Εισαγωγή στο Συναρτησιακό Προγραμματισμό

Γιάννης Κασσιός

Σε αυτό το μάθημα θα εξερευνήσουμε ένα σπουδαίο μοντέλο προγραμματισμού, το *συναρτησιακό προγραμματισμό*. Θα δούμε το συναρτησιακό προγραμματισμό εν δράσει χρησιμοποιώντας μία από τις πιο δημοφιλείς συναρτησιακές γλώσσες, τη Haskell. Κατόπιν, θα εξετάσουμε το λ-λογισμό, τη μαθηματική θεωρία πίσω από το συναρτησιακό προγραμματισμό. Θα καταλήξουμε παρουσιάζοντας μερικά θέματα υλοποίησης συναρτησιακών γλωσσών.

Εδώ κάνουμε μια μικρή επισκόπηση στο τι είναι ο συναρτησιακός προγραμματισμός και ποια είναι η συνεισφορά του μοντέλου αυτού στον προγραμματισμό γενικότερα. Για μια πιο λεπτομερή ανάλυση, δείτε το άρθρο [Hug89].

1 Συναρτησιακός Προγραμματισμός

Ως είθισται, η παρουσίαση μας του συναρτησιακού μοντέλου προγραμματισμού γίνεται σε αντιπαραβολή με το μακράν επικρατέστερο μοντέλο του *προστακτικού προγραμματισμού*, με το οποίο οι περισσότεροι προγραμματιστές είναι επαρκώς εξοικειωμένοι.

Στο οικείο μοντέλο του προστακτικού προγραμματισμού, ένας υπολογισμός περιγράφεται από μια αλληλουχία *εντολών* προς τον υπολογιστή, οι οποίες έχουν άμεση επίδραση στην *κατάσταση* του προγράμματος. Οι λεγόμενες "*παρενέργειες*" των εντολών αυτών, δηλαδή οι αλλαγές στην κατάσταση του προγράμματος, είναι κεντρικό στοιχείο αυτού του τύπου προγραμματισμού.

Το προστακτικό μοντέλο είναι πολύ κοντά σε αυτό που πράγματι συμβαίνει στον υπολογιστή μας όταν εκτελούμε ένα πρόγραμμα: διάφορες οδηγίες που επεξεργάζεται ο υπολογιστής έχουν ως αποτέλεσμα την αλλαγή των περιεχομένων της μνήμης του. Έτσι, το προστακτικό μοντέλο θεωρείται σχετικά "χαμηλού επιπέδου" προγραμματισμός. Η γλώσσα μηχανής, η πιο χαμηλού επιπέδου γλώσσα και η μόνη που καταλαβαίνει άμεσα ο υπολογιστής, είναι μια καθαρά προστακτική γλώσσα.

Ως γνωστόν, οι γλώσσες πολύ χαμηλού επιπέδου δεν είναι πρακτικές για τη συγγραφή μεγάλων και δύσκολων προγραμμάτων. Οι σχεδιαστές γλωσσών προγραμματισμού αντιμετώπισαν αυτό το πρόβλημα με ποικίλους τρόπους. Σε μερικές περιπτώσεις παρέμειναν στο προστακτικό προγραμματισμό, αλλά εφηύραν αφαιρετικές έννοιες πιο "υψηλού επιπέδου", δηλαδή έννοιες που είναι πλησιέστερες στον άνθρωπο/προγραμματιστή και πιο μακρυά από τον υπολογιστή. Η ιδέα της *μεταβλητής*, μιας θέσης μνήμης στην οποία ο προγραμματιστής μπορεί να δώσει

όνομα, είναι το πιο χαρακτηριστικό και το πιο θεμελειώδες παράδειγμα μιας τέτοιας αφαίρεσης.

