Κεφάλαιο 10 : Συναρτησιακές Γλώσσες

Αρχές Γλωσσών Προγραμματισμού και Μεταφραστών

Ιστορικά Στοιχεία

- Οι μαθηματικοί διατύπωσαν τη διάκριση μεταξύ
  - μιας ***κατασκευαστικής* απόδειξης** (που δείχνει πώς λαμβάνεται ένα μαθηματικό αντικείμενο με κάποια επιθυμητή ιδιότητα)
  - μιας ***μη κατασκευαστικής* απόδειξης** (που απλώς δείχνει ότι ένα τέτοιο αντικείμενο πρέπει να υπάρχει, π.χ. με **απαγωγή σε άτοπο**)

- **Ο λογικός προγραμματισμός (λογικό, λογισμικό Η/Υ)** είναι στενά συνδεδεμένος με την έννοια της **κατασκευαστικής απόδειξης**, αλλά σε ένα πιο **αφηρημένο** επίπεδο:
  - ο προγραμματιστής στο λογικό μοντέλο προγραμματισμού, γράφει ένα **σύνολο *αξιωμάτων* (συντακτικο)** που επιτρέπει στον *υπολογιστή* να **ανακαλύψει μια κατασκευαστική απόδειξη** για κάθε συγκεκριμένο **σύνολο εισόδων**

Εισαγωγή

- **Οι διαφορές των ΓΠ στις *συντακτικές δομές* τους, είναι πολύ μεγαλύτερες** από τις διαφορές τους στις εννοιολογικές δομές. Π.χ. το στοιχείο του πίνακα Α στη θέση 1, γράφεται:

  A(1) στις FORTRAN, COBOL, PL/1, Ada

  A[1] στις Pascal, C

  A<1> στη SNOBOL

- Στόχος συντακτικού:

  Κανόνες επικοινωνίας της πληροφορίας μεταξύ προγραμματιστή και μεταφραστή/διερμηνέα.

Ιστορικά Στοιχεία

- Τα μοντέλα του προστακτικού και του συναρτησιακού προγραμματισμού προέκυψαν από το έργο των Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, κλπ. ~δεκαετία 1930
  - διαφορετικοί φορμαλισμοί για την έννοια του αλγορίθμου ή της *αποτελεσματικής διαδικασίας,* βασισμένοι στα αυτόματα, τη συμβολική επεξεργασία, τους αναδρομικούς ορισμούς συναρτήσεων, και τη συνδυαστική (python)
- Τα αποτελέσματα αυτά οδήγησαν τον Church στην εικασία ότι *οποιοδήποτε* διαισθητικά αποδεκτό μοντέλο υπολογισμών θα είναι και αυτό εξίσου ισχυρό
  - η εικασία αυτή είναι γνωστή ως *θέση του Church*

Ιστορικά Στοιχεία

- Το μοντέλο υπολογισμών του Turing ήταν η *μηχανή Turing*, ένα είδος αυτόματου στοίβας που χρησιμοποιούσε μια «ταινία» με απεριόριστο πλήθος αποθηκευτικών θέσεων
  - η μηχανή Turing κάνει υπολογισμούς με προστακτικό τρόπο, αλλάζοντας τις τιμές θέσεων στην ταινία της – όπως τα προστακτικά προγράμματα υψηλού επιπέδου εκτελούν υπολογισμούς αλλάζοντας τις τιμές των μεταβλητών

Ιστορικά Στοιχεία

- Το μοντέλο υπολογισμού του Church ονομάζεται *λογισμός λάμβδα*
  - βασίζεται στην έννοια των παραμετρικών εκφράσεων (κάθε παράμετρος εισάγεται με μια εμφάνιση του γράμματος λ – από όπου και το όνομα της σημειογραφίας)
  - ο λογισμός λάμβδα ήταν η έμπνευση για το συναρτησιακό προγραμματισμό
  - *ο υπολογισμός γίνεται με την αντικατάσταση παραμέτρων σε εκφράσεις, ακριβώς όπως σε ένα συναρτησιακό πρόγραμμα υψηλού επιπέδου ο υπολογισμός γίνεται με τη μεταβίβαση ορισμάτων σε συναρτήσεις*

