **Question 15** Écrire un analyseur syntaxique pour la grammaire G_2.

```
val parse_regex_3 : string -> regex_ast
```

2.3 Traduction en postfixe

Nous avons écrit (TP **??**) un clone très simplifié de `grep` qui accepte en entrée une expression régulière en notation postfixe (et un nom de fichier). Le but de cette partie est d'écrire un petit programme OCaml faisant exactement le même travail mais acceptant une expression régulière en notation infixe. L'idée n'est pas de tout reprogrammer, mais simplement de faire la conversion puis d'appeler le programme C.

On rappelle la syntaxe utilisée par notre pseudo-`grep` :

- l'alternative est notée `|` ;
- la concaténation est explicitement notée `@` ;
- les autres caractères spéciaux (`'.'`, `'*'` et `'?'`) ont leur rôle habituel.

▶ **Question 16** Écrire une fonction `postfix` qui prend en entrée un `regex_ast` et renvoie la chaîne de caractères correspondant à sa représentation postfixe comme définie ci-dessus.

```
val postfix : regex_ast -> string
```

▶ **Question 17** La fonction `Sys.command` exécute une ligne de commande (passée sous forme de chaîne de caractères). Écrire un programme OCaml ayant le comportement décrit en introduction de cette partie.

2.4 Preuve de non-ambiguïté

Soit $\Sigma_p = \Sigma \sqcup \{p_o, p_f\}$ un alphabet (on note p_o et p_f les parenthèses ouvrantes et fermantes par souci de clarté).

- Pour un mot $u \in \Sigma_p^*$, on définit $\delta(u) = |u|_{p_o} - |u|_{p_f}$.
- On dit que v est un *préfixe propre* de u si v est un préfixe de u différent de ε et de u.
- On dit que u est *entièrement parenthésé* si :
  - $\delta(u) = 0$;
  - pour tout préfixe propre v de u, $\delta(v) > 0$.
- On dit que u est *bien parenthésé* s'il vérifie les conditions suivantes :
  - $\delta(u) = 0$;
  - pour tout préfixe v de u, $\delta(v) \geqslant 0$.

▶ **Question 18** Montrer qu'un mot w a au plus un préfixe non vide entièrement parenthésé.

▶ **Question 19** On note $\mathcal{L}_{G_1}(P)$ le langage engendré par la grammaire ayant les mêmes règles que G_1 mais P comme symbole initial. Montrer (rigoureusement) que tout mot de $\mathcal{L}_{G_1}(P)$ est entièrement parenthésé, et que tout mot de $\mathcal{L}(G_1)$ est bien parenthésé.

▶ **Question 20** Montrer (rigoureusement) que la grammaire G_1 n'est pas ambiguë.

3 Expressions arithmétiques

On fournit une série de fichiers, contenant chacun un certain nombre d'expressions arithmétiques (une par ligne), suivant des grammaires plus ou moins compliquées. On donne ici pour chaque fichier la grammaire la plus simple engendrant le langage, qui n'est bien sûr pas utilisable telle quelle pour l'analyse syntaxique : il faudra « désambiguer » en utilisant les règles de priorité et associativité usuelles.

$$\begin{aligned} E &\to N \mid (E) \mid E + E \mid E \times E \\ N &\to 0D \mid \cdots \mid 9D \\ D &\to 0D \mid \cdots \mid 9D \mid \varepsilon \end{aligned}$$

Figure 19.3 – Fichier `plus-fois.txt`

$$\begin{aligned} E &\to N \mid (E) \mid E + E \mid E \times E \mid E - E \\ N &\to 0D \mid \cdots \mid 9D \\ D &\to 0D \mid \cdots \mid 9D \mid \varepsilon \end{aligned}$$

Figure 19.4 – Fichier `plus-fois-moins.txt`

$$\begin{aligned} E &\to N \mid (E) \mid E + E \mid E \times E \mid E - E \mid E! \\ N &\to 0D \mid \cdots \mid 9D \\ D &\to 0D \mid \cdots \mid 9D \mid \varepsilon \end{aligned}$$

Figure 19.5 – Fichier `plus-fois-moins-fact.txt`

$$\begin{aligned} E &\to N \mid (E) \mid E + E \mid E \times E \mid E - E \mid E! \mid -E \\ N &\to 0D \mid \cdots \mid 9D \\ D &\to 0D \mid \cdots \mid 9D \mid \varepsilon \end{aligned}$$

Figure 19.6 – Fichier `plus-fois-moins-fact-moins-unaire.txt`

▸ **Question 21** Pour chacun de ces fichiers, calculer la somme des valeurs des expressions qui y sont présentes. Il y aura des dépassements de capacité mais on s'intéresse à la somme modulo 2^{63} ramenée entre -2^{62} et $2^{62} - 1$ (autrement dit, on fera les calculs sur des `int` OCaml sans se soucier des dépassements). On garantit que les factorielles seront toujours appliquées à des arguments raisonnables (positifs et pas trop grands).

