Linguaggi di Programmazione, laboratorio

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a ldurante le esercitazioni in laboratorio. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Grazie mille e buono studio!

Capitolo 2

Laboratorio

2.1 Laboratorio 1

Esercizio 1. *Esempio base:*

```
lavora_per(bill, google). %questo è un fatto
lavora_per(mario, oracle).
lavora_per(steve, apple).  % con ; scorro le opzioni
lavora_per(jane, google).

collega(X, Y) :- lavora_per(X, Z),
                 lavora_per(Y, Z),
                 X \= Y. /* questa è una regola e con \=
                            evita che X = X sia un risultato
```

si possono avere le seguenti richieste con i seguenti output:

```
?- lavora_per(X, google).
X = bill ;
X = jane.

?- lavora_per(bill, X).
X = google.

?- collega(bill, jane).
true.
```

Esercizio 2. *esercizio 2, logica dei Naturali:*

```
/* definisco i naturali: */
nat(0). % 0 è naturale e lo sono anche i successori di un naturale
nat(s(N)) :- nat(N). /* s(N) indica il successore,
                        in compilatore sarà s(s(0)) etc...*/

/* definisco la somma come n + m = (n+1) + (m-1) se m>0 n + 0 = n */

succ(N, s(N)). % defisco il successore
sum(N, 0, N). % il terzo è il risultato
sum(N, s(M), s) :- sum(N, M, T),
                   succ(T, s).

/* si può anche fare così; */
summ(N, 0, N).
summ(N, s(M), s(T)) :- summ(N, M, T).
summ(N, M, S) :-  summ(s(N), P, S),
                  succ(P, M).
```

si hanno i seguenti output:

```
?- nat(0).
true.

?- nat(s(0)).
true.

?- sum(s(s(0)), 0, X).
X = s(s(0)) ;
false.

?- summ(s(0), 0, X).
X = s(0).
```

Esercizio 3. *Calcolo il fattoriale:*

```
fact(0, 1). % fattoriale di 0 è 1, caso base
fact(N, M) :- N>0,
              N1 is N-1,
              fact(N1, M1),
              M is N * M1. /* ogni volta salva in M1 il
                              risultato parziale, N>0
                               porterà a non fare il caso
                                base*/
```

```
?- fact(3, X).
X = 6 ;
false.
```

Esercizio 4. *Primi esercizi sulle liste:*

```
/* trovo primo elemento della lista */
intesta(X, [X|_]). /* | separa testa e coda (che non
                      interessa e indico con _,
                       senza non si sa che la lista può
                        continuare. Interrogo con, per
                         esempio  intesta(2, [2,3]). */

/* trovo ennesimo elemento*/
nth(0, [X|_], X). % se cerco lo 0, caso base
nth(N, [_|T], X) :- N1 is N-1,
                    nth(N1, T, X). /* se non è all'inizio
                                      cerco nella coda ad
                                       ogni passo la head
                                        aumenta di uno
                                        tengo solo T che è
                                         la coda, ovvero
                                         tutto tranne il
                                          primo*/

/*lista contiene l'elemento X?*/
contains(X, [X|_]). % controllo testa
contains(X,[_|T]) :- contains(X, T). % controllo ricorsivcamente la coda
```

Si hanno i seguenti output:

```
?- intesta(2, [2, 3]).
true.

?- intesta(2, [1, 3]).
false.

?- nth(0, [1,2,3], X).
X = 1 ;
false.

?- nth(2, [1,2,3], X).
X = 3 ;
false.

?- nth(18, X, 1).
X = [_594, _600, _606, _612, _618, _624|...] ;

?- contains(5, [5, 6, 7]).
true ;
false.

?- contains(4, [5, 6, 7]).
false.
```

dove l'ultimo risultato di nth ci dice che non può fare nulla con l'istruzione data, cerca all'infinito una risposta senza trovarla. Il primo di contains da false ad una seconda richiesta di risultato in quanto prova ad usare un'altra regola il compilatore di prolog

Esercizio 5. *Ancora sulle liste:*

```
/* funzione append per concatenare */
append([], L, L). % vuota più L = L
append([H|T], L, [H|X]) :- append(T, L, X).
/* l'inizio della lista finale è H e la fine è la fine
    delle liste concatenate */

/* chiedo se una lista  è ordinata */

sorted([]). % lista vuota ordinata
sorted([_]). % lista di un elemento è ordinata
sorted([X, Y| T]) :- X =< Y,
                     sorted([Y| T]). /* confronto sempre l'elemento col
                     resto della tail */

/* chiedo ultimo elemento */

last([X], X). %lista di un elemento ha come ultimo quell'elemento
last([_|T], X) :- last(T, X). /*controllo ricorsivamente
                                 la coda della lista finché
                                  non ho solo T e posso usare
                                   il caso base */
/* posso anche chieder e una lista che finisca con un certo N
                                 per esempio 4  last(X, 4). */

/* tolgo tutte le occorrenze */

remove_all([], _, []). % la lista vuota non ha nulla da rimuovere */
remove_all([X|T], X, L) :- remove_all(T, X, L).
/* se la testa è quel numero rimuovo tutte
    le occorrenze... ma se la testa \= X non va*/
remove_all([H|T], X, [H|L]) :- H \= X,
                               remove_all(T, X, L).
                               /* se H\=X  faccio il
                                   controllo senza H */

/* somma elementi lista */

somma_lista([], 0).
somma_lista([H|T], X) :-  somma_lista(T, N),
```

```
                                X is H + N.
/* sommo tutte le tail ricorsivamente e poi ci sommo la H */

/* ricordiamo le basi delle code */
coda([_|T], T).

/* duplico lista [1,2]->[1,1,2,2] */

duplico([], []).
duplico([H|T], [H, H|X]) :- duplico(T, X).
/* a priori duplico H riscivendo nel risultato
    poi duplico il tail, dove di volta in volta
     ogni head verrà duplicato. Se inverto e metto
      ( x, [lista]) mi toglie i duplicati*/
```

si hanno i seguenti output:

```
?- contains(5, [5, 6, 7]).
true ;
false.

?- contains(4, [5, 6, 7]).
false.

?- append([4], [5, 6, 7], X).
X = [4, 5, 6, 7].

?- append([1, 2, 3], X, [1, 2, 3]).
X = [].

?- append([1, 2, 3], X, [1, 2, 3, 6]).
X = [6].

?- append(X, Y, [1, 2, 3, 6]).
X = [],
Y = [1, 2, 3, 6] ;
X = [1],
Y = [2, 3, 6] ;
X = [1, 2],
```

```
Y = [3, 6] ;
X = [1, 2, 3],
Y = [6] ;
X = [1, 2, 3, 6],
Y = [] ;
false.

- sorted([1,2,1]).
false.

?- sorted([1, 2, 1]).
false.

?- sorted([1, 2, 1]).
false.

?- last([1, 2, 3], 3).
true.

?- last([1, 2, 3], X).
X = 3.

?- remove_all([1, 2, 1], X, L).
X = 1,
L = [2] ;
false.

?- somma_lista([1, 2, 3, 4], X).
X = 10.

coda([1,2,3], X).
X = [2,3]

?- duplico([1, 2, 3], X).
X = [1, 1, 2, 2, 3, 3].
```