ΠΑΡΑΔΕΙΓΜΑ....

lambda Function

Here we rewrite our function **raise_to_power** as a **lambda function**. **After the keyword lambda**, we specify the **names of the arguments;** then, we use a **colon followed by the expression** that specifies what we wish the function to return.

```
raise_to_power = lambda x, y: x ** y

raise_to_power(2, 3)
```

POWERED BY DATACAMP WORKSPACE

8

POWERED BY DATACAMP WORKSPACE

As mentioned, the lambda functions allow you to write functions in a **quick and dirty way**. (lambda x, y: x ** y)(2,3)

ΠΑΡΑΔΕΙΓΜΑ....

map() and lambda Function

The map function **takes two arguments**, **a function and a sequence such as a list** and **applies the function over all the elements of the sequence.**

We can pass lambda function to the map without even naming them, and in this case, we refer to them as **anonymous** functions.

In the following example, we use **map() on the lambda function**, which squares all elements of the list, and we store the result in **square_all**.

ΠΑΡΑΔΕΙΓΜΑ...

```
nums = [48, 6, 9, 21, 1]

square_all = map(lambda num: num ** 2, nums)

print(list(square_all))
```

```
[2304, 36, 81, 441, 1]
```

ΠΑΡΑΔΕΙΓΜΑ….

```
def echo_word(word1, echo): return  word1 * echo
echo_word('hey', 5)
```

heyheyheyheyhey

```
(lambda word1, echo: word1 * echo) ('hey', 5)
```

heyheyheyheyhey

Έννοιες Συναρτησιακού Προγραμματισμού

- Οι γλώσσες συναρτησιακού προγραμματισμού όπως η **Lisp**, η **Scheme**, η **FP**, η **ML**, η **Miranda**, και η **Haskell** καθώς και οι **συνδυαστικές SCALA** και **PYTHON** (**Imperative (procedural or object oriented) and Functional**) είναι μια προσπάθεια να πραγματωθεί ο λογισμός λάμβδα του Church σε μια πρακτική μορφή σαν γλώσσα προγραμματισμού
- η βασική ιδέα: τα πάντα γίνονται με τη **σύνθεση συναρτήσεων**
  - δεν υπάρχει μεταβλητή κατάσταση, όπως συμβαίνει στον προστακτικό προγραμματισμό (turing machine).
  - δεν υπάρχουν παρενέργειες (κάθε κλήση της ίδιας συνάρτησης στο ίδιο όρισμα, επιστρέφει το ίδιο αποτέλεσμα : **referential transparency – Unlike Java or C**)

Non - Referential Transparency Java

```
public static void main(String... args) {
    printFibs(10);
}

public static void printFibs(int limit) {
    Fibs fibs = new Fibs();
    for (int i = 0; i < limit; i++) {
        System.out.println(fibs.next());
    }
}

static class Fibs {
    private int previous = -1;
    private int last = 1;

    public Integer next() {
        last = previous + (previous = last);
        return previous + last;
    }
}
```

0,1,1,2,3,5,8,13,21,34,.....

Here, the method is designed to return a different value on each call.

Using such non referentially transparent methods requires a strong discipline in order not to share the mutable state involved in the computation.

Functional style avoids such methods in favor of referentially transparent versions
(Αυτο βοηθάει στην παραλληλοποίηση και στη θεμελίωση του parallel and distributed computing)

Έννοιες Συναρτησιακού Προγραμματισμού

- Απαραίτητες δυνατότητες, αρκετές από τις οποίες λείπουν από κάποιες προστακτικές γλώσσες
  - **συναρτήσεις πρώτης κατηγορίας και υψηλότερης τάξης**
  - **ισχυρός πολυμορφισμός**
  - Αν όλος ο κώδικας είναι γραμμένος χωρίς αναφορά συγκεκριμένων τύπων και ως εκ τούτου μπορεί να χρησιμοποιηθεί διαφανώς με οποιοδήποτε αριθμό νέων τύπων, ονομάζεται παραμετρικά πολυμορφικός. Ο John C. Reynolds (και αργότερα ο Jean-Yves Girard) ανέπτυξαν τυπικά την έννοια αυτή του πολυμορφισμού ως μια προέκταση του λάμδα λογισμού (που καλείται πολυμορφικός λάμδα λογισμός, ή Σύστημα F).
  - **ισχυρές λειτουργίες για λίστες**
  - **αναδρομή**
  - **δομημένα αποτελέσματα συναρτήσεων**

