Creative Scala

Dave Gurnell and Noel Welsh

underscore

Creative Scala

Dave Gurnell και Noel Welsh

Φεβρουάριος 2017

Creative Scala

Published by Underscore Consulting LLP, Brighton, UK.

Αντίγραφα αυτού του βιβλίου και σχετικών θεμάτων μπορούν να βρεθούν στο http://underscore.io/training. Όταν υπάρχουν εκπτώσεις για ομάδες, αναρτώνται στην ίδια διεύθυνση. Για θέματα σχετικά με το κείμενο της αγγλικής έκδοσης, επικοινωνήστε με τους συγγραφείς στο: hello@underscore.io.

Τα μαθήματά, τα εργαστήρια και τα άλλα μας προϊόντα, μπορούν να βοηθήσουν εσάς και την ομάδα σας να δημιουργήσετε καλύτερο λογισμικό με ευχάριστο τρόπο. Για περισσότερες πληροφορίες, καθώς και για τους τελευταίους τίτλους της Underscore, επισκευθείτε την διεύθυνση http://underscore.io/training.

Δήλωση αποποίησης ευθύνης: *Κατά την προετοιμασία αυτού του βιβλίου, πάρθηκαν όλες οι απαραίτητες προφυλάξεις. Παρόλα αυτά,* ***οι συγγραφείς και η Underscore Consulting LLP αποποιούνται οποιαδήποτε ευθύνη σχετικά με λάθη, παραλείψεις ή καταστροφές*** *που μπορεί να προκληθούν από την χρήση των πληροφοριών που υπάρχουν σε αυτό το βιβλίο (συμπεριλαμβανομένων των καταχωρήσεων προγραμμάτων).*

Μετάφραση: Μαρία Μπάτσου mbatsou@gmail.com

Επιμέλεια μετάφρασης: Δημοσθένης Μιχαηλίδης
mrdimosthenis@hotmail.com

Για πληροφορίες σχετικές με την μετάφραση του βιβλίου επισκεφθείτε το

GitHub repository mrdimosthenis/creative-scala

Creative Scala

ΠΡΟΛΟΓΟΣ

Το βιβλίο Creative Scala απευθύνεται σε προγραμματιστές που δεν έχουν προηγούμενη εμπειρία με την Scala. Υποθέτουμε ότι είστε εξοικειωμένοι με κάποια άλλη γλώσσα προγραμματισμού αλλά έχετε λίγη ή καθόλου εμπειρία με την Scala ή άλλες γλώσσες συναρτησιακού προγραμματισμού.

Σε αυτό το βιβλίο θέσαμε 3 στόχους:

1. Να γνωρίσετε και να εξοικειωθείτε με τον συναρτησιακό προγραμματισμό, να μπορείτε να φτιάξετε προγράμματα αλλά και να είστε σε θέση να καταλάβετε και άλλα σχετικά εισαγωγικά βιβλία.
2. Να σας διδάξουμε αρκετή Scala ώστε να μπορέσετε να εξερευνήσετε και δικά σας ενδιαφέροντα χρησιμοποιώντας την.
3. Να τα παρουσιάσουμε όλα αυτά με διασκεδαστικό, ήπιο και ενδιαφέρον τρόπο μέσω δισδιάστατων γραφικών.

Ως κίνητρο είχαμε τις δικές μας εμπειρίες εκμάθησης και μελέτης συναρτησιακού προγραμματισμού καθώς και διδασκαλίας της Scala σε επαγγελματίες προγραμματιστές.

Πρώτα απ’ όλα, πιστεύουμε ότι ο συναρτησιακός προγραμματισμός είναι το μέλλον. Αφού υποθέτουμε ότι έχετε μικρή εμπειρία στον προγραμματισμό, δεν θα μπούμε σε λεπτομέρειες σχετικά με τις διαφορές του συναρτησιακού με τον αντικειμενοστραφή προγραμματισμό, με τον οποίο μπορεί να έχετε ήδη μία εξοικείωση. Αρκεί να πούμε ότι υπάρχουν πολλοί διαφορετικοί τρόποι να σκεφτούμε και να γράψουμε προγράμματα και εμείς επιλέξαμε την χρήση του συναρτησιακού προγραμματισμού.

Οι λόγοι όμως που χρησιμοποιήσαμε συναρτησιακό προγραμματισμό είναι λίγο πιο ενδιαφέροντες από όσο αφήσαμε να εννοηθεί παραπάνω. Η διδασκαλία προγραμματισμού χρησιμοποιώντας ένα “bag of syntax” είναι πολύ συχνό φαινόμενο. Σε αυτή την περίπτωση μία γλώσσα προγραμματισμού διδάσκεται ως μία συλλογή συντακτικών στοιχείων (μεταβλητές, βρόχοι for και while, μέθοδοι κλπ) και οι μαθητές προσπαθούν να καταλάβουν μόνοι τους πότε πρέπει να

χρησιμοποιήσουν το κάθε ένα από αυτά τα στοιχεία. Έχουμε δει αυτή τη μέθοδο να αποτυγχάνει πολλές φορές, πρώτα όταν ήμασταν οι ίδιοι φοιτητές αλλά και αργότερα καθώς διδάσκαμε προγραμματισμό, αφού οι μαθητές δεν διδάσκονταν έναν συστηματικό τρόπο διάσπασης προβλημάτων σε μικρά κομμάτια τα οποία θα μπορούσαν να μετατρέψουν σε κώδικα ευκολότερα. Ως αποτέλεσμα, οι περισσότεροι τα παρατούσανε λόγω της κακής ποιότητας διδασκαλίας. Οι μαθητές οι οποίοι τα κατάφερναν μέχρι το τέλος, όπως και εμείς, είχαν ήδη αρκετή εμπειρία στον προγραμματισμό.

Ας θυμηθούμε τα μαθηματικά του δημοτικού και συγκεκριμένα την πρόσθεση κατά στήλη. Αυτός είναι ο βασικός τρόπος με τον οποίο μαθαίναμε να προσθέτουμε αριθμούς οι οποίοι είναι πολλύ μεγάλοι για να τους προσθέσουμε στο μυαλό μας. Έτσι, για παράδειγμα, αν έπρεπε να προσθέσουμε 266+385, θα βάζαμε τους αριθμούς αυτούς τον έναν κάτω από τον άλλο, θα υπολογίζαμε τα κρατούμενα και ούτω καθεξής. Μπορεί τα μαθηματικά να μην ήταν το αγαπημένο σας μάθημα αλλά αυτή η εμπειρία μπορεί να μας διδάξει μερικά πράγματα. Πρώτον, μας δίνονταν ένας συστηματικός τρόπος για να φτάσουμε στην λύση. Δεύτερον, δεν χρειάζονταν να καταλάβουμε γιατί αυτός ο τρόπος λειτουργεί (παρόλο που βοηθάει) ώστε να τον χρησιμοποιήσουμε. Αν ακολουθούσαμε τα βήματα, θα φτάναμε στην σωστή απάντηση.

Αυτό που είναι αξιοθαύμαστο στον συναρτησιακό προγραμματισμό είναι ότι λειτουργεί ακριβώς όπως η πρόσθεση κατά στήλη. Έχουμε συνταγές που εγγυώνται ότι θα μας δώσουν την σωστή απάντηση αν τις ακολουθήσουμε σωστά. Αυτή την διαδικασία την αποκαλούμε *υπολογισμό* προγράμματος. Αυτό δεν σημαίνει όμως ότι από τον προγραμματισμό λείπει η δημιουργικότητα, αλλά ότι η πρόκληση είναι το να καταλάβουμε την δομή του προγράμματος και μόλις το καταφέρουμε αυτό θα ακολουθήσει αμέσως και η συνταγή που πρέπει να χρησιμοποιήσουμε. Ο κώδικας από μόνος του δεν είναι το ενδιαφέρον μέρος.

Διδάσκουμε συναρτησιακό προγραμματισμό χρησιμοποιώντας Scala, αλλά δεν διδάσκουμε την ίδια την Scala. Η Scala είναι μία γλώσσα που βρίσκεται σε μεγάλη ζήτηση αυτή την εποχή. Όσοι προγραμματίζουν σε Scala μπορούν να βρουν δουλειά σχετικά εύκολα σε διάφορες επιχειρήσεις και αυτό αποτελεί ένα κίνητρο για να ασχοληθεί κανείς με

αυτή. Ένας από τους λόγους που είναι τόσο δημοφιλής είναι επειδή συνδυάζει την αντικειμενοστρέφεια, τον παλιό τρόπο προγραμματισμού καθώς και τον συναρτησιακό προγραμματισμό. Υπάρχει πολύς κώδικας που είναι γραμμένος χρησιμοποιώντας αντικειμενοστρέφεια και πολλοί οι προγραμματιστές που έχουν συνηθίσει σ' αυτόν τον τρόπο. Η Scala παρέχει έναν ήπιο τρόπο για να πάει κανείς από τον αντικειμενοστραφή προγραμματισμό στον συναρτησιακό. Αυτό όμως σημαίνει ότι η Scala είναι μία τεράστια γλώσσα και η αλληλεπίδραση μεταξύ των δυο προαναφερόμενων τρόπων προγραμματισμού μπορεί να φανεί περίπλοκη. Πιστεύουμε ότι ο συναρτησιακός προγραμματισμός είναι πολύ πιο αποτελεσματικός από τον αντικειμενοστραφή και ότι δεν χρειάζεται να προσθέσουμε περισσότερη σύγχυση στους νέους προγραμματιστές με την παράλληλη εκμάθηση των τεχνικών αντικειμενοστρέφειας, καθώς μαθαίνουν για τον συναρτησιακό. Αυτό μπορεί να συμβεί αργότερα. Οπότε, σ' αυτό το βιβλίο χρησιμοποιούνται αποκλειστικά οι συναρτησιακές τεχνικές της Scala.

Τέλος, για να εξερευνήσουμε τον συναρτησιακό προγραμματισμό και την Scala, επιλέξαμε μία μέθοδο που ελπίζουμε ότι θα σας φανεί ευχάριστη: τα γραφικά υπολογιστών. Υπάρχουν πάρα πολλές εισαγωγές για την Scala, αλλά οι περισσότερες χρησιμοποιούν παραδείγματα που αναφέρονται σε επιχειρήσεις ή σε μαθηματικά. Για παράδειγμα, μία από τις πρώτες ασκήσεις στη δημοφιλή σειρά μαθημάτων Coursera, είναι να δημιουργηθούν σύνολα χρησιμοποιώντας δείκτες. Πιστεύουμε ότι αν σας αρέσει αυτού του είδους το εκπαιδευτικό περιεχόμενο τότε έχετε ήδη αρκετό υλικό στη διάθεσή σας. Εμείς θέλουμε να στοχεύσουμε σε μία διαφορετική κατηγορία ανθρώπων: σε αυτούς που πιστεύουν ότι τα μαθηματικά δεν τους ταιριάζουν αλλά παρόλα αυτά διαθέτουν κάποιο ενδιαφέρον ή εκτίμηση για τις οπτικές τέχνες. Δεν θα πούμε ψέματα: υπάρχουν μαθηματικά στο βιβλίο, αλλά ελπίζουμε ότι θα μπορέσουμε να σας παρακινήσουμε και να οπτικοποιήσουμε τις διάφορες έννοιες που αναλύουμε, έτσι ώστε να γίνουν λιγότερο τρομακτικές.

Ενώ το βιβλίο θα σας παρέχει το βασικό τρόπο σκέψης για να γίνετε ικανός στην χρήση της Scala, δεν θα γνωρίζετε τα πάντα ώστε να μπορείτε να εργάζεστε μόνοι σας. Για να προχωρήσετε σε υψηλότερο επίπεδο σας προτείνουμε να δείτε ένα από τα πολλά υπέροχα εγχειρίδια για Scala, συμπεριλαμβανομένου και του δικού μας Essential Scala.

Αν δουλεύετε τις εργασίες μόνοι σας, συνιστούμε να μπείτε στο δικό μας Gitter chat room ώστε να βρείτε βοήθεια αλλά και να πείτε και την δική σας γνώμη για το βιβλίο.

Το κείμενο του βιβλίου Creative Scala είναι ανοιχτού κώδικα, όπως είναι και ο κώδικα της βιβλιοθήκης ζωγραφικής Doodle που χρησιμοποιείται στις ασκήσεις. Μπορείτε να πάρετε τον κώδικα από τον λογαριασμό μας στο Github Github account. Μπορείτε επίσης να επικοινωνήστε μαζί μας με email ή μέσω του Gitter εάν θέλετε να συμβάλλετε.

Ευχαριστούμε που κατεβάσατε το βιβλίο και ευχόμαστε καλό και δημιουργικό προγραμματισμό!

—Dave και Noel

Παρατηρήσεις για την πρώτη έκδοση

> Αυτή η έκδοση της Creative Scala δεν είναι το τελικό προϊόν . Μπορεί να υπάρχουν ορθογραφικά και άλλα λάθη στο κείμενο και στα παραδείγματα.
>
> Εαν εντοπίσετε λάθη ή θα θέλατε να παρέχετε ανατροφοδότηση, παρακαλούμε να μας ενημερώσετε μέσω του Gitter chat room ή με email:
>
> - Dave Gurnell (dave@underscore.io)
> - Noel Welsh (noel@underscore.io)

Ευχαριστίες

Η Creative Scala γράφτηκε από τον Dave Gurnell και τον Noel Welsh. Πολλές ευχαριστίες στους Richard Dallaway, Jonathan Ferguson, και στην ομάδα της Underscore για την ανεκτίμητη συνεισφορά τους και τις διορθώσεις που έκαναν.

Ευχαριστούμε επίσης τους ανθρώπους που μας υπέδειξαν λάθη ή έκαναν προτάσεις ώστε να βελτιωθεί το βιβλίο: Neil Moore, Kelley Robinson, Julie Pitt, και τους υπόλοιπους διοργανωτές της ScalaBridge,

d43, τον Matt Kohl καθώς και όλους τους μαθητές που εργάστηκαν για την Creative Scala στην ScalaBridge, σε άλλες εκδηλώσεις ή από μόνοι τους. Ευχαριστούμε επίσης και τα πολλά και υπέροχα μέλη της κοινότητας της Scala που μας έδωσαν τα σχόλιά τους και τις προτάσεις τους από κοντά. Τέλος, τρέφουμε πολύ μεγάλη ευγνωμοσύνη για την Bridgewater και κυρίως για την Lauren Cipicchio, η οποία ίσως και ασυνείδητα χρηματοδότησε ένα μεγάλο κομμάτι της αρχικής ανάπτυξης της δεύτερης έκδοσης της Creative Scala και παρείχε τους πρώτους μαθητές.

1 Ξεκινώντας

Το πρώτο μας βήμα είναι να εγκαταστήσουμε το λογισμικό που χρειαζόμαστε για να δουλέψουμε με την Creative Scala. Επιλέξτε έναν από τους παρακάτω τρόπους:

1. Μπορείτε να δουλέψετε με έναν text editor και ένα τερματικό. *Προτείνουμε αυτόν τον τρόπο σε αυτούς που είναι εντελώς νέοι στον προγραμματισμό*.
2. Μπορείτε να δουλέψετε με το IntelliJ IDEA. Προτείνουμε αυτόν τον τρόπο σε αυτούς που έχουν συνηθίσει να χρησιμοποιούν ένα IDE ή δεν αισθάνονται άνετα χρησιμοποιώντας τερματικό.

Αν είστε έμπειρος προγραμματιστής και είστε ευχαριστημένος με το λογισμικό που χρησιμοποιείτε, κρατήστε το και προσαρμόστε τις οδηγίες που δίνονται παρακάτω όπως εσείς κρίνετε απαραίτητο.

Εάν όλα αυτά είναι καινούρια για εσάς, παρακάτω θα βρείτε περισσότερες διευκρινιστικές πληροφορίες.

1.1 Εγκατάσταση Τερματικού και Text Editor

Σε αυτήν την ενότητα δίνεται ο τρόπος με τον οποίο προτείνουμε εμείς να εργαστούν όσοι είναι νέοι στον προγραμματισμό. Περιγράφουμε το πώς μπορείτε να εγκαταστήσετε και να χρησιμοποιήσετε έναν text editor και το τερματικο ώστε να κάνετε τις ασκήσεις της Creative Scala. Θα πρέπει να εγκαταστήσετε:

- το JVM;
- το Git;
- έναν text editor και
- το template project για την Creative Scala.

1.1.1 OS X

Ανοίξτε το τερματικό. (Πατήστε τον μεγεθυντικό φακό στην πάνω δεξί

μέρος της εργαλειοθήκης. Πληκτρολογήστε την λέξη “terminal”.)

Εγκαταστήστε την Java: Πληκτρολογήστε στο τερματικό

```
java
```

Εάν εκτελεστεί τότε έχετε ήδη εγκατεστημένη την Java. Σε αντίθετη περίπτωση θα εμφανιστεί μία προτροπή για την εγκατάστασή της.

Εγκαταστήστε το homebrew. Επικολλήστε στο τερματικό την παρακάτω εντολή:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubuse
rcontent.com/Homebrew/install/master/install)"
```

Εγκαταστήστε το `git` χρησιμοποιώντας το homebrew. Στο τερματικό, πληκτρολογήστε:

```
brew install git
```

Τώρα εγκαταστήστε τον text editor Atom. Για ακόμα μία φορά πληκτρολογήστε στο τερματικό:

```
brew install Caskroom/cask/atom
```

Κάντε τις παρακάτω ρυθμίσεις για να υποστηρίζεται η Scala από τον Atom: Settings > Install > language-scala

Τώρα θα χρησιμοποιήσουμε το Git για να λάβουμε ένα SBT project το οποίο θα δουλέψουμε στην Creative Scala. Πληκτρολογήστε:

```
git clone https://github.com/underscoreio/creative-s
cala-template.git
```

Κοινοποίηση των εργασιών σας

Υπάρχει και ένας εναλλακτικός τρόπος εργασίας ο οποίος απαιτεί να γίνει αρχικά forking του template project της Creative Scala και στην συνέχεια να κλωνοποιηθεί στον υπολογιστή σας. Αυτόν τον

> τρόπο θα ήταν καλό να τον επιλέξετε εάν θέλετε να μοιράζεστε την δουλειά σας και με άλλους ανθρώπους. Για παράδειγμα μπορεί να θέλετε να εργαστείτε στην Creative Scala με έναν απομακρυσμένο εκπαιδευτή ή απλώς να θέλετε να δείξετε (και δικαίως) ότι είστε περήφανος γι' αυτά που καταφέρατε.
>
> Αν ακολουθήσετε αυτόν τον τρόπο πρέπει πρώτα να κάνετε *fork* το template της Creative Scala. Μετά δημιουργείστε έναν κλώνο του fork *σας*. Αυτός ο εναλλακτικός τρόπος περιγράφεται με περισσότερες λεπτομέρειες στην ενότητα για το Github αργότερα σε αυτό το κεφάλαιο.

Τώρα επισκευθείτε στον κατάλογο που δημιουργήσαμε και τρέξτε το SBT.

```
cd creative-scala-template
./sbt.sh
```

Το SBT πρέπει να ξεκινήσει. Μέσα στο SBT πληκτρολογήστε `console`. Τέλος πληκτρολογήστε

```
Example.image.draw
```

και θα πρέπει να εμφανιστεί μία εικόνα με τρεις κύκλους!

Εάν έχετε φτάσει μέχρι εδώ τότε έχετε καταφέρει επιτυχώς να εγκαταστήσετε όλο το λογισμικό που θα χρειαστείτε για να δουλέψετε με την Creative Scala.

Το τελικό βήμα είναι να φορτώσετε το Atom και να τον χρησιμοποιήσετε για να ανοίξετε το `Example.scala`, το οποίο μπορείτε να βρείτε στο μονοπάτι `src/main/scala`.

1.1.2 Windows

Κατεβάστε και εγκαταστήστε την Java. Ψάξτε για το "JDK" (Java development kit). Αυτό θα σας οδηγήσει στην ιστοσελίδα της Oracle. Αποδεχτείτε την άδειά τους και κατεβάστε το JDK. Τρέξτε αυτό που μόλις κατεβάσατε.

Κατεβάστε και εγκαταστήστε το Atom. Μεταβείτε στο `https://atom.io/` και κατεβάστε το Atom για Windows. Τρέξτε αυτό που μόλις κατεβάσατε.

Κατεβάστε και εγκαταστήστε το Git. Μεταβείτε στο `https://git-scm.com/` και κατεβάστε το Git για Windows. Τρέξτε αυτό που μόλις κατεβάσατε. Στο τέλος θα σας δοθεί η επιλογή να ανοίξετε το Git. Αποδεχθείτε την. Θα ανοίξει ένα παράθυρο με μία γραμμή εντολών. Πληκτρολογήστε:

```
git clone https://github.com/underscoreio/creative-scala-template.git
```

Κοινοποίηση των εργασιών σας

Υπάρχει και ένας εναλλακτικός τρόπος εργασίας ο οποίος απαιτεί να γίνει αρχικά forking του template project της Creative Scala και στην συνέχεια να κλωνοποιηθεί στον υπολογιστή σας. Αυτόν τον τρόπο θα ήταν καλό να τον επιλέξετε εάν θέλετε να μοιράζεστε την δουλειά σας και με άλλους ανθρώπους. Για παράδειγμα μπορεί να θέλετε να εργαστείτε στην Creative Scala με έναν απομακρυσμένο εκπαιδευτή ή απλώς να θέλετε να δείξετε (και δικαίως) ότι είστε περήφανος γι' αυτά που καταφέρατε.

Αν ακολουθήσετε αυτόν τον τρόπο πρέπει πρώτα να κάνετε *fork* το template της Creative Scala. Μετά δημιουργείστε έναν κλώνο του fork *σας*. Αυτός ο εναλλακτικός τρόπος περιγράφεται με περισσότερες λεπτομέρειες στην ενότητα για το Github αργότερα σε αυτό το κεφάλαιο.

Ανοίξτε μία συνηθισμένη γραμμή εντολών. Κάντε click στο εικονίδιο των Windows που βρίσκεται στο κάτω αριστερό μέρος της οθόνης. Στο κουτί αναζήτησης πληκτρολογήστε "cmd" και εκτελέστε το πρόγραμμα που βρέθηκε. Στο παράθυρο που άνοιξε πληκτρολογήστε

```
cd creative-scala-template
```

και θα οδηγηθείτε στον κατάλογο του template project της Creative Scala που μόλις κατεβάσατε. Πληκτρολογήστε

```
sbt.bat
```

ώστε να ξεκινήσει το SBT. Μέσα στο SBT πληκτρολογήστε `console`. Τέλος πληκτρολογήστε

```
Example.image.draw
```

και θα πρέπει να εμφανιστεί μία εικόνα με τρεις κύκλους!

Εάν έχετε φτάσει μέχρι εδώ τότε έχετε καταφέρει επιτυχώς να εγκαταστήσετε όλο το λογισμικό που θα χρειαστείτε για να δουλέψετε με την Creative Scala.

Το τελικό βήμα είναι να μπείτε στον Atom και να τον χρησιμοποιήσετε για να ανοίξετε το `Example.scala`, το οποίο μπορείτε να βρείτε στο μονοπάτι `src/main/scala`.

1.1.3 Linux

Ακολουθείστε τις εντολές για το OS X, αλλά αντί για το Homebrew χρησιμοποιήστε τον διαχειριστή πακέτων διανομών (distributions package manager) για την εγκατάσταση των πακέτων λογισμικού.

1.2 IntelliJ

Το IntelliJ είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης λογισμικού (IDE) για Scala καθώς και για άλλες γλώσσες προγραμματισμού. Έχει συμπεριλάβει πολλά προγραμματιστικά εργαλεία μέσα σε μία μόνο εφαρμογή και προτείνουμε την χρήση του σε άτομα που έχουν συνηθίσει να χρησιμοποιούν και άλλα IDEs όπως το Visual Studio ή το Eclipse.

Ξεκινήστε κατεβάζοντας και εγκαθιστώντας το IntelliJ. Για την εργασία σας με την Creative Scala μπορείτε να χρησιμοποιήσετε την έκδοση που διατίθεται δωρεάν. Καθώς γίνεται η εγκατάσταση του IntelliJ θα πρέπει να απαντήσετε σε μερικές ερωτήσεις. Στις περισσότερες περιπτώσεις μπορείτε να αποδεχτείτε τις προκαθορισμένες επιλογές. Όταν ερωτηθείτε για τα “featured plugins”, *βεβαιωθείτε ότι επιλέξατε την εγκατάσταση του plug-in για την Scala*.

Όταν ολοκληρώσετε τις ρυθμίσεις, θα εμφανιστεί ένα παράθυρο διαλόγου που θα σας ρωτάει αν θέλετε να δημιουργήσετε ένα νέο project, να φορτώσετε ένα project, να ανοίξετε ένα αρχείο, ή να κάνετε checkout από το version control. Διαλέξτε το “checkout from version control” και μετά επιλέξτε Github.

Στο νέο παράθυρο που θα ανοίξει, αλλάξτε το “Auth type” σε Token. Τώρα επισκεφθείτε το Github σε έναν περιηγητή. Επιλέξτε τον λογαριασμό σας (πάνω δεξιά στην σελίδα). Διαλέξτε “Settings” και μετά “Personal access tokens”. Δημιουργήστε ένα token. Ονομάστε το “intellij” και επιλέξτε το “repo”. Αντιγράψτε την ακολουθία αριθμών και γραμμάτων και επικολλήστε την στο πλαίσιο “Token”. Τώρα κάντε login στο Github.

Εγκαταστήστε το add-on για το SBT.

1.3 Χρήσιμες Πληροφορίες

Σε αυτή την ενότητα παρουσιάζονται κάποιες πληροφορίες σχετικά με τα εργαλεία που θα χρησιμοποιήσουμε. Εάν είστε πεπειραμένος προγραμματιστής πολλά από τα παρακάτω θα σας φανούν γνωστά, οπότε μπορείτε να τα προσπεράσετε. Εάν δεν είστε, ελπίζουμε ότι αυτά που θα σας πούμε θα σας βοηθήσουν να κατανοήσετε καλύτερα το λογισμικό με το οποίο θα δουλέψουμε.

1.3.1 Το τερματικό

Παλιά, όταν ο κόσμος ήταν ακόμη νέος και οι υπολογιστές σε βρεφική ηλικία, η πλέον συνηθισμένη διεπαφή χρήστη με τα παράθυρα, ο κέρσορας που ελέγχεται από το ποντίκι και η άμεση αλληλεπίδραση με τον υπολογιστή, δεν υπήρχαν. Αντί γι’ αυτά οι χρήστες πληκτρολογούσαν εντολές σε μία συσκευή που ονομάζονταν *τερματικό*. Η χρήση διεπαφής για άμεσο χειρισμό είναι πολύ βολική στις περισσότερες περιπτώσεις αλλά υπάρχουν φορές που προτιμάται το τερματικό ή *γραμμή εντολών*. Για παράδειγμα, εάν θέλαμε να υπολογίσουμε πόσο χώρο καταλαμβάνουν τα αρχεία των οποίων το όνομα ξεκινάει από `data` στα Linux ή στο OS X θα μπορούσαμε να εκτελέσουμε την εντολή

```
du -hs data*
```

Μπορούμε να χωρίσουμε αυτή την εντολή σε τρία επιμέρους στοιχεία προς εξέταση:

- η εντολή `du` δείχνει τη χρήση δίσκου (disk usage);
- η σημαία `-hs` εμφανίζει μία περίληψη που μπορεί να διαβαστεί από τους ανθρώπους και
- το `data*` δίνει όλα τα αρχεία των οποίων το όνομα ξεκινάει από `data`.

Για να γίνει αυτός ο υπολογισμός μέσω της διεπαφής άμεσης διαχείρισης θα απαιτούνταν πολύ περισσότερος χρόνος από το να πληκτρολογήσουμε αυτή την εντολή στο τερματικό.

Είναι δύσκολο να μάθει κανείς να χρησιμοποιεί την γραμμή εντολών όμως στο τέλος θα έχει στην διάθεσή του ένα πολύ δυνατό εργαλείο. Στην περίπτωσή μας, η χρήση του τερματικού θα είναι πολύ περιορισμένη, γι' αυτό μην ανησυχείτε αν βρήκατε το παραπάνω παράδειγμα τρομακτικό!

1.3.2 Text Editors

Το πιο πιθανό είναι ότι ήδη ξέρετε πώς να γράψετε ένα κείμενο σε κάποιον επεξεργαστή κειμένου. Οι επεξεργαστές κειμένου μας δίνουν την δυνατότητα να γράψουμε κείμενο και να ρυθμίσουμε τον τρόπο με τον οποίο αυτό εμφανίζεται όταν εκτυπωθεί σε μία σελίδα (κάτι που πλέον συμβαίνει πιο σπάνια). Οι επεξεργαστές κειμένου έχουν πολλές και δυνατές εντολές, όπως ορθογραφικό έλεγχο και αυτόματη δημιουργία πίνακα περιεχομένων, ώστε να κάνουν την διαδικασία της επεξεργασίας ευκολότερη.

Ένας *text editor* είναι ένας επεξεργαστής κειμένου για κώδικα. Αν θέλαμε να διαμορφώσουμε το πώς φαίνεται οπτικά ένα κείμενο, θα χρησιμοποιούσαμε κάποιον επεξεργαστή κειμένου ενώ αν θέλαμε να γράψουμε κώδικα θα χρησιμοποιούσαμε κάποιον text editor. Οι text editors παρέχουν διάφορες ειδικές λειτουργίες που τους κάνουν κατάλληλους για προγραμματισμό. Τυπικά παραδείγματα τέτοιων λειτουργειών είναι τα δυνατά εργαλεία αναζήτησης και αντικατάστασης κειμένου, καθώς και η δυνατότητα άμεσης αναπήδησης ανάμεσα στα διάφορα αρχεία που αποτελούν ένα project.

Οι text editors υπάρχουν από τότε που υπάρχουν και τα τερματικά και

μερικοί από αυτούς χρησιμοποιούνται ακόμη. Οι δύο παλαιότεροι αλλά ένδοξοι text editors που έχουν καταφέρει να επιβιώσουν μέχρι και σήμερα είναι ο Emacs και ο Vim. Είναι πολύ διαφορετικοί (όχι όμως πάντα) και οι προγραμματιστές συνήθως προτιμούν έναν από τους δύο. Εμείς χρησιμοποιούμε τον Emacs εδώ και περίπου είκοσι χρόνια οπότε γνωρίζουμε βαθιά μέσα μας ότι αυτός είναι ο καλύτερος text editor από όλους και ότι ο Vim είναι ένα υποδεέστερο εργαλείο και όσοι τον χρησιμοποιούν είναι κακόγουστοι. Χωρίς καμία αμφιβολία οι χρήστες του Vim σκέφτονται τα ίδια για εμάς.

Εάν υπάρχει κάτι που ενώνει τους χρήστες του Vim και του Emacs είναι η σίγουρη γνώση ότι οι μοντέρνοι text editors όπως ο Sublime Text και ο Atom φέρνουν την πτώση του πολιτισμού μας. Παρόλα αυτά συνιστούμε την χρήση του Atom εάν είστε νέος στον χώρο του text editing. Ο Vim και ο Emacs δημιουργήθηκαν πριν τις γνωστές διεπαφές εργασίας που υπάρχουν σήμερα και για να τους χρησιμοποιήσει κανείς απαιτείται η εκμάθηση ένος πολύ διαφορετικού τρόπου εργασίας.

1.3.3 Ο Μεταγλωττιστής (Compiler)

Ο κώδικας που γράφουμε σε έναν text editor δεν μπορεί να εκτελεστεί απευθείας από έναν υπολογιστή. Γι' αυτό, χρησιμοποιείται ένας *μεταγλωττιστής* ο οποίος μεταφράζει τον κώδικα σε κάτι που μπορεί να καταλάβει και να εκτελέσει ο υπολογιστής. Κατά τη διάρκεια της μεταγλώττισης γίνονται διάφοροι έλεγχοι. Εάν αυτοί οι έλεγχοι δεν "περάσουν" τότε ο κώδικας δεν θα μεταγλωττιστεί και θα εμφανιστεί κάποιο μήνυμα λάθους. Στη συνέχεια αυτού του βιβλίου θα μάθουμε περισσότερα για το τι μπορεί να ελέγξει ένας μεταγλωττιστής και τι όχι.

Όταν είπαμε παραπάνω ότι ο μεταγλωττιστής μεταφράζει τον κώδικα σε κάτι που μπορεί να εκτελέσει ένας υπολογιστής, αυτό δεν ήταν ολόκληρη η αλήθεια όσον αφορά την Scala. Το αποτέλεσμα της μεταγλώττισης είναι κάτι που ονομάζεται bytecode. Ένα άλλο πρόγραμμα, που ονομάζεται Java Virtual Machine (JVM), εκτελεί αυτόν τον κώδικα 1.

1.3.4 Ολοκληρωμένα περιβάλλοντα ανάπτυξης (Integrated Development

Environments)

Τα ολοκληρωμένα περιβάλλοντα ανάπτυξης (IDEs) αποτελούν μία εναλλακτική προσέγγιση στην ανάπτυξη λογισμικού η οποία συμπεριλαμβάνει έναν text editor, έναν μεταγλωττιστή και διάφορα άλλα προγραμματιστικά εργαλεία μέσα σε ένα μόνο πρόγραμμα. Κάποιο προγραμματιστές παίρνουν όρκο για το πόσο καλά είναι τα IDEs, αλλά κάποιοι άλλοι προτιμούν το τερματικό σε συνδυασμό με κάποιον text editor. Εάν είστε νέος στον προγραμματισμό προτείνουμε το τερματικό και τον text editor. Εάν είστε ήδη εξοικειωμένοι με κάποιο IDE τότε το IntelliJ IDEA είναι η καλύτερη επιλογή για ανάπτυξη κώδικα σε Scala.

1.3.5 Version Control

Το version control είναι το τελευταίο εργαλείο που θα χρησιμοποιήσουμε. Το σύστημα version control είναι ένα πρόγραμμα το οποίο μας επιτρέπει να καταγράψουμε τις αλλαγές που έγιναν σε μία ομάδα αρχείων. Είναι χρήσιμο γιατί επιτρέπει την εργασία πολλών ανθρώπων πάνω στο ίδιο project την ίδια στιγμή χωρίς να φοβάται κανείς ότι θα διαγράψει κατά λάθος αλλαγές που έκανε κάποιος άλλος. Το version control δεν είναι κάτι που θα μας απασχολήσει πολύ στην Creative Scala, αλλά θεωρήσαμε ότι είναι καλό να το γνωρίσετε από τώρα.

Το λογισμικό για version control που θα χρησιμοποιήσουμε εμείς ονομάζεται Git. Είναι πολύ δυνατό αλλά περίπλοκο. Τα καλά νέα είναι ότι δεν χρειάζεται να μάθουμε και πολλά για το Git. Θα χρησιμοποιούμε το Git περισσότερο μέσω μίας ιστοσελίδας που ονομάζεται Github και επιτρέπει την κοινοποίηση λογισμικού που είναι αποθηκευμένο στο Git. Εμείς χρησιμοποιούμε το Github ώστε να κοινοποιήσουμε το λογισμικό που χρησιμοποιήσαμε στην Creative Scala.

1.3.6 Συνεχίζοντας!

Τώρα που αποκτήσαμε μερικές πληροφορίες για τα εργαλεία που θα χρησιμοποιήσουμε ας προχωρήσουμε στην εγκατάσταση του λογισμικού που θα χρειαστούμε για να γράψουμε κώδικα σε Scala.

1.4 Github

Έχουμε δημιουργήσει για σας ένα template με όλο τον κώδικα που θα σας χρειαστεί καθώς δουλεύετε με την Creative Scala. Αυτό το template είναι αποθηκευμένο στο Github, μία ιστοσελίδα για κοινοποίηση κώδικα.

Μπορείτε να χρησιμοποιήσετε την επιλογή cloning που θα βρείτε στο Git ώστε να αντιγράψετε το template στον υπολογιστή σας. Με αυτόν τον τρόπο όμως δεν θα μπορείτε να αποθηκεύετε στο Github τις αλλαγές που θα κάνετε ώστε να μπορούν να τις δουν και άλλοι.

Εάν θέλετε να έχετε την δυνατότητα κοινοποίησης των αλλαγών σας, τότε πρέπει να δημιουργήσετε ένα αντίγραφο του template στο δικό σας Github. Στο Git αυτή η διαδικασία ονομάζεται forking. Πρέπει να κάνετε fork το repository στο Github και μετά να κλωνοποιήσετε στον υπολογιστή σας αυτό το *fork* που μόλις δημιουργήσατε. Έτσι θα μπορείτε να αποθηκεύετε στο Github τις αλλαγές που κάνετε στο fork σας.

Για να ξεκινήσετε αυτή τη διαδικασία πρέπει να δημιουργήσετε έναν λογαριασμό στο Github, εάν δεν έχετε ήδη.

Μόλις αποκτήσετε λογαριασμό, επισκεφθείτε το template project στον περιηγητή σας. Πάνω δεξιά υπάρχει ένα το κουμπί με την ονομασία “Fork”. Πατήστε αυτό το κουμπί για να δημιουργήσετε το δικό σας αντίγραφο του template. Θα μεταφερθείτε σε μία σελίδα στην οποία θα εμφανίζεται το δικό σας fork του template. Να θυμάστε ότι το όνομα του repository (πρέπει να μοιάζει με αυτό `yourname/creative-scala-template` όπου το `yourname` είναι το δικό σας όνομα χρήστη στο Github).

Τώρα η κλωνοποίηση του fork σας είναι τόσο εύκολη όσο η εκτέλεση της παρακάτω εντολής. Το μόνο που πρέπει να κάνετε είναι να αντικαταστήσετε το `yourname` με το όνομα χρήστη σας στο Github.

```
git clone git@github.com:yourname/creative-scala.git
```

Τώρα θα μπορείτε να στέλνετε στο fork σας στο Github όλες τις αλλαγές που κάνετε. Αυτή η διαδικασία στο Git είναι λίγο περίπλοκη. Όταν αλλάζετε κάτι και θέλετε να το στείλετε στο fork σας, θα πρέπει να κάνετε τα εξής βήματα:

- κάντε `add` την αλλαγή στο index του Git
- κάντε `commit` την αλλαγή και τέλος
- κάντε την `push` στο fork.

Παρακάτω μπορείτε να δείτε ένα παράδειγμα του πώς μπορεί να γίνει αυτό χρησιμοποιώντας την γραμμή εντολών.

```
git add
git commit -m "Explain here what you did"
git push
```

Το Github έχει δημιουργήσει ένα δωρεάν εργαλείο για την χρήση του Git που ονομάζεται Github Desktop. Είναι ίσως ο ευκολότερος τρόπος να χρησιμοποιήσετε το Git αν είστε ακόμη αρχάριοι.

---

1. Αυτή δεν είναι όμως όλη η αλήθεια! Συνήθως εκτελούμε τον κώδικα της Scala στη JVM αλλά στην πραγματικότητα μπορούμε να μεταγλωττίσουμε την Scala σε τρεις διαφορετικές μορφές. Η πρώτη και πιο κοινή είναι η JVM bytecode. Μπορούμε επίσης να μεταγλωττίσουμε σε Javascript, η οποία είναι και αυτή μία γλώσσα προγραμματισμού και επιτρέπει την εκτέλεση κώδικα Scala σε περιηγητή διαδικτύου (web browser). Τέλος, η Scala Native μπορεί να μεταγλωττίσει Scala σε κάτι που ένας υπολογιστής *μπορεί* να εκτελέσει κατευθείαν χωρίς να χρειάζεται η JVM. ↩

2 Εκφράσεις, Τιμές και Τύποι

Τα προγράμματα της Scala αποτελούνται από τρία βασικά δομικά στοιχεία: *τις εκφράσεις*, *τις τιμές* και *τους τύπους*. Η ενότητα αυτή είναι αφιερωμένη σ' αυτές τις τρεις έννοιες.

Παρακάτω μπορείτε να δείτε μία πολύ απλή έκφραση:

```
1 + 2
```

Μία *έκφραση* στην Scala είναι ένα μικρό κομμάτι κώδικα. Μπορούμε να γράψουμε εκφράσεις σε έναν text editor, σε ένα χαρτί, σε έναν τοίχο ή οπουδήποτε αλλού.

Οι εκφράσεις είναι όπως ο γραπτός λόγος. Όπως κάτι που είναι γραμμένο πρέπει να διαβαστεί από κάποιον ώστε να έχει αξία (και εννοείται ότι αυτός που το διαβάζει πρέπει να καταλαβαίνει την γλώσσα στην οποία είναι γραμμένο), έτσι και ο υπολογιστής πρέπει να *εκτελέσει* μία έκφραση ώστε αυτή να παράξει κάποιο αποτέλεσμα. Το αποτέλεσμα της εκτέλεσης μίας έκφρασης είναι μία *τιμή*. Οι τιμές ζουν στην μνήμη του υπολογιστή, όπως ζει κάτι που θα διαβάσει κάποιος μέσα στο κεφάλι του. Για να περιγράψουμε την διαδικασία μετατροπής των εκφράσεων σε τιμές πρέπει να πούμε ότι αυτές *αξιολογούνται* ή *εκτελούνται*.

Μπορούμε να αξιολογήσουμε εκφράσεις πολύ εύκολα αν τις γράψουμε στην κονσόλα και μετά πατήσουμε "Enter" (ή "Return"). Δοκιμάστε το τώρα.

```
1 + 2
// res1: Int = 3
```

Η κονσόλα θα μας επιστρέψει την τιμή με την οποία αξιολογείται η έκφραση, καθώς και τον τύπος της.

Η έκφραση `1 + 2` αξιολογείται με την τιμή `3`. Μπορούμε να καταγράψουμε τον αριθμό τρία σε μία σελίδα αλλά η αληθινή τιμή είναι κάτι που αποθηκεύεται στην μνήμη του υπολογιστή. Στη συγκεκριμένη περίπτωση, είναι ένας ακέραιος 32-bit αριθμός που αναπαρίσταται με συμπλήρωμα ως προς δύο. Το νόημα αυτής της αναπαράστασης του

ακεραίου δεν είναι σημαντικό. Απλώς το αναφέραμε ώστε να δώσουμε έμφαση στο γεγονός ότι η αναπαράσταση της τιμής `3` για τον υπολογιστή, είναι διαφορετική από τον αριθμό που είναι γραμμένος εδώ ή στην κονσόλα.

Οι *τύποι* είναι το τελευταίο κομμάτι του παζλ. Ένας τύπος είναι οτιδήποτε μπορούμε να συμπεράνουμε για το πρόγραμμα *χωρίς* να το τρέξουμε. Ο τύπος της έκφρασης `1 + 2` είναι `Int`. Από αυτό μπορούμε να καταλάβουμε ότι η τιμή της έκφρασης θεωρείται ακέραιος αριθμός. Αυτό σημαίνει επίσης ότι μπορούμε να γράψουμε και άλλες εκφράσεις που να χρησιμοποιούν το αποτέλεσμα αυτής της έκφρασης αλλά θα πρέπει να είναι τέτοιες ώστε να βγάζουν νόημα όταν χρησιμοποιούν ακεραίους. Μπορούμε να προσθέσουμε, να αφαιρέσουμε, να πολλαπλασιάσουμε ή να διαιρέσουμε αλλά για παράδειγμα δεν θα μπορούσαμε να μετατρέψουμε έναν ακέραιο σε πεζό γράμμα.

Οι τύποι μας δείχνουν το πώς πρέπει να εκλάβουμε μία τιμή (που υπάρχει μέσα στην μνήμη του υπολογιστή) η οποία ήταν το αποτέλεσμα μίας έκφρασης. Πρέπει να την εκλάβουμε ως έναν ακέραιο αριθμό ή ως μία ακολουθία σημείων που καθορίζουν την θέση του ποντικιού μία συγκεκριμένη στιγμή; Οι τύποι μπορούν να μας δώσουν αυτές τις απαντήσεις. Μπορούμε να χρησιμοποιήσουμε τύπους και για άλλα πράγματα, συμπεριλαμβανομένων και αυτών που δεν φαίνονται κατά τη διάρκεια της εκτέλεσης. Αυτές οι χρήσεις τύπων είναι λίγο πιο προχωρημένες από το τωρινό μας επίπεδο αλλά μην κάνετε το λάθος να πιστέψετε ότι οι τύποι αντιστοιχούν σε τιμές. Οι τύποι στην Scala υπάρχουν μόνο κατά την διάρκεια της μεταγλώττισης. Κατά την διάρκεια της εκτέλεσης, η παρουσία του τύπου μίας έκφρασης που είχε ως αποτέλεσμα μία συγκεκριμένη τιμή, δεν είναι απαραίτητη.

Πριν την εκτέλεση ενός προγράμματος Scala, πρέπει να γίνει η *μεταγλώττιση*. Η μεταγλώττιση ελέγχει αν ένα πρόγραμμα βγάζει νόημα. Πρέπει να είναι συντακτικά σωστό, δηλαδή να είναι γραμμένο σύμφωνα με τους κανόνες της Scala. Για παράδειγμα η έκφραση `(1 + 2)` είναι συντακτικά σωστή ενώ η `(1 + 2` δεν είναι. Επίσης, στη μεταγλώττιση γίνεται και *έλεγχος τύπων*, που σημαίνει ότι οι τύποι πρέπει να είναι οι κατάλληλοι για αυτό που θέλουμε να κάνουμε. Η έκφραση `1 + 2` περνάει από τον έλεγχο τύπων (προσθέτουμε ακέραιους) αλλά η `1.toUpperCase` δεν περνάει αφού στους αριθμούς δεν υπάρχει η έννοια των πεζών και

κεφαλαίων.

Στην φάση της εκτέλεσης μπορούν να περάσουν μόνο τα προγράμματα που μεταγλωττίζονται επιτυχώς. Θα μπορούσαμε να πούμε ότι η μεταγλώττιση ενός προγράμματος είναι ανάλογη με τον έλεγχο των κανόνων γραμματικής στον γραπτό λόγο. Η πρόταση “F$Rf fjrmn;l df.fd” είναι συντακτικά λάθος στα Αγγλικά. Η σειρά με την οποία έχουν μπει τα γράμματα δεν σχηματίζει λέξεις. Η πρόταση “σκυλί πετάει ένα εδώ όχι” αποτελείται από υπαρκτές λέξεις αλλά η σειρά τους δεν είναι σύμφωνη με τους κανόνες της γραμματικής—τα αντίστοιχα ισχύουν και για τους ελέγχους τύπων που εκτελεί η Scala.

Θα αναφερόμαστε στον *χρόνο-μεταγλώττισης* ως τον χρόνο κατά τον οποίο ο κώδικας μεταγλωττίζεται και στον *χρόνο-εκτέλεσης* ως τον χρόνο κατά τον οποίο εκτελείται.

2.1 Κυριολεκτικές Εκφράσεις

Σ’ αυτή την ενότητα θα εξερευνήσουμε τις διάφορες μορφές εκφράσεων στη Scala, ξεκινώντας από τις πιο απλές, τις *κυριολεκτικές*. Παρακάτω μπορείτε να δείτε ένα παράδειγμα κυριολεκτικής έκφρασης:

```
3
// res0: Int = 3
```

Μία κυριολεκτική έκφραση (ο αγγλικός όρος είναι “literal”) εκτιμάται ως ο ίδιος του ο ευατός. Ο τρόπος με τον οποίο γράφουμε την έκφραση και ο τρόπος με τον οποίο εκτυπώνει η κονσόλα την τιμή της έφρασης αυτής, είναι ο ίδιος. Πρέπει να θυμάστε όμως, ότι η γραπτή αναπαράσταση μίας τιμής είναι διαφορετική από την πραγματική της αναπαράσταση στην μνήμη του υπολογιστή.

Στη Scala υπάρχουν διάφορες μορφές κυριολεκτικών εκφράσεων. Μία από αυτές είναι το `Int`. Για τους *αριθμούς κινητής υποδιαστολής*, χρησιμοποιείται ένας διαφορετικός τύπος και μία διαφορετική σύνταξη. Αυτό αντιστοιχεί στο πώς αντιλαμβάνεται ο υπολογιστής τους πραγματικούς αριθμούς. Δείτε ένα παράδειγμα:

```
0.1
```

```
// res1: Double = 0.1
```

Όπως μπορείτε να δείτε, ο τύπος αυτής της κυριολεκτικής έκφρασης είναι ο `Double`.

Καλοί οι αριθμοί, αλλά τι γίνεται με το κείμενο; Η Scala μας παρέχει τον τύπο `String` για αναπαράσταση μίας ακολουθίας χαρακτήρων. Μπορούμε να γράψουμε κυριολεκτικές εκφράσεις strings βάζοντας το περιεχόμενό τους μέσα σε διπλά εισαγωγικά (“…”).

```
"To be fond of dancing was a certain step towards fa
lling in love."
// res2: String = To be fond of dancing was a certai
n step towards falling in love.
```

Αν θέλουμε να γράψουμε ένα string το οποίο θα χρειαστεί πολλές γραμμές για να χωρέσει, μπορούμε να χρησιμοποιήσουμε τρία διπλά εισαγωγικά (“””…”””), όπως φαίνεται παρακάτω.

```
"""
A new, a vast, and a powerful language is developed
for the future use of analysis,
in which to wield its truths so that these may becom
e of more speedy and accurate
practical application for the purposes of mankind th
an the means hitherto in our
possession have rendered possible.

  -- Ada Lovelace, the world's first programmer
"""
// res3: String =
// "
// A new, a vast, and a powerful language is develop
ed for the future use of analysis,
// in which to wield its truths so that these may be
come of more speedy and accurate
// practical application for the purposes of mankind
 than the means hitherto in our
```

```
// possession have rendered possible.
//
//   -- Ada Lovelace, the world's first programmer
// "
```

Ένα `String` είναι μία ακολουθία χαρακτήρων. Οι χαρακτήρες έχουν δικό τους τύπο, τον `Char` και γράφονται μέσα σε μονά εισαγωγικά.

```
'a'
// res4: Char = a
```

Τέλος, θα δούμε τις κυριολεκτικές αναπαραστάσεις του τύπου `Boolean`, ο οποίος πήρε το όνομά του από τον άγγλο μαθηματικό George Boolean. Αυτό το φανταχτερό όνομα σημαίνει απλώς ότι μία τιμή μπορεί να είναι είτε `true` (σωστή) είτε `false` (λάθος). Παρακάτω μπορείτε να δείτε τον τρόπο με τον οποίο γράφουμε τα κυριολεκτικά τύπου boolean:

```
true
// res5: Boolean = true

false
// res6: Boolean = false
```

Με τις κυριολεκτικές εκφράσεις μπορούμε να δημιουργήσουμε τιμές, όμως αυτό δεν είναι πολύ χρήσιμο αν δεν μπορούμε να αλληλεπιδράσουμε μαζί τους. Είδαμε ήδη μερικά παραδείγματα πιο περίπλοκων εκφράσεων όπως το `1 + 2`. Στην επόμενη ενότητα θα μάθουμε για τα αντικείμενα και τις μεθόδους, ώστε να μπορέσουμε να κατανοήσουμε πώς δουλεύουν τέτοιες και άλλου είδους πιο ενδιαφέρουσες εκφράσεις.

2.2 Οι Τιμές είναι Αντικείμενα

Στην Scala όλες οι τιμές είναι *αντικείμενα*. Ένα αντικείμενο είναι ένα σύνολο που αποτελείται από δεδομένα και διαδικασίες που τα αφορούν. Για παράδειγμα, ο αριθμός 2 είναι ένα αντικείμενο. Ο ακέραιος 2 είναι το δεδομένο του συνόλου, ενώ οι διαδικασίες είναι οι πράξεις όπως η

πρόσθεση (+), η αφαίρεση (-) και ούτω καθεξής. Αυτές τις διαδικασίες ενός αντικειμένου, τις ονομάζουμε *μεθόδους*.

2.2.1 Κλήσεις Μεθόδων

Αλληλεπιδρούμε με τα αντικείμενα *καλώντας* μεθόδους. Για παράδειγμα, μπορούμε να αλλάξουμε τα γράμματα ενός `String` σε κεφαλαία καλώντας την αντίστοιχη μέθοδό που ονομάζεται `toUpperCase`.

```
"Titan!".toUpperCase
// res0: String = TITAN!
```

Μερικές μέθοδοι δέχονται *παραμέτρους*, οι οποίες καθορίζουν το αποτέλεσμά της. Για παράδειγμα, η μέθοδος `take`, παίρνει χαρακτήρες από ένα `String`. Για να δείξουμε στην `take` πόσους χαρακτήρες θέλουμε να πάρουμε, πρέπει να της περάσουμε μία παράμετρο.

```
"Gilgamesh went abroad in the world".take(3)
// res1: String = Gil

"Gilgamesh went abroad in the world".take(9)
// res2: String = Gilgamesh
```

Μία κλήση μεθόδου είναι ουσιαστικά μία έκφραση και άρα θεωρείται αντικείνο. Αυτό σημαίνει ότι μπορούμε να παραθέσουμε κλήσεις μεθόδων την μία μετά την άλλη ώστε να φτιάξουμε πιο περίπλοκα προγράμματα:

```
"Titan!".toUpperCase.toLowerCase
// res3: String = titan!
```

Συντακτικό Κλήσης Μεθόδων

Το συντακτικό για κλήσεις μεθόδων είναι:

```
anExpression.methodName(param1, ...)
```

ή

```
anExpression.methodName
```

όπου

- `anExpression` είναι οποιδήποτε έκφραση (η οποία θεωρείται αντικείμενο)
- `methodName` είναι το όνομα της μεθόδου
- και το προαιρετικό `param1, ...` είναι μία ή περισσότερες εκφράσεις που λειτουργούν ως παράμετροι για την μέθοδο.

2.2.2 Τελεστές

Μέχρι τώρα έχουμε πει ότι όλες οι τιμές είναι αντικείμενα και ότι χρησιμοποιούμε το συντακτικό `object.methodName(parameter)` για να καλέσουμε μεθόδους. Πώς όμως εξηγούνται εκφράσεις όπως η `1 + 2`;

Στην Scala, εκφράσεις που γράφονται ως `a.b(c)` μπορούν να γραφούν ως `a b c`. Άρα είναι ισοδύναμες μεταξύ τους:

```
1 + 2
// res4: Int = 3

1.+(2)
// res5: Int = 3
```

Γραφή Ενδιάμεσου Τελεστή

Οποιαδήποτε έκφραση στην Scala η οποία είναι γραμμένη έτσι `a.b(c)` μπορεί επίσης να γραφεί και έτσι `a b c`.

Παρατηρήστε ότι το `a b c d e` είναι ισοδύναμο με το `a.b(c).d(e)` και όχι με το `a.b(c, d, e)`.

2.3 Τύποι

Τώρα που είμαστε σε θέση να γράψουμε περίπλοκες εκφράσεις ήρθε η ώρα να μιλήσουμε λίγο παραπάνω για τους τύπους.

Μία χρήση των τύπων είναι να μας αποτρέπουν από την κλήση μεθόδων που δεν υπάρχουν. Ο τύπος μίας έκφρασης λέει στον μεταγλωττιστή ποιες μέθοδοι υπάρχουν για την τιμή με την οποία αξιολογείται. Αν προσπαθήσουμε να καλέσουμε μία μέθοδο που δεν υπάρχει, ο κώδικάς μας δεν θα μεταγλωττιστεί. Παρακάτω δίνονται μερικά απλά παραδείγματα.

```
"Brontë" / "Austen"
// <console>:13: error: value / is not a member of S
tring
//        "Brontë" / "Austen"
//                 ^

1.take(2)
// <console>:13: error: value take is not a member o
f Int
//        1.take(2)
//          ^
```

Πραγματικά, ο τύπος μία έκφρασης, είναι αυτός που καθορίζει ποιες μεθόδους μπορούμε να καλέσουμε, κάτι που μπορούμε να επιδείξουμε καλώντας μεθόδους που αντιστοιχούν σε αποτελέσματα πιο περίπλοκων εκφράσεων.

```
(1 + 3).take(1)
// <console>:13: error: value take is not a member o
f Int
//        (1 + 3).take(1)
//                ^
```

Αυτή η διαδικασία *ελέγχου του τύπου* εφαρμόζεται και στις παραμέτρους των μεθόδων.

```
1.min("zero")
// <console>:13: error: type mismatch;
```

```
//  found   : String("zero")
//  required: Int
//        1.min("zero")
//                ^
```

Οι τύποι είναι ιδιότητα των εκφράσεων, επομένως υπάρχουν μόνο κατά τον χρόνο μεταγλώττισης (όπως έχουμε αναφέρει προηγουμένως). Αυτό σημαίνει ότι μπορούμε να καθορίσουμε τον τύπο μίας έκφρασης ακόμη και αν η αξιολόγισή της έχει ως αποτέλεσμα την εμφάνιση λάθους κατά τον χρόνο εκτέλεσης (run-time error). Για παράδειγμα, η διαίρεση ενός ακεραίου με το μηδέν θα προκαλέσει ένα τέτοιο λάθος.

```
1 / 0
// java.lang.ArithmeticException: / by zero
//   ... 131 elided
```

Η έκφραση `1 / 0` έχει τύπο και μπορούμε να μάθουμε ποιος είναι αυτός, χρησιμοποιώντας την κονσόλα όπως φαίνεται παρακάτω.

```
:type 1 / 0
// Int
```

Ακόμη, ο λόγος εμφάνισης λάθους κατά την εκτέλεση μπορεί να είναι μία υπό-έκφραση μέσα σε μία σύνθετη έκφραση όπως παρακάτω.

```
(2 + (1 / 0) + 3)
// java.lang.ArithmeticException: / by zero
//   ... 157 elided
```

Ακόμη και αυτή η έκφραση έχει τύπο.

```
:type (2 + (1 / 0) + 3)
// Int
```

2.4 Ασκήσεις

2.4.0.1 Αριθμητική

Γράψτε μία έκφραση η οποία αποτελείται από ακεραίους, πρόσθεση, αφαίρεση και αξιολογείται με την τιμή 42.

See the solution

2.4.0.2 Ενώνοντας Strings

Ενώστε δύο strings (τεχνική γνωστή ως *appending* strings) χρησιμοποιώντας την μέθοδο `++`. Γράψτε ισοδύναμες εκφράσεις χρησιμοποιώντας τον κλασικο τροπο με κλήση μεθόδου αλλά και με την χρήση τελεστή.

See the solution

2.4.0.3 Προτεραιότητα

Στα μαθηματικά έχουμε μάθει ότι κάποιοι τελεστές έχουν *προτεραιότητα* έναντι κάποιων άλλων. Για παράδειγμα, στην έκφραση `1 + 2 * 3` ο πολλαπλασιασμός θα πρέπει να γίνει πριν την πρόσθεση. Ισχύουν οι ίδιοι κανόνες στην Scala;

See the solution

2.4.0.4 Τύποι και Τιμές

Ποιες από τις παρακάτω εκφράσεις δεν θα μεταγλωττιστούν; Ποίος είναι ο τύπος αυτών που θα μεταγλωττιστούν; Ποιες εκφράσεις θα αποτύχουν κατά τον χρόνο εκτέλεσης;

```
1 + 2
```

```
"3".toInt
```

```
"Electric blue".toInt
```

```
"Electric blue".take(1)
```

```
"Electric blue".take("blue")
```

```
1 + ("Moonage daydream".indexOf("N"))
```

```
1 / 1 + ("Moonage daydream".indexOf("N"))
```

```
1 / (1 + ("Moonage daydream".indexOf("N")))
```

See the solution

2.4.0.5 Αστοχίες κινητής υποδιαστολής

Όταν σας συστήσαμε τον τύπο Double, σας είπαμε ότι είναι μία προσέγγιση των πραγματικών αριθμών. Γιατί νομίζετε ότι συμβαίνει αυτό; Σκεφτείτε την αναπαράσταση αριθμών όπως το ⅓ και το π. Πόσος χώρος θα χρειάζονταν για να αναπαρασταθούν αυτοί οι αριθμοί στο δεκαδικό;

See the solution

2.4.0.6 Πέρα από τις Εκφράσεις

Στο τωρινό μας υπολογιστικό μοντέλο υπάρχουν μόνο τρία συστατικά στοιχεία: οι εκφράσεις (το κείμενο του προγράμματος), οι αντίστοιχοι τύποι και οι τιμές (που υπάρχουν μέσα στη μνήμη του υπολογιστή). Είναι όμως αυτό αρκετό; Θα μπορούσαμε να γράψουμε ένα πρόγραμμα που θα χρησιμοποιηθεί στο χρηματιστήριο ή ένα παιχνίδι για υπολογιστή μόνο με αυτό το μοντέλο; Μπορείτε να σκεφτείτε τρόπους επέκτασης αυτού του μοντέλου;

See the solution

3 Δουλεύοντας με Εικόνες

Μέχρι τώρα έχουμε δουλέψει με αριθμούς, με strings καθώς και με άλλα απλά αντικείμενα. Όμως τίποτα από αυτά δεν μας ενθουσίασε ιδιαίτερα. Από εδώ και πέρα θα συγκεντρώσουμε την προσοχή μας στο να εργαζόμαστε με εικόνες και αργότερα και με animations (κινούμενες εικόνες). Οι εικόνες μας προσφέρουν την ευκαιρία να εκφραστούμε δημιουργικά και κάνουν το αποτέλεσμα του προγράμματός μας πιο απτό με έναν τρόπο που άλλες μέθοδοι δεν μπορούν να καταφέρουν.

Για να δημιουργήσουμε γραφικά, Θα χρησιμοποιήσουμε μία βιβλιοθήκη που ονομάζεται Doodle. Σ' αυτό το κεφάλαιο θα μάθουμε τα βασικά αυτής της βιβλιοθήκης.

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

3.1 Εικόνες

Ας ξεκινήσουμε με μερικά απλά σχήματα, προγραμματίζοντας στην κονσόλα όπως και πριν.

```
Image.circle(10)
// res0: doodle.core.Image = Circle(10.0)
```

Τι συμβαίνει εδώ; Το `Image` είναι ένα αντικείμενο και το `circle` μία μέθοδος αυτού του αντικειμένου. Περνάμε μία παράμετρο στο `circle`,

την `10`, η οποία αντιστοιχεί στην ακτίνα του κύκλου που κατασκευάζουμε. Παρατηρήστε τον τύπο του αποτελέσματος —είναι `Image`.

Μπορούμε επίσης απλώς να γράψουμε `circle(10)`, αφού εκτελώντας την κονσόλα μέσα στο Doodle είναι σαν να δίνεται αυτόματα η ικανότητα σε αυτή την μέθοδο αλλά και σε άλλες, να κατασκευάζουν εικόνες.

```
circle(10)
// res1: doodle.core.Image = Circle(10.0)
```

Σχεδιάζουμε τον κύκλο, δηλαδή τον εμφανίζουμε στην οθόνη, καλώντας την μέθοδο `draw`.

```
circle(10).draw
```

Μετά την εκτέλεση της παραπάνω εντολής θα πρέπει να εμφανιστεί ένα παράθυρο όπως φαίνεται στην εικόνα fig. 1.

Figure 1: Ένας κύκλος

Το Doodle υποστηρίζει πολλές “βασικές” εικόνες: κύκλους, ορθογώνια και τρίγωνα. Ας προσπαθήσουμε να ζωγραφίσουμε ένα ορθογώνιο.

```
rectangle(100, 50).draw
```

Το αποτέλεσμα φαίνεται στην εικόνα fig. 2.

Figure 2: Ένα ορθογώνιο

Τέλος, ας προσπαθήσουμε να δημιουργήσουμε το τρίγωνο που φαίνεται στην εικόνα fig. 3.

```
triangle(60, 40).draw
```

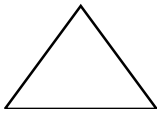


Figure 3: Ένα τρίγωνο

Ασκήσεις

Κάνοντας Κύκλους

Δημιουργήστε κύκλους με πλάτος 1, 10, και 100. Σχεδιάστε τους!

See the solution

Τύποι Σχημάτων

Ποιός είναι ο τύπος ενός κύκλου; Ενός ορθογωνίου; Ενός τριγώνου;

See the solution

Τύποι Σχεδίων

Ποιός είναι ο τύπος του *σχεδίου* μίας εικόνας; Τι σημαίνει αυτό;

See the solution

3.2 Διάταξη

Στην προηγούμενη ενότητα είδαμε πώς μπορούμε να κατασκευάσουμε τα βασικά σχήματα. Αν θέλουμε να δημιουργήσουμε πιο περίπλοκες εικόνες μπορούμε να χρησιμοποιήσουμε μεθόδους διάταξης. Δοκιμάστε τον παρακάτω κώδικα —θα πρέπει να δείτε έναν κύκλο και ένα ορθογώνιο το ένα δίπλα στο άλλο, όπως φαίνεται στην εικόνα fig. 4.