2.2 Laboratorio 2

Esercizio 6. *ancora sulle liste*

```
/* trovo min lista*/

% la lista da in automatico false
min([H], H).
min([H | T], H) :- min(T, X),
                   H =< X. % fa solo H
min([H | T], X) :- min(T, X),
                   H > X. % fa tutto il resto
```

```
?- min([3, 2, 2], X).
X = 2 .

?- min([2, 3, 4], X).
X = 2.
```

Esercizio 7. *ancora sulle liste*

```
/* rimuovo valore (una sola copia) da lista salvando in lista */
remove_one([], _, []).
remove_one([H | T], H, T). % se è la testa la rimuovo
remove_one([H | T], X, [H | S]):- remove_one(T, X, S),
                                  X \= H.
 /* la H la tengo e riattacco la nuova S
       e uso il primo casoi base e non il secondo
         usando il \=*/
```

```
?- remove_one([1, 2, 3, 4], 2, L).
L = [1, 3, 4] ;
false.

?- remove_one([1, 1, 2, 3, 4], 1, L).
L = [1, 2, 3, 4] ;
false.
```

Esercizio 8. *ancora sulle liste*

```
/* selection sort */
selection_sort([], []).
selection_sort(X, [H| T]) :- min(X, H), % minimo in testa
                             remove_one(X, H, X1), %toglie il minimo
                             selection_sort(X1, T). % rifa conl T

/* unisco liste in ordine */
merge([], X, X).
merge(X, [], X).
merge([H1 | T1], [H2 | T2], [H1 | T]) :- H1 =< H2,
                             merge(T1, [H2 | T2], T).
merge([H1 | T1], [H2 | T2], [H1 | T]) :- H2 =< H1,
                             merge([H1 | T1], T2, T).

/* spezzo in due lista */
split_in_two([],[],[]).
split_in_two([X],[X],[]).
split_in_two([H1, H2 | T],[H1 | T1],[H2 | T2]) :- split_in_two(T, T1, T2).

/* mergesort */
/* divide in 2 la lista, fa il mergesort delle due e merge dei risultati*/
mergesort([], []).
mergesort([X], [X]).
mergesort(L1, L2) :-  split_in_two(L1, X1, Y1),
                      mergesort(X1, X2),
                      mergesort(Y1, Y2),
                      merge(X2, Y2, L2).
```

```
?- selection_sort([2, 1, 4], L).
L = [1, 2, 4] ;
false.

?- merge([2,4], [1,3,5], X).
X = [1, 2, 3, 4, 5].

?- split_in_two([1,2,3,4],X, Y).
X = [1, 3],
```

```
Y = [2, 4].

?- split_in_two([1,2,4],X, Y).
X = [1, 4],
Y = [2] ;

?- mergesort([2,4,1,7,6,2,12,3], X).
X = [1, 2, 2, 3, 4, 6, 7, 12] .
```

Esercizio 9. *ancora sulle liste*

```
/* da più listre annidate a una */
/* da [[1, 3],[[4]]] */
/* uso definizione di lista e append */
listp([]).
listp([_ | _]). % vero solo se ho lista

append([], X, X).
append([H|T], L, [H|X]) :- append(T, L, X).

flatten([], []).
flatten([H | T], L) :- listp(H),
                       flatten(H, X),
                       flatten(T, Y),
                       append(X, Y, L). /* se testa è lista la appiattisco
                                           e la aggiungo al risultato */
flatten([H | T], [H | X]) :- flatten(T, X). /* altrimenti appiattisco la tail
                                                ricorsivamente tenendo la head*/
```

```
?- flatten([[1,2,3],[[4]]], X).
X = [1, 2, 3, 4]
```

2.3 Laboratorio 3

Esercizio 10

```
/* Esercizio del tris
  uso una lista a 9 entrate per le caselle
  e una con le x e gli o */

/* vedo se c'è posto in una lista, prendendo il primo libero,
    dove c'è ancora un numero,
    [1,2,x,o,o,x,7,8,o] -> X=[1, 2, 7, 8] */

find_free_position([], []).
find_free_position([x|T], X) :- find_free_position(T, X). % no x in H
find_free_position([o|T], X) :- find_free_position(T, X). % no o in H
find_free_position([H|T], [H|T2]) :- find_free_position(T, T2).

/* stampa della tabella */

print_board([]).
print_board([X|T]) :-  write(X),
                       print_board2(T). % prima colonna
print_board2([X|T]) :-  write(X),
                        print_board3(T). % seconda colonna
print_board3([X|T]) :-  write(X),
                        nl,
                        print_board(T). % terza colonna, nl=newline


/* giocatore vince se tre X di fila, orizzontale, verticale e diagonale */

/* righe */
win(X, [X, X, X, _, _, _, _, _, _]).
win(X, [_, _, _, _X, X, X, _, _, _]).
win(X, [_, _, _, _, _, _, X, X, X]).

/* verticali */
win(X, [X, _, _, X, _, _, X, _, _]).
win(X, [_, X, _, _, X, _, _, X, _]).
win(X, [_, _, X, _, _, X, _, _, X]).

/*diagonali*/
win(X, [X, _, _, _, X, _, _, _, X]).
```

```
win(X, [_, _, X, _, X, _, X, _, _]).

/* nth0(Pos, Lista, Elem, Rest) è gia in prolog */

/* replace (Pos, Lista, Elem, Lista2) per le giocate
   replace(2, [1,3,8], 4, X) -> X=[1, 3, 4]*/

/* rimuovo qualsiasi cosa in pos e salvo la lista rimanente
   a quel punto uso nth0 per chiedere la lista L2 tale che
   ci sia I in pos a partire da L3 che non ha più l'elemento
   che aveva prima */

replace(X, L, I, L2) :- nth0(X, L, _, L3), % rimuove qualsiasi cosa in pos=X
                        nth0(X, L2, I, L3). % aggiunge in pos=X  elem=I

/* richiedo mossa e salvo la mossa */
 /* member mi dice se elemento è in lista */

player_move(Board, Player, NewBoard) :- print_board(Board),
                                        write("Dove vuoi fare la mossa?"),
                                        nl,
                                        read(X),
                                        find_free_position(Board, FP),
                                        X<10,
                                        member(X, FP),  % libero?
                                        Pos is X-1, % si parte da 0
                                        replace(Pos, Board, Player, NewBoard).

player_move(Board, Player, NewBoard) :- write("Posizione non valida"),
                                        nl,
                                        player_move(Board, Player, NewBoard).

/* passiamo al gioco, scegliendo x oppure o */

game(Board, _) :- win(x, Board),
                  writeln("Ha vinto il giocatore x").
game(Board, _) :- win(o, Board),
                  writeln("Ha vinto il giocatore o").
/* se non ho più mosse possibili è pareggio */
game(Board, _) :- find_free_position(Board, []),
                  writeln("Pareggio").
```

```
game(Board, x) :- player_move(Board, x, NewBoard),
                  game(NewBoard, o).
game(Board, o) :- player_move(Board, o, NewBoard),
                  game(NewBoard, x).

/* faccio iniziare il gioco */

one_game :- write("Inizia x oppure o?"),
            nl,
            read(X),
            member(X, [x,o]),
            game([1, 2, 3, 4, 5, 6, 7, 8, 9], X).
one_game :- write("errore"), % se non si isnerisce x o
            nl,
            one_game.
```

```
?- find_free_position([x,1,3,o,4,x,o,5], X).
X = [1, 3, 4, 5]

?- print_board([1,2,3,4,5,6,7,8,9]).
123
456
789
true.

?- win(x, [x,x,x,o,x,o,x,x,o]).
true

?- win(x,X).
X = [x, x, x, _3878, _3884, _3890, _3896, _3902,_3908];
X = [_3860, _3866, _3872, x, x, x, _3896, _3902,_3908];
X = [_3860, _3866, _3872, _3878, _3884 _3890, x, x, x];
X = [x, _3866, _3872, x, _3884, _3890, x, _3902,_3908];
X = [_3860, x, _3872, _3878, x, _3890, _3896, x,_3908];
X = [_3860, _3866, x, _3878, _3884, x, _3896,_3902, x];
X = [x, _3866, _3872, _3878, x, _3890, _3896,_3902, x];
X = [_3860, _3866, x, _3878, x, _3890, x, _3902,_3908].
```

```
?- replace(2, [1,3,8], 4, X).
X = [1, 3, 4].

?- player_move([1,2,3,x,o,x,7,8,9],o,X).
123
xox
789
Dove vuoi fare la mossa?
|     5.
Posizione non valida
123
xox
789
Dove vuoi fare la mossa?
|     3.
12o
xox
789
X = [1, 2, o, x, o, x, 7, 8, 9] .

?- one_game.
Inizia x oppure o?
|     o.
1 2 3
4 5 6
7 8 9
Dove vuoi fare la mossa?
|     1.
o 2 3
4 5 6
7 8 9
Dove vuoi fare la mossa?
|     ^C
^C

Action (h for help) ? abort
Action (h for help) ? Unknown option (h for help)
Action (h for help) ? abort
?- one_game.
```

```
Inizia x oppure o?
|     x.
1 2 3
4 5 6
7 8 9
Dove vuoi fare la mossa?
|     1.
x 2 3
4 5 6
7 8 9
Dove vuoi fare la mossa?
...

Dove vuoi fare la mossa?
|     1
|     .
Posizione non valida
x23
456
789
Dove vuoi fare la mossa?
|     3.
x2o
456
789
Dove vuoi fare la mossa?
|     4.
x2o
x56
789
Dove vuoi fare la mossa?
|     6.
x2o
x5o
789
Dove vuoi fare la mossa?
|     7.
Ha vinto il giocatore x
true .
```