Έννοιες Συναρτησιακού Προγραμματισμού

- Πώς κάνουμε κάτι σε μια συναρτησιακή γλώσσα;
  - Η αναδρομή (ειδικά η αναδρομή ουράς) αντικαθιστά την επανάληψη
  - Γενικά, μπορείτε να έχετε το ίδιο αποτέλεσμα με μια σειρά αναθέσεων

```
x := 0      ...
x := expr1  ...
x := expr2  ...
```

  - με την `x=f3(f2(f1(0)))`, όπου κάθε f περιμένει την τιμή του x σαν παράμετρο, η f1 επιστρέφει expr1, η f2 επιστρέφει expr2, η f3 επιστρέφει expr3…….

Έννοιες Συναρτησιακού Προγραμματισμού

- Η αναδρομή αντικαθιστά με επιτυχία ακόμα και τους βρόχους

```
x := 0; i := 1; j := 100;
while i < j do
    x := x + i*j; i := i + 1;
          j := j - 1
end while
return x
```

γίνεται `f(0,1,100)`, όπου

```
f(x,i,j) == if i < j then
f (x+i*j, i+1, j-1) else x
```

Έννοιες Συναρτησιακού Προγραμματισμού

- Το να θεωρούμε όμως ότι η αναδρομή είναι μια απευθείας, μηχανική αντικατάσταση της επανάληψης είναι ο λάθος τρόπος να βλέπουμε τα πράγματα
  - Πρέπει να συνηθίσουμε να σκεπτόμαστε σε αναδρομικό στυλ
- Πιο σημαντική έννοια και από την αναδρομή είναι οι ***συναρτήσεις υψηλότερης τάξης***
  - Παίρνουν μια συνάρτηση ως παράμετρο, ή επιστρέφουν μια συνάρτηση ως αποτέλεσμα
  - Χρήσιμες στην κατασκευή πραγμάτων

Έννοιες Συναρτησιακού Προγραμματισμού

- Η Lisp (**Lis**t **p**rocessing) έχει επίσης τα εξής (δεν τα έχουν όλες οι συναρτησιακές γλώσσες)
  - ομοεικονική
  - αυτό-ορισμός (μπορεί να μην έχει όνομα, να μην έχει data types)
  - ανάγνωση-αποτίμηση-εκτύπωση
- Εκδόσεις της LISP
  - Αμιγής Lisp (η πρώτη Lisp)
  - Common Lisp
  - Scheme

Έννοιες Συναρτησιακού Προγραμματισμού

- Το γεγονός ότι ο κώδικας είναι απαράλλακτος από τα δεδομένα, δίνει στη Lisp μια χαρακτηριστική σύνταξη που αναγνωρίζεται εύκολα.
- Όλος ο κώδικας του προγράμματος γράφεται ως λίστες μέσα σε παρενθέσεις.
- Η κλήση μιας συνάρτησης γράφεται ως μια λίστα όπου το όνομα της συνάρτησης είναι πρώτο, και ακολουθούν τα ορίσματα.
- Για παράδειγμα, μια συνάρτηση `f` που παίρνει τρία ορίσματα μπορεί να καλεσθεί με `(f x y z)`.