Ας δούμε λοιπόν πώς περιγράφεται ένας απλός υπολογισμός, π.χ. ο υπολογισμός του παραγοντικού ενός ακεραίου `n`, σε μία προστακτική γλώσσα σχετικά υψηλού επιπέδου όπως η Pascal:

```
factorial:=1;
while (n > 0) do begin
  factorial := factorial * n;
  n:=n-1
end
```

Η περιγραφή είναι όντως μια αλληλουχία εντολών που βασίζεται στις παρενέργειες:

- Αρχικοποίησε τη μεταβλητή `factorial` σε 1
- Όσο το `n` είναι μεγαλύτερο του 0 εκτέλεσε τα παρακάτω:
  - Πολλαπλασίασε το `factorial` με το `n`
  - Μείωσε την τιμή του `n` κατά 1

Στην περίπτωση του συναρτησιακού προγραμματισμού, το παραδοσιακό μοντέλο των εντολών και παρενεργειών εγκαταλείπεται χάριν μίας καθαρότερης και υψηλότερου επιπέδου αφαίρεσης, αυτήν της μαθηματικής *συνάρτησης*. Ο υπολογισμός πλέον περιγράφεται σα μία συνάρτηση που συσχετίζει τις παραμέτρους της (είσοδος) με το αποτέλεσμά της (έξοδος). Η έννοια της κατάστασης προγράμματος και της παρενέργειας δεν υπάρχει σε αυτό το μοντέλο υπολογισμού. Ας δούμε πώς περιγράφεται ο υπολογισμός του παραγοντικού στο συναρτησιακό μοντέλο (και συγκεκριμένα σε Haskell):

```
factorial :: Int -> Int
factorial n
  | n==0  = 1
  | n>0   = n*factorial(n-1)
```

Η δήλωση αυτή περιγράφει τον υπολογισμό ως εξής: η συνάρτηση `factorial` επιστρέφει

- 1 αν η παράμετρός της `n` είναι 0
- `n*factorial(n-1)` αν η παράμετρός της `n` είναι μεγαλύτερη από το 0

Δίνουμε έμφαση στο γεγονός ότι οι συνάρτησεις στο μοντέλο μας είναι *μαθηματικές*. Το αποτέλεσμα που επιστρέφουν εξαρτάται αποκλειστικά και μόνο από την είσοδο που τους δίνουμε. Δύο κλήσεις της ίδιας συνάρτησης με τις ίδιες παραμέτρους θα παράγουν αναγκαστικά το ίδιο αποτέλεσμα. Αυτό λέγεται σε αντιπαραβολή με τις "συναρτήσεις" προστακτικών γλωσσών προγραμματισμού όπως η C. Ο όρος "συνάρτηση" σε αυτές τις γλώσσες είναι καταχρηστικός, καθώς, λόγω παρενεργειών, μία τέτοια "συνάρτηση" μπορεί να επιστρέφει διαφορετικό αποτέλεσμα

κάθε φορά που καλείται, ακόμα και αν οι παράμετροι είναι ίδιες. Στον παρακάτω προστακτικό κώδικα Pascal, η "συνάρτηση" p επιστρέφει διαφορετική τιμή κάθε φορά που καλείται:

```
var somevar:integer;
function p():integer;
begin
  somevar:=somevar+1;
  p:=somevar
end
```

2 Το Κίνητρο για τη Χρήση Συναρτήσεων

Ας δούμε τώρα τους βασικούς λόγους για τους οποίους το συναρτησιακό μοντέλο απέρριψε τις έννοιες της κατάστασης και της παρενέργειας για αυτό το υψηλότερου επιπέδου μοντέλο υπολογισμού, τη συνάρτηση. Το διαρκές ζητούμενο στον προγραμματισμό είναι η παραγωγή *ορθού* και *εύκολα συντηρήσιμου* κώδικα. Για την επίτευξη αυτού των στόχων, είναι σημαντικό ο κώδικας να είναι

- Καλογραμμένος και ευανάγνωστος.
- Εύκολος στη χρήση από μαθηματικές θεωρίες ορθότητας.
- Διαχωρισμένος σε όσο το δυνατόν πιο ανεξάρτητα μεταξύ τους κομμάτια, που μπορούν να δημιουργηθούν, να ελεγχθούν και να αποσφαλματωθούν ανεξάρτητα το ένα από το άλλο. Η πολή σημαντική αυτή ιδέα ονομάζεται *τμηματοποίηση (modularization)* του κώδικα.