2.4 Laboratorio 4

atom_string fa da variabile a stringa, number_string da numero a stringa e string_codes da stringa a ascii. I caratteri si indicano con 0'char.

```
?- atom_string(test, X).
X = "test".

?- atom_string(test, X), X=test.
false.

?- number_string(QD, ""42.0).
QD = 42.0

?- number_string(X, "10").
X = 10.

?- number_string(X, "a").
false.

?- string_codes""(42, Cs).
Cs = [52, 50]
```

Esercizio 11

```
/* data stringa e carattere rimuova tutte le copie di quel carattere */

remove([], _,[]).
remove([H|T], H, L) :- remove(T, H, L).
remove([H|T], X, [H|S]) :- H\= X,
                           remove(T, X, S).

remove_char([], _, []).
remove_char([X], X, []).
remove_char(String, Char, NewString) :- atom_string(Char, C),
                                        string_codes(C, [Charc]),
                                        string_codes(String, S),
                                        remove(S, Charc, NewString2),
                                        string_codes(NewString, NewString2).
```

```
remove_char([a,b,c,c],c,X).
X = "ab"

?- remove_char([a,b,c,c],"c",X).
X = "ab"
```

così uso una libreria

```
library(readutil).
```

Capitolo 3

Lisp

3.1 Laboratorio 1

```
;; Definiamo una funzione per moltiplicare
;; ogni valore di una lista per due
(defun double-list (lst)
  (if (null lst)
      nil
      (cons (* 2 (car lst)) (double-list (cdr lst)))))

;; Ora ne definiamo una versione iterativa
(defun double-list-iter (lst acc)
  (if (null lst)
      (reverse acc)
      (double-list-iter (cdr lst) (cons (* 2 (car lst)) acc))))

;; Possiamo generalizzare la visita della lista
;; applicando una funzione
;; ad ogni elemento della lista
(defun map-list (lst func)
  (if (null lst)
      nil
      (cons (funcall func (car lst)) (map-list (cdr lst) func))))
;; Recuperiamo la vecchia funzione di raddoppiare liste utilizzando
;; (map-list lst (lambda (x) (* 2 x)))
;; Common Lisp fornisce già una funzione che fa quello che fa map-list:
;; (mapcar funzione list)
```

```
;; Dato che possiamo passare e ritornare
;; funzioni come argomento possiamo
;; avere applicazioni parziali. In questo caso di una funzione di due
;; elementi
(defun partial-apply (func x)
  (lambda (y) (funcall func x y)))
;; In questo modo possiamo riottenere
;; la funzione per raddoppiare i valori
;; in una lista come una applicazione parziale
;; di mapcar in cui il primo
;; argomento è fissato a (lambda (x) (* 2 x)).
;; Ad esempio:
(let ((list-doubler (partial-apply #'mapcar (lambda (x) (* 2 x)))))
  (format t "~a~%" (funcall list-doubler '(1 2 3)))
  (format t "~a~%" (funcall list-doubler '(0 -3 8 9 -6)))
  (format t "~a~%" (funcall list-doubler '(8 7 6))))
;; Nel precedente codice "list-doubler" è il
;; risultato dell'applicazione di
;; mapcar in cui abbiamo fissato il primo argomento
;; ad una lista, ottenenedo
;; quindi un "raddoppiatore" del valore
;; contenuto in una lista

;; Definizione della struttura di nodo con tre campi:
;; value (valore contenuto)
;; left e right, i due sottoalberi.
;; Di default sono tutte a nil
(defstruct node
  value
  left
  right)
;; make-node (per creare un nodo), node-value, node-left e node-right
;; sono creati in automatico quando la stuttura node viene definita.

;; RItorna un albero in cui è stato inserito il nuovo valore "value"
(defun tree-insert (root value)
  (let ((v (unless (null root) (node-value root))))
    (cond ((null root) (make-node :value value))
     ((= v value) root)
     ((> v value)
      (make-node :value v
```

```
            :left (tree-insert (node-left root) value)
            :right (node-right root)))
      ((< v value)
       (make-node :value v
            :left (node-left root)
            :right (tree-insert (node-right root) value))))))

;; Funzione che effettua la ricerca
;; all'interno di un albero binario di ricerca
(defun tree-search (root value)
  (let ((v (unless (null root) (node-value root))))
    (cond ((null root) nil)
      ((= v value) t)
      ((> v value) (tree-search (node-left root) value))
      ((< v value) (tree-search (node-right root) value)))))
```

