“Φιλοσοφίες” γλωσσών προγραμματισμού

Νίκος Παπασπύρου, Στάθης Ζάχος

4 κύρια “μοντέλα”
ή “στυλ”
ή “φιλοσοφίες”
προγραμματισμού

#1

Προστακτικός προγραμματισμός
Imperative programming

Προστακτικός προγραμματισμός

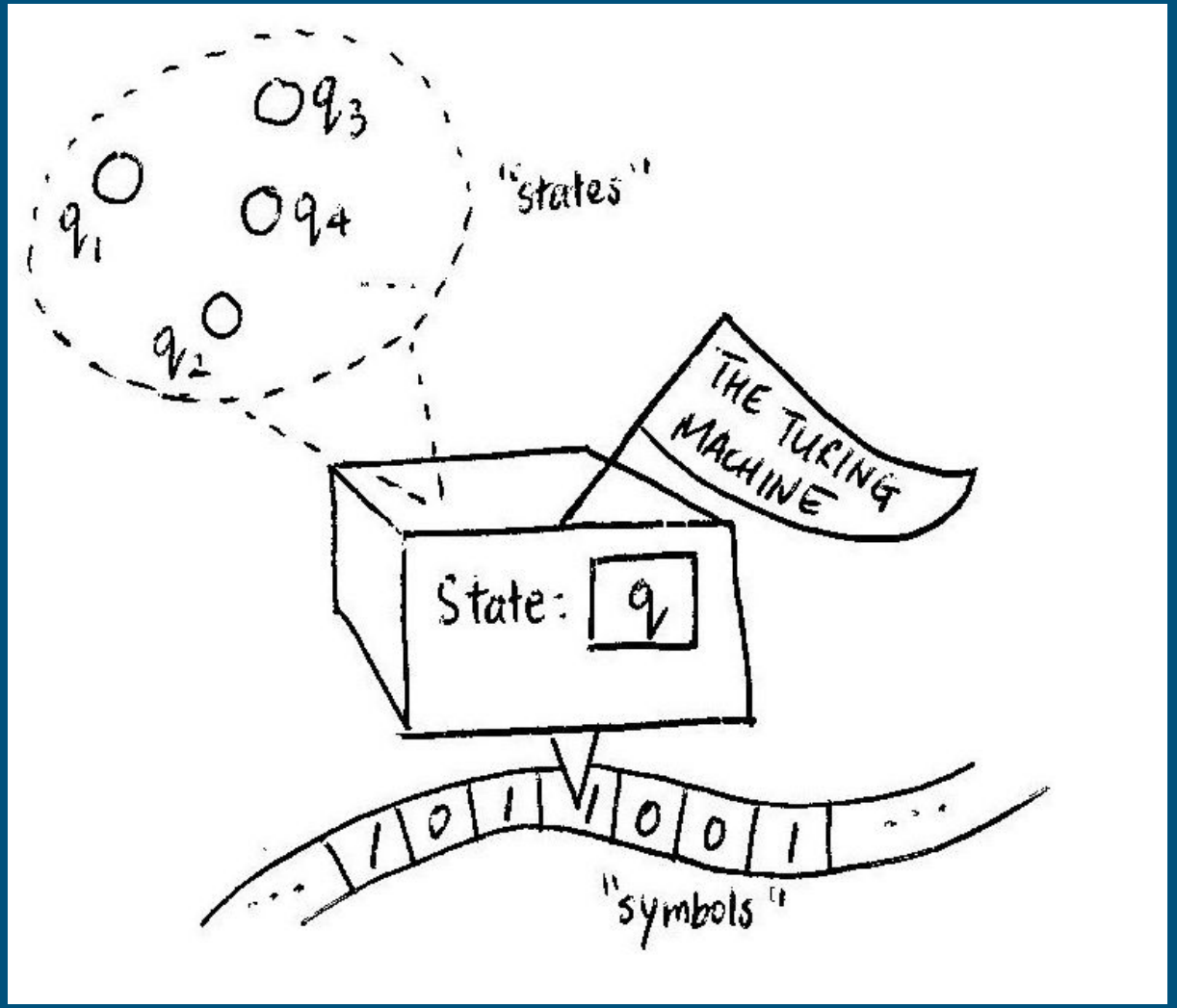
Fortran, Algol, Cobol
Pascal, C, Ada

- Ουσιαστικά, το στυλ προγραμματισμού που μάθαμε σε αυτό το εξάμηνο
- **Εντολές** που αλλάζουν την **κατάσταση** του υπολογιστή
- Ο υπολογισμός οδηγείται από τη σειρά των εντολών
- **Μεταβλητές** και **ανάθεση**
- **Βρόχοι** και **επανάληψη**