Ο συναρτησιακός προγραμματισμός, χάρη στην απλότητα του μαθηματικού του μοντέλου, επιτυγχάνει τους δύο πρώτους στόχους αρκετά καλύτερα από τον προστακτικό προγραμματισμό. Όσον αφορά τον τρίτο στόχο, στο συναρτησιακό προγραμματισμό έχουν αναπτυχθεί ιδιώματα που προσφέρουν δυνατότητες τμηματοποίησης που είναι ανέφικτες με τους κλασσικούς μηχανισμούς του προστακτικού προγραμματισμού (τμήματα (modules), πακέτα (packages), κλάσεις (classes) κτλ.). Τέτοια ιδιώματα είναι οι *συναρτήσεις ως τιμές πρώτης κατηγορίας (functions as first-class values)* (και κατά συνέπεια οι *συναρτήσεις υψηλότερης τάξης (higher order functions)*) και η *οκνηρή αποτίμηση (lazy evaluation)*, που θα εξετάσουμε λεπτομερώς στη συνέχεια.

Εφόσον οι στόχοι της αναγνωσιμότητας, της ευχρηστίας και της τμηματοποίησης του κώδικα αφορούν όλους τους προγραμματιστές, τα ιδιώματα του συναρτησιακού προγραμματισμού αποκτούν ενδιαφέρουν και για αυτούς που προτιμούν τις παραδοσιακές προστακτικές γλώσσες. Εξάλλου, πολλές από τις ιδέες του συναρτησιακού προγραμματισμού μπορούν να μεταφερθούν και να χρησιμοποιηθούν σε προστακτικές γλώσσες.

Στα παρακάτω, τα παραδείγματα που χρησιμοποιούνται είναι σκοπίμως πολύ απλά, ώστε να γίνουν ευκολότερα κατανοητά από τον αρχάριο αναγνώστη. Αυτό δε θα πρέπει όμως να δημιουργήσει την εντύπωση ότι η χρήση του συναρτησιακού προγραμματισμού περιορίζεται σε τόσο απλοϊκά και μικρά προγράμματα.

2.1 Καθαρότητα Κώδικα / Αποδείξεις Ορθότητας

Η χρήση μαθηματικά απλών αντικειμένων όπως οι συναρτήσεις έχει ως αποτέλεσμα να διευκολύνεται η χρήση μαθηματικών για την απόδειξη της ορθότητας ενός προγράμματος. Αντίθετα, παρουσία παρενεργειών, ακόμα και θεμελειώδεις αρχές των μαθηματικών δε μπορούν να χρησιμοποιηθούν. Για παράδειγμα, ο παρακάτω κώδικας Pascal

```
if f(1)=f(1) then print("Yes") else print("No")
```

δε μπορεί να απλοποιηθεί σε

```
print("Yes")
```

Η θεμελειώδης *ανακλαστική* μαθηματική ιδιότητα $x = x$ της ισότητας δεν ισχύει εδώ, αφού η "συνάρτηση" `f` θα μπορούσε να επιστρέψει διαφορετική τιμή στις δύο κλήσεις της. Δεν είναι τυχαίο ότι οι θεωρίες ορθότητας των προστακτικών γλωσσών κατά κανόνα διαχωρίζουν τις εκφράσεις (των οποίων η αποτίμηση δεν έχει παρενέργειες) από τις εντολές, ώστε να μπορέσουν να χρησιμοποιήσουν στοιχειώδη μαθηματικά. Τέτοια προβλήματα δεν υπάρχουν στο συναρτησιακό προγραμματισμό.