3.2 Laboratorio 2

```
;; Vediamo come leggere un file in Common Lisp

(defun read-all-lines ()
  (with-open-file (f "example.txt" :direction :input)
    (read-all-lines-helper f)))

(defun read-all-lines-helper (stream)
  (let ((line (read-line stream nil nil))) ;; leggi la linea ritornando nil in
    (when line (cons line (read-all-lines-helper stream)))))

;; Ridefiniamo usando una funzione definita localmente per la lettura delle li

(defun read-all-lines2 ()
  (with-open-file (f "example.txt" :direction :input)
    (labels ((read-helper ()
         (let ((line (read-line f nil nil)))
     (when line (cons line (read-helper))))))
      (read-helper))))

;; Ricerca di un valore in una stringa (funziona anche per sottosequenze)
;; (search "test" "this is a test") => 10 (posizione del match)
```

```
;; Esercizio

(defun read-matching (match)
  (with-open-file (f "example.txt" :direction :input)
    (labels ((read-helper ()
         (let ((line (read-line f nil nil)))
     (when line
       (if (search match line)
           (cons line (read-helper))
           (read-helper))))))
      (read-helper))))

;; Parsing di un valore salvato in una stringa

(defun convert-string (str)
  (when (string/=  str "")
    (multiple-value-bind (value num-chars) (read-from-string str nil)
      (when value
  (cons value (convert-string (subseq str num-chars)))))))

;; Format to write to files
;; ~A (auto)
;; ~D (decimal number)
;; ~F (floating point)
;; ~R (number as words)
;; ~% (newline)

(defun write-one-per-line (lst)
  (when lst
    (format t "~R~%" (car lst))
    (write-one-per-line (cdr lst))))

;; vediamo ora come scrivere su file

(defun write-one-per-line-on-file (lst filename)
  (with-open-file (f filename :direction :output)
    (labels ((helper (lst)
         (when lst
     (format f "~R~%" (car lst))
     (helper (cdr lst)))))
      (helper lst))))
```

```
;; Vediamo ora come implementare una calcolatrice RPN

(defun rpn-calc (stack)
  (format t "stack:~A~%" stack)
  (format t "> ")
  (let ((op (read)))
    (cond ((numberp op) (rpn-calc (cons op stack)))
    ((eq op '+) (rpn-calc (cons
         (+ (car stack) (cadr stack))
         (cddr stack))))
    ((eq op '-) (rpn-calc (cons
         (- (car stack) (cadr stack))
         (cddr stack))))
    ((eq op '*) (rpn-calc (cons
         (* (car stack) (cadr stack))
         (cddr stack))))
    ((eq op '/) (rpn-calc (cons
         (/ (car stack) (cadr stack))
         (cddr stack))))
    ((eq op 'quit) (format t "Quit~%"))
    (t nil))))

;; Proviamo ad aggiungere la stampa degli argomenti ad una funzione

(defun add-logging (func)
  (lambda (&rest args)
    (print args)
    (apply func args)))

;; Esempio di applicazione

(defun double-values (list)
  (mapcar (add-logging (lambda (x) (* x 2))) list))

;; Rendiamo più leggibile con gli argomenti opzionali

(defun add-logging2 (func &optional text)
  (lambda (&rest args)
    (if (null text)
  (format t "~A~%" args)
```

```
  (format t "~A: ~A~%" text args))
    (apply func args)))

(defun double-values2 (list)
  (mapcar (add-logging2 (lambda (x) (* x 2)) "Chiamo (lambda (x) (* x 2)) con ar

;; Esercizio: definire "add-logging3" in cui è possibile vedere anche
;; il valore di ritorno della funzione chiamata e che possa stampare anche un
;; a scelta prima di stampare l'output

(defun add-logging3 (func &optional text-before text-after)
  (lambda (&rest args)
    (if (null text-before)
  (format t "~A~%" args)
  (format t "~A: ~A~%" text-before args))
    (let ((result (apply func args)))
      (if (null text-after)
    (format t "~A~%" result)
    (format t "~A: ~A~%" text-after result))
      result)))

(defun double-values3 (list)
  (mapcar (add-logging3 (lambda (x) (* x 2))
      "Chiamo (lambda (x) (* x 2)) con argomenti"
      "Ottenendo come risultato")
    list))
```