Έννοιες Συναρτησιακού Προγραμματισμού

- **Η αμιγής Lisp είναι αμιγώς συναρτησιακή**· όλες οι άλλες Lisp έχουν προστακτικά χαρακτηριστικά
- Όλες οι πρώτες Lisp έχουν δυναμική εμβέλεια
  - Δεν γνωρίζουμε αν αυτό ήταν ηθελημένο ή συνέβη κατά λάθος
- Η Scheme και η Common Lisp έχουν στατικές εμβέλειες
  - Η Common Lisp επιτρέπει τη δυναμική εμβέλεια για ρητά δηλωμένες *ειδικές* συναρτήσεις
  - Η Common Lisp είναι τώρα Η πρότυπη Lisp
    - Πολύ μεγάλη· πολύπλοκη (η Ada του συναρτησιακού προγραμματισμού)

```c
// A C program to demonstrate static scoping.
#include<stdio.h>
int x = 10;
// Called by g()
int f()
{
   return x;
}
// g() has its own variable named as x and calls f()
int g()
{
   int x = 20;
   return f();
}
int main()
{
  printf("%d", g());
  printf("\n");
  return 0;
}
```

Output : 10

```c
// Language that uses dynamic scoping.
int x = 10;
// Called by g()
int f()
{
   return x;
}
// g() has its own variable named as x and calls f()
int g()
{
   int x = 20;
   return f();
}
main()
{
  printf(g());
}
```

Output : 20

Έννοιες Συναρτησιακού Προγραμματισμού

- Η Scheme είναι μια ιδιαίτερα κομψή Lisp
- Άλλες συναρτησιακές γλώσσες
  - ML
  - Miranda
  - Haskell
  - FP
  - PYTHON (Συνδυαστική)
  - SCALA (Συνδυαστική)
- Η Haskell είναι η κυρίαρχη γλώσσα στην **έρευνα** στο συναρτησιακό προγραμματισμό

Μια Ανασκόπηση/Σύνοψη της Scheme

- Όπως αναφέρθηκε, η Scheme είναι μια ιδιαίτερα κομψή Lisp
  - Ο διερμηνέας εκτελεί έναν βρόχο ανάγνωσης-αποτίμησης-εκτύπωσης
  - Ό,τι εισάγεται στο διερμηνέα αποτιμάται (αναδρομικά) μόνο μια φορά
  - Αν κάτι είναι μέσα σε παρενθέσεις, είναι κλήση συνάρτησης (εκτός αν είναι σε παράθεση)
  - Οι παρενθέσεις ΔΕΝ υπάρχουν μόνο για ομοαδοποίηση, όπως στις γλώσσες της οικογένειας της Algol
    - Η προσθήκη ενός επιπέδου παρενθέσεων αλλάζει τη σημασία (οι συναρτήσεις και οι τελεστές είναι σε prefix notation)

    `(+ 3 4)` $\Rightarrow$ `7`

    `((+ 3 4))` $\Rightarrow$ `error`

    (το βέλος '$\Rightarrow$' σημαίνει «αποτιμάται σε»)

Μια Ανασκόπηση/Σύνοψη της Scheme

- Scheme:
  - Λογικές τιμές #t και #f
  - Αριθμοί
  - Εκφράσεις λάμβδα (οι συναρτήσεις είναι σε prefix notation)
  - Παράθεση

```
(+ 3 4) ⇒ 7
(quote (+ 3 4)) ⇒ (+ 3 4)
'(+ 3 4) ⇒ (+ 3 4)
```

  - Μηχανισμοί για τη δημιουργία νέων εμβελειών

```
(let ((square (lambda (x) (* x x))) (plus +))
(sqrt (plus (square a) (square b))))
```

Μια Ανασκόπηση/Σύνοψη της Scheme

- Scheme:
  - Εκφράσεις συνθήκης

```
(if (< 2 3) 4 5) ⇒ 4
(cond
 ((< 3 2) 1)
 ((< 4 3) 2)
 (else 3)) ⇒ 3
```

  - Προστακτικές δυνατότητες
    - αναθέσεις
    - ακολουθιακή εκτέλεση (begin)
    - επανάληψη
    - είσοδος-έξοδος (read, display)

Μια Ανασκόπηση/Σύνοψη της Scheme

- Πρότυπες συναρτήσεις της Scheme (η λίστα δεν είναι πλήρης):
  - αριθμητικές
  - λογικοί τελεστές
  - ισοδυναμία
  - τελεστές λιστών
  - **symbol**?
  - number?
  - complex?
  - real?
  - rational?
  - integer?