Υπολογιστικό μοντέλο:

Μηχανή Turing
(Turing machine)

[Turing, ~1930]



#2

Συναρτησιακός προγραμματισμός
Functional programming

Συναρτησιακός προγραμματισμός

Lisp, Scheme, Erlang, ML, Haskell

- **Συναρτήσεις** που δηλώνουν τον τρόπο **εξάρτησης** μεταξύ διαφόρων **τιμών δεδομένων**
- Ο υπολογισμός οδηγείται από τις εξαρτήσεις
- **“Μεταβλητές”** και συναρτήσεις όπως στα μαθηματικά
- **Αναδρομή**

Υπολογιστικό μοντέλο:

Λογισμός λ
(λ-calculus)

[Church, ~1930]

$$
\begin{aligned}
& (\lambda f.\lambda x.f\ (f\ x))(\lambda f.\lambda x.f\ (f\ x)) \\
\rightsquigarrow_{\alpha}\ & (\lambda f.\lambda x.f\ (f\ x))(\lambda g.\lambda y.g\ (g\ y)) \\
\rightsquigarrow_{\beta}\ & \lambda x.(\lambda g.\lambda y.g\ (g\ y))((\lambda g.\lambda y.g\ (g\ y))\ x) \\
\rightsquigarrow_{\beta}\ & \lambda x.(\lambda g.\lambda y.g\ (g\ y))(\lambda y.x\ (x\ y)) \\
\rightsquigarrow_{\alpha}\ & \lambda x.(\lambda g.\lambda y.g\ (g\ y))(\lambda z.x\ (x\ z)) \\
\rightsquigarrow_{\beta}\ & \lambda x.\lambda y.(\lambda z.x\ (x\ z))((\lambda z.x\ (x\ z))\ y) \\
\rightsquigarrow_{\beta}\ & \lambda x.\lambda y.(\lambda z.x\ (x\ z))(x\ (x\ y))) \\
\rightsquigarrow_{\beta}\ & \lambda x.\lambda y.x\ (x\ (x\ (x\ y)))
\end{aligned}
$$

Αρχές του συναρτησιακού προγραμματισμού

- Αγνές συναρτήσεις — pure functions
- Αμετάβλητα δεδομένα — immutable data
- Αναδρομή — recursion
- Διαφάνεια αναφοράς — referential transparency
- Συναρτήσεις υψηλής τάξης — higher-order functions
- Currying — currying
- Πρόθυμη / οκνηρή αποτίμηση — eager / lazy evaluation

Αγνές συναρτήσεις

Επιστρέφουν πάντα το **ίδιο αποτέλεσμα** για τις **ίδιες τιμές των παραμέτρων**

Το αποτέλεσμα δεν εξαρτάται από κανενός είδους “**κατάσταση**”

Δεν προκαλούν “**παρενέργειες**” (side effects)

```
int pure_function(int x) {
  return x*(x+1);
}
```

Αγνές συναρτήσεις

```
bool valid(date d) {
  int days_of_month;
  switch (d.month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
      days_of_month = 31; break;
    case 4: case 6: case 9: case 11:
      days_of_month = 30; break;
    case 2:
      if (leap(d.year)) days_of_month = 29;
      else              days_of_month = 28;
      break;
    default:
      return false;
  }
  return 1 <= d.day && d.day <= days_of_month;
}
```

Μη αγνές συναρτήσεις

```
int x = 0;

int global_state() {
  return x++;
}
```

```
void doing_IO(string filename) {
  ifstream f(filename);
  string line;
  getline(f, line);
  cout << "Here's the first line: "
       << line << endl;
}
```

```
string reading_the_time() {
  auto t = time(nullptr);
  auto tm = localtime(&t);
  if (tm->tm_hour < 8) return "sleeping";
  else                 return "working";
}
```

```
inc :: Int → Int
inc n = n + 1

f :: Int → Int
f t = t * inc t

g :: (Int, Int) → Int
g (a,b) = f a + f (f b)

x :: Bool
x = g (1, 2) < 42
```

