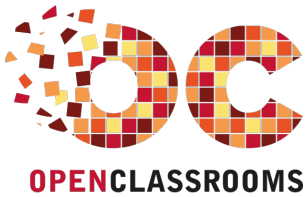
Parser un format simple en Haskell avec Parsec

Par Mr.Tambourine



www.openclassrooms.com

Dernière mise à jour le 16/08/2010

Sommaire

Parser un format simple en Haskell avec Parsec

Par  Mr.Tambourine

Mise à jour : 16/08/2010
Difficulté : Intermédiaire

Ce tutoriel a pour but de vous présenter les bases de l'utilisation de Parsec, une bibliothèque écrite en Haskell. Parsec facilite l'écriture d'analyseur syntaxique (*parser*) en fournissant des *parsers* plus ou moins basiques ainsi que des combinateurs pour les lier. Le *parsing* étant une tâche courante en programmation (que ce soit pour lire un fichier de configuration, des résultats dans une base de données ou interpréter un langage), la connaissance de Parsec peut très largement vous simplifier la vie.
Contrairement à bison ou à Happy, avec Parsec, la grammaire du langage à parser s'écrit directement en Haskell. Pour suivre ce tutoriel, il est donc seulement nécessaire d'avoir des bases dans ce langage. Si vous ne savez pas ce que sont une monade ou un foncteur, reportez-vous au cours de gnomnain (à ce jour, il est suffisant pour comprendre de quoi il retourne dans ce tutoriel).
Sommaire du tutoriel :

Prémices

Outils

Pour suivre le tutoriel, vous n'aurez besoin que d'un éditeur, d'un compilateur Haskell (de préférence avec un mode interactif) et de Parsec.
Parsec est disponible sur HackageDB. Vous pouvez télécharger le paquet directement et suivre les instructions du README ou utiliser cabal-install, voir les paquets de votre distribution. Attention tout de même, ce tutoriel est prévu pour la version 3.*. Pensez à vérifier quelle version vous installez (précisez là si vous utilisez cabal-install).

Objectifs

Afin d'en apprendre un peu plus sur Parsec, nous allons écrire un *parser* pour un format simple. J'ai choisi d'implémenter un *parser* pour les "Desktop Entries". Ce sont des fichiers de configuration donnant des informations sur la façon d'ouvrir un programme, comment l'afficher dans un menu … La spécification de ce format fait partie de celles proposées par le groupe Freedesktop.org et est utilisée, entre autres, par GNOME et KDE. Cette spécification a le mérite d'être courte, simple, disponible librement et en ligne (bien qu'en anglais), tout en permettant de jouer avec les bases de Parsec.
Voici à quoi ressemble le type de fichier que nous allons analyser:

Code : Autre - exemple.desktop

```
[Desktop Entry]
Cle=Valeur
Name=XMonad
Comment=Lightweight tiling window manager
Comment[fr]=Un gestionnaire de fenêtre pavant et léger
Exec=xmonad
Icon=xmonad
Type=Application
Version=0.9
NoDisplay=true
```

```
# Un commentaire
[Autre Groupe]
X-IsAwesome=true
X-CeNestPasUnBooleen=True
X-DonotKnow=
X-
JaiBesoinDunExemple=des chaînes contenant un caractère spécial:\t \n etc.
```

L'exemple parle de lui-même: un fichier .desktop est une suite de clef/valeur (comme Name=XMonad) appartenant à un groupe (X-IsAwesome=True appartient au groupe Autre Groupe) et les lignes commençant par un dièse sont des commentaires. Toutes les clefs se trouvant après la déclaration d'un groupe appartiennent à celui-ci.
Notre *parser* respectera les règles suivantes:

- Un nom de groupe peut contenir n'importe quel caractère, sauf les crochets ([]) et les caractères spéciaux (\n, \t…), et doit être unique.
- Un nom de clef peut contenir uniquement des caractères alphanumériques non accentués ainsi que le tiret (-) et les crochets.
- Au sein d'un même groupe, deux clefs ne peuvent avoir le même nom.
- La spécification retient 4 types pour les valeurs: string, localestring, boolean et numeric. Nous conserverons ces types à l'exception de localestring, qui sera traité comme string (la différence entre les deux se situant dans l'encodage, ça ne me semble pas intéressant d'en parler).

Nous ajouterons deux ou trois choses au cours du tutoriel, mais il est plus que temps de lancer notre éditeur et d'utiliser Parsec.

Premier contact

Dans la partie précédente, nous avons distingué trois éléments principaux dans notre spécification: les groupes, les commentaires et les paires clefs/valeurs. Le but de cette partie sera de *parser* chacun d'entre eux.

