ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ
ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΣΥΝΑΡΤΗΣΙΑΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΤΗΝ JAVASCRIPT

FUNCTIONAL PROGRAMMING IN JAVASCRIPT

Διπλωματική Εργασία

του

Σαφαρίδη Φέλιξ

Θεσσαλονίκη, Ιούνιος 2019

ΣΥΝΑΡΤΗΣΙΑΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΤΗΝ JAVASCRIPT

FUNCTIONAL PROGRAMMING IN JAVASCRIPT

Σαφαρίδης Φέλιξ

Πτυχίο Μηχανικού Τηλεπικοινωνιών-Ηλεκτρονικών, Σχολή Ικάρων, 2013

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 21/06/2019

....

Σαφαρίδης Φέλιξ
.....

Περίληψη

Αντικείμενο της παρούσας διπλωματικής εργασίας αποτελεί η ανάλυση και η μελέτη των αρχών του συναρτησιακού προγραμματισμού γενικά καθώς και ο βαθμός κατά τον οποίο δύναται η υιοθέτηση αυτών από το ειδικό προγραμματιστικό περιβάλλον που ορίζει η προγραμματιστική γλώσσα της JavaScript (ES2015 και παραπέρα). Θα εξεταστούν σε βάθος μία πληθώρα τεχνικών που πηγάζουν από τη συναρτησιακή μαθηματική θεωρία και θα γίνει ορατός στον αναγνώστη ο τρόπος με τον οποίο μπορούμε να εκμεταλλευτούμε εγγενείς μαθηματικές λειτουργικότητες με σκοπό την παραγωγή εύρωστου, επεκτάσιμου και αρθρωτού κώδικα. Αρχικά θα γίνει μία ιστορική αναδρομή στην οποία αποτυπώνεται η θεωρία που διέπει τον λογισμό λάμδα, το βασικό μοντέλο πάνω στο οποίο στηρίζεται ο συναρτησιακός προγραμματισμός. Στη συνέχεια θα γίνει αναφορά στη γλώσσα της JavaScript και το ειδικό προγραμματιστικό πλαίσιο που επιτρέπει τη συνύπαρξη συναρτησιακού και αντικειμενοστρεφή κώδικα πάνω στη βασική αρχή ότι ο συναρτησιακός προγραμματισμός και η αντικειμενοστρεφής προσέγγιση είναι έννοιες ορθογώνιες και σε καμία περίπτωση αμοιβαία αποκλειόμενες. Επιπρόσθετα θα παρουσιαστούν τεχνικές που επιτρέπουν το σχηματισμό δηλωτικών προγραμμάτων κατά τις αρχές που διέπουν τη συναρτησιακή προσέγγιση και έρχονται σε αντίθεση με το προστακτικό μοντέλο γραφής στο οποίο είναι συνηθισμένοι οι προγραμματιστές. Η επίλυση προβλημάτων θα παρουσιαστεί ως μία αυστηρή διαδικασία διάσπασης-σύνθεσης. Τέλος θα δοθούν κατευθυντήριες οδηγίες για την πρόσβαση στην προγραμματιστική βιβλιοθήκη που θα αναπτυχθεί στα πλαίσια της παρούσας διπλωματικής εργασίας, μίας βιβλιοθήκης που σκοπό θα έχει την αρωγή του προγραμματιστή στην υιοθέτηση έτοιμων συναρτησιακών υλοποιήσεων που θα μπορεί να αξιοποιήσει κατά την επίλυση ενός προβλήματος στο περιβάλλον της JavaScript.

Λέξεις-κλειδιά:

συναρτησιακός προγραμματισμός, λογισμός λάμδα, δηλωτικός, προστακτικός, JavaScript, συναρτήσεις υψηλής τάξης, αφαίρεση, αναδρομή, δίπλωση, συνάρτηση, απεικόνιση, αλυσίδωση, σωλήνωση, σύνθεση, σύνθεση Kleisli, σύστημα τύπων, θεωρία κατηγοριών, μορφισμός, συναρτήτης, εφαρμοστής, μονάδα, μονοειδή, αντικείμενο, δομή

Abstract

The aim of the present thesis is to analyze and feature the functional programming concepts on general, as well as highlight those under the wealthy context that constitutes the programming language of JavaScript (ES2015 and beyond). We will introduce an in-depth analysis of a plethora of functional programming techniques that are based on strongly structured mathematical principles, and we will show how to apply those principles in order to produce highly robust, extensible and modular code. We will examine the formal system of computation, namely Lambda Calculus, on which functional programming has its roots on. As it is established, this formal system encourages a declarative style of programming as opposed to the imperative style that many of us are strongly coupled with, since our first steps in the computer programming field. The core of the language (JavaScript), as well as computation principles, does not constrain us of using both functional programming approach and object orientation as a mean to produce a solution to a problem because of the fact that those concepts are orthogonal. In fact merging them enables the programmer with a new perception on how he can manipulate the data and how he can produce strongly decoupled functionalities that can be combined to produce new ones. Finally, the programmer approaches a stage in which he is able to decide what functions to apply in the form of a transformation on his data or objects to achieve a solution. Lastly, we are going to build a library, wrapping all the concepts of functional programming in a form of higher order functions that will aid programmers to embrace the functional approach during development.

Keywords:

functional programming, lambda calculus, declarative, imperative, JavaScript, higher order functions, abstraction, recursion, folding, function, mapping, chaining, pipelining, composition, Kleisli composition, type system, category theory, morphism, functor, applicative, monad, monoid, object, structure

Πρόλογος-Ευχαριστίες

Ευχαριστώ εγκάρδια τον επιβλέποντα καθηγητή της παρούσας διπλωματικής εργασίας κ. Κασκάλη Χ. Θεόδωρο, τα μαθήματα του οποίου μου γέννησαν ενθουσιασμό για τον μαγικό κόσμο της JavaScript. Η παρακίνηση του για την αξιοποίηση μίας διαφορετικής αντίληψης κατά το στάδιο της ανάπτυξης λογισμικού στην JavaScript, είχε ως αποτέλεσμα όχι μόνο τη δημιουργία της παρούσας εργασίας, άλλα και τη γνωριμία μου με έννοιες που παρουσιάζουν απότομη καμπύλη μάθησης και δίχως την παροχή κατάλληλου κινήτρου, συνήθως αποφεύγονται. Το κίνητρο, στην παρούσα περίπτωση, αλλά και σε κάθε περίπτωση, ήταν, και θα πρέπει να είναι πάντα, ο ενθουσιασμός. Κύριε Καθηγητά ευχαριστώ.

Περιεχόμενα

Πίνακας Σχημάτων

1 Εισαγωγή

1.1 Ορισμός

Στην επιστήμη των υπολογιστών ο συναρτησιακός προγραμματισμός (functional programming) αποτελεί ένα προγραμματιστικό μοντέλο (όπως για παράδειγμα ο αντικειμενοστρεφής προγραμματισμός) στο οποίο τη δομική μονάδα συγκρότησης των προγραμμάτων αποτελούν οι συναρτήσεις. Οι συναρτήσεις αυτές ακολουθούν κανόνες οι οποίες επιτρέπουν στο μοντέλο να ταυτιστεί πλήρως με μαθηματικές θεωρίες (όπως η θεωρία συνόλων, set theory) και ως εκ τούτου να αποκτήσουν ένα πανίσχυρο θεωρητικό υπόβαθρο ικανό να επιλύσει πολύπλοκα υπολογιστικά προβλήματα (Hughes, 1990).

Λόγω της μαθηματικής προέλευσης του μοντέλου, ο συναρτησιακός προγραμματισμός ακολουθεί έναν δηλωτικό τρόπο σύνταξης (declarative style of programming) σε αντίθεση με το γνωστό προστακτικό μοντέλο (imperative style of proramming) που χρησιμοποιείται κατά κόρον από προγραμματιστές, σε δημοφιλής γλώσσες όπως η C, Java κτλ (PopularitY of Programming Language, 2019). Πρακτικά αυτό σημαίνει ότι τον προγραμματιστή απασχολεί εννοιολογικά το τι κάνει μία μονάδα υπολογισμού, και όχι πώς θα το κάνει, δίνοντας του έτσι τη δυνατότητα μέσω δηλώσεων να συγκροτήσει τη λογική του προγράμματος που τελικώς θα αποτελέσει λύση στο πρόβλημα που καλείται να αντιμετωπίσει.

1.2 Ιστορική αναδρομή

Ο συναρτησιακός προγραμματισμός έχει τις ρίζες του στον λογισμό λάμδα (lambda calculus), ένα θεωρητικό πλαίσιο το οποίο περιγράφει τη δομική μονάδα της συνάρτησης και τις μεθόδους αποτιμήσεων αυτής. Αν και αποτελεί καθαρά μία αφηρημένη μαθηματική έννοια, αποτελεί τη βάση για όλες τις συναρτησιακές γλώσσες προγραμματισμού στις μέρες μας. Πιο συγκεκριμένα επινοήθηκε από τον Ιταλό μαθηματικό Alonzo Church στη δεκαετία του 1930 και εδραιώθηκε ως ένα καθολικό μοντέλο υπολογισμού (universal model of computation) το οποίο είναι ικανό να προσομοιώσει οποιαδήποτε μηχανή Turing μονής ταινίας (single-tape Turing Machine). Αυτό σημαίνει ότι ο λογισμός λάμδα επαρκεί για να υλοποιήσει οποιοδήποτε προγραμματιστικό αλγόριθμο.

Η πρώτη γλώσσα συναρτησιακού προγραμματισμού ή πιο σωστά συναρτησιακού τύπου (functional-flavored) ήταν η Lisp, η οποία δημιουργήθηκε στα τέλη του 1950 από τον John McCarthy στο MIT. Όμως στη συνέχεια και καθώς αναπτύχθηκαν άλλα προγραμματιστικά μοντέλα τα οποία υιοθέτησε η Lisp, ο τίτλος της πρώτης συναρτησιακής γλώσσας προγραμματισμού αφαιρέθηκε και τη θέση της πήρε η IPL (Information Processing Language). Η IPL αποτελούσε μία γλώσσα επιπέδου assembly στην οποία ο κώδικας μπορούσε να χρησιμοποιηθεί ως δεδομένο και συνεπώς να χρησιμοποιηθεί ως έκφραση (expression).

Καθώς περνούσαν τα χρόνια ο συναρτησιακός προγραμματισμός αποτελούσε κυρίως αντικείμενο μελέτης και έρευνας της ακαδημαϊκής κοινότητας. Ο πρώτος σταθμός που αποτέλεσε προπομπό για μία διαφορετική αντιμετώπιση του συναρτησιακού προγραμματισμού ήταν η διάλεξη του John Backus στα βραβεία Turing (Turing Awards) με τίτλο "Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs". Στη διάλεξη αυτή ο John Backus υποστηρίζει ότι οι γλώσσες προγραμματισμού που ακολουθούν το μοντέλο του von Neumann διέπονται από μία ανελαστικότητα κυρίως λόγω του τρόπου λειτουργίας του μοντέλου. Για παράδειγμα οι εντολές εκχώρησης (store instruction) που προσομοιώνουν κελιά της μνήμης ή οι εντολές μεταπήδησης (jump instruction), φόρτωσης (load instruction), ή οτιδήποτε άλλο σε επίπεδο προγραμματισμού επιφορτίζουν τη γλώσσα με ασήμαντη λεπτομέρεια και κυρίως λόγω της μεταβαλλόμενης κατάστασης (mutable state) που εισάγει το μοντέλο αυτό, καταστρατηγεί τα μαθηματικά υπόβαθρα που θα μπορούσε να εκμεταλλευτεί. Αυτό, σύμφωνα με τον John Backus, οδηγεί στη συμφόρηση von Neumann των γλωσσών προγραμματισμού (von Neumann bottleneck of computer languages). Σε αντίθεση ο συναρτησιακός προγραμματισμός και οι αλγεβρικές ιδιότητες που διέπουν τις συναρτήσεις, όπως υποστήριζε, μπορούν να προσδώσουν τεράστια ευχέρεια και δυνατότητες δημιουργίας σύνθετων υπολογιστικών συστημάτων (Backus, 1978).

Το παραπάνω γεγονός, σε συνδυασμό με τη συγγραφή του γνωστού βιβλίου "Structure and Interpration of Computer Programs" του Harold Abelson και Gerald Jay Sussman, επέστησαν την προσοχή της κοινότητας των επιστημόνων της πληροφορικής και εγκαθίδρυσαν τον συναρτησιακό προγραμματισμό ως ένα πανίσχυρο μοντέλο με τεράστια

δυναμική. Τέλος σταθμός αποτέλεσε και η δημιουργία της Haskell, μιας συναρτησιακής γλώσσας προγραμματισμού με σημαντική απήχηση (Abelson & Sussman, 1996).

1.3 Σκοπός

Αντικείμενο λοιπόν της παρούσας διπλωματικής εργασίας αποτελεί η μελέτη των βασικών αρχών του συναρτησιακού προγραμματισμού γενικότερα, καθώς και η ανάλυση της συμπεριφοράς του μοντέλου υπό το πρίσμα της δημοφιλούς προγραμματιστικής γλώσσας, JavaScript. Θα δοθεί έμφαση σε ένα πλήθος τεχνικών που όχι μόνο έχουν τη δυνατότητα να διευκολύνουν τον προγραμματιστή αλλά θα τον βοηθήσουν στη διαδικασία σχηματισμού μίας νέας αντίληψης στην οποία τα προγραμματιστικά προβλήματα αντιμετωπίζονται αυστηρά από στάδια αποσύνθεσης και σύνθεσης. Παράλληλα, η βασική αρχή στην οποία στηρίζεται η υιοθέτηση του συναρτησιακού μοντέλου αποτελεί όχι η ανάπτυξη ενός ριζικά νέου και αυστηρού τρόπου προσέγγισης ανάπτυξης λογισμικού στην JavaScript, αλλά η ομαλή συνύπαρξη με το υπάρχον αντικειμενοστρεφές περιβάλλον καθότι όπως αποδεικνύεται μία τέτοια προσέγγιση αποτελεί τη βέλτιστη λύση. Τέλος, απώτερος στόχος είναι η διεύρυνση των ικανοτήτων ενός προγραμματιστή με δυνατότητες ανάπτυξης αφηρημένων λειτουργιών υπό τη μορφή συναρτήσεων, η εφαρμογή των οποίων δίνει λύσεις στο συγκεκριμένο προγραμματιστικό πρόβλημα που καλείται να αντιμετωπίσει.

1.4 Διάρθρωση της μελέτης

Λόγω της φύσης του θέματος, η εργασία παράλληλα με τον βασικό σκοπό της ανάδειξης του συναρτησιακού μοντέλου γραφής στην προγραμματιστική γλώσσα JavaScript, θα μεταφέρει στον αναγνώστη πληθώρα γνώσεων όσον αφορά στον συναρτησιακό προγραμματισμό ως ευρύτερη έννοια που δεν περιορίζεται στα πλαίσια μίας μόνο γλώσσας, άλλα εμφανίζει καθολική απήχηση.

Στο 2^{ο} κεφάλαιο θα γίνει αντικείμενο μελέτης το θεμελιώδες, για τον συναρτησιακό προγραμματισμό, υπολογιστικό σύστημα του λογισμού λάμδα. Θα δοθεί έμφαση στις βασικές ιδιότητες του και θα παρακολουθήσουμε τον τρόπο με τον οποίο οι

λύσεις των προβλημάτων ερμηνεύονται ως αφαιρέσεις (**abstractions**) και εφαρμογές (**applications**).

Στο 3ο κεφάλαιο θα δούμε τις ιδιότητες της προγραμματιστικής γλώσσας JavaScript, καθώς και το βαθμό στον οποίο δύναται η μεταφορά των αρχών του συναρτησιακού προγραμματισμού στο ειδικό περιβάλλον της γλώσσας.

Στο 4ο κεφάλαιο, που αποτελεί και τον κορμό της παρούσας εργασίας, θα πραγματοποιηθεί πλήρης ανάλυση του συναρτησιακού μοντέλου προγραμματισμού, με την ευρύτερη έννοια του όρου, μέσω μίας πληθώρας τεχνικών των οποίων η υλοποίηση θα γίνει στην JavaScript.

Στο 5ο κεφάλαιο θα παρατεθούν τα συμπεράσματα της εργασίας και θα αναλυθεί ο βαθμός της δυναμικής που παρουσιάζει τελικά ο συναρτησιακός προγραμματισμός στο περιβάλλον της JavaScript.

Στο παράρτημα Α, θα δοθούν κατευθυντήριες οδηγίες και παραδείγματα για τον τρόπο με τον οποίο μπορεί κάποιος να χρησιμοποιήσει την προγραμματιστική βιβλιοθήκη που υλοποιήθηκε στα πλαίσια της εργασίας. Η βιβλιοθήκη θα προσανατολίζεται στην παροχή έτοιμου κώδικα που θα βοηθάει τον προγραμματιστή στη διαδικασία υιοθέτησης συναρτησιακών προγραμματιστικών τεχνικών κατά την ανάπτυξη λογισμικού.

2 Λάμδα Λογισμός (Lambda Calculus)

2.1 Ιστορία

Για την καλύτερη κατανόηση του λογισμού λάμδα, για τον οποίο έγινε μια μικρή ιστορική αναφορά στην εισαγωγή της παρούσας εργασίας, κρίνεται απαραίτητη η ανίχνευση των μαθηματικών ριζών του συναρτησιακού προγραμματισμού.

Ο Συναρτησιακός προγραμματισμός έχει τις ρίζες του στη μαθηματική λογική (mathematical logic). Τα άτυπα λογικά συστήματα (informal logical systems) είναι σε χρήση εδώ και δύο χιλιετίες αλλά η πρώτη απόπειρα τυποποίησης τους έγινε για πρώτη φορά στα μέσα του 19ου αιώνα από τους Hamilton, De Morgan και Boole. Από τα αποτελέσματα του έργου τους διακρίνονται δύο μεγάλες μαθηματικές θεωρίες, ο προτασιακός λογισμός (propositional logic) και η κατηγορηματική λογική (predicate logic).

Ο προτασιακός λογισμός αποτελεί ένα μαθηματικό σύστημα που χρησιμοποιεί τις εκφράσεις Αληθής, Ψευδής (**True**, **False**) ως βασικές τιμές (values) και τις εκφράσεις Και, Ή, Όχι (**And**, **Or**, **Not**) ως πράξεις (**operations**). Χρησιμοποιώντας τον προτασιακό λογισμό και στηριζόμενος σε μαθηματικά αξιώματα (axioms) κάποιος είναι σε θέση να αποδείξει το εάν μία τυχαία έκφραση μπορεί να αποτελεί και θεώρημα (πάντα Αληθές). Στη συνέχεια, επαγωγικά κάνοντας χρήση των θεωρημάτων μπορεί να κατασκευάσει άλλα και τελικά να δημιουργήσει μία θεωρία (Theory). Ο προτασιακός λογισμός αποτελεί συνεπώς μία θεωρητική υποστήριξη για τη δημιουργία πανίσχυρων λογικών συστημάτων που στις μέρες μας έχουν απήχηση από τα ηλεκτρονικά και ψηφιακά κυκλώματα έως τα σύνθετα τηλεπικοινωνιακά συστήματα (Thompson, 1991).

Η κατηγορηματική λογική επεκτείνει τον προτασιακό λογισμό δίνοντας τη δυνατότητα ενσωμάτωσης μη-λογικών τιμών όπως είναι οι αριθμοί (numbers), τα αλφαριθμητικά στοιχεία (strings) και τα σύνολα (sets). Η υλοποίηση αυτής της δυνατότητας επιτυγχάνεται με τα κατηγορήματα (predicates) τα οποία γενικεύουν τις λογικές εκφράσεις με την ικανότητα αποτύπωσης μη-λογικών τιμών, αλλά και τις συναρτήσεις (functions) που γενικεύουν τις πράξεις που μπορούν να εφαρμοστούν σε αυτά τα κατηγορήματα. Επιπρόσθετα η κατηγορηματική λογική εισάγει και την έννοια των ποσοδεικτών (quantifiers) που βοηθούν στην αναπαράσταση ιδιοτήτων μίας σειράς τιμών. Οι δύο πιο γνωστοί ποσοδεικτές στην κατηγορηματική λογική είναι ο καθολικός

ποσοδείκτης (universal quantifier) ή "για όλα" ("for all, ∀x ") (Για παράδειγμα όλα τα στοιχεία ενός συνόλου υπακούνε σε μία ιδιότητα) και ο υπαρξιακός ποσοδείκτης (existential quantifier) ή "υπάρχει τουλάχιστον ένα / για μερικά" ("there is at least one / for some, ∃x") (Για παράδειγμα τουλάχιστον ένα στοιχείο ενός συνόλου υπακούει σε μία συγκεκριμένη ιδιότητα). Η κατηγορηματική λογική εφαρμόζεται σε διάφορα προβλήματα μέσα από τη θεωρητική ανάπτυξη κατηγορημάτων, συναρτήσεων, αξιωμάτων και συμπερασμάτων. Για παράδειγμα η κατηγορηματική λογική που εφαρμόζεται στη θεωρία αριθμών χρησιμοποιείται για τον λογισμό αυτών (από ανθρώπους) (Thompson, 1991).

Και στις δύο αυτές θεωρίες αξίζει να σημειωθεί ότι οι συσχετίσεις μεταξύ ονομάτων και τιμών δεν μεταβάλλονται. Επίσης η αποτίμηση των εκφράσεων δεν έχει κάποια συγκεκριμένη σειρά.

Στις αρχές του 20ού αιώνα σε μια απόπειρα απόδειξης των Russell και Whitehead ότι η μαθηματική αλήθεια πηγάζει εξ' ολοκλήρου από τη λογική αλήθεια, όπως αναφέρεται στο έργο τους "Principia Mathematica", ο Hilbert προτάσσει μία θεωρία επονομαζόμενη "Program" η οποία στόχο είχε να υποστηρίξει το "Principia Mathematica" και να αποδείξει ότι το έργο των προηγουμένων είχε συνοχή και ήταν ολοκληρωμένο. Το 1931 όμως η απόδειξη του Gödel περί μη πληρότητάς των συστημάτων όσον αφορά την περιγραφή της βασικής αριθμητικής έβαλε φρένο στο "Program" του Hilbert.

Μπορεί το έργο του Hilbert να επισκιάστηκε από τη θεωρία του Godel, δεν έγινε το ίδιο όμως και στην απήχηση που είχε το "Program" στη θεωρία της υπολογισιμότητας (theory of computability). Το έργο του βοήθησε στην ανάπτυξη τυπικών συστημάτων περιγραφής υπολογισμών. Συγκεκριμένα το 1936 παρουσιάστηκαν τρεις (3) διαφορετικές προσεγγίσεις υπολογισιμότητας.

- Μηχανές Turing (Turing Machines)
- Αναδρομική θεωρία Συναρτήσεων (Recursive Function Theory) του Kleene
- Λογισμός Λάμδα (Lambda Calculus) του Church

Καθεμία από τις τρεις (3) μορφές παρουσιάζει ένα οργανωμένο σύνολο από πρωταρχικές (primitive) πράξεις και υποστηρίζεται από ένα απλό πλαίσιο κανόνων δόμησης. Το κοινό χαρακτηριστικό ανάμεσα σε αυτές είναι ισοδυναμία μεταξύ τους αλλά

και η δυνατότητα πλήρους εφαρμογής τους στις μηχανές von Neumann (τους ψηφιακούς υπολογιστές). Αυτό πρακτικά σημαίνει ότι το αποτέλεσμα που προκύπτει από τη χρήση ενός συστήματος θα είναι το ίδιο σε περίπτωση που στη θέση του χρησιμοποιηθεί άλλο διαφορετικό, αλλά και επιπροσθέτως υπάρχει η δυνατότητα μοντελοποίησης του ενός από το άλλο. Συνακόλουθα με βάση τα παραπάνω, οι γλώσσες προγραμματισμού που αναπτύχθηκαν για τις μηχανές von Neumann έχουν την ικανότητα να περιγράψουν και να υλοποιήσουν οποιοδήποτε από τα τρία (3) συστήματα.

Μία από τις βασικές διαφορές ανάμεσα στις μηχανές Turing, και στα άλλα δύο πλαίσια υπολογισιμότητας, ήταν το γεγονός ότι η μηχανή Turing υλοποιούσε του υπολογισμούς ως μία μηχανική αλληλεπίδραση συμβόλων στηριζόμενη σε εντολές εκχώρησης (assignment) και στην αυστηρή χρονικά σειρά αποτίμησης εκφράσεων (time ordered evaluation). Η Αναδρομική Θεωρία Συναρτήσεων και ο Λογισμός Λάμδα από την άλλη αναδείκνυε τους υπολογισμούς ως μία δομημένη εφαρμογή συναρτήσεων (function application) η οποία δεν παρουσίαζε εξάρτηση από τη χρονική σειρά εκτέλεσης των αποτιμήσεων.

Ο Turing υπέδειξε ότι είναι αδύνατο να πεις εκ των προτέρων ότι μία τυχαία μηχανή Turing θα περατώσει τη λειτουργία της. Το πρόβλημα αυτό γνωστό ως πρόβλημα περάτωσης (halting problem) δεν επιδέχεται λύση. Το ίδιο ίσχυε και για τα άλλα δύο συστήματα υπολογισιμότητας. Δεν μπορούσες να γνωρίζεις με βεβαιότητα ότι ή αποτίμηση μίας τυχαίας έκφρασης λάμδα ή μίας αναδρομικής κλήσης θα περατωθεί. Όμως σε αντίθεση με το μοντέλο του Turing, ο Church και ο Barkley Rosser κατάφεραν να αποδείξουν ότι στο μοντέλο του λογισμού λάμδα δύο διαφορετικές σειρές αποτίμησης που παρουσιάζουν περατότητα έχουν πάντοτε το ίδιο αποτέλεσμα. Επίσης υποστήριξαν ότι μπορεί μία συγκεκριμένη σειρά αποτίμησης να εμφανίσει μεγαλύτερη πιθανότητα περάτωσης από οποιαδήποτε άλλη. Συνεπώς με βάση το συμπέρασμα των ερευνών τους, ήταν δυνατή η προσπέλαση ενός προγράμματος σε διαφορετικά κομμάτια καθένα από τα οποία μπορούσε να χρησιμοποιήσει διαφορετική σειρά αποτίμησης, και πιο συγκεκριμένα λόγω αυτής της ανεξαρτησίας παρουσιάζονταν η δυνατότητα παράλληλης εκτέλεσης (parallel execution) των υποπρογραμμάτων (Wikipedia/Functional_programming, n.d.).

Στις μέρες μας, ο λογισμός λάμδα και η αναδρομική θεωρία συναρτήσεων δεν αποτελούν μόνο τη ραχοκοκαλιά του Συναρτησιακού Προγραμματισμού αλλά και ένα ευρύτερο πεδίο εφαρμογής υπολογισμών.

2.2 Λάμδα Εκφράσεις (Lambda Expressions)

Οι λάμδα εκφράσεις αποτελούν λοιπόν το εργαλείο πάνω στο οποίο στηρίζεται το σύστημα υπολογισιμότητας της θεωρίας του λογισμού λάμδα. Θα λέγαμε ότι είναι η γλώσσα που χρησιμοποιεί η θεωρία για να εκφράσει τον εαυτό της και συνεπώς να προσδώσει λύση στα προβλήματα που καλείται να αντιμετωπίσει.

Λάμδα έκφραση μπορεί να είναι ένα όνομα (**name**) το οποίο προσδιορίζει ένα σημείο (**point**) σε μία αφαίρεση (**abstraction**), μία συνάρτηση (**function**) που εισάγει τον ορισμό μίας αφαίρεσης ή τέλος μία εφαρμογή συνάρτησης (**function application**) που εξειδικεύει μία αφαίρεση. Έχουμε

<expression> ::= <name> | <function> | <application>

- Ένα όνομα ορίζεται ως μία ακολουθία από χαρακτήρες, εξαιρουμένου του κενού πχ.

    - *Felix*
    - *22*
    - +
    - →

- Μία λάμδα συνάρτηση αποτελεί τον ορισμό μίας αφαίρεσης και έχει τη μορφή <function> ::= λ<name>.<body>, όπου <body> αποτελεί με τη σειρά του μία έκφραση <expression>

    - *λx.x*
    - *λfirst.λsecond.first*
    - *λf.λa. (f a)*

Στη λάμδα έκφραση λx.x το x αποτελεί δεσμευμένη μεταβλητή (bound variable) για το σώμα της συνάρτησης x. Η τελεία (.) χρησιμοποιείται για να διαχωρίσει το όνομα της μεταβλητής από το σώμα της συνάρτησης.

- Μία εφαρμογή συνάρτησης (function application) εξειδικεύει μία συνάρτηση λ και έχει την παρακάτω μορφή

<application> ::= (<function expression> <argument expression>),

όπου

<function expression> ::= <expression>

και

<argument expression> ::= <expression>

  - *(λx.x λa.λb.b)*
  - *(λz.z+3 5)*

Η συγκεκριμενοποίηση της συνάρτησης που επιτελεί η εφαρμογή μίας συνάρτησης εισάγεται με την αντιστοίχιση μίας τιμής στη θέση ενός ονόματος σε μία έκφραση λάμδα. Στην έκφραση (λz.z+3 5) για παράδειγμα η δεσμευμένη μεταβλητή z αντικαθίσταται από τη συγκεκριμένη αρχέγονη (primitive) τιμή 5. Η εφαρμογή δίνει σαν αποτέλεσμα το 8.

Η εφαρμογή μίας συνάρτησης μπορεί να γίνει με δύο τρόπους ανάλογα με τη μέθοδο αποτίμησης που θα επιλεχθεί (**evaluation strategy**).

1) Applicative order (call by value)
2) Normal order (call by name)

Στην πρώτη περίπτωση η αποτίμηση των παραμέτρων προηγείται της εφαρμογής της συνάρτησης επάνω τους Στη δεύτερη η έκφραση που αναπαριστάνει μία παράμετρος αντικαθίσταται στο σώμα της συνάρτησης δίχως να γίνει αποτίμηση της. Για παράδειγμα αν έχουμε τη λ-έκφραση (λz.z+3 1+2) τότε

1) Κατά την εφαρμογή του call by value μοντέλου η έκφραση 1+2 αρχικά θα αποτιμηθεί και στη συνέχεια θα αντικατασταθεί στο σώμα z+3.

(λz.z+3 1+2) = (λz.z+3 3) = 6

2) Κατά την εφαρμογή του call by name μοντέλου η έκφραση 1+2 θα αντικατασταθεί ακριβώς ίδια στο σώμα της έκφρασης λ, όπου υπάρχει παράμετρος z. Δηλαδή

(λz.z+3 1+2) = (1+2) + 3 = 6

Η κανονική σειρά (call by name) θεωρείται πιο ισχυρή, γενικά πλην όμως εισάγει αρκετές φορές περιττούς υπολογισμούς. Παρακάτω θα δούμε μερικά παραδείγματα βασικών συναρτήσεων που αποτυπώνονται με λάμδα εκφράσεις.

2.2.1 Identity Function (Ταυτοτική Συνάρτηση)

Η έκφραση λx.x αποτελεί την identity function στις εκφράσεις λ και όπως γίνεται αντιληπτό από το σώμα της συνάρτησης δουλεία της είναι η επιστροφή της παραμέτρου πάνω στην οποία εφαρμόζεται η συγκεκριμένη έκφραση. Η συνάρτηση ορίζεται με παράμετρο τη δεσμευμένη μεταβλητή x την οποία επιστρέφει αφού το σώμα της συνάρτησης ορίζεται ως x. Δηλαδή

(λx.x 3) = 3
(λx.x felix) = felix
(λx.x λx.x) = λx.x = identity function

2.2.2 Self Application Function (Συνάρτηση Αυτοεφαρμογής)

Η συνάρτηση αυτοεφαρμογής έχει την παρακάτω μορφή

λs.(s s)

και ουσιαστικά κάνει εφαρμογή της παραμέτρου που δέχεται επάνω στην ίδια την παράμετρο. Για παράδειγμα αν εφαρμόσουμε τη συνάρτηση ταυτότητας με παράμετρο τη συνάρτηση αυτοεφαρμογής έχουμε

(λx.x λs.(s s)) = λs.(s s) = self application function,

αφού η identity function επιστρέφει πάντοτε την παράμετρο με την οποία καλείται Αν τώρα εφαρμόσουμε τη συνάρτηση αυτοεφαρμογής επάνω στη συνάρτηση ταυτότητας το αποτέλεσμα θα έχει ως εξής:

(λs.(s s) λx.x) = (λx.x λx.x) = λx.x = identity function

Τέλος η εφαρμογή της συνάρτησης αυτοεφαρμογής στον εαυτό της

(λs.(s s) λs.(s s)) = (λs.(s s) λs.(s s)) = (λs.(s s) λs.(s s)) = ... ατέρμων

επιστρέφει τον εαυτό της με παράμετρο τη συνάρτηση αυτοεφαρμογής και συνεπώς η παραπάνω παράσταση δεν περατώνεται.

2.2.3 Function Application Function (Συνάρτηση Εφαρμογής Συνάρτησης)

Η έκφραση λάμδα

λfunc.λarg.(func arg),

έχει ως δεσμευμένη μεταβλητή την func και ως σώμα τη συνάρτηση λarg.(func arg) ή οποία με τη σειρά της έχει ως δεσμευμένη τη μεταβλητή arg και ως σώμα την εφαρμογή της func στην παράμετρο arg, (func arg). Η εφαρμογή της αρχικής έκφρασης λάμδα με μία παράμετρο y επιστρέφει μία νέα συνάρτηση z, η εφαρμογή της οποίας σε μία παράμετρο x θα επιστρέφει z(x) = y(x). Για παράδειγμα έστω η έκφραση

((λfunc.λarg.(func arg) λx.x) λs.(s s)),

Η παράσταση (λfunc.λarg.(func arg) λx.x) επιστρέφει τη συνάρτηση λarg.(λx.x arg). Έπειτα όπως βλέπουμε η συνάρτηση αυτή λarg.(λx.x arg) εφαρμόζεται στη λs.(s s) οπότε έχουμε

(λarg.(λx.x arg) λs.(s s)),

ή αν αντικαταστήσουμε τη δεσμευμένη μεταβλητή arg με την παράμετρο λs.(s s) έχουμε

λx.x λs.(s s) = λs.(s s)

Για την απλοποίηση των παραστάσεων των λάμδα εκφράσεων και καθώς αυτές αυξάνουν σε μέγεθος και πολυπλοκότητα, κατά την ανάλυση αυτών δύναται να χρησιμοποιηθούν μέθοδοι μετονομασίας που επιτρέπουν στον αναλυτή να δημιουργεί ένα επίπεδο αφαίρεσης απαλλάσσοντας τον από την πληθώρα των λεπτομερειών. Έτσι για παράδειγμα ή έκφραση της συνάρτησης ταυτότητας

λx.x, ορίζεται ως identity function με τον παρακάτω τρόπο
def identity = λx.x, ή γενικά
def <name> = <function>

Για τα τρία βασικά είδη συναρτήσεων που μελετήθηκαν παραπάνω έχουμε.

def identity = λx.x
def self_apply = λs.(s s)
def apply = λfunc.λarg.(func arg)

Χρησιμοποιώντας συνδυασμούς των παραπάνω συναρτήσεων μπορούμε να επαναδημιουργησούμε τις ίδιες συναρτήσεις. Για παράδειγμα η συνάρτηση ταυτότητας μπορεί να γραφεί και ως

def identity2 = λx.((apply identity) x) αφού,
identity2 = λx.((λfunc.λarg.(func arg) λx.x) x),

identity2 = λx.(λarg.(λy.y arg)) x),

Έστω arg1 τυχαία παράμετρος τότε αν εφαρμόσουμε τις συναρτήσεις στην παράμετρο αυτή έχουμε,

identity2 arg1 = λx.(λarg.(λy.y arg)) x) arg1,
identity2 arg1 = λarg.(λy.y arg) arg1,
identity2 arg1 = (λy.y arg1)

ή

identity2 = λy.y

Επίσης χρησιμοποιώντας τη συνάρτηση εφαρμογής συνάρτησης μπορούμε να ορίσουμε μία νέα συνάρτηση που έχει τα ίδια αποτελέσματα με την αρχική. Για παράδειγμα έστω ότι έχουμε τη συνάρτηση <function>.

Τότε,

(apply <function>) =
(λf.λa.(f a) <function>) =
λa.(<function> a)

Εφαρμόζοντας την τελευταία συνάρτηση πάνω σε μία τυχαία παράμετρο arg1 έχουμε,

(λa.(<function> a) arg1) == (<function> arg1)

το οποίο τελικώς αποτελεί εφαρμογή της αρχικής συνάρτησης <function> στην παράμετρο arg1. Κάνοντας χρήση της apply συνεπώς προσθέσαμε ένα στρώμα τύπου **β-reduction**. β-reduction ονομάζεται η αντικατάσταση μίας δεσμευμένης μεταβλητής με μία παράμετρο σε ένα σώμα μίας συνάρτησης. Πιο απλά αποτελεί την εφαρμογή μίας συνάρτησης σε μία παράμετρο.

2.2.4 Επιλογή Παραμέτρων

Παρακάτω θα δούμε συναρτήσεις που χρησιμεύουν στην επιλογή παραμέτρων στις εμφωλευμένες κλήσεις συναρτήσεων. Για την επιλογή της πρώτης παραμέτρου χρησιμοποιούμε

def select_first = λfirst.λsecond.first,

Η εφαρμογή της παραπάνω συνάρτησης σε μία παράμετρο x επιστρέφει μία νέα συνάρτηση η εφαρμογή της οποίας σε οποιαδήποτε παράμετρο y επιστρέφει την τιμή x. Για παράδειγμα

((select_first a) b) == ((λfirst.λsecond.first a) b) == (λsecond.a b) == a

ή

((select_first identity) apply) == identity,

Για την επιλογή της δεύτερης παραμέτρου χρησιμοποιούμε

def select_second = λfirst.λsecond.second

Η εφαρμογή της παραπάνω συνάρτησης σε μία παράμετρο x επιστρέφει μία νέα συνάρτηση η εφαρμογής της οποίας σε μία παράμετρο y επιστρέφει πάντα την τιμή y.

((select_second a) b) == ((λfirst.λsecond.second a) b) == (λsecond.second b) = b

ή

((select_second identity) apply) == apply

2.2.5 Ζευγάρωμα Παραμέτρων

Ας υποθέσουμε ότι έχουμε την παρακάτω συνάρτηση

def make_pair = λfirst.λsecond.λfunc.((func first) second),

Παρατηρούμε ότι για τιμή func = select_first τότε έχουμε επιστροφή της παραμέτρου first ενώ για τιμή func = select_second έχουμε επιστροφή της τιμής second. Για παράδειγμα έστω

((make_pair identity) apply),

τότε έχουμε

((λfirst.λsecond.λfunc.((func first) second) identity) apply) =
λfunc.((func identity) apply)

Με λίγα λόγια έχουμε επιστροφή μίας συνάρτησης ή οποία είναι έτοιμη με την εφαρμογή της κατάλληλης συνάρτησης επιλογής (select_first, select_second) να εξάγει την αντίστοιχη παράμετρο (identity, apply) Έτσι για func = select_first έχουμε,

(λfunc.((func identity) apply) select_first) =
((select_first identity) apply) =
identity,

Ενώ για func = select_second,

(λfunc.((func identity) apply) select_second) =
((select_second identity) apply) =
apply

Τα παραπάνω αποτελούν θεμελιώδη θεωρία για την κατασκευή λιστών στον συναρτησιακό προγραμματισμό. Γλώσσες που ακολουθούν τον συναρτησιακό μοντέλο προγραμματισμό όπως ή Haskell υλοποιούν τις λίστες κάνοντας χρήση της παραπάνω θεωρίας.

2.2.6 Δεσμευμένες και ελεύθερες μεταβλητές

Στην έκφραση λάμδα

(λf. (f λx.x) λs.(s s))

υπάρχουν τρείς συναρτήσεις. Η πρώτη έχει σαν δεσμευμένη μεταβλητή την f, η δεύτερη την x και η τρίτη την s. Έτσι απλοποιώντας την παράσταση καταλήγουμε στην έκφραση

(λs.(s s) λx.x) == (λx.x λx.x) == λx.x

Υπάρχει όμως μεγάλη πιθανότητα οι δεσμευμένες μεταβλητές συναρτήσεων μίας έκφρασης λάμδα να έχουν ίδια ονόματα. Για παράδειγμα

(λf. (f λf.f) λs.(s s))

Η παραπάνω έκφραση πρέπει να επιστρέφει τη συνάρτηση ταυτότητας (identity) όπως είδαμε προηγουμένως. Η αντικατάσταση της παραμέτρου f με τη λs.(s s) πρέπει να γίνει σωστά. Αυτό σημαίνει ότι το f που θα αλλάξει τιμή είναι το εξωτερικό και όχι το εσωτερικό. Η εσωτερική συνάρτηση λf.f είναι αυτόνομη και έχει σαν δεσμευμένη μεταβλητή το δικό της f που τυχαίνει λόγω συνωνυμίας να συσχετίζεται λανθασμένα με το εξωτερικό. Για την αποφυγή και τη διαλεύκανση αυτών των περιπτώσεων πρέπει να υπάρχει συγκεκριμένος τρόπος συσχέτισης παραμέτρων με το σώμα μίας συνάρτησης.

Έτσι για μία τυχαία συνάρτηση λ<name>.<body> η δεσμευμένη μεταβλητή <name> συσχετίζεται με τις εμφανίσεις που παρουσιάζει η ονομασία αυτή εντός του σώματος <body> της συνάρτησης και πουθενά αλλού. Πιο επίσημα θα λέγαμε η εμβέλεια (scope) της μεταβλητής <name> είναι το <body>. Για παράδειγμα στην έκφραση

λf.λs.(f (s s))

η δεσμευμένη μεταβλητή f έχει σαν εμβέλεια το σώμα λs.(f (s s))

Στην έκφραση

(λf.λg.λa(f (g a)) λg.(g g))

η δεσμευμένη μεταβλητή f έχει σαν εμβέλεια το σώμα λg.λa(f (g a)) και τίποτα παραπέρα. Επίσης η δεσμευμένη μεταβλητή g στην έκφραση λg.(g g)
έχει σαν εμβέλεια μόνο το (g g) και τίποτα άλλο (δεν έχει σχέση δηλαδή με το αριστερό g). Γενικά στις συναρτήσεις υπάρχουν δύο είδη μεταβλητών. Οι δεσμευμένες (bound variables) και οι ελεύθερες (free variables). Δεσμευμένη μεταβλητή είναι αυτή που εμφανίζεται σαν όνομα παραμέτρου σε μία συνάρτηση και συσχετίζεται με το σώμα της ίδιας συνάρτησης εκτός από περιπτώσεις συνωνυμίας όπως είδαμε προηγουμένως. Σε όλες τις υπόλοιπες περιπτώσεις μιλάμε για ελεύθερες μεταβλητές. Για παράδειγμα στο σώμα της συνάρτησης

λf.(f λf.f)

δηλαδή το

(f λ.f.f)

το πρώτο f αποτελεί ελεύθερη μεταβλητή αλλά τα υπόλοιπα f είναι δεσμευμένα.

Ξέροντας πλέον τη διαφορά ανάμεσα σε δεσμευμένες και ελεύθερες μεταβλητές μπορούμε να προχωρήσουμε σε έναν πιο επίσημο ορισμό του **β-reduction**. Έχοντας την έκφραση

(λ<name>.<body> <argument>)

η μέθοδος του β-reduction υλοποιεί αντικατάσταση όλων των ελεύθερων μεταβλητών <name> στην έκφραση <body> με την παράμετρο <argument> . Αυτό πρακτικά σημαίνει αντικατάσταση της δεσμευμένης μεταβλητής <name> στην έκφραση

(λ<name>.<body> <argument>)

Για παράδειγμα στην έκφραση

(λf. (f λf.f) λs.(s s)),

η ελεύθερη μεταβλητή f στο (f λf.f) αντικαθίσταται με το λs.(s s),

(λs.(s s) λf.f) == (λf.f λf.f) == λf.f

2.2.7 Συγκρούσεις Ονομάτων και a-conversion

Έστω ότι έχουμε τη συνάρτηση

def apply = λfunc.λarg.(func arg)

και την έκφραση

((apply arg) foo)

Τότε παρατηρούμε ότι αν προχωρήσουμε στη μέθοδο του β-reduction θα εμφανιστεί σύγχυση ονομάτων καθώς η arg εντός της apply είναι δεσμευμένη μεταβλητή ενώ η arg που εμφανίζεται στην έκφραση ((apply arg) foo) είναι ελεύθερη. Θα είχαμε λοιπόν

((apply arg) foo) =
((λfunc.λarg.(func arg) arg) foo) =
(λarg.(arg arg) foo) =
(foo foo)

το οποίο όμως είναι εσφαλμένο καθώς τα δύο arg που εμφανίζονται έχουν διαφορετικό ρόλο αλλά ίδιο όνομα. Έτσι λοιπόν λέμε ότι έχουμε σύγχυση ονομάτων όταν μια εφαρμογή τύπου β-reduction εισάγει στην εμβέλεια μιας δεσμευμένης μεταβλητής, μία ελεύθερη με το ίδιο όνομα. Για την αντιμετώπιση αυτών των περιπτώσεων χρησιμοποιείται η μέθοδος

α-conversion που μετονομάζει τη δεσμευμένη μεταβλητή της συνάρτησης για την αποφυγή συγχύσεων. Πιο επίσημα θα λέγαμε ότι για μία συνάρτηση

λ<name1>.<body>

η μέθοδος του α-conversion αντικαθιστεί όλες τις ελεύθερες μεταβλητές <name1> στο <body> με ένα νέο όνομα <name2>. Συνεπώς η έκφραση γίνεται

λ<name2>.<body>

2.2.8 Συνθήκες και τύποι δεδομένων αληθείας

Οι τιμές αληθείας της άλγεβρας Boole, ΑΛΗΘΗΣ (**TRUE**), ΨΕΥΔΗΣ (**FALSE**) καθώς και οι λογικοί τελεστές ΟΧΙ (**NOT**), ΚΑΙ (**AND**), Η (**OR**) μπορούν να αναπαρασταθούν πλήρως από τις εκφράσεις λάμδα με τη μορφή συναρτήσεων. Αν θεωρήσουμε την αναπαράσταση επιλογής που χρησιμοποιείται από την C (ternary operator) τότε η έκφραση

<condition> ? <expression1> : <expression2>

επιστρέφει την τιμή <expression1> αν ή συνθήκη <condition> είναι ΑΛΗΘΗΣ (TRUE) και την τιμή <expression2> αν το <condition> είναι ΨΕΥΔΗΣ (FALSE).

Για τη μοντελοποίηση της παραπάνω αναπαράστασης στη μορφή των λάμδα εκφράσεων και συνεπώς για την εισαγωγή της δομής επιλογής στις συναρτησιακές γλώσσες γίνεται χρήση των συναρτήσεων select_first, select_second, make_pair που είδαμε πιο πάνω. Συνεπώς ορίζουμε τη συνάρτηση

def cond = λe1.λe2.λc.((c e1) e2)

Η εφαρμογή της παραπάνω συνάρτησης στις παραμέτρους <expression1> και <expression2> επιστρέφει

((cond <expression1>) <expression2>) =
λc.((c <expression1>) <expression2>)

Η τελευταία παράσταση αποτελεί συνάρτηση με δεσμευμένη τη μεταβλητή c. Αν γίνει εφαρμογή της στην select_first τότε έχουμε

(λc.((c <expression1>) <expression2>) select_first) =
((select_first <expression1>) <expression2>) =
<expression1>

Αντίθετα αν σαν c ορίσουμε την select_second τότε

(λc.((c <expression1>) <expression2>) select_second) =
((select_ second <expression1>) <expression2>) =
<expression2>

Εύλογα λοιπόν συμπεραίνουμε ότι ανάλογα με την τιμή της συνθήκης c έχουμε αντίστοιχη επιστροφή παραμέτρου. Αν c = select_first τότε επιστρέφεται <expression1> ενώ όταν c = select_second τότε επιστρέφεται το <expression2>. Οι συναρτήσεις select_first και select_second συνεπώς αποτελούν τις τιμές TRUE και FALSE στις εκφράσεις λάμδα ή

def true = select_first
def false = select_second

Τελικώς δηλαδή

(((cond <expression1>) <expression2>) true) == <expression1>,
(((cond <expression1>) <expression2>) false) == <expression2>

2.2.9 NOT

Ο μοναδιαίος τελεστής (unary operator) NOT παίρνει τη μορφή

NOT <operand>

και ο πίνακας αληθείας του επιστρέφει

X	**NOT X**
FALSE	TRUE
TRUE	FALSE

Σχήμα 2-1 Πίνακας αληθείας NOT

Χρησιμοποιώντας τον λογισμό λάμδα μπορούμε να ορίσουμε συνάρτηση που υλοποιεί τη λειτουργία του τελεστή NOT, ορίζουμε ως

def not = λx.(((cond false) true) x)

Απλοποιώντας το σώμα της παραπάνω συνάρτησης έχουμε

(((cond false) true) x) =
(((λe1.λe2.λc.((c e1) e2) false) true) x) =
((x false) true),

ή

def not = λx.((x false) true)

Για παράδειγμα η έκφραση NOT TRUE επιστρέφει

NOT TRUE =
not true =
((select_first false) true) == false == FALSE

ενώ η έκφραση NOT FALSE

NOT FALSE=
not false =
((select_second false) true) == true == TRUE

Παρατηρούμε ότι οι παραπάνω τιμές ακολουθούν τον πίνακα αληθείας του τελεστή NOT

2.2.10 AND

Ο δυαδικός τελεστής AND παίρνει τη μορφή

<operand> AND <operand>

και ο πίνακας αληθείας του τελεστή αποτυπώνει τα παρακάτω αποτελέσματα

X	**Y**	**X AND Y**
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

Σχήμα 2-2 Πίνακας αληθείας AND

Από τον πίνακα παρατηρούμε ότι όταν ό αριστερός τελεστέος (X) έχει την τιμή TRUE τότε το αποτέλεσμα του τελεστή (X AND Y) είναι η λογική τιμή του δεύτερου τελεστέου (Y) ενώ όταν αυτός έχει την τιμή FALSE τότε η παράσταση (X AND Y) επιστρέφει πάντα FALSE. Δηλαδή συνοψίζοντας μπορούμε να αποτυπώσουμε την παρακάτω λογική έκφραση

X ? Y : FALSE

Χρησιμοποιώντας τον λογισμό λάμδα και την παραπάνω λογική έκφραση μπορούμε να αναπαραστήσουμε τη λειτουργία του τελεστή υπό τη μορφή μίας συνάρτησης λ. Έτσι ορίζουμε

def and = λx.λy.(((cond y) false) x)

Απλοποιώντας το σώμα της παραπάνω συνάρτησης έχουμε

(((cond y) false) x) =
(((λe1.λe2.λc.((c e1) e2) y) false) x) =
((x y) false),

ή

def and = λx.λy.((x y) false)

Για παράδειγμα η έκφραση TRUE AND FALSE επιστρέφει

TRUE AND FALSE =
((and true) false) == ((true false) false)

όμως true = select_first = λfirst.λsecond.first άρα

((true false) false) == ((λfirst.λsecond.first false) false) =
false = FALSE

το οποίο αποτελεί μία περίπτωση του πίνακα αληθείας. Ομοίως αποδεικνύουμε και τις άλλες περιπτώσεις.

2.2.11 OR

Ο δυαδικός τελεστής OR παίρνει τη μορφή

<operand> OR <operand>

και ο πίνακας αληθείας του τελεστή αποτυπώνει τα παρακάτω αποτελέσματα

X	**Y**	**X OR Y**
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

Σχήμα 2-3Πίνακας αληθείας OR

Από τον πίνακα παρατηρούμε ότι όταν ό αριστερός τελεστέος (X) έχει την τιμή FALSE τότε το αποτέλεσμα του τελεστή (X OR Y) είναι η λογική τιμή του δεύτερου τελεστέου (Y) ενώ όταν αυτός έχει την τιμή TRUE τότε η παράσταση (X OR Y) επιστρέφει πάντα TRUE. Δηλαδή συνοψίζοντας μπορούμε να αποτυπώσουμε την παρακάτω λογική έκφραση

X ? TRUE : Y

Χρησιμοποιώντας τον λογισμό λάμδα και την παραπάνω λογική έκφραση μπορούμε να αναπαραστήσουμε τη λειτουργία του τελεστή υπό τη μορφή μίας συνάρτησης λ. Έτσι ορίζουμε

def or = λx.λy.(((cond true) y) x)

Απλοποιώντας το σώμα της παραπάνω συνάρτησης έχουμε

(((cond true) y) x) =
(((λe1.λe2.λc.((c e1) e2) true) y) x) =

((x true) y),

ή

def or = λx.λy.((x true) y)

Για παράδειγμα η έκφραση FALSE OR TRUE επιστρέφει

FALSE OR TRUE =
((or false) true) == ((false true) true)

όμως false = select_second = λfirst.λsecond.second άρα

((false true) true) =
((λfirst.λsecond.second true) true) =
true = TRUE

το οποίο μία περίπτωση του πίνακα αληθείας. Ομοίως αποδεικνύουμε και τις άλλες περιπτώσεις (Michaelson, 2011).

Κάπου εδώ τελειώνει η εισαγωγή για τον λογισμό λάμδα και τις λάμδα εκφράσεις. Ο βαθμός ανάλυσης και το πεδίο που μπορούν αυτές να επεκταθούν είναι πολύ μεγάλο και κάτι τέτοιο δεν είναι απαραίτητο στα πλαίσια της παρούσας διπλωματικής. Στόχος ήταν να δοθεί στον αναγνώστη μία βασική εικόνα για τη δυνατότητα αναπαράστασης σύνθετων εκφράσεων ως συναρτήσεις που συνακόλουθα μπορούν να χρησιμοποιηθούν σε γλώσσες προγραμματισμού συναρτησιακού προσανατολισμού. Για την ακρίβεια ο λογισμός λάμδα, όπως προαναφέραμε, αποτελεί το βασικό αλφάβητο έκφρασης των γλωσσών αυτών. Στα επόμενα κεφάλαια θα δούμε παραδείγματα μετάφρασης εκφράσεων λάμδα στη βασική γλώσσα με την οποία ασχολείται η παρούσα εργασία, δηλαδή την JavaScript. Τέλος σε αυτό το σημείο αξίζει να τονιστεί ότι το μοντέλο του λογισμού λάμδα σαφώς είναι καθορισμένο και αυστηρό, όχι όμως και ο βαθμός στον οποίο το ακολουθεί μία γλώσσα προγραμματισμού. Το γεγονός αυτό καθορίζει και το κατά ποσό μία γλώσσα ανήκει στη συναρτησιακή σχολή ή όχι.