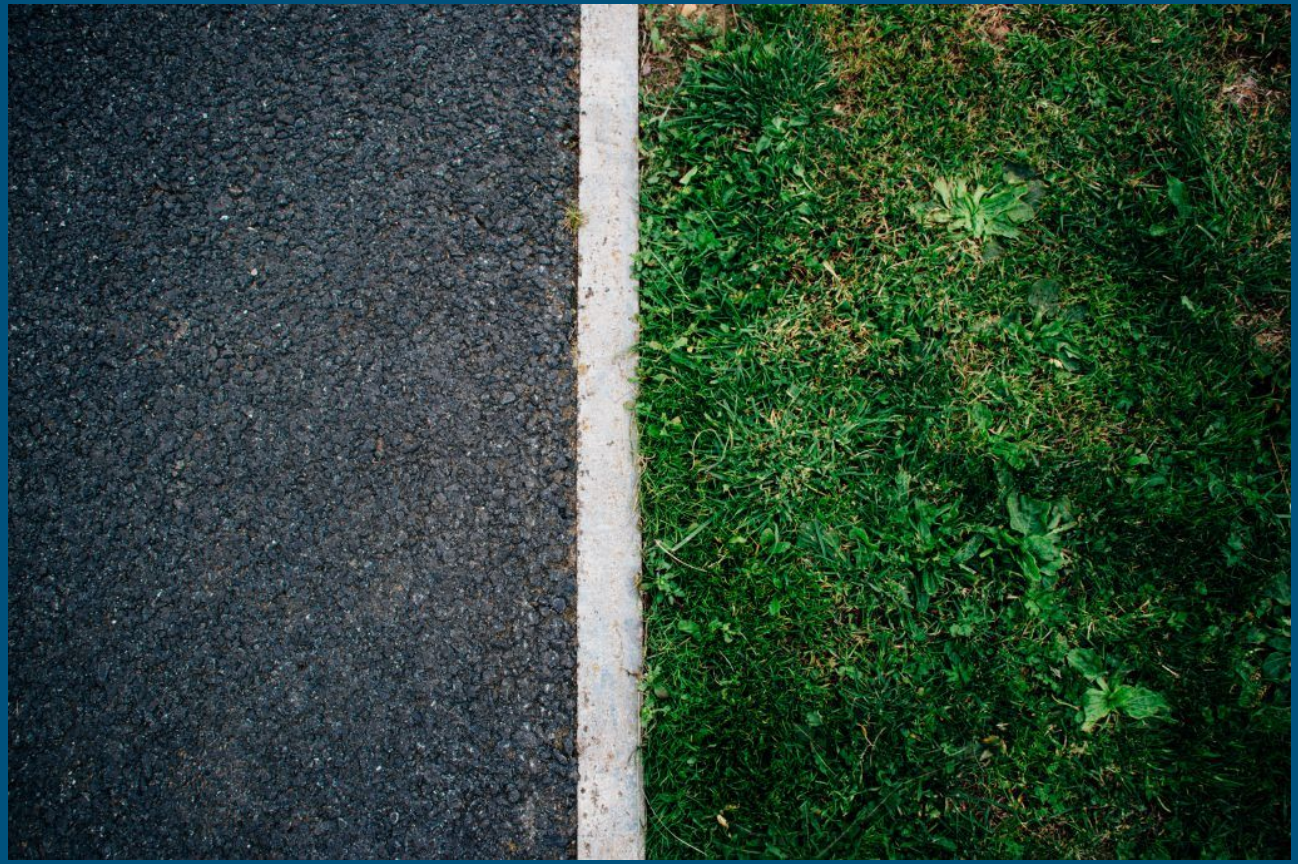
```
    f 6
→   6 * inc 6
→   6 * (6 + 1)
→   6 * 7
→   42
```

αναγωγές (reductions)

Αμετάβλητα δεδομένα

Το **περιεχόμενό** τους **δεν μπορεί να μεταβληθεί**

Οι “**μεταβλητές**” στο συναρτησιακό προγραμματισμό δεν είναι παρά “**σταθερές**”



```
int main() {
  int x = 42;
  int y = 17;
  ...
  x = x * y;
  ...
  y++;
  ...
}
```