```
(circle(10) beside rectangle(10, 20)).draw
```

Figure 4: Ένας κύκλος δίπλα σε ένα ορθογώνιο

Το Doodle έχει αρκετές μεθόδους διάταξης για συνδυασμό εικόνων. Μπορείτε να τις βρείτε στον πίνακα tbl. 1. Μπορείτε να τις δοκιμάσετε και να δείτε τι κάνουν.

Table 1: Οι διαθέσιμες μέθοδοι διάταξης του Doodle

Operator	Τύπος	Περιγραφή	Παράδειγμα
`Image beside Image`	`Image`	Τοποθετεί τις	`circle(10) beside circle(20)`

		εικόνες οριζόντια την μία δίπλα στην άλλη.	
`Image above Image`	`Image`	Τοποθετεί τις εικόνες κάθετα την μία πάνω από την άλλη.	`circle(10) above circle(20)`
`Image below Image`	`Image`	Τοποθετεί τις εικόνες κάθετα την μία κάτω από την άλλη.	`circle(10) below circle(20)`
`Image on Image`	`Image`	Τοποθετεί τις εικόνες κεντραρισμένες την μία πάνω στην άλλη	`circle(10) on circle(20)`
`Image under Image`	`Image`	Τοποθετεί τις εικόνες κεντραρισμένες την μία κάτω από την άλλη	`circle(10) under circle(20)`

Ασκήσεις

Η Διάμετρος ενός Κύκλου

Φτιάξτε την εικόνα fig. 5 χρησιμοποιώντας τις μεθόδους διάταξης και τα βασικά σχήματα που έχουμε μάθει μέχρι τώρα.

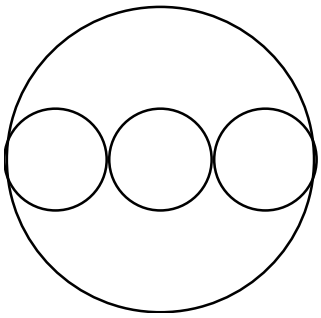


Figure 5: Η διάμετρος ενός κύκλου

See the solution

3.3 Χρώμα

Εκτός από τις μεθόδους διάταξης, το Doodle μας δίνει την δυνατότητα να χρωματίσουμε τις εικόνες μας με την βοήθεια απλών τελεστών. Δοκιμάστε τις μεθόδους που περιγράφονται στον πίνακα tbl. 2 για να δείτε τι κάνουν.

Table 2: Μερικές από τις μεθόδους χρωματισμού στο Doodle.

Τελεστής	Τύπος	Περιγραφή	Παράδειγμα
`Image fillColor Color`	`Image`	Γεμίζει την εικόνα με το χρώμα της επιλογής μας.	`circle(10) fillColor Color.red`
`Image lineColor Color`	`Image`	Χρωματίζει το περίγραμμα της εικόνας.	`circle(10) lineColor Color.blue`
`Image lineWidth Int`	`Image`	Ορίζει το πλάτος του περιγράμματος μίας εικόνας.	`circle(10) lineWidth 3`

Το Doodle δημιουργεί χρώματα με διάφορους τρόπους. Ο πιο απλός, είναι η χρήση των προκαθορισμένων χρωμάτων που υπάρχουν στο CommonColors.scala. Στον παρακάτω πίνακα tbl. 3 περιγράφονται

μερικά από αυτά που χρησιμοποιούνται πιο συχνά .

Table 3: Μερικά από τα προκαθορισμένα χρώματα που χρησιμοποιούνται συχνότερα.

Χρώμα	Τύπος	Παράδειγμα
`Color.red`	`Color`	`circle(10) fillColor Color.red`
`Color.blue`	`Color`	`circle(10) fillColor Color.blue`
`Color.green`	`Color`	`circle(10) fillColor Color.green`
`Color.black`	`Color`	`circle(10) fillColor Color.black`
`Color.white`	`Color`	`circle(10) fillColor Color.white`
`Color.gray`	`Color`	`circle(10) fillColor Color.gray`
`Color.brown`	`Color`	`circle(10) fillColor Color.brown`

Ασκήσεις

Το Κακό Μάτι

Φτιάξτε την εικόνα fig. 6. Είναι σχεδιασμένη έτσι ώστε να μοιάζει με παραδοσιακό φυλακτό προστασίας από το κακό μάτι. Εμείς για να την φτιάξουμε χρησιμοποιήσαμε το χρώμα `cornflowerBlue` για την ίριδα και το `darkBlue` για τον εξωτερικό κύκλο αλλά εσείς μπορείτε να πειραματιστείτε και με δικές σας επιλογές χρωμάτων!

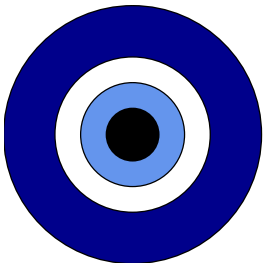


Figure 6: Φυλαχτό!

See the solution

3.4 Δημιουργώντας Χρώματα

Στην προηγούμενη ενότητα είδαμε πώς μπορούμε να χρησιμοποιούμε τα προκαθορισμένα χρώματα στις εικόνες μας. Αλλά τι γίνεται αν θέλουμε να χρησιμοποιήσουμε και άλλα χρώματα εκτός από αυτά; Σε αυτή την ενότητα θα δούμε πώς μπορούμε να δημιουργήσουμε δικά μας χρώματα και πώς να μετατρέψουμε τα ήδη υπάρχοντα σε νέα.

3.4.1 Χρώματα RGB

Οι υπολογιστές χρησιμοποιούν χρώματα που φτιάχνονται συνδυάζοντας διαφορετικές ποσότητες κόκκινου, πράσινου και μπλε. Αυτό το μοντέλο ονομάζεται “RGB” και είναι ένα προσθετικό μοντέλο (additive model) χρωμάτων. Κάθε ένα από τα συστατικά του στοιχεία, δηλαδή το κόκκινο, το πράσινο και το μπλε, μπορούν να πάρουν μία τιμή από το μηδέν ως το 255. Αν δοθεί η μέγιστη τιμή και στα τρία, δηλαδή το 255, το αποτέλεσμα του συνδυασμού τους είναι το καθαρό λευκό. Αν τους δοθεί η τιμή μηδέν, τότε θα προκύψει το μαύρο.

Μπορούμε να δημιουργήσουμε τα δικά μας χρώματα RGB χρησιμοποιώντας την μέθοδο `rgb` στο αντικείμενο `Color`. Αυτή η μέθοδος παίρνει τρεις παραμέτρους: το κόκκινο, το πράσινο και το μπλε. Αυτές οι παράμετροι είναι αριθμοί από το 0 ως το 255 και ονομάζονται `UnsignedBytes`1. Δεν υπάρχει κυριολεκτική έκφραση για το `UnsignedByte` όπως υπάρχει για το `Int` και έτσι πρέπει να μετατρέψουμε τον `Int` σε `UnsignedByte`. Αυτό μπορούμε να το κάνουμε χρησιμοποιώντας την μέθοδο `uByte`. Ένας `Int` μπορεί να πάρει μεγαλύτερες τιμές από ότι ένας `UnsignedByte`, έτσι αν ο αριθμός είναι πολύ μικρός ή πολύ μεγάλος για να

αναπαρασταθεί ως `UnsignedByte`, τότε θα μετατραπεί στην κοντινότερη τιμή από 0 ως 255. Στα παρακάτω παραδείγματα μπορείτε να δείτε μερικές τέτοιες μετατροπές.

```
0.uByte
// res0: doodle.core.UnsignedByte = UnsignedByte(-12
8)

255.uByte
// res1: doodle.core.UnsignedByte = UnsignedByte(127
)

128.uByte
// res2: doodle.core.UnsignedByte = UnsignedByte(0)

-100.uByte // Πολύ μικρό, μετατρέπεται σε 0
// res3: doodle.core.UnsignedByte = UnsignedByte(-12
8)

1000.uByte // Πολύ μεγάλο, μετατρέπεται σε 255
// res4: doodle.core.UnsignedByte = UnsignedByte(127
)
```

(Παρατηρήστε ότι το `UnsignedByte` είναι ένα στοιχείο του Doodle. Δεν είναι κάτι που παρέχει η ίδια η Scala.)

Τώρα που γνωρίζουμε πώς να φτιάξουμε `UnsignedBytes` μπορούμε να δημιουργήσουμε και χρώματα RGB.

```
Color.rgb(255.uByte, 255.uByte, 255.uByte) // Άσπρο
Color.rgb(0.uByte, 0.uByte, 0.uByte) // Μαύρο
Color.rgb(255.uByte, 0.uByte, 0.uByte) // κόκκινο
```

3.4.2 Χρώματα HSL

Η χρήση των χρωμάτων RGB δεν είναι πολύ εύκολη. Η αναπαράσταση χρωμάτων HSL- hue-saturation-lightness (απόχρωση- κορεσμός-

φωτεινότητα) είναι πιο κοντά στον τρόπο με τον οποίο αντιλαμβανόμαστε τα χρώματα. Σε αυτή την αναπαράσταση ένα χρώμα αποτελείται από:

- το *hue*, από 0 εώς 360, που είναι η γωνία περιστροφής στον τροχό των χρωμάτων
- το *saturation*, από 0 εώς 1, είναι ο αριθμός που δίνει την ένταση του χρώματος, από βαθύ γκρι μέχρι καθαρό χρώμα
- το *lightness*, μεταξύ 0 και 1, δίνει στο χρώμα την φωτεινότητά του, από μαύρο μέχρι καθαρό λευκό

Η εικόνα fig. 7 δείχνει πώς διαφέρουν τα χρώματα καθώς αλλάζουμε την απόχρωση (hue) και την φωτεινότητα (lightness) και η εικόνα fig. 8 δείχνει το πώς επηρεάζει η αλλαγή του κορεσμού (saturation).

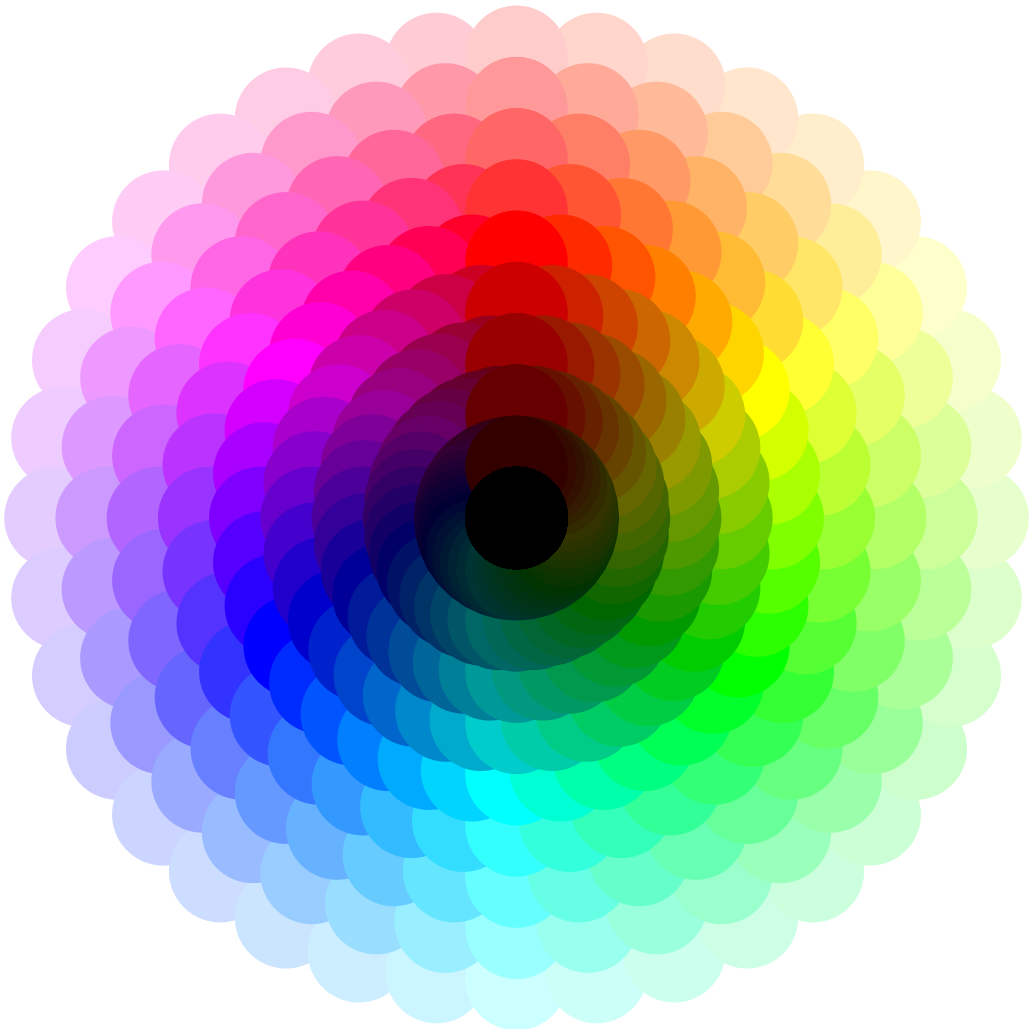


Figure 7: Ένας τροχός χρωμάτων που δείχνει τις αλλαγές στην απόχρωση (περιστροφές) και την φωτεινότητα (απόσταση από το κέντρο), όταν ο κορεσμός έχει σταθερή τιμή 1.

Figure 8: Εικόνα που δείχνει πώς επηρεάζεται το χρώμα καθώς αλλάζει ο κορεσμός, όταν η απόχρωση και η φωτεινότητα μένουν σταθερές. Ο κορεσμός στα αριστερά είναι μηδέν και στα δεξιά είναι ένα.

Μπορούμε να κατασκευάσουμε ένα χρώμα στην αναπαράσταση HSL χρησιμοποιώντας την μέθοδο `Color.hsl`. Αυτή η μέθοδος παίρνει ως παραμέτρους το hue, το saturation, και το lightness. Η αριθμητική τιμή του hue δηλώνει τις μοίρες μίας γωνίας, άρα ο τύπος του είναι `Angle`. Μπορούμε να μετατρέψουμε έναν `Double` σε `Angle` χρησιμοποιώντας τις μεθόδους `degrees` ή `radians`.

```
0.degrees
// res8: doodle.core.Angle = Angle(0.0)

180.degrees
// res9: doodle.core.Angle = Angle(3.141592653589793
)

3.14.radians
// res10: doodle.core.Angle = Angle(3.14)
```

Ο κορεσμός και η φωτεινότητα είναι κανονικοποιημένες τιμές μεταξύ του 0.0 και του 1.0. Μπορούμε να μετατρέψουμε ένα `Double` σε κανονικοποιημένη τιμή χρησιμοποιώντας την μέθοδο `.normalized`.

```
0.0.normalized
// res11: doodle.core.Normalized = Normalized(0.0)

1.0.normalized
// res12: doodle.core.Normalized = Normalized(1.0)

1.2.normalized // Πολύ μεγάλο, μετατρέπεται σε 1.0
// res13: doodle.core.Normalized = Normalized(1.0)
```

```
-1.0.normalized // Πολύ μικρό, μετατρέπεται σε 0.0
// res14: doodle.core.Normalized = Normalized(0.0)
```

Τώρα, είμαστε έτοιμοι να δημιουργήσουμε χρώματα χρησιμοποιώντας την αναπαράσταση HSL.

```
Color.hsl(0.degrees, 0.8.normalized, 0.6.normalized)
 // Ένα παστέλ κόκκινο
```

Για να δούμε αυτό το χρώμα, μπορούμε να το χρησιμοποιήσουμε σε μία εικόνα. Για παράδειγμα, δείτε την εικόνα fig. 9.

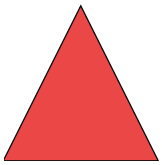


Figure 9: Χρωματίζοντας ένα τρίγωνο κόκκινο παστέλ

3.4.3 Χειρισμός Χρωμάτων

Η εντύπωση που κάνει μία σύνθετη εικόνα, πολύ συχνά εξαρτάται τόσο από τον συνδυασμό των χρωμάτων όσο και από τα ίδια τα χρώματα που χρησιμοποιούνται. Υπάρχουν διάφορες μέθοδοι οι οποίες μας επιτρέπουν να δημιουργήσουμε ένα νέο χρώμα από ένα που υπάρχει ήδη. Αυτές που χρησιμοποιούνται πιο συχνά είναι:

- η `spin`, η οποία αλλάζει την απόχρωση σύμφωνα με κάποια `Angle`;
- η `saturate` και η `desaturate`, οι οποίες προσθέτουν ή αφαιρούν αντίστοιχα μία `Normalised` τιμή από ένα χρώμα, και
- η `lighten` και η `darken`, οι οποίες προσθέτουν και αφαιρούν αντίστοιχα μία `Normalised` τιμή από την φωτεινότητα ενός χρώματος.

Για παράδειγμα ο παρακάτω κώδικας,

```
((circle(100) fillColor Color.red) beside
  (circle(100) fillColor Color.red.spin(15.degrees))
 beside
    (circle(100) fillColor Color.red.spin(30.degrees
))).lineWidth(5.0)
```

παράγει την εικόνα fig. 10.

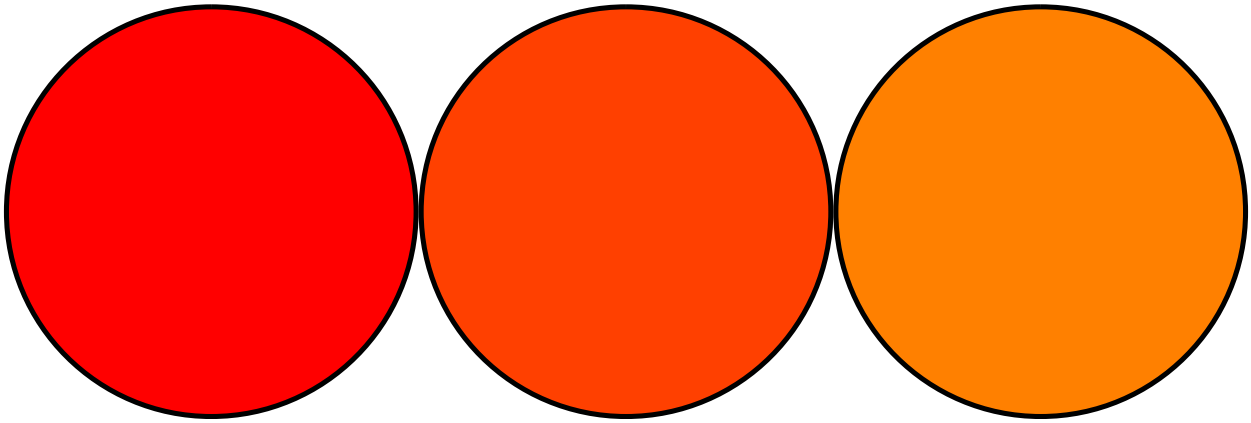


Figure 10: Τρεις κύκλοι. Το χρώμα του πρώτου είναι Color.red και στους διαδοχικούς η απόχρωση αλλάζει κατά γωνία 15 μοιρών

Στην εικόνα fig. 11 μπορείτε να δείτε ένα παρόμοιο παράδειγμα αλλά αυτή τη φορά για τον κορεσμό και την φωτεινότητα.

```
(((circle(20) fillColor (Color.red darken 0.2.normal
ized))
  beside (circle(20) fillColor Color.red)
  beside (circle(20) fillColor (Color.red lighten 0.
2.normalized))) above
((rectangle(40,40) fillColor (Color.red desaturate 0
.6.normalized))
  beside (rectangle(40,40) fillColor (Color.red desa
turate 0.3.normalized))
  beside (rectangle(40,40) fillColor Color.red)))
```

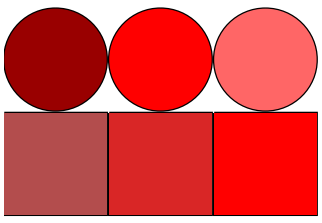


Figure 11: Οι τρεις κύκλοι δείχνουν την αλλαγή της φωτεινότητας και τα τρία τετράγωνα την αλλαγή του κορεσμού.

3.4.4 Διαφάνεια

Μπορούμε να ορίσουμε στα χρώματά μας έναν βαθμό διαφάνειας,

δίνοντάς τους μία τιμή *alpha*. Όταν αυτή οριστεί στο 0.0, αναπαριστά ένα εντελώς διάφανο χρώμα, ενώ αν της δοθεί η τιμή 1.0 το χρώμα είναι εντελώς αδιαφανές. Οι μέθοδοι `Color.rgba` και `Color.hsla` παίρνουν και μία τέταρτη παράμετρο, η οποία είναι μία `Normalised` τιμή alpha. Ακόμη, μπορούμε να δημιουργήσουμε ένα νέο χρώμα με διαφορετικό βαθμό διαφάνειας χρησιμοποιώντας την μέθοδο `alpha` σε ένα χρώμα. Δείτε ένα παράδειγμα στην εικόνα fig. 12 για να καταλάβετε.

```
((circle(40) fillColor (Color.red.alpha(0.5.normaliz
ed))) beside
 (circle(40) fillColor (Color.blue.alpha(0.5.normali
zed))) on
 (circle(40) fillColor (Color.green.alpha(0.5.normal
ized))))
```

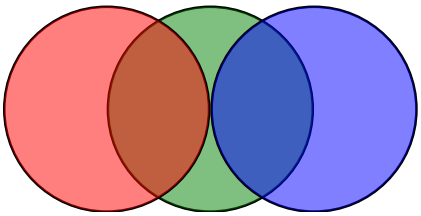


Figure 12: Κύκλοι όπου το alpha είναι 0.5

Ασκήσεις

Συμπληρωματικά Τρίγωνα

Δημιουργήστε τρία τρίγωνα, διατεταγμένα μέσα σε ένα τρίγωνο, έτσι ώστε να έχουν χρώματα παρόμοιας απόχρωσης (hue). Δείτε ένα (αρκετά περίπλοκο) παράδειγμα στην εικόνα fig. 13.

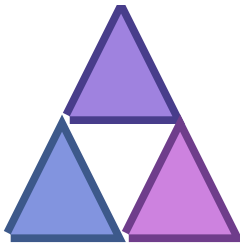


Figure 13: Συμπληρωματικά τρίγωνα. Τα χρώματα που επιλέχθηκαν είναι παραλλαγές του `darkSlateBlue`

See the solution

3.5 Ασκήσεις

3.5.1 Σύνθετος Στόχος

Δημιουργήστε με γραμμές μία εικόνα ενός στόχου τοξοβολίας με τρεις ομόκεντρες ζώνες, όπως φαίνεται στην εικόνα fig. 14.

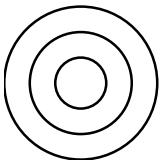


Figure 14: Απλός στόχος τοξοβολίας

Για περισσότερη εξάσκηση προσθέστε ένα στήριγμα ώστε να μπορούμε να τοποθετήσουμε τον στόχο σε οποιαδήποτε απόσταση θέλουμε, όπως φαίνεται στην εικόνα fig. 15.

Figure 15: Χρωματιστός στόχος τοξοβολίας

See the solution

3.5.2 Μείνετε στον Στόχο

Χρωματίστε τον στόχο κόκκινο και άσπρο. Αν φτιάξατε το στήριγμα, χρωματίστε το καφέ. Χρωματίστε επίσης μία πράσινη περιοχή η οποία θα αναπαριστά το έδαφος πάνω στο οποίο στέκεται ο στόχος όπως φαίνεται στην εικόνα fig. 16.

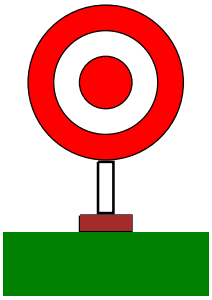


Figure 16: Στόχος τοξοβολίας με στήριγμα

See the solution

1. Το byte είναι ένας αριθμός με 256 πιθανές τιμές και χρειάζεται χώρο 8 bits για να αναπαρασταθεί σε υπολογιστή. Ένα signed byte παίρνει ακέραιες τιμές από το -128 μέχρι το 127, ενώ ένα unsigned byte από το 0 ως το 255. ↩

4 Γράφοντας Μεγαλύτερα Προγράμματα

Έχουμε φτάσει στο σημείο που το να γράφουμε προγράμματα στην κονσόλα δεν είναι πλέον βολικό. Σ' αυτό το κεφάλαιο θα μάθουμε δύο εργαλεία που θα μας βοηθήσουν να γράφουμε μεγαλύτερα προγράμματα:

- την αποθήκευση προγραμμάτων σε αρχεία ώστε να μην χρειάζεται να γράφουμε κώδικά ξανά και ξανά,
- την απόδοση ονομάτων σε τιμές ώστε να μπορούμε να τις επαναχρησιμοποιήσουμε.

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

4.1 Δουλεύοντας στην Κονσόλα

Ο text editor ή το IDE σας, επιτρέπουν την αποθήκευση κώδικα σε αρχεία, τα οποία όμως θα πρέπει να βρίσκονται σε σημεία που να μπορεί να το βρει ο μεταγλωττιστής της Scala. Αν δουλεύετε στο template του Doodle, θα πρέπει να αποθηκεύσετε τον κώδικά σας στον κατάλογο `src/main/scala/`.

Πώς μπορούμε να χρησιμοποιήσουμε στην κονσόλα, κώδικα που έχουμε αποθηκεύσει σε αρχείο; Υπάρχει μία ειδική εντολή, η οποία δουλεύει

μόνο στην κονσόλα. Ονομάζεται `:paste`[1] και ακολουθείται από το όνομα του αρχείου που θέλουμε να εκτελέσουμε. Για παράδειγμα, αν στο αρχείο `src/main/scala/Example.scala` αποθηκεύσουμε την έκφραση

```
circle(100) fillColor Color.paleGoldenrod lineColor
Color.indianRed
```

τότε μπορούμε να εκτελέσουμε αυτόν τον κώδικα γράφοντας

```
:paste src/main/scala/Example.scala
// res0: doodle.core.Image = ContextTransform(<funct
ion1>,ContextTransform(<function1>,Circle(100.0)))
```

Παρατηρήστε ότι στο παραπάνω παράδειγμα, δόθηκε το όνομα `res0` στο αποτέλεσμα. Αν γράφετε και εσείς τα παραδείγματα του βιβλίου στην κονσόλα, το όνομα αυτό μπορεί να έχει διαφορετικό αριθμό ανάλογα με το τι έχετε γράψει προηγουμένως. Μπορούμε να δημιουργήσουμε την εικόνα αξιολογώντας το `res0.draw` (ή το αντίστοιχο όνομα στην δική σας κονσόλα).

4.1.1 Συμβουλές για την Χρήση της Κονσόλας

Παρακάτω μπορείτε να δείτε μερικές συμβουλές για πιο παραγωγική χρήση της κονσόλας:

- Αν πατήσετε το πάνω βελάκι στο πληκτρολόγιο θα εμφανιστεί το τελευταίο πράγμα που γράψατε στην κονσόλα. Αυτό, είναι για παράδειγμα χρήσιμο ώστε να μην πληκτρολογείτε κάθε φορά μεγάλα ονόματα αρχείων! Μπορείτε να πατήσετε πολλές φορές ώστε να δείτε όλο το ιστορικό των εντολών σας στην κονσόλα.
- Μπορείτε να πατήσετε το πλήκτρο `Tab` ώστε να λάβετε προτάσεις για συμπλήρωση του κώδικα που γράφετε. Για παράδειγμα, αν πληκτρολογήσετε `Stri` και μετά πατήσετε το `Tab`, η κονσόλα θα εμφανίσει πιθανές εντολές. Πληκτρολογείστε `Strin` και η κονσόλα θα το συμπληρώσει σε `String`. Δυστυχώς τα ονόματα αρχείων αποτελούν εξαίρεση και δεν συμπληρώνονται.

> Αν γράψουμε κώδικα σε αρχείο, θα παρατηρήσουμε ότι την επόμενη φορά που θα ξεκινήσουμε το SBT, ο μεταγλωττιστής θα μας εμφανίσει ένα μήνυμα λάθους. Διαβάστε την επόμενη ενότητα για να δείτε πώς μπορεί να διορθωθεί αυτό το πρόβλημα.

4.2 Προγραμματίζοντας Εκτός Κονσόλας

Ο κώδικας που γράψαμε στην κονσόλα θα δημιουργήσει προβλήματα αν εκτελεστεί κάπου αλλού. Για παράδειγμα, βάλτε τον παρακάτω κώδικα μέσα στο `Example.scala` στον κατάλογο `src/main/scala`.

```
Image.circle(100) fillColor Color.paleGoldenrod line
Color Color.indianRed
```

Τώρα επανεκκινήστε το SBT και προσπαθήστε να εισέλθετε και πάλι στην κονσόλα. Λογικά, θα δείτε ένα μήνυμα λάθους παρόμοιο με το παρακάτω

```
[error] src/main/scala/Example.scala:1: expected cla
ss or object definition
[error] circle(100) fillColor Color.paleGoldenrod li
neColor Color.indianRed
[error] ^
[error] one error found
```

Αν χρησιμοποιείτε κάποιο IDE, θα συμβεί και εκεί κάτι παρόμοιο.

Το πρόβλημα είναι το εξής:

- Η Scala προσπαθεί να μεταγλωττίσει όλο τον κώδικά μας πριν ξεκινήσει η κονσόλα, και
- υπάρχουν κάποιοι περιορισμοί που ισχύουν για κώδικα που είναι γραμμένος σε αρχεία αλλά όχι για αυτόν που γράφεται απευθείας στην κονσόλα.

Πρέπει να ξέρουμε αυτούς τους περιορισμούς και να αλλάξουμε τον

τρόπο που γράφουμε κώδικα σε αρχεία, αναλόγως.

Το μήνυμα λάθους μας δίνει ένα στοιχείο: `expected class or object definition` (αναμένονταν ορισμός κλάσης ή αντικειμένου). Δεν γνωρίζουμε ακόμη τι είναι μία κλάση αλλά γνωρίζουμε για τα αντικείμενα —όλες οι τιμές είναι αντικείμενα. Στην Scala, ο κώδικας που βρίσκεται σε ένα αρχείο πρέπει να γραφτεί μέσα σε ένα αντικείμενο ή σε μία κλάση. Μπορούμε να ορίσουμε ένα αντικείμενο εύκολα, φτιάχνοντας μία έκφραση όπως την παρακάτω.

```
object Example {
  (circle(100) fillColor Color.paleGoldenrod lineCol
or Color.indianRed).draw
}
```

Τώρα ο κώδικας δεν θα μεταγλωττιστεί για έναν άλλο λόγο. Θα δείτε πολλά λάθη όπως τα παρακάτω

```
[error] doodle/shared/src/main/scala/doodle/examples
/Example.scala:2: not found: value circle
[error]   (circle(100) fillColor Color.paleGoldenrod
 lineColor Color.indianRed).draw
[error]    ^
```

Ο μεταγλωττιστής λέει ότι έχουμε χρησιμοποιήσει ένα όνομα, το `circle` αλλά δεν ξέρει σε ποια τιμή αναφέρεται. Το πρόβλημα είναι παρόμοιο με αυτό που προέκυψε για το `Color` στο παραπάνω κομμάτι κώδικα. θα μιλήσουμε με περισσότερες λεπτομέρειες για τα ονόματα σε λίγο. Τώρα όμως ας πούμε στον μεταγλωττιστή πού μπορεί να βρει τις τιμές γι' αυτά τα ονόματα προσθέτοντας μερικά `imports`. Το όνομα `Color` βρίσκεται μέσα σε ένα *package* που ονομάζεται `doodle.core` και το `circle` μέσα στο αντικείμενο `Image` του `doodle.core`. Μπορούμε να πούμε στον μεταγλωττιστή να χρησιμοποιήσει όλα τα ονόματα του `doodle.core` και όλα τα ονόματα του αντικείμενου `Image`, γράφοντας

```
import doodle.core._
import doodle.core.Image._
```

Υπάρχουν και μερικά ακόμη ονόματα που πρέπει να βρει ο μεταγλωττιστής ώστε να δουλέψει ο κώδικας. Μπορούμε να τα εισάγουμε με αυτές τις γραμμές

```
import doodle.syntax._
import doodle.jvm.Java2DFrame._
import doodle.backend.StandardInterpreter._
```

Θα πρέπει να τοποθετήσουμε όλα τα imports στο πάνω μέρος του αρχείου. Ο τελικός κώδικας θα μοιάζει με τον παρακάτω

```
import doodle.core._
import doodle.core.Image._
import doodle.syntax._
import doodle.jvm.Java2DFrame._
import doodle.backend.StandardInterpreter._

object Example {
  (circle(100) fillColor Color.paleGoldenrod lineCol
or Color.indianRed).draw
}
```

Τώρα θα πρέπει να μεταγλωττιστεί χωρίς προβλήματα.

Όταν επιστρέψουμε στην κονσόλα μέσα στο SBT, μπορούμε να αναφερθούμε στον κώδικά μας, χρησιμοποιώντας το όνομα `Example` που του δώσαμε προηγουμένως.

```
Example // δημιουργεί την εικόνα
```

Άσκηση

Αν δεν το έχετε κάνει ήδη, αποθηκεύστε τον παραπάνω κώδικα στο αρχείο `src/main/scala/Example.scala`. Ελέγξτε ότι μεταγλωττίζεται και ότι η πρόσβαση σε αυτόν από την κονσόλα είναι δυνατή.

4.3 Ονόματα

Στην προηγούμενη ενότητα είδαμε πολλές καινούριες έννοιες. Σε αυτήν, θα εξερευνήσουμε μία: την απόδοση ονομάτων σε τιμές.

Χρησιμοποιούμε ονόματα για να αναφερθούμε σε πράγματα. Για παράδειγμα, το όνομα “Professeur Emile Perrot” αναφέρεται σε μία ποικιλία μυρωδικών τριανταφύλλων, ενώ τα “Cherry Parfait” είναι ανθεκτικά σε ασθένειες αλλά δεν έχουν σχεδόν καθόλου άρωμα. Η συζήτηση για το πώς ακριβώς λειτουργεί αυτή η σχέση ονόματος και έννοιας στον προφορικό λόγο είναι πολύ μεγάλη. Οι γλώσσες προγραμματισμού είναι πιο περιορισμένες, κάτι που μας επιτρέπει να είμαστε πιο ακριβείς: τα ονόματα αναφέρονται σε τιμές. Μερικές φορές θα λέμε ότι κάποια ονόματα είναι *συνδεμένα* με τιμές ή ότι ένα όνομα εισάγει μία *σύνδεση*. Από δω και πέρα, αντί να γράφουμε μία τιμή, μπορούμε να χρησιμοποιούμε το όνομά της, αν της έχει αποδοθεί κάποιο. Με άλλα λόγια, ένα όνομα αξιολογείται με την τιμή στην οποία αναφέρεται. Έτσι, γεννιέται το εξής ερώτημα: πώς δίνουμε ονόματα στις τιμές; Υπάρχουν πολλοί τρόποι για να γίνει αυτό στη Scala. Ας δούμε μερικούς.

4.3.1 Κυριολεκτικές Εκφράσεις Αντικειμένων

Έχουμε ήδη δεί ένα παράδειγμα δήλωσης κυριολεκτικής έκφρασης αντικειμένου.

```
object Example {
  (circle(100) fillColor Color.paleGoldenrod lineCol
or Color.indianRed).draw
}
```

Η παραπάνω είναι μία κυριολεκτική έκφραση, όπως άλλες που έχουμε δει, αλλά σε αυτή την περίπτωση δημιουργεί ένα αντικείμενο με το όνομα `Example`. Όταν χρησιμοποιούμε το όνομα `Example` σε ένα πρόγραμμα, αυτό αξιολογείται με το αντικείμενο στο οποίο αντιστοιχεί.

```
Example
// Example.type = Example$@76c39258
```

Δοκιμάστε το μερικές φορές στην κονσόλα. Βλέπετε κάποια διαφορά; Ίσως παρατηρήσατε ότι την *πρώτη* φορά που χρησιμοποιήσατε το όνομα

`Example` δημιουργήθηκε μία εικόνα αλλά τις επόμενες φορές δεν έγινε το ίδιο. Την πρώτη φορά που χρησιμοποιούμε το όνομα ενός αντικειμένου, αυτό αξιολογείται και δημιουργείται. Στις επόμενες χρήσεις του ονόματος, το αντικείμενο υπάρχει ήδη και δεν αξιολογείται ξανά. Μπορούμε να καταλάβουμε ότι αυτή η περίπτωση είναι διαφορετική αφού η έκφραση μέσα στο αντικείμενο καλεί την μέθοδο `draw`. Αν την αντικαταστήσουμε με κάτι όπως το `1 + 1` (ή απλώς αφαιρέσουμε την κλήση της `draw`) δεν θα μπορέσουμε να καταλάβουμε την διαφορά. Θα πούμε περισσότερα γι' αυτό σε επόμενο κεφάλαιο.

Ίσως αναρωτιέστε ποιός είναι ο τύπος του αντικειμένου που μόλις δημιουργήσαμε. Μπορούμε να ρωτήσουμε την κονσόλα.

```
:type Example
// Example.type
```

Ο τύπος του `Example` είναι ο `Example.type`. Καμία άλλη τιμή δεν έχει τον ίδιο.

4.3.2 Δηλώσεις `val`

Η δήλωση μίας κυριολεκτικής έκφρασης αντικειμένου συνδυάζει την δημιουργία αντικειμένου και τον ορισμό ονόματος. Αν είχαμε την δυνατότητα να τα χωρίσουμε, θα μας ήταν χρήσιμο αφού έτσι θα μπορούσαμε να δώσουμε όνομα σε κάποιο προ υπάρχον αντικείμενο. Οι δηλώσεις `val`, μας δίνουν αυτήν ακριβώς την δυνατότητα.

Χρησιμοποιούμε το `val` γράφοντας

```
val <name> = <value>
```

και αντικαθιστώντας το `<name>` και το `<value>` με το όνομα και την αντίστοιχη τιμή με την οποία αξιολογείται. Για παράδειγμα

```
val one = 1
val anImage = Image.circle(100).fillColor(Color.red)
```

Αυτές οι δύο δηλώσεις ορίζουν τα ονόματα `one` και `anImage`. Μπορούμε να χρησιμοποιήσουμε αυτά τα ονόματα για να αναφερθούμε σ' αυτές τις

τιμές αργότερα στον κώδικά μας.

```
one
// res0: Int = 1

anImage
// res1: doodle.core.Image = ContextTransform(doodle
.core.Image$$Lambda$15201/517283623@1ca5004e,Circle(
100.0))
```

4.3.3 Δηλώσεις

Παραπάνω μιλήσαμε για δηλώσεις και ορισμούς. Τώρα θα δούμε τι ακριβώς σημαίνουν αυτοί οι όροι και θα αναλύσουμε σε βάθος τις διαφορές μεταξύ `object` και `val`.

Έχουμε ήδη μιλήσει για εκφράσεις. Είναι ένα μέρος του προγράμματος που αξιολογείται με κάποια τιμή. Μία *δήλωση* ή ένας *ορισμός* είναι ένα άλλο μέρος του προγράμματος αλλά δεν αξιολογείται με κάποια τιμή. Αντί γι' αυτό, οι δηλώσεις δίνουν ένα όνομα σε κάτι—όχι πάντα σε κάποια τιμή, αφού στη Scala, μπορείτε να δηλώσετε και τύπους. Εμείς σ' αυτό το βιβλίο δεν θα αφιερώσουμε πολύ χρόνο σ' αυτό. Τα `object` και `val` είναι δηλώσεις.

Μία συνέπεια του διαχωρισμού των δηλώσεων από τις εκφράσεις, είναι ότι δεν μπορούμε να γράψουμε προγράμματα όπως το παρακάτω:

```
val one = ( val aNumber = 1 )
// <console>:2: error: illegal start of simple expre
ssion
// val one = ( val aNumber = 1 )
//             ^
```

αφού το `val aNumber = 1` δεν είναι έκφραση και άρα δεν αξιολογείται με κάποια τιμή.

Παρόλα αυτά, μπορούμε να γράψουμε:

```
val aNumber = 1
// aNumber: Int = 1

val one = aNumber
// one: Int = 1
```

4.3.4 To Top-Level

Το να δηλώνουμε `object` και `val`, δεν είναι καθόλου ικανοποιητικό αφού και τα δύο δίνουν ονόματα σε τιμές. Γιατί να μην έχουμε μόνο την `val` για δήλωση ονομάτων και το `object` μόνο για να δημιουργεί αντικείμενα χωρίς να τα ονομάζει; Μπορείτε να δηλώσετε μία κυριολεκτική έκφραση αντικειμένου χωρίς όνομα;

See the solution

Η Scala κάνει μία διάκριση μεταξύ αυτού που αποκαλείται *top-level* κώδικας και του υπόλοιπου. Ο κώδικας στο top-level είναι αυτός που δεν περικλείεται από άλλο κώδικα. Με άλλα λόγια είναι κάτι που μπορούμε να γράψουμε μέσα σε ένα αρχείο και η Scala μπορεί να το μεταγλωττίσει χωρίς να το βάλει μέσα σε `object`.

Έχουμε δει ότι οι εκφράσεις δεν επιτρέπονται στο top-level. Ούτε και οι ορισμοί `val`. Όμως, οι κυριολεκτικές εκφράσεις αντικειμένων επιτρέπονται.

Αυτή η διάκριση είναι λίγο ενοχλητική. Υπάρχουν γλώσσες που δεν έχουν αυτόν τον περιορισμό. Η Scala τον έχει επειδή βασίζεται στο Java Virtual Machine (JVM), το οποίο σχεδιάστηκε για να εκτελεί κώδικα Java. Η Java είναι μια από τις γλώσσες που κάνουν την διάκριση μεταξύ top-level και του υπόλοιπου κώδικα και έτσι η Scala είναι υποχρεωμένη να κάνει και αυτή το ίδιο ώστε να μπορεί να δουλέψει με το JVM. Η κονσόλα της Scala *δεν* κάνει αυτή τη διάκριση (για παράδειγμα ό,τι γράψαμε στην κονσόλα, δεν περικλείονταν από κάποιο αντικείμενο) κάτι που μπορεί να οδηγήσει σε σύγχυση όταν ξεκινάμε την χρήση της.

Αν μία κυριολεκτική έκφραση αντικειμένου επιτραπεί στο top-level αλλά ένας ορισμός `val` όχι, τότε μπορούμε να δηλώσουμε μία `val` μέσα σε μία

κυριολεκτική έκφραση αντικειμένου; Αν μπορούμε να δηλώσουμε μία `val` μέσα σε μία κυριολεκτική έκφραση αντικειμένου, τότε μπορούμε αργότερα να αναφερθούμε σ' αυτό το όνομα;

See the solution

4.3.5 Εμβέλεια

Εάν κάνατε την τελευταία άσκηση (και την κάνατε, δεν την κάνατε;) θα είδατε ότι ένα όνομα δηλωμένο μέσα σε ένα αντικείμενο, δεν μπορεί να χρησιμοποιηθεί έξω από αυτό χωρίς αναφορά στο αντικείμενο μέσα στο οποίο βρίσκεται. Συγκεκριμένα, αν δηλώσουμε:

```
object Example {
  val hi = "Hi!"
}
// defined object Example
```

δεν μπορούμε να γράψουμε

```
hi
// <console>:28: error: not found: value hi
//         hi
//         ^
```

Πρέπει να πούμε στην Scala να ψάξει το `hi` μέσα στο `Example`.

```
Example.hi
// res5: String = Hi!
```

Λέμε ότι ένα όνομα είναι *ορατό* εκεί όπου μπορεί να χρησιμοποιηθεί χωρίς περισσότερες πληροφορίες και ονομάζουμε *εμβέλεια* αυτού του ονόματος, τα μέρη στα οποία είναι ορατό. Έτσι χρησιμοποιώντας την νέα αυτή φανταχτερή ορολογία, το `hi` δεν είναι ορατό εκτός του `Example` ή αλλιώς η εμβέλεια του `hi` δεν υπερβαίνει το `Example`.

Πώς αναγνωρίζουμε την εμβέλεια ενός ονόματος; Ο κανόνας είναι αρκετά απλός: ένα όνομα είναι ορατό από το σημείο της δήλωσής του μέχρι το τέλος των κοντινότερων αγκυλών (οι αγκύλες είναι η `{` και η `}`). Στο

παραπάνω παράδειγμα, το `hi` βρίσκεται μέσα στις αγκύλες του `Example` και άρα εκεί είναι ορατό. Δεν είναι ορατό πουθενά αλλού.

Μπορούμε να δηλώσουμε κυριολεκτικές εκφράσεις αντικειμένων μέσα σε κυριολεκτικές εκφράσεις αντικειμένων, κάτι που μας επιτρέπει να παρατηρήσουμε καλύτερα τις εμβέλειες. Για παράδειγμα στον παρακάτω κώδικα

```
object Example1 {
  val hi = "Hi!"

  object Example2 {
    val hello = "Hello!"
  }
}
```

το `hi` έχει στην εμβέλειά του το `Example2` (το `Example2` ορίζεται μέσα στις αγκύλες που περικλείουν το `hi`). Όμως η εμβέλεια του `hello` περιορίζεται στο `Example2` και άρα έχει μικρότερη εμβέλεια από το `hi`.

Τι θα συμβεί αν δηλώσουμε ένα όνομα μέσα στην εμβέλεια στην οποία έχει ήδη δηλωθεί; Αυτό είναι γνωστό ως *shadowing (επισκίαση)*. Στον παρακάτω κώδικα, ο ορισμός του `hi` μέσα στο `Example2` επισκιάζει τον ορισμό του `hi` στο `Example1`

```
object Example1 {
  val hi = "Hi!"

  object Example2 {
    val hi = "Hello!"
  }
}
```

Η Scala μας το επιτρέπει αλλά η χρήση του είναι γενικώς κακή ιδέα αφού μπορεί πολύ εύκολα να μας μπερδέψει.

Δεν χρειάζεται να χρησιμοποιήσουμε κυριολεκτικές εκφράσεις αντικειμένων για να δημιουργήσουμε νέες εμβέλειες. Η Scala μας επιτρέπει να δημιουργήσουμε μία νέα εμβέλεια οπουδήποτε,

χρησιμοποιώντας απλώς αγκύλες. Έτσι μπορούμε να γράψουμε:

```
object Example {
  val good = "Good"

  // Δημιουργία νέας εμβέλειας
  {
    val morning = good ++ " morning"
    val toYou = morning ++ " to you"
  }

  val day = good ++ " day, sir!"
}
```

το `morning` (και το `toYou`) είναι δηλωμένα μέσα σε μία νέα εμβέλεια. Δεν υπάρχει τρόπος να αναφερθούμε σ' αυτήν την εμβέλεια από έξω (δεν έχει όνομα) άρα δεν μπορούμε να αναφερθούμε στο `morning` εκτός της εμβέλειας μέσα στην οποία είναι δηλωμένο. Αν είχαμε μυστικά που δεν θέλαμε να τα μάθει το υπόλοιπο πρόγραμμα, αυτός θα ήταν ένας πολύ καλός τρόπος να τα κρύψουμε.

Ο τρόπος με τον οποίο λειτουργούν οι εμφωλευμένες εμβέλειες στην Scala καλείται *lexical scoping*. Το lexical scoping δεν υπάρχει σε όλες τις γλώσσες. Για παράδειγμα, η Ruby και η Python δεν το έχουν, ενώ η Javascript το απέκτησε πρόσφατα. Η γνώμη των συγγραφέων είναι ότι η δημιουργία γλώσσας χωρίς lexical scoping είναι σαν να τρως ένα ματσάκι πιπεριών Γουατεμάλας και μετά να πηγαίνεις στην τουαλέτα χωρίς να πλύνεις τα χέρια σου.

Ασκήσεις

Εξετάστε πόσο καλά κατανοήσατε τα ονόματα και τις εμβέλειες βρίσκοντας την τιμή του `answer` σε κάθε μία από τις παρακάτω περιπτώσεις.

```
val a = 1
val b = 2
val answer = a + b
```

See the solution

```
object One {
  val a = 1

  object Two {
    val a = 3
    val b = 2
  }

  object Answer {
    val answer = a + Two.b
  }
}
```

See the solution

```
object One {
  val a = 5
  val b = 2

  object Answer {
    val a = 1
    val answer = a + b
  }
}
```

See the solution

```
object One {
  val a = 1
  val b = a + 1
  val answer = a + b
}
```

See the solution

```
object One {
```

```
  val a = 1

  object Two {
    val b = 2
  }

  val answer = a + b
}
```

See the solution

```
object One {
  val a = b - 1
  val b = a + 1

  val answer = a + b
}
```

See the solution

4.4 Αφαιρετικότητα

Στην προηγούμενη ενότητα μάθαμε πολλά για τα ονόματα. Αν θέλαμε ως προγραμματιστές να χρησιμοποιήσουμε φανταχτερές λέξεις, θα λέγαμε ότι *τα ονόματα είναι πιο αφηρημένα από τις εκφράσεις*. Αυτή η φράση εξηγεί τι κάνει ο ορισμός ονομάτων, οπότε ας την αναλύσουμε.

Το να αφαιρείς σημαίνει να βγάζεις τις ασήμαντες λεπτομέρειες. Για παράδειγμα οι αριθμοί είναι μία “αφαίρεση”. Ο αριθμός “ένα” δεν βρίσκεται ποτέ στην φύση ως καθαρή έννοια. Είναι πάντα ένα αντικείμενο, όπως ένα μήλο ή ένα αντίτυπο της Creative Scala. Στην αριθμητική, η έννοια των αριθμών μας επιτρέπει να αφαιρέσουμε τις ασήμαντες λεπτομέρειες του εξεταζόμενου αντικειμένου και να χειριστούμε τους αριθμούς ως έχουν.

Παρομοίως, ένα όνομα αντιπροσωπεύει μία έκφραση. Η έκφραση μας λέει πώς να φτιάξουμε μία τιμή. Εάν αυτή η τιμή έχει όνομα, τότε δεν χρειάζεται να ξέρουμε κάτι άλλο για το πώς κατασκευάστηκε. Η έκφραση μπορεί να έχει μια αυθαίρετη πολυπλοκότητα αλλά δεν χρειάζεται να

ενδιαφερθούμε γι' αυτήν όταν χρησιμοποιούμε το όνομά της. Αυτό ακριβώς εννοούμε όταν λέμε ότι τα ονόματα είναι πιο αφηρημένα από τις εκφράσεις. Όποτε έχουμε μία έκφραση, μπορούμε να την αντικαταστήσουμε με το όνομα που αναφέρεται στην ίδια τιμή.

Η έννοια της αφαιρετικότητας προσδίδει ευκολία στην ανάγνωση και στη γραφή του κώδικα. Ας πάρουμε ως παράδειγμα την δημιουργία μίας σειράς από κουτιά όπως φαίνεται στην εικόνα fig. 17.

Figure 17: Έξι κουτιά χρώματος Royal Blue

Μπορούμε να δημιουργήσουμε την εικόνα γράφοντας μόνο μια έκφραση.

```
(
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue) beside
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue) beside
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue) beside
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue) beside
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue)
)
```

Σ' αυτόν τον κώδικα κρύβεται ένα απλό μοίβο που όμως είναι δύσκολο να

διακριθεί. Μπορείτε μετά από μία μόνο ματιά να καταλάβετε ότι όλα τα ορθογώνια είναι ακριβώς ίδια; Αν χρησιμοποιήσουμε την έννοια της αφαίρετικότητας και δώσουμε όνομα στο βασικό κουτί, τότε η ανάγνωση του κώδικα θα γίνει πολύ πιο εύκολη.

```
val box =
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue)

box beside box beside box beside box beside box
```

Τώρα μπορούμε εύκολα να δούμε πώς δημιουργείται το κουτί. Η τελική εικόνα είναι το ίδιο κουτί που επαναλαμβάνεται πέντε φορές.

Ασκήσεις

Τοξοβολία και Πάλι

Ας επιστρέψουμε στον στόχο τοξοβολίας που είχαμε δημιουργήσει σε προηγούμενο κεφάλαιο, όπως φαίνεται στην εικόνα fig. 18.

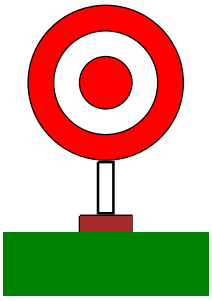


Figure 18: Ο Στόχος Τοξοβολίας

Όταν φτιάξαμε αυτή την εικόνα δεν γνωρίζαμε πώς να δώσουμε ονόματα σε τιμές. Τώρα όμως μπορούμε να γράψουμε μία μεγάλη έκφραση. Αυτή τη φορά, δώστε ονόματα στα στοιχεία της εικόνας ώστε να γίνει πιο εύκολο για κάποιον να καταλάβει πώς κατασκευάστηκε. Αποφασίσετε εσείς ποια μέρη θα πρέπει να ονομαστούν και ποια όχι.

See the solution

Σκηνικό Δρόμου

Για να γίνει η χρήση των ονομάτων πιό ενδιαφέρουσα, κατασκευάστε ένα σκηνικό δρόμου όπως φαίνεται στην εικόνα fig. 19. Ονομάσετε τα διαφορετικά στοιχεία της εικόνας ώστε να αποφύγετε τις πολλές επαναλήψεις.

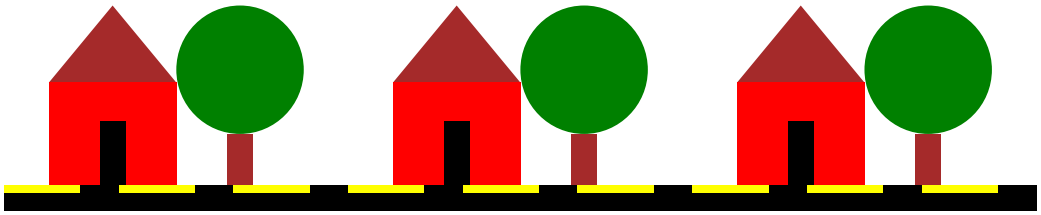


Figure 19: Ένα σκηνικό δρόμου

See the solution

4.5 Packages και Imports

Όταν αλλάξαμε τον κώδικά μας ώστε να μεταγλωττίζεται, χρειάστηκε να προσθέσουμε πολλές δηλώσεις *import*. Σ' αυτή την ενότητα θα μάθουμε γι' αυτές.

Είδαμε ότι ένα όνομα μπορεί να επισκιάσει ένα άλλο. Αυτό μπορεί να προκαλέσει προβλήματα σε μεγαλύτερα προγράμματα αφού πολλά μέρη τους μπορεί να θέλουν να χρησιμοποιήσουν το ίδιο όνομα αλλά για διαφορετικούς σκοπούς. Μπορούμε να δημιουργήσουμε εμβέλειες ώστε να κρύψουμε κάποια ονόματα από τον εξωτερικό κώδικα αλλά και πάλι θα πρέπει να ασχοληθούμε με τα ονόματα που ορίστηκαν στο top-level.

Το ίδιο πρόβλημα αντιμετωπίζουμε και στην φυσική γλώσσα. Για παράδειγμα, αν ο αδερφός σας αλλά και ο φίλος σας ονομάζονταν "Ziggy" θα έπρεπε κάθε φορά που χρησιμοποιείτε αυτό το όνομα να εξηγείτε ποιόν εννοείτε. Ίσως θα μπορούσατε να καταλάβετε από τα συμφραζόμενα ή ίσως ο φίλος σας να ήταν ο "Ziggy S" ενώ ο αδερφός σας ο "Ziggy".

Στην Scala, για να οργανώσουμε τα ονόματα, μπορούμε να χρησιμοποιήσουμε *packages*. Ένα package δημιουργεί μία εμβέλεια για τα ονόματα που έχουν οριστεί στο top-level. Όλα τα ονόματα στο top-level που βρίσκονται στο ίδιο package έχουν την ίδια εμβέλεια. Για να βάλουμε τα ονόματα ενός package σε μία άλλη εμβέλεια, θα πρέπει να τα κάνουμε *import*.

Η δημιουργία ενός package είναι εύκολη: απλώς γράφουμε

```
package <name>
```

στο πάνω μέρος του αρχείου και αντικαθιστούμε το `<name>` με το όνομα του package που μας ενδιαφέρει.

Όταν θέλουμε να χρησιμοποιήσουμε ονόματα που έχουν οριστεί μέσα σε ένα package, τότε χρησιμοποιούμε μία δήλωση `import`, διευκρινίζοντας το όνομα του πακέτου ακολουθούμενου από `_` αν θέλουμε όλα τα ονόματά του ή αλλιώς απλά το όνομα ή τα ονόματα που θέλουμε.

Δείτε ένα παράδειγμα.

Δεν μπορείτε να ορίσετε packages στην κονσόλα. Για να δουλέψει ο παρακάτω κώδικας, θα πρέπει να τον βάλετε στο package `example` μέσα σε ένα αρχείο και να το μεταγλωττίσετε.

Ας ξεκινήσουμε ορίζοντας μερικά ονόματα μέσα σε ένα package.

```
package example

object One {
  val one = 1
}

object Two {
  val two = 2
}

object Three {
  val three = 3
}
```

Τώρα, για να βάλουμε αυτά τα ονόματα σε μία εμβέλεια θα πρέπει να τα κάνουμε import. Αν θέλουμε, μπορούμε να κάνουμε import μόνο ένα όνομα.

```
import example.One
```

```
One.one
```

Ή δύο. Το `One` και το `Two`.

```
import example.{One, Two}

One.one + Two.two
```

Ή όλα τα ονόματα που υπάρχουν στο `example`.

```
import example._

One.one + Two.two + Three.three
```

Στην Scala μπορούμε να κάνουμε import οτιδήποτε ορίζει μία εμβέλεια, συμπεριλαμβανομένων και των αντικειμένων. Ο παρακάτω κώδικας φέρνει το `one` στην εμβέλειά μας.

```
import example.One._

one
```

4.5.1 Οργάνωση των Packages

Τα packages αποτρέπουν την σύγκρουση των ονομάτων που υπάρχουν στο top-level. Τι γίνεται όμως στην περίπτωση που δύο packages έχουν το ίδιο όνομα; Η οργάνωση των packages σε μία ιεραρχία είναι μία κλασική στρατηγική ώστε να αποφευχθούν συγκρούσεις. Για παράδειγμα, στο Doodle το package `core` είναι ορισμένο μέσα στο package `doodle`. Όταν χρησιμοποιούμε την δήλωση

```
import doodle.core._
```

υποδεικνύουμε ότι θέλουμε το package `core` που βρίσκεται μέσα στο package `doodle` και όχι κάποιο άλλο package που μπορεί να ονομάζεται `core`.

---

1. Υπάρχει και η εντολή `:load` που λειτουργεί με έναν ελαφρώς διαφορετικό τρόπο από την `:paste`. Μεταγλωττίζει και εκτελεί κάθε γραμμή του αρχείου ξεχωριστά, ενώ αντίθετα η `:paste` μεταγλωττίζει και εκτελεί ολόκληρο το αρχείο. Οι διαφορές τους είναι σημασιολογικές. Ο τρόπος με τον οποίο λειτουργεί η `:paste` είναι πιο κοντά σ' αυτόν με τον οποίο λειτουργεί ο κώδικας εκτός κονσόλας, οπότε θα χρησιμοποιούμε αυτήν αντί για την `:load`. ↩

5 Το Μοντέλο Αντικατάστασης για Αξιολόγηση

Μέσα στο μυαλό μας, πρέπει να χτίσουμε ένα μοντέλο για τον τρόπο με τον οποίο αξιολογούνται οι εκφράσεις στην Scala ώστε να καταλαβαίνουμε τι ακριβώς κάνουν τα προγράμματά μας. Μέχρι τώρα χρησιμοποιούσαμε ένα αυθαίρετο μοντέλο. Σ' αυτή την ενότητα θα το κάνουμε λίγο πιο συγκεκριμένο. Θα μάθουμε το *μοντέλο αντικατάστασης* για αξιολόγηση. Στον προγραμματισμό συνηθίζεται να χρησιμοποιούμε φανταχτερές λέξεις για απλές έννοιες. Σίγουρα έχετε ακούσει για την αντικατάσταση στο μάθημα της άλγεβρας στο σχολείο. Τώρα, απλώς θα βάλουμε αυτές τις ιδέες που ήδη γνωρίζετε μέσα σε ένα άλλο πλαίσιο.

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

5.1 Αντικατάσταση

Με την μέθοδο της αντικατάστασης, όπου βλέπουμε μία έκφραση μπορούμε να την αντικαταστήσουμε με την τιμή με την οποία αξιολογείται. Για παράδειγμα, όπου βλέπουμε

```
1 + 1
```

μπορούμε να το αντικαταστήσουμε με το `2`. Αυτό με τη σειρά του

σημαίνει ότι όπου βλέπουμε μία σύνθετη έκφραση όπως

```
(1 + 1) + (1 + 1)
```

μπορούμε να αντικαταστήσουμε το `1 + 1` με `2` και να πάρουμε

```
2 + 2
```

που αξιολογείται με την τιμή `4`.

Αυτός ο τρόπος σκέψης, μας είναι γνωστός από την άλγεβρα. Τον χρησιμοποιούσαμε για την απλοποίηση εκφράσεων. Φυσικά, η επιστήμη των υπολογιστών έχει πιο φανταχτερές λέξεις γι' αυτή τη διαδικασία. Εκτός από την αντικατάσταση, μια άλλη ονομασία που χρησιμοποιείται είναι η *απλοποίηση μίας έκφρασης*, ή *εξισωτική συλλογιστική*.

Η αντικατάσταση μας δίνει τον τρόπο με τον οποίο μπορούμε να καταλάβουμε τι κάνουν τα προγράμματά μας. Μπορούμε να εφαρμόσουμε την μέθοδο της αντικατάστασης σχεδόν σε όλες τις εκφράσεις που έχουμε δει μέχρι τώρα. Σ΄αυτή την περίπτωση είναι πιο εύκολο να χρησιμοποιήσουμε παραδείγματα με αριθμούς και strings, παρά με εικόνες, οπότε θα επιστρέψουμε σε ένα παράδειγμα που είδαμε σε προηγούμενο κεφάλαιο:

```
1 + ("Moonage daydream".indexOf("N"))
```

Το παραπάνω παράδειγμα το είχαμε προσπεράσει λίγο πρόχειρα. Τώρα όμως θα δείξουμε τα βήματα που θα κάνει ο υπολογιστής, με μεγαλύτερη ακρίβεια. Έτσι κι αλλιώς, αυτό που προσπαθούμε να κάνουμε είναι να τον μιμηθούμε.

Η έκφραση που περιέχει τον τελεστή `+` αποτελείται από δύο υπό-εκφράσεις, την `1` και την `("Moonage daydream".indexOf("N"))`. Πρέπει να αποφασίσουμε ποια θέλουμε να αξιολογήσουμε πρώτη: την δεξιά ή την αριστερή. Ας διαλέξουμε αυθαίρετα την δεξιά υπό-έκφραση.

Η υπό-έκφραση `("Moonage daydream".indexOf("N"))` αποτελείται και αυτή από δύο υπό-εκφράσεις, την `"Moonage daydream"` και την `"N"`. Ας αξιολογήσουμε και πάλι πρώτα την δεξιά, έχοντας στο μυαλό μας ότι οι κυριολεκτικές εκφράσεις δεν είναι τιμές και άρα πρέπει να αξιολογηθούν.

Η κυριολεκτική έκφραση `"N"` αξιολογείται με την τιμή `"N"`. Για να αποφύγουμε την πιθανή σύγχυση, ας γράψουμε την τιμή ως `|"N"|`. Τώρα μπορούμε να την αντικαταστήσουμε με την έκφραση, κάνοντας έτσι τα πρώτα μας βήματα

```
1 + ("Moonage daydream".indexOf(|"N"|))
```

Έπειτα, μπορούμε να αξιολογήσουμε την αριστερή πλευρά της υπό-έκφρασης, αντικαθιστώντας την κυριολεκτική έκφραση `"Moonage daydream"` με την τιμή της `|"Moonage daydream"|`. Αυτό μας δίνει

```
1 + (|"Moonage daydream"|.indexOf(|"N"|))
```

Τώρα είμαστε σε θέση να αξιολογήσουμε ολόκληρη την έκφραση `(|"Moonage daydream"|.indexOf(|"N"|))`, η οποία αξιολογείται με `|-1|` (και πάλι, μπορείτε να ξεχωρίσετε την ακέραια τιμή από την κυριολεκτική έκφραση, από τις οριζόντιες γραμμές). Εφαρμόζοντας ξανά την μέθοδο της αντικατάστασης έχουμε

```
1 + |-1|
```

Θα πρέπει να αξιολογήσουμε την αριστερή κυριολεκτική έκφραση `1`, που δίνει `|1|`. Κάνοντας αντικατάσταση παίρνουμε

```
|1| + |-1|
```

Τώρα μπορούμε να αξιολογήσουμε ολόκληρη την έκφραση. Το αποτέλεσμα που θα πάρουμε είναι

```
|0|
```

Μπορούμε να ζητήσουμε από την Scala να αξιολογήσει ολόκληρη την έκφραση ώστε να ελέγξουμε την δουλειά μας.

```
1 + ("Moonage daydream".indexOf("N"))
// res4: Int = 0
```

Σωστό!

Σε αυτό το σημείο μπορούμε να κάνουμε κάποιες παρατηρήσεις:

- αν κάνουμε την αντικατάσταση τόσο αυστηρά όσο ο υπολογιστής, μπορεί να χρειαστούν πολλά βήματα
- ο πιο σύντομος υπολογισμός που ίσως κάνατε στο μυαλό σας, μάλλον σας οδήγησε στην σωστή απάντηση
- η φαινομενικά αυθαίρετη επιλογή μας να αξιολογήσουμε από τα δεξιά προς τα αριστερά, μας οδήγησε στην σωστή απάντηση.

Μήπως με κάποιον τρόπο καταφέραμε να χρησιμοποιήσουμε την ίδια σειρά αξιολόγησης που χρησιμοποιεί και η Scala; Δεν έγινε έτσι αλλά αυτό είναι κάτι που δεν έχουμε ερευνήσει ακόμα. Μήπως δεν έχει όντως σημασία η σειρά που επιλέγουμε; Πότε μπορούμε να κάνουμε συντομεύσεις και να πάρουμε το σωστό αποτέλεσμα, όπως κάναμε στο πρώτο παράδειγμα με την πρόσθεση; Θα εξετάσουμε αυτές τις ερωτήσεις σε λίγο αλλά πρώτα ας μιλήσουμε για το πώς χρησιμοποιείται η μέθοδος της αντικατάστασης στα ονόματα.

5.1.1 Ονόματα

Ο κανόνας για αντικατάσταση ονομάτων είναι να αντικαθίσταται το όνομα με την τιμή στην οποία αξιολογείται. Χωρίς να το ξέρουμε, έχουμε ήδη χρησιμοποιήσει αυτόν τον κανόνα. Τώρα απλώς τον επισημοποιούμε.

Για παράδειγμα, στον παρακάτω κώδικα

```
val name = "Ada"
name ++ " " ++ "Lovelace"
```

μπορούμε να εφαρμόσουμε την μέθοδο της αντικατάστασης ώστε να πάρουμε

```
"Ada" ++ " " ++ "Lovelace"
```

που αξιολογείται ως

```
"Ada Lovelace"
```

Για να γίνουμε λίγο πιο τυπικοί με την διαδικασία αντικατάστασης, μπορούμε να χρησιμοποιήσουμε ονόματα. Ας επιστρέψουμε στο πρώτο

μας παράδειγμα

```
1 + 1
```

μπορούμε να δώσουμε ένα όνομα σ' αυτή την έκφραση:

```
val two = 1 + 1
```

Όταν βλέπουμε μία σύνθετη έκφραση όπως η

```
(1 + 1) + (1 + 1)
```

η μέθοδος της αντικατάστασης μας λέει ότι μπορούμε να αντικαταστήσουμε το `1 + 1` με `two` ώστε να πάρουμε

```
two + two
```

Θυμηθείτε ότι όταν εξετάσαμε την έκφραση

```
1 + ("Moonage daydream".indexOf("N"))
```

αναγκαστήκαμε να την χωρίσουμε σε υπό-εκφράσεις, τις οποίες αξιολογήσαμε και στην συνέχεια αντικαταστήσαμε. Αυτή η διαδικασία είναι αρκετά δυσνόητη. Με την χρήση όμως μερικών δηλώσεων `val` μπορούμε να την συμπυκνώσουμε και να την κάνουμε πιο κατανοητή. Παρακάτω μπορείτε να δείτε την ίδια αυτή έκφραση, διασπασμένη στα συστατικά της στοιχεία.

