Bonjour

Méthodologie de Programmation avec Objective CAML

Cours 10

Vincent Poirriez

courriel :Vincent.Poirriez@univ-valenciennes.fr

Un exemple complet avec analyse syntaxique par flots

Implantation d’une calculette logique

Le langage logique

Présentation des propositions

Le langage des propositions comporte deux valeurs : `Vrai` et `Faux`, parfois représentées par `1` et `0`.

Des *connecteurs* : ou, et, non, implique, équivalent. Parfois, *ou* est noté `+` et *et* est noté .

Et des variables qui sont prises dans un ensemble dénombrable.

Une proposition est une tautologie si elle est toujours vraie, quelquesoit les valeurs prises par ses variables (libres). (exemple `x <==> x`)

objectifs

Réaliser un interprète du langage logique : *une calculette logique*

Exemple d'utilisations – Lancer la caculette

```
$ ./calclog
calclog>
```

– Vérifier un théorème :

```
calclog> x <=> x
Théorème: pour toute proposition x,
x <=> x
```

– Deux autres :

```
 calclog> 1 => x
1 => x n'est pas un théorème.
 la proposition est fausse quand:
 x est fausse
calclog> 0 => x
Théorème: pour toute proposition x,
0 => x
```

Plan de travail

- Il nous faut définir un type pour représenter les propositions : *syntaxe abstraite*.
- Écrire les fonctions d'évaluation et de vérification de tautologie sur cette représentation.
- Passer de la syntaxe concrète à la syntaxe abstraite.
- Construire une boucle interactive.

1. Les deux premiers points dans un module `Proposition`
2. Le passage de la syntaxe concrète à la syntaxe abstraite sera traité dans deux modules : `Lexer` et `Parser`
3. La boucle interactive dans `Maintauto`.

module Proposition

Le type

proposition.mli

```
type t =
  | Vrai
  | Faux
  | Non of t
  | Et of t * t
  | Ou of t * t
  | Implique of t * t
  | Équivalent of t * t
  | Variable of string
```

Il s'agit d'un type arborescent : un arbre de syntaxe.

Vu de l'extérieur : `Proposition.t`

Les valeurs exportées

proposition.mli(suite)

```
(** [to_string p] renvoie [p] sous la forme d'une chaîne de
    caractères compréhensible*)
val to_string : t -> string
(** [variables_libres p] renvoie la liste des variables libres
   dans la  proposition [p] *)
val variables_libres : t -> string list
(**[vérifie_tautologie p vlibres] teste si [p] est une tautologie
  en instanciant par toutes les valeurs possibles les variables
  de [vlibres]. Déclenche l'exception [Réfutation liste] où
  [liste] est une instance des variables [vlibres] telle que
  [p] n'est pas vraie si [p] n'est pas une tautologie.*)
exception Réfutation of (string * bool) list
val vérifie_tautologie : t -> string list -> unit
```

Passons à l'implantation...

Trouver toutes les variables d'une proposition

Une fonction auxilaire, récursive terminale :

proposition.ml

```
let rec variables_aux accu proposition = match proposition with
| Variable v -> if List.mem v accu then accu else v::accu
| Non p -> variables_aux accu p
| Et (p1,p2) -> variables_aux (variables_aux accu p1) p2
| Ou (p1,p2) -> variables_aux (variables_aux accu p1) p2
| Implique (p1,p2) -> variables_aux (variables_aux accu p1) p2
| Équivalent (p1,p2) -> variables_aux (variables_aux accu p1) p2
| _ -> accu
```

La fonction exportée :

```
let variables_libres proposition = variables_aux [] proposition
```

Évaluer une proposition

```
(** Renvoie [true] ou [false] suivant que [p] s'évalue en
   [Vrai] ou [Faux] compte tenu des liaisons des variables. Suppose
   que toutes les variables sont présentes dans [liaisons] *)
let rec évalue_dans liaisons p = match p with
| Vrai -> true | Faux -> false
| Non p' -> not(évalue_dans liaisons p')
| Et (p1,p2) ->
    (évalue_dans liaisons p1) && (évalue_dans liaisons p2)
| Ou (p1,p2) ->
    (évalue_dans liaisons p1) || (évalue_dans liaisons p2)
| Implique (p1,p2) ->
    (not (évalue_dans liaisons p1)) || (évalue_dans liaisons p2)
| Équivalent (p1,p2) ->
    (évalue_dans liaisons p1) = (évalue_dans liaisons p2)
| Variable v -> List.assoc v liaisons
```