Mutable

```
class intset {
  intset();              // constructor: empty
  bool   isEmpty();
  bool   member(int x);
  void   add(int x);
  void   remove(int x);
};

int main() {
  intset S;                          // {}
  S.add(17);                         // {17}
  S.add(42);                         // {17,42}
  cout << S.isEmpty() << endl;       // false
  cout << S.member(17) << endl       // true
  S.remove(17);                      // {42}
  cout << S.member(17) << endl;      // false
  cout << S.member(42) << endl       // true
  cout << S.isEmpty() << endl;       // false
}
```

```
int main() {
  const int x = 42;
  const int y = 17;
  const int z = x * y;
  const int w = f(z);
  ...
}
```

Immutable

```
class intset {
  intset();              // constructor: empty
  bool   isEmpty();
  bool   member(int x);
  intset add(int x);
  intset remove(int x);
};

int main() {
  intset S0;                          // {}
  intset S1 = S0.add(17);             // {17}
  intset S2 = S1.add(42);             // {17,42}
  cout << S0.isEmpty() << endl;       // true
  cout << S1.member(42) << endl       // false
  cout << S2.member(42) << endl       // true
  intset S3 = S1.remove(17);          // {}
  cout << S1.member(17) << endl;      // true
  cout << S3.isEmpty() << endl;       // true
}
```

```haskell
small :: [Int]
small = [1,2,3,4,5]

bigger :: [Int]
bigger = [17..42]

length :: [a] → Int
length []     = 0
length (h:t) = 1 + length t

n :: Int
n = length bigger
```

Λίστες

Αναδρομή
recursion

Ταίριασμα προτύπων
pattern matching

Haskell

Διαφάνεια αναφοράς

(links: #1, #2)

» Μπορείς να **αντικαταστήσεις** κάτι με κάτι άλλο **“ίσο”**.

Σε ένα πρόγραμμα, μπορεί κανείς να **αντικαταστήσει** μία έκφραση με μία άλλη **“ίση”** (ισοδύναμη) έκφραση, χωρίς να μεταβάλει τη συμπεριφορά του προγράμματος.

βλ. Μαθηματικά, Φιλοσοφία, Γλωσσολογία...

Διαφάνεια αναφοράς

$$a + a = 2a$$

$$x = a \Rightarrow a + a = x + x$$

```
int main() {
  cout << f(42) + f(42) << endl;
}
```

```
int main() {
  cout << 2 * f(42) << endl;
}
```

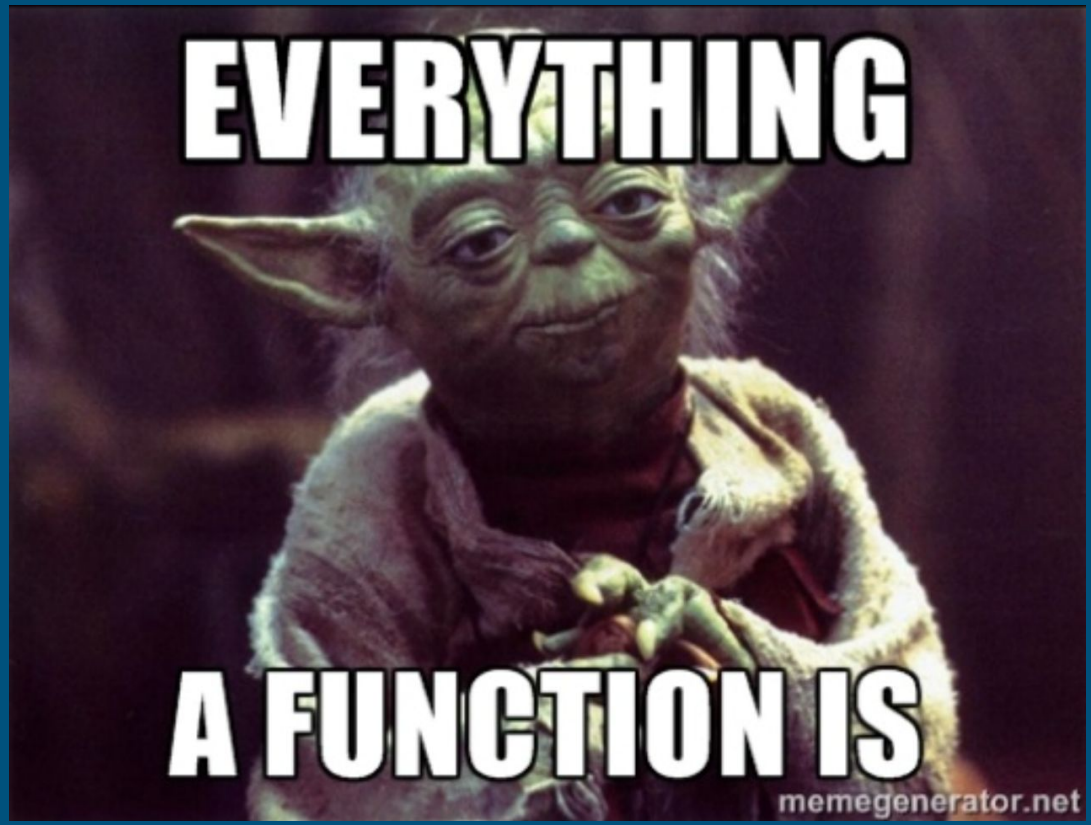
```
int main() {
  int x = f(42);
  cout << x + x << endl;
}
```