Στη scheme ΔΕΝ ΥΠΑΡΧΕΙ ο κλασσικός τύποςenumerate, όπως π.χ. στη C

```
// The name of enumeration is "flag"
and the constant // are the values of
the flag. By default, the values // of
the constants are as follows: //
constant1 = 0, constant2 = 1,
constant3 = 2 and // so on. enum
flag{constant1, constant2, constant3,
..... };
```

Π.χ. enum { north, south, east, west }
Στη C, εάν τυπώσω το east θα εμαφανιστεί το 2.
Στη scheme ΔΕΝ ΙΣΧΥΕΙ ΑΥΤΟ!!!
Όλα είναι symbols, δηλαδή strings

Η Scheme π*ροσομοιώνει ΝΠΑ*

- Η περιγραφή του αυτομάτου είναι μια λίστα τριών πραγμάτων:
  - αρχική κατάσταση
  - συνάρτηση μετάβασης
  - το σύνολο των τελικών καταστάσεων
- Η συνάρτηση μετάβασης είναι μια λίστα από ζεύγη
  - το πρώτο στοιχείο κάθε ζεύγους είναι ένα ζεύγος, του οποίου το πρώτο στοιχείο είναι μια κατάσταση και το δεύτερο στοιχείο είναι το σύμβολο εισόδου
  - αν η τρέχουσα κατάσταση ταιριάζει με το επόμενο σύμβολο εισόδου, τότε το πεπερασμένο αυτόματο μεταβαίνει στην κατάσταση που αναφέρεται στο δεύτερο στοιχείο του ζεύγους

Επιστροφή στη Σειρά Αποτίμησης

- Εφαρμοστική σειρά
  - την έχετε συνηθίσει από τις προστακτικές γλώσσες
  - συνήθως γρηγορότερη (αποτιμά την παράμετρο προτού τη χρειαστεί)
- Κανονική σειρά
  - όπως η κλήση κατ' όνομα: δεν αποτιμά την παράμετρο μέχρι να τη χρειαστεί
  - μερικές φορές πιο γρήγορη
  - αν ο υπολογισμός μπορεί γενικά να τερματιστεί, η αποτίμηση θα τελειώσει (θεώρημα Church-Rosser)

Επιστροφή στη Σειρά Αποτίμησης

- Στη Scheme
  - **οι συναρτήσεις χρησιμοποιούν εφαρμοστική σειρά που ορίζεται με εκφράσεις λάμβδα**
  - **οι ειδικές μορφές συναρτήσεων (μακροεντολές) χρησιμοποιούν κανονική σειρά που ορίζεται με συντακτικούς κανόνες**
- Μια ***αυστηρή*** γλώσσα απαιτεί όλα τα ορίσματά της να είναι καλά ορισμένα, επομένως μπορεί να χρησιμοποιηθεί **εφαρμοστική σειρά**
- Μια ***μη αυστηρή*** **γλώσσα** δεν απαιτεί όλα τα ορίσματά της να είναι καλά ορισμένα· χρειάζεται **αποτίμηση κανονικής σειράς**

Συναρτήσεις Υψηλής Τάξης

- Συναρτήσεις υψηλότερης τάξης
  - Δέχονται ως όρισμα μια συνάρτηση, ή επιστρέφουν ως αποτέλεσμα μια συνάρτηση
  - Πολύ χρήσιμες για την δημιουργία πραγμάτων
  - Πέρασμα παραμέτρων με Currying (το όνομα προέρχεται από τον Haskell Curry, από τον ίδιο προέρχεται και το όνομα της Haskell)
    - Για λεπτομέρειες, δείτε το λογισμό λάμβδα σε επόμενες διαφάνειες
    - Η ML, η Miranda, και η Haskell διευκολύνουν ιδιαίτερα τον ορισμό συναρτήσεων με currying

Θεωρητικές Βάσεις : Λογισμός - λ

- Λογισμός λάμβδα:
  - Μια σημειογραφία/μοντέλο υπολογισμών που βασίζεται στον αμιγή συντακτικό χειρισμό συμβόλων, τα πάντα είναι συναρτήσεις
  - Αναπτύχθηκε από τον Alonzo Church στη δεκαετία του 1930 σαν μοντέλο υπολογισιμότητας
  - Ο Church άνηκε σε ένα πλήθος ανθρώπων μεταξύ των οποίων συγκαταλέγονται οι Chomsky, Turing, Kleene, και Rosser