```
val a = 1
val b = "Moonage daydream"
val c = "N"
val d = b.indexOf(c)
val e = a + d
```

Εάν ορίσουμε (αυθαίρετα σε αυτό το σημείο) ότι η αξιολόγηση γίνεται από πάνω προς τα κάτω, τότε μπορούμε να πειραματιστούμε με την σειρά των εντολών ώστε να δούμε ποιες θα είναι οι διαφορές στο αποτέλεσμα.

Για παράδειγμα, ο παρακάτω κώδικας

```
val c = "N"
val b = "Moonage daydream"
val a = 1
val d = b.indexOf(c)
val e = a + d
```

δίνει το ίδιο αποτέλεσμα με πριν. Όμως δεν μπορούμε να χρησιμοποιήσουμε το παρακάτω

```
val e = a + d
val a = 1
val b = "Moonage daydream"
val c = "N"
val d = b.indexOf(c)
```

αφού το `e` εξαρτάται από το `a` και το `d`, και με την από πάνω προς το κάτω σειρά που επιλέξαμε, τα `a` και `d` δεν έχουν αξιολογηθεί ακόμα. Θα μπορούσαμε να ισχυριστούμε ότι είναι χαζό να το επιχειρήσουμε. Το `e` είναι η έκφραση που προσπαθούμε να αξιολογήσουμε αλλά από τα `a`, `b`, `c` και `d` είναι υπό-εκφράσεις του `e`. Φυσικά πρέπει να αξιολογήσουμε πρώτα τις υπό-εκφράσεις πριν αξιολογήσουμε την έκφραση.

5.2 Σειρά Αξιολόγησης

Σ' αυτή την ενότητα θα ασχοληθούμε με την σειρά αξιολόγησης. Ίσως αναρωτιέστε αν όντως έχει σημασία. Στα παραδείγματα που είδαμε μέχρι τώρα, η σειρά δεν φάνηκε να είχε κάποια ιδιαίτερη σημασία, εκτός βέβαια από την περίπτωση στην οποία δεν μπορούσαμε να αξιολογήσουμε μία έκφραση, πριν αξιολογήσουμε τις υπό-εκφράσεις της.

Για να ερευνήσουμε αυτά τα θέματα περισσότερο θα πρέπει να εισάγουμε μία νέα έννοια. Μέχρι εδώ έχουμε ασχοληθεί μόνο με *pure* εκφράσεις. Τέτοιες είναι οι εκφράσεις στις οποίες μπορούμε να κάνουμε αντικαταστάσεις με οποιαδήποτε σειρά 1.

Στις *impure* εκφράσεις, η σειρά αξιολόγησης παίζει καθοριστικό ρόλο. Έχουμε ήδη χρησιμοποιήσει μία τέτοια έκφραση, την μέθοδο `draw`. Αν αξιολογήσουμε το

```
Image.circle(100).draw
Image.rectangle(100, 50).draw
```

και το

```
Image.rectangle(100, 50).draw
Image.circle(100).draw
```

οι εικόνες θα εμφανιστούν με διαφορετική σειρά. Σε αυτό το παράδειγμα δεν φαίνεται η σημασία της διαφοράς.

Το κλειδί για να ξεχωρίζουμε τις impure εκφράσεις είναι ότι η αξιολόγησή τους έχει ως αποτέλεσμα κάποια αλλαγή που μπορούμε να παρατηρήσουμε. Για παράδειγμα, η αξιολόγηση της `draw` έχει ως αποτέλεσμα την εμφάνιση μίας εικόνας. Αυτές τις εύκολα ορατές αλλαγές τις ονομάζουμε *side effects* ή απλώς *effects* για συντομία. Σε ένα πρόγραμμα το οποίο περιέχει side effects δεν μπορούμε να κάνουμε αντικαταστάσεις ελεύθερα. Μπορούμε όμως να χρησιμοποιήσουμε side effects ώστε να ερευνήσουμε την σειρά αξιολόγησης. Το εργαλείο που χρησιμοποιούμε σ' αυτή την περίπτωση είναι η μέθοδος `println`.

Η μέθοδος `println` εμφανίζει κείμενο στην κονσόλα (ένα side effect) και αξιολογείται ως unit. Δείτε ένα παράδειγμα:

```
println("Hello!")
// Hello!
```

Το side-effect του `println`—εμφάνιση κειμένου στην κονσόλα—μας δίνει έναν βολικό τρόπο για να ερευνήσουμε την σειρά αξιολόγησης. Για παράδειγμα, το αποτέλεσμα της εκτέλεσης του παρακάτω κώδικα

```
println("A")
// A

println("B")
// B

println("C")
// C
```

υποδεικνύει ότι οι εκφράσεις αξιολογούνται από πάνω προς τα κάτω. Ας χρησιμοποιήσουμε το `println` για περαιτέρω διερεύνηση.

Ασκήσεις

Αντικαταστάσεις και Println

Σε ένα pure πρόγραμμα, μπορούμε να δώσουμε όνομα σε οποιαδήποτε έκφραση. Όταν την χρησιμοποιήσουμε ξανά, μπορούμε να την αντικαταστήσουμε με το όνομά της. Συνεπώς, μπορούμε να ξαναγράψουμε την έκφραση

```
(2 + 2) + (2 + 2)
```

ως

```
val a = (2 + 2)
a + a
```

και το αποτέλεσμα του προγράμματος δεν αλλάζει.

Χρησιμοποιώντας την `println` βλέπουμε ότι οι impure εκφράσεις, με τα side effects τους, χαλάνε την διαδικασία της αντικατάστασης.

See the solution

Τρέλα στις Μεθόδους μας

Όταν μιλήσαμε για τις εμβέλειες, μιλήσαμε και για εκφράσεις block. Τότε όμως δεν τις είχαμε ονομάσει. Ένα block δημιουργείται με αγκύλες (`{}`) και αξιολογεί όλες τις εκφράσεις μέσα σε αυτές, με το τελικό αποτέλεσμα να είναι αυτό της τελευταίας έκφρασης του.

```
// Αξιολογείται ως 3
{
  val one = 1
  val two = 2
  one + two
}
// res13: Int = 3
```

Μπορούμε να χρησιμοποιήσουμε εκφράσεις block ώστε να ερευνήσουμε την σειρά με την οποία αξιολογούνται οι παράμετροι μεθόδων, βάζοντας την έκφραση `println` μέσα σε ένα block το οποίο αξιολογείται με κάποια άλλη χρήσιμη τιμή.

Η χρήση για παράδειγμα του `Image.rectangle` ή του `Color.hsl` και οι εκφράσεις block, καθορίζουν το αν η Scala θα αξιολογήσει τις παραμέτρους της μεθόδου σε μία συγκεκριμένη σειρά και ποια θα είναι αυτή.

Σημειώστε ότι μπορείτε να γράψετε blocks μικρότερης έκτασης, χωρίζοντας τις εκφράσεις με ερωτηματικά (`;`). Γενικά, αυτός δεν είναι καλός τρόπος αλλά μπορεί να φανεί χρήσιμος στα πειράματά μας. Δείτε ένα παράδειγμα.

```
// Αξιολογείται ως 3
{ val one = 1; val two = 2; one + two }
// res15: Int = 3
```

See the solution

Η Τελευταία Σειρά

Με ποιά σειρά αξιολογούνται οι εκφράσεις στην Scala; Κάντε δικά σας πειράματα ώστε να βρείτε την απάντηση που σας ικανοποιεί. Μπορείτε να συμπεράνετε ότι η Scala εφαρμόζει τους ίδιους κανόνες για όλες τις εκφράσεις. Δεν υπάρχουν ειδικές περιπτώσεις.

See the solution

5.3 Τοπική Συλλογιστική

Είδαμε ότι η σειρά αξιολόγησης είναι σημαντική μόνο όταν υπάρχουν side effects. Για παράδειγμα, αν οι επόμενες εκφράσεις δημιουργούν side effects

```
disableWarheads()
launchTheMissles()
```

θα πρέπει να σιγουρευτούμε ότι αξιολογούνται από πάνω προς τα κάτω

ώστε οι κεφαλές να αφοπλιστούν πριν την εκτόξευση των πυραύλων.

Κάθε χρήσιμο πρόγραμμα πρέπει να έχει κάποιο effect, αφού έτσι αλληλεπιδρά με τον έξω κόσμο. Το effect μπορεί απλώς να τυπώνει κάτι όταν τελειώνει η εκτέλεση του προγράμματος. Ο περιορισμός των side effects αποτελεί στόχο κλειδί στον συναρτησιακό προγραμματισμό συνεπώς θα αφιερώσουμε μερικές λέξεις ακόμη γι' αυτό το θέμα.

Το μοντέλο της αντικατάστασης είναι εύκολα κατανοητό. Όταν η σειρά αξιολόγησης δεν έχει σημασία, τότε κανένας άλλος κώδικας δεν έχει την δυνατότητα να αλλάξει το νόημα του κώδικα που κοιτάμε. Η έκφραση `1 + 1` αξιολογείται πάντα με `2` και ο υπόλοιπος κώδικας του προγράμματός μας δεν μπορεί να το αλλάξει αυτό. Όμως το effect του `launchTheMissles()` εξαρτάται από το αν έχουμε αφοπλίσει ήδη τις κεφαλές ή όχι.

Αυτό έχει ως αποτέλεσμα, ο pure κώδικας να μπορεί να γίνει κατανοητός ακόμη και αποκομμένος από το υπόλοιπο πρόγραμμα. Από την στιγμή που κανένας άλλος κώδικας δεν μπορεί να αλλάξει το νόημά του, αν ενδιαφερόμαστε μόνο για ένα κομμάτι, μπορούμε να αγνοήσουμε όλα τα υπόλοιπα. Από την άλλη, το νόημα του impure κώδικα, εξαρτάται από τον κώδικα που έχει εκτελεστεί πριν από αυτόν. Αυτή η ιδιότητα είναι γνωστή ως *τοπική συλλογιστική* (local reasoning). Ο pure κώδικας την έχει ενώ ο impure όχι.

Καθώς τα προγράμματα γίνονται όλο και μεγαλύτερα, γίνεται πιο δύσκολο για εμάς να συγκρατούμε όλες αυτές τις λεπτομέρειες στο μυαλό μας. Επειδή το κεφάλι μας έχει συγκεκριμένη χωρητικότητα, η μόνη λύση είναι να εισάγουμε την έννοια της αφαιρετικότητας. Θυμηθείτε ότι αυτή η διαδικασία είναι ουσιαστικά η απομάκρυνση ασήμαντων λεπτομερειών. Ο pure κώδικας είναι η απόλυτη αφαίρεση, αφού μας λέει ότι όλα τα υπόλοιπα είναι άσχετες λεπτομέρειες. Αυτή είναι μία από τις ιδιότητες που κάνει τον συναρτησιακό προγραμματισμό συναρπαστικό: η ικανότητα να γίνονται κατανοητά τα μεγάλα προγράμματα. Αυτό δεν σημαίνει ότι ο συναρτησιακός προγραμματισμός αποφεύγει τα effects. Όλα τα χρήσιμα προγράμματα έχουν effects. Σημαίνει όμως, ότι η διαχείρισή τους γίνεται έτσι ώστε το μεγαλύτερο μέρος του κώδικα να μπορεί να συμβαδίσει με την λογική του απλού μοντέλου της αντικατάστασης.

5.3.1 Το Νόημα του Νοήματος

Παραπάνω μιλήσαμε αρκετά για το νόημα του κώδικα, υπονοώντας ότι το “νόημα” είναι το αποτέλεσμα με το οποίο αξιολογείται ο κώδικας και ίσως τα side effects του.

Με άλλα λόγια, το νόημα ενός προγράμματος είναι ακριβώς αυτό στο οποιο αξιολογείται. Συνεπώς, δύο προγράμματα είναι ίσα αν αξιολογούνται στο ίδιο αποτέλεσμα. Αυτός είναι και ο λόγος για τον οποίο όταν υπάρχουν side effects δεν μπορεί να γίνει αντικατάσταση: το μοντέλο της αντικατάστασης δεν έχει επίγνωση των side effects και δεν μπορεί να διακρίνει προγράμματα με διαφορετικά effects.

Υπάρχουν και άλλοι τρόποι με τους οποίους μπορεί να διαφέρουν δύο προγράμματα. Για παράδειγμα, ένα πρόγραμμα μπορεί να χρειαστεί περισσότερο χρόνο από ένα άλλο για να φέρει το ίδιο αποτέλεσμα. Και πάλι, το μοντέλο της αντικατάστασης δεν μπορεί να τα ξεχωρίσει.

Η αντικατάσταση είναι μία αφαίρεση και οι λεπτομέρειες που απομακρύνει είναι οτιδήποτε εκτός από την τιμή. Τα side effects, ο χρόνος και η χρήση της μνήμης, είναι όλα ασήμαντα για την διαδικασία της αντικατάστασης αλλά ίσως δεν ισχύει το ίδιο και για τους ανθρώπους που φτιάχνουν ή τρέχουν το πρόγραμμα. Εδώ γίνεται ένας συμβιβασμός. Μπορούμε να χρησιμοποιήσουμε πιο πλούσια μοντέλα που κρατάν περισσότερες λεπτομέρειες αλλά είναι πολύ πιο δύσκολα στην χρήση τους. Για τους περισσότερους ανθρώπους, η αντικατάσταση αποτελεί μια απλή και χρήσιμη διαδικασία.

---

1. Αυτή δεν είναι όλη η αλήθεια. Υπάρχουν μερικές ειδικές περιπτώσεις στις οποίες η σειρά αξιολόγησης έχει σημασία ακόμη και στις pure εκφράσεις. Δεν θα ασχοληθούμε με αυτές τις περιπτώσεις. Εάν όμως θέλετε να μάθετε περισσότερα, μπορείτε να ψάξετε και να διαβάσετε για την “eager evaluation” και την “lazy evaluation”. ↩

6 Μέθοδοι

Έχουμε ήδη συναντήσει διάφορες μεθόδους και έχουμε μάθει οτι τις χρησιμοποιούμε ώστε να αλληλεπιδράσουμε με τα αντικείμενα. Σ' αυτό το κεφάλαιο θα μάθουμε πώς να γράφουμε τις δικές μας μεθόδους.

Τα ονόματα μας επιτρέπουν να χρησιμοποιήσουμε την μέθοδο της αφαίρεσης στις εκφράσεις. Οι μέθοδοι μας επιτρέπουν να χρησιμοποιήσουμε την μέθοδο της αφαίρεσης και να *γενικεύσουμε* τις εκφράσεις. Με τον όρο γενίκευση εννοούμε την ικανότητα να εκφράσουμε μία ομάδα πραγμάτων σχετικών μεταξύ τους, δηλαδή σ' αυτή την περίπτωση, εκφράσεις. Οι μέθοδοι αποτελούν templates για εκφράσεις και δίνουν την δυνατότητα σε αυτόν που τις καλεί, να περάσει παραμέτρους και να συμπληρώσει κομμάτια αυτού του template.

Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.

```
import doodle.core._
import doodle.core.Image._
import doodle.syntax._
import doodle.jvm.Java2DFrame._
import doodle.backend.StandardInterpreter._
```

6.1 Μέθοδοι

Σε ένα από τα προηγούμενα κεφάλαια, δημιουργήσαμε την εικόνα που φαίνεται στην εικόνα fig. 20 χρησιμοποιώντας το παρακάτω πρόγραμμα

Figure 20: Έξι κουτιά σε χρώμα Royal Blue

```
val box =
  Image.rectangle(40, 40).
    lineWidth(5.0).
    lineColor(Color.royalBlue.spin(30.degrees)).
    fillColor(Color.royalBlue)

box beside box beside box beside box beside box
```

Αν θέλουμε να αλλάξουμε μόνο το χρώμα αυτών των κουτιών, με αυτά που ξέρουμε ως τώρα, για κάθε διαφορετικό χρώμα που επιλέγουμε, θα πρέπει να γράψουμε τον κώδικα από την αρχή.

```
val paleGoldenrod = {
  val box =
    Image.rectangle(40, 40).
      lineWidth(5.0).
      lineColor(Color.paleGoldenrod.spin(30.degrees)
).
      fillColor(Color.paleGoldenrod)

  box beside box beside box beside box beside box
}

val lightSteelBlue = {
  val box =
    Image.rectangle(40, 40).
      lineWidth(5.0).
      lineColor(Color.lightSteelBlue.spin(30.degrees
)).
      fillColor(Color.lightSteelBlue)

  box beside box beside box beside box beside box
}

val mistyRose = {
  val box =
    Image.rectangle(40, 40).
```

```
      lineWidth(5.0).
      lineColor(Color.mistyRose.spin(30.degrees)).
      fillColor(Color.mistyRose)

  box beside box beside box beside box beside box
}
```

Αυτό είναι βαρετό. Οι εκφράσεις για τα διαφορετικά χρώματα διαφέρουν ελάχιστα μεταξύ τους. Θα ήταν πιο βολικό αν είχαμε ένα γενικό ?? μοτίβο?? στο οποίο μπορούμε να αλλάζουμε το χρώμα ενώ ο υπόλοιπος κώδικας παραμένει ίδιος. Οι μέθοδοι, μας δίνουν ακριβώς αυτήν την δυνατότητα.

```
def boxes(color: Color): Image = {
  val box =
    Image.rectangle(40, 40).
      lineWidth(5.0).
      lineColor(color.spin(30.degrees)).
      fillColor(color)

  box beside box beside box beside box beside box
}

// Δημιουργία κουτιών με διαφορετικά χρώματα
boxes(Color.paleGoldenrod)
boxes(Color.lightSteelBlue)
boxes(Color.mistyRose)
```

Δοκιμάστε μόνοι σας και δείτε ότι χρησιμοποιώντας μεθόδους παίρνετε το ίδιο αποτέλεσμα όπως όταν τα γράφατε όλα με το χέρι.

Τώρα που είδαμε ένα παράδειγμα δήλωσης μεθόδου, πρέπει να εξηγήσουμε τη σύνταξή της. Παρακάτω θα δούμε πώς γράφονται οι μέθοδοι, την σημασιολογία των κλήσεών τους και πώς λειτουργούν σε σχέση με την μέθοδο της αντικατάστασης.

6.2 Συντακτικό Μεθόδων

Έχουμε ήδη δει ένα παράδειγμα δήλωσης μεθόδου.

```
def boxes(color: Color): Image = {
  val box =
    Image.rectangle(40, 40).
      lineWidth(5.0).
      lineColor(color.spin(30.degrees)).
      fillColor(color)

  box beside box beside box beside box beside box
}
```

Ας το χρησιμοποιήσουμε ως μοντέλο για να κατανοήσουμε το συντακτικό της δήλωσης μίας μεθόδου. Αρχικά, έχουμε την *λέξη-κλειδί* `def`. Είναι μία ειδική λέξη που υποδεικνύει κάτι σημαντικό στον μεταγλωττιστή της Scala —σ' αυτήν την περίπτωση, ότι θα δηλώσουμε μία μέθοδο. Έχουμε ήδη δει τις λέξεις-κλειδιά `object` και `val`.

Το `def` ακολουθείται από το όνομα της μεθόδου, που στην δική μας περίπτωση είναι το `boxes`. Το ίδιο συνέβαινε και με τα `val` και `object`, που ακολουθούνταν από το όνομα που δηλώνουν. Όπως μία δήλωση `val`, μία δήλωση μεθόδου δεν είναι δήλωση top-level και όταν γράφεται σε αρχείο πρέπει να περικλείεται μέσα σε μία δήλωση `object` (ή σε μία άλλη top-level δήλωση).

Μετά, έχουμε τις παραμέτρους της μεθόδου, οι οποίες ορίζονται μέσα σε παρενθέσεις (`()`). Οι παράμετροι είναι τα κομμάτια εκείνα, που αυτός που κάνει την κλήση μπορεί να "βάλει" μέσα στην έκφραση που αξιολογεί η μέθοδος. Όταν δηλώνουμε παραμέτρους μεθόδων πρέπει να τους δώσουμε ένα όνομα και έναν τύπο. Το όνομα και ο τύπος χωρίζονται από μία άνω-κάτω τελεία (`:`). Μέχρι στιγμής, σ' αυτό το βιβλίο δεν είχε χρειαστεί να δηλώσουμε κάποιον τύπο. Στις περισσότερες περιπτώσεις η Scala μπορεί να βρει τους τύπους από μόνη της. Αυτή Η διαδικασία ονομάζεται *type inference*. Όμως το type inference, δεν μπορεί να συμπεράνει τους τύπους των παραμέτρων μίας μεθόδου οπότε πρέπει να τους δηλώσουμε εμείς.

Μετά τις παραμέτρους, έχουμε τον τύπο του αποτελέσματος. Ο τύπος του αποτελέσματος είναι ο τύπος της τιμής με την οποία αξιολογείται η

μέθοδος όταν καλείται. Αντίθετα με τους τύπους των παραμέτρων, η Scala μπορεί να συμπεράνει τον τύπο του αποτελέσματος. Παρόλα αυτά, είναι καλή τακτική να τον δηλώνουμε από μόνοι μας. Ετσι θα κάνουμε παρακάτω.

Τέλος, η έκφραση που υπάρχει στο σώμα της μεθόδου, υπολογίζει το αποτέλεσμα της κλήσης της. Το σώμα μπορεί να είναι μία block expression, όπως στην `boxes` που είδαμε παραπάνω ή απλώς μία μόνο έκφραση.

Συντακτικό Δήλωσης Μεθόδου

Το συντακτικό μίας δήλωσης μεθόδου είναι το παρακάτω

```
def methodName(param1: Param1Type, ...): ResultType =
  bodyExpression
```

όπου

- το `methodName` είναι το όνομα της μεθόδου,
- τα προαιρετικά `param1 : Param1Type, ...` είναι ένα ή περισσότερα ζεύγη ονομάτων και τύπων παραμέτρων,
- το προαιρετικό `ResultType` είναι ο τύπος του αποτελέσματος που επιστρέφει η μεθόδος, και
- το `bodyExpression` είναι η έκφραση η οποία αξιολογείται ώστε να παραχθεί το αποτέλεσμα κλήσης της μεθόδου.

Ασκήσεις

Ας εξασκηθούμε στην δήλωση μεθόδων γράφοντας μερικά απλά παραδείγματα.

Τετράγωνο

Γράψτε μία μέθοδο `square` η οποία δέχεται έναν `Int` και επιστρέφει το `Int` υψωμένο στο τετράγωνο.

See the solution

Μισό

Γράψτε μία μέθοδο `halve` η οποία δέχεται μία παράμετρο `Double` και επιστρέφει ένα `Double` το οποίο είναι το μισό από αυτή την παράμετρο.

See the solution

6.3 Σημασιολογία Μεθόδων

Τώρα που ξέρουμε πώς να δηλώνουμε μεθόδους, ας στρέψουμε την προσοχή μας στη σημασιολογία τους. Τι γίνεται με τις κλήσεις μεθόδων σε σχέση με το μοντέλο αντικατάστασης;

Ξέρουμε ήδη ότι μπορούμε να αντικαταστήσουμε μία κλήση μεθόδου με την τιμή στην οποία αξιολογείται. Όμως χρειαζόμαστε ένα πιο λεπτομερές μοντέλο ώστε να βρούμε ποια θα είναι αυτή η τιμή. Το επεκταμένο μοντέλο μας είναι το παρακάτω: όταν βλέπουμε μία κλήση μεθόδου θα δημιουργούμε ένα νέο block και μέσα σε αυτό θα συνδέουμε τις παραμέτρους με τις αντίστοιχες εκφράσεις που δίνονται στην κλήση της μεθόδου και θα αντικαθιστούμε το σώμα της μεθόδου.

Μετά μπορούμε να χρησιμοποιήσουμε το μοντέλο της αντικατάστασης ως συνήθως.

Ας δούμε ένα απλό παράδειγμα. Με δεδομένη την παρακάτω μέθοδο

```
def square(x: Int): Int =
  x * x
```

μπορούμε να επεκτείνουμε την κλήση της μεθόδου

```
square(2)
```

χρησιμοποιώντας ένα block

```
{
  square(2)
}
```

συνδέοντας την παράμετρο `x` με την έκφραση `2`

```
{
  val x = 2
  square(2)
}
```

και αντικαθιστώντας το σώμα της μεθόδου

```
{
  val x = 2
  x * x
}
```

Μπορούμε τώρα να κάνουμε αντικατάσταση ως συνήθως

```
{
  2 * 2
}
```

και τελικά

```
{
  4
}
```

Για άλλη μία φορά βλέπουμε ότι εμπλέκεται το μοντέλο της αντικατάστασης αλλά κανένα βήμα δεν ήταν ιδιαίτερα δύσκολο.

Άσκηση

Την τελευταία φορά που ασχοληθήκαμε με την αντικατάσταση, αφιερώσαμε αρκετό χρόνο στην διερεύνηση της σειράς αξιολόγησης. Στην παραπάνω περιγραφή, αναφέραμε ότι οι παράμετροι μίας μεθόδου αξιολογούνται πριν από το σώμα. Τα πράγματα όμως δεν είναι ακριβώς έτσι. Για παράδειγμα, θα μπορούσαμε να αξιολογήσουμε τις παραμέτρους μίας μεθόδου μόνο στο σημείο που χρειάζονται. Αυτό θα μπορούσε να μας εξοικονομήσει λίγο χρόνο στην περίπτωση που μία

μέθοδος δεν χρησιμοποιεί κάποια από τις παραμέτρους της. Στην Scala, χρησιμοποιώντας τον παλιό μας φίλο `println`, μπορούμε να προσδιορίσουμε το πότε αξιολογούνται οι παράμετροι μίας μεθόδου.

See the solution

6.4 Συμπεράσματα

Σ' αυτό το κεφάλαιο μάθαμε πώς να γράφουμε τις δικές μας απλές μεθόδους και είδαμε πώς μπορούμε να χρησιμοποιήσουμε το μοντέλο της αντικατάστασης για την αξιολόγηση των μεθόδων.

Είδαμε ότι οι μέθοδοι μπορούν να εφαρμόσουν την διαδικασία της αφαίρεσης στις εκφράσεις, με τον ίδιο τρόπο όπως τα ονόματα και ότι μπορούν επίσης να γενικεύσουν εκφράσεις, επιτρέποντάς μας να ομαδοποιήσουμε εκφράσεις σχετικές μεταξύ τους κάτω από ένα όνομα.

Γράψαμε αρκετές ενδιαφέρουσες μεθόδους αλλά ακόμα έχουμε πολύ περισσότερο επαναλαμβανόμενο κώδικα από όσο θα θέλαμε (σκεφτείτε τις επαναλαμβανόμενες κλήσεις του `box` και του `circle` στις ασκήσεις). Στο επόμενο κεφάλαιο, θα μάθουμε πώς μπορούμε να γενικεύουμε πάνω στον κώδικα χρησιμοποιώντας δομημένη αναδρομή (structural recursion) με φυσικούς αριθμούς.

7 Δομημένη Αναδρομή

Σε αυτό το κεφάλαιο θα δούμε το πρώτο μας σημαντικό μοτίβο δόμισης υπολογισμών: την *δομημένη αναδρομή (structural recursion) με φυσικούς αριθμούς*. Ας αναλύσουμε αυτή την έννοια λίγο καλύτερα:

- Με τον όρο μοτίβο, εννοούμε ότι θα μάθουμε έναν τρόπο για να γράφουμε κώδικα που θα μπορεί να χρησιμοποιηθεί σε πολλές διαφορετικές περιπτώσεις. Την δομημένη αναδρομή θα την συναντήσουμε αρκετές φορές.
- Με τον όρο φυσικοί αριθμοί, εννοούμε τους θετικούς ακέραιους αριθμούς όπως το 0, 1, 2, κλπ.
- Με τον όρο αναδρομή, εννοούμε κάτι που αναφέρεται στον εαυτό του. Δομημένη αναδρομή, είναι αυτή η οποία ακολουθεί την δομή των δεδομένων που επεξεργάζεται. Αν τα δεδομένα αυτά είναι αναδρομικά (δηλαδή αναφέρονται στον εαυτό τους) τότε και η δομημένη αναδρομή θα αναφέρεται στον εαυτό της. Σε λίγο θα δούμε πιο αναλυτικά τι σημαίνει αυτό.

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

7.1 Μία Σειρά από Κουτιά

Ας ξεκινήσουμε με ένα παράδειγμα σχεδιασμού κουτιών σε μία γραμμή ή στήλη όπως φαίνεται στην εικόνα fig. 21.

Figure 21: Πέντε κουτιά χρωματισμένα με Royal Blue

Για αρχή, ας ορίσουμε ένα μόνο κουτί.

```
val aBox = Image.rectangle(20, 20).fillColor(Color.r
oyalBlue)
// aBox: doodle.core.Image = ContextTransform(doodle
.core.Image$$Lambda$9266/1047889822@43432aa4,Rectang
le(20.0,20.0))
```

Μετά, ένα κουτί σε μία γραμμή είναι απλώς

```
val oneBox = aBox
// oneBox: doodle.core.Image = ContextTransform(dood
le.core.Image$$Lambda$9266/1047889822@43432aa4,Recta
ngle(20.0,20.0))
```

Αν θέλουμε να έχουμε δύο κουτιά το ένα δίπλα στο άλλο, τότε μπορούμε πολύ εύκολα να το πετύχουμε γράφοντας τον παρακάτω κώδικα.

```
val twoBoxes = aBox beside oneBox
// twoBoxes: doodle.core.Image = Beside(ContextTrans
form(doodle.core.Image$$Lambda$9266/1047889822@43432
aa4,Rectangle(20.0,20.0)),ContextTransform(doodle.co
re.Image$$Lambda$9266/1047889822@43432aa4,Rectangle(
20.0,20.0)))
```

Αντίστοιχα και για τρια κουτιά.

```
val threeBoxes = aBox beside twoBoxes
// threeBoxes: doodle.core.Image = Beside(ContextTra
nsform(doodle.core.Image$$Lambda$9266/1047889822@434
32aa4,Rectangle(20.0,20.0)),Beside(ContextTransform(
doodle.core.Image$$Lambda$9266/1047889822@43432aa4,R
ectangle(20.0,20.0)),ContextTransform(doodle.core.Im
age$$Lambda$9266/1047889822@43432aa4,Rectangle(20.0,
20.0))))
```

Ούτω καθεξής για όσα κουτιά θέλουμε να φτιάξουμε.

Μπορεί να σκέφτεστε ότι αυτός ο τρόπος δημιουργίας των παραπάνω εικόνων είναι κάπως ασυνήθιστος. Για παράδειγμα γιατί να μην γράψουμε απλώς κάτι σαν το παρακάτω;

```
val threeBoxes = aBox beside aBox beside aBox
// threeBoxes: doodle.core.Image = Beside(Beside(Con
textTransform(doodle.core.Image$$Lambda$9266/1047889
822@43432aa4,Rectangle(20.0,20.0)),ContextTransform(
doodle.core.Image$$Lambda$9266/104788922@43432aa4,R
ectangle(20.0,20.0))),ContextTransform(doodle.core.I
mage$$Lambda$9266/104788922@43432aa4,Rectangle(20.0
,20.0)))
```

Αυτοί οι δύο ορισμοί είναι ίσοι μεταξύ τους. Για κάθε νέα εικόνα που ορίσαμε, επιλέξαμε να χρησιμοποιήσουμε την προηγούμενη ώστε να δώσουμε έμφαση στην δομή. Κάπως έτσι οδηγούμαστε σιγά σιγά στην δομημένη αναδρομή.

Αυτός ο τρόπος δημιουργίας εικόνων μπορεί να γίνει πολύ κουραστικός. Πιο βολικό θα ήταν να μπορούσαμε με κάποιον τρόπο να πούμε στον υπολογιστή πόσα κουτιά θέλουμε. Χρησιμοποιώντας πιο τεχνικούς όρους, θα θέλαμε να εφαρμόσουμε την μέθοδο της αφαίρεσης στις παραπάνω εκφράσεις. Σε προηγούμενο κεφάλαιο, μάθαμε ότι οι μέθοδοι χρησιμοποιούν την μέθοδο της αφαίρεσης πάνω στις εκφράσεις, οπότε ας προσπαθήσουμε και εμείς να γράψουμε μία μέθοδο ώστε να λύσουμε αυτό το πρόβλημα.

Θα ξεκινήσουμε γράφοντας τον σκελετό της μεθόδου, στον οποίο ως συνήθως θα ορίζεται το τι εισάγεται στην μέθοδο και το πώς αυτό αξιολογείται. Σ' αυτή την περίπτωση θα δώσουμε ως είσοδο μία παράμετρο `count` τύπου `Int`, η οποία αντιπροσωπεύει τον αριθμό των κουτιών που θέλουμε και θα πάρουμε πίσω μία `Image`.

```
def boxes(count: Int): Image =
  ???
// boxes: (count: Int)doodle.core.Image
```

Τώρα ξεκινάει η εφαρμογή της νέας έννοιας, δηλαδή της *δομημένης αναδρομής*. Παρατηρήσαμε προηγουμένως ότι το `threeBoxes` ορίζεται σε σχέση με το `twoBoxes` και το `twoBoxes` σε σχέση με το `box`. Θα μπορούσαμε ακόμη να ορίσουμε και το `box` σε σχέση με το *κανένα* κουτί, όπως παρακάτω:

```
val oneBox = aBox beside Image.empty
// oneBox: doodle.core.Image = Beside(ContextTransfo
rm(doodle.core.Image$$Lambda$9266/1047889822@43432aa
4,Rectangle(20.0,20.0)),Empty)
```

Για να αναπαραστήσουμε το “κανένα” κουτί, χρησιμοποιήσαμε το `Image.empty`.

Φανταστείτε ότι έχουμε ήδη δημιουργήσει την μέθοδο `boxes`. Μπορούμε να πούμε ότι οι παρακάτω ιδιότητες της `boxes` ισχύουν πάντα εφόσον είναι σωστά φτιαγμένες:

- `boxes(0) == Image.empty`
- `boxes(1) == aBox beside boxes(0)`
- `boxes(2) == aBox beside boxes(1)`
- `boxes(3) == aBox beside boxes(2)`

Οι τρεις τελευταίες ιδιότητες έχουν όλες το ίδιο βασικό σχήμα. Μπορούμε να τις περιγράψουμε όλες, καθώς και οποιαδήποτε άλλη περίπτωση για `n > 0`, χρησιμοποιώντας μία μόνο ιδιότητα: την `boxes(n) == aBox beside boxes(n - 1)`. Έτσι, μένουμε τελικά με τις δύο ιδιότητες:

- `boxes(0) == Image.empty`
- `boxes(n) == aBox beside boxes(n-1)`

Αυτές οι δύο, ορίζουν την συμπεριφορά της `boxes`. Μπορούμε να δημιουργήσουμε την `boxes` μετατρέποντας αυτές τις ιδιότητες σε κώδικα.

Η `boxes` ολοκληρωμένη:

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
```

```
    }
// boxes: (count: Int)doodle.core.Image
```

Δοκιμάστε να την εκτελέσετε και δείτε τι αποτελέσματα θα πάρετε! Αυτός ο τρόπος είναι λίγο πιο περίπλοκος από τις ιδιότητες που γράψαμε παραπάνω, όμως είναι ακόμα η πρώτη φορά που χρησιμοποιήσαμε δομημένη αναδρομή με φυσικούς αριθμούς.

Σε αυτό το σημείο πρέπει να απαντήσουμε σε δύο ερωτήσεις. Πρώτον, πώς δουλεύει η έκφραση `match`; Δεύτερον, υπάρχει κάποιο γενικό μοντέλο που μπορούμε να χρησιμοποιούμε ώστε να φτιάχνουμε μόνοι μας μεθόδους σαν αυτή; Ας απαντήσουμε στις ερωτήσεις με την σειρά.

Άσκηση: Στοιβάζοντας Κουτιά

Πριν εξηγήσουμε την έκφραση `match`, θα πρέπει να είστε σε θέση να αναδιαμορφώσετε την μέθοδο `boxes` ώστε να δημιουργήσετε την παρακάτω εικόνα fig. 22.

Θέλουμε να συνηθίσουμε την σύνταξη της έκφρασης `match`, οπότε καλό θα ήταν να γράψετε τον κώδικα με το χέρι αντί να κάνετε αντιγραφή-επικόλληση.

Figure 22: Τρία κουτιά, το ένα πάνω στο άλλο, γεμισμένα με χρώμα Royal Blue

See the solution

7.2 Εκφράσεις Match

Στην προηγούμενη ενότητα είδαμε την παρακάτω έκφραση `match`

```
count match {
  case 0 => Image.empty
  case n => aBox beside boxes(n-1)
}
```

Πώς μπορούμε να κατανοήσουμε αυτό το νέο είδος έκφρασης και να γράψουμε και εμείς μία δική μας? Ας την αναλύσουμε.

Το πρώτο που πρέπει να πούμε είναι ότι η `match` είναι όντως έκφραση και άρα αξιολογείται με μία τιμή. Αν ο παραπάνω ισχυρισμός δεν ίσχυε, τότε η μέθοδος `boxes` δεν θα δούλευε!

Για να καταλάβουμε πώς αξιολογείται η match, πρέπει να μάθουμε περισσότερες λεπτομέρειες. Γενικώς, μία έκφραση `match` έχει το παρακάτω σχήμα

```
<anExpression> match {
  case <pattern1> => <expression1>
  case <pattern2> => <expression2>
  case <pattern3> => <expression3>
  ...
}
```

ένα `<anExpression>`, συγκεκριμένα το `count` για την παραπάνω περίπτωση, είναι μία έκφραση η οποία θα αξιολογηθεί με μία τιμή που θα χρησιμοποιηθεί παρακάτω προς αντιπαραβολή. Το `<pattern1>` και τα υπόλοιπα μοτίβα αντιπαραβάλλονται με αυτή την τιμή. Μέχρι τώρα έχουμε δει δύο είδη μοτίβων:

- μία κυριολεκτική έκφραση (όπως το `case 0`) το οποίο αντιστοιχίζεται με την ακριβή τιμή της κυριολεκτικής έκφρασης, και
- έναν μπαλαντέρ (όπως το `case n`) το οποίο αντιστοιχίζεται με *οτιδήποτε* άλλο και ταυτόχρονα εισάγει μία σύνδεση στο δεξί μέρος της έκφρασης.

Τέλος, οι εκφράσεις που βρίσκονται στην δεξιά πλευρά, όπως η `<expression1>`, είναι απλώς εκφράσεις όπως πολλές άλλες που έχουμε γράψει ως τώρα. Ολόκληρη η έκφραση `match`, αξιολογείται με την τιμή της δεξιάς έκφρασης του *πρώτου* μοτίβου που μπορεί να αντιστοιχηθεί με

την έκφραση προς αντιπαραβολή. Έτσι, όταν καλούμε την `boxes(0)`, και τα δύο μοτίβα μπορούν αντιστοιχηθούν (αφού ο μπαλαντέρ αντιστοιχίζεται με οτιδήποτε) αλλά αφού είναι πρώτο το κυριολεκτικό pattern, θα αξιολογηθεί η έκφραση `Image.empty`.

Μία έκφραση `match` που ελέγχει όλες τις πιθανές περιπτώσεις καλείται εξαντλητική match (exhaustive match). Αν υποθέσουμε ότι το `count` είναι πάντα ίσο ή μεγαλύτερο από το μηδέν τότε η `match` στην `boxes` είναι εξαντλητική.

Αφού εξοικειωθούμε με τις εκφράσεις `match` και πριν εξηγήσουμε την δομημένη αναδρομή, πρέπει να μελετήσουμε την δομή των φυσικών αριθμών.

Ασκήσεις

Μαντέψτε το Αποτέλεσμα

Ας ελέγξουμε πόσο καλά κατανοούμε την match μαντεύοντας το αποτέλεσμα των παρακάτω εκφράσεων και γιατί αξιολογούνται έτσι.

```
"abcd" match {
  case "bcde" => 0
  case "cdef" => 1
  case "abcd" => 2
}
```

```
1 match {
  case 0 => "zero"
  case 1 => "one"
  case 1 => "two"
}
```

```
1 match {
  case n => n + 1
  case 1 => 1000
}
```

```
1 match {
  case a => a
  case b => b + 1
  case c => c * 2
}
```

See the solution

Όταν Δεν Υπάρχει Αντιστοίχιση

Τι γίνεται όμως αν κανένα μοτίβο δεν μπορεί να αντιστοιχηθεί σε μία έκφραση `match`? Μαντέψτε! Βρείτε μία έκφραση `match` η οποία απέτυχε και δείτε αν μαντέψατε σωστά. (Σύμφωνα με αυτά που ξέρουμε ως τώρα δεν έχουμε κανέναν λόγο να περιμένουμε κάποια συγκεκριμένη συμπεριφορά οπότε οποιαδήποτε λογική υπόθεση είναι αποδεκτή.)

See the solution

7.3 Οι Φυσικοί Αριθμοί

Φυσικοί είναι οι ακέραιοι αριθμοί που είναι μεγαλύτεροι από το 0 ή ίσοι με αυτό. Με άλλα λόγια είναι οι αριθμοί 0, 1, 2, 3, … (Μερικοί ορίζουν τους φυσικούς αριθμούς σαν να ξεκινάν από το 1 και όχι από το 0. Στην δική μας περίπτωση δεν έχει πολύ μεγάλη σημασία ποιον ορισμό προτιμάτε αλλά εμείς θα υποθέσουμε ότι ξεκινάν από το 0.)

Μία ενδιαφέρουσα ιδιότητα των φυσικών αριθμών είναι ότι μπορούν να οριστούν αναδρομικά. Αυτό σημαίνει ότι μπορούμε να τους ορίσουμε σε σχέση με τον εαυτό τους. Θα μπορούσε κανείς να σκεφτεί ότι αυτός ο κυκλικός ορισμός οδηγεί σε παραλογισμό. Για να αποφευχθεί κάτι τέτοιο χρησιμοποιούμε μία *βασική περίπτωση (base case)*, η οποία σταματάει την αναδρομή. Επομένως, ο ορισμός είναι ο εξής:

Ένας φυσικός αριθμός `n` είναι

- 0, ή
- 1 + `m`, όπου `m` είναι ένας φυσικός αριθμός.

Η περίπτωση 0 είναι η βασική περίπτωση, ενώ η επόμενη είναι αναδρομική αφού ορίζει τον φυσικό αριθμό `n` σε σχέση με τον φυσικό

αριθμό `m`. Επειδή ο `m` είναι πάντα μικρότερος από τον `n` και η βασική υπόθεση είναι ο μικρότερος φυσικός αριθμός, αυτός ο ορισμός ορίζει όλους τους φυσικούς αριθμούς.

Εάν μας δοθεί ένας φυσικός αριθμός, για παράδειγμα ο 3, μπορούμε να τον διασπάσουμε χρησιμοποιώντας τον παραπάνω ορισμό, ως εξής:

3 = 1 + 2 = 1 + (1 + 1) = 1 + (1 + (1 + 0))

Χρησιμοποιούμε τον κανόνα της αναδρομής ώστε να απλοποιήσουμε την παραπάνω εξίσωση μέχρι να φτάσουμε σε σημείο που δεν μπορούμε να χρησιμοποιήσουμε πιά τον κανόνα. Χρησιμοποιούμε την βασική περίπτωση ώστε να σταματήσουμε την αναδρομή.

7.4 Δομημένη Αναδρομή

Τώρα είμαστε έτοιμοι να δούμε την δομημένη αναδρομή. Η μορφή της δομημένης αναδρομής με φυσικούς αριθμούς μας δίνει:

- έναν σκελετό κώδικα για την επεξεργασία οποιουδήποτε φυσικού αριθμού που μπορούμε να χρησιμοποιήσουμε σε πολλές περιπτώσεις, και
- εγγύηση ότι μπορούμε να χρησιμοποιήσουμε αυτόν τον σκελετό για *οποιονδήποτε* υπολογισμό με φυσικούς αριθμούς.

Θυμηθείτε ότι προηγουμένως γράψαμε την `boxes` όπως παρακάτω

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
// boxes: (count: Int)doodle.core.Image
```

Όταν φτιάχναμε την `boxes` πέσαμε πάνω σ' αυτή τη μορφή που προαναφέραμε. Μπορούμε εύκολα να δούμε ότι η παραπάνω μορφή ακολουθεί τον ορισμό των φυσικών αριθμών. Θυμηθείτε τον αναδρομικό ορισμό των φυσικών αριθμών: ένας φυσικός αριθμός `n` είναι

- 0, ή

- 1 + `m`, όπου `m` είναι ένας φυσικός αριθμός.

Η μορφή της έκφρασης `match` ταιριάζει ακριβώς μ' αυτόν τον ορισμό. Η παρακάτω έκφραση

```
count match {
  case 0 => ???
  case n => ???
}
```

σημαίνει ότι ελέγχουμε το `count` για δύο περιπτώσεις, την περίπτωση όπου το `count` είναι 0 και την περίπτωση που το `count` είναι οποιοσδήποτε άλλος φυσικός αριθμός `n` (δηλαδή `1 + m`).

Η δεξιά πλευρά της έκφρασης `match` μας λέει τι κάνουμε σε κάθε περίπτωση. Στην περίπτωση μηδέν αντιστοιχεί το αποτέλεσμα `Image.empty`. Στην περίπτωση `n` αντιστοιχεί το `aBox beside boxes(n-1)`.

Τώρα θα μπούμε στο πιό σημαντικό κομμάτι. Παρατηρήστε οτι η δομή της δεξιάς πλευράς αντικατοπτρίζει την δομή των φυσικών αριθμών που προσπαθούμε να αντιστοιχίσουμε. Όταν αντιστοιχίζουμε την βασική περίπτωση 0, το αποτέλεσμα είναι η βασική περίπτωση `Image.empty`. Όταν αντιστοιχίζουμε την αναδρομική περίπτωση `n`, η δομή της δεξιάς πλευράς ταιριάζει με την δομή της αναδρομικής περίπτωσης του ορισμού των φυσικών αριθμών. Στον ορισμό δηλώνεται ότι το `n` είναι `1 + m`. Στην δεξιά πλευρά αντικαθιστούμε το 1 με το `aBox`, το + με το `beside` και καλούμε αναδρομικά την `boxes` με το `m` (που είναι `n-1`) όπου σύμφωνα με τον ορισμό γίνεται η αναδρομή.

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
// boxes: (count: Int)doodle.core.Image
```

Για να συνοψίσουμε, η αριστερή πλευρά της έκφρασης `match` ταιριάζει ακριβώς με τον ορισμό των φυσικών αριθμών. Η δεξιά πλευρά ταιριάζει με τον ορισμό αλλά αντικαταστήσαμε τους φυσικούς αριθμούς με εικόνες.

Η εικόνα που είναι ισοδύναμη με το μηδέν είναι η `Image.empty`. Η εικόνα που είναι ισοδύναμη με το `1 + m` είναι η `aBox beside boxes(m)`.

Αυτή η γενική μορφή ισχύει για οτιδήποτε θέλουμε να γράψουμε και μετατρέπει τους φυσικούς αριθμούς σε κάποιον άλλο τύπο. Πάντα έχουμε μία έκφραση `match`. Πάντα έχουμε τις δύο μορφές, μία για την βασική και μία για την αναδρομική περίπτωση. Οι εκφράσεις της δεξιάς πλευράς είναι πάντα η βασική περίπτωση και η αναδρομική, η οποία περιέχει κάτι που αντικαθιστά το `1` και το `+`, και μία αναδρομική κλήση για το `n-1`.

Δομημένη Αναδρομη στη Δομή των Φυσικών Αριθμών

Παρακάτω μπορείτε να δείτε την γενική μορφή της δομημένης αναδρομής με φυσικούς αριθμούς

```
def name(count: Int): Result =
  count match {
    case 0 => resultBase
    case n => resultUnit add name(n-1)
  }
```

όπου τα `Result`, `resultBase`, `resultUnit` και το `add`, είναι συγκεκριμένα για το πρόβλημα που επιλύουμε. Έτσι, για να εφαρμόσουμε την δομημένη αναδρομή με φυσικούς αριθμούς πρέπει να

- αναγνωρίσουμε ότι η μέθοδος που γράφουμε παίρνει ως είσοδο έναν φυσικό αριθμό,
- βρούμε τον τύπο του αποτελέσματος, και
- αποφασίσουμε ποια πρέπει να είναι η βασική περίπτωση, η μονάδα και η πρόσθεση για το αποτέλεσμα.

Τώρα είμαστε έτοιμοι να διασκεδάσουμε εξερευνώντας αυτό το βασικό αλλά πολύ δυνατό εργαλείο.

7.4 Αποδείξεις και Προγράμματα

Αν έχετε μελετήσει μαθηματικά, το πιό πιθανό είναι ότι έχετε συναντήσει την μέθοδο της επαγωγής σε διάφορες αποδείξεις. Η γενική μορφή για απόδειξη μέσω επαγωγής μοιάζει πολύ με την γενική μορφή της δομημένης αναδρομής με φυσικούς αριθμούς. Δεν είναι σύμπτωση! Υπάρχει μία σχέση μεταξύ αυτών των δύο. Μπορούμε να δούμε την δομημένη αναδρομή με φυσικούς αριθμούς ως μία απόδειξη μέσω επαγωγής. Η δυνατότητα χρήσης της ιδιότητας της μετατροπής των φυσικών αριθμών σε σχέση με τον σκελετό της δομημένης αναδρομής, βασίζεται πάνω σε μαθηματικά που χρησιμοποιούμε έμμεσα. Ακόμη, μπορούμε να αποδείξουμε ιδιότητες του κώδικά μας χρησιμοποιώντας την σύνδεση αυτών των δύο: οποιαδήποτε δομημένη αναδρομή ορίζει εμμέσως μία απόδειξη κάποιας ιδιότητας.

Αυτή η γενική σύνδεση μεταξύ αποδείξεων και προγραμμάτων είναι γνωστή ως *Ισομορφισμός Howard-Curry*.

Ασκήσεις

Σταυρός

Στην πρώτη άσκηση θέλουμε να φτιάξουμε μία μέθοδο με όνομα `cross` που θα δημιουργεί εικόνες σταυρών. Στην εικόνα fig. 23 φαίνονται τέσσερις σταυροί, οι οποίοι αντιστοιχούν στην κλήση της μεθόδου `cross` από το `0` ως το `3`.

Ο σκελετός της μεθόδου είναι ο παρακάτω

```
def cross(count: Int): Image =
  ???
// cross: (count: Int)doodle.core.Image
```

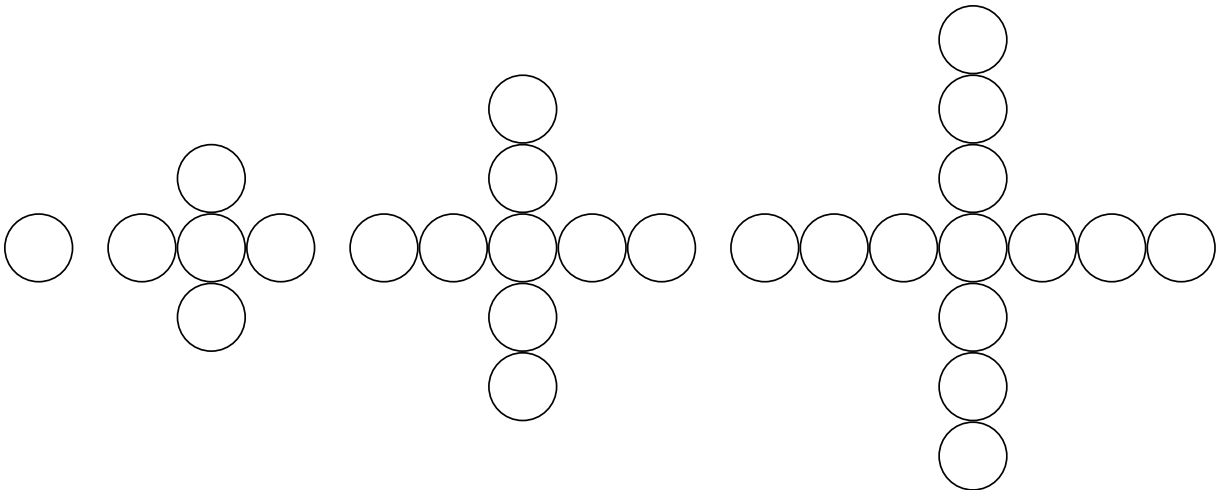


Figure 23: Οι σταυροί δημιουργούνται χρησιμοποιώντας ένα `count` από το 0 ως το 3.

Ποια μορφή θα χρησιμοποιήσουμε για να συμπληρώσουμε το σώμα της `cross`? Γράψτε την μόνοι σας.

See the solution

Τώρα που βρήκαμε ποια μορφή θα χρησιμοποιήσουμε πρέπει να συμπληρώσουμε τα επιμέρους κομμάτια του προβλήματος:

- την βασική περίπτωση
- τους τελεστές της μονάδας και της πρόσθεσης.

Βοήθεια: χρησιμοποιήστε την εικόνα fig. 23 για να αναγνωρίσετε τα παραπάνω ζητούμενα στοιχεία.

See the solution

Τώρα συμπληρώστε όλα τα στοιχεία της μεθόδου `cross` ώστε να λειτουργεί επιτυχώς.

See the solution

Σκακιέρα

Στην άσκηση με τον σταυρό είδαμε οτι το δύσκολο κομμάτι είναι το να αναγνωρίσουμε την αναδρομική δομή μέσα σ' αυτό που προσπαθούμε να δημιουργήσουμε. Μόλις όμως το καταφέραμε, η συμπλήρωση των υπόλοιπων στοιχείων της μορφής της δομημένης αναδρομής ήταν εύκολη.

Σ' αυτή και στην επόμενη άσκηση, θα προσπαθήσουμε να εξασκήσουμε

το μάτι σας στην δομημένη αναδρομή. Η αποστολή σας σ' αυτήν την άσκηση είναι να αναγνωρίσετε πώς μπορεί να χρησιμοποιηθεί η δομημένη αναδρομή σε μία σκακιέρα και να φτιάξετε μία μέθοδο που σχεδιάζει σκακιέρες. Ο σκελετός της μεθόδου είναι ο παρακάτω

```
def chessboard(count: Int): Image =
  ???
```

Στην εικόνα fig. 24 μπορείτε να δείτε παραδείγματα με σκακιέρες που έχουν σχεδιαστεί με το `count` να ξεκινάει από το `0` και να φτάνει μέχρι το `2`. Βοήθεια: παρατηρήστε ότι το `count` αυτή τη φορά δεν μας δίνει το πλάτος της σκακιέρας αλλά μας λέει τον αριθμό των "μονάδων σκακιέρας" που πρέπει να συνδυάσουμε.

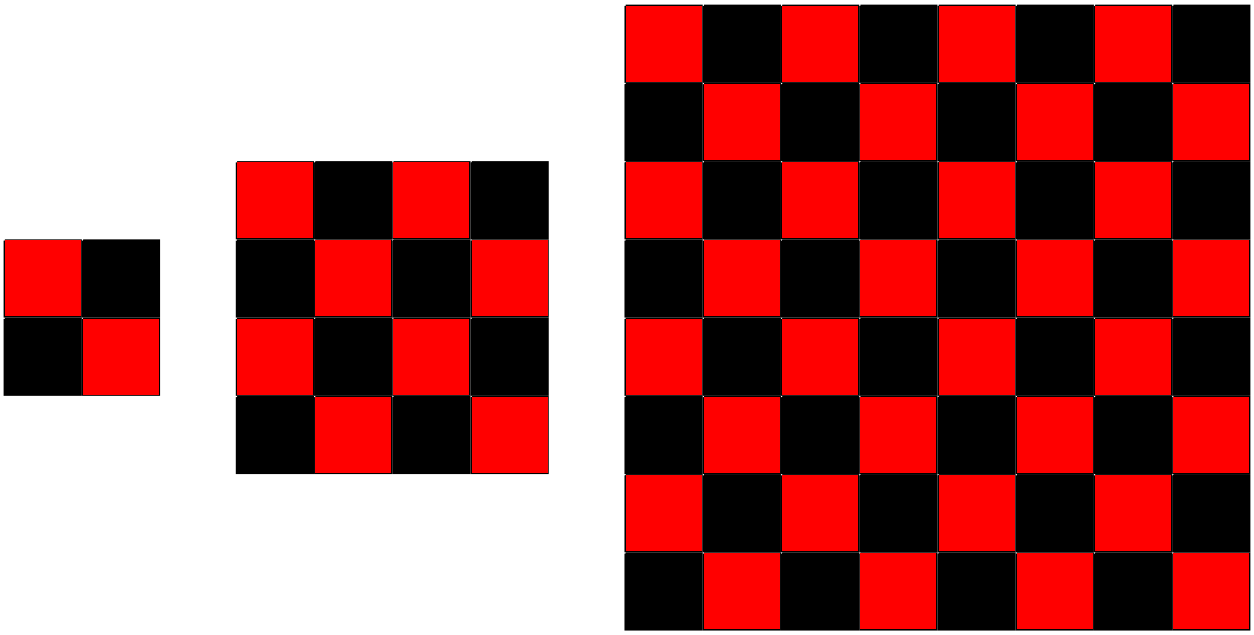


Figure 24: Σκακιέρες που δημιουργούνται χρησιμοποιώντας `count` από 0 ως 2.

Φτιάξτε την μέθοδο `chessboard`.

See the solution

Τρίγωνο Sierpinkski

Το τρίγωνο Sierpinski, όπως φαίνεται στην εικόνα fig. 25, είναι ένα διάσημο fractal. (Η εικόνα fig. 25 δείχνει ένα τρίγωνο Sier*pink*ski.)

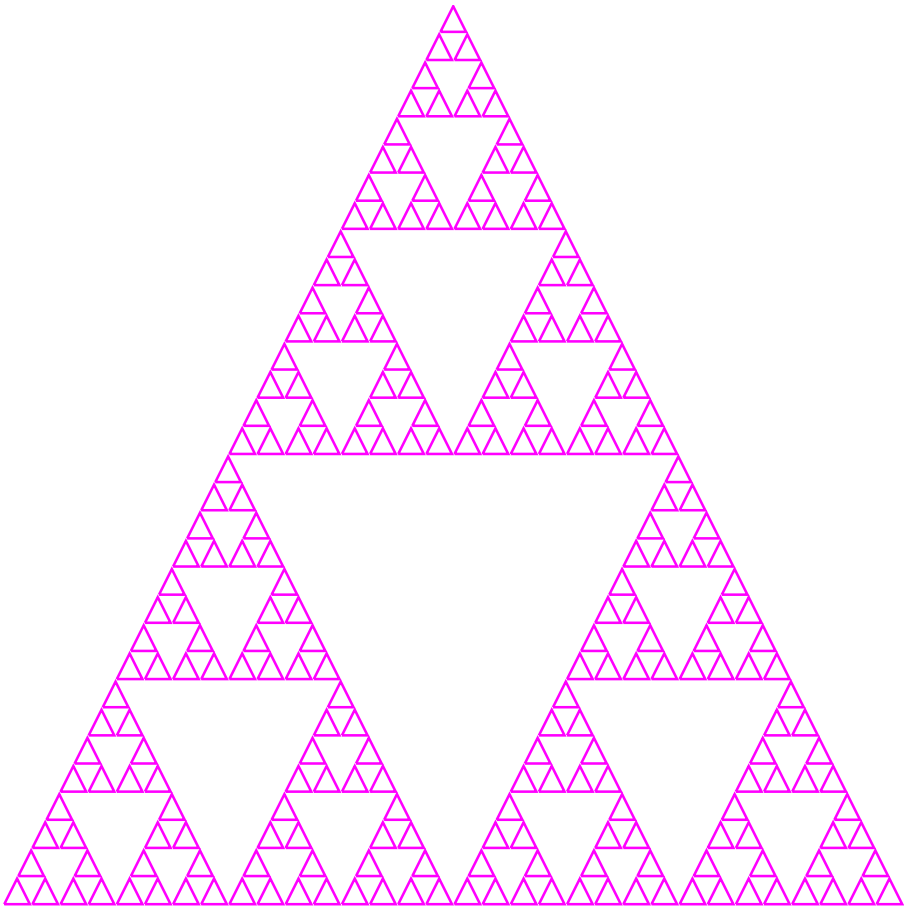


Figure 25: Το τρίγωνο Sierpinski.

Μπορεί να φαίνεται αρκετά περίπλοκο αλλά μπορούμε να διασπάσουμε την δομή του σε μία μορφή την οποία μπορούμε να φτιάξουμε χρησιμοποιώντας δομημένη αναδρομή με φυσικούς αριθμούς. Φτιάξτε μία μέθοδο με τον παρακάτω σκελετό

```
def sierpinski(count: Int): Image =
  ???
// sierpinski: (count: Int)doodle.core.Image
```

Αυτή τη φορά δεν θα σας δοθεί βοήθεια. Έχουμε ήδη δει όλα όσα χρειαζόμαστε για να το λύσουμε.

See the solution

7.5 Κατανοώντας την Αναδρομή

Έχουμε γίνει πλέον έμπειροι χρήστες της δομημένης αναδρομής με

φυσικούς αριθμούς. Ας επιστρέψουμε όμως στο μοντέλο αντικατάστασης για να δούμε αν μπορεί να χρησιμοποιηθεί σε συνδυασμό με το νέο εργαλείο που μάθαμε, την αναδρομή.

Θυμηθείτε ότι αυτή η μέθοδος μας επιτρέπει να αντικαταστήσουμε την τιμή μίας έκφρασης οπουδήποτε και αν την συναντήσουμε. Στην περίπτωση κλήσης μεθόδου, μπορούμε να αντικαταστήσουμε το σώμα της μεθόδου μετονομάζοντας τις παραμέτρους.

Το πρώτο παράδειγμα που είδαμε με την μέθοδο της αναδρομής ήταν η `boxes` και η σύνταξή της ήταν η παρακάτω:

```
val aBox = Image.rectangle(20, 20).fillColor(Color.r
oyalBlue)

def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
```

Ας προσπαθήσουμε να εφαρμόσουμε την μέθοδο της αντικατάστασης στην `boxes(3)` για να δούμε τι θα πάρουμε.

Η πρώτη αντικατάσταση που μπορούμε να κάνουμε είναι η παρακάτω

```
boxes(3)
// Αντικατάσταση του σώματος της `boxes`
3 match {
  case 0 => Image.empty
  case n => aBox beside boxes(n-1)
}
```

Γνωρίζοντας πώς να αξιολογήσουμε μία έκφραση `match` και πώς να χρησιμοποιήσουμε την αντικατάσταση, μπορούμε να κάνουμε την παρακάτω αλλαγή

```
3 match {
  case 0 => Image.empty
```

```
  case n => aBox beside boxes(n-1)
}
// Αντικατάσταση της δεξιάς πλευράς της έκφρασης της
 `case n`
aBox beside boxes(2)
```

Μπορούμε να κάνουμε άλλη μία αντικατάσταση στην `boxes(2)` και θα πάρουμε

```
aBox beside boxes(2)
// Αντικατάσταση του σώματος της `boxes`
aBox beside {
  2 match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
}
// Αντικατάσταση της δεξιάς πλευράς της έκφρασης της
 `case n`
aBox beside {
  aBox beside boxes(1)
}
```

Επαναλαμβάνοντας την ίδια διαδικασία μερικές ακόμη φορές θα πάρουμε

```
aBox beside {
  aBox beside {
    1 match {
      case 0 => Image.empty
      case n => aBox beside boxes(n-1)
    }
  }
}
// Αντικατάσταση της δεξιάς πλευράς της έκφρασης της
 `case n`
aBox beside {
  aBox beside {
      aBox beside boxes(0)
```

```
  }
}
// Αντικατάσταση του σώματος της `boxes`
aBox beside {
  aBox beside {
    aBox beside {
      0 match {
        case 0 => Image.empty
        case n => aBox beside boxes(n-1)
      }
    }
  }
}
// Αντικατάσταση της δεξιάς πλευράς της έκφρασης της
 `case 0`
aBox beside {
  aBox beside {
    aBox beside {
      Image.empty
    }
  }
}
```

Το τελικό αποτέλεσμα έχει απλουστευτεί σε

```
aBox beside aBox beside aBox beside Image.empty
```

και είναι ακριβώς αυτό που περιμέναμε. Η μέθοδος της αντικατάστασης μας βοηθάει να καταλάβουμε την λογική της αναδρομής. Τέλεια! Παρόλα αυτά, οι αντικαταστάσεις είναι περίπλοκες και η παρακολούθησή τους είναι δύσκολη χωρίς να γράφουμε κάπου τις αλλαγές. Ένας πιο πρακτικός τρόπος να καταλάβουμε την αναδρομή είναι να υποθέσουμε απλώς ότι λειτουργεί και να ανησυχούμε μόνο για το τι καινούριο θα φέρει το κάθε βήμα.

Για παράδειγμα, όταν προσπαθούμε να καταλάβουμε τι γίνεται στην `boxes`

```
def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
```

εξετάζοντάς την, μπορούμε να πούμε ότι η βασική περίπτωση είναι σωστή. Κοιτώντας την περίπτωση της αναδρομής *υποθέτουμε* ότι το `boxes(n-1)` θα κάνει αυτό που πρέπει. Μετά ρωτάμε τον εαυτό μας “είναι σωστό αυτό που κάνουμε στο βήμα της περίπτωσης της αναδρομής; Είναι η ίδια η αναδρομή σωστή;” Η απάντηση είναι ναι, αν η αναδρομή στο `boxes(n-1)` δημιουργεί `n-1` κουτιά σε σειρά, το να βάλουμε ένα κουτί πριν από αυτή τη σειρά είναι η σωστή κίνηση. Αυτός ο τρόπος κατανόησης της λογικής της αναδρομής είναι πολύ πιο εύκολος από την χρήση της αντικατάστασης *και* εγγυάται ότι θα λειτουργήσει *αν* χρησιμοποιούμε δομημένη αναδρομή.