3 Συναρτησιακή JavaScript (Functional JavaScript)

3.1 Γλώσσα

Η JavaScript ή συχνά αποκαλούμενη JS, αποτελεί μία γλώσσα υψηλού επιπέδου που συναντά μεγάλη δημοφιλία στις μέρες μας αφού μαζί με την HTML και CSS αποτελούν τον βασικό κορμό του παγκόσμιου ιστού (World Wide Web). Χρησιμοποιείται κυρίως για τη δημιουργία διαδραστικών ιστοσελίδων η πιο σωστά δικτυακών εφαρμογών φιλικών προς τον χρήστη. Η πλειονότητα των ιστοσελίδων κάνει εκτεταμένη χρήση της γλώσσας και συνεπώς όλοι οι γνωστοί περιηγητές την υποστηρίζουν (firefox, chrome, safari, edge, opera). Η εκτέλεση της γλώσσας υλοποιείται από ενσωματωμένες μηχανές JavaScript (JavaScript engines) που ακολουθούν τις προδιαγραφές που ορίζει ο αρμόδιος φορέας ECMAScript. Ως πολυπαραδειγματική γλώσσα, η JavaScript υποστηρίζει πολλά μοντέλα προγραμματισμού όπως χειρισμού γεγονότων (event-driven), συναρτησιακού (functional) και προστακτικού (συμπεριλαμβανομένου του αντικειμενοστρεφή προγραμματισμού) και διαθέτει βιβλιοθήκες (API) για διαχείριση κειμένου, ημερομηνιών, πινάκων, κανονικών εκφράσεων και χειρισμού του DOM (Document Object Model) αλλά δεν υποστηρίζει είσοδο-έξοδο (I/O) όπως δικτυακές κλήσεις, κλήσεις προς δίσκο ή απεικόνιση γραφικών.

Η JavaScript συχνά λόγω των συντακτικών ομοιοτήτων αλλά και της ονομασίας της συγχέεται λανθασμένα με την Java. Οι δύο γλώσσες έχουν τεράστια σχεδιαστικές διαφορές και η άποψη ότι η γνώση της μίας συνεπάγεται και γνώση της άλλης (λόγω συντακτικών ομοιοτήτων κυρίως) μπορεί να αποβεί μοιραία για έναν προγραμματιστή. Αντίθετα η JavaScript επηρεάστηκε σε μεγάλο βαθμό από τις γλώσσες Self και Scheme υιοθετώντας από την πρώτη την πρωτότυπη κληρονομικότητα (prototypal inheritance) και αντικειμενοστρέφεια και από τη δεύτερη δυνατότητες συναρτησιακού προγραμματισμού.

3.2 Ιδιότητες

Προστακτική και δομημένη (Imperative and Structured)

Η JavaScript υποστηρίζει πλήρως τις τεχνικές του δομημένου προγραμματισμού που ακολουθούν γλώσσες όπως η C (if statements, while loops, switch statements, do while

loops etc) με τη μοναδική διαφορά να παρουσιάζεται στον χειρισμό της εμβέλειας των μεταβλητών. Η JavaScript μέχρι πρότινος υποστήριζε μόνο τη συναρτησιακή εμβέλεια (function scoping) σε αντίθεση με τις περισσότερες γλώσσες που χρησιμοποιούν την τμηματική εμβέλεια (block scoping). Η έκδοση έξι (6) της γλώσσας (ECMAScript 2015 v6) εισήγαγε αρκετές δυνατότητες και μία από αυτές ήταν η δήλωση μεταβλητών τμηματικής εμβέλειας (με τη χρήση των λέξεων **let** και **const** αντί για **var**). Τέλος η γλώσσα κάνει διάκριση των εκφράσεων και των δηλώσεων (expression and statements), ένα γεγονός που όπως θα δούμε δεν ευνοεί τη συναρτησιακή διάσταση της γλώσσας.

Δυναμική (Dynamic)

Η JavaScript είναι μία γλώσσα δυναμικών τύπων (dynamically typed) υπό την έννοια ότι οι τύποι (String, Number, Boolean) είναι συσχετισμένοι με τις τιμές που μπορεί να πάρει μία μεταβλητή και όχι με την ίδια τη μεταβλητή. Μία λοιπόν ανάθεση τύπου String σε μία μεταβλητή δεν απαγορεύει την επανανάθεση της ίδιας μεταβλητής σε μία τιμή τύπου Number. Επίσης η γλώσσα διαθέτει τη συνάρτηση **eval** που επιτρέπει την εκτέλεση αλφαριθμητικών και συνεπώς δυναμικών δηλώσεων.

Πρωτότυπη δομή (Prototype based object oriented)

Η JavaScript δομείται πλήρως από αντικείμενα (objects). Τα αντικείμενα στην JavaScript έχουν τη μορφή ενός λεξικογραφικού πίνακα (dictionaries) που συσχετίζουν αλφαριθμητικά κλειδιά (object property names) σε τιμές (property value). Η προσπέλαση των τιμών των αντικείμενων μπορεί να γίνει με δύο τρόπους. Στην πρώτη περίπτωση χρησιμοποιείτε το όνομα του αντικειμένου ακολουθούμενο από τελεία και το όνομα του κλειδιού (dot notation) και στη δεύτερη το όνομα του αντικειμένου ακολουθούμενο από αγκύλες εντός των οποίων αναγράφεται ως αλφαριθμητικό το όνομα του κλειδιού (bracket notation). Αν για παράδειγμα έχουμε το αντικείμενο

```
var obj = {a: "lizard"};
```

τότε για να ανακτήσουμε την τιμή "lizard" είτε γράφουμε obj.a είτε obj["a"]. Η δεύτερη αναπαράσταση προσδίδει δυναμικότητα καθώς είναι δυνατή η χρήση μεταβλητών για την ανάκτηση τιμών κατά το στάδιο της εκτέλεσης (run-time).

Οι συναρτήσεις της JavaScript μπορούν να χρησιμοποιηθούν και ως κατασκευαστές αντικείμενων. Η κλήση μίας συνάρτησης με το πρόθεμα **new** δημιουργεί ένα αντικείμενο που έχει σαν πρωτότυπο (κληρονομεί δηλαδή ιδιότητες και χαρακτηριστικά) το αντικείμενο <όνομα συνάρτησης>.prototype . Η αλυσίδα κληρονομικότητας αυτή ονομάζεται πρωτότυπη γιατί η συσχέτιση παιδιού-γονέα (child-parent class) γίνεται μεταξύ προσπελάσιμων αντικειμένων μέσω δεικτών και όχι βάση σχεδίου (blueprint) όπως υλοποιείται η κλασσική κληρονομικότητα κλάσεων. Αν για παράδειγμα έχουμε τη συνάρτηση

```
function Person() {
    this.name = "felix"
}
```

η κλήση

```
var a = new Person();
```

θα δημιουργήσει ένα αντικείμενο με όνομα a που θα κληρονομεί τα χαρακτηριστικά του προτότυπου αντικειμένο Person.prototype το οποίο με τη σειρά του θα κληρονομεί τα χαρακτηριστικά του καθολικού αντικειμένου Object.prototype (όπως Object Class στην Java). Τα αντικείμενα αυτά είναι ανοιχτά για επέκταση ή διαγραφή ιδιοτήτων τους κατά την εκτέλεση αλλά η τροποποίηση των πρωτοτύπων που παρέχει η γλώσσα όπως (Object.prototype, Function.prototype) θεωρείται κακή πρακτική και πρέπει να αποφεύγεται. Τέλος οι συναρτήσεις της JavaScript μπορούν να χρησιμοποιηθούν και ως μέθοδοι αντικείμενων δίχως να αλλάζει η υπόσταση της εσωτερικής λειτουργίας της συνάρτησης. Με άλλα λόγια οι συναρτήσεις της JavaScript (είτε αυτές είναι κατασκευαστές, είτε μέθοδοι, είτε απλές συναρτήσεις) αντιμετωπίζονται ως μία κοινή έννοια από τη μηχανή. Αυτό που αλλάζει είναι ο τρόπος κλήσης τους (Crockford, 2008).

Συναρτησιακή (functional)

Οι συναρτήσεις στην JavaScript αντιμετωπίζονται ως πρώτης τάξης πολίτες (first-class citizens) αφού είναι αντικείμενα και συνακόλουθα μπορούν να χρησιμοποιηθούν ως παράμετροι κλήσης συναρτήσεων αλλά και να επιστραφούν από κλήσεις αυτών. Η δυνατότητα αυτή επιτρέπει τη δημιουργία συναρτήσεων υψηλού βαθμού (higher order functions) και συνεπώς ενεργοποιεί τη συναρτησιακή διάσταση της γλώσσας (Crockford, 2008).

Αναθετική (Delegative)

Η JavaScript υποστηρίζει την άμεση και έμμεση ανάθεση (explicit and implicit delegation) μέσω συναρτησιακών υλοποιήσεων όπως είναι τα Traits, Mixins. Η άμεση ανάθεση γίνεται μέσω κλήσεων των μεθόδων call, apply κάνοντας χρήση της λέξης this εσωτερικά της συνάρτησης. Η έμμεση ανάθεση πραγματοποιείται συνεχώς κατά τη διάσχιση της αλυσίδας κληρονομικότητας προς αναζήτηση μία μεθόδου. Όταν αυτή βρεθεί, τότε καλείται εντός του πλαισίου (context) του αρχικού αντικειμένου.

Πολυποίκιλη (Miscellaneous)

Το περιβάλλον στο οποίο εκτελείται η JavaScript βρίσκεται ενσωματωμένο στους περιηγητές και έτσι ο κώδικας εκτελείται στα πλαίσια αυτών, προσφέροντας δυνατότητες αλληλεπίδρασης με σελίδες (HTML) όπως το DOM (Document Object Model) αλλά και με στοιχεία που προσφέρουν οι ίδιοι οι περιηγητές μέσω του BOM (Browser Object Model). Τα DOM, BOM αποτελούν ουσιαστικά αυστηρά καθορισμένες διεπαφές επικοινωνίας υπό τη μορφή ενός API, που επιτρέπουν την αλληλεπίδραση ενός προγράμματος JavaScript με διάφορες σελίδες και περιηγητές αντίστοιχα.

NodeJS

Καθώς η δημοτικότητα της γλώσσας αυξήθηκε γεννήθηκε η ιδέα απεξάρτησης των μηχανών εκτέλεσης JavaScript από τους περιηγητές και η φιλοξενία τους πλέον απευθείας από το λειτουργικό σύστημα. Έτσι δημιουργήθηκε η NodeJS που ουσιαστικά υιοθέτησε τη μηχανή εκτέλεσης JavaScript του Chrome (V8 engine), πρόσθεσε σε αυτήν λειτουργίες που έλειπαν όπως η είσοδος-έξοδος (Input/Output) και παρέδωσε στους προγραμματιστές ένα περιβάλλον που μπορεί να υλοποιήσει πολλαπλές λειτουργίες σε γλώσσα JavaScript.

3.3 Συναρτησιακός προγραμματισμός και JavaScript

Ο συναρτησιακός προγραμματισμός σε γενικές γραμμές, όπως προαναφέρθηκε στα δύο προηγούμενα κεφάλαια, δίνει μεγάλη έμφαση στην ανάπτυξη λογισμικού μέσω συναρτήσεων. Αποτελεί εύλογο ερώτημα για πολλούς το γεγονός ότι στη σχολή του προστακτικού προγραμματισμού (σε γλώσσες όπως η C, Java) γίνεται ήδη χρήση συναρτήσεων ή μεθόδων και μάλιστα σε μεγάλο βαθμό. Τι είναι αυτό όμως που διαφοροποιεί τις δύο (2) καταστάσεις μεταξύ τους; Ο συναρτησιακός προγραμματισμός προϋποθέτει μία διαφορετική ιδεολογική προσέγγιση στη λύση ενός προβλήματος. Οι συναρτήσεις δεν αποτελούν απλώς εργαλεία στα οποία εισάγουμε δεδομένα και αναμένουμε εξόδους αλλά, δομούνται με έναν τέτοιο τρόπο, ώστε να επικροτούν την απόκρυψη μέσω ενός αφαιρετικού πλαισίου, της ροής των ελέγχων και διαδικασιών πάνω στα δεδομένα του προβλήματος με απώτερο σκοπό την αποφυγή παρενεργειών (side effects) και την τροποποίηση της κατάστασης (mutation of state) της εφαρμογής.

Για παράδειγμα έστω ότι έχουμε τον παρακάτω κώδικα

```
document.querySelector("#msg").innerHTML = "<h1>Hello World!</h1>";
```

Ο κώδικας αυτός όπως βλέπουμε είναι αρκετά απλός αφού το μόνο που υλοποιεί είναι η εμφάνιση του κειμένου "Hello World!" εντός του html στοιχείου με id = "msg". Όμως παρόλα αυτά υπάρχει ένας μεγάλος βαθμός ανελαστικότητας καθώς όλα τα στοιχεία είναι σταθερά (hardcoded). Αν υποθέσουμε κάποια απαίτηση στην αλλαγή του μηνύματος εκτύπωσης, στην αλλαγή του στυλ (formatting) ή στην αλλαγή του σημείου εμφάνισης τότε προφανώς αυτή δεν μπορεί να ικανοποιηθεί με την παραπάνω σύνταξη. Για να το επιτύχουμε αυτό μία λύση θα ήταν η παραμετροποίηση των προαναφερθέντων παραγόντων και η ενσωμάτωση του στα πλαίσια μιας συνάρτησης. Για παράδειγμα

```
function printMessage(elementId, format, message) {
    document.querySelector(`#${elementId}`).innerHTML =
        `<${format}>${message}</${format}>`;
}

printMessage("#msg", "h1", "Hello World");
```

Παρατηρούμε ότι το παραπάνω κομμάτι κώδικα αποκτά πλέον μία ευελιξία καθώς η συνάρτηση μπορεί να επαναχρησιμοποιηθεί χρησιμοποιώντας διαφορετικές παραμέτρους. Παρόλα αυτά δεν αποτελεί μία αμιγώς λύση συναρτησιακού προγραμματισμού. Ο συναρτησιακός προγραμματισμός προχωράει ένα βήμα παραπάνω σε αφαιρετικό επίπεδο και συνιστά τη χρήση συναρτήσεων ως παράγοντες άλλων συναρτήσεων. Παρακάτω δίνεται ο κώδικας που επιτελεί το ίδιο έργο με τα δύο παραπάνω παραδείγματα, αλλά σε αυτή την περίπτωση διαφαίνεται η συναρτησιακή φύση της ροής του προγράμματος.

```
let printMessage = run(addToDom("msg"), h1, echo);

printMessage("Hello World");
```

Δίχως καμία αμφιβολία η παραπάνω λύση είναι αρκετά διαφορετική από τις προηγούμενες. Παρατηρούμε ότι οι παράμετροι της συνάρτησης **run** αποτελούν ανεξάρτητες συναρτήσεις (**addToDom**, **h1**, **echo**). Το τι ακριβώς επιτελεί η συνάρτηση **run** θα γίνει αντικείμενο μελέτης παρακάτω. Διαισθητικά αντιλαμβανόμαστε ότι η **run** επιστρέφει εκ νέου μία συνάρτηση που περιμένει ως όρισμα ένα αλφαριθμητικό το οποίο θα απεικονίσει σε μία σελίδα. Επίσης διαφαίνεται η διαφορά μεταξύ των δύο μοντέλων προγραμματισμού, του προστακτικού (που ακολουθούν τα δύο πρώτα παραδείγματα) και του δηλωτικού (που ακολουθεί το τελευταίο). Η έκφραση

```
run(addToDom("msg"), h1, echo);
```

δηλώνει τη δημιουργία μίας νέας λειτουργικότητας ή οποία κατασκευάζεται από τρεις (3) υπάρχουσες μικρότερες και συνήθως απλούστερες. Η έκφραση δεν ασχολείται με το πώς θα επιτελέσει μία λειτουργία εμβαθύνοντας σε λεπτομέρειες, αλλά το τι θα κάνει συνδυάζοντας τις τρείς επιμέρους εκφράσεις. Η πολυπλοκότητα κρύβεται πίσω από τις συναρτήσεις, οι οποίες συχνά κατασκευάζονται από άλλες μικρότερες και απλούστερες ακολουθώντας τις τεχνικές του συναρτησιακού προγραμματισμού. Η αρθρωτή αυτή προσέγγιση επιτρέπει στον προγραμματιστή να δημιουργήσει επαναχρησιμοποιούμενο και εύρωστο κώδικα τον οποίο είναι σε θέση να εξηγήσει δομημένα, συνδυάζοντας ή διασπώντας στο μυαλό του τα επιμέρους αρθρώματα που ακολουθούν μια προγραμματιστική συναρτησιακή λογική. Επί του πρακτέου στο παράδειγμα μας οι συναρτήσεις συνδυάζονται χρησιμοποιώντας τη συνάρτηση **run** η οποία επιτρέπει μία αλυσιδωτή προσπέλαση των συναρτήσεων με μία είσοδο και μία έξοδο. Αν θα θέλαμε να

παραλλάξουμε τη λειτουργικότητα της printMessage η συναρτησιακή σύνταξη της μας το επιτρέπει πανεύκολα αλλάζοντας τις λειτουργικότητες των παραμέτρων με άλλες κατασκευασμένες ίδιας φύσεως. Για παράδειγμα η printMessage μπορεί να τροποποιηθεί άμεσα και απλά

```
let printMessage = run(console.log, repeat(3), h2, echo);

printMessage("Have to get Functional");
```

Ο αναγνώστης μπορεί εύκολα να αντιληφθεί το έργο που επιτελεί ή νέα printMessage ξέροντας την παλιά. Στην προκειμένη περίπτωση το κείμενο "<h2>Have to get Functional</h2>" εμφανίζεται τρεις φορές στην κονσόλα. Εάν αντί για **console.log** χρησιμοποιήσουμε την **addToDom("msg")** τότε το κείμενο "<h2>Have to get Functional</h2>" θα εμφανιστεί εντός του πεδίου με id = "msg" και θα ερμηνευτεί ανάλογα από τον περιηγητή.

Η δυνατότητα αυτή του συναρτησιακού προγραμματισμού στηρίζεται σε τέσσερις βασικές αρχές που ακολουθεί το μαθηματικό μοντέλο που περιβάλλει τις συναρτήσεις . Οι αρχές αυτές σε κάποιες γλώσσες (συναρτησιακές) όπως η Haskell είναι εμπεδωμένες σε επίπεδο μεταγλωττιστή (compiler) και ως εκ τούτου ο προγραμματιστής είναι υποχρεωμένος να τις ακολουθεί. Σε άλλες γλώσσες γενικού προσανατολισμού όπως ή JavaScript ή η Python ο προγραμματιστής τις ακολουθεί από μόνος του εάν επιθυμεί να στηριχτεί σε μία συναρτησιακού περιεχομένου λύση. Τα τέσσερα βασικά αυτά σημεία είναι (Atencio, 2016)

- Δηλωτικός Προγραμματισμός (Declarative Programming)
- Αγνές Συναρτήσεις (Pure Functions)
- Αναφορική Διαφάνεια (Referential Transparency)
- Μη Μεταβλητότητα (Immutability)

3.3.1 Ο Συναρτησιακός Προγραμματισμός είναι Δηλωτικός

Ο συναρτησιακός προγραμματισμός όπως προαναφέρθηκε υπάγεται στην κατηγορία του δηλωτικού μοντέλου προγραμματισμού. Αποτελεί ένα μοντέλο που εκφράζει το σύνολο των λειτουργιών ενός προγράμματος δίχως την αποκάλυψη των λεπτομερειών της υλοποίησης. Στην αντιπέρα όχθη, το πιο διαδομένο μοντέλο αποτελεί ο προστακτικός ή διαδικαστικός προγραμματισμός που στις μέρες μας υποστηρίζεται σε

δομημένες και αντικειμενοστρεφής γλώσσες όπως η Java, C#, C++ και άλλες. Ο προστακτικός προγραμματισμός αντιμετωπίζει ένα πρόγραμμα ως μια διαδοχική αλληλουχία εντολών από πάνω προς τα κάτω που αλλάζουν την κατάσταση του συστήματος με σκοπό να υπολογίσουν ένα αποτέλεσμα. Ας δούμε ένα απλό παράδειγμα υλοποίησης σε προστακτική μορφή. Υποθέτουμε ότι θέλουμε να υπολογίσουμε τα τετράγωνα όλων των στοιχείων ενός πίνακα. Παρακάτω βλέπουμε την οικεία λύση που θα υιοθετούσε ένας μεγάλος αριθμός προγραμματιστών

```
let array = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
for (let i = 0; i < array.length; i++) {
    array[i] = Math.pow(array[i], 2);
}

console.log(array);
//-> [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

Η εκδοχή προστακτικού προγραμματισμού υποδεικνύει στον υπολογιστή σε μεγάλος βάθος πως θα υλοποιήσει έναν υπολογισμό (εισαγωγή σε βρόχο επανάληψης, για κάθε στοιχείο αντικατάσταση αυτού με το τετράγωνο του). Η λύση αυτή είναι αρκετά κοινή και σίγουρα είναι η πρώτη που έρχεται στο μυαλό κάθε προγραμματιστή.

Ο δηλωτικός προγραμματισμός από την άλλη διαχωρίζει την περιγραφή του προγράμματος από την αποτίμηση. Εστιάζει στη χρήση εκφράσεων για να περιγράψει τη λογική του προγράμματος δίχως απαραίτητα να προδιαγράφει τον έλεγχο ροής η την αλλαγή των καταστάσεων αυτού. Ένα κλασσικό παράδειγμα δηλωτικών εκφράσεων αποτελούν τα ερωτήματα στην SQL. Τα ερωτήματα περιγράφουν το είδος των αποτελεσμάτων που αναμένει ο προγραμματιστής, τοποθετώντας ένα επίπεδο αφαίρεσης στον τρόπο ανάκτησης των δεδομένων (Atencio, 2016).

Στο παρακάτω παράδειγμα θα δούμε τη συναρτησιακή λύση του προβλήματος τετραγονοποίησης των στοιχείων ενός πίνακα

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].map(function (num) {
    return Math.pow(num, 2);
});

console.log(array);
//-> [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

Συγκρίνοντας τις δύο υλοποιήσεις παρατηρούμε ότι η δηλωτική μορφή απαλείφει τον προγραμματιστή από την ανάγκη συγκέντρωσης του σε λεπτομέρειες (τις οποίες φυσικά πρέπει να γνωρίζει) όπως είναι ο βρόχος και οι δείκτες πινάκων. Το γεγονός αυτό ισοδυναμεί με ένα πρόγραμμα που είναι πιο απλό στην κατανόηση του και φυσικά λιγότερο ευάλωτο σε σφάλματα που προκύπτουν από τον θόρυβο της λεπτομέρειας. Πιο συγκεκριμένα παρατηρούμε ότι στον πίνακα εφαρμόζεται μία συνάρτηση (**map**) η οποία επιτελεί ένα έργο και παράγει ως αποτέλεσμα έναν νέο πίνακα. Επίσης βλέπουμε ότι η **map** δέχεται σαν όρισμα την έκφραση

```
function (num) {
    return Math.pow(num, 2);
}
```

Η έκφραση αυτή αποτελεί μία συνάρτηση στη γλώσσα της JavaScript και όπως εύκολα συμπεραίνει κανείς, δέχεται μία παράμετρο, και επιστρέφει το τετράγωνο αυτής. Συνεπώς αυτό που πραγματοποιεί η συνάρτηση **map** είναι η προσπέλαση των στοιχείων του πίνακα εισόδου ένα προς ένα, η εφαρμογή της παραπάνω συνάρτησης σε αυτά τα στοιχεία ένα προς ένα, η συγκέντρωση των αποτελεσμάτων σε έναν νέο πίνακα και επιστροφή αυτού ως έξοδο του συστήματος. Ουσιαστικά η συνάρτηση **map** δέχεται δύο παραμέτρους, έναν πίνακα και μία συνάρτηση, και επιστρέφει ένα καινούργιο πίνακα που έχει ως τιμές αυτές που ορίζει η συνάρτηση εισόδου. Παρατηρούμε λοιπόν ότι με τη χρήση μίας συνάρτησης μπορούμε να αποκρύψουμε τις λεπτομέρειες υλοποίησης ενός βρόχου. Επιπρόσθετα με τη χρήση μίας άλλης συνάρτησης ορίζουμε το αποτέλεσμα που μας ικανοποιεί. Και φυσικά όλα αυτά συσχετίζονται άμεσα με τα στοιχεία που αναφέρθηκαν στο 1ο κεφάλαιο καθώς οι συναρτήσεις αυτές αποτελούν εκφράσεις λάμδα. Η JavaScript από την έκδοση ECMAScript 6 και έπειτα, έκδοση που χρησιμοποιεί η παρούσα διπλωματική, για να κάνει πιο εμφανής αυτή την ομοιότητα, δίνει στον προγραμματιστή τη δυνατότητα πιο σύντομης αποτύπωσης των συναρτήσεων ακριβώς όπως πράττουν οι εκφράσεις λάμδα. Έτσι η παραπάνω συνάρτηση μπορεί να πάρει τη μορφή

```
num => Math.pow(num, 2)
```

που θυμίζει αρκετά τη μορφή μίας αντίστοιχης έκφρασης λάμδα

```
((((λx.λy.λz.λnum.x y z num) pow) Math) 2)

ή

λnum.(pow Math 2 num)

ή

λnum.<Math.pow(num, 2)>
```

Παρατηρούμε πλέον ότι το ισχυρό μαθηματικό υπόβαθρο των εκφράσεων λάμδα μπορεί να υιοθετηθεί από τη γλώσσα της JavaScript προσδίδοντας της έτσι έναν νέο ορίζοντα (ήδη διευρυμένο) στον τρόπο σκέψης και γραφής της γλώσσας.

3.3.2 Αγνές Συναρτήσεις

Ο συναρτησιακός προγραμματισμός στηρίζεται στο γεγονός ότι το πρόγραμμα αποτελείται από μη μεταβαλλόμενες δομές και στοιχεία όπως είναι οι αγνές συναρτήσεις (**pure functions**). Οι αγνές συναρτήσεις παρουσιάζουν τα παρακάτω χαρακτηριστικά:

- Εξαρτώνται μόνο από την είσοδο και όχι από κρυφή ή εξωτερική κατάσταση που μπορεί να μεταβληθεί κατά την εκτέλεση του προγράμματος
- Δεν τροποποιεί καταστάσεις εκτός των ορίων δικαιοδοσίας της (function scope) όπως η αλλαγή μίας τιμής του global object ή η τροποποίηση ενός οποιουδήποτε αντικειμένου που θα δεχτεί ως όρισμα

Όποια συνάρτηση καταστρατηγεί τα παραπάνω κριτήρια θεωρείται μη αγνή (**impure function**). Η διαδικασία του προγραμματισμού με την αρχή της μη μεταβλητότητας απαιτεί εξοικείωση καθώς οι περισσότεροι προγραμματιστές είμαστε συνηθισμένοι στο μοντέλο του προστακτικού προγραμματισμού που κατά βάση μεταβάλλει τιμές και δομές δεδομένων κατά την εκτέλεση του προγράμματος. Για παράδειγμα η παρακάτω συνάρτηση

```
var counter = 0;
function increment() {
    return ++counter;
}
```

Είναι μη αγνή (**impure**) καθώς η συνάρτηση διαβάζει/τροποποιεί μία μεταβλητή, την counter, που δεν ανήκει στο πεδίο της (function scope). Η εκτέλεση της δηλαδή τροποποιεί την τιμή της counter μεταβάλλοντας έτσι την κατάσταση του καθολικού αντικειμένου της JavaScript (Global Object). Γενικά οι συναρτήσεις παρουσιάζουν παρενέργειες (**side effects**) όταν διαβάζουν ή γράφουν σε εξωτερικές θέσεις. Ένα άλλο παράδειγμα μη αγνής συνάρτησης αποτελεί η Date.now() η οποία παρέχει μεταβαλλόμενη χρονικά έξοδο (εξαρτάται από τον εξωτερικό παράγοντα του χρόνου). Λειτουργίες που προσδιορίζουν μία μη αγνή συνάρτηση είναι

- Η αλλαγή τιμής μίας μεταβλητής ή μίας δομής δεδομένων καθολικά
- Τροποποίηση της αρχικής τιμής μίας παραμέτρου συνάρτησης
- Επεξεργασία εισόδου από χρήστη
- Εξαγωγή εξαίρεσης (exception throwing), εκτός αν γίνει εντός της συνάρτησης
- Εκτύπωση σε συσκευή εξόδου
- Αναζήτηση σε βάσεις δεδομένων

Φυσικά εύλογα κάποιος αναρωτιέται ποια πρακτική χρήση μπορεί να έχουν οι αγνές συναρτήσεις όταν δεν επιτρέπεται ούτε η λειτουργία της εμφάνισης ενός αποτελέσματος στην οθόνη. Οι αγνές συναρτήσεις δύσκολα εφαρμόζονται σε έναν κόσμο που κατακλύζεται από δυναμικές συμπεριφορές και συστήματα. Όμως όπως προαναφέραμε ο συναρτησιακός προγραμματισμός δεν απαγορεύει όλες τις μεταβολές καταστάσεων παρά αποτελεί ένα πλαίσιο ικανό να ελαχιστοποιεί τις μεταβολές αυτές ανάλογα με τις απαιτήσεις του προγραμματιστή. Όσο πιο αγνά γράφει κάποιος τόσο μεγαλύτερα τα οφέλη που μπορεί να αποκομίσει από τον συναρτησιακό προγραμματισμό. Αξίζει να σημειωθεί ότι το επίπεδο αφαίρεσης που προσεγγίζει ο λογισμός λάμδα και συνεπώς ο συναρτησιακός προγραμματισμός είναι απειροστικός. Αυτό σημαίνει ότι όλες οι παραπάνω λειτουργίες μπορούν να θεωρηθούν αγνές σε ένα σύστημα που έχει διαφορετικό σημείο αναφοράς. Η γλώσσα προγραμματισμού Haskell τηρεί αυστηρά τα πρότυπα του συναρτησιακού μοντέλου και δίνει λύση στο πρόβλημα της μη αγνότητας (**impureness**).

3.3.3 Αναφορική Διαφάνεια (Referential Transparency)

Η αναφορική διαφάνεια είναι ένας όρος άμεσα συνυφασμένος με τις αγνές συναρτήσεις. Πρακτικά προσδιορίζει την αγνή συσχέτιση μεταξύ των εισόδων μίας αγνής συνάρτησης και των εξόδων της. Έτσι αφού οι αγνές συναρτήσεις επιστρέφουν αυστηρά το ίδιο αποτέλεσμα για την ίδια είσοδο, μπορούμε να πούμε ότι είναι διαφανής ή αναφορικά διαφανής (**referential transparent**) με την έννοια ότι το αποτέλεσμα είναι ντετερμινιστικό. Για παράδειγμα η συνάρτηση increment που είδαμε πιο πάνω είναι μη αγνή και συνεπώς δεν είναι αναφορικά διαφανής σε αντίθεση με την παρακάτω τροποποιημένη εκδοχή της

```
var increment = counter => counter + 1;
```

Η παραπάνω συνάρτηση είναι αγνή καθώς για την ίδια είσοδο έχουμε με απόλυτη ακρίβεια την ίδια έξοδο. Δεν επηρεάζει εξωτερικές καταστάσεις ούτε επηρεάζεται από τέτοιες. Η αναφορική διαφάνεια είναι κάτι που πρέπει να το επιζητούμε καθώς όχι μόνο κάνει τα προγράμματα μας πιο ευανάγνωστα, αλλά κάνει και πιο εύκολη τη διαδικασία εκλογίκευσης τους αφού το τελικό αποτέλεσμα μπορεί να διακριθεί μέσω αντικατάστασης (substitution). Στο παράδειγμα που ακολουθεί διαφαίνεται η έννοια της αναφορικής διαφάνειας που ενεργοποιεί η χρήση αγνών συναρτήσεων. Έστω ότι έχουμε ένα πρόγραμμα που υπολογίζει τον μέσο όρο των στοιχείων ενός συγκεκριμένου πίνακα

```
let input = [80, 90, 100, 110, 120];
let sum = (total, current) => total + current;
let total = arr => arr.reduce(sum);
let size = arr => arr.length;
let divide = (a, b) => a / b;
let average = arr => divide(total(arr), size(arr));

let result = average(input);
console.log(result);
//-> 100
```

Όλες οι συναρτήσεις του παραπάνω προγράμματος είναι αγνές καθώς για την ίδια είσοδο έχουν πάντα την ίδια έξοδο. Ως εκ τούτου η αναφορικής διαφάνεια επιτρέπει την αντικατάσταση των αποτελεσμάτων τους σε κάθε κλήση τους με το αποτέλεσμα που προκύπτει από τον αρχικό υπολογισμό τους. Αν για παράδειγμα η κλήση **total(input)** επιστρέφει την τιμή 500 τότε μπορούμε να είμαστε βέβαιοι ότι οπουδήποτε στο πρόγραμμα αντικρίζουμε τη συγκεκριμένη κλήση, αυτή μπορεί να αντικατασταθεί από την τιμή 500 δεδομένου φυσικά ότι το input δεν έχει μεταβληθεί κατά τη ροή του προγράμματος. Αυτός

είναι και ένας από τους βασικούς λόγους που ο συναρτησιακός προγραμματισμός προϋποθέτει την αρχή της μη μεταβλητότητας (**Immutability**) που θα αναφερθεί αμέσως μετά. Η αναφορική διαφάνεια και οι αγνές συναρτήσεις γεννάνε ένα άλλο ισχυρό προτέρημα του συναρτησιακού μοντέλου που είναι η μνημόνευση κλήσεων συνάρτησης (**Memoization**) για την οποία θα γίνει αναφορά σε επόμενο κεφάλαιο.

3.3.4 Μη Μεταβλητότητα (Immutability)

Τα αμετάβλητα δεδομένα είναι τα δεδομένα των οποίων η τιμή δεν μπορεί να τροποποιηθεί κατά την εκτέλεση του προγράμματος. Στην JavaScript όπως και σε πολλές άλλες γλώσσες, όλοι οι πρωτόγονοι τύποι δεδομένων (**String**, **Number**, **Boolean**) είναι αμετάβλητοι. Δεν συμβαίνει όμως το ίδιο για τα υπόλοιπα αντικείμενα. Για παράδειγμα οι πίνακες στην JavaScript είναι μεταβλητοί και τροποποιήσιμοι κατά την εκτέλεση ακόμα και αν περαστούν ως παράμετροι σε κλήση κάποιας συνάρτησης. Για παράδειγμα έστω ότι έχουμε την παρακάτω συνάρτηση

```
let sortDesc = arr => arr.sort((a, b) => b - a);
```

Και εισάγουμε ως είσοδο σε αυτή τον πίνακα input

```
let sortDesc = arr => arr.sort((a, b) => b - a);
let input = [1, 2, 3, 4, 5];
let output = sortDesc(input);
console.log(output);
//-> [ 5, 4, 3, 2, 1 ]
```

Τότε το αποτέλεσμα που προκύπτει είναι όπως βλέπουμε ένας πίνακας ταξινομημένος σε φθίνουσα σειρά, όπως δηλαδή είναι και ο στόχος της συνάρτησης sortDesc. Όμως μία πιο προσεκτική μελέτη της συνάρτησης αποκαλύπτει ότι το αποτέλεσμα που εξάγει η συνάρτηση δεν είναι ένας νέος πίνακας ανεξάρτητος από την είσοδο του συστήματος, αλλά είναι η ίδια η είσοδος παραλλαγμένη υπό την εφαρμογή της συνάρτησης Array.sort . Με άλλα λόγια η εκτύπωση της μεταβλητής input μετά την εκτέλεση της sortDesc επιστρέφει

```
console.log(input);
//-> [ 5, 4, 3, 2, 1 ]
console.log(input === output);
//-> true
```

Οι δύο μεταβλητές (input, output) δείχνουν τελικά στις ίδιες θέσεις μνήμης. Οι συναρτήσεις αυτές όπως αναφέρθηκε δεν είναι αγνές, και συνεπώς η χρήση τους εναντιώνεται στις αρχές του συναρτησιακού προγραμματισμού αναιρώντας έτσι τα οφέλη του. Στις περιπτώσεις αυτές είναι σαφώς προτιμότερο η συνάρτηση να επιστρέφει ένα καινούργιο αντικείμενο (στην περίπτωση μας έναν νέο πίνακα) το οποίο είναι στον έλεγχο του προγραμματιστή να το χρησιμοποιήσει καταλλήλως χωρίς να υπεισέρχεται κάποια παρενέργεια (side effect).

3.4 Ενθαρρύνοντας την αποσύνθεση των πολύπλοκων λειτουργιών

Σε υψηλό επίπεδο, θα μπορούσαμε να πούμε ότι ο συναρτησιακός προγραμματισμός είναι η διαδικασία κατακερματισμού ενός προβλήματος σε πολλά μικρότερα, η εύρεση λύσεων για τα μικρά αυτά προβλήματα και τελικά η κατάλληλη σύνθεση των λύσεων με σκοπό την επίλυση του αρχικού προβλήματος. Η δυική αυτή φύση της διαδικασίας προσφέρει στο μοντέλο του συναρτησιακού προγραμματισμού την αρθρωτή του δυνατότητα και συνεπώς την ευελιξία και αποτελεσματικότητα που τελικώς εμφανίζει. Η μονάδα άρθρωσης του μοντέλου που μελετάμε είναι η συνάρτηση. Συνεπώς η συναρτησιακή σκέψη προτρέπει τον προγραμματιστή να σπάει το πρόβλημα του σε μικρότερα, δίνοντας λύση σε αυτά με τη χρήση συναρτήσεων. Αν το πρόβλημα δεν επιλύεται τότε η αποσύνθεση συνεχίζεται μέχρις ότου το πρόβλημα γίνει επιλύσιμο. Όταν τα προβλήματα είναι απλά η δυνατότητα συγγραφής αγνών, ανεξάρτητων συναρτήσεων είναι απλούστερη με αποτέλεσμα να ακολουθείται η αρχή της μοναδικότητας (singularity principle) που αναφέρει ότι ο σκοπός μίας συνάρτησης πρέπει να είναι μοναδικός. Τέλος αφού ο προγραμματιστής διαθέτει ένα πλήθος από αγνές συναρτήσεις πρέπει να είναι σε θέση να τις συνθέσει σωστά για να επιλύσει το τελικό πρόβλημα. Η αναφορική διαφάνεια στην οποία αναφερθήκαμε πιο πάνω επιτρέπει στον προγραμματιστή την απόκτηση τεχνικής διαίσθησης για τους τρόπους με τους οποίους συναρτήσεις διαφορετικών εισόδων και εξόδων μπορούν να δεθούν αποτελεσματικά προσφέροντας ένα ορθό και ερμηνεύσιμο αποτέλεσμα. Παρακάτω θα δούμε ένα παράδειγμα που αναδεικνύει τα προαναφερθέντα Έστω ότι έχουμε την παρακάτω συνάρτηση **Student** που παράγει αντικείμενα μαθητών.

```
const Student = (ssn, firstname, lastname) => ({
    ssn,
    firstname,
    lastname
});
```

Ας υποθέσουμε ότι μια λειτουργικότητα του προγράμματος μας επιστρέφει έναν πίνακα με εκατοντάδες τέτοιους μαθητές. Θέλουμε να υλοποιήσουμε έναν πρόγραμμα το οποίο θα αναζητεί με ένα συγκεκριμένο κριτήριο κάποιους μαθητές και θα εκτυπώνει τα στοιχεία τους στην κονσόλα ή στο DOM.

Μία συναρτησιακή προσέγγιση του προβλήματος θα χώριζε τις αρμοδιότητες σε διαφορετικές μονάδες λειτουργικότητας (συναρτήσεις). Με μία προσεκτική ανάλυση του προβλήματος εύκολα διαπιστώνει κάποιος ότι αυτό μπορεί να χωριστεί σε επιμέρους τμήματα, όπως πχ αναζήτησης, διαμόρφωσης της πληροφορίας εξόδου και τελικά υλοποίησης της απεικόνισης. Το στάδιο της αναζήτησης αναμένει έναν πίνακα με τα αντικείμενα των μαθητών, και εφαρμόζοντας το κατάλληλο κριτήριο θα επιστρέφει έναν μαθητή που θα το ικανοποιεί. Μία πρώτη προσέγγιση θα ήταν η εξής:

```
const find = collection => ssn => collection.filter(student => stu-
dent.ssn === ssn)[0] || Student("dump", "dump", "dump");
```

Η παραπάνω συνάρτηση δέχεται έναν πίνακα με μαθητές και επιστρέφει μία συνάρτηση που είναι έτοιμη να φιλτράρει τον πίνακα με βάση το **ssn** (μοναδικός αριθμός) που θα εισαχθεί σε αυτή. Έτσι αν υποθέσουμε ότι έχουμε έναν πίνακα με μαθητές

```
const collection = [
    Student("1111", "Alonzo", "Church"),
    Student("2222", "Haskell", "Curry"),
    Student("3333", "Alan", "Turing"),
    Student("4444", "Stephen", "Cleene")
];
```

Τότε η κλήση

```
const findBySSN = find(collection);
```

Επιστρέφει τη συνάρτηση **findBySSN** η οποία δέχεται ένα **ssn** και επιστρέφει τον μαθητή που διαθέτει το συγκεκριμένο **ssn**.

```
findBySSN("1111");
//-> { ssn: '1111', firstname: 'Alonzo', lastname: 'Church' }
```

Στο επόμενο στάδιο συγκεντρωνόμαστε στην υλοποίηση διαμόρφωσης της χρήσιμης πληροφορίας. Δημιουργούμε μία συνάρτηση που θα δέχεται έναν μαθητή και θα επιστρέφει έναν αλφαριθμητικό που εμπεριέχει τα κατάλληλα για εμάς στοιχεία. Ας πούμε ότι δημιουργούμε την **makeInfo**.

```
const makeInfo = student => `Student Info {firstname:
${student.firstname}, lastname: ${student.lastname}, ssn:
${student.ssn}}`;
```

Αν καλέσουμε την **makeInfo** με παράμετρο έναν μαθητή θα έχουμε το παρακάτω αποτέλεσμα.

```
makeInfo(Student("3333", "Alan", "Turing"));
//-> Student Info {firstname: Alan, lastname: Turing, ssn: 3333}
```

Συνεπώς διαμορφώσαμε την πληροφορία εξόδου με τον τρόπο που θέλαμε.

Τώρα πρέπει να υλοποιήσουμε μία επιπλέον συνάρτηση που θα απεικονίζει τα δεδομένα στην κονσόλα ή σε ένα html element. Βλέπουμε ότι απαιτούνται δύο διαφορετικές λειτουργικότητες. Οι δύο αυτές λειτουργικότητες μπορούν να εκφραστούν όπως μάθαμε με τη χρήση συναρτήσεων. Ας πούμε ότι δημιουργούμε τις εξής δύο συναρτήσεις.

```
const consoleLogger = info => console.log(info);

const htmlLogger = id => info =>
document.querySelector(`#${id}`).innerHTML = info;
```

Η **consoleLogger** δέχεται ένα αλφαριθμητικό και το εκτυπώνει στην κονσόλα. Η **htmlLogger** δέχεται το id του element στο οποίο θέλουμε να εκτυπώσουμε το αποτέλεσμα και επιστρέφει μία συνάρτηση που αναμένει την πληροφορία εκτύπωσης. Όταν αυτή του δοθεί η εκτύπωση θα λάβει μέρος. Βλέπουμε ότι με τη χρήση παραμέτρων και μάλιστα **curried** συναρτήσεων (συναρτήσεων που λαμβάνουν μία παράμετρο και επιστρέφουν συναρτήσεις που είναι πιο συγκεκριμένες και αναμένουν τις υπόλοιπες παραμέτρους, θα γίνει ανάλυση στα επόμενα κεφάλαια) χτίζουμε ένα αφαιρετικό μοντέλο που προσεγγίζει όχι μόνο τη συγκεκριμένη λύση άλλα ενεργοποιεί τον προγραμματιστή να προσδώσει λύση και σε άλλα διαφορετικά σενάρια κάνοντας μικρές η και καθόλου αλλαγές στον κώδικα του.

Όσον αφορά στο παράδειγμα μας μπορούμε να υλοποιήσουμε επιπρόσθετα άλλη μία συνάρτηση **printInfo** που θα καλεί την **consoleLogger** ή **htmlLogger**. Το σώμα της συνάρτησης φαίνεται παρακάτω

```
const printInfo = printFn => info => printFn(info);
```

Έτσι η **printInfo** μπορεί να κληθεί με τους παρακάτω τρόπους.

```
printInfo(consoleLogger)("Hello FP");
//-> Hello FP
printInfo(htmlLogger("fp"))("Hello FP");
//-> <div id="fp">Hello FP</div>
```

Προσεγγίσαμε το πρόβλημα, δημιουργήσαμε μικρές ανεξάρτητες οντότητες (συναρτήσεις) που αντιμετωπίζουν επιμέρους ευκολότερα προβλήματα που προκύπτουν έπειτα από ανάλυση και τώρα απομένει η συγκόλληση και σύνθεση (composition) των λύσεων με τρόπο τέτοιο ώστε να επιλύεται το αρχικό πρόβλημα. Έστω ότι είχαμε το πίνακα collection του σχήματος … και θέλαμε να εκτυπώσουμε τα στοιχεία του μαθητή με ssn = 2222 στην κονσόλα. Μία λύση θα ήταν

```
const student = findBySSN("2222");
const info = makeInfo(student);
printInfo(consoleLogger)(info);
//-> Student Info {firstname: Haskell, lastname: Curry, ssn: 2222}
```

Φυσικά η λύση αυτή είναι σωστή αλλά δεν είναι ευέλικτη καθώς πιθανή νέα ζητούμενη αναζήτηση και εκτύπωση θα απαιτούσε επανάληψη της διαδικασίας. Ας δούμε μία πιο συναρτησιακή προσέγγιση.