Parser une ligne

Dans un fichier .desktop, les lignes ont une particularité intéressante: elles contiennent un et un seul élément: une déclaration de groupe, un commentaire ou une paire (éventuellement rien, mais ce ne sera pas un problème). Commençons donc par *parser* une ligne.

Parser un commentaire

Les commentaires sont les lignes les plus simples à *parser*. En effet, il suffit de vérifier que la ligne commence par un dièse. Voyons comment implémenter ça avec Parsec. Les deux premières lignes du code permettent d'importer Parsec. Par la suite, on les considérera comme sous-entendues.

Code : Haskell

```haskell
import Text.Parsec
import Text.Parsec.String (Parser)

commentaire :: Parser String
commentaire = char '#' >> (many $ noneOf "\n")
```

Tout d'abord, un commentaire sur le type de la fonction `commentaire` . `Parser String` est un alias pour le type `ParsecT String () Identity String` . `ParsecT s u m a` est la monade (en fait, le transformeur de monade) utilisé par Parsec. Son premier paramètre (`s`) est le type du flux à *parser*. Un flux est une instance de la classe `Stream` , définie par Parsec et dont le but est entre autres de gérer la position actuelle dans le flux. Ici, nous allons *parser* une liste de caractères. Le second (`u`) ne nous sera pas utile et donc nous n'en parlerons pas. Le troisième (`m`) est une monade. Comme `ParsecT` est un transformeur de monade, on peut le composer avec d'autres monades pour profiter des caractéristiques de chacune. Mais ici, nous utilisons la monade `Identity` qui permet d'obtenir en fait une monade classique. Il n'est pas nécessaire de bien comprendre ce qu'est un transformeur de monade pour la suite puisque nous allons l'utiliser comme une simple monade. Toutefois, ceux souhaitant en savoir plus peuvent lire le chapitre de Real World Haskell sur le sujet. Enfin, le dernier (`a`) est le

type de retour du *parser*.

Après ce point légèrement délicat, jetons un coup d'œil au corps de la fonction. L'une des choses qu'on peut remarquer en premier est l'utilisation de l'opérateur `>>` . Comme je l'ai dit, Parsec est une bibliothèque **monadique**: on va donc pouvoir utiliser toutes les fonctions sur les monades et la notation `do` . Dans Parsec, `>>` va appliquer le *parser* à sa droite si celui à sa gauche a réussi. On retrouve un peu la logique de la monade `Maybe` qui court-circuite toute la suite d'action si l'une d'entre elles retourne `Nothing` . Là, si un *parser* échoue, tout le reste échoue. Sinon, on évalue la fonction suivante.

À gauche de `>>` se trouve la fonction `char` . C'est une fonction fournie par Parsec. Elle prend en paramètre un caractère (en Haskell, on note les caractères entre guillemets simples) et va tenter de le *parser*. Si elle réussit, c'est donc qu'on a affaire à un commentaire.

Il ne nous reste plus qu'à analyser le reste de la ligne. `noneOf` prend en paramètre une liste de caractère et réussi si le caractère à *parser* n'est **pas** dans la liste. Parsec contient également une fonction s'appelant `oneOf` qui fait l'opposé (elle *parse* seulement les caractères qui sont dans la liste qui lui est fournie). Comme un commentaire peut contenir n'importe quoi à l'exception d'un retour à la ligne, on utilise `noneOf "\n"` .

`many` est un combinateur très courant qui va appliquer le *parser* qu'on lui passe en argument autant de fois que possible (ça peut être 0, 1 ou n'importe quel nombre de fois, tant que le *parser* réussi) et retourner une liste des éléments *parsés*. `noneOf` retournant le caractère analysé — tout comme `char` d'ailleurs — `commentaire` nous retournera une liste de caractère. Cette liste sera le commentaire, privé du dièse initial.

Chargez le code précédent avec GHCi et essayons de *parser* un texte.

Code : Console

```
*Main>parse commentaire "" "#Salut"
Right "Salut"
```

Comme on s'y attendait, la fonction `parse` retourne le commentaire, le dièse en moins. La fonction `parse` prend trois arguments : un *parser*, une chaîne de caractère utilisée comme label pour les erreurs (le nom du fichier en général) et le texte à analyser. Elle retourne une valeur du type `Either ParseError a` où `a` est le type de retour du *parser* passé en argument.

Essayons maintenant avec autre chose qu'un commentaire.

Code : Console

```
*Main>parse commentaire "" "Salut"
Left (line 1, column 1):
unexpected "S"
expecting "#"
```

Parsec renvoie une erreur, mais précise aussi ce qui a causé cette erreur. Ici, le premier caractère doit être un dièse, pas un S. On verra avant la fin du tutoriel qu'on peut personnaliser le message d'erreur grâce au combinateur `<?>` .