Ασκήσεις

Παρακάτω μπορείτε να δείτε μερικά εύκολα παραδείγματα δομημένης αναδρομής. Βρείτε αν οι μέθοδοι κάνουν αυτό που υποστηρίζουν, *χωρίς* να τις εκτελέσετε.

```
// Παίρνοντας ως δεδομένο έναν φυσικό αριθμό, επιστρ
έφει αυτόν τον αριθμό
// Παραδείγματα:
//   identity(0) == 0
//   identity(3) == 3
def identity(n: Int): Int =
  n match {
    case 0 => 0
    case n => 1 + identity(n-1)
  }
```

See the solution

```
// Παίρνοντας ως δεδομένο έναν φυσικό αριθμό, το απο
```

```
τέλεσμα πρέπει να είναι το διπλάσιό του
// Παραδείγματα:
//   double(0) == 0
//   double(3) == 6
def double(n: Int): Int =
  n match {
    case 0 => 0
    case n => 2 * double(n-1)
  }
```

See the solution

7.6 Βοηθητικές Παράμετροι

Είδαμε πώς μπορούμε να χρησιμοποιήσουμε την δομημένη αναδρομή με φυσικούς αριθμούς ώστε να γράψουμε ενδιαφέροντα προγράμματα. Σ' αυτή την ενότητα θα δούμε μία επέκταση των προηγουμένων η οποία θα μας επιτρέψει να γράψουμε πιο σύνθετα προγράμματα χρησιμοποιώντας *βοηθητικές παραμέτρους*. Η βοηθητική παράμετρος είναι μία επιπρόσθετη παράμετρος στην μέθοδό μας που μας επιτρέπει να περάσουμε περισσότερες πληροφορίες στην αναδρομική κλήση.

Για παράδειγμα, φανταστείτε ότι δημιουργείτε την εικόνα fig. 26, η οποία απεικονίζει μία σειρά από κουτιά, το μέγεθος των οποίων αυξάνεται καθώς τοποθετούνται στην σειρά.

Figure 26: Κουτιά που μεγαλώνουν με κάθε αναδρομική κλήση.

Πώς μπορούμε να φτιάξουμε αυτήν την εικόνα;

Ξέρουμε ότι πρέπει να είναι μία δομημένη αναδρομή με φυσικούς αριθμούς και άρα μπορούμε αμέσως να γράψουμε τον σκελετό της μεθόδου

```
def growingBoxes(count: Int): Image =
  count match {
    case 0 => base
```

```
    case n => unit add growingBoxes(n-1)
  }
```

Χρησιμοποιώντας αυτά που μάθαμε προηγουμένως, μπορούμε να γράψουμε το παρακάτω

```
def growingBoxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => Image.rectangle(???,???) beside growin
gBoxes(n-1)
  }
```

Η πρόκληση βρίσκεται στο πώς θα κάνουμε το κουτί να μεγαλώνει καθώς κινείται προς τα δεξιά.

Υπάρχουν δύο τρόποι για να τα καταφέρουμε. Ο πρώτος τρόπος είναι να αλλάξουμε την σειρά στην αναδρομική περίπτωση και να ορίσουμε το μέγεθος των κουτιών σε συνάρτηση με το `n`. Δείτε παρακάτω τον αντίστοιχο κώδικα.

```
def growingBoxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => growingBoxes(n-1) beside Image.rectang
le(n*10, n*10)
  }
// growingBoxes: (count: Int)doodle.core.Image
```

Αφιερώστε λίγο χρόνο στην κατανόηση του παραπάνω κώδικα πριν προχωρήσουμε στην λύση που θα χρησιμοποιεί βοηθητική παράμετρο.

Η χρήση βοηθητικής παραμέτρου σημαίνει απλώς την πρόσθεση μίας ακόμη παραμέτρου στην μέθοδο `growingBoxes` η οποία θα καθορίζει το πόσο μεγάλο πρέπει να είναι το κάθε κουτί. Το μέγεθος αλλάζει καθώς πραγματοποιείται η αναδρομή. Δείτε τον παρακάτω κώδικα.

```
def growingBoxes(count: Int, size: Int): Image =
  count match {
```

```
    case 0 => Image.empty
    case n => Image.rectangle(size, size) beside gro
wingBoxes(n-1, size + 10)
  }
// growingBoxes: (count: Int, size: Int)doodle.core.
Image
```

Η χρήση της βοηθητικής παραμέτρου έχει δύο πλεονεκτήματα: το μόνο που πρέπει να σκεφτούμε είναι αυτό που αλλάζει από την μία αναδρομική κλήση στην άλλη (στην συγκεκριμένη περίπτωση μεγαλώνει το μέγεθος του κουτιού) και επιτρέπει σ' αυτόν που κάνει την κλήση της μεθόδου να αλλάξει την επιπρόσθετη αυτή παράμετρο (για παράδειγμα, μπορεί να κάνει το αρχικό κουτί μικρότερο ή μεγαλύτερο).

Τώρα που είδαμε την μέθοδο της βοηθητικής παραμέτρου, ας εξασκηθούμε στην χρήση της.

Κουτιά που Αλλάζουν Χρώμα

Σ' αυτή την άσκηση θα δημιουργήσουμε την εικόνα fig. 27. Γνωρίζουμε ήδη πώς να φτιάξουμε μία σειρά από κουτιά. Η πρόκληση σ' αυτή την άσκηση είναι να κάνουμε το χρώμα να αλλάζει σε κάθε βήμα.

Βοήθεια: μπορείτε να χρησιμοποιήσετε το `spin` για να χρωματίσετε τα κουτιά με αναδρομή.

Figure 27: Πέντε κουτιά με διαφορετικά χρώματα ξεκινώντας από το Royal Blue

See the solution

Ομόκεντροι κύκλοι

Τώρα ας κάνουμε μία παραλλαγή και ας ζωγραφίσουμε ομόκεντρους κύκλους όπως αυτούς που φαίνονται στην εικόνα fig. 28. Σ' αυτή την περίπτωση αλλάζουμε το μέγεθος σε κάθε βήμα και όχι το χρώμα. Αλλιώς το σχέδιο θα μένει το ίδιο. Προσπαθήστε να το φτιάξετε μόνοι σας.

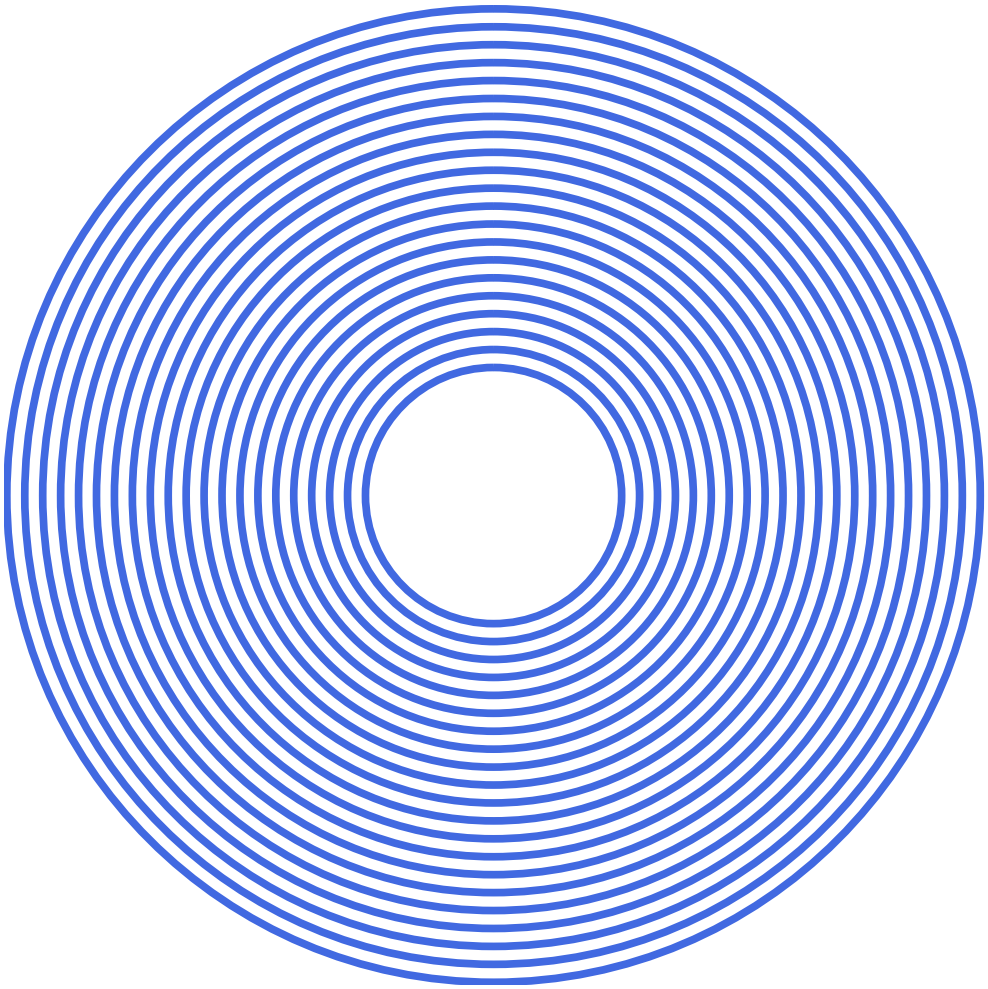


Figure 28: Ομόκεντροι κύκλοι σε χρώμα Royal Blue

See the solution

Άλλη μία Άσκηση

Τώρα ας συνδυάσουμε τις δύο τεχνικές για να αλλάξουμε το μέγεθος και το χρώμα σε κάθε βήμα, για να πάρουμε ως αποτέλεσμα την εικόνα fig. 29. Πειραματιστείτε μέχρι να βρείτε κάτι που σας αρέσει.

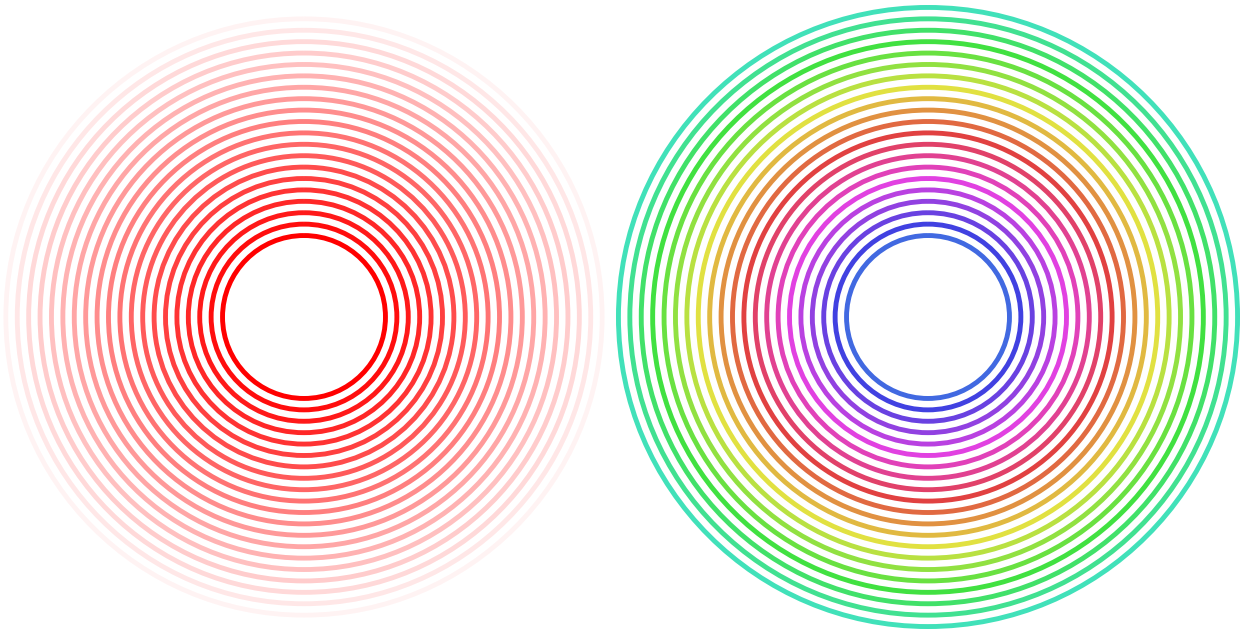


Figure 29: Ομόκεντροι κύκλοι διαφορετικών χρωμάτων

See the solution

7.7 Εμφωλευμένες Μέθοδοι

Μία μέθοδος είναι μία δήλωση. Το σώμα μίας μεθόδου μπορεί να περιέχει δηλώσεις και εκφράσεις. Άρα, η δήλωση μίας μεθόδου μπορεί να περιέχει δηλώσεις άλλων μεθόδων.

Για να καταλάβουμε γιατί κάτι τέτοιο είναι χρήσιμο, ας δούμε μία μέθοδο που γράψαμε προηγουμένως:

```
def cross(count: Int): Image = {
  val unit = Image.circle(20)
  count match {
    case 0 => unit
    case n => unit beside (unit above cross(n-1) abo
ve unit) beside unit
  }
}
// cross: (count: Int)doodle.core.Image
```

Έχουμε δηλώσει την `unit` μέσα στην μέθοδο `cross`. Αυτό σημαίνει ότι η δήλωση της `unit` έχει στην εμβέλειά της μόνο το σώμα της `cross`. Ο

περιορισμός της εμβέλειας των δηλώσεων στα σημεία στα οποία χρησιμοποιούνται αποτελεί μία καλή τακτική για να αποφευχθεί η κατά λάθος επισκίαση άλλων δηλώσεων. Όμως αν σκεφτούμε την συμπεριφορά της `cross` κατά την διάρκεια της εκτέλεσης, θα δούμε ότι έχει κάποια ανεπιθύμητα χαρακτηριστικά.

Θα χρησιμοποιήσουμε το μοντέλο αντικατάστασης για να επεκτείνουμε την `cross(1)`.

```
cross(1)
// Επεκτείνεται σε
{
  val unit = Image.circle(20)
  1 match {
    case 0 => unit
    case n => unit beside (unit above cross(n-1) abo
ve unit) beside unit
  }
}
// Επεκτείνεται σε
{
  val unit = Image.circle(20)
  unit beside (unit above cross(0) above unit) besid
e unit
}
// Επεκτείνεται σε
{
  val unit = Image.circle(20)
  unit beside (unit above
  {
    val unit = Image.circle(20)
    0 match {
      case 0 => unit
      case n => unit beside (unit above cross(n-1) a
bove unit) beside unit
    }
  }
  above unit) beside unit
```

```
}
// Επεκτείνεται σε
{
  val unit = Image.circle(20)
  unit beside (unit above
  {
    val unit = Image.circle(20)
    unit
  }
  above unit) beside unit
}
```

Το νόημα αυτής της τεράστιας επέκτασης είναι να δείξουμε ότι δημιουργούμε την `unit` κάθε φορά που κάνουμε αναδρομή μέσα στην `cross`. Μπορούμε να αποδείξουμε ότι όντως συμβαίνει έτσι, εκτυπώνοντας κάτι, κάθε φορά που δημιουργείται η `unit`.

```
def cross(count: Int): Image = {
  val unit = {
    println("Creating unit")
    Image.circle(20)
  }
  count match {
    case 0 => unit
    case n => unit beside (unit above cross(n-1) abo
ve unit) beside unit
  }
}
// cross: (count: Int)doodle.core.Image

cross(1)
// Creating unit
// Creating unit
// res0: doodle.core.Image = Beside(Beside(Circle(20
.0),Above(Above(Circle(20.0),Circle(20.0)),Circle(20
.0))),Circle(20.0))
```

Η εκτύπωση του “Creating unit”, δεν έχει πολύ μεγάλη σημασία για την `unit`, αφού είναι πολύ μικρή. Παρόλα αυτά, είναι καλό να αποφεύγεται όπου δεν είναι απαραίτητο γιατί καταναλώνεται πολύ μνήμη και χρόνος.

Μπορούμε να το λύσουμε μετακινώντας την `unit` έξω από την `cross`.

```
val unit = {
  println("Creating unit")
  Image.circle(20)
}
// Creating unit
// unit: doodle.core.Image = Circle(20.0)

def cross(count: Int): Image = {
  count match {
    case 0 => unit
    case n => unit beside (unit above cross(n-1) abo
ve unit) beside unit
  }
}
// cross: (count: Int)doodle.core.Image

cross(1)
// res1: doodle.core.Image = Beside(Beside(Circle(20
.0),Above(Above(Circle(20.0),Circle(20.0)),Circle(20
.0))),Circle(20.0))
```

Αυτό δεν είναι επιθυμητό γιατί η `unit` έχει μεγαλύτερη εμβέλεια από όσο χρειάζεται. Μία καλύτερη λύση είναι να χρησιμοποιηθεί μία εμφωλευμένη ή εσωτερική μέθοδος.

```
def cross(count: Int): Image = {
  val unit = {
    println("Creating unit")
    Image.circle(20)
  }
  def loop(count: Int): Image = {
    count match {
```

```
      case 0 => unit
      case n => unit beside (unit above loop(n-1) ab
ove unit) beside unit
    }
  }

  loop(count)
}
// cross: (count: Int)doodle.core.Image

cross(1)
// Creating unit
// res2: doodle.core.Image = Beside(Beside(Circle(20
.0),Above(Above(Circle(20.0),Circle(20.0)),Circle(20
.0))),Circle(20.0))
```

Έτσι επιτυγχάνουμε την συμπεριφορά που θέλουμε, δημιουργώντας την `unit` μόνο μία φορά και ελαχιστοποιώντας την εμβέλειά της. Η εσωτερική μέθοδος `loop` χρησιμοποιεί δομημένη αναδρομή ακριβώς όπως είδαμε προηγουμένως. Το μόνο που πρέπει να κάνουμε είναι να σιγουρευτούμε ότι την καλούμε μέσα στην `cross`. Συνήθως ονομάζουμε αυτού του είδους τις μεθόδους `loop` (βρόγχος) ή `iter` (επανάληψη) (συντομία του iterate) ώστε να δείξουμε ότι εκτελείται ένας βρόγχος.

Αυτή η τεχνική είναι απλώς μία μικρή παραλλαγή των όσων έχουμε ήδη κάνει μέχρι τώρα. Ας δούμε μερικές ασκήσεις για να σιγουρευτούμε ότι μπορούμε να την χειριστούμε.

Ασκήσεις

Σκακιέρα

Ξαναγράψτε την μέθοδο `chessboard` χρησιμοποιώντας μία εμφωλευμένη μέθοδο ώστε τα `blackSquare`, `redSquare` και `base` να δημιουργούνται μόνο μία φορά, όταν καλείται η `chessboard`.

```
def chessboard(count: Int): Image = {
  val blackSquare = Image.rectangle(30, 30) fillColo
```

```
r Color.black
  val redSquare   = Image.rectangle(30, 30) fillColo
r Color.red

  val base =
    (redSquare   beside blackSquare) above (blackSqu
are beside redSquare)
  count match {
    case 0 => base
    case n =>
      val unit = cross(n-1)
      (unit beside unit) above (unit beside unit)
  }
}
// chessboard: (count: Int)doodle.core.Image
```

See the solution

Έξυπνα Κουτιά

Ξαναγράψτε την `boxes`, η οποία φαίνεται παρακάτω, έτσι ώστε το `aBox` να έχει ως εμβέλεια μόνο την `boxes` και να δημιουργείται μόνο μία φορά όταν καλείται η `boxes`.

```
val aBox = Image.rectangle(20, 20).fillColor(Color.r
oyalBlue)

def boxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
```

See the solution

7.8 Συμπεράσματα

Σ' αυτό το κεφάλαιο είδαμε την *δομημένη αναδρομή με φυσικούς*

αριθμούς. Είδαμε επίσης πολλά παραδείγματα για την δημιουργία εικόνων. Μπορούμε να χρησιμοποιήσουμε την δομημένη αναδρομή με φυσικούς αριθμούς για οτιδήποτε μετατρέπει έναν φυσικό αριθμό σε κάτι άλλο (συμπεριλαμβανομένου και άλλων φυσικών αριθμών).

Θα χρησιμοποιήσουμε αυτή τη δομή καθώς και άλλες παραλλαγές της σε όλο το υπόλοιπο βιβλίο.

8 Κηπουρική και Higher-order Συναρτήσεις

Σ' αυτό το κεφάλαιο θα μάθουμε πώς να ζωγραφίζουμε λουλούδια και πώς να χρησιμοποιούμε τις συναρτήσεις ως τιμές πρώτης-τάξεως.

Γνωρίζουμε οτι τα προγράμματα λειτουργούν χρησιμοποιώντας τιμές αλλά δεν είναι όλες *πρώτης-τάξεως*. Μία πρώτης-τάξεως τιμή είναι κάτι που περνάμε ως παράμετρο σε μία μέθοδο ή κάτι που επιστρέφεται ως αποτέλεσμα μίας κλήσης μεθόδου.

Αν περάσουμε μία συνάρτηση ως παράμετρο σε μία άλλη συνάρτηση, τότε:

- η συνάρτηση που περνάμε χρησιμοποιείται ως τιμή πρώτης-τάξεως, και
- η συνάρτηση η οποία δέχεται αυτή την παράμετρο ονομάζεται *higher-order συνάρτηση*.

Η τεχνολογία που μόλις περιγράψαμε δεν είναι πολύ σημαντική αλλά θα την συναντήσετε και σε άλλα βιβλία, οπότε θεωρήσαμε ότι είναι καλό να την γνωρίζετε (έστω και ελάχιστα). Αφού δούμε κάποια παραδείγματα θα γίνει πιό ξεκάθαρη.

Μέχρι τώρα θεωρούσαμε τους όρους *συνάρτηση* και *μέθοδος* ταυτόσημους. Σύντομα όμως θα δούμε ότι αυτοί οι όροι έχουν διαφορετική σημασία.

Προχωρώντας παρακάτω θα δούμε:

- πώς δημιουργούμε συναρτήσεις στην Scala, και
- πώς χρησιμοποιούμε συναρτήσεις πρώτης-τάξεως σε δομημένα προγράμματα.

Για να σας δώσουμε κίνητρο να ασχοληθείτε, το παράδειγμά μας θα είναι ο σχεδιασμός λουλουδιών όπως αυτό που φαίνεται στην εικόνα fig. 30.

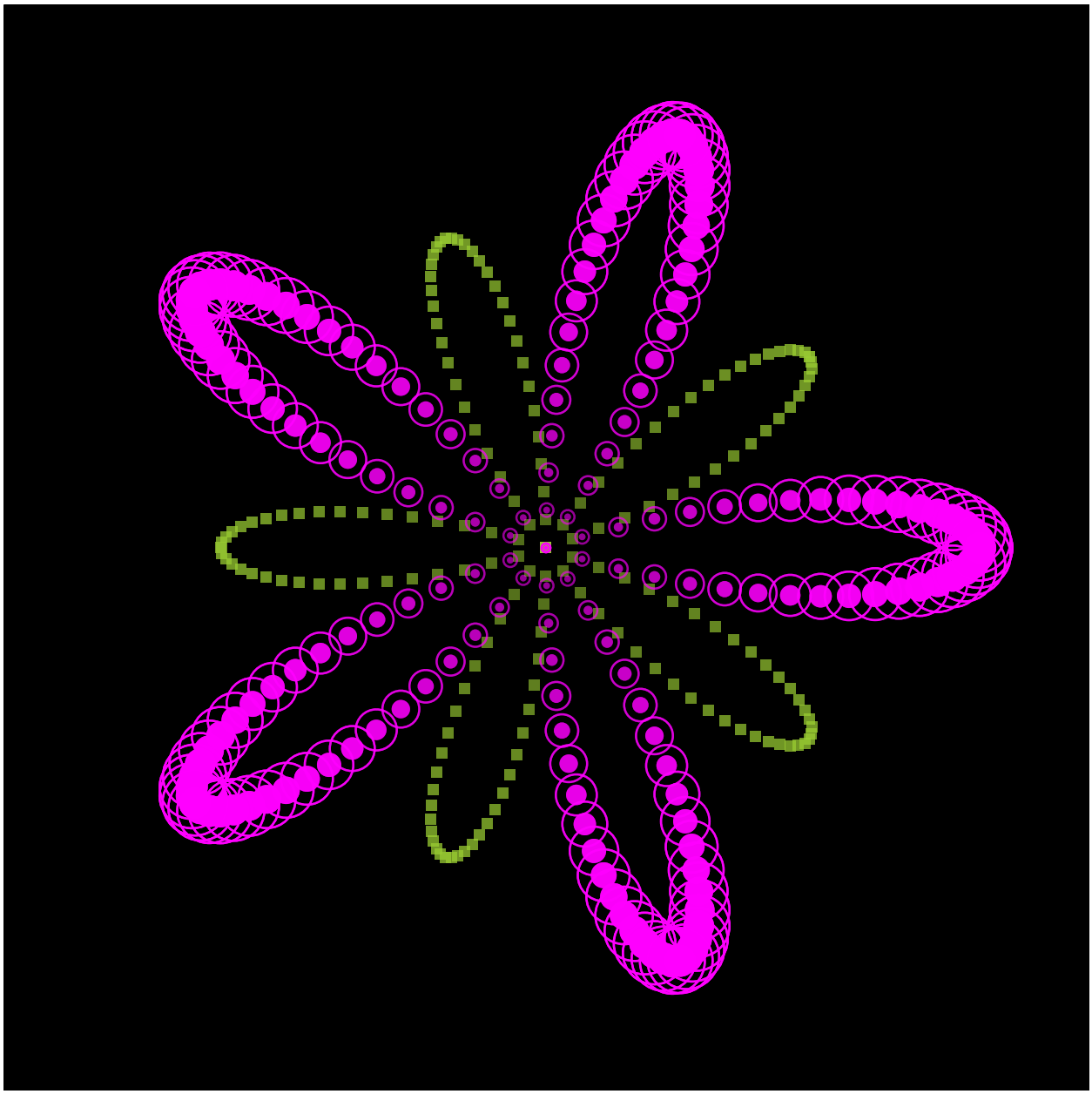


Figure 30: Λουλούδι που δημιουργήθηκε χρησιμοποιώντας τις τεχνικές αυτού του κεφαλαίου

Τα προγράμματα θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που βρίσκεται μέσα στο Doodle. Αν όχι, τότε θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.

```
import doodle.core._
import doodle.core.Image._
```

```
import doodle.syntax._
import doodle.jvm.Java2DFrame._
import doodle.backend.StandardInterpreter._
```

8.1 Παραμετρικές Καμπύλες

Μέχρι τώρα, ξέρουμε μόνο πώς να φτιάχνουμε βασικά σχήματα όπως κύκλους και ορθογώνια. Ο σκοπός μας είναι να σχεδιάσουμε λουλούδια αλλά για να το κάνουμε αυτό θα χρειαστούμε περισσότερα εργαλεία. Γι' αυτο, θα χρησιμοποιήσουμε ένα μαθηματικό εργαλείο γνωστό ως *παραμετρική εξίσωση* ή *παραμετρική καμπύλη*.

Μία παραμετρική εξίσωση είναι μία συνάρτηση που δέχεται μία είσοδο (παράμετρος) και επιστρέφει ένα σημείο (τοποθεσία στον χώρο). Για παράδειγμα, η παραμετρική εξίσωση ενός κύκλου είναι μία συνάρτηση που δέχεται μία `Angle` (γωνία) και επιστρέφει ένα σημείο.

```
def parametricCircle(angle: Angle): Point =
  ???
```

Μπορούμε να κατασκευάσουμε μικρές κουκκίδες ή άλλα σχήματα σ' αυτά τα σημεία και έτσι να δημιουργήσουμε μεγάλα σχήματα όπως αυτά που θέλουμε να ζωγραφίσουμε.

Στην εικόνα fig. 31 μπορείτε να δείτε ένα παράδειγμα μικρών κύκλων που έχουν δημιουργηθεί με την παραμετρική εξίσωση κύκλου. Πηγαίνοντας από τα αριστερά προς τα δεξιά, ζωγραφίζουμε σημεία κάθε 90, 45, και 22.5 μοίρες. Μπορείτε να δείτε το περίγραμμα του σχήματος να γίνεται όλο και πιο έντονο καθώς ζωγραφίζουμε περισσότερα σημεία.

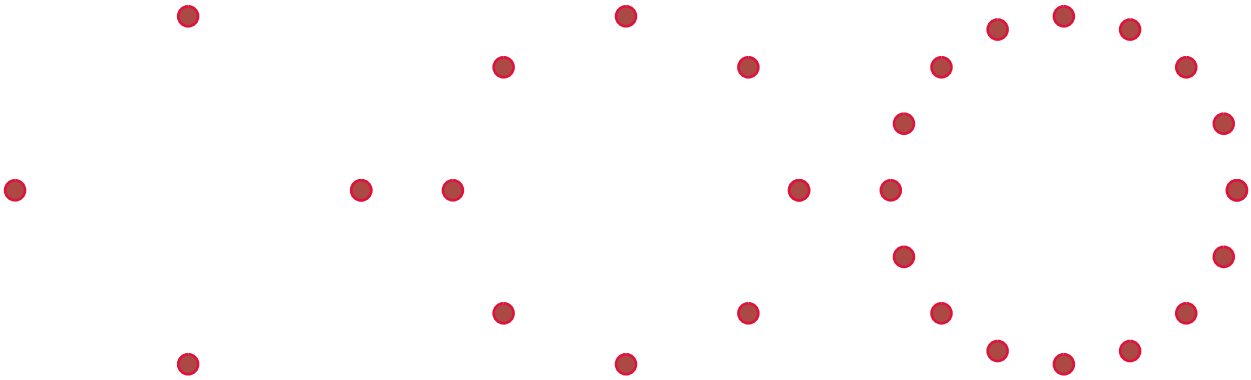


Figure 31: Παραμετρικός κύκλος που δημιουργείται από τα αριστερά προς τα δεξιά με σημεία σε κάθε 90, 45, και 22.5 μοίρες.

Για να δημιουργήσουμε παραμετρικές καμπύλες πρέπει να μάθουμε πώς να φτιάχνουμε σημεία με το Doodle, πώς να εμφανίζουμε εικόνες σε συγκεκριμένο σημείο στον χώρο και να θυμηθούμε λίγη γεωμετρία γυμνασίου.

8.2 Σημεία

Στο Doodle, για αναπαράσταση μίας θέσης σε δισδιάστατο χώρο, υπάρχει ο τύπος `Point`. Έχουμε στην διάθεσή μας δύο ισοδύναμες αναπαραστάσεις:

- οι συντεταγμένες x και y, δηλαδή το καρτεσιανό σύστημα αναπαράστασης συντεταγμένων, και
- η αναπαράσταση με πολικές συντεταγμένες όπου χρησιμοποιείται μία γωνία και η απόστασή της από ένα αρχικό σημείο (δηλαδή η ακτίνα).

Μπορούμε να δημιουργήσουμε σημεία καρτεσιανής αναπαράστασης χρησιμοποιώντας το `Point.cartesian` και για την πολική αναπαράσταση μπορούμε να χρησιμοποιήσουμε το `Point.polar`. Στον παρακάτω πίνακα μπορείτε να δείτε τις βασικές μεθόδους του `Point`.

Τελεστής	Τύπος	Περιγραφή	Παράδειγμα
`Point.cartesian(Double, Double)`	`Point`	Κατασκευάζει ένα `Point` (σημείο) χρησιμοποιώντας	`Point.cartesian(1. 1.0)`

		καρτεσιανή αναπαράσταση	
`Point.polar(Double, Angle)` `Point(Double, Angle)`	`Point`	Κατασκευάζει ένα `Point` χρησιμοποιώντας πολική αναπαράσταση	`Point.polar(1.0, 90.degrees)`
`Point.zero`	`Point`	Κατασκευάζει ένα `Point` όπου τα x και y είναι 0	`Point.zero`
`Point.x`	`Double`	Επιστρέφει την συντεταγμένη x του `Point`.	`Point.zero.x`
`Point.y`	`Double`	Επιστρέφει την συντεταγμένη y του `Point`.	`Point.zero.y`
`Point.r`	`Double`	Επιστρέφει την ακτίνα του `Point`.	`Point.zero.r`
`Point.angle`	`Angle`	Επιστρέφει την γωνία του `Point`.	`Point.zero.angle`

8.3 Ευέλικτη Διάταξη

Μπορούμε να τοποθετήσουμε μία εικόνα σ' ένα συγκεκριμένο σημείο; Μέχρι τώρα ξέρουμε μόνο πώς διατάσσουμε εικόνες χρησιμοποιώντας τα `on`, `beside`, και `above`. Για να πετύχουμε μεγαλύτερη ευελιξία στον τρόπο διάταξης, χρειαζόμαστε ακόμη ένα εργαλείο, την μέθοδο `at`. Παρακάτω μπορείτε να δείτε ένα παράδειγμα σχεδιασμού κύκλων στις γωνίες ενός τετραγώνου.

```
val dot = Image.circle(5).lineWidth(3).lineColor(Col
or.crimson)
val squareDots =
  dot.at(0, 0).
    on(dot.at(0, 100)).
    on(dot.at(100, 100)).
    on(dot.at(100, 0))
```

Ο παραπάνω κώδικας δημιουργεί την εικόνα fig. 32.

Figure 32: Χρήση της `at` για την τοποθέτηση τεσσάρων κουκκίδων στις γωνίες ενός τετραγώνου.

Για να καταλάβετε πώς λειτουργεί το `at` και γιατί πρέπει να τοποθετήσουμε τις κουκκίδες τη μία πάνω στην άλλη χρησιμοποιώντας το `on`, πρέπει να ξέρουμε πώς χειρίζεται τις διατάξεις το Doodle.

Κάθε εικόνα στο Doodle έχει μία *αρχική θέση*. Για τις περισσότερες περιπτώσεις η θέση αυτή βρίσκεται στο κέντρο, χωρίς αυτό να είναι πάντα απαραίτητο. Όταν μία διάταξη του Doodle συνδυάζει εικόνες, τοποθετεί σε μία ευθεία γραμμή τις αρχικές τους θέσεις. Για παράδειγμα, αν μερικές εικόνες έχουν διαταχθεί με την χρήση του `above` τότε οι αρχικές τους θέσεις βρίσκονται σε μία κάθετη ευθεία και η αρχική θέση της σύνθετης εικόνας βρίσκεται στην μέση της νοητής γραμμής που συνδέει τις αρχικές θέσεις των μεμονωμένων εικόνων. Στην εικόνα fig. 33 μπορείτε να δείτε ένα παράδειγμα διάταξης με `beside` στο οποίο φαίνεται το πώς ευθυγραμμίζονται οι αρχικές θέσεις (οι κόκκινοι κύκλοι). Τέλος, με το `on` οι αρχικές θέσεις τοποθετούνται η μία πάνω στην άλλη έτσι ώστε να μοιράζονται μία κοινή αρχική θέση.

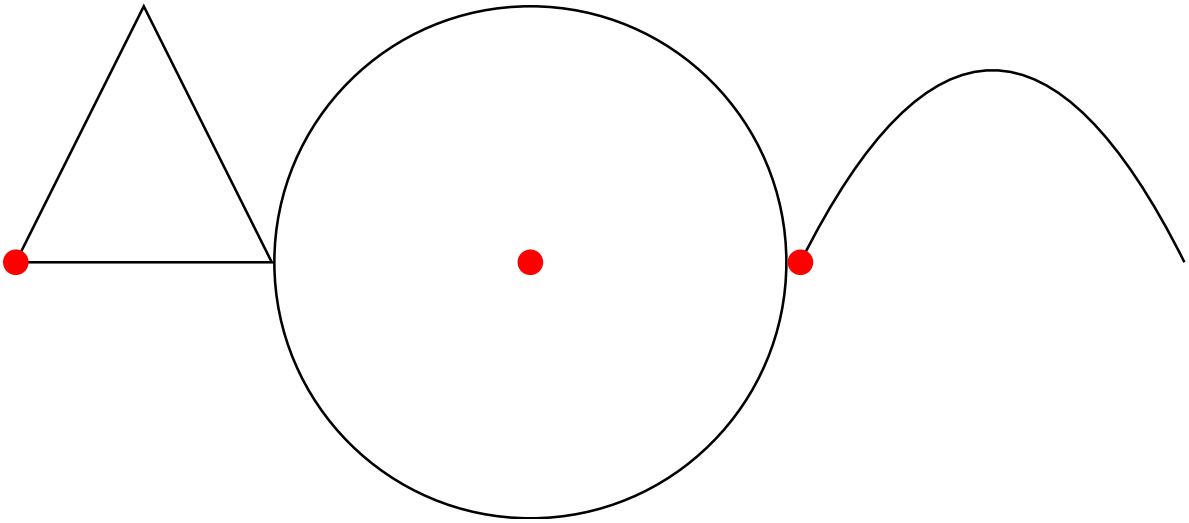


Figure 33: Ένα παράδειγμα οριζόντιας διάταξης (με χρήση του `beside`), στο οποίο φαίνεται πώς ευθυγραμμίζονται οι αρχικές θέσεις.

Χρησιμοποιώντας το `at` μπορούμε να μετακινήσουμε μία εικόνα σε άλλη θέση από την αρχική της. Στα παραδείγματα που ακολουθούν θέλουμε όλα τα στοιχεία της εικόνας να μοιράζονται μία κοινή αρχική θέση και γι' αυτό χρησιμοποιήσαμε το `on` ώστε να συνδυάσουμε εικόνες που έχουμε μετακινήσει χρησιμοποιώντας το `at`.

Υπάρχουν δύο τρόποι κλήσης του `at`:

- περνώντας τις συντεταγμένες x και y, όπως για παράδειγμα `dot.at(100, 100)`, ή
- περνώντας ένα `Vec` (διάνυσμα) το οποίο δίνει τις συντεταγμένες `dot.at(Vec(100, 100))`.

Μπορούμε να μετατρέψουμε ένα `Point` σε `Vec` χρησιμοποιώντας την μέθοδο `toVec`.

```
Point.cartesian(1.0, 1.0).toVec
// res0: doodle.core.Vec = Vec(1.0,1.0)
```

8.4 Γεωμετρία

Το τελευταίο στοιχείο που χρειάζεται να ξέρουμε για να τοποθετήσουμε τις εικόνες σε διάφορα σημεία, είναι λίγη γεωμετρία. Αν ένα σημείο είναι τοποθετημένο σε απόσταση `r` από το αρχικό σημείο και σε γωνία `a`, οι

συντεταγμένες x και y είναι `(a.cos) * r` και `(a.sin) * r` αντίστοιχα. Εναλλακτικά μπορούμε απλώς να χρησιμοποιήσουμε πολικές συντεταγμένες!

```
val polar = Point.polar(1.0, 45.degrees)
// polar: doodle.core.Point = Polar(1.0,Angle(0.7853
981633974483))

val cartesian = Point.cartesian((45.degrees.cos) * 1
.0, (45.degrees.sin) * 1.0)
// cartesian: doodle.core.Point = Cartesian(0.707106
7811865476,0.7071067811865475)

// Είναι ίδια
polar.toCartesian == cartesian
// res2: Boolean = true

cartesian.toPolar == polar
// res3: Boolean = true
```

8.5 Συνδυάζοντας

Μπορούμε να συνδιάσουμε τα παραπάνω, ώστε να δημιουργήσουμε έναν παραμετρικό κύκλο. Ο κώδικας ενός παραμετρικού κύκλου με ακτίνα 200 σε καρτεσιανές συντεταγμένες είναι ο παρακάτω

```
def parametricCircle(angle: Angle): Point =
  Point.cartesian(angle.cos * 200, angle.sin * 200)
```

Σε πολική μορφή είναι ο παρακάτω

```
def parametricCircle(angle: Angle): Point =
  Point.polar(200, angle)
```

Μπορούμε ακόμη να διαλέξουμε ομοιόμορφα κατανεμημένα σημεία του κύκλου. Ζωγραφίζοντας κάτι σε κάθε σημείο μπορούμε να δημιουργήσουμε μία εικόνα (για παράδειγμα ένα τρίγωνο).

```
def sample(start: Angle, samples: Int): Image = {
  // Το Angle.one είναι μια ολόκληρη περιστροφή. Δηλ
αδή 360 μοίρες
  val step = Angle.one / samples
  val dot = triangle(10, 10)
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n =>
        dot.at(parametricCircle(angle).toVec) on loo
p(n - 1)
    }
  }

  loop(samples)
}
```

Το παραπάνω είναι μία δομημένη αναδρομή που ελπίζουμε να σας φαίνεται αρκετά οικεία.

Αν το εκτελέσουμε, θα δούμε να σχηματίζεται ένας κύκλος με την βοήθεια τριγώνων. Δείτε την εικόνα fig. 34, στην οποία φαίνονται τα αποτελέσματα του `sample(0.degrees, 72)`.

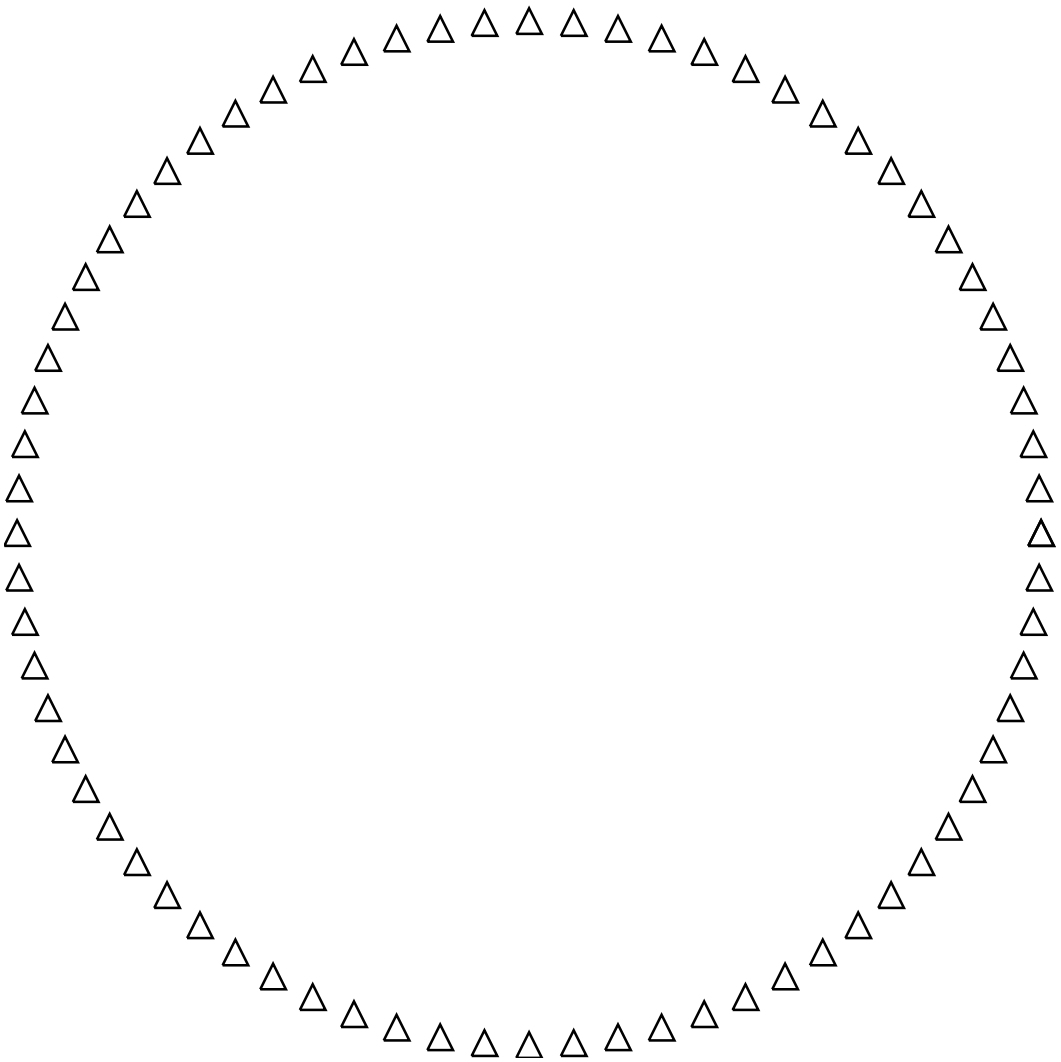


Figure 34: Τρίγωνα σε κύκλο που προέκυψαν από τον κώδικα `sample`.

8.5.1 Λουλούδια

Το επόμενο βήμα για την δημιουργία του λουλουδιού είναι η χρήση ενός πιο ενδιαφέροντος σχήματος από τον κύκλο. Αυτό σημαίνει ότι θα πρέπει να αλλάξουμε το `parametricCircle` σε μία πιο ενδιαφέρουσα εξίσωση. Ίσως σαν την `rose` που μπορείτε να δείτε παρακάτω. Αυτό είναι ένα παράδειγμα καμπύλης τριαντάφυλλου, με μέγιστη ακτίνα 200. Μπορούμε να αλλάξουμε την τιμή που πολλαπλασιάζεται με την γωνία (στην συγκεκριμένη περίπτωση είναι 7) ώστε να πάρουμε ένα διαφορετικό σχήμα.

```
// Παραμετρική εξίσωση για το τριαντάφυλλο με k = 7
```

```
def rose(angle: Angle) =
  Point.polar((angle * 7).cos * 200, angle)
```

Μπορείτε να δείτε ένα παράδειγμα στην εικόνα fig. 35.

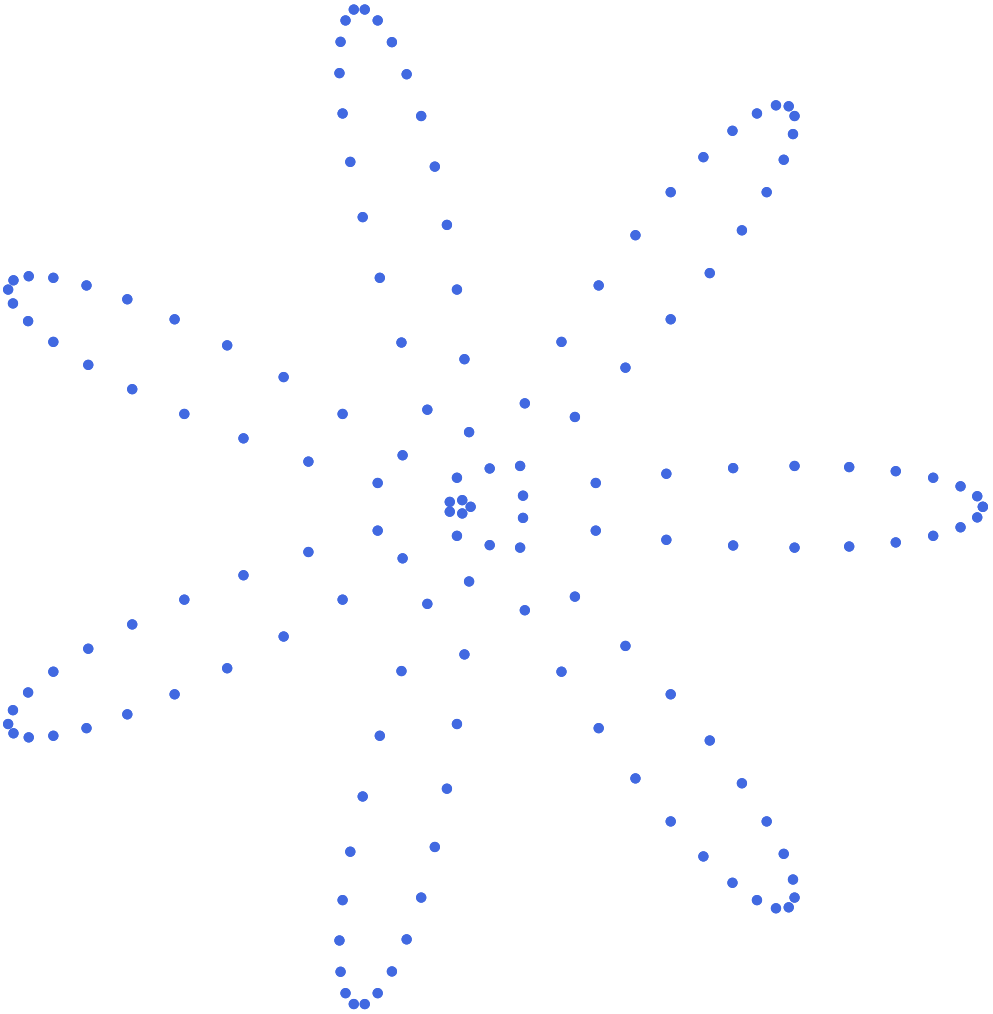


Figure 35: Ένα παράδειγμα της καμπύλης του τριαντάφυλλου.

Μπορούμε να αλλάξουμε την `sample` ώστε να καλεί την `rose` αντί για την `parametricCircle` αλλά και πάλι το αποτέλεσμα δεν θα είναι ικανοποιητικό. Ας πειραματιστούμε με διαφορετικές παραμετρικές εξισώσεις! Θα ήταν πολύ βολικό αν μπορούσαμε να περάσουμε ως παράμετρο στην `sample` την μέθοδο που δημιουργεί σημεία (δηλαδή την παραμετρική εξίσωση). Μπορούμε όμως να κάνουμε κάτι τέτοιο; Για να το κάνουμε πρέπει:

- να γράψουμε τον τύπο μίας μεθόδου
- να διαχωρίσουμε την κλήση μίας μεθόδου (πχ την `rose(0.degrees)`)

από την μέθοδο καθαυτήν.

Ας δούμε το δεύτερο πρόβλημα. Αν προσπαθήσουμε να αναφερθούμε σε μία μέθοδο χωρίς να την καλέσουμε θα λάβουμε ένα μήνυμα λάθους.

```
rose
// <console>:29: error: missing argument list for me
thod rose
// Unapplied methods are only converted to functions
 when a function type is expected.
// You can make this conversion explicit by writing
`rose _` or `rose(_)` instead of `rose`.
//        rose
//        ^
```

Το μήνυμα λάθους μας λέει από μόνο του τι πρέπει να διορθώσουμε και αυτό είναι ένα πολύ καλό σημείο για να εισάγουμε επιτέλους τις συναρτήσεις.

8.6 Συναρτήσεις

Όπως μας υποδεικνύει και το μήνυμα λάθους που είδαμε στην προηγούμενη ενότητα, μπορούμε να μετατρέψουμε μία μέθοδο σε συνάρτηση χρησιμοποιώντας τον τελεστή `_` και να την καλέσουμε με τις ίδιες παραμέτρους.

```
// Παραμετρική εξίσωση για το τριαντάφυλλο με k = 7
def rose(angle: Angle) =
  Point.cartesian((angle * 7).cos * angle.cos, (angl
e * 7).cos * angle.sin)
```

```
rose _
// res1: doodle.core.Angle => doodle.core.Point = $$
Lambda$16616/1522025270@618fe50d

(rose _)(0.degrees)
// res2: doodle.core.Point = Cartesian(1.0,0.0)
```

Βασικά μία συνάρτηση είναι μία μέθοδος, την οποία όμως μπορούμε να χρησιμοποιήσουμε ως τιμή πρώτης-τάξης:

- μπορούμε να την περάσουμε ως argument ή ως παράμετρο σε μία μέθοδο ή σε μία συνάρτηση,
- μπορούμε να την επιστρέψουμε από μία μέθοδο ή μία συνάρτηση, και
- μπορούμε να της δώσουμε ένα όνομα χρησιμοποιώντας το `val`.

```
val roseFn = rose _
// roseFn: doodle.core.Angle => doodle.core.Point =
$$Lambda$16618/355055240@7b4cc981

roseFn(0.degrees)
// res3: doodle.core.Point = Cartesian(1.0,0.0)
```

8.6.1 Τύποι Συναρτήσεων

Για να περάσουμε μία συνάρτηση σε μία μέθοδο πρέπει να γνωρίζουμε τον τύπο της (αφού όταν δηλώνουμε μία παράμετρο πρέπει να δηλώσουμε και τον τύπο της).

Ένας τύπος συνάρτησης γράφεται ως `(A, B) => C` όπου `A` και `B` είναι οι τύποι των παραμέτρων και `C` ο τύπος του αποτελέσματος. Η ίδια αυτή μορφή μπορεί να γενικευτεί για συναρτήσεις με οποιονδήποτε αριθμό παραμέτρων.

Στο παράδειγμά μας θέλουμε το `f` να είναι μία συνάρτηση η οποία δέχεται δυο παραμέτρους τύπου `Int` και να επιστρέφει έναν `Int`. Έτσι μπορούμε να την γράψουμε και ως `(Int, Int) => Int`.

Δήλωση Τύπου Συνάρτησης

Για να δηλώσουμε τον τύπο μιας συνάρτησης γράφουμε

```
(A, B, ...) => C
```

όπου

- τα `A, B, ...` είναι οι τύποι των εισαγόμενων παραμέτρων, και
- το `C` ο τύπος του αποτελέσματος.

Αν μία συνάρτηση έχει μόνο μία παράμετρο, οι παρενθέσεις δεν είναι απαραίτητες:

```
A => B
```

8.6.2 Κυριολεκτικές Εκφράσεις Συναρτήσεων

Υπάρχει μία κυριολεκτική σύνταξη για συναρτήσεις. Για παράδειγμα, παρακάτω μπορείτε να δείτε μία συνάρτηση η οποία προσθέτει τον αριθμό `42` στην παράμετρο της.

```
(x: Int) => x + 42
// res4: Int => Int = $$Lambda$16619/1477171709@7acfe7e9
```

Μπορούμε να εφαρμόσουμε την συνάρτηση σε ένα argument με τον γνωστό τρόπο.

```
val add42 = (x: Int) => x + 42
// add42: Int => Int = $$Lambda$16620/1362887180@4a447a42

add42(0)
// res5: Int = 42
```

Σύνταξη Κυριολεκτικών Συναρτήσεων

Η δήλωση μιας κυριολεκτικής συνάρτησης είναι η παρακάτω

```
(parameter: type, ...) => expression
```

όπου

- τα προαιρετικά `parameter` είναι τα ονόματα των παραμέτρων της συνάρτησης
- τα `type` είναι οι τύποι των παραμέτρων της
- το `expression` καθορίζει το αποτέλεσμα της.

8.6.3 Συναρτήσεις και Αντικείμενα

Επειδή η Scala είναι αντικειμενοστραφής γλώσσα, όλες οι τιμές πρώτης-τάξης είναι αντικείμενα. Αυτό σημαίνει ότι οι συναρτήσεις μπορούν να έχουν μεθόδους, συμπεριλαμβανομένου μερικών πολύ χρήσιμων για σύνθεση:

```
val addTen = (a: Int) => a + 10
// addTen: Int => Int = $$Lambda$16621/192613224@66d788fd

val double = (a: Int) => a * 2
// double: Int => Int = $$Lambda$16622/1938257021@317a0e35

val combined = addTen andThen double // this composes the two functions
// combined: Int => Int = scala.Function1$$Lambda$3966/1978108880@643f818c

combined(5)
// res6: Int = 30
```

Ασκήσεις

Τύποι Συναρτήσεων

Ποιος είναι ο τύπος της παραπάνω συνάρτησης `roseFn`; Τι σημαίνει αυτός ο τύπος;

See the solution

Κυριολεκτικές Εκφράσεις Συναρτήσεων

Γράψτε την `roseFn` ως κυριολεκτική έκφραση συνάρτησης.

See the solution

8.7 Higher Order Μέθοδοι και Συναρτήσεις

Γιατί είναι χρήσιμες οι συναρτήσεις; Αν θέλουμε να “πακετάρουμε” και να ονομάσουμε ένα επαναχρησιμοποιήσιμο κομμάτι κώδικα, μπορούμε να χρησιμοποιήσουμε μεθόδους. Τι πλεονεκτήματα έχει η χρήση κώδικα ως τιμή; Έχουμε πει ότι:

- μπορούμε να περάσουμε συναρτήσεις ως παραμέτρους σε άλλες συναρτήσεις ή μεθόδους,
- μπορούμε να δημιουργήσουμε μεθόδους οι οποίες επιστρέφουν συναρτήσεις.

Ας πάρουμε ως παράδειγμα την άσκηση με τους ομόκεντρους κύκλους:

```
def concentricCircles(count: Int, size: Int): Image
=
  count match {
    case 0 => Image.empty
    case n => Image.circle(size) on concentricCircle
s(n-1, size + 5)
  }
```

Ο παραπάνω κώδικας μας επιτρέπει να δημιουργήσουμε διαφορετικές εικόνες αλλάζοντας την χρήση της `Image.circle`. Όμως, κάθε φορά που αλλάζουμε τον ορισμό της `Image.circle`, πρέπει να αλλάξουμε και τον ορισμό της `concentricCircles` αντιστοίχως.

Μπορούμε να κάνουμε την `concentricCircles` γενικότερη περνώντας την `Image.circle` ως παράμετρο: Εδώ μετονομάσαμε την `concentricShapes`, αφού δεν περιοριζόμαστε πλέον στο να φτιάχνουμε μόνο κύκλους, και ορίσαμε την `singleShape` ως υπεύθυνη για τον σχεδιασμό ενός σχήματος κατάλληλου μεγέθους.

```
def concentricShapes(count: Int, singleShape: Int =>
 Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n) on concentricShapes(n-1
, singleShape)
  }
```

Τώρα, μπορούμε να επαναχρησιμοποιήσουμε τον ορισμό της `concentricShapes` για να φτιάξουμε απλούς κύκλους, κύκλους διαφορετικής απόχρωσης, κύκλους με διαφορετική διαφάνεια και ούτω καθεξής. Το μόνο που πρέπει να κάνουμε είναι να δώσουμε τον κατάλληλο ορισμό στην `singleShape`:

```
// Απευθείας πέρασμα κυριολεκτικού συνάρτησης:
val blackCircles: Image =
  concentricShapes(10, (n: Int) => Image.circle(50 +
 5*n))

// Μετατροπή μεθόδου σε συνάρτηση:
def redCircle(n: Int): Image =
  Image.circle(50 + 5*n) lineColor Color.red

val redCircles: Image =
  concentricShapes(10, redCircle _)
```

Ασκήσεις

Χρώμα και Σχήμα

Ξεκινώντας με τον παρακάτω κώδικα, γράψτε συναρτήσεις για το χρώμα και το σχήμα ώστε το αποτέλεσμα να είναι η εικόνα fig. 36.

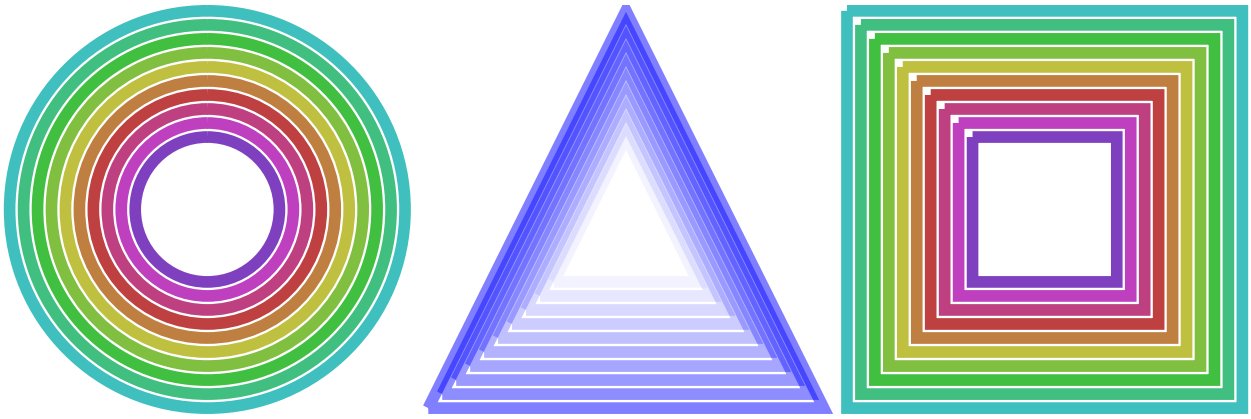


Figure 36: Χρώματα και Σχήματα

```
  def concentricShapes(count: Int, singleShape: Int
=> Image): Image =
    count match {
      case 0 => Image.empty
      case n => singleShape(n) on concentricShapes(n
-1, singleShape)
    }
```

Η μέθοδος `concentricShapes` είναι ισοδύναμη με την μέθοδο `concentricCircles` που είδαμε σε προηγούμενες ασκήσεις. Η διαφορά τους είναι ότι περνάμε τον ορισμό της `singleShape` ως παράμετρο.

Ας σκεφτούμε λίγο το πρόβλημα που μας δίνεται. Πρέπει να κάνουμε δύο πράγματα:

1. να γράψουμε κατάλληλο ορισμό για την `singleShape` για κάθε ένα από τα τρία σχήματα της εικόνας που έχουμε ως στόχο, και
2. να καλέσουμε την `concentricShapes` τρεις φορές, περνώντας κάθε φορά τον αντίστοιχο ορισμό για την `singleShape` και να διατάξουμε τα αποτελέσματα με την `beside` (το ένα δίπλα στο άλλο).

Ας δούμε τον ορισμό της `singleShape` λεπτομερώς. Ο τύπος της παραμέτρου είναι `Int => Image`, που σημαίνει ότι είναι μία συνάρτηση η οποία δέχεται μία παράμετρο `Int` και επιστρέφει ένα `Image`. Μπορούμε να δηλώσουμε μία μέθοδο τέτοιου τύπου όπως παρακάτω:

```
def outlinedCircle(n: Int) =
  Image.circle(n * 10)
```

Μπορούμε να περάσουμε μία παράμετρο στην μέθοδο `concentricShapes` ώστε να δημιουργήσουμε μία εικόνα ομόκεντρων κύκλων με μαύρο περίγραμμα:

```
concentricShapes(10, outlinedCircle _)
```

Έτσι παράγεται το αποτέλεσμα της εικόνας fig. 37.

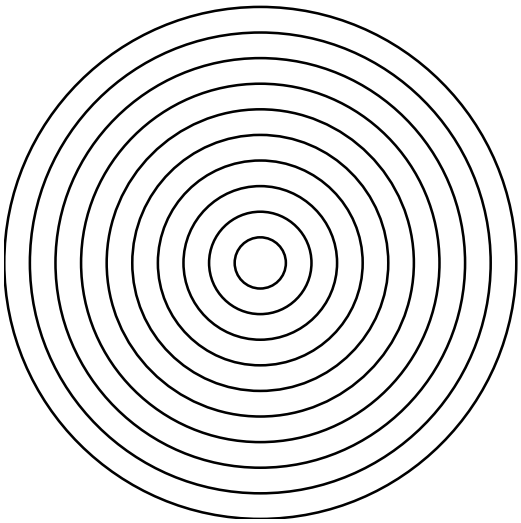


Figure 37: Σχεδιασμένοι κύκλοι

Η επίλυση της άσκησης είναι θέμα αντιγραφής, μετονομασίας και μορφοποίησης της συνάρτησης ώστε να παράγονται οι επιθυμητοί συνδυασμοί χρωμάτων και σχημάτων:

```
def circleOrSquare(n: Int) =
  if(n % 2 == 0) Image.rectangle(n*20, n*20) else Im
age.circle(n*10)

(concentricShapes(10, outlinedCircle) beside concent
ricShapes(10, circleOrSquare))
```

Δείτε το αποτέλεσμα στην εικόνα fig. 38.

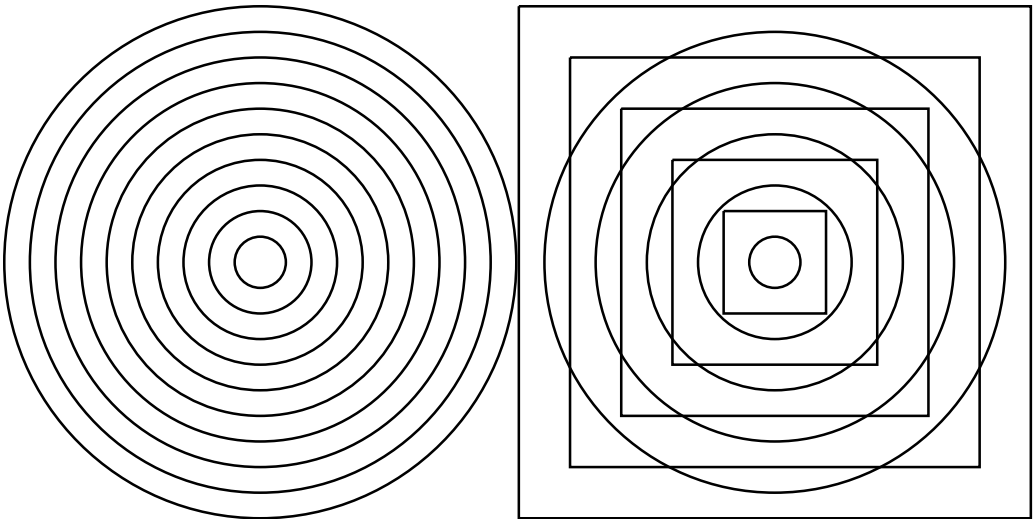


Figure 38: Σχεδιασμένοι κύκλοι δίπλα σε κύκλους και τετράγωνα

Για περισσότερη εξάσκηση, αφού γράψετε τον κώδικα για τα βασικά σχήματα που ζητούνται παραπάνω, αλλάξτε τον έτσι ώστε να έχετε δύο ομάδες βασικών συναρτήσεων—μία που παράγει χρώματα και μία που παράγει σχήματα. Συνδέστε τις συναρτήσεις χρησιμοποιώντας έναν *combinator* όπως μπορείτε να δείτε παρακάτω και χρησιμοποιήστε το αποτέλεσμα ως παράμετρο για την `concentricShapes`

```
def colored(shape: Int => Image, color: Int => Color
): Int => Image =
  (n: Int) => ???
```

See the solution

8.8 Ασκήσεις

Τώρα που έχουμε μάθει τόσα για τις συναρτήσεις, θα επιστρέψουμε στο πρόβλημα της δημιουργίας λουλουδιών. Από εδώ και πέρα θα σας ζητάμε να σχεδιάζετε σχήματα πιο συχνά.

Η εργασία σας θα είναι να σπάσετε την δημιουργία του λουλουδιού σε μικρότερες συναρτήσεις που δουλεύουν μαζί. Αφήστε την δημιουργικότητά σας ελεύθερη καθώς μετατρέπετε το κάθε ξεχωριστό στοιχείο του κώδικα σε συνάρτηση.