αν η f είναι αγνή συνάρτηση... 🙂
αν έχει παρενέργειες... 😔
αν τροποποιεί το περιεχόμενο μεταβλητών δεδομένων... 😔

Συναρτήσεις υψηλής τάξης

- Δέχονται συναρτήσεις ως **παραμέτρους**
- Επιστρέφουν συναρτήσεις ως **αποτέλεσμα**

Με λίγα λόγια, στο συναρτησιακό προγραμματισμό, οι συναρτήσεις δε διαφέρουν από τα άλλα είδη τιμών



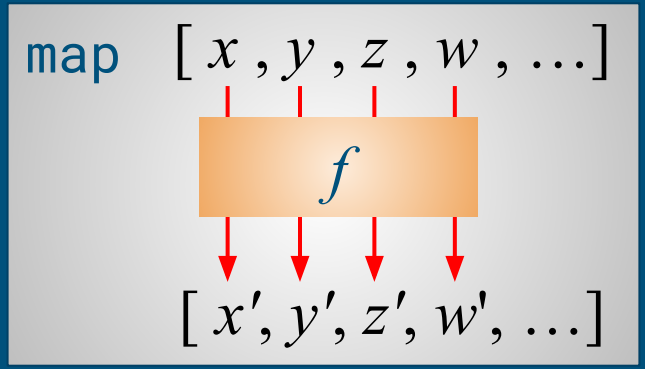
```
find_root :: (Double → Double) → Double
find_root = ...

f :: Double → Double
f x = 2*x - 10

g :: Double → Double
g x = x^3 - 2*x^2 + x - 7

x1,x2 :: Double
x1 = find_root f
x2 = find_root g
```

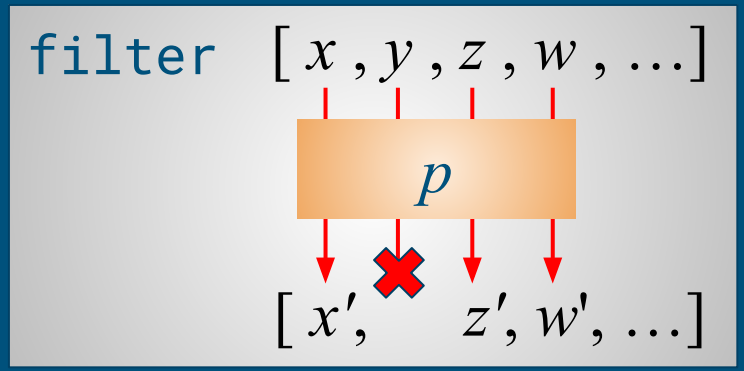
x1 = 5.0

x2 = 2.6311

Haskell

```haskell
map :: (a → b) → [a] → [b]
map f []     = []
map f (h:t) = f h : map f t

filter :: (a → Bool) → [a] → [a]
filter p []     = []
filter p (h:t) = if p h then h : filter p t
                        else filter p t
```





Haskell

Currying

Συναρτήσεις που δέχονται **περισσότερες** παραμέτρους, **μία τη φορά**!

