Συναρτησιακός προγραμματισμός : Η άλλη πλευρά του νομίσματος

Απόστολος Συρόπουλος
28ης Οκτωβρίου 366
671 00 ΞΑΝΘΗ
apostolo@obelix.ee.duth.gr

Οκτώβριος 1999

Δημοσιεύτηκε στο τεύχος Οκτωβρίου 1999 του περιοδικού RAM, σελ. 248–253

1 Εισαγωγή

Πριν από μερικά χρόνια συζητούσα με κάποιο καθηγητή μου το ζήτημα της μελλοντικής μου επαγγελματικής αποκατάστασης. Ένα πρόβλημα που με απασχολούσε ιδιαίτερα ήταν αυτό της γνώσης κάποιων γλωσσών προγραμματισμού. Σκεφτόμουν ότι πολύ πιθανά κάποια εταιρεία να μου ζητούσε να γνωρίζω κάποια γλώσσα την οποία όμως εγώ δεν θα γνώριζα, ενώ θα γνώριζα κάποια παραπλήσια. Ο Bror, έτσι λένε τον καθηγητή μου, μου είπε απλά στην περίπτωση που συμβεί αυτό να πω ότι γνωρίζω την γλώσσα και το ίδιο βράδυ να βρω ένα βιβλίο εκμάθησης της γλώσσας και να την μάθω! Μου φάνηκε υπερβολικό, αλλά όμως μετά από λίγο συνειδητοποίησα το πόσο δίκαιο είχε. Σκεφτήκατε ποτέ σε ποίο ουσιαστικό σημείο διαφέρει η C από την Pascal ή η Ada από τη Modula-2, την Java ή την C++? Ο πυρήνας όλων αυτών των γλωσσών προγραμματισμού είναι η εντολή ανάθεσης, δηλαδή η εντολή με την οποία αλλάζουμε ή δίνουμε τιμή (αξία) σε μία μεταβλητή. Σκεφτείτε το! Κάνοντας ένα βήμα παραπέρα: Έχει άραγε νόημα μια γλώσσα προγραμματισμού να μην διαθέτει την εντολή ανάθεσης?

Η απάντηση στο προηγούμενο ερώτημα, προς έκπληξη πολλών φαντάζομαι αναγνωστών, είναι : ΝΑΙ! Καταρχήν οι γλώσσες προγραμματισμού χωρίζονται σε δύο μεγάλες κατηγορίες : τις *προστακτικές* (imperative) και τις *δηλωτικές* (declarative). Έχοντας κανείς να λύσει ένα πρόβλημα, χρησιμοποιώντας τις πρώτες περιγράφει ακριβώς τι πρέπει να κάνει ο Η/Υ ώστε να λυθεί υπολογιστικά το πρόβλημά του. Αν όμως κανείς χρησιμοποιεί μια δηλωτική γλώσσα, απλά περιγράφει πως πρέπει να λυθεί το πρόβλημα του. Ας δούμε ένα παράδειγμα. Έστω ότι θέλουμε να γράψουμε ένα πρόγραμμα το οποίο θα υπολογίζει το άθροισμα των πρώτων n φυσικών αριθμών. Στην περίπτωση μιας προστακτικής γλώσσας, όπως η C++, το πρόγραμμά μας θα έχει την παρακάτω μορφή:

```
sum=0;
cin >> n;
for(i=0; i<=n; i++)
  sum += i;
cout << sum << '\n';
```

Δηλαδή χρησιμοποιώντας έναν αθροιστή και μία εντολή επανάληψης (και όχι εντολή ανακύκλωσης όπως κακώς αναφέρεται σε πολλά κείμενα), βρίσκουμε το τελικό αποτέλεσμα. Στην περίπτωση μιας δηλωτικής γλώσσας θα πρέπει να ανακαλύψουμε τις ιδιότητες του αθροίσματος και με βάση αυτές να γράψουμε το πρόγραμμά μας. Δηλαδή ότι το άθροισμα των n αριθμών είναι ίσο με n συν το άθροισμα των $n - 1$

αριθμών, ενώ το άθροισμα το των $n-1$ αριθμών με $n-1$ συν το άθροισμα των $n-2$ αριθμών, κ.ο.κ. Ιδού το αντίστοιχο πρόγραμμα σε μορφή ψευδό-κώδικα:

```
sum n = if n==0 then 0 else n + sum(n-1)
n = inputstream
show sum n
```

Παρατηρήστε ότι στις δηλωτικές γλώσσες «παίζουμε» με αναδρομικές δηλώσεις λειτουργιών (*functions* όπως αυτές της Pascal ή της C). Μια διαδικασία ή μια λειτουργία ονομάζεται αναδρομική όταν ορίζεται με βάση τον ίδιο της τον εαυτό, π.χ., το παραγοντικό ενός αριθμού n ορίζεται ως εξής:

$$n! = \begin{cases} 1, & \text{an } n = 1 \\ n \cdot (n-1)! & \text{αλλιώς} \end{cases}$$

Είναι φανερό ότι το παραγοντικό του n ορίζεται με βάση την τιμή του παραγοντικού του $n-1$, του $n-1$ με βάση αυτό του $n-2$, κ.ο.κ. Επιπλέον, υπάρχει μια τερματική συνθήκη με την οποία καθορίζουμε που θα σταματήσει ο υπολογισμός. Έτσι αν θέλουμε να υπολογίσουμε το 3! θα το κάνουμε ως εξής:

$$\begin{aligned} 3! &= 3 \cdot (3-1)! \\ &= 3 \cdot 2! \\ &= 3 \cdot 2 \cdot (2-1)! \\ &= 3 \cdot 2 \cdot 1! \\ &= 3 \cdot 2 \cdot 1 \\ &= 6. \end{aligned}$$

Προσέξτε την σημασία της τερματικής συνθήκης.

