Rappresentazione delle formule e definizioni ricorsive su di esse

Osservazione: non rappresentiamo la doppia implicazione

```
type form =
    True
  | False
  | Prop of string
  | Not of form
  | And of form * form
  | Or of form * form
  | Imp of form * form
```

$\neg(p \rightarrow q)$:

```
Not
(Imp
  (Prop "p",Prop "q"))
```

Rappresentazione delle formule e definizioni ricorsive su di esse

Osservazione: non rappresentiamo la doppia implicazione

```
type form =
    True
  | False
  | Prop of string
  | Not of form
  | And of form * form
  | Or of form * form
  | Imp of form * form
```

$\neg(p \rightarrow q)$:

```
Not
(Imp
  (Prop "p",Prop "q"))
```

```
(* atomlist :
       form -> string list *)
let rec atomlist = function
    True | False -> []
  | Prop s -> [s]
  | Not f -> atomlist f
  | Or(f1,f2) ->
      union (atomlist f1)
            (atomlist f2)
  | And(f1,f2) ->
      union (atomlist f1)
            (atomlist f2)
  | Imp(f1,f2) ->
      union (atomlist f1)
            (atomlist f2)
```

Moduli e file

Vogliamo arrivare a scrivere un **parser** per le formule proposizionali.
A questo scopo conviene racchiudere la dichiarazione del tipo **form** in un file a parte, che andrà a costituire un modulo, come i moduli della libreria standard di OCaml.

Creiamo il file **language.ml** che contiene soltanto la dichiarazione del tipo **form** (**CODICE/11-logica/language.ml**).

Occorre poi **compilare** il modulo, dando il comando (da una command shell):

```
> ocamlc -c language.ml
```

che genera i file **language.cmi** (l'interfaccia compilata del modulo) e **language.cmo** (bytecode del modulo **Language**).

Per caricare in memoria il modulo **Language**:

```
# #load "language.cmo";;
```

Ma i costruttori sono relativi al modulo **Language**:

```
# Prop "p";;      ====> Unbound constructor Prop
# Language.Prop "p";;
- : Language.form = Language.Prop "p"
```

Aprire i moduli

Per evitare di scrivere **Language.Prop, Language.And**, ..., occorre “aprire” il modulo:

```
# open Language;;
# Prop "p";;
- : Language.form = Prop "p"
```

D’ora in poi assumiamo di scrivere le nostre definizioni in un file (**logic.ml**) che inizia con:

```
#load "language.cmo";;
open Language
```

(il file, ovviamente, deve stare nella stessa directory in cui si trovano **language.cmi** e **language.cmo**).

Stampa di una formula

Vai al codice

file: CODICE/11-logica/logic.ml

Un parser per formule proposizionali

Utilizziamo i **costruttori di parser**: **ocamllex** e **ocamlyacc**.

Generatore di parser (compiler-compiler): strumento per la generazione del codice sorgente di un parser, a partire dalla descrizione data nella forma di grammatica (in genere BNF).

I nomi derivano dai due più noti generatori di parser, **Lex** e **Yacc**.
Lex: genera codice C per un *lex*ical analyzer.

Yacc: "Yet Another Compiler Compiler!". A partire dalla grammatica, genera un *parser* o analizzatore sintattico (in C): la sequenza di *token* prodotta dall'analisi lessicale è organizzata in un albero sintattico.

In OCaml: **ocamllex** produce il codice OCaml dell'analizzatore lessicale corrispondente ai pattern specificati in un file con estensione **mll**.

ocamlyacc produce il codice OCaml di un parser, corrispondente alla grammatica specificata nel file (con estensione **mly**).

La dichiarazione dei simboli terminali

Un file **mly** inizia con un **header** (codice OCaml che verrà copiato all'inizio del file con il parser) e la dichiarazione dei simboli terminali (token) della grammatica.

Nel nostro caso, il file **parser.mly** (in CODICE/11-logica) inizia con:

```
%{ open Language %}

%token <string> ATOM
%token TRUE FALSE NOT AND OR IMP IFF RPAREN LPAREN EOF END
%left IFF
%left IMP
%left OR
%left AND
%nonassoc NOT
```

Le dichiarazioni **%left, %right** e **%nonassoc** associano la precedenza e l'associatività ai simboli corrispondenti.
Ogni simbolo ha precedenza più alta di quelli che lo precedono.