Capitolo 4

Progetto

4.1 Prolog

```
%%%% Little Man Computer
%%%% Progetto di Linguaggi di Programmazione
%%%% Anno accademico 2018-2019
%%%% Appello di Gennaio 2019
%%%% Davide Cozzi 829827
%%%% Gabriele De Rosa 829835

%%% Addizione
%%% Instruction: 1xx
addizione(Acc, Pointer, Mem, X, flag) :-
    nth0(Pointer, Mem, Value, _),
    Y is Acc + Value,
    X is ((Acc + Value) mod 1000),
    Y > 999,
    !.

addizione(Acc, Pointer, Mem, X, noflag) :-
    nth0(Pointer, Mem, Value, _),
    Y is Acc+Value,
    X is ((Acc+Value) mod 1000),
    Y < 1000.

%%% Sottrazione
%%% Instruction: 2xx
sottrazione(Acc, Pointer, Mem, X, flag) :-
```

```
    nth0(Pointer, Mem, Value, _),
    Y is Acc-Value,
    X is ((Acc-Value)mod 1000),
    Y < 0,
    !.

sottrazione(Acc, Pointer, Mem, X, noflag) :-
    nth0(Pointer, Mem, Value, _),
    Y is Acc-Value,
    X is ((Acc-Value)mod 1000),
    Y >= 0.

%%% Store
%%% Instruction: 3xx
store(Acc, Pointer, MemIn, MemOut) :-
    nth0(Pointer, MemIn, _, Varmem),
    nth0(Pointer, MemOut, Acc, Varmem).

%%% Load
%%% Instruction: 5xx
load(Acc, Pointer, MemIn) :-
    nth0(Pointer, MemIn, Acc, _).

%%% Branch
%%% Instruction: 6xx
branch(Pc, Pc).

%%% Branch if zero
%%% Instruction: 7xx
branchifzero(_, 0, Pointer, noflag, Pointer) :-
    !.

branchifzero(Pc, _, _, noflag, NewPc) :-
    NewPc is ((Pc + 1) mod 100).

branchifzero(Pc, _, _, flag, NewPc) :-
    NewPc is ((Pc + 1) mod 100).

%%% Branch if positive
%%% Instruction: 8xx
branchifpositive(_, Pointer, noflag, Pointer) :-
```

```
    !.

branchifpositive(Pc, _, flag, NewPc) :-
    NewPc is ((Pc + 1) mod 100).

%%% Input
%%% Instruction: 901
input(Acc, [Acc|NewQueueIn], NewQueueIn).

%%% Output
%%% Instruction: 902
output(Acc, QueueOut, NewQueueOut) :-
    append(QueueOut, [Acc], NewQueueOut).

%%% Little Man Computer

%%% Istruzione in memoria
instr_in_mem(Pc, Mem, Instruction) :-
    nth0(Pc, Mem, Instruction, _).

%%% Puntatore nell'istruzione
extract_pointer(I, X) :-
    X is (I mod 100).

%%% One Instruction
%%% Passaggio da uno stato iniziale ad uno stato finale
%%% a seconda del valore dell'opcode (Istr)
one_instruction(state(Acc, Pc, Mem, In, Out, Flag),
                halted_state(Acc, Pc, Mem, In, Out, Flag)) :-
    instr_in_mem(Pc, Mem, Istr),
    Istr >= 0,
    Istr < 100,
    !.

one_instruction(state(Acc, Pc, Mem, In, Out, _),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 100,
    Istr < 200,
    addizione(Acc, Pointer, Mem, Acc2, Flag2),
```

```
    Pc2 is ((Pc + 1) mod 100),
    append([], Mem, Mem2),
    append([], In, In2),
    append([], Out, Out2).

one_instruction(state(Acc, Pc, Mem, In, Out, _),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 200,
    Istr < 300,
    sottrazione(Acc, Pointer, Mem, Acc2, Flag2),
    Pc2 is ((Pc + 1) mod 100),
    append([], Mem, Mem2),
    append([], In, In2),
    append([], Out, Out2).

one_instruction(state(Acc, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 300,
    Istr < 400,
    store(Acc, Pointer, Mem, Mem2),
    Acc2 is Acc,
    Pc2 is ((Pc + 1) mod 100),
    append([], In, In2),
    append([], Out, Out2),
    copy_term(Flag, Flag2).

one_instruction(state(_, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 500,
    Istr < 600,
    load(Acc2, Pointer, Mem),
    Pc2 is ((Pc + 1) mod 100),
    append([], Mem, Mem2),
    append([], In, In2),
    append([], Out, Out2),
```

```
    copy_term(Flag, Flag2).

one_instruction(state(Acc, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 600,
    Istr < 700,
    branch(Pc2, Pointer),
    Acc2 is Acc,
    append([], Mem, Mem2),
    append([], In, In2),
    append([], Out, Out2),
    copy_term(Flag, Flag2).

one_instruction(state(Acc, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 700,
    Istr < 800,
    branchifzero(Pc, Acc, Pointer, Flag, Pc2),
    Acc2 is Acc,
    append([], Mem, Mem2),
    append([], In, In2),
    append([], Out, Out2),
    copy_term(Flag, Flag2).

one_instruction(state(Acc, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    extract_pointer(Istr, Pointer),
    Istr >= 800,
    Istr < 900,
    branchifpositive(Pc, Pointer, Flag, Pc2),
    Acc2 is Acc,
    append([], Mem, Mem2),
    append([], In, In2),
    append([], Out, Out2),
    copy_term(Flag, Flag2).
```

```
one_instruction(state(_, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    Istr = 901,
    proper_length(In, InEmpty),
    InEmpty \= 0,
    input(Acc2, In, In2),
    Pc2 is ((Pc + 1) mod 100),
    append([], Mem, Mem2),
    append([], Out, Out2),
    copy_term(Flag, Flag2).

one_instruction(state(Acc, Pc, Mem, In, Out, Flag),
                state(Acc2, Pc2, Mem2, In2, Out2, Flag2)) :-
    instr_in_mem(Pc, Mem, Istr),
    Istr = 902,
    output(Acc, Out, Out2),
    Acc2 is Acc,
    Pc2 is ((Pc + 1) mod 100),
    append([], Mem, Mem2),
    append([], In, In2),
    copy_term(Flag, Flag2).

%%% Check List
%%% Controlla che una lista non abbia valori superiori a 999
%%% E che abbia solo valori positivi
checklist([]).

checklist([H|T]) :-
    H =< 999,
    H >= 0,
    !,
    checklist(T).

%%% Execution Loop
%%% Cicla dallo stato iniziale allo stato finale
%%% Restituisce la coda di output quando viene raggiunto uno stato di halt
execution_loop(halted_state(_, _, _, _, Out, _), Out).

execution_loop(state(Acc, Pc, Mem, In, Out, Flag), OutTot) :-
    integer(Pc),
```

```
    Pc < 100,
    integer(Acc),
    Acc < 1000,
    proper_length(Mem, Len),
    Len =< 100,
    member(Flag, [flag, noflag]),
    checklist(Mem),
    checklist(In),
    one_instruction(state(Acc, Pc, Mem, In, Out, Flag), NewState),
    !,
    execution_loop(NewState, OutTot).

%%% Remove Comment
%%% Rimuove i commenti da una stringa
remove_comment(Row, Command) :- %% pulizia labels
    split_string(Row, "//", " ", X),
    nth0(0, X, Command, _).

%%% del_blank
%%% Elimina tutti gli elementi uguali alla stringa vuota da una lista
del_blank(_, [], []) :-
    !.

del_blank(X, [X|Xs], Y) :-
    !,
    del_blank(X, Xs, Y).

del_blank(X, [T|Xs], Y) :-
    !,
    del_blank(X, Xs, Y2),
    append([T], Y2, Y).

%%% no_instr
%%% Controlla che un elemento non sia presente in una lista
no_instr(_,[]) :-
    !.

no_instr(X,[X|_]) :-
    !,
    fail.
```

```
no_instr(X,[_|T]) :-
    !,
    no_instr(X,T).

%%% Exec
%%% Esecuzione di un comando assemply
%%% differenziato a seconda del numero di parole contenute
%%% e dalla eventuale presenza di label
exec(_, Row, _) :-
    remove_comment(Row, Command),
    split_string(Command, " ", "", Y),
    del_blank("", Y, Words),
    proper_length(Words, WordsNum),
    WordsNum = 0, !.

exec(_, Row, Instruction) :-
    remove_comment(Row, Command),
    split_string(Command, " ", "", Y),
    del_blank("", Y, Words),
    proper_length(Words, WordsNum),
    WordsNum = 1,
    !,
    single_command(Words, Instruction).

exec(FirstEmptyIndex, Row, Instruction) :-
    remove_comment(Row, Command),
    split_string(Command, " ", "", Y),
    del_blank("", Y, Words),
    proper_length(Words, WordsNum),
    WordsNum = 2,
    nth0(0, Words, Elem),
    string_lower(Elem, Eleml),
    no_instr(Eleml, ["add", "sub", "sta", "lda", "bra", "brz"]),
    no_instr(Eleml, ["brp", "inp", "out", "hlt", "dat"]),
    !,
    command_with_label2(Words, Instruction, FirstEmptyIndex).

exec(_, Row, Instruction) :-
    remove_comment(Row, Command),
    split_string(Command, " ", "", Y),
    del_blank("", Y, Words),
```

```
    proper_length(Words, WordsNum),
    WordsNum = 2,
    !,
    command(Words, Instruction).

exec(FirstEmptyIndex, Row, Instruction) :-
    remove_comment(Row, Command),