```
const findBySSNAndPrintInConsole = ssn =>
printInfo(consoleLogger)(makeInfo(findBySSN(ssn)));

findBySSNAndPrintInConsole("3333");
//-> Student Info {firstname: Alan, lastname: Turing, ssn: 3333}
```

Η λύση αυτή είναι αρκετά καλύτερη και προσθέτει ένα στάδιο αφαίρεσης ως προς το πεδίο ssn.

Αν παρατηρήσουμε όμως πιο προσεκτικά θα δούμε ότι οι συναρτήσεις που δημιουργήσαμε αλληλοσυνδέονται με την έννοια ότι η είσοδος της μίας αποτελεί έξοδο της άλλης. Το μοτίβο αυτό χρησιμοποιείται κατά κόρον στον συναρτησιακό μοντέλο προγραμματισμού και επιτρέπει τη δημιουργία σύνθετων λειτουργιών συνθέτοντας απλούστερες συναρτήσεις. Η πράξη αυτή ονομάζεται σύνθεση συναρτήσεων (**composition**) και είναι άμεσα συνυφασμένη με τα μαθηματικά. Ανάλυση της συνθετικής πράξης θα γίνει στα επόμενα κεφάλαια, όμως αρκεί να πούμε απλοϊκά ότι εάν έχουμε δύο συναρτήσεις f, g τότε το αποτέλεσμα της σύνθεσης τους παράγει μία νέα συνάρτηση για την οποία ισχύει $f \circ g$ είναι $f \circ g\ (x) = f(g(x))$. Από τον τύπο μπορούμε να γράψουμε τη δική μας **composeTwo** που θα επιστρέφει στη σύνθεση δύο συναρτήσεων. Όπως παρακάτω

```
const composeTwo = (f, g) => x => f(g(x));
```

Και επειδή η πράξη της σύνθεσης όπως θα δούμε είναι προσεταιριστική μπορούμε να την επεκτείνουμε γράφοντας

```
const composeThree = (h, f, g) => composeTwo(h, composeTwo(f, g));
```

Υλοποιήσαμε δηλαδή μία συνάρτηση υψηλής τάξης (**higher order function**) που θα συνθέτει τρεις λειτουργικότητες σε μία. Η συνάρτηση **findBySSNAndPrintInConsole** θα μπορούσε να γραφεί και

```
const findBySSNAndPrintInConsole =
composeThree(printInfo(consoleLogger), makeInfo, find(collection));

findBySSNAndPrintInConsole("1111");
//-> Student Info {firstname: Alonzo, lastname: Church, ssn: 1111}
```

ή αν θέλαμε να εκτυπώσουμε στο DOM

```
const findBySSNAndPrintInDOM =
composeThree(printInfo(htmlLogger("fp")), makeInfo, find(collection));

findBySSNAndPrintInDOM("1111");
//-> <div id="fp">Student Info {firstname: Alonzo, lastname: Church,
ssn: 1111}</div>
```

Βλέπουμε λοιπόν ότι η συναρτησιακή προσέγγιση του προβλήματος, αν και ασυνήθιστη για κάποιον που είναι εξοικειωμένος με τον αντικειμενοστρεφή προγραμματισμό, μπορεί να προσδώσει λύση και μάλιστα αφαιρετική με σχετική ευκολία. Υλοποιώντας μικρές μονάδες συναρτήσεων με ορίσματα δημιουργούμε αφαιρετικά στρώματα τα οποία ξετυλίγονται κατά την κλήση των συναρτήσεων. Θα λέγαμε ότι είναι ένα παιχνίδι μεταξύ αφαιρέσεων και εφαρμογών (**abstractions** and **applications**), οι δύο δηλαδή θεμελιώδεις έννοιες στις οποίες στηρίζεται ο λογισμός λ όπως είδαμε στο προηγούμενο κεφάλαιο.

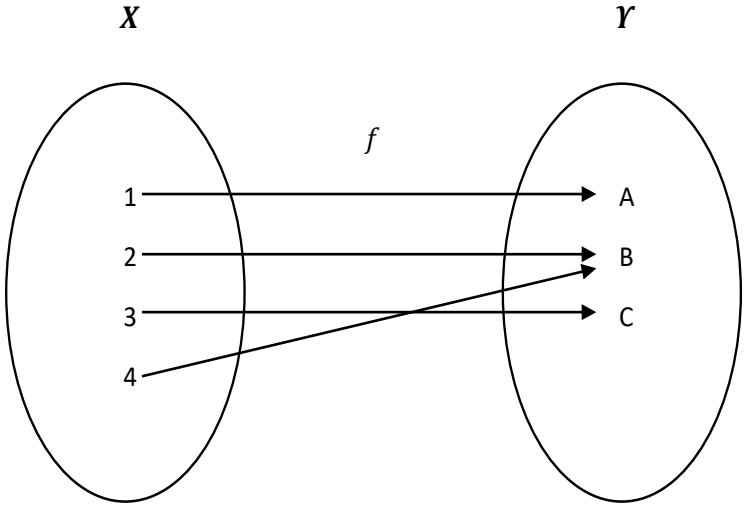
3.5 Συστήματα Τύπων

Αν αναλύσει κάποιος όμως προσεκτικά τη λύση του προηγούμενου προβλήματος θα δει ότι για να βγάλει κάποιος ένα λογικό συμπέρασμα συνθέτοντας τις επιμέρους συναρτήσεις πρέπει αυτές να ταιριάζουν ως προς τους τύπους. Για παράδειγμα εάν μία συνάρτηση περιμένει ως όρισμα ένα αντικείμενο τύπου **Student** και εμείς της δώσουμε έναν αριθμό που είναι τύπου **Number** τότε σίγουρα, μπορούμε να υποθέσουμε εσφαλμένο αποτέλεσμα. Πράγματι αν μελετήσει κανείς τον συναρτησιακό μοντέλο προγραμματισμού θα δει ότι η δήλωση τύπων είναι βασικό συστατικό του. Γλώσσες όπως η **Haskell** ή η **Closure** απαιτούν τη δήλωση τύπων σε εισόδους και εξόδους συναρτήσεων κάτι που δεν κάνει η JavaScript καθώς είναι **dynamically typed** γλώσσα. Η δήλωση τύπων σε μία **statically typed** γλώσσα προσφέρει μία ασπίδα στον προγραμματιστή που του επιτρέπει την ανίχνευση λαθών κατά τον χρόνο μεταγλώττισης. Επιπρόσθετα όμως επειδή ο συναρτησιακός προγραμματισμός μπορεί να γίνει αρκετά πολύπλοκος στην ανάλυση του η δήλωση των τύπων βοηθάει τον προγραμματιστή να ερμηνεύσει νοητικά το τι πάει πού όταν έχει να αντιμετωπίσει αρκετά σύνθετο κώδικα. Η JavaScript δυστυχώς για το πρώτο πρόβλημα δεν μπορεί να δώσει λύση (ανίχνευση λαθών κατά τη μεταγλώττιση), βέβαια το γεγονός ότι δεν δηλώνεις τύπους είναι ταυτόχρονα και προτέρημα λόγω της ευελιξίας που προσφέρει. Ο προγραμματιστής στην JavaScript έχει μεγάλο βαθμό ελευθερίας αλλά παράλληλα έχει και τεράστια ευθύνη όσον αφορά την ορθότητα του αποτελέσματος που παράγει (Kereki, 2017). Για το δεύτερο κομμάτι, αυτό δηλαδή της αποκωδικοποίησης των τύπων με σκοπό την κατανόηση λειτουργικοτήτων, τη λύση προσφέρουν οι ίδιοι οι προγραμματιστές γράφοντας ως σχόλιο την υπογραφή κάθε συνάρτησης πάνω από τη

δήλωση της. Επιπρόσθετα, μία πιο αυστηρή προσέγγιση περιλαμβάνει και την παράθεση δομών ελέγχου των τύπων μέσω του τελεστή **typeof**, σχηματίζοντας έτσι έναν μηχανισμό ενημέρωσης του προγραμματιστή που θα ενεργοποιείται κατά το στάδιο της εκτέλεσης ενός προγράμματος. Ας δούμε όμως αναλυτικά τον μαθηματικό ορισμό της συνάρτησης και την έννοια των τύπων. Αναφέραμε στο εισαγωγικό κεφάλαιο ότι τον συναρτησιακό μοντέλο προγραμματισμού στηρίζεται στην έννοια της συνάρτησης υπό τη μαθηματική της ερμηνεία. Στα μαθηματικά ορίζουμε ως συνάρτηση

$$f(x) = y, \quad x \in X, y \in Y \quad \text{ή} \quad f: X \to Y$$

μία αντιστοίχιση μεταξύ δύο συνόλων X, Y που καλούνται σύνολο ορισμού και σύνολο τιμών, κατά την οποία κάθε στοιχείο του πεδίου ορισμού αντιστοιχίζεται σε ένα και μόνο στοιχείο του πεδίου τιμών.



Σχήμα 3-1Αντιστοιχία Συνόλων

Στο παραπάνω σχήμα το σύνολο $\boldsymbol{X}$ με στοιχεία $\{1, 2, 3, 4\}$ αντιστοιχείται μέσω της συνάρτησης f στα στοιχεία $\{A, B, C\}$ του συνόλου $\boldsymbol{Y}$. Έτσι τα ζευγάρια $\{(1, A), (2, B), (3, C), (4, B)\}$ αποτελούν λύσεις της συνάρτησης $\boldsymbol{f}$ καθώς την ικανοποιούν. Η συνάρτηση $\boldsymbol{f}$ είναι αυστηρά καθορισμένη και απεικονίζει στοιχεία ενός τύπου (συνόλου) σε στοιχεία ενός άλλου τύπου (συνόλου). Όπως εύκολα αντιλαμβάνεται ο

αναγνώστης ο παράγοντας του χρόνου δεν υφίσταται στον μαθηματικό ορισμό μίας συνάρτησης και ως εκ τούτου, αν ορίσουμε μία συνάρτηση $\boldsymbol{g}: \mathbb{Z} \to \mathbb{Z}$ για την οποία $g(1) = 10$ τότε η τελευταία σχέση θεωρείται δεδομένη ανεξαρτήτου χρονικής σφραγίδας. Η δήλωση αυτή σχετίζεται άμεσα με την αγνότητα (**pureness**) μίας προγραμματιστικής συνάρτησης, βασική αρχή πάνω στην οποία στηρίζεται τον συναρτησιακό μοντέλο προγραμματισμού που όπως είδαμε ενθαρρύνει τη συγγραφή αγνών συναρτήσεων. Ο λόγος όπως αντιλαμβανόμαστε είναι η εκμετάλλευση του πανίσχυρου μοντέλου που προσφέρει η επιστήμη των Μαθηματικών. Γράφοντας αγνές συναρτήσεις ένας προγραμματιστής προσπαθεί να προσεγγίσει τον μαθηματικό ορισμό μίας συνάρτησης, γεγονός που θα του ξεκλειδώσει βασικές αρχές των Μαθηματικών οι οποίες θα τον βοηθήσουν να γράψει εύρωστο, επεκτάσιμο και αφαιρετικό κώδικα. Για να επιστρέψουμε στο αρχικό πρόβλημα που αναφέραμε οι τύποι μίας συνάρτησης, τα σύνολα δηλαδή που αντιστοιχεί, αποτελούν κομμάτι του ορισμού της και συνεπώς είναι απαραίτητα. Το ίδιο συμβαίνει και με τις προγραμματιστικές συναρτήσεις. Αν για παράδειγμα ορίσουμε την παρακάτω συνάρτηση

```
const add5 = x => x + 5;
```

τότε εύκολα μπορεί να συμπεράνει κανείς ότι η συνάρτηση **add5** θα πρέπει να δεχτεί σαν παράμετρο έναν αριθμό **Number** και επιστρέφει επίσης έναν αριθμό **Number**. Υπό μαθηματική σκοπιά θα λέγαμε δηλαδή ότι η **add5** είναι μία συνάρτηση με σύνολο ορισμού το **Number** και σύνολο τιμών το **Number**

ή

$$add5: Number \to Number$$

Όπου **Number** εξιδανικεύοντας θα μπορούσαμε να πούμε ότι είναι το $\mathbb{R}$. Ασφαλώς επειδή η JavaScript είναι γλώσσα προγραμματισμού δυναμικής γραφής (dynamically typed) η εφαρμογή της παραπάνω συνάρτησης σε στοιχείο διαφορετικού συνόλου (τύπου) θα προχωρήσει κανονικά, αλλά στο 99% των περιπτώσεων το αποτέλεσμα δεν είναι αυτό που επιζητούμε. Η κλήση δηλαδή

```
add5("FP");
//-> FP5
```

υλοποιεί αντιστοίχιση στα σύνολα $String \rightarrow String$ κάτι το οποίο δεν θα θέλαμε. Γλώσσες στατικής γραφής (statically typed) όπως η Haskell θα πετούσε σφάλμα κατά τη μεταγλώττιση του προγράμματος καθώς ο ορισμός των τύπων μίας συνάρτησης είναι αυστηρός, ακριβώς όπως και στα Μαθηματικά.

Αφού λοιπόν η JavaScript δεν είναι σε θέση να καθορίσει τύπους (σε αντίθεση με διαλέκτους της όπως η Typescript και η Elm, που προσφέρουν αυτή τη δυνατότητα καθώς διαθέτουν στάδιο μεταγλώττισης μέσω transpilers π.χ Babel) αποτελεί καλή πρακτική για τους προγραμματιστές, ακόμα και για αυτούς που γράφουν σε μεγάλο βαθμό αντικειμενοστρεφή κώδικα, να αναγράψουν σε μορφή σχολίου πάνω από τον ορισμό της συνάρτησης ή της μεθόδου, την υπογραφή της. Για παράδειγμα στην περίπτωση της **add5** θα ορίζαμε ως υπογραφή

```
// add5 :: Number -> Number
const add5 = x => x + 5;
```

που δηλώνει ξεκάθαρα το είδος της παραμέτρου που δέχεται η **add5** καθώς και το είδος της παραμέτρου που αυτή επιστρέφει. Ενναλακτικά θα μπορούσαμε να χρησιμοποιήσουμε μία δομή ελέγχου για την αποτροπή μίας αναντιστοιχίας τύπων. Για παράδειγμα

```
// add5 :: Number -> Number
const add5 = x => typeof x === "number" ? x + 5 : console.log(Error("Type of x:
Not a Number"));

add5("felix");
//-> Error: Type of x: Not a Number
```

Φυσικά μία τέτοια προσέγγιση επιβαρύνει αρκετά τη λεγόμενη αναφορική ακεραιότητα που διέπει τον συναρτησιακό προγραμματισμό (λόγω των πολλαπλών δομών ελέγχου που απαιτούνται) και ως εκ τούτου στην παρούσα διπλωματική θα αποφευχθεί.

Η υπογραφή μίας συνάρτησης ακολουθεί ορισμένους κανόνες τους οποίους ορίζει αυστηρά το σύστημα αλγεβρικών τύπων των **Hindley** και **Milner**. Το σύστημα αυτό επηρέασε σε μεγάλο βαθμό τον τρόπο που υλοποιούν τους τύπους τους γνωστές γλώσσες προγραμματισμού εκ των οποίων και η Haskell (Thompson, 1991). Δίχως να γίνει εκτεταμένη ανάλυση, καθώς αυτή θα ξέφευγε από τα πλαίσια που ορίζει το θέμα της παρούσας διπλωματικής, θα αναφέρουμε περιληπτικά τις βασικές αρχές με τις οποίες θα μπορούμε εξάγουμε την υπογραφή οποιασδήποτε συνάρτησης ορισμένης στη γλώσσα που μας ενδιαφέρει, δηλαδή την JavaScript.

Αυτές είναι:

- Το όνομα της συνάρτησης γράφεται πρώτο και ακολουθείται από τον τελεστή :: που μπορεί να ερμηνευτεί ως «έχει τύπους».
- Πιθανοί περιορισμοί του τύπου εισάγονται πριν τον τελεστή =>
- Αν η συνάρτηση είναι μέθοδος τότε εισάγεται ο τελεστής ⇝
- Ο τύπος εισόδου που εισάγεται στη συνάρτηση ακολουθείται από το τελεστή →
- Τέλος αναγράφεται ο τύπος εξόδου της συνάρτησης.

Λαμβάνοντας υπόψιν τα παραπάνω ας αναλύσουμε ξανά το παράδειγμα των μαθητών. Αρχικά κατασκευάσαμε μία συνάρτηση αναζήτησης η οποία δέχεται τύπους μαθητών (**Student**) και αλφαριθμητικών (**String**) και επιστρέφει ένα αντικείμενο μαθητή. Η υπογραφή θα λέγαμε ότι έχει τη μορφή που φαίνεται παρακάτω

```
// find :: ([Student], String) -> Student
const find = collection => ssn => collection.filter(student => stu-
dent.ssn === ssn)[0] || Student("dump", "dump", "dump");
```

ή

$$find\colon ([Student], String) \rightarrow String$$

Με λίγη παρατηρητικότητα παραπάνω όμως θα προσέχαμε ότι η υπογραφή της συνάρτησης **find**, δεν είναι ορθή. Αν υποθέταμε ότι ήταν ορθή τότε η κλήση της **find** με ορίσματα τον πίνακα των μαθητών **students** μαζί με ένα αλφαριθμητικό **x** θα μας επέστρεφε έναν από τους μαθητές του πίνακα εάν το **x** θα ήταν ίσο με το **ssn** του συγκεκριμένου μαθητή, ή αλλιώς θα μας επέστρεφε τον dump Student (ανύπαρκτο μαθητή). Όμως η κλήση αυτή επιστρέφει συνάρτηση

```
find(collection, "1111");
//-> [Function]
```

Ο λόγος που εμφανίζεται η παρούσα συμπεριφορά είναι γιατί ο τρόπος συγγραφής της **find** είναι τέτοιας μορφής που επιτρέπει τη δημιουργία συναρτήσεων σταδιακά καθώς σε αυτή περνάμε ορίσματα ένα ένα. Η τεχνική αυτή, όπως προαναφέραμε, είναι γνωστή ως

currying και εκμεταλλεύεται τα στατικά περιβάλλοντα (**closures**) που οικοδομούνται κατά την εκτέλεση συναρτήσεων, και πηγάζουν από τον τρόπο με τον οποίο καθορίζεται η εμβέλεια των μεταβλητών (**Lexical Scoping**), στην JavaScript (αλλά και σε πολλές άλλες γλώσσες). Η τεχνική αυτή θα μελετηθεί στο επόμενο κεφάλαιο αναλυτικότερα. Επί του παρόντος και αφού αντιλαμβανόμαστε ότι η κλήση της **find** με μια παράμετρο επιστρέφει συνάρτηση, μία ορθότερη υπογραφή της θα ήταν η

```
// find :: [Student] -> (String -> Student)
const find = collection => ssn => collection.filter(student =>
student.ssn === ssn)[0] || Student("dump", "dump", "dump");
```

ή

$$find\colon [Student] \rightarrow (String \rightarrow Student)$$

ή αν απαλείψουμε τις παρενθέσεις

$$find\colon [Student] \rightarrow String \rightarrow Student$$

Η δυνατότητα επιστροφής συναρτήσεων ή η εφαρμογή συναρτήσεων με ορίσματα άλλες συναρτήσεις, δηλαδή η μεταχείριση των συναρτήσεων ως δεδομένα και πολίτες πρώτης κατηγορίας (**fist class citizens**), όπως συχνά αποκαλούμε, αποτελεί βασικό συστατικό του συναρτησιακού προγραμματισμού και υποστηρίζει τις τέσσερις βασικές αρχές του στις οποίες έγινε αναφορά στην αρχή του παρόντος κεφαλαίου.

Στη συνέχεια ορίσαμε τη συνάρτηση **makeInfo.** Η υπογραφή της εν λόγω συνάρτησης διαφαίνεται αρκετά εύκολα και έχει την εξής μορφή:

```
// makeInfo :: Student -> String
const makeInfo = student => `Student Info {firstname:
${student.firstname}, lastname: ${student.lastname}, ssn:
${student.ssn}}`;
```

ή

$$makeInfo\colon Student \rightarrow String$$

Η συνάρτηση δέχεται ως παράμετρο ένα στοιχείο του τύπου **Student** (αντικείμενο μαθητή) και επιστρέφει ένα αλφαριθμητικό από το σύνολο που ορίζει ο τύπος **String**. Το γεγονός ότι η συνάρτηση δέχεται ένα αντικείμενο ως είσοδο την καθιστά ευάλωτη σε

παρενέργειες (side effects) που όπως είδαμε καταστρατηγούν την αρχή της αγνότητας (pureness). Βέβαια η συμπεριφορά μίας συνάρτησης κρίνεται ασφαλώς από τον τρόπο χρήσης αλλά και το σώμα της (function body). Αυτό σημαίνει ότι το γεγονός αυτό από μόνο του δεν εμφανίζει κάποια αρνητική συνέπεια που υποσκάπτει τη συναρτησιακή προγραμματιστή λογική. Στο παρόν παράδειγμα η συνάρτηση δεν επιτελεί κάποια τροποποίηση στα χαρακτηριστικά του αντικειμένου όπως για παράδειγμα αυτή η παραλλαγή της

```
const makeInfo2 = student => (student.firstname = "Random", `Student
Info {firstname: ${student.firstname}, lastname: ${student.lastname},
ssn: ${student.ssn}}`);
```

Η κλήση της **makeInfo2** με όρισμα ένα αντικείμενο τύπου μαθητή σε μια γλώσσα όπως η JavaScript, που χρησιμοποιεί τη γνωστή στρατηγική αποτίμησης (**evaluation strategy**) **"call by value"** θα είχε σαν αποτέλεσμα μία παρενέργεια (side effect). Στην αποτίμηση **"call by value"** η θέση μνήμης του αντικειμένου αντιγράφεται ως τιμή στην παράμετρο κλήσης της συνάρτησης και οποιαδήποτε τροποποίηση του αντικειμένου αντανακλάται στο φαινομενικά αρχικό αντικείμενο (φαινομενικά γιατί πρόκειται για το ίδιο αντικείμενο). Αυτό αντιβαίνει στις αρχές που προσδιορίζουν τον συναρτησιακό μοντέλο προγραμματισμού αφού η συνάρτηση παύει να είναι αγνή, και συνεπώς χάνει τα προτερήματα που αναδεικνύει η μαθηματική θεωρία. Βέβαια όπως προαναφέραμε στο πρώτο κεφάλαιο, ο προγραμματιστής κάποια στιγμή θα κληθεί να κάνει κάποια ενέργεια που θα αλλάξει την κατάσταση (**state**) του προγράμματος του για να παράγει ένα αποτέλεσμα ωφέλιμο για τον κόσμο μας που είναι γεμάτος από καταστάσεις. Ο συναρτησιακός προγραμματισμός δεν εξαιρεί τη δυνατότητα αυτή, ειδάλλως θα μας ήταν αχρείαστος, αλλά ενθαρρύνει τη χρήση της στις περιπτώσεις που πραγματικά χρειάζεται και την υλοποιεί πολύ πιο οργανωμένα. Έτσι για να επιστρέψουμε στο πρόβλημα μας, η **makeInfo** θα λέγαμε ότι είναι μία αγνή συνάρτηση αρκεί ο προγραμματιστής να έχει λίγο προσοχή στο περιβάλλον του προγράμματος καθώς τα αντικείμενα τροποποιούνται και δεν είναι σταθερά. Αυτό σημαίνει ότι η κλήση

```
const alan = Student("3333", "Alan", "Turing");
makeInfo(alan);
//-> Student Info {firstname: Alan, lastname: Turing, ssn: 3333}
```

Θα επιστρέφει πάντα το ίδιο αλφαριθμητικό αρκεί να μην υπάρχει κάποιο άλλο σημείο του προγράμματος που πιθανόν θα τροποποιήσει (mutate) το αντικείμενο **alan.** Σε συναρτησιακές γλώσσες προγραμματισμού η τροποποίηση μίας δομής δεδομένων δεν είναι δυνατή. Οποιαδήποτε αλλαγή σε αυτή συνοδεύεται από τη δημιουργία μία νέας ίδιας δομής, με τις αλλαγές που έχει ζητήσει ο προγραμματιστής. Η JavaScript ακολουθεί το μοντέλο των προστακτικών γλωσσών και ως εκ τούτου μπορεί να υποστηρίξει την παραπάνω δήλωση μόνο κάνοντας κάποιες παραπάνω ενέργειες. Στη συνέχεια είδαμε τη συνάρτηση

```
// printInfo :: (String -> ()) -> String -> ()
const printInfo = printFn => info => printFn(info);
```

Με υπογραφή

$$printInfo: (String \rightarrow ()) \rightarrow String \rightarrow ()$$

Η συνάρτηση αυτή είναι βοηθητική και δέχεται σαν παράμετρο μία συνάρτηση αρχικά. Η κλήση της επιστρέφει μία νέα συνάρτηση που αναμένει ως είσοδο ένα αλφαριθμητικό (**String**) στοιχείο και επιστρέφει το Unit type (Μοναδικός τύπος, στη γλώσσα JavaScript θα λέγαμε ότι επιστρέφει το **undefined**). Έτσι η συνάρτηση

```
const printInfo = printFn => info => printFn(info);
const consoleLogger = info => console.log(info);

// consolePrint :: String -> ()
const consolePrint = printInfo(consoleLogger);
```

consolePrint δέχεται ένα **String** και το εκτυπώνει στην κονσόλα ενώ η

```
const printInfo = printFn => info => printFn(info);
const htmlLogger = id => info =>
document.querySelector(`#${id}`).innerHTML = info;

// htmlPrint :: String -> ()
const htmlPrint = printInfo(htmlLogger("fp"));
```

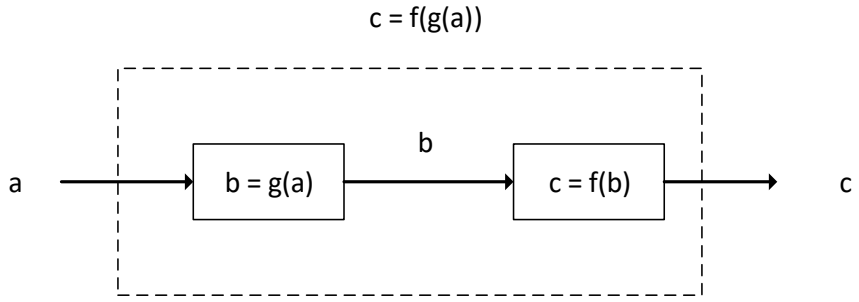
htmlPrint δέχεται ένα **String** και το εισάγει στο DOM.

Βλέπουμε ότι η συνάρτηση **printInfo** μέσω των συναρτήσεων **consoleLogger** και **htmlLogger** προκαλεί παράπλευρες αλλαγές στην κατάσταση και ως εκ τούτου το δεύτερο στάδιο κλήσης της είναι μη αγνό. Παρατηρούμε επίσης πως η **printInfo** γράφτηκε επίσης ως curried για να εισάγει ένα στάδιο αφαίρεσης μεταξύ του τρόπου με τον οποίο

εκτυπώνεται ένα αλφαριθμητικό και το αλφαριθμητικό στο οποίο θα επέμβει ο τρόπος αυτός. Τώρα που αναλύσαμε του τύπους που δέχονται και επιστρέφουν οι συναρτήσεις του προβλήματος μπορούμε να διαισθανθούμε πιο εύκολα τον τρόπο με τον οποίο αυτές θα δεθούν για να εξάγουν ένα επιθυμητό, λογικό αποτέλεσμα. Ο τρόπος σύνδεσης αυτών γίνεται δια μέσου της μαθηματικής σύνθεσης τους. Στο παράδειγμα μας χρησιμοποιήσαμε τη συνάρτηση **composeTwo** με υπογραφή και υλοποίηση

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (f, g) => x => f(g(x));
```

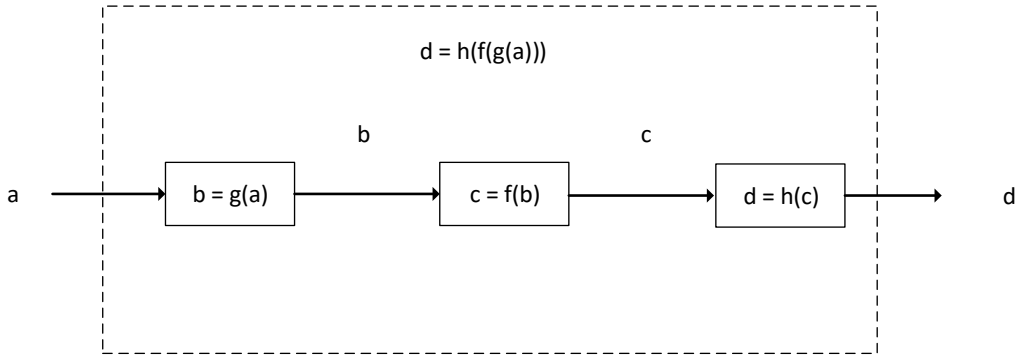
Η συνάρτηση αυτή είναι σαν να δημιουργεί μία ροή μεταξύ στοιχείων ενός τύπου **a**, τα οποία απεικονίζονται σε στοιχεία τύπου **b**, και τέλος αντιστοιχούνται σε στοιχεία του συνόλου **c**.



Σχήμα 3-2 Σύστημα διπλής σύνθεσης

Επεκτείνοντας αυτή τη λογική κατασκευάσαμε την **composeThree** που συνθέτει τρεις συναρτήσεις με τη βοήθεια της **composeTwo.**

```
// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, f, g) => composeTwo(h, composeTwo(f, g));
```



Σχήμα 3-3 Σύστημα τριπλής σύνθεσης

Τελικά καλέσαμε την **composeThree** με ορίσματα τις συναρτήσεις

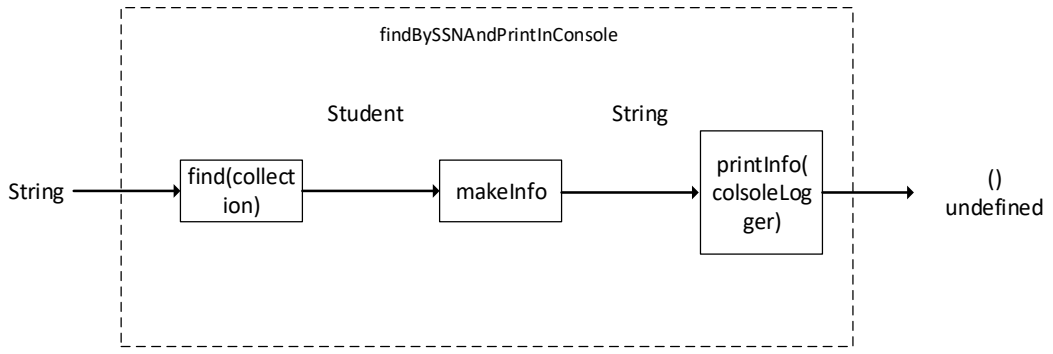
$$find(collection): String \rightarrow Student$$

$$makeInfo: Student \rightarrow String$$

$$printInfo(consoleLogger): String \rightarrow ()$$

```
// findBySSNAndPrintInConsole :: String -> ()
const findBySSNAndPrintInConsole =
composeThree(printInfo(consoleLogger), makeInfo, find(collection));
```

Αν αντικαταστήσουμε τις συναρτήσεις **g, f, h** με τις αντίστοιχες που ορίζει η **findBySSNAndPrintInConsole** τότε το αποτέλεσμα του σχήματος 3.3 θα πάρει τη μορφή



Σχήμα 3-4 Σύστημα σύνθεσης findBySSN

Παρατηρούμε δηλαδή ότι οι είσοδοι και οι έξοδοι των εν λόγω συναρτήσεων ταιριάζουν όπως σε ένα κομμάτι puzzle, έτσι ώστε το αποτέλεσμα της σύνθεσης τους να είναι ορθό και ερμηνεύσιμο. Το πλαίσιο που ορίζει το σύστημα των τύπων μας επιτρέπει συνεπώς να αιτιολογήσουμε (reason) των κώδικα που παράγουμε ως προγραμματιστές, γεγονός που οδηγεί σε εύρωστο και με λιγότερα λάθη πρόγραμμα. Και αφού όπως προαναφέραμε η JavaScript δεν υποστηρίζει εγγενώς τη δήλωση τύπων, όπως οι γλώσσες στατικής γραφής, προσφεύγουμε νοητικά στην οικοδόμηση ενός μοντέλου που λειτουργεί ως αρωγός για τη συγγραφή συναρτησιακού κώδικα και εκφράζεται με ένα απλό τυποποιημένο σχόλιο πάνω από τον ορισμό των συναρτήσεων μας. Φυσικά δεν είναι το ίδιο αποδοτικό αλλά βοηθάει αρκετά όπως είδαμε και στο παράδειγμα μας.

Τέλος, αξίζει να σημειωθεί ότι η αινιγματική λειτουργικότητα της συνάρτησης **run** του προηγούμενου παραδείγματος επιτελεί ακριβώς την ίδια πράξη της σύνθεσης συναρτήσεων (**composition**) που είδαμε και παραπάνω.

Αφού είδαμε λοιπόν πώς περίπου προσεγγίζουμε ένα πρόβλημα με τον συναρτησιακό μοντέλο γραφής, στο επόμενο κεφάλαιο θα μπούμε στην καρδιά του συναρτησιακού προγραμματισμού παρουσιάζοντας διάφορα παραδείγματα και τεχνικές που θα διευκολύνουν τον αναγνώστη να αντιληφθεί τη λογική που κρύβεται από πίσω του. Όπως είπαμε το να αποδώσει κάποιος ένα πρόγραμμα, επωφελούμενος από το μαθηματικό υπόβαθρο που στηρίζει τον συναρτησιακό προγραμματισμό, αποτελεί έργο αρκετά δύσκολο στην αρχή. Η κατεύθυνση της σκέψης μας είναι στραμμένη κατά κύριο λόγο σε αντικείμενα διότι αφενός έχουμε εξοικείωση με τον αντικειμενοστρεφή προγραμματισμό από τα πρώτα βήματα μας ως προγραμματιστές και αφετέρου γιατί ο κόσμος μας προσομοιώνεται σε μεγάλο βαθμό από την αντικειμενοστρεφή προσέγγιση. Βέβαια αν αλλάξουμε την οπτική με την οποία βλέπουμε τον κόσμο μας και τα προβλήματα του θα δούμε ότι τα μαθηματικά κυριαρχούν σε αυτόν. Είτε υπό αφηρημένη προσέγγιση είτε υπό συγκεκριμένη τα μαθηματικά είναι ικανά να μοντελοποιήσουν και να αναλύσουν οποιοδήποτε πρόβλημα. Αν ο άνθρωπος δεν μπορούσε να δει τον κόσμο μας με αυτή τη διάσταση τότε η εξέλιξη της ανθρωπότητας θα ήταν ακόμη σε νηπιακό στάδιο. Αυτήν ακριβώς τη μαθηματική προσέγγιση των προγραμματιστικών προβλημάτων προσπαθεί να ενθαρρύνει τον συναρτησιακό μοντέλο γραφής. Το επόμενο κεφάλαιο λοιπόν θα μας βοηθήσει να δούμε τα προγράμματα μας υπό ένα πρίσμα διαφορετικό από αυτό που τόσα χρόνια πρόσφερε και προσφέρει (φυσικά σε κάποιες περιπτώσεις αποδοτικά) η αντικειμενοστρεφής ανάλυση και σχεδίαση.

4 Συναρτησιακός Προγραμματισμός στην JavaScript (Functional Programming in JavaScript)

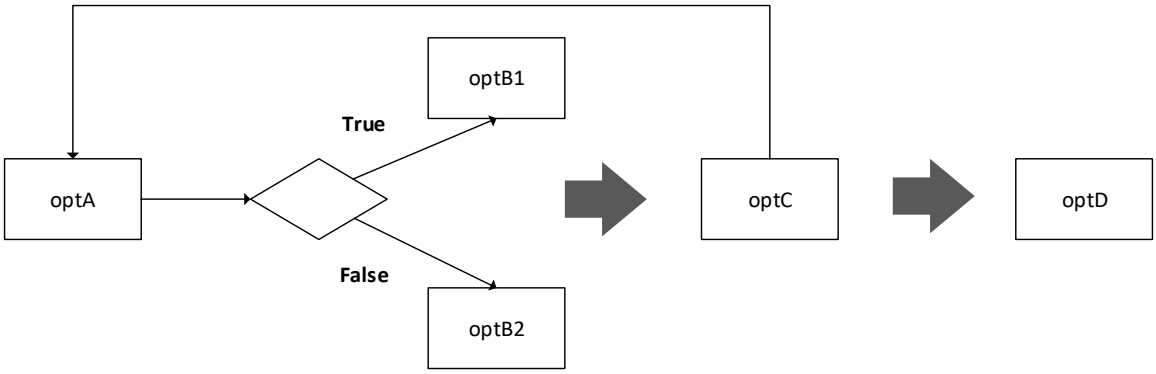
Στο κεφάλαιο αυτό θα αναλυθούν οι πτυχές του συναρτησιακού προγραμματισμού που θα μας βοηθήσουν να κατανοήσουμε καλύτερα και σε μεγαλύτερο βάθος την πρακτική του διάσταση. Συνεπώς θα μάθουμε να προσεγγίζουμε οποιοδήποτε προγραμματιστικό πρόβλημα με έναν νέο τρόπο σκέψης που δεν θα αναιρεί σε καμία περίπτωση το αντικειμενοστρεφές μοντέλο ερμηνείας, καθότι το μοντέλο του συναρτησιακού προγραμματισμού είναι κάθετο (orthogonal) με αυτό, αλλά θα λέγαμε ότι ο νέος αυτός τρόπος σκέψης θα είναι εμπλουτισμένος με ένα πλαίσιο αφαιρετικής αντίληψης των διαδικασιών που απαιτούνται για να οδηγηθούμε σε λύση.

4.1 Κατανοώντας τη ροή της εφαρμογής

Η διαδρομή που ακολουθεί ένα πρόγραμμα κατά το στάδιο της εκτέλεσης του αποτελεί τον έλεγχο ροής του προγράμματος (**control flow**). Αν αναλύσουμε μία εφαρμογή που υλοποιήθηκε στηριζόμενη στο προστακτικό μοντέλο προγραμματισμού θα δούμε ότι ο έλεγχος ροής της είναι ορατός με αρκετά μεγάλη λεπτομέρεια. Όλες οι εντολές ελέγχου συνθηκών, οι βρόχοι επανάληψης και γενικά οι προστακτικές δηλώσεις εκθέτουν όλα εκείνα τα απαραίτητα στοιχεία που καθορίζουν με μεγάλη ευκρίνεια την ακριβή διαδρομή που ακολουθεί ένα πρόγραμμα. Συνήθως το προστακτικό μοντέλο παράγει προγράμματα που ακολουθούν το παρακάτω πρότυπο

```
let loop = optC();
while(loop) {
    let condition = optA();
    if (condition) {
        optB1();
    } else {
        optB2();
    }
    loop = optC();
}
optD();
```

Το διάγραμμα ροής βλέπουμε ότι είναι ευδιάκριτο από τη λεπτομέρεια των εντολών και τον τρόπο με τον οποίο αυτές τοποθετούνται σε μία προστακτική φύση.



Σχήμα 4-1 Προστακτικό μοντέλο γραφής

Από την άλλη πλευρά ο δηλωτικός τρόπος γραφής που εισάγει ο συναρτησιακός προγραμματισμός προσθέτει στα προγράμματα μας ένα επιπλέον επίπεδο αφαίρεσης που απλοποιεί τον έλεγχο ροής τους. Η ροή του προγράμματος είναι σωληνωτή και απαρτίζεται από λειτουργικότητες συναρτήσεων που εφαρμόζονται καταλλήλως στην κατάσταση του προγράμματος και την τροποποιούν ελαχιστοποιώντας το βαθμό εξάρτησης της κατάστασης με τις λειτουργικότητες που εφαρμόζονται σε αυτή. Τα προγράμματα τείνουν να ακολουθούν το παρακάτω πρότυπο γραφής

```
optA().optB().optC().optD();
```

και συνεπώς ο έλεγχος της ροής της εφαρμογής αποκτά μινιμαλιστικό χαρακτήρα και παραπέμπει σε σωλήνωση

Σχήμα 4-2 Δηλωτικό μοντέλο γραφής

Η αλυσιδωτή αυτή διάταξη περιγράφει αποδοτικά την οργανωτική συνοχή που διακατέχει τα προγράμματα που αναπτύσσονται κάτω από τον δηλωτικό τρόπο σύνταξης.

4.2 Αλυσίδωση Μεθόδων (Method Chaining)

Η αλυσίδωση μεθόδων αποτελεί ένα μοτίβο που έχει τις ρίζες του στον αντικειμενοστρεφή προγραμματισμό και επιτρέπει την κλήση πολλαπλών μεθόδων με μία δήλωση. Όταν οι μέθοδοι αυτοί εφαρμόζονται στο ίδιο αντικείμενο τότε η τεχνική αυτή είναι γνωστή ως **method cascading.** Αν και το μοτίβο αυτό εμφανίζεται αρκετά συχνά στον αντικειμενοστρεφή προγραμματισμό, υπό βασικές προϋποθέσεις, μπορεί να χρησιμοποιηθεί και στη συναρτησιακή προσέγγιση. Η προϋπόθεση αυτή όπως εύλογα μπορεί να αντιληφθεί κανείς δεν είναι άλλη από την τήρηση της μη μεταβλητότητας των δομών που τροποποιούμε. Με λίγα λόγια η τροποποίηση ενός αντικειμένου η μίας σύνθετης δομής μέσω μίας μεθόδου, πρέπει να συνοδεύεται από την παραγωγή μίας νέας δομής τα στοιχεία της οποία παραμένουν κοινά πλην αυτών που τροποποιούνται. Η αρχή αυτή πρέπει να ακολουθείται για να είμαστε σε θέση να ικανοποιήσουμε τους κανόνες που ορίζουν τα μαθηματικά πίσω από τον συναρτησιακό προγραμματιστικό μοντέλο. Ο κώδικας μας συνεπώς επωφελείται και η διαδικασία αιτιολογίας (**reasoning**) απλοποιείται.

Παρακάτω θα δούμε ένα παράδειγμα μετατροπής αλφαριθμητικών (**String**) στην JavaScript με την τεχνική της αλυσίδωσης μεθόδων. Ο τύπος **String** στην JavaScript είναι πρωτόγονος και ως εκ τούτου αμετάβλητος. Αυτό σημαίνει ότι οποιαδήποτε τροποποίηση της τιμής ενός αλφαριθμητικού οδηγεί στην παραγωγή νέου που ασφαλώς αντανακλά τη μεταβολή. Το γεγονός αυτό καθιστά ιδανική τη χρήση του τύπου των αλφαριθμητικών, και γενικά όλων των πρωτόγονων τιμών, στη συναρτησιακή προσέγγιση της JavaScript.

```
console.log("Functional Programming in Javascript".substring(0,
22).toLocaleLowerCase().concat(" is fun"));
//-> functional programming is fun
```

Παραπάνω βλέπουμε τη μετατροπή του αλφαριθμητικού "*Functional Programming in JavaScript*" με χρήση της τεχνικής της αλυσίδωσης μεθόδων. Αξίζει να σημειωθεί ότι η JavaScript αντιμετωπίζει τα αλφαριθμητικά ως πρωτόγονα αλλά επιτρέπει τη χρήση μεθόδων σε αυτά μέσω της εσωτερικής μετατροπής σε αντικείμενα τύπου **String** (διαδικασία **autoboxing**). Η πρόταση αυτή ευθυγραμμίζεται άμεσα με την τοποθέτηση που έγινε πιο πάνω και αναφέρει ότι η αλυσίδωση μεθόδων στον συναρτησιακό προγραμματισμό επιτρέπεται με την προϋπόθεση ότι πιθανή τροποποίηση συνοδεύεται

από τη δημιουργία ενός νέου αντικειμένου. Αυτό ακριβώς δηλαδή που κάνει η JavaScript για εμάς όσον αφορά στους πρωτόγονους τύπους δεδομένων.

Η κλήση substring επεμβαίνει στο αλφαριθμητικό που δέχεται έμμεσα (implicit) μέσω του this και μετατρέπει το "*Functional Programming in JavaScript*" σε "*Functional Programming*". Στη συνέχεια το νέο αλφαριθμητικό προωθείται στη μέθοδο toLowerCase που επιστρέφει ένα νέο αλφαριθμητικό με όλους τους χαρακτήρες πεζούς. Τέλος η μέθοδος **concat** επιστρέφει ένα νέο αλφαριθμητικό που παράγεται από την πράξη της ένωσης στις άμεσες και έμμεσες παραμέτρους που δέχεται. Στην περίπτωση μας έμμεση παράμετρος είναι το αλφαριθμητικό "*functional programming*" και άμεση το *" is fun*". Το τελικό αποτέλεσμα είναι ένα νέο αλφαριθμητικό με τιμή "*functional programming is fun*". Η ίδια εκδοχή του προγράμματος σε μία καθαρά συναρτησιακή υλοποίηση θα είχε τη μορφή που φαίνεται παρακάτω.

```
concat(toLocaleLowerCase(substring("Functional Programming in
Javascript", 0, 22)), " is fun");
//-> functional programming is fun
```

Η παραπάνω υλοποίηση ακολουθεί κατά γράμμα τη συναρτησιακή προσέγγιση που ορίζεται από τη συναρτησιακή ανάλυση και σχεδίαση. Δεν παρουσιάζει παράπλευρες αλλαγές και όλες οι παράμετροι δηλώνονται άμεσα. Παρόλα αυτά μπορούμε να πούμε ότι η δεύτερη υλοποίηση δεν είναι το ίδιο εκφραστική και εύκολα κατανοητή με την πρώτη. Ως εκ τούτου η τεχνική της αλυσίδωσης των μεθόδων σε δομές δεδομένων ορισμένες σε γλώσσες προσανατολισμένες στην αντικειμενοστρέφεια, όπως η JavaScript, είναι επιτρεπτή και σε αρκετές περιπτώσεις πιο εκφραστική. Σε γλώσσες καθαρά συναρτησιακές η αντίστοιχή λειτουργία της αλυσίδωσης, επιτελείται διά μέσου της σύνθεσης συναρτήσεων (βλ. **composeTwo**, **composeThree**).

Παρακάτω θα δούμε ένα παράδειγμα το οποίο υπογραμμίζει τη σημασία της χρήσης αμετάβλητων αντικειμένων στον συναρτησιακό προγραμματισμό. Θα δούμε πώς επιτελεί την αλυσίδωση μεθόδων η αντικειμενοστρεφής προσέγγιση και πώς η συναρτησιακή παρουσιάζοντας τρόπους υλοποίησης.

4.3 Αντικειμενοστρεφής Ανάλυση

Έστω ότι έχουμε την παρακάτω κλάση που παράγει αντικείμενα τύπου **Flock** τα οποία αντιπροσωπεύουν σμήνη πουλιών.

```
class Flock {
    constructor(n) {
        this.members = n;
    }

    join(other) {
        this.members += other.members;
        return this;
    }

    breed(other) {
        this.members *= other.members;
        return this;
    }

    printMembers() {
        console.log(`Member Quantity of flock: ${this.members}`);
    }

}
```

Ο κατασκευαστής της κλάσης δέχεται έναν αριθμό που απεικονίζει τον αριθμό των μελών του σμήνους και τον αποθηκεύει στη μεταβλητή **members.** Ο αριθμός αυτός θα λέγαμε περιγράφει την κατάσταση (state) του αντικειμένου ανά πάσα χρονική στιγμή. Η κλάση επίσης ορίζει τις μεθόδους **join**, **breed** και **printMembers** οι οποίες μεταβάλλουν την κατάσταση των αντικειμένων με ορισμένο τρόπο. Πιο ειδικά η μέθοδος **join** δέχεται ως άμεση παράμετρο ένα άλλο αντικείμενο τύπου σμήνους και επηρεάζει την τιμή των μελών στο έμμεσα εμπλεκόμενο αντικείμενο (μέσω **dot notation**) πραγματοποιώντας την πράξη της πρόσθεσης. Η μέθοδος **breed** επίσης δέχεται ως άμεση παράμετρο ένα άλλο αντικείμενο τύπου σμήνους και μεταβάλλει την τιμή των μελών στο έμμεσα εμπλεκόμενο αντικείμενο (μέσω dot notation) πραγματοποιώντας την πράξη του πολλαπλασιασμού. Η **printMembers** εκτυπώνει το πλήθος των μελών του σμήνους από το οποίο καλείται (dot notation). Παρατηρούμε ότι οι μέθοδοι που μεταβάλλουν την τιμή των μελών επιστρέφουν τη μεταβλητή **this** αφού υλοποιήσουν του υπολογισμούς επιτρέποντας έτσι την εφαρμογή

της αλυσίδωσης. Επίσης περιττό να αναφέρουμε ότι και οι τρεις (3) μέθοδοι μεταβάλλουν κάποια κατάσταση, είτε των αντικειμένων (mutation) είτε του εξωτερικού περιβάλλοντος (IO), πράγμα λογικό αφού η λύση σχεδιάστηκε με βάση την αντικειμενοστρέφεια, και συνεπώς η προσπάθεια συγχώνευσης των μεθόδων σε συναρτησιακά μοντέλα γραφής θα καταλήξει σε σφάλματα δίχως τις απαραίτητες προεργασίες. Έστω λοιπόν ότι έχουμε τρία (3) σμήνη πουλιών τα οποία αλληλοεπιδρούνε μεταξύ τους, ή πιο σωστά επηρεάζουν τον αριθμό των μελών του σμήνους **FlockA** με το παρακάτω τρόπο.

```
const flockA = new Flock(10);
const flockB = new Flock(12);
const flockC = new Flock(2);
flockA.breed(flockB).join(flockA.breed(flockC)).printMembers();
```

Αν προσπαθήσουμε να ακολουθήσουμε τον κώδικα για να μαντέψουμε τον αριθμό που θα εκτυπώσει η **printMembers** θα δούμε ότι εκ πρώτης όψεως η λύση είναι εύκολη, και πράγματι είναι αν τηρηθούν οι αρχές που ορίζει ο συναρτησιακός προγραμματισμός, αλλά κατά 99% θα μαντέψουμε λάθος αποτέλεσμα.

Βήμα βήμα έχουμε **flockA** = 10, **flockB** = 12, **flockC** = 2. Η δήλωση **flockA.breed(flockB)** ορίζει τον αριθμό των μελών του σμήνους **flockA** σε 10 * 12 = 120. Στη συνέχεια καλείται η **join** με παράμετρο το **flockA.breed(flockC)** δηλαδή θα λέγαμε 120 + 120 * 2 = 360. Αν εκτελέσουμε όμως την τελευταία γραμμή παρατηρούμε

```
flockA.breed(flockB).join(flockA.breed(flockC)).printMembers();
//-> Member Quantity of flock: 480
flockA.printMembers();
//-> Member Quantity of flock: 480
```

ότι ο αριθμός των μελών του σμήνους A έχει μεταβληθεί σε 480. Και πράγματι η δήλωση **flockA.breed(flockC)** μεταβάλλει τον αριθμό σε 240 και στη συνέχεια αυτός προστίθεται με το υποτιθέμενο αρχικό ποσό που λόγω της τροποποίησης (**mutation**) είναι πλέον 240. Δηλαδή 240 + 240 = 480. Το πρόγραμμα αυτό υπολογίσαμε ότι θα έπρεπε να επιστρέφει 360 έχοντας κατά νου ότι τροποποιούμε την κατάσταση του **flockA**, όμως επιστρέφει τελικά 480 λόγω της μη ορατής μεταβολής, και στην ιδανική και εύκολα ανιχνεύσιμη μορφή του θα έπρεπε να επιστρέφει 10 * 12 + 10 * 2 = 140.

Βλέπουμε λοιπόν ότι η μεταβολή της κατάστασης εμποδίζει την ομαλή και εύκολη εξαγωγή συμπερασμάτων στη διαδικασία αιτιολογίας του κώδικα μας καθώς η μεταβολή αυτή μπορεί να γίνει απρόσμενα.

Παρακάτω θα δούμε δύο (2) υλοποιήσεις του ίδιου προβλήματος υπό συναρτησιακό πρίσμα. Η πρώτη θα κάνει χρήση της αλυσίδωσης μεθόδων που όπως προαναφέραμε αν και μοτίβο του αντικειμενοστρεφούς προγραμματισμού, υπό ορισμένες συνθήκες μπορεί να χρησιμοποιηθεί και στη συναρτησιακή προσέγγιση, και ο δεύτερος τρόπος είναι αμιγώς συναρτησιακός και διαχωρίζει πλήρως τα δεδομένα με τις λειτουργίες (συναρτήσεις) που τα συνοδεύουν.

Ας δούμε την πρώτη υλοποίηση

```
// Flock :: Number -> Flock
const Flock = n => ({
    members: n,
    // join :: Flock ↝ Flock -> Flock
    join: function (other) {
        return Flock(this.members + other.members);
    },
    // breed :: Flock ↝ Flock -> Flock
    breed: function (other) {
        return Flock(this.members * other.members);
    },
    // printMembers :: Flock ↝ () -> ()
    printMembers: function () {
        console.log(`Member Quantity of flock: ${this.members}`);
    }
});
```

Στην προσέγγιση αυτή κάθε μεταβολή της κατάστασης ενός αντικειμένου τύπου **Flock** δια μέσω των μεθόδων **join**, **breed** συνοδεύεται από τη δημιουργία ενός νέου αντικειμένου με την ενημερωμένη κατάσταση. Έτσι αν καλέσουμε την αλυσιδωτή έκφραση που χρησιμοποιήσαμε προηγουμένως το αποτέλεσμα που προκύπτει είναι πολύ πιο εύκολα ερμηνεύσιμο.

```
const flockA = Flock(10);
const flockB = Flock(12);
const flockC = Flock(2);
flockA.breed(flockB).join(flockA.breed(flockC)).printMembers();
//-> Member Quantity of flock: 140
flockA.printMembers();
//-> Member Quantity of flock: 10
```

Βλέπουμε λοιπόν ότι επειδή ακριβώς η συναρτησιακή υλοποίηση φροντίζει να συνοδεύει τις μεταβολές που την παραγωγή νέων αντικειμένων, η αιτιολόγηση του κώδικα γίνεται σε ένα μεγάλο βαθμό αποδοτικότερη (Lonsdorf, 2015). Και όλα αυτά γιατί οι συναρτήσεις, αν και μέθοδοι λόγω της τεχνικής της αλυσίδωσης, στο παρόν πρόβλημα θεωρούνται αγνές. Έτσι η αναφορική ακεραιότητα του συστήματος αυξάνεται με αποτέλεσμα η ερμηνεία του να γίνεται σημαντικά ευκολότερη. Για παράδειγμα παρατηρούμε ότι το **flockA** παραμένει σταθερό σε όλη τη ροή της εκτέλεσης. Η προϋπόθεση αυτή είναι δεδομένη στον συναρτησιακό προγραμματισμό και συνεπώς ο προγραμματιστής που γνωρίζει το μοντέλο, αυτομάτως αναγνωρίζει ότι οι εντολές ανάθεσης είναι ορατές και σταθερές. Η αμιγώς συναρτησιακή υλοποίηση που θα δούμε παρακάτω, απεμπλέκει τις λειτουργικότητες της δομής **Flock** από την κατάσταση της σε μια προσπάθεια να δημιουργήσει αφαιρετικές συναρτήσεις. Παρόλο που όπως είδαμε αρχικά η συναρτησιακή αλυσίδωση δημιουργεί έναν περισσότερο εύκολα ερμηνεύσιμο και αναγνωρίσιμο κώδικα, η συναρτησιακή προσέγγιση κατακρίνει την έμμεση ανάθεση παραμέτρων σε κλήσεις συναρτήσεων (όπως τα **Flock** που περνάνε μέσω της μεταβλητής **this**, μεθόδων δηλαδή).

Μία αμιγώς συναρτησιακή υλοποίηση θα είχε την εξής αρχική μορφή

```
// Flock :: Number -> Flock
const Flock = n => ({
    members : n,
});

// joinFlocks :: (Flock, Flock) -> Flock
const joinFlocks = (fa, fb) => Flock(fa.members + fb.members);

// breedFlocks :: (Flock, Flock) -> Flock
const breedFlocks = (fa, fb) => Flock(fa.members * fb.members);

// printMembersOfAFlock :: Flock -> ()
const printMembersOfAFlock = f => console.log(`Member Quantity of
flock: ${f.members}`);
```

Και η αντίστοιχη δήλωση αλυσίδωσης στο προηγούμενο παράδειγμα θα παρουσιαζόταν ως

```
const flockA = Flock(10);
const flockB = Flock(12);
const flockC = Flock(2);

printMembersOfAFlock(joinFlocks(breedFlocks(flockA, flockB),
breedFlocks(flockA, flockC)));
//-> Member Quantity of flock: 140
```

Η παραπάνω υλοποίηση είναι σαφώς έγκυρη και τηρεί τις αρχές της συναρτησιακής σχεδίασης. Όμως από τη φύση του ο συναρτησιακός προγραμματισμός και ειδικότερα οι συναρτήσεις τείνουν πάντα να προσθέτουν επίπεδα αφαίρεσης έτσι ώστε να προβλέπουν πιθανές αλλαγές στο τρόπο υλοποίησης ενός προβλήματος. Ο συναρτησιακός προγραμματισμός θα λέγαμε είναι εκ φύσεως επεκτάσιμος, τροποποιήσιμος και εύρωστος, χαρακτηριστικά τα οποία αντλεί από τη μαθηματική του υπόσταση και τον λογισμό λάμδα. Συνεπώς μία άλλη εκδοχή αποδοτικότερη θα ήταν η παρακάτω:

```
// Flock :: Number -> Flock
const Flock = n => ({
    members: n,
});

// get :: String -> a -> b
const get = attr => obj => obj[attr];

// getMembers :: a -> b
const getMembers = get("members");

// add: (Number, Number) -> Number
const add = (x, y) => x + y;

// multiply: (Number, Number) -> Number
const multiply = (x, y) => x * y;

// createJoinedContainer :: (a -> b) -> (b -> c) -> ((c, c) -> a) ->
(b, b) -> b
const createJoinedContainer = wrapper => attrFn => joinFn => (objA,
objB) => wrapper(joinFn(attrFn(objA), attrFn(objB)));

// joinFlocks :: (Flock, Flock) -> Flock
const joinFlocks = createJoinedContainer(Flock)(getMembers)(add);

// breedFlocks :: (Flock, Flock) -> Flock
const breedFlocks =
createJoinedContainer(Flock)(getMembers)(multiply);

// printAttr :: (b -> c) -> a -> ()
const printAttr = attrFn => obj => console.log(`Member Quantity of
flock: ${attrFn(obj)}`);

// printMembersOfAFlock :: Flock -> ()
const printMembersOfAFlock = printAttr(getMembers);
```

Και η αντίστοιχη δήλωση αλυσίδωσης στο προηγούμενο παράδειγμα θα παρουσιαζόταν ως

```
const flockA = Flock(10);
const flockB = Flock(12);
const flockC = Flock(2);

printMembersOfAFlock(joinFlocks(breedFlocks(flockA, flockB),
breedFlocks(flockA, flockC)));
//-> Member Quantity of flock: 140
```

Βλέπουμε ότι αυτή η εκδοχή, αν και παρατραβηγμένα αναλυτική για το παράδειγμα που μελετάμε, επεκτείνει αφαιρετικά τη λύση του προβλήματος δημιουργώντας όσο το δυνατόν πιο γενικές συναρτήσεις. Οι πιο ειδικές λειτουργικότητες παράγονται από την εφαρμογή των γενικών συναρτήσεων με τα κατάλληλα ορίσματα, που πολλές φορές όπως βλέπουμε και στο παράδειγμα, είναι και οι ίδιες συναρτήσεις. Δημιουργούμε δηλαδή συναρτήσεις υψηλής τάξης (**higher order functions**) και στη συνέχεια παράγουμε από αυτές ειδικές συμπεριφορές τις οποίες συνθέτουμε και καταλήγουμε στο επιθυμητό αποτέλεσμα. Οι συναρτήσεις υψηλής τάξης αντικαθιστούν επάξια τις αρχές της κληρονομικότητας και του πολυμορφισμού που παρουσιάζει ο αντικειμενοστρεφής προγραμματισμός και μάλιστα σε αντίθεση με τον τελευταίο, ο συναρτησιακός προγραμματισμός εξ ορισμού συνθέτει και ακολουθεί πιστά το ρητό που είναι γνωστό στη σχολή της αντικειμενοστρέφειας "Prefer Composition over Inheritance" (Wikipedia/Composition_over_inheritance, n.d.)

4.4 Λίστες και Αλυσίδωση (List Chaining)

Η μαζική προσπέλαση δεδομένων ίδιου τύπου με τον ίδιο τρόπο είναι αρκετά συχνή πρακτική για έναν προγραμματιστή. Η αποθήκευση των στοιχείων προς προσπέλαση συγκεντρώνεται αρχικά σε μία δομή, όπου στη συνέχεια γίνεται επεξεργασία τους μέσω μίας επαναληπτικής δήλωσης και εξάγεται το τελικό αποτέλεσμα των στοιχείων σε μία δομή του ιδίου τύπου. Η γλώσσα της JavaScript, επηρεασμένη σε μεγάλο βαθμό από την **LISP** (List processing), δεν προσφέρει εγγενώς μεγάλη ποικιλία σε γνωστές δομές δεδομένων αλλά ενθαρρύνει τον προγραμματιστή στη χρήση της δομής των πινάκων (λίστες). Φυσικά ο ορισμός του αντικειμένου και ο τρόπος με τον οποίο η JavaScript τα υλοποιεί, πρακτικά είναι σαν λεξικογραφικοί ή συσχετιστικοί πίνακες (βλ. associative

arrays php, dictionaries python), προσφέρει μεγάλη ευελιξία ενεργοποιώντας τη δυνατότητα υλοποίησης οποιασδήποτε δομής δεδομένων.

Προσανατολισμένη στους πίνακες λοιπόν, η JavaScript προσφέρει ένα οπλοστάσιο μεθόδων, που διευκολύνουν τον προγραμματιστή στην οργανωμένη και αφαιρετική επεξεργασία τους με τη χρήση συναρτήσεων υψηλής τάξης. Όπως θα δούμε μάλιστα, η προσέγγιση που ακολουθούν οι εν λόγω μέθοδοι είναι συναρτησιακής φύσεως που τηρούν πιστά τις αρχές που αναλύθηκαν στο προηγούμενο υποκεφάλαιο και υποστηρίζουν την τεχνική της αλυσίδωσης.