Parser une déclaration de groupe

Si vous avez bien compris comment *parser* un commentaire, vous devriez déjà être capable de *parser* une déclaration de groupe. Voici une première implémentation n'utilisant que ce qu'on a vu pour les commentaires :

Code : Haskell

```
groupe :: Parser String
groupe = do
    char '['
    nom <- many $ noneOf ("[]\127"++['\0'..'\31'])
    char ']'
    return nom
```

La liste fournie à `noneOf` est la seule chose nouvelle. Dans la première partie, nous avons fixé que les noms de groupe pouvaient contenir tous les caractères ASCII à l'exception des crochets et des caractères de contrôle. En Haskell, on peut écrire les caractères soit en les tapant directement comme `'A'` soit en donnant leur représentation numérique (`'\65'` par exemple). Les caractères de contrôles sont tous ceux ayant un code inférieur à 32 ou égal à 127. Nous autorisons l'utilisation de `'\32'` car c'est l'espace. La notation `[a..b]` permet de lister tout les éléments entre a et b (avec a=2 et b=5, la liste sera `[2,3,4,5]`).

Cette implémentation marche correctement, mais on peut faire beaucoup plus concis en regardant dans les combinateurs proposés par Parsec. `between` est un combinateur prenant trois *parsers* en paramètre. Il va *parser* le premier puis le troisième et après, le second et retourner le résultat du troisième. Dans notre cas, la valeur de retour sera donc le nom du groupe, sans les crochets. C'est exactement ce que fait notre fonction `groupe` . En utilisant `between` , elle devient ceci :

Code : Haskell

```haskell
groupe :: Parser String
groupe = between (char '[') (char ']') (many $ noneOf
("[]\127"++['\0'..'\31']))
```

C'est déjà plus esthétique, non? Il faut tout de même faire attention lorsqu'on utilise `between` à ce que le troisième *parser* n'inclue pas le délimiteur de fin, sinon `between` échouera forcément.
Si nous avions écrit `groupe` de cette façon (`anyChar` *parse* n'importe quel caractère), `groupe = between (char '[') (char ']') $ many anyChar` , lors de l'exécution de notre parser, nous aurions eu droit à une erreur du type "unexpected end of input", indiquant que le *parser* a consommé tout le fichier sans pour autant réussir.

Parser une paire

Après tout cela, *parser* une paire se révèle être extrêmement simple. Je vous recommande d'ailleurs d'essayer d'implémenter vous même le *parser* avant de voir la solution que voici :

Code : Haskell

```haskell
paire :: Parser String
paire = do
    c <- clef
    char '='
    v <- valeur
    return $ c ++ " vaut " ++ v

clef :: Parser String
clef = many (alphaNum <|> oneOf "[]_-@")

valeur :: Parser String
valeur = many $ noneOf "\n"
```

La grande nouveauté est le combinateur `<|>` . Il applique tout d'abord le *parser* à sa gauche. S'il réussit, il retourne le résultat. S'il échoue **sans modifier l'état du flux** (le flux est ici la chaîne de caractère à lire, par exemple, le contenu d'un fichier .desktop), il applique le second. Un *parser* consomme le flux à partir du moment où il *parse* quelque chose, et ce même s'il rencontre une erreur après. Prenons le *parser* suivant :

Code : Haskell

```haskell
test = anyChar >> char '#'
```

Comme `anyChar` réussi pour tout caractère, on est quasiment sûr que ce *parser* va consommer une partie du flux. Si on l'utilise avec `<|>` , le *parser* à droite ne sera jamais essayé. Nous verrons ce problème plus en détail par la suite. Pour l'instant, aucun

problème puisqu'`alphaNum` ne consomme rien lorsqu'il échoue. Une autre chose à laquelle nous devons prendre garde lorsqu'on utilise `<|>`, c'est que les deux *parsers* doivent être du même type, et donc retourner des données du même type. Par ailleurs, `alphaNum` est la seule autre nouveauté de ce code. Il réussit dans le cas où le caractère à *parser* est un chiffre ou une lettre.

La fonction `paire` retourne la clef et la valeur sous la forme d'une chaîne de caractère.

Parser un fichier

Pouvoir *parser* une ligne, c'est cool mais c'est assez limité surtout quand la plupart des fichiers utilisant le format que nous devons *parser* sont généralement composés de plusieurs dizaines de lignes. Toutefois, nous n'avons pas perdu notre temps: nous avons un *parser* pour chaque élément de base de notre spécification. Il suffit maintenant d'assembler le tout et ce à l'aide, évidemment, des combinateurs de Parsec.