    split_string(Command, " ", "", Y),
    del_blank("", Y, Words),
    proper_length(Words, WordsNum),
    WordsNum = 3,
    !,
    command_with_label(Words, Instruction, FirstEmptyIndex).

%%% Normalize
%%% Restituisce qualunque numero in 2 cifre
%%% aggiungendo uno 0 prima nel caso sia un numero compreso tra 0 e 10
normalize(Number, NumberNorm) :-
    string_length(Number, Leng),
    Leng = 1,
    !,
    string_concat("0", Number, NumberNorm).

normalize(Number, Number).

%%% Single Command
%%% Restituisce l'istruzione numerica associata al comando assembly
%%% nel caso di un comando con una singola parola
single_command([Command], Instruction) :-
    string_lower(Command, CommandLower),
    CommandLower = "inp",
    !,
    copy_term("901", Instruction).

single_command([Command], Instruction) :-
    string_lower(Command, CommandLower),
    CommandLower = "out",
    !,
    copy_term("902", Instruction).

single_command([Command], Instruction) :-
```

```
    string_lower(Command, CommandLower),
    CommandLower = "hlt",
    !,
    copy_term("000", Instruction).

single_command([Command], Instruction) :-
    string_lower(Command, CommandLower),
    CommandLower = "dat",
    !,
    copy_term("000", Instruction).

%%% Command
%%% Restituisce l'istruzione numerica associata al comando assembly
%%% nel caso di un comando con 2 parole
%%% (label + singolo comando oppure comando + valore)
command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "add",
    !,
    string_concat("1", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "add",
    !,
    string_concat("1", ValueNorm, Instruction).

command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "sub",
    !,
    string_concat("2", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
```

```
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "sub",
    !,
    string_concat("2", ValueNorm, Instruction).

command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "sta",
    !,
    string_concat("3", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "sta",
    !,
    string_concat("3", ValueNorm, Instruction).

command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "lda",
    !,
    string_concat("5", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "lda",
    !,
    string_concat("5", ValueNorm, Instruction).
```

```
command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "bra",
    !,
    string_concat("6", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "bra",
    !,
    string_concat("6", ValueNorm, Instruction).

command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "brz",
    !,
    string_concat("7", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "brz",
    !,
    string_concat("7", ValueNorm, Instruction).

command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "brp",
    !,
```

```
    string_concat("8", ValueNorm, Instruction).

command([Command, Label], Instruction) :-
    string_upper(Label, LabelUpper),
    tag(LabelUpper, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "brp",
    !,
    string_concat("8", ValueNorm, Instruction).

command([Command, Value], Instruction) :-
    number_string(_, Value),
    string_lower(Command, CommandLower),
    normalize(Value, ValueNorm),
    CommandLower = "dat",
    !,
    copy_term(ValueNorm, Instruction).

%%% Command with Label
%%% Restituisce l'istruzione numerica associata al comando assembly
%%% nel caso di un comando con 3 parole
%%% (label + comando + valore)
command_with_label([_, Command, Value], Instruction, _) :-
    string_lower(Command, CommandLower),
    command([CommandLower, Value], Instruction).

command_with_label2([_, Command], Instruction, _) :-
    string_lower(Command, CommandLower),
    single_command([CommandLower], Instruction).

%%% Replace
%%% Sostituisce l'elemento X nella lista L in posizione I
%%% nel parametro L2
replace(X, L, I, L2) :-
    nth0(X, L, _, L3),
    nth0(X, L2, I, L3).

%%% Row to Mem
%%% Richiama ricorsivamente exec riempiendo la memoria
row_to_mem([], [], 0, []) :-
```

```
    !.

row_to_mem([LastRow], Mem, Pc, MemOut) :-
    !,
    exec(Pc, LastRow, Instruction),
    Pc =< 100,
    replace(Pc, Mem, Instruction, MemOut).

row_to_mem([Row|OtherRows], Mem, Pc, MemOut) :-
    exec(Pc, Row, Instruction),
    Pc =< 100,
    replace(Pc, Mem, Instruction, MemOutNew),
    PcNew is Pc+1,
    row_to_mem(OtherRows, MemOutNew, PcNew, MemOut).

%%% Save Labels
%%% Salva tutte le label presenti nel file .lmc
%%% aggiungendoli con gli assert alla base di conoscenza
save_labels([], 0) :-
    !.

save_labels([LastRow], Pc) :-
    Pc < 100,
    remove_comment(LastRow, Command),
    split_string(Command, " ", " ", Words),
    proper_length(Words, WordsNum),
    WordsNum < 2,
    !.

save_labels([LastRow], Pc) :-
    Pc < 100,
    remove_comment(LastRow, Command),
    split_string(Command, " ", " ", Words),
    proper_length(Words, WordsNum),
    WordsNum = 2,
    nth0(0, Words, Label),
    no_instr(Label, ["add", "sub", "sta", "lda", "bra", "brz"]),
    no_instr(Label, ["brp", "inp", "out", "hlt", "dat"]),
    !,
    string_upper(Label, LabelUpper),
    assertz(tag(LabelUpper, Pc)).
```

```
save_labels([LastRow], Pc) :-
    Pc < 100,
    remove_comment(LastRow, Command),
    split_string(Command, " ", " ", Words),
    proper_length(Words, WordsNum),
    WordsNum = 2,
    !.

save_labels([LastRow], Pc) :-
    Pc < 100,
    remove_comment(LastRow, Command),
    split_string(Command, " ", " ", Words),
    proper_length(Words, WordsNum),
    WordsNum = 3,
    !,
    nth0(0, Words, Label),
    string_upper(Label, LabelUpper),
    assertz(tag(LabelUpper, Pc)).

save_labels([Row|OtherRows], Pc) :-
    !,
    save_labels([Row], Pc),
    PcNew is Pc+1,
    save_labels(OtherRows, PcNew).

%%% no_comment
%%% Rimuove ogni riga di commento dalla lista di stringhe
no_comment([], []).

no_comment([H | T], [H2| T2]) :-
    remove_comment(H, H2),
    no_comment(T, T2).

%%% memg
%%% Riempie il fondo della Mem con gli 0
memg(L, NewMem) :-
    proper_length(L, X),
    X<100,
    append(["0"], L, Mem),
    memg(Mem, NewMem),
```

```
    !.

memg(Mem, Mem) :-
    !.

%%% mem_to_number
%%% Converte la lista di stringhe in lista di interi
mem_to_number(Mem, MemNumber, X) :-
    X<100,
    nth0(X, Mem, Elem),
    number_string(Num, Elem),
    NewX is X+1,
    replace(X, Mem, Num, MemNumber2),
    mem_to_number(MemNumber2, MemNumber, NewX),
    !.

mem_to_number(Mem, Mem, _) :-
    !.

%%% lmc_load
%%% Legge il file .lmc e genera la memoria
%%% dopo gli opportuni controlli
%%% (rimozione commenti, salvataggio e sostituzione label)
lmc_load(Filename, Mem) :-
    open(Filename, read, Input),
    read_string(Input, _, FileTxt),
    split_string(FileTxt, "\n", " ", Rows),
    del_blank("", Rows, ClearRows),
    no_comment(ClearRows, NoCommentList),
    delete(NoCommentList, "", CommandList),
    proper_length(CommandList, NumberCommand),
    NumberCommand =< 100,
    save_labels(CommandList, 0),
    memg([], MemV),
    row_to_mem(CommandList, MemV, 0, Mem).

%%% lmc_run
%%% Esegue il lmc_load del file .lmc,
%%% converte gli elementi della memoria in numeri interi
%%% e la passa come memoria dello stato iniziale dell'execution_loop.
%%% Infine rimuove tutte le label aggiunte alla base di conoscenza
```

```
lmc_run(Filename, In, Output) :-
    %% Generazione della memoria
    lmc_load(Filename, Mem),
    %% Conversione memoria in interi
    mem_to_number(Mem, MemNumber, 0),
    %% Pulizia delle labels
    retractall(tag(_,_)),
    %% Esecuzione
    execution_loop(state(0, 0, MemNumber, In, [], noflag), Output),
    !.
```