4.4.1 Συνάρτηση map

Έστω ότι έχουμε έναν πίνακα με στοιχεία ενός συνόλου A και τον συμβολίζουμε με $[A]$. Τότε η συνάρτηση **map** δέχεται ως παράμετρο μία συνάρτηση f που αντιστοιχεί στοιχεία του συνόλου A σε στοιχεία του συνόλου B, δηλαδή $f: A \to B$, και επιστρέφει έναν νέο πίνακα με στοιχεία του συνόλου B, $[B]$ που προκύπτουν από εφαρμογή της $f, \forall\, Ai \in A$. Η υπογραφή της μεθόδου έχει την εξής μορφή

$$\boldsymbol{Array.prototype.map}: \boldsymbol{Array} \rightsquigarrow (\boldsymbol{a} \to \boldsymbol{b}) \to \boldsymbol{Array}$$

Ενώ η υπογραφή της συνάρτησης

$$\boldsymbol{map}: \big((\boldsymbol{a} \to \boldsymbol{b}), [\boldsymbol{a}]\big) \to [\boldsymbol{b}] \text{ ή } \boldsymbol{map}: (\boldsymbol{a} \to \boldsymbol{b}) \to [\boldsymbol{a}] \to [\boldsymbol{b}]\ (\boldsymbol{curried})$$

Ας δούμε όμως ένα πρακτικό παράδειγμα που εκθέτει την εφαρμογή της εν λόγω συνάρτησης. Έστω ότι έχουμε μία λίστα από αντικείμενα τύπου **Person** όπως φαίνεται πιο κάτω.

```
const Person = (name, age) => ({
    name,
    age
});

let p1 = Person("Alonzo Church", 44);
let p2 = Person("Haskell Curry", 21);
let p3 = Person("Alan Turing", 25);
let p4 = Person("Stephen Cleene", 33);

const arr = [p1, p2, p3, p4];
```

Και θέλουμε να εξάγουμε τα ονόματα των ανθρώπων σε έναν άλλο πίνακα. Τότε η προστακτική μορφή της λύσης θα ήταν η χρήση μία δομής επανάληψης όπως είδαμε στο παράδειγμα της εισαγωγής. Η **συνάρτηση υψηλής τάξης**, map, μας επιτρέπει να αφαιρέσουμε τη λογική της υλοποίησης και να συγκεντρωθούμε στο τι θέλουμε να κάνουμε με τα δεδομένα της λίστας μας. Η δυνατότητα αυτή σε συνδυασμό με το γεγονός ότι η μέθοδος-συνάρτηση τηρεί την αρχή της μη μεταβλητότητας κάνει ιδανική τη χρήση της σε συναρτησιακά προσκείμενες υλοποιήσεις. Στην πραγματικότητα, όπως θα δούμε αργότερα, η συνάρτηση **map** έχει τις ρίζες της στη μαθηματική θεωρία των κατηγοριών (**Category Theory**) και είναι ιδιότητα που παρουσιάζουν οι τύποι των **Functors.** Η λογική της αντιστοιχίας μεταξύ συνόλων, εντός μίας δομής (Container) είναι πολύ δημοφιλής στον συναρτησιακό προγραμματισμό και χρησιμοποιείται σε όλες τις συναρτησιακές γλώσσες. Στην προκειμένη περίπτωση η δομή της λίστας περιέχει στοιχεία που μπορούν να αντιστοιχηθούν σε στοιχεία κάποιου άλλου συνόλου. Για την εξαγωγή των ονομάτων του πίνακα **arr** η **map** θα πάρει ως όρισμα μία συνάρτηση

$$f: Person \rightarrow String$$

και θα επιστρέψει ένα νέο Container (λίστα) με όλα τα ονόματα των ανθρώπων όπως φαίνεται παρακάτω

```
// getNames :: Person -> String
const getNames = person => person.name;

const peopleNames = arr.map(getNames);
//-> [ 'Alonzo Church', 'Haskell Curry', 'Alan Turing', 'Stephen
Cleene' ]
```

Στην αμιγώς συναρτησιακή της μορφή η μέθοδος **map** μπορεί να μετατραπεί σε συνάρτηση δύο ορισμάτων

```
// map :: (a -> b, [a]) -> [b]
const map = (f, arr) => Array.prototype.map.call(arr, f);

const peopleNames = map(getNames, arr);
//-> [ 'Alonzo Church', 'Haskell Curry', 'Alan Turing', 'Stephen
Cleene' ]
```

Το γεγονός ότι η μέθοδος **map** επιστρέφει ένα νέο δοχείο (λίστα) κάθε φορά που καλείται μας επιτρέπει να χρησιμοποιήσουμε ορθά με αναφορική ακεραιότητα την τεχνική της αλυσίδωσης μεθόδων. Αν λόχου χάρη θα θέλαμε να εμφανίζονται μόνο τα μικρά ονόματα των ανθρώπων τότε θα καλούσαμε την **map** διπλά όπως φαίνεται

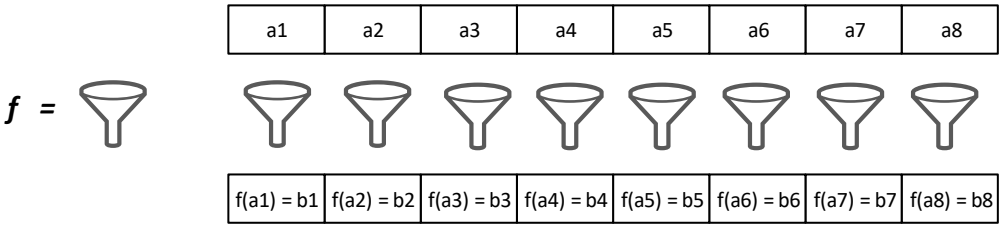
```
// getNames :: Person -> String
const getNames = person => person.name;

// getFirstNames :: String -> String
const getFirstNames = name => name.split(" ")[0];

const peopleFirstNames = arr.map(getNames).map(getFirstNames);
//-> [ 'Alonzo', 'Haskell', 'Alan', 'Stephen' ]
```

Αξίζει να σημειωθεί ότι σημαντικό στοιχείο της συνάρτησης **map** σε λίστες (και γενικά σε Functors) είναι η συνθετική ιδιότητα. Ανάλυση θα γινεί στο κεφάλαιο που θα δούμε τα Functors, όμως αρκεί να αναφέρουμε ότι στο τελευταίο παράδειγμα τα μικρά ονόματα θα μπορούσαν να εξαχθούν και με μία νέα συνάρτηση που θα προέκυπτε από τη σύνθεση των δύο λειτουργιών **getNames** και **getFirstNames.** Αν δανειστούμε τη συνάρτηση composeTwo του προηγούμενου υποκεφαλαίου τότε θα είχαμε

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (f, g) => x => f(g(x));

// get :: Person -> String
const get = composeTwo(getFirstNames, getNames);

const peopleFirstNames = arr.map(get);
//-> [ 'Alonzo', 'Haskell', 'Alan', 'Stephen' ]
```

Τέλος παραθέτουμε ένα σχήμα που περιγράφει συνοπτικά τη λειτουργία της **map**

map : (a -> b, [a]) -> [b],
όπου
f : a -> b



Σχήμα 4-3 Λειτουργία συνάρτησης map

4.4.2 Συνάρτηση filter

Έστω ότι έχουμε έναν πίνακα με στοιχεία ενός συνόλου A και τον συμβολίζουμε με $[A]$. Τότε η συνάρτηση **filter** δέχεται ως παράμετρο μία συνάρτηση f που αντιστοιχεί στοιχεία του συνόλου A σε στοιχεία του συνόλου $Boolean$ $\{True, False\}$, δηλαδή $f: A \rightarrow Boolean$, και επιστρέφει έναν νέο πίνακα με στοιχεία ενός συνόλου $B \subseteq A$, $[B]$ που προϋπόθεση έχουν να ικανοποιούν τη συνθήκη $f(A_i) = True,\ Ai\ \in A$. Η υπογραφή της μεθόδου έχει την εξής μορφή

$$\boldsymbol{Array.prototype.filter}: \boldsymbol{Array} \rightsquigarrow (\boldsymbol{a} \rightarrow \boldsymbol{Boolean}) \rightarrow \boldsymbol{Array}$$

Ενώ η υπογραφή της συνάρτησης

$$\boldsymbol{filter}: \big((\boldsymbol{a} \rightarrow \boldsymbol{Boolean}), [\boldsymbol{a}]\big) \rightarrow [\boldsymbol{a}] \text{ ή } \boldsymbol{map}: (\boldsymbol{a} \rightarrow \boldsymbol{Boolean}) \rightarrow [\boldsymbol{a}] \rightarrow [\boldsymbol{a}]\ (\boldsymbol{curried})$$

Ας δούμε όμως την πρακτική απήχηση της συνάρτησης υπό τη μορφή παραδείγματος. ‘Εστω ότι έχουμε τον πίνακα του προηγούμενου παραδείγματος arr και θέλουμε τους ανθρώπους που εμφανίζουν ηλικία μεγαλύτερη από 26 χρόνια. Τότε αγνοώντας μία προστακτική λύση του προβλήματος, που συνοπτικά μέσω ενός βρόχου επανάληψης και μία δομής **if** θα προσπέλαζε όλα τα στοιχεία του πίνακα arr και θα δημιουργούσε έναν νέο πίνακα με τα ζητούμενα στοιχεία (ή πιθανόν να τροποποιούσε τον υπάρχοντα πίνακα arr), θα μπορούσαμε να αντιμετωπίσουμε το πρόβλημα συναρτησιακά. Αντί δηλαδή να εμπλακούμε στο βάθος της λεπτομέρειας προτιμάμε να πούμε σε έναν έτοιμο μηχανισμό (τον οποίο εμείς μπορούμε να συντάξουμε) το τι (**what**) θέλουμε να κάνει και όχι πώς θα το κάνει (**how**). Στην προκειμένη περίπτωση ένας τέτοιος μηχανισμός είναι η συνάρτηση υψηλής τάξης **filter**. Η συνάρτηση αυτή θα δεχτεί ως είσοδο έναν πίνακα με στοιχεία του συνόλου **Person** και μία συνάρτηση τύπου **Predicate** (δηλαδή συνάρτηση που αντιστοιχεί στοιχεία ενός συνόλου στο σετ Boolean) και θα επιστρέψει έναν νέο πίνακα που θα περιέχει όλα τα στοιχεία του συνόλου **Person** του πίνακα εισόδου για τα οποία η συνάρτηση **Predicate** επιστρέφει θετική τιμή (true). Ας δούμε το παράδειγμα για να το κατανοήσουμε καλύτερα.

```
// checkAge :: Person -> Boolean
const checkAge = person => person.age > 26;

const peopleOverAge26 = arr.filter(checkAge);
//-> [{name: 'Alonzo Church', age: 44}, {name: 'Stephen Cleene', age:
33}]
```

Όπως και στην περίπτωση της **map** η **filter** μπορεί να μετατραπεί στην αμιγώς συναρτησιακή της μορφή

```
// filter :: (a -> Boolean, [a]) -> [a]
const filter = (f, arr) => Array.prototype.filter.call(arr, f);

const peopleOverAge26 = filter(checkAge, arr);
//-> [{name: 'Alonzo Church', age: 44}, {name: 'Stephen Cleene', age:
33}]
```

Επίσης το γεγονός ότι η μέθοδος **filter** επιστρέφει ένα νέο δοχείο (λίστα) κάθε φορά που καλείται μας επιτρέπει να χρησιμοποιήσουμε ορθά και με αναφορική ακεραιότητα την τεχνική της αλυσίδωσης μεθόδων. Αν για παράδειγμα θέλαμε όλους του ανθρώπους που έχουν ηλικία μικρότερη των 40 και αριθμό γραμμάτων μικρού ονόματος μεγαλύτερο από 5 τότε μία προσέγγιση θα είχε την εξής μορφή

```
// checkAge :: Person -> Boolean
const checkAge = person => person.age < 40;

// checkLetter :: Person -> Boolean
const checkLetter = person => person.name.split(" ")[0].length > 5;

const ageLowerThan40nameOverThan5 =
arr.filter(checkAge).filter(checkLetter);
//-> [{name: 'Haskell Curry', age: 21}, {name: 'Stephen Cleene', age:
33}]
```

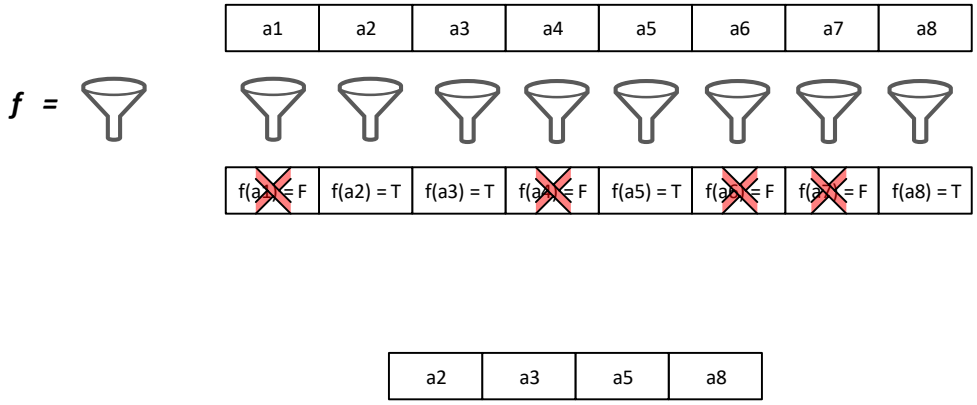
Αξίζει να σημειωθεί πως η συνάρτηση **filter** ακολουθεί, όπως ακριβώς και η **map**, την ιδιότητα της σύνθεσης. Σε αντίθεση όμως με την **map** που αντιστοιχεί στοιχεία ενός συνόλου X σε ένα σύνολο Y και συνεπώς ως σύνθεση ορίζεται η πράξη $f \circ g$ μεταξύ δύο μορφισμών στην κατηγορία των συναρτήσεων, η συνάρτηση filter αντιστοιχεί στοιχεία ενός συνόλου στο σετ Boolean και συνεπώς ως σύνθεση στη κατηγορία της λογικής (Predicate logic) ορίζεται ή πράξη $f \wedge g$ ή $f \&\& g$. Με άλλα λόγια το παραπάνω παράδειγμα μπορούσε να συνταχθεί όπως φαίνεται παρακάτω.

```
// composeTwo :: (a -> Boolean, a -> Boolean) -> a -> a
const composeTwo = (f, g) => x => f(x) && g(x);

// checkAgeAndLetter :: a -> Boolean
const checkAgeAndLetter = composeTwo(checkAge, checkLetter);

const ageLowerThan40nameOverThan5 = arr.filter(checkAgeAndLetter);
//-> [{name: 'Haskell Curry', age: 21}, {name: 'Stephen Cleene', age:
33}]
```

Τέλος παραθέτουμε ένα σχήμα που περιγράφει συνοπτικά τη λειτουργία της **filter**

filter : (a -> Boolean, [a]) -> [a],
όπου

f : a -> Boolean



Σχήμα 4-4 Λειτουργία συνάρτησης filter

4.4.3 Συνάρτηση reduce

Έστω ότι έχουμε έναν πίνακα με στοιχεία ενός συνόλου A και τον συμβολίζουμε με $[A]$. Τότε η συνάρτηση **reduce** δέχεται ως παράμετρο μία συνάρτηση f που αντιστοιχεί στοιχεία του συνόλου B και A σε στοιχεία του συνόλου B, δηλαδή $f: (B, A) \to B$, μία αρχική τιμή από το σύνολο B, και επιστρέφει μία τελική τιμή που ανήκει στο σύνολο B, και προκύπτει από διαδοχική εφαρμογή της f με παραμέτρους τα στοιχεία του πίνακα και επαγωγικά το προηγούμενο αποτέλεσμα της κλήσης της. Η υπογραφή της μεθόδου έχει την εξής μορφή

$$\boldsymbol{Array.prototype.reduce}: \boldsymbol{Array} \rightsquigarrow ((\boldsymbol{b}, \boldsymbol{a}) \to \boldsymbol{b}, \boldsymbol{b}) \to \boldsymbol{b}$$

Ενώ η υπογραφή της συνάρτησης

$$\boldsymbol{reduce}: \big((\boldsymbol{b}, \boldsymbol{a}) \to \boldsymbol{b}, \boldsymbol{b}, [\boldsymbol{a}]\big) \to \boldsymbol{b} \text{ ή } \boldsymbol{reduce}: \big((\boldsymbol{b}, \boldsymbol{a} \to \boldsymbol{b})\big) \to \boldsymbol{b} \to [\boldsymbol{a}] \to \boldsymbol{b} \ (\boldsymbol{curried})$$

Η **reduce**, γνωστή και ως **fold** (στην JavaScript με μικρές διαφορές) σε ορισμένες γλώσσες προγραμματισμού, χρησιμοποιείται στην εξαγωγή αποτελέσματος από μία δομή (λίστα, foldable δομή) ελαττώνοντας τις τιμές της δομής μέσω μίας συνάρτησης γνωστή και ως **reducer function**. Αποτελεί μία επαναληπτική δομή και το αποτέλεσμα που επιστρέφει μπορεί να ανήκει σε ένα σύνολο διαφορετικό από αυτό των στοιχείων του πίνακα στο οποίο εφαρμόζεται. Ας δούμε όπως ένα παράδειγμα για να κατανοήσουμε τη λειτουργία της καθώς η χρήση της απαιτεί εξοικείωση.

Έστω ότι έχουμε έναν πίνακα με αριθμούς και επιθυμούμε να εξάγουμε το άθροισμα των στοιχείων του. Τότε η μέθοδος **reduce** είναι ή πλέον κατάλληλη καθώς συμμαζεύει τα στοιχεία ενός πίνακα σε μία τελική τιμή βάσει μίας αθροιστικής συνάρτησης εισόδου.

```
const arr = [26, -33, 12, -15, 10, 37];

// add :: (Number, Number) -> Number
const add = (x, y) => x + y;

const result = arr.reduce(add, 0);
//-> 37
```

Αν αναπαραστήσουμε τη λειτουργία της **reduce** σε έναν πίνακα τότε θα δούμε πως προσθέτει ένα επίπεδο αφαίρεσης σε μία αναδρομική λειτουργία στην οποία το αποτέλεσμα της κλήσης της συνάρτησης **add** εισάγεται ως παράμετρος στην επόμενη κλήσης της. Παρακάτω βλέπουμε εκχωρημένα σε πίνακα τις τιμές για κάθε βήμα.

[26, -33, 12, -15, 10, 37].reduce(add, 0)

initVal	x	y	add(x,y)	A_i	result
BHMA-1	0	26	26	26	26
BHMA-2	26	-33	-7	-33	-7
BHMA-3	-7	12	5	12	5
BHMA-4	5	-15	-10	-15	10
BHMA-5	-10	10	0	10	0
BHMA-6	0	37	37	37	37

Βλέπουμε ότι η τιμή x του βήματος i είναι το αποτέλεσμα το αποτέλεσμα της συνάρτησης **add** του βήματος i-1. Στην αμιγώς συναρτησιακή της μορφή η **reduce** παίρνει τη μορφή που φαίνεται παρακάτω.

```
const arr = [26, -33, 12, -15, 10, 37];

// add :: (Number, Number) -> Number
const add = (x, y) => x + y;

// reduce :: (((b, a) -> b), b, [a]) -> b
const reduce = (f, initialValue, array) =>
Array.prototype.reduce.call(array, f, initialValue);

const result = reduce(add, 0, arr);
//-> 37
```

Η **reduce** όπως θα δούμε στη συνέχεια του κεφαλαίου χρησιμοποιείται για τη δημιουργία **συναρτήσεων υψηλής τάξης** που απαιτούνε επαγωγική προσέγγιση. Για παράδειγμα ο τελεστής της συνάθροισης συναρτήσεων ∘ μπορεί να υλοποιήσει μία συνάρτηση που συναθροίζει όλες τις συναρτήσεις που περιέχονται σε μία λίστα ακριβώς όπως ο τελεστής του αθροίσματος εξάγει το άθροισμα όλων των αριθμών μίας λίστας, όπως είδαμε στο προηγούμενο παράδειγμα. Επίσης αξίζει να σημειωθεί ότι η JavaScript διαθέτει και τη συνάρτηση **reduceRight** η οποία προσπελαύνει τα στοιχεία του πίνακα από δεξιά προς τα αριστερά. Η υπογραφή της μεθόδου και της συνάρτησης είναι ίδια με αυτή της **reduce.** Παρακάτω εξάγουμε το άθροισμα των στοιχείων του πίνακα **arr** με τη βοήθεια της **reduceRight.**

```
const arr = [26, -33, 12, -15, 10, 37];

// add :: (Number, Number) -> Number
const add = (x, y) => x + y;

const result = arr.reduceRight(add, 0);
//-> 37
```

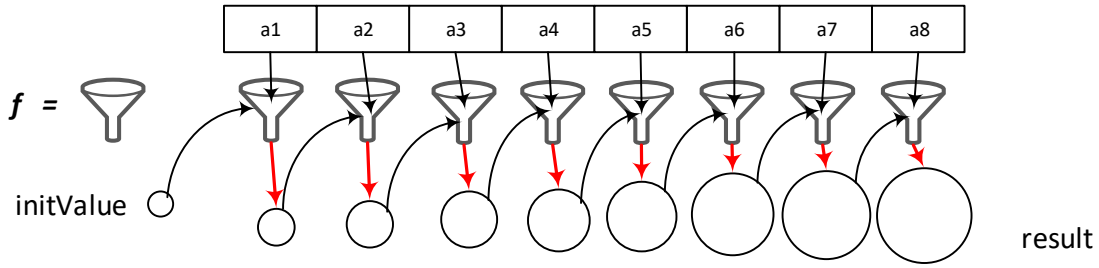
και ή συναρτησιακή υλοποίηση

```
// reduceRight :: (((b, a) -> b), b, [a]) -> b
const reduceRight = (f, initialValue, array) =>
Array.prototype.reduceRight.call(array, f, initialValue);

const result = reduceRight(add, 0, arr);
//-> 37
```

Παρακάτω παραθέτουμε ένα σχεδιάγραμμα που απεικονίζει τη ροή των συναρτήσεων **reduce** και **reduceRight**

reduce : (((b, a) -> b), b, [a]) -> b, όπου

f : (b, a) -> b



Σχήμα 4-5 Λειτουργία συνάρτησης reduce

ή

$$reduce(f, acc, [a1, a2, a3, a4, \dots, a8]) = f(\dots f(f(f(acc, a1), a2), a3) \dots, a8)$$

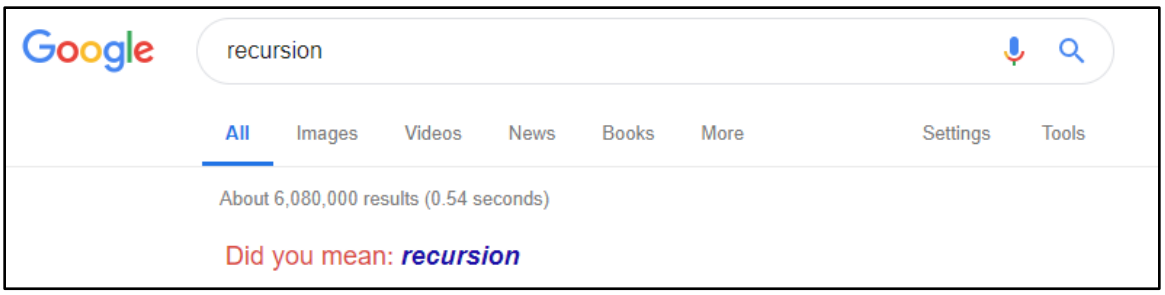
Ομοίως τα ίδια ισχύουν και για την **reduceRight** μόνο που η προσπέλαση των στοιχείων γίνεται από δεξία προς τα αριστερά. Εδώ αξίζει να σημειωθεί ότι στην JavaScript ενώ η **reduce** παρουσιάζει την ίδια λειτουργικότητα με τη γνωστή σε άλλες συναρτησιακές γλώσσες, **foldLeft**, η **reduceRight** της JavaScript δεν έχει την ίδια υλοποίηση με την **foldRight** που έχουν άλλες γλώσσες αλλά όπως είπαμε ακολουθεί ακριβώς την ίδια υλοποίηση με τη **reduce** σε αντίστροφο πίνακα. Οι ακριβείς υλοποιήσεις των συναρτήσεων **foldLeft** και **foldRight** θα δοθούν στο επόμενο κεφάλαιο στα πλαίσια της συναρτησιακής βιβλιοθήκης που θα συνοδεύει την παρούσα εργασία.

4.5 Αναδρομή (Recursion)

Η αναδρομική υλοποίηση της λύσης ενός προβλήματος αποτελεί μία πολύ σημαντική τεχνική του συναρτησιακού προγραμματισμού. Συναρτησιακές γλώσσες όπως η Haskell, δεν προσφέρουν ενναλακτική προσέγγιση σε δομές επανάληψης, όπως οι **for**, **while**, **do-while,** αλλά απαιτούν την τεχνική της αναδρομής στις περιπτώσεις που παρουσιάζονται επαναληπτικές διαδικασίες. Η JavaScript όντας μία υβριδική γλώσσα, που υποστηρίζει την αντικειμενοστρέφεια αλλά επιτρέπει και μάλιστα στις μέρες μας ενθαρρύνει τη συναρτησιακή επίλυση των πραγμάτων, προσφέρει στον προγραμματιστή και τους δύο τρόπους. Μία βασική αρχή της επιστήμης των υπολογιστών είναι ότι οτιδήποτε μπορεί να λυθεί χρησιμοποιώντας τις κλασσικές δομές επανάληψης, μπορεί να επιλυθεί και με χρήση της αναδρομής, και το ανάποδο. Επίσης ένα άλλο γεγονός είναι ότι η τεχνική της αναδρομής, αν και σύνθετη κυρίως λόγω της ελλιπούς διδασκαλίας σε νέους προγραμματιστές, προσφέρει αρκετά ευκολότερη και εύληπτη υλοποίηση πολλών γνωστών αλγορίθμων. Η αναδρομή, θα λέγαμε ότι αποτελεί μία φυσική διαδικασία επίλυσης προβλημάτων όπως:

- Μαθηματικοί ορισμοί όπως οι ακολουθίες, αθροίσματα, σειρές και γενικά επαγωγικές μέθοδοι
- Δομές δεδομένων που ορίζονται αναδρομικά, όπως οι λίστες, δέντρα, γράφοι κτλ.
- Συντακτική ανάλυση και εξέταση κώδικα που επιτελούν οι μεταγλωττιστές
- Γνωστά προβλήματα όπως οι πύργοι του Ανόι (Tower of Hanoi), N-Βασίλισσές (N-Queens)

Μία αναδρομική συνάρτηση λοιπόν είναι η συνάρτηση που στην προσπάθεια της να επιλύσει ένα πρόβλημα, καλεί τον εαυτό της, ελαττώνοντας έτσι τη λύση του γενικού προβλήματος σε άλλα μικρότερα ίδιας φύσεως. Κάποια στιγμή τα μικρά προβλήματα απλοποιούνται, επιλύονται, και με τη σειρά τους βοηθούν στην επίλυση των μεγαλύτερων προβλημάτων ώσπου τελικώς δίνεται λύση στο αρχικό πρόβλημα. Η αρχή αυτή μπορεί να μην μας είναι κατανοητή αλλά η ενασχόληση μας με χαρακτηριστικά προβλήματα που ενθαρρύνουν την αναδρομική επίλυση θα καθαρίσει το θολό τοπίο.



Σχήμα 4-6 Ερώτημα αναζήτησης της φράσης "recursion" στο Google s. Engine. Επιστρέφεται χιουμοριστικά ο ίδιος όρος

Ένα κλειδί στην αναδρομική προσέγγιση αποτελεί η υπόθεση ότι η συνάρτηση μας ήδη επιλύει το πρόβλημα που χρειάζεται να αντιμετωπίσει και συνεπώς η κλήση της ευσταθεί. Ακούγεται λίγο περίεργο αλλά η παραπάνω πρόταση αντικατοπτρίζει τον ορισμό της αναδρομικής συνάρτησης και ως εκ τούτου ισχύει. Αν στον αντίποδα προσπαθήσουμε νοητικά να ακολουθήσουμε τη ροή της αναδρομικής ακολουθίας τότε με μεγάλη πιθανότητα θα αποτύχουμε καθώς θα μπερδευτούμε. Φυσικά η ακολουθία της ροής και το πως ακριβώς δουλεύει η αναδρομική υλοποίηση είναι κάτι που πρέπει να γίνεται γιατί ο προγραμματιστής πρέπει να είναι σε θέση να αιτιολογεί τον κώδικα του, όμως αυτό πρέπει να γίνεται εκ των υστέρων, και όχι κατά την αναζήτηση της αναδρομικής συνάρτησης που θα επιλύει το πρόβλημα του. Πρακτικά η μέθοδος που ακολουθείται στηρίζεται σε τέσσερα βήματα.

1) Υποθέτουμε ότι ήδη έχουμε μία συνάρτηση η κλήση της οποίας επιλύει το πρόβλημα

2) Στη συνέχεια αναζητούμε συσχέτιση ολόκληρου του προβλήματος με ίδιας μορφής μικρότερα

3) Λύση των μικρότερων προβλημάτων χρησιμοποιώντας τη συνάρτηση που ορίσαμε νοητικά στο πρώτο βήμα

4) Ορισμός απλών βασικών καταστάσεων (**base cases**) που επιστρέφουν βασικές τιμές και δεν κάνουν χρήση της αναδρομής. Οι βασικές αυτές καταστάσεις χρησιμεύουν στην ορθή ξεδίπλωση την αναδρομικής λύσης για αποφυγή υπερχειλίσεων της στοίβας (**stack overflow**).

Υλοποιώντας τους τέσσερις παραπάνω βηματισμούς δημιουργούμε τη βασική δομής της αναδρομικής λύσεως με αποτέλεσμα η προσέγγιση λύσης να είναι ευκολότερη. Επίσης υπάρχουν τρεις διαφορετικές προσεγγίσεις με τις οποίες μπορεί να ερμηνευτεί η μέθοδος των τεσσάρων βημάτων.

1) Μείωνε και βασίλευε (**Decrease and Conquer**). Αποτελεί την απλούστερη ερμηνεία, με βάση την οποία η επίλυση ενός προβλήματος εξαρτάται πλήρως από την επίλυση του ίδιου προβλήματος αλλά απλούστερης μορφής.

2) Διαίρει και βασίλευε (**Divice and Conquer**). Αποτελεί μία γενικότερη εκδοχή της παραπάνω μεθόδου, με βάση την οποία διαιρούμε το πρόβλημα σε δύο μικρότερα των οποίων η αναδρομική επίλυση επιτρέπει την έκβαση λύσης για το αρχικό πρόβλημα.

3) Δυναμικός προγραμματισμός (**Dynamic Programming**). Η ερμηνεία αυτή αναφέρει ότι η λύση ενός προβλήματος δίδεται από την επίλυση απλούστερων προβλημάτων ίδιας φύσεως, όπως ακριβώς με την τεχνική του διαίρει και βασίλευε, με τη διαφορά ότι οι λύσεις των απλούστερων προβλημάτων μνημονεύονται και ανακαλούνται όταν ξαναζητηθούν, γλιτώνοντας έτσι το κόστος που προέρχεται από την επίλυση ήδη λυμένων προβλημάτων. Ο δυναμικός προγραμματισμός υλοποιείται είτε δια μέσου της μνημόνευσης (memorization, top-down approach) είτε δια μέσου της πινακοποίησης (tabulation, bottom-up approach). Δεν θα γίνει ανάλυση των πτυχών του δυναμικού προγραμματισμού όμως να αξίζει να σημειωθεί ότι η μνημόνευση αποτελεί κυρίως τεχνική του συναρτησιακού προγραμματισμού και θα γίνει εκτενής αναφορά αργότερα.

Αρκετά με τη θεωρία όμως, ας δούμε ορισμένα παραδείγματα που λύνονται αναδρομικά. Έστω ότι έχουμε έναν πίνακα με n στοιχεία ενός συνόλου Α, και αναζητούμε σε αυτόν την ύπαρξη ενός στοιχειού x του ίδιου συνόλου. Το πρόβλημα της γραμμικής αναζήτησης σε μία λίστα είναι σύνηθες και οι περισσότεροι προγραμματιστές προχωράνε σε μία προστακτική υλοποίηση της λύσης χρησιμοποιώντας έναν βρόχο επανάληψης όπως φαίνεται παρακάτω

```
// search :: ([a], a) -> Boolean
const search = (arr, x) => {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === x) return true;
    }
    return false;
};


search([2, 5, 1, 10, 14, 7, 22, 39], 1);
//-> true
```

Ο συναρτησιακός προγραμματισμός μάθαμε ότι ενθαρρύνει τον προγραμματιστή να ασχολείται με το τι ακριβώς θέλει και να μην στρέφει την προσοχή του στις λεπτομέρειες για το πως ακριβώς θα το κάνει. Στη συγκεκριμένη περίπτωση βλέποντας τις συναρτήσεις υψηλής τάξης που εφαρμόζονται σε πίνακες (map, filter, reduce, κα) που είδαμε στο προηγούμενο υποκεφάλαιο κάποιος θα μπορούσε να δώσει λύση στο πρόβλημα πολύ πιο εύκολα με τη χρήση των βοηθητικών συναρτήσεων όπως απεικονίζεται παρακάτω

```
// search :: ([a], a) -> Boolean
const search = (arr, x) => arr.filter(e => e === x).length > 0;

search([2, 5, 1, 10, 14, 7, 22, 39], 0);
//-> false
```

Βλέπουμε ότι η λύση αυτή είναι «καθαρή» και κατανοητή. Τι σχέση έχει όμως με την τεχνική της αναδρομής; Όπως θα δούμε όλες οι συναρτησιακές γλώσσες επιλύουν τα προβλήματα επανάληψης αναδρομικά συσχετίζοντας τις λύσεις των προβλημάτων με τις λύσεις των απλούστερων κοκ. Στην προκειμένη περίπτωση η συνάρτηση **filter** μπορεί κάλλιστα και μάλιστα επιβάλλεται από τις συναρτησιακές γλώσσες να επιλυθεί αναδρομικά καθώς η λύση ακούγεται φυσική αν αναγάγουμε το πρόβλημα σε ένα ίδιας φύσεως αλλά απλούστερο, όπως προτείνει η αναδρομική προσέγγιση. Ας κάνουμε όμως μία παύση με την **filter** και ας προσπαθήσουμε να αναγάγουμε το αρχικό πρόβλημα της αναζήτησης σε κάτι απλούστερο δίνοντας έτσι αναδρομική λύση. Σε μία αόριστη λίστα n μεγέθους θα λέγαμε ότι το αποτέλεσμα της αναζήτησης ενός στοιχείου στη λίστα ισοδυναμεί με το αποτέλεσμα του ελέγχου του πρώτου στοιχείου της λίστας με το στοιχείο της αναζήτησης, σε περίπτωση που το στοιχείο βρεθεί επιστρέφεται η τιμή **true**,

διαφορετικά ανάγουμε το πρόβλημα στο ίδιο αλλά πλέον σε λίστα μεγέθους n – 1. Αφού εντοπίσαμε τη ροή της αναδρομής στο σημείο αυτό πρέπει να σκεφτούμε ότι πρέπει να ορίσουμε μία συνθήκη περάτωσης (base case) που θα τερματίζει την αναδρομή γιατί διαφορετικά θα είναι ατέρμων και εσφαλμένη. Στο συγκεκριμένο πρόβλημα η συνθήκη αυτή λογικώς καθορίζεται από το μέγεθος της λίστας εισόδου. Όταν η αναδρομική υλοποίηση δεχτεί μία λίστα με μηδενικό μέγεθος τότε θα λέγαμε ότι επιστρέφει την τιμή false. Η πρόταση αυτή γίνεται καλύτερα κατανοητή αν θεωρήσουμε ως είσοδο τη λίστα με μηδενικό αριθμό στοιχείων σε μία αρχική κλήση της αναδρομικής συνάρτησης και όχι όταν αυτή καλείται από μόνη της παράγοντας το αποτέλεσμα της βασικής κατάστασης. Ας δούμε όμως μία υλοποίηση που θα μας διευκολύνει να καταλάβουμε καλύτερα

```
// search :: ([a], a) -> Boolean
const search = (arr, x) => {
    if (arr.length === 0) return false;
    else if (arr[0] === x) return true;
    else return search(arr.slice(1), x);
};

search([2, 5, 1, 10, 14, 7, 22, 39], 14);
//-> true
```

Στη λύση που απεικονίζεται βλέπουμε δύο συνθήκες περάτωσης που τερματίζουν την αναδρομή και επιστρέφουν απτό αποτέλεσμα και μία διακλάδωση που θα λέγαμε υλοποιεί την επανάληψη ή τη ροή της αναδρομής. Όταν λοιπόν η λίστα που εισάγεται σαν είσοδος δεν είναι μηδενική και το στοιχείο αναζήτησης δεν ισοδυναμεί με το πρώτο στοιχείο της λίστας εισόδου, τότε η αναδρομική συνάρτηση καλεί τον εαυτό της με συρρικνωμένη λίστα εισόδου ανάγοντας θεωρητικά το αρχικό πρόβλημα σε μικρότερο. Υπό την έννοια αυτή το πρόβλημα της αναζήτησης ενός αριθμού $\boldsymbol{x}$ σε μία λίστα $\boldsymbol{arr_n}$, όπου **n** το πλήθος των στοιχείων τις λίστας, ισοδυναμεί με την επίλυση των προβλημάτων του ελέγχου του πλήθος των στοιχείων της λίστας εισόδου, του ελέγχου ισότητας του πρώτου στοιχείου της λίστας με το στοιχείο αναζήτησης και τέλος (εδώ υπεισέρχεται η αναδρομή) της αναζήτησης του ίδιου αριθμού $\boldsymbol{x}$ στην ίδια λίστα $\boldsymbol{arr_{n-1}}$, με απαλειμμένο το στοιχείο σύγκρισης του προηγούμενο υποπροβλήματος (στη λύση μας είναι το πρώτο στοιχείο της λίστας). Η αναδρομική λύση του προβλήματος της αναζήτησης ανήκει στη μεθοδολογία του «Μείωνε και βασίλευε» καθώς το πρόβλημα ανάγεται στον εαυτό του σε

μία απλούστερη μορφή. Επίσης αξίζει να σημειωθεί ότι σε αναδρομικές υλοποιήσεις συναρτήσεων πρέπει να δίνεται ιδιαίτερη έμφαση στο τύπο της επιστροφής και στους πιθανούς τελεστές που μπορούν να εφαρμοστούν. Στην προκειμένη περίπτωση η συνάρτηση search επιστρέφει λογική τιμή που ανήκει στο σύνολο Boolean. Αν μελετήσουμε καλύτερα το πρόβλημα θα λέγαμε ότι η αναγωγή της λύσης περιορίζεται σε πρόβλημα σύζευξης μεταξύ της συνθήκης ισότητας και της ροής της αναδρομής. Στο παρακάτω παράδειγμα φαίνεται η λύση της αναζήτησης με χρήση του συζευκτικού τελεστή ||

```
// search :: ([a], a) -> Boolean
const search = (arr, x) => arr.length === 0 ? false : arr[0] === x ||
search(arr.slice(1), x);

search([2, 5, 1, 10, 14, 7, 22, 39], 13);
//-> false
```

Ο τελεστής σύζευξης || στην JavaScript έχει την ιδιότητα της βραχυκύκλωσης της αποτίμησης (**short- circuit evaluation**), ένα αρκετά ισχυρό χαρακτηριστικό που μας δίνει τη δυνατότητα της αποτίμησης μίας έκφρασης ανάλογα με τη λογική τιμή εξόδου μίας άλλης έκφρασης. Στο παράδειγμα μας ο τελεστής || δέχεται ως είσοδο τις εκφράσεις,

```
arr[0] === x || search(arr.slice(1), x)
```

Αν η αποτίμηση της πρώτης έκφρασης *arr[0] === x* ισούται ή ανάγεται (**coersion**) στη λογική τιμή **false,** τότε ο τελεστής συνεχίζει και επιστρέφει την αποτίμηση της δεύτερης έκφρασης, διαφορετικά, αν δηλαδή *arr[0] === x* ισούται η ανάγεται στην τιμή **true**, επιστρέφει την αποτίμηση της πρώτης. Φυσικά παρόμοια αλλά αντιστροφή λογική ακολουθεί και ο τελεστής της διάζευξης **&&**. Αν η πρώτη έκφραση είναι αληθής, τότε αποτιμά και επιστρέφει την αμέσως επόμενη έκφραση, διαφορετικά επιστρέφει την πρώτη. Αν θέλουμε να εισάγουμε και τον διαζευκτικό τελεστή στη λύση μπορούμε με το παρακάτω τρόπο

```
// search :: ([a], a) -> Boolean
const search = (arr, x) => arr.length > 0 && (arr[0] === x ||
search(arr.slice(1), x));

search([2, 5, 1, 10, 14, 7, 22, 39], 22);
//-> true
```

Οι διάφοροι τελεστές που προκύπτουν από τη θεωρία κατηγοριών είναι τρομερά χρήσιμοι στον συναρτησιακό προγραμματισμό καθότι, πρώτον μπορούν να αναπαρασταθούν υπό τη μορφή συναρτήσεων και συνεπώς να επωφεληθούν από τις ιδιότητες που τις διέπουν και δεύτερον μας επιτρέπουν τη δημιουργία σύνθετων εκφράσεων που δίνουν λύσεις σε σύνθετα προβλήματα. Αφού είδαμε την αναδρομική υλοποίηση της αναζήτησης μπορούμε με σχετική ευκολία να συντάξουμε και την αναδρομική εκδοχή της συνάρτησης **filter.** Το αποτέλεσμα κάνει χρήση των δηλώσεων **if**, ενώ μπορεί να υλοποιηθεί και μέσω των τεχνικών **gathering** και **spreading**, που προσφέρει η JavaScript.

```
// filter :: ([a], a -> Boolean) -> [a]
const filter = (arr, f) => {
    if (arr.length === 0) return [];
    else if (f(arr[0])) return [arr[0]].concat(filter(arr.slice(1),
f));
    else return filter(arr.slice(1), f);
};

filter([1, 2, 3, 4, 5, 3], x => x < 3);
//-> [1, 2]
```

Βλέπουμε ότι η αναδρομική λύση της συνάρτησης **filter** έχει πολλά όμοια στοιχεία με αυτή της αναζήτησης. Ως βασική κατάσταση ορίζεται πάλι ο πίνακας με μηδενικό αριθμό στοιχείων όπου η συνάρτηση επιστρέφει το ουδέτερο στοιχείου του τελεστή συνάθροισης πινάκων, δηλαδή έναν κενό πίνακα. Η συνάθροισή πινάκων υλοποιείται στο δεύτερο παρακλάδι της δήλωσης **if** όπου ελέγχεται το πρώτο στοιχείο του πίνακα εισόδου με εφαρμογή της λογικής συνάρτησης **f** σε αυτό. Εάν το αποτέλεσμα είναι ή αναγάγεται σε θετικό τότε η συνάρτηση επιστρέφει τη συνάθροιση ενός πίνακα με μόνο στοιχείο το στοιχείο ελέγχου και ενός άλλου πίνακα που προκύπτει από την αναδρομική κλήση της συνάρτησης **filter** στο συρρικνωμένο κατά ένα στοιχείο νέο πίνακα εισόδου. Εάν η απεικόνιση του στοιχείου στο σύνολο **Boolean** μέσω της f είναι **false** τότε η συνάρτηση απλώς επιστρέφει την κλήση του εαυτού της με πίνακα εισόδου έναν νέο πίνακα που δεν περιέχει το στοιχείο που ελέγχθηκε κατά την προηγούμενη κλήση. Μία πιο διακριτική υλοποίηση θα ήταν η παρακάτω

```
// filter :: ([a], a -> Boolean) -> [a]
const filter = ([first, ...rest], f) => first === undefined ? [] :
f(first) ? [first, ...filter(rest, f)] : filter(rest, f);

filter([1, 2, 3, 4, 5, 3], x => x > 2);
//-> [3, 4, 5, 3]
```

Στο παραπάνω παράδειγμα χρησιμοποιούνται οι τεχνικές **gathering** και **spreading** που εισήχθησαν στη γλώσσα της JavaScript κατά το πρότυπο ECMAScript 2015 (ES6). Παρατηρούμε ότι μέσω αυτών είναι δυνατή η πράξη της συνάθροισης πινάκων (έμμεση μορφή) σε μία απόπειρα αντικατάστασης της άμεσης συνάρτησης ή μεθόδου **concat** που προσφέρει η γλώσσα για την εφαρμογή της ομώνυμης πράξης. Αφού μελετήσαμε το πρόβλημα της αναδρομικής αναζήτησης που όπως είπαμε ανάγεται στην κατηγορία 'Μείωνε και Βασίλευε' προσεγγίσεων, ας δούμε και την αναδρομική υλοποίηση ενός τρόπου ταξινόμησης που εκ φύσεως ανάγεται σε πρόβλημα που αντιμετωπίζεται με τη μέθοδο του '**Διαίρει και βασίλευε**'. Διαισθητικά η φύση της ταξινόμησης προβλέπει τη διαίρεση ενός προβλήματος σε δύο υποπροβλήματα ίδιας μορφής που με κατάλληλη σύνθεση των επιμέρους λύσεων οδηγείται στην επίλυση του αρχικού προβλήματος. Αυτό πρακτικά σημαίνει ότι η επίλυση των προβλημάτων συνήθως ακολουθεί λογαριθμική πολυπλοκότητα λόγω της διαίρεσης που υποκρύπτεται. Στον αντίποδα η μεθοδολογία των 'Μείωνε και Βασίλευε' προβλημάτων ακολουθούν συνήθως γραμμική πολυπλοκότητά όπως είδαμε και στο παράδειγμα της αναζήτησης παραπάνω.

Έστω λοιπόν ότι έχουμε μία λίστα με συγκρίσιμα στοιχεία. Συγκρίσιμα είναι τα στοιχεία ενός συνόλου που υπακούνε στο **Ord specification** κατά τον συναρτησιακό προγραμματισμό ή θα λέγαμε υλοποιούν το **Comparable interface** κατά την αντικειμενοστρεφή προσέγγιση. Ένα τέτοιο παράδειγμα φυσικά είναι των σύνολο των ακέραιων αριθμών $\mathbb{Z}$, οι οποίοι όπως έχουμε μάθει υπονοούν ένα μέγεθος και ως εκ τούτου μπορούν να διαταχθούν με βάση αυτήν τους την ερμηνεία. Ας υποθέσουμε λοιπόν ότι έχουμε έναν πίνακα με στοιχεία από το σύνολο $\mathbb{Z}$, τότε ένας αλγόριθμός που επιτελεί την ταξινόμηση τους και ενθαρρύνει την αναδρομή θα ήταν ο εξής:

Αν ο πίνακας που εισάγεται ως είσοδος στην αναδρομική συνάρτηση έχει ένα ή κανένα στοιχείο τότε επιστρέφεται ο ίδιος πίνακας καθότι αυτός είναι ήδη ταξινομημένος, ειδάλλως διάλεξε ένα τυχαίο στοιχείο του πίνακα και ονόμασε το **pivot,** φτιάξε δύο

υποπίνακες όπου στον πρώτο μάζεψε όλα τα στοιχεία του αρχικού πίνακα που είναι μικρότερα από το **pivot**, και στον δεύτερο αυτά που είναι μεγαλύτερα ή ίσα από το **pivot** και επέστρεψε τελικώς το πίνακα που προκύπτει από την αναδρομική κλήση της συνάρτησης ταξινόμησης για τον πρώτο υποπίνακα, το στοιχείο ελέγχου **pivot**, και την αναδρομική κλήση για τον δεύτερο υποπίνακα. Ας δούμε την υλοποίηση του αλγορίθμου στη γλώσσα της JavaScript

```
// sort :: Ord a => [a] -> [a]
const sort = ([first, ...rest]) => first === undefined ? [] :
[...sort(rest.filter(e => e < first)), first, ...sort(rest.filter(e =>
e >= first))];

sort([5, 1, 3, 9, 15, 44, 18, 17, 9, 9, 1, 22, -3, -1, 109, -13]);
//-> [ -13, -3, -1, 1, 1, 3, 5, 9, 9, 9, 15, 17, 18, 22, 44, 109 ]
```

Ο αλγόριθμός αναζήτησης αυτός είναι γνωστός και ως **quicksort algorithm,** και από ότι φαίνεται η αναδρομική εκδοχή της υλοποίησης του μπορεί να εκφραστεί σε μία μόλις γραμμή κώδικα.

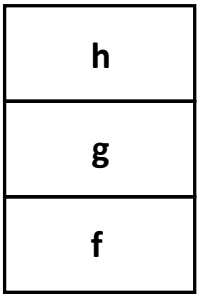
Ο συναρτησιακός προγραμματισμός αντικαθιστά τις επαναληπτικές δομές με τη χρήση των αναδρομικών συναρτήσεων σε μία απόπειρα προσέγγισης των προβλημάτων αφ' υψηλού και στο ευρύτερο πλαίσιο διαχωρισμού των λειτουργιών που προβλέπει. Υιοθετώντας την τεχνική της αναδρομής, ένας προγραμματιστής υιοθετεί παράλληλα έναν νέο τρόπο ερμηνείας και επίλυσης των προβλημάτων που καλείται να αντιμετωπίσει. Φυσικά όμως, όπως και ο συναρτησιακός προγραμματισμός, η τεχνική της αναδρομής απαιτεί μεγάλη εξοικείωση αλλά σε βάθος χρόνου η προγραμματιστική αντίληψη βελτιώνεται προσθέτοντας στο οπλοστάσιο της ένα ισχυρό εργαλείο.

4.5.1 Τεχνική της αναδρομής και στοίβα

Κλείνοντας το κεφάλαιο της αναδρομής πρέπει να γίνει αναφορά στον τρόπο με τον οποίο αντιμετωπίζει η στοίβα της εκάστοτε γλώσσας προγραμματισμού τις κλήσεις των συναρτήσεων. Ως γνωστόν η κλήση μίας συνάρτησης, τοποθετεί τη συνάρτηση αυτή και το εσωτερικό της περιβάλλον σε μία εσωτερική στοίβα (**Stack**) την οποία διαχειρίζεται η μηχανή της JavaScript. Αν η συνάρτηση αυτή καλεί εσωτερικά της κάποια άλλη, τότε το περιβάλλον αυτής τοποθετείται επάνω στη στοίβα σε μία λογική LIFO (Last-In-First-Out). Αν για παράδειγμα είχαμε τις παρακάτω συναρτήσεις με τις αντίστοιχες λειτουργίες

```
function f() {
    g();
}

function g() {
    h();
}

function h() {
    // doSomething
}
```

Τότε το στιγμιότυπο της στοίβας κατά την εκτέλεση της συνάρτησης **h** θα είχε τη μορφή

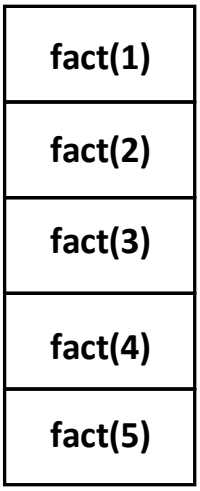


Stack

Φυσικά το μέγεθος της στοίβας αυτό είναι πεπερασμένο και εδώ έγκειται και το πρόβλημα μας στις περιπτώσεις που χρησιμοποιούμε την αναδρομή. Οι αναδρομικές κλήσεις όπως είδαμε καλούνε μία συνάρτηση τόσες φόρες όσες απαιτούνται για να συρρικνωθεί ένα πρόβλημα και να γίνει επιλύσιμο. Οι διαδοχικές αυτές κλήσεις έχουν ως αποτέλεσμα πολλές φορές να φορτώνουν τη στοίβα και σε αρκετές περιπτώσεις να ξεπερνούν το μέγεθος κλήσεων που αυτή επιτρέπει να φιλοξενήσει οδηγώντας έτσι το πρόγραμμα στο σφάλμα που είναι γνωστό και ως υπερχείλιση της στοίβας (**Stack Overflow**). Ας δούμε όμως ένα παράδειγμα για να καταλάβουμε καλύτερα το πρόβλημα. Έστω ότι θέλουμε να υπολογίσουμε το παραγοντικό ενός φυσικού αριθμού n. Το πρόβλημα αυτό είναι γνωστό για την απλή αναδρομική του υλοποίηση και χρησιμοποιείται κατά κόρον σε παραδείγματα αναδρομής. Η λύση φαίνεται παρακάτω

```
// fact :: Natural n => n -> n
const fact = n => n < 2 ? 1 : n * fact(n - 1);

fact(5);
//-> 120
```

Η **fact** καλεί τον εαυτό της πέντε φορές μέχρι να ξεδιπλωθεί (εξαγωγή κλήσεων από στοίβα) και να επιστραφεί το τελικό αποτέλεσμα. Το στιγμιότυπο της στοίβας κατά την τελευταία κλήση θα έχει τη μορφή που απεικονίζεται παρακάτω



Stack

Όπως αναφέραμε το μέγεθος της στοίβας είναι περιορισμένο και ως εκ τούτου η κλήση της συνάρτησης για έναν μεγάλο αριθμό n θα έχει ως αποτέλεσμα την υπερχείλιση της και την εξαγωγή σφάλματος. Στο παράδειγμα μας, με τη μηχανή JavaScript V8 που χρησιμοποιεί ο περιηγητής **Google Chrome** αλλά και η **Node**, για οποιοδήποτε αριθμό μικρότερο του n = 8357 το πρόγραμμα τρέχει κανονικά δίχως κάποιο πρόβλημα. Για αριθμούς όμως μεγαλύτερους ή ίσο με το συγκεκριμένο όριο το πρόγραμμα πετάει σφάλμα το *"RangeError: Maximum call stack size exceeded"* και αυτό διότι η στοίβα δεν μπορεί να φιλοξενήσει άλλες κλήσεις. Φυσικά κάποιος μπορεί να πει ότι το παραγοντικό ενός τέτοιου αριθμού είναι αχρείαστο αλλά το πρόβλημα του περιορισμένου χώρου της στοίβας μπορεί να επεκταθεί και σε άλλα παραδείγματα όπως θα ήταν μία γραμμική υλοποίηση ταξινόμησης ενός πίνακα με 100000 στοιχεία, ένα πρόβλημα αρκετά συχνό όταν έχουμε να κάνουμε με πολλά δεδομένα. Η JavaScript κατά το πρότυπο (ES6) δίνει λύσει σε αυτά

τα προβλήματα υλοποιώντας τη γνωστή βελτιστοποίηση **tail-call optimization** όπου ο προγραμματιστής ενεργοποιεί τη δυνατότητα χρήσης σταθερού χώρου (constant-space) γράφοντας αναδρομικές συναρτήσεις υπό μορφή **tail-called.** Για την ακρίβεια η συγκεκριμένη βελτιστοποίηση δεν έχει υλοποιηθεί ακόμα από τις σύγχρονες μηχανές JavaScript (V8, SpiderMonkey) αλλά κάτι τέτοιο θα γίνει μελλοντικά και ως εκ τούτου αξίζει να μελετηθεί (compatibility table ES6, n.d.).

Τι σημαίνει όμως tail-called function; Tail-called λέγονται οι συναρτήσεις που το τελευταίο πράγμα που επιτελούν λειτουργικά είναι η κλήση κάποιας άλλης συνάρτησης, επιτρέποντας έτσι τη μηχανή της JavaScript να προχωρήσει εκ νέου στην κλήση της επόμενης αυτής συνάρτησης αποφεύγοντας έτσι τη δημιουργία μίας επιπρόσθετης εγγραφής στη στοίβα και εξοικονομώντας πολύτιμο χώρο. Είναι η fact του παραπάνω παραδείγματος tail-called; Η απάντηση είναι όχι και αυτό προκύπτει από τη μελέτη του σώματος της συνάρτησης. Παρατηρούμε ότι το αποτέλεσμα της κλήσης μίας συνάρτησης ορίζεται από την πράξη του πολλαπλασιασμού ενός αριθμού που ορίζεται στο υπάρχον περιβάλλον και της αμέσως επόμενης κλήσης που ορίζει ένα νέο περιβάλλον μεταβλητών. Το γεγονός αυτό αφαιρεί τη δυνατότητα από τη μηχανή να αποδεσμευτεί από τα προηγούμενα περιβάλλοντα που δημιουργούνται από τις κλήσεις των συναρτήσεων με αποτέλεσμα αυτά να μένουν στη στοίβα. Η λύση που προτείνουν οι tail-called συναρτήσεις είναι η αποφυγή της άμεσης εξάρτησης των περιβαλλόντων που δημιουργούνται από τα σώματα των συναρτήσεων χρησιμοποιώντας ενναλακτικά τις συναρτησιακές παραμέτρους ως μέθοδο επικοινωνίας μεταξύ αυτών. Στην περίπτωση μας η fact μπορεί να εκφραστεί και με τον παρακάτω, tail-called optimized αυτή τη φορά, τρόπο.