Il faut tout d'abord définir une fonction capable de *parser* **n'importe quelle ligne** qu'on puisse trouver dans un fichier .desktop, c'est-à-dire soit une déclaration de groupe, soit un commentaire, soit une paire. Si vous avez pensé utiliser le combinateur `<|>`, bravo, vous avez bien suivi. Mais dans quel ordre? On sait que si un parser consomme une partie de l'entrée, les autres options sont ignorées. Il faut donc d'abord préciser des *parsers* qui ne consommeront rien dans le cas où ils échouent. On sait que `commentaire` ne posera aucun problème puisqu'il ne peut échouer que si `char '#'` échoue, et lorsque `char` échoue, cela veut dire qu'il n'a rien *parsé*, et s'il réussi, on se trouve nécessairement face à un commentaire. Le problème se pose pour `groupe` et `paire`. En effet, dans le cas d'une ligne commençant par un crochet ouvrant ([), `groupe` va parser tout les caractères suivants n'étant pas des caractères de contrôle. Le problème est qu'on pourrait avoir en fait affaire à une clef, ce qu'on ne saura qu'au moment où nous trouverons le crochet fermant ou le retour à la ligne. Ça semble mal engagé.

On peut envisager deux solutions : la première serait de récupérer l'état actuel du flux avant d'utiliser `groupe` puis, dans le cas où il échoue, remettre le flux dans son état précédent et appliquer `paire`. Parsec nous permet de faire cela. La seconde serait d'appliquer d'abord le *parser* `paire` tout en vérifiant d'abord que le premier caractère n'est pas un crochet. En effet, les crochets dans les noms de clefs sont sensés servir à indiquer la localisation et donc ne pas se trouver au début du nom. Il paraît raisonnable et en accord avec la spécification d'interdire l'utilisation du crochet comme premier caractère.
Dans les deux cas, on veut pouvoir faire une « recherche en avant » (l'expression consacrée en anglais est *lookahead*). Voici l'implémentation de la fonction `ligne` dans chacun des deux cas :

Code : Haskell - Première solution

```
ligne :: Parser String
ligne = commentaire <|> try groupe <|> paire
```

Code : Haskell - Seconde solution

```
ligne :: Parser String
ligne = commentaire <|> paire <|> groupe

clef = lookAhead (noneOf "[") >> many (alphaNum <|> oneOf "[]_@-")
```

La première solution utilise la fonction `try`. `try` va essayer d'appliquer le *parser* qu'on lui passe en argument. S'il échoue, le flux sera remis dans son état précédent. Il faut noter que, lorsqu'on utilise l'opérateur `<|>`, `try` n'a d'intérêt que s'il se trouve à sa gauche.
Dans la seconde solution, c'est dans la façon de *parser* une clef que se trouve la différence. lookAhead permet d'appliquer un parser sans altérer le flux. Le *parser* `cle` va échouer lorsqu'un crochet se trouvera en début de ligne sans pour autant le *parser*. Ensuite, le *parser* groupe sera appliqué. Les deux solutions sont plutôt bonnes mais nous allons choisir la seconde (utilisant `lookAhead`), car elle sera plus adaptée pour la suite.

Bien. Maintenant que nous pouvons *parser* n'importe quelle ligne, il va falloir être capable de le faire autant de fois que nécessaire. Ça ressemble un peu à la définition de `many` tout ça, non? Mais il y a un problème: notre *parser* `ligne` ne consomme pas le caractère de retour à la ligne ('\n'). Si nous faisons simplement un `fichier = many ligne`, le *parser* va s'arrêter à la fin de la première ligne. On va légèrement modifier notre définition de `ligne` pour prendre en compte ce problème.

Code : Haskell

```
ligne :: Parser String
ligne = do
    resultat <- commentaire <|> paire <|> groupe
    many1 newline
    return resultat
```

`newline` est un synonyme de `char '\n'`. On en profite également pour gérer les lignes vides, puisque `many1 newline` va consommer toutes les lignes vides entre deux éléments (`many1` fonctionne comme `many` sauf qu'il doit au moins *parser* une fois pour réussir). Maintenant, on peut simplement définir la fonction `fichier` de cette façon :

Code : Haskell

```
fichier :: Parser [String]
fichier = do
    resultat <- many ligne
    eof
    return resultat
```

`eof` sert simplement à *parser* l'indicateur de fin de fichier. Néanmoins, Parsec nous permet encore une fois de faire mieux. `endBy` est un combinateur semblable à `many` mais qui prend deux *parsers* en argument et va appliquer le premier puis le second et ce autant de fois que possible (de 0 à ∞). `endBy` retourne ensuite une liste des résultats fournis par le premier *parser*.

