

OCaml (autore: Vittorio Albertoni)

Premessa

A metà anni '70 del secolo scorso, subito dopo la nascita dei linguaggi a paradigma procedurale/imperativo Pascal e C, un'equipe guidata da Robin Milner presso l'Università di Edimburgo creò il linguaggio ML (che sta per MetaLanguage), ispirato al paradigma funzionale introdotto anni prima dal linguaggio Lisp.

In realtà ML è un linguaggio funzionale impuro in quanto consente anche la programmazione imperativa.

Questa stessa caratteristica la si trova nel linguaggio CAML, nato nel 1984 nei laboratori dell'INRIA (un importante istituto di ricerca francese) per diretta derivazione dal linguaggio ML. Il nome del linguaggio è l'acronimo di Categorical Abstract Machine Language.

Nel 1996 un'equipe guidata da Xavier Leroy, Jérôme Vouillon e Damien Doligez, sempre nei laboratori dell'INRIA, aggiunse al linguaggio CAML funzionalità orientate agli oggetti e nacque così un linguaggio multi-paradigma chiamato OCaml, come acronimo di Objective CAML.

OCaml unisce i paradigmi di programmazione funzionale, imperativo e orientato agli oggetti, ed è noto per le sue elevate prestazioni e l'efficienza del suo sistema di tipi.

In omaggio alle lontane origini, i source scritti con questo linguaggio vengono identificati dall'estensione `.ml`.

Si tratta di un prodotto open source distribuito con licenza GNU, nel tempo arricchito da molteplici package che ne estendono la potenzialità.

In questo manualetto illustrerò le basi del linguaggio, a portata dei dilettanti cui mi rivolgo, nell'intento di rimediare al fatto che la documentazione ufficiale esistente è molto criptica, non soltanto a causa della lingua inglese con cui è scritta.

Non mancherà un accenno, sul finale, ad alcuni package di estensione, il cui utilizzo è un tantino più difficoltoso.

Indice

1	Installazione	3
2	Come funziona	3
3	Tipi	4
3.1	Tipi fondamentali	4
3.2	Tipi contenitore	5
4	Variabili	6
5	Operatori	7
6	Funzioni intrinseche	7
6.1	Funzioni trigonometriche	8
6.2	Funzioni iperboliche	8
6.3	Funzioni varie:	8
6.4	Funzioni di valore assoluto:	8
6.5	Funzioni per la conversione di tipo:	8
6.6	Funzioni di Input/Output	9
7	Creare funzioni	9
8	Strutture di controllo	9
8.1	if-then-else	10
8.2	for	10
8.3	while	10
9	Classi	11
10	Struttura del programma	12
11	Esempi di programmi	12
12	Estensioni del linguaggio	13
13	Esempi di programmi con utilizzo di estensioni	15

1 Installazione

Tutto ciò che c'è da sapere su OCaml si trova all'indirizzo <https://ocaml.org/>.

A cominciare dalle istruzioni per l'installazione, accessibili cliccando sul pulsante INSTALL. OCaml è disponibile per tutti i sistemi operativi Linux, Windows e Mac.

Chi lavora su Linux può installare OCaml dal repository della distro, sapendo che:

- . per avere semplicemente la base del linguaggio e il compilatore basta installare `ocaml`,
- . se si vuole lavorare con packages di estensione del linguaggio occorre installare anche `ocaml-findlib` e il gestore di pacchetti `opam`.

Se si sceglie questa seconda strada e si lavora su Debian o derivate (Ubuntu, MXLinux, ecc.) basta installare `opam`.

Se si lavora su distro che fanno capo al repository Red Hat (Fedora), per avere tutto occorre installare `ocaml`, `ocaml-findlib` e `opam`.

Una volta installato `opam` occorre inicializzarlo inserendo a terminale i comandi

```
opam init
eval $(opam env)
```

Sempre se si prevede di lavorare con packages di estensione del linguaggio è consigliabile installare un pacchetto che consiste in una shell dedicata al linguaggio OCaml, che si chiama `utop`. Possiamo installarla con `opam` con il comando

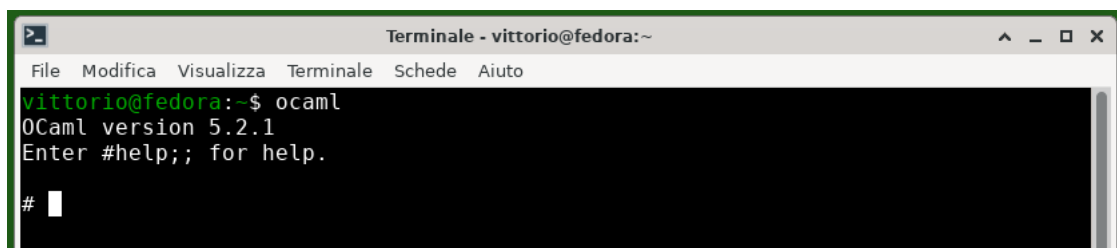
```
opam install utop.
```

Se apriamo la pagina LEARN troviamo indicazioni sulla documentazione esistente per imparare ed utilizzare il linguaggio.

