TD10: Les parseurs

1 Reconnaissance de langage

On rappelle le type abstrait de Flux, auquel on a ajouté également les traits monadiques. L'implantation est laissée libre (véritables flux, itérateurs, etc):

```
type 'a t ;;
val vide : 'a t ;;
val cons : 'a -> 'a t -> 'a t;;
val uncons : 'a t -> ('a * 'a t) option;;
val apply : ('a -> 'b) t -> ('a t -> 'b t);;
val unfold : ('b -> ('a * 'b) option) -> ('b -> 'a t);;
val filter : ('a -> bool) -> 'a t -> 'a t;;
(* monade additive des flux *)
val map : ('a -> 'b) -> 'a t -> 'b t
val ( >>=) : 'a t -> ('a -> 'b t) -> 'b t
(* return a = cons a vide *)
val return : 'a -> 'a t
(* zero = vide *)
val zero : 'a t
(* f1 ++ f2 = append f1 f2 *)
val ( ++ ) : 'a t -> 'a t -> 'a t
```

La reconnaissance de langage consiste à tester si une séquence/écriture donnée de symboles correspond à une phrase bien formée d'un langage. Cette reconnaissance syntaxique est le prélude à une analyse sémantique, qui consiste à interpréter cette phrase, lui donner un sens. On peut prendre pour exemple les langages informatiques.

Remarque : On ne va pas introduire formellement la notion de langage, ni les outils de description de ceux-ci. Il faudra donc se contenter d'exemples simples.

Les constructions dont on aura besoin pour décrire/reconnaître des langages sont les suivantes:

- les langages de base qui reconnaissent:
  - aucun flux
  - tous les flux
  - le flux vide
  - les flux commençant par un symbole donné.
- les combinaisons:
  - la séquence: une phrase commence par ceci, PUIS par cela.
  - le choix: une phrase est comme ceci OU comme cela.
  - l'option: une phrase peut contenir ceci ou non.
  - la répétition: (pour décrire des listes par exemple).

1.1 Les parsers

À la reconnaissance de langage est associée la notion de **parser**. Un parser est une fonction, qui prend un flux d'entrée et qui renvoie l'ensemble des flux résiduels, après consommation/filtrage des éléments du flux

initial conformes au langage. Ce flux d'entrée est unique pour toute l'application et on se contentera de lire ces éléments à l'aide de uncons, sans utiliser d'autres fonctionnalités. L'ensemble des flux résiduels peut: soit être vide, si le mot n'est pas reconnu par le parser; soit contenir un nombre fini (voire infini) de flux. On représente également cet ensemble de solutions, potentiellement très grand, par un flux dont on utilisera cette fois-ci les traits monadiques avec la sémantique "ensembliste" de NDET. Pour clarifier les notations, on appelle Flux l'implantation du flux de lecture et Solution l'implantation du flux des solutions. On peut utiliser la même implantation pour les deux flux.

On définit donc les types suivants:

```
type 'a result = 'a Flux.t Solution.t ;;
type 'a parser = 'a Flux.t -> 'a result ;;
```

1.2 Parsers élémentaires

On définit alors les parsers et opérations suivants:

- le parser nul: qui réussit toujours et ne consomme rien.

```
(* pnul : 'a parser *)
let pnul flux =
 Solution.return flux ;;
```

- le parser erreur: qui échoue toujours.

```
(* perreur : 'a parser *)
let perreur flux =
 Solution.zero ;;
```

- le parser vide: qui réussit uniquement si le flux est vide (et ne consomme rien).

```
(* pvide : 'a parser *)
let pvide flux =
 match Flux.uncons flux with
  | None   -> Solution.return flux
  | Some _ -> Solution.zero;;
```

- le test: qui réussit uniquement si l'élément de tête du flux satisfait un prédicat (et consomme cet élément).

```
(* ptest : ('a -> bool) -> 'a parser *)
let ptest p flux =
 match Flux.uncons flux with
  | None        -> Solution.zero
  | Some (t, q) -> if p t then Solution.return q else Solution.zero ;;
```

1.3 Combinaisons simples de parsers

▷ **Exercice 1** *Définir les opérations sur les parsers suivantes:*

- *la séquence: qui tente d'appliquer deux parsers à la suite sur un flux.*
- *le choix: qui applique un premier parser sur un flux puis un second parser sur ce même flux.*

2 Les parsers comme dénotations de langages

Le type suivant nous permettra de décrire simplement la catégorie de langages, dits langages **réguliers**, à laquelle on s'intéresse. On pourra se reporter à l'UE *Modélisation* en 1A-SN pour plus de détails:

```
type 'a language =
| Nothing (* langage vide *)
| Empty  (* langage reduit au mot vide *)
| Letter of 'a
| Sequence of 'a language * 'a language
| Choice of 'a language * 'a language
| Repeat of 'a language;;
```

On cherche à associer à chaque langage décrit par le type `'a language` un parser reconnaissant les flux de lettres appartenant à ce langage.