Haskell B. Curry
(1900-1982)

$f : A \times B \to C$

$f(a, b) = \ldots$

$f(17, 42)$

$f' : A \to (B \to C)$

$f'(a) = g_a$

$g_a : B \to C$

$g_a(b) = \ldots$

$g(17)(42)$

```haskell
max :: (Int,Int) → Int
max (a,b) = if a < b then b else a

max' :: Int → Int → Int
max' a b = if a < b then b else a

list :: [Int]
list = [1..10]

mapped :: [Int]
mapped = map (max' 5) list
```

Uncurried

Curried

[5,5,5,5,5,6,7,8,9,10]

Haskell

Πρόθυμη / οκνηρή αποτίμηση

Πρόθυμη (eager) αποτίμηση:
πριν κληθεί μια συνάρτηση, αποτιμώνται οι παράμετροί της

Οκνηρή (lazy) αποτίμηση:
οι παράμετροι μιας συνάρτησης δεν αποτιμώνται παρά μόνο όταν χρειαστεί η τιμή τους

Στρατηγική αποτίμησης:
σειρά που γίνονται οι αναγωγές

Οι περισσότερες γλώσσες προγραμματισμού (συναρτησιακού ή μη) χρησιμοποιούν πρόθυμη αποτίμηση.

Εξαίρεση, η **Haskell**.

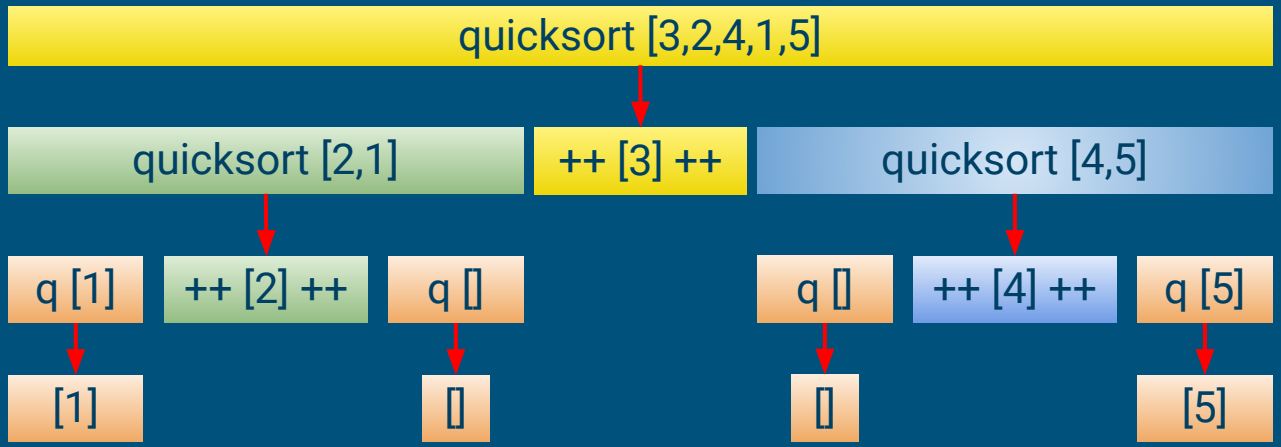
```haskell
and :: Bool → Bool → Bool
and p q = if p then q else False

start_counting :: Int → Bool
start_counting n = start_counting (n+1)
```

```
        and False (start_counting 0)
  →     if False then start_counting 0 else False
  →     False
```

Τι θα γινόταν σε μια γλώσσα με πρόθυμη αποτίμηση;

```
quicksort :: [Int] → [Int]
quicksort []     = []
quicksort (x:xs) = quicksort lt ++ [x] ++ quicksort ge
  where lt = [y | y ← xs, y < x]
        ge = [y | y ← xs, y >= x]
```

quicksort [3,2,4,1,5]




```
x = 3
xs = [2,4,1,5]
lt = [2,1]
ge = [4,5]
```

→ [1,2,3,4,5]

Haskell

Συναρτησιακός προγραμματισμός: + και −

+ Συντομία κώδικα
+ Ευκολία στην κατανόηση
+ Αφαίρεση και συνθεσιμότητα
+ Ευκολία στη μαθηματική απόδειξη ορθότητας
+ Αυτόματη διαχείριση μνήμης