2 Come funziona

Per familiarizzare con il linguaggio o per utilizzarlo interattivamente come si trattasse di una calcolatrice abbiamo a disposizione la shell.

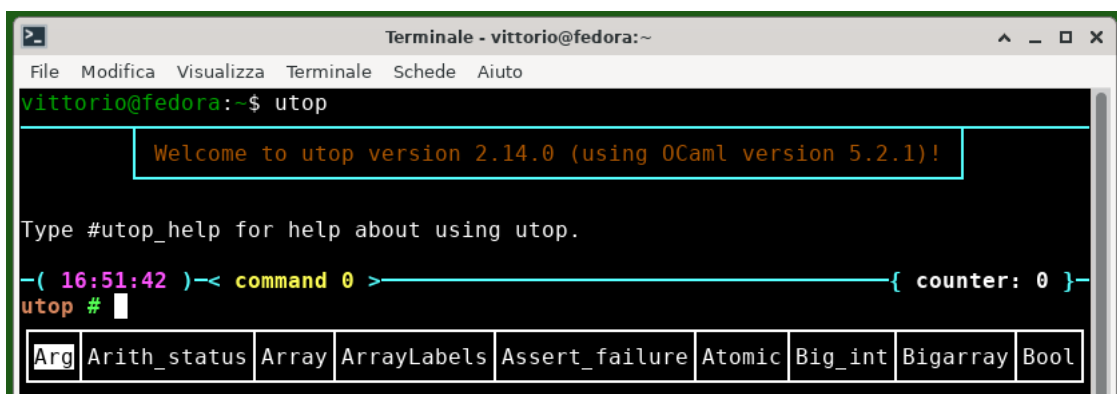
Possiamo avvalerci di una shell semplice semplice aprendola nel terminale con il comando `ocaml`



```
Terminale - vittorio@fedora:~
File Modifica Visualizza Terminale Schede Aiuto
vittorio@fedora:~$ ocaml
OCaml version 5.2.1
Enter #help;; for help.
#
```

Se abbiamo installato il relativo pacchetto possiamo aprire la più sofisticata shell `utop` con il comando

```
utop
```



```
Terminale - vittorio@fedora:~
File Modifica Visualizza Terminale Schede Aiuto
vittorio@fedora:~$ utop
Welcome to utop version 2.14.0 (using OCaml version 5.2.1)!
Type #utop_help for help about using utop.
-( 16:51:42 )-< command 0 > { counter: 0 }-
utop #
Arg Arith_status Array ArrayLabels Assert_failure Atomic Big_int Bigarray Bool
```

In ogni caso, inserendo una istruzione in linguaggio OCaml al prompt # , chiudendola con un doppio punto e virgola (; ;) e premendo INVIO, otteniamo nella riga successiva il risultato dell'istruzione.

Per elaborazioni più complesse dobbiamo scrivere le istruzioni in un file di testo e memorizzare il file con estensione .ml.

Per scrivere questo file può servire un qualsiasi editor di testo (non un word processor). Una buona scelta potrebbe essere l'editor Geany, che riconosce il linguaggio OCaml, ne colora le parole chiave ed è predisposto per l'eventuale compilazione in bytecode.

Il file che abbiamo così creato può essere eseguito come fosse uno script con il comando a terminale

```
ocaml <file.ml>
```

utilizzando l'interprete di OCaml.

Per esecuzioni più veloci, che peraltro si percepiscono e apprezzano soltanto in presenza di programmi di una certa complessità e pesantezza, possiamo compilare in bytecode il programma con il comando a terminale

```
ocamlc -o <eseguibile> <file.ml>
```

utilizzando il compilatore di OCaml.

L'eseguibile in bytecode richiede meno tempo di compilazione ma è più lento nell'esecuzione, sempre essendo molto più veloce che non nel caso di uso dell'interprete.

Esso si può lanciare richiamando la macchina virtuale che lo esegue, con il comando a terminale

```
ocamlrun <eseguibile>
```

oppure semplicemente richiamandone il nome.