Code : Haskell

```
fichier :: Parser [String]
fichier = ligne `endBy` many1 newline

ligne :: Parser String
ligne = commentaire <|> paire <|> groupe
```

Et voilà, vous venez d'implémenter un *parser* complet pour notre format. On va maintenant utiliser la fonction `parse` pour récupérer le résultat. Voici le code complet de notre *parser* :

Code : Haskell

```
import Text.Parsec
import Text.Parsec.String (Parser)

parserDesktop :: SourceName -> String -> Either ParseError [String]
parserDesktop = parse fichier

parserDesktopF :: FilePath -> IO (Either ParseError [String])
parserDesktopF chemin = fmap (parserDesktop chemin) $ readFile
chemin

fichier :: Parser [String]
fichier = ligne `endBy` many1 newline

ligne :: Parser String
ligne = commentaire <|> paire <|> groupe

clef :: Parser String
clef = lookAhead (noneOf "[") >> many (alphaNum <|> oneOf "[]_@-")

valeur :: Parser String
valeur = many $ noneOf "\n"

paire :: Parser String
paire = do
```

```
        c <- clef
        char '='
        v <- valeur
        return $ c ++ " vaut " ++ v

groupe :: Parser String
groupe = between (char '[') (char ']') (many $ noneOf
("[]\127"++['\0'..'\31']))

commentaire :: Parser String
commentaire = char '#' >> (many $ noneOf "\n")
```

Il n'y a que deux nouvelles fonctions : `parserDesktop` et `parserDesktopF` . La première n'est qu'un raccourci pour `parse` . La seconde prend en paramètre le chemin d'un fichier et va le *parser*. C'est une simple fonction de test. Si vous voulez voir ce que fait notre parser, vous trouverez probablement des fichiers .desktop dans le dossier /usr/share/applications/ si vous utilisez un *nix. Sinon, il reste l'exemple de la première partie.

Le point mode

Avant de nous attaquer plus profondément à notre *parser*, nous allons parler un peu de style.

Si vous regardez nos fonctions, vous verrez que nous n'avons utilisé la notation **do** qu'une seule fois: dans la fonction `paire` pour pouvoir stocker le résultat de nos *parsers* dans des variables pour pouvoir les utiliser par la suite. La notation **do** est certes très pratique, mais pousse également à adopter un style impératif, ce qui n'est pas franchement désirable en Haskell. On peut utiliser la fonction `liftM2` issue du module `Control.Monad` pour revenir à quelque chose de plus propre.

Code : Haskell

```
import Text.Parsec
import Text.Parsec.String (Parser)
import Control.Monad

paire :: Parser String
paire = liftM2 (\c v -> c ++" vaut "++ v)  cle  (char '=' >> valeur)
```

Toutefois, un certain nombre de programmeurs préfèrent utiliser `ParsecT` comme un foncteur applicatif (étant une monade, `ParsecT` est nécessairement un foncteur applicatif).
Depuis la version 3.0, `ParsecT` est une instance des classes `Applicative` et `Alternative` . Il suffit donc d'importer le module `Control.Applicative` pour utiliser `ParsecT` comme un foncteur applicatif. Il faut également penser à ne pas importer les fonctions `many` , `optional` et `<|>` car celles-ci sont aussi définies dans `Control.Applicative` . `optional` est un combinateur qui va tenter d'appliquer un *parser* (si le *parser* échoue **sans modifier l'état du flux**, `optional` ne renvoie pas d'erreur et retourne `()`).

En utilisant `ParsecT` en tant que foncteur applicatif, notre code devient ceci :

Code : Haskell

```
import Text.Parsec hiding (many,optional,(<|>))
import Text.Parsec.String (Parser)
import Control.Applicative

parserDesktop :: SourceName -> String -> Either ParseError [String]
parserDesktop = parse fichier

parserDesktopF :: FilePath -> IO (Either ParseError [String])
parserDesktopF chemin = fmap (parserDesktop chemin) $ readFile
chemin

fichier :: Parser [String]
fichier = ligne `endBy` many1 newline
```

```
ligne :: Parser String
ligne = commentaire <|> paire <|> groupe

clef :: Parser String
clef = lookAhead (noneOf "[") *> many (alphaNum <|> oneOf "[]_@-")

valeur :: Parser String
valeur = many $ noneOf "\n"

paire :: Parser String
paire = liftA2 (\c v -> c ++ " vaut " ++ v) clef (char '=' *>
valeur)

groupe :: Parser String
groupe = between (char '[') (char ']') (many $ noneOf
("[]\127"++['\0'..'\31']))

commentaire :: Parser String
commentaire = char '#' *> (many $ noneOf "\n")
```

Quasiment rien n'a changé : les `>>` sont devenus `*>` et `liftM2` a été transformé en `liftA2` . L'opérateur `*>` va appliquer les deux fonctions et retourner le résultat de la seconde (celle qui est à sa droite). `<*` fait la même chose mais retourne le résultat de la première (à sa gauche).
Vu le peu de modifications, on peut se demander si ça vaut vraiment le coup. C'est vrai dans notre cas, car il est plutôt trivial, mais dans d'autres les modifications sont beaucoup plus importantes. Comme les foncteurs applicatifs favorisent un style de programmation plus fonctionnelle, ce sont eux que nous allons privilégier dans le reste du tutoriel. Toutefois, ça ne nous empêche absolument pas d'utiliser parfois l'aspect monadique de `ParsecT` (et donc `do`).