Όσον αφορά την καθαρότητα και την αναγνωσιμότητα του κώδικα, είναι φανερό ότι η απλότητα του παραπάνω προγράμματος είναι απατηλή. Ο αναγνώστης του προγράμματος μπορεί να εξαπατηθεί χρησιμοποιώντας την αυτονόητη μαθηματική παραδοχή $x = x$. Αυτή η παγίδα μπορεί να οδηγήσει σε λάθος τροποποιήσεις του προγράμματος ή σε σφάλματα που είναι δύσκολο να εντοπιστούν.

2.2 Συναρτήσεις Ανώτερης Τάξης

Ένα σημαντικό ιδίωμα που εισήγαγε ο συναρτησιακός προγραμματισμός είναι η χρήση των συναρτήσεων ως *τιμές πρώτης κατηγορίας (first-class values)*. Αυτό σημαίνει ότι μία συνάρτηση μπορεί να χρησιμοποιηθεί όπως μία οποιαδήποτε βαθμωτή τιμή. Μπορεί να περάσει σαν παράμετρος σε μια άλλη συνάρτηση και μπορεί να είναι η τιμή επιστροφής μίας άλλης συνάρτησης. Αυτό μας οδηγεί και στην έννοια των *συναρτήσεων υψηλότερης τάξης (higher-order functions)*. Μία συνάρτηση ανώτερης τάξης είναι μία συνάρτηση που παίρνει ως παράμετρο μία άλλη συνάρτηση.

Ας δούμε πώς το ιδίωμα αυτό μας βοηθάει να τμηματοποιούμε τις προγραμματιστικές λύσεις μας με καινούριους και ενδιαφέροντες τρόπους που δε συναντάμε στο δομημένο και αντικειμενοστρεφή προστακτικό προγραμματισμό. Τα παραδείγματα που θα χρησιμοποιήσουμε προέρχονται από το άρθρο [Hug89].

Έστω ότι έχουμε μία λίστα ακέραιων αριθμών των οποίων θέλουμε να βρούμε το άθροισμα. Στη Haskell γράφουμε:

```
sum::[Int]->Int
sum l
  | l==[]      = 0
  | otherwise  = head l + sum (tail l)
```

Τα παραπάνω σημαίνουν:

- Ορίζουμε τη `sum` ως μία συνάρτηση που παίρνει ως παράμετρο μία λίστα ακεραίων και επιστρέφει έναν ακέραιο.
- Η `sum` εφαρμοζόμενη στην κενή λίστα επιστρέφει 0.
- Η `sum` εφαρμοζόμενη σε μία μη κενή λίστα επιστρέφει το άθροισμα του πρώτου στοιχείου της λίστας με το αποτέλεσμα της εφαρμογής της `sum` στην υπόλοιπη λίστα.

Αυτός είναι ένας κλασσικός τρόπος ορισμού μιας αναδρομικής συνάρτησης σε λίστες. Αν προσέξουμε όμως, θα δούμε ότι ο τρόπος με τον οποίο υπολογίζουμε το άθροισμα είναι πανομοιότυπος με τον τρόπο με τον οποίο θα υπολογίζαμε πολλές άλλες συσσωρευτικές πράξεις πάνω στη λίστα, πχ. το γινόμενο όλων των αριθμών, το μέγιστο όλων των αριθμών κτλ. Θα μπορούσαμε να απομονώσουμε αυτό το μοτίβο, χρησιμοποιώντας μία συνάρτηση ανώτερης τάξης με όνομα `reduce`. Η `reduce` παίρνει ως παράμετρο μία αρχική τιμή `i` μία συνάρτηση `f` και μία λίστα `l` και παράγει το αποτέλεσμα που έχει η συσσωρευτική αποτίμηση της `f` στη λίστα `l`:

```
reduce::Int->(Int->Int->Int)->[Int]->Int
reduce i f l
  | l==[]      = i
  | otherwise  = f (head l) (reduce i f (tail l))
```

Η αναδρομή που βλέπουμε εδώ είναι ίδια με αυτήν που είδαμε στον ορισμό της `sum`, μόνο που εδώ έχουμε γενικεύσει: αντί για πρόσθεση έχουμε βάλει μία οποιαδήποτε συνάρτηση `f` που παίρνει δύο ακεραίους και επιστρέφει έναν ακέραιο και αντί για 0 έχουμε βάλει έναν οποιονδήποτε ακέραιο `i`.