Οι δηλωτικές γλώσσες χωρίζονται σε δύο κατηγορίες : στις λογικές (logic) και στις συναρτησιακές (functional). Οι λογικές γλώσσες προγραμματισμού έχουν τα μαθηματικά τους θεμέλια στην μαθηματική θεωρία αποδείξεων (proof theory) και ουσιαστικά αποτελούν εργαλεία απόδειξης θεωρημάτων. Με άλλα λόγια, κάθε πρόγραμμα είναι ένα θεώρημα το οποίο ζητάμε να αποδείξει η υλοποίηση της γλώσσας. Η πιο γνωστές λογικές γλώσσες προγραμματισμού είναι η Prolog και οι διάφορες παραλλαγές της, όπως η λ-Prolog, κ.ά. Οι συναρτησιακές έχουν τα μαθηματικά τους θεμέλια στον λογισμό λ του Alonzo Church (βλέπε πλαίσιο για μια σύντομη περιγραφή του λογισμού λ). Σε μία συναρτησιακή γλώσσα προγραμματισμού κανείς ορίζει συναρτήσεις των οποίων η σύνθεση (βλέπε επόμενη ενότητα), συνήθως, αποτελεί τη λύση στο πρόβλημα μας. Οι πιο γνωστές συναρτησιακές γλώσσες προγραμματισμού είναι η Lisp[1] , η SML, η LML, η Miranda, η SISAL, η Haskell, κ.ά. Όλα σχεδόν τα κείμενα που πραγματεύονται γενικά τον συναρτησιακό προγραμματισμό, χρησιμοποιούν μια γλώσσα προγραμματισμού για να δείξουν τα ιδιαίτερα χαρακτηριστικά αυτής της προγραμματιστικής φιλοσοφίας. Εμείς εδώ διαλέξαμε την Haskell[2], ως γλώσσα αναφοράς μιας και αποτελεί την καλύτερη κατά τεκμήριο συναρτησιακή γλώσσα προγραμματισμού.

[1] Να σημειώσουμε ότι η Lisp δεν είναι συναρτησιακή γλώσσα όπως οι υπόλοιπες, αλλά έχει τα ίδια μαθηματικά θεμέλια με τις υπόλοιπες γλώσσες.

[2] Το όνομά της είναι το όνομα του μεγάλου λογικολόγου (logician) Haskell B. Curry.

(Αν δεν είστε εξοικειωμένος με την έννοια της συνάρτησης απλά διαβάστε την αρχή της δεύτερης ενότητας όπου γίνεται μια σύντομή περιγραφή της έννοιας.) Ο λογισμός λ συλλήφθηκε από τον Alonzo Church, ως μέρος μια γενικότερης θεωρίας των συναρτήσεων και της λογικής που προορίζονταν να γίνει θεμέλιο των μαθηματικών. Μολονότι, όπως αποδείχθηκε, το όλο σύστημα είχε αδυναμίες, το υποσύστημα που πραγματεύονταν μόνο τις συναρτήσεις έγινε ένα επιτυχημένο μοντέλο για τις «υπολογίσιμες συναρτήσεις», δηλαδή όλων εκείνων των συναρτήσεων τις οποίες μπορεί να υπολογίσει μια μηχανή Turing. Η αναπαράσταση μιας υπολογίσιμης συνάρτησης με έναν όρο του λογισμού λ, οδηγεί στον συναρτησιακό προγραμματισμό.
Ο λογισμός λ έχει δύο βασικές πράξεις. Η πρώτη είναι η *εφαρμογή*. Έτσι η παράσταση FA υποδηλώνει ότι τα δεδομένα F θεωρούμενα ως αλγόριθμος εφαρμόζονται στο A θεωρούμενο ως είσοδος. Είναι ενδιαφέρον να τονίσουμε ότι μπορούμε να έχουμε ως είσοδο τον ίδιο τον αλγόριθμο, πράγμα που συμβαίνει σε γλώσσες όπως η Lisp. Η άλλη βασική πράξη είναι η *αφαίρεση*. Αν γενικά το M είναι μια (αλγεβρική) έκφραση που περιέχει (ή εξαρτάται από) το x, τότε η παράσταση $\lambda x.$ δηλώνει μια συνάρτηση από το x στο πεδίο τιμών της παράστασης $[x]$. Για να γίνουμε πιο συγκεκριμένοι: έστω ότι $\ = 3x + 4$, τότε η αφαίρεση $\lambda x.3x + 4$ δηλώνει την συνάρτηση που αντιστοιχεί κάθε x στην τιμή $3x + 4$. Αν έχουμε μια αφαίρεση λάμδα, μπορούμε να την εφαρμόσουμε σε κάποια είσοδο αντικαθιστώντας κάθε εμφάνιση της μεταβλητής με την είσοδο. Για παράδειγμα η εφαρμογή $(\lambda x.3x+1/x)6$ έχει ως αποτέλεσμα την παράσταση $3\cdot 6+1/6$. Αν και μιλήσαμε για αφαιρέσεις λάμδα με μια εξαρτώμενη μεταβλητή, δεν υπάρχει κανένα πρόβλημα για αφαιρέσεις με πολλές μεταβλητές, π.χ., $\lambda x.\lambda y.\lambda z.x + 3y + 2z$, κ.λπ. (Σημειώστε ότι μόνο η τελευταία τελεία είναι υποχρεωτική.) Φυσικά υπάρχουν πάρα πολλά ακόμη να λεχθούν για τον λογισμό λ, όπως για παράδειγμα το γεγονός ότι σε κάθε μαθηματική λογική (κλασική, γραμμική, κ.λπ.) αντιστοιχεί ένας λογισμός λ με ιδιαίτερα χαρακτηριστικά, εντούτοις εμείς δε μπορούμε να επεκταθούμε περισσότερο. Ο αναγνώστης που ενδιαφέρεται να μάθει περισσότερα μπορεί να διαβάσει το κλασικό πλέον βιβλίο του H.P. Barendregt με τίτλο «*The lambda calculus: its syntax and semantics*», το οποίο εκδίδει ο οίκος North-Holland.

2 Συναρτήσεις και συναρτησιακός προγραμματισμός

Στα μαθηματικά μια συνάρτηση είναι ένας μηχανισμός που παρέχει μια αντιστοιχία από αντικείμενα ενός συνόλου, το πεδίο ορισμού, σε αντικείμενα ενός άλλου συνόλου, το πεδίο τιμών.

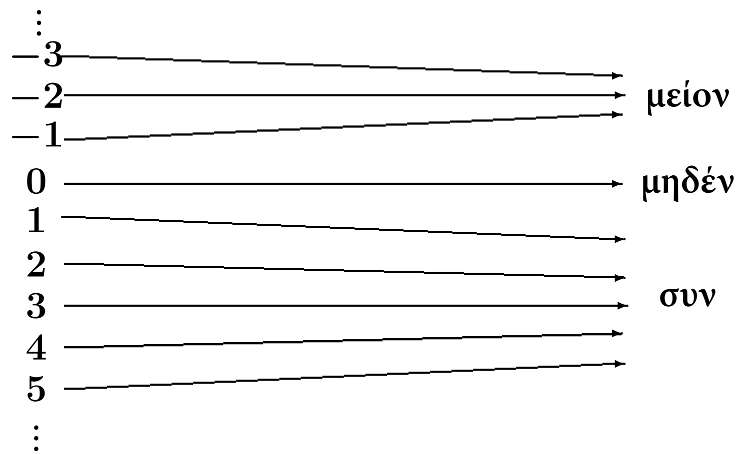
Ένα απλό παράδειγμα συνάρτησης είναι η συνάρτηση που αντιστοιχεί σε κάθε ακέραιο αριθμό μία από τις λέξεις συν, πλην ή μηδέν, ανάλογα του αν ο αριθμός είναι θετικός, αρνητικός ή μηδέν. Ονομάζουμε αυτή την συνάρτηση πρόσημο αντανακλώντας την αντιστοιχία που επιτελεί. Υπάρχουν τρεις τρόποι ορισμού μιας συνάρτησης:

1. μέσω| ενός διαγράμματος που δείχνει σε ποιο στοιχείο του πεδίου τιμών αντιστοιχεί το οποιοδήποτε στοιχείο του πεδίου ορισμού,
2. μέσω| μιας σειράς εξισώσεων όπου δίνουμε την τιμή της συνάρτησης για κάθε δυνατή τιμή, και
3. μέσω| της περιγραφής της αντιστοιχίας που επιτελεί η συνάρτηση.

Στο σχήμα 1 φαίνονται οι τρεις αυτές μέθοδοι ορισμού συνάρτησης για την περίπτωση της συνάρτησης πρόσημο. Ειδικά στην τρίτη περίπτωση το x ονομάζεται παράμετρος και αναπαριστά οποιοδήποτε στοιχείο του πεδίου ορισμού. Όταν το x πάρει μια συγκεκριμένη τιμή, αυτή ονομάζεται όρισμα της συνάρτησης, π.χ., στην παράσταση πρόσημο(4) το 4 είναι όρισμα της συνάρτησης. Η τρίτη μέθοδος είναι αυτή που συνήθως χρησιμοποιούμε στον συναρτησιακό προγραμματισμό, ενώ σε αρκετές περιπτώσεις χρησιμοποιούμε και την δεύτερη. Φυσικά μια συνάρτηση μπορεί να έχει παραπάνω από μία παραμέτρους. Μια συνάρτηση λέγεται καλώς ορισμένη όταν κάθε στοιχείο του πεδίου ορισμού αντιστοιχεί σε μόνο ένα στοιχείο του πεδίου τιμών.

Όπως αναφέρθηκε παραπάνω ένα βασικό συστατικό του συναρτησιακού προγραμματισμού είναι η δυνατότητα σύνθεσης συναρτήσεων. Τι σημαίνει όμως αυτό? Έστω ότι θέλουμε να υπολογίσουμε το μέγιστο δύο ακεραίων. Η παρακάτω συνάρτηση βρίσκει το μέγιστο δύο αριθμών:

α′



β′

$$\begin{array}{rcl} & \vdots & \\ \text{πρόσημο}(-3) & = & \text{μείον} \\ \text{πρόσημο}(-2) & = & \text{μείον} \\ \text{πρόσημο}(-1) & = & \text{μείον} \\ \text{πρόσημο}(0) & = & \text{μηδέν} \\ \text{πρόσημο}(1) & = & \text{συν} \\ \text{πρόσημο}(2) & = & \text{συν} \\ \text{πρόσημο}(3) & = & \text{συν} \\ & \vdots & \end{array}$$

γ′

$$\text{πρόσημο}(x) = \begin{cases} \text{μείον, αν } x < 0 \\ \text{μηδέν, αν } x = 0 \\ \text{συν, αν } x > 0 \end{cases}$$

Σχήμα 1: Μέθοδοι ορισμού συνάρτησης.

```
max x y = if x>=y then x else y
```

Το πεδίο ορισμού της συνάρτησης είναι εκείνο το σύνολο που αποτελείται από ζεύγη ακεραίων και το πεδίο τιμών το σύνολο των ακεραίων. (Βέβαια για μια πεπερασμένη μηχανή όπως ένας Η/Υ το σύνολο των ακεραίων δεν είναι παρά ένα πολύ μικρό υποσύνολό του.)‘

Αν θέλουμε να υπολογίσουμε το μέγιστο τριών αριθμών, θα πρέπει να ορίσουμε μια νέα συνάρτηση:

```
max3 x y z | x>=y && x>z || x>=z && x>y = x
           | y>=x && y>z || y>=z && y>x = y
           | z>=x && z>y || z>=y && z>x = z
           | otherwise                  = x
```

(Το σύμβολο `&&` σημαίνει *και*, το σύμβολο `||` *ή* και το σύμβολο `|` *αν*, δηλαδή η τρίτη γραμμή της δήλωσης σημαίνει ό,τι το μέγιστο των `x`, `y` και `z` είναι το `z` αν `z>=x && z>y || z>=y && z>x`. Τα άλλα σύμβολα, τελεστές σύγκρισης, είναι αυτονόητα.) Τώρα, ας προσπαθήσουμε να γράψουμε τον παραπάνω ορισμό με απλούστερο τρόπο. Ασφαλώς θα συμφωνείτε ότι ο παρακάτω ορισμός είναι σαφέστατα πιο κατανοητός και πιο απλός:

```
max3 x y x = max(x,max(y,z))
```

Με τον τρόπο αυτό κανείς μπορεί δημιουργεί νέες συναρτήσεις βασιζόμενος σε προηγούμενη δουλειά. (Ως άσκηση, προσπαθήστε να δημιουργήσετε μια συνάρτηση η οποία θα υπολογίζει το μέγιστο τεσσάρων ακεραίων αριθμών.) Μάλιστα, αυτός είναι και ο λόγος που κάθε τυπική εγκατάσταση της Haskell συνοδεύεται από ένα μεγάλο αριθμό προκαθορισμένων συναρτήσεων.

Ένα ακόμη βασικό χαρακτηριστικό κάθε συναρτησιακής γλώσσας προγραμματισμού είναι η έλλειψη *παρενεργειών*, η δυνατότητα χρήσης των οποίων αποτελεί βασικό χαρακτηριστικό κάθε προστακτικής γλώσσας προγραμματισμού. Όμως τι είναι οι παρενέργειες? Αν ορίσουμε μια συνάρτηση σε κάποια προστακτική γλώσσα προγραμματισμού, η οποία επιστρέφει κάποια τιμή αλλά και ταυτόχρονα αλλάζει την τιμή κάποιας καθολικής μεταβλητής, τότε λέμε ότι η συνάρτηση έχει παρενέργειες. Όσο αθώες και αν φαίνονται οι παρενέργειες, εντούτοις μπορούν εύκολα και απλά να παραβιάσουν βασικούς κανόνες των μαθηματικών και έτσι να έχουμε αναπάντεχα αποτελέσματα! Το παρακάτω πρόγραμμα σε C++ δείχνει ακριβώς το τι εννοούμε. Το πρόγραμμα απλά τυπώνει στην οθόνη του Η/Υ το άθροισμα $f(1) + f(2)$ και το άθροισμα $f(2) + f(1)$. Η f επιστρέφει είτε το όρισμα της, είτε το διπλάσιο του ορίσματος της, ανάλογα της τιμής της καθολικής μεταβλητής `flag`, της οποία όμως την τιμή μεταβάλει πριν επιστρέψει τον έλεγχο στο κύριο πρόγραμμα.

```
#include <iostream.h>

bool flag;

int f(int n)
{
    return (flag=!flag, (flag ? n : 2*n));
}

int main()
{
    flag=true;
    cout << f(1)+f(2) << '\n';
    cout << f(2)+f(1) << '\n';
}
```