```
// fact :: Natural n => (n, 1) -> n
const fact = (n, p) => n < 2 ? p : fact(n - 1, n * p);

fact(5, 1);
//-> 120
```

Παρατηρούμε ότι πλέον η επικοινωνία μεταξύ των τελεστών του γινομένου γίνεται διά μέσου της παραμέτρου p. Αυτό έχει ως αποτέλεσμα την αντιγραφή του στοιχείου αυτού στο νέο περιβάλλον κλήσης και ως εκ τούτου την αποδέσμευση της εξάρτησης από το προηγούμενο περιβάλλον. Αυτό σημαίνει ότι η στοίβα δεν προσθέτει εγγραφή και λειτουργεί σε σταθερό χώρο. Βέβαια στην υλοποίηση αυτή ο χρήσης της fact πρέπει να

θυμάται ότι πρέπει να την καλέσει κάνοντας χρήση της ουδέτερης τιμής του πολλαπλασιασμού, δηλαδή το 1. Ας βελτιώσουμε λίγο την υλοποίηση λοιπόν

```
// fact :: Natural n => n -> n
const fact = n => {
    const innerFact = (n, p) => n < 2 ? p : innerFact(n - 1, n * p);
    return innerFact(n, 1);
};

fact(8);
//-> 40320
```

Προσθέτοντας ένα επίπεδο αφαίρεσης παραπάνω η διαδικασία υλοποίησης μία συνάρτησης σε μορφή tail-called μπορεί να πραγματοποιηθεί από τον συναρτησιακό προγραμματισμό κάνοντας χρήση της συνέχειας (**Continuation Passing Style**). Πρόκειται για μία προσέγγιση που όπως είπαμε επιτρέπει την επικοινωνία μεταξύ δύο περιβαλλόντων που προκύπτουν από κλήσεις διαδοχικών συναρτήσεων μέσω μίας τρίτης γενικής συνάρτησης. Το παράδειγμα του παραγοντικού με χρήση της τεχνικής CPS θα είχε την παρακάτω μορφή

```
// fact :: Natural n => (n, n -> n) -> n
const fact = (n, contf) => n < 2 ? contf(1) : fact(n - 1, x => contf(n
* x));

fact(8, x => x);
//-> 40320
```

Και αν θα θέλαμε να αποδεσμεύσουμε τον καταναλωτή της συνάρτησης από την ανάγκη να την καλέσει χρησιμοποιώντας ως δεύτερη παράμετρο τη συνάρτηση **identity** (ή **I** συνδυαστής) μπορούμε να γράψουμε

```
const fact = n => {
    const innerFact = (n, contf) => n < 2 ? contf(1) : innerFact(n -
1, x => contf(n * x));
    return innerFact(n, x => x);
};

fact(8);
//-> 40320
```

Αφού είδαμε λοιπόν τους τρόπους με τους οποίους μπορεί να υλοποιηθεί μία αναδρομική διαδικασία και μελετήσαμε τα οφέλη που προσφέρει η αναδρομική προσέγγιση των επαναληπτικών προβλημάτων στον συναρτησιακό προγραμματισμό, πάμε να δούμε μία άλλη ισχυρή ιδιότητα του τελευταίου που πηγάζει από τη χρήση αγνών συναρτήσεων και την έλλειψη παράπλευρων τροποποιήσεων.

4.6 Μνημόνευση (Memoization)

Όπως είδαμε στην εισαγωγή του 2ου κεφαλαίου μία από τις βασικές αρχές του συναρτησιακού προγραμματισμού είναι η χρήση αγνών συναρτήσεων. Σε μία προσπάθεια υιοθέτησης της μαθηματικής θεωρίας, οι αγνές συναρτήσεις στον προγραμματισμό αποτελούν ένα πανίσχυρο εργαλείο που ενεργοποιεί στον προγραμματιστή τη δυνατότητα να δημιουργεί αυτοτελής λειτουργικότητες που δέχονται μία είσοδο και παράγουν μία έξοδο. Η ιδιότητα τους αυτή, δηλαδή η μη ενασχόληση τους με το εξωτερικό περιβάλλον και η σταθερή συσχέτιση μεταξύ του συνόλου των εισόδων και εξόδων, γεννά άλλο ένα χρήσιμο χαρακτηριστικό τους που έχει να κάνει με την εξοικονόμηση πόρων (προγραμματιστική σκοπιά). Για να αντιληφθούμε καλύτερα το πρόβλημα ας δούμε ένα απλό παράδειγμα το όποιο σίγουρα έχουμε ζήσει όλοι ως μαθητές. Έστω ότι στην τάξη των μαθηματικών ο καθηγητής μας δίνει ένα πρόβλημα στο οποίο μεταξύ των άλλων μας δίνεται μία συνάρτηση της παρακάτω μορφής

$$f: \mathbb{R} \to \mathbb{R}, f(x) = x^3 + 2019$$

και σε κάποιο σημείο μας ζητείται η απεικόνιση της f για την τιμή $x = 33$, δηλαδή το $f(33)$. Όντας καλοί μαθητές, και αφού ο καθηγητής απαγορεύει τις αριθμομηχανές, κάνουμε στο χαρτί τους πολλαπλασιασμούς και τις προσθέσεις και καταλήγουμε μετά από χρόνο τριών λεπτών ότι $f(33) = 37956$. Καταγράψαμε το αποτέλεσμα στο χαρτί και συνεχίζουμε το πρόβλημα. Ας υποθέσουμε ότι μετά από αρκετά ερωτήματα εμφανίζεται μία νέα συνάρτηση για τη οποία

$$g: \mathbb{R} \to \mathbb{R}, g(x) = 2 * f(x)$$

Και μας ζητείται η απεικόνιση της g για την τιμή $x = 33$, δηλαδή το $g(33)$. Το βασικό ερώτημα είναι, πόσο χρόνο θα χρειαστούμε για να βρούμε το αποτέλεσμα; Και φυσικά ο

χρόνος αυτός ισοδυναμεί με τον χρόνο που χρειάζεται κάποιος για να κάνει την πράξη $g(33) = 2 * f(33) = 2 * 37956 = 75912$ δηλαδή περίπου 15 δευτερόλεπτα. Κάποιος θα πει ότι ο χρόνος αυτός είναι τόσο σύντομος διότι είχαμε ήδη υπολογισμένο το αποτέλεσμα της f για $x = 33$ και αυτό ακριβώς είναι το σημείο που πρέπει να προσέξουμε. Ο μαθητής έκανε τη σωστή υπόθεση στο μυαλό του ότι αφού υπολογίσαμε το $f(33)$ σε προηγούμενο ερώτημα τότε η τιμή αυτή είναι έγκυρη και ισχύει για όλη τη διάρκεια που νοητικά ορίζεται η f, η τάξη, και το πρόβλημα που καλούμαστε να αντιμετωπίσουμε. Και αυτό δεν είναι κάτι αυθαίρετο αλλά βασική αρχή των μαθηματικών συναρτήσεων. Στο συγκεκριμένο παράδειγμα αποφύγαμε τον επαναϋπολογισμό της τιμής $f(33)$ (εξοικονομήσαμε δηλαδή περίπου 3 λεπτά) καθώς στηριχτήκαμε στην ιδιότητα ότι αφού η απεικόνιση της f για $x = 33$ ισούται με 37956 τότε η πρόταση αυτή ισχύει για πάντα και ως εκ τούτου οπουδήποτε αντικρίζουμε το $f(33)$ μπορούμε να το αντικαταστήσουμε με την αντίστοιχη τιμή.

Η συνάρτηση f στον συναρτησιακό προγραμματισμό είναι αγνή (**pure function**) και η δυνατότητα προνοητικής αντικατάστασης των τιμών όπως η $f(33)$ ονομάζεται όπως είδαμε αναφορική διαφάνεια (**referential transparency**).

Τα αντίστοιχα οφέλη μπορούμε να αποκομίσουμε και ως προγραμματιστές κάνοντας χρήση φυσικά του συναρτησιακού μοντέλου γραφής και συνακόλουθα των αγνών συναρτήσεων. Αν έχουμε στη διάθεση μας μία αγνή συνάρτηση η οποία κάνει κάποιο σύνθετο υπολογισμό με βάση μία συγκεκριμένη είσοδο και παράγει μία συγκεκριμένη έξοδο, τότε γιατί να μην εκμεταλλευτούμε το γεγονός ότι η έξοδος θα είναι ίδια στην περίπτωση που θα ζητηθεί επανυπολογισμός της συνάρτησης με την ίδια είσοδο αποφεύγοντας έτσι επεξεργαστικό κόστος. Η τεχνική που μας το επιτρέπει ονομάζεται μνημόνευση (**memoization**). Ας δούμε όμως πως θα υλοποιούνταν προγραμματιστικά το παράδειγμα του μαθητή και της συνάρτηση f προτού δούμε κάτι πιο σύνθετο.

Έστω ότι έχουμε τη συνάρτηση f στο παρακάτω πρόγραμμα

```
// f :: Number -> Number
const f = x => x * x * x + 2019;
```

Και θέλουμε να υπολογίσουμε την τιμή της f για $x = 33$. Επίσης ας υποθέσουμε ότι ο συγκεκριμένος υπολογισμός εμφανίζεται αρκετές φορές στο πρόγραμμα μας όπως φαίνεται παρακάτω

```
// f :: Number -> Number
const f = x => x * x * x + 2019;

f(33);
//-> 37956
f(33);
//-> 37956
f(33);
//-> 37956
```

Το πρόβλημα μας είναι ότι εφόσον μετά το πρώτο υπολογισμό της f για $x = 33$ γνωρίζουμε ότι $f(33) = 37956$ υποβάλουμε το πρόγραμμα σε δύο περιττούς υπολογισμούς της f για την ίδια είσοδο. Για να γίνει ορατό ότι η f τρέχει τον υπολογισμό τρεις φορές και όχι μία όπως θα θέλαμε ας τροποποιήσουμε λίγο τον ορισμό της συνάρτησης.

```
const logger = console.log.bind(console);

// f :: Number -> Number
const f = x => {
    logger("I am doing the calculation");
    return x * x * x + 2019;
};

f(33);
//-> I am doing the calculation
//-> 37956
f(33);
//-> I am doing the calculation
//-> 37956
f(33);
//-> I am doing the calculation
//-> 37956
```

Βλέπουμε ότι όντος η τιμή $f(33)$ υπολογίζεται τρεις φορές παρότι ως αγνή, ο επαναϋπολογισμός μπορεί να αποφευχθεί λόγω της τεχνικής της μνημόνευσης. Πως θα μπορούσαμε όμως να το υλοποιήσουμε; Σίγουρα χρησιμοποιώντας έναν χώρο ως μνήμη. Ας δούμε την παρακάτω υλοποίηση

```
const logger = console.log.bind(console);

const cache = Object.create(null);
// f :: Number -> Number
const f = x => {
    if (!cache[x]) {
        logger("I am doing the calculation");
        cache[x] = x * x * x + 2019;
    }
    return cache[x];
};

f(33);
//-> I am doing the calculation
//-> 37956
f(33);
//-> 37956
f(33);
//-> 37956
f(34);
//-> I am doing the calculation
//-> 41323
f(34);
//-> 41323
```

Βλέπουμε ότι κατά την πρώτη κλήση της $f(33)$ η συνάρτηση ελέγχει αν υπάρχει αποθηκευμένο το αποτέλεσμα στη μνήμη **cache** (πρακτικά εδώ το object χρησιμοποιείται ως dictionary). Εφόσον δεν υπάρχει κατά την πρώτη κλήση προχωράει στον υπολογισμό, στην αποθήκευση στην **cache,** και έπειτα επιστρέφει το αποτέλεσμα. Κατά τη δεύτερη κλήση τώρα εφόσον η τιμή **cache["33"]** υφίσταται, η συνάρτηση επιστρέφει απευθείας την υπολογισμένη τιμή δίχως να επανεκτελεί τον υπολογισμό. Το ίδιο γίνεται για την τρίτη κλήση. Η τέταρτη κλήση της f αυτή τη φορά με είσοδο το $x = 34$ προχωράει κανονικά στην επίλυση της μαθηματικής παράστασης καθότι η τιμή **cache["34"]** δεν υφίσταται. Στην τελευταία κλήση της $f(34)$ η τιμή 41323 είναι αποθηκευμένη στη μνήμη και οπότε επιστρέφεται κατευθείαν. Έτσι λοιπόν το αλφαριθμητικό 'I am doing the calculation' εμφανίζεται συνολικά δύο φόρες όσες δηλαδή και οι διαφορετικού είσοδοι (33, 34). Φυσικά ο συναρτησιακός προγραμματισμός ενθαρρύνει τη δημιουργία συναρτήσεων υψηλής τάξης για την προσθήκη τέτοιου είδους επιπρόσθετων λειτουργιών σε μία

συνάρτηση αποφεύγοντας έτσι την τροποποίηση του εξωτερικού περιβάλλοντος, όπως γίνεται στη συγκεκριμένη περίπτωση με τη μεταβλητή **cache** που είναι ορατή από όλους. Θα μπορούσαμε να γράψουμε

```
const logger = console.log.bind(console);
// f :: Number -> Number
const f = x => x * x * x + 2019;

const tap = f => x => (logger("I am doing the calculation"), f(x));

// memo :: (a -> b) -> a -> b
const memo = h => {
    const cache = Object.create(null);
    return x => cache[x] || (cache[x] = h(x), cache[x]);
};

const fm = memo(tap(f));

fm(33);
//-> I am doing the calculation
//-> 37956
fm(33);
//-> 37956
fm(33);
//-> 37956
```

Όπου **memo** ορίζεται ως μία συνάρτηση υψηλής τάξης που δέχεται την αγνή συνάρτηση για την οποία θέλουμε χρησιμοποιηθεί η τεχνική της μνημόνευσης. Η συνάρτηση **tap** δεν έχει σχέση με την υλοποίηση και δημιουργήθηκε καθαρά για να γίνει ορατός ο πραγματικός αριθμός κλήσεων της συνάρτησης **f** ανά στοιχείο εισόδου. Στην προκειμένη περίπτωση βλέπουμε ότι ο υπολογισμός της τιμής $f(33)$ πραγματοποιήθηκε μόνο μία φορά όπως ορίζει η τεχνική της μνημόνευσης (αφού η φράση "I am doing the calculation" εμφανίστηκε μία φορά) και τις υπόλοιπες δύο η τιμή επιστράφηκε απευθείας από το αντικείμενο **cache** γλιτώνοντας έτσι υπολογιστικό κόστος.

Πάμε να δούμε όμως άλλο ένα πιο σύνθετο παράδειγμα στο οποίο η τεχνική της μνημόνευσης βοηθάει το πρόγραμμα να είναι λειτουργικό καθώς οι υπολογισμοί που γίνονται είναι ιδιαίτερα χρονοβόροι. Προτού όμως δούμε τη συνάρτηση που θα δοκιμάσουμε, και εφόσον μιλάμε για υπολογιστικά κόστη, θα ήταν συνετό να

χρησιμοποιήσουμε και μία λειτουργικότητα που θα μας εκτυπώνει τον χρόνο εφαρμογής μίας συνάρτησης προς δοκιμή. Και στην περίπτωση αυτή θα χρησιμοποιήσουμε πάλι μία συνάρτηση υψηλής τάξης που λειτουργεί ως περιτύλιγμα (**wrapper**) της αρχικής συνάρτησης προσφέροντας την επιπρόσθετη λειτουργικότητα. Παρακάτω βλέπουμε τη μορφή της

```
// withTime :: ((a -> b), () -> Time, () -> ()) -> a -> b
const withTime = (f, getTime, log) => (...args) => {
    const tStart = getTime();
    const result = f(...args);
    const tEnd = getTime();
    log(`Input: ${args} | Output: ${result} | Total time: ${tEnd -
tStart}ms`);
    return result;
};
```

Η **withTime** δέχεται σαν είσοδο μία συνάρτηση **f**, της οποίας οι κλήσεις θέλουμε να χρονομετρηθούν, μία συνάρτηση χρονικής αποτύπωσης, και μία συνάρτηση εκτύπωσης και επιστρέφει μία νέα συνάρτηση η οποία λειτουργεί ακριβώς όπως η αρχική **f** με τη διαφοροποίηση ότι σε κάθε κλήση της εκτυπώνεται η χρονική διάρκεια που χρειάστηκε η εκάστοτε μηχανή εκτέλεσης για να υπολογίσει το αποτέλεσμα. Επιπρόσθετα παρακάτω φαίνεται η σύνταξη μίας αναδρομικής συνάρτησης που υπολογίζει και επιστρέφει τον ν-οστό όρο της γνωστής ακολουθίας Fibonacci. Επιλέξαμε τη συνάρτηση αυτή καθότι οι υπολογισμοί της μπορούν να γίνουν ιδιαίτερα χρονοβόροι λόγω των πολλών περιττών επανυπολογισμών. Ας δούμε πως συντάσσεται

```
// fib :: Natural n => n -> n
const fib = n => n <= 1 ? n : fib(n - 1) + fib(n - 2);
```

Κάθε όρος της ακολουθίας **Fibonacci** προκύπτει ως γνωστόν από το άθροισμα των δύο προηγούμενων όρων. Ως βασικές καταστάσεις ορίζονται οι τιμές n = 0 και n = 1 όπου επιστρέφεται η τιμή n, αφού οι δύο πρώτοι όροι της ακολουθίας είναι το {0, 1}. Ας εμπλουτίσουμε τη συνάρτηση **fib** με τη λειτουργικότητα της **withTime** για να δούμε τους χρόνους που επιστρέφονται στην αναζήτηση συγκεκριμένων όρων.

```
// logger :: () -> ()
const logger = console.log.bind(console);

// withTime :: ((a -> b), () -> Time, () -> ()) -> a -> b
const withTime = (f, getTime, log) => (...args) => {
    const tStart = getTime();
    const result = f(...args);
    const tEnd = getTime();
    log(`Input: ${args} | Output: ${result} | Total time: ${tEnd -
tStart}ms`);
    return result;
};

// fib :: Natural n => n -> n
const fib = n => n <= 1 ? n : fib(n - 1) + fib(n - 2);
// fib :: Natural n => n -> n
const fibT = withTime(fib, () => performance.now(), logger);


fibT(35);
//-> Input: 35 | Output: 9227465 | Total time: 131.8725999891758ms
fibT(35);
//-> Input: 35 | Output: 9227465 | Total time: 132.75299900770187ms
fibT(40);
//-> Input: 40 | Output: 102334155 | Total time: 1443.4167000055313ms
fibT(40);
//-> Input: 40 | Output: 102334155 | Total time: 1410.7007009983063ms
```

Παρατηρούμε ότι για να βρούμε τον 35ο όρο της ακολουθίας, τουλάχιστον στον υπολογιστή στον οποίο τρέχει το παραπάνω πρόγραμμα, απαιτούνται περίπου $\mathbf{130}\boldsymbol{ms}$ ενώ για να βρούμε τον 40ο χρειάζονται περίπου $\mathbf{1.5}\boldsymbol{s}$. Καλούμε τις κλήσεις για κάθε είσοδο δύο φορές για να δείξουμε ότι η χρονική διάρκεια της κλήσης κάθε επόμενη φορά πέραν της πρώτης δεν μεταβάλλεται. Αν εμπλουτίσουμε τη λειτουργικότητα της συνάρτησης **fib** με τη συνάρτηση υψηλής τάξης **memo** θα δούμε ότι τα αποτελέσματα είναι διαφορετικά για κάθε επόμενη κλήση καθώς η **memo** όπως είδαμε αποφεύγει τον επανυπολογισμό αφού το αποτέλεσμα είναι αποθηκευμένο στην **cache**. Και μίας και έχουμε εξοικειωθεί με την πράξη της σύνθεσης συναρτήσεων από το προηγούμενο κεφάλαιο, πάμε να κατασκευάσουμε μία νέα συνάρτηση υψηλής τάξης που να περιέχει τις λειτουργικότητες

της **withTime** και **memo** μαζί. Για τη δημιουργία της θα χρησιμοποιήσουμε την **composeTw**. Το αποτέλεσμα φαίνεται παρακάτω

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (f, g) => x => f(g(x));

// logger :: () -> ()
const logger = console.log.bind(console);

// withTime :: ((a -> b), () -> Time, () -> ()) -> a -> b
const withTime = (f, getTime = () => performance.now(), log = logger)
=> (...args) => {
    const tStart = getTime();
    const result = f(...args);
    const tEnd = getTime();
    log(`Input: ${args} | Output: ${result} | Total time: ${tEnd -
tStart}ms`);
    return result;
};

// fib :: Natural n => n -> n
const fib = n => n <= 1 ? n : fib(n - 1) + fib(n - 2);

const withTimeAndMemo = composeTwo(withTime, memo);

const fibT = withTimeAndMemo(fib);


fibT(35);
//-> Input: 35 | Output: 9227465 | Total time: 131.8725999891758ms
fibT(35);
//-> Input: 35 | Output: 9227465 | Total time: 0.002400010824203491ms
fibT(40);
//-> Input: 40 | Output: 102334155 | Total time: 1443.4167000055313ms
fibT(40);
//-> Input: 40 | Output: 102334155 | Total time: 0.0023989975452423 1ms
```

Παρατηρούμε ότι τα αποτελέσματα είναι θεαματικά. Τη δεύτερη φορά που καλείται η **fibT** για να βρεθεί ο 35ος όρος ο χρόνος που χρειάζεται για να επιστρέψει το αποτέλεσμα η συνάρτηση είναι μειωμένος κατά 54946 φορές περίπου. Αντίστοιχα για τον 40ο όρο ο χρόνος προσπέλασης της συνάρτησης στη δεύτερη κλήση είναι $\mathbf{0.0024ms}$ σε αντίθεση με το προηγούμενο παράδειγμα όπου ο χρόνος ήταν περίπου $\mathbf{1.5}s$ όσα δηλαδή

χρειάζεται και η πρώτη κλήση. Επιπρόσθετα αξίζει να σημειωθεί ότι με την παραπάνω υλοποίηση ο υπολογισμός του $40^{ου}$ όρου δεν χρησιμοποιεί τους υπολογισμούς προηγούμενων όρων αφού αυτοί δεν έχουν μνημονευτεί. Αυτό φυσικά οφείλεται καθαρά στο γεγονός της ιδιαιτερότητας της fib, καθώς η υλοποίηση είναι αναδρομική και ο υπολογισμός του n-όρου στηρίζεται στους όρους n-1 και n-2. Αν θα θέλαμε να γίνεται χρήση των προηγούμενων υπολογισμών για μεγαλύτερη αποδοτικότητα τότε η fibT θα έπρεπε να γραφεί με τον παρακάτω τρόπο δηλώνοντας ουσιαστικά στις αναδρομικές κλήσεις να ακολουθούνε την υλοποίηση που υπόκειται σε μνημόνευση.

```
const withTimeAndMemo = composeTwo(withTime, memo);

const fibT = withTimeAndMemo(n => n <= 1 ? n : fibT(n - 1) + fibT(n - 2));
```

Ένα εύλογο ερώτημα που προκύπτει εάν μελετήσει κανείς τα προηγούμενα είναι το τι γίνεται όταν η είσοδος σε μία συνάρτηση που έχει εμπλουτιστεί με την ιδιότητα της μνημόνευσης είναι κάποια σύνθετη δομή δεδομένων και όχι κάποια πρωτόγονη όπως στα συγκεκριμένα παραδείγματα. Οι συναρτήσεις ***f*** και ***fib*** δέχονται ως είσοδο αριθμούς που είναι εξ ορισμού αμετάβλητοι και απεικονίζονται μοναδικά στη χρήση τους ως κλειδιά στον αντικείμενο της **cache.** Επίσης στους τύπους των αλφαριθμητικών καθώς και των αριθμών η συνθήκη ισότητας μεταξύ δύο κλειδιών είναι ξεκάθαρη, πράγμα το οποίο δεν ισχύει για δύο σύνθετες δομές δεδομένων. Αν για παράδειγμα είχαμε δύο αντικείμενα **a** και **b** ενός συνόλου ***C***, τότε η ισότητα μεταξύ τους είναι κάτι αυθαίρετο που πρέπει να σκιαγραφείτε από τον προγραμματιστή. Για κάποιον δύο αντικείμενα αυτού του τύπου θεωρούνται ίσα όταν τα πρωτόγονα χαρακτηριστικά τους **y** είναι ίσα. Δηλαδή αν το απεικονίσουμε σε κώδικα

```
const C = (name, value) => ({
    name,
    y: value
});

const a = C("random1", 15);
const b = C("random2", 15);

a === b;
//-> True only if a.y === b.y
```

Αυτό πρακτικά σημαίνει ότι αν καλέσουμε μία συνάρτηση ***g*** που υλοποιεί τη λειτουργικότητα της μνημόνευσης με παράμετρο το αντικείμενο **a** αρχικά και στη συνέχεια την ίδια συνάρτηση αλλά με είσοδο το στοιχείο **b** τότε κατά τη δεύτερη κλήση της ***g***, δηλαδή ***g*(*b*)**, θα μας επιστραφεί η τιμή που είναι αποθηκευμένη στην **cache** δηλαδή το ***g*(*a*)**.

Η πιο γενική περίπτωση και αυτή που θα εξετάσουμε εμείς είναι η ισότητα των σύνθετων δομών δεδομένων να ισχύει όταν όλα τα χαρακτηριστικά τους είναι ίσα ένα προς ένα και μάλιστα σε βάθος (**deep equality**). Για να το καταφέρουμε αυτό χρησιμοποιούμε ένα εργαλείο μετατροπής της σύνθετης δομής σε αλφαριθμητικό (σαν **serialization**), και συνεπώς η ισότητα μεταξύ δύο τέτοιων αντικειμένων θα ανάγεται σε ισότητα μεταξύ των αλφαριθμητικών στα οποία απεικονίζονται. Το εργαλείο αυτό είναι η συνάρτηση **JSON.stringify .** Ας δούμε τα αλφαριθμητικά που εξάγει για διάφορες σύνθετες εισόδους.

```
JSON.stringify({name: "felix", age: 27});
//-> {"name":"felix","age":27}
JSON.stringify([10, 11, 12, 24, 25, 26]);
//-> [10,11,12,24,25,26]
JSON.stringify({ name: "random", values: [10, 11, 12, 24, 25, 26]});
//-> {"name":"random","values":[10,11,12,24,25,26]}
```

Βλέπουμε ότι η συνάρτηση JSON.stringify μετατρέπει επιτυχώς τις σύνθετες δομές σε τύπους **String.** Για την ακρίβεια τα μετατρέπει σε μορφή ερμηνεύσιμη από τη σύμβαση JSON (JavaScript Object notation) που αποτελεί μία τεχνική σειριοποίησης για αποστολή των δεδομένων μέσω πρωτοκόλλου **HTTP.** Για τη συγγραφή λοιπόν τις αντίστοιχης **memo** απαιτούνται μικρές αλλαγές όπως φαίνονται παρακάτω

```
// memoS :: (a -> b) -> a -> b
const memoS = h => {
    const cache = Object.create(null);
    return x => {
        const key = JSON.stringify(x);
        return cache[key] || (cache[key] = h(x), cache[key]);
    }
};
```

Παρατηρούμε ότι αντί για **cache[x]** που είχαμε στην περίπτωση της memo, υπολογίζουμε αρχικά το κλειδί key που είναι το αποτέλεσμα κλήσης της **JSON.stringify** στην είσοδο της συνάρτησης, και έπειτα με βάση αυτό το κλειδί κάνουμε αναζήτηση η

προσθήκη αντίστοιχα στην **cache**. Ας δούμε πως θα λειτουργούσε η συνάρτηση υψηλής τάξης αν δεχόταν σαν είσοδο τη συνάρτηση ταξινόμησης (**quicksort**) που μελετήθηκε στο υποκεφάλαιο της αναδρομής. Επιλέγουμε αυτή τη συνάρτηση γιατί δέχεται ως παραμέτρους λίστες που αποτελούν μία μη πρωτόγονη δομή δεδομένων

```
// memoS :: (a -> b) -> a -> b
const memoS = h => {
    const cache = Object.create(null);
    return x => {
        const key = JSON.stringify(x);
        return cache[key] || (cache[key] = h(x), cache[key]);
    }
};

const tap = f => x => (logger("I am doing the calculation"), f(x));

// sort :: Ord a => [a] -> [a]
const sort = ([first, ...rest]) => first === undefined ? [] :
[...sort(rest.filter(e => e < first)), first, ...sort(rest.filter(e =>
e >= first))];

const sortM = memoS(tap(sort));

sortM([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
//-> I am doing the calculation
//-> [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
sortM([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
//-> [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

Βλέπουμε ότι αφού καλέσαμε τη συνάρτηση **sortM** δύο φορές με την ίδια είσοδο (λίστα [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]) η συνάρτηση **sort** έκανε τον υπολογισμό μόνο μία φορά όπως φαίνεται και βοηθητικά από τη συνάρτηση υψηλής τάξης **tap.** Τη δεύτερη φορά ή **memoS** παρουσίασε **hit** στην cache και συνεπώς το αποτέλεσμα επιστράφηκε απευθείας δίχως να γίνει ο ακριβός υπολογισμός της ταξινόμησης. Και σαν τελευταία προσθήκη αξίζει να σημειωθεί ότι η σύνταξη της **memoS** θα μπορούσε να υλοποιηθεί υπό πιο αυστηρή συναρτησιακή προσέγγιση κάνοντας χρήση της περιβόητης **composeTwo** όπως απεικονίζεται παρακάτω

```
// memo :: (a -> b) -> a -> b
const memo = h => {
    const cache = Object.create(null);
    return (key, x) => cache[key] || (cache[key] = h(x), cache[key]);
};

// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (f, g) => x => f(g(x));

const stringify = f => x => f(JSON.stringify(x), x);

const memoS = composeTwo(stringify, memo);

const tap = f => x => (logger("I am doing the calculation"), f(x));

// sort :: Ord a => [a] -> [a]
const sort = ([first, ...rest]) => first === undefined ? [] :
[...sort(rest.filter(e => e < first)), first, ...sort(rest.filter(e =>
e >= first))];

const sortM = memoS(tap(sort));

sortM([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
//-> I am doing the calculation
//-> [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
sortM([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]);
//-> [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

Στο σημείο αυτό ολοκληρώνουμε με την ισχυρή τεχνική της μνημόνευσης που όπως πήγαμε πηγάζει από τις βασικές αρχές που διέπουν τον συναρτησιακό προγραμματισμό. Φυσικά πρέπει να σημειωθεί ότι η υλοποίηση της τεχνικής αυτής μπορεί να γίνει πολύ πιο σύνθετη και αποδοτική αλλά η ανάλυση της ξεφεύγει από τα πλαίσια της παρούσας εργασίας όπου σκοπός είναι η μετάδοση της βασικής ιδέας. Στο επόμενο υποκεφάλαιο θα μελετηθεί εις βάθος η πράξη της σύνθεσης συναρτήσεων.

4.7 Σύνθεση Συναρτήσεων (Function Composition)

Στο προηγούμενο κεφάλαιο έγινε μία νύξη για το πως ακριβώς επιτελείται η πράξη της σύνθεσης συναρτήσεων και την έντονη σημασία της στο μοντέλο του συναρτησιακού προγραμματισμού αλλά και της αντίστοιχης μαθηματικής θεωρίας στην οποία αυτός στηρίζεται. Πράγματι η σύνθεση συναρτήσεων, υπό την έννοια της σύνθεσης ποικίλων λειτουργιών αποτελεί τη βασική ιδέα του προγραμματισμού ανεξαρτήτως του μοντέλου (paradigm) το οποίο υιοθετείται. Όταν προσπαθούμε να λύσουμε ένα πολύπλοκο πρόβλημα, το διασπάμε σε μικρότερα, απλούστερα, δίνοντας λύσεις σε αυτά σε μία διαδικασία που θα λέγαμε αποσυνθέτει και κατακερματίζει το αρχικό πρόβλημα. Με τον ίδιο τρόπο λοιπόν, όταν το στάδιο της αποσύνθεσης έχει ολοκληρωθεί, πρέπει να είμαστε σε θέση να προσδώσουμε έναν τρόπο ικανό να ενώσει τα διάσπαρτα κομμάτια, που αρχικά μπορεί να φαίνονται ασυσχέτιστα μεταξύ τους αλλά εάν η αποσύνθεση ευσταθεί λογικά τότε υπάρχει τουλάχιστον ένας ορθός τρόπος σύνθεσης, σε μία σωστή διάταξη που θα έχει νόημα και τελικώς θα επιλύει το αρχικό πρόβλημα. Το στάδιο αυτό ονομάζεται στάδιο σύνθεσης (**composition**) και αποτελεί τον βασικό πυλώνα του προγραμματισμού. Είτε πρόκειται για προστακτικό μοντέλο γραφής, είτε υιοθετείται ο αντικειμενοστρεφής σχεδιασμός, είτε τον συναρτησιακό μοντέλο είτε οτιδήποτε άλλο, ο κύκλος της σύνθεσης και της αποσύνθεσης θα είναι παρόν γιατί πολύ απλά έτσι λειτουργεί ή ανθρώπινη αντίληψη. Συνεπώς οτιδήποτε στον κόσμο μας ακολουθεί αυτή τη θεωρία, εφόσον είναι προϊόν της ανθρώπινης επέμβασης, με χαρακτηριστικό παράδειγμα τις επιστήμες που χτίζονται και εξελίσσονται διά της σύνθεσης (Milewski, 2018).

Αρκετά με τη φιλοσοφική θεώρηση, που όμως στόχο είχε να τονίσει τη σημασία τη σύνθεσης ευρύτερα, και ας δούμε την απήχηση της στον συναρτησιακό προγραμματισμό και την υπόσταση της στο περιβάλλον της JavaScript.

Γνωρίζουμε ότι ο συναρτησιακός προγραμματισμός ενθαρρύνει την επίλυση προγραμματιστικών προβλημάτων κάνοντας χρήση συναρτήσεων υπό τη μαθηματική τους όμως υπόσταση (αγνές και χωρίς παράπλευρες τροποποιήσεις). Συνεπώς, αν τον συναρτησιακό μοντέλο αντιμετωπίζει ένα πρόβλημα διαιρώντας το σε μικρότερα, και για το καθένα από αυτά προσφέρει λύση υπό τη μορφή μίας συνάρτησης που υποστηρίζει τη μαθηματική της ιδιότητα, τότε το στάδιο της επίλυσης του αρχικού προβλήματος, αυτό της σύνθεσης δηλαδή, θα πρέπει να μοιάζει με τον μαθηματικό ορισμό της σύνθεσης

συναρτήσεων (**function composition**). Και πράγματι έτσι έχει. Πρώτα όμως ας ξαναδούμε τη μαθηματική θεωρία πίσω από τη σύνθεση συναρτήσεων.

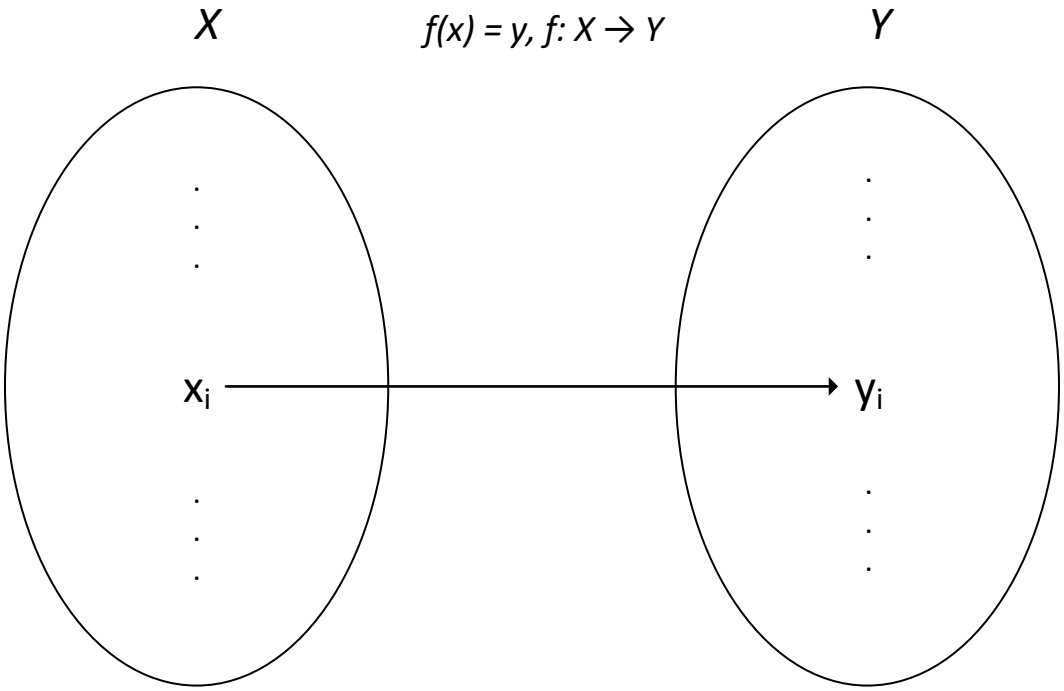
Έστω ότι έχουμε τις συναρτήσεις

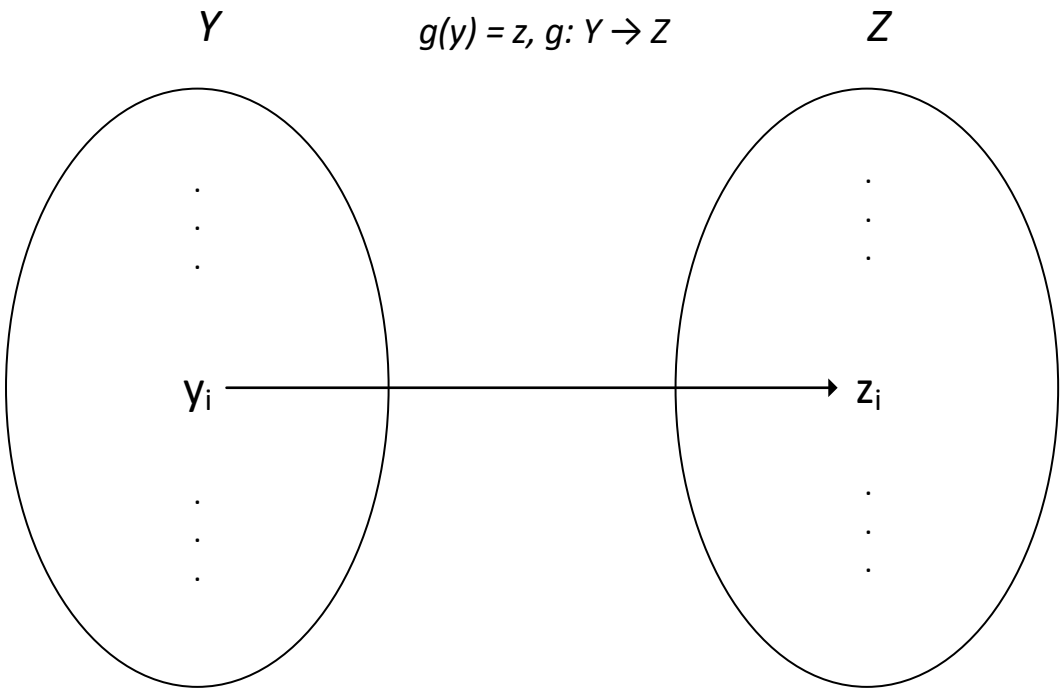
$$f(x) = y, \quad x \in X, y \in Y \quad \text{ή} \quad f: X \to Y$$

και

$$g(y) = z, \quad y \in Y, z \in Z \quad \text{ή} \quad g: Y \to Z$$

τότε η f εκφράζει τις απεικονίσεις των στοιχείων του συνόλου X στο σύνολο Y και ή g εκφράζει τις απεικονίσεις των στοιχείων του συνόλου Y στο σύνολο Z όπως φαίνεται στα παρακάτω σχήματα



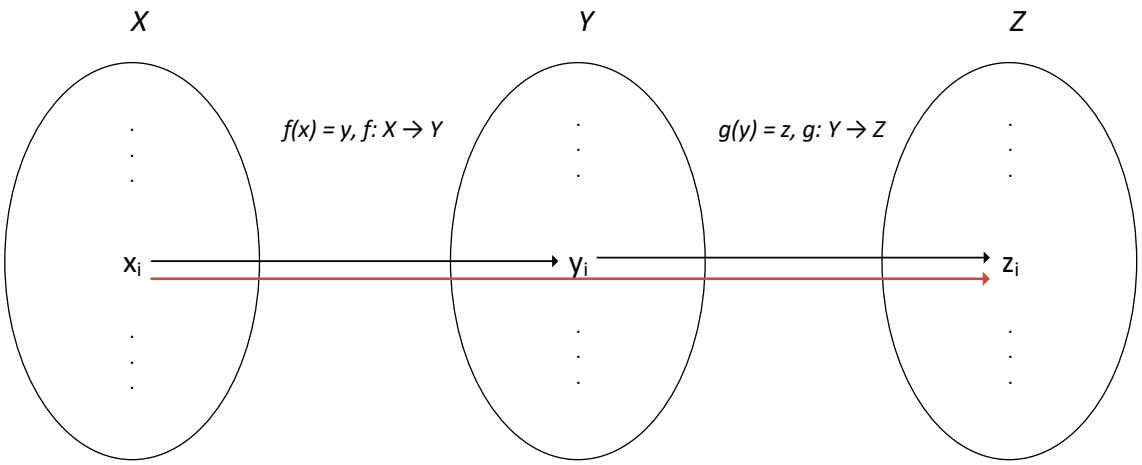


Τότε με την προϋπόθεση ότι το πεδίο τιμών μία συνάρτησης (στην προκειμένη περίπτωση της f) αποτελεί το πεδίο ορισμού μίας άλλης (στην προκειμένη περίπτωση της g) , μπορούμε να εκφράσουμε μία σύνθετη συνάρτηση η οποία απεικονίζει το σύνολο του πεδίο ορισμού της πρώτης X, στο σύνολο τιμών της δεύτερης Y. Η σύνθετη αυτή συνάρτηση συμβολίζεται με $g \circ f$ και για αυτή ισχύει

$$(g \circ f)(x) = z, \qquad x \in X, z \in Z \text{ ή } g \circ f : X \to Z$$

Στην πραγματικότητα, αν δούμε το αποτέλεσμα της σύνθεσης σχηματικά υπό τη μορφή βελών θα δούμε ότι πρόκειται για μία απλή επέκταση των απεικονίσεων που πηγάζει τον μαθηματικό ορισμό της συνάρτησης.

$(g \circ f)(x) = z,\ g \circ f : X \to Z$



Σχήμα 4-7 Απεικόνιση σύνθεσης

Στο σημείο αυτό πιστεύω θα ήταν ωφέλιμο να γίνει μία μικρή αναφορά στη θεωρία κατηγοριών η οποία προσφέρει ένα είδος τυποποίησης των μαθηματικών δομών σε μια αναπαράσταση κατευθυνόμενων γράφων, θα λέγαμε, των οποίων οι κόμβοι (nodes) ονομάζονται αντικείμενα (**objects**) και οι ακμές (edges), βέλη (arrows) ή μορφισμοί (**morphisms**)**.** Ως κατηγορία λοιπόν νοείται οποιαδήποτε υψηλού επιπέδου μαθηματική δομή η οποία μπορεί να τυποποιηθεί με βάση τη συγκεκριμένη αναπαράσταση και υπακούει σε δύο βασικούς νόμους. Ο πρώτος είναι η δυνατότητα της σύνθεσης των μορφισμών υπακούοντας την προσεταιριστική ιδιότητα και ο δεύτερος η ύπαρξη του ταυτοτικού μορφισμού (**identity morphism**) για κάθε αντικείμενο. Τι σχέση έχουν όμως όλα αυτά με τη σύνθεση συναρτήσεων. Όπως αποδεικνύεται στη θεωρία κατηγοριών και συγκεκριμένα στη κατηγορία των συνόλων (category of sets) οι μορφισμοί αναπαρίστανται ως συναρτήσεις οι οποίες απεικονίζουν τα αντικείμενα της κατηγορίας (τα σύνολα) μεταξύ τους. Φυσικά αν συγκεκριμενοποιήσουμε ακόμα περισσότερό την ανάλυση μπορούμε να ορίσουμε ως αντικείμενα όλα τα στοιχεία ενός συνόλου, ως μορφισμούς όλες τις συναρτήσεις που εφαρμόζονται και μετασχηματίζουν τα αντικείμενα και υπακούνε στους δύο βασικούς προαναφερόμενους νόμους, και δημιουργήσαμε από μόνοι μας μία κατηγορία (Milewski, 2018).

Έτσι στο παράδειγμα μας μπορούμε να υποθέσουμε ότι οι συναρτήσεις f, g αποτελούν μορφισμούς και τα στοιχεία x_i, y_i, z_i των συνόλων X, Y, Z αποτελούν αντικείμενα. Για την εγκαθίδρυση όμως μίας κατηγορίας απαιτείται και η απόδειξη των δύο νόμων που προαναφέραμε.

1) Οι μορφισμοί μπορούν να υποβληθούν στην πράξη της σύνθεσης υπακούοντας στην προσεταιριστική ιδιότητα

Με λίγα λόγια αν ορίσουμε τρεις μορφισμούς (συναρτήσεις)

$$f(x) = y, \quad x \in X, y \in Y \quad \text{ή} \quad f: X \to Y$$

και

$$g(y) = z, \quad y \in Y, z \in Z \quad \text{ή} \quad g: Y \to Z$$

και

$$h(z) = w, \quad z \in Z, w \in W \quad \text{ή} \quad h: Z \to W$$

πρέπει να αποδείξουμε ότι

$$h \circ g \circ f = (h \circ g) \circ f = h \circ (g \circ f) \ : X \to W$$

Η παραπάνω ισότητα ισχύει καθώς

$$h \circ g \circ f = h\Big(g\big(f(x)\big)\Big)$$

$$(h \circ g) \circ f = h\big(g(x)\big)(f) = h(g\big(f(x)\big)$$

$$h \circ (g \circ f) = h \circ \Big(g\big(f(x)\big)\Big) = h(g(f(x)))$$

Από τις παραπάνω σχέσεις συμπεραίνουμε ότι ισχύει και η προς απόδειξη ισότητα

2) Η ύπαρξη ταυτοτικού μορφισμού για κάθε αντικείμενο Α ο οποίος αν βρεθεί ως παράγοντας στην πράξη της σύνθεσης με έναν οποιοδήποτε μορφισμό Μ (βέλος) που αρχίζει από το Α η καταλήγει στο Α, επιστρέφει τον ίδιο μορφισμό Μ.

Ο ταυτοτικός μορφισμός στον κόσμο των συναρτήσεων ορίζεται ως μία συνάρτηση της μορφής

$$id(\alpha) = \alpha, \quad \alpha \in A \quad \text{ή} \quad id: A \to A, \text{όπου } A \text{ ένα αυθαίρετο σύνολο}$$

και ονομάζεται **identity morfism**,

Έτσι αν έχουμε μία συνάρτηση της μορφής

$$f(x) = y, \quad x \in X, y \in Y \quad \text{ή} \quad f: X \to Y$$

Τότε για κάθε $x_i \in X$ πρέπει να αποδείξουμε ότι

$$f \circ id = id \circ f = f$$

Πράγματι αν αναλύσουμε την πράξη της σύνθεσης έχουμε

$$f \circ id = f\big(id(x)\big) = f(x) = f \quad και \quad id \circ f = id\big(f(x)\big) = f(x) = f$$

Συνεπώς υπάρχει συνάρτηση που ικανοποιεί την παραπάνω συνθήκη και όπως είπαμε αυτή αποτελεί τον ταυτοτικό μορφισμό της κατηγορίας.

Ο μορφισμός identity ονομάζεται και ταυτοτική συνάρτηση στην κατηγορία των συνόλων ή ταυτοτικός συνδυαστής στη συνδυαστική λογική.

Αρκετά όμως με τη μαθηματική ανάλυση. Σκοπός της ήταν να μεταδώσει στον αναγνώστη ότι γύρω από την πράξη της σύνθεσης μπορούν να χτιστούν μαθηματικές δομές τις οποίες μπορεί να αξιοποιήσει ο συναρτησιακός προγραμματισμός. Το πρόγραμμα προς επίλυση, διασπάται σε ξεχωριστές λειτουργίες υπό τη μορφή αγνών συναρτήσεων, και η πράξη της σύνθεσης προσφέρει τη συγκολλητική ουσία όπου οι λειτουργικότητες αυτές ενώνονται για να χρησιμοποιηθούν ως μία οντότητα. Τελικώς, τα δεδομένα του προβλήματος εισάγονται, μετασχηματίζονται από τη συνθετική συνάρτηση σε μία αναπαραστατική μορφή ροής, θα λέγαμε, και εξάγονται διαμορφώνοντας το προσδοκώμενο αποτέλεσμα.

Στον προγραμματισμό η πράξη της σύνθεσης συναρτήσεων παίρνει τη μορφή της σύνθεσης λειτουργιών. Συναρτησιακές γλώσσες όπως η Haskell προσφέρουν εγγενή υλοποίηση της πράξης (χρησιμοποιώντας ως τελεστή την τελεία .) με δεξιά προς αριστερά σειρά σύνθεσης. Επίσης στα λειτουργικά συστήματα των UNIX, η σύνθεση εκφράζεται με τον τελεστή | (pipe operator) όπου η σειρά σύνθεσης αλλάζει και τα δεδομένα ρέουν από αριστερά προς τα δεξιά (ουσιαστικά υλοποιείται η πράξη της σωλήνωσης). Η JavaScript όντας γλώσσα αντικειμενοστρεφής αλλά με δυνατότητα συναρτησιακής προσέγγισης (όπως αποδεικνύει η παρούσα διπλωματική) δεν προσφέρει έτοιμη υλοποίηση για πράξη της σύνθεσης συναρτήσεων (αν και υπάρχουν ενδείξεις ότι θα προστεθεί σε μελλοντικές εκδόσεις της γλώσσας), συνεπώς οι προγραμματιστές θα πρέπει να συντάξουν τη συνάρτηση από μόνοι τους ή φυσικά να κάνουν χρήση μίας βιβλιοθήκης.