Per esecuzioni ancora più veloci possiamo compilare in codice nativo il programma con il comando

```
ocamlopt -o <eseguibile> <file.ml>
```

L'eseguibile in codice nativo richiede più tempo di compilazione ma è di velocità insuperabile nell'esecuzione.

Purtroppo non si tratta di un eseguibile come quelli generati con i linguaggi Pascal o C che hanno massima portabilità. Nel caso di OCaml la compilazione avviene nativamente per l'architettura e il sistema operativo della macchina su cui lavoriamo.

Esso si lancia semplicemente richiamandone il nome.

3 Tipi

OCaml è un linguaggio fortemente tipizzato a tipizzazione statica.

I tipi dei dati elaborabili con questo linguaggio sono i seguenti.

3.1 Tipi fondamentali

Tipi numerici

`int` numeri interi,

`float` numeri con decimali (floating point).

I float sono a doppia precisione. Pertanto contengono almeno 15 cifre esatte.

Gli int ereditano la precisione dal sistema operativo e se ne può calcolare il valore massimo elevando 2 ad un esponente pari ai bit del sistema operativo meno 2 (in quanto impegnati altrimenti), il tutto meno 1.

Pertanto:

. in un sistema operativo a 32 bit il massimo intero trattabile è

$$2^{30} - 1 = 1.073.741.823$$

. in un sistema operativo a 64 bit il massimo intero trattabile è

$$2^{62} - 1 = 4.611.686.018.427.387.903$$

Come dire che in un sistema operativo a 32 bit non riusciamo a calcolare il fattoriale di 13 e in un sistema operativo a 64 bit non riusciamo a calcolare il fattoriale di 21.

Tipi carattere

`char` per un carattere singolo racchiuso tra apici semplici (' e '),
`string` per una stringa di caratteri racchiusi tra apici doppi (" e ").

Tipo booleano

`bool` rappresenta i valori booleani `true` e `false`.

3.2 Tipi contenitore

I tipi contenitore sono strutture di dati.

Tupla

La tupla è una collezione immutabile di valori di tipo arbitrario, separati da virgole.

Viene rappresentata con i valori racchiusi tra parentesi tonde, ma può essere costruita senza indicare le parentesi.

Lista

La lista è una collezione immutabile di valori numerici uguali o superiori a 0, separati da punto e virgola e racchiusi tra parentesi quadre ([e]).

Viene rappresentata con i valori racchiusi tra parentesi quadre e si costruisce indicando le parentesi quadre.

Array

L'array è una collezione di valori dello stesso tipo, separati da punto e virgola e racchiusi tra parentesi quadre e carattere | ([| e |]).

Viene rappresentato con gli stessi caratteri e la sintassi per costruirlo è
`[| ...; ...; ... |]`.

Per costruire un array abbiamo a disposizione la funzione `make` del modulo `Array`, incluso nel linguaggio base

```
Array.make <elementi> <valore>
```

crea un array di `n` `<elementi>` assegnando inizialmente a tutti il valore `<valore>`.

Gli elementi dell'array così inizializzato sono indicizzati partendo da zero e possono essere modificati con la seguente sintassi

```
<nome_array>.( <indice> ) <- <valore>
```

e possono essere letti con la seguente sintassi

```
<nome_array>.( <indice> )
```

Un array di array è una matrice e per costruirla abbiamo a disposizione la funzione `make_matrix` del modulo `Array`

```
Array.make_matrix <righe> <colonne> <valore>
```

crea una matrice di `n` `<righe>` e `n` `<colonne>` assegnando inizialmente a tutti gli elementi il valore `<valore>`.

Gli elementi della matrice così inizializzata sono indicizzati partendo da zero e possono essere modificati con la seguente sintassi

```
<nome_matrice>.( <indice_riga> ).( <indice_colonna> ) <- <valore>
```

e possono essere letti con la seguente sintassi

```
<nome_matrice>.( <indice_riga> ).( <indice_colonna> )
```

4 Variabili

Come in qualsiasi linguaggio di programmazione, la variabile è una posizione di memoria in cui inseriamo un valore per averlo a disposizione per le elaborazioni previste nel programma.

Per default le variabili del linguaggio OCaml non sono modificabili, cioè non è possibile riassegnare ad esse un valore diverso da quello con cui sono state create, per cui è improprio chiamarle variabili.

Esiste tuttavia la possibilità di creare variabili mutabili, in tutto simili alle variabili che troviamo negli altri linguaggi di programmazione.

Vediamo come funziona il tutto.