Un parser, un vrai

Le *parser* que nous venons d'implémenter présente plusieurs problèmes.

Le premier est le résultat qu'il fournit. Si vous ne l'avez pas testé, voici un exemple de résultat obtenu en parsant un fichier .desktop :

Code : Console

```
Right ["Desktop Entry","Version vaut 1.0","Encoding vaut UTF-
8","Name vaut rxvt-unicode",
"Comment vaut An Unicode capable rxvt clone","Exec vaut urxvt","Icon vaut rxvt-
unicode","Terminal vaut false",
"Type vaut Application","Categories vaut Application;System;TerminalEmulator;"]
```

Utilisez la fonction `lines` sur le même fichier et vous obtiendrez à quelques détails près la même chose. Plusieurs dizaines de lignes pour récupérer une fonction de `Prelude` , ça semble cher payé. En plus, le résultat obtenu est presque moins utilisable que le fichier brut et on récupère toutes les valeurs sous forme de texte, alors que nous avions décidé que l'on conserverait leur type. Malgré cela, le *parser* est fonctionnel. Il sait reconnaître une clef, un groupe, un commentaire. Le travail à effectuer se trouve en fait sur la façon de traiter ces éléments une fois extraits du texte brut.

Le second problème est que les clefs ne sont pas liées à leur groupe. On perd tout l'intérêt d'avoir des groupes si on procède de cette façon. En fait, nous avons traité les paires, les groupes et les commentaires plus ou moins de la même façon dans notre résultat. Il faudrait hiérarchiser ces informations.

On veut des types

Nous allons commencer par nous intéresser au type des valeurs. Rappelez-vous, nous en avions retenu trois : string, numeric et boolean. Nous allons donc créer un type algébrique pour les gérer.

Code : Haskell

```
data DValeur = DString String | DNum Float | DBool Bool deriving
```

```
    (Show, Eq)
```

Maintenant, il va falloir reconnaître ces types. Un booléen est représenté par la chaîne false ou true. Il faut vérifier la présence de ces chaînes et surtout qu'elles ne soient pas suivies par autre chose qu'un retour à la ligne, sinon, la valeur est du type string. L'utilisation de `lookAhead` semble toute indiquée.

Code : Haskell

```
pBool = (True <$ string "true" <|> False <$ string "false") <*
lookAhead newline
```

`string` est très semblable à `char` sauf qu'au lieu de chercher un caractère, on cherche toute une chaîne. On utilise `lookAhead` pour vérifier qu'on a bien affaire aux mots clefs true ou false et pas à une chaîne. Souvent, pour rechercher des mots clefs, on utilise le combinateur `notFollowedBy` . Il prend en paramètre un *parser* et l'applique. Si le *parser* rencontre une erreur, `notFollowedBy` réussi. Ce combinateur à la particularité de ne pas modifier l'état du flux et donc peut être très utile pour faire une « recherche en avant » . `notFollowedBy` retourne l'unité (`()`). Toutefois, ici nous utilisons `lookAhead` car la liste des caractères qui ne doivent pas suivre notre mot clef est plus longue que celle des caractères qu'il faut trouver (il n'y en a qu'un).

Ensuite, il va nous falloir parser un nombre. Haskell possède justement un module répondant au doux nom de `Numeric` qui contient des fonctions permettant de lire les flottants. `readSigned readFloat s` (où `s` est du type `String`) renvoie une liste du type `[(Float,String)]` . Dans le cas où nous aurions effectivement à faire à un nombre, cette liste contiendra un seul couple dont le premier élément sera le nombre et le second, le reste de la chaîne. Ça semble très séduisant, mais le problème est que ces fonctions n'interagissent pas avec Parsec et ne modifient donc pas l'état du flux, quand bien même elles *parseraient* quelque chose. Comment faire pour les utiliser ? On va devoir passer par les étapes suivantes :

- Nous devons d'abord récupérer le flux qui n'a pas encore été consommé depuis `ParsecT` . Pour ce faire, on va utiliser la fonction `getInput` .
- Ensuite, il va falloir utiliser la fonction `readSigned readFloat` .
- Si ce *parser* retourne une liste contenant un seul tuple, c'est que nous avons affaire à un nombre. Il faut alors retourner ce nombre et modifier l'état du flux en utilisant le second élément du couple et la fonction `setInput` de Parsec. Il faut aussi penser à vérifier que le caractère suivant est un retour à la ligne.
- Si on récupère autre chose, il va falloir signaler à Parsec que notre *parser* a échoué. On utilise pour cela la fonction `parserZero` , qui est un *parser* qui échoue tout le temps (on peut aussi utiliser `empty` , qui est un alias).