Θεωρητικές Βάσεις : Λογισμός - λ

Ο λ-λογισμός είναι μία τυπική γλώσσα Λ, η σύνταξη της οποίας δίνεται από τον ακόλουθο επαγωγικό ορισμό:

Έστω V ένα αριθμήσιμο σύνολο μεταβλητών. Το σύνολο Λ των όρων του λ-λογισμού είναι το μικρότερο σύνολο, που ικανοποιεί τις παρακάτω ιδιότητες:

$$x \in V \Rightarrow x \in \Lambda$$

$$M, N \in \Lambda \Rightarrow (M\ N) \in \Lambda$$

$$x \in V,\ M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda$$

Θεωρητικές Βάσεις : Λογισμός - λ

Τα στοιχεία του συνόλου Λ ονομάζονται επίσης λ-όροι (λ-terms). Υπάρχουν τριών ειδών:

- Μεταβλητές (Variables), δηλαδή στοιχεία του συνόλου V
- Εφαρμογές (Applications), με μορφή (M N), όπου M και N είναι λ-όροι
- Αφαιρέσεις (Abstractions), με μορφή (λx.M), όπου x μεταβλητή και M λ-όρος

Θεωρητικές Βάσεις : Λογισμός - λ

- Κατά σύμβαση χρησιμοποιούνται μικρά γράμματα του λατινικού αλφαβήτου (x,y,z κλπ) για συμβολισμό μεταβλητών και κεφαλαία (M,N,F,G, P κλπ) για λ-όρους.

Θεωρητικές Βάσεις : Λογισμός - λ

- Χρησιμοποιώντας αφηρημένη σύνταξη BNF και θεωρώντας ότι η συντακτική κλάση των μεταβλητών παριστάνεται με το μη-τερματικό σύμβολο (var), η γλώσσα Λ των λ-όρων περιγράφεται ισοδύναμα

⟨term⟩ ::= ⟨var⟩

| (⟨term⟩ ⟨term⟩)

| (λ ⟨var⟩·⟨term⟩)

Θεωρητικές Βάσεις : Λογισμός - λ

- Μπορούμε να ορίσουμε πράγματα όπως οι ακέραιοι με βάση μια διακεκριμένη συνάρτηση (όπως η ταυτοτική) που να αναπαριστά το μηδέν, και μια συνάρτηση του «επόμενου» που μας δίνει όλους τους υπόλοιπους αριθμούς
- Γίνεται εύκολο να οριστούν οι αριθμητικοί τελεστές στη σημειογραφία
  - Στην πράξη αυτό είναι κουραστικό
    - **θα υποθέσουμε την ύπαρξη της αριθμητικής και διακεκριμένων *σταθερών συναρτήσεων* για τους αριθμούς**

(Αριθμοειδή του Church – Church numerals). Για κάθε φυσικό αριθμό $n \in \mathbb{N}$ ορίζεται ένας όρος $c_n \in \Lambda$ ως: $c_n \equiv \lambda f.\ \lambda x.\ f^n(x)$

Το αριθμοειδές, που αντιστοιχεί στον αριθμό 0 είναι το $c_0 \equiv \lambda f.\ \lambda x.\ x$, στον αριθμό 1 το $c_1 \equiv \lambda f.\ \lambda x.\ f\ x$, στον αριθμό 2 το $c_2 \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$ κοκ.

Θεωρητικές Βάσεις : Λογισμός - λ

- Παράδειγμα εκφράσεων λάμβδα

```
id          λx.x
const       λx.2
plus        λx.λy.x + y
square      λx.x * x
hypot       λx. λy.sqrt
(plus (square x) (square y))
```

Θεωρητικές Βάσεις : Λογισμός - λ

- Συνήθως η εφαρμογή προσεταιρίζει από αριστερά προς τα δεξιά, επομένως η `f A B` είναι `(f A) B`, και όχι `f (A B)`
- Επίσης, η εφαρμογή έχει υψηλότερη προτεραιότητα από την αφαίρεση, επομένως η λ`x.A B` είναι λ`x.(A B)`, και όχι `(`λ`x.A) B` – π.χ., ML
- Οι παρενθέσεις χρησιμοποιούνται για σαφήνεια, ή για την παράβαση των κανόνων