Προσπαθήστε να το κάνετε τώρα. Αν αντιμετωπίσετε προβλήματα δείτε παρακάτω τον δικό μας τρόπο.

8.8.1 Ξεχωριστά Στοιχεία

Αναγνωρίσαμε δύο στοιχεία στον σχεδιασμό λουλουδιών:

- την παραμετρική εξίσωση και
- την δομημένη αναδρομή με γωνίες.

Τι άλλα στοιχεία θα μπορούσαμε να βρούμε στις συναρτήσεις; Ποιοι είναι οι τύποι τους; (Αυτή η ερώτηση είναι τόσο γενική εσκεμμένα! Εξερευνήστε μόνοι σας!)

See the solution

8.8.2 Σύνθεση

Τώρα που απομονώσαμε τα στοιχεία, μπορούμε να τα συνδυάσουμε ώστε να δημιουργήσουμε ενδιαφέροντα αποτελέσματα. Προσπαθήστε το.

See the solution

8.8.3 Πείραμα

Τώρα πειραματιστείτε με την δημιουργικότητά σας!

See the solution

9 Σχήματα, Ακολουθίες και Αστέρια

Σε αυτό το κεφάλαιο θα μάθουμε πώς να φτιάχνουμε τα δικά μας σχήματα χρησιμοποιώντας τις βασικές γραμμές και καμπύλες με οποίες φτιάχναμε ως τώρα τρίγωνα, ορθογώνια και κύκλους. Έτσι θα μάθουμε την αναπαράσταση και τον χειρισμό μιας ακολουθίας δεδομένων μέσω higher-order συναρτήσεων που χρησιμοποιούν την μέθοδο της αφαίρεσης πάνω στην δομημένη αναδρομή. Παρουσιάσαμε πολλούς νέους όρους αλλά θα δείτε ότι δεν είναι τόσο δύσκολοι όσο ακούγονται!

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

9.1 Μονοπάτια

Όλα τα σχήματα στο Doodle αναπαριστώνται ως μονοπάτια. Φανταστείτε τα μονοπάτια ως μία ακολουθία κινήσεων ενός φανταστικού μολυβιού, που ξεκινάει να ζωγραφίζει από ένα αρχικό σημείο. Οι κινήσεις του μολυβιού χωρίζονται σε τρεις κατηγορίες:

- κίνηση του μολυβιού σε ένα σημείο χωρίς όμως να ζωγραφίζει γραμμή,
- σχεδιασμός ευθείας γραμμής από την θέση που βρίσκεται μέχρι ένα σημείο και
- σχεδιασμός μίας καμπύλης Bezier από την θέση που βρίσκεται μέχρι ένα σημείο, με την καμπυλότητά της να ορίζεται από δύο *σημεία*

ελέγχου.

Τα μονοπάτια χωρίζονται σε δύο κατηγορίες:

- τα ανοιχτά μονοπάτια, όπου το τέλος του μονοπατιού δεν είναι απαραιτήτως το σημείο εκκίνησης και
- τα κλειστά μονοπάτια, όπου τελειώνουν εκεί από όπου ξεκίνησαν—σε περίπτωση που δεν γίνει έτσι θα προστεθεί μία γραμμή ώστε το μονοπάτι να τελειώνει εκεί από όπου ξεκίνησε.

Στην εικόνα fig. 39 μπορείτε να δείτε τα στοιχεία που απαρτίζουν ένα μονοπάτι καθώς και την διαφορά μεταξύ ανοιχτού και κλειστού μονοπατιού.

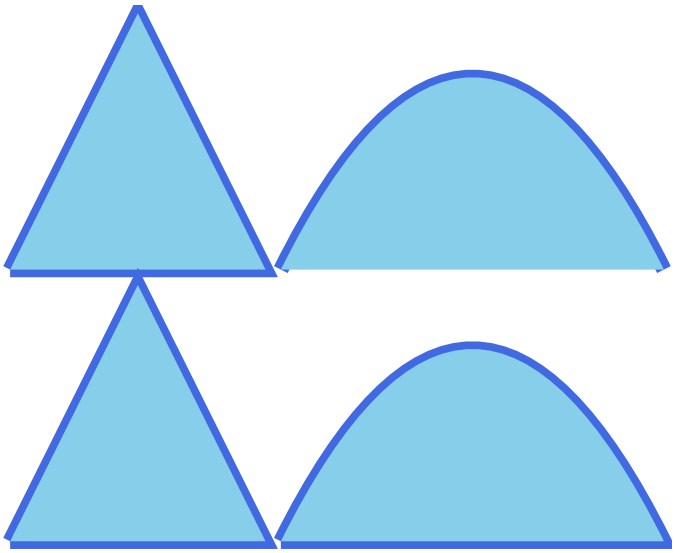


Figure 39: Τα ίδια σχήματα σχεδιασμένα με ανοιχτά (πάνω) και κλειστά (κάτω) μονοπάτια. Παρατηρήστε ότι το ανοιχτό τρίγωνο δεν είναι σωστά ενωμένο στην κάτω αριστερή του γωνία και γι' αυτό προστίθεται μία ευθεία γραμμή ώστε να κλείσει.

9.1.1 Δημιουργία Μονοπατιών

Γνωρίζοντας μερικά πράγματα για τα μονοπάτια, μπορούμε να τα δημιουργήσουμε στο Doodle; Παρακάτω βλέπετε τον κώδικα που δημιούργησε την εικόνα.

```
import doodle.core.Point._
import doodle.core.PathElement._
```

```
val triangle =
  List(
    lineTo(cartesian(50, 100)),
    lineTo(cartesian(100, 0)),
    lineTo(cartesian(0, 0))
  )

val curve =
  List(curveTo(cartesian(50, 100), cartesian(100, 10
0), cartesian(150, 0)))

def style(image: Image): Image =
  image.
    lineWidth(6.0).
    lineColor(Color.royalBlue).
    fillColor(Color.skyBlue)

val openPaths =
  style(openPath(triangle) beside openPath(curve))

val closedPaths =
  style(closedPath(triangle) beside closedPath(curve
))

val paths = openPaths above closedPaths
```

Από τον παραπάνω κώδικα μπορούμε να καταλάβουμε ότι δημιουργούμε μονοπάτια χρησιμοποιώντας τις μεθόδους `openPath` και `closePath` για μία `Image` ακριβώς όπως κάνουμε και με άλλα σχήματα. Ένα μονοπάτι δημιουργείται από μία `List` με `PathElement`. Τα διαφορετικά είδη `PathElement` δημιουργούνται με κλήση μεθόδων του αντικειμένου `PathElement` όπως περιγράφεται στον πίνακα tbl. 4.

Table 4: Χρήση του `PathElement`.

Μέθοδος	Περιγραφή	Παράδειγμα
`moveTo(Point)`		`moveTo(cartesian(1,`

	Τοποθετήστε το μολύβι στο `Point` χωρίς να σχεδιάζεται η γραμμή.	`1))`
`lineTo(Point)`	Σχεδιάστε μία ευθεία γραμμή μέχρι το `Point`	`lineTo(cartesian(2, 2))`
`curveTo(Point, Point, Point)`	Σχεδιάστε μία καμπύλη. Τα δύο πρώτα σημεία καθορίζουν τα σημεία ελέγχου και το τελευταίο είναι αυτό στο οποίο τελειώνει η γραμμή.	`curveTo(cartesian(1,0), cartesian(0,1), cartesian(1,1))`

Η δημιουργία μίας `List` είναι αρκετά εύκολη: απλώς καλούμε την `List` με τα στοιχεία που θέλουμε να περιέχει. Παρακάτω μπορείτε να δείτε μερικά παραδείγματα.

```
// Λίστα με ακεραίους
List(1, 2, 3)
// res7: List[Int] = List(1, 2, 3)

// Λίστα με εικόνες
List(circle(10), circle(20), circle(30))
// res9: List[doodle.core.Image] = List(Circle(10.0)
, Circle(20.0), Circle(30.0))

// Λίστα με χρώματα
List(Color.paleGoldenrod, Color.paleGreen, Color.pal
```

```
eTurquoise)
// res11: List[doodle.core.Color] = List(RGBA(Unsign
edByte(110),UnsignedByte(104),UnsignedByte(42),Norma
lized(1.0)), RGBA(UnsignedByte(24),UnsignedByte(123)
,UnsignedByte(24),Normalized(1.0)), RGBA(UnsignedByt
e(47),UnsignedByte(110),UnsignedByte(110),Normalized
(1.0)))
```

Παρατηρήστε ότι ο τύπος της `List` συμπεριλαμβάνει και τον τύπο των περιεχομένων της μέσα σε αγκύλες. Άρα ο τύπος μίας λίστας ακεραίων γράφεται ως `List[Int]` και μία λίστα με `PathElement` γράφεται ως `List[PathElement]`.

Ασκήσεις

Πολύγωνα

Φτιάξτε μονοπάτια για να ορίσετε ένα τρίγωνο, ένα τετράγωνο και ένα πεντάγωνο. Το αποτέλεσμα πρέπει να μοιάζει με την εικόνα fig. 40. *Βοήθεια:* μπορεί να σας φανεί ευκολότερο αν για τον ορισμό των πολυγώνων χρησιμοποιήσετε πολικές συντεταγμένες .

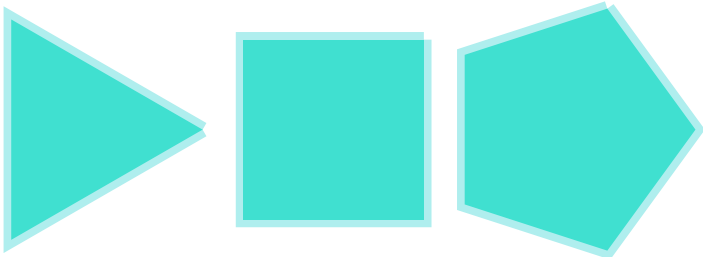


Figure 40: Ένα τρίγωνο, ένα τετράγωνο και ένα πεντάγωνο σχεδιασμένα με μονοπάτια.

See the solution

Καμπύλες

Επαναλάβετε την παραπάνω άσκηση αλλά αυτή τη φορά χρησιμοποιήστε καμπύλες αντί για ευθείες γραμμές ώστε να κατασκευάσετε ενδιαφέροντα σχήματα. Στην εικόνα fig. 41 μπορείτε να δείτε τα δικά μας πολύγωνα. *Βοήθεια:* θα είναι πιο εύκολο αν μετατρέψετε σε μέθοδο τον κώδικα για την δημιουργία καμπύλης .

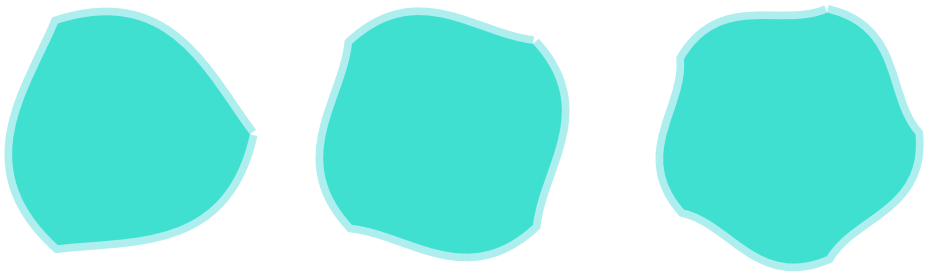


Figure 41: Τρίγωνο, τετράγωνο και πολύγωνο σχεδιασμένα με καμπύλες και ορισμένα από μονοπάτια.

See the solution

9.2 Δουλεύοντας με Λίστες

Σ' αυτό το σημείο ίσως σκέφτεστε ότι θα ήταν καλή ιδέα να δημιουργήσουμε μία μέθοδο που δημιουργεί πολύγωνα αντί να τα κατασκευάζουμε κάθε φορά από την αρχή. Είναι φανερό ότι υπάρχουν επαναλαμβανόμενα κομμάτια στην κατασκευή τους και άρα θα μπορούσαμε να τα γενικεύσουμε αν γνωρίζαμε πώς να φτιάχνουμε λίστες μη προκαθορισμένου μεγέθους. Ήρθε η ώρα να μάθουμε περισσότερα για την δημιουργία και την διαχείριση λιστών.

9.2.1 Η Αναδρομική Δομή των Λιστών

Ίσως θυμάστε ότι όταν μιλήσαμε πρώτη φορά για την δομημένη αναδρομή με φυσικούς αριθμούς, είπαμε ότι θα μπορούσαμε να μετατρέψουμε την αναδρομική τους δομή σε οποιαδήποτε άλλη αναδρομική δομή. Το εφαρμόσαμε στους ομόκεντρους κύκλους καθώς και σε διάφορες άλλες περιπτώσεις.

Οι λίστες έχουν μία αναδρομική δομή η οποία είναι παρόμοια με αυτή της δομής των φυσικών αριθμών. Μία λίστα μπορεί να είναι

- άδεια `Nil`, ή
- ένα ζεύγος που αποτελείται από ένα στοιχείο `a` και μία `List` και γράφεται ως `a :: tail`, όπου το `tail` είναι το υπόλοιπο της λίστας.

Για παράδειγμα, μπορούμε να γράψουμε την λίστα `List(1, 2, 3, 4)` και με την παρακάτω μορφή

```
1 :: 2 :: 3 :: 4 :: Nil
// res0: List[Int] = List(1, 2, 3, 4)
```

Παρατηρήστε την ομοιότητα με τους φυσικούς αριθμούς. Προηγουμένως είπαμε ότι μπορούμε να επεκτείνουμε την δομή ενός φυσικού αριθμού ώστε να γράψουμε για παράδειγμα, τον αριθμό `3` ως `1 + 1 + 1 + 0`. Αν αντικαταστήσουμε το `+` με `::` και το `0` με `Nil` τότε παίρνουμε `List 1 :: 1 :: 1 :: Nil`.

Τι σημαίνει όμως αυτό; Σημαίνει ότι μπορούμε πολύ εύκολα να μετατρέψουμε ένα φυσικό αριθμό σε `List` χρησιμοποιώντας το γνωστό εργαλείο της δομημένης αναδρομής1. Παρακάτω μπορείτε να δείτε ένα πολύ απλό παράδειγμα, στο οποίο με δεδομένο έναν αριθμό, δημιουργείται μία λίστα αντίστοιχου μήκους, η οποία περιέχει το `String` “Hi”.

```
def sayHi(length: Int): List[String] =
  length match {
    case 0 => Nil
    case n => "Hi" :: sayHi(n - 1)
  }
// sayHi: (length: Int)List[String]

sayHi(5)
// res1: List[String] = List(Hi, Hi, Hi, Hi, Hi)
```

Ο παραπάνω κώδικας μετατρέπει:

- το `0` σε `Nil`, για την βασική περίπτωση και
- το `n` (το οποίο θυμηθείτε ότι το θεωρούμε σαν `1 + m`) σε `"Hi" :: sayHi(n - 1)`, μετατρέποντας το `1` σε `"Hi"`, το `+` σε `::` και ως συνήθως κάνοντας αναδρομή στο `m` (το οποίο είναι `n - 1`).

Ακόμη, αυτή η αναδρομική δομή μας επιτρέπει να μετατρέψουμε λίστες σε άλλες αναδρομικές δομές, όπως φυσικούς αριθμούς, άλλες λίστες, σκακιέρες κλπ. Αυξάνουμε κάθε στοιχείο της λίστας—που σημαίνει ότι μετατρέπουμε την λίστα σε άλλη λίστα—χρησιμοποιώντας δομημένη αναδρομή.

```
def increment(list: List[Int]): List[Int] =
  list match {
    case Nil => Nil
    case hd :: tl => (hd + 1) :: increment(tl)
  }
// increment: (list: List[Int])List[Int]

increment(List(1, 2, 3))
// res2: List[Int] = List(2, 3, 4)
```

Προσθέτουμε τα στοιχεία μία λίστας ακεραίων—δηλαδή μετατρέπουμε την λίστα με φυσικό αριθμό—χρησιμοποιώντας δομημένη αναδρομή.

```
def sum(list: List[Int]): Int =
  list match {
    case Nil => 0
    case hd :: tl => hd + sum(tl)
  }
// sum: (list: List[Int])Int

sum(List(1, 2, 3))
// res3: Int = 6
```

Παρατηρήστε ότι όταν ξεχωρίζουμε τα στοιχεία της `List` για να εφαρμόσουμε τις περιπτώσεις της match, χρησιμοποιούμε το ίδιο συντακτικό `hd :: tl` που χρησιμοποιήσαμε και για να την κατασκευάσουμε. Αυτή η συμμετρία είναι σκόπιμη.

9.2.2 Μεταβλητές Τύπων

Τι γίνεται αν θέλουμε να βρούμε το μήκος μίας λίστας; Γνωρίζουμε ότι μπορούμε να χρησιμοποιήσουμε το συνηθισμένο μας εργαλείο, δηλαδή την δομημένη αναδρομή, ώστε να γράψουμε μία μέθοδο που θα κάνει ακριβώς αυτό. Παρακάτω μπορείτε να δείτε τον κώδικα για τον υπολογισμό του μήκους μίας λίστας `List[Int]`.

```
def length(list: List[Int]): Int =
```

```
  list match {
    case Nil => 0
    case hd :: tl => 1 + length(tl)
  }
// length: (list: List[Int])Int
```

Παρατηρήστε ότι δεν ασχολούμαστε με τα στοιχεία της λίστας—δεν μας ενδιαφέρει ο τύπος τους. Χρησιμοποιώντας τον ίδιο σκελετό μπορούμε να υπολογίσουμε το ίδιο εύκολα το μήκος μίας `List[Int]` όπως και μίας `List[HairyYak]` αλλά μέχρι τώρα δεν γνωρίζουμε πώς να ορίσουμε μία λίστα για την οποία δεν μας ενδιαφέρει ο τύπος των στοιχείων της.

Η Scala μας επιτρέπει να γράφουμε μεθόδους οι οποίες μπορούν να δουλέψουν με οποιονδήποτε τύπο, χρησιμοποιώντας κάτι που ονομάζεται *μεταβλητή τύπου*. Η μεταβλητή τύπου γράφεται μέσα σε αγκύλες, δηλαδή `[A]`, και βρίσκεται μετά το όνομα της μεθόδου και πριν την λίστα παραμέτρων. Μία μεταβλητή τύπου, μπορεί να αντικαταστήσει οποιονδήποτε τύπο και μπορούμε να την χρησιμοποιήσουμε στην λίστα παραμέτρων ή στον τύπο του αποτελέσματος, ώστε να αντικαταστήσει κάποιον τύπο που δεν γνωρίζουμε. Για παράδειγμα, παρακάτω μπορείτε να δείτε πώς μπορούμε να γράψουμε την `length` ώστε να δουλεύει με λίστες κάθε τύπου.

```
def length[A](list: List[A]): Int =
  list match {
    case Nil => 0
    case hd :: tl => 1 + length(tl)
  }
// length: [A](list: List[A])Int
```

Δομημένη Αναδρομή με Λίστα

Μία `List` στοιχείων τύπου `A` είναι:

- άδεια `Nil`, ή
- ένα στοιχείο `a` τύπου `A` και μία `tail` τύπου `List[A]`: `a :: tail`

Το σχήμα της δομημένης αναδρομής για την μετατροπή μίας `list` με τύπο `List[A]` σε έναν τύπο `B` είναι το παρακάτω

```
def doSomething[A,B](list: List[A]): B =
  list match {
    case Nil => ??? // Βασική περίπτωση για τον τύπο B
    case hd :: tl => f(hd, doSomething(tl))
  }
```

όπου το `f` είναι μία συγκεκριμένη μέθοδος για προβλήματα η οποία συνδυάζει το `hd` και το αποτέλεσμα της αναδρομικής κλήσης ώστε να παράξει κάτι με τύπο `B`.

Ασκήσεις

Κατασκευή Λιστών

Κάνοντας τις παρακάτω ασκήσεις θα γίνουμε πιο έμπειροι στην κατασκευή λιστών με χρήση της δομημένης αναδρομής με φυσικούς αριθμούς

Γράψτε μία μέθοδο με όνομα `ones` η οποία δέχεται έναν ακέραιο `n` και επιστρέφει μία λίστα `List[Int]` με μήκος `n` όπου όλα τα στοιχεία της είναι `1`. Για παράδειγμα

```
ones(3)
// res4: List[Int] = List(1, 1, 1)
```

See the solution

Γράψτε μία μέθοδο με όνομα `descending` η οποία δέχεται έναν φυσικό αριθμό `n` και επιστρέφει μία `List[Int]` η οποία περιέχει τους φυσικούς αριθμούς από το `n` ως το `1` ή επιστρέφει μία άδεια λίστα αν το `n` είναι μηδέν. Για παράδειγμα

```
descending(0)
// res6: List[Int] = List()
```

```
descending(3)
// res7: List[Int] = List(3, 2, 1)
```

See the solution

Γράψτε μία μέθοδο με όνομα `ascending` η οποία δέχεται έναν φυσικό αριθμό `n` και επιστρέφει μία `List[Int]` η οποία περιέχει τους φυσικούς αριθμούς από το `1` ως το `n` ή επιστρέφει μία άδεια λίστα αν το `n` είναι μηδέν.

```
ascending(0)
// res10: List[Int] = List()

ascending(3)
// res11: List[Int] = List(1, 2, 3)
```

See the solution

Φτιάξτε μία μέθοδο με όνομα `fill` η οποία δέχεται έναν φυσικό αριθμό `n` και ένα στοιχείο `a` τύπου `A` και κατασκευάζει μία λίστα μήκους `n` όπου όλα της τα στοιχεία είναι `a`.

```
fill(3, "Hi")
// res14: List[String] = List(Hi, Hi, Hi)

fill(3, Color.blue)
// res15: List[doodle.core.Color] = List(RGBA(UnsignedByte(-128),UnsignedByte(-128),UnsignedByte(127),Normalized(1.0)), RGBA(UnsignedByte(-128),UnsignedByte(-128),UnsignedByte(127),Normalized(1.0)), RGBA(UnsignedByte(-128),UnsignedByte(-128),UnsignedByte(127),Normalized(1.0)))
```

See the solution

Μετατροπή Λιστών

Σ' αυτή την άσκηση θα εξασκηθούμε στην άλλη πλευρά του χειρισμού λιστών—στην μετατροπή τους σε στοιχεία άλλων τύπων (και μερικές

φορές σε διαφορετικές λίστες).

Γράψτε μία μέθοδο με όνομα `double` η οποία δέχεται μία `List[Int]` και επιστρέφει μία λίστα όπου το κάθε της στοιχείο είναι διπλασιασμένο.

```
double(List(1, 2, 3))
// res18: List[Int] = List(2, 4, 6)

double(List(4, 9, 16))
// res19: List[Int] = List(8, 18, 32)
```

See the solution

Γράψτε μία μέθοδο με όνομα `product` η οποία δέχεται μία `List[Int]` και υπολογίζει το γινόμενο όλων των στοιχείων της.

```
product(Nil)
// res22: Int = 1

product(List(1,2,3))
// res23: Int = 6
```

See the solution

Γράψτε μία μέθοδο με όνομα `contains` η οποία δέχεται μία `List[A]` και ένα στοιχείο τύπου `A` και επιστρέφει true αν η λίστα περιέχει το στοιχείο ή σε αντίθετη περίπτωση false.

```
contains(List(1,2,3), 3)
// res26: Boolean = true

contains(List("one", "two", "three"), "four")
// res27: Boolean = false
```

See the solution

Γράψτε μία μέθοδο με όνομα `first` η οποία δέχεται μία `List[A]` και ένα στοιχείο τύπου `A` και επιστρέφει το πρώτο στοιχείο της λίστας αν αυτή δεν είναι άδεια ή αλλιώς το στοιχείο τύπου `A` που περάσαμε ως παράμετρο.

```
first(Nil, 4)
// res30: Int = 4

first(List(1,2,3), 4)
// res31: Int = 1
```

See the solution

Άσκηση Πρόκληση: Αντιστροφή

Γράψτε μία μέθοδο με όνομα `reverse` η οποία δέχεται μία λίστα `List[A]` και την αντιστρέφει.

```
reverse(List(1, 2, 3))
// res34: List[Int] = List(3, 2, 1)

reverse(List("a", "b", "c"))
// res35: List[String] = List(c, b, a)
```

See the solution

Πολύγωνα!

Τέλος, ας επιστρέψουμε στο παράδειγμα με τον σχεδιασμό πολυγώνων. Γράψτε μία μέθοδο με όνομα `polygon` η οποία δέχεται τον αριθμό των πλευρών του πολυγώνου και την αρχική γωνία περιστροφής και παράγει μία `Image` με το αντίστοιχο πολύγωνο. *Βοήθεια:* χρησιμοποιήστε έναν εσωτερικό συσσωρευτή.

Χρησιμοποιήστε αυτόν τον κώδικα για να δημιουργήσετε μία ωραία εικόνα συνδυάζοντας πολύγωνα. Μπορείτε να δείτε το δικό μας ευφάνταστο παράδειγμα στην εικόνα fig. 42. Είμαστε σίγουροι ότι εσείς μπορείτε και καλύτερα από αυτό.

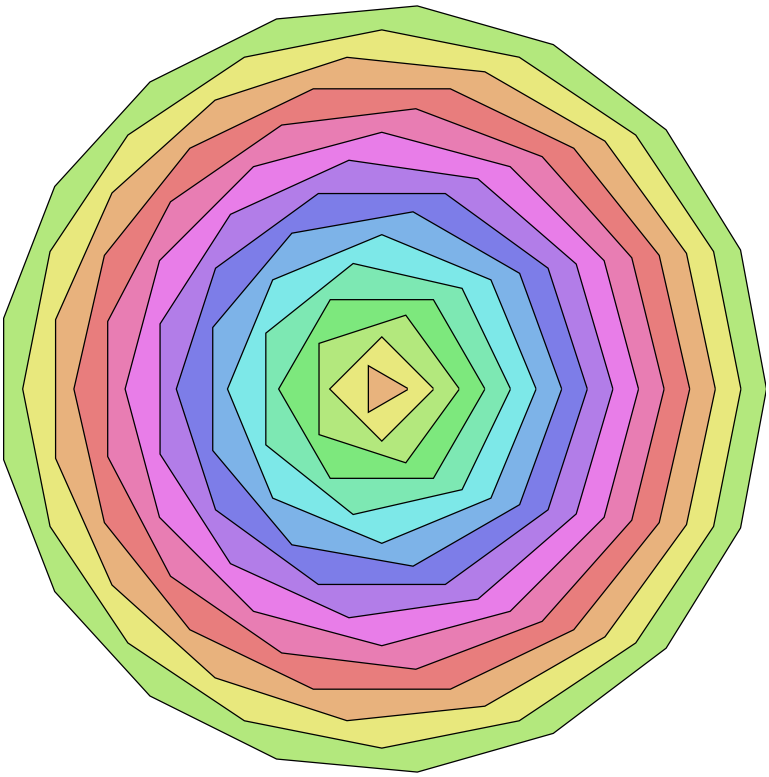


Figure 42: Ομόκεντρα πολύγωνα σε παστέλ αποχρώσεις.

See the solution

9.3 Μετατρέποντας Ακολουθίες

Έχουμε δει ότι χρησιμοποιώντας δομημένη αναδρομή μπορούμε να δημιουργήσουμε λίστες και να τις μετασχηματίσουμε σε κάτι άλλο. Αυτός ο τρόπος είναι αρκετά απλός στην χρήση και στην κατανόησή του όμως απαιτεί το γράψιμο του ίδιου σκελετού ξανά και ξανά. Σε αυτή την ενότητα θα μάθουμε ότι σε μερικές περιπτώσεις μπορούμε να αντικαταστήσουμε την δομημένη αναδρομή χρησιμοποιώντας μία μέθοδο της `List` η οποία ονομάζεται `map`. Θα δούμε επίσης ότι και άλλοι τύποι δεδομένων μπορούν να μας παρέχουν αυτή τη μέθοδο και ότι μπορούμε να την χρησιμοποιήσουμε ως έναν εύκολο τρόπο διαχείρισης μιας ακολουθίας.

9.3.1 Μετατρέποντας Στοιχεία σε μια Λίστα

Στην προηγούμενη ενότητα είδαμε σε διάφορα παραδείγματα λίστες να μετατρέπονται σε άλλες λίστες. Για παράδειγμα, προσθέσαμε μία τιμή σε όλα τα στοιχεία μίας λίστας χρησιμοποιώντας τον παρακάτω κώδικα.

```
def increment(list: List[Int]): List[Int] =
  list match {
    case Nil => Nil
    case hd :: tl => (hd + 1) :: tl
  }
// increment: (list: List[Int])List[Int]

increment(List(1, 2, 3))
// res0: List[Int] = List(2, 2, 3)
```

Σ' αυτό το παράδειγμα, δεν αλλάζει η *δομή* της λίστας. Αν ξεκινήσουμε με τρία στοιχεία, τελειώνουμε με τρία στοιχεία. Μπορούμε να αντικαταστήσουμε τον παραπάνω τρόπο με μία μέθοδο που ονομάζεται `map`. Αν έχουμε μία λίστα με δεδομένα τύπου `A`, περνάμε στην `map` μία συνάρτηση τύπου `A => B` και παίρνουμε πίσω μία λίστα της οποίας τα στοιχεία έχουν τύπο `B`. Για παράδειγμα, μπορούμε να υλοποιήσουμε την `increment` χρησιμοποιώντας την `map` με την συνάρτηση `x => x + 1`.

```
def increment(list: List[Int]): List[Int] =
  list.map(x => x + 1)
// increment: (list: List[Int])List[Int]

increment(List(1, 2, 3))
// res1: List[Int] = List(2, 3, 4)
```

Κάθε στοιχείο μετασχηματίζεται σύμφωνα με την συνάρτηση που περνάμε στην `map`. Σ' αυτή την περίπτωση έχουμε την `x => x + 1`. Με την `map` μπορούμε να μετασχηματίσουμε τα στοιχεία μίας λίστας αλλά δεν μπορούμε να αλλάξουμε το πλήθος τους.

Η παρακάτω αναπαράσταση με γραφικά θα σας βοηθήσει στην κατανόηση της `map`. Αν έχουμε έναν τύπο `Circle` τότε μπορούμε να φανταστούμε την `List[Circle]` σαν ένα κουτί που περιέχει έναν κύκλο, όπως φαίνεται στην εικόνα fig. 43.

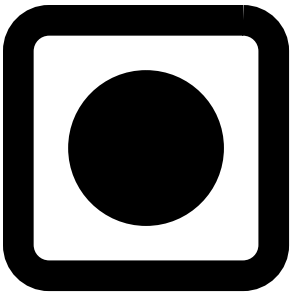


Figure 43: Μία `List[Circle]` που αναπαρίσταται από έναν κύκλο μέσα σε ένα κουτί

Τώρα μπορούμε να σχεδιάσουμε μία εξίσωση για την `map` όπως φαίνεται στην εικόνα fig. 44. Αν προτιμάτε τα σύμβολα από τις εικόνες, η εικόνα αναπαριστά το `List[Circle] map (Circle => Triangle) = List[Triangle]`. Ένα πλεονέκτημα της γραφικής αυτής αναπαράστασης είναι ότι παρουσιάζει πολύ καλά το ότι η `map` δεν μπορεί να δημιουργήσει ένα νέο "κουτί" που θα αναπαριστά την δομή της λίστας—ή πιο σωστά, δεν μπορεί να αλλάξει τον αριθμό των στοιχείων της και την σειρά τους.

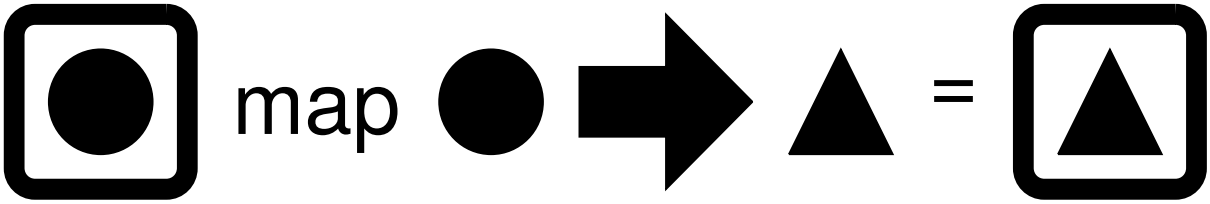


Figure 44: Γραφική αναπαράσταση της `map`

Η γραφική αναπαράσταση της `map` δείχνει τι ακριβώς συμβαίνει με τους τύπους που χρησιμοποιεί. Αν το προτιμάτε, μπορούμε να παρουσιάσουμε το ίδιο και με σύμβολα:

```
List[A] map (A => B) = List[B]
```

Στην αριστερή πλευρά της εξίσωσης υπάρχει ο τύπος της λίστας που περνάμε από την map καθώς και η συνάρτηση που θα χρησιμοποιήσουμε για την μετατροπή. Στην δεξιά πλευρά, βρίσκεται ο τύπος του αποτελέσματος.

9.3.2 Μετατρέποντας Ακολουθίες Αριθμών

Έχουμε ήδη γράψει πολλές μεθόδους που μετατρέπουν φυσικούς αριθμούς σε λίστες. Επίσης συζητήσαμε περιληπτικά για το πώς

μπορούμε να αναπαραστήσουμε έναν φυσικό αριθμό ως λίστα. Το `3` είναι ισοδύναμο με το `1 + 1 + 1 + 0`, το οποίο με την σειρά του θα μπορούσε να αναπαρασταθεί από μία λίστα `List(1, 1, 1)`. Άρα; Αυτό σημαίνει ότι θα μπορούσαμε να γράψουμε πολλές από τις μεθόδους οι οποίες δέχονται φυσικούς αριθμούς, ως μεθόδους που δουλεύουν με λίστες.

Για παράδειγμα, αντί για

```
def fill[A](n: Int, a: A): List[A] =
  n match {
    case 0 => Nil
    case n => a :: fill(n-1, a)
  }
// fill: [A](n: Int, a: A)List[A]

fill(3, "Hi")
// res2: List[String] = List(Hi, Hi, Hi)
```

θα μπορούσαμε να γράψουμε

```
def fill[A](n: List[Int], a: A): List[A] =
  n.map(x => a)
// fill: [A](n: List[Int], a: A)List[A]

fill(List(1, 1, 1), "Hi")
// res3: List[String] = List(Hi, Hi, Hi)
```

Είναι πιο εύκολο να γράψουμε αυτήν την εκδοχή της `fill` αλλά για τον χρήστη είναι πιο βολικό να την καλέσει με το `3` αντί για το `List(1, 1, 1)`.

Αν θέλουμε να δουλέψουμε με ακολουθίες αριθμών, καλύτερα να χρησιμοποιήσουμε τα `Ranges` (διαστήματα). Μπορούμε να τα δημιουργήσουμε χρησιμοποιώντας την μέθοδο `until` με `Int` ή `Double`:

```
0 until 10
// res4: scala.collection.immutable.Range = Range 0
until 10
```

```
0.0 until 5.0
// res5: Range.Partial[Double,scala.collection.immut
able.NumericRange[Double]] = Range requires step
```

τα `Ranges` έχουν μία μέθοδο `by` η οποία μας επιτρέπει να αλλάξουμε το βήμα μεταξύ των στοιχείων της ακολουθίας:

```
0 until 10 by 2
// res6: scala.collection.immutable.Range = Range 0
until 10 by 2

0.0 until 1.0 by 0.3
// res7: scala.collection.immutable.NumericRange[Dou
ble] = NumericRange 0.0 until 1.0 by 0.3
```

τα `Ranges` έχουν και αυτά μία μέθοδο `map`, ακριβώς όπως και η `List`

```
(0 until 3) map (x => x + 1)
// res8: scala.collection.immutable.IndexedSeq[Int]
= Vector(1, 2, 3)
```

Θα παρατηρήσατε ότι ο τύπος του αποτελέσματος είναι ο `IndexedSeq` ο οποίος τελικά υλοποιείται ως `Vector`—δύο τύποι που δεν τους έχουμε δει ακόμα. Μπορούμε να φτιάξουμε ένα `IndexedSeq` όπως φτιάχνουμε και μία `List` αλλά για λόγους απλότητας μπορούμε να μετατρέψουμε ένα `Range` ή ένα `IndexedSeq` σε ένα `List` χρησιμοποιώντας την μέθοδο `toList`.

```
(0 until 7).toList
// res9: List[Int] = List(0, 1, 2, 3, 4, 5, 6)

(0 until 3).map(x => x + 1).toList
// res10: List[Int] = List(1, 2, 3)
```

Με τα `Ranges` στην εργαλειοθήκη μας, μπορούμε να γράψουμε την `fill` ως εξής

```
def fill[A](n: Int, a: A): List[A] =
  (0 until n).toList.map(x => a)
```

```
// fill: [A](n: Int, a: A)List[A]

fill(3, "Hi")
// res11: List[String] = List(Hi, Hi, Hi)
```

Ασκήσεις

Ranges, Lists και map

Γράψτε ξανά τις παρακάτω μεθόδους χρησιμοποιώντας τα νέα εργαλεία που μάθαμε παραπάνω.

Γράψτε μία μέθοδο με όνομα `ones` η οποία δέχεται έναν `Int` με όνομα `n` και επιστρέφει μία `List[Int]` μήκους `n` όπου όλα της τα στοιχεία είναι `1`. Για παράδειγμα

```
ones(3)
// res12: List[Int] = List(1, 1, 1)
```

See the solution

Γράψτε μία μέθοδο με όνομα `descending` η οποία δέχεται έναν φυσικό αριθμό `n` και επιστρέφει μία `List[Int]` η οποία περιέχει τους φυσικούς αριθμούς από το `n` ως το `1`. Αν η λίστα είναι άδεια, δηλαδή το `n` είναι μηδέν, επιστρέφει την άδεια λίστα. Για παράδειγμα

```
descending(0)
// res14: List[Int] = List()

descending(3)
// res15: List[Int] = List(3, 2, 1)
```

See the solution

Γράψτε μία μέθοδο με όνομα `ascending` η οποία δέχεται έναν φυσικό αριθμό `n` και επιστρέφει μία `List[Int]` η οποία περιέχει τους φυσικούς αριθμούς από το `1` μέχρι το `n`. Αν το `n` είναι μηδέν, επιστρέφει την άδεια λίστα.

```
ascending(0)
```

```
// res18: List[Int] = List()

ascending(3)
// res19: List[Int] = List(1, 2, 3)
```

See the solution

Γράψτε μία μέθοδο με όνομα `double` η οποία δέχεται μία `List[Int]` και επιστρέφει μία λίστα όπου κάθε της στοιχείο είναι διπλασιασμένο.

```
double(List(1, 2, 3))
// res22: List[Int] = List(2, 4, 6)

double(List(4, 9, 16))
// res23: List[Int] = List(8, 18, 32)
```

See the solution

Πολύγωνα και Πάλι!

Χρησιμοποιώντας τα νέα μας εργαλεία, γράψτε και πάλι την μέθοδο `polygon` που είχαμε δει στην προηγούμενη ενότητα.

See the solution

Άσκηση Πρόκληση: Πέρα από την map

Μπορούμε να χρησιμοποιήσουμε την `map` για να αντικαταστήσουμε όλες τις χρήσεις της δομημένης αναδρομής; Αν όχι, μπορείτε να αναφέρετε τα προβλήματα που δεν μπορούμε να λύσουμε με την `map` αλλά μπορούμε να λύσουμε με γενική δομημένη αναδρομή με λίστες;

See the solution

9.3.3 Εργαλεία Range

Είδαμε την μέθοδο `until` για την κατασκευή της `Range` και την `by` για την αλλαγή του βήματός της. Υπάρχει όμως και άλλη μία μέθοδος η οποία είναι χρήσιμη: η `to`.

```
1 until 5
// res26: scala.collection.immutable.Range = Range 1
```

```
 until 5

1 to 5
// res27: scala.collection.immutable.Range.Inclusive
 = Range 1 to 5
```

Ασκήσεις

Χρησιμοποιώντας το Range

Γράψτε μία μέθοδο με όνομα `ascending` η οποία δέχεται έναν φυσικό αριθμό `n` και επιστρέφει μία `List[Int]` η οποία περιέχει τους φυσικούς αριθμούς από το `1` ως το `n`. Αν το `n` είναι μηδέν, επιστρέφει την άδεια λίστα. *Βοήθεια:* χρησιμοποιήστε την `to`

```
ascending(0)
// res28: List[Int] = List()

ascending(3)
// res29: List[Int] = List(1, 2, 3)
```

See the solution

9.4 Θεέ μου, Πόσα Αστέρια!

Ας χρησιμοποιήσουμε τα νέα εργαλεία που μάθαμε για να ζωγραφίσουμε μερικά αστέρια. Για τους σκοπούς αυτής της άσκησης, ας υποθέσουμε ότι ένα αστέρι είναι ένα πολύγωνο με `p` κορυφές. Όμως, αντί να ενώσουμε την κάθε κορυφή με τις γειτονικές της, θα τις ενώσουμε με την νιοστή κορυφή της περιφέρειας του κύκλου.

Για παράδειγμα, η εικόνα fig. 45 δείχνει αστέρια με `p=11` και `n=1 to 5`. Το `n=1` παράγει ένα συνηθισμένο πολύγωνο. Όταν το `n` παίρνει τιμές από `2` και πάνω, παράγονται αστέρια με όλο και πιο μυτερές κορυφές:

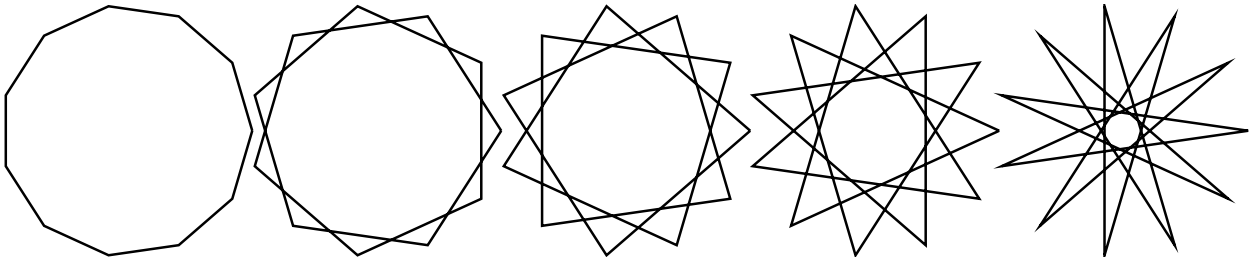


Figure 45: Αστέρια με `p=11` και `n=1 to 5`

Γράψτε κώδικα που δημιουργεί την παραπάνω εικόνα. Ξεκινήστε γράφοντας μία μέθοδο η οποία παράγει ένα `star` (αστέρι) παίρνοντας ώς παράμετρο τα `p` και `n`:

```
def star(p: Int, n: Int, radius: Double): Image =
  ???
```

Βοήθεια: χρησιμοποιήστε την τεχνική που είδαμε προηγουμένως για την μέθοδο `polygon`.

See the solution

Χρησιμοποιώντας δομημένη αναδρομή και την `beside` γράψτε μία μέθοδο με όνομα `allBeside` με την μορφή

```
def allBeside(images: List[Image]): Image =
  ???
// allBeside: (images: List[doodle.core.Image])doodl
e.core.Image
```

See the solution

Αφού τελειώσετε με την σειρά από αστέρια, προσπαθήστε να φτιάξετε μία μεγαλύτερη εικόνα για διαφορετικές τιμές των `p` και `n`. Μπορείτε να δείτε ένα παράδειγμα στην εικόνα fig. 46. *Βοήθεια:* Θα πρέπει να φτιάξετε μία μέθοδο `allAbove` παρόμοια με την `allBeside`.

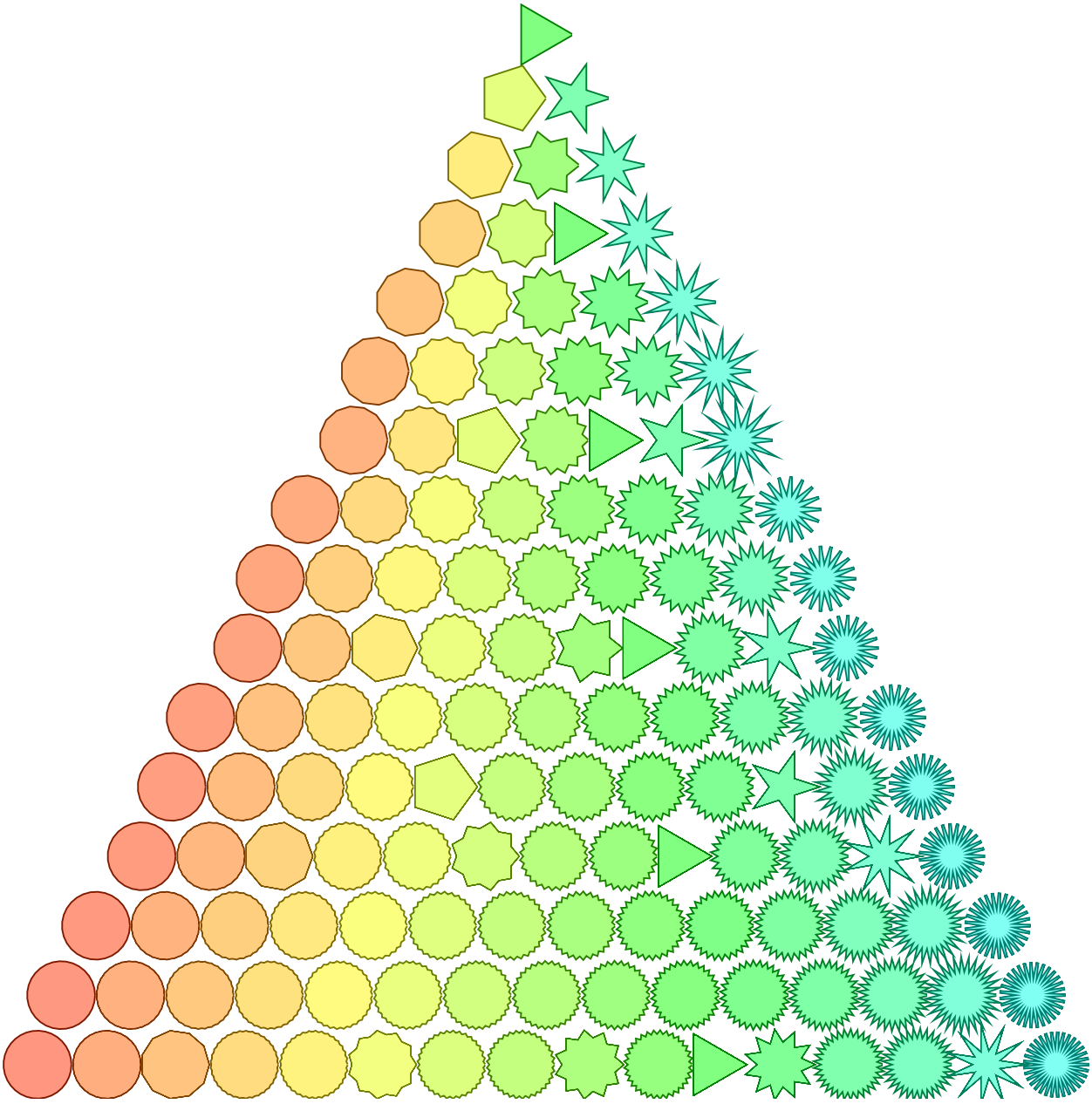


Figure 46: Αστέρια με `p=3 to 33 by 2` και `n=1 to p/2`

See the solution

---

1. Η σύνδεσή τους είναι πιο βαθιά από όσο φαίνεται αρχικά. Μπορούμε να χρησιμοποιήσουμε την μέθοδο της αφαίρεσης σε ότι “προστίθεται” για να εφαρμόσουμε μία έννοια η οποία ονομάζεται monoid, στην οποία μία λίστα αντιπροσωπεύει έναν συγκεκριμένο τύπο monoid που ονομάζεταιι free monoid. Δεν θα χρησιμοποιήσουμε αυτή τη μέθοδο στην Creative Scala αλλά

μπορείτε να την ψάξετε μόνοι σας! ↩

10 Άλγεβρα Turtle και Αλγεβρικοί Τύποι Δεδομένων

Σ' αυτό το κεφάλαιο θα εξερευνήσουμε νέους τρόπους δημιουργίας μονοπατιών—με γραφικά turtle—και θα μάθουμε μερικούς καινούριους τρόπους χειρισμού λιστών και συναρτήσεων.

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

10.1 Γραφικά Turtle

Μέχρι τώρα τα μονοπάτια που είδαμε χρησιμοποιούσαν ένα απόλυτο σύστημα συντεταγμένων. Για παράδειγμα, αν θέλαμε να σχεδιάσουμε ένα τετράγωνο, θα γράφαμε κάτι τέτοιο

```
import doodle.core.PathElement._

val path =
  Image.openPath(
    List(moveTo(10,10), lineTo(-10,10), lineTo(-10,-
10), lineTo(10, -10), lineTo(10, 10))
  )
```

Πολλές φορές όμως, είναι πιο εύκολο να οριστούν μονοπάτια με σχετικές

συντεταγμένες. Ειδικά όταν πρέπει να καθοριστεί πόσο έχουμε κινηθεί προς τα μπροστά ή πόσο έχουμε στρίψει, σε σχέση με την θέση στην οποία βρισκόμαστε. Έτσι ακριβώς λειτουργεί ένα σύστημα γραφικών turtle. Παρακάτω μπορείτε να δείτε ένα παράδειγμα.

```
import doodle.turtle._
import doodle.turtle.Instruction._

// Δημιούργησε μία λίστα εντολών για το turtle
val instructions: List[Instruction] =
  List(forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10))

// Ζήτα από το turtle να ζωγραφίσει αυτές τις εντολέ
ς δημιουργώντας μία εικόνα
val path: Image = Turtle.draw(instructions)
```

Πού είναι το turtle μέσα σ' όλα αυτά; Αυτό το μοντέλο δημιουργήθηκε την δεκαετία του '60 από τον Seymour Papert για την γλώσσα προγραμματισμού Logo. Η Logo μπορούσε να ελέγξει ένα ρομπότ που ζωγράφιζε με μολύβι πάνω σε χαρτί. Αυτό το ρομπότ ονομάζονταν turtle, λόγω του στρογγυλού του σχήματος και έτσι ο τρόπος προγραμματισμού του ρομπότ έγινε γνωστός ως γραφικά turtle.

Με την χρήση των γραφικών turtle και το L-system, μπορούμε να δημιουργήσουμε εικόνες οι οποίες μιμούνται την φύση, όπως για παράδειγμα το φυτό στην εικόνα fig. 47.

Figure 47: Ένα φυτό που φτιάχτηκε χρησιμοποιώντας γραφικά turtle.

10.2 Ελέγχοντας το Turtle

Θα δούμε το API για τα γραφικά turtle και θα το χρησιμοποιήσουμε για να φτιάξουμε μερικές εικόνες.

10.2.1 Εντολές

Ελέγχουμε το turtle δίνοντάς του εντολές. Αυτές οι εντολές, ορίζονται ως μέθοδοι του αντικειμένου `doodle.turtle.Instruction` (παρόμοια με τις μεθόδους του αντικειμένου `doodle.core.Image`).

Αφού εισάγουμε τις μεθόδους αυτές με την χρήση του import θα είμαστε έτοιμοι να δημιουργήσουμε εντολές.

```
import doodle.turtle._
import doodle.turtle.Instruction._
```

```
forward(10)
// res0: doodle.turtle.Instruction = Forward(10.0)

turn(5.degrees)
// res1: doodle.turtle.Instruction = Turn(Angle(0.08
726646259971647))
```

Ο παραπάνω κώδικας δεν μπορεί να κάνει τίποτα χρήσιμο αν δεν ομαδοποιήσουμε αυτές τις εντολές ώστε να δημιουργήσουμε μια εικόνα. Για να το κάνουμε αυτό, φτιάχνουμε μία λίστα εντολών και ζητάμε από το turtle (για την ακρίβεια από το `doodle.turtle.Turtle`) να τις σχεδιάσει σε μια `Image`.

```
val instructions =
  List(forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10), turn(90.degrees),
       forward(10))

val path = Turtle.draw(instructions)
```

Δημιουργείται έτσι ένα μονοπάτι—μία `Image`—την οποία μπορούμε μετά να ζωγραφίσουμε ως συνήθως. Το αποτέλεσμα του παραπάνω κώδικα είναι η εικόνα fig. 48. Δεν είναι κάτι ιδιαίτερο αλλά μπορούμε να αλλάξουμε το χρώμα, το πάχος της γραμμής καθώς και πολλά άλλα στοιχεία της ώστε να δημιουργήσουμε κάτι πιο ενδιαφέρον.

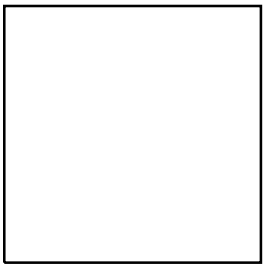


Figure 48: Ένα τετράγωνο που δημιουργήθηκε με την βοήθεια του συστήματος γραφικών turtle.

Στον πίνακα tbl. 5 μπορείτε να δείτε την πλήρη λίστα εντολών για το turtle

Table 5: οι εντολές που καταλαβαίνει το turtle.

Εντολή	Περιγραφή	Παράδειγμα
`forward(distance)`	Προχώρα όσο είναι η `distance` (απόσταση), τύπου `Double`.	`forward(100.0)`
`turn(angle)`	Στρίψε όσο είναι η `angle` (γωνία) από την τωρινή σου κατεύθυνση.	`turn(10.degrees)`
`branch(instruction, ...)`	Αποθήκευσε την θέση και την κατεύθυνσή σου, ζωγράφισε σύμφωνα με τις `instructions` (εντολές) και μετά επέστρεψε στην αρχική σου θέση ώστε να ζωγραφίσεις τις υπόλοιπες εντολές.	`branch(turn(10.degrees), forward(10))`
`noop`	Μην κάνεις τίποτα!	`noop`

Ασκήσεις

Πολύγωνα

Στο προηγούμενο κεφάλαιο γράψαμε μία μέθοδο για την δημιουργία ενός πολυγώνου. Φτιάξτε και πάλι αυτή την μέθοδο. Αυτή τη φορά χρησιμοποιήστε γραφικά turtle. Η δήλωση της μεθόδου θα πρέπει να μοιάζει με την παρακάτω

```
def polygon(sides: Int, sideLength: Double): Image =
  ???
```

Για να βρείτε την σωστή γωνία περιστροφής θα πρέπει να χρησιμοποιήσετε λίγη γεωμετρία. Αυτή είναι η μισή διασκέδαση, οπότε δεν θα σας το χαλάσουμε δίνοντάς σας την απάντηση.

See the solution

10.2.1.1 Το Τετράγωνο Σπειροειδές

Στην εικόνα fig. 49 μπορείτε να δείτε ένα τετράγωνο σπειροειδές σχήμα. Γράψτε μία μέθοδο που δημιουργεί τετράγωνα σπειροειδή, χρησιμοποιώντας γραφικά turtle.

Αυτή η άσκηση απαιτεί περισσότερη δουλειά από όσο σας ζητάμε συνήθως. Θα πρέπει να βρείτε πώς κατασκευάζεται το τετράγωνο σπειροειδές (βοήθεια: ξεκινάει από το κέντρο) και μετά να κατασκευάσετε την μέθοδο που το σχεδιάζει.

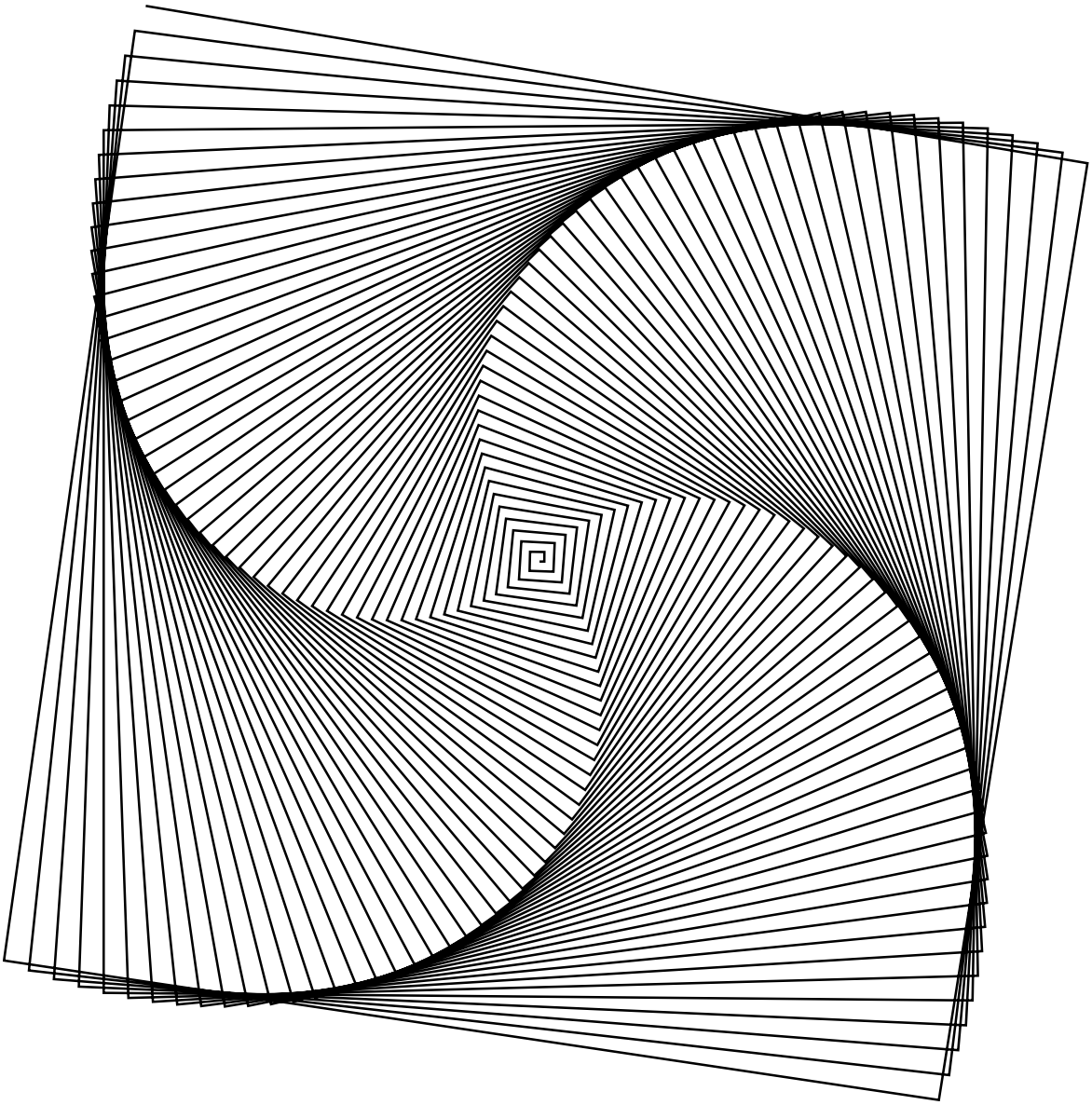


Figure 49: Το τετράγωνο σπειροειδές!

See the solution

Γραφικά Turtle vs Πολικές Συντεταγμένες

Μπορούμε να δημιουργήσουμε πολύγωνα με πολικές συντεταγμένες χρησιμοποιώντας ένα `Range` και ένα `map` όπως φαίνεται παρακάτω.

```
import doodle.core.Point._

def polygon(sides: Int, size: Int): Image = {
  val rotation = Angle.one / sides
```

```
  val elts =
    (1 to sides).toList.map { i =>
      PathElement.lineTo(polar(size, rotation * i))
    }
  closedPath(PathElement.moveTo(polar(size, Angle.ze
ro)) :: elts)
}
```

Όμως δεν είναι το ίδιο εύκολο να γράψουμε την ίδια μέθοδο για παραγωγή εντολών turtle χρησιμοποιώντας ένα `Range` και ένα `map`. Γιατί; Τι μας λείπει;

See the solution

10.3 Δομές Διακλαδώσεων

> Εκτός από τα imports που δίνονται στην αρχή κάθε κεφαλαίου, σ' αυτή την ενότητα θα προσθέσουμε και τα παρακάτω:
>
> ```
> import doodle.turtle._
> import doodle.turtle.Instruction._
> ```

Χρησιμοποιώντας την εντολή `branch` του turtle μπορούμε να δημιουργήσουμε σχήματα που θα ήταν δύσκολο να τα φτιάξουμε χωρίς αυτό. Η εντολή `branch` δέχεται μία λίστα `List[Instruction]`. Αποθηκεύει την τρέχουσα κατάσταση του turtle (την θέση του και την κατεύθυνσή του), σχεδιάζει τις γραμμές σύμφωνα με τις εντολές που του έχουν δοθεί και επαναφέρει το turtle στην αποθηκευμένη κατάσταση.

Δείτε τον παρακάτω κώδικα που δημιουργεί την εικόνα fig. 50. Έχει σχεδιαστεί εύκολα με turtle και την εντολή branch.

```
val y = Turtle.draw(List(
          forward(100),
          branch(turn(45.degrees), forward(100)),
          branch(turn(-45.degrees), forward(100))))
```

```
        )
// y: doodle.core.Image = OpenPath(List(MoveTo(Carte
sian(0.0,0.0)), MoveTo(Cartesian(0.0,0.0)), LineTo(C
artesian(100.0,0.0)), LineTo(Cartesian(170.710678118
65476,70.71067811865474)), MoveTo(Cartesian(100.0,0.
0)), LineTo(Cartesian(170.71067811865476,-70.7106781
1865474)), MoveTo(Cartesian(100.0,0.0)))
```

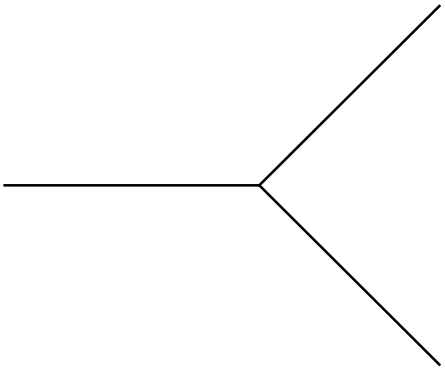


Figure 50: Μία εικόνα της οποίας ο σχεδιασμός με το turtle και την εντολή branch είναι εύκολος.

Χρησιμοποιώντας διακλαδώσεις μπορούμε να μοντελοποιήσουμε κάποιες μορφές βιολογικής ανάπτυξης, ώστε να δημιουργήσουμε για παράδειγμα εικόνες φυτών, όπως την fig. 47. Ένα τέτοιο μοντέλο είναι γνωστό ως *L-system*. Ένα L-system αποτελείται από δύο μέρη:

- έναν αρχικό *seed* για να ξεκινήσει η ανάπτυξη και
- τους *κανόνες αναπαραγωγής*, οι οποίοι καθορίζουν πώς γίνεται η ανάπτυξη.

Ένα παράδειγμα αυτής της διαδικασίας φαίνεται στην εικόνα fig. 51. Το σχήμα στην αριστερή πλευρά είναι το seed. Οι κανόνες αναπαραγωγής είναι οι ακόλουθοι:

- κάθε ευθεία γραμμή διπλασιάζεται σε μέγεθος και
- ο βλαστός (το διαμάντι στην άκρη της γραμμής) μεγαλώνει σε δύο άλλα κλαδιά.

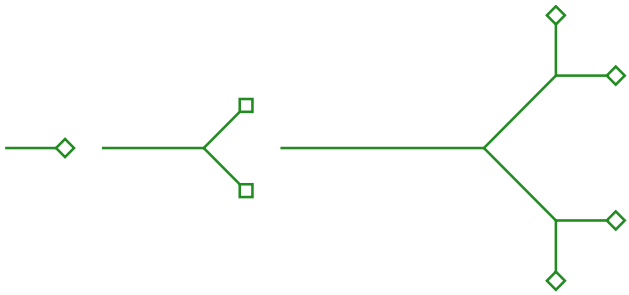


Figure 51: Αναπαράσταση της ανάπτυξης ενός φυτού χρησιμοποιώντας κανόνες αναπαραγωγής.

Επομένως, μπορούμε να γράψουμε αυτούς τους κανόνες ως μετασχηματισμό της `Instruction`, υποθέτοντας ότι χρησιμοποιούμε την `NoOp` για την αναπαράσταση ενός βλαστού.

```
val stepSize = 10
// stepSize: Int = 10

def rule(i: Instruction): List[Instruction] =
  i match {
    case Forward(_) => List(forward(stepSize), forwa
rd(stepSize))
    case NoOp =>
      List(branch(turn(45.degrees), forward(stepSize
), noop),
           branch(turn(-45.degrees), forward(stepSiz
e), noop))
    case other => List(other)
  }
// rule: (i: doodle.turtle.Instruction)List[doodle.t
urtle.Instruction]
```

Παρατηρήστε ότι χρησιμοποιήσαμε την match στην `Instruction`, όπως κάναμε και σε άλλους αλγεβρικούς τύπους—όπως τους φυσικούς αριθμούς και την `List`—που έχουμε δει μέχρι τώρα. Εισάγοντας την `doodle.turtle.Instruction._` έχουμε πρόσβαση σε όλες τις εντολές της `Instruction`, οι οποίες είναι οι παρακάτω

- το `Forward(distance)`, όπου το `distance` είναι τύπου `Double`,
- το `Turn(angle)`, όπου το `angle` είναι μία γωνία,

- το `NoOp` και
- το `Branch(instructions)`, όπου το `instructions` είναι μία `List[Instruction]`.