Solutions

▸ **Question 1**

```
let expect s c =
  match peek s with
  | Some c' when c = c' ->
    s.index <- s.index + 1
  | _ -> printf "Expected %c\n" c; error s
```

▸ **Question 2**

```
let discard s =
  if s.index = String.length s.str then error s
  else s.index <- s.index + 1
```

▸ **Question 3**

```
let is_letter c =
  let i = int_of_char c in
  (int_of_char 'a' <= i && i <= int_of_char 'z')
  ||
  (int_of_char 'A' <= i && i <= int_of_char 'Z')
```

▸ **Question 4** On a deux dérivations gauches distinctes (et donc deux arbres de dérivation distincts) pour $a + b + c$:

- $E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow a + E + E \Rightarrow a + b + E \Rightarrow a + b + c$
- $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E + E \Rightarrow a + b + E \Rightarrow a + b + c$

▸ **Question 5**

1. Oui : $R \Rightarrow PP \Rightarrow (R)P \Rightarrow (a)P \Rightarrow (a)(R) \Rightarrow (a)(P*) \Rightarrow (a)((R)*) \Rightarrow (a)((b)*)$
2. Non.
3. Non.
4. Non.
5. Non.
6. Oui : $R \Rightarrow P + P \Rightarrow (R) + P \Rightarrow (a) + P \Rightarrow (a) + (R) \Rightarrow (a) + (b)$
7. Non.

▸ **Question 6** Si le mot est de longueur 1, c'est forcément un point ou une lettre, avec une unique dérivation. Sinon, il commence par une parenthèse ouvrante, qui est uniquement appariée à une parenthèse fermante qui lui correspond. On a alors $u = (v)w$, avec $v \in \mathcal{L}_{G_1}(R)$ et w qui commence soit par $+$, soit par $*$, soit par ?, soit par une parenthèse ouvrante. Ce caractère permet de déterminer la règle à la racine de l'arbre de dérivation, et ensuite on conclut par récurrence sur la longueur du mot (ou la hauteur de l'arbre de dérivation).

▸ **Question 7** Pour passer d'un arbre de dérivation à un AST :

- on efface les feuilles étiquetées par des parenthèses ;
- on fait remonter l'étiquette des feuilles $+$, $*$ et ? sur leur père, puis on efface ces feuilles.

▶ **Question 8**

```
let rec regex s =
  match peek s with
  | Some '(' ->
    begin
      let p = paren s in
      match peek s with
      | Some '+' ->
        discard s;
        let p' = paren s in
        Sum (p, p')
      | Some '*' -> discard s; Star p
      | Some '?' -> discard s; Maybe p
      | Some '(' ->
        let p' = paren s in
        Concat (p, p')
      | _ -> error s
    end
  | Some '.' -> discard s; Any
  | Some c when is_letter c -> discard s; Char c
  | _ -> error s
and paren s =
  expect s '(';
  let r = regex s in
  expect s ')';
  r
```

▶ **Question 9** On écrit une fonction générique, on pourra la réutiliser plus loin :

```
let parse initial_symbol str =
  let s = new_stream str in
  let tree = initial_symbol s in
  match peek s with
  | None -> tree
  | Some c -> printf "Expected input to end\n"; error s

let parse_regex str = parse regex str
```

On vérifie sur les exemples vus plus haut :

```
# parse_regex "(a)((b)*)";;
- : regex_ast = Concat (Char 'a', Star (Char 'b'))
# parse_regex "ab";;
Expected input to end
Unexpected token b at position 1
Exception: SyntaxError.
# parse_regex "(.)";;
Unexpected end of input
Exception: SyntaxError.
# parse_regex "a+b";;
Expected input to end
Unexpected token + at position 1
Exception: SyntaxError.
```

```
# parse_regex "(a+b)";;
Expected )
Unexpected token + at position 2
Exception: SyntaxError.
# parse_regex "(a)+(b)";;
- : regex_ast = Sum (Char 'a', Char 'b')
# parse_regex "((a)+(b))";;
Unexpected end of input
Exception: SyntaxError.
```