Ας μελετήσουμε την υπόσταση της στη γλώσσα της JavaScript. Στην πραγματικότητα η υλοποίηση της δεν διαφέρει από αυτήν που είδαμε στο παράδειγμα του προηγούμενου κεφαλαίου. Αναλυτικά εάν θέλουμε να συνθέσουμε δύο συναρτήσεις έχουμε

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));
```

Αν η συνάρτηση **g** απεικονίζει στοιχεία του συνόλου **b** στο σύνολο **c**, και η συνάρτηση **f** απεικονίζει στοιχεία του συνόλου **a** στο **b** τότε η σύνθεση τους είναι μία νέα συνάρτηση που απεικονίζει τα στοιχεία του συνόλου **a** απευθείας στο σύνολο **c.** Παρατηρούμε ότι η **composeTwo**, δεν κάνει τίποτα παραπάνω από το να υλοποιεί τη μαθηματική αλγεβρική πράξη της σύνθεσης συναρτήσεων όπως αυτή ορίστηκε στον παράδειγμα της θεωρίας κατηγοριών. Επίσης εμβαθύνοντας και αφού αποδείξαμε ότι η πράξη της σύνθεσης ακολουθεί τους δύο βασικούς νόμους που προϋποθέτει η σύσταση μίας κατηγορίας, μπορούμε να γράψουμε

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(composeTwo(h, g), f);
```

ή

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(h, composeTwo(g, f));
```

Όπου **h** μία συνάρτηση που απεικονίζει τα στοιχεία ενός συνόλου **c** στο σύνολο **d**. Εκμεταλλευόμενοι λοιπόν την προσεταιριστική ιδιότητα η σύνθεση παραπάνω από δύο συναρτήσεων παίρνει τη μορφή που απεικονίζεται πιο πάνω. Θα δούμε λίγο πιο κάτω πως ορίζεται η σύνθεση **N** συναρτήσεων ή **composeN**, αλλά με αυτά που είπαμε πρέπει να έχουμε συλλάβει μία κατεύθυνση (αναδρομή). Επίσης για την ταυτοτική συνάρτηση **id** ισχύει

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// id :: z -> z
const id = x => x;

composeTwo(f, id);
//-> f

composeTwo(id, f);
//-> f
```

Η ταυτοτική συνάρτηση δέχεται στοιχεία ενός αυθαίρετου συνόλου **z** και τα απεικονίζει στον εαυτό τους. Συνεπώς η σύνθεση αυτής της συνάρτησης με οποιαδήποτε άλλη **f,** επιστρέφει την ίδια την **f** όπως και γίνεται. Η συνάρτηση αυτή αποτελεί το ουδέτερο στοιχείο σε πράξεις μεταξύ συναρτήσεων και χρησιμοποιείται κατά κόρον σε συναρτήσεις υψηλών τάξεων με το ίδιο τρόπο που χρησιμοποιείται το μηδέν στην άλγεβρα.

Αφού είδαμε τον προγραμματιστικό ορισμό της σύνθεσης πάμε να δούμε ένα πολύ απλό παράδειγμα χρήσης και αναπαράστασης της.

Θεωρούμε ότι έχουμε το απλό αριθμητικό πρόβλημα της μετατροπής μίας θερμοκρασίας από την κλίμακα των Fahrenheit στην κλίμακα των Celsius. Πως θα το προσεγγίζαμε χρησιμοποιώντας την πράξη της σύνθεσης;

Ως γνωστόν ο τύπος της μετατροπής είναι

$$c = (f - 32) * 5/9,$$ όπου f οι βαθμοί Fahrenheit

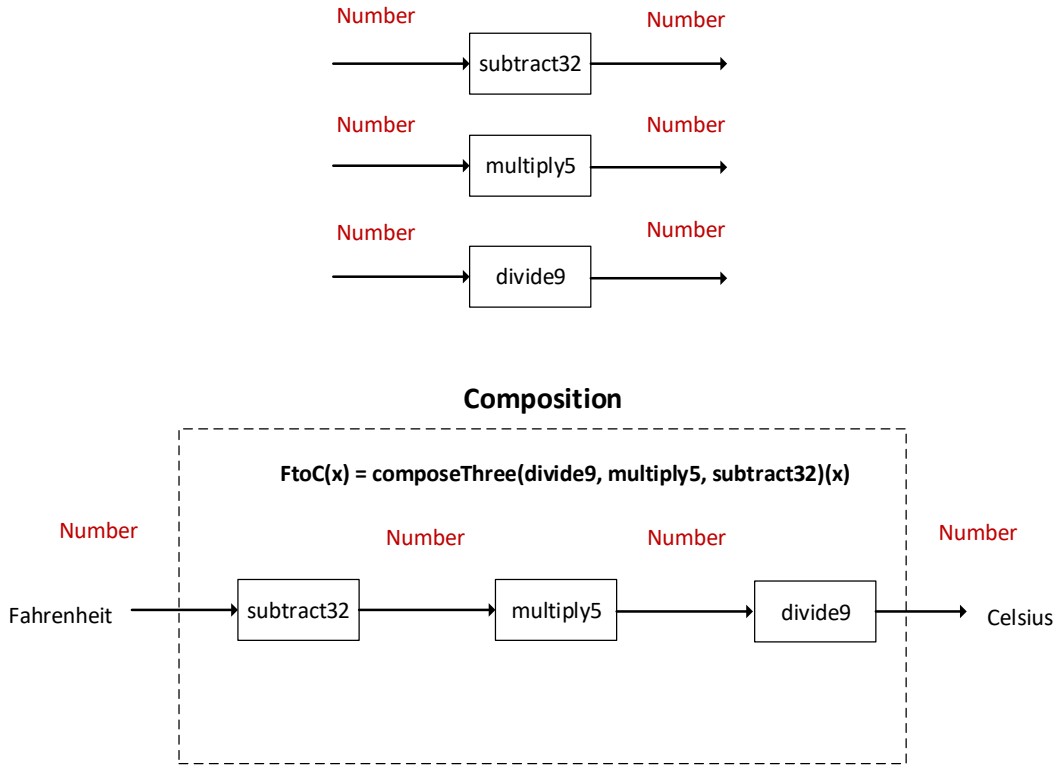
Φυσικά λόγω της απλότητας του παραδείγματος θα μπορούσαμε να συντάξουμε απευθείας τη συνάρτηση μετατροπής αλλά θα χάναμε την ουσία του παραδείγματος. Αναφέραμε ότι η προσέγγιση οποιουδήποτε προβλήματος πραγματοποιείται από το στάδιο της διάσπασης των σύνθετων λειτουργιών, την εύρεση λειτουργικοτήτων για αυτές και τελικώς τη σύνθεση τους για την επίλυση του αρχικού προβλήματος. Στην προκειμένη περίπτωση αν θεωρήσουμε ότι ο τύπος αναπαριστά το πολύπλοκο πρόβλημα προς επίλυση, θα μπορούσαμε να προχωρήσουμε στη διάσπαση του σε απλούστερα υποπροβλήματα υπό τη μορφή των τριών διαφορετικών αριθμητικών πράξεων που

εμφανίζονται. Στη συνέχεια με την κατάλληλη σύνθεση των επιμέρους λειτουργιών θα καταλήγαμε στη σύνταξη του τελικού τύπου και συνεπώς επίλυσης του προγραμματιστικού προβλήματος. Ας δούμε την υλοποίηση στην JavaScript

```
// subtract32 :: Number -> Number
const subtract32 = x => x - 32;

// multiply5 :: Number -> Number
const multiply5 = x => x * 5;

// divide9 :: Number -> Number
const divide9 = x => x / 9;

// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(h, composeTwo(g, f));

// FtoC :: Number -> Number
const FtoC = composeThree(divide9, multiply5, subtract32);

FtoC(32);
//-> 0
```

Παρατηρούμε ότι δημιουργήσαμε τρεις απλές λειτουργικότητες υπό τη μορφή συναρτήσεων. Η **subtract32** αναλαμβάνει το έργο της αφαίρεσης με αφαιρέτη τον αριθμό 32, η **multiply5** τον πολλαπλασιασμό με τον αριθμό 5, και η **divide9** υλοποιεί την πράξη της διαίρεσης με διαιρέτη τον αριθμό 9. Φυσικά η δήλωση των συναρτήσεων δεν είναι τυχαία αλλά προκύπτει, όπως είπαμε από την ερμηνεία και διάσπαση του αρχικού προβλήματος που στην προκειμένη περίπτωση είναι ο τύπος της μετατροπής που διατυπώνεται στην εξίσωση $c = (f - 32) * 5/9$. Στη συνέχεια ακολουθεί η πράξη της σύνθεσης που στόχο έχει να ενώσει τις επιμέρους λειτουργικότητες σε μία ορθή διάταξη που θα δώσει επίλυση στο αρχικό πρόβλημα. Έτσι η συνάρτηση **composeThree** δέχεται ως παραμέτρους τις συναρτήσεις **divide9**, **multiply5** και **subtract32** σχηματίζοντας έτσι μία σύνθετη διάταξη που αναλαμβάνει να μετασχηματίσει τις πιθανές εισόδους που θα δεχθεί σε εξόδους που ικανοποιούν το πρόβλημα προς επίλυση, δηλαδή τον τύπο παραπάνω τύπο. Αν θα θέλαμε να αναπαραστήσουμε σχηματικά την επίλυση του

προβλήματος τότε θα ήταν ωφέλιμο από πλευράς ερμηνείας, αυτή να γίνει υπό τη μορφή συστημάτων όπως απεικονίζεται παρακάτω



Παρατηρούμε ότι οι τρεις διαφορετικές λειτουργικότητες στις οποίες διασπάστηκε το αρχικό πρόβλημα μπορούν να λειτουργήσουν αυτόνομα ως μεμονωμένα συστήματα που δέχονται μία είσοδο και εξάγουν μία έξοδο. Η πράξη της σύνθεσης αναλαμβάνει να ενώσει τα αυτόνομα αυτά συστήματα σε ένα νέο σύστημα που επίσης έχει μία είσοδο και εξάγει ένα αποτέλεσμα ως έξοδο. Το νέο αυτό σύστημα είναι επίσης πλήρως αυτόνομο και έτοιμο να υποβληθεί σε μία πράξη νέας σύνθεσης, αν αυτό απαιτηθεί, για να δημιουργήσει με τη σειρά του νέες λειτουργικότητες. Αντιλαμβανόμαστε δηλαδή ότι έχουμε μια φυσική επαγωγική προσέγγιση ικανή να χρησιμοποιηθεί ως μοντέλο λύσης για οποιοδήποτε πρόβλημα καλούμαστε να αντιμετωπίσουμε. Και πάνω από όλα η προσέγγιση αυτή, που στην πραγματικότητα περιγράφει στη βάση της ο συναρτησιακός προγραμματισμός, στηρίζεται σε μαθηματικές έννοιες, όπως είναι ο λογισμός λάμδα, που έχουν μελετηθεί και αποδειχτεί εδώ και αρκετά χρόνια.

Επιπρόσθετα όσον αφορά στο παραπάνω πρόβλημα πρέπει να πούμε ότι ο συναρτησιακός προγραμματισμός ενθαρρύνει τη συγγραφή αφαιρετικών λειτουργικοτήτων με σκοπό να προνοήσει πιθανές εκδοχές παρόμοιων προβλημάτων που θα προκύψουν στο μέλλον. Με τον τρόπο αυτό τον συναρτησιακό μοντέλο γραφής πετυχαίνει ότι και οι αφηρημένες κλάσεις ή διεπαφές στην αντικειμενοστρέφεια, εξαλείφοντας έτσι το τεχνικό κόστος που θα πρόεκυπτε από πιθανή αφαίρεση η προσθήκη μίας λειτουργικότητας που καλύπτει η συγκεκριμένη αφαίρεση (abstraction). Στην προκειμένη περίπτωση, αν και το παράδειγμα είναι εξαιρετικά απλοϊκό, οι συναρτήσεις **divide9**, **multiply5** και **subtract32** θα μπορούσαν να δημιουργούνται δυναμικά από μία πηγή στην οποία θα εισάγεται κάτι συγκεκριμένο και θα παράγεται μία συγκεκριμένη λειτουργικότητα. Η τελευταία πρόταση περιγράφεται πλήρως από τον κύκλο του λογισμού λάμδα που αναλύθηκε στο εισαγωγικό κεφάλαιο, και αναφέρεται στην αφαίρεση λάμδα (**lambda abstraction**) και στην εφαρμογή (**application**). Και αφού γνωρίζουμε ότι οι εκφράσεις λάμδα αποτελούν στην ουσία μαθηματικές συναρτήσεις αρκεί να πούμε ότι η προσθήκη αφαιρετικών πλαισίων επιτυγχάνεται με συναρτήσεις. Στο συγκεκριμένο παράδειγμα, θα μπορούσαμε να γράψουμε

```
// subtract :: Number -> Number -> Number
const subtract = x => y => y - x;

// multiply :: Number -> Number -> Number
const multiply = x => y => x * y;

// divide :: Number -> Number -> Number
const divide = x => y => y / x;

// subtract32 :: Number -> Number
const subtract32 = subtract(32);

// multiply5 :: Number -> Number
const multiply5 = multiply(5);

// divide9 :: Number -> Number
const divide9 = divide(9);

// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(h, composeTwo(g, f));

// FtoC :: Number -> Number
const FtoC = composeThree(divide9, multiply5, subtract32);

FtoC(32);
//-> 0
```

Παρατηρούμε ότι οι συναρτήσεις **divide**, **multiply** και **subtract** δέχονται ως είσοδο μία παράμετρο και επιστρέφουν μία νέα λειτουργικότητα που ικανοποιεί έναν πιο συγκεκριμένο στόχο. Για παράδειγμα η **multiply5** που πηγάζει από τη συνάρτηση πολλαπλασιασμού **multiply,** επιτελεί την πράξη του πολλαπλασιασμού όπως ορίζει ακριβώς και ο παραγωγός της με τη διαφορά ότι ο ένας παράγοντας της πράξης είναι ο αριθμός 5, η παράμετρος δηλαδή που δέχτηκε η αφαίρεση λάμδα (συνάρτηση **multiply**). Προχωρώντας ένα βήμα παραπάνω θα μπορούσαμε να γράψουμε μία ακόμα πιο γενική συνάρτηση, από την οποία θα παράγονται οι συναρτήσεις **divide**, **multiply**, **subtract** και συνακόλουθα τα παράγωγα τους **divide9**, **multiply5** και **subtract32.** Η συνάρτηση αυτή φαίνεται παρακάτω

```
// binaryOp :: String -> Number -> Number
const binaryOp = operator => x => y => {
    switch (operator) {
        case "+": return y + x;
        case "-": return y - x;
        case "/": return y / x;
        case "*": return y * x;
    }
};
```

Η **binaryOp** δέχεται ως παράμετρο το σύμβολο της πράξης που θέλουμε να υλοποιήσουμε, και επιστρέφει μία νέα συνάρτηση, η οποία αναμένει μία νέα παράμετρο που θα καθορίσει τον έναν τελεστέο της πράξης και θα επιστρέψει μία νέα συνάρτηση η οποία περιμένει τον δεύτερο τελεστέο για να επιστρέψει το τελικό αποτέλεσμα. Χρησιμοποιώντας την **binaryOp** θα γράφαμε

```
// subtract :: Number -> Number -> Number
const subtract = binaryOp("-");

// multiply :: Number -> Number -> Number
const multiply = binaryOp("*");

// divide :: Number -> Number -> Number
const divide = binaryOp("/");

// subtract32 :: Number -> Number
const subtract32 = subtract(32);

// multiply5 :: Number -> Number
const multiply5 = multiply(5);

// divide9 :: Number -> Number
const divide9 = divide(9);

// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(h, composeTwo(g, f));

// FtoC :: Number -> Number
const FtoC = composeThree(divide9, multiply5, subtract32);

FtoC(32);
//-> 0
```

Βλέπουμε συνεπώς γενικά ότι ο συναρτησιακός προγραμματισμός αρχικά διασπάει το πρόβλημα σε απλούστερα (**decomposition**), προσφέρει αφαιρετικές λειτουργικότητες ικανές να προσεγγίσουν τα απλούστερα προβλήματα (**lambda abstractions**), περιορίζει το φάσμα των λειτουργικοτήτων μέσω εφαρμογών συγκεκριμένων προϋποθέσεων (**applications**), και τέλος συνθέτει τις επιμέρους λειτουργικότητες (**composition**) δίνοντας λύση στο αρχικό πρόβλημα.

Προτού αναλύσουμε ένα πιο πολύπλοκο παράδειγμα συναρτησιακής προσέγγισης ενός προβλήματος δια μέσου της σύνθεσης, πάμε να δούμε την υλοποίηση της σύνθεσης **compose** με έναν ακαθάριστο αριθμό συναρτήσεων εισόδου. Με λίγα λόγια αφού ορίσαμε τις **composeTwo, composeThree** ψάχνουμε τον γενικό ορισμό που ακολουθεί η πράξη της σύνθεσης για N πλήθος συναρτήσεων, δηλαδή **composeN.** Παρατηρούμε από τον ορισμό της σύνθεσης συναρτήσεων και από τις ιδιότητες τις οποίες αυτή ικανοποιεί (προσεταιριστική) ότι μπορεί να γραφεί

$$a \circ b \circ \dots x \circ y \circ z = a \circ b \circ \dots x \circ (y \circ z) = a \circ b \circ \dots \big(x \circ (y \circ z)\big)$$
$$= a \circ \Big(b \circ \dots \big(x \circ (y \circ z)\big)\Big) = \Big(a \circ \Big(b \circ \dots \big(x \circ (y \circ z)\big)\Big)\Big)$$

ή

$$a \circ b \circ \dots x \circ y \circ z = (a \circ b) \circ \dots x \circ y \circ z = \big((a \circ b) \circ \dots \circ x\big) \circ y \circ z$$
$$= \Big(\big((a \circ b) \circ \dots \circ x\big) \circ y\Big) \circ z = \Big(\Big(\big((a \circ b) \circ \dots \circ x\big) \circ y\Big) \circ z\Big)$$

Για την πρώτη έκφραση η οποία είναι εφαρμόζεται από τα δεξιά προς τα αριστερά (right associative) η προγραμματιστική αναπαράσταση θα είχε την παρακάτω μορφή

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(h, composeTwo(g, f));

// composeFour :: (d -> e, c -> d, b -> c, a -> b) -> a -> e
const composeFour = (w, h, g, f) => composeTwo(w, composeTwo(h,
composeTwo(g, f)));
```

Παρατηρούμε ότι διακρίνουμε μία επαναληπτική διαδικασία η οποία ως γνωστόν μπορεί να υλοποιηθεί αναδρομικά. Πράγματι μία εκδοχή της συνάρτησης σύνθεσης αυθαιρέτου αριθμού συναρτήσεων απεικονίζεται παρακάτω

```
// composeN :: (y -> z, x -> y, ..., b -> c, a -> b) -> a -> z
const composeN = (f, ...fns) => f === undefined ? x => x :
composeTwo(f, composeN(...fns));
```

και πιο σωστά αν θα θέλαμε να κρύψουμε την **composeTwo** από το εξωτερικό περιβάλλον θα γράφαμε

```
// composeN :: (y -> z, x -> y, ..., b -> c, a -> b) -> a -> z
const composeN = (f, ...fns) => {
    const composeTwo = (g, f) => x => g(f(x));
    return f === undefined ? x => x : composeTwo(f, composeN(...fns));
}
```

Για τη δεύτερη έκφραση η οποία είναι εφαρμόζεται από τα αριστερά προς τα δεξιά (**left associative**) η προγραμματιστική αναπαράσταση θα είχε την αντίστοιχη μορφή που απεικονίζεται παρακάτω

```
// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeThree :: (c -> d, b -> c, a -> b) -> a -> d
const composeThree = (h, g, f) => composeTwo(composeTwo(h, g), f);

// composeFour :: (d -> e, c -> d, b -> c, a -> b) -> a -> e
const composeFour = (w, h, g, f) =>
composeTwo(composeTwo(composeTwo(w, h), g), f);
```

Πάλι διακρίνουμε μία αναδρομική διαδικασία της οποίας η υλοποίηση διαφέρει από αυτή που φαίνεται στον κώδικα της παραπάνω **composeN.**

```
// composeN :: (y -> z, x -> y, ..., b -> c, a -> b) -> a -> z
const composeN = (f, ...fns) => fns.length === 0 ? f :
composeN(composeTwo(f, fns[0]), ...fns.slice(1));
```

Και αποκρύπτοντας τη δήλωση της **composeTwo** θα έχουμε

```
// composeN :: (y -> z, x -> y, ..., b -> c, a -> b) -> a -> z
const composeN = (f, ...fns) => {
    const composeTwo = (g, f) => x => g(f(x));
    return fns.length === 0 ? f : composeN(composeTwo(f, fns[0]),
...fns.slice(1));
}
```

Οι δύο εκφράσεις που είδαμε για την **composeN** είναι ισοδύναμες ως προς το αποτέλεσμα τους αλλά διαφέρουν στην υλοποίηση προσφέροντας η καθεμία της πλεονεκτήματα και μειονεκτήματα όσον αφορά ορισμένες προγραμματιστικές τεχνικές. Στην πραγματικότητα οι υλοποιήσεις αυτές περιγράφονται αφαιρετικά, από τις γνωστές στον συναρτησιακό προγραμματισμό συναρτήσεις, **foldLeft** και **foldRight,** (οι οποίες με μικρές τροποποιήσεις μπορούν να εκφραστούν και ως **reduceLeft**, **reduceRight**) που όπως είδαμε αποτελούν συναρτήσεις υψηλής τάξης προσπέλασης λιστών. Αυτό πρακτικά σημαίνει ότι η σύνθεση συναρτήσεων μπορεί να εκφραστεί και μέσω αυτών των δομών.

Τέλος ο κώδικας επίλυσης του προβλήματος της μετατροπής θερμοκρασιών θα μπορούσε να γραφεί με τον εξής τρόπο.

```
// subtract :: Number -> Number -> Number
const subtract = binaryOp("-");

// multiply :: Number -> Number -> Number
const multiply = binaryOp("*");

// divide :: Number -> Number -> Number
const divide = binaryOp("/");

// subtract32 :: Number -> Number
const subtract32 = subtract(32);

// multiply5 :: Number -> Number
const multiply5 = multiply(5);

// divide9 :: Number -> Number
const divide9 = divide(9);

// composeTwo :: (b -> c, a -> b) -> a -> c
const composeTwo = (g, f) => x => g(f(x));

// composeN :: (y -> z, x -> y, ..., b -> c, a -> b) -> a -> z
const composeN = (f, ...fns) => f === undefined ? x => x :
composeTwo(f, composeN(...fns));

// FtoC :: Number -> Number
const FtoC = composeN(divide9, multiply5, subtract32);

FtoC(32);
//-> 0
```

κάνοντας πλέον χρήση της **composeN**

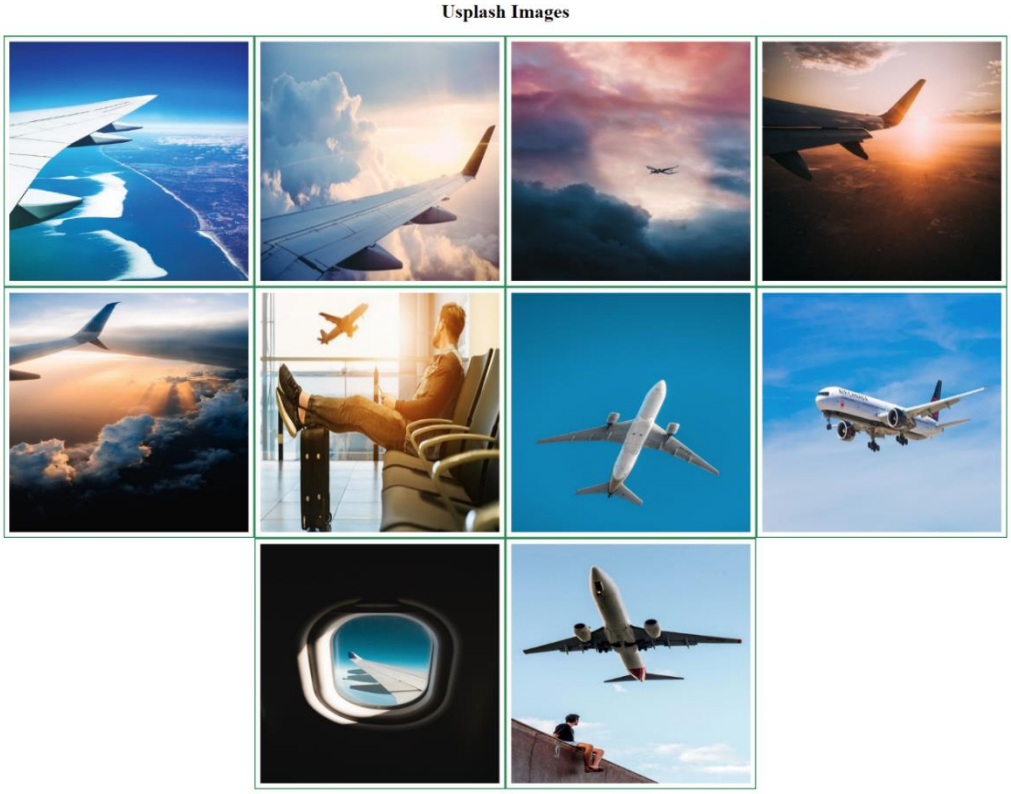
Αφού είδαμε πως μπορεί να υλοποιηθεί η συνάρτηση σύνθεσης αυθαιρέτου αριθμού συναρτήσεων **composeN**, πάνε να δούμε πως θα αντιμετώπιζε ο συναρτησιακός προγραμματισμός ένα πιο σύνθετο προγραμματιστικό πρόβλημα. Στο σημείο αυτό θα ήθελα να κάνω μία παρέμβαση και να αναφέρω ότι το μοντέλο του συναρτησιακού προγραμματισμού υπό τη μαθηματική του ερμηνεία και φτάνοντας σε ένα πολύ υψηλό αφαιρετικό επίπεδο μπορεί να προσεγγίσει οποιοδήποτε πρόβλημα, αλλά το γεγονός αυτό περιγράφει τη μία όψη του. Η άλλη όψη του αναλύεται υπό το πρίσμα της θεώρησης ότι ο κόσμος μας, και η ανθρώπινη αντίληψη για αυτόν και τα περιεχόμενα του, πηγάζει από το

γεγονός της χρονικής μεταβολής των καταστάσεων που τον διέπουν. Με λίγα λόγια δεν αρκεί η αόριστη αφαιρετική προσέγγιση ενός προβλήματος, γιατί πολύ απλά μία τέτοια λύση δεν θα γίνοταν ποτέ αντιληπτή από εμάς. Φανταστείτε ένα πρόγραμμα που δεν μεταβάλλει καμία κατάσταση. Πώς μπορεί να αποδείξει κάποιος ότι ένα τέτοιο πρόγραμμα εκτελείται και πραγματοποιεί του υπολογισμούς που πρέπει να πραγματοποιήσει. Για αυτό τον λόγο ο συναρτησιακός προγραμματισμός ενθαρρύνει από τη μία τη σύνταξη αφαιρετικών, αγνών συναρτησιακών δομών αλλά από την άλλη μεταβιβάζει στον χρήστη τους την ευθύνη για το στάδιο της εφαρμογής τους, το στάδιο δηλαδή στο οποίο έχουμε πραγματική μεταβολή των καταστάσεων που περιγράφουν τη λύση του προβλήματος. Και φυσικά δεν νοείται πρόγραμμα στο οποίο δεν υπάρχει το δεύτερο στάδιο, το στάδιο δηλαδή στο οποίο έχουμε μεταβολή κάποιας κατάστασης. Ούτως ή άλλως οι ηλεκτρονικοί υπολογιστές στηρίζονται λειτουργικά στο γεγονός της μεταβολής κάποιας κατάστασης σε συγκεκριμένους καταχωρητές, οπότε τα προγράμματα που θα γραφούν σε μία γλώσσα θα εκτελεστούν τελικά με βάση την αρχή που μόλις αναφέραμε. Οπότε καταλήγοντας τον συναρτησιακό μοντέλο προγραμματισμού δεν αγνοεί την ύπαρξη κάποιας μεταβολής στην κατάσταση του προγράμματος μας, κάτι τέτοιο θα ήταν ανώφελο άλλωστε, αλλά ευνοεί και ενθαρρύνει το σαφή διαχωρισμό των λειτουργιών που μεταβάλλουν κάποια κατάσταση στο πρόγραμμα μας, και αυτών που δεν το κάνουν (των αγνών λειτουργιών δηλαδή), κάνοντας έτσι πιο εύκολη την αιτιολόγηση του κώδικα τον οποίο καλούμαστε να συντάξουμε.

Ας δούμε λοιπόν ένα πιο σύνθετο πρόβλημα. Έστω ότι θέλουμε να φτιάξουμε ένα δυναμικό πρόγραμμα το οποίο δέχεται ως είσοδο μία λέξη και παράγει ως έξοδο μία λίστα από εικόνες οι οποίες μπορούν να εμφανιστούν στο σώμα μίας σελίδας που εμφανίζεται μέσω ενός περιηγητή. Πιο συγκεκριμένα το πρόγραμμα, θα πραγματοποιεί ασύγχρονα ένα **GET request** στη δικτυακή πλατφόρμα εικόνων **usplash.com** , και κατόπιν επεξεργασίας της απάντησης (στην οποία θα περιέχονται οι διευθύνσεις των ζητούμενων εικόνων), θα δημιουργούνται κόμβοι εικόνων που θα τοποθετούνται στο DOM (Document Object Model) μίας σελίδας. Η τελική λειτουργικότητα του προγράμματος θα έχει την παρακάτω μορφή.

```
// app :: String -> Promise
app("airplane");
```

και το αποτέλεσμα του υπολογισμού της έκφρασης **app(<term>)** θα δημιουργεί ένα DOM που οποιοσδήποτε περιηγητής (browser) θα μεταφράσει με τη μορφή που φαίνεται από κάτω



Η ανταλλαγή δεδομένων μεταξύ διαφορετικών οντοτήτων, στη προκειμένη περίπτωση, μεταξύ ενός καταναλωτή (η εφαρμογή μας) και ενός παραγωγού (διακομιστή **usplash.com**) αποτελεί ένα συνηθισμένο φαινόμενο στον προγραμματισμό και είναι το μοντέλο που έκανε τον παγκόσμιο ιστό να έχει τη μορφή που παρουσιάζει στις μέρες μας. Η JavaScript, δεν είναι κρυφό ότι αποτελεί το βασικό εργαλείο με το οποίο υλοποιούνται τέτοιες λειτουργικότητες για πολλά χρόνια τώρα, συνεπώς το πρόβλημα που παρουσιάζουμε είναι σίγουρα γνώριμο σε οποιοδήποτε προγραμματιστή που έχει ασχοληθεί με την ανάπτυξη web προγραμμάτων. Αυτό όμως που δεν είναι γνωστό στην πλειοψηφία είναι η προσέγγιση με την οποία θα αναλύσουμε το πρόβλημα. Και αυτή όπως

θα δούμε διαφέρει αρκετά με την πιθανή υλοποίηση που θα προτείναμε όντας εξοικειωμένη με το προστακτικό μοντέλο προγραμματισμού και την αντικειμενοστρέφεια. Ας δούμε αργά αργά την υλοποίηση.

Αρχικά όπως θα κάναμε σε οποιαδήποτε περίπτωση εξάρτησης του κώδικα μας από κάποια εξωτερική βιβλιοθήκη έτσι και εδώ απαιτείται η ανάλυση του **usplash.com API** για να κατανοήσουμε τον τρόπο με τον οποίο θα εξάγουμε τη χρήσιμη για εμάς πληροφορία. Διαβάζοντας βρίσκουμε το endpoint που πρέπει να «χτυπήσουμε προγραμματιστικά» για την αποκόμιση των δεδομένων. Ο όρος «χτυπάμε προγραμματιστικά» σημαίνει η αποστολή κάποιων δεδομένων (υπό μορφή ερωτήματος) διά μέσου ενός συγκεκριμένου πρωτοκόλλου σε μία υπηρεσία που αναμένει τέτοιου είδους ερωτήματα. Η υπηρεσία, ανάλογα με τα δικαιώματα που διέπουν ένα συγκεκριμένο ερώτημα, αποφασίζει για το εάν πρέπει και τι πρέπει να προσφέρει σαν απάντηση στην οντότητα που δημιούργησε το ερώτημα. Αν όλα πάνε καλά η απάντηση δίδεται πίσω δια μέσω ενός ίδιου η παρόμοιου πρωτοκόλλου και περιέχει τις πληροφορίες που ζητήθηκαν στο αρχικό ερώτημα. Να πούμε περιληπτικά ότι τα ερωτήματα στέλνονται με τις μεθόδους του **HTTP** πρωτοκόλλου **GET** ή **POST** , οι απαντήσεις των ερωτημάτων έρχονται με τη μορφή που ορίζει το πρωτόκολλο **JSON**, και το πρωτόκολλο που ορίζει την εξουσιοδότηση είναι το **OAUTH (2.0** πλέον). Αρά λοιπόν μέχρι τώρα γνωρίζουμε ότι πρέπει αρχικά να φτιάξουμε ένα ερώτημα, να το αποστείλουμε, να περιμένουμε την απάντηση, και όταν αυτή έρθει θα πρέπει να πάρουμε τα δεδομένα που μας ενδιαφέρουν (σύνδεσμοι), να φτιάξουμε του κόμβους των εικόνων και τελικά να τους τοποθετήσουμε στο DOM της σελίδας. Αν παρατηρήσουμε τα ζητούμενα, θα δούμε ότι στο συγκεκριμένο πρόβλημα υπεισέρχεται και ο χρονικός παράγοντας. Η JavaScript ως γνωστόν είναι μία μονονηματική γλώσσα δομημένη με κατεύθυνση τον προγραμματισμό συμβάντων (event driver programming). Αυτό πρακτικά σημαίνει ότι οι ασύγχρονες δομές αντιμετωπίζονται εγγενώς από τη γλώσσα με τη βοήθεια των συμβάντων. Στην προκειμένη περίπτωση η πράξη της αποστολής του ερωτήματος, η αναμονή της απάντησης, και η ενέργεια που θα ακολουθηθεί, σκεπάζονται αφαιρετικά από τη δομή των **promises**. Θα δούμε στη συνέχεια πως θα υλοποιηθεί ακριβώς η δομή των promises. Αρχικά πάμε να δούμε τη δομή της απάντησης που θα μας αποστείλει το **API** στέλνοντας ένα ερώτημα μέσω του περιηγητή

και όχι προγραμματιστικά. Παρατηρούμε ότι πληκτρολογώντας στο URL του περιηγητή το ερώτημα

επιστρέφει ως απάντηση η παρακάτω μορφή JSON απεικόνισης


```
{
    total: 352,
    total_pages: 36,
  - results: [
      - {
            id: "TWBkfxTQin8",
            created_at: "2018-05-03T21:18:14-04:00",
            updated_at: "2018-08-28T21:01:50-04:00",
            width: 4162,
            height: 2379,
            color: "#2B3443",
            description: "Israel transportation plane",
          - urls: {
                raw: "https://images.unsplash.com/photo-1525396524423-64f7b55f5b33?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjUwNjA0fQ",
                full: "https://images.unsplash.com/photo-1525396524423-64f7b55f5b33?ixlib=rb-1.2.1&q=85&fm=jpg&crop=entropy&cs=srgb&ixid=eyJhcHBfaWQiOjUwNjA0fQ",
                regular: "https://images.unsplash.com/photo-1525396524423-64f7b55f5b33?ixlib=rb-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=1080&fit=max&ixid=eyJhcHBfaWQiOjUwNjA0fQ",
                small: "https://images.unsplash.com/photo-1525396524423-64f7b55f5b33?ixlib=rb-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=400&fit=max&ixid=eyJhcHBfaWQiOjUwNjA0fQ",
                thumb: "https://images.unsplash.com/photo-1525396524423-64f7b55f5b33?ixlib=rb-1.2.1&q=80&fm=jpg&crop=entropy&cs=tinysrgb&w=200&fit=max&ixid=eyJhcHBfaWQiOjUwNjA0fQ"
            },
          + links: {…},
            categories: [ ],
            sponsored: false,
            sponsored_by: null,
            sponsored_impressions_id: null,
            likes: 12,
            liked_by_user: false,
            current_user_collections: [ ],
          + user: {…},
          + tags: […],
          + photo_tags: […]
        },
      + {…},
      + {…},
      + {…},
      + {…},
      + {…},
      + {…},
      + {…},
      + {…},
      + {…}
    ]
}
```


Τα δεδομένα που θα χρειαστούμε έρχονται λοιπόν υπό τη μορφή ενός αντικειμένου, έστω x, και περιέχονται στη διαδρομή **x.results[0-10].urls.small**, όπου **x.results** λίστα με αντικείμενα και **urls.small** ο τελικός σύνδεσμος που θα εξαχθεί από τα επιμέρους αντικείμενα. Πάμε αρχικά να γράψουμε τη συνάρτηση που θα δημιουργεί και θα επιστρέφει το ερώτημα αναζήτησης. Ως ορίσματα θα δέχεται τον όρο με βάση των οποίο επιστρέφει τα σχετικά αποτελέσματα (πχ «aeroplane» για εικόνες με αεροπλάνα) και το κλειδί αυθεντικοποίησης clientId (το απαιτεί το πρωτόκολλο OAUTH2). Έχουμε

```
// apiURL :: String -> String -> String
const apiURL = clientId => queryTerm =>
`https://api.unsplash.com/search/photos?page=1&query=${queryTerm}&clie
nt_id=${clientId}`;
```

Επίσης γράφουμε τις βοηθητικές συναρτήσεις της σύνθεσης **composeN**, απεικόνισης **map** και εξαγωγής χαρακτηριστικό από ένα αντικείμενο **pluck**

```
// composeN :: (y -> z, x -> y, ..., b -> c, a -> b) -> a -> z
const composeN = (f, ...fns) => {
    const composeTwo = (g, f) => x => g(f(x));
    return f === undefined ? x => x : composeTwo(f, composeN(...fns));
}

// map :: (a -> b) -> [a] -> [b]
const map = f => arr => Array.prototype.map.call(arr, f);

// pluck :: String -> a -> b
const pluck = attr => obj => obj[attr];
```

Τώρα που έχουμε στην κατοχή μας τις βοηθητικές συναρτήσεις, θα ξεκινήσουμε την υλοποίηση του προγράμματος, υποθέτοντας ότι η έχουμε στη διάθεση μας την απάντηση από την υπηρεσία του domain **usplash.com** . Με λίγα λόγια θα ξεκινήσουμε από το δεύτερο σκέλος του προγράμματος και έπειτα θα ασχοληθούμε με το πρόβλημα της ασύγχρονης χρονικά δομής (μέσω των promises). Διασπώντας λοιπόν το δεύτερο σκέλος του προγράμματος, παρατηρούμε ότι πρέπει να συντάξουμε μία λειτουργικότητα η οποία θα δέχεται το σύνδεσμο με τα δεδομένα της εικόνας και θα δημιουργεί έναν κόμβο εικόνας (image element). Η σύνταξη φαίνεται παρακάτω

```
// createImage :: String -> ImageElement
const createImage = src => {
    const img = document.createElement("img");
    img.setAttribute("src", src);
    return img;
};
```

Στη συνέχεια θέλουμε μία λειτουργικότητα που θα δέχεται σαν είσοδο τέτοιους κόμβους και θα τους τοποθετεί στο DOM της σελίδας. Η ενέργεια αυτή σαφώς περιέχει παρενέργειες (side effects) αφού ξεκάθαρα αλλάζει την κατάσταση του DOM και συνεπώς αλλάζει την κατάσταση του προγράμματος. Όπως είπαμε όλα τα προγράμματα αργά η γρήγορα θα τροποποιήσουν μία κατάσταση έτσι ώστε να υπάρξει αντιληπτό από τον χρήστη αποτέλεσμα. Πώς είναι δυνατόν να αλλάξουμε το εικονιζόμενο αποτέλεσμα ενός περιηγητή εάν δεν μπορούμε να τροποποιήσουμε το DOM; Ο συναρτησιακός προγραμματισμός όμως προσφέρει σαφή διαχωρισμό ανάμεσα στις αγνές και τις μη αγνές δομές. Επίσης ο τρόπος γραφής μη αγνών δομών, δομών δηλαδή που μεταβάλλουν κάποια κατάσταση, με τη μορφή συναρτήσεων που εκτελούνται συνήθως στο τέλος της

αλγοριθμικής ροής ενός προγράμματος βοηθάνε τον προγραμματιστή να αιτιολογήσει καλύτερα τον κώδικα του. Ας δούμε τη συνάρτηση που τοποθετεί του κόμβους εικόνων πάνω στο DOM.

```
// hookTo(Impure) :: String -> Element -> ()
const hookTo = id => el =>
document.querySelector(`#${id}`).appendChild(el);
```

Δέχεται ως είσοδο το id του στοιχείου πάνω μέσα στο οποίο θα τοποθετηθούν οι εικόνες και επιστρέφει μία νέα συνάρτηση που αναμένει τα εικονοστοιχεία. Παρατηρούμε ότι η συνάρτηση **hookTo** και **createImage** μπορούν να συντεθούν σε μία νέα συνάρτηση που θα έχει την παρακάτω μορφή

```
// createImageAndHookToContainer :: String -> ()
const createImageAndHookToContainer = composeN(hookTo("container"),
createImage);
```

Η νέα αυτή συνάρτηση αναμένει έναν σύνδεσμο μορφής **String** και υλοποιεί την τοποθέτηση μίας εικόνας με το συγκεκριμένο σύνδεσμο πάνω στο στοιχείο του DOM που έχει **id="container".** Το συγκεκριμένο id προκύπτει από το σκελετό της σελίδας μέσα στον οποίο θα εκτελεστεί το παρόν πρόγραμμα. Ο σκελετός έχει τη μορφή που φαίνεται παρακάτω

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title>Usplash Photo Gen</title>
    <meta name="viewport" content="width=device-width, initial-
scale=1">
    <link rel="stylesheet" type="text/css" media="screen"
href="app.css" />
</head>
<body>
    <h1>Usplash Images</h1>
    <div id="container"></div>
    <script src="app.js"></script>
</body>
</html>
```

Αφού δημιουργήσαμε τη συνάρτηση τοποθέτησης, πάμε να συντάξουμε τη λειτουργικότητα εξαγωγής του ζητούμενων δεδομένων (των συνδέσμων δηλαδή) από το αντικείμενο JSON που θα έρθει ως απάντηση. Προφανώς η λειτουργικότητα αυτή θα προηγείται από αυτή της τοποθέτησης των εικόνων στο DOM. Αρχικά πρέπει να εξάγουμε το αντικείμενο **results** που περιέχει το αρχικό JSON

```
// getResults :: ResultObject -> [UrlsObject]
const getResults = pluck("results");
```

Στη συνέχεια, και αφού γνωρίζουμε ότι το αποτέλεσμα της επιστροφής της **getResults** είναι ένας πίνακας, δημιουργούμε μία συνάρτηση που επιστρέφει το σύνδεσμο από ένα στοιχείο του πίνακα αυτού, όπως φαίνεται παρακάτω

```
// getImageSrc :: UrlsObject -> Link(String)
const getImageSrc = composeN(pluck("small"), pluck("urls"));
```

και εισάγουμε τη συνάρτηση αυτή **getImageSrc** στη συνάρτηση υψηλής τάξης **map,** για να ενεργοποιηθεί η δυνατότητα μαζικής προσπέλασης των στοιχείων ενός πίνακα με βάση τη συγκεκριμένη λειτουργικότητα. Πρακτικά το αποτέλεσμα της **getResults** (λίστα) θα εισαχθεί στην **map**, και η **map** θα εφαρμόσει τη συνάρτηση **getImageSrc** σε όλα τα στοιχεία αυτής της λίστας. Στη συνέχεια μπορούμε να επεκτείνουμε τη λειτουργικότητα εφαρμόζοντας τελικά σε όλα τα μετασχηματισμένα στοιχεία τη συνάρτηση **createImageAndHookToContainer** όπως βλέπουμε παρακάτω

```
// createImagesAndHookToContainer :: ResultObject -> [()]
const createImagesAndHookToContainer =
composeN(map(createImageAndHookToContainer), map(getImageSrc),
getResults);
```

Επίσης μπορούμε να εκμεταλλευτούμε τη συνθετική ιδιότητα της συνάρτησης υψηλής τάξης **map** και η συνάρτηση **createImagesAndHookToContainer** να πάρει τη μορφή

```
// createImagesAndHookToContainer :: ResultObject -> [()]
const createImagesAndHookToContainer =
composeN(map(composeN(createImageAndHookToContainer, getImageSrc)),
getResults);
```

Αποφεύγοντας έτσι την άσκοπη διπλή προσπέλαση των στοιχείων της λίστας που επιστρέφει η **getResults**. Έχουμε ολοκληρώσει τις λειτουργίες που θα εκτελεστούν από τη στιγμή που θα φτάσει η απάντηση από το **usplash api endpoint** (JSON Object) και απομένει η δημιουργία και αποστολή του ερωτήματος. Όπως είπαμε μία τέτοια λειτουργία

επειδή είναι χρονοβόρα, χρειάζεται να υλοποιηθεί σε ασύγχρονο χρόνο χρησιμοποιώντας την εγγενή στην JavaScript δομή των **promises**. Ένα **promise** επιτρέπει στον προγραμματιστή να χτίσει έναν υπολογισμό με βάση μία μελλοντική τιμή (future value), αλυσιδωτά. Πρακτικά επιτρέπει την ευπαρουσίαστη αναπαράσταση των ασύγχρονων δομών ξεπερνώντας τα εμπόδια που γνώριζε το μοντέλο των callback functions (callback hell, inversion of control) (Simpson, 2015). Επίσης αξίζει να αναφέρουμε ότι τα **promises** αν και δεν ικανοποιούν όλες τις προϋποθέσεις που ορίζουν οι μονάδες (**monads**) από τη θεωρία κατηγοριών, συμπεριφέρονται ως τέτοιες στην πλειοψηφία των περιπτώσεων (θα δούμε αργότερα τι ορίζουμε ως μονάδα και τα πλεονεκτήματα της). Στη συγκεκριμένη περίπτωση για την αποστολή του ερωτήματος θα κάνουμε χρήση της συνάρτησης **fetch** που δέχεται τα στοιχεία του ερωτήματος και επιστρέφει ένα **promise** με μελλοντική τιμή την απάντηση που θα μας στείλει το **api endpoint**. Έχοντας το promise στη διάθεση μας μπορούμε να αιτιολογήσουμε (reason) το πρόγραμμα μας κανονικά υποθέτοντας ότι έχουμε ήδη στην κατοχή μας την τιμή που θα επιστρέψει οποιαδήποτε ασύγχρονη κλήση. Αυτό είναι και το βασικό προτέρημα που προφέρουν τα promises. Αφαιρούν τον χρονικό παράγοντα που εισάγουν οι χρονοβόρες (ασύγχρονες) διαδικασίες. Πάμε να δούμε το τελευταίο μέρος της λύσης. Γράφουμε μία συνάρτηση της μορφής

```
// getImages(Impure) = (ResultObject -> [()]) -> String -> Promise
const getImages = f => url => fetch(url).then(res =>
res.json()).then(f);
```

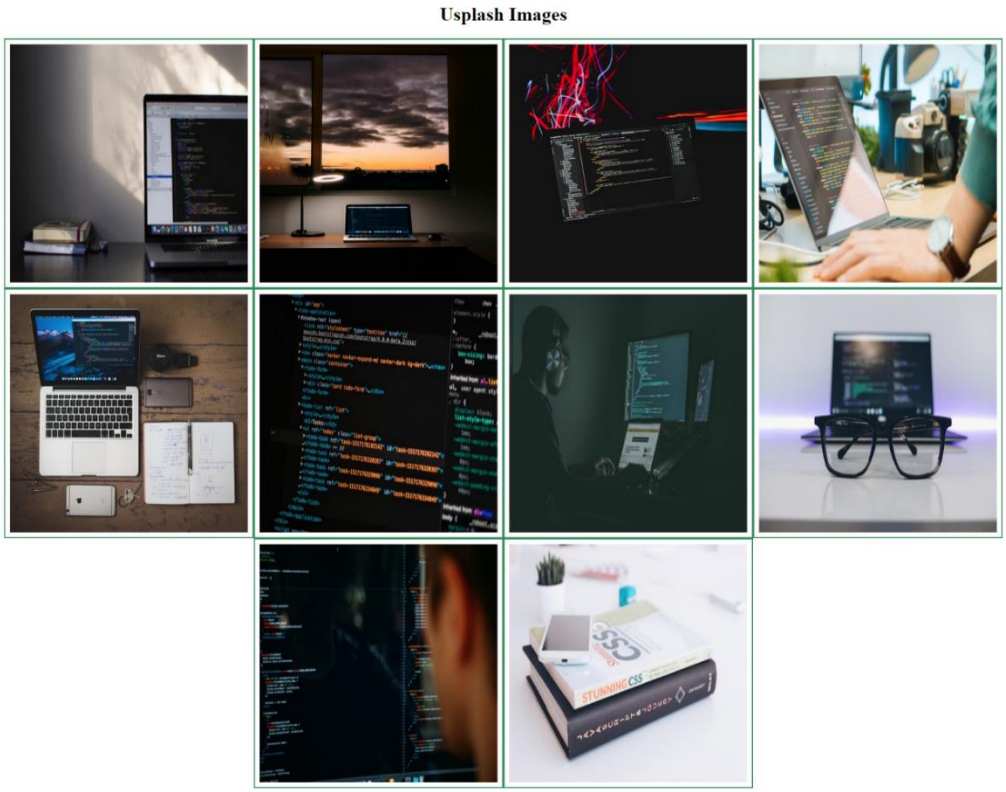
Η συνάρτηση **getImages** δέχεται μία συνάρτηση f, που θα δέχεται το αποτέλεσμα της απάντησης (JSON Object) και θα υλοποιεί ότι είδαμε παραπάνω, δηλαδή τη λειτουργικότητα **createImagesAndHookToContainer,** και ως δεύτερη παράμετρο το URL του ερωτήματος και θα επιστρέφει ένα promise το οποίο θα εκτελέσει τη συνάρτηση **createImagesAndHookToContainer** με παράμετρο την απάντηση από το api endpoint. Τελικά συνθέτουμε τη συνάρτηση δημιουργίας του URL **apiURL** με την **getImages** και έχουμε

```
const CLIENTID =
"bf1e80fabb597dadac4f35cb19e9acb2f1cbb1189bdbcabcfbc0d7f24ea710f3";

// apiURL :: String -> String -> String
const apiURL = clientId => queryTerm =>
`https://api.unsplash.com/search/photos?page=1&query=${queryTerm}&clie
nt_id=${clientId}`;

// app :: String -> Promise
const app = composeN(getImages(createImagesAndHookToContainer),
apiURL(CLIENTID));
```

Και το πρόγραμμα είναι έτοιμο. Το αλφαριθμητικό **CLIENTID** χρειάζεται για λόγους αυθεντικοποίησης της εφαρμογής στην υπηρεσία που προσφέρει το **usplash.com .** Η συνάρτηση **app** περιμένει ως παράμετρο έναν αγγλικό όρο, και επιστρέφει ένα DOM πάνω στο οποίο είναι τοποθετημένοι κόμβοι εικόνων σχετικές με τον όρο που εισήχθη. Για παράδειγμα η έκφραση

```
// app :: String -> Promise
app("programming");
```

δημιουργεί την παρακάτω σελίδα



Παρατηρούμε ότι η συναρτησιακή προσέγγιση διαφέρει αρκετά από το κλασσικό μοντέλο προστακτικής γραφής στο οποίο είναι εξοικειωμένη η μεγάλη πλειοψηφία των προγραμματιστών. Στον παρόν παράδειγμα, αναλύσαμε το πρόβλημα σε υποπροβλήματα για καθένα από τα οποία προσφέραμε αυστηρά καθορισμένη και ευέλικτη λύση υπό τη μορφή μίας συνάρτησης. Λαμβάνοντας υπόψη της εισόδους και εξόδους αυτών, προχωρήσαμε στη σύνθεση τους με τρόπο τέτοιο ώστε η λύση να μπορεί να εκφραστεί δηλωτικά, όπως δηλαδή ενθαρρύνει ο συναρτησιακός προγραμματισμός. Ταυτόχρονα όλες οι παράμετροι που μπορούν να δεχτούν οι συναρτήσεις μας, εκφράζουν το βαθμό ελευθερίας και αφαίρεσης του προγράμματος με την έννοια ότι η λειτουργικότητα είναι εύκολα τροποποιήσιμη. Για παράδειγμα το κλειδί αυθεντικοποίησης **CLIENTID**, ορίζεται κατόπιν δημιουργίας του **URL** και το γεγονός ότι δηλώνεται ως παράμετρος στη συνάρτησης **apiURL** σημαίνει ότι μπορεί εύκολα να αλλάξει σε περίπτωση που θέλουμε να χρησιμοποιήσουμε κάποιο άλλο κλειδί. Φυσικά αυτό δεν είναι κάτι καινούργιο στον προγραμματισμό στον οποίο οι μεταβλητές έχουν ακριβώς αυτό το σκοπό. Η διαφορά είναι ότι στη συναρτησιακή προσέγγιση, οι μεταβλητές μας πλέον δεν αποθηκεύουν μόνο πρωτόγονες τιμές, αλλά ολόκληρες αυτοτελής λειτουργικότητες (πολλές φορές με καθορισμένο περιβάλλον, **closure**), (Simpson, You Don't Know JS_ Scope & Closures, 2014) με αποτέλεσμα να έχουμε τη δυνατότητα αποδοτικότερης ευελιξίας. Αν για παράδειγμα αντί να επέμβουμε στο DOM μίας σελίδας, θα θέλαμε να γράψουμε τις εικόνες σε αρχεία, αρκεί να αντικαταστήσουμε τη συνάρτηση **createImageAndHookToContainer** με την αντίστοιχη νέα λειτουργικότητα.

Για να κλείσουμε το κεφάλαιο της σύνθεσης και αφού μιλάμε για τη σχετική ευκολία που εισάγει η συναρτησιακή ανάλυση όσον αφορά την επεκτασιμότητα, ας το δούμε και στην πράξη. Παρατηρούμε ότι το **URL** του ερωτήματος που κατασκευάζεται για να αποσταλεί στην υπηρεσία **api** του **usplash.com**, μπορεί να δεχτεί ως παράμετρο και τη σελίδα από την οποία θέλουμε να έρθουν τα αποτελέσματα των εικόνων. Ο προκαθορισμένος αριθμός είναι ο 1, φέρνοντας έτσι όλες τις εικόνες που χωράνε σε μία σελίδα με μέγιστο τις 10/σελίδα. Έστω λοιπόν ότι θέλουμε να τροποποιήσουμε το πρόγραμμα μας και πλέον, κατά την κλήση της τελικής συνάρτησης να εισάγουμε εκτός από τον όρο αναζήτησης, και τις σελίδες από τις οποίες θέλουμε να έρθουν τα αποτελέσματα. Έτσι αν καλέσουμε την τελική συνάρτηση με παραμέτρους έναν πίνακα με

τις σελίδες που θέλουμε, και τον όρο αναζήτησης, το αποτέλεσμα θα είναι ένα DOM το οποίο περιέχει όλες τις εικόνες των σελίδων που θέλουμε.