Un vérificateur de tautologie

Définir l'exception qui a été déclarée :

```
exception Réfutation of (string * bool) list
```

Une proposition est une tautologie, ou un théorème, si elle est vraie quelquesoit les valeurs prises par les variables présentes.

```
let rec vérifie_table_de_vérité proposition liaisons variables =
 match variables with
 | v :: suite ->
    vérifie_table_de_vérité proposition((v, true)::liaisons) suite;
    vérifie_table_de_vérité proposition((v, false)::liaisons) suite
 |[] -> if not ( évalue_dans liaisons proposition) then
     raise (Réfutation liaisons)
```

La fonction exportée :

```
let vérifie_tautologie proposition variables =
  vérifie_table_de_vérité proposition [] variables
```

Syntaxe concrète

Ce que nous avons n’est pas vraiment utilisable en tant que calculette.

Ce n’est pas très usuel de devoir lire et écrire :

```
Ou(Et(Variable "P", Variable "Q"),Variable "S")
```

On préfère lire et écrire :

```
(P et Q) ou S
```

De syntaxe abstraite vers syntaxe concrète

Une fonction d'impression :

proposition.ml(fin)

```
let rec to_string p = match p with
| Vrai -> "1"
| Faux -> "0"
| Non (prop) -> "non("^(to_string prop)^")"
| Et(a,b) -> "("^(to_string a)^" . "^(to_string b)^")"
| Ou(a,b) -> "("^(to_string a)^" + "^(to_string b)^")"
| Implique(a,b) -> "("^(to_string a)^" => "^(to_string b)^")"
| Équivalent(a,b) -> "("^(to_string a)^" <=> "^(to_string b)^")"
| Variable s -> s

let print p = Format.print_string (to_string p)
```

Directive pour utiliser la fonction d’impression

Dans la boucle interactive

```
#install_printer print_proposition
```

Analyses lexicale et syntaxique

Syntaxe concrète $\Rightarrow$ Syntaxe abstraite

C’est plus compliqué. C’est ce que l’on appelle l’analyse lexicale et syntaxique.

Vous verrez dans le cours de compilation des outils `Lex` et `Yacc` qui ont leurs versions caml.

Nous utilisons ici les *flux* ou *stream* (in english).
Qui existent aussi en java.

Qui permettent de faire de l’analyse récursive gauche.

Présentation des flux

Introduction du type `stream`

– Un flux est une structure *paresseuse* et *consommée*

– A été conçu pour faire de l'analyse syntaxique.

– Possède une syntaxe concrète :

```
[< >]              un flux vide
[< s ; t>]         un flux qui contient deux flux
[< '1; '2; '3>]    un flux qui contient trois entiers
[< ''a' >]         un flux qui contient un caractère
```

– Depuis les dernières versions, obligent d'utiliser le pré-processeur `camlp4` donc, compilation : `ocamlc -c -pp camlp4.o toto.ml`
Dans la boucle interactive, faire `#load"camlp4o.cma";;`

Fonctions sur les flux

Dans le module Stream :

```
# let s = [< ''a'>];;
val s : char Stream.t = <abstr>
# let t = [< ''b';''c'>];;
val t : char Stream.t = <abstr>
# let u = [< s; t >];;
val u : char Stream.t = <abstr>
# Stream.next u;;
- : char = 'a'
# Stream.next s;;
- : char = 'a'
# Stream.next t;;
- : char = 'b'
# Stream.next u;;
- : char = 'c'
# Stream.next t;;
- : char = 'c'
# Stream.next t;;
Uncaught exception: Stream.Failure.
```

Des fonctions de conversions

```
# let fc = Stream.of_string "abc efg";;
val fc : char Stream.t = <abstr>
# Stream.next fc;;
- : char = 'a'
```

Il y en a trois autres :

`Stream.of_channel: in_channel -> char Stream.t` Qui renvoie le flux de caractères lus sur son argument.