► **Question 10** On peut prendre la grammaire suivante :

$$S \to T \mid T + S \quad T \to F \mid FT \quad F \to (S) \mid . \mid a \in \Sigma_0$$

► **Question 11** Cette grammaire donne l'arbre correspondant à $a + (b + c)$. On obtiendrait l'autre arbre avec la grammaire suivante :

$$S \to T \mid S + T \quad T \to F \mid FT \quad F \to (S) \mid . \mid a \in \Sigma_0$$

► **Question 12**

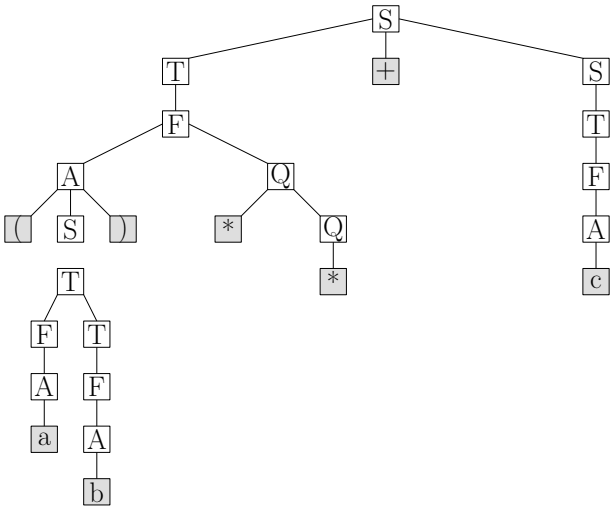
```
let rec regex_2 s =
  let t = term s in
  match peek s with
  | Some '+' ->
    discard s;
    let e = regex_2 s in
    Sum (t, e)
  | _ -> t
and term s =
  let f = factor s in
  match peek s with
  | None | Some ')' | Some '+' -> f
  | _ ->
    let t = term s in
    Concat (f, t)
and factor s =
  match peek s with
  | Some '(' ->
    discard s;
    let e = regex_2 s in
    expect s ')';
    e
  | Some c when is_letter c -> discard s; Char c
  | Some '.' -> discard s; Any
  | _ -> error s

let parse_regex_2 str = parse regex_2 str
```

► **Question 13** On peut prendre la grammaire suivante :

$$\begin{aligned} S &\to T \mid T + S \\ T &\to F \mid FT \\ F &\to A \mid AQ \\ Q &\to * \mid ? \mid *Q \mid ?Q \\ A &\to (S) \mid . \mid a \in \Sigma_0 \end{aligned}$$

▶ **Question 14**



▶ **Question 15**

```
let rec regex_3 s =
  let t = term s in
  match peek s with
  | Some '+' ->
    discard s;
    let e = regex_3 s in
    Sum (t, e)
  | _ -> t
and term s =
  let f = factor s in
  match peek s with
  | None | Some ')' | Some '+' -> f
  | _ ->
    let t = term s in
    Concat (f, t)
and factor s =
  let a = atom s in
  quantif s a
and quantif s a =
  match peek s with
  | Some '*' -> discard s; quantif s (Star a)
  | Some '?' -> discard s; quantif s (Maybe a)
  | _ -> a
and atom s =
  match peek s with
  | Some '(' ->
    discard s;
    let e = regex_3 s in
    expect s ')';
    e
  | Some c when is_letter c -> discard s; Char c
  | Some '.' -> discard s; Any
  | _ -> error s

let parse_regex_3 str = parse regex_3 str
```

▸ **Question 16** On utilise pour simplifier un objet de type `Bytes.t`, qui est une version mutable d'une chaîne (il s'agit donc moralement d'un tableau de `bytes`). On commence par calculer la taille de l'arbre, qui correspond exactement au nombre de caractères de la représentation postfixe et donc à la taille du « tableau » à créer.

```
let rec size = function
  | Char _ | Any -> 1
  | Star e | Maybe e -> 1 + size e
  | Concat (e, f) | Sum (e, f) -> 1 + size e + size f


let postfix expr =
  let n = size expr in
  let t = Bytes.make n '!' in
  let i = ref 0 in
  let put c =
    Bytes.set t !i c;
    incr i in
  let rec fill = function
    | Char c -> put c
    | Any -> put '.'
    | Star e -> fill e; put '*'
    | Maybe e -> fill e; put '?'
    | Concat (e, f) -> fill e; fill f; put '@'
    | Sum (e, f) -> fill e; fill f; put '|' in
  fill expr;
  String.of_bytes t
```