Dichiarazione degli entry point

Il file prosegue con la dichiarazione degli **entry point** della grammatica:

```
%start formula
%type <Language.form> formula
```

Per ogni entry point, il modulo prodotto da **ocamlyacc** avrà una funzione con lo stesso nome.

```
val formula :
  (Lexing.lexbuf -> token)     (* tipo della funzione
                                  di lexing *)
            -> Lexing.lexbuf
                     (* tipo dei lexer buffer,
                        tipo astratto del modulo Lexing *)
            -> Language.form
```

La grammatica

Ogni simbolo non terminale deve essere definito mediante regole sintattiche, che associano un **attributo** (espressione OCaml) a ogni caso di espansione.

```
formula :
    ATOM                    { Prop($1) }
  | TRUE                    { True }
  | FALSE                   { False }
  | NOT formula             { Not $2}
  | formula IFF formula     { And(Imp($1,$3),Imp($3,$1)) }
  | formula IMP formula     { Imp($1,$3) }
  | formula OR formula      { Or($1,$3) }
  | formula AND formula     { And($1,$3) }
  | LPAREN formula RPAREN { $2 } ;
```

A ogni sequenza di simboli (parte sinistra) è associato un “attributo semantico”, nella parte destra: questo sarà valutato a riportato come valore nel caso corrispondente.

Nella parte destra: **$1** corrisponde all’attributo del primo simbolo della corrispondente parte sinistra, **$2** quello del secondo simbolo, ecc.

L'analisi lessicale (I)

I file con estensione **mll** contengono un insieme di espressioni regolari con associata azione semantica (un **Parser.token**).

Vai al codice

CODICE/11-logica/lexer.mll

Il file inizia con un *header*:

```
{      open Parser     }
```

E la definizione dell'*entry point* **token**:

```
rule token = parse              (*  keywords *)
  .........
```

Per ogni entry point (possono essercene diversi), il file **lexer.ml** prodotto da **ocamllex** conterrà la definizione di una corrispondente funzione di *lexing*:

```
val token : Lexing.lexbuf -> Parser.token
```

L'analisi lessicale (II)

```
rule token = parse
  [' ' '\t' '\n' ] { token lexbuf }
| "T" { TRUE }
| "F" { FALSE }
.....
```

La funzione è definita per casi: a sinistra c'è un'espressione regolare, a destra l'attributo semantico associato.

token lexbuf: applicare la funzione **token** al (resto del) lexer buffer.

```
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*
    { ATOM(Lexing.lexeme lexbuf) }
```

A sinistra: espressione regolare, a destra: **Lexing.lexeme lexbuf** riporta la stringa corrispondente all'espressione regolare letta.

I moduli

Abbiamo compilato il file **language.ml**, generando il file **language.cmo** (il modulo **Language**):

```
ocamlc -c language.ml
```

Ora è possibile generare il parser:

```
ocamlyacc parser.mly
ocamllex lexer.mll
```

OCamlyacc genera **parser.ml** (modulo) e **parser.mli** (l'interfaccia); OCamllex genera **lexer.ml**.
Ora possiamo compilare questi file:

```
ocamlc -c parser.mli parser.ml lexer.ml
```

generando i corrispondenti file **cmi** e **cmo**.
Per utilizzarli, da una shell OCaml:

```
#load "language.cmo";;
#load "parser.cmo";;
#load "lexer.cmo";;
```

Ora abbiamo a disposizione i **moduli Language**, **Parser** e **Lexer**.

Uso del parser (e dei moduli) in modalità interattiva

Lexer.token: Lexing.lexbuf -> Parser.token
Parser.formula: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Language.form
Parser.formula Lexer.token: Lexing.lexbuf -> Language.form

Per analizzare una stringa, si deve trasformare in un Lexing.lexbuf
Lexing.from_string: string -> lexbuf

```
#load "language.cmo";;
open Language;;
#load "parser.cmo";;
#load "lexer.cmo";;

(* parse: string -> Language.form *)
(* solleva Parsing.parse_error in caso di fallimento *)
let parse stringa =
  let lexbuf = Lexing.from_string stringa
  in Parser.formula Lexer.token lexbuf
```

Vai al codice

Compilazione separata e codice eseguibile (ocamlc)

I nomi dei file

- **<nome-file>.mli**: codice sorgente per interfacce. La compilazione produce un'interfaccia compilata nel file **<nome-file>.cmi**
- **<nome-file>.ml**: codice sorgente per unità di compilazione. La compilazione produce bytecode nel file **<nome-file>.cmo**

Per compilare separatamente un'unità o un'interfaccia:

```
ocamlc -c <nome-file>.mli
ocamlc -c <nome-file>.ml
```

Mediante *linking* del bytecode (file **.cmo**) si ottiene un programma eseguibile indipendente.

```
ocamlc -o <nome-programma> <file_1.cmo> ... <file_n.cmo>
ocamlc -o <nome-programma> <f1.mli> <f1.ml> ... <fn.ml>
```

Per eseguire il programma, si invoca l'interprete di bytecode:

```
ocamlrun <nome-programma>
```

Compilazione di programmi stand-alone

```
ocamlc -custom -o <nome-programma> ........
```

(per la creazione di programma stand-alone su sistemi Windows, vedere il sito di OCaml)

Moduli e file

Contenuto del file **queue.mli**:

```
(* interfaccia
   per le code *)
type  'a t
exception Empty
val empty : 'a t
val dequeue : 'a t -> 'a t
val head : 'a t -> 'a
val enqueue :
     'a t -> 'a -> 'a t
val null : 'a t -> bool
```

Contenuto del file **queue.ml**:

```
type 'a t =
    'a list * 'a list
exception Empty
let norm = function ...
let empty = ([],[])
let dequeue = ...
let head = ...
let enqueue (h,t) x = ...
let null q = q = ([],[])
```

Il compilatore riconosce un **modulo** **Queue**, la cui **segnatura** è quella specificata in **queue.mli**.

queue.mli definisce l'interfaccia del file di nome **queue.ml**.