Αν «τρέξετε» το πρόγραμμα αυτό θα διαπιστώσετε ότι παίρνουμε ως έξοδο τους αριθμούς 4 και 5, ενώ είναι γνωστό από τα μαθηματικά του γυμνασίου ότι $a + b = b + a$, πράγμα που εδώ προφανώς δεν ισχύει. Από την άλλη, δεν υπάρχει καμία πιθανότητα μια συναρτησιακή γλώσσα προγραμματισμού να παρουσιάσει μια ανακολουθία αυτής της μορφής. Αυτή η ιδιότητα των συναρτησιακών γλωσσών είναι γνωστή στη βιβλιογραφία ως referential transparency («αναφορική διαφάνεια»). Επίσης εντολές ανάθεσης που δημιουργούν παρενέργειες χαρακτηρίζονται ως καταστροφικές. Συμπερασματικά λέμε ό,τι αυτό που χαρακτηρίζει κάθε συναρτησιακή γλώσσα προγραμματισμού είναι ότι προγραμματίζουμε με συναρτήσεις τις οποίες μπορούμε να συνθέτουμε και οι οποίες δεν παραβιάζουν τα βασικά αξιώματα των μαθηματικών.

3 Σύντομη εισαγωγή στον συναρτησιακό προγραμματισμό με την Haskell

Χονδρικά μιλώντας, όταν λέμε «τύπο» σε μια γλώσσα προγραμματισμού εννοούμε ένα σύνολο τιμών και ορισμένες πράξεις που γίνονται με στοιχεία του συνόλου αυτού. (Οι αναγνώστες με κάποιο μαθηματικό υπόβαθρο, μπορούν χονδρικά να ταυτίσουν ένα τύπο με κάποια αλγεβρική δομή, π.χ., ομάδα, δακτύλιο, κ.λπ.) Κάθε συναρτησιακή γλώσσα προγραμματισμού είναι μια γλώσσα με ένα «ισχυρό» σύστημα τύπων. Αυτό σημαίνει ότι η υλοποίηση της γλώσσας ελέγχει πάντα αν κάποια έκφραση είναι σύμφωνη με τους κανόνες που διέπουν το σύστημα τύπων της. Για παράδειγμα δεν επιτρέπεται να προσθέτουμε ένα ακέραιο με ένα χαρακτήρα. Εκείνο όμως που κάνει ξεχωριστό το σύστημα τύπων κάθε συναρτησιακής γλώσσας σε σχέση με το αντίστοιχο κάποιας προστακτικής γλώσσας είναι το γεγονός ότι δεν απαιτείται να δηλώσουμε τον τύπο των συναρτήσεων και των διαφόρων παραστάσεων, καθώς αυτός εξάγεται από την υλοποίηση της γλώσσας με απόλυτα ασφαλή μέθοδο.

Οι βασικοί τύποι της Haskell είναι οι εξής:

- Οι τύποι `Int` και `Integer` που αναφέρονται σε ακεραίους με πεπερασμένο και απεριόριστο αριθμό ψηφίων αντίστοιχα.
- Οι τύποι `Float` και `Double` που αναφέρονται σε δεκαδικούς απλής και διπλής ακρίβειας αντίστοιχα.
- Ο τύπος `Bool` με τις δύο δυνατές τιμές: `True` και `False`.
- Ο τύπος `Char` που αναφέρεται σε χαρακτήρες.

Σημειώστε ότι όλα τα ονόματα τύπων γράφονται με κεφαλαίο το πρώτο τους γράμμα. Εκτός όμως από τους βασικούς τύπους, υπάρχουν και αρκετές μέθοδοι δημιουργίας νέων τύπων.

- Αν με το γράμμα `a` δηλώνουμε ένα τύπο, ο τύπος `[a]` δηλώνει μια ακολουθία από αντικείμενα τύπου `a`. Οι ακολουθίες διαφέρουν από τις παρατάξεις (arrays) στο γεγονός ότι δε μπορούμε να έχουμε πρόσβαση σε οποιοδήποτε στοιχείο τους, ακριβώς όπως για παράδειγμα συμβαίνει με μία ουρά: κανείς πάει πάντα στο τέλος και περιμένει την σειρά του. Μια ακολουθία αντικειμένων είναι μια δομή δεδομένων η οποία είναι είτε κενή, οπότε γράφεται `[]`, είτε μη-κενή και αποτελείται από ένα αντικείμενο `h`, την κεφαλή, και μία άλλη ακολουθία αντικειμένων `t`, την ουρά. Μια μη-κενή ακολουθία γράφεται `h:t`. Για παράδειγμα η ακολουθία `1:2:3:[]` αποτελείται από τους αριθμούς 1, 2 και 3, ενώ η άδεια ακολουθία μπαίνει απλά για να ορίσουμε το τέλος της ακολουθίας. Εναλλακτικά κανείς μπορεί να γράψει την προηγούμενη ακολουθία ως εξής: `[1,2,3]`. Τα κορδόνια, `String`, δεν είναι παρά `[Char]`, δηλαδή ακολουθίες χαρακτήρων τα οποία όμως επιτρέπεται να γράφονται και σε συμπαγή μορφή: η ακολουθία `['A',' ','S','t','r','i','n','g']` γράφεται `"A String"`.

- Αν τα a και b δηλώνουν κάποιους τύπους, ο τύπος (a,b) δηλώνει τον τύπο ζευγών που το πρώτο μέλος είναι τύπου a και το δεύτερο τύπου b. Φυσικά μπορούμε να δημιουργήσουμε τριάδες, τετράδες, κ.ο.κ., αλλά ζεύγη όπου ο ένας τύπος είναι μια τριάδα, κ.λπ. Γενικότερα οι νιάδες αντιστοιχούν σε records της Pascal ή σε structures της C.