▸ **Question 17** On suppose que toutes les fonctions écrites jusqu'à présent ont été placées dans un fichier `recursive_descent.ml` situé dans le même répertoire que le fichier que nous créons. On suppose aussi que l'exécutable issu du programme C s'appelle `mygrep` et est également situé dans le répertoire courant.

```
open Recursive_descent
open Printf

let () =
  let regex_string = Sys.argv.(1) in
  let filename = Sys.argv.(2) in
  let ast = parse_regex_3 regex_string in
  let postfix_string = postfix ast in
  let cmd = Printf.sprintf "./mygrep '%s' %s" postfix_string filename in
  ignore (Sys.command cmd)
```

Remarque

On utilise `ignore (Sys.command cmd)` puisque la fonction `Sys.command` renvoie un entier (le code de retour de l'exécution de la commande dans le *shell*, donc normalement zéro si tout s'est bien passé) et que nous souhaitons ignorer cette valeur.

▸ **Question 18** Si u et v sont deux préfixes non vides de w, alors u est un préfixe de v ou v un préfixe de u : supposons u préfixe de v, sans perte de généralité. On a alors $\delta(u) = 0$ puisque u est bien parenthésé, donc u n'est pas un préfixe propre de v puisque v est entièrement parenthésé. Or u est non vide, donc $u = v$.

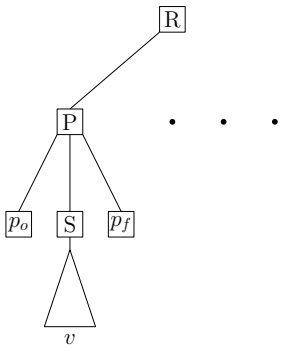
▸ **Question 19** On montre les deux propriétés simultanément par récurrence forte sur la longueur n du mot.

- Un mot de $L(R)$ de longueur $n \leqslant 1$ est réduit à un point ou une lettre, aucun mot de $L(P)$ n'est de longueur inférieure ou égale à 1.

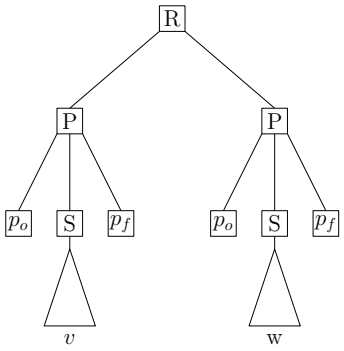
- Si $u \in L(R)$ est de longueur $n \geqslant 2$, on en considère une dérivation.
  - — Si elle commence par $R \Rightarrow P + P$, alors $u = v + w$ avec $v, w \in L(P)$. On peut leur appliquer l'hypothèse de récurrence, ils sont donc entièrement parenthésés et on en déduit immédiatement que u est bien parenthésé.
  - — On raisonne de même si la dérivation commence par $R \Rightarrow PP$, $R \Rightarrow P*$ ou $R \Rightarrow P?$.
- Si $v \in L(P)$ avec $|v| \geqslant 2$, une dérivation de v commence nécessairement par $P \Rightarrow p_o R p_f$, donc v est de la forme $p_o w p_f$ avec $w \in L(P)$. Par hypothèse de récurrence, w est bien parenthésé. Un préfixe propre de v est de la forme $p_o z$ avec z préfixe de w, donc $\delta(z) \geqslant 0$ et $\delta(p_o z) > 0$. Comme de plus $\delta(p_o w p_f) = \delta(w) = 0$, on a v entièrement parenthésé.

► **Question 20** On montre par récurrence sur n qu'aucun mot $u \in L(R)$ tel que $|u| \leqslant n$ n'est ambigu pour G.

- C'est évident si $n \leqslant 1$.
- Si $n > 2$, on considère un arbre de dérivation pour u. Quelle que soit la règle utilisée à la racine, le sous-arbre gauche doit donner un mot de $L(R)$, donc entièrement parenthésé. Or u a au plus (et donc exactement) un préfixe v entièrement parenthésé. Donc l'arbre est nécessairement de la forme suivante :



Le caractère qui suit v dans u est soit $*$, soit ?, soit $+$, soit p_o, et une seule règle est possible à la racine dans chaque cas. Si ce caractère est par exemple p_o, l'arbre est de la forme :



v est entièrement déterminé (préfixe entièrement parenthésé), donc w aussi. En leur appliquant l'hypothèse de récurrence, on obtient l'unicité de l'arbre. Les autres cas se traitent de même.