Ως συνάρτηση, η `rule` έχει τύπο `Instruction => List[Instruction]`, αφού είναι πολύ πιθανόν να μετασχηματίσουμε την κάθε εντολή σε πολλές άλλες εντολές (όπως κάνουμε στην περίπτωση της `Forward`). Πώς μπορούμε να εφαρμόσουμε αυτόν τον κανόνα στην `List[Instruction]` ώστε να δημιουργήσουμε μία `List[Instruction]` (για παράδειγμα να το εφαρμόσουμε στο `List[noop]`); Μπορούμε να χρησιμοποιήσουμε την `map`;

See the solution

Υπάρχει μία μέθοδος της `List` που ονομάζεται `flatten` και μπορεί να μετατρέψει μία `List[List[A]]` σε `List[A]`. Θα *μπορούσαμε* να χρησιμοποιήσουμε έναν συνδυασμό της `map` και της `flatten` αλλά έχουμε μία ακόμη καλύτερη λύση. Ο συνδυασμός αυτών των δύο προκύπτει τόσο συχνά—και μάλιστα σε διάφορες περιπτώσεις που θα δούμε αργότερα—που δημιουργήθηκε μία ξεχωριστή μέθοδος για τον χειρισμό του. Η μέθοδος αυτή ονομάζεται `flatMap`.

Η εξίσωση τύπων για την `flatMap` είναι

```
List[A] flatMap (A => List[B]) = List[B]
```

και παρουσιάζεται γραφικά στην εικόνα fig. 52. Μπορούμε να δούμε ότι η `flatMap` έχει τον κατάλληλο τύπο ώστε να συνδυάσει την `rule` με την `List[Instruction]` για να ξαναδημιουργήσει την `List[Instruction]`.

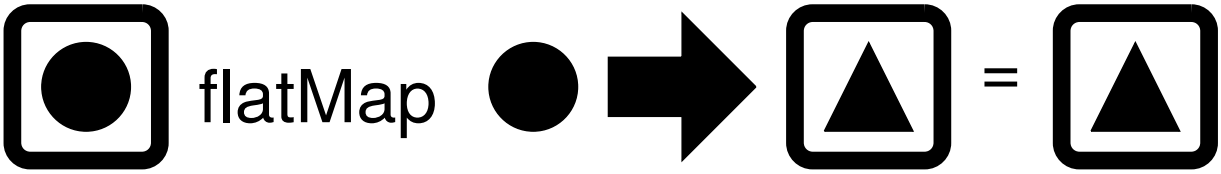


Figure 52: Γραφική αναπαράσταση της εξίσωσης τύπων για την flatMap.

Σε προηγούμενες αναφορές μας στην `map`, είπαμε ότι δεν μας επιτρέπει να αλλάξουμε τον αριθμό των στοιχείων μία λίστας. Δεν μπορούμε να δημιουργήσουμε ένα νέο “κουτί” με γραφικά χρησιμοποιώντας την `map`. Με την `flatMap` μπορούμε να αλλάξουμε το κουτί. Όσον αφορά τις λίστες,

μπορούμε να αλλάξουμε τον αριθμό των στοιχείων τους.

Ασκήσεις

Όλα Διπλά

Χρησιμοποιώντας την `flatMap`, γράψτε μία μέθοδο με όνομα `double` η οποία θα μετατρέπει μία `List` σε `List` στην οποία όλα τα στοιχεία θα εμφανίζονται δύο φορές. Για παράδειγμα

```
double(List(1, 2, 3))
// res0: List[Int] = List(1, 1, 2, 2, 3, 3)

double(List("do", "ray", "me"))
// res1: List[String] = List(do, do, ray, ray, me, m
e)
```

See the solution

Ή Τίποτα

Χρησιμοποιώντας την `flatMap`, γράψτε μία μέθοδο με όνομα `nothing` η οποία μετατρέπει μία `List` σε μία κενή `List`. Για παράδειγμα

```
nothing(List(1, 2, 3))
// res2: List[Int] = List()

nothing(List("do", "ray", "me"))
// res3: List[String] = List()
```

See the solution

Ξαναγράφοντας τους Κανόνες

Γράψτε μία μέθοδο με όνομα `rewrite` με δήλωση όπως την παρακάτω

```
def rewrite(instructions: List[Instruction],
            rule: Instruction => List[Instruction]):
 List[Instruction] =
  ???
```

Αυτή η μέθοδος πρέπει να εφαρμόζει την `rule` για να γραφούν ξανά οι εντολές στην `instructions`, εκτός από μερικά κλαδιά που θα χρειαστεί να τα χειριστείτε ξεχωριστά. Αν βρείτε μπροστά σας μία διακλάδωση, τότε θα πρέπει να ξαναγράψετε όλες τις εντολές που υπάρχουν μέσα της αλλά να αφήσετε την ίδια την διακλάδωση άθικτη.

Σημείωση: Θα πρέπει να περάσετε μία `List[Instruction]` στην `branch`, αφού η `branch` δέχεται μηδέν ή παραπάνω εντολές (οι οποίες ονομάζονται *varargs*). Για να μετασχηματίσετε την `List[Instruction]` σε μορφή την οποία δέχεται η `branch`, βάλτε μετά τις παραμέτρους το σύμβολο `:_*` όπως παρακάτω

```
val instructions = List(turn(45.degrees), forward(10
))
// instructions: List[doodle.turtle.Instruction] = L
ist(Turn(Angle(0.7853981633974483)), Forward(10.0))

branch(instructions:_*)
// res4: doodle.turtle.Instruction.Branch = Branch(L
ist(Turn(Angle(0.7853981633974483)), Forward(10.0)))
```

See the solution

Το Δικό σας L-System

Είμαστε πλέον έτοιμοι να δημιουργήσουμε ένα πλήρες L-system. Χρησιμοποιώντας την `rewrite` από παραπάνω, δημιουργήστε μία μέθοδο με όνομα `iterate` με την παρακάτω δήλωση

```
def iterate(steps: Int,
            seed: List[Instruction],
            rule: Instruction => List[Instruction]):
 List[Instruction] =
  ???
// iterate: (steps: Int, seed: List[doodle.turtle.In
struction], rule: doodle.turtle.Instruction => List[
doodle.turtle.Instruction])List[doodle.turtle.Instru
ction]
```

Θα πρέπει να εφαρμόζει αναδρομικά την `rule` στη `seed` για όσες επαναλήψεις ορίζει η `steps`.

See the solution

Φυτά και Άλλες Δημιουργίες

Δημιουργήστε την εικόνα fig. 53 και την fig. 54.

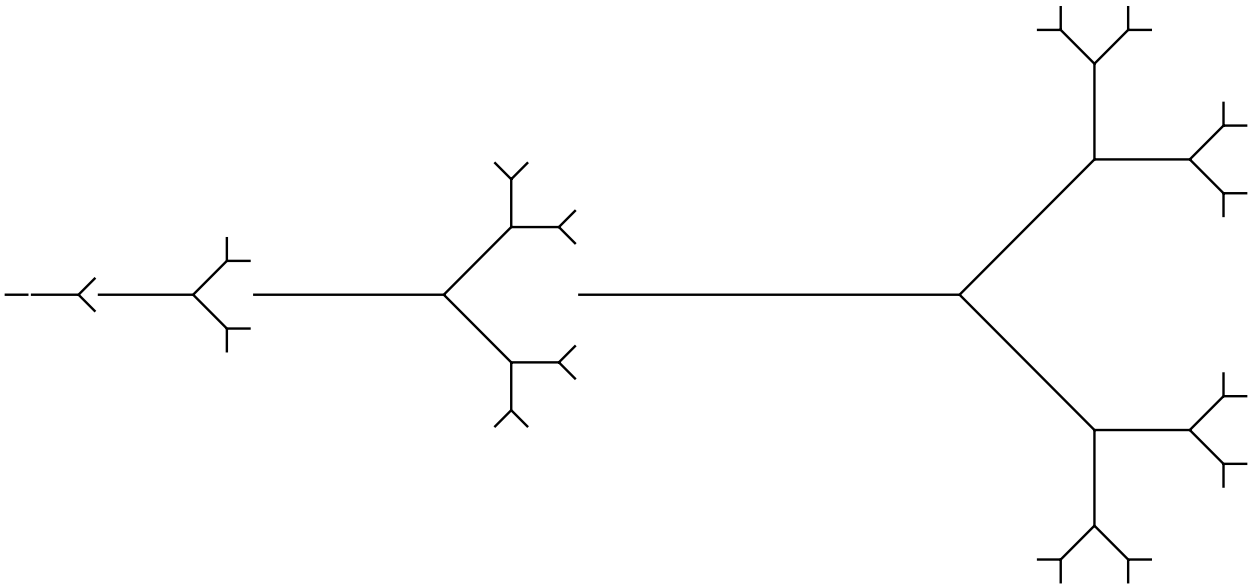


Figure 53: Πέντε επαναλήψεις ενός απλού L-system.

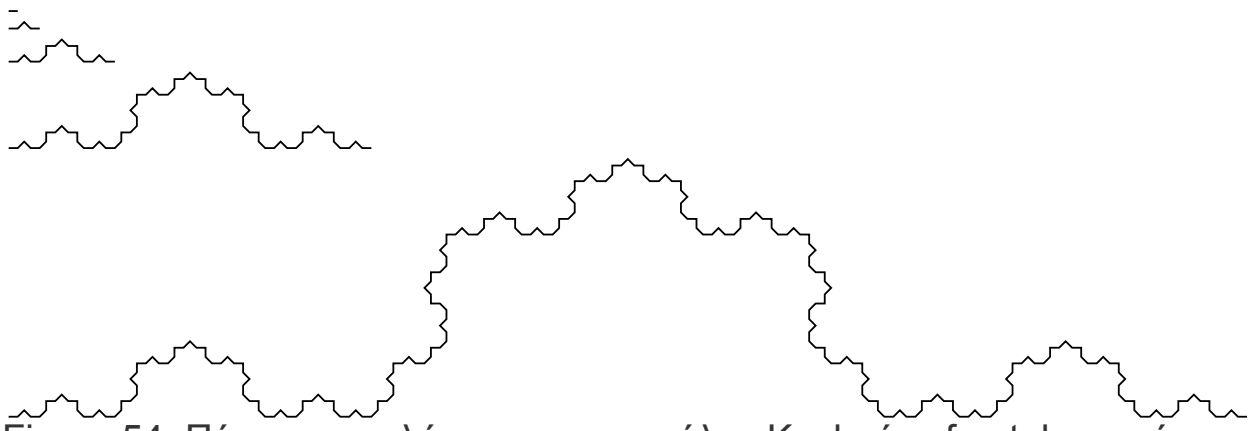


Figure 54: Πέντε επαναλήψεις της καμπύλης Koch, ένα fractal που είναι εύκολο να δημιουργηθεί χρησιμοποιώντας ένα L-System.

10.4 Ασκήσεις

10.4.1 Επίπεδο Πολύγωνο

Χρησιμοποιώντας τις μεθόδους του Turtle, την `Range` και την `flatMap`, ξαναγράψτε την μέθοδό σας ώστε να δημιουργήσετε ένα πολύγωνο. Η δήλωση της `polygon` είναι η παρακάτω

```
def polygon(sides: Int, sideLength: Double): Image =

  ???
```

See the solution

10.4.2 Επίπεδο Σπειροειδές

Χρησιμοποιώντας τις μεθόδους του Turtle, την `Range` και την `flatMap`, ξαναγράψτε την μέθοδό `squareSpiral` για να δημιουργήσετε ένα τετράγωνο σπειροειδές. Η δήλωση της `squareSpiral` είναι η παρακάτω

```
def squareSpiral(steps: Int, distance: Double, angle
: Angle, increment: Double): Image =
  ???
```

See the solution

10.4.3 Τέχνη με L-System

Σ' αυτή την άσκηση θέλουμε να χρησιμοποιήσετε την δημιουργικότητά σας για να κατασκευάσετε μία εικόνα ενός φυσικού αντικειμένου, χρησιμοποιώντας το δικό σας L-system. Έχετε ήδη δει αρκετά παραδείγματα και μπορείτε να τα χρησιμοποιήσετε ως έμπνευση.

11 Σύνθεση Αναπαραγωγικής Τέχνης

Σ’ αυτό το κεφάλαιο θα εξερευνήσουμε τεχνικές της αναπαραγωγικής τέχνης, οι οποίες με την σειρά τους θα μας επιτρέψουν να εξερευνήσουμε έννοιες-κλειδιά του συναρτησιακού προγραμματισμού. Θα δούμε:

- χρήσεις της `map` και της `flatMap` οι οποίες προχωράν πολύ μακρύτερα από τον χειρισμό συλλογών δεδομένων που έχουμε δει σε προηγούμενα κεφάλαια,
- πώς μπορούμε να χειριστούμε side effects (παρενέργειες) χωρίς να βάλουμε σε κίνδυνο την έννοια της αντικατάστασης που είδαμε σε προηγούμενο κεφάλαιο και
- μερικές ενδιαφέρουσες και όμορφες εικόνες που συνδυάζουν στοιχεία δομής αλλά και τυχαιότητας.

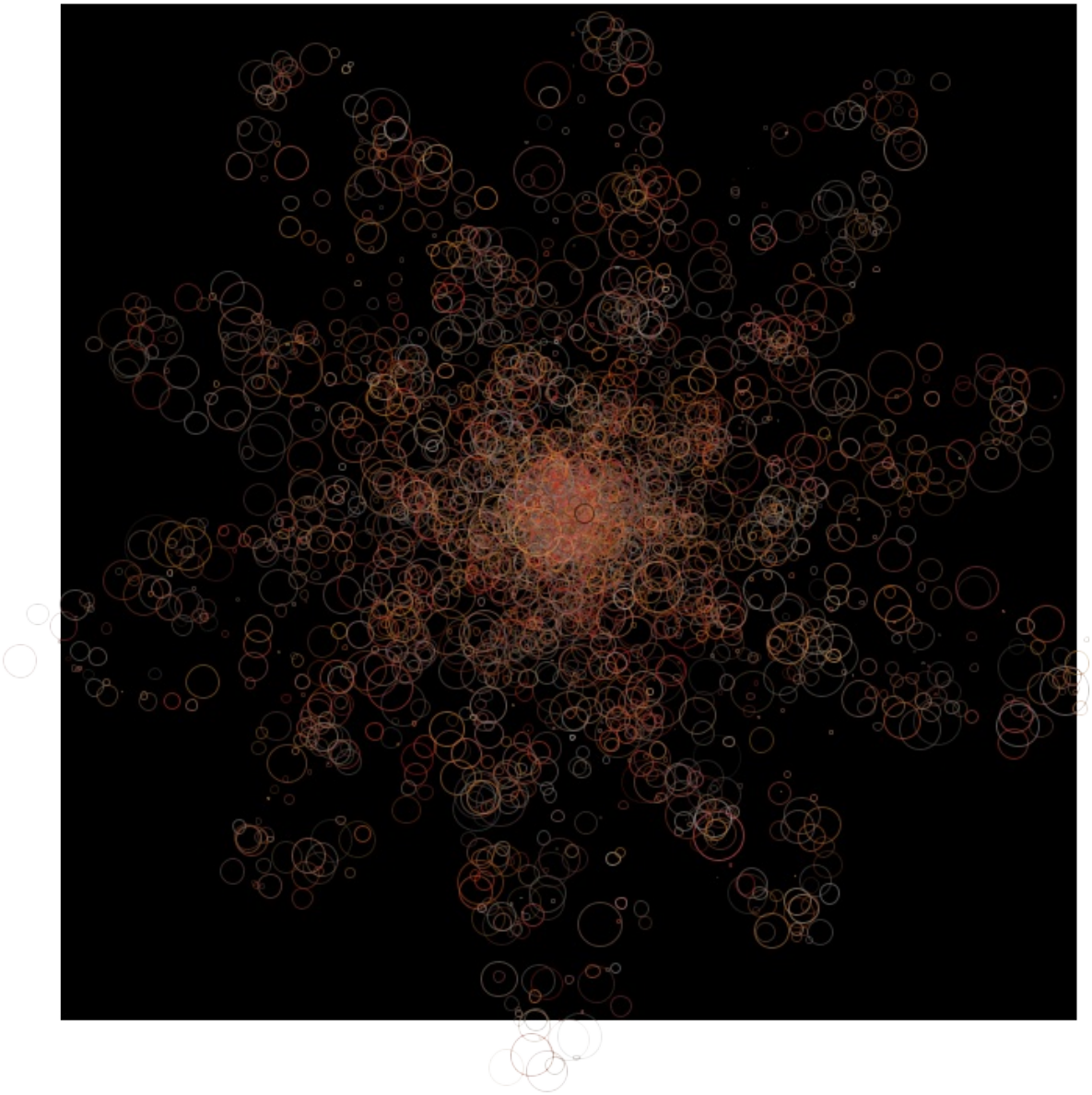


Figure 55: Ένα παράδειγμα εικόνας που έχει δημιουργηθεί με την χρήση τεχνικών αυτού του κεφαλαίου

Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.

```
import doodle.core._
import doodle.core.Image._
```

```
import doodle.syntax._
import doodle.jvm.Java2DFrame._
import doodle.backend.StandardInterpreter._
```

11.1 Αναπαραγωγική Τέχνη

Αναπαραγωγική αποκαλούμε την τέχνη της οποίας κάποιο μέρος της σύνθεσής της οφείλεται σε μία αυτόνομη διαδικασία. Συνεπώς, για εμάς αυτό σημαίνει ότι θα πρέπει να προσθέσουμε ένα στοιχείο τυχαιότητας.

Ας δούμε ένα πολύ απλό παράδειγμα. Μάθαμε προηγουμένως πώς να φτιάχνουμε ομόκεντρους κύκλους.

```
def concentricCircles(n: Int): Image =
  n match {
    case 0 => circle(10)
    case n => concentricCircles(n-1) on circle(n * 10)
  }
```

(Πλέον ξέρουμε ότι θα μπορούσαμε να έχουμε χρησιμοποιήσει την `Range` και κάποια μέθοδο όπως η `allOn`.)

Μάθαμε επίσης πώς μπορούμε να φτιάξουμε χρωματιστούς κύκλους χρησιμοποιώντας μία δεύτερη παράμετρο.

```
def concentricCircles(n: Int, color: Color): Image =
  n match {
    case 0 => circle(10) fillColor color
    case n => concentricCircles(n-1, color.spin(15.degrees)) on (circle(n * 10) fillColor color)
  }
```

Οι εικόνες που έχουν κατασκευαστεί με τον παραπάνω τρόπο είναι ωραίες αλλά και κάπως βαρετές, αφού είναι πολύ συνηθισμένες. Τι θα γίνονταν αν θέλαμε να κάνουμε μία τυχαία αλλαγή στην απόχρωση του χρώματος σε κάθε βήμα;

Η Scala μας παρέχει μερικές μεθόδους για παραγωγή *τυχαίων αριθμών*. Μία τέτοια μέθοδος είναι η `math.random`. Κάθε φορά που την καλούμε, λαμβάνουμε μία διαφορετική τιμή τύπου `Double` ανάμεσα στο 0.0 και το 1.01.

```
math.random
// res0: Double = 0.17724404214134482

math.random
// res1: Double = 2.3729438104180822E-4
```

Με την `math.random` θα μπορούσαμε να φτιάξουμε μία μέθοδο που θα επιστρέφει μία τυχαία `Angle` όπως παρακάτω.

```
def randomAngle: Angle =
  math.random.turns
// randomAngle: doodle.core.Angle

randomAngle
// res2: doodle.core.Angle = Angle(2.687226663780781
)

randomAngle
// res3: doodle.core.Angle = Angle(3.381094491539957
7)
```

Όμως γιατί δεν θέλουμε να το κάνουμε αυτό; Ποια βασική αρχή καταργείται;

See the solution

Τι μπορούμε να κάνουμε; Να υποφέρουμε από τις σφεντόνες και τα βέλη αυτών των ανόητων υπολογιστικών μοντέλων ή να παλέψουμε μέσα σε μία θάλασσα από side-effects και τελικά να νικήσουμε; Η επιλογή είναι βασικά μία.

11.2 Τυχαιότητα χωρίς Επιπτώσεις

Η λύση στο πρόβλημά μας είναι να διαχωρίσουμε τον τρόπο χρήσης των

τυχαίων αριθμών από την διαδικασία της πραγματικής παραγωγής τους. Ακούγεται μπερδεμένο αλλά στην πραγματικότητα κάναμε το ίδιο με τις `Image` σε όλο το βιβλίο.

- περιγράφουμε μία `Image` χρησιμοποιώντας μονοπάτια και μεθόδους όπως οι `beside`, `above` και `on` και
- σχεδιάζουμε την `Image` μόνο όταν καλούμε την `draw`.

Το ίδιο ακριβώς κάνουμε και με τον τύπο `Random` του Doodle. Για να έχουμε πρόσβαση σ' αυτόν πρέπει πρώτα να φορτώσουμε το πακέτο `doodle.random`.

```
import doodle.random._
```

Τώρα μπορούμε να φτιάξουμε στοιχεία που περιγράφουν την δημιουργία τυχαίων αριθμών

```
val randomDouble = Random.double
// randomDouble: doodle.random.Random[Double] = Free
(...)
```

Όμως κανένας τυχαίος αριθμός δεν παράγεται πραγματικά πριν καλέσουμε την μέθοδο `run`.

```
randomDouble.run
// res0: Double = 0.7975173865220604
```

Ο τύπος `Random[Double]` μας προϊδεάζει για την παραγωγή ενός τυχαίου `Double` αριθμού. Ακριβώς όπως και με την `Image` και την `draw`, η αντικατάσταση της `Random` ισχύει μέχρι να την εκτελέσουμε.

Ο πίνακας tbl. 6 δείχνει μερικούς τρόπους κατασκευής τέτοιων στοιχείων.

Table 6: Μερικές από τις μεθόδους που δημιουργούν στοιχεία παραγωγής τυχαίων τιμών.

Μέθοδος	**Περιγραφή**	**Παράδειγμα**
`Random.always(value)`	Ένα στοιχείο που	`Random.always(10)`

	παράγει πάντα την δοσμένη τιμή.	
`Random.double`	Ένα στοιχείο που παράγει τιμές `Double`, ομοιόμορφα κατανεμημένες μεταξύ του `0.0` και του `1.0`.	`Random.double`
`Random.int`	Ένα στοιχείο που παράγει τιμές `Int`, ομοιόμορφα κατανεμημένες.	`Random.int`
`Random.natural(limit)`	Ένα στοιχείο που παράγει τιμές `Int`, ομοιόμορφα κατανεμημένες, μεγαλύτερες ή ίσες με το `0` και μικρότερες από το `1`.	`Random.natural(10)`
`Random.oneOf(value, ...)`	Ένα στοιχείο που παράγει μία από τις δοσμένες τιμές, με ίση πιθανότητα.	`Random.oneOf("A", "B", "C")`

11.2.1 Συντάσσοντας Τυχαίες Τιμές

Είδαμε πώς δημιουργούμε στοιχεία παραγωγής τυχαίων τιμών. Πώς τα συντάσσουμε μέσα σε πιο ενδιαφέροντα προγράμματα; Για παράδειγμα, πώς μπορούμε να μετατρέψουμε μία τυχαία `Double` σε τυχαία `Angle`; Ίσως είναι δελεαστική η κλήση της `run` κάθε φορά που θέλουμε να χειριστούμε ένα τυχαίο αποτέλεσμα αλλά αυτό θα απειλούσε την έννοια

της αντικατάστασης, κάτι που θέλουμε να αποφύγουμε.

Θυμηθείτε ότι όταν μιλήσαμε για την `map` σε προηγούμενα κεφάλαια είπαμε ότι μπορεί να μετασχηματίσει τα στοιχεία μίας λίστας αλλά κρατάει την δομή της (τον αριθμό των στοιχείων που περιέχει). Η ίδια αναλογία εφαρμόζεται και στην μέθοδο `map` για τυχαίους αριθμούς. Μας επιτρέπει να μετασχηματίσουμε το τυχαίο στοιχείο—την τιμή που παράγει όταν εκτελείται—αλλά δεν μας επιτρέπει να αλλάξουμε την δομή του. Σ' αυτή την περίπτωση, με την έννοια "δομή" εννοούμε την εισαγωγή περισσότερης τυχαιότητας.

Μπορούμε να δημιουργήσουμε ένα στοιχείο παραγωγής τυχαίων τιμών και να εφαρμόσουμε έναν *ντετερμινιστικό* μετασχηματισμό χρησιμοποιώντας την `map`, αλλά δεν μπορούμε να δημιουργήσουμε μία τυχαία τιμή και μετά να την χρησιμοποιήσουμε ως είσοδο σε μία διαδικασία που δημιουργεί κάποια άλλη τυχαία τιμή.

Παρακάτω μπορείτε να δείτε πώς μπορούμε να δημιουργήσουμε μία τυχαία γωνία.

```
val randomAngle: Random[Angle] =
  Random.double.map(x => x.turns)
```

Όταν τρέξουμε την `RandomAngle` θα παραχθούν τυχαίες γωνίες

```
randomAngle.run
// res1: doodle.core.Angle = Angle(2.6036940741128944)

randomAngle.run
// res2: doodle.core.Angle = Angle(4.222971141292186)
```

Ασκήσεις

Τυχαία Χρώματα

Με δεδομένη την `randomAngle` από παραπάνω, φτιάξτε μία μέθοδο η οποία δέχεται τον κορεσμό και την φωτεινότητα και παράγει ένα τυχαίο

χρώμα. Η δήλωση της μεθόδου σας θα πρέπει να είναι όπως η παρακάτω

```
def randomColor(s: Normalized, l: Normalized): Rando
m[Color] =
  ???
```

Αυτός είναι ένας ντετερμινιστικός μετασχηματισμός της εξόδου της `randomAngle` και μπορούμε να τον υλοποιήσουμε χρησιμοποιώντας την `map`.

```
def randomColor(s: Normalized, l: Normalized): Rando
m[Color] =
  randomAngle map (hue => Color.hsl(hue, s, l))
```

Τυχαίοι Κύκλοι

Γράψτε μία μέθοδο που δέχεται την ακτίνα και ένα `Random[Color]` (τυχαίο χρώμα) και παράγει έναν κύκλο με την συγκεκριμένη ακτίνα, χρωματισμένος τυχαία. Η δήλωση της μεθόδου θα πρέπει να είναι όπως η παρακάτω

```
def randomCircle(r: Double, color: Random[Color]): R
andom[Image] =
  ???
```

Για άλλη μία φορά, είναι ένας ντετερμινιστικός μετασχηματισμός τυχαίου χρώματος και άρα μπορούμε να χρησιμοποιήσουμε την `map`.

```
def randomCircle(r: Double, color: Random[Color]): R
andom[Image] =
  color map (fill => Image.circle(r) fillColor fill)
```

11.3 Συνδυάζοντας Τυχαίες Τιμές

> Εκτός από τα παραπάνω imports, σ' αυτή την ενότητα θα

προσθέσουμε και το παρακάτω:

```
import doodle.random._
```

Μέχρι τώρα έχουμε δει πώς να συντάσσουμε συναρτήσεις για παραγωγή τυχαίων αριθμών χρησιμοποιώντας τον τύπο `Random` καθώς και πώς να φτιάχνουμε ντετερμινιστικούς μετασχηματισμούς για κάποια τυχαία τιμή χρησιμοποιώντας την `map`. Σ' αυτή την ενότητα, θα δούμε πώς μπορούμε να κάνουμε έναν τυχαίο (ή στοχαστικό αν προτιμάτε τους πιο τεχνικούς όρους) μετασχηματισμό μίας τυχαίας τιμής χρησιμοποιώντας την `flatMap`.

Θα ξεκινήσουμε με την δημιουργία της μεθόδου `randomConcentricCircles`, η οποία κατασκευάζει ομόκεντρους κύκλους με τυχαία απόχρωση, χρησιμοποιώντας τις μεθόδους που αναπτύξαμε στην προηγούμενη ενότητα.

Αρχικά, θα γράψουμε τον κώδικα που δημιουργεί τους ομόκεντρους κύκλους με ντετερμινιστικά χρώματα, με την βοήθεια των εργαλείων που αναπτύξαμε προηγουμένως.

```
def concentricCircles(count: Int, size: Int, color:
Color): Image =
  count match {
    case 0 => Image.empty
    case n =>
      Image.circle(size).fillColor(color) on concent
ricCircles(n-1, size + 5, color.spin(15.degrees))
  }

def randomAngle: Random[Angle] =
  Random.double.map(x => x.turns)

def randomColor(s: Normalized, l: Normalized): Rando
m[Color] =
  randomAngle map (hue => Color.hsl(hue, s, l))
```

```
def randomCircle(r: Double, color: Random[Color]): R
andom[Image] =
  color map (fill => Image.circle(r) fillColor fill)
```

Ας φτιάξουμε έναν σκελετό για την μέθοδο `randomConcentricCircles`.

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  ???
```

Η σημαντική αλλαγή εδώ, είναι ότι επιστρέφουμε μία `Random[Image]` και όχι απλώς μία `Image`. Ξέρουμε ότι είναι περίπτωση δομημένης αναδρομής με φυσικούς αριθμούς άρα το μόνο που πρέπει να κάνουμε είναι να συμπληρώσουμε το σώμα της μεθόδου.

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => ???
    case n => ???
  }
```

Η βασική περίπτωση είναι η `Random.always(Image.empty)`, που είναι το ισοδύναμο της `Image.empty` για ντετερμινιστικές περιπτώσεις.

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n => ???
  }
```

Τι γίνεται με την αναδρομική περίπτωση; Θα μπορούσαμε να προσπαθήσουμε να χρησιμοποιήσουμε το παρακάτω

```
val randomPastel = randomColor(0.7.normalized, 0.7.n
ormalized)
```

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => Image.empty
    case n =>
      randomCircle(size, randomPastel) on randomConc
entricCircles(n-1, size + 5)
  }
// <console>:35: error: type mismatch;
//  found   : doodle.core.Image
//  required: doodle.random.Random[doodle.core.Image
]
//     (which expands to)  cats.free.Free[doodle.ran
dom.RandomOp,doodle.core.Image]
//            case 0 => Image.empty
//                                ^
// <console>:37: error: value on is not a member of
doodle.random.Random[doodle.core.Image]
//              randomCircle(size, randomPastel) on
randomConcentricCircles(n-1, size + 5)
//                                                  ^
```

αλλά δεν μεταγλωττίζεται. Η `randomConcentricCircles` αλλά και η `randomCircle` αξιολογούνται με `Random[Image]`. Δεν υπάρχει μέθοδος `on` στην `Random[Image]`, άρα αυτός ο κώδικας δεν μπορεί να λειτουργήσει.

Αφού αυτός είναι ένας μετασχηματισμός δύο τιμών `Random[Image]`, είναι φανερό ότι χρειαζόμαστε κάποια μέθοδο που θα μας επιτρέψει να μετασχηματίσουμε *δύο* `Random[Image]` και όχι μόνο μία όπως μπορούμε να κάνουμε με την `map`. Θα μπορούσαμε να ονομάσουμε αυτή την μέθοδο `map2` και να φανταστούμε ότι μπορούμε να γράψουμε κώδικα όπως τον παρακάτω

```
randomCircle(size, randomPastel).map2(randomConcentr
icCircles(n-1, size + 5)){
  (circle, circles) => circle on circles
}
```

Πιθανόν να χρειαζόμασταν και `map3`, `map4`, και ούτω καθεξής. Αντί γι' αυτές τις ειδικές περιπτώσεις, μπορούμε να χρησιμοποιήσουμε την `flatMap` σε συνδυασμό με την `map`.

```
randomCircle(size, randomPastel) flatMap { circle =>
  randomConcentricCircles(n-1, size + 5) map { circl
es =>
    circle on circles
  }
}
```

Μπορείτε να δείτε τον ολοκληρωμένο κώδικα παρακάτω

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      randomCircle(size, randomPastel) flatMap { cir
cle =>
        randomConcentricCircles(n-1, size + 5) map {
 circles =>
          circle on circles
        }
      }
  }
```

Ένα παράδειγμα εξόδου του παραπάνω κώδικα φαίνεται στην εικόνα fig. 56.

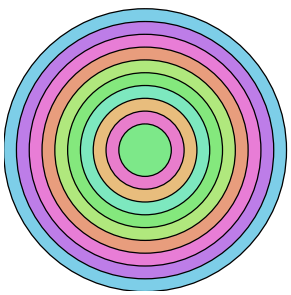


Figure 56: Το αποτέλεσμα μίας εκτέλεσης της `randomConcentricCircles(10, 10).run.draw`

Ας δούμε όμως πιο αναλυτικά αυτόν τον συνδυασμό της `flatMap` και της `map`, για να καταλάβουμε πώς λειτουργεί.

11.3.1 Άλγεβρα Τύπων

Κατά την γνώμη μας, ο πιο εύκολος τρόπος για να καταλάβετε τον παρακάτω κώδικα είναι να προσέξετε τους τύπους.

```
randomCircle(size, randomPastel) flatMap { circle =>
  randomConcentricCircles(n-1, size + 5) map { circl
es =>
    circle on circles
  }
}
```

Ξεκινώντας από μέσα προς τα έξω, έχουμε

```
{ circles =>
    circle on circles
}
```

η οποία είναι μία συνάρτηση με τύπο

```
Image => Image
```

Γύρω από αυτό έχουμε

```
randomConcentricCircles(n-1, size + 5) map { circles
 =>
    circle on circles
  }
```

Ξέρουμε ότι η `randomConcentricCircles(n-1, size + 5)` έχει τύπο `Random[Image]`. Αντικαθιστώντας με τον τύπο `Image => Image` που βρήκαμε προηγουμένως, έχουμε

```
Random[Image] map (Image => Image)
```

Τώρα μπορούμε να ασχοληθούμε με ολόκληρη την έκφραση

```
randomCircle(size, randomPastel) flatMap { circle =>
  randomConcentricCircles(n-1, size + 5) map { circl
es =>
    circle on circles
  }
}
```

Η `randomCircle(size, randomPastel)` έχει τύπο `Random[Image]`. Κάνοντας πάλι αντικαταστάσεις, παίρνουμε μία εξίσωση τύπων για όλη την έκφραση.

```
Random[Inage] flatMap (Random[Image] map (Image => I
mage))
```

Τώρα μπορούμε να εφαρμόσουμε αυτές τις εξισώσεις τύπων στην `map` και στην `flatMap` που είδαμε προηγουμένως:

```
F[A] map (A => B) = F[B]
F[A] flatMap (A => F[B]) = F[B]
```

Δουλεύοντας και πάλι από μέσα προς τα έξω, χρησιμοποιούμε πρώτα την εξίσωση τύπου για την `map`, η οποία απλοποιεί την έκφραση τύπων ως εξής

```
Random[Inage] flatMap (Random[Image])
```

Τώρα μπορούμε να εφαρμόσουμε την εξίσωση τύπων και στην `flatMap` παράγοντας μόνο

```
Random[Image]
```

Αυτό μας λέει ότι το αποτέλεσμα έχει τον τύπο που θέλουμε. Παρατηρήστε ότι εφαρμόσαμε την μέθοδο της αντικατάστασης σε επίπεδο τύπων—την ίδια τεχνική χρησιμοποιούμε συνήθως και σε επίπεδο τιμών.

Ασκήσεις

Μην ξεχάσετε να κάνετε import την `doodle.random._` πριν προσπαθήσετε να λύσετε τις παρακάτω ασκήσεις.

Τυχαιότητα και Τυχαιότητα

Ποια είναι η διαφορά μεταξύ του αποτελέσματος της `programOne` και της `programTwo` που μπορείτε να δείτε παρακάτω; Γιατί διαφέρουν;

```
def randomCircle(r: Double, color: Random[Color]): R
andom[Image] =
  color map (fill => Image.circle(r) fillColor fill)

def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      randomCircle(size, randomPastel) flatMap { cir
cle =>
        randomConcentricCircles(n-1, size + 5) map {
 circles =>
          circle on circles
        }
      }
  }

val circles = randomConcentricCircles(5, 10)
val programOne =
  circles flatMap { c1 =>
    circles flatMap { c2 =>
      circles map { c3 =>
        c1 beside c2 beside c3
      }
    }
  }
```

```
val programTwo =
  circles map { c => c beside c beside c }
```

See the solution

Χρωματιστά Κουτιά

Ας επιστρέψουμε σ' ένα πρόβλημα που είχαμε συναντήσει στην αρχή του βιβλίου: τον σχεδιασμό χρωματιστών κουτιών. Αυτή τη φορά θα κάνουμε την διαβάθμιση των χρωμάτων λίγο πιο ενδιαφέρουσα αναθέτοντάς τους τυχαία χρώματα.

Θυμηθείτε την βασική δομημένη αναδρομή για δημιουργία μίας σειράς από κουτιά

```
def rowOfBoxes(count: Int): Image =
  count match {
    case 0 => rectangle(20, 20)
    case n => rectangle(20, 20) beside rowOfBoxes(n-
1)
  }
// rowOfBoxes: (count: Int)doodle.core.Image
```

Ας αλλάξουμε τον παραπάνω κώδικα, όπως κάναμε με τους ομόκεντρους κύκλους, ώστε να χρωματίσουμε τυχαία το κάθε κουτί. *Βοήθεια:* μπορεί να βρείτε βολική την χρήση του κώδικα που γράψαμε προηγουμένως για την `randomConcentricCircles`. Στην εικόνα fig. 57 μπορείτε να δείτε ένα παράδειγμα εξόδου.

Figure 57: Κουτιά τυχαίων χρωμάτων.

See the solution

11.4 Εξερευνώντας την Random

Μέχρι τώρα έχουμε δει μόνο τα βασικά για την χρήση της `Random`. Σ' αυτή την ενότητα θα δούμε περισσότερα χαρακτηριστικά της και θα τα χρησιμοποιήσουμε για να δημιουργήσουμε πιο ενδιαφέρουσες εικόνες.

Εκτός από τα imports που εισάγουμε συνήθως, σ' αυτή την ενότητα θα προσθέσουμε και τα παρακάτω:

```
import doodle.random._
```

11.4.1 Κανονικές Κατανομές

Πολύ συχνά, όταν χρησιμοποιούμε τυχαίους αριθμούς στην αναπαραγωγική τέχνη, επιλέγουμε συγκεκριμένες κατανομές, λόγω των σχημάτων που μπορουν να δημιουργήσουν. Για παράδειγμα, η εικόνα fig. 58 δείχνει χίλια τυχαία σημεία που έχουν παραχθεί χρησιμοποιώντας μία ομοιόμορφη κατανομή, μία κανονική κατανομή (ή Γκαουσιανή) και μία κανονική κατανομή στο τετράγωνο.

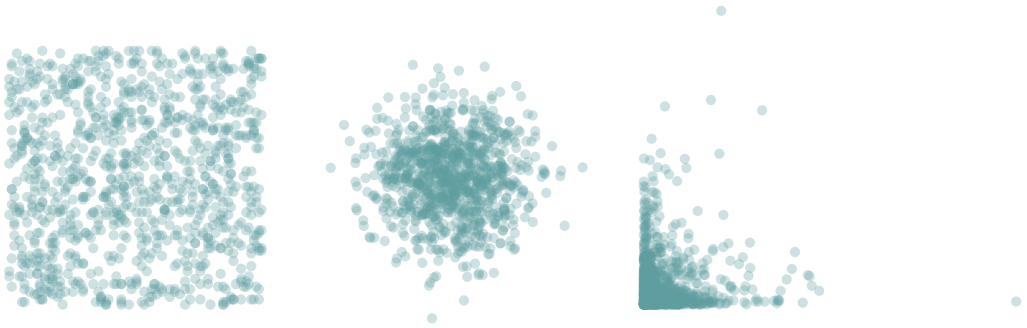


Figure 58: Σημεία που έχουν τοποθετηθεί σύμφωνα με την ομοιόμορφη κατανομή, την κανονική κατανομή και την κανονική κατανομή υψωμένη στο τετράγωνο.

Όπως μπορείτε να δείτε, η κανονική κατανομή παράγει περισσότερα σημεία προς το κέντρο από την ομοιόμορφη κατανομή.

Το Doodle παρέχει δύο μεθόδους για την δημιουργία κανονικά κατανεμημένων αριθμών, με τις οποίες μπορούμε να φτιάξουμε και άλλες κατανομές. Η κανονική κατανομή ορίζεται από δύο παραμέτρους, την *μέση τιμή* (mean), η οποία ορίζει το κέντρο της κατανομής και την *τυπική απόκλιση* (standard deviation), η οποία ορίζει το εύρος της κατανομής. Οι αντίστοιχες μέθοδοι στο Doodle είναι οι παρακάτω

- η `Random.normal`, που παράγει έναν `Double` από μία κανονική κατανομή με μέση τιμή 0 και τυπική απόκλιση 1.0 και

- η `Random.normal(mean, stdDev)`, που παράγει έναν `Double` από μία κανονική κατανομή με δοσμένη από τον χρήστη μέση τιμή και τυπική απόκλιση.

11.4.2 Δομημένη Τυχαιότητα

Απο δομημένες εικόνες περάσαμε σε τυχαίες. Θα ήταν ωραία αν μπορούσαμε να βρούμε μία μέση λύση που να περιλαμβάνει στοιχεία τυχαιότητας και δομής. Μπορούμε να χρησιμοποιήσουμε την `flatMap` για να το πετύχουμε—με τη `flatMap` μπορούμε να χρησιμοποιήσουμε μία τυχαία παραγόμενη τιμή ώστε να δημιουργήσουμε μία άλλη τυχαία τιμή. Έτσι δημιουργείται μία εξάρτηση μεταξύ των τιμών—η προηγούμενη τυχαία τιμή επηρεάζει την επόμενη τυχαία τιμή που θα παραχθεί.

Για παράδειγμα, μπορούμε να φτιάξουμε μία μέθοδο η οποία με δεδομένο ένα χρώμα θα αλλάζει τυχαία την απόχρωσή του.

```
def nextColor(color: Color): Random[Color] = {
  val spin = Random.normal(15.0, 10.0)
  spin map { s => color.spin(s.degrees) }
}
```

Χρησιμοποιώντας την `nextColor`, μπορούμε να φτιάξουμε μία σειρά από κουτιά με διαβάθμιση χρωμάτων που θα είναι τυχαία και δομημένη: κάθε χρώμα της διαβάθμισης είναι μία διαφοροποίηση του προηγούμενου.

```
def coloredRectangle(color: Color, size: Int): Image
 =
  rectangle(size, size).
    lineWidth(5.0).
    lineColor(color.spin(30.degrees)).
    fillColor(color)

def randomGradientBoxes(count: Int, color: Color): R
andom[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
```

```
      val box = coloredRectangle(color, 20)
      val boxes = nextColor(color) flatMap { c => ra
ndomGradientBoxes(n-1, c) }
      boxes map { b => box beside b }
  }
```

Ένα παράδειγμα εξόδου μπορείτε να δείτε στην εικόνα fig. 59.

Figure 59: Κουτιά χρωματισμένα με διαβαθμίσεις χρωμάτων που είναι μερικώς τυχαίες.

Ασκήσεις

Συστήματα Σωματιδίων

Ένα *σύστημα σωματιδίων* είναι μία τεχνική που χρησιμοποιείται στα γραφικά υπολογιστών ώστε να δημιουργηθεί μεγάλος αριθμός “σωματιδίων” που κινούνται σύμφωνα με απλούς κανόνες. Στην εικόνα fig. 60 μπορείτε να δείτε ένα παράδειγμα ενός συστήματος σωματιδίων που προσομοιώνει μία φωτιά και καπνό. Γι’ αυτούς που τους αρέσουν τα μαθηματικά, ένα σύστημα σωματιδίων είναι βασικά μία *στοχαστική διαδικασία* ή ένας *τυχαίος περίπατος*.

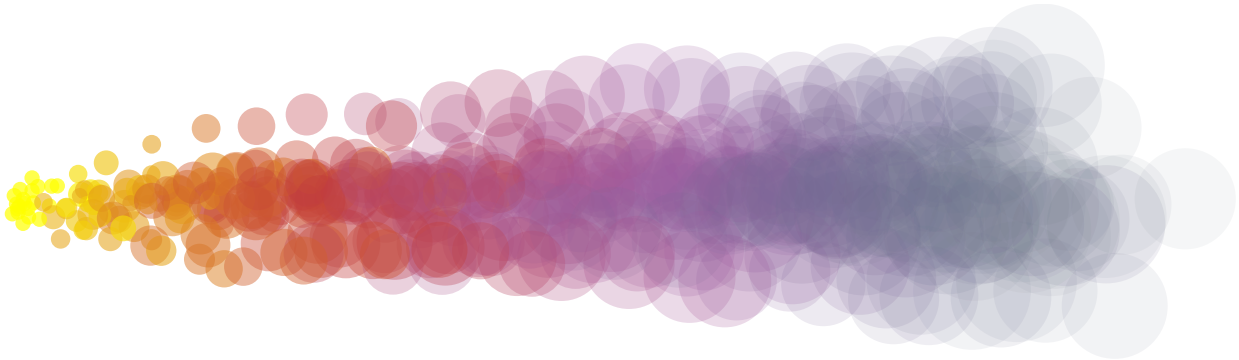


Figure 60: Μία προσομοίωση φωτιάς με καπνό που δημιουργήθηκε με την χρήση ενός συστήματος σωματιδίων.

Σ’ αυτή την άσκηση θα δημιουργήσουμε ένα σύστημα σωματιδίων που θα σας δείξει ένα ευέλικτο σύστημα για να πειραματιστείτε. Θα ξεκινήσουμε με ένα κλασικό σύστημα και μετά θα το αλλάξουμε έτσι ώστε

να δημιουργήσουμε επαναχρησιμοποιήσιμα κομμάτια.

Παρακάτω μπορείτε να δείτε πώς δουλεύει ένα σύστημα σωματιδίων. Για να ζωγραφίσουμε ένα μεμονωμένο σωματίδιο πρέπει

- να επιλέξουμε μία αρχική θέση,
- να αποφασίσουμε για πόσα χρονικά βήματα θέλουμε να κινήσουμε το σύστημα σωματιδίων και
- σε κάθε χρονικό βήμα, η νέα θέση του σωματιδίου να είναι ίση με την θέση του προηγούμενου βήματος συν κάποιον τυχαίο θόρυβο (και πιθανόν και κάποια μη-τυχαία (ντετερμινιστική) κίνηση, όπως για παράδειγμα ταχύτητα ή επιτάχυνση).

Ένα σύστημα σωματιδίων είναι απλώς μία συλλογή σωματιδίων—στην εικόνα fig. 60 μπορείτε να δείτε 20 σωματίδια μετά από 20 βήματα.

Στην παραπάνω περιγραφή, σπάσαμε σε μικρά κομμάτια τα στοιχεία που αποτελούν ένα σύστημα σωματιδίων. Τώρα το μόνο που έχουμε να κάνουμε είναι να τα υλοποιήσουμε.

Η αρχική θέση μπορεί να είναι ένα οποιοδήποτε `Random[Point]` (τυχαίο σημείο). Δημιουργήστε ένα!

See the solution

Ας φτιάξουμε μία μέθοδο με όνομα `step`, η οποία θα πραγματοποιεί ένα βήμα στο σύστημα σωματιδίων. Ο σκελετός της θα είναι ο παρακάτω

```
def step(current: Point): Random[Point] =
  ???
```

Πρέπει να αποφασίσουμε πώς θα χρησιμοποιήσουμε το σημείο στο οποίο βρισκόμαστε τώρα ώστε να δημιουργείται το επόμενο σημείο. Προτείνουμε την πρόσθεση τυχαίου θορύβου και μία σταθερά που θα ονομάζεται “drift”. Μ’ αυτόν τον τρόπο θα εξασφαλιστεί η κίνηση των σημείων προς μία συγκεκριμένη κατεύθυνση. Για παράδειγμα, μπορούμε να αυξήσουμε την συντεταγμένη `x` κατά 10 μονάδες και έτσι θα προκληθεί η κλίση των σωματιδίων προς την δεξιά πλευρά της οθόνης καθώς και κάποιος θόρυβος, κανονικά κατανεμημένος, στις συντεταγμένες `x` και `y`.

See the solution

Τώρα που μπορούμε να δώσουμε ένα βήμα (`step`) σ' ενα σωματίδιο, πρέπει να συνδέσουμε μία σειρά βημάτων ώστε να πάρουμε έναν "περίπατο" (`walk`). Υπάρχει κάτι που πρέπει να προσέξουμε εδώ: θέλουμε να ζωγραφίσουμε και τα ενδιάμεσα στάδια, οπότε θα ορίσουμε δύο μεθόδους:

- μία μέθοδο που μετατρέπει ένα σημείο σε εικόνα
- μία μέθοδο με όνομα `walk` που παράγει μία `Random[Image]` (τυχαία εικόνα)

Οι σκελετοί τους είναι οι παρακάτω

```
def render(point: Point): Image =
  ???

def walk(steps: Int): Random[Image] =
  ???
```

Η υλοποίηση της μεθόδου `render` μπορεί να γίνει με όποιον τρόπο θέλετε. Για την υλοποίηση της `walk`, θα πρέπει να καλέσετε την `step` ώστε να πάρετε το επόμενο `Point` (σημείο) και μετά να καλέσετε την `render` ώστε να μετατρέψετε το σημείο σε κάτι που μπορεί να σχεδιαστεί. Θα πρέπει επίσης να έχετε έναν συσσωρευτή για την εικόνα που έχει δημιουργηθεί μέχρι τότε. Βοήθεια: μπορεί να σας φανεί χρήσιμος ο ορισμός μίας βοηθητικής παραμέτρου στην `walk`.

See the solution

Θα πρέπει να είστε σε θέση να καλέσετε την `walk` για να πάρετε κάποιο αποτέλεσμα.

Το τελικό βήμα είναι η δημιουργία των σωματιδίων. Δημιουργήστε μία μέθοδο με όνομα `particleSystem`, με τον παρακάτω σκελετό

```
def particleSystem(particles: Int, steps: Int): Rand
om[Image] =
  ???
```

ο οποίος είναι κατάλληλος γι' αυτό που θέλουμε να κάνουμε.

See the solution

Δείτε το αποτέλεσμα και “πειράξτε το” μέχρι να φτιάξετε κάτι που σας αρέσει. Εμείς δεν είμαστε πολύ ευχαριστημένοι με το αποτέλεσμα του κώδικά μας. Τα σωματίδια είναι πολύ κοντά μεταξύ τους και τα χρώματα δεν είναι πολύ ωραία. Για ένα πιο ενδιαφέρον αποτέλεσμα θα μπορούσαμε να προσθέσουμε λίγο περισσότερο θόρυβο, να αλλάξουμε το αρχικό χρώμα και να μειώσουμε το φάσμα των χρωμάτων.

Τυχαίες Αλλαγές

Η υλοποίηση της `particleSystem` που είδαμε παραπάνω αφορά μόνο μία συγκεκριμένη επιλογή συστήματος σωματιδίων. Για να κάνουμε πιο εύκολο τον πειραματισμό θα μπορούσαμε να κάνουμε κάποιες αλλαγές στην `walk` και την `start`. Πώς πιστεύετε ότι μπορούμε να το κάνουμε αυτό;

See the solution

Φτιάξτε το.

See the solution

Αυτός ο κώδικας δεν είναι ικανοποιητικός. Οι περισσότερες παράμετροι της `particleSystem` πρέπει να περαστούν μόνο στην `walk`. Αυτές οι παράμετροι δεν αλλάζουν μέσα στην δομημένη αναδρομή της `particleSystem`. Σ’ αυτό το σημείο μπορούμε να εφαρμόσουμε την μέθοδο της αντικατάστασης—μπορούμε να αντικαταστήσουμε μία κλήση μεθόδου με την τιμή με την οποία αξιολογείται—και να αφαιρέσουμε την `walk` και τις σχετικές παραμέτρους από την `particleSystem`.

```
def particleSystem(particles: Int, walk: Random[Imag
e]): Random[Image] = {
  particles match {
    case 0 => Random.always(Image.empty)
    case n => walk flatMap { img1 =>
      particleSystem(n-1, walk) map { img2 =>
        img1 on img2
      }
    }
  }
}
```

Αν έχετε συνηθίσει να προγραμματίζετε με δομημένο προγραμματισμό τότε ίσως η παραπάνω τεχνική σας φανεί πολύ διαφορετική στον τρόπο σκέψης. Θυμηθείτε ότι κάναμε μεγάλη προσπάθεια για να σιγουρευτούμε ότι η χρήση τυχαίων αριθμών δεν θα παραβιάσει την αρχή της αντικατάστασης μέχρι την στιγμή που καλείται η `run`. Η μέθοδος `walk`, στην πραγματικότητα δεν δημιουργεί έναν τυχαίο περίπατο. Αντιθέτως, περιγράφει την δημιουργία ενός τυχαίου περιπάτου καθώς εκτελείται ο κώδικάς της. Αυτός ο διαχωρισμός μεταξύ περιγραφής και πράξης σημαίνει ότι μπορεί να χρησιμοποιηθεί αντικατάσταση. Η περιγραφή της υλοποίησης ενός τυχαίου περιπάτου μπορεί να χρησιμοποιηθεί για την δημιουργία πολλών και διαφορετικών τυχαίων περιπάτων.

11.5 For Comprehension

> Σε αυτήν την ενότητα, εκτός από τα imports που δίνονται στην αρχή του κεφαλαίου, χρειαζόμαστε και το παρακάτω:
>
> ```
> import doodle.random._
> ```

Στην Scala, υπάρχει μία ειδική σύνταξη που ονομάζεται *for comprehension*, η οποία μας επιτρέπει να γράψουμε απλούστερα τις εκφράσεις που περιέχουν μεθόδους όπως η `map` και η `flatMap`.

Για παράδειγμα, ο κώδικας της `randomConcentricCircles` καλεί την `map` και την `flatMap`.

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      randomCircle(size, randomPastel) flatMap { cir
cle =>
        randomConcentricCircles(n-1, size + 5) map {
 circles =>
```

```
          circle on circles
        }
      }
  }
```

Αυτό μπορεί να αντικατασταθεί από μία for comprehension.

```
def randomConcentricCircles(count: Int, size: Int):
Random[Image] =
  count match {
    case 0 => Random.always(Image.empty)
    case n =>
      for {
        circle  <- randomCircle(size, randomPastel)
        circles <- randomConcentricCircles(n-1, size
 + 5)
      } yield circle on circles
  }
```

Συνήθως, το for comprehension είναι πιο ευανάγνωστο από την `map` και την `flatMap`.

Γενικά, το for comprehension

```
for {
  x <- a
  y <- b
  z <- c
} yield e
```

μεταφράζεται σε:

```
a.flatMap(x => b.flatMap(y => c.map(z => e)))
```

Δηλαδή κάθε `<-`, εκτός από το τελευταίο, μετατρέπεται σε `flatMap`, ενώ το τελευταίο `<-` μετατρέπεται σε `map`.

Ο μεταγλωττιστής μεταφράζει το for comprehensions σε `flatMap` και `map`. Δεν συμβαίνει τίποτα μαγικό! Έχουμε απλά έναν διαφορετικό τρόπο

σύνταξης ο οποίος αποφεύγει βαθιές εμφωλευμένες εκφράσεις.

Σημειώστε ότι η σύνταξη του for comprehension είναι πιο ευέλικτη από όσα έχουμε παρουσιάσει. Για παράδειγμα, αν παραλείψετε το `yield`, ο κώδικα θα μεταγλωττιστεί. Δεν θα επιστρέψει όμως κάποιο αποτέλεσμα.

11.6 Ασκήσεις

11.6.1 Διασκορπισμένα Σχέδια

Σ' αυτή την άσκηση θα φτιάξουμε σχέδια διασκορπισμένα στον χώρο, όπως αυτά στην εικόνα fig. 58. Πειραματιστείτε με διάφορες κατανομές (προσπαθήστε να δημιουργήσετε τις δικές σας κατανομές μετατρέποντας αυτές που έχουμε γράψει).

Όταν δημιουργούμε ένα τέτοιο σχέδιο:

- πρέπει να δημιουργήσουμε τα σημεία που θα σχεδιάσουμε,
- να τοποθετήσουμε τις εικόνες την μία πάνω στην άλλη στο ίδιο σύστημα συντεταγμένων ώστε να δημιουργήσουμε το σχέδιο και
- να μετασχηματίσουμε ένα σημείο σε εικόνα, έτσι ώστε να την εμφανίσουμε.

Θα αναλύσουμε το κάθε ένα από τα παραπάνω σημεία ξεχωριστά.

Ξεκινήστε γράφοντας μία μέθοδο με όνομα `makePoint` που δέχεται μία `Random[Double]` για τις συντεταγμένες x και y ενός σημείου και επιστρέφει ένα `Random[Point]`. Θα πρέπει να έχει τον ακόλουθο σκελετό:

```
def makePoint(x: Random[Double], y: Random[Double]):
 Random[Point] =
  ???
```

Χρησιμοποιήστε έναν βρόγχο for.

See the solution

Τώρα δημιουργείστε χίλια τυχαία σημεία χρησιμοποιώντας τις τεχνικές που μάθαμε στο προηγούμενο κεφάλαιο για τις λίστες και επιλέξτε μία κατανομή που προτιμάτε. Θα πρέπει να καταλήξετε με μία `List[Random[Point]]`.

See the solution

Ας μετατρέψουμε την `List[Random[Point]]` σε `List[Random[Image]]`. Κάντε το σε δύο βήματα: πρώτα γράψτε μία μέθοδο που μετατρέπει ένα σημείο σε εικόνα και στην συνέχεια μία άλλη που μετατρέπει την `List[Random[Point]]` σε `List[Random[Image]]`.

See the solution

Τώρα δημιουργήστε μία μέθοδο μετατροπής της `List[Random[Image]]` σε `Random[Image]` τοποθετώντας όλα τα σημεία το ένα πάνω στο άλλο χρησιμοποιώντας την μέθοδο `on`. Είναι ισοδύναμο με την μέθοδο `allOn` που φτιάξαμε προηγουμένως αλλά τώρα λειτουργεί με δεδομένα μέσα στην `Random`.

See the solution

Τώρα βάλτε τα όλα μαζί για να δημιουργηθεί το τελικό σχέδιο με τα διασκορπισμένα σημεία.

See the solution

11.6.2 Παραμετρικός Θόρυβος

Σ' αυτή την άσκηση θα συνδυάσουμε τις παραμετρικές εξισώσεις που είδαμε σε προηγούμενα κεφάλαια με την τυχαιότητα. Ας ξεκινήσουμε φτιάχνοντας μία μέθοδο με όνομα `perturb` που προσθέτει τυχαίο θόρυβο σ' ένα σημείο. Η μέθοδος αυτή θα πρέπει να έχει τον παρακάτω σκελετό

```
def perturb(point: Point): Random[Point] =
  ???
```

Επιλέξτε όποια συνάρτηση επιθυμείτε.

See the solution

Τώρα φτιάξτε μία παραμετρική συνάρτηση. Μπορείτε να χρησιμοποιήσετε την συνάρτηση που γράψαμε για τα τριαντάφυλλα (που είδαμε σε προηγούμενο κεφάλαιο) ή να φτιάξετε μία δική σας από την αρχή. Δείτε τον ορισμό της rose παρακάτω.

```
def rose(k: Int): Angle => Point =
```

```
  (angle: Angle) => {
    Point.cartesian((angle * k).cos * angle.cos, (an
gle * k).cos * angle.sin)
  }
```

Συνδυάστε την παραμετρική συνάρτηση και την `perturb` για να δημιουργήσετε μία μέθοδο με τύπο `Angle => Random[Point]`. Γράψτε την χρησιμοποιώντας την μέθοδο `andThen`. Παρακάτω δίνεται ένα παράδειγμα της `andThen` στο οποίο φαίνεται πώς μπορεί να υπολογιστεί η τέταρτη δύναμη ενός αριθμού χρησιμοποιώντας το τετράγωνό του.

```
val square = (x: Double) => x * x
val quartic = square andThen square
```

See the solution

Τώρα, χρησιμοποιώντας την `allOn` φτιάξτε μία εικόνα που να συνδυάζει τυχαιότητα και δομή. Προσθέστε χρώμα, διαφάνειες ή οτιδήποτε μπορείτε να φανταστείτε.

See the solution

---

1. Οι ψευδο-τυχαίοι αριθμοί δεν είναι πραγματικά τυχαίοι. Το αποτέλεσμα καθορίζεται από μία τιμή που ονομάζεται *seed*. Αν ξέρουμε την τιμή της seed μπορούμε να προβλέψουμε τα αποτελέσματα που θα πάρουμε από την κλήση της `math.random` με απόλυτη ακρίβεια. Όμως, αν το προσπαθήσουμε από την αντίθετη πλευρά—δηλαδή να προβλέψουμε το seed έχοντας μία σειρά αποτελεσμάτων—είναι πολύ δύσκολο. Οι αριθμοί που παράγονται μ' αυτόν τον τρόπο καλούνται *ψευδο-τυχαίοι*, επειδή δεν είναι πραγματικά τυχαίοι αλλά παρόλα αυτά, είναι πολύ δύσκολο να προβλεφθούν. ↩

12 Δικοί μας Αλγεβρικοί Τύποι Δεδομένων

Σ' αυτό το κεφάλαιο θα μάθουμε πώς να δημιουργούμε τους δικούς μας αλγεβρικούς τύπους δεδομένων και πώς να τους χρησιμοποιούμε μαζί τα εργαλεία που γνωρίζουμε.

Μέχρι τώρα στην Creative Scala χρησιμοποιούσαμε τους (αλγεβρικούς) τύπους δεδομένων που μας παρείχε η Scala ή το Doodle, όπως για παράδειγμα οι τύποι `List` και `Point`. Σ' αυτή την ενότητα θα μάθουμε πώς να δημιουργούμε τους δικούς μας αλγεβρικούς τύπους δεδομένων, ανοίγοντας έτσι καινούριες δυνατότητες σε σχέση με τα προγράμματα που μπορούμε να γράψουμε.

> Τα προγράμματά σας θα δουλέψουν αν τα εκτελείτε από την κονσόλα SBT που υπάρχει μέσα στο Doodle. Αν όχι, θα πρέπει να ξεκινήσετε τον κώδικά σας με τα παρακάτω imports ώστε να κάνετε το Doodle διαθέσιμο.
>
> ```
> import doodle.core._
> import doodle.core.Image._
> import doodle.syntax._
> import doodle.jvm.Java2DFrame._
> import doodle.backend.StandardInterpreter._
> ```

12.1 Αλγεβρικοί Τύποι Δεδομένων

Στην Creative Scala έχουμε χρησιμοποιήσει πολλές φορές αλγεβρικούς τύπους δεδομένων αλλά δεν τους έχουμε περιγράψει ποτέ επίσημα. Σ' αυτό το σημείο όμως είναι χρήσιμη λίγη αυστηρότητα.

Ένας αλγεβρικός τύπος, χτίζεται από δύο συστατικά: - τα *λογικά ή* και - τα *λογικά και*.

Ο τύπος δεδομένων `List` είναι ένα πολύ καλό παράδειγμα αλγεβρικού

τύπου, αφού χρησιμοποιεί και τα δύο παραπάνω στοιχεία. Μία λίστα είναι `Nil` *ή* είναι ένα ζεύγος (λογικό ή). Ένα ζεύγος αποτελείται από μία κεφαλή *και* μία ουρά (λογικό και). Ένα σημείο `Point` είναι ένα ακόμη παράδειγμα. Είναι είτε καρτεσιανό είτε πολικό. Ένα καρτεσιανό σημείο έχει μία συντεταγμένη x και μία y, ενώ ένα πολικό έχει μία ακτίνα και μία γωνία. Σημειώστε ότι δεν είναι απαραίτητη η χρήση και των δύο μορφών ώστε ο τύπος δεδομένων να είναι αλγεβρικός.

Αφού είμαστε συναρτησιακοί προγραμματιστές, διαθέτουμε όπως ήταν αναμενόμενο μερικές πιο επίστημονικές λέξεις για τους τύπους του λογικού “ή” και του λογικού “και”.

Μπορείτε να τους δείτε παρακάτω: - έναν *τύπο αθροίσματος* για το λογικό “ή” - έναν *τύπο γινομένου* για το λογικό “και”.

Η έννοια του αλγεβρικού τύπου δεδομένων δεν είναι πολύ συγκεκριμένη στην Scala. Ας εξασκηθούμε λίγο.

Ασκήσεις

Στοιχεία Μονοπατιών

Ο τύπος `PathElement` χρησιμοποιείται για την κατασκευή μονοπατιών και είναι ένας απλός αλγεβρικός τύπος δεδομένων. Έχουμε ήδη χρησιμοποιήσει τον `PathElement` αρκετά. Πώς πιστεύετε ότι ορίζεται ο `PathElement` χρησιμοποιώντας τους τύπους του αθροίσματος και του γινομένου;

See the solution

Εντελώς Turtles

Ο τύπος `Instruction` που χρησιμοποιήσαμε για τον έλεγχο του turtle είναι επίσης ένας αλγεβρικός τύπος δεδομένων. Πώς πιστεύετε ότι ορίζεται ο `Instruction`;

See the solution

12.1.1 Ορίζοντας Αλγεβρικούς Τύπους Δεδομένων

Τώρα που καταλάβαμε πώς μπορούμε να μοντελοποιήσουμε δεδομένα

με αλγεβρικούς τύπους δεδομένων, ας δούμε και πώς μπορούμε να ορίσουμε τους δικούς μας.

Η μορφή είναι η παρακάτω:

- Αν ο `A` είναι `B` ή `C` τότε γράψτε το παρακάτω

```
sealed abstract class A extends Product with Seriali
zable
// defined class A

final case class B() extends A
// defined class B

final case class C() extends A
// defined class C
```

Παραπάνω υπάρχουν πολλοί όροι που δεν έχουμε δει αλλά μπορούμε να τους αγνοήσουμε και να αποδεχτούμε ότι πρέπει να τους γράφουμε μαζί με τον υπόλοιπο κώδικα. Όμως αν ενδιαφέρεστε για το τι σημαίνουν (και πολύ πιθανόν να έχετε προηγούμενη εμπειρία με τον αντικειμενοστραφή προγραμματισμό), τότε εξερευνήστε τους.

Για να ορίσουμε τον τύπο `PathElement` θα μπορούσαμε να ξεκινήσουμε με το παρακάτω

```
sealed abstract class PathElement extends Product wi
th Serializable
// defined class PathElement
// warning: previously defined object PathElement is
 not a companion to class PathElement.
// Companions must be defined together; you may wish
 to use :paste mode for this.

final case class MoveTo() extends PathElement
// defined class MoveTo

final case class LineTo() extends PathElement
// defined class LineTo
```

```
final case class CurveTo() extends PathElement
// defined class CurveTo
```

Το άλλο μισό της μορφής είναι το ακόλουθο

- Αν ο `A` έχει έναν `B` και έναν `C`, τότε γράψτε

```
final case class A(b: B, c: C)
```

Περιγράψτε τις παραμέτρους του constructor εδώ.