Θεωρητικές Βάσεις : Λογισμός - λ

- Αυτοί οι κανόνες σημαίνουν ότι η εμβέλεια της τελείας φτάνει δεξιά μέχρι την πρώτη δεξιά παρένθεση που δεν της αντιστοιχεί αριστερή παρένθεση, ή μέχρι το τέλος της έκφρασης αν δεν υπάρχει τέτοια παρένθεση
  - Στην (λx. λy. λz.e) a b c, η αρχική συνάρτηση δέχεται μια παράμετρο και επιστρέφει μια συνάρτηση (μιας παραμέτρου) που επιστρέφει μια συνάρτηση (μιας παραμέτρου)
  - *Για την αναγωγή της έκφρασης, αντικαθιστάτε την a σε κάθε x στη λy. λz.e, στη συνέχεια αντικαθιστάτε την b σε κάθε y στην υπόλοιπη έκφραση, και την c σε κάθε z σε ό,τι μένει στο τέλος*

Θεωρητικές Βάσεις : Λογισμός - λ

- Παράδειγμα:

```
(λx. λy.x + y) 3 4
λy.(3 + y) 4
(3 + 4)
7
```

- Ελεύθερες και δεσμευμένες μεταβλητές: μια μεταβλητή είναι δεσμευμένη αν εισάγεται από ένα λάμβδα
  - Για παράδειγμα, στη λx. λy.(* x y) έχουμε δυο ένθετες εκφράσεις λάμβδα
    - η x είναι ελεύθερη στην εσωτερική (λy.(* x y)), αλλά δεσμευμένη στην εξωτερική

Θεωρητικές Βάσεις : Λογισμός - λ

- Η αποτίμηση των εκφράσεων λάμβδα γίνεται μέσω
  - (1) αντικατάστασης των παραμέτρων (*βήτα αναγωγή*)
    `(λx.times x x) y  =>  times y y`
  - (2) μετονομασίας των μεταβλητών (*άλφα μετατροπή*) (συχνά για να αποφύγουμε τις συγκρούσεις ονομάτων)
    `(λx.times x x) y ==(λz.times z z) y`
  - (3) απλοποίηση «εκτός σειράς» (*ήτα αναγωγή*)
    `(λx.f x)  =>  f`
    - Ο τελευταίος κανόνας είναι δυσνόητος· ΔΕΝ είναι ο ίδιος με τη βήτα αναγωγή

Απόψεις περί Συναρτ. Προγραμματισμού

- Πλεονεκτήματα των συναρτησιακών γλωσσών
  - η έλλειψη ρητής σειράς αποτίμησης (σε μερικές γλώσσες) προσφέρει την πιθανότητα παράλληλης αποτίμησης (π.χ. MultiLisp)
  - η έλλειψη παρενεργειών (referential transparency) και ρητής σειράς αποτίμησης απλοποιούν κάποια πράγματα για το μεταγλωττιστή
  - τα προγράμματα συχνά είναι εκπληκτικά μικρά
  - η γλώσσα μπορεί να είναι εξαιρετικά μικρή αλλά ισχυρή

Απόψεις περί Συναρτ. Προγραμματισμού

- Προβλήματα – Μειονεκτήματα:
  - πολλές αντιγραφές δεδομένων μέσω παραμέτρων
  - (φαινομενική) ανάγκη να δημιουργείται ένας νέος πίνακας όταν πρέπει να αλλάξει ένα στοιχείο του
  - συχνή χρήση δεικτών (προβλήματα χώρου/χρόνου και τοπικότητας)
  - συχνές κλήσεις διαδικασιών
  - η αναδρομή χρησιμοποιεί σημαντικό χώρο
  - απαιτεί συλλογή σκουπιδιών
  - απαιτεί ένα διαφορετικό τρόπο σκέψης από τον προγραμματιστή
  - δύσκολο να ενσωματωθεί η είσοδος-έξοδος στο αμιγώς συναρτησιακό μοντέλο