La sintassi per creare una variabile è

```
let <nome> = <valore>
```

In realtà con questa istruzione non creiamo una variabile ma creiamo un binding attraverso il quale leghiamo il <nome> al <valore>. Come praticamente avviene per quelle che in altri linguaggi si chiamano costanti.

Il linguaggio, in base al <valore> indicato, assegna automaticamente il tipo alla «variabile» così creata.

Se scriviamo nella shell di OCaml l'istruzione

```
let a = 6;;
```

otteniamo in risposta

```
val a : int = 6
```

cioè la conferma che il valore legato al nome a è di tipo intero ed è 6.

Con il comando a otteniamo questo valore.

Se vogliamo che a abbia valore diverso non possiamo semplicemente assegnare il valore diverso alla variabile a che abbiamo creato ma dobbiamo crearne un'altra, con lo stesso nome e con la stessa sintassi, inizializzandola con il valore diverso, che può essere anche di altro tipo.

Per creare una variabile mutabile, alla quale, cioè possa essere assegnato un valore diverso da quello della inizializzazione, dobbiamo far precedere al valore assegnato la parola chiave `ref`.

Con l'istruzione

```
let a = ref 6;;
```

otteniamo in risposta

```
val a : int ref = {contents = 6}
```

A questa variabile possiamo assegnare un valore diverso da 6 con l'istruzione

```
a := <valore>
```

ovviamente indicando un valore dello stesso tipo di quello originariamente assegnato con `ref`.

Con il comando `!a` otteniamo questo valore.

Esempio:

Supponiamo di creare una variabile contatore per governare un ciclo e di dover incrementare il suo valore di 1 ad ogni ciclo.

Se abbiamo creato il contatore con

```
let contatore = 1
```

lo incrementiamo con

```
let contatore = contatore + 1
```

creando ogni volta una nuova variabile contenente il valore della vecchia aumentato di 1.

Se abbiamo creato il contatore con

```
let contatore = ref 1
```

lo incrementiamo con

```
contatore := !contatore + 1
```

mantenendo la vecchia variabile e modificandone il valore aumentandolo di 1.

Un tantino più complicato che in altri linguaggi ma, tutto sommato, gestibile.

5 Operatori

Gli operatori collegano tra loro operandi di varia natura in espressioni che forniscono un risultato.

Operatori per valori numerici

In ordine di esecuzione, per operare con numeri interi abbiamo i seguenti:

- * per la moltiplicazione
- / per la divisione (intesa con risultato intero senza i decimali)
- mod per il resto della divisione intera
- + per la somma
- per la sottrazione

Sempre in ordine di esecuzione, per operare con numeri di tipo float abbiamo i seguenti:

- ** per l'elevamento a potenza
- *. per la moltiplicazione
- /. per la divisione
- +. per la somma
- . per la sottrazione

Notare come, per ottenere il quadrato di 2 non possiamo scrivere `2**2` ma dobbiamo scrivere `2.**2.`, indicando il numero 2, sia come base che come esponente, con tipo float. Ciò in quanto, per i numeri interi, non esiste un operatore per l'elevamento a potenza.

Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano.

Sono i seguenti e si applicano tra valori dello stesso tipo.

- = uguale,
- <> non uguale,
- < minore,
- <= minore o uguale,
- > maggiore,
- >= maggiore o uguale,

Operatori logici

Sono i seguenti.

- && che sta per and,
- || che sta per or,
- not che sta per not.

Operatori per stringhe

- ^ per concatenare stringhe.

Operatori per liste

- @ per concatenare liste,
- :: per creare una nuova lista aggiungendo un primo elemento a una lista già esistente.

6 Funzioni intrinseche

Oltre agli operatori il linguaggio ci fornisce altre funzioni preconfezionate che possiamo utilizzare per le nostre elaborazioni.

6.1 Funzioni trigonometriche

Con l'argomento x di tipo float abbiamo:

`sin(x)` ritorna il seno di x radianti

`cos(x)` ritorna il coseno di x radianti

`tan(x)` ritorna la tangente di x radianti

`asin(x)` ritorna l'arcoseno di x in radianti

`acos(x)` ritorna l'arcocoseno di x in radianti

`atan(x)` ritorna l'arcotangente di x in radianti

Se l'argomento è un numero positivo possiamo omettere le parentesi.