Επιστρέφοντας στις `PathElement`, `MoveTo` και `LineTo`, όλες τους έχουν ένα σημείο (τον προορισμό) και η `CurveTo` έχει ένα σημείο προορισμού και δύο σημεία ελέγχου. Οπότε μπορούμε να γράψουμε το παρακάτω.

```
sealed abstract class PathElement extends Product wi
th Serializable
final case class MoveTo(to: Point) extends PathEleme
nt
final case class LineTo(to: Point) extends PathEleme
nt
final case class CurveTo(cp1: Point, cp2: Point, to:
 Point) extends PathElement
```

Αυτός είναι ο τρόπος ορισμού του `PathElement` στο Doodle.

Άσκηση

Ορίστε τον δικό σας αλγεβρικό τύπο δεδομένων για αναπαράσταση του `Instruction`.

See the solution

12.2 Χτίστε το Δικό σας Turtle

Παρακάτω μπορείτε να δείτε τον τύπο `Instruction` που ορίσαμε στην προηγούμενη ενότητα.

```
sealed abstract class Instruction extends Product wi
```

```
th Serializable
// defined class Instruction

final case class Forward(distance: Double) extends I
nstruction
// defined class Forward

final case class Turn(angle: Angle) extends Instruct
ion
// defined class Turn

final case class Branch(instructions: List[Instructi
on]) extends Instruction
// defined class Branch

final case class NoOp() extends Instruction
// defined class NoOp
```

Τώρα που ορίσαμε τον δικό μας τύπο `Instruction`, ας προχωρήσουμε ένα βήμα παρακάτω και ας δημιουργήσουμε το δικό μας `Turtle`. Για να ολοκληρώσουμε το turtle μας, πρέπει να φτιάξουμε μία `draw`. Μπορούμε να ξεκινήσουμε έτσι:

```
object Turtle {
  def draw(instructions: List[Instruction]): Image =
    ???
}
// defined object Turtle
```

Ο `Instruction` είναι ένας αλγεβρικός τύπος δεδομένων, άρα ξέρουμε ότι μπορούμε να τον επεξεργαστούμε χρησιμοποιώντας δομημένη αναδρομή. Όμως, για να το κάνουμε αυτό, πρέπει να αποθηκεύσουμε την τωρινή κατάσταση του turtle: θα χρειαστούμε την τοποθεσία του (ένα `Vec`) και την κατεύθυνση (μία `Angle`). Φτιάξτε έναν τύπο για την αποθήκευση αυτών των δεδομένων.

See the solution

Όταν επεξεργαστούμε τις εντολές, θα τις μετατρέψουμε σε `List[PathElement]` και αργότερα θα τις βάλουμε μέσα σ' ένα ανοιχτό μονοπάτι για να δημιουργήσουμε μία εικόνα. Η μετατροπή για κάθε εντολή, θα είναι μία συνάρτηση της τωρινής κατάστασης του turtle και της ίδιας της εντολής και θα επιστρέφει την ενημερωμένη κατάσταση και μία `List[PathElement]`.

Φτιάξτε μία μέθοδο με όνομα `process`, η οποία θα υλοποιεί την παραπάνω περιγραφή και η δήλωσή της θα είναι η παρακάτω

```
def process(state: TurtleState, instruction: Instruc
tion): (TurtleState, List[PathElement]) =
  ???
// process: (state: TurtleState, instruction: Instru
ction)(TurtleState, List[doodle.core.PathElement])
```

Υλοποιήστε το πρώτα χωρίς εντολές διακλάδωσης. Θα επιστρέψουμε σ' αυτό αργότερα.

See the solution

Χρησιμοποιώντας την `process`, γράψτε μία δομημένη αναδρομή στην `List[Instruction]` για την μετατροπή των εντολών σε `List[PathElement]`. Ονομάστε αυτή την μέθοδο `iterate` και ξεκινήστε την όπως παρακάτω

```
def iterate(state: TurtleState, instructions: List[I
nstruction]): List[PathElement] =
  ???
// iterate: (state: TurtleState, instructions: List[
Instruction])List[doodle.core.PathElement]
```

See the solution

Μπορείτε να προσθέσετε στην `process` διακλαδώσεις, χρησιμοποιώντας την `iterate`.

```
def process(state: TurtleState, instruction: Instruc
tion): (TurtleState, List[PathElement]) = {
  import PathElement._
```

```
  instruction match {
    case Forward(d) =>
      val nowAt = state.at + Vec.polar(d, state.head
ing)
      val element = lineTo(nowAt.toPoint)

      (state.copy(at = nowAt), List(element))
    case Turn(a) =>
      val nowHeading = state.heading + a

      (state.copy(heading = nowHeading), List())
    case Branch(is) =>
     val branchedElements = iterate(state, is)

     (state, moveTo(state.at.toPoint) :: branchedEle
ments)
    case NoOp() =>
      (state, List())
  }
}
// process: (state: TurtleState, instruction: Instru
ction)(TurtleState, List[doodle.core.PathElement])
```

Τώρα υλοποιήστε την `draw` χρησιμοποιώντας την `iterate`.

See the solution

12.2.1 Επεκτάσεις

Τα turtles που μπορούν να κάνουν τυχαίες επιλογές μπορούν να οδηγήσουν σε πιο πρωτότυπες εικόνες. Μπορείτε να φτιάξετε κάτι τέτοιο;

13 Σύνοψη

Σ' αυτό το βιβλίο καλύψαμε αρκετά από τα εργαλεία που χρησιμοποιούνται στον συναρτησιακό προγραμματισμό και είναι διαθέσιμα στην Scala.

13.1 Αναπαραστάσεις και Μεταφραστές

Ξεκινήσαμε γράφοντας εκφράσεις για να δημιουργήσουμε εικόνες. Τα προγράμματα που γράψαμε πέρασαν από δύο διακριτές φάσεις:

1. Σχεδίασαν μία εικόνα
2. Κάλεσαν την μέθοδο `draw` για να την εμφανίσουν

Μέσα από αυτή τη διαδικασία παρατηρούμε δύο πολύ σημαντικά στοιχεία του συναρτησιακού προγραμματισμού: την *κατασκευή ενδιάμεσων αναπαραστάσεων* του επιθυμητού αποτελέσματος και την *μετάφραση αυτών των αναπαραστάσεων* ώστε να παράγουμε αποτέλεσμα.

13.2 Αφαιρετικότητα

Η κατασκευή μίας ενδιάμεσης αναπαράστασης μας επιτρέπει να μοντελοποιήσουμε πλευρές του αποτελέσματος που πιστεύουμε ότι είναι σημαντικές και να χρησιμοποιήσουμε την έννοια της *αφαιρετικότητας* ώστε να απομακρύνουμε άσχετες λεπτομέρειες.

Για παράδειγμα, το Doodle αναπαριστά απευθείας τα βασικά σχήματα και τις γεωμετρικές σχέσεις των σχεδίων μας και έτσι δεν χρειάζεται να ανησυχούμε για τις λεπτομέρειες της υλοποίησης, όπως τις συντεταγμένες στην οθόνη. Έτσι, ο κώδικάς μας παραμένει καθαρός, η τροποποίηση του εύκολη και περιορίζονται οι "μαγικοί αριθμοί" που πρέπει να γράψουμε. Για παράδειγμα, είναι πιο εύκολο να καταλάβουμε ότι ο παρακάτω κώδικας στο Doodle θα παράγει ένα σπίτι:

```
def myImage: Image =
  Triangle(50, 50) above Rectangle(50, 50)
// myImage: Image = // ...
```

από ότι αυτός ο κώδικας σε Java2D:

```
def drawImage(g: Graphics2D): Unit = {
  g.setStroke(new BasicStroke(1.0f))
  g.setPaint(new Color(0, 0, 0))
  val path = new Path2D.Double()
  path.moveTo(25, 0)
  path.lineTo(50, 50)
  path.lineTo(0, 50)
  path.lineTo(25, 0)
  path.closePath()
  g.draw(path)
  f.drawRect(50, 50, 50, 50)
}
```

Είναι σημαντικό να κατανοήσετε ότι ο βασικός σε Java2D βρίσκεται μέσα στο Doodle. Η διαφορά είναι ότι τον έχουμε κρύψει μέσα στην μέθοδο `draw`. Η `draw` έχει τον ρόλο του *μεταφραστή* για τις εικόνες μας, συμπληρώνοντας όλες τις λεπτομέρειες σχετικά με τις συντεταγμένες, τα μονοπάτια και τα γραφικά που δεν θέλουμε να σκεφτόμαστε καθώς γράφουμε τον κώδικά μας.

Ξεχωρίζοντας την τιμή από τον μεταφραστή, έχουμε την δυνατότητα να αλλάξουμε τον τρόπο που γίνεται η μετάφραση. Το Doodle έχει ήδη δύο μεταφραστές. Ο ένας από αυτούς σχεδιάζει στην δομή της Java2D ενώ ο άλλος στον καμβά της HTML. Θα μπορούσε να φανταστεί κανείς μεταφραστές και για άλλες χρήσεις, όπως για παράδειγμα έναν για σχεδιασμό εικόνων που φαίνονται σαν ζωγραφισμένες με το χέρι.

13.3 Σύνθεση

Εκτός από το να κρατάμε τα προγράμματά μας καθαρά, η συναρτησιακή προσέγγιση που έχει προτιμήσει το Doodle, μας επιτρέπει να *συνθέσουμε* εικόνες χρησιμοποιώντας προϋπάρχουσες. Για παράδειγμα, μπορούμε να επαναχρησιμοποιήσουμε τον κώδικα για το σπίτι ώστε να ζωγραφίσουμε έναν δρόμο:

```
val house = Triangle(50, 50) above Rectangle(50, 50)
```

```
// house: Image = // ...

val street = house beside house beside house
// street: Image = // ...
```

Οι τιμές που δημιουργούμε για τις εικόνες και τα χρώματα είναι αμετάβλητες. Μπορούμε εύκολα να χρησιμοποιήσουμε ένα σπίτι τρεις φορές μέσα στην ίδια εικόνα.

Αυτή η προσέγγιση, μας επιτρέπει να χωρίσουμε μία σύνθετη εικόνα σε μικρότερα κομμάτια που μπορούμε να συνδυάσουμε μεταξύ τους, ώστε να δημιουργήσουμε το επιθυμητό αποτέλεσμα.

Η επαναχρησιμοποίηση αμετάβλητων δεδομένων, τεχνική που ονομάζεται *structure sharing*, είναι η βάση πολλών γρήγορων και αποτελεσματικών, σε σχέση με την μνήμη, αμετάβλητων δομών δεδομένων. Ένα πολύ αντιπροσωπευτικό παράδειγμα είναι το τρίγωνο Sierpinski στο οποίο χρησιμοποιήσαμε μόνο ένα αντικείμενο `Triangle` για να δημιουργήσουμε μία εικόνα η οποία περιλαμβάνει περίπου 20,000 χρωματιστά τρίγωνα.

13.4 Προγραμματισμός Προσανατολισμένος σε Εκφράσεις

Η Scala μας παρέχει ένα πολύ βολικό συντακτικό για απλοποίηση της δημιουργίας δομών δεδομένων με συναρτησιακό τρόπο. Κατασκευές όπως οι δομές υπόθεσης (πχ if), οι βρόγχοι και τα blocks, είναι *εκφράσεις*, που μας επιτρέπουν να γράφουμε μικρές μεθόδους, χωρίς να πρέπει να δηλώσουμε πολλές ενδιάμεσες μεταβλητές. Έτσι υιοθετούμε την συνήθεια να γράφουμε μικρές μεθόδους των οποίων ο σκοπός είναι να επιστρέψουν μία τιμή.

13.5 Τύποι και Δίχτυ Ασφαλείας

Το σύστημα τύπων της Scala μας βοηθάει να ελέγξουμε το προγράμμά μας. Κάθε έκφραση έχει έναν τύπο ο οποίος ελέγχεται κατά την μεταγλώττιση, ώστε να γίνει σίγουρο ότι ταιριάζει με τον υπόλοιπο

κώδικα. Μπορούμε ακόμη και να ορίσουμε τους δικούς μας τύπους, με μοναδικό σκοπό να προστατέψουμε τον εαυτό μας από λάθη που μπορεί να κάνουμε.

Ένα πολύ απλό παράδειγμα είναι ο τύπος `Angle` του Doodle, ο οποίος μας αποτρέπει από το να μπερδέψουμε τους αριθμούς, τις γωνίες, τις μοίρες και τα ακτίνια:

```
90
// res0: Int = 90

90.degrees
// res1: doodle.core.Angle = Angle(1.5707963267948966)

90.radians
// res2: doodle.core.Angle = Angle(2.0354056994857643)

90.degrees + 90.radians
// res3: doodle.core.Angle = Angle(3.606202026280661)

90 + 90.degrees
// <console>:20: error: overloaded method value + with alternatives:
//   (x: Double)Double <and>
//   (x: Float)Float <and>
//   (x: Long)Long <and>
//   (x: Int)Int <and>
//   (x: Char)Int <and>
//   (x: Short)Int <and>
//   (x: Byte)Int <and>
//   (x: String)String
// δεν μπορεί να εφαρμοστεί στο (doodle.core.Angle)
//              90 + 90.degrees
//                   ^
```

13.6 Οι Συναρτήσεις ως Τιμές

Καθώς προγραμματίζουμε, ξοδεύουμε πολύ χρόνο στο να γράφουμε μεθόδους που παράγουν τιμές. Οι μέθοδοι μας επιτρέπουν να χρησιμοποιήσουμε την έννοια της αφαιρετικότητας, όσον αφορά τις παραμέτρους. Για παράδειγμα, η παρακάτω μέθοδος χρησιμοποιεί αυτήν την έννοια για να σχεδιάσει κουκκίδες διαφορετικού χρώματος:

```
def dot(color: Color): Image =
  Circle(10) lineWidth 0 fillColor color
// dot: Color => Image = // ...
```

Προερχόμενες από αντικειμενοστραφείς γλώσσες, οι μέθοδοι δεν είναι κάτι ιδιαίτερο. Η ικανότητα της Scala να μετατρέπει μεθόδους σε *συναρτήσεις* που μπορούν να περαστούν ως τιμές, είναι πολύ πιο ενδιαφέρουσα. Δείτε παρακάτω:

```
def spectrum(shape: Color => Image): Image =
  shape(Color.red) beside shape(Color.blue) beside s
hape(Color.green)
// spectrum: (Color => Image) => Image = // ...

spectrum(dot)
// res0: Image = // ...
```

Γράψαμε πολλά προγράμματα που χρησιμοποιούν συναρτήσεις ως τιμές αλλά το πιο αντιπροσωπευτικό παράδειγμα ήταν η μέθοδος `map` στις λίστες. Στο κεφάλαιο "Συλλογές" είδαμε πώς η `map` μας επιτρέπει να μετασχηματίσουμε τις ακολουθίες χωρίς να μεταφέρουμε ή να "εισάγουμε" νέες τιμές σε ενδιάμεσους buffers (μνήμες):

```
List(1, 2, 3).map(x => x * 2)
// res0: List[Int] = List(2, 4, 6)
```

Οι συναρτήσεις και ο ορισμούς τους ως τιμές πρώτης τάξης, είναι πολύ σημαντικά στοιχεία για την γραφή απλού και κατανοητού κώδικα.

13.7 Επίλογος

Ο σκοπός αυτού του βιβλίου ήταν να σας συστήσει τα κομμάτια της Scala που έχουν να κάνουν με τον συναρτησιακό προγραμματισμό. Αυτά είναι που διαφοροποιούν την Scala από παλαιότερες εμπορικές γλώσσες όπως η Java και η C. Όμως αυτό είναι μόνο ένα μέρος της ιστορίας της Scala. Πολλές σύγχρονες γλώσσες όπως η Ruby, η Python, η Javascript, και η Clojure υποστηρίζουν τον συναρτησιακό προγραμματισμό. Πώς σχετίζεται η Scala μ' αυτές τις γλώσσες και γιατί να την προτιμήσετε έναντι των άλλων;

Ίσως το πιο σημαντικό πλεονέκτημα της Scala είναι το αυστηρό σύστημα τύπων που διαθέτει. Αυτό διαφοροποιεί την Scala από άλλες δημοφιλείς γλώσσες όπως η Ruby, η Python, η Javascript και η Clojure, οι οποίες διαθέτουν δυναμικό σύστημα τύπων. Η χρήση στατικών τύπων είναι ένας αναγκαίος συμβιβασμός—η σύνταξη του κώδικα είναι πιο αργή αφού πρέπει να ικανοποιεί τον μεταγλωττιστή σε όλα τα στάδια. Όμως, αφού μεταγλωττιστεί ο κώδικάς μας μπορούμε να είμαστε βέβαιοι για την ποιότητά του.

Άλλο ένα μεγάλο πλεονέκτημα της Scala είναι η ανάμειξη των τεχνικών αντικειμενοστραφούς και συναρτησιακού προγραμματισμού. Αυτό το είδαμε στο πρώτο κεφάλαιο—κάθε τιμή είναι ένα αντικείμενο με μεθόδους, πεδία και μία κλάση (τύπο). Παρόλα αυτά, σ' αυτό το βιβλίο δεν δημιουργήσαμε δικούς μας τύπους δεδομένων. Η δημιουργία τύπων είναι συνώνυμο της δήλωσης κλάσεων και η Scala υποστηρίζει μία μεγάλη ποικιλία τέτοιων χαρακτηριστικών όπως για παράδειγμα, τις κλάσεις, τα traits, την κληρονομικότητα και τα generics.

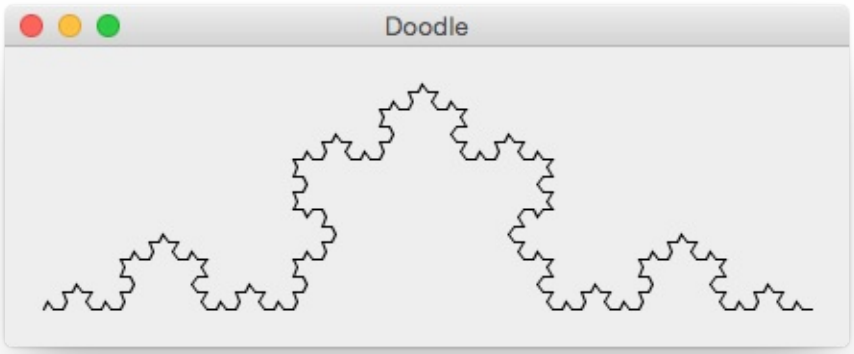
Τέλος, μία σημαντική ικανότητα της Scala είναι η συμβατότητά της με την Java. Η Scala θα μπορούσε να θεωρηθεί υπερσύνολο της Java και η σχέση τους είναι αρκετά εμφανής. Έτσι ανοίγεται ένας ολόκληρος κόσμος από βιβλιοθήκες της Java που μπορούν να χρησιμοποιηθούν από εφαρμογές της Scala. Η μετάφραση εφαρμογών Java σε Scala είναι πολύ εύκολη.

13.8 Επόμενα Βήματα

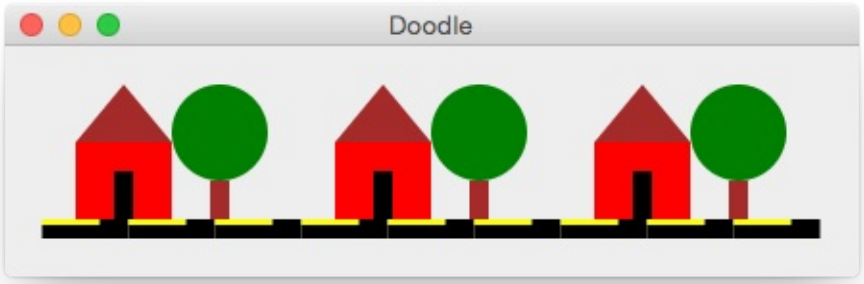
Ελπίζουμε να σας άρεσε η Creative Scala και ο σχεδιασμός εικόνων με το Doodle. Αν θέλετε να μάθετε περισσότερα για αυτή, θα προτείναμε να επιλέξετε ένα από τα υπέροχα βιβλία που είναι διαθέσιμα γι' αυτή τη γλώσσα.

Το δικό μας βιβλίο, Essential Scala, είναι διαθέσιμο στην ιστοσελίδα μας και συνεχίζει την προσέγγιση της Creative Scala σε σχέση με τον τρόπο εκμάθησης Scala, συζητώντας και παρουσιάζοντας εικόνες.

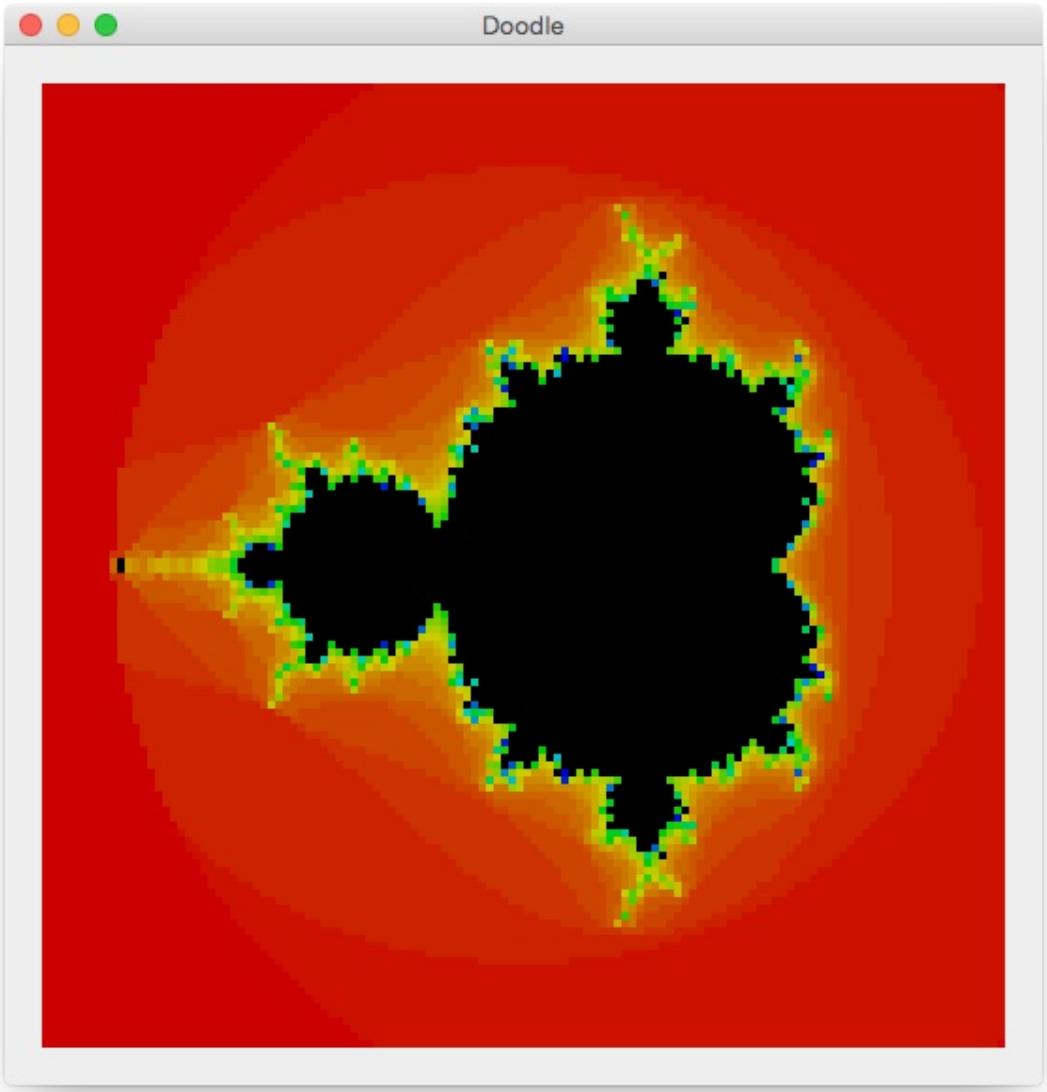
Αν θέλετε να δυσκολέψετε τον εαυτό σας, προσπαθήστε να ζωγραφίσετε κάτι πιο περίπλοκο με το Doodle και μοιραστείτε το μαζί μας μέσω του Gitter. Υπάρχουν πολλά που μπορείτε να δοκιμάσετε—δείτε τον κατάλογο `examples` στο Doodle για προτάσεις:



Τρίγωνο Koch (Koch.scala)



Αστικό τοπίο (Street.scala)



Mandelbrot Fractal του Mat Moore (Mandelbrot.scala)

14 Γρήγορη Αναφορά στο Συντακτικό

14.1 Κυριολεκτικές Εκφράσεις

```
// Κυριολεκτικές Εκφράσεις:
123      // Int
123.0    // Double
"Hello!" // String
true     // Boolean

// Μαθηματικά:
10 + 2   // Int + Int    = Int
10 + 2.0 // Int + Double = Double
10 / 2   // Int / Int    = Double

// Πράξεις Boolean:
true && false // logical AND
true || false // logical OR
!true         // logical NOT

// Σύνδεση String:
"abc" + "def" // String
"abc" + 123   // αυτόματη μετατροπή από Int σε Strin
g

// Κλήσεις μεθόδων και infix operators:
1.+(2)    // κλήση μεθόδου
1 + 2     // infix operator
1 + 2 + 3 // ισοδύναμο με το 1.+(2).+(3)

// Υποθετικές εκφράσεις:
if(booleanExpression) expressionA else expressionB
```

```
// Blocks:
{
  sideEffectExpression1
  sideEffectExpression2
  resultExpression
}
```

14.2 Δηλώσεις Τιμών και Μεθόδων

```
// Συντακτικό δήλωσης τιμών:
val valueName: SomeType = resultExpression // δήλωση
 με συγκεκριμένο τύπο
val valueName = resultExpression           // δήλωση
 της οποίας ο τύπος υπονοείται

// Μέθοδος με λίστα παραμέτρων και με συγκεκριμένο τ
ύπο αποτελέσματος:
def methodName(argName: ArgType, argName: ArgType):
ReturnType =
  resultExpression

// Μέθοδος με λίστα παραμέτρων με μη συγκεκριμένο τύ
πο αποτελέσματος:
def methodName(argName: ArgType, argName: ArgType) =
  resultExpression

// Μέθοδος με πολλές εκφράσεις (με χρήση block):
def methodName(argName: ArgType, argName: ArgType):
ReturnType = {
  sideEffectExpression1
  sideEffectExpression2
  resultExpression
}

// Μέθοδος χωρίς λίστα παραμέτρων:
def methodName: ReturnType =
```

```
  resultExpression

// Κλήση μεθόδου με λίστα παραμέτρων:
methodName(arg, arg)

// Κλήση μεθόδου χωρίς λίστα παραμέτρων:
methodName
```

14.3 Συναρτήσεις και Τιμές

Οι συναρτήσεις ως τιμές, γράφονται έτσι `(argName: ArgType, ...) => resultExpression`:

```
val double = (num: Int) => num * 2
// double: Int => Int = <function1>

val sum = (a: Int, b: Int) => a + b
sum: (Int, Int) => Int = <function2>
```

Οι συναρτήσεις με πολλές γραμμές κώδικα γράφονται μέσα σε block:

```
val printAndDouble = (num: Int) => {
  println("The number was " + num)
  num * 2
}
// printAndDouble: Int => Int = <function1>

scala> printAndDouble(10)
// Ο αριθμός ήταν 10
// res0: Int = 20
```

Πρέπει να γράφουμε τους τύπους των συναρτήσεων των παραμέτρων και των αποτελεσμάτων. Η σύνταξη είναι αυτή: `ArgType => ResultType` ή `(ArgType, ...) => ResultType`:

```
def doTwice(value: Int, func: Int => Int): Int =
  func(func(value))
```

```
// doTwice: (value: Int, func: Int => Int)Int

doTwice(1, double)
// res0: Int = 4
```

Οι συναρτήσεις ως τιμές μπορούν να γραφούν όπως ακριβώς και οι κανονικές εκφράσεις:

```
doTwice(1, (num: Int) => num * 10)
// res1: Int = 100
```

Μερικές φορές μπορούμε να παραλείψουμε τους τύπους των παραμέτρων, υποθέτοντας ότι θα τους βρει ο μεταγλωττιστής:

```
doTwice(1, num => num * 10)
// res2: Int = 100
```

14.4 Οδηγός Αναφοράς για το Doodle

14.4.1 Imports

```
// Μ' αυτά τα imports μπορείτε να κάνετε τα πάντα:
import doodle.core._
import doodle.syntax._
```

14.4.2 Δημιουργώντας Εικόνες

```
// Βασικές εικόνες (μαύρο περίγραμμα, χωρίς γέμισμα)
:
val i: Image = Circle(radius)
val i: Image = Rectangle(width, height)
val i: Image = Triangle(width, height)

// Σύνθετες εικόνες χρησιμοποιώντας σύνταξη με opera
tors:
```

```
val i: Image = imageA beside imageB // οριζόντια διάταξη
val i: Image = imageA above  imageB // κάθετη διάταξη
val i: Image = imageA below  imageB // κάθετη διάταξη
val i: Image = imageA on     imageB // η μία πάνω από την άλλη
val i: Image = imageA under  imageB // η μία κάτω από την άλλη

// Σύνθετες εικόνες χρησιμοποιώντας σύνταξη με κλήσεις μεθόδων:
val i: Image = imageA.beside(imageB)
// κλπ...
```

14.4.3 Μορφοποίηση Εικόνων

```
// Μορφοποίηση εικόνων με operators:
val i: Image = image fillColor color   // νέο χρώμα γεμίσματος (δεν γίνεται αλλαγή γραμμής)
val i: Image = image lineColor color   // νέο χρώμα γραμμής (δεν αλλάζει το χρώμα γεμίσματος)
val i: Image = image lineWidth integer // νέο πλάτος γραμμής (δεν αλλάζει το χρώμα γεμίσματος)
val i: Image = image fillColor color lineColor otherColor // νέο χρώμα γέμισματος και γραμμής

// Μορφοποίηση εικόνων με κλήση μεθόδων:
val i: Image = imageA.fillColor(color)
val i: Image = imageA.fillColor(color).lineColor(otherColor)
// κλπ...
```

14.4.4 Χρώματα

```
// Βασικά Χρώματα:
val c: Color = Color.red                          // προ
οκαθορισμένα χρώματα
val c: Color = Color.rgb(255.uByte, 127.uByte, 0.uBy
te)          // χρώματα RGB
val c: Color = Color.rgba(255.uByte, 127.uByte, 0.uB
yte, 0.5.normalized)    // χρώματα RGBA
val c: Color = Color.hsl(15.degrees, 0.25.normalized
, 0.5.normalized)       // χρώμα HSL
val c: Color = Color.hsla(15.degrees, 0.25.normalize
d, 0.5.normalized, 0.5.normalized) // χρώμα HSLA

// Μετατροπή/μίξη χρωμάτων χρησιμοποιώντας operators
:
val c: Color = someColor spin       10.degrees     /
/ αλλαγή απόχρωσης
val c: Color = someColor lighten    0.1.normalized /
/ αλλαγή φωτεινότητας
val c: Color = someColor darken     0.1.normalized /
/ αλλαγή φωτεινότητας
val c: Color = someColor saturate   0.1.normalized /
/ αλλαγή κορεσμού
val c: Color = someColor desaturate 0.1.normalized /
/ αλλαγή κορεσμού
val c: Color = someColor fadeIn     0.1.normalized /
/ αλλαγή αδιαφάνειας
val c: Color = someColor fadeOut    0.1.normalized /
/ αλλαγή αδιαφάνειας

// Μετατροπή/μίξη χρωμάτων χρησιμοποιώντας κλήσεις μ
εθόδων:
val c: Color = someColor.spin(10.degrees)
val c: Color = someColor.lighten(0.1.normalized)
// κλπ...
```

14.4.5 Μονοπάτια

```
// Δημιουργία μονοπατιού από λίστα με PathElements:
val i: Image = OpenPath(List(
  MoveTo(Vec(0, 0).toPoint),
  LineTo(Vec(10, 10).toPoint)
))

// Δημιουργία μονοπατιού από ακολουθία PathElements:
val i: Image = OpenPath(
  (0 until 360 by 30) map { i =>
    LineTo(Vec.polar(i.degrees, 100).toPoint)
  }
)

// Τύποι στοιχείων:
val e1: PathElement = MoveTo(toVec.toPoint)
                // χωρίς γραμμή
val e2: PathElement = LineTo(toVec.toPoint)
                // ευθεία γραμμή
val e3: PathElement = BezierCurveTo(cp1Vec.toPoint,
cp2Vec.toPoint, toVec.toPoint) // καμπύλη

// ΣΗΜΕΙΩΣΗ: Αν το πρώτο στοιχείο δεν είναι το MoveT
o,
//       τότε μετατρέπεται σε τέτοιο
```

14.4.6 Γωνίες και Διανύσματα

```
val a: Angle = 30.degrees                    // γωνία σε
 μοίρες
val a: Angle = 1.5.radians                   // γωνία σε
 ακτίνια
val a: Angle = math.Pi.radians               // ακτίνια
π
```

```
val a: Angle = 1.turns                          // γωνία σε πλήρη περιστροφή

val v: Vec = Vec.zero                           // μηδενικό διάνυσμα (0,0)
val v: Vec = Vec.unitX                          // διάνυσμα μονάδα για την x (1,0)
val v: Vec = Vec.unitY                          // διάνυσμα μονάδα για την y (0,1)

val v: Vec = Vec(3, 4)                          // διάνυσμα καρτεσιανών συντεταγμένων
val v: Vec = Vec.polar(30.degrees, 5)           // διάνυσμα πολικών συντεταγμένων
val v: Vec = Vec(2, 1) * 10                     // πολλαπλασιασμός μήκους
val v: Vec = Vec(20, 10) / 10                   // διαίρεση μήκους
val v: Vec = Vec(2, 1) + Vec(1, 3)              // πρόσθεση διανυσμάτων
val v: Vec = Vec(5, 5) - Vec(2, 1)              // αφαίρεση διανυσμάτων
val v: Vec = Vec(5, 5) rotate 45.degrees // περιστροφή αντίθετα από την φορά του ρολογιού

val x: Double = Vec(3, 4).x                     // συντεταγμένη x
val y: Double = Vec(3, 4).y                     // συντεταγμένη y
val a: Angle  = Vec(3, 4).angle                 // γωνία που σχηματίζεται με το διάνυσμα (1, 0)
val l: Double = Vec(3, 4).length                // μήκος
```

15 Λύσεις Ασκήσεων

15.1 Εκφράσεις, Τιμές και Τύποι

15.1.1 Solution to: Αριθμητική

Ο σκοπός αυτής της άσκησης είναι να συνηθίσετε να γράφετε κώδικα σε Scala. Παρακάτω δίνεται μία πιθανή λύση.

```
1 + 43 - 2
// res0: Int = 42
```

Return to the exercise

15.1.2 Solution to: Ενώνοντας Strings

Παρακάτω μπορείτε να δείτε μία πιθανή λύση.

```
"It is a truth ".++("universally acknowledged")
// res1: String = It is a truth universally acknowle
dged

"It is a truth " ++ "universally acknowledged"
// res2: String = It is a truth universally acknowle
dged
```

Return to the exercise

15.1.3 Solution to: Προτεραιότητα

Λίγη εξάσκηση με την κονσόλα θα πρέπει να σας πείσει ότι ναι, στη Scala ισχύουν οι ίδιοι κανόνες προτεραιότητας. Δείτε το επόμενο παράδειγμα.

```
1 + 2 * 3
// res3: Int = 7
```

```
1 + (2 * 3)
// res4: Int = 7

(1 + 2) * 3
// res5: Int = 9
```

Return to the exercise

15.1.4 Solution to: Τύποι και Τιμές

```
1 + 2
// res14: Int = 3
```

Ο τύπος της έκφρασης είναι `Int` και το αποτέλεσμα είναι `3`.

```
"3".toInt
// res15: Int = 3
```

Ο τύπος της έκφρασης είναι `Int` και το αποτέλεσμα είναι `3`.

```
"Electric blue".toInt
// java.lang.NumberFormatException: For input string
: "Electric blue"
//   at java.lang.NumberFormatException.forInputStri
ng(NumberFormatException.java:65)
//   at java.lang.Integer.parseInt(Integer.java:580)
//   at java.lang.Integer.parseInt(Integer.java:615)
//   at scala.collection.immutable.StringLike.toInt(
StringLike.scala:301)
//   at scala.collection.immutable.StringLike.toInt$
(StringLike.scala:301)
//   at scala.collection.immutable.StringOps.toInt(S
tringOps.scala:29)
//   ... 267 elided
```

Ο τύπος της έκφρασης είναι `Int` αλλά θα αποτύχει κατά τον χρόνο εκτέλεσης.

```
"Electric blue".take(1)
```

Ο τύπος της έκφρασης είναι `String` και το αποτέλεσμα είναι `"E"`.

```
"Electric blue".take("blue")
// <console>:13: error: type mismatch;
//  found   : String("blue")
//  required: Int
//        "Electric blue".take("blue")
//                             ^
```

Η έκφραση θα αποτύχει κατά τον χρόνο μεταγλώττισης άρα δεν έχει τύπο.

```
1 + ("Moonage daydream".indexOf("N"))
// res19: Int = 0
```

Ο τύπος της έκφρασης είναι `Int` και το αποτέλεσμα είναι `0`.

```
1 / 1 + ("Moonage daydream".indexOf("N"))
// res20: Int = 0
```

Ο τύπος της έκφρασης είναι `Int` και λόγω των κανόνων προτεραιότητας το αποτέλεσμα είναι `(1 / 1) + -1`, άρα `0`.

```
1 / (1 + ("Moonage daydream".indexOf("N")))
```

Ο τύπος της έκφρασης είναι `Int` αλλά και αυτή θα αποτύχει κατά τον χρόνο εκτέλεσης λόγω διαίρεσης με το μηδέν.

Return to the exercise

15.1.5 Solution to: Αστοχίες κινητής υποδιαστολής

Ο `Double` αποτελεί μία προσέγγιση καθώς πρέπει να χωρέσουν μέσα στην πεπερασμένη μνήμη του υπολογιστή. Ένας `Double` καταλαμβάνει ακριβώς 64-bits, που είναι αρκετός χώρος για να αποθηκευτούν πολλά

ψηφία αλλά όχι τόσος ώστε να αποθηκευτούν αριθμοί όπως το π, το οποίο έχει άπειρα δεκαδικά ψηφία.

Ο αριθμός ⅓ έχει επίσης άπειρα δεκαδικά ψηφία. Οι `Doubles` αποθηκεύονται στο δυαδικό. Υπάρχουν όμως και αριθμοί που μπορούν να αναπαρασταθούν με έναν πεπερασμένο αριθμό δεκαδικών ψηφίων αλλά δεν έχουν πεπερασμένη αναπαράσταση στο δυαδικό. Το 0.1 είναι ένας τέτοιος αριθμός.

Γενικά, οι αριθμοί κινητής υποδιαστολής μπορεί να οδηγήσουν σε άσχημες καταστάσεις αν περιμένετε από αυτούς να φερθούν όπως οι πραγματικοί. Η χρήση τους αρκεί για αυτά που θα κάνουμε στην Creative Scala, αλλά καλύτερα να μην τους χρησιμοποιήσετε για να γράψετε λογιστικό λογισμικό!

Return to the exercise

15.1.6 Solution to: Πέρα από τις Εκφράσεις

Αυτή είναι μία ερώτηση με πολύ γενική απάντηση. Υπάρχουν διάφοροι τρόποι ώστε να ξεπεράσουμε το μοντέλο που ξέρουμε μέχρι τώρα.

Για να είναι τα προγράμματά μας χρήσιμα, πρέπει να είναι ικανά να δημιουργούν έναν αντίκτυπο—αλλαγές στον κόσμο που ξεπερνούν τα όρια της μνήμης του υπολογιστή. Για παράδειγμα, να μπορούν να εμφανίζουν πράγματα στην οθόνη, να αναπαράγουν ήχους, να στέλνουν μηνύματα σε άλλους υπολογιστές και ούτω καθεξής. Η κονσόλα κάνει κάποια από αυτά για εμάς, εκτυπώνοντας τιμές στην οθόνη. Θα πρέπει να πάμε και λίγο πιο πέρα από αυτά για να καταφέρουμε να φτιάξουμε πιό χρήσιμα προγράμματα.

Επίσης, δεν έχουμε μάθει ακόμη κάποιον τρόπο ώστε να μπορούμε να ορίζουμε δικά μας αντικείμενα και μεθόδους ή να επαναχρησιμοποιήσουμε τιμές στα προγράμματά μας. Αν, για παράδειγμα, θέλαμε να χρησιμοποιήσουμε το όνομα κάποιου πολλές φορές σε ένα πρόγραμμα μας, θα έπρεπε κάθε φορά που το χρειαζόμαστε να το ξαναγράφουμε. Πρέπει να μάθουμε περισσότερες μεθόδους *αφαιρετικότητας* και θα φτάσουμε σε αυτό το επίπεδο πολύ σύντομα.

Return to the exercise

15.2 Δουλεύοντας με Εικόνες

15.2.1 Solution to: Κάνοντας Κύκλους

Σε αυτή την άσκηση ελέγχουμε αν η εγκατάσταση του Doodle δουλεύει σωστά και εξοικειωνόμαστε με την βιβλιοθήκη. Ένα από τα σημαντικά στοιχεία του Doodle είναι ότι διαχωρίζουμε τον *ορισμό μίας εικόνας* από τον *σχεδιασμό της*. Θα μιλήσουμε περισσότερο γι' αυτό αργότερα στο βιβλίο.

Μπορούμε να δημιουργήσουμε τους κύκλους με τον παρακάτω κώδικα.

```
circle(1)
circle(10)
circle(100)
```

Μπορούμε να σχεδιάσουμε τους κύκλους αυτούς καλώντας την μέθοδο `draw` για κάθε έναν ξεχωριστά.

```
circle(1).draw
circle(10).draw
circle(100).draw
```

Return to the exercise

15.2.2 Solution to: Τύποι Σχημάτων

Όλα έχουν τον τύπο `Image` όπως μπορούμε να διαπιστώσουμε και από την κονσόλα.

```
:type circle(10)
// doodle.core.Image
:type rectangle(10, 10)
// doodle.core.Image
:type triangle(10, 10)
// doodle.core.Image
```

Return to the exercise

15.2.3 Solution to: Τύποι Σχεδίων

Για άλλη μία φορά, μπορούμε να κάνουμε αυτή την ερώτηση στην κονσόλα.

```
:type circle(10).draw
// Unit
```

Βλέπουμε ότι ο τύπος του σχεδίου μίας εικόνας είναι ο `Unit`. Ο `Unit` είναι ένας τύπος που χρησιμοποιείται για εκφράσεις που δεν επιστρέφουν κάποια ενδιαφέρουσα τιμή. Αυτό συμβαίνει και στην περίπτωση της `draw`. Την καλούμε επειδή θέλουμε να εμφανιστεί κάτι στην οθόνη και όχι επειδή έχουμε μία χρήση για την τιμή που επιστρέφει. Υπάρχει μόνο μία τιμή με τύπο `Unit`. Αυτή η τιμή ονομάζεται επίσης unit και αν γραφεί ως κυριολεκτική έκφραση τότε θα είναι η `()`

Θα παρατηρήσετε ότι η κονσόλα δεν εκτυπώνει την unit από μόνη της.

```
()
```

Μπορούμε να ζητήσουμε τον τύπο από την κονσόλα ώστε να δείξουμε ότι όντως υπάρχει η unit.

```
:type ()
// Unit
```

Return to the exercise

15.2.4 Solution to: Η Διάμετρος ενός Κύκλου

Είναι τρεις μικροί κύκλοι πάνω από έναν μεγαλύτερο κύκλο. Μπορούμε να φτιάξουμε αυτή την εικόνα με τον παρακάτω κώδικα.

```
(circle(20) beside circle(20) beside circle(20)) on
circle(60)
// res0: doodle.core.Image = On(Beside(Beside(Circle
(20.0),Circle(20.0)),Circle(20.0)),Circle(60.0))
```

Return to the exercise

15.2.5 Solution to: Το Κακό Μάτι

Παρακάτω μπορείτε να δείτε τον κώδικα για το δικό μας φυλαχτό:

```
((circle(10) fillColor Color.black) on
 (circle(20) fillColor Color.cornflowerBlue) on
 (circle(30) fillColor Color.white) on
 (circle(50) fillColor Color.darkBlue))
// res0: doodle.core.Image = On(On(On(ContextTransfo
rm(doodle.core.Image$$Lambda$4075/1773387545@4956730
1,Circle(10.0)),ContextTransform(doodle.core.Image$$
Lambda$4075/1773387545@5d35731a,Circle(20.0))),Conte
xtTransform(doodle.core.Image$$Lambda$4075/177338754
5@5ecb0318,Circle(30.0))),ContextTransform(doodle.co
re.Image$$Lambda$4075/1773387545@2a7434af,Circle(50.
0)))
```

Return to the exercise

15.2.6 Solution to: Συμπληρωματικά Τρίγωνα

Τα παραδείγματα γίνονται αρκετά μεγάλα για να τα γράφουμε στην κονσόλα. Αργότερα θα δούμε έναν διαφορετικό τρόπο.

```
((triangle(40, 40)
       lineWidth 6.0
       lineColor Color.darkSlateBlue
       fillColor (Color.darkSlateBlue lighten 0.3.no
rmalized saturate 0.2.normalized spin 10.degrees)) a
bove
  ((triangle(40, 40)
      lineWidth 6.0
      lineColor (Color.darkSlateBlue spin (-30.degre
es))
      fillColor (Color.darkSlateBlue lighten 0.3.nor
malized saturate 0.2.normalized spin (-20.degrees)))
```

```
 beside
     (triangle(40, 40)
        lineWidth 6.0
        lineColor (Color.darkSlateBlue spin (30.degr
ees))
        fillColor (Color.darkSlateBlue lighten 0.3.n
ormalized saturate 0.2.normalized spin (40.degrees))
)))
// res19: doodle.core.Image = Above(ContextTransform
(doodle.core.Image$$Lambda$5529/665359163@5142eb32,C
ontextTransform(doodle.core.Image$$Lambda$5531/14250
08609@c913fa2,ContextTransform(doodle.core.Image$$La
mbda$5530/259686800@340d130d,Triangle(40.0,40.0)))),
Beside(ContextTransform(doodle.core.Image$$Lambda$55
29/665359163@1f762a82,ContextTransform(doodle.core.I
mage$$Lambda$5531/142500609@59ab5925,ContextTransfo
rm(doodle.core.Image$$Lambda$5530/259686800@5c42119d
,Triangle(40.0,40.0))),ContextTransform(doodle.core
.Image$$Lambda$5529/665359163@8be463,ContextTransfor
m(doodle.core.Image$$Lambda$5531/142500609@2e87ecca
,ContextTransform(doodle.core.Image$$Lambda$5530/259
686800@2de6ea4e,Triangle(40.0,40.0))))))
```

Return to the exercise

15.2.7 Solution to: Σύνθετος Στόχος

Η πιο απλή λύση είναι να δημιουργήσουμε τρεις ομόκεντρους κύκλους χρησιμοποιώντας τον τελεστή `on`:

```
(circle(10) on circle(20) on circle(30))
```

Για περισσότερη εξάσκηση δημιουργήσαμε ένα στήριγμα χρησιμοποιώντας δύο ορθογώνια:

```
(
  circle(10) on
  circle(20) on
```

```
  circle(30) above
  rectangle(6, 20) above
  rectangle(20, 6)
)
```

Return to the exercise

15.2.8 Solution to: Μείνετε στον Στόχο

Το κόλπο εδώ είναι η χρήση παρενθέσεων ώστε να ελέγξετε την σειρά με την οποία χρωματίζεται η σύνθεση. Κάθε μία από τις μεθόδους `fillColor()`, `lineColor()`, και `lineWidth()` εφαρμόζεται σε μία μόνο εικόνα. Τέλος, πρέπει να σιγουρευτούμε ότι η εικόνα αποτελείται από τα σωστά σχήματα:

```
(
  ( circle(10) fillColor Color.red ) on
  ( circle(20) fillColor Color.white ) on
  ( circle(30) fillColor Color.red lineWidth 2 ) abo
ve
  ( rectangle(6, 20) above rectangle(20, 6) fillColo
r Color.brown ) above
  ( rectangle(80, 25) lineWidth 0 fillColor Color.gr
een )
)
```

Return to the exercise

15.3 Γράφοντας Μεγαλύτερα Προγράμματα

15.3.1 Solution to: To Top-Level

Όχι, η Scala δεν μας το επιτρέπει. Για παράδειγμα, δεν μπορούμε να γράψουμε:

```
object {}
// <console>:2: error: identifier expected but '{' f
ound.
// object {}
//        ^
```

Πρέπει να δίνουμε ονόματα σε όλες τις κυριολεκτικές εκφράσεις αντικειμένων που δημιουργούμε.

Return to the exercise

15.3.2 Solution to: To Top-Level Part 2

Φυσικά και μπορούμε!

Μπορούμε να τοποθετήσουμε μία `val` μέσα σε μία κυριολεκτική έκφραση αντικειμένου όπως παρακάτω:

```
object Example {
  val hi = "Hi!"
}
```

Μπορούμε μετά να αναφερθούμε σε αυτό χρησιμοποιώντας την σύνταξη με την τελεία `.` που έχουμε δει επανειλημμένως.

```
Example.hi
// res2: String = Hi!
```

Παρατηρήστε ότι δεν μπορούμε να χρησιμοποιήσουμε το `hi` μόνο του

```
hi
// <console>:28: error: not found: value hi
//        hi
//        ^
```

Πρέπει να πούμε στην Scala ότι θέλουμε να αναφερθούμε στο όνομα `hi` που έχει οριστεί μέσα στο αντικείμενο `Example`.

Return to the exercise

15.3.3 Solution to: Ασκήσεις

Ένα απλό παράδειγμα για να ξεκινήσετε. Το `answer` είναι `1 + 2`, δηλαδή `3`.

Return to the exercise

15.3.4 Solution to: Ασκήσεις Part 2

Ακόμα ένα απλό παράδειγμα. Το `answer` είναι `1 + 2`, δηλαδή `3`. Η εμβέλεια του `Two.a` δεν φτάνει μέχρι εκεί που είναι ορισμένο το `answer`.

Return to the exercise

15.3.5 Solution to: Ασκήσεις Part 3

Εδώ το `Answer.a` επισκιάζει το `One.a` άρα το `answer` είναι `1 + 2`, δηλαδή `3`.

Return to the exercise

15.3.6 Solution to: Ασκήσεις Part 4

Αυτό είναι μία χαρά. Η έκφραση `a + 1` στην δεξιά πλευρά της δήλωσης του `b` είναι μία έκφραση όπως όλες τις άλλες άρα το `answer` είναι και πάλι `3`.

Return to the exercise

15.3.7 Solution to: Ασκήσεις Part 5

Αυτός ο κώδικας δεν θα μεταγλωττιστεί αφού η εμβέλεια του `b` δεν φτάνει μέχρι εκεί που έχει οριστεί το `answer`.

Return to the exercise

15.3.8 Solution to: Ασκήσεις Part 6

Ερώτηση παγίδα! Αυτός ο κώδικας δεν δουλεύει. Εδώ τα `a` και `b` είναι ορισμένα έτσι ώστε να έχουν σχέση μεταξύ τους κατι που όδηγεί σε

φαύλο κύκλο.

Return to the exercise

15.3.9 Solution to: Τοξοβολία και Πάλι

Εμείς αποφασίσαμε να δώσουμε όνομα στον στόχο, στο στήριγμα και στο έδαφος, όπως φαίνεται παρακάτω. Έτσι γίνεται σαφές το πώς κατασκευάστηκε η τελική εικόνα. Το να ονομάσουμε περισσότερα στοιχεία μας φάνηκε ότι δεν θα βοηθούσε στην κατανόηση.

```
val coloredTarget =
  (
    Image.circle(10).fillColor(Color.red) on
      Image.circle(20).fillColor(Color.white) on
      Image.circle(30).fillColor(Color.red)
  )

val stand =
  Image.rectangle(6, 20) above Image.rectangle(20, 6
).fillColor(Color.brown)

val ground =
  Image.rectangle(80, 25).lineWidth(0).fillColor(Col
or.green)

val image = coloredTarget above stand above ground
```

Return to the exercise

15.3.10 Solution to: Σκηνικό Δρόμου

Παρακάτω θα βρείτε την δική μας λύση. Όπως μπορείτε να δείτε, χωρίσαμε το σκηνικό σε μικρότερα κομμάτια ώστε να περιορίσουμε την έκταση του κώδικά μας.

```
val roof = Image.triangle(50, 30) fillColor Color.br
own
```

```
val frontDoor =
  (Image.rectangle(50, 15) fillColor Color.red) abov
e (
    (Image.rectangle(10, 25) fillColor Color.black)
on
    (Image.rectangle(50, 25) fillColor Color.red)
  )

val house = roof above frontDoor

val tree =
  (
    (Image.circle(25) fillColor Color.green) above
    (Image.rectangle(10, 20) fillColor Color.brown)
  )

val streetSegment =
  (
    (Image.rectangle(30, 3) fillColor Color.yellow)
beside
    (Image.rectangle(15, 3) fillColor Color.black) a
bove
    (Image.rectangle(45, 7) fillColor Color.black)
  )

val street = streetSegment beside streetSegment besi
de streetSegment

val houseAndGarden =
  (house beside tree) above street

val image = (
  houseAndGarden beside
  houseAndGarden beside
  houseAndGarden
) lineWidth 0
```

Return to the exercise

15.4 Το Μοντέλο Αντικατάστασης για Αξιολόγηση

15.4.1 Solution to: Αντικαταστάσεις και Println

Παρακάτω μπορείτε να βρείτε ένα απλό παράδειγμα με println. Τα δύο επόμενα προγράμματα είναι εμφανώς διαφορετικά.

```
println("Happy birthday to you!")
// Happy birthday to you!

println("Happy birthday to you!")
// Happy birthday to you!

println("Happy birthday to you!")
// Happy birthday to you!

val a = println("Happy birthday to you!")
// Happy birthday to you!
// a: Unit = ()

a

a

a
```

Έτσι βλέπουμε ότι δεν μπορούμε να χρησιμοποιήσουμε την μέθοδο της αντικατάστασης ελεύθερα υπό την παρουσία των side effects και πρέπει να λάβουμε υπόψη την σειρά αξιολόγησης.

Return to the exercise

15.4.2 Solution to: Τρέλα στις Μεθόδους μας

Στο επόμενο παράδειγμα φαίνεται ότι οι παράμετροι μεθόδων αξιολογούνται από τα αριστερά προς τα δεξιά.

```
Color.hsl(
  {
    println("a")
    0.degrees
  },
  {
    println("b")
    1.normalized
  },
  {
    println("c")
    1.normalized
  }
)
// a
// b
// c
// res16: doodle.core.Color = HSLA(Angle(0.0),Normal
ized(1.0),Normalized(1.0),Normalized(1.0))
```

Μπορούμε να το γράψουμε εν συντομία όπως παρακάτω

```
Color.hsl({ println("a"); 0.degrees },
          { println("b"); 1.normalized },
          { println("c"); 1.normalized })
// a
// b
// c
// res17: doodle.core.Color = HSLA(Angle(0.0),Normal
ized(1.0),Normalized(1.0),Normalized(1.0))
```

Return to the exercise

15.4.3 Solution to: Η Τελευταία Σειρά

Έχουμε ήδη δει ότι οι εκφράσεις αξιολογούνται από πάνω προς τα κάτω και οι παράμετροι μεθόδων από τα αριστερά προς τα δεξιά. Παρακάτω θα ελέγξουμε αν όλες οι εκφράσεις, αξιολογούνται από τα αριστερά προς τα δεξιά. Αυτό μπορούμε να το δείξουμε αρκετά εύκολα.

```
{ println("a"); 1 } + { println("b"); 2 } + { println
n("c"); 3}
// a
// b
// c
// res18: Int = 6
```

Το συμπέρασμα που προκείπτει είναι ότι οι εκφράσεις στην Scala αξιολογούνται από πάνω προς τα κάτω και από τα αριστερά προς τα δεξιά.

Return to the exercise

15.5 Μέθοδοι

15.5.1 Solution to: Τετράγωνο

Η λύση είναι η παρακάτω

```
def square(x: Int): Int =
  x * x
```

Μπορούμε να φτάσουμε στην λύση με τα ακόλουθα βήματα.

Μας δίνεται το όνομα (`square`), ο τύπος της παραμέτρου (`Int`) και ο τύπος του αποτελέσματος (`Int`). Από αυτά μπορούμε να γράψουμε τον σκελετό της μεθόδου

```
def square(x: Int): Int =
```

```
  ???
```

όπου έχουμε επιλέξει (αυθαίρετα) το `x` ως το όνομα της παραμέτρου. Στις περιπτώσεις όπου δεν υπάρχει κάποιο όνομα με συγκεκριμένο νόημα συχνά θα βλέπετε να χρησιμοποιούνται ονόματα που αποτελούνται από ένα γράμμα όπως `x`, `v`, ή `i`.

Παρεμπιπτόντως, αυτός ο κώδικας είναι έγκυρος. Βάλτε τον στην κονσόλα και δείτε! Τι συμβαίνει όταν καλείτε την `square` όπως είναι ορισμένη παραπάνω;

Τώρα πρέπει να ολοκληρώσουμε το σώμα. Ξέρουμε ότι το τετράγωνο είναι ο πολλαπλασιασμός ενός αριθμού με τον εαυτό του, άρα θα αντικαταστήσουμε τα `???` με το `x * x`. Δεν χρειάζεται να το τοποθετήσουμε μέσα σε αγκύλες, αφού στο σώμα υπάρχει μόνο μία έκφραση.

Return to the exercise

15.5.2 Solution to: Μισό

```
def halve(x: Double): Double =
 x / 2.0
```

Για να φτάσουμε στην λύση, μπορούμε να ακολουθήσουμε την ίδια διαδικασία όπως κάναμε παραπάνω με την `square`.

Return to the exercise

15.5.3 Solution to: Άσκηση

Το παρακάτω πρόγραμμα δείχνει ότι οι παράμετροι αξιολογούνται πριν το σώμα της μεθόδου.

```
def example(a: Int, b: Int): Int = {
  println("In the method body!")
  a + b
}
// example: (a: Int, b: Int)Int
```

```
example({ println("a"); 1 }, { println("b"); 2 })
// a
// b
// In the method body!
// res6: Int = 3
```

Ο εναλλακτικός τρόπος που περιγράψαμε παραπάνω χρησιμοποιείται από μερικές γλώσσες, κυρίως στην Haskell και είναι γνωστός ως lazy (τεμπέλα) αξιολόγηση ή non-strict (μη-αυστηρή).

Return to the exercise

15.6 Δομημένη Αναδρομή

15.6.1 Solution to: Άσκηση: Στοιβάζοντας Κουτιά

Το μόνο που πρέπει να κάνετε είναι να αλλάξε το `beside` σε `above`.

```
def stackedBoxes(count: Int): Image =
  count match {
    case 0 => Image.empty
    case n => aBox beside stackedBoxes(n-1)
  }
// stackedBoxes: (count: Int)doodle.core.Image
```

Return to the exercise

15.6.2 Solution to: Μαντέψτε το Αποτέλεσμα

Το πρώτο παράδειγμα αξιολογείται με `2`, αφού το μοτίβο `"abcd"` είναι το μόνο που αντιστοιχίζεται με την κυριολεκτική έκφραση `"abcd"`.

Το δεύτερο παράδειγμα αξιολογείται με `"one"`, αφού αξιολογείται η πρώτη περίπτωση η οποία αντιστοιχίζεται με την έκφραση προς αντιπαραβολή.

Το τρίτο παράδειγμα αξιολογείται με `2`, αφού το `case n` ορίζει ένα μοτίβο

μπαλαντέρ το οποίο μπορεί να αντιστοιχηθεί με οτιδήποτε.

Το τελευταίο παράδειγμα αξιολογείται με `1` αφού αξιολογείται η πρώτη περίπτωση.

Return to the exercise

15.6.3 Solution to: Όταν Δεν Υπάρχει Αντιστοίχιση

Παρακάτω μπορείτε να δείτε τρείς λογικές πιθανότητες που σκεφτήκαμε εμείς. Εσείς σκεφτήκατε κάτι διαφορετικό;

- Η έκφραση θα μπορούσε να αξιολογείται με κάτι προκαθορισμένο, όπως το `Image.empty` (όμως πώς θα ήξερε η Scala ποιο θα ήταν αυτό;)
- Ο μεταγλωττιστής της Scala δεν θα σας άφηνε να γράψετε τέτοιου είδους κώδικα.
- Η έκφραση `match` θα αποτύχει κατά την διάρκεια της εκτέλεσης.

Παρακάτω μπορείτε να δείτε μία έκφραση match που θα αποτύχει.

```
2 match {
  case 0 => "zero"
  case 1 => "one"
}
// scala.MatchError: 2 (of class java.lang.Integer)
//   ... 309 elided
```

Η σωστή απάντηση είναι μία από τις δύο τελευταίες, δηλαδή είτε να εμφανιστεί αποτυχία κατά την μεταγλώττιση είτε κατά την εκτέλεση. Στο παραπάνω παράδειγμα το πρόβλημα θα εμφανιστεί στην εκτέλεση. Η ακριβής απάντηση εξαρτάται από το πώς είναι ρυθμισμένη η Scala (μπορούμε να πούμε στον μεταγλωττιστή να μην μεταγλωττίζει matches που δεν είναι εξαντλητικές, αλλά αυτή δεν είναι η προκαθορισμένη του συμπεριφορά).

Return to the exercise

15.6.4 Solution to: Σταυρός

Θα χρησιμοποιήσουμε δομημένη αναδρομή με φυσικούς αριθμούς. Αυτό που γράψατε θα πρέπει να μοιάζει με το παρακάτω

```
def cross(count: Int): Image =
  count match {
    case 0 => <resultBase>
    case n => <resultUnit> <add> cross(n-1)
  }
```

Return to the exercise

15.6.5 Solution to: Σταυρός Part 2

Από την εικόνα μπορούμε να καταλάβουμε ότι η βασική περίπτωση είναι ένας κύκλος.

Διαδοχικά στοιχεία στην εικόνα προσθέτουν κύκλους πάνω, κάτω, δεξιά και αριστερά. Άρα η μονάδα είναι η βασική περίπτωση, δηλαδή ένας κύκλος, αλλά ο τελεστής της πρόσθεσης δεν είναι ένα απλό `beside` ή ένα `above` όπως είχαμε δει προηγουμένως. Σ' αυτή την περίπτωση είναι το `unit beside (unit above cross(n-1) above unit) beside unit`.

Return to the exercise

15.6.6 Solution to: Σταυρός Part 3

Παρακάτω μπορείτε να δείτε την δική μας λύση.

```
def cross(count: Int): Image = {
  val unit = Image.circle(20)
  count match {
    case 0 => unit
    case n => unit beside (unit above cross(n-1) abo
ve unit) beside unit
  }
}
```

Return to the exercise

15.6.7 Solution to: Σκακιέρα

Η `chessboard` είναι μία δομημένη αναδρομή με φυσικούς αριθμούς, οπότε μπορούμε να γράψουμε απευθείας τον σκελετό της.

```
def chessboard(count: Int): Image =
  count match {
    case 0 => resultBase
    case n => resultUnit add cross(n-1)
  }
```

Όπως και προηγουμένως, πρέπει να αποφασίσουμε πώς θα είναι η βασική περίπτωση, η μονάδα και η πρόσθεση για το αποτέλεσμα. Σας δώσαμε μία βοήθεια, παρουσιάζοντας την εξέλιξη της σκακιέρας στην εικόνα fig. 24. Καταλαβαίνουμε ότι η βασική περίπτωση είναι μία σκακιέρα 2 επί 2.

```
val blackSquare = Image.rectangle(30, 30) fillColor
Color.black
val redSquare   = Image.rectangle(30, 30) fillColor
Color.red

val base =
  (redSquare beside blackSquare) above (blackSquare
beside redSquare)
```

Τώρα πρέπει να βρούμε την μονάδα και την πρόσθεση. Εδώ βλέπουμε κάτι διαφορετικό σε σχέση με τα προηγούμενα παραδείγματα. Η μονάδα είναι η τιμή που παίρνουμε από την αναδρομική κλήση `chessboard(n-1)`. Η πρόσθεση είναι το `(unit beside unit) above (unit beside unit)`.

Βάζοντάς τα όλα μαζί παίρνουμε τον παρακάτω κώδικα

```
def chessboard(count: Int): Image = {
  val blackSquare = Image.rectangle(30, 30) fillColo
r Color.black
```

```
  val redSquare   = Image.rectangle(30, 30) fillColo
r Color.red

  val base =
    (redSquare   beside blackSquare) above (blackSqu
are beside redSquare)
  count match {
    case 0 => base
    case n =>
      val unit = chessboard(n-1)
      (unit beside unit) above (unit beside unit)
  }
}
```

Αν έχετε προηγούμενη εμπειρία στον προγραμματισμό, ίσως σκεφτήκατε ότι μπορείτε να φτιάξετε μία σκακιέρα χρησιμοποιώντας δύο εμφωλευμένους βρόγχους. Εδώ εμείς ακολουθήσαμε μία διαφορετική προσέγγιση ορίζοντας μία μεγάλη σκακιέρα ως σύνθεση μικρότερων. Η κατανόηση αυτής της διαφορετικής προσέγγισης για αποσύνθεση προβλημάτων είναι ένα βήμα κλειδί για να γίνετε ικανοί στον συναρτησιακό προγραμματισμό.

Return to the exercise

15.6.8 Solution to: Τρίγωνο Sierpinkski

Το βήμα κλειδί είναι να αναγνωρίσουμε ότι η βασική μονάδα του τριγώνου Sierpinski είναι η `triangle above (triangle beside triangle)`. Μόλις κατανοήσουμε το παραπάνω, θα δούμε ότι ο κώδικας έχει ακριβώς την ίδια δομή με την `chessboard`. Παρακάτω μπορείτε να δείτε την δικιά μας λύση.

```
def sierpinski(count: Int): Image = {
  val triangle = Image.triangle(10, 10) lineColor Co
lor.magenta
  count match {
    case 0 => triangle above (triangle beside triang
le)
```

```
    case n =>
      val unit = sierpinski(n-1)
      unit above (unit beside unit)
  }
}
// sierpinski: (count: Int)doodle.core.Image
```

Return to the exercise

15.6.9 Solution to: Ασκήσεις

Λειτουργεί όπως πρέπει! Η βασική περίπτωση είναι ξεκάθαρη. Κοιτώντας την αναδρομική περίπτωση, υποθέτουμε ότι το `identity(n-1)` επιστρέφει το identity για `n-1` (το οποίο είναι ακριβώς `n-1`). Το identity για `n` είναι `1 + identity(n-1)`.