▷ **Exercice 2** *Définir les fonctions d'interprétations suivantes:*

- `eval: 'a language -> 'a parser`, *qui à tout langage fait correspondre un parser, qui reconnaît les flux appartenant à ce langage.*
- `belongs : 'a language -> 'a Flux.t -> bool`, *qui à tout langage et tout flux, teste si le flux appartient bien au langage.*

3 Parsing plus général

3.1 Langages plus généraux

Sans introduire de représentation "syntaxique" pour dénoter des langages plus généraux, comme des grammaires pour les langages non-contextuels par exemple, on peut tout de même décrire des parsers pour ces langages à l'aide de la récursivité, utilisée avec précaution. Considérons par exemple un langage d'expressions arithmétiques simples représenté par la grammaire suivante:

$Expr \rightarrow (Expr{+}Expr)$
$Expr \rightarrow$ `variable`

Le parser ci-dessous reconnaît ce langage. Les choix et séquences sont rangés dans des listes et parsés par des fonctions spéciales n-aires pour plus de lisibilité et d'extensibilité. Ici, une première couche lexicale serait fort utile pour regrouper des caractères en lexèmes et éliminer les espaces.

```
let pchoice_n l = List.fold_right pchoice l perreur;;
let psequence_n l = List.fold_right psequence l pnul;;
let paro = ptest ((=) '(');;
let parf = ptest ((=) ')');;
let plus = ptest ((=) '+');;
let var  = ptest (fun v -> v >= 'a' && v <= 'z');;
let rec expr flux =
 pchoice_n [psequence_n [var];
            psequence_n [paro; expr; plus; expr; parf]
 ]
 flux;;
```

Remarque : Avec l'introduction du paramètre flux à la définition, on a plus besoin de protéger explicitement les appels récursifs. OCaml sait que expr est une fonction, il n'évaluera rien, et en particulier pas les appels récursifs, sans un argument de type flux.

3.2 Parsing plus général

Les applications du *parsing* ne se limitent pas à la simple reconnaissance de mots mais y ajoutent également un traitement "sémantique", *a minima* et le plus souvent sous la forme de la construction d'un arbre syntaxique abstrait.

Dès lors, aux types associés au *parsing* est ajouté le type de la valeur construite 'b, ce qui donne:

```
type ('a, 'b) result = ('b * 'a Flux.t) Solution.t
type ('a, 'b) parser = 'a Flux.t -> ('a, 'b) result
```

Pour un type 'a fixé, on peut reconnaître une structure monadique additive avec les primitives suivantes:

```
let map : ('b -> 'c) -> ('a, 'b) parser -> ('a, 'c) parser =
  fun fmap parse f -> Solution.map (fun (b, f') -> (fmap b, f')) (parse f);;

let return : 'b -> ('a, 'b) parser = fun b f -> Solution.return (b, f);;

let (>>=) : ('a, 'b) parser -> ('b -> ('a, 'c) parser) -> ('a, 'c) parser =
  fun parse dep_parse f -> Solution.(parse f >>= fun (b, f') -> dep_parse b f');;

let zero : ('a, 'b) parser = fun f -> Solution.zero;;

let (++) : ('a, 'b) parser -> ('a, 'b) parser -> ('a, 'b) parser =
  fun parse1 parse2 f -> Solution.(parse1 f ++ parse2 f);;

let run : ('a, 'b) parser -> 'a Flux.t -> 'b Solution.t =
  fun parse f -> Solution.(map fst (filter (fun (b, f') -> Flux.uncons f' = None) (parse f)));;
```

▷ **Exercice 3** *En fonction du nouveau type* ('a, 'b) parser*, redéfinir si nécessaire les primitives* ptest*,* psequence*,* pchoice*,* pvide*,* perreur *et* pnul*, en commençant par déterminer leurs types.*

3.3 Application à la construction d'AST

▷ **Exercice 4** *Modifier le parser défini dans la section 3.1 afin de lui faire construire un arbre abstrait du type suivant:*

```
type ast = Plus of ast * ast | Var of char
```