6.2 Funzioni iperboliche

Con l'argomento x di tipo float abbiamo:

`sinh(x)` ritorna il seno iperbolico di x radianti

`cosh(x)` ritorna il coseno iperbolico di x radianti

`tanh(x)` ritorna la tangente iperbolica di x radianti

`asinh(x)` ritorna l'arcoseno iperbolico di x in radianti

`acosh(x)` ritorna l'arcocoseno iperbolico di x in radianti

`atanh(x)` ritorna l'arcotangente iperbolica di x in radianti

Se l'argomento è un numero positivo possiamo omettere le parentesi.

6.3 Funzioni varie:

Con l'argomento x di tipo float abbiamo:

`sqrt(x)` per la radice quadrata di x

`exp(x)` per l'esponenziale di $x e^x$

`log(x)` per il logaritmo naturale di x

`log10(x)` per il logaritmo decimale x

`ceil(x)` per arrotondare x all'intero maggiore

`floor(x)` per varrotondare x all'intero minore

`max_int` indica il valore massimo dell'intero elaborabile

`min_int` indica il valore minimo dell'intero elaborabile

`max_float` indica il valore massimo del numero a virgola mobile elaborabile

`min_float` indica il valore minimo del numero a virgola mobile elaborabile

Se l'argomento è un numero positivo possiamo omettere le parentesi.

6.4 Funzioni di valore assoluto:

`abs(x)` per il valore assoluto di x intero

`abs_float(x)` per il valore assoluto di x float

Se l'argomento è un numero positivo possiamo omettere le parentesi.

6.5 Funzioni per la conversione di tipo:

`int_of_char` converte da carattere a codice ASCII

`char_of_int` converte da codice ASCII a carattere

`float_of_int` converte da intero a float

`int_of_float` converte da float a intero

`int_of_string` converte da stringa a intero

`string_of_int` converte da intero a stringa

`float_of_string` converte da stringa a float

`string_of_float` converte da float a stringa

`bool_of_string` converte da stringa a valore booleano

`string_of_bool` converte da valore booleano a stringa

6.6 Funzioni di Input/Output

Per acquisire dati digitati sulla tastiera, cioè dallo standard input, abbiamo a disposizione queste funzioni:

`read_line()` legge il dato e lo acquisisce come stringa

`read_int()` legge il dato e lo acquisisce come intero

`read_float()` legge il dato e lo acquisisce come float

Per scrivere i risultati delle elaborazioni sullo schermo, cioè sullo standard output, abbiamo a disposizione queste funzioni:

`print_char()` scrive un carattere e non va a capo

`print_string()` scrive una stringa e non va a capo

`print_int()` scrive un numero intero e non va a capo

`print_float()` scrive un numero a virgola mobile e non va a capo

`print_newline()` per andare a capo

`print_endline()` scrive una stringa e va a capo

Se l'argomento è un carattere, una sola stringa o un solo numero positivo si possono evitare le parentesi.

Tra le parentesi l'argomento può essere un letterale o una espressione che fornisce un risultato da stampare.

Il linguaggio base contiene il modulo `Printf` per produrre output formattato. Questo modulo contiene la funzione `printf` richiamabile con

`Printf.printf()`

che consente di comporre l'output formattato utilizzando i seguenti segnaposti:

`%c` per un carattere

`%s` per una stringa

`%d` per un intero

`%f` per un numero a virgola mobile

`%.nf` per un numero a virgola mobile, limitando a `n` i decimali da esporre.

7 Creare funzioni

Possiamo noi stessi creare una funzione e, tra i tanti modi che il linguaggio ci offre per farlo, direi che il più sicuro è con la sintassi

```
let <nome_funzione>(<argomento: tipo>) (<argomento: tipo>) ... = <espressione>
```

Richiamando il nome della funzione e indicando gli argomenti tra le parentesi tonde otteniamo il risultato dell'espressione.

Possiamo creare una funzione per calcolare l'area del triangolo così

```
let area_triangolo(base:float) (altezza:float) = base *. altezza /. 2.
```

Notare come, avendo scelto il tipo `float` per gli argomenti della funzione, l'espressione debba contenere operatori adatti a quel tipo e ciò comporti che i dati inseriti nell'espressione debbano uniformarsi al tipo degli operatori (vedi il 2 che deve essere 2.).

Con

```
area_triangolo 12.5 10. otteniamo il risultato 62.5
```

Se la funzione è ricorsiva va creata con

```
let rec .....
```

8 Strutture di controllo

Per controllare l'esecuzione del programma abbiamo istruzioni per condizionare l'esecuzione di un blocco al verificarsi di certe condizioni oppure per la ripetizione di blocchi.