`Stream.of_list: 'a list -> 'a Stream.t` qui renvoie le flux des éléments de la liste.

`Stream.from : (int -> 'a option) -> 'a Stream.t`

Stream.from f renvoie un flux construit par les applications successives de la fonction `f`. Le flux construit est : `[< f(0); f(1);f(2);.....>]`

Le résultat de `f(i)` est soit `Some(v)`, soit `None` spécifiant alors la fin du flux.

module Lexer

Flux et analyse lexicale

Il s'agit de vérifier qu'une chaîne de caractères utilise uniquement les mots licites du langage.

Les mots licites du langages sont appelés lexèmes et classés par catégorie.
On veut donc transformer une suite (un flux) de caractères en suite de lexèmes.
Nous proposons une implantation d'un analyseur lexicale *universel*. Nous aurions pu utiliser le module `Genlex`.

lexer.ml

```
type lexème =
  | MC of string     (* pour les mots clefs *)
  | Ident of string (* les identificateurs qui ne sont pas des mots clefs *)
  | Entier of int    (* les entiers *)
```

Lire un entier

```
let int_of_char c = Char.code(c) - Char.code('0')
```

```
let rec lire_int accu flux  = match flux with parser
| [< '('0' .. '9' as c) >] ->
    lire_int (10 * accu + int_of_char c) flux
| [<>] -> accu
```

notez le mot clef parser

le premier cas filtre seulement le premier caractère de flux, si ce premier caractère est un chiffre, alors on passe par ce cas, le caractère est consommé, l'appel récursif est fait sur ce qui reste dans flux.

Le deuxième cas de filtrage ne correspond pas à un flux vide. C'est le cas où le premier caractère n'a pas été consommé par les filtres précédents. Le flux peut ne pas être vide.

Utilisons pour filtrer un entier

```
# let s = Stream.of_string "123ajk34";;
val s : char Stream.t = <abstr>

# let x = lire_int 0 s;;
val x : int = 123

# Stream.next s;;
- : char = 'a'

# let y = lire_int 0 s;;
val y : int = 0
```

Lire un mot

Un tampon pour stocker le mot lu, on limite la taille d'un symbole :

```
let taille_maxi = 16
let tampon = String.create taille_maxi
```

Un mot est une suite de caractères alpha-numériques, de ' et de _ :

```
let rec lire_mot pos flux = match flux with parser
| [< '( 'A'..'Z'|'a'..'z'|'0'..'9'|'_'|'\''|'é'|'à'|'è'|'ù'|
       'â'|'ê'|'î'|'ô'|'û'|'ä'|'ë'|'ï'|'ö'|'ü'|'ç'|'É'|'À'|'È'|'Ù'|
       'Â'|'Ê'|'Î'|'Ô'|'Û'|'Ä'|'Ë'|'Ï'|'Ö'|'Ü'|'Ç' as c) >] ->
         if pos < taille_maxi then tampon.[pos] <- c;
         lire_mot (pos+1) flux
| [<>] -> String.sub tampon 0 (min pos taille_maxi)
```

Lire un symbole

Sur le même *moule*

```
let rec lire_symbole pos flux = match flux with parser
| [< '( '!'|'$'|'%'|'&'|'*'|'+'|'-'|'.'|'/'|':'|';'|'<'|'>'|'='|
        '?'|'@'|'^'|'|'|'~' as c) >] ->
          if pos < taille_maxi then tampon.[pos] <- c;
          lire_symbole (pos+1) flux
| [<>] -> String.sub tampon 0 (min pos taille_maxi)
```

Prévoyons des commentaires simples

Un commentaire commence par un # et se termine en fin de ligne :

```
let rec lire_comment flux = match flux with parser
| [< ''\n' >] -> ()
| [< 'c >] -> lire_comment flux
```

Utilisons

```
# let fl =Stream.of_string"Bonjour␣123␣#ceci␣est␣un␣commentaire\n␣Au␣revoir";;
val fl : char Stream.t = <abstr>

# let m1 = lire_mot 0 fl;;
val m1 : string = "Bonjour"

# Stream.next fl;;
- : char = ' '

# let i1 = lire_int 0 fl;;
val i1 : int = 123

# Stream.next fl;;
- : char = ' '

# let c = lire_comment fl;;
val c : unit = ()
```

Utilisons(suite)

```
# Stream.next fl;;
- : char = ' '

# let m2 = lire_mot 0 fl;;
val m2 : string = "Au"

# let m3 = lire_mot 0 fl;;
val m3 : string = ""
```

Il nous reste à sauter les *blancs* et à reconnaître si c'est un entier, un mot ou un commentaire qu'il faut lire.