Με αυτόν τον τρόπο, έχουμε ορίσει το συσσωρευτικό υπολογισμό οποιασδήποτε συνάρτησης και όχι μόνο της πρόσθεσης. Ο τρόπος να ορίσουμε τη συσσωρευτική πρόσθεση είναι απλά να δώσουμε στην παράμετρο `f` την τιμή `(+)` που είναι η συνάρτηση πρόσθεσης δύο ακεραίων, και στην παράμετρο `i` την τιμή 0:

```
sum::[Int]->Int
sum = reduce 0 (+)
```

Ο συσσωρευτικός πολλαπλασιασμός γίνεται επίσης με τον ίδιο τρόπο, χωρίς επιπλέον κόπο από την πλευρά του προγραμματιστή:

```
mult::[Int]->Int
mult = reduce 1 (*)
```

Βλέπουμε λοιπόν πώς οι συναρτήσεις υψηλής τάξης μας επιτρέπουν τρόπους διαχωρισμού του λογισμικού που δεν είναι δυνατοί στον απλό δομημένο προγραμματισμό.

Πολλές προστακτικές γλώσσες προγραμματισμού επιτρέπουν το ιδίωμα των συναρτήσεων ως πολίτες πρώτης κατηγορίας. Όσο όμως αυτές δεν είναι πραγματικές συναρτήσεις, αλλά διαδικασίες με παρενέργειες, το πρόβλημα της απόδειξης ορθότητας σε αυτές τις γλώσσες είναι εξαιρετικά δύσκολο.

Παρατηρήστε ακόμα ότι στο παράδειγμα αυτό, μπορούμε να χρησιμοποιήσουμε συναρτήσεις χωρίς να τους περάσουμε όλα τα ορίσματά τους. Στον ορισμό της `sum` και της `mult`, στη συνάρτηση `reduce` δίνουμε μόνο δύο ορίσματα. Αυτό που συμβαίνει στην προκειμένη περίπτωση είναι η επιστροφή μίας νέας συνάρτησης, η οποία περιμένει και το τρίτο όρισμα: τη λίστα πάνω στην οποία θα γίνει ο συσσωρευτικός υπολογισμός. Αυτή η ευελιξία είναι ακόμα μία συνέπεια της χρήσης των συναρτήσεων ως πολιτών πρώτης κατηγορίας. Σε παραδοσιακές προστακτικές γλώσσες, τέτοια χρήση δεν επιτρέπεται.

2.3 Οκνηρή Αποτίμηση

Η *οκνηρή αποτίμηση (lazy evaluation)* αποτελεί ένα άλλο χαρακτηριστικό ιδίωμα που ξεκίνησε από το συναρτησιακό προγραμματισμό. Στην οκνηρή αποτίμηση, οι τιμές που κατασκευάζονται από έναν ορισμό ή που περνάνε σε μία συνάρτηση δεν αποτιμώνται αμέσως. Μία τιμή υπολογίζεται μόνο όταν χρειαστεί από κάποιον υπολογισμό. Το αντίθετο της οκνηρής αποτίμησης, δηλαδή να αποτιμώνται τα ορίσματα αμέσως μόλις οριστούν ή περαστούν σαν παράμετροι σε μία συνάρτηση ονομάζεται *πρόθυμη αποτίμηση (eager evaluation)*.

Ας δούμε ένα πολύ απλό παράδειγμα. Στη Haskell, γλώσσα που υποστηρίζει οκνηρή αποτίμηση, μπορούμε να δημιουργήσουμε μία λίστα με άπειρα στοιχεία ως εξής:

```
listOfOnes::[Int]
listOfOnes = [1] ++ listOfOnes
```

Εδώ ο τελεστής `++` είναι ο τελεστής *συνένωσης (concatenation)* λιστών. Από τον αναδρομικό ορισμό βλέπουμε ότι η λιστα `listOfOnes` είναι μία μη πεπερασμένη δομή δεδομένων (περιέχει το στοιχείο 1 άπειρες φορές).