Code : Haskell

```
pNum = do
    f <- getInput
    case readSigned readFloat f of
        [(n,f')] -> n <$ (setInput f' <* lookAhead newline)
        _        -> parserZero
```

Il ne nous reste plus que le type string à identifier. Comme les chaînes peuvent contenir n'importe quoi, c'est trivial : `pString = many (noneOf "\n")`

A présent, il nous faut réécrire le *parser* de manière à gérer les types.

Code : Haskell

```
valeur = try (DBool <$> pBool) <|> try (DNum <$> pNum) <|> DString
pString
```

L'utilisation de `try` est nécessaire puisque `pNum` tout comme `pBool` peuvent modifier l'état du flux et échouer (par exemple, sans `try`, nous ne pourrons pas *parser* "true is not false" comme valeur).
Quand on commence à enchaîner les `<|>`, il peut être intéressant d'utiliser le combinateur `choice`. Il prend une liste de *parser* en paramètre et va modifier le constructeur de liste `:` en `<|>`. C'est un bon exemple d'une utilisation de `foldr`. Nous pouvons utiliser `choice` pour écrire `valeur` :

Code : Haskell

```
valeur = choice [ try (DBool <$> pBool)
                 ,try (DNum <$> pNum)
                 ,DString <$> pString ]
```

Nous voulons des informations

Nous avons résolu notre premier problème et notre *parser* est en bonne voie pour devenir utile. Il faut maintenant songer à la manière d'organiser les données qu'il récupère. Les "Desktop Entries" sont des fichiers de configuration donc on s'en sert surtout pour lire des données. Il faut une structure qui permette de récupérer facilement la valeur d'une clef. Il existe dans `GHC.List` une fonction nommée `lookup` qui permet de récupérer une valeur depuis une liste associative, autrement dit une liste du type `Eq a => [(a,b)]`. Remplacez `a` par le nom d'une clef et `b` par sa valeur et on obtient une structure permettant un accès simple à une valeur depuis sa clef.

Mais comment inscrire les groupes dans cette structure ? Souvent, lorsqu'on a besoin de hiérarchiser des données, on utilise un arbre. C'est très bien pour une expression mathématique, du HTML … Mais comme un groupe ne peut pas avoir de sous-groupe, on peut se contenter de beaucoup plus simple. Tout comme il était intéressant de pouvoir accéder à une clef en fonction de sa valeur, il serait intéressant de pouvoir accéder aux éléments d'un groupe simplement avec son nom. Là encore, une liste associative ne semble pas un mauvais choix. Au final, notre *parser* renverra une liste de type `[(String,[(String,DValeur)])]`.

Des clefs et des valeurs

Voici la fonction `paire` telle que nous l'avons écrite dans la seconde partie : `paire = liftA2 (\c v -> c ++ " vaut " ++ v) clef (char '=' *> valeur)`. Pour se retrouver avec une fonction `paire` retournant un couple, nous avons juste besoin de changer la fonction anonyme en `(,)`.

Code : Haskell

```
paire = liftA2 (,) clef (char '=' *> valeur)
```

Voilà.

Des groupes et des paires

On aimerait bien faire exactement la même chose avec les groupes pour récupérer notre couple. Sauf que c'est un peu (mais vraiment un peu) plus compliqué. Tout d'abord, nous allons modifier (et renommer) le *parser* `fichier` pour qu'il ne gère plus les groupes. Voici le *parser* `bloc` :

Code : Haskell

```
bloc = (commentaire <|> paire) `endBy` many1 newline
```

Oui, sauf que si vous essayez de compiler ça, vous aurez droit à une erreur. En effet, les *parsers* avec lesquels on utilise `<|>` doivent être du même type. Sauf que `paire` a pour type `paire :: Parser (String, DValeur)` alors que `commentaire` a pour type `commentaire :: Parser String`. On pourrait envisager d'utiliser le combinateur `skipMany` qui fonctionne de la même façon que `many` mais retourne `()`. Sauf que, si nous voulions par exemple modifier les valeurs de notre fichier puis les réécrire, on perdrait les commentaires. Nous allons donc simplement ajouter un constructeur au type `DValeur` pour les commentaires. Et, en guise de clef, nous donnerons le numéro de la ligne.