Nous avons aussi à distinguer les mots clefs des autres identificateurs.

Distinguer mot clef et identificateur

Les mots clefs sont donnés, dans un premier temps, par une liste de mots :

```
let mc_ou_ident table_des_mots_clefs ident =
  if List.mem ident table_des_mots_clefs then MC ident
  else Ident ident
```

Il faut gérer aussi les mots clefs *mono-caractère* comme les "(" :

```
let mc_ou_erreur table_des_mots_clefs caract =
  let ident = Char.escaped caract in
  if List.mem ident table_des_mots_clefs then MC ident
  else raise (Stream.Error ident)
```

Lire un lexème

Nous lisons le premier caractère pour décider si c'est un espace, un ident, un mot clef ou un entier.

```
let rec lire_lexème table flux = match flux with parser
| [< '( ' ' | '\n' | '\r' | '\t' ) >] -> 
    lire_lexème table flux
| [< ''#' >] -> lire_comment flux; lire_lexème table flux
| [< '( 'A'..'Z'|'a'..'z'|'_'|'\''|'é'|'à'|'è'|'ù'|
       'â'|'ê'|'î'|'ô'|'û'|'ä'|'ë'|'ï'|'ö'|'ü'|'ç'|'É'|'À'|'È'|
       'Ù'|'Â'|'Ê'|'Î'|'Ô'|'Û'|'Ä'|'Ë'|'Ï'|'Ö'|'Ü'|'Ç' as c) >] ->
         tampon.[0] <- c;
         mc_ou_ident table (lire_mot 1 flux)
| [< '( '!'|'$'|'%'|'&'|'*'|'+'|'.'|'/'|':'|';'|'<'|'>'|'='|
       '?'|'@'|'^'|'|'|'~' as c) >] ->
         tampon.[0] <- c;
         mc_ou_ident table (lire_symbole 1 flux)
| [< '('0' .. '9' as c) >] -> Entier (lire_int (int_of_char c) flux)
```

Lire un lexème (suite)

```
| [< ''-' >] -> (* distinguer le - unaire du - binaire *)
    begin match flux with parser
    |  [< '('0' .. '9' as c) >] ->
        Entier (-(lire_int (int_of_char c) flux))
    |[<>] ->
        tampon.[0] <- '-';
        mc_ou_ident table (lire_symbole 1 flux)
    end
| [<'c>] -> mc_ou_erreur table c
```

L'analyseur lexicale peut donc s'écrire en utilisant la fonction Stream.from qui construit le flux des applications :

```
let analyseur mots_clefs flux =
  Stream.from
    (function i ->
       try Some(lire_lexème mots_clefs flux ) with
       | Stream.Failure -> None)
```

Utilisons l'analyseur lexicale

```
# let lf = lexeur (Stream.of_string "␣(p␣et␣q)␣ou␣1");;
val lf : lexème Stream.t = <abstr>
# let l1 = Stream.next lf;;
val l1 : lexème = MC "("
# let l2 = Stream.next lf;;
val l2 : lexème = Ident "p"
# let l3 = Stream.next lf;;
val l3 : lexème = MC "et"
# let l4 = Stream.next lf;;
val l4 : lexème = Ident "q"
# let l5 = Stream.next lf;;
val l5 : lexème = MC ")"
# let l6 = Stream.next lf;;
val l6 : lexème = MC "ou"
# let l7 = Stream.next lf;;
val l7 : lexème = MC "1"
# let l8 = Stream.next lf;;
Uncaught exception: Stream.Failure.
```

module Parser

Analyse syntaxique

À partir d'un flux de lexèmes, on veut produire un arbre de syntaxe abstraite ssi les règles de la syntaxe sont vérifiées.
Nous voulons de plus implanter les règles de précédence des opérateurs logiques.
L'ordre de précédence est celui-ci :

```
non; et; ou; =>; <=>
```

Nous utilisons une possibilité très puissante des flux. Il est possible de faire appel à un analyseur de flux, *au sein* d'un motif de flux.