Σε μία γλώσσα με πρόθυμη αποτίμηση, αυτός ο ορισμός θα είχε ως συνέπεια να προσπαθήσει ο υπολογιστής να κατασκευάσει αμέσως αυτή τη δομή δεδομένων. Αφού αυτή η δομή είναι άπειρη, αυτό θα είχε ως αποτέλεσμα έναν υπολογισμό που δε θα τερματίσει ποτέ.

Στη Haskell αυτό δε συμβαίνει, καθώς η δομή `listOfOnes` δεν αποτιμάται αμέσως. Αντίθετα, κάθε στοιχείο της δομής θα αποτιμηθεί όταν ζητηθεί η αποτίμησή του. Για παράδειγμα, αν ζητήσουμε από τη Haskell να υπολογίσει το πρώτο στοιχείο της λίστας, τότε η λίστα θα αποτιμηθεί μέχρι το πρώτο της στοιχείο ώστε να δωθεί απάντηση στο ερώτημα. Η ερώτηση που θα κάνουμε στη Haskell είναι

```
head listOfOnes
```

και η απάντηση που θα πάρουμε είναι, φυσικά:

```
1
```

Ομοίως, αν ζητήσουμε το τρίτο στοιχείο, η λίστα θα αποτιμηθεί μέχρι το τρίτο στοιχείο: η ερώτηση είναι

```
head (tail (tail listOfOnes))
```

και η απάντηση είναι και πάλι

```
1
```

Η χρησιμότητα αυτού του σχήματος είναι ότι διαχωρίζει τη διαδικασία κατασκευής μίας πολύπλοκης και πιθανώς άπειρης δομής δεδομένων, από τη διαδικασία χρήσης αυτής της δομής σε ένα πρόγραμμα. Αυτό δεν είναι δυνατό με την πρόθυμη αποτίμηση, στην οποία είμαστε αναγκασμένοι να παράγουμε τη δομή μαζί με τον κώδικα που τη χρησιμοποιεί.

Αυτό το πλεονέκτημα μπορεί να μη φαίνεται πολύ σπουδαίο στο απλοϊκό παράδειγμά μας, αλλά αποκτά μεγάλη αξία σε πιο πολύπλοκα παραδείγματα. Στο πιο εμπεριστατωμένο παράδειγμα που δίνεται στο άρθρο [Hug89] έχουμε την περίπτωση ενός παιχνιδιού (όπως είναι η τρίλιζα ή το σκάκι). Σε αυτήν την περίπτωση, μπορούμε να διαχωρίσουμε την κατασκευή του δέντρου καταστάσεων του παιχνιδιού (μία δομή δεδομένων που είναι τεράστια) από το πρόγραμμα που χρησιμοποιεί αυτό το δέντρο (πιθανώς ένας αλγόριθμος τεχνητής νοημοσύνης που παίζει το παιχνίδι). Ο προγραμματιστής της δομής μπορεί να την κατασκευάσει αδιαφορώντας για το μέγεθός της, καθώς ξέρει ότι στην πραγματικότητα θα αποτιμηθούν μόνο όσα τμήματά της χρειάζονται. Ο προγραμματιστής του κυρίως προγράμματος μπορεί να ασχοληθεί με τον αλγοριθμό του, αδιαφορώντας για την κατασκευή της δομής, την οποία μπορεί να χειριστεί σαν να ήταν ήδη κατασκευασμένη.

3 Μερικές Επιπλέον Σημειώσεις

Πολλοί ερευνητές στο πεδίο του Συναρτησιακού Προγραμματισμού, καθώς και σχεδιαστές και χρήστες συναρτησιακών γλωσσών θεωρούν το συναρτησιακό μοντέλο ανώτερο από το προστακτικό και προτάσουν τη χρήση συναρτησιακών γλωσσών για κάθε περίπτωση. Υπάρχει και αντίλογος από την πλευρά του προστακτικού προγραμματισμού, όπου ομοίως πολλοί θεωρούν ότι ο προστακτικός προγραμματισμός είναι ανώτερος από το συναρτησιακό.