8.1 if-then-else

È l'istruzione condizionale che, verificata la condizione introdotta con `if`, se essa è verificata esegue l'istruzione (o il blocco di istruzioni) introdotta con `then`, altrimenti esegue l'istruzione (o il blocco di istruzioni) introdotta con `else`.

Un blocco di istruzioni si esprime racchiudendo le istruzioni, separate da un punto e virgola, tra parentesi tonde.

Contrariamente a ciò che avviene con altri linguaggi, in OCaml è sempre necessario utilizzare entrambe le vie `then` e `else`.

Esempi:

Date le variabili `x = 12` e `y = 22`,

```
if x<15 then print_string "OK" ritorna OK
```

```
if x>15 then print_string "NO" ritorna NO
```

```
if x<15 && y>20 then print_string "OK" else print_string "NO" ritorna OK
```

```
if x<15 && y<20 then
```

```
  print_string "OK"
```

```
else
```

```
  (
```

```
    print_string "NO";
```

```
    print_newline();
```

```
    print_string "non è vero che y è minore di 20"
```

```
  )
```

```
ritorna
```

```
  NO
```

```
  non è vero che y è minore di 20
```

8.2 for

Serve per ripetere l'esecuzione di una istruzione (o di un blocco di istruzioni) per un numero definito di volte.

La sintassi è

```
for <contatore> = <inizio> to <fine> do <istruzione/i> done
```

Con

```
for i = 1 to 5 do
```

```
  (
```

```
    print_string "Ciao, ";
```

```
    print_string "Pippo!";
```

```
    print_newline()
```

```
  )
```

```
done
```

si scrive per cinque volte, andando a capo, Ciao, Pippo!.

Con

```
for i = 1 to 3 do print_int i done
```

si scrive 123.

8.3 while

Altro modo di ripetere l'esecuzione di una istruzione (o di un blocco di istruzioni) per un numero definito di volte, fino a quando una condizione booleana è vera.

La sintassi di questa istruzione è

```
while <condizione_booleana> do <istruzione/i> done
```

Con questo costrutto, un altro modo di scrivere cinque volte Ciao, Pippo! è il seguente:

```

let i = ref 1;;
while !i<=5 do
(
print_endline "Ciao, Pippo!";
i := !i + 1
)
done

```

Qui la condizione booleana è verificata attraverso una variabile contatore che si incrementa di una unità ad ogni ciclo.

In quest'altro esempio si chiede all'utente di indicare numeri da sommare fino a quando viene indicato il numero 0 e, quando ciò avviene viene ritornata la somma dei numeri inseriti:

```

let somma = ref 0;;
let dato = ref "";
while !dato <> "0" do
(
print_string "Scrivi un intero da sommare (0 per terminare): ";
dato := read_line ();
somma := !somma + int_of_string !dato;
)
done;;
Printf.printf "La somma è: %d" !somma

```

Qui la condizione booleana è verificata attraverso l'introduzione di un carattere da tastiera.

9 Classi

Da quando Caml ha messo la O iniziale ed è diventato OCaml supporta la programmazione a oggetti.

Come avviene in tutti i linguaggi che hanno questa peculiarità abbiamo pertanto il costrutto Classe, che è lo stampo con cui possiamo creare un oggetto.

La sintassi per creare una classe è la seguente

```

class <nome> <variabile> <variabile> ... =
  object
    val <variabile> = <variabile>
    val <variabile> = <variabile>
    ...
    method <nome> = <espressione>
    method <nome> = <espressione>
    ...
  end

```

Con la parola chiave val si acquisiscono al costruttore dell'oggetto le variabili indicate nella definizione della classe e con la parola chiave method si definiscono i metodi che dovrà avere l'oggetto.

Questa è una classe per costruire un oggetto rettangolo dotato di metodi per determinarne area e perimetro:

```

class rettangolo base altezza =
  object
    val base = base
    val altezza = altezza
    method area = base * altezza
    method perimetro = 2*base+2*altezza
  end

```

Costruiamo un rettangolo di nome r con base 2 e altezza 3 con

```
let r = new rettangolo 2 3
  Con
r#area otteniamo la sua area,
r#perimetro otteniamo il suo perimetro.
```

In questo esempio possiamo operare con un rettangolo avente base e altezza identificati con numeri interi.

Per operare con un rettangolo avente base e altezza identificati con numeri decimali dobbiamo scrivere le espressioni dei metodi con operatori adatti a numeri float, così:

```
method area = base *. altezza
method perimetro = 2.*.base+.2.*.altezza
```

10 Struttura del programma

Come detto nel Capitolo 2 un programma OCaml è un file di testo che salviamo con estensione .ml.