- Αν τα a και b δηλώνουν κάποιους τύπους, ο τύπος a->b δηλώνει τον τύπο συναρτήσεων που έχουν ως πεδίο ορισμού τον τύπο a και ως πεδίο τιμών τον τύπο b. Για παράδειγμα η συνάρτηση strlen η οποία βρίσκει το μήκος ενός κορδονιού έχει τύπο [Char]->Int.

Αφού μάθαμε τους διάφορους δυνατούς τύπους που παρέχει η γλώσσα Haskell, ας δούμε πώς γράφουμε συναρτήσεις. Κάθε συνάρτηση έχει όνομα και χαρακτηρίζεται από τον αριθμό των παραμέτρων που δέχεται καθώς και το τι υπολογίζει, το οποίο είναι μια γενικευμένη αλγεβρική παράσταση. Την ονομάζουμε γενικευμένη απλά γιατί μπορεί να περιέχει παραστάσεις συνθήκης, απλές αλγεβρικές παραστάσεις, κ.ά. Για παράδειγμα οι συναρτήσεις

```
square y = y*y
factorial x = product [1..x]
sign n = if n>0 then "plus" else if n=0 then "zero" else "minus"
```

υπολογίζουν το τετράγωνο, το παραγοντικό ενός αριθμού και το πρόσημο ενός αριθμού. Η συνάρτηση factorial ορίζεται με βάση την συνάρτηση product η οποία υπολογίζει το γινόμενο όλων των μελών μιας ακολουθίας αριθμών. Προσέξτε ότι η ακολουθία [x..y] αποτελείται από όλους τους ακεραίους από το x ως και το y. Ενώ η ακολουθία [x,y..z] αποτελείται από τους ακεραίους από το x ως το z ανά y-x, δηλαδή η ακολουθία [1,3..10] είναι ισοδύναμη με την [1,3,5,7,9]. Ένα ιδιαίτερο χαρακτηριστικό της Haskell είναι ότι υπολογίζει τα ορίσματα κάθε συνάρτησης μόνο αν χρειάζεται και όσο αυτό χρειάζεται. Έτσι αν ορίσουμε την συνάρτηση

```
myfunc x y = x
```

και ζητήσουμε από κάποια υλοποίηση της γλώσσας να υπολογίσει την παράσταση myfunx 3 3/0 το αποτέλεσμα θα είναι 3, επειδή σύμφωνα με τον ορισμό το δεύτερο όρισμα απλά δεν λαμβάνεται υπ' όψη. Από την άλλη καμία προστακτική γλώσσα δεν θα μπορούσε να υπολογίσει την αντίστοιχη παράσταση. Αυτό το ιδιαίτερο χαρακτηριστικό πολλών συναρτησιακών γλωσσών προγραμματισμού είναι γνωστό στη διεθνή βιβλιογραφία ως lazy evaluation.

Εκτός όμως από τη δυνατότητα ορισμού ακολουθιών με απλή αναγραφή των μελών της μπορούμε να βάζουμε τη γλώσσα να υπολογίζει τα μέλη μιας ακολουθίας, ακριβώς όπως όταν ορίζουμε ένα σύνολο περιγραφικά, π.χ., η ακολουθία

```
 [ x | x <- [0..100], odd x ]
```

αποτελείται από τους περιττούς αριθμούς που βρίσκονται μεταξύ του 0 και του 100. Προσέξτε ότι ο ορισμός είναι σχεδόν ίδιος με τον αντίστοιχο μαθηματικό: $\{x|0 \leq x \leq 100 \text{ και } x \text{ περιττός}\}$. Αυτή ακριβώς η δυνατότητα χρησιμοποιείται από την Haskell για τον ορισμό παρατάξεων (arrays). Έτσι αν θέλουμε να ορίσουμε μια παράταξη, πρέπει να ορίσουμε τα άκρα της καθώς και τον τρόπο δημιουργίας των στοιχείων της:

```
    squares = array(1,100)[(i,i*i) | i <- [1..100]]
```

Βλέπουμε ότι ορίζουμε η παράταξη να έχει 100 στοιχεία με δείκτες από το 1 ως το 100. Το ζεύγος στις αγκύλες χρησιμεύει στο να δείξουμε σε πιο στοιχείο της παράταξης πρέπει να βάλουμε πια τιμή. Αν θέλουμε το έβδομο στοιχείο της παράταξης, απλά γράφουμε squares!7.

Στις περισσότερες προστακτικές γλώσσες προγραμματισμού, υπάρχει η δυνατότητα ορισμού επαγωγικών τύπων, όπως δένδρων, λιστών, σωρών, κ.λπ. Μόνο που εκεί συνήθως ένας ορισμός ενός επαγωγικού τύπου γίνεται με τη βοήθεια δεικτών. Οι συναρτησιακές γλώσσες δίνουν αυτή τη δυνατότητα, αλλά

επιπλέον το κάνουν με ένα πολύ απλό και σχεδόν φυσικό τρόπο. Έστω ότι θέλουμε να ορίσουμε έναν επαγωγικό τύπο που να αναπαριστά τους φυσικούς αριθμούς, τότε στην Haskell θα γράψουμε

```
data Nat = Zero | Succ Nat
```

όπου `Zero` ο αριθμός μηδέν και `Succ` ο επόμενος κάποιου φυσικού αριθμού (το σύμβολο `|` διαβάζεται, σε αυτή την περίπτωση, *ή*). Το παράδειγμα στηρίζεται στην θεώρηση ότι κάθε φυσικός αριθμός μπορεί να παρασταθεί με βάση το μηδέν και μια συνάρτηση που υπολογίζει τον επόμενο κάποιου αριθμού. Έτσι ο αριθμός $2 =$ `SuccSuccZero`. Αν θέλουμε να ορίσουμε για παράδειγμα ένα δυαδικό δένδρο, τότε πρέπει να καταφύγουμε στον ορισμό ενός παραμετρικού επαγωγικού τύπου:

```
data BinTree a = Null | Node a (BinTree a) (BinTree a)
```

Προσέξτε ότι εδώ ορίζουμε το δυαδικό δένδρο να είναι είτε κενό, είτε να αποτελείται από ένα κόμβο που περιέχει μια πληροφορία τύπου `a` και δύο υπό-δένδρα, τα οποία με την σειρά των μπορούν να είναι είτε κενά, είτε να περιέχουν πληροφορίες, κ.ο.κ.