− Μικρότερη ταχύτητα εκτέλεσης
− Μεγαλύτερη ανάγκη σε μνήμη
− Δυσκολία στην κωδικοποίηση προστακτικών αλγορίθμων

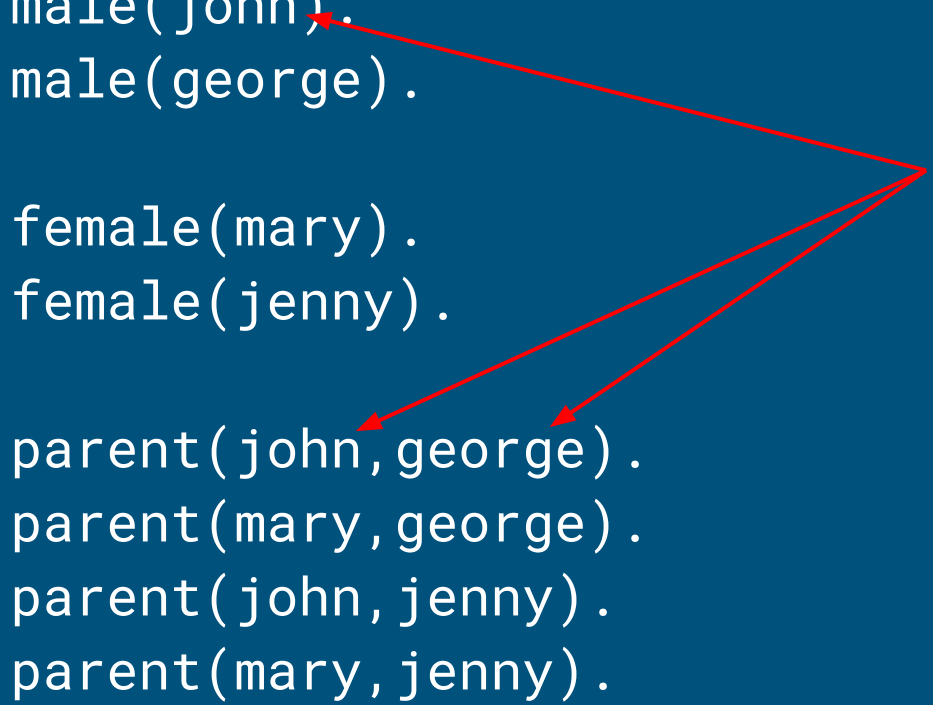
#3

Λογικός προγραμματισμός
Logic programming

Λογικός προγραμματισμός

Prolog, (Datalog)

- **Λογικές προτάσεις** που περιγράφουν **γεγονότα** και **συμπερασματικούς** κανόνες
- Ο υπολογισμός οδηγείται από τη διαδικασία της **επίλυσης** (resolution)
- **Κατηγορήματα** όπως στη μαθηματική λογική
- **Αναδρομή**



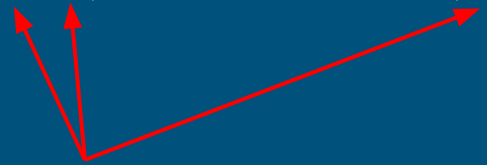
Άτομα (atoms)
π.χ. `john`, `george`, `mary`

Κατηγορήματα (predicates)
π.χ. `male/1`, `female/1`, `parent/2`

Γεγονότα (facts)

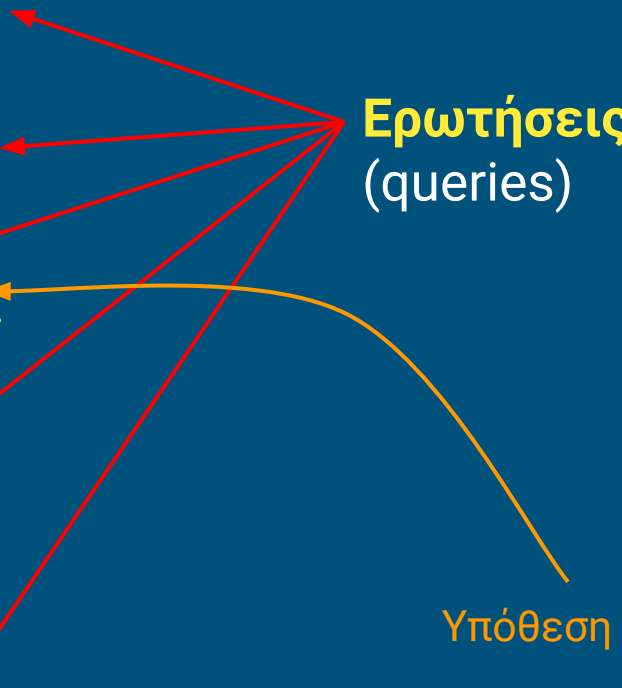
```
father(X,Y) :- parent(X,Y), male(X).

mother(X,Y) :- parent(X,Y), female(X).

human(X) :- male(X).
human(X) :- female(X).

brother(X,Y) :- male(X), parent(Z,X), parent(Z,Y).

sister(X,Y) :- female(X), parent(Z,X), parent(Z,Y).
```