Per la sua scrittura teniamo presente che OCaml è sensibile alle minuscole e alle maiuscole e che praticamente tutte le parole chiave del linguaggio sono in minuscolo. Iniziano con la lettera maiuscola solo i nomi dei moduli e dei packages.

Per introdurre commenti dobbiamo racchiuderli tra i simboli (* e *).

Nella prima riga del programma dobbiamo indicare gli eventuali moduli o packages che intendiamo usare con

```
open <nome_con_iniziale_maiuscola>
```

Se non facciamo questo, ogniqualvolta nel programma richiamiamo una funzione contenuta in un modulo o in un package dobbiamo farlo con la sintassi

```
<nome_con_iniziale_maiuscola>.<funzione>
```

come ho indicato parlando degli array nel Paragrafo 3.2 e dell'output formattato nel Paragrafo 6.6.

Una cosa molto problematica nella scrittura dei programmi OCaml è l'uso del carattere punto e virgola (;).

In linea di massima, mentre le istruzioni che inseriamo nella shell, come abbiamo visto nel Capitolo 2, devono necessariamente terminare con un doppio punto e virgola (;;) ad indicare all'interprete che l'istruzione è terminata, nei programmi l'uso del doppio punto e virgola non è necessario, anzi è sconsigliato.

Capita però che qualche punto e virgola serva anche nei programmi quando occorre delimitare nettamente istruzioni toplevel, cioè le definizioni e le espressioni scritte senza essere dentro un'altra costruzione.

In questi casi, se siamo all'interno di funzioni si usa il punto e virgola semplice.

Empiricamente ho osservato che ogniqualvolta l'interprete o il compilatore evidenziano con riferimento a una riga del programma errori inspiegabili e incomprensibili o la scritta «The extra argument is not expected», occorre terminare con il doppio punto e virgola la riga precedente.

11 Esempi di programmi

Cominciamo con un programmino che chiede all'utente il nome per salutarlo.

```
print_string "Come ti chiami?\n"
let nome = read_line();;
print_string ("Ciao, " ^ nome ^ "!\n")
```

Questo programma, se non si mette il doppio punto e virgola dopo la seconda istruzione, non funziona.

Funziona benissimo, alla faccia della raccomandazione di non usare i doppi punto e virgola nei programmi, se mettiamo il doppio punto e virgola alla fine di ogni istruzione.

Quest'altro programma, redatto secondo il paradigma imperativo/funzionale, mostra i valori della circonferenza e dell'area di un cerchio il cui raggio è indicato dall'utente

```
let pi = acos(-1.);;
print_string "Inserisci, in numero decimale, il raggio di un cerchio:\n"
let r = float_of_string(read_line())
let circonferenza r = 2. *. pi *. r
let area r = r *. r *. pi
let c = circonferenza r
let a = area r;;
print_string("Per un cerchio di raggio " ^ string_of_float(r) ^ ":");;
print_newline();;
Printf.printf("la circonferenza è: %.3f") c;;
print_newline();;
Printf.printf("l'area è: %.3f") a;;
print_newline();;
```

Notare la creazione della variabile per un valore approssimato di π attraverso l'utilizzo di una funzione trigonometrica e la formattazione della stampa dei risultati delle elaborazioni con limitazione a 3 cifre decimali.

I doppi punto e virgola sono stati inseriti ove necessario per poter compilare il programma.

Questo è lo stesso programma. redatto secondo il paradigma della programmazione per oggetti

```
class cerchio raggio =
  object
    val r = raggio
    val pi = acos(-1.)
    method circonferenza = 2. *. pi *. r
    method area = r *. r *. pi
  end;;
print_string "Inserisci, in numero decimale, il raggio di un cerchio:\n"
let r = read_float()
let c = new cerchio r;;
print_string("Per un cerchio di raggio " ^ string_of_float(r) ^ ":");;
print_newline();;
Printf.printf("la circonferenza è: %.3f") c#circonferenza;;
print_newline();;
Printf.printf("l'area è: %.3f") c#area;;
print_newline();;
```

Ora un programma per calcolare il fattoriale di un numero indicato dall'utente

```
let rec fattoriale n =
  if n = 0 then 1
  else n * fattoriale(n-1);;
print_string "Immetti il numero di cui vuoi calcolare il fattoriale\n"
let n = int_of_string(read_line());;
print_int(fattoriale(n));;
```

Come abbiamo visto nel Paragrafo 3.1 con questo programma che tratta interi a 64 bit possiamo arrivare a calcolare il fattoriale di 20.