Ας δούμε τι θα αλλάζαμε στον κώδικα. Αρχικά προσθέτουμε μία παράμετρο στο ερώτημα αναζήτησης, το οποίο θα αντικατοπτρίζει τον αριθμό της σελίδας όπως φαίνεται παρακάτω

```
// apiURL :: String -> Number -> String -> String
const apiURL = clientId => page => queryTerm =>
`https://api.unsplash.com/search/photos?page=${page}&query=${queryTerm
}&client_id=${clientId}`;
```

Στη συνέχεια, αυτό που πρέπει να γίνει ουσιαστικά είναι ο μετασχηματισμός της συνάρτησης **app**, σε μία συνάρτηση η οποία θα ενεργεί σε πίνακες για κάθε τιμή της λίστας των σελίδων. Έτσι αν η app είχε τη μορφή

```
// app :: String -> Promise
const app = composeN(getImages(createImagesAndHookToContainer),
apiURL(CLIENTID));
```

η νέα της μορφή θα είναι περίπου αυτή

```
const app = composeN(map(getImages(createImagesAndHookToContainer)),
map(apiURL(CLIENTID)));
```

Αυτό που απομένει είναι η σωστή χρήση των παραμέτρων στην τελική συνάρτηση. Η **map(apiURL(CLIENTID))** περιμένει ως είσοδο έναν πίνακα με τις σελίδες προς αναζήτηση, οπότε η **app** θα πάρει τη μορφή

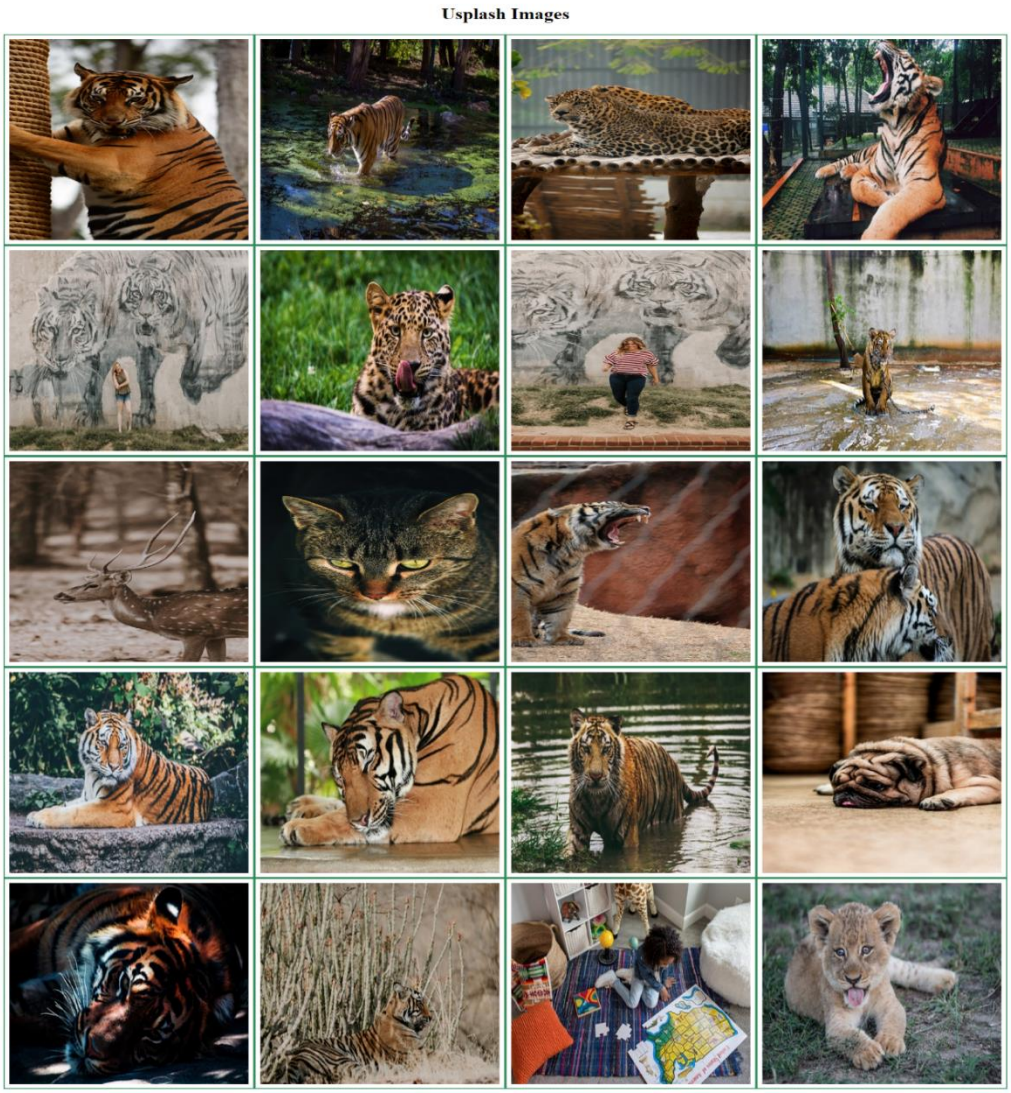
```
const app = pages =>
composeN(map(getImages(createImagesAndHookToContainer)),
map(apiURL(CLIENTID)))(pages);
```

Αυτή τη στιγμή το δεξί τμήμα της σύνθεσης επιστρέφει έναν πίνακα με το αποτέλεσμα της κλήσης **apiURL(clientId)(page)** το οποίο είναι συναρτήσεις έτοιμες να δεχτούν τον όρο αναζήτησης και να επιστρέψουν το τελικό αλφαριθμητικό που θα χρησιμοποιηθεί στο get ερώτημα. Συνεπώς τελικώς μπορούμε να γράψουμε

```
// app :: [Number] -> String -> [Promise]
const app = pages => term => composeN(map(getImages(createImage-
sAndHookToContainer)), map(f => f(term)), map(apiURL(CLIEN-
TID)))(pages);
```

Έτσι μπορούμε να τρέξουμε το νέο μας πρόγραμμα όπως απεικονίζεται στη συνέχεια

Το αποτέλεσμα είναι η δημιουργία 2 ερωτημάτων στο **API endpoint** του **usplash.com,** ένα για κάθε σελίδα, και αν οι σελίδες είναι γεμάτες, το πλήθος των εικόνων είναι το πλήθος των σελίδων επί δέκα (10), δηλαδή είκοσι (20) εικόνες στο παρόν παράδειγμα.

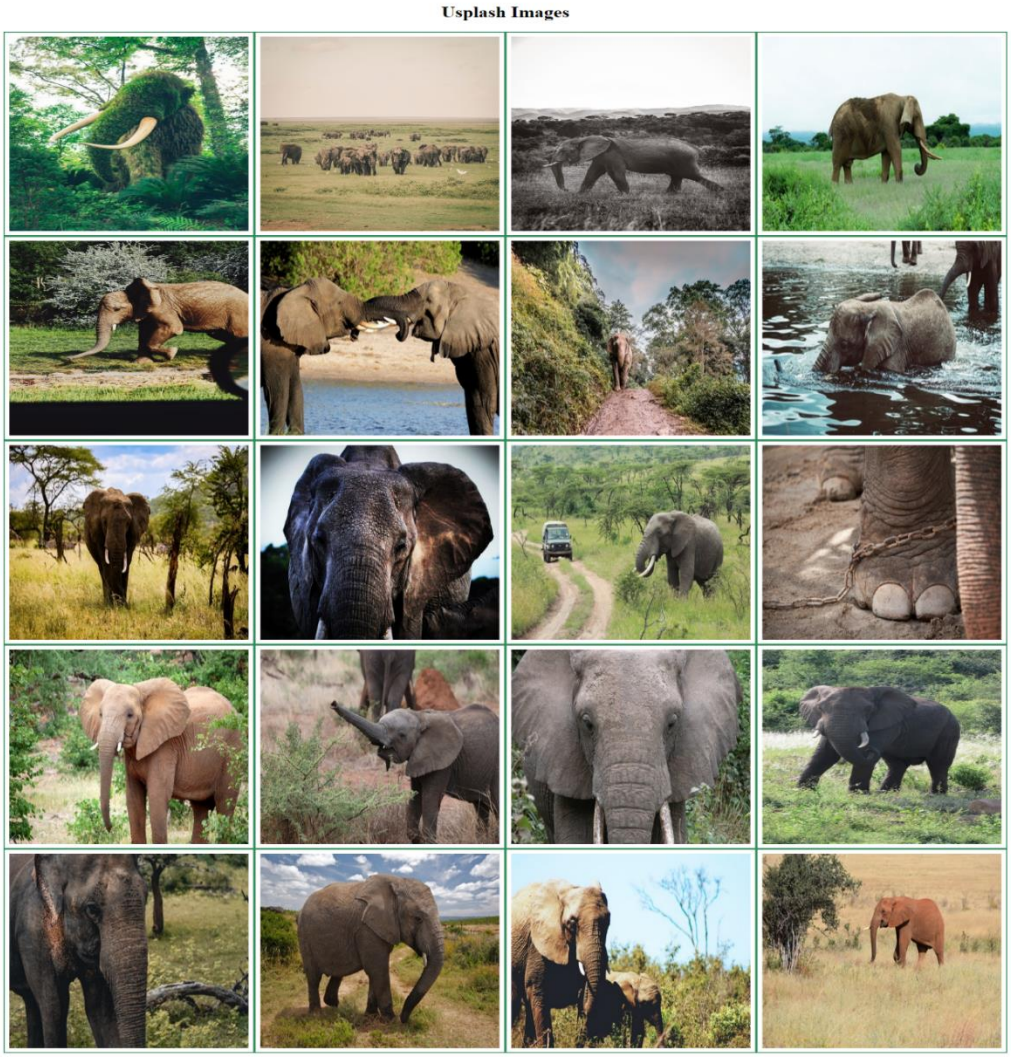


Τέλος εάν θέλουμε να βελτιστοποιήσουμε των κώδικα μας μπορούμε να γράψουμε για την app

```
// app :: [Number] -> String -> [Promise]
const app = pages => term =>
map(composeN(getImages(createImagesAndHookToContainer), f => f(term),
apiURL(CLIENTID)))(pages);
```

Μετατρέποντας τη σύνθεση των συναρτήσεων **map** σε **map** της σύνθεσης γλιτώνοντας έτσι το υπολογιστικό κόστος από τις περιττές προσπελάσεις των πινάκων. Έτσι

```
// appForPages :: String -> [Promise]
const appForPages = app([1, 2]);

appForPages("elephant");
```



4.8 Σωλήνωση Συναρτήσεων (Function Pipelining)

Η σωλήνωση συναρτήσεων ή **function pipelining** όπως είναι γνωστός ο όρος στην αγγλική ορολογία, είναι η προγραμματιστική τεχνική που υλοποιεί την πράξη της σύνθεσης διπλώνοντας τις λειτουργικότητες με αντίθετη σειρά από αυτή της σύνθεσης. Σε αντίθεση με την πράξη της σύνθεσης η σωλήνωση δεν έχει κάποιο αντίστοιχο μαθηματικό τελεστή, όμως η μαθηματική θεωρία που διέπει τη συνθετική πράξη ισχύει και για τη σωλήνωση. Πρακτικά όπως είπαμε η σωλήνωση επιτελεί την πράξη της σύνθεσης απλά με ανάποδη σειρά. Στην επιστήμη των υπολογιστών η σωλήνωση συμβολίζεται συνήθως με τον τελεστή |, που είναι γνωστός από τα λειτουργικά συστήματα των UNIX και επιτρέπει τη μετάδοση της εξόδου μίας διεργασίας στην είσοδο μίας άλλης όπως φαίνεται στο παρακάτω παράδειγμα

```
felix@imbalance:/mnt/c/Users/fsafa$ cat some.scala | wc -m
103
```

Σχήμα 4-8 Παράδειγμα εντολής σωλήνωση σε περιβάλλον Unix

Στο παραπάνω παράδειγμα η διεργασία **cat** που διαβάζει τα περιεχόμενα του αρχείου **some.scala** και τα τροφοδοτεί μέσω της σωλήνωσης στη διεργασίας **wc** η οποία παίρνει το περιεχόμενο του αρχείου και επιστρέφει το πλήθος των χαρακτήρων του αρχείου (με την επιλογή -m), δηλαδή τον αριθμό 104 . Αν θα θέλαμε να απεικονίσουμε μαθηματικά την τεχνική της σωλήνωσης τότε θα λέγαμε ότι έστω

$$f(x) = y, \quad x \in X, y \in Y \quad \text{ή} \quad f: X \to Y$$

και

$$g(y) = z, \quad y \in Y, z \in Z \quad \text{ή} \quad g: Y \to Z$$

Ως σωλήνωση ορίζεται η σύνθετη συνάρτηση για τη οποία έχουμε

$$(f \mid g)(x) = z, \qquad x \in X, z \in Z \ \text{ή} \ f \mid g : X \to Z$$

ή

$$(f \mid g)(x) = (g \circ f)(x) = z, \qquad x \in X, z \in Z \ \text{ή} \ f \mid g, g \circ f \ : X \to Z$$

Βλέπουμε λοιπόν ότι πρακτικά ισχύει

$$f_1 \mid f_2 \mid f_3 \mid \ldots \mid f_n = f_n \circ \ldots \circ f_3 \circ f_2 \circ f_1$$

Όσον αφορά στην προγραμματιστική διάσταση του θέματος στα πλαίσια του συναρτησιακού προγραμματισμού η σωλήνωση προσφέρει μία πιο συνηθισμένη αντίληψη της ροής των δεδομένων, αφού αυτά πλέον ρέουν από αριστερά προς τα δεξιά. Η σειρά αυτή μας είναι αρκετά πιο γνώριμη, αφού χρησιμοποιείται από τα συστήματα γραφής της πλειονότητας των γλωσσών καθώς και της ερμηνείας της πλειοψηφίας των μαθηματικών παραστάσεων. Πέρα από αυτό το διαχωρισμό αυτό η πράξη της σωλήνωσης είναι ακριβώς ίδια με αυτή της σύνθεσης και ασφαλώς οτιδήποτε υλοποιείται με την πρώτη μπορεί να υλοποιηθεί και με τη δεύτερη με αλλαγή της σειράς των παραμέτρων όπως υποδεικνύει η σχέση $(f \mid g)(x) = (g \circ f)(x) = z$. Ας δούμε όμως πως υλοποιείται η πράξη της σωλήνωσης στη γλώσσα της JavaScript

```
// pipeTwo :: (a -> b, b -> c) -> a -> c
const pipeTwo = (f, g) => x => g(f(x));
```

Παρατηρούμε ότι η πρώτη παράμετρος της συνάρτησης **pipeTwo,** δηλαδή η συνάρτηση **f**, εκτελείται πρώτη και τροφοδοτεί την έξοδο της ως είσοδο στη δεύτερη παράμετρο δηλαδή τη συνάρτηση **g.** Η σωλήνωση συναρτήσεων ακολουθεί τους ίδιους νόμους που ισχύουν και για τη σύνθεση συναρτήσεων. Συγκεκριμένα αναφερόμαστε στην προσαιτεριστική ιδιότητα. Αν θέλουμε να απεικονίσουμε το πρόβλημα στη μαθηματική του διάσταση θα λέγαμε ότι αν ορίσουμε τις τρείς συναρτήσεις f, g, h για τις οποίες ισχύουν οι παρακάτω ορισμοί, τότε ισχύει και η τελική σχέση.

Αν δηλαδή έχουμε

$$f(x) = y, \quad x \in X, y \in Y \quad \text{ή} \quad f: X \to Y$$

και

$$g(y) = z, \quad y \in Y, z \in Z \quad \text{ή} \quad g: Y \to Z$$

και

$$h(z) = w, \quad z \in Z, w \in W \quad \text{ή} \quad h: Z \to W$$

Τότε ισχύει ότι

$$f \mid g \mid h = (f \mid g) \mid h = f \mid (g \mid h) \ : X \to W$$

Η απόδειξη ανάγεται στην πράξη της σύνθεσης αφού ισχύει $f \mid g = g \circ f$ και ως εκ τούτου είναι περιττή. Τοποθετώντας τις παραπάνω σχέσεις προγραμματιστικά θα έχουμε

```
// pipeTwo :: (a -> b, b -> c) -> a -> c
const pipeTwo = (f, g) => x => g(f(x));

// pipeThree :: (a -> b, b -> c, c -> d) -> a -> d
const pipeThree = (f, g, h) => pipeTwo(f, pipeTwo(g, h));
```

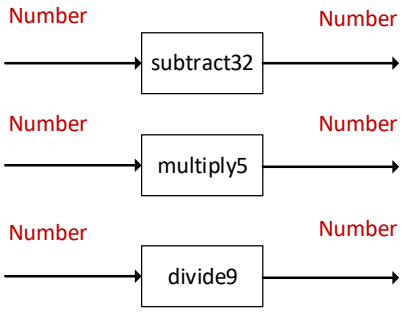
ή

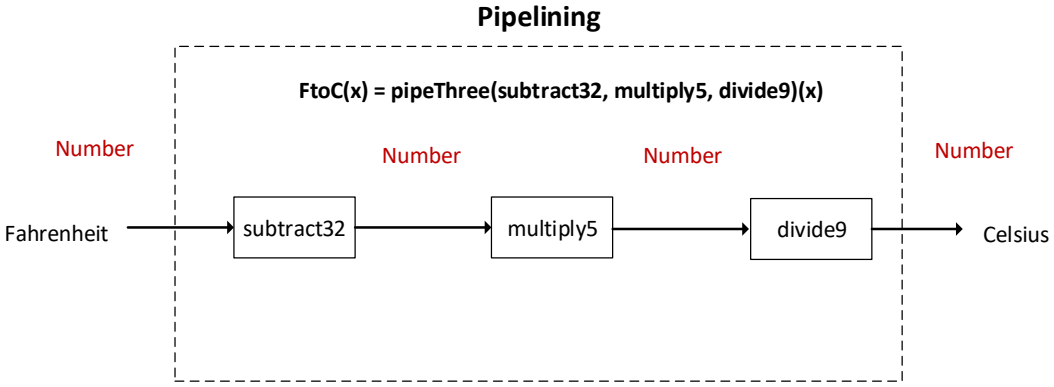
```
// pipeTwo :: (a -> b, b -> c) -> a -> c
const pipeTwo = (f, g) => x => g(f(x));

// pipeThree :: (a -> b, b -> c, c -> d) -> a -> d
const pipeThree = (f, g, h) => pipeTwo(pipeTwo(f, g), h);
```

Έτσι αν θα θέλαμε να αναπαραστήσουμε το πρόβλημα της μετατροπής θερμοκρασίας από βαθμούς Fahrenheit σε Celsius (το πρόβλημα που είδαμε στο υποκεφάλαιο της σύνθεσης δηλαδή) υπό το πρίσμα της σωλήνωσης, θα είχαμε

```
// subtract :: Number -> Number -> Number
const subtract = binaryOp("-");

// multiply :: Number -> Number -> Number
const multiply = binaryOp("*");

// divide :: Number -> Number -> Number
const divide = binaryOp("/");

// subtract32 :: Number -> Number
const subtract32 = subtract(32);

// multiply5 :: Number -> Number
const multiply5 = multiply(5);

// divide9 :: Number -> Number
const divide9 = divide(9);

// pipeTwo :: (a -> b, b -> c) -> a -> c
const pipeTwo = (f, g) => x => g(f(x));

// pipeThree :: (a -> b, b -> c, c -> d) -> a -> d
const pipeThree = (f, g, h) => pipeTwo(pipeTwo(f, g), h);

// FtoC :: Number -> Number
const FtoC = pipeThree(subtract32, multiply5, divide9);

FtoC(32);
//-> 0
```

Και το αντίστοιχο σχήμα υπό τη μορφή μπλοκ συστημάτων που υπόκεινται στην πράξη της σωλήνωσης θα απεικονίζονταν όπως φαίνεται παρακάτω





Παρατηρούμε ότι η σειρά εφαρμογής των παραμέτρων στη συνάρτηση **pipeThree** έρχεται σε πλήρη στοίχιση με τη σειρά προσπέλασης των αντίστοιχων μπλοκ συστημάτων τους ακολουθώντας την κατεύθυνση που υποδεικνύουν τα βέλη από αριστερά προς τα δεξία. Κατί τέτοιο δεν συνέβη με το αντίστοιχο διάγραμμα που είδαμε στην πράξη της σύνθεσης όπου η σειρά των παραμέτρων ήταν ανάποδη.

Τέλος πάμε να δούμε την υλοποίηση της σωλήνωσης **pipe** με έναν ακαθάριστο αριθμό συναρτήσεων εισόδου. Με λίγα λόγια αφού ορίσαμε τις **pipeTwo, pipeThree** ψάχνουμε τον γενικό ορισμό που ακολουθεί η πράξη της σύνθεσης για N πλήθος συναρτήσεων, δηλαδή **pipeN.** Παρατηρούμε από τον ορισμό της σωλήνωσης συναρτήσεων και από τις ιδιότητες τις οποίες αυτή ικανοποιεί (προσεταιρεστική) ότι μπορεί να γραφεί

$$a \mid b \mid \dots x \mid y \mid z = a \mid b \mid \dots x \mid (y \mid z) = a \mid b \mid \dots \big(x \mid (y \mid z)\big) = a \mid \Big(b \mid \dots \big(x \mid (y \mid z)\big)\Big)$$

$$= \Big(a \mid \Big(b \mid \dots \big(x \mid (y \mid z)\big)\Big)\Big)$$

ή

$$a \mid b \mid \ldots x \mid y \mid z = (a \mid b) \mid \ldots x \mid y \mid z = \big((a \mid b) \mid \ldots \mid x\big) \mid y \mid z$$
$$= \Big(\big((a \mid b) \mid \ldots \mid x\big) \mid y\Big) \mid z = \Big(\Big(\big((a \mid b) \mid \ldots \mid x\big) \mid y\Big) \mid z\Big)$$

Για την πρώτη έκφραση η οποία εφαρμόζεται από τα δεξιά προς τα αριστερά (right associative) η προγραμματιστική αναπαράσταση θα είχε την παρακάτω μορφή

```
// pipeTwo :: (a -> b, b -> c) -> a -> c
const pipeTwo = (f, g) => x => g(f(x));

// pipeThree :: (a -> b, b -> c, c -> d) -> a -> d
const pipeThree = (f, g, h) => pipeTwo(f, pipeTwo(g, h));

// pipeFour :: (a -> b, b -> c, c -> d, d -> e) -> a -> e
const pipeFour = (f, g, h, w) => pipeTwo(f, pipeTwo(g, pipeTwo(h,
w)));
```

Παρατηρούμε ότι διακρίνουμε μία επαναληπτική διαδικασία η οποία ως γνωστόν μπορεί να υλοποιηθεί αναδρομικά. Πράγματι μία εκδοχή της συνάρτησης σωλήνωσης αυθαιρέτου αριθμού συναρτήσεων απεικονίζεται παρακάτω

```
// pipeN :: (a -> b, b -> c, ... , x -> y, y -> z) -> a -> z
const pipeN = (f, ...fns) => f === undefined ? x => x : pipeTwo(f,
pipeN(...fns));
```

και πιο σωστά αν θα θέλαμε να κρύψουμε την **pipeTwo** από το εξωτερικό περιβάλλον θα γράφαμε

```
// pipeN :: (a -> b, b -> c, ... , x -> y, y -> z) -> a -> z
const pipeN = (f, ...fns) => {
    const pipeTwo = (f, g) => x => g(f(x));
    return f === undefined ? x => x : pipeTwo(f, pipeN(...fns));
}
```

Για τη δεύτερη έκφραση η οποία είναι εφαρμόζεται από τα αριστερά προς τα δεξιά (**left associative**) η προγραμματιστική αναπαράσταση θα είχε την αντίστοιχη μορφή που απεικονίζεται παρακάτω

```
// pipeTwo :: (a -> b, b -> c) -> a -> c
const pipeTwo = (f, g) => x => g(f(x));

// pipeThree :: (a -> b, b -> c, c -> d) -> a -> d
const pipeThree = (f, g, h) => pipeTwo(pipeTwo(f, g), h);

// pipeFour :: (a -> b, b -> c, c -> d, d -> e) -> a -> e
const pipeFour = (f, g, h, w) => pipeTwo(pipeTwo(pipeTwo(f, g), h),
w);
```

Πάλι διακρίνουμε μία αναδρομική διαδικασία της οποίας η υλοποίηση διαφέρει από αυτή που φαίνεται στον κώδικα της **pipeN**

```
// pipeN :: (a -> b, b -> c, ... , x -> y, y -> z) -> a -> z
const pipeN = (f, ...fns) => fns.length === 0 ? f : pipeN(pipeTwo(f,
fns[0]), ...fns.slice(1));
```

Και αποκρύπτοντας τη δήλωση της **pipeTwo** θα έχουμε

```
// pipeN :: (a -> b, b -> c, ... , x -> y, y -> z) -> a -> z
const pipeN = (f, ...fns) => {
    const pipeTwo = (f, g) => x => g(f(x));
    return fns.length === 0 ? f : pipeN(pipeTwo(f, fns[0]),
...fns.slice(1));
}
```

Οι δύο εκφράσεις που είδαμε για την **pipeN** είναι ισοδύναμες ως προς το αποτέλεσμα τους αλλά διαφέρουν στην υλοποίηση ακριβώς με τον ίδιο τρόπο που διαφέρουν οι υλοποιήσεις **composeN** τις οποίες γνωρίσαμε στο προηγούμενο υποκεφάλαιο

4.9 Τεχνική Currying και Μερική Εφαρμογή (Currying and Partial Application)

Αφού αναλύσαμε τη μαθηματική θεωρία της σύνθεσης και είδαμε τον τρόπο με τον οποίο αυτή υιοθετείται από το προγραμματιστικό μοντέλο του συναρτησιακού προγραμματισμού, πάμε να δούμε άλλες δύο στενά συνυφασμένες έννοιες που παρουσιάζουν επίσης μεγάλη βαρύτητα, στη συναρτησιακή προσέγγιση.

Η τεχνική currying, το όνομα της οποίας πηγάζει από τον λογικολόγο-μαθηματικό Haskell Curry (η ομώνυμη γλώσσα ονομάστηκε επίσης προς τιμήν του), έχει καθορισμένο μαθηματικό υπόβαθρο και αποτελεί λογικό παράγωγο του θεωρητικού μοντέλου που αναλύει ο λογισμός λάμδα. Το γεγονός αυτό καθώς και η άμεση σχέση της έννοιας με τις

μαθηματικές συναρτήσεις έχουν ως αποτέλεσμα την ευρεία χρήση της τεχνικής στον συναρτησιακό προγραμματισμό. Προτού μελετήσουμε όμως το πρόβλημα υπό την προγραμματιστική του σκοπιά, ας δούμε αρχικά τη μαθηματικά του ερμηνεία.

Έστω ότι έχουμε την αλγεβρική συνάρτηση

$$f(x,y) = x + y^2, \quad f: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

η οποία δέχεται ως είσοδο δύο πραγματικού αριθμούς και επιστρέφει στην έξοδο έναν πραγματικό αριθμό. Η τεχνική currying μεταφράζει τη συνάρτηση f σε μία άλλη συνάρτηση, έστω h, η οποία δέχεται στην είσοδο της έναν πραγματικό αριθμό και επιστρέφει για τον καθένα μία νέα συνάρτηση ορισμένη στο πεδίο $\mathbb{R} \to \mathbb{R}$. Η συνάρτηση h μαθηματικά ορίζεται

$$h = curry(f), \qquad h(x) = h_x = x + y^2, \qquad \forall x \in \mathbb{R} \ , \qquad h: \mathbb{R} \to \mathbb{R}^{\mathbb{R}}$$

ή

$$h(x)(y) = x + y^2$$

Έτσι για παράδειγμα η μαθηματική έκφραση $h(2)$ επιστρέφει μία νέα συνάρτηση όπως φαίνεται παρακάτω

$$h(2) = 2 + y^2 = g(y), \ \ g: \mathbb{R} \to \mathbb{R}$$

Η παραπάνω λογική ισχύει για συνάρτηση με ν πλήθος παραμέτρων (n-ary function). Η g είναι μία νέα συνάρτηση με πεδίο ορισμού το $\mathbb{R}$ και σύνολο τιμών επίσης το $\mathbb{R}$ που προκύπτει από εφαρμογή της curried συνάρτησης h στον πραγματικό αριθμό 2 (όρισμα). Αν θέλουμε να αναπαραστήσουμε το πρόβλημα με βάση το θεωρητικό μοντέλο του λογισμού λάμδα, το οποίο υιοθετεί ο συναρτησιακός προγραμματισμός, τότε θα γράφαμε την παρακάτω αφαίρεση λάμδα

$$h = curry(f) = \lambda x. \Big(\lambda y. \big(f(x,y)\big)\Big) = \lambda x. (\lambda y. (x + y^2)), \qquad \mu\varepsilon \ f: X \times Y \to Z$$

και επειδή η πράξη της εφαρμογής (application) στον λογισμό λάμδα έχει αριστερή προτεραιότητα (left associative) μπορεί να γραφεί

$$h = \lambda x. \lambda y. x + y^2$$

με ορισμό

$$h: \big(X \to (Y \to Z)\big)$$

και αφού ο τελεστής $\to$ έχει δεξιά προτεραιότητα (right associative) μπορούμε να γράψουμε

$$h: X \to Y \to Z$$

Η παραπάνω έκφραση λάμδα κατόπιν εφαρμογής της με μία παράμετρο έστω x_0 επιστρέφει μία νέα αφαίρεση λάμδα και ως εκ τούτου επιστρέφει μία νέα συνάρτηση. Όπως είδαμε στο αρχικό κεφάλαιο, εξ ορισμού η θεμελιώδης αρχή του λογισμού λάμδα είναι ο κύκλος που δημιουργεί η πράξη της δημιουργίας αφαιρέσεων (**abstraction**) και της εφαρμογής τους (**application**) μέσω της αποτίμησης των εκφράσεων. Θα λέγαμε λοιπόν ότι το γεγονός της σταδιακής αποτίμησης που ακολουθεί ο λογισμός λάμδα γεννά και την ιδιότητα **currying**. Έτσι ο τρόπος απεικόνισης μιας πολυπαραμετρικής αφαίρεσης λάμδα (συνάρτησης δηλαδή) είναι μοναδικός και μεταφράζεται απευθείας σε μία **curried** μορφή. Έτσι εφαρμόζοντας για παράδειγμα τον αριθμό 2 στην έκφραση h θα έχουμε

$$\mathrm{h}\ 2 = (\lambda x.\lambda y.x + y^2)\ \ 2 = \lambda y.2 + y^2 = g$$

Η συνάρτηση g αποτελεί μία νέα αφαίρεση λάμδα έτοιμη να εφαρμοστεί σε μία νέα παράμετρο, από το σύνολο Y για να επιστρέψει τελικά ένα στοιχείο από το σύνολο Z. Αν εφαρμόσουμε την g στον αριθμό 3 για παράδειγμα θα έχουμε τελικώς

$$g\ 3 = 2 + \ 3^2 = 11 = h\ 2\ 3 = f(2,3)$$

Αφού είδαμε τη βασική θεωρία πάνω στην οποία οικοδομείται η τεχνική **currying**, καθώς και η μερική εφαρμογή (**partial application**), πάμε να μελετήσουμε τον προγραμματιστικό ορισμό των εννοιών αυτών και στη συνέχεια την υλοποίηση τους στη γλώσσα της JavaScript. Στην επιστήμη των υπολογιστών και συγκεκριμένα στη συναρτησιακή προγραμματιστική θεωρία οι τεχνικές currying και partial application αποτελούν στενά αλληλένδετες έννοιες. **Currying** ονομάζεται η διαδικασία μετατροπής μίας συνάρτησης ν-παραμέτρων (n-ary arity) σε μία σειριακή διάταξη από ν συναρτήσεις μίας παραμέτρου (unary functions), καθεμία από τις οποίες δέχεται διαδοχικά από μία παράμετρο της αρχικής αυθεντικής συνάρτησης, ξεκινώντας από αριστερά (Δηλαδή η πρώτη συνάρτηση δέχεται την πρώτη παράμετρο της αρχικής συνάρτησης, η δεύτερη τη δεύτερη κοκ). Η κλήση μίας εξ αυτών των συναρτήσεων με ορίσματα, δημιουργεί και επιστρέφει την αμέσως επόμενη στη σειρά συνάρτηση. Η διαδικασία αυτή συνεχίζεται μέχρις ότου η τελευταία συνάρτηση θα δεχτεί ως είσοδο το τελευταί όρισμα και θα επιστρέψει το τελικό αποτέλεσμα. **Partial Application** από την άλλη είναι η διαδικασία

μετατροπής μίας συνάρτησης ν-παραμέτρων (n-ary arity) σε μία νέα συνάρτηση ν-μ παραμέτρων ({n-m}ary arity, όπου μ το πλήθος των παραμέτρων με την οποία κλήθηκε η αρχική αυθεντική συνάρτηση). Κάθε φορά που η νέα παραγόμενη συνάρτηση εφαρμόζεται σε ένα x πλήθος παραμέτρων, η κλήση γεννά μία νέα συνάρτηση που είναι έτοιμη να δεχτεί τις υπολειπόμενες παραμέτρους. Η διαδικασία αυτή συνεχίζεται ώσπου το σύνολο των ορισμάτων που εισήχθη στις παραγόμενες συναρτήσεις ισοδυναμεί με το πλήθος των ορισμάτων της αρχικής συνάρτησης, όπου και τελικά αποτιμάται το τελικό αποτέλεσμα. Η διαφορά που διακρίνει τις δύο τεχνικές είναι ότι η μερική εφαρμογή (**partial application**) μπορεί να δεχτεί τα ορίσματα μαζικά αλλά και σε διαφορετική σειρά εισάγοντας ενδιάμεσα την τιμή **undefined.** Τέλος υπάρχει και μία τρίτη τεχνική που συνήθως ονομάζεται **partial currying** που επιτρέπει επίσης την παροχή περισσότερο από μίας παραμέτρου στην τροποποιημένη συνάρτηση και παράγει μία νέα συνάρτηση που αναμένει στην είσοδο της τις υπολειπόμενες παραμέτρους. Όταν το σύνολο των ορισμάτων δοθεί η συνάρτηση επιστρέφει το τελικό αποτέλεσμα. Αξίζει να σημειωθεί ότι η υλοποίηση αυτή είναι η ίδια με αυτή που προσφέρει η τεχνική currying απλά προσφέρεται η δυνατότητα εφαρμογής της συνάρτησης σε περισσότερες από μία παραμέτρους. Ως εκ τούτου πολλές φορές στον χώρο μιλάμε για curried συναρτήσεις που στην πραγματικότητα όμως υλοποιούνται ως partial curried λειτουργικότητες, χωρίς αυτό να δημιουργεί κάποιο πρόβλημα αφού όπως είπαμε μία partial curried συνάρτηση είναι και curried. Αρκετά όμως με τη θεωρία, πάμε να δούμε την απήχηση της τεχνικής στον συναρτησιακό προγραμματισμό.

Στην πραγματικότητα η τεχνική αυτή έχει χρησιμοποιηθεί ήδη αρκετές φορές σε προηγούμενα παραδείγματα, δίχως όμως τη συνοδεία μίας ολοκληρωμένης λειτουργικότητας και θεωρίας που μας επιτρέπει να αντιληφθούμε τη θέση της στον συναρτησιακό μοντέλο προγραμματισμού.

Έστω λοιπόν ότι έχουμε τη γνωστή συνάρτηση άθροισης υλοποιημένη στην JavaScript

```
// add :: (Number, Number) -> Number
const add = (x, y) => x + y;
```

Η συγκεκριμένη συνάρτηση δέχεται δύο αριθμητικές παραμέτρους και επιστρέφει το άθροισμα τους. Αν θέλουμε να υπολογίσουμε για παράδειγμα το άθροισμα 2 + 7 τότε αρκεί να εφαρμόσουμε τη συνάρτηση στα ορίσματα μας όπως φαίνεται παρακάτω

```
add(2, 7);
//-> 9
```

Η τεχνική currying προσθέτει ένα αφαιρετικό επίπεδο που επιτρέπει την αξιοποίηση της συνάρτησης **add** ως μία γεννήτρια συναρτήσεων(αθροιστών στην προκειμένη περίπτωση) προσφέροντας έτσι νέες λειτουργικότητες έτοιμες προς χρήση σε δομές που χαράζει η συναρτησιακή προσέγγιση(πχ **composition**). Έτσι αν υποθέσουμε ότι η συνάρτηση **add** είναι στην curried μορφή της, τότε το ζητούμενο άθροισμα θα προκύπτει πλέον από την έκφραση

```
add(2)(7);
//-> 9
```

Η αποτίμηση της έκφρασης **add(2)** επιστρέφει μία νέα συνάρτηση που είναι έτοιμη να δεχτεί ένα αριθμητικό όρισμα και να επιστρέψει το τελικό αποτέλεσμα το οποίο ορίζει το σώμα της συνάρτησης (άθροισμα). Σε μία άμεση μορφή της η συγκεκριμένη λειτουργικότητα θα μπορούσε να υλοποιηθεί από την παρακάτω αναπαράσταση κώδικα

```
// add :: Number -> Number -> Number
const add = x => y => x + y;
```

μετασχηματίζοντας την αρχική συνάρτηση **add** με υπογραφή

$$add :: (Number, Number) \rightarrow Number$$

σε μία νέα συνάρτηση με υπογραφή

$$add :: Number \rightarrow Number \rightarrow Number$$

όπως δηλαδή έχει περιγράφει και από το μαθηματικό μοντέλο στο οποίο στηρίζεται η τεχνική. Στη συγκεκριμένη περίπτωση μία συνάρτηση δύον παραμέτρων μετετράπη σε μία σειρά συναρτήσεων μίας παραμέτρου, η κλήση των οποίων επιστρέφει είτε νέες συναρτήσεις πχ. **add(2),** είτε το τελικό αποτέλεσμα πχ. **add(2)(7)** ανάλογα με τον αριθμό των παραμέτρων. Έτσι ουσιαστικά με την παροχή ενός ορίσματος έχουμε στη διάθεση μας μία νέα λειτουργικότητα η οποία μπορεί να πάρει υπόσταση ονοματίζοντας την

```
// adder2 :: Number -> Number
const adder2 = add(2);
```

Η τεχνική αυτή αποτελεί εξαιρετικά χρήσιμη πρακτική στον συναρτησιακό προγραμματισμό καθώς πρώτον, οι συναρτήσεις έρχονται σε πλήρη ταύτιση με τις αφαιρέσεις λάμδα και δεύτερον, αποτελούν έναν τρόπο σταδιακής εφαρμογής των αφαιρετικών λειτουργικοτήτων προσφέροντας έτσι στον προγραμματιστή τη δυνατότητα τροποποίησης ή εμπλουτισμού τους στους άξονες που ορίζει τον συναρτησιακό μοντέλο γραφής. Μάλιστα πρέπει να τονίσουμε ότι γνωστές γλώσσες συναρτησιακού προγραμματισμού προσφέρουν εγγενώς τη συγκεκριμένη ιδιότητα. Για παράδειγμα στην Haskell, που η συνάρτηση άθροισης συμβολίζεται με το γνωστό σύμβολο της πρόσθεσης + , η έκφραση (+ 1 2) επιστρέφει το αποτέλεσμα 3, όμως η αποτίμηση της έκφρασης (+ 1) επιστρέφει μία νέα συνάρτηση που αναμένει την επόμενη παράμετρο για να προχωρήσει στη αποτίμηση του αποτελέσματος που θα προκύψει από το άθροισμα της πρώτης παραμέτρου με την αμέσως επόμενη (δηλαδή τον αριθμό 1). Η JavaScript μπορεί να μην προσφέρει κάτι τέτοιο, αλλά όπως έχουμε πει μεταχειρίζεται τις συναρτήσεις ως πολίτες πρώτης κατηγορίας (first class citizens) και αυτό είναι αρκετό για την εφαρμογή του συναρτησιακού τρόπου προσέγγισης. Έτσι στην προκειμένη περίπτωση μπορούμε να προχωρήσουμε στη σχεδίαση μίας νέας βοηθητικής λειτουργικότητας που θα μετατρέπει τις κλασσικές **uncurried** συναρτήσεις στην **curried** τους μορφή. Με λίγα λόγια μπορούμε να δημιουργήσουμε μία συνάρτηση υψηλής τάξης (**higher order function**), που θα δέχεται ως είσοδο μία πολυπαραμετρική συνάρτηση, και θα επιστρέφει τη συνάρτηση αυτή στην **curried** της μορφή (δηλαδή μια σειρά από μονοπαραμετρικές συναρτήσεις διατηρώντας τη λειτουργικότητα της αρχικής συνάρτησης). Για να γίνει κατανοητό καλύτερα το ζητούμενο, ας υποθέσουμε ότι έχουμε στην κατοχή μας μία τέτοια συνάρτηση, πώς θα τη χρησιμοποιούσαμε; Παρακάτω βλέπουμε ένα απλό παράδειγμα

```
// add :: (Number, Number) -> Number
const add = (x, y) => x + y;

const curriedAdd = curry(add);

const adder2 = curriedAdd(2);

adder2(7);
//-> 9
```

Η συνάρτηση **curry** μετατρέπει την **add** κατά τα πρότυπα της τεχνικής currying με αποτέλεσμα να μην χρειάζεται να γράψουμε την **add** με τη μορφή που φαίνεται πιο κάτω

```
// add :: Number -> Number -> Number
const add = x => y => x + y;
```

Πως υλοποιείται όμως μία τέτοια συνάρτηση μετατροπής συναρτήσεων ακαθάριστου αριθμού παραμέτρων σε curried μορφές; Πάμε να δούμε την υλοποίηση βήμα βήμα. Ας υποθέσουμε ότι για αρχή θέλουμε η συνάρτηση curry να δέχεται συναρτήσεις δύον παραμέτρων, όπως η **add.** Τότε η παρακάτω υλοποίηση φαίνεται λογική

```
// curry :: ((a, b) -> c) -> a -> b -> c
const curry = fn => x => y => fn(x, y);

// add :: (Number, Number) -> Number
const add = (x, y) => x + y;

// curriedAdd :: Number -> Number -> Number
const curriedAdd = curry(add);
```

και ευσταθεί για συναρτήσεις δύο παραμέτρων, όχι όμως περισσότερων.

```
// curry :: ((a, b) -> c) -> a -> b -> c
const curry = fn => x => y => fn(x, y);

// mul :: (Number, Number, Number) -> Number
const mul = (a, b, c) => a * b * c;

// curriedAdd :: Number -> Number -> Number
const curriedMul = curry(mul);

curriedMul(3)(4)(5);
//-> TypeError: curriedMul is not a function
```

Συνεπώς πρέπει να σκεφτούμε μία πιο γενική υλοποίηση που φυσικά να ικανοποιεί και να καλύπτει τα πλαίσια στα οποία στηρίζεται η βασική θεωρία της τεχνικής. Αν αναλύσουμε προσεκτικότερα την υλοποίηση της **curry** δύο παραμέτρων, θα παρατηρήσουμε ότι αυτή επιτυγχάνεται μέσω της δομής των **closures**. Πράγματι πολλές από τις συναρτησιακές τεχνικές στην JavaScript, και όχι μόνο, δύναται να υλοποιηθούν χάρη στη λειτουργικότητα που προσφέρουν τα closures (δηλαδή η δυνατότητα διατήρησης του περιβάλλοντος εντός του οποίου ορίζεται μία συνάρτηση) και στην εύκολη ερμηνεία των παραστάσεων κώδικα μέσω του σταθερού scope (ή πιο γνωστά lexical scoping) το

οποίο έρχεται σε αρμονία με τη λογική αποτίμησης που ακολουθεί ο λογισμός λάμδα. Συνακόλουθα πρέπει να σκεφτούμε μία δομή η οποία θα δημιουργεί μία νέα λειτουργικότητα, όσο ο αριθμός των παραμέτρων που έχει εισαχθεί είναι μικρότερος από αυτόν της αυθεντικής συνάρτησης, μεταφέροντας όμως τις προηγούμενες παραμέτρους στο νέο αυτό περιβάλλον. Αν θα θέλαμε να ορίσουμε την curry με δυνατότητα μετατροπής συναρτήσεων τριών παραμέτρων, θα είχαμε

```
// curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
const curry3 = fn => x => y => z => fn(x, y, z);
```

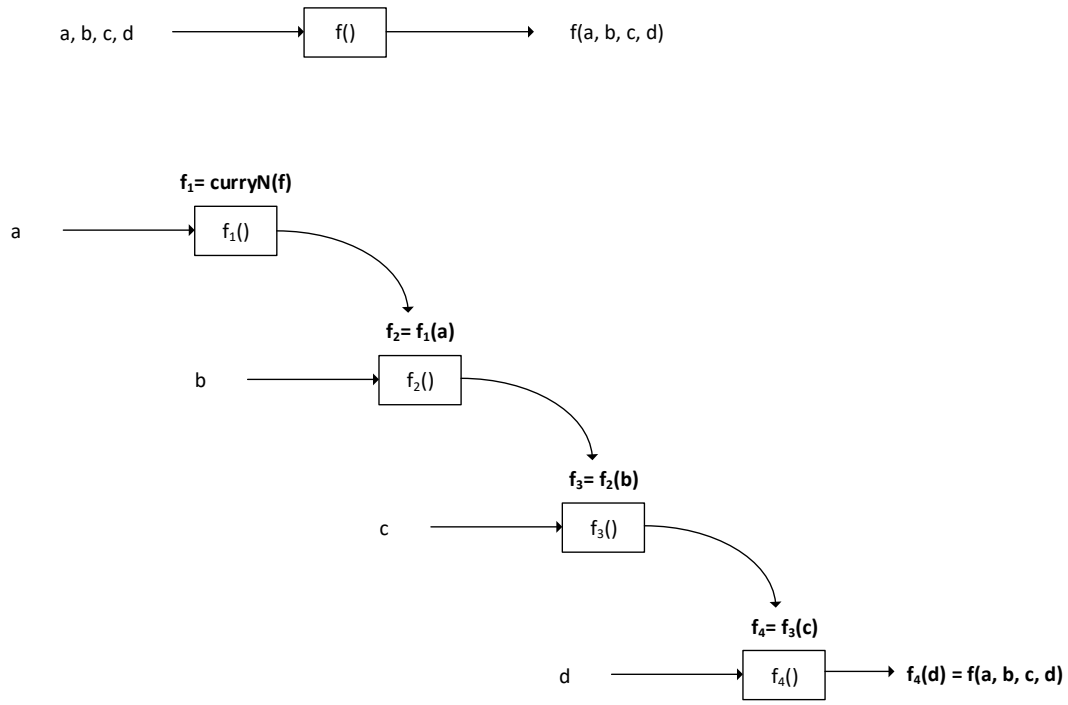
Παρατηρούμε ότι το πλήθος των παραμέτρων που δέχεται η συνάρτηση **fn** καθορίζει και το πλήθος των εμφωλευμένων **closures** που ορίζονται από το σώμα

```
x => y => z => fn(x, y, z);
```

Συνεπώς εύλογα μπορούμε να συμπεράνουμε ότι θα χρειαστούμε μία αναδρομική διαδικασία που θα είναι σε θέση να δημιουργεί δυναμικά τα νέα αυτά περιβάλλοντα, και τελικά να επιστρέφει την αποτίμηση της αρχικής συνάρτησης με εφαρμογή στις δοθείσες παραμέτρους. Παρακάτω βλέπουμε μία λύση, που χρησιμοποιεί **closures** και αναδρομή.

```
// curryN :: ((a, b, c, ..., y) -> z) -> a -> b -> c -> ... -> y -> z
const curryN = fn => {
    const curried = (...args) => x => (args.length + 1 < fn.length ?
curried : fn)(...args, x);
    return curried();
};
```

Η συνάρτηση **curryN** λειτουργεί ως decorator τροποποιώντας η εμπλουτίζοντας τη λειτουργικότητα της συνάρτησης που θα εισαχθεί. Εσωτερικά ορίζεται η συνάρτηση **curried** η οποία σε κάθε κλήση της αναλαμβάνει τη δημιουργία μία νέας συνάρτησης της οποίας όμως το περιβάλλον θυμάται όλες τις παραμέτρους που έχουν εισαχθεί μέχρι το σημείο εκείνο. Η λειτουργικότητα αυτή είναι απλοποιημένη χάρη στον τελεστή … (gathering, spreading) που μας επιτρέπει τη διαχείριση ακαθάριστου αριθμού μεταβλητών. Επίσης παρατηρούμε ότι η συνθήκη (args.length + 1 < fn.length ?) ελέγχει τη συνάρτηση επιστροφής και, όπως ξέρουμε από τη θεωρία της τεχνικής **currying**, είτε επιστρέφει την επόμενη στη σειρά μονοπαραμετρική συνάρτηση, είτε αποτιμά το τελικό αποτέλεσμα καλώντας την αρχική συνάρτηση **fn** με τις παραμέτρους που πρέπει. Αν θα θέλαμε να δούμε σχηματικά τη λειτουργία της **curryN**, τότε αυτή θα είχε τη μορφή που απεικονίζεται πιο κάτω.



Σχήμα 4-9 Λειτουργία συνάρτησης curry

Φαίνεται χαρακτηριστικά η σταδιακή δημιουργία των μονοπαραμετρικών συναρτήσεων έως ότου συμπληρωθεί και η τελευταία παράμετρος και επιστρέφεται η τιμή **f(a, b, c, d).** Οι συναρτήσεις f_1, f_2, f_3, f_4 προκύπτουν από την πράξη της εφαρμογής (application) και η αρχική συνάρτηση f μπορούμε να πούμε ότι συμπεριφέρεται ακριβώς όπως και μία αφαίρεση λάμδα. Και όλα αυτά μέσω της τεχνικής **currying.** Αν ορίσουμε λοιπόν ως f τη συνάρτηση **add** (άθροιση τεσσάρων παραμέτρων) που εικονίζεται παρακάτω

```
// add :: (Number, Number, Number, Number) -> Number
const add = (x, y, z, w) => x + y + z + w;
```

Τότε εισάγοντας τη στη συνάρτηση υψηλής τάξης **curryN** θα έχει ως αποτέλεσμα τη δημιουργία των σταδιακών συναρτήσεων f_1, f_2, f_3, f_4 όπως φαίνεται στο σχήμα 4.9, και παρακάτω στην JavaScript

```
// f1 :: Number -> Number -> Number -> Number -> Number
const f1 = curryN(add);
// f2 :: Number -> Number -> Number -> Number
const f2 = f1(1);
// f3 :: Number -> Number -> Number
const f3 = f2(2);
// f4 :: Number -> Number
const f4 = f3(3);

const result = f4(4);
//-> 10
```

Αφού είδαμε τον τρόπο υλοποίησης της currying λειτουργικότητας, πάμε να εμβαθύνουμε σε μία άλλη παρόμοια έννοια γνωστή στον συναρτησιακό προγραμματισμό ως μερική εφαρμογή (**partial application**). Πράγματι η μερική εφαρμογή έχει ως σκοπό να δώσει πρακτική υπόσταση σε μία αφαίρεση λάμδα, προσφέροντας τις παραμέτρους, ακριβώς όπως και η τεχνική currying, με τη διαφορά όμως ότι στην πρώτη περίπτωση δύναται η μαζική, και σε ακαθόριστη σειρά χρήση των παραμέτρων μέσω της τιμής **undefined.** Παρακάτω φαίνεται η υλοποίηση της

```
// partialApp :: ((a, b, c, ..., y) -> z) -> (a | undefined -> b |
undefined -> c | undefined -> ... -> y | undefined -> z) | ((a |
undefined, b | undefined) -> c | undefined -> ... -> y | undefined ->
z) | ((a | undefined, b | undefined, c | undefined) -> ... -> y |
undefined -> z) | ... | (a | undefined, b | undefined, c | undefined,
..., y | undefined) -> z
const partialApp = fn => {
    const partialize = (...args1) => (...args2) => {
        const newParams = [...args1.map(x => x === undefined ?
args2.shift() : x), ...args2];
        return (newParams.includes(undefined) || args1.length +
args2.length < fn.length ? partialize : fn)(...newParams);
    };

    return partialize();
};
```

Η εσωτερική συνάρτηση **partialize** αναλαμβάνει τη στοίχιση των παραμέτρων στη σωστή διάταξη λαμβάνοντας υπόψιν το γεγονός της ύπαρξης της **undefined** τιμής εντός αυτών. Οι τιμές undefined χρησιμοποιούνται ως θέσεις καθυστερημένης εισαγωγής παραμέτρων επιτρέποντας έτσι τον καθορισμό τους αγνοώντας τη σειρά προσπέλασης

τους, που είναι από αριστερά προς τα δεξιά. Για να γίνει κατανοητή καλύτερα η λειτουργία της πάμε να δούμε ένα παράδειγμα. Έστω ότι έχουμε την παρακάτω συνάρτηση

```
// strParams :: (String, String, String, String) -> String
const strParams = (a, b, c, d) => `1st param: ${a}, 2nd param: ${b},
3rd param: ${c}, 4th param: ${d}`;
```

Τότε κάνοντας χρήση της **partialApp** μπορούμε να πετύχουμε τα παρακάτω αποτελέσματα

```
pstrParams("random1", undefined, "random3")(undefined,
"random4")("random2");
//-> 1st param: random1, 2nd param: random2, 3rd param: random3, 4th
param: random4
```

ή

```
pstrParams(undefined, undefined, undefined, "random4")("random1",
"random2", "random3");
//-> 1st param: random1, 2nd param: random2, 3rd param: random3, 4th
param: random4
```

ή

```
const partialApplicationWithRandoms = pstrParams(undefined, "random2",
undefined, "random4");

partialApplicationWithRandoms("felix1", "felix3");
//-> 1st param: felix1, 2nd param: random2, 3rd param: felix3, 4th
param: random4
```

Παρατηρούμε ότι βάζοντας την τιμή **undefined** δηλώνουμε ότι η παράμετρος θα συμπληρωθεί σε κάποια επόμενη κλήση των **curried** συναρτήσεων. Επίσης αξίζει να σημειωθεί ότι η μερική εφαρμογή επιτρέπει και τη μαζική εισαγωγή παραμέτρων σε αντίθεση με την αυστηρά καθορισμένη τεχνική **currying** που προβλέπει μονοπαραμετρικές εισαγωγές. Μάλιστα όπως θα δούμε, η συγκεκριμένη δυνατότητα αποδεικνύεται εξαιρετικά χρήσιμη και ως εκ τούτου μπορούμε να υλοποιήσουμε και μία διαφορετική εκδοχή της τεχνικής **currying**, γνωστή ως **partialCurrying,** εμπλουτισμένη με την προαναφερθείσα λειτουργικότητα. Έτσι τροποποιώντας ελαφρός την υλοποίηση της currying όπως απεικονίζεται παρακάτω

```
// curryN :: ((a, b, c, ..., y) -> z) -> (a -> b -> c -> ... -> y) ->
z | (a -> b -> c ...) -> y -> z | ... | a -> b -> c ... -> y -> z
const pcurryN = fn => {
    const curried = (...args1) => (...args2) => (args1.length +
args2.length < fn.length ? curried : fn)(...args1, ...args2);
    return curried();
};
```

Χρησιμοποιώντας δηλαδή την τεχνική **gathering** μέσω της μεταβλητής **…args2** μπορούμε πλέον να κάνουμε χρήση των μετασχηματισμένων συναρτήσεων εισάγοντας σε αυτές παραπάνω από μία παράμετρο.

```
// add :: (Number, Number, Number, Number) -> Number
const add = (x, y, z, w) => x + y + z + w;

pcurryN(add)(1, 2, 3)(4);
//-> 10
```

αντί για

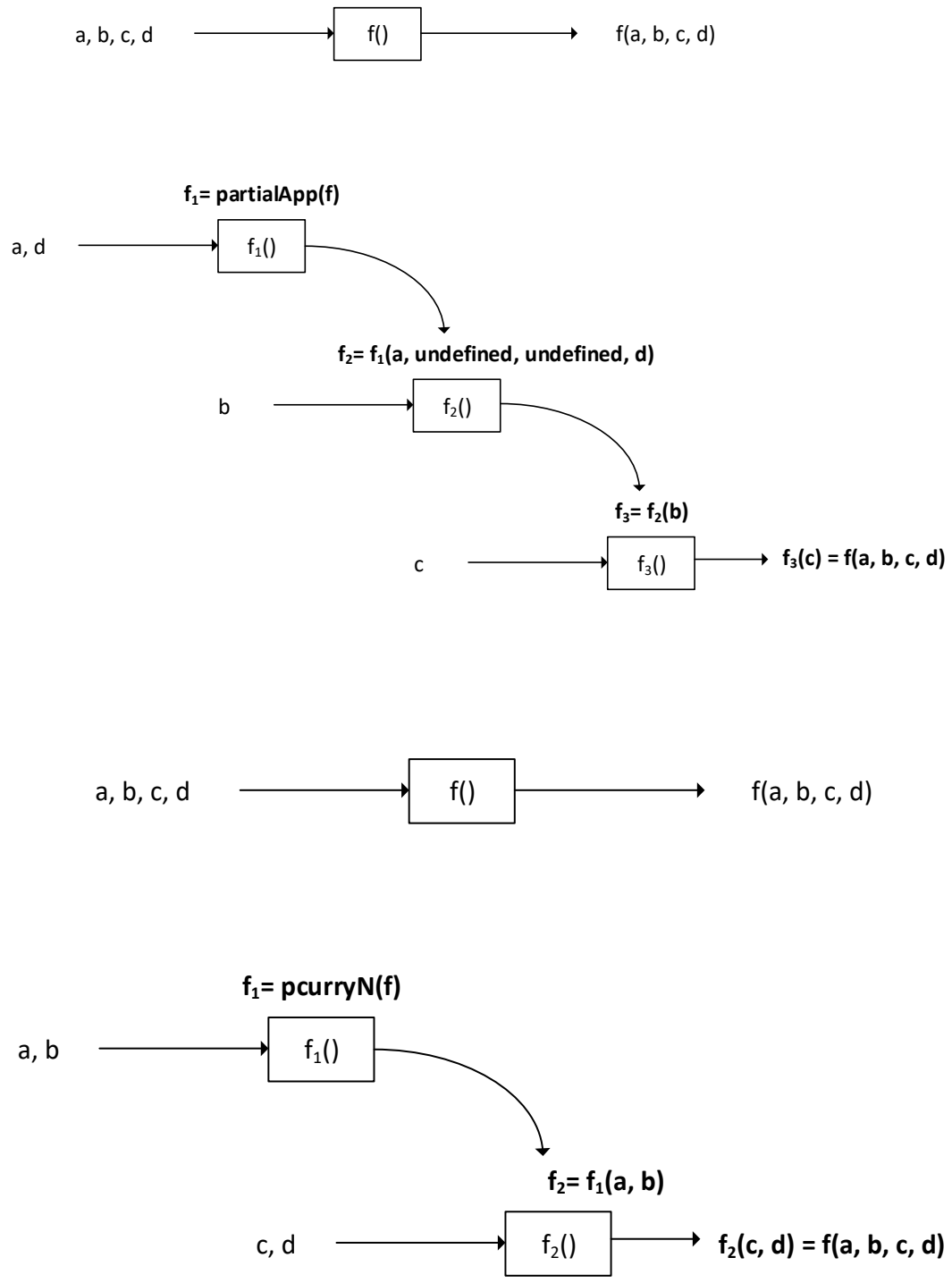
```
curryN(add)(1)(2)(3)(4);
//-> 10
```

Αξίζει να επισημάνουμε ότι η λειτουργία της **pcurryN** καλύπτει και αυτή της κλασσικής συνάρτησης **curryN** και συνεπώς για λόγους χρηστικότητας είναι αυτή που συνήθως επιλέγεται όταν θέλουμε να προσδώσουμε σε μία συνάρτηση την υπόσταση μίας αφαίρεσης λάμδα. Με λίγα λόγια μπορούμε να γράψουμε

```
pcurryN(add)(1)(2)(3)(4);
//-> 10
```

χωρίς κάποιο πρόβλημα.

