SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

MPIL(3I008)
Année 2017/2018
Séance 5 de TD/TME

Objectifs

1. Langage de modules.
2. Signatures.
3. Modules paramétrés (foncteurs).
4. Notions de structuration de programmes.

Travaux Dirigés

Exercice 1 : Texte formaté

On cherche à représenter des lignes de texte formaté. Une ligne est une suite de caractères formatés, un caractère formaté contenant le caractère brut et une liste de marques de formatage.

Pour cet exercice, on se limitera à prendre en charge quelques marques de formatage comme *gras*, *italique*, etc.

Q.1.1 Donnez les types `fmark`, `fchar` et `fline` représentant respectivement une marque de formatage, un caractère formatté et une ligne de texte formaté. Si besoin, définissez des types intermédiaires.

Q.1.2 Pour simplifier la gestion des listes de marqueurs, définissez les quelques fonctions utilitaires suivantes. Veillez à préférer l'utilisation des combinateurs prédéfinis du module `List`.

- `set_mark l a` (resp. `unset_mark`) prend un ensemble `l` de marqueurs et renvoie un nouvel ensemble dans lequel le marqueur `a` est (resp. n'est pas) présent et la présence des autres marqueurs est conservée.
- `present_mark l a` renvoie la présence du marqueur `a` dans `l`
- `change_mark l a` renvoie un nouvel ensemble ou la présence du marqueur `a` est inversée.

Exercice 2 : Processeur de texte formaté

Le but de cet exercice est de concevoir une architecture de traitement de texte, générique en la source et le format d'entrée d'une part et en la cible et le format de sortie d'autre part.

Le traitement principal fonctionne de la façon suivante :

1. la source et la cible sont initialisées, éventuellement avec des paramètres spécifiques (chemin pour une source fichier, adresse pour une source réseau, etc.).
2. le texte de la source est lu ligne par ligne, et chaque ligne est transmise à la cible.
3. lorsque la source a été lue en entier (déclenchement d'une exception spécifique `End`), la source et la cible doivent être fermées et le traitement termine.

Il est implanté par le foncteur suivant :

```
module Processor (Input  : INPUT)
                 (Output : OUTPUT) =
struct
  type input_param = Input.param
  type output_param = Output.param
  let process input_param output_param =
    let input = Input.new_input input_param in
    let output = Output.new_output output_param in
    try
      while true do
        let line = Input.input_line input in
        Output.output_line output line
      done
    with
      | Input.End ->
        Input.close_input input ;
        Output.close_output output
end
```

Q.2.1 Donnez une signature `INPUT` fonctionnant avec le code du foncteur `Processor`.

Q.2.2 Même question pour `OUTPUT`.

Q.2.3 On donne la signature `PROCESSOR` suivante.

```
module type PROCESSOR = sig
  type input_param
  type output_param
  val process : input_param -> output_param -> unit
end
```

Comment peut-on contraindre le résultat du foncteur `Processor` à vérifier signature ?

Q.2.4 Expliquez pourquoi ce n'est pas une bonne idée, et corrigez le programme pour qu'il vérifie que `Processor` est bien une instance correcte de `PROCESSOR`, tout en corrigeant le problème décelé.

Exercice 3 : Copieur verbatim de fichiers

Q.3.1 Donnez un module `VerbatimFileInput`, instance d'`INPUT`, lisant ligne par ligne un fichier, sans se soucier du formatage.

Q.3.2 Donnez un module `VerbatimFileOutput`, instance d'`OUTPUT`, imprimant ligne par ligne dans un fichier, en ignorant les informations de formatage.

Q.3.3 À l'aide des deux questions précédentes, créez un processeur spécialisé dans la duplication verbatim de fichiers.

Q.3.4 Lancez ce duplicateur de fichier fraîchement créé, en utilisant les arguments de la ligne de commande comme source et cible.

Exercice 4 : Processeur interactif Markdown→HTML

Q.4.1 Donnez un module `MarkdownConsoleInput`, instance d'`INPUT`, lisant ligne par ligne sur l'entrée clavier en interprétant une syntaxe de type MarkDown.

La syntaxe employée est la suivante :

- le texte entre deux caractères `*` doit être en gras
- le texte entre deux caractères `/` doit être en italique
- le texte entre deux caractères `_` doit être souligné
- etc.

Q.4.2 Donnez un module `HtmlConsoleOutput`, instance d'`OUTPUT`, imprimant ligne par ligne dans la console, en utilisant une syntaxe HTML. Pour l'instant, on se satisfera d'une solution où `a*bc*d` est rendu en `a<b>b</b><b>c</b>d` plutôt que `a<b>bc</b>d`.

Q.4.3 Construisez le processeur `MarkdownToHtmlProcessor` et testez-le.

Travaux sur Machines Encadrés

Exercice 5 : Découplage lecture/analyse

L'architecture vue en TD n'est pas très souple, car elle lie la lecture de la ligne depuis la source à l'analyse de son contenu. Afin d'améliorer la solution, nous allons affiner l'architecture

afin de séparer la source de lecture (fichier, console, réseau, etc.) de l'analyse de son format (verbatim, markdown, etc.).

Q.5.1 Donnez deux signatures `INPUT_SOURCE` et `INPUT_ANALYZER`. La première décrit un module se chargeant de lire ligne par ligne une source sous forme de chaînes. La seconde permet de transformer chaque chaîne en une ligne formattée.

Q.5.2 Donnez un foncteur `Input`, permettant à partir de deux modules correspondant aux signatures de la question précédente, de confectionner un module conforme à la signature `INPUT`.

Q.5.3 Donnez deux instances `File` et `Console` de la signature `INPUT_SOURCE`.

Q.5.4 Donnez deux instances `Verbatim` et `Markdown` de la signature `INPUT_ANALYSER`.

Exercice 6 : Découplage rendu/écriture

De la même façon, on procède au découplage de la sortie.

Q.6.1 Donnez deux signatures `OUTPUT_DEST` et `OUTPUT_RENDERER`. La première décrit un module se chargeant d'écrire ligne par ligne dans la destination à partir de chaînes. La seconde permet de transformer chaque ligne formattée en chaîne.

Q.6.2 Donnez un foncteur `Output`, permettant à partir de deux modules correspondant aux signatures de la question précédente, de confectionner un module conforme à la signature `OUTPUT`.

Q.6.3 Donnez deux instances `File` et `Console` de la signature `OUTPUT_DEST`.

Q.6.4 Donnez deux instances `Verbatim` et `Html` de la signature `OUTPUT_RENDERER`.

Exercice 7 : Affichage graphique (bonus)

À la place d'une représentation linéaire des chaînes de caractères formatées, on veut utiliser la représentation arborescente suivante.

```
type ftree =
  | Sequence of ftree list
  | Format of fmark list * ftree
  | Text of string
```

Q.7.1 Donnez la fonction de linéarisation `fline_of_ftree`.

Q.7.2 Donnez la fonction inverse `ftree_of_fline`.

Q.7.3 Montrez que pour une ligne de texte en représentation linéaire, il existe une (et une seule) représentation arborescente minimale (en nombre de constructeurs). Vérifiez que votre fonction calcule cette représentation minimale, corrigez-là si ce n'est pas le cas.

Q.7.4 En utilisant la représentation arborescente, donnez un meilleur afficheur HTML.

Q.7.5 Donnez un module `GraphicsOutput` de signature `OUTPUT`, permettant l'affichage graphique du texte formatté.