BNF de la syntaxe du langage des propositions

Grammaire non ambigüe factorisée à gauche :

- `<proposition0>` : := *Entier 0* | *Entier 1* |*Ident s* | (`<proposition5>`)
- `<proposition1>` : := non `<proposition0>`
- `<proposition2>` : := `<proposition1>` `<reste2>`
- `<reste2>` : := et `<proposition1>` `<reste2>` | ϵ
- `<proposition3>` : := `<proposition2>` `<reste3>`
- `<reste3>` : := ou `<proposition2>` `<reste3>` | ϵ
- `<proposition4>` : := `<proposition3>` `<reste4>`
- `<reste4>` : := => `<proposition3>` `<reste4>` | ϵ
- `<proposition5>` : := `<proposition4>` `<reste5>`
- `<reste5>` : := <=> `<proposition4>` `<reste5>` | ϵ

Analyse respectant les précédences

⇒ Notez que ces fonctions d'analyse sont mutuellement récursives.

⇒ C'est la traduction directe de la grammaire BNF.

⇒ On nomme les résultats intermédiaires pour produire l'arbre de syntaxe.

Commençons par ouvrir pour éviter de toujours préfixer.

parser.ml

```
open Proposition
open Lexer
```

La fonction d'analyse

Le point d'entrée :

```
let rec lire_proposition f = proposition5 f
```

Lire une proposition de niveau 0 :

```
and proposition0 = parser
  | [< 'Ident s >] -> Variable s
  | [< 'Entier 1 >] -> Vrai
  | [< 'Entier 0 >] -> Faux
  | [< 'MC "("; p =lire_proposition  ;'MC ")" >] -> p
  |[<>] -> begin Printf.printf "Hasta␣La␣Vista\n"; exit 0 end
```

Lire une proposition de niveau 1 :

```
and proposition1 fl= match fl with  parser
  | [< 'MC "non" >] -> Non (proposition0 fl)
  |[<>] -> proposition0 fl
```

suite de l’analyse

```
and proposition2 = parser
  | [< p1 = proposition1;  q =(reste2 p1)  >] -> q
and reste2 p1 = parser
  | [< 'MC "et"; p2 = proposition1 ; q = (reste2 (Et(p1,p2))) >] -> q
  | [<>] -> p1
```

```
and proposition3 = parser
  | [< p1 = proposition2 ; q = (reste3 p1) >] -> q
and reste3 p1 = parser
  | [< 'MC "ou"; p2 = proposition2; q = (reste3 (Ou(p1,p2)))>] -> q
  | [<>] -> p1

and proposition4 = parser
  | [< p1 = proposition3 ; q = (reste4 p1) >] -> q
and reste4 p1 = parser
  | [< 'MC "=>"; p2 = proposition3 ; q = (reste4 (Implique(p1,p2)))>] -> q
  | [<>] -> p1

and proposition5 = parser
  | [< p1 = proposition4 ; q = (reste5 p1) >] -> q
and reste5 p1 = parser
  | [< 'MC "<=>"; p2 = proposition4; q = (reste5 (Équivalent(p1,p2)))>] -> q
  | [<>] -> p1
```

Écrire la fonction exportée

```
let analyse_lexicale_prop =
  analyseur ["et";"ou";"non";"=>";"<=>";"(";")"]
```

Analyser une chaîne c'est vérifier la validité des mots utilisés *analyse lexicale* puis vérifier que les règles de la grammaire sont respectées et produire un arbre de syntaxe abstraite.