Code : Haskell

```
data DValeur = DString String | DNum Float | DBool Bool | DCom
String
    deriving (Show, Eq)

commentaire = liftA2 (,) ( show . sourceLine <$> getPosition) (DCom
<$> (char '#' *> many (noneOf  "\n")))
```

`getPosition` renvoie une valeur du type `SourcePos` qui contient des informations telles que le nom du fichier, la ligne ou la colonne. `sourceLine` récupère le numéro de la ligne comme un entier.
Maintenant, `bloc` fonctionne correctement. Pour *parser* un groupe, il nous suffit de réutiliser `liftA2` :

Code : Haskell

```
groupe' = liftA2 (,) groupe bloc
```

Et enfin, il faut créer une fonction `fichier` qui utilise la fonction `groupe'` .

Code : Haskell

```
fichier = manyTill groupe' eof
```

`manyTill` va appliquer le *parser* `groupe'` jusqu'à trouver la fin du fichier (`eof`).

On pourrait croire que c'est fini, mais il y a un dernier détail à régler. La fonction `groupe` *parse* une déclaration de groupe mais laisse le caractère de retour à la ligne. Lorsqu'on va utiliser `bloc` , on aura une erreur puisque nous nous attendons en fait à trouver un commentaire ou une paire. Il faut consommer ce caractère dans le *parser* `groupe` :

Code : Haskell

```
groupe = between (char '[') (char ']') (many $ noneOf
("[]\127"++['\0'..'\31'])) <* newline
```

Voici le code final de notre *parser* :

Code : Haskell

```
import Text.Parsec hiding ((<|>), many, optional)
import Text.Parsec.String (Parser)
import Control.Applicative
import Numeric

data DValeur = DString String | DNum Float | DBool Bool | DCom
String
    deriving (Show, Eq)

parserDesktopF :: FilePath -> IO (Either ParseError
[(String,[(String,DValeur)])])
parserDesktopF x = fmap (parserDesktop x) $ readFile x

parserDesktop :: SourceName -> String -> Either ParseError
[(String,[(String,DValeur)])]
parserDesktop = parse fichier

fichier :: Parser [(String,[(String,DValeur)])]
fichier = manyTill groupe' eof
```

```
groupe' :: Parser (String,[(String,DValeur)])
groupe' = liftA2 (,) groupe bloc

groupe :: Parser String
groupe = between (char '[') (char ']') (many $ noneOf
("[]\127"++['\0'..'\31'])) <* newline

bloc :: Parser [(String,DValeur)]
bloc = (commentaire <|> paire) `endBy` many1 newline

paire :: Parser (String,DValeur)
paire = liftA2 (,) clef (char '=' *> valeur)

commentaire :: Parser (String,DValeur)
commentaire = liftA2 (,) ( show . sourceLine <$> getPosition) (DCom
<$> (char '#' *> many (noneOf  "\n")))

clef :: Parser String
clef = lookAhead (noneOf "[") *> many (alphaNum <|> oneOf "[]_@-")

valeur :: Parser DValeur
valeur = choice [ try (DBool <$> pBool)
                 ,try (DNum <$> pNum)
                 ,DString <$> pString ] <?> "a valid desktop value"


pBool :: Parser Bool
pBool = (True <$ string "true" <|> False <$ string "false") <*
lookAhead newline

pNum :: Parser Float
pNum = do
    f <- getInput
    case readSigned readFloat f of
        [(n,f')] -> n <$ (setInput f' <* lookAhead newline)
        _        -> parserZero

pString :: Parser String
pString = many (noneOf "\n")
```

J'ai juste rajouté l'opérateur `<?>` dans la fonction `valeur`, à titre d'exemple. Ce combinateur se comporte de façon similaire à `<|>` sauf que, dans le cas où le *parser* à sa gauche échoue **sans modifier l'état du flux**, il renvoie un message d'erreur utilisant la chaîne à sa droite pour indiquer ce qu'on devrait trouver.

Si vous tester à nouveau ce code, voici le type de retour qu'on obtient :

Code : Console

```
y = Right [("Desktop Entry",[("Version",DNum 1.0),("Encoding",DString "UTF-8"),("Na
          ,("Comment",DString "An Unicode capable rxvt clone"),("Exec",DString "urx
          ,("Terminal",DBool False),("Type",DString "Application"),("Categories",DS
```

On peut simplement utiliser `either (const Nothing) (\x -> lookup "Desktop Entry" x >>= lookup "Name") y` pour récupérer le nom de l'application.
Ce tutoriel touche à sa fin et vous en savez déjà beaucoup sur Parsec mais il vous reste encore beaucoup de choses à apprendre sur Parsec, notamment sur les modules `Token` et `Expr`, particulièrement utile lorsqu'on s'attaque à un projet un peu plus sérieux que de *parser* un petit fichier de configuration.