Κανόνες (rules)



Μεταβλητές (variables)
π.χ. X, Y, Z, Student

```
?- male(john).
yes
?- male(mary).
no
?- male(peter).
no
?- male(X).
X = john ;
X = george ;
no
?- brother(george,jenny).
yes
```



Ερωτήσεις
(queries)

```
male(john).
male(george).

parent(john,george).
parent(mary,george).
parent(john,jenny).
parent(mary,jenny).

brother(X,Y) :-
  male(X),
  parent(Z,X),
  parent(Z,Y).
```

Υπόθεση "**κλειστού κόσμου**".

Τα μόνα γνωστά δεδομένα είναι αυτά που αναφέρονται στα γεγονότα και τους κανόνες.

Δηλαδή, ό,τι δεν είναι γνωστό πως είναι αληθές θεωρείται ψευδές.

```
?- human(X).
X = john;
X = george;
X = mary;
X = jenny;
no

?- mother(X,george).
X = mary;
no

?- sister(X,Y).
X = jenny, Y = george;
X = jenny, Y = jenny;
X = jenny, Y = george;
X = jenny, Y = jenny;
no
```

```
male(john).
male(george).

female(mary).
female(jenny).

parent(john,george).
parent(mary,george).
parent(john,jenny).
parent(mary,jenny).

mother(X,Y) :-
  parent(X,Y),
  female(X).

human(X) :- male(X).
human(X) :- female(X).

sister(X,Y) :-
  female(X),
  parent(Z,X),
  parent(Z,Y).
```

Prolog

```
rational(X) :-
  X = A/B,
  integer(A),
  integer(B),
  not(B = 0).

factorial(0,1).
factorial(N,NF) :-
  N > 0,
  P is N-1,
  factorial(P,PF),
  NF is N*PF.
```

```
gcd(X,0,X) :- X > 0.

gcd(0,Y,Y) :- Y > 0.

gcd(X,Y,Z) :-
  X > Y, Y > 0,
  M is X mod Y,
  gcd(M,Y,Z).

gcd(X,Y,Z) :-
  X <= Y, X > 0,
  M is Y mod X,
  gcd(X,M,Z).
```

Prolog

```
mergesort([],[]).

mergesort(U,S) :-
  split(U,L,R),
  mergesort(L,SL),
  mergesort(R,SR),
  merge(SL,SR,S).

split([],[],[]).

split([X],[X],[]).

split([L,R|T],[L|LT],[R|RT]) :-
  split(T,LT,RT).
```

```
merge([],RS,RS).

merge(LS,[],LS).

merge([L|LS],[R|RS],[L|T]) :-
  L =< R,
  merge(LS,[R|RS],T).

merge([L|LS],[R|RS],[R|T]) :-
  L > R,
  merge([L|LS],RS,T).
```

Prolog

#4

Αντικειμενοστρεφής προγραμματισμός
Object-oriented programming

Αντικειμενοστρεφής προγραμματισμός

Simula, Smalltalk, Eiffel,
C++, Java, C#, ...

- **Αντικείμενα** που επικοινωνούν μεταξύ τους ανταλλάσσοντας **“μηνύματα”**
- Κλάσεις, αφαίρεση δεδομένων, κληρονομικότητα, πολυμορφισμός
- Περισσότερα με τη C++ στο 2ο εξάμηνο

Περισσότερα στα μαθήματα της ροής Λ:

- **Γλώσσες Προγραμματισμού Ι** (6ο εξάμηνο)
- **Γλώσσες Προγραμματισμού ΙΙ** (9ο εξάμηνο)