4.2 Lisp

```
;;;; Little Man Computer
;;;; Progetto di Linguaggi di Programmazione
;;;; Anno accademico 2018-2019
;;;; Appello di Gennaio 2019
;;;; Davide Cozzi 829827
;;;; Gabriele De Rosa 829835

;;; Replace
;;; Sostituisce tutte le occorrenze di un valore in una lista
(defun repl (list n elem)
  (if (= n 0)
      (append (list elem) (cdr list))
      (append (list (car list)) (repl (cdr list) (- n 1) elem))))

;;; Is In List
;;; Restituisce T se un elemento è nella lista
(defun is-in-list (elem list)
  (cond ((equal elem (car list)) T)
        ((equal list NIL) NIL)
        (T (is-in-list elem (cdr list)))))

;;; List Parse Integer
;;; Converte tutti gli elementi numerici della lista in interi
(defun list-parse-integer (l)
  (cond ((equal l NIL)
         NIL)
        (T
```

```
          (cons (parse-integer (car l)) (list-parse-integer (cdr l))))))

;;; List of Zero
;;; Crea una lista di n zeri
(defun list-of-zero (n)
  (cond ((= n 0) NIL)
        (T (cons "0" (list-of-zero (- n 1))))))

;;; Addizione
;;; Instruction: 1xx
(defun addizione (Acc Pointer Mem)
  (cons (mod (+ Acc (nth Pointer Mem)) 1000)
        (if (< (+ Acc (nth Pointer Mem)) 1000)
            'NOFLAG
            'FLAG)))

;;; Sottrazione
;;; Instruction: 2xx
(defun sottrazione (Acc Pointer Mem)
  (cons (mod (- Acc (nth Pointer Mem)) 1000)
        (if (>= (- Acc (nth Pointer Mem)) 0)
            'NOFLAG
            'FLAG)))

;;; Store
;;; Instruction: 3xx
(defun store (Acc Pointer Mem)
  (repl Mem Pointer Acc))

;;; Load
;;; Instruction: 5xx
(defun lload (Pointer Mem)
  (nth Pointer Mem))

;;; Branch
;;; Instruction: 6xx
(defun branch (Pc) (+ Pc 0))

;;; Branch If Zero
;;; Instruction: 7xx
(defun branch-if-zero (Pc Acc Pointer Flag)
```

```
  (if (and (= Acc 0) (equal Flag 'NOFLAG))
      Pointer
      (mod (+ Pc 1) 100)))

;;; Branch If Zero
;;; Instruction: 8xx
(defun branch-if-positive (Pc Pointer Flag)
  (if (equal Flag 'NOFLAG)
      Pointer
      (mod (+ Pc 1) 100)))

;;; Input
;;; Instruction: 901
(defun input (In)
  (cons (first In) (rest In)))

;;; Output
;;; Instruction: 902
(defun output (Acc Out)
  (append Out (list Acc)))

;;; One Instruction
;;; Passaggio da uno stato iniziale ad uno stato finale
;;; a seconda del valore dell'opcode (Istr)
(defun one-instruction (state)
  (let ((ACC (nth 2 state))
        (PC (nth 4 state))
        (MEM (nth 6 state))
        (IN (nth 8 state))
        (OUT (nth 10 state))
        (FLAG (nth 12 state)))
    (let ((ISTR (nth PC MEM)))
      (let ((POINTER (mod ISTR 100)))
        (cond ((and (>= ISTR 0) (< ISTR 100))
               (list 'halted-state ':acc ACC ':pc PC ':mem MEM
                     ':in IN ':out OUT ':flag FLAG))
              ((and (> ISTR 99) (< ISTR 200))
               (list 'state ':acc (car (addizione ACC POINTER MEM))
                     ':pc (mod (+ PC 1) 100) ':mem MEM ':in IN
                     ':out OUT ':flag (cdr (addizione ACC POINTER MEM))))
              ((and (> ISTR 199) (< ISTR 300))
```

```
                (list 'state ':acc (car (sottrazione ACC POINTER MEM))
                      ':pc (mod (+ PC 1) 100) ':mem MEM ':in IN
                      ':out OUT ':flag (cdr (sottrazione ACC POINTER MEM))))
               ((and (> ISTR 299) (< ISTR 400))
                (list 'state ':acc ACC ':pc (mod (+ PC 1) 100)
                      ':mem (store ACC POINTER MEM)
                      ':in IN ':out OUT ':flag FLAG))
               ((and (> ISTR 499) (< ISTR 600))
                (list 'state ':acc (lload POINTER MEM) ':pc (mod (+ PC 1) 100)
                      ':mem MEM ':in IN ':out OUT ':flag FLAG))
               ((and (> ISTR 599) (< ISTR 700))
                (list 'state ':acc ACC ':pc (branch POINTER) ':mem MEM
                      ':in IN ':out OUT ':flag FLAG))
               ((and (> ISTR 699) (< ISTR 800))
                (list 'state ':acc ACC
                      ':pc (branch-if-zero PC ACC POINTER FLAG)
                      ':mem MEM ':in IN ':out OUT ':flag FLAG))
               ((and (> ISTR 799) (< ISTR 900))
                (list 'state ':acc ACC
                      ':pc (branch-if-positive PC POINTER FLAG) ':mem MEM
                      ':in IN ':out OUT ':flag FLAG))
               ((and (= ISTR 901) (not (= (list-length IN) 0)))
                (list 'state ':acc (car (input IN)) ':pc (mod (+ PC 1) 100)
                      ':mem MEM ':in (cdr (input IN)) ':out OUT ':flag FLAG))
               ((= ISTR 902)
                (list 'state ':acc ACC ':pc (mod (+ PC 1) 100) ':mem MEM
                      ':in IN ':out (output ACC OUT) ':flag FLAG)))))))

;;; Check List
;;; Controlla che una lista non abbia valori superiori a 999
(defun checklist (l)
  (cond ((equal l NIL) T)
    (T (if (<= ( car l) 999) (checklist (cdr l))))))

;;; Execution Loop
;;; Cicla dallo stato iniziale allo stato finale
;;; Dopo aver controllato lo stato inziale
;;; Restituisce la coda di output quando viene raggiunto uno stato di halt
(defun execution-loop (state)
  (cond ((not (equal (nth 1 state) ':acc)) NIL)
        ((not (integerp (nth 2 state))) NIL) ; check acc
```

```
        ((not (equal (nth 3 state) ':pc)) NIL)
        ((not (integerp (nth 4 state))) NIL) ; check pc
        ((not (equal (nth 5 state) ':mem)) NIL)
        ((> (list-length (nth 6 state)) 100) NIL) ; check mem length
        ((not (checklist (nth 6 state))) NIL) ; check mem
        ((not (equal (nth 7 state) ':in)) NIL)
        ((not (checklist (nth 8 state))) NIL) ; check input
        ((not (equal (nth 9 state) ':out)) NIL)
        ((not (equal (nth 11 state) ':flag)) NIL)
        ((not (is-in-list (nth 12 state) '(FLAG NOFLAG))) NIL) ; check flag
        ((equal state NIL) NIL) ; istruzione errata
        ((and (< (nth 4 state) 100) (equal (nth 0 state) 'state))
         (execution-loop (one-instruction state)))
        ((equal (nth 0 state) 'halted-state)
          (nth 10 state))))

;;; Remove Comment