D'où :

```
let proposition chaîne =
  lire_proposition(analyse_lexicale_prop (Stream.of_string chaîne))
```

L'interface, dans le fichier `parser.mli`

parser.mli

```
(** [Parser.proposition s] vérifie la correction lexicale et
    syntaxique de [s] et renvoie un arbre de syntaxe abstraite
    correspondant. *)
val proposition : string -> Proposition.t
```

module Maintauto

Lire et dire si c'est un théorème

maintauto.ml

```
open Proposition

let examine chaîne =
  let prop = Parser.proposition chaîne in
  let variables = variables_libres prop in
  try
    vérifie_tautologie prop variables;
    begin match variables with
    | [] -> print_string "Théorème: "
    | [v] -> print_string ("Théorème: pour toute proposition " ^v^ ", ")
    | _ ->
        print_string "Théorème: pour toutes propositions ";
        List.iter (Printf.printf "%s, ") variables
    end;
    print_string ("\n" ^ chaîne ^"\n")
```

Lire et dire si c'est un théorème(2)

```
with Réfutation liaisons ->
  print_string (chaîne ^" n'est pas un théorème. \n ");
  match liaisons with
  | [] -> print_string "la proposition est toujours fausse\n"
  | _ -> begin
      print_string "la proposition est fausse quand:\n ";
      List.iter
        (fun (v,b) ->
          if b then Printf.printf "%s est vraie\n" v
          else Printf.printf "%s est fausse\n" v)
        liaisons
  end
```

Une boucle d'interaction

```
let boucle () =
  try while true do
    print_string "calclog> "; examine (read_line())
  done
  with End_of_file -> exit 0
```

Mise en oeuvre

Un Makefile pour compiler le tout

```
MODULES= proposition. lexer. parser. maintauto.

SOURCES=$(MODULES:.=.ml)
BCOBJS=$(MODULES:.=.cmo)
OBJS=$(MODULES:.=.cmx)
EXEC=calclog

#*********Compilateurs et options de compilation

OCAMLC=ocamlc
OCAMLOPT=ocamlopt
OPTFLAGS= -pp camlp4o
BCFLAGS=-pp camlp4o
INCLUDES=

exec:$(BCOBJS)
        $(OCAMLC) $(BCFLAGS) -o $(EXEC) $(BCOBJS)

opt:$(OBJS)
        $(OCAMLOPT) $(OPTFLAGS) -o $(EXEC).opt $(OBJS)
```

Un Makefile pour compiler le tout(2)

```
.SUFFIXES:
.SUFFIXES: .ml .mli .cmo .cmi .cmx .mll .mly

.ml.cmo:
	$(OCAMLC) $(BCFLAGS) -c $<

.mli.cmi:
	$(OCAMLC) -c $<

.ml.cmx:
	$(OCAMLOPT) -c $(OPTFLAGS) $<

depend: $(SOURCES)
	ocamldep -pp camlp4o *.mli *.ml > .depend

clean:
	rm -f *.cm[iox] *~ .*~ *.o *.aux *.log \#*\#
allclean: clean
	rm -f $(EXEC) $(EXEC).opt

include .depend
```

On compile

```
$ touch .depend
$ make depend
ocamldep -pp camlp4o *.mli *.ml > .depend

$ make exec
ocamlc -c proposition.mli
ocamlc -pp camlp4o -c proposition.ml
ocamlc -pp camlp4o -c lexer.ml
ocamlc -c parser.mli
ocamlc -pp camlp4o -c parser.ml
ocamlc -pp camlp4o -c maintauto.ml
ocamlc -pp camlp4o -o calclog proposition.cmo lexer.cmo parser.cmo ma
```

On essaye

```
$ ./calclog
calclog> p => p
Théorème: pour toute proposition p,
p => p

calclog> ( p ou q) et (non p) => q
Théorème: pour toutes propositions q, p,
( p ou q) et (non p) => q

calclog> (p => q) <=> ((non q) => (non p))
Théorème: pour toutes propositions q, p,
(p => q) <=> ((non q) => (non p))
```

On essaye (fin)

```
calclog> (p et non q) => (p et q)
(p et non q) => (p et q) n'est pas un théorème.
 la proposition est fausse quand:
 p est vraie
q est fausse
calclog> p ou 1
Théorème: pour toute proposition p,
p ou 1
calclog> ((p => q) => p) => p
Théorème: pour toutes propositions q, p,
((p => q) => p) => p
```

La structure du logiciel

obtenue par :

```
$ ocamldoc -pp camlp4o -dot -dot-include-all *.mli *.ml -o calclog.do
$ dot -Tps  calclog.dot -o calclogdot.ps
```