Στο μάθημα αυτό δε υιοθετούμε καμία από τις δύο απόψεις. Ο τρόπος που αντιλαμβάνεται κανείς τον υπολογισμό είναι σε μεγάλο βαθμό υποκειμενική υπόθεση. Αυτό που είναι εύχρηστο και κομψό για κάποιον μπορεί να θεωρηθεί δύσχρηστο και κακογραμμένο από κάποιον άλλον.

Ο σκοπός αυτού του μαθήματος είναι να παρουσιάσει το συναρτησιακό μοντέλο προγραμματισμού, καθώς και τις σημαντικές ιδέες που έχουν αναπτυχθεί γύρω από αυτό το μοντέλο. Δε βλέπουμε το συναρτησιακό μοντέλο ως κάτι που θα πρέπει να *αντικαταστήσει* τον προστακτικό προγραμματισμό, αλλά ως ένα σύνολο ιδεών που μπορούν να *συμπληρώσουν* το προστακτικό μοντέλο και να διορθώσουν τις ελλείψεις του. Οι ιδέες που θα παρουσιαστούν σε αυτό το μάθημα είναι χρήσιμες και για τους χρήστες προστακτικών γλωσσών. Παρ' όλα αυτά, το μάθημα θα κινηθεί καθαρά στα πλαίσια του συναρτησιακού προγραμματισμού για λόγους καθαρότητας της παρουσίασης.

4 Επισκόπηση

Σε αυτήν την ενότητα είδαμε τα εξής:

- Ο συναρτησιακός προγραμματισμός είναι ένα μοντέλο προγραμματισμού το οποίο χρησιμοποιεί τη *μαθηματική συνάρτηση* ως το μοναδικό εργαλείο για να περιγράψει τον υπολογισμό.
- Οι κύριοι λόγοι για τους οποίους αναπτύχθηκε ο συναρτησιακός προγραμματισμός είναι:
  - Η συγγραφή *καλογραμμένου* και *ευανάγνωστου* κώδικα.
  - Η χρήση "καθαρών" μαθηματικών δομών για την περιγραφή του υπολογισμού, ώστε να διευκολυνθούν οι *αποδείξεις ορθότητας* των προγραμμάτων.
  - Η δημιουργία εναλλακτικών τρόπων *τμηματοποίησης* του κώδικα.
- Δύο σημαντικά ιδιώματα που αναπτύχθηκαν στο συναρτησιακό προγραμματισμό είναι οι *συναρτήσεις ως πολίτες πρώτης κατηγορίας* και η *οκνηρή αποτίμηση*.
- Η χρήση των συναρτήσεων ως πολίτες πρώτης κατηγορίας βοηθάει στην τμηματοποίηση του κώδικα με τρόπους που δεν είναι δυνατοί στο δομημένο προγραμματισμό. Στο παράδειγμά μας, μπορέσαμε να διαχωρίσουμε τη δομή παραγωγής συσσωρευτικού αποτελέσματος σε μία λίστα από τη συνάρτηση που χρησιμοποιείται για να παράγει αυτό το συσσωρευτικό αποτέλεσμα.
- Η οκνηρή αποτίμηση βοηθάει στο διαχωρισμό του κώδικα *παραγωγής* πολύπλοκων δομών δεδομένων από τον κώδικα που *χρησιμοποιεί* τις δομές αυτές.
- Σε αυτό το μάθημα δε θεωρούμε ότι το συναρτησιακό μοντέλο είναι ανώτερο από το προστακτικό. Θεωρούμε ότι οι ιδέες του συναρτησιακού μοντέλου μπορούν να *συμπληρώσουν* τον προστακτικό προγραμματισμό.

Αναφορές

[Hug89] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98--107, 1989.