TD10: Les parseurs

Thèmes et objectifs

- reconnaissance de langage
- utilisation des flux
- les parseurs

▷ **Support étudiant**

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.

Remarque : Si on choisissait de ne calculer que la première solution, on devrait gérér les retours en arrière: il suffit de penser au parser d'un langage de la forme $(\texttt{a}+\texttt{ab})\texttt{.c}$ confronté à l'entrée `abc`, la première branche `a` filtre un préfixe du mot, mais amène à un échec global, il faut alors revenir à la seconde branche `ab`, suivi de `c`. Bien sûr, il faut des continuations pour cela, et une bonne dose, ce qui entraîne une solution très/trop complexe. En choisissant de représenter les ensembles de solutions par des flux paresseux, on pourra en extraire simplement le premier élément. À noter enfin qu'on pourrait considérer des implantations différentes pour les deux usages des flux.

▷ **Support étudiant**

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.*

▷ ***Solution***

```
(* psequence : 'a parser -> 'a parser -> 'a parser *)
let psequence parser1 parser2 flux =
 Solution.(parser1 flux >>= parser2);;

(* pchoice : 'a parser -> 'a parser -> 'a parser *)
let pchoice parser1 parser2 flux =
 Solution.(parser1 flux ++ parser2 flux);;
```

▷ **Support étudiant**

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.*

▷ ***Solution***

La solution immédiate (et incorrecte) devrait ressembler à ceci, au dépliage de code près, notamment dans le cas Repeat *où on se contente de déplier la définition récursive de l'étoile de Kleene. La définition de* belongs *nécessite d'abord de filtrer les solutions à flux résiduel vide (un mot est reconnu s'il est consommé entièrement), puis de tester si cet ensemble est vide. Dans notre cas simple, c'est un overkill, puisqu'on calcule toutes les solutions qui sont au final équivalentes (seules les preuves de l'appartenance peuvent être multiples, mais ce n'est pas ce que l'algorithme produit).*

```
(* eval : 'a language -> 'a parser *)
let rec eval lang =
 match lang with
  | Nothing          -> perreur
  | Empty            -> pnul
  | Letter   a       -> ptest ((=) a)
  | Sequence (l1, l2) -> psequence (eval l1) (eval l2)
  | Choice   (l1, l2) -> pchoice (eval l1) (eval l2)
  | Repeat   l1      -> eval (Choice (Empty, Sequence (l1, Repeat l1)));;

(* belongs : 'a language -> 'a Flux.t -> bool *)
let belongs lang flux =
  Solution.uncons (Solution.filter (fun f' -> Flux.uncons f' = None) (eval lang flux)) <> None;;
```

On constate que l'évaluation de eval (Repeat l1) *va finir par appeler à nouveau* eval *sur le même argument. Comme* OCaml *est strict, l'évaluation va boucler avant d'avoir lu le moindre flux. Il faut protéger l'appel récursif dangereux avec une fonction explicite de* flux *qui retarde toute évaluation jusqu'à l'application d'un flux.*

```
let rec eval lang =
  match lang with
   | Nothing          -> perreur
   | Empty            -> pnul
   | Letter   a       -> ptest ((=) a)
   | Sequence (l1, l2) -> psequence (eval l1) (eval l2)
   | Choice   (l1, l2) -> pchoice (eval l1) (eval l2)
   | Repeat   l1      -> pchoice pnul (psequence (eval l1) (fun flux -> eval (Repeat l1) flux));;

(* belongs : 'a language -> 'a Flux.t -> bool *)
let belongs lang flux =
  Solution.uncons (Solution.filter (fun f' -> Flux.uncons f' = None) (eval lang flux)) <> None;;
```

▷ **Support étudiant**

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.

Remarque : Classiquement, la récursivité à gauche des grammaires poserait des problèmes de terminaison, et donc de reconnaissance de mot. Comme on cherche toutes les solutions, mais paresseusement, il serait *a minima* nécessaire de parser en premier choix les productions non-récursives à gauche pour obtenir quelques solutions. Toutefois, après avoir épuisé ces solutions, tous les choix restants aboutiront à des échecs. Malheureusement, les cas récursifs à gauche permettent d'engendrer une infinité de choix infructueux et on rentrera dans une boucle infinie, sans pouvoir produire la moindre solution suivante. La moralité est qu'il ne faut pas définir de telles grammaires lorsqu'on construit des parsers dits "descendants récursifs".

▷ **Support étudiant**

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.*

▷ ***Solution***

On pourra déterminer les types et les relations avec les primitives monadiques (le cas échéant) interactivement. Les parsers pvide pnul *ne renvoient rien et* ptest *renvoie l'élément testé. Enfin,* run *remplace* belongs*.*

```
(* perreur : ('a, 'b) parser *)
let perreur = fun f -> zero f;;
(* pnul : ('a, unit) parser *)
let pnul = fun f -> return () f;;
(* pvide : ('a, unit) parser *)
let pvide f =
 match Flux.uncons f with
  | None   -> Solution.return ((), f)
  | Some _ -> Solution.zero;;
(* ptest : ('a -> bool) -> ('a, 'a) parser *)
let ptest p f =
 match Flux.uncons f with
  | None          -> Solution.zero
  | Some (t, q) -> if p t then Solution.return (t, q) else Solution.zero;;
(* pchoice : ('a, 'b) parser -> ('a, 'b) parser -> ('a, 'b) parser *)
let pchoice p1 p2 = (p1 ++ p2);;
(* psequence : ('a, 'b) parser -> ('a, 'c) parser -> ('a, 'b * 'c) parser *)
let psequence parse1 parse2 =
 parse1 >>= fun b -> parse2 >>= fun c -> return (b, c);;
```

Une solution alternative pour la séquence peut être la suivante, qui permettrait de redéfinir psequence_n*:*

```
(* psequence : ('a, 'b list) parser -> ('a, 'b list) parser -> ('a, 'b list) parser *)
let psequence parse1 parse2 =
 parse1 f >>= fun b -> parse2 >>= fun c -> return (b@c);;
```

▷ **Support étudiant**

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
```

▷ ***Solution***

On peut utiliser psequence *et filtrer les tuples d'arguments produits, mais il est plus direct de récupérer ces arguments en utilisant* >>= *et d'insérer des constructeurs du type* ast *à la fin. Les parsers* paro, parf, plus *et* var *gardent la même définition.*

```
let rec expr flux =
  (
   var >>= fun c ->
   return (Var c)
   ++
   paro >>= fun _ -> expr >>= fun e1 -> plus >>= fun _ -> expr >>= fun e2 -> parf >>= fun _ ->
   return (Plus (e1, e2))
  ) flux ;;
```

Remarque: Les parsers présentés risquent de souffrir de problèmes d'efficacité, en plus de leur aversion mortelle pour la récursivité à gauche, en cas de grammaire fortement ambigüe, le nombre de tous les *parsing* possibles pouvant être exponentiel. On pourrait introduire, en cassant les lois monadiques, un combinateur first : ('a, 'b) parser -> ('a, 'b) parser, similaire au cut de Prolog, qui ne garde que la première solution indépendamment de la suite du *parsing*, mais cela comporte évidemment des risques. Plus raisonnablement, on pourrait *memoizer* les parsers, afin de ne pas ré-engendrer tous les résultats possibles lorsqu'un même parser examine une nouvelle fois un même flux d'entrée.