Όταν έχουμε επαγωγικούς τύπους μπορούμε να χρησιμοποιήσουμε τον ορισμό των για να ορίσουμε συναρτήσεις που θα τους διαχειρίζονται. Οι συναρτήσεις αυτές ορίζονται με πρότυπα, δηλαδή γενικές περιγραφές της δομής κάποιου ορίσματος. Για παράδειγμα στο σχήμα 2 δίνουμε την υλοποίηση του αλγορίθμου ταξινόμησης QuickSort του C.A.R. Hoare σε Haskell και C/C++ για να καταδείξουμε την εκφραστική δύναμη του συναρτησιακού προγραμματισμού αλλά και για να δείξουμε ένα παράδειγμα της χρήσης προτύπων (pattern matching). Προσέξτε ότι μια ακολουθία είναι είτε άδεια είτε περιέχει κάποια στοιχεία. Αν είναι άδεια, τότε απλά επιστρέφουμε την άδεια ακολουθία. Αν είναι γεμάτη, τότε πρέπει να γνωρίζουμε το πρώτο της στοιχείο και «συλλογικά» τα υπόλοιπα. Έτσι αν θέλουμε να υπολογίσουμε το μήκος μιας ακολουθίας η παρακάτω συνάρτηση κάνει με το παραπάνω την δουλειά μας:

```
len []       = 0
len (x:xs)   = 1 + len xs
```

Το ενδιαφέρον σε αυτό τον ορισμό είναι ότι μπορεί να εφαρμοσθεί σε κάθε είδους ακολουθία. Η δυνατότητα που έχουν πολλές συναρτησιακές γλώσσες να δηλώνει κανείς συναρτήσεις για ένα ευρύ φάσμα ομοειδών δομών ονομάζεται πολυμορφία. Ας δούμε ένα ακόμη παράδειγμα μιας πολύμορφης συνάρτησης η οποία αντιστρέφει μια ακολουθία:

```
reverse []    = []
reverse (h:t) = reverse t ++ [h]
```

Προσέξτε ότι εδώ δημιουργούμε νέες ακολουθίες που αποτελούνται από ένα μόνο στοιχείο.

Εκτός από τους επαγωγικούς τύπους μπορούμε να έχουμε και άπειρες δομές δεδομένων, η ύπαρξη και χρήση των οποίων είναι αποτέλεσμα του lazy evaluation. Για παράδειγμα, οι παρακάτω συναρτήσεις ορίζουν η μεν πρώτη μια άπειρη ακολουθία αποτελούμενη μόνο από το ψηφίο 1, ενώ η δε δεύτερη μια ακολουθία αποτελούμενη από τους αριθμούς από κάποιο ακέραιο n ως το άπειρο. Η τρίτη αποτελεί ένα εναλλακτικό τρόπο γραφής της δεύτερης. Η παράσταση `[m..]` αντιστοιχεί σε μία ακολουθία από το m ως το άπειρο. Αυτή η δυνατότητα, δηλαδή το να μπορεί κανείς να γράφει κάτι με κομψό τρόπο, ονομάζεται syntactic sugar.

```
ones = 1:ones
numsFrom n = n:numsFrom(n+1)
numbersFrom m = [m..]
```

(Μπορείτε μήπως να βρείτε τον τύπο της δεύτερης συνάρτησης?)

Γλώσσες όπως η C και Pascal δίνουν τη δυνατότητα σε κάποιο χρήστη να ορίζει λειτουργίες ή διαδικασίες των οποίων κάποιο όρισμα είναι μια άλλη λειτουργία ή διαδικασία. Η ίδια δυνατότητα παρέχεται και στις συναρτησιακές γλώσσες προγραμματισμού. Μάλιστα, αν μια συνάρτηση έχει ως παράμετρο μια άλλη συνάρτηση, ονομάζεται συνάρτηση υψηλότερης τάξης. Ιδού μια τέτοια συνάρτηση:

```
qsort []     = []
qsort (x:xs) =  qsort [y | y <- xs, y<x]
             ++ [x]
             ++ qsort [y | y <- xs, y>=x]
```

Ο αλγόριθμος QuickSort σε Haskell. Ο τελεστής ++ ενώνει δύο ακολουθίες σε μία.

```
void qsort(void *v[], int left, int right,
           int (*comp)(void *, void *))
{
   int i, last;
   void swap(void *v[], int, int);

   if(left>=right)
      return;
   swap(v,left,(left+right)/2);
   last=left;
   for(i=left+1; i<=right; i++)
      if((*comp)(v[i],v[left])<0)
         swap(v,++last,i);
   swap(v,left,last);
   qsort(v.left,last-1,comp);
   qsort(v,last+1,right,comp);
}
```

Ο αλγόριθμος QuickSort σε C/C++.

Σχήμα 2: Υλοποίηση του αλγορίθμου QuickSort σε Haskell και C/C++.

```
map f []      = []
map f (x:xs)  = f x : map f xs
```

Έτσι η παράσταση `map square [1,2,3,4,5]` θα δημιουργήσει την ακολουθία `[1,4,9,16,25]`. Προσέξτε ότι δεν χρειάζεται να καταφύγουμε σε κόλπα με δείκτες, όπως στην C, ή σε μονολιθικές δηλώσεις, όπως στην Pascal, για να ορίσουμε μια συνάρτηση υψηλότερης τάξης. Η υλοποίηση αναλαμβάνει τα πάντα!

Ένα άλλο ενδιαφέρον χαρακτηριστικό των συναρτησιακών γλωσσών αποτελεί η δυνατότητα ορισμού κάποιου συμβόλου ή λέξης ως δυαδικού τελεστή, όπως, π.χ., το +, το *, κ.λπ. Ορίστε πως ορίζεται στην Haskell ο τελεστής ++:

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Βέβαια, δεν καλύψαμε πλήρως τη γλώσσα, αλλά δώσαμε μια γεύση των δυνατοτήτων της. Το σημαντικό ερώτημα που πολλοί νέοι στο χώρο θέτουν είναι: Όντως είναι ενδιαφέρουσες και μαθηματικά κομψές οι συναρτησιακές γλώσσες προγραμματισμού, έχουν όμως χρησιμοποιηθεί σε «πραγματικές» εφαρμογές;

4 Συναρτησιακές γλώσσες και εφαρμογές

Οι συναρτησιακές γλώσσες προγραμματισμού έχουν χρησιμοποιηθεί επιτυχώς σε αρκετά σημαντικά προγράμματα (projects). Για παράδειγμα ο Mobility Server, ένα προϊόν που πωλείται σε δώδεκα ευρωπαϊκές χώρες, ελέγχει τα κινητά τηλέφωνα του Ευρωπαϊκού Κοινοβουλίου στο Στρασβούργο. Το σύστημα αυτό είναι γραμμένο στην Erlang, μια συναρτησιακή γλώσσα προγραμματισμού σχεδιασμένη από την Ericsson για τηλεπικοινωνιακές εφαρμογές. Η φιλοσοφία της εταιρείας είναι στα επόμενα χρόνια ότι λογισμικό παράγει, να γράφεται σε Erlang (το όνομα της δεν προέρχεται από τα αρχικά των λέξεων Ericsson language, αλλά από το όνομα του δανού μαθηματικού A.K. Erlang). Περισσότερες πληροφορίες σχετικά με την Erlang μπορείτε να βρείτε στο URL: `http://www.erlang.org/`. Μια άλλη εφαρμογή στον τηλεπικοινωνιακό τομέα είναι το σύστημα Pdiff το οποίο είναι γραμμένο σε SML και αποτελεί τμήμα

του τηλεπικοινωνιακού συστήματος 5ESS (phone switch system). Το σύστημα αυτό παράγεται από τα Εργαστήρια Bell.