Return to the exercise

15.6.10 Solution to: Ασκήσεις Part 2

Κάτι δεν πάει καλά! Αυτή η μέθοδος είναι λάθος για δύο διαφορετικούς λόγους. Πρώτον, αφού υπάρχει πολλαπλασιασμός μέσα στην αναδρομική περίπτωση, στο τέλος θα καταλήξουμε να πολλαπλασιάζουμε με την βασική υπόθεση, δηλαδή με το μηδέν. Άρα το αποτέλεσμα θα είναι πάντα μηδέν.

Μπορούμε να προσπαθήσουμε να το διορθώσουμε προσθέτοντας μία περίπτωση για το `1` (και μετά να αναρωτηθούμε γιατί μας απογοήτευσε ο σκελετός της δομημένης αναδρομής).

```
def double(n: Int): Int =
  n match {
    case 0 => 0
    case 1 => 1
    case n => 2 * double(n-1)
  }
```

Παρόλα, αυτά πάλι δεν παίρνουμε σωστό αποτέλεσμα! Κάνουμε κάτι λάθος στην αναδρομική περίπτωση: θα έπρεπε να προσθέτουμε αντί να

πολλαπλασιάζουμε.

Λίγη άλγεβρα:

```
2(n-1 + 1) == 2(n-1) + 2
```

Άρα αν το `double(n-1)` είναι ισοδύναμο με το `2(n-1)` τότε θα πρέπει να *προσθέσουμε* 2 και όχι να πολλαπλασιάσουμε με το 2. Η σωστή υλοποίηση της μεθόδου είναι η παρακάτω

```
def double(n: Int): Int =
  n match {
    case 0 => 0
    case n => 2 + double(n-1)
  }
```

Return to the exercise

15.6.11 Solution to: Κουτιά που Αλλάζουν Χρώμα

Υπάρχουν δύο τρόποι για να φτάσετε στην λύση. Κατά την χρήση της βοηθητικής παραμέτρου, θα προστεθεί μία ακόμη παράμετρος στην `gradientBoxes`. Έτσι θα περάσουμε το `Color` (χρώμα) μέσα στη δομημένη αναδρομή.

```
def gradientBoxes(n: Int, color: Color): Image =
  n match {
    case 0 => Image.empty
    case n => aBox.fillColor(color) beside gradientB
oxes(n-1, color.spin(15.degrees))
  }
// gradientBoxes: (n: Int, color: doodle.core.Color)
doodle.core.Image
```

Μπορούμε ακόμη να βάλουμε την fill color σε μία συνάρτηση `n`, όπως κάναμε προηγουμένως με το size για το μέγεθος των κουτιών στην `growingBoxes`.

```
def gradientBoxes(n: Int): Image =
  n match {
    case 0 => Image.empty
    case n => aBox.fillColor(Color.royalBlue.spin((15*n).degrees)) beside gradientBoxes(n-1)
  }
// gradientBoxes: (n: Int)doodle.core.Image
```

Return to the exercise

15.6.12 Solution to: Ομόκεντροι κύκλοι

Η λύση είναι παρόμοια με την `growingBoxes`.

```
def concentricCircles(count: Int, size: Int): Image =
  count match {
    case 0 => Image.empty
    case n => Image.circle(size) on concentricCircles(n-1, size + 5)
  }
// concentricCircles: (count: Int, size: Int)doodle.core.Image
```

Return to the exercise

15.6.13 Solution to: Άλλη μία Άσκηση

Δείτε παρακάτω την δική μας λύση, στην οποία προσπαθήσαμε να σπάσουμε το πρόβλημα σε επαναχρησιμοποιήσιμα κομμάτια ώστε να μειωθεί ο επαναλαμβανόμενος κώδικας. Παρόλα αυτά, υπάρχει ακόμη αρκετός κώδικας που επαναλαμβάνεται, αφού ακόμα δεν έχουμε τα κατάλληλα εργαλεία για να τον μειώσουμε περισσότερο. Θα επιστρέψουμε σ' αυτό σύντομα.

```
def circle(size: Int, color: Color): Image =
  Image.circle(size).lineWidth(3.0).lineColor(color)
```

```
// circle: (size: Int, color: doodle.core.Color)dood
le.core.Image

def fadeCircles(n: Int, size: Int, color: Color): Im
age =
  n match {
    case 0 => Image.empty
    case n => circle(size, color) on fadeCircles(n-1
, size+7, color.fadeOutBy(0.05.normalized))
  }
// fadeCircles: (n: Int, size: Int, color: doodle.co
re.Color)doodle.core.Image

def gradientCircles(n: Int, size: Int, color: Color)
: Image =
  n match {
    case 0 => Image.empty
    case n => circle(size, color) on gradientCircles
(n-1, size+7, color.spin(15.degrees))
  }
// gradientCircles: (n: Int, size: Int, color: doodl
e.core.Color)doodle.core.Image

def image: Image =
  fadeCircles(20, 50, Color.red) beside gradientCirc
les(20, 50, Color.royalBlue)
// image: doodle.core.Image
```

Return to the exercise

15.6.14 Solution to: Σκακιέρα

Δείτε παρακάτω πώς το υλοποιήσαμε εμείς. Είναι ακριβώς ο ίδιος τρόπος που χρησιμοποιήσαμε στην `boxes`.

```
def chessboard(count: Int): Image = {
  val blackSquare = Image.rectangle(30, 30) fillColo
```

```
r Color.black
  val redSquare   = Image.rectangle(30, 30) fillColo
r Color.red
  val base =
    (redSquare   beside blackSquare) above (blackSqu
are beside redSquare)
  def loop(count: Int): Image =
    count match {
      case 0 => base
      case n =>
        val unit = loop(n-1)
        (unit beside unit) above (unit beside unit)
    }

  loop(count)
}
// chessboard: (count: Int)doodle.core.Image
```

Return to the exercise

15.6.15 Solution to: Έξυπνα Κουτιά

Μπορούμε να το κάνουμε σε δύο στάδια, πρώτα πρέπει να μετακινήσουμε το `aBox` μέσα στην `boxes`.

```
def boxes(count: Int): Image = {
  val aBox = Image.rectangle(20, 20).fillColor(Color
.royalBlue)
  count match {
    case 0 => Image.empty
    case n => aBox beside boxes(n-1)
  }
}
```

Στην συνέχεια μπορούμε να χρησιμοποιήσουμε μία εσωτερική μέθοδο ώστε να αποφύγουμε την δημιουργία του `aBox` σε κάθε αναδρομή.

```
def boxes(count: Int): Image = {
```

```
  val aBox = Image.rectangle(20, 20).fillColor(Color
.royalBlue)
  def loop(count: Int): Image =
    count match {
      case 0 => Image.empty
      case n => aBox beside loop(n-1)
    }

  loop(count)
}
```

Return to the exercise

15.7 Κηπουρική και Higher-order Συναρτήσεις

15.7.1 Solution to: Τύποι Συναρτήσεων

Ο τύπος της είναι `Angle => Point`. Αυτό σημαίνει ότι η `roseFn` είναι μία συνάρτηση η οποία παίρνει μόνο μία παράμετρο τύπου `Angle` και επιστρέφει μία τιμή με τύπο `Point`. Με άλλα λόγια, η `roseFn` μετατρέπει ένα `Angle` σε `Point`.

Return to the exercise

15.7.2 Solution to: Κυριολεκτικές Εκφράσεις Συναρτήσεων

```
val roseFn = (angle: Angle) =>
  Point.cartesian((angle * 7).cos * angle.cos, (angl
e * 7).cos * angle.sin)
// roseFn: doodle.core.Angle => doodle.core.Point =
$$Lambda$16623/1515324787@1689ad6b
```

Return to the exercise

15.7.3 Solution to: Χρώμα και Σχήμα

Η πιο απλή λύση είναι να ορίσουμε τρεις `singleShapes` όπως παρακάτω:

```
def concentricShapes(count: Int, singleShape: Int =>
 Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n) on concentricShapes(n-1
, singleShape)
  }

def rainbowCircle(n: Int) = {
  val color = Color.blue desaturate 0.5.normalized s
pin (n * 30).degrees
  val shape = Image.circle(50 + n*12)
  shape lineWidth 10 lineColor color
}

def fadingTriangle(n: Int) = {
  val color = Color.blue fadeOut (1 - n / 20.0).norm
alized
  val shape = Image.triangle(100 + n*24, 100 + n*24)
  shape lineWidth 10 lineColor color
}

def rainbowSquare(n: Int) = {
  val color = Color.blue desaturate 0.5.normalized s
pin (n * 30).degrees
  val shape = Image.rectangle(100 + n*24, 100 + n*24
)
  shape lineWidth 10 lineColor color
}

val answer =
  (concentricShapes(10, rainbowCircle) beside
   concentricShapes(10, fadingTriangle) beside
```

```
    concentricShapes(10, rainbowSquare))
```

Όμως, εδώ υπάρχουν κάποια περιττά στοιχεία: συγκεκριμένα το `rainbowCircle` και το `rainbowTriangle`, χρησιμοποιούν τον ίδιο ορισμό για το `color`. Υπάρχουν επίσης επαναλαμβανόμενες κλήσεις του `lineWidth(10)` και του `lineColor(color)` οι οποίες μπορούν να εξαλειφθούν. Αυτά τα δύο υλοποιούνται στην `colored`:

```
def concentricShapes(count: Int, singleShape: Int =>
 Image): Image =
  count match {
    case 0 => Image.empty
    case n => singleShape(n) on concentricShapes(n-1
, singleShape)
  }
// concentricShapes: (count: Int, singleShape: Int =
> doodle.core.Image)doodle.core.Image

def colored(shape: Int => Image, color: Int => Color
): Int => Image =
  (n: Int) =>
    shape(n) lineWidth 10 lineColor color(n)
// colored: (shape: Int => doodle.core.Image, color:
 Int => doodle.core.Color)Int => doodle.core.Image

def fading(n: Int): Color =
  Color.blue fadeOut (1 - n / 20.0).normalized
// fading: (n: Int)doodle.core.Color

def spinning(n: Int): Color =
  Color.blue desaturate 0.5.normalized spin (n * 30)
.degrees
// spinning: (n: Int)doodle.core.Color

def size(n: Int): Double =
  50 + 12 * n
// size: (n: Int)Double
```

```
def circle(n: Int): Image =
  Circle(size(n))
// circle: (n: Int)doodle.core.Image

def square(n: Int): Image =
  Image.rectangle(2*size(n), 2*size(n))
// square: (n: Int)doodle.core.Image

def triangle(n: Int): Image =
  Image.triangle(2*size(n), 2*size(n))
// triangle: (n: Int)doodle.core.Image

val answer =
  (concentricShapes(10, colored(circle, spinning)) b
eside
   concentricShapes(10, colored(triangle, fading)) b
eside
   concentricShapes(10, colored(square, spinning)))
// answer: doodle.core.Image = Beside(Beside(On(Cont
extTransform(doodle.core.Image$$Lambda$16593/1190974
287@64bd53da,ContextTransform(doodle.core.Image$$Lam
bda$16592/1864527597@625a8951,Circle(170.0))),On(Con
textTransform(doodle.core.Image$$Lambda$16593/119097
4287@5e466a9,ContextTransform(doodle.core.Image$$Lam
bda$16592/1864527597@43cd762d,Circle(158.0))),On(Con
textTransform(doodle.core.Image$$Lambda$16593/119097
4287@4de0ce0d,ContextTransform(doodle.core.Image$$La
mbda$16592/1864527597@7bc6c6e1,Circle(146.0))),On(Co
ntextTransform(doodle.core.Image$$Lambda$16593/11909
74287@5db1261e,ContextTransform(doodle.core.Image$$L
ambda$16592/1864527597@75654f9b,Circle(134.0))),On(C
ontextTransform(doodle.core.Image$$Lambda$16593/1190
974287@3e3e679c,ContextTransform(doodle.core.I...
```

Return to the exercise

15.7.4 Solution to: Ξεχωριστά Στοιχεία

Όταν σχεδιάζουμε τις παραμετρικές καμπύλες το πιο πιθανό είναι ότι θα θελήσουμε να αλλάξουμε την ακτίνα των διαφορετικών καμπυλών. Μπορούμε να το κάνουμε με μία συνάρτηση. Ποιος πρέπει να είναι ο τύπος αυτή της συνάρτης; Ίσως η πιο φανερή προσέγγιση είναι να θέσουμε ως τύπο της συνάρτησης τον `(Point, Double) => Point`, όπου η παράμετρος `Double` είναι η ποσότητα με την οποία μετράμε το σημείο. Όμως η χρήση αυτού του τύπου είναι κάπως ενοχλητική. Πρέπει να περνάμε συνεχώς γύρω από την `Double`, η οποία δεν μπορεί να πάρει διαφορετική τιμή από την αρχική της.

Μία καλύτερη δομή είναι να φτιάξουμε μία συνάρτηση με τύπο `Double => (Point => Point)`. Αυτή είναι μία συνάρτηση στην οποία περνάμε τον συντελεστή της κλίμακας. Επιστρέφεται μία συνάρτηση η οποία μετατρέπει ένα `Point` σύμφωνα με τον συντελεστή της κλίμακας. Μ' αυτόν τον τρόπο ξεπερνάμε το ότι η παράμετρος δεν μπορούσε να αλλάξει τιμή. Ένας τρόπος λύσης είναι ο παρακάτω

```
def scale(factor: Double): Point => Point =
  (pt: Point) => {
    Point.polar(pt.r * factor, pt.angle)
  }
```

Προηγουμένως είπαμε ότι θα προτιμούσαμε να ξεχωρίσουμε την παραμετρική εξίσωση από την `sample`. Αυτό μπορούμε εύκολα να το κάνουμε όπως εδώ

```
def sample(start: Angle, samples: Int, location: Ang
le => Point): Image = {
  // Το Angle.one είναι μία ολόκληρη περιστροφή, δηλ
αδή 360 μοίρες
  val step = Angle.one / samples
  val dot = triangle(10, 10)
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
```

```
      case n =>
        dot.at(location(angle).toVec) on loop(n - 1)
    }
  }

  loop(samples)
}
```

Ίσως ακόμη να θέλουμε να απομονώσουμε την επιλογή της βασικής εικόνας (το `dot` ή το `Image.triangle` παραπάνω). Για να το καταφέρουμε, μπορούμε να αλλάξουμε το `location` ώστε να γίνει μία ξεχωριστή συνάρτηση `Angle => Image`.

```
def sample(start: Angle, samples: Int, location: Ang
le => Image): Image = {
  // Το Angle.one είναι μία ολόκληρη περιστροφή, δηλ
αδή 360 μοίρες
  val step = Angle.one / samples
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n => location(angle) on loop(n - 1)
    }
  }

  loop(samples)
}
```

Μπορούμε επίσης να απομονώσουμε όλο το κομμάτι της δομημένης αναδρομής. Όπου είχαμε

```
def loop(count: Int): Image = {
  val angle = step * count
  count match {
    case 0 => Image.empty
    case n => location(angle) on loop(n - 1)
```

```
    }
  }
```

μπορούμε να ξεχωρίσουμε την αρχική περίπτωση (`Image.empty`) και την αναδρομή (`location(angle) on loop(n - 1)`). Το προηγούμενο ήταν απλά μία `Image` αλλά το τελευταίο είναι μία συνάρτηση με τύπο `(Angle, Image) => Image`. Μπορείτε να δείτε το τελικό αποτέλεσμα παρακάτω

```
def sample(start: Angle, samples: Int, empty: Image,
 combine: (Angle, Image) => Image): Image = {
  //To Angle.one είναι μία ολόκληρη περιστροφή, δηλα
δή 360 μοίρες
  val step = Angle.one / samples
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => empty
      case n => combine(angle, loop(n - 1))
    }
  }

  loop(samples)
}
```

Η παραπάνω συνάρτηση έχει μία μεγάλη δόση αφαιρετικότητας. Περιμένουμε ότι οι περισσότεροι άνθρωποι δεν θα δουν αυτή την αφαίρεση, αλλά αν ενδιαφέρεστε μπορείτε να εξερευνήσετε τα folds και τα monoids.

Return to the exercise

15.7.5 Solution to: Σύνθεση

Μπορεί να καταλήξατε και εσείς με κάτι σαν το παρακάτω.

```
def parametricCircle(angle: Angle): Point =
  Point.cartesian(angle.cos, angle.sin)
```

```
def rose(angle: Angle): Point =
  Point.cartesian((angle * 7).cos * angle.cos, (angl
e * 7).cos * angle.sin)

def scale(factor: Double): Point => Point =
  (pt: Point) => {
    Point.polar(pt.r * factor, pt.angle)
  }

def sample(start: Angle, samples: Int, location: Ang
le => Point): Image = {
  // το Angle.one είναι μία ολόκληρη περιστροφή, δηλ
αδή 360 μοίρες
  val step = Angle.one / samples
  val dot = triangle(10, 10)
  def loop(count: Int): Image = {
    val angle = step * count
    count match {
      case 0 => Image.empty
      case n => dot.at(location(angle).toVec) on loo
p(n - 1)
    }
  }

  loop(samples)
}

def locate(scale: Point => Point, point: Angle => Po
int): Angle => Point =
  (angle: Angle) => scale(point(angle))

// Τριαντάφυλλο σε κύκλο
val flower = {
  sample(0.degrees, 200, locate(scale(200), rose _))
 on
  sample(0.degrees, 40, locate(scale(150), parametri
cCircle _))
```

```
}
```

Return to the exercise

15.7.6 Solution to: Πείραμα

Η λύση που χρησιμοποιήσαμε για την δημιουργία της εικόνας fig. 30 βρίσκεται στο Flowers.scala. Τι φτιάξατε εσείς; Αν θέλετε να μας το δείξετε, τα email μας είναι `noel@underscore.io` και `dave@underscore.io`.

Return to the exercise

15.8 Σχήματα, Ακολουθίες και Αστέρια

15.8.1 Solution to: Πολύγωνα

Η χρήση πολικών συντεταγμένων κάνει τον ορισμό της τοποθεσίας των κορυφών των πολυγώνων πολύ πιο εύκολο. Κάθε κορυφή τοποθετείται σε απόσταση μίας συγκεκριμένης γωνίας από την προηγούμενη. Αφού τοποθετήσουμε όλες τις κορυφές, θα πρέπει να έχουμε κάνει μία πλήρη κυκλική περιστροφή. Για παράδειγμα, σε ένα πεντάγωνο η κάθε κορυφή θα τοποθετηθεί με διαφορά (360 / 5) = 72 μοιρών από την προηγούμενη. Αν ξεκινήσουμε από τις 0 μοίρες, οι κορυφές θα τοποθετηθούν στις 0, 72, 144, 216 και 288 μοίρες. Η απόσταση από το αρχικό σημείο είναι σταθερή για κάθε περίπτωση. Δεν χρειάζεται να σχεδιάσουμε την γραμμή που ενώνει την τελευταία κορυφή με την αρχή—χρησιμοποιώντας κλειστό μονοπάτι, αυτό θα γίνει αυτόματα.

Παρακάτω μπορείτε να δείτε τον κώδικά μας για τον σχεδιασμό της εικόνας fig. 40, ο οποίος εφαρμόζει την παραπάνω ιδέα. Σε κάποιες περιπτώσεις, ξεκινήσαμε την τοποθέτηση των κορυφών από γωνία διαφορετική του 0, ώστε να μπορούμε να περιστρέψουμε το σχήμα.

```
import doodle.core.Image._
import doodle.core.PathElement._
import doodle.core.Point._
import doodle.core.Color._
```

```
val triangle =
  closedPath(List(
               moveTo(polar(50, 0.degrees)),
               lineTo(polar(50, 120.degrees)),
               lineTo(polar(50, 240.degrees))
             ))

val square =
  closedPath(List(
               moveTo(polar(50, 45.degrees)),
               lineTo(polar(50, 135.degrees)),
               lineTo(polar(50, 225.degrees)),
               lineTo(polar(50, 315.degrees))
             ))

val pentagon =
  closedPath((List(
                moveTo(polar(50, 72.degrees)),
                lineTo(polar(50, 144.degrees)),
                lineTo(polar(50, 216.degrees)),
                lineTo(polar(50, 288.degrees)),
                lineTo(polar(50, 360.degrees))
              )))

val spacer =
  rectangle(10, 100).noLine.noFill

def style(image: Image): Image =
  image.lineWidth(6.0).lineColor(paleTurquoise).fill
Color(turquoise)

val image =
  style(triangle) beside spacer beside style(square)
 beside spacer beside style(pentagon)
```

Return to the exercise

15.8.2 Solution to: Καμπύλες

Ο πυρήνας της άσκησης βρίσκεται στην αλλαγή της έκφρασης `lineTo` σε `curveTo`. Μπορούμε να τοποθετήσουμε την δημιουργία των καμπυλών μέσα σε μία μέθοδο η οποία δέχεται μία αρχική γωνία και την αύξηση της και κατασκευάζει σημεία ελέγχου σε προκαθορισμένα σημεία της περιστροφής. Αυτό ακριβώς κάνουμε στην μέθοδο `curve` για να πάρουμε καμπύλες χωρίς να χρειάζεται να επαναλάβουμε κάθε φορά τους ίδιους υπολογισμούς. Κάνοντας τα παραπάνω, η αλλαγή των σημείων ελέγχου ώστε να δημιουργήσουμε διαφορετικά αποτελέσματα γίνεται πιο εύκολη.

```
import doodle.core.Image._
import doodle.core.Point._
import doodle.core.PathElement._
import doodle.core.Color._

def curve(radius: Int, start: Angle, increment: Angl
e): PathElement = {
  curveTo(
    polar(radius *  .8, start + (increment * .3)),
    polar(radius * 1.2, start + (increment * .6)),
    polar(radius, start + increment)
  )
}

val triangle =
  closedPath(List(
               moveTo(polar(50, 0.degrees)),
               curve(50, 0.degrees, 120.degrees),
               curve(50, 120.degrees, 120.degrees),
               curve(50, 240.degrees, 120.degrees)
             ))

val square =
  closedPath(List(
               moveTo(polar(50, 45.degrees)),
               curve(50, 45.degrees, 90.degrees),
```

```
                curve(50, 135.degrees, 90.degrees),
                curve(50, 225.degrees, 90.degrees),
                curve(50, 315.degrees, 90.degrees)
              ))

val pentagon =
  closedPath((List(
                moveTo(polar(50, 72.degrees)),
                curve(50, 72.degrees, 72.degrees),
                curve(50, 144.degrees, 72.degrees),
                curve(50, 216.degrees, 72.degrees),
                curve(50, 288.degrees, 72.degrees),
                curve(50, 360.degrees, 72.degrees)
              )))

val spacer =
  rectangle(10, 100).noLine.noFill

def style(image: Image): Image =
  image.lineWidth(6.0).lineColor(paleTurquoise).fill
Color(turquoise)

val image = style(triangle) beside spacer beside sty
le(square) beside spacer beside style(pentagon)
```

Return to the exercise

15.8.3 Solution to: Κατασκευή Λιστών

Είναι δομημένη αναδρομή με φυσικούς αριθμούς!

```
def ones(n: Int): List[Int] =
  n match {
    case 0 => Nil
    case n => 1 :: ones(n - 1)
  }
// ones: (n: Int)List[Int]
```

```
ones(3)
// res5: List[Int] = List(1, 1, 1)
```

Return to the exercise

15.8.4 Solution to: Κατασκευή Λιστών Part 2

Για άλλη μία φορά, μπορούμε να χρησιμοποιήσουμε δομημένη αναδρομή με φυσικούς αριθμούς.

```
def descending(n: Int): List[Int] =
  n match {
    case 0 => Nil
    case n => n :: descending(n - 1)
  }
// descending: (n: Int)List[Int]

descending(0)
// res8: List[Int] = List()

descending(3)
// res9: List[Int] = List(3, 2, 1)
```

Return to the exercise

15.8.5 Solution to: Κατασκευή Λιστών Part 3

Είναι δομημένη αναδρομή με φυσικούς αριθμούς αλλά αυτή τη φορά έχει έναν εσωτερικό συσσωρευτή.

```
def ascending(n: Int): List[Int] = {
  def iter(n: Int, counter: Int): List[Int] =
    n match {
      case 0 => Nil
      case n => counter :: iter(n - 1, counter + 1)
    }
```

```
  iter(n, 1)
}
// ascending: (n: Int)List[Int]

ascending(0)
// res12: List[Int] = List()

ascending(3)
// res13: List[Int] = List(1, 2, 3)
```

Return to the exercise

15.8.6 Solution to: Κατασκευή Λιστών Part 4

Σ' αυτή την άσκηση σας ζητάμε να χρησιμοποιήσετε μία μεταβλητή τύπου. Κατά τα άλλα, έχουμε την ίδια μορφή.

```
def fill[A](n: Int, a: A): List[A] =
  n match {
    case 0 => Nil
    case n => a :: fill(n-1, a)
  }
// fill: [A](n: Int, a: A)List[A]

fill(3, "Hi")
// res16: List[String] = List(Hi, Hi, Hi)

fill(3, Color.blue)
// res17: List[doodle.core.Color] = List(RGBA(Unsign
edByte(-128),UnsignedByte(-128),UnsignedByte(127),No
rmalized(1.0)), RGBA(UnsignedByte(-128),UnsignedByte
(-128),UnsignedByte(127),Normalized(1.0)), RGBA(Unsi
gnedByte(-128),UnsignedByte(-128),UnsignedByte(127),
Normalized(1.0)))
```

Return to the exercise

15.8.7 Solution to: Μετατροπή Λιστών

Αυτή είναι μία δομημένη αναδρομή με λίστα. Η αποδόμηση της εισόδου αντικατοπτρίζεται στην κατασκευή της εξόδου.

```
def double(list: List[Int]): List[Int] =
  list match {
    case Nil => Nil
    case hd :: tl => (hd * 2) :: double(tl)
  }
// double: (list: List[Int])List[Int]

double(List(1, 2, 3))
// res20: List[Int] = List(2, 4, 6)

double(List(4, 9, 16))
// res21: List[Int] = List(8, 18, 32)
```

Return to the exercise

15.8.8 Solution to: Μετατροπή Λιστών Part 2

Είναι μία δομημένη αναδρομή με λίστα η οποία χρησιμοποιεί την ίδια μορφή με την `sum` που είδαμε παραπάνω.

```
def product(list: List[Int]): Int =
  list match {
    case Nil => 1
    case hd :: tl => hd * product(tl)
  }
// product: (list: List[Int])Int

product(Nil)
// res24: Int = 1

product(List(1,2,3))
```

```
// res25: Int = 6
```

Return to the exercise

15.8.9 Solution to: Μετατροπή Λιστών Part 3

Είναι η ίδια μορφή με πριν αλλά τώρα χρησιμοποιούμε μία μεταβλητή τύπου ώστε να επιτρέψουμε όλους τους διαφορετικούς τύπους.

```
def contains[A](list: List[A], elt: A): Boolean =
  list match {
    case Nil => false
    case hd :: tl => (hd == elt) || contains(tl, elt
)
  }
// contains: [A](list: List[A], elt: A)Boolean

contains(List(1,2,3), 3)
// res28: Boolean = true

contains(List("one", "two", "three"), "four")
// res29: Boolean = false
```

Return to the exercise

15.8.10 Solution to: Μετατροπή Λιστών Part 4

Αυτή η μέθοδος είναι παρόμοια με την `contains` αλλά χρησιμοποιήσαμε μία μεταβλητή τύπου για τον τύπο του αποτελέσματος καθώς και για τους τύπους των παραμέτρων.

```
def first[A](list: List[A], elt: A): A =
  list match {
    case Nil => elt
    case hd :: tl => hd
```

```
  }
// first: [A](list: List[A], elt: A)A

first(Nil, 4)
// res32: Int = 4

first(List(1,2,3), 4)
// res33: Int = 1
```

Return to the exercise

15.8.11 Solution to: Άσκηση Πρόκληση: Αντιστροφή

Το κόλπο είναι να χρησιμοποιήσετε έναν συσσωρευτή ώστε να να κρατήσετε την μερικώς αντεστραμμένη λίστα. Αν καταφέρατε να το βρείτε μόνοι σας, συγχαρητήρια!—έχετε καταλάβει την δομημένη αναδρομή πολύ καλά!

```
def reverse[A](list: List[A]): List[A] = {
  def iter(list: List[A], reversed: List[A]): List[A
] =
    list match {
      case Nil => reversed
      case hd :: tl => iter(tl, hd :: reversed)
    }

  iter(list, Nil)
}
// reverse: [A](list: List[A])List[A]

reverse(List(1, 2, 3))
// res36: List[Int] = List(3, 2, 1)

reverse(List("a", "b", "c"))
// res37: List[String] = List(c, b, a)
```

Return to the exercise

15.8.12 Solution to: Πολύγωνα!

Δείτε παρακάτω τον κώδικά μας! Παρατηρήστε ότι οργανώσαμε τον κώδικά μας σε μικρότερα κομμάτια—θα μπορούσαμε να τα κάνουμε και ακόμα πιο μικρά αν θέλαμε. (Μπορείτε να καταλάβετε πώς το καταφέραμε; Βοήθεια: Είναι απαραίτητο να περάσουμε για παράδειγμα, το `start` σε κάθε κλήση της `makeColor` όταν αυτή δεν αλλάζει;)

```
import Point._
import PathElement._

def polygon(sides: Int, size: Int, initialRotation:
Angle): Image = {
  def iter(n: Int, rotation: Angle): List[PathElemen
t] =
    n match {
      case 0 =>
        Nil
      case n =>
        LineTo(polar(size, rotation * n + initialRot
ation)) :: iter(n - 1, rotation)
    }
  closedPath(moveTo(polar(size, initialRotation)) ::
 iter(sides, 360.degrees / sides))
}

def style(img: Image): Image = {
  img.
    lineWidth(3.0).
    lineColor(Color.mediumVioletRed).
    fillColor(Color.paleVioletRed.fadeOut(0.5.normal
ized))
}

def makeShape(n: Int, increment: Int): Image =
```

```
    polygon(n+2, n * increment, 0.degrees)

def makeColor(n: Int, spin: Angle, start: Color): Co
lor =
  start spin (spin * n)

val baseColor = Color.hsl(0.degrees, 0.7.normalized,
 0.7.normalized)

def makeImage(n: Int): Image = {
  n match {
    case 0 =>
      Image.empty
    case n =>
      val shape = makeShape(n, 10)
      val color = makeColor(n, 30.degrees, baseColor
)
      makeImage(n-1) on (shape fillColor color)
  }
}

val image = makeImage(15)
```

Return to the exercise

15.8.13 Solution to: Ranges, Lists και map

Για να το καταφέρουμε, μπορούμε να χρησιμοποιήσουμε την `map` για ένα συγκεκριμένο `Range`.

```
def ones(n: Int): List[Int] =
  (0 until n).toList.map(x => 1)
// ones: (n: Int)List[Int]

ones(3)
// res13: List[Int] = List(1, 1, 1)
```

Return to the exercise

15.8.14 Solution to: Ranges, Lists και map Part 2

Μπορούμε να χρησιμοποιήσουμε ένα `Range` αλλά θα πρέπει να ρυθμίσουμε και το βήμα. Διαφορετικά, το πρόγραμμα δεν θα γνωρίζει τι να κάνει.

```
def descending(n: Int): List[Int] =
  (n until 0 by -1).toList
// descending: (n: Int)List[Int]

descending(0)
// res16: List[Int] = List()

descending(3)
// res17: List[Int] = List(3, 2, 1)
```

Return to the exercise

15.8.15 Solution to: Ranges, Lists και map Part 3

Μπορούμε να χρησιμοποιήσουμε το `Range` και πάλι αλλά θα πρέπει να ξεκινήσουμε από το `0` και να αυξάνουμε τα στοιχεία κατά `1` ώστε να έχουμε τον σωστό αριθμό στοιχείων.

```
def ascending(n: Int): List[Int] =
  (0 until n).toList.map(x => x + 1)
// ascending: (n: Int)List[Int]

ascending(0)
// res20: List[Int] = List()

ascending(3)
```

```
// res21: List[Int] = List(1, 2, 3)
```

Return to the exercise

15.8.16 Solution to: Ranges, Lists και map Part 4

Εδώ ξεκάθαρα πρέπει να χρησιμοποιήσουμε την `map`.

```
def double(list: List[Int]): List[Int] =
  list map (x => x * 2)
// double: (list: List[Int])List[Int]

double(List(1, 2, 3))
// res24: List[Int] = List(2, 4, 6)

double(List(4, 9, 16))
// res25: List[Int] = List(8, 18, 32)
```

Return to the exercise

15.8.17 Solution to: Πολύγωνα και Πάλι!

Πρακάτω μπορείτε να δείτε μία πιθανή λύση. Είναι πολύ πιο εύκολο να την διαβάσετε!

```
def polygon(sides: Int, size: Int, initialRotation:
Angle): Image = {
  import Point._
  import PathElement._

  val step = (Angle.one / sides).toDegrees
  val path =
    (0.0 to 360.0 by step).toList.map{ deg =>
      lineTo(polar(size, initialRotation + deg.degre
es))
    }
```

```
  closedPath(moveTo(polar(size, initialRotation)) ::
 path)
}
```

Return to the exercise

15.8.18 Solution to: Άσκηση Πρόκληση: Πέρα από την map

Έχουμε δει πολλά παραδείγματα στα οποία δεν θα μπορούσαμε να χρησιμοποιήσουμε την `map`: οι μέθοδοι `product`, `sum`, `find`, καθώς και άλλες πολλές στην προηγούμενη ενότητα δεν μπορούν να υλοποιηθούν με την `map`.

Γενικά, οι μέθοδοι που υλοποιούνται με map ακολουθούν την παρακάτω εξίσωση:

```
List[A] map A => B = List[B]
```

Αν το αποτέλεσμα δεν είναι τύπου `List[B]` τότε δεν μπορούμε να την υλοποιήσουμε με `map`. Για παράδειγμα, μέθοδοι όπως οι `product` και `sum` μετατρέπουν μία `List[Int]` σε `Int` και άρα δεν μπορούμε να τις υλοποιήσουμε χρησιμοποιώντας `map`.

Η `map` μετασχηματίζει τα στοιχεία μίας λίστας αλλά δεν μπορεί να αλλάξει τον αριθμό των στοιχείων στο αποτέλεσμα. Ακόμη και αν μία μέθοδος ταιριάζει στην παραπάνω εξίσωση, αν απαιτείται αλλαγή του αριθμού στοιχείων της λίστας, τότε δεν θα μπορέσει να υλοποιηθεί με την `map`.

Return to the exercise

15.8.19 Solution to: Χρησιμοποιώντας το Range

Τώρα που μάθαμε την `to`, η υλοποιηση της `ascending` είναι πολύ εύκολη.

```
def ascending(n: Int): List[Int] =
```

```
  (1 to n).toList
// ascending: (n: Int)List[Int]

ascending(0)
// res30: List[Int] = List()

ascending(3)
// res31: List[Int] = List(1, 2, 3)
```

Return to the exercise

15.8.20 Solution to: Θεέ μου, Πόσα Αστέρια!

Παρακάτω μπορείτε να δείτε την μέθοδο `star`. Μετονομάσαμε τα `p` και `n` σε `points` (κορυφές) και `skip` (αριθμός των κορυφών που θα παραλειφθούν) για καλύτερη κατανόηση:

```
def star(sides: Int, skip: Int, radius: Double): Ima
ge = {
  import Point._
  import PathElement._

  val rotation = 360.degrees * skip / sides

  val start = moveTo(polar(radius, 0.degrees))
  val elements = (1 until sides).toList map { index
=>
    val point = polar(radius, rotation * index)
    lineTo(point)
  }

  closedPath(start :: elements) lineWidth 2
}
```

Return to the exercise

15.8.21 Solution to: Θεέ μου, Πόσα Αστέρια!

Part 2

Θα χρησιμοποιήσουμε την `allBeside` ώστε να δημιουργήσουμε μία σειρά από αστέρια. Για την δημιουργία αυτής της εικόνας θα χρειαστεί να χρησιμοποιήσουμε μόνο τιμές της `skip` από το `1` ως το `sides/2` (στρογγυλοποιημένα προς τα κάτω). Για παράδειγμα:

```
allBeside(
  (1 to 5).toList map { skip =>
    star(11, skip, 100)
  }
)
```

Return to the exercise

15.8.22 Solution to: Θεέ μου, Πόσα Αστέρια! Part 3

Για την δημιουργία της εικόνας fig. 45, ξεκινήσαμε φτιάχνοντας μία μέθοδο για την μορφή των αστεριών.

```
def style(img: Image, hue: Angle): Image = {
  img.
    lineColor(Color.hsl(hue, 1.normalized, .25.norma
lized)).
    fillColor(Color.hsl(hue, 1.normalized, .75.norma
lized))
}
```

Μετά φτιάξαμε την μέθοδο `allAbove`, η οποία όπως θα δείτε μοιάζει πολύ με την `allBeside` (Δεν θα ήταν πολύ ωραίο να μπορούσαμε να τυποποιήσουμε αυτή τη μορφή;)

```
def allAbove(imgs: List[Image]): Image =
  imgs match {
    case Nil => Image.empty
    case hd :: tl => hd above allAbove(tl)
```

```
  }
```

Η τελική εικόνα:

```
allAbove((3 to 33 by 2).toList map { sides =>
  allBeside((1 to sides/2).toList map { skip =>
    style(star(sides, skip, 20), 360.degrees * skip
/ sides)
  })
})
```

Return to the exercise

15.9 Άλγεβρα Turtle και Αλγεβρικοί Τύποι Δεδομένων

15.9.1 Solution to: Πολύγωνα

Παρακάτω μπορείτε να δείτε την δική μας λύση. Είναι μία δομημένη αναδρομή με φυσικούς αριθμούς. Η γωνία είναι ίδια με την γωνία περιστροφής που χρησιμοποιήσαμε για την κατασκευή πολυγώνων με πολικές συντεταγμένες, σε προηγούμενο κεφάλαιο. Όμώς εδώ, η διαδικασία σχεδιασμού είναι εντελώς διαφορετική.

```
def polygon(sides: Int, sideLength: Double): Image =
 {
  val rotation = Angle.one / sides
  def iter(n: Int): List[Instruction] =
    n match {
      case 0 => Nil
      case n => turn(rotation) :: forward(sideLength
) :: iter(n-1)
    }

  Turtle.draw(iter(sides))
}
```

Return to the exercise

15.9.2 Solution to: Το Τετράγωνο Σπειροειδές

Παρακάτω μπορείτε να δείτε τα βασικά στοιχεία για τον σχεδιασμό του τετράγωνου σπειροειδούς:

- κάθε στροφή είναι λίγο μικρότερη από 90 μοίρες
- κάθε βήμα μπροστά είναι λίγο μεγαλύτερο από το προηγούμενο

Μόλις κατανοήσετε τα παραπάνω, θα δείτε ότι η δομή του κώδικα είναι σχεδόν ίδια με αυτή του σχεδιασμού ενός πολυγώνου. Δείτε την λύση μας.

```
def squareSpiral(steps: Int, distance: Double, angle
: Angle, increment: Double): Image = {
  def iter(n: Int, distance: Double): List[Instructi
on] = {
    n match {
      case 0 => Nil
      case n => forward(distance) :: turn(angle) ::
iter(steps-1, distance + increment)
    }
  }

  Turtle.draw(iter(steps, distance))
}
// squareSpiral: (steps: Int, distance: Double, angl
e: doodle.core.Angle, increment: Double)doodle.core.
Image
```

Return to the exercise

15.9.3 Solution to: Γραφικά Turtle vs Πολικές Συντεταγμένες

Κάθε πλευρά του πολυγώνου απαιτεί δύο εντολές turtle: μία `forward` και μία `turn`. Άρα, για να ζωγραφίσουμε ένα πεντάγωνο θα χρειαστούν δέκα εντολές. Γενικότερα `n` πλευρές θα χρειαστούν `2n` εντολές. Χρησιμοποιώντας την `map` δεν μπορούμε να αλλάξουμε τον αριθμό των στοιχείων μίας λίστας. Δηλαδή, το να έχουμε μία map από `1` ως `n`, όπως κάναμε παραπάνω, δεν θα λειτουργήσει σ' αυτή την περίπτωση. Θα μπορούσαμε να έχουμε ένα map για `1 ως (n*2)` και στους περιττούς αριθμούς να κινείται μπροστά ενώ στους άρτιους να στρίβει αλλά αυτός ο τρόπος δεν είναι πολύ κομψός. Θα ήταν πολύ βολικό να είχαμε μία δομή σαν την map, που μας επιτρέπει την αλλαγή του πλήθους των στοιχείων μίας λίστας.

Return to the exercise

15.9.4 Solution to: Δομές Διακλαδώσεων

Όπως μας λενε οι τυποι, αυτή τη φορα η `map` δεν είναι η καταλληλη λύση. Θυμηθείτε την εξισωση τύπων για την `map`

```
List[A] map (A => B) = List[B]
```

Αν

- έχουμε μία `List[Instruction]` και
- χρησιμοποιήσουμε την `map` σε μία συνάρτηση `Instruction => List[Instruction]`, τότε
- το αποτέλεσμα που θα πάρουμε θα είναι `List[List[Instruction]]`

όπως μπορούμε να δούμε και από την εξίσωση τύπων.

Το turtle όμως δεν ξέρει πώς να σχεδιάσει την `List[List[Instruction]]`, οπότε το παραπάνω δεν θα δουλέψει.

Return to the exercise

15.9.5 Solution to: Όλα Διπλά

Υπάρχουν δύο σημεία που πρέπει να προσέξουμε:

- να αναγνωρίσουμε τον τρόπο με τον οποίο θα χρησιμοποιήσουμε

την `flatMap` και
- να θυμηθούμε πώς χρησιμοποιούνται οι μεταβλητές τύπου.

```
def double[A](in: List[A]): List[A] =
  in.flatMap { x => List(x, x) }
```

Return to the exercise

15.9.6 Solution to: Ή Τίποτα

Θα μπορούσαμε πολύ εύκολα να γράψουμε αυτή την μεθοδο ως

```
def nothing[A](in: List[A]): List[A] =
  List() // ή List.empty ή Nil
```

αλλά το νόημα της άσκησης είναι να εξοικειωθούμε με την χρήση της `flatMap`. Με την `flatMap` μπορούμε να γράψουμε την μέθοδο όπως παρακάτω

```
def nothing[A](in: List[A]): List[A] =
  in.flatMap { x => List.empty }
```

Return to the exercise

15.9.7 Solution to: Ξαναγράφοντας τους Κανόνες

Υπάρχουν δύο σημεία που πρέπει να προσέξουμε:

- να αναγνωρίσουμε ότι πρέπει να χρησιμοποιήσουμε την `flatMap`, για τους λόγους που αναφέρθηκαν παραπάνω και
- να καταλάβουμε ότι πρέπει να καλέσουμε την `rewrite` αναδρομικά, για την επεξεργασία των περιεχομένων της διακλάδωσης.

Ένα παράδειγμα δομημένης αναδρομής σε λίγο πιο περίπλοκη μορφή από αυτήν που έχουμε συνηθίσει.

```
def rewrite(instructions: List[Instruction], rule: I
nstruction => List[Instruction]): List[Instruction]
```

```
=
  instructions.flatMap { i =>
    i match {
      case Branch(i) =>
        List(branch(rewrite(i, rule):_*))
      case other =>
        rule(other)
    }
  }
```

Return to the exercise

15.9.8 Solution to: Το Δικό σας L-System

Είναι μία απλή δομημένη αναδρομή με φυσικούς αριθμούς όπου η `rewrite` κάνει την δύσκολη δουλειά.

```
def iterate(steps: Int,
            seed: List[Instruction],
            rule: Instruction => List[Instruction]):
 List[Instruction] =
  steps match {
    case 0 => seed
    case n => iterate(n - 1, rewrite(seed, rule), ru
le)
  }
```

Return to the exercise

15.9.9 Solution to: Επίπεδο Πολύγωνο

Χρησιμοποιώντας την `flatMap` μπορούμε να κάνουμε τον κώδικα πιο συμπαγή από ότι αν χρησιμοποιούσαμε δομημένη αναδρομή όπως παλαιότερα.

```
def polygon(sides: Int, sideLength: Double): Image =
 {
  val rotation = Angle.one / sides
```

```
  Turtle.draw((1 to sides).toList.flatMap { n =>
    List(turn(rotation), forward(sideLength))
  })
}
```

Return to the exercise

15.9.10 Solution to: Επίπεδο Σπειροειδές

Και πάλι, το αποτέλεσμα θα είναι πιο συμπαγές από ότι ήταν την προηγούμενη φορά. Είναι πιο εύκολη η ανάγνωση της μεθόδου; Εμάς μας φαίνεται περίπου το ίδιο. Πιστεύουμε ότι η κατανόηση είναι θέμα εξοικείωσης και ελπίζουμε ότι έχουμε φτάσει σε καλό σημείο με την `flatMap`.

```
def squareSpiral(steps: Int, distance: Double, angle
: Angle, increment: Double): Image = {
  Turtle.draw((1 to steps).toList.flatMap { n =>
   List(forward(distance + (n * increment)), turn(an
gle))
  })
}
```

Return to the exercise

15.10 Σύνθεση Αναπαραγωγικής Τέχνης

15.10.1 Solution to: Αναπαραγωγική Τέχνη

Η παραγωγή τυχαίων αριθμών με αυτόν το τρόπο απειλεί την έννοια της αντικατάστασης. Όπως θυμάστε από προηγούμενο κεφάλαιο, η μέθοδος της αντικατάστασης έλεγε ότι όπου βλέπουμε μία έκφραση θα πρέπει να μπορούμε να αντικαταστήσουμε την τιμή με την οποία αξιολογείται, χωρίς να αλλάξουμε το νόημα του προγράμματος. Επομένως, αυτό σημαίνει ότι

οι παρακάτω κώδικες

```
val result1 = randomAngle
// result1: doodle.core.Angle = Angle(4.800086372129
377)

val result2 = randomAngle
// result2: doodle.core.Angle = Angle(3.180264537595
8147)
```

και

```
val result1 = randomAngle
// result1: doodle.core.Angle = Angle(4.572027113527
563)

val result2 = result1
// result2: doodle.core.Angle = Angle(4.572027113527
563)
```

θα έπρεπε να είναι το ίδιο πρόγραμμα, κάτι που φανερά δεν ισχύει.

Return to the exercise

15.10.2 Solution to: Τυχαιότητα και Τυχαιότητα

Η έξοδος της `programOne` είναι τρεις διαφορετικοί κύκλοι σε μία γραμμή ενώ η έξοδος της `programTwo` επαναλαμβάνει τον ίδιο κύκλο τρεις φορές. Η `circles` αντιπροσωπεύει ένα πρόγραμμα που παράγει μία εικόνα με τυχαία χρωματισμένους ομόκεντρους κύκλους. Θυμηθείτε ότι η `map` αναπαριστά έναν ντετερμινιστικό μετασχηματισμό. Η έξοδος του `programTwo` πρέπει να είναι ο ίδιος κύκλος που επαναλαμβάνεται τρεις φορές χωρίς να γίνονται τυχαίες επιλογές. Στο `programOne` συγχωνεύουμε το `circle` με τον εαυτό του τρεις φορές. Ίσως πιστεύετε ότι έτσι θα έπρεπε να υπάρχει μόνο μία τυχαία εικόνα που επαναλαμβάνεται τρεις φορές και όχι τρεις διαφορετικές αλλά θυμηθείτε ότι η `Random` διατηρεί την

αντικατάσταση. Μπορούμε να γράψουμε το `programOne` όπως παρακάτω και να πάρουμε το ίδιο αποτέλεσμα

```
val programOne =
  randomConcentricCircles(5, 10) flatMap { c1 =>
    randomConcentricCircles(5, 10) flatMap { c2 =>
      randomConcentricCircles(5, 10) map { c3 =>
        c1 beside c2 beside c3
      }
    }
  }
// programOne: cats.free.Free[doodle.random.RandomOp
,doodle.core.Image] = Free(...)
```

Έτσι φαίνεται πιο καθαρά ότι φτιάχνουμε τρεις διαφορετικούς κύκλους.

Return to the exercise

15.10.3 Solution to: Χρωματιστά Κουτιά

Αυτός ο κώδικας έχει ακριβώς την ίδια μορφή με την `randomConcentricCircles`.

```
val randomAngle: Random[Angle] =
  Random.double.map(x => x.turns)
// randomAngle: doodle.random.Random[doodle.core.Ang
le] = Free(...)

val randomColor: Random[Color] =
  randomAngle map (hue => Color.hsl(hue, 0.7.normali
zed, 0.7.normalized))
// randomColor: doodle.random.Random[doodle.core.Col
or] = Free(...)

def coloredRectangle(color: Color): Image =
  rectangle(20, 20) fillColor color
// coloredRectangle: (color: doodle.core.Color)doodl
e.core.Image
```

```
def randomColorBoxes(count: Int): Random[Image] =
  count match {
    case 0 => randomColor map { c => coloredRectangl
e(c) }
    case n =>
      val box = randomColor map { c => coloredRectan
gle(c) }
      val boxes = randomColorBoxes(n-1)
      box flatMap { b =>
        boxes map { bs => b beside bs }
      }
  }
// randomColorBoxes: (count: Int)doodle.random.Rando
m[doodle.core.Image]
```

Return to the exercise

15.10.4 Solution to: Συστήματα Σωματιδίων

Το παρακάτω θα κάνει αυτό που θέλουμε. Αν θέλετε, μπορείτε να δημιουργήσετε μία πιο περίπλοκη (και ενδιαφέρουσα) κατανομή στην αρχική θέση.

```
val start = Random.always(Point.zero)
```

Return to the exercise

15.10.5 Solution to: Συστήματα Σωματιδίων Part 2

Επιλέξαμε την χρήση κανονικά κατανεμημένου θορύβου που είναι ίδιος και στις δύο κατευθύνσεις. Μία αλλαγή στον θόρυβο θα άλλαζε και το σχήμα του αποτελέσματος—αξίζει να “παίξετε” με διάφορες ρυθμίσεις.

```
def step(current: Point): Random[Point] = {
  val drift = Point(current.x + 10, current.y)
```

```
  val noise =
    Random.normal(0.0, 5.0) flatMap { x =>
      Random.normal(0.0, 5.0) map { y =>
        Vec(x, y)
      }
    }

  noise.map(vec => drift + vec)
}
```

Return to the exercise

15.10.6 Solution to: Συστήματα Σωματιδίων Part 3

Στον ορισμό μας για την `render`, δείξαμε πώς μπορούμε να χρησιμοποιήσουμε πληροφορίες από το σημείο ώστε να αλλάξουμε το σχήμα με ενδιαφέρον τρόπο.

Ο ορισμός της `walk` είναι μία δομημένη αναδρομή με φυσικούς αριθμούς και έναν εσωτερικό συσσωρευτή και την `flatMap`.

```
def render(point: Point): Image = {
  val length = (point - Point.zero).length
  val sides = (length / 20).toInt + 3
  val hue = (length / 200).turns
  val color = Color.hsl(hue, 0.7.normalized, 0.5.nor
malized)
  Image.
    star(sides, 5, 3, 0.degrees).
    noFill.
    lineColor(color).
    at(point.toVec)
}

def walk(steps: Int): Random[Image] = {
  def loop(count: Int, current: Point, image: Image)
```

```
: Random[Image] = {
    count match {
      case 0 => Random.always(image on render(curren
t))
      case n =>
        val next = step(current)
        next flatMap { pt =>
          loop(count - 1, pt, image on render(curren
t))
        }
    }
  }

  start flatMap { pt => loop(steps, pt, Image.empty)
 }
}
```

Return to the exercise

15.10.7 Solution to: Συστήματα Σωματιδίων Part 4

Για άλλη μία φορά έχουμε δομημένη αναδρομή με φυσικούς αριθμούς. Σε αντίθεση με την `walk`, η αναδρομή γίνεται στην `map` και όχι στην `flatMap`. Αυτό συμβαίνει επειδή η `particleSystem` δεν προσθέτει νέες τυχαίες επιλογές.

```
def particleSystem(particles: Int, steps: Int): Rand
om[Image] = {
  particles match {
    case 0 => Random.always(Image.empty)
    case n => walk(steps) flatMap { img1 =>
      particleSystem(n-1, steps) map { img2 =>
        img1 on img2
      }
    }
  }
```

```
}
```

Return to the exercise

15.10.8 Solution to: Τυχαίες Αλλαγές

Θα μπορούσαμε να κάνουμε τις `walk` `start` και `render`, παραμέτρους της `particleSystem` και τις `start` και `render` παραμέτρους της `walk`.

Return to the exercise

15.10.9 Solution to: Τυχαίες Αλλαγές Part 2

Αν προσθέσουμε παραμέτρους με σωστά ονόματα και τύπους, οι αλλαγές στον κώδικα θα είναι ελάχιστες. Είναι σαν να κάνουμε το αντίθετο της αντικατάστασης—βγάζουμε ολόκληρες αναπαραστάσεις από τον κωδικά μας και τις αντικαθιστούμε με παραμέτρους μεθόδων. Δείτε παρακάτω

```
def walk(
  steps: Int,
  start: Random[Point],
  render: Point => Image
): Random[Image] = {
  def loop(count: Int, current: Point, image: Image)
: Random[Image] = {
    count match {
      case 0 => Random.always(image on render(curren
t))
      case n =>
        val next = step(current)
        next flatMap { pt =>
          loop(count - 1, pt, image on render(curren
t))
        }
    }
  }
```

```
  start flatMap { pt => loop(steps, pt, Image.empty)
 }
}

def particleSystem(
  particles: Int,
  steps: Int,
  start: Random[Point],
  render: Point => Image,
  walk: (Int, Random[Point], Point => Image) => Rand
om[Image]
): Random[Image] = {
  particles match {
    case 0 => Random.always(Image.empty)
    case n => walk(steps, start, render) flatMap { i
mg1 =>
      particleSystem(n-1, steps, start, render, walk
) map { img2 =>
        img1 on img2
      }
    }
  }
}
```

Return to the exercise

15.10.10 Solution to: Διασκορπισμένα Σχέδια

Αυτή η άσκηση είναι ένα καλό παράδειγμα σύνθεσης τυχαίων αριθμών.

```
def makePoint(x: Random[Double], y: Random[Double]):
 Random[Point] =
  for {
    theX <- x
    theY <- y
  } yield Point.cartesian(theX, theY)
```

```
// makePoint: (x: doodle.random.Random[Double], y: doodle.random.Random[Double])doodle.random.Random[doodle.core.Point]
```

Return to the exercise

15.10.11 Solution to: Διασκορπισμένα Σχέδια Part 2

Κάτι σαν το παρακάτω θα πρέπει να δουλέψει.

```
val normal = Random.normal(50, 15)
val normal2D = makePoint(normal, normal)

val data = (1 to 1000).toList.map(_ => normal2D)
```

Return to the exercise

15.10.12 Solution to: Διασκορπισμένα Σχέδια Part 3

Μπορούμε να μετατρέψουμε ένα σημείο σε εικόνα χρησιμοποιώντας την μέθοδο `point` που υπάρχει παρακάτω. Παρατηρήστε ότι κάναμε το κάθε σημείο του σχεδίου σχεδόν διαφανές—έτσι γίνεται ευκολότερο να δει κανείς πού συγκεντρώνονται τα περισσότερα σημεία.

```
def point(loc: Point): Image =
  circle(2).fillColor(Color.cadetBlue.alpha(0.3.normalized)).noLine.at(loc.toVec)
```

Η μετατροπή των λιστών είναι απλώς θέμα κλήσης της `map`.

```
val points = data.map(r => r.map(point _))
```

Return to the exercise

15.10.13 Solution to: Διασκορπισμένα Σχέδια Part 4

Ίσως αναγνωρίσετε την παρακάτω μορφή. Είναι σαν αυτή που χρησιμοποιήσαμε στη `allOn` αλλά προσθέσαμε και την `flatMap`, η οποία χρησιμοποιείται και στην `randomConcentricCircles` (καθώς και σε πολλά άλλα παραδείγματα).

```
def allOn(points: List[Random[Image]]): Random[Image
] =
  points match {
    case Nil => Random.always(Image.empty)
    case img :: imgs =>
      for {
        i  <- img
        is <- allOn(imgs)
      } yield (i on is)
  }
```

Return to the exercise

15.10.14 Solution to: Διασκορπισμένα Σχέδια Part 5

Για να γίνει αυτό πρέπει απλώς να καλέσουμε μερικές μεθόδους και να χρησιμοποιήσουμε τιμές που έχουμε ήδη ορίσει.

```
val plot = allOn(points)
```

Return to the exercise

15.10.15 Solution to: Παραμετρικός Θόρυβος

Παρακάτω μπορείτε να δείτε την λύση μας. Έχουμε δει παρόμοιο κώδικα στην υλοποίηση των διασκορπισμένων σημείων.

```
def perturb(point: Point): Random[Point] =
  for {
    x <- Random.normal(0, 10)
    y <- Random.normal(0, 10)
  } yield Point.cartesian(point.x + x, point.y + y)
```

Return to the exercise

15.10.16 Solution to: Παραμετρικός Θόρυβος Part 2

Με χρήση της `andThen` παίρνουμε έναν ωραίο και συμμαζεμένο κώδικα.

```
def perturbedRose(k: Int): Angle => Random[Point] =
  rose(k) andThen perturb
```

Return to the exercise

15.10.17 Solution to: Παραμετρικός Θόρυβος Part 3

Παρακάτω μπορείτε να δείτε τον κώδικα που χρησιμοποιήσαμε για να δημιουργήσουμε την εικόνα fig. 55. Είναι λίγο μεγαλύτερος από όσα έχουμε δει ως τώρα. Θα πρέπει να καταλαβαίνετε πώς δουλεύουν τα στοιχεία που τον αποτελούν.

```
object ParametricNoise {
  def rose(k: Int): Angle => Point =
    (angle: Angle) => {
      Point.cartesian((angle * k).cos * angle.cos, (
angle * k).cos * angle.sin)
    }

  def scale(factor: Double): Point => Point =
    (pt: Point) => {
      Point.polar(pt.r * factor, pt.angle)
```

```
    }

  def perturb(point: Point): Random[Point] =
    for {
      x <- Random.normal(0, 10)
      y <- Random.normal(0, 10)
    } yield Point.cartesian(point.x + x, point.y + y
)

  def smoke(r: Normalized): Random[Image] = {
    val alpha = Random.normal(0.5, 0.1) map (a => a.
normalized)
    val hue = Random.double.map(h => (h * 0.1).turns
)
    val saturation = Random.double.map(s => (s * 0.8
).normalized)
    val lightness = Random.normal(0.4, 0.1) map (a =
> a.normalized)
    val color =
      for {
        h <- hue
        s <- saturation
        l <- lightness
        a <- alpha
      } yield Color.hsla(h, s, l, a)
    val c = Random.normal(5, 5) map (r => circle(r))

    for {
      circle <- c
      line   <- color
    } yield circle.lineColor(line).noFill
  }

  def point(
    position: Angle => Point,
    scale: Point => Point,
    perturb: Point => Random[Point],
```

```
    image: Normalized => Random[Image],
    rotation: Angle
  ): Angle => Random[Image] = {
    (angle: Angle) => {
      val pt = position(angle)
      val scaledPt = scale(pt)
      val perturbed = perturb(scaledPt)

      val r = pt.r.normalized
      val img = image(r)

      for {
        i  <- img
        pt <- perturbed
      } yield (i at pt.toVec.rotate(rotation))
    }
  }

  def iterate(step: Angle): (Angle => Random[Image])
=> Random[Image] = {
    (point: Angle => Random[Image]) => {
      def iter(angle: Angle): Random[Image] = {
        if(angle > Angle.one)
          Random.always(Image.empty)
        else
          for {
            p  <- point(angle)
            ps <- iter(angle + step)
          } yield (p on ps)
      }

      iter(Angle.zero)
    }
  }

  val image: Random[Image] = {
    val pts =
```

```
      for(i <- 28 to 360 by 39) yield {
        iterate(1.degrees){
          point(
            rose(5),
            scale(i),
            perturb _,
            smoke _,
            i.degrees
          )
        }
      }
    val picture = pts.foldLeft(Random.always(Image.e
mpty)){ (accum, img) =>
      for {
        a <- accum
        i <- img
      } yield (a on i)
    }
    val background = (rectangle(650, 650) fillColor
Color.black)

    picture map { _ on background }
  }
}
```

Return to the exercise

15.11 Δικοί μας Αλγεβρικοί Τύποι Δεδομένων

15.11.1 Solution to: Στοιχεία Μονοπατιών

Ο `PathElement` είναι από μόνος του τύπος αθροίσματος: - μία `MoveTo`, ή - μία `LineTo` ή - μία `CurveTo`.

Μία `MoveTo` είναι τύπος γινομένου που κρατάει αποθηκευμένο ένα σημείο

(εκεί που θα μετακινηθεί).

Μία `LineTo` είναι τύπος γινομένου που κρατάει αποθηκευμένο ένα σημείο (το τελευταίο σημείο της γραμμής).

Μία `CurveTo` είναι τύπος γινομένου που κρατάει αποθηκευμένα τρία σημεία: δύο σημεία ελέγχου και το τελευταίο σημείο της γραμμής.

Return to the exercise

15.11.2 Solution to: Εντελώς Turtles

Ένας τύπος `Instruction` είναι: - μία `Forward`, ή - μία `Turn`, ή - μία `Branch`, ή - μία `NoOp`

Άρα ο `Instruction` είναι τύπος αθροίσματος. Οι `Forward`, `Turn` και `Branch` έχουν όλες τύπο γινομένου.

Η `Forward` έχει αποθηκευμένη μία απόσταση, τύπου `Double`.

Η `Turn` έχει αποθηκευμένη μία γωνία, τύπου `Angle`.

Η `Branch` έχει αποθηκευμένη μία `List[Instruction]`—άρα ο τύπος `Instruction` ορίζεται σε σχέση με τον εαυτό του, ακριβώς όπως κάνει και μία λίστα.

Η `NoOp` δεν έχει τίποτα αποθηκευμένο.

Return to the exercise

15.11.3 Solution to: Άσκηση

Μπορούμε να μετατρέψουμε απευθείας την περιγραφή σε κώδικα, χρησιμοποιώντας τα παρακάτω.

```
sealed abstract class Instruction extends Product wi
th Serializable
// defined class Instruction

final case class Forward(distance: Double) extends I
nstruction
// defined class Forward
```

```
final case class Turn(angle: Angle) extends Instruct
ion
// defined class Turn

final case class Branch(instructions: List[Instructi
on]) extends Instruction
// defined class Branch

final case class NoOp() extends Instruction
// defined class NoOp
```

Return to the exercise

15.11.4 Solution to: Χτίστε το Δικό σας Turtle

Ο παρακάτω είναι ένας τύπος γινομένου.

```
final case class TurtleState(at: Vec, heading: Angle
)
// defined class TurtleState
```

Return to the exercise

15.11.5 Solution to: Χτίστε το Δικό σας Turtle Part 2

Ο πυρήνας της λύσης είναι μία δομημένη αναδρομή αλλά οι λεπτομέρειες αυτής της περίπτωσης είναι πιο περίπλοκες από όσο έχουμε συνηθίσει μέχρι τώρα. Πρέπει να δημιουργήσουμε τα στοιχεία του μονοπατιού αλλά και να ενημερώσουμε το ‘state’ (κατάσταση).

```
def process(state: TurtleState, instruction: Instruc
tion): (TurtleState, List[PathElement]) = {
  import PathElement._
```

```
  instruction match {
    case Forward(d) =>
      val nowAt = state.at + Vec.polar(d, state.head
ing)
      val element = lineTo(nowAt.toPoint)

      (state.copy(at = nowAt), List(element))
    case Turn(a) =>
      val nowHeading = state.heading + a

      (state.copy(heading = nowHeading), List())
    case Branch(i) =>
      // Ignoring for now
      (state, List())
    case NoOp() =>
      (state, List())
  }
}
// process: (state: TurtleState, instruction: Instru
ction)(TurtleState, List[doodle.core.PathElement])
```

Return to the exercise

15.11.6 Solution to: Χτίστε το Δικό σας Turtle Part 3

```
def iterate(state: TurtleState, instructions: List[I
nstruction]): List[PathElement] =
  instructions match {
    case Nil =>
      Nil
    case i :: is =>
      val (newState, elements) = process(state, i)
      elements ++ iterate(newState, is)
  }
// iterate: (state: TurtleState, instructions: List[
```

```
Instruction])List[doodle.core.PathElement]
```

Return to the exercise

15.11.7 Solution to: Χτίστε το Δικό σας Turtle Part 4

Παρακάτω μπορείτε να δείτε το turtle ολοκληρωμένο.

```
object Turtle {
  def draw(instructions: List[Instruction]): Image =
 {
    def iterate(state: TurtleState, instructions: Li
st[Instruction]): List[PathElement] =
      instructions match {
        case Nil =>
          Nil
        case i :: is =>
          val (newState, elements) = process(state,
i)
          elements ++ iterate(newState, is)
      }

    def process(state: TurtleState, instruction: Ins
truction): (TurtleState, List[PathElement]) = {
      import PathElement._

      instruction match {
        case Forward(d) =>
          val nowAt = state.at + Vec.polar(d, state.
heading)
          val element = lineTo(nowAt.toPoint)

          (state.copy(at = nowAt), List(element))
        case Turn(a) =>
          val nowHeading = state.heading + a
```

```
          (state.copy(heading = nowHeading), List())
        case Branch(is) =>
         val branchedElements = iterate(state, is)

         (state, moveTo(state.at.toPoint) :: branche
dElements)
        case NoOp() =>
          (state, List())
      }
    }

    openPath(iterate(TurtleState(Vec.zero, Angle.zer
o), instructions))
  }
}
// defined object Turtle
```

Return to the exercise