12 Estensioni del linguaggio

Fin qui abbiamo visto il linguaggio base OCaml (la libreria standard, chiamata Stdlib) con qualche piccola incursione in due moduli che ne estendono la potenzialità e che sono inclusi

nella libreria standard: il modulo `Printf`, che contiene funzioni per formattare l'output, e il modulo `Array`, che contiene funzioni per creare array e matrici.

Complessivamente i moduli compresi nella libreria standard sono 45 e sono i seguenti:

- Array – operazioni su array mutabili.
- ArrayLabels – variante con etichette sui parametri.
- Buffer – buffer estendibili di stringhe.
- Bytes – stringhe mutabili.
- BytesLabels – versione etichettata.
- Callback – registrazione di funzioni richiamabili da C.
- Char – funzioni sui caratteri.
- Complex – numeri complessi.
- Digest – funzioni hash MD5.
- Either – tipo somma con due varianti (Left, Right).
- Ephemeron – riferimenti deboli con chiave e valore.
- Filename – gestione di nomi di file.
- Float – operazioni sui float.
- Format – formattazione avanzata.
- Fun – combinatori funzionali semplici.
- Gc – controllo del garbage collector.
- Hashtbl – tabelle hash mutabili.
- Int – operazioni sugli interi.
- Int32, Int64, Nativeint – interi a 32, 64 e dimensione macchina.
- Lazy – valori calcolati su richiesta.
- Lexing – supporto ai lexer generati da `ocamllex`.
- List – funzioni su liste.
- ListLabels – variante con etichette.
- Map – dizionari immutabili.
- Marshal – serializzazione binaria.
- MoreLabels – estensioni etichettate di `Hashtbl`, `Map`, `Set`.
- Obj – accesso basso livello alla rappresentazione interna.
- Option – operazioni sul tipo `option`.
- Parsing – supporto ai parser generati da `ocamlyacc`.
- Printexc – gestione delle eccezioni.
- Printf – formattazione stile C.
- Queue – code mutabili.
- Random – numeri pseudocasuali.
- Result – tipo ('a, 'b) result e funzioni associate.
- Scanf – input formattato.
- Seq – sequenze lazy.
- Set – insiemi immutabili.
- Stack – pile mutabili.
- Stdlib – il modulo stesso, che raccoglie tutti i valori e moduli di base.
- Stream – flussi di token (deprecated).
- String – stringhe immutabili.
- StringLabels – versione etichettata.
- Sys – interfacce al sistema operativo.
- Uchar – caratteri Unicode.
- Weak – riferimenti deboli.

All'indirizzo https://ocaml.org/manual/5.1/api/index_modules.html troviamo accesso alla descrizione dei contenuti di questi moduli.

Ulteriori estensioni del linguaggio si trovano in packages aggiuntivi, non compresi nella libreria standard e che occorre installare a parte ricorrendo al tool `opam` di cui ho parlato nel Capitolo 1.

Altro package di grande utilità può essere quello per la grafica, che possiamo installare con il comando a terminale

```
opam install graphics
```

Questo è un esempio di programma che utilizza questo package per creare il grafico di una curva cardioide.

```
Graphics.open_graph " 300x200";;  
Graphics.moveto 200 150;;  
for i = 0 to 200 do  
  let th = atan 1. *. float i /. 25. in  
  let r = 50. *. (1. -. sin th) in  
  Graphics.lineto (150 + truncate (r *. cos th)) (150 + truncate (r *. sin th))  
done;  
ignore (Graphics.read_key ())
```

Invece di iniziare con

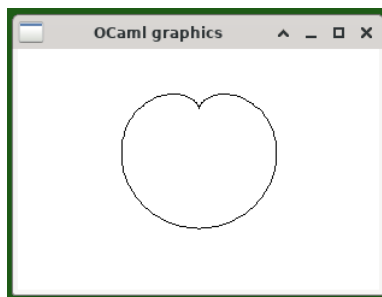
```
open Graphics
```

ho richiamato il package con `Graphics.` ogniqualvolta ne ho utilizzato una funzione.

Salvato in un file `cardioide.ml` e compilato con il comando

```
ocamlfind ocamlc -package graphics -linkpkg cardioide.ml -o cardioide
```

una volta lanciato produce questo



Altro divertente programmino grafico

```
open Graphics;;  
open_graph " 640x480";;  
for i = 10 downto 1 do  
  let radius = i * 20 in  
  set_color (if (i mod 2) = 0 then red else yellow);  
  fill_circle 320 240 radius  
done;  
ignore (read_key())
```

che produce questo