Ένα άλλο είδος εφαρμογών στο οποίο χρησιμοποιούνται κύρια συναρτησιακές γλώσσες προγραμματισμού είναι τα συστήματα αυτόματης απόδειξης θεωρημάτων. Για παράδειγμα το σύστημα HOL(Higher Order Logic) χρησιμοποιήθηκε στην παραγωγή των επεξεργαστών HP9000 της εταιρείας Hewlett-Packard. Μάλιστα μπόρεσε να διαγνώσει προβλήματα τα οποία δεν ήταν δυνατό να ανακαλυφθούν μετά από έξι μήνες προσομοίωσης. Επίσης ο Οργανισμός Αμυντικής Τεχνολογίας και Επιστήμης του Υπουργείου Αμύνης της Αυστραλίας, χρησιμοποιεί το σύστημα Isabelle για την επαλήθευση των συνθηκών οπλισμού παγίδων βλημάτων. Τα δύο αυτά συστήματα είναι γραμμένα σε SML. Η γλώσσα SML είναι απόγονος της γλώσσας ML η οποία χρησιμοποιούταν ως «μεταγλώσσα» στο σύστημα LCF, ενός πρωτοποριακού συστήματος απόδειξης θεωρημάτων.

Κάθε πτήση από ή προς τα αεροδρόμια Orly και Roissy του Παρισιού επεξεργάζεται από ένα έμπειρο σύστημα το οποίο ονομάζεται Ivanhoe το οποίο είναι γραμμένο στο κέλυφος εμπείρων συστημάτων Natural Expert. Το κέλυφος αυτό παρέχει μια γλώσσα προγραμματισμού η οποία μοιάζει πάρα πολύ με την Haskell. Έτσι κανείς ουσιαστικά προγραμματίζει έμπειρα συστήματα σε μία συναρτησιακή γλώσσα προγραμματισμού. Παράλληλα, στην Γαλλία η εταιρεία Polygram, η οποία ελέγχει το ένα τρίτο της ευρωπαϊκής αγοράς CD και κασετών, χρησιμοποιεί το έμπειρο σύστημα Colisage στην γραμμή πακεταρίσματος. Το σύστημα αυτό είναι γραμμένο στο κέλυφος Natural Expert.

Η Ensemble είναι μια βιβλιοθήκη πρωτοκόλλων η οποία μπορεί να χρησιμοποιηθεί για την εύκολη αλλά και γρήγορη δημιουργία κατανεμημένων (distributed) εφαρμογών. Το σύστημα Ensemble βρίσκεται σε καθημερινή χρήση στο Πανεπιστήμιο Cornell, όπου μεταξύ άλλων χρησιμοποιείται για την υποστήριξη της αποθήκευσης ηχητικών CD αλλά και της αναπαραγωγής των. Το ίδιο σύστημα χρησιμοποιείται από μια σειρά εταιρειών (BBN, Lockheed Martin, Microsoft, κ.ά.) για τη δημιουργία εμπορικών προϊόντων. Το πρόγραμμα είναι γραμμένο στη γλώσσα Objective Caml, μιας διαλέκτου της SML. Επιπλέον σε μια προσπάθεια να φανούν οι δυνατότητες του συναρτησιακού προγραμματισμού, ερευνητές του Πανεπιστημίου Carnegie Mellon δημιούργησαν όλα τα εργαλεία που απαιτούνται για το στήσιμο ενός συστήματος πρόσβασης σε ιστοσελίδες (web server). Για περισσότερες πληροφορίες για το σύστημα μπορείτε να διαβάσετε τα σχετικά στο URL: `http://foxnet.cs.cmu.edu/`. Περισσότερες πληροφορίες για μεγάλες εφαρμογές των συναρτησιακών γλωσσών προγραμματισμού μπορείτε να βρείτε στο παρακάτω URL: `http://www.cs.bell-labs.com/who/wadler/realworld/index.html`.

Αφού λοιπόν έχουν τόσες εφαρμογές οι συναρτησιακές γλώσσες προγραμματισμού, γιατί δεν είναι ευρέως διαδεδομένες; Καταρχήν ένας σημαντικός λόγος είναι το γεγονός ότι οι περισσότερες από τις συναρτησιακές γλώσσες αναπτύχθηκαν σε ερευνητικά κέντρα στα οποία ενδιαφέρονταν για γλώσσες μαθηματικώς «καθαρές». Η πραγματικότητα όμως επιβάλει τη δυνατότητα κλήσης ρουτινών σε άλλες γλώσσες (κυρίως σε C), μιας και δεν υπάρχει κανένας λόγος να ανακαλύπτουμε συνεχώς το τροχό. Αντιλαμβανόμενοι αυτό το μειονέκτημα, πολλά ενεργά μέλη της κοινότητας του συναρτησιακού προγραμματισμού προχώρησαν σε βήματα προς τον ορθό δρόμο. Έτσι η Erlang παρέχει τη δυνατότητα κλήσης λειτουργιών σε C, ενώ η SISAL υπορουτινών σε FORTRAN. Επίσης πρόσφατες εξελίξεις επιτρέπουν κάθε πρόγραμμα Haskell να γίνεται ένα στοιχείο COM, ενώ κάθε τέτοιο στοιχείο μπορεί να χρησιμοποιείται από οποιοδήποτε πρόγραμμα σε Haskell.