Il **tipo concreto** utilizzato per rappresentare le code è nascosto.

(vedere capitolo 5 del libro)

Accesso al contenuto di una unità di compilazione

Le altre unità di compilazione (altri file), possono accedere al contenuto del file **queue.mli** (al modulo **Queue**) nello stesso modo in cui accedono ai moduli della libreria standard:

```
if Queue.null (Queue.enqueue Queue.empty 3)
then ...
```

Ovviamente “vedono” soltanto ciò che è specificato nell’interfaccia.

Per caricare bytecode in modalità interattiva:

```
#load "queue.cmo";;
```

Esecuzione di un programma compilato

Se vogliamo eseguire un programma creato con

```
ocamlc -o <nome-programma> ....
```

il file “principale” (quello dal quale non dipende nessun altro), deve contenere una richiesta di valutazione che dia l’avvio all’esecuzione.
Ad esempio:

```
let _ = <Espressione-da-valutare>
```

È anche possibile scrivere programmi che accettano argomenti: vedere il modulo **Arg** della libreria standard.

Esempio

Abbiamo scritto i file **language.ml**, **lexer.mll** e **parser.mly**, abbiamo creato il parser (i file **lexer.ml, parser.mli** e **parser.mly**):

```
ocamlc -c language.ml
ocamlyacc parser.mly
ocamllex lexer.mll
```

Ora scriviamo un file **prova.ml** con il seguente contenuto:

Vai al codice

CODICE/11-logica/prova.ml

Compiliamo, creando il file **prova**, e possiamo eseguirlo:

```
ocamlc -o prova language.ml parser.mli parser.ml \
          lexer.ml prova.ml
ocamlrun prova
```

Interpretazioni e verità in OCaml

Rappresentazione di un'interpretazione come una lista di stringhe. Un atomo `Prop p` è vero nell'interpretazione rappresentata dalla lista `emme` se e soltanto se `p` è un elemento di `emme`.

```
type interpretation = string list

(* models: form -> interpretation -> bool *)
(* models f emme = true se e solo se f e' vera in emme *)
let rec models f emme =
  match f with
    True -> true
  | False -> false
  | Prop name -> List.mem name emme
  | Not f1 -> not(models f1 emme)
  | And(f1,f2) -> models f1 emme && models f2 emme
  | Or(f1,f2) -> models f1 emme || models f2 emme
  | Imp(f1,f2) -> not(models f1 emme) || models f2 emme
```

Validità di una formula e costruzione della sua tavola di verità

- Collezionare tutte le variabili proposizionali che occorrono nella formula:

  atomlist: form $\rightarrow$ string list
- Costruire tutte le possibili interpretazioni di tali variabili:

  powerset: α list $\rightarrow$ α list list
- Validità: determinare se la formula è vera in ogni interpretazione $\mathcal{M}$:

  valid: form $\rightarrow$ bool
- Costruzione della tavola di verità: per ciascuna interpretazione $\mathcal{M}$, determinare il valore della formula in $\mathcal{M}$:

  truthtable: form $\rightarrow$ ((interpretation $\times$ bool) list

Vai al codice

Controllo di validità mediante le tabelle di verità

Esempio: per dimostrare che la legge di Pierce è una tautologia, possiamo costruirne la tavola di verità e verificare che è vera in ogni interpretazione

```
# let pierce = parse "((A->B)->A)->A";;
val pierce : Language.form =
  Imp (Imp (Imp (Prop "A", Prop "B"), Prop "A"), Prop "A")
# truthtable pierce;;
- : (string list * bool) list =
[([], true); (["A"], true); (["B"], true);
 (["B"; "A"], true)]
```

In generale

```
(* valid_tt : form -> bool *)
let valid_tt f =
   List.for_all (function (_,value) -> value)
                (truthtable f)
```

Oppure usiamo **valid**: form $\rightarrow$ bool

Formule valide e contraddittorie

```
(* valid : form -> bool *)
(* valid f = f e' valida *)
let valid f =
  List.for_all (models f) (powerset (atomlist f));;


(* non : ('a -> bool) -> 'a -> bool *)
let non p x = not(p x)

(* contradiction : form -> bool *)
(* contradiction f = f e' una contraddizione *)
let contradiction f =
  List.for_all (non (models f))
               (powerset (atomlist f))
```

La validità si può anche verificare utilizzando le tavole di verità

Vai al codice

Soddisfacibilità, equivalenza logica, conseguenza logica, trasformazione in forma normale negativa

Vai al codice