;;; Rimuove i commenti da una stringa
(defun remove-comment (row)
  (string-trim " " (subseq row 0 (search "//" row))))

;;; Split
;;; Splitta una stringa
;;; Restituisce una lista
(defun split-core (str index)
  (cond ((= (length str) 0) (list str))
        ((>= index (length str)) (list str))
        ((equal (char str index) #\ )
         (append (list (subseq str 0 index))
                 (split-core (subseq str (+ index 1)) 0)))
        (T (split-core str (+ index 1)))))

(defun split (str)
  (split-core str 0))

;;; Remove Blank
;;; Elimina tutti gli elementi uguali alla stringa vuota da una lista
(defun remove-blank (lista)
  (cond ((= (list-length lista) 0) NIL)
        ((equal (remove-comment (car lista)) "") (remove-blank (cdr lista)))
        (T  (cons (car lista) (remove-blank (cdr lista))))))
```

```
;;; Normalize
;;; Restituisce qualunque numero in 2 cifre
;;; aggiungendo uno 0 prima nel caso sia un numero compreso tra 0 e 10
(defun normalize (num)
  (cond ((= (length num ) 1) (concatenate 'string "0" num))
        (T num)))

;;; Get Value
;;; Restituisce il valore associato ad una etichetta
;;; o il valore stesso
(defun get-value-of (tag tags)
  (cond ((equal tags NIL) NIL)
        ((equal (string-upcase tag)
                (string-upcase (first (first tags)))) (cdr (first tags)))
        (T (get-value-of tag (cdr tags)))))

(defun value-of (word tags)
  (cond ((equal (get-value-of word tags) NIL) word)
        (T (get-value-of word tags))))

;;; Command to Istr
;;; Restituisce l'istruzione numerica associata al comando assembly
(defun command-to-instr (command pointer tags)
  (cond ((and (equal (string-upcase (car command)) "ADD")
              (not (equal (cdr command) NIL)))
         (concatenate 'string "1"
                              (normalize (value-of (second command) tags))))
        ((and (equal (string-upcase (car command)) "SUB")
              (not (equal (cdr command) NIL)))
         (concatenate 'string "2"
                              (normalize (value-of (second command) tags))))
        ((and (equal (string-upcase (car command)) "STA")
              (not (equal (cdr command) NIL)))
         (concatenate 'string "3"
                              (normalize (value-of (second command) tags))))
        ((and (equal (string-upcase (car command)) "LDA")
              (not (equal (cdr command) NIL)))
         (concatenate 'string "5"
                              (normalize (value-of (second command) tags))))
        ((and (equal (string-upcase (car command)) "BRA")
```

```
            (not (equal (cdr command) NIL)))
       (concatenate 'string "6"
                            (normalize (value-of (second command) tags))))
      ((and (equal (string-upcase (car command)) "BRZ")
            (not (equal (cdr command) NIL)))
       (concatenate 'string "7"
                            (normalize (value-of (second command) tags))))
      ((and (equal (string-upcase (car command)) "BRP")
            (not (equal (cdr command) NIL)))
       (concatenate 'string "8"
                            (normalize (value-of (second command) tags))))
      ((and (equal (string-upcase (car command)) "INP")
            (equal (cdr command) NIL))
       "901")
      ((and (equal (string-upcase (car command)) "OUT")
            (equal (cdr command) NIL))
       "902")
      ((and (equal (string-upcase (car command)) "HLT")
            (equal (cdr command) NIL))
       "000")
      ((equal (string-upcase (car command)) "DAT")
       (cond ((equal (cdr command) NIL) "000")
             (T (value-of (second command) tags))))
      (T
        (command-to-instr (cdr command) pointer tags))))

;;; Read Lines
;;; Legge il file
;;; Restituisce una lista contenente le righe del file di testo letto
(defun read-all-lines-helper (stream)
  (let ((line (read-line stream NIL NIL)))
    (when line (cons line (read-all-lines-helper stream)))))

(defun read-all-lines (filename)
  (with-open-file (f filename :direction :input)
                  (read-all-lines-helper f)))

;;; Is Istr
(defun is-istr (command)
  (is-in-list (string-upcase (car command))
              (list "ADD" "SUB" "STA" "LDA" "BRA" "BRZ"
```

```
                    "BRP" "INP" "OUT" "HLT" "DAT")))

;;; Get Tags
;;; Restituisce la lista di etichette del file
(defun get-tags (commands pc)
  (cond ((equal commands NIL) NIL)
        (T
    (cond ((not
            (is-istr (remove-blank (split (remove-comment (car commands))))))
      (append (list (cons (string-upcase
        (car (remove-blank (split (remove-comment (car commands))))))
                            (write-to-string pc)))
                    (get-tags (cdr commands) (+ pc 1))))
      (T (get-tags (cdr commands) (+ pc 1)))))))

;;; Get Mem
;;; Riempie il fondo della Mem con gli 0
(defun get-mem (mem)
  (nconc mem (list-of-zero (- 100 (list-length mem)))))

;;; Commands to Mem
;;; Aggiunge i vari comandi assembly alla memoria
(defun commands-to-mem (commands pointer tags)
  (cond ((equal commands NIL) NIL)
        (T (cons (command-to-instr (remove-blank (split
                                        (remove-comment (car commands))))
                                    pointer tags)
                 (commands-to-mem (cdr commands) (+ pointer 1) tags)))))

;;; LMC Load
;;; Legge il file .lmc e genera la memoria convertendola in interi
;;; dopo gli opportuni controlli
(defun lmc-load (filename)
  (list-parse-integer
    (get-mem (commands-to-mem (remove-blank (read-all-lines filename))
                              0 (get-tags (remove-blank
                                            (read-all-lines filename))
                                          0)))))

;;; LMC Run
;;; Esegue il lmc_load del file .lmc,
```

```
;;; e passa la memoria ottenuta allo stato iniziale dell'execution-loop.
(defun lmc-run (filename input)
  (execution-loop (list 'state ':acc 0 ':pc 0 ':mem (lmc-load filename)
                        ':in input ':out '() ':flag 'NOFLAG)))
```