Ένα άλλο σημαντικό στοιχείο αποτελεί το γεγονός ότι οι περισσότερες συναρτησιακές γλώσσες έχουν παρά μόνο βασικές βιβλιοθήκες. Από την άλλη η Java δεν οφείλει την επιτυχία της τόσο στο ότι είναι μια πραγματικά σπουδαία γλώσσα, αλλά στο γεγονός ότι παρέχει μια τεράστια βιβλιοθήκη με δυνατότητες σχεδιασμού γραφικών, επεξεργασίας εικόνας, τηλεφωνίας, κ.ο.κ. Βέβαια υπάρχουν βιβλιοθήκες και για την Haskell για, π.χ., σχεδιασμό γραφικών και δημιουργία εφαρμογών τύπου GUI, δηλαδή με γραφικό διάμεσο, αλλά με κανένα τρόπο δεν φτάνουν την πληθώρα των βιβλιοθηκών της Java. Αυτό ίσως να σχετίζεται και με το γεγονός ότι μόνο ελάχιστες συναρτησιακές γλώσσες προγραμματισμού είναι πραγματικά εμπορικά προϊόντα όπως: η Erlang, η ML Works της Harlequin (`http://www.harlequin.com/products/ads/ml/`), η Miranda της Research Software Limited, και μερικές ακόμη.

Στην εποχή που η Java και η Perl διαφημίζονται ως γλώσσες στις οποίες γράφεις ένα πρόγραμμα και αυτό τρέχει σε οποιαδήποτε υπολογιστική πλατφόρμα, οι συναρτησιακές γλώσσες δυστυχώς δεν έχουν υλοποιηθεί παρά στις πλέον γνωστές. Βέβαια αυτό δεν είναι μεγάλο πρόβλημα, αφού κανείς μπορεί εύκολα να φτιάξει μια *εν δυνάμει μηχανή* (virtual machine) σε C και αυτή να λειτουργεί σε οποιαδήποτε πλατφόρμα, ακριβώς όπως συμβαίνει με την Java. Στο σημείο αυτό αξίζει να αναφέρουμε ότι η Java έγινε γνωστή και για το γεγονός ότι δεν απαιτεί από τον χρήστη της να διαχειρίζεται τη μνήμη μιας και το κάνει αυτό αυτόματα (η τεχνική αυτή ονομάζεται «αποκομιδή σκουπιδιών» ή garbage collection). Όμως η τεχνική αυτή είναι γνωστή στους κύκλους των συναρτησιακών προγραμματιστών εδώ και…δεκαετίες!

Ένας άλλος λόγος που κάνει σχετικά δύσκολη τη χρήση των συναρτησιακών γλωσσών προγραμματισμού είναι η ελλιπής κατάρτιση των προγραμματιστών. Δεν αρκεί κάποιος να γνωρίζει μία (προστακτική) γλώσσα ώστε να μπορεί να μάθει μια συναρτησιακή. Απαιτούνται δύο περίπου βδομάδες ώστε κάποιος να μάθει να προγραμματίζει συνθέτοντας συναρτήσεις και μη έχοντας στη διάθεσή του μεταβλητές, παρά μόνο σταθερές. Βέβαια έχουν γίνει βήματα για την αντιμετώπιση του προβλήματος, αλλά αυτό που ουσιαστικά απαιτείται είναι οι φοιτητές των τμημάτων των υπολογιστικών επιστημών να αποκτούν εξοικείωση με αυτού του είδους τις γλώσσες. Να σημειωθεί ότι η εκμάθηση της Lisp είναι όχι μόνο παρωχημένη αλλά και δεν προσφέρει τίποτα το ουσιαστικό πλέον.

Όλοι οι παραπάνω είναι λόγοι που αποτρέπουν αρκετό κόσμο να ασχοληθεί σοβαρά με τον συναρτησιακό προγραμματισμό, ενώ υπάρχουν και δύο βασικοί μύθοι οι οποίοι διαψεύδονται από τα γεγονότα. Μύθος πρώτος: οι υλοποιήσεις των συναρτησιακών γλωσσών είναι αργές. Αυτό δεν ισχύει αφού σε πολλές περιπτώσεις υπάρχουν συναρτησιακά προγράμματα που «τρέχουν» γρηγορότερα από τα αντίστοιχα προγράμματα σε C. Επιπλέον, η Java που είναι τόσο δημοφιλής είναι μια σχετικά αργή γλώσσα, ακόμη και σε Solaris! Μύθος δεύτερος: ο συναρτησιακός προγραμματισμός είναι δύσκολος. Μια ανακρίβεια που οφείλεται στο γεγονός ότι απαιτείται κανείς να μάθει να σκέφτεται διαφορετικά. Όμως το ίδιο δεν συμβαίνει όταν κανείς μαθαίνει μια άλλη (φυσική) γλώσσα, π.χ., γαλλικά, ρωσικά, κ.ά.

5 Περισσότερες πληροφορίες

Αν έχετε πιεσθεί ότι αξίζει τον κόπο να ασχοληθείτε με τον συναρτησιακό προγραμματισμό, θα πρέπει να «κατεβάσετε» μια γλώσσα για το υπολογιστικό σας σύστημα. Αυτό είναι πολύ απλό: επισκεφθείτε το URL: `http://www.haskell.org`. Εκεί θα βρείτε πάρα πολλές πληροφορίες σχετικά με την Haskell, την υλοποίησης της γλώσσας αλλά και βιβλιογραφία σχετικά με τον συναρτησιακό προγραμματισμό γενικότερα. Εμείς για αρχή σας προτείνουμε να πειραματιστείτε με την Hugs, έναν διερμηνευτή της Haskell. Αρκετές πληροφορίες, σχετικά με τον συναρτησιακό προγραμματισμό αλλά και άλλες γλώσσες προγραμματισμού, μπορείτε να βρείτε στις σελίδες των ερευνητικών ομάδων στα αντίστοιχα τμήματα των Πανεπιστημίων Chalmers & Goteborg της Σουηδίας (`http://www.cs.chalmers.se/Cs/Research/Functional/`), του Πανεπιστημίου της Γλασκόβης στη Σκοτία (`http://www.dcs.glasgow.ac.uk/fp/`) και του Πανεπιστημίου Yale στις ΗΠΑ (`http://www.cs.yale.edu/haskell/yale-fp.html`). Επιπλέον, μπορείτε να διαβάζεται το newsgroup `comp.lang.faunctional`. Επίσης υπάρχει και ένα επιστημονικό περιοδικό σχετικά με τον συναρτησιακό προγραμματισμό: Journal of Functional Programming (`http://www.cup.cam.ac.uk/scripts/webjrnl.asp?mnemonic=jfp`) που εκδίδεται από τον εκδοτικό οίκο Cambridge University Press. Τέλος, υπάρχει και μια στήλη στο περιοδικό SIGPLAN Notices της ACM με θέματα σχετικά με τον συναρτησιακό προγραμματισμό.

Ελπίζουμε η σύντομη αυτή παρουσίαση να συντελέσει στην άνοδο της χρήσης αυτού του τόσο γοητευτικού τρόπου επίλυσης υπολογιστικών προβλημάτων!