Προτού κλείσουμε το συγκεκριμένο υποκεφάλαιο, πρέπει να πούμε ότι η τεχνική **currying** μαζί με τη σύνθεση συναρτήσεων (**composition**) αποτελούν τα πιο σημαντικά εργαλεία εφαρμογής του συναρτησιακού προγραμματισμού με την έννοια ότι αποτελούν αλφάβητο για οποιοδήποτε θέλει να εξοικειωθεί με τη συγκεκριμένη προσέγγιση. Συνεπώς πρέπει να δίδεται η απαραίτητη προσοχή και έμφαση. Τέλος παραθέτουμε δύο σχήματα, ένα για καθεμία από τις τεχνικές **partialApplication** και **partialCurrying**, που περιγράφουν παραστατικά τις συγκεκριμένες λειτουργικότητες.



Σχήμα 4-10 Λειτουργίες συνάρτησης partialCurry

4.10 Συναρτησιακές Δομές (Functional Data Structures)

Για να δούμε τι ακριβώς μελετάει το παρόν κεφάλαιο θα ήταν ωφέλιμο να γίνει μία μικρή ιστορική αναδρομή. Λόγος θα γίνει για τη γνωστή γλώσσα **LISP** (List processing) ή οποία ουσιαστικά ήταν η πρώτη γλώσσα στην οποία μπορούσες να υιοθετήσεις τις βασικές αρχές του λογισμού λάμδα και συνεπώς μπορούσες να χρησιμοποιήσεις μία πιο συναρτησιακή προσέγγιση, αν και αυτό δεν ήταν απαραίτητο (δεν σε ανάγκαζε η γλώσσα όπως για παράδειγμα κάνει η **Closure** η οποία δανείστηκε χαρακτηριστικά της **LISP**). Η LISP, όπως διαφαίνεται και από το όνομα της, έκανε βαριά χρήση της δομής των λιστών σε μία υλοποίηση η οποία ήταν τρομερά αποδοτική για τους επεξεργαστές της εποχής καθότι χρησιμοποιούσε τις λεγόμενες κυψέλες cons (**cons cells**) σε μία σειριακή διάταξη με τους καταχωρητές να λαμβάνουν σε επίπεδο υλικού απευθείας τις τιμές των κυψελών (Braithwaite, 2017). Μία κυψέλη **cons** φιλοξενούσε δύο τιμές, η πρώτη της οποίας η προσπέλαση γινότανε με τη μακροεντολή **car** περιείχε συνήθως μία πρωτόγονη τιμή, και η δεύτερη η οποία φιλοξενούσε τη διεύθυνση της μνήμης στην οποία ήταν αποθηκευμένη η επόμενη κυψέλη. Η εξαγωγή της διεύθυνσης επιτυγχάνονταν μέσω της κλήσης **cdr.** Η διάταξη αυτή, όπως πολλοί μαντεύουμε, ήταν μία σειρά από διασυνδεμένες κυψέλες που στο σύνολο τους υλοποιούσαν δομή δεδομένων γνωστή και ως διασυνδεμένη λίστα (**Linked List**). Προσομοιώνοντας την προσέγγιση αυτή στη γλώσσα της JavaScript θα μπορούσαμε να γράψουμε τις παρακάτω συνθήκες.

```
const cons = (x, y) => [x, y];

const car = ([x, y]) => x;

const cdr = ([x, y]) => y;
```

Έτσι αν θα θέλαμε να σχηματίσουμε μία λίστα με στοιχεία του αριθμούς από 0 έως 4 θα γράφαμε

```
const zeroToFour = cons(0, cons(1, cons(2, cons(3, cons(4, null)))));
//-> [0, [1, [2, [3, [4, null]]]]]

car(zeroToFour);
//-> 0

cdr(zeroToFour);
//-> [1, [2, [3, [4, null]]]]
```

Το ερώτημα που τίθεται είναι αν μία τέτοια δομή είναι ακόμα επίκαιρη από πλευράς απόδοσης η χρηστικότητας. Γιατί δηλαδή να μην ομαδοποιήσουμε τα στοιχεία μας σε έναν απλό πίνακα

```
const zeroToFour = [0, 1, 2, 3, 4];
//-> [0, 1, 2, 3, 4]
```

Από πλευράς απόδοσης θα λέγαμε ότι οι διαφορές των δύο δομών ανάγονται στις γνωστές διαφορές που παρουσιάζουν η δομή της διασυνδεμένης λίστας (Linked List) με αυτή ενός πίνακα (Array, List). Λόγος γίνεται για την πολυπλοκότητα των αλγορίθμων εισαγωγής και πρόσβασης που συνοδεύουν τις δομές αυτές. Η πολυπλοκότητα του αλγορίθμου εισαγωγής ενός στοιχείου σε μία διασυνδεμένη λίστα είναι μικρότερη από αυτή της εισαγωγής σε έναν πίνακα. Από την άλλη η πρόσβαση ενός στοιχείου ενός πίνακα είναι ακαριαία, σε αντίθεση με την πρόσβαση σε μία διασυνδεμένη λίστα όπου θα απαιτηθεί γραμμική προσπέλαση της. Από πλευράς χρηστικότητας τώρα, θα λέγαμε ότι η απεικόνιση μία δομής κατά αυτόν τον τρόπο, ενεργοποιεί τον συναρτησιακό προγραμματισμό αφού δίνεται η δυνατότητα ερμηνείας των δομών μέσω της αρχής της αυτό-ομοιότητας (self-similarity). Με λίγα λόγια η αναπαράσταση μίας δομής μέσω του εαυτού της (όπως η δομή cons(cons(cons…))) επιτρέπει στον συναρτησιακό προγραμματισμό τη δημιουργία αγνών δομών δεδομένων οι οποίες παρουσιάζουν τις ίδιες λειτουργικότητες με τις κλασσικές υλοποιήσεις των δομών αυτών (πχ μέσω αντικειμενοστρέφειας) με τη διαφορά ότι όντας συναρτήσεις, μπορούν να επωφεληθούν από τις τεχνικές που εισάγει η συναρτησιακή προσέγγιση (πχ συναρτήσεις υψηλής τάξης). Το σημείο κλειδί, και ο λόγος που έγινε αναφορά στην υλοποίηση της λίστας στην LISP, είναι η δυνατότητα αναπαράστασης των δομών μέσω του εαυτού τους. Η προσέγγιση αυτή ταιριάζει απόλυτα με τη θεωρία στην οποία στηρίζεται η αναδρομή και ως εκ τούτου οι δομές αυτές εκφράζονται εγγενώς από αυτήν. Ο συναρτησιακός προγραμματισμός χρησιμοποιεί λοιπόν την αναδρομική τεχνική για να απεικονίσει γνωστές δομές δεδομένων, όπως οι λίστες, τα δέντρα, οι γράφοι κα, και να κάνει εφικτή την ομαδοποίηση στοιχείων με τον τρόπο που αυτός γνωρίζει (δηλαδή χρήση συναρτήσεων). Φυσικά η προσέγγιση αυτή, στέκεται κάτω από την ομπρέλα του αυστηρού συναρτησιακού προγραμματισμού, και δεν είναι απαραίτητη η υιοθέτηση της σε μία γλώσσα όπως είναι η JavaScript. Άλλες γλώσσες, καθαρά συναρτησιακές, στηρίζονται εγγενώς σε αναδρομικές

συναρτησιακές δομές. Παρά ταύτα, αποτελεί όπως είπαμε έναν διαφορετικό τρόπο ανάγνωσης και ερμηνείας των δόμων και ξεκλειδώνει άλλες δυνατότητες οι οποίες είναι ευκολότερα υλοποιήσιμες στη συναρτησιακή τους υπόσταση ακόμα και στην JavaScript. Ως εκ τούτου είναι συνετό να δούμε τον τρόπο με τον οποίο αυτές ορίζονται.

Προτού δούμε την υλοποίηση της δομής της λίστας στην αμιγώς συναρτησιακή της μορφή, πάμε να δούμε λίγο πως θα μπορούσαμε να χρησιμοποιήσουμε τη δομή **cons** για τη δυναμική δημιουργία μίας ομαδοποιημένης διάταξης. Όπως είπαμε, λόγω της φύσης της δομής, αυτή θα επιτευχθεί χρησιμοποιώντας αναδρομική λύση. Για παράδειγμα

```
// range :: (Number, Number) -> Cons Number
const range = (x, y) => x < y ? cons(x, range(x + 1, y)) : null;

const zeroToFour = range(0, 5);
//-> [0, [1, [2, [3, [4, null]]]]]
```

Η συνάρτηση **range** δέχεται δύο αριθμούς x, y στην είσοδο και επιστρέφει μία διασυνδεμένη λίστα από κελία **cons** η οποία περιέχει ως στοιχεία τους ακέραιους αριθμούς που ορίζονται από το εύρος [x, y). Θα μπορούσαμε επίσης να φτιάξουμε μία συνάρτηση που να μετατρέπει τη δομή ενός πίνακα σε διασυνδεμένη λίστα.

```
// consFrom :: [a] -> Cons a
const consFrom = ([first, ...rest]) => first === undefined ? null :
cons(first, consFrom(rest));

consFrom(["random1", "random2", "random3", "random4"])
//-> [ 'random1', [ 'random2', [ 'random3', ['random4', null] ] ] ]
```

Η **consFrom** δέχεται έναν πίνακα και επιστρέφει την ισοδύναμη μορφή σε διασυνδεμένη λίστα. Παρατηρούμε ότι όλες οι υλοποιήσεις περιέχουν αναδρομικές κλήσεις των συναρτήσεων σε μία απόπειρα δημιουργίας της εμφωλευμένης επαναληπτικής διάταξης. Μάλιστα αν εμβαθύνουμε περισσότερο, μπορούμε να δούμε ότι οι αναδρομικές κλήσεις συμβαίνουν εσωτερικά της δομής **cons** και η πράξη της δημιουργίας του κελιού **cons** είναι δεξιά προσεταιριστική (right-associative). Πιο πάνω και συγκεκριμένα στην πράξη της σύνθεσης συναρτήσεων αντιμετωπίσαμε μια παρόμοια περίπτωση στην οποία είχαμε κάνει αναφορά στις συναρτήσεις **foldLeft** και **foldRight**. Οι συναρτήσεις αυτές είναι αρκετά χρήσιμες καθώς επιτρέπουν την αφαιρετική προσέγγιση διαφόρων αναδρομικών λειτουργιών, όπως στην προκειμένη περίπτωση είναι η δομή **cons.**

Η δομή αυτή αποδεικνύεται ότι μπορεί εύκολα να αναπαρασταθεί λόγω της μορφής της από τη συνάρτηση υψηλής τάξης **foldRight** (Lipovaca, 2011)**.** Στην Haskell ο ορισμός της συνάρτησης φαίνεται παρακάτω

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Αν μεταφέρουμε τον ορισμό στη γλώσσα της JavaScript έχουμε

```
// foldr :: ((a, b) -> b, b, [a]) -> b
const foldr = (f, z, [first, ...rest]) => first === undefined ? z :
f(first, foldr(f, z, rest));
```

Μετατρέποντας την **foldr** στην curried της μορφή, μπορούμε να γράψουμε την **consFrom** ως εξής

```
// consFrom :: [a] -> Cons a
const consFrom = curryN(foldr)(cons)(null);

consFrom(["r1", "r2", "r3", "r4", "r5"]);
//-> [ 'r1', [ 'r2', [ 'r3', ['r4', ['r5', null] ] ] ] ]
```

Βλέπουμε λοιπόν ότι αν εισάγουμε στη συνάρτηση **foldr** τη συνάρτηση **cons** ως παράμετρο και ως αρχική τιμή z την τιμή **null**, αυτό που προκύπτει είναι μία συνάρτηση που αναμένει έναν πίνακα και τον μετατρέπει σε διασυνδεμένη λίστα από κελία **cons.** Αυτό που επιστρέφεται με λίγα λόγια είναι η συνάρτηση **consFrom.** Αντιλαμβανόμαστε λοιπόν ότι η προσέγγιση των δομών ως μία αναδρομική έννοια εισάγει στο παιχνίδι διαφορετικές οπτικές, πάνω στις οποίες μπορούμε να στηριχτούμε για να διευκολύνουμε την επίλυση του προβλήματος που καλούμαστε να αντιμετωπίσουμε.

4.10.1 Διασυνδεμένη Λίστα pair

Η επίδειξη της δομής **cons** καθάρισε λίγο το θολό τοπίο των αναδρομικών δομών, όμως η υλοποίηση δεν είναι αμιγώς συναρτησιακή. Πως θα ήταν όμως μίας αμιγώς συναρτησιακή υλοποίηση; Αντί λοιπόν να χρησιμοποιούμε τους πίνακες βοηθητικά (όπως στη δομή cons) για να αποθηκεύσουμε την κατάσταση της δομής μας, μπορούμε να πράξουμε το ίδιο εκμεταλλευόμενοι τις συναρτησιακές παραμέτρους και την τεχνική των **closures** χτίζοντας έτσι μία αμιγώς αναδρομική συναρτησιακή δομή. Έστω λοιπόν ότι ορίζουμε τη δομή **pair** η οποία παρουσιάζει τη λειτουργικότητα της παρακάτω αφαίρεσης λάμδα.

```
pair = λx.λy.λs.(s x y)
```

Στην JavaScript η έκφραση παίρνει τη μορφή

```
const pair = x => y => s => s(x)(y);
```

Αποδεικνύεται ότι η δομή αυτή μπορεί να χρησιμοποιηθεί ακριβώς με τον ίδιο τρόπο που χρησιμοποιήθηκαν και τα κελιά **cons,** με τη διαφορά ότι πλέον θα έχουμε μία διασυνδεμένη λίστα υλοποιημένη αμιγώς συναρτησιακά αφού πλέον η κατάσταση της λίστας θα είναι εμπεδωμένη εντός του περιβάλλοντος που θα ορίζει η αλληλουχία των συναρτησιακών κλήσεων. Για να αναπαραστήσουμε λοιπόν την **oneToFour** κάνοντας χρήση της νέας μας δομής θα γράφαμε

```
const pair = x => y => s => s(x)(y);

const oneToFour = pair(1)(pair(2)(pair(3)(pair(4)(null))));
//-> Function
```

Βλέπουμε ότι κάθε κλήση της **pair** συνοδεύεται από την πρώτη παράμετρο που είναι το στοιχείο αποθήκευσης, και τη δεύτερη (σε **curried** μορφή) που αποτελεί την αμέσως επόμενη δομή pair, σχηματίζοντας έτσι μία συναρτησιακή διασυνδεμένη λίστα. Η διάταξη της λίστας έχει ορισθεί και η κατάσταση της παραμένει σταθερή, αλλά τα δεδομένα της πρέπει να γίνουν διαθέσιμα. Με λίγα λόγια πρέπει να καθορίσουμε παράθυρα, που μας επιτρέπουν την προσπέλαση της διάταξης, γιατί αλλιώς η δομή μας είναι αχρείαστη. Και όπως πολύ μαντεύουν, από τον ορισμό της **pair**, τα παράθυρα αυτά θα έχουν να κάνουν με την παράμετρο **s** (selector), η οποία είναι η τρίτη παράμετρος που εισάγεται σε μία συνάρτηση της μορφής **pair**, και έχει στόχο διά της εφαρμογής της (**application**) με παραμέτρους το αριστερό και δεξιό στοιχείο μία δομής pair, να εξάγει το επιθυμητό κομμάτι. Παρακάτω φαίνονται οι αφαιρέσεις λάμδα που μπορούν να χρησιμοποιηθούν ως είσοδοι στην παράμετρο **s** της **pair.**

```
car = λx.λy.x

cdr = λx.λy.y
```

Η αφαίρεση λάμδα **car** είναι γνωστή στη συνδυαστική λογική (combinatorial logic) και ως συνδυαστής Kestrel (είναι γνωστό ότι πολλοί από τους συνδυαστές που μελετά η συνδυαστική λογική δανείζονται τα ονόματα τους από γνωστά είδη πουλιών) ή σκέτο **K**. Επίσης η αφαίρεση **cdr** μπορεί να προκύψει από εφαρμογή του ήδη γνωστού σε εμάς

συνδυαστή **I** (identity function) ή όπως ορίσαμε ταυτοτικού μορφισμού. Έτσι μπορεί να γραφεί

```
K = λx.λy.x
I = λx.x

car = K
cdr = K(I)
```

Μεταφέροντας τις συνθήκες αυτές στο πεδίο της JavaScript θα έχουμε

```
const pair = x => y => s => s(x)(y);
const oneToFour = pair(1)(pair(2)(pair(3)(pair(4)(null))));

const K = x => y => x;
const I = x => x;

const car = K;
const cdr = K(I);

oneToFour(car);
//-> 1
oneToFour(cdr)(car);
//-> 2
```

Βλέπουμε ότι οι συναρτήσεις **car** και **cdr** χρησιμοποιούνται ως παράμετροι στη δομή **pair** της μεταβλητής **oneToFour.** Για να μοιάσει το μοντέλο παραπάνω με αυτό που είδαμε στη δομή από κελία **cons**, μπορούμε να τροποποιήσουμε τις συναρτήσεις επέμβασης **car** και **cdr** κατά τέτοιο τρόπο ώστε να δέχονται στην είσοδο τους μία δομή **pair** και στη συνέχεια να υλοποιείται η κλασσική λειτουργικότητα. Με λίγα λόγια οι νέες συναρτήσεις θα συμπεριλαμβάνουν και την παράμετρο της δομής στην αφαίρεση τους. Στο πεδίο του λογισμού λάμδα θα πάρουν την παρακάτω μορφή

```
const K = x => y => x;
const I = x => x;

const car = λp.p K;
const cdr = λp.p (K I);
```

Συνεπώς πλέον η επέμβαση στα στοιχεία μίας δομής τύπου **pair** θα γίνεται με την κλήση **car(<pair>)** και **cdr(<pair>)** όπως απεικονίζεται παρακάτω στη γλώσσα της JavaScript

```
const K = x => y => x;
const I = x => x;

const car = p => p(K);
const cdr = p => p(K(I));

car(oneToFour);
//-> 1
car(cdr(cdr(oneToFour)));
//-> 3
```

Παρατηρούμε ότι μπορέσαμε να φτιάξουμε μία λίστα από στοιχεία χρησιμοποιώντας μία αμιγώς συναρτησιακή υλοποίηση. Ο συναρτησιακός προγραμματισμός ενθαρρύνει αυτού του είδους τις δομές καθώς πέρα από την ευέλικτη δυνατότητα επεξεργασίας τους, λόγω του ότι είναι συναρτήσεις, είναι και αγνές αφού το μοναδικό παράθυρο που υπάρχει για την τροποποίηση της είναι η τροποποίηση (mutation) ενός σύνθετου στοιχείου (όπως πίνακας, αντικείμενα). Δυστυχώς, όπως προαναφέραμε, στη γλώσσα της JavaScript ο συναρτησιακός τρόπος γραφής πρέπει να συνοδεύεται από αυξημένη μέριμνα όσον αφορά στη μη ελεγχόμενη μεταβολή κάποιας κατάστασης που μπορεί να καταστρατηγήσει και να ακυρώσει πολλά από τα οφέλη της συναρτησιακής προσέγγισης. Φυσικά αυτό είναι εσκεμμένο καθότι η γλώσσα όπως τονίσαμε υποστηρίζει και τις δύο σχολές προγραμματισμού προσφέροντας τεράστια ευελιξία στους προγραμματιστές στη διαδικασία ανάπτυξης λογισμικού. Παρακάτω φαίνεται η τροποποίηση ενός στοιχείου σε μία διασυνδεμένη λίστα

```
const oneToFour = pair({1: 1})(pair({2: 2})(pair({3: 3})(pair({4:
4})(null))));
const x = car(oneToFour);
x["1"] = "mutated";
car(oneToFour);
//-> { '1': 'mutated' }
```

Βέβαια να πούμε ότι στην περίπτωση των κελιών **cons** που χρησιμοποιούσε εσωτερικά τη δομή του πίνακα, ο κίνδυνος τροποποίησης ενός στοιχείου ήταν πολύ μεγαλύτερος καθώς ήταν εφικτή η τροποποίηση και του δείκτη που επέστρεφε η συνάρτηση **cdr**, πιθανότατα διαλύοντας έτσι τη δομή. Στην περίπτωση της **pair** αυτό είναι εφικτό μόνο με μη συμβατικές ενέργειες καθότι είναι συνάρτηση και η τροποποίηση του

σώματος γίνεται μόνο με επανάθεση της τιμής με νέα. Αφού ορίσαμε λοιπόν τη δομή μας, πάμε να δούμε πως ο συναρτησιακός προγραμματισμός είναι σε θέση να προσφέρει λειτουργικότητες εμπλουτίζοντας την.

Για παράδειγμα μπορούμε να υλοποιήσουμε τη συνάρτηση **range** που επιστρέφει σε μία δομή **pair** τους ακέραιους αριθμούς που ορίζονται από το εύρος [x, y).

```
// range :: (Number, Number) -> Pair
const range = (x, y) => x < y ? pair(x)(range(x + 1, y)) : null;

const zeroToFour = range(0, 5);
//-> pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(null)))))
```

Επίσης, όπως αναφέραμε και στη δομή των **cons**, η αναδρομική φύση που διέπει τη δομή pair καθώς η ιδιότητα του δεξιά προσεταιρισμού (**right-associative**) που διατηρεί επιτρέπει την αξιοποίηση της συνάρτησης **foldr** για τη δυναμική δημιουργία της. Επειδή η συνάρτηση **foldr** έχει ταυτιστεί με λειτουργικότητα που σχετίζεται με τη δομή ενός πίνακα, παρόλο που κάτι τέτοιο δεν ισχύει καθώς μπορούμε να τη χρησιμοποιήσουμε για οποιαδήποτε δομή που ακολουθεί τη διεπαφή **iterable,** θα δούμε αρχικά μια υλοποίηση που μετασχηματίζει τη δομή ενός πίνακα σε μία δομή τύπου **pair**, όπως ακριβώς δηλαδή έγινε και στην περίπτωση της συνάρτησης **consFrom** . Μπορεί να γραφεί λοιπόν

```
// foldr :: ((a, b) -> b, b, [a]) -> b
const foldr = (f, z, [first, ...rest]) => first === undefined ? z :
f(first, foldr(f, z, rest));

// range :: [a] -> Pair a
const range = arr => foldr((x, y) => pair(x)(y), null, arr);

const zeroToFour = range([0, 1, 2, 3, 4]);
//-> pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(null)))))

car(cdr(cdr(cdr(cdr(zeroToFour)))));
//-> 4
```

ή αν θέλουμε να διατηρήσουμε την αρχική λειτουργικότητα της **range** δίνοντας δηλαδή τα όρια για το εύρος αριθμών το οποίο θέλουμε να περιέχονται στη διασυνδεμένη λίστα τύπου **pair,** τότε πρέπει να τροποποιήσουμε ελαφρός τη συνάρτηση **foldr** όπως φαίνεται παρακάτω

```
// foldr :: ((a, b) -> b, b, (Number, Number) -> Boolean, Number, Number) -> b
const foldr = (f, z, pred, x, y) => pred(x, y) ? f(x, foldr(f, z, pred, x + 1, y)) : z;
// range :: (Number, Number) -> Pair Number
const range = (x, y) => foldr((x, y) => pair(x)(y), null, (x, y) => x < y, x, y);

const zeroToFour = range(0, 5);
//-> pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(null)))))
car(cdr(cdr(cdr(cdr(zeroToFour)))));
//-> 4
```

Και αν θέλουμε να δώσουμε λύση σε **point-free** μορφή μπορούμε να χρησιμοποιήσουμε την τεχνική **partial currying** και να ευθυγραμμίσουμε τις παραμέτρους της **range** με αυτή της **foldr**. Το τελευταίο σχήμα μπορεί να πάρει τη μορφή

```
// foldr :: ((a, b) -> b, b, (Number, Number) -> Boolean, Number, Number) -> b
const foldr = (f, z, pred, x, y) => pred(x, y) ? f(x, foldr(f, z, pred, x + 1, y)) : z;
// range :: (Number, Number) -> Pair Number
const range = pcurryN(foldr)((x, y) => pair(x)(y), null, (x, y) => x < y);

const zeroToFour = range(0, 5);
//-> pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(null)))))
car(cdr(cdr(cdr(cdr(zeroToFour)))));
//-> 4
```

Προτού προχωρήσουμε παρακάτω, πρέπει να τονίσουμε δύο βασικά σημεία που θα μας επιτρέψουν να συλλάβουμε τη μεγάλη εικόνα σχετικά με το τι προσπαθούμε να αποδείξουμε. Το πρώτο έχει να κάνει με την ανάγκη των προγραμματιστών να ομαδοποιούν ένα πλήθος από δεδομένα σε μία οργανωμένη διάταξη, η μορφή και οι λειτουργικότητες της οποίας καθορίζονται ανάλογα με το εκάστοτε πρόβλημα που καλείται να αντιμετωπίσει. Ο συναρτησιακός προγραμματισμός δεν απαλείφει την ανάγκη αυτή, αλλά αντιθέτως επιτρέπει τη χρήση τέτοιων δομών ενθαρρύνοντας παράλληλα την αντιμετώπιση τους με έναν τρόπο διαφορετικό. Είδαμε για παράδειγμα πως πολλές από τις δομές προκύπτουν από την πράξη της σύνθεσης με τον εαυτό τους, πράγμα που η συναρτησιακή προσέγγιση ερμηνεύει εγγενώς τη χρήση της αναδρομής για την αντιμετώπιση τους. Το δεύτερο έχει να κάνει με την ευελιξία που προσφέρει η συναρτησιακή σχεδίαση κατά την αξιοποίηση τέτοιων δομών. Όπως θα δούμε και αργότερα οι δομές αυτές αντιμετωπίζονται ως συναρτήτες (**Functors**) που απεικονίζουν (**map**) ένα αντικείμενο κάποιας κατηγορίας, σε ένα άλλο μίας άλλης. Πρακτικά επιτρέπουν την αλλαγή του περιβάλλοντος με βάση το οποίο υφίσταται ένα αντικείμενο διατηρώντας έτσι την αγνότητα (pureness) που επιζητά τον συναρτησιακό μοντέλο σχεδίασης. Ένα τέτοιο παράδειγμα αποτελούν και οι πίνακες της JavaScript. Πέρα από αυτό η χρήση συναρτησιακών δομών, επιτρέπει στον προγραμματιστή την αξιοποίηση γνωστών

συναρτήσεων υψηλής τάξης (**curry**, **compose**, **pipe**) καθώς και όπως θα δούμε τη δημιουργία νέων στα πρότυπα που χαράσσει η συναρτησιακή προσέγγιση. Πάμε να δούμε την ευκολία με την οποία μπορούμε να εμπλουτίσουμε τις λειτουργικότητες της διασυνδεμένης λίστας τύπου **pair.**

Η διασυνδεμένη λίστα όπως και ένας πίνακας μπορεί να προσπελαστεί (υπακούει στο **iterable** interface) και ως εκ τούτου μπορούν να υλοποιηθούν λειτουργικότητες υψηλής τάξης όπως **mapping**, **filtering, folding** κτλ. Επίσης δύο λίστες μπορούν να συντεθούν (**concat**) παράγοντας μία νέα που περιέχει στοιχεία των επιμέρους. Για την πράξη της συνάθροισης μπορούμε να γράψουμε

```
// prepend :: (a, Pair a) -> Pair a
const prepend = (x, p) => pair(x)(p);
// concat :: (Pair a, Pair a) -> Pair a
const concat = (p1, p2) => p1 === null ? p2 : prepend(car(p1),
concat(cdr(p1), p2));

const zeroToFour = range(0, 5);
const fiveToNine = range(5, 10);

concat(zeroToFour, fiveToNine);
//-> pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(...pair(9)(null))))))
```

Η συνάρτηση **prepend** δημιουργεί το ζεύγος **pair** καλώντας την ομώνυμη συνάρτηση. Μπορούσε να χρησιμοποιηθεί και στην κλήση της **foldr** ως παράμετρος της **f** για τη δημιουργία της **range.** Πάμε να δούμε πως μπορούμε να υλοποιήσουμε μία αντίστοιχη **map** με αυτή των πινάκων στην JavaScript. Έχουμε

```
// prepend :: (a, Pair a) -> Pair a
const prepend = (x, p) => pair(x)(p);
//map :: (a -> b, Pair a) -> Pair b
const map = (f, p) => p === null ? null : prepend(f(car(p)), map(f,
cdr(p)));

const zeroToFour = range(0, 5);

map(x => x + 10, zeroToFour);
//-> pair(10)(pair(11)(pair(12)(pair(13)(pair(14)(null)))))
```

Και φυσικά η map αυτή δεν διαφέρει σε τίποτα από τη γνωστή map που μεταχειρίζεται τους πίνακες. Διατηρεί αυστηρά τις ιδιότητες της σύνθεσης αφού όπως θα δούμε η Pair είναι συναρτήτης. Μπορούμε να γράψουμε δηλαδή

```
const zeroToFour = range(0, 5);

map(x => x + 10, map(x => x * 2, zeroToFour));
//-> pair(10)(pair(12)(pair(14)(pair(16)(pair(18)(null)))))
```

Η **filter** ορίζεται με τον ίδιο τρόπο και φαίνεται παρακάτω

```
// prepend :: (a, Pair a) -> Pair a
const prepend = (x, p) => pair(x)(p);
// filter :: (a -> Boolean, Pair a) -> Pair a
const filter = (pred, p) => p === null ? null : pred(car(p)) ?
prepend(car(p), filter(pred, cdr(p))) : filter(pred, cdr(p));

const zeroToNine = range(0, 10);

filter(x => x % 2 === 0, zeroToNine);
//-> pair(0)(pair(2)(pair(4)(pair(6)(pair(8)(null)))))
```

Δέχεται ως είσοδο μία συνάρτηση Predicate (a -> Boolean), μία διασυνδεμένη λίστα μορφής pair και επιστρέφει μία νέα διασυνδεμένη λίστα με όλα τα στοιχεία τύπου a για τα οποία ισχύει (Predicate(a) = True). Επιπρόσθετα της **map** και **filter** μπορούμε να υλοποιήσουμε και τις συναρτήσεις **foldl** και **foldr.** Με βάση τους ορισμούς των δύο συναρτήσεων στην **Haskell** για τη δομή των πινάκων

```
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Η μετάφραση των υλοποιήσεων είναι αρκετά απλή υπόθεση στην JavaScript καθώς η δομή **pair** μοιάζει πολύ με αυτή των πινάκων, καθώς είναι σειριακή, και διαθέτει κεφάλι **(head / x / car)** και ουρά **(tail / xs / cdr).** Για την ακρίβεια όπως προαναφέραμε η δομή της λίστας στην Haskell υλοποιείται εσωτερικά με μία συναρτησιακή αναπαράσταση (λόγω της γλώσσας) που μοιάζει με αυτή της **pair.** Έχουμε λοιπόν

```
// foldl :: ((b, a) -> b, b, [a]) -> b
const foldl = (f, z, p) => p === null ? z : foldl(f, f(z, car(p)), cdr(p));
```

```
// foldr :: ((a, b) -> b, b, [a]) -> b
const foldr = (f, z, p) => p === null ? z : f(car(p), foldr(f, z, cdr(p)));
```

Αν θέλουμε λοιπόν να βρούμε το άθροισμα μίας διασυνδεμένης λίστας που περιέχει τους αριθμούς από το 1 έως το 100, θα έχουμε

```
// add :: (Number, Number) -> Number
const add = (x, y) => x + y;
// sum1 :: Pair Number -> Number
const sum1 = p => foldl(add, 0, p);
// sum2 :: Pair Number -> Number
const sum2 = p => foldr(add, 0, p);

const oneToHundred = range(1, 101);

sum1(oneToHundred);
//-> 5050
sum2(oneToHundred);
//-> 5050
```

Η πράξη της άθροισης, όπως βλέπουμε, υπακούει στην προσεταιριστική ιδιότητα και ως εκ τούτου υλοποιείται ομοιόμορφα και με τις δύο υλοποιήσεις **folding.** Επιπρόσθετα είναι άξιο επισήμανσης το γεγονός ότι η **foldl** λειτουργικότητα προσομοιώνει υλοποιήσεις που είναι αριστερά προσεταιριστικές (left-associative) ενώ η **foldr** δεξιά προσεταιριστικές (right-associative). Το αποτέλεσμα αυτής της πρότασης επιτρέπει στην πρώτη υλοποίηση (**foldl**) να μπορεί να σχεδιαστεί σε μία αναδρομική δομή που είναι **tail-case optimized** (συναρτήσεις δηλαδή που μπορούν να ξεπεράσουν τον περιορισμό στις εγγραφές τις στοίβας, αντιγράφοντας το περιβάλλον μίας κλήσης της στην αμέσως επόμενη). Η δεύτερη υλοποίηση (**foldr**), η οποία μάλιστα σκιαγραφεί την υλοποίηση της δομής **pair,** λόγω της φύσης της δεν διέπεται από την ιδιότητα αυτή, αλλά το γεγονός ότι οι αναδρομικές κλήσεις εμφωλεύονται προς τα δεξία επιτρέπει την καθυστέρηση της αποτίμησης (**Lazy evaluation**) της δεξιά έκφρασης σε χρόνο μελλοντικό, δημιουργώντας έτσι μία προσομοίωση της στρατηγικής αποτίμησης (**evaluation strategy**) γνωστή και ως **call-by-name**. Όπως αναφέραμε στο εισαγωγικό κεφάλαιο η JavaScript ακολουθεί το μοντέλο αποτίμησης **applicative order** και συγκεκριμένα τη μέθοδο **call-by-value**, στο οποίο οι εκφράσεις που ορίζονται ως

παράμετροι σε μία συνάρτηση αποτιμώνται απευθείας, από αριστερά προς τα δεξιά. Θα δούμε σε λίγο λοιπόν πως μπορούμε με τη βοήθεια των συναρτήσεων να χρησιμοποιήσουμε την τεχνική call-by-name, και θα ανακαλύψουμε τα οφέλη που μπορούν να προκύψουν από αυτήν.

Τέλος πάμε να δούμε πως μπορούμε να υλοποιήσουμε τις συναρτήσεις **map** και **filter** με τη βοήθεια της **foldr** και παράλληλα αξιοποιώντας τη συνάρτηση υψηλής τάξης **curry.**

```
// prepend :: (a, Pair a) -> Pair a
const prepend = (x, p) => pair(x)(p);

//map :: (a -> b, Pair a) -> Pair b
const map = (f, p) => foldr((x, y) => prepend(f(x), y), null, p);

// filter :: (a -> Boolean, Pair a) -> Pair a
const filter = (f, p) => foldr((x, y) => f(x) ? prepend(x, y) : y, null, p);
```

Βλέπουμε λοιπόν ότι με τη βοήθεια της λειτουργικότητας **folding** μπορούν να υλοποιηθούν όλες οι βοηθητικές συναρτήσεις υψηλής τάξης, ακριβώς με τον ίδιο τρόπο που η συνάρτηση **reduce** (ένα είδος folding) στην JavaScript χρησιμοποιείται για την υλοποίηση των ίδιων δομών στους πίνακες. Και προσθέτοντας τη λειτουργικότητα της **curryN** έχουμε

```
const curryN = fn => {
    const curried = (...args) => x => (args.length + 1 < fn.length ? curried :
fn)(...args, x);
    return curried();
};

const composeN = (f, ...fns) => {
    const composeTwo = (g, f) => x => g(f(x));
    return f === undefined ? x => x : composeTwo(f, composeN(...fns));
}

// prepend :: (a, Pair a) -> Pair a
const prepend = (x, p) => pair(x)(p);

//map :: (a -> b) -> Pair a -> Pair b
const map = curryN((f, p) => foldrr((x, y) => prepend(f(x), y), null, p));

// filter :: (a -> Boolean) -> Pair a -> Pair a
const filter = curryN((f, p) => foldrr((x, y) => f(x) ? prepend(x, y) : y, null,
p));

// mul21 :: Number -> Number
const mul21 = x => x * 21;
// selectEven :: Number -> Boolean
const selectEven = x => x % 2 === 0;
// mul21AndEven :: Pair Number -> Pair Number
const mul21AndEven = composeN(filter(selectEven), map(mul21));

mul21AndEven(range(0, 21));
//->
pair(0)(pair(42)(pair(84)(pair(126)(pair(168)(pair(210)(...pair(420)(null)))))))
```

Παρατηρούμε ότι είμαστε σε θέση να εμπλουτίσουμε τις λειτουργικότητες **map**, **filter** παρέχοντας τις συναρτήσεις ως παραμέτρους στη συνάρτηση υψηλής τάξης **curryN.** Όπως είδαμε και πιο πάνω αυτό μας επιτρέπει να χρησιμοποιήσουμε τις συγκεκριμένες συναρτήσεις επιτελώντας την πράξης της εφαρμογής (**application**) σταδιακά ακριβώς όπως συμβαίνει δηλαδή και με τις αφαιρέσεις λάμδα. Έτσι εφαρμόζοντας στις συναρτήσεις τις λειτουργικότητες **mul21** και **selectEven**, δημιουργούμε νέες συναρτήσεις που αναμένουν μία δομή τύπου **pair** στην είσοδο τους, για να εξάγουν τελικώς την αποτίμηση της λειτουργίας τους. Αυτό μας επιτρέπει να συνθέσουμε τις επιμέρους λειτουργικότητες σε μία, αξιοποιώντας τη γνωστή συνάρτηση **composeN,** υλοποιώντας έτσι πετυχημένα ένα νέο σύστημα (**mul21AndEven**), που δέχεται μία διασυνδεμένη λίστα **Pair Number**,

πολλαπλασιάζει τα στοιχεία της με τον αριθμό 21, απαλείφει τα περιττά στοιχεία, και επιστρέφει το τελικό αποτέλεσμα (μία δομή Pair Number) στην έξοδο του.

4.10.2 Δέντρα trees

Προτού εμβαθύνουμε στην τεχνική της καθυστερημένης αποτίμησης (Lazy evaluation) και δούμε πως με τη βοήθεια της μπορούμε να κατασκευάσουμε μία ροή δεδομένων (Stream), πάμε να δούμε και μία άλλη γνωστή δομή η υλοποίηση της οποίας όμως ακολουθεί συναρτησιακή προσέγγιση. Πρόκειται για τα γνωστά δυαδικά δέντρα (**Binary Trees**) που στοιβάζουν τα δεδομένα τους σε μία δενδροειδή διάταξη, και ανάλογα με τη μορφή που έχουν και τις προϋποθέσεις που ικανοποιούν, ευνοούν συγκεκριμένες λειτουργικότητες. Τα δέντρα αυτά απαρτίζονται από κόμβους και δείκτες που υλοποιούν μία διασύνδεση στην οποία κάθε κόμβος (γονέας/parent) δείχνει σε άλλους δύο (παιδία/children). Η τριγωνική αυτή διάταξη αποκτά επαγωγική υπόσταση με καθένα από τα παιδιά να γίνεται γονέας και να δείχνει σε άλλους δύο κόμβους (παιδιά), χτίζοντας έτσι πετυχημένα μία δομή δέντρου η οποία εξ' ορισμού απαρτίζεται από δομές του εαυτού της. Η τελευταία πρόταση επιτρέπει την προσέγγιση των δομών αυτών (όπως και της λίστας **pair**) με αναδρομικές διαδικασίες οι οποίες όπως είδαμε αποκτάνε πλεονεκτήματα ερμηνείας και ευελιξίας σε ένα συναρτησιακό μοντέλο γραφής.

Η δομή του δυαδικού δέντρου, όπως προαναφέραμε, έχει πολλές διαφορετικές θεωρητικές προσεγγίσεις. Για παράδειγμα μία γνωστή εκδοχή είναι αυτή του δυαδικού δέντρου αναζήτησης (**Binary Search Tree**) στην οποία τα δεδομένα που εμπεριέχονται ακολουθούν τον τύπο **Ord,** είναι δηλαδή συγκρίσιμα, και η τοποθέτηση των στοιχείων έχει μια λογική στην οποία τα μικρότερα στοιχεία τοποθετούνται σε αριστερή κατεύθυνση ενώ τα μεγαλύτερα στη δεξιά. Αυτό έχει ως αποτέλεσμα η πράξη της αναζήτησης ενός στοιχείου να αποκτά λογαριθμική πολυπλοκότητα, επιτρέποντας την να εφαρμόζεται σε τεράστιο αριθμό στοιχείων. Μία άλλη προσέγγιση είναι αυτή του δέντρου (**Binary Heap**) που επιτρέπει την αποδοτική ταξινόμηση των στοιχείων που ακολουθούν τον τύπο **Ord**.

Όπως είπαμε σκοπός της ενότητας είναι η ανάδειξη της συναρτησιακής υλοποίησης μία τέτοιας δομής και ως εκ τούτου θα δώσουμε έμφαση αρχικά στη δημιουργία ενός κλασσικού ισορροπημένου (**balanced**) δυαδικού δέντρου, και στη συνέχεια θα δούμε μία παραλλαγή του φτιάχνοντας ένα δυαδικό δέντρο αναζήτησης.

Ο ορισμός ενός κόμβου σε ένα δυαδικό συναρτησιακό δέντρο μπορεί να πάρει την παρακάτω μορφή έκφρασης στον λογισμό λάμδα

```
tree = λx.λl.λr.λs.λe.(s x l r) | empty = λs.λe.(e ())
```

Η παράμετρος **x** συμβολίζει την τιμή που έχει αποθηκευμένη ο κόμβος, η παράμετρος **l** δέχεται το αριστερό κλαδί (παιδί-δέντρο) στο οποίο είναι γονέας αυτός ο κόμβος, η παράμετρος **r** απεικονίζει το δεξί κλαδί (παιδί-δέντρο) στο οποίο είναι γονέας πάλι αυτός ο κόμβος, και τέλος η παράμετρος **s** είναι η διεπαφή προσπέλασης των εσωτερικών στοιχείων του κόμβου μέσω μίας έκφρασης λάμδα. Η κάθετη μπάρα συμβολίζει την ενναλακτική τιμή που μπορεί ένας κόμβος (**empty**) για να σηματοδοτήσει το πέρας της επιμέρους δομής δηλώνοντας ανυπαρξία. Μπορούμε να το προσομοιώσουμε ως το ουδέτερο στοιχείο της δομής για να γίνει ευκολότερα κατανοητό. Στην πραγματικότητα ο ορισμός της δομής κατά αυτόν τον τρόπο μας επιτρέπει τη δημιουργίας ενός αφαιρετικού επιπέδου, καθιστώντας εφικτή τη χρήση της δομής της επιλογής (**if**), εκμεταλλευόμενοι τη σειρά των παραμέτρων που δέχεται για εφαρμογή ένα κόμβος τύπου **tree** ή **empty.** Έτσι μπορούμε να ορίσουμε ένα δέντρο με την παρακάτω έκφραση.

```
myTree =
= tree 10 empty empty =
= (((λx.λl.λr.λs.λe.(s x l r) 10) empty) empty) =
= λs.λe.(s 10 empty empty)
```

Πρόκειται για έναν μοναδικό κόμβο ο οποίος αποθηκεύει την τιμή 10, και ορίζει αριστερό και δεξί παιδί την έκφραση **empty** για να δηλώσει ανυπαρξία. Για να εξάγουμε την τιμή ενός κόμβου μπορούμε να ορίσουμε την παρακάτω έκφραση λάμδα

```
val = λt.(t (λx.λl.λr.x))
```

Και συνεπώς μπορούμε να δούμε ότι

```
val myTree = λt.(t (λx.λl.λr.x))  λs.λe.(s 10 empty empty) =
= λs.λe.(s 10 empty empty) (λx.λl.λr.x) =
= λe.((((λx.λl.λr.x) 10) empty) empty)
```

Παρατηρούμε ότι επιστρέφεται μία έκφραση η οποία κατόπιν εφαρμογής της με οποιαδήποτε παράμετρο επιστρέφει

```
λe.((((λx.λl.λr.x) 10) empty) empty) () =
= ((((λx.λl.λr.x) 10) empty) empty) =
= 10
```

Κρατάμε λίγο τη συμπεριφορά αυτή στο μυαλό μας και πάμε παρακάτω. Τι θα γίνει στην περίπτωση που η συνάρτηση **val** δεχτεί ως είσοδο ένα κόμβο **empty.** Η απάντηση είναι ότι επιστρέφει μία συνάρτηση της μορφής

```
λe.(e ())
```

αφού

```
val empty = λt.(t (λx.λl.λr.x))  λs.λe.(e ()) =
= λs.λe.(e ()) (λx.λl.λr.x) =
= λe.(e ())
```

Αυτό πρακτικά σημαίνει ότι μπορούμε να επαναπροσδιορίσουμε τη συνάρτηση **val** εκμεταλλευόμενοι τον ορισμό της λάμδα αφαίρεσης **empty,** και είμαστε πλέον σε θέση να εκτελέσουμε μία νέα λειτουργικότητα στις περιπτώσεις που το όρισμα είναι ένας κενός κόμβος, όπως επίσης επιτυγχάνουμε και την εξαγωγή του αποθηκευμένου στοιχείου στις περιπτώσεις που έχουμε μία δομή **tree**. Η σωστή **val**, αλλά και κάθε λειτουργικότητα που θέλουμε να διαχειρίζεται κόμβους τύπου **empty | tree**, πρέπει να δέχεται και μία δεύτερη αφαίρεση λάμδα στον ορισμό της, όπως φαίνεται παρακάτω

```
val = λt.(t (λx.λl.λr.x) λx.null)
```

Με τη νέα έκφραση **val** έχουμε

```
val myTree = λt.(t (λx.λl.λr.x) λx.null)  λs.λe.(s 10 empty empty) =
= λs.λe.(s 10 empty empty) (λx.λl.λr.x) (λx.null) =
= λe.((((λx.λl.λr.x) 10) empty) empty) (λx.null)=
= ((((λx.λl.λr.x) 10) empty) empty) =
= 10
```

Όπως επίσης και

```
val empty = λt.(t (λx.λl.λr.x) λx.null)  λs.λe.(e ()) =
= λs.λe.(e ()) (λx.λl.λr.x) (λx.null)=
= (λx.null) () =
= null
```

Τέλος, όπως αναφέραμε, η φύση της συναρτησιακής αυτή δομής είναι αναδρομική αφού κάθε δέντρο αποτελείται από κόμβους διασυνδεμένους σε τριγωνική διάταξη σχηματίζοντας σχέσεις γονέα-παιδιών. Για τη δημιουργία ενός τέτοιου τριγώνου, όπου για παράδειγμα έχουμε ως γονέα ένα κόμβο με τον αριθμό 10, και ως παιδιά δύο κόμβους με τους αριθμούς 5 και 15 αντίστοιχα, αρκεί να γράψουμε την έκφραση

```
myTree = tree 10 (tree 5 empty empty) (tree 15 empty empty)
```

Και φυσικά η λογική αυτή επεκτείνεται βάση επαγωγής δικαιολογώντας έτσι την ύπαρξη μίας αναδρομικής συναρτησιακής δομής

Πάμε να δούμε πως μπορούμε να μεταφέρουμε τα παραπάνω από το πεδίο του λογισμού λάμδα, στο πρακτικό περιβάλλον της γλώσσας προγραμματισμού που ασχολείται η παρούσα διπλωματική, δηλαδή την JavaScript.

Η δομή **tree** και **empty** μπορούν να γραφούν με τον παρακάτω τρόπο

```
// tree :: a -> Tree a -> Tree a -> (a, Tree, Tree) -> a
const tree = value => treeLeft => treeRight => (s, _) => s(value,
treeLeft, treeRight);

// emtpy :: (() -> a) -> a
const empty = (_, e) => e();
```

Βλέπουμε ότι για λόγους απλότητας αποφεύγουμε να χρησιμοποιήσουμε τις **curried** εκδοχές των δύο συναρτήσεων καταστρατηγώντας σε έναν βαθμό την ακριβή αντιστοιχία τους με τον λογισμό λάμδα. Παρά ταύτα αφού μας δίνεται η δυνατότητα στην JavaScript να ορίσουμε συναρτήσεις με πλήθος παραμέτρων περισσότερων του ενός, τότε στην προκειμένη περίπτωση μπορούμε να το πράξουμε, καθώς η συμπεριφορά και η λειτουργικότητα της δομής παραμένει ίδια. Έτσι για την κατασκευή ενός δέντρου με γονέα, κόμβο με τον αριθμό 10, και αριστερό και δεξί παιδί τους κόμβους 5 και 15 αντίστοιχα, θα είχαμε

```
const myTree = tree(10)(tree(5)(empty)(empty))(tree(15)(empty)(empty));
```

Ωραία, πάμε να δούμε πως θα φτιάχναμε ορισμένες βοηθητικές συναρτήσεις που θα μας διευκολύνουν με την απεικόνιση της δομής στη συσκευή εξόδου, καθότι η δομή όπως είδαμε είναι μία αφαίρεση λάμδα (συνάρτηση) και συνεπώς η παρακάτω έκφραση

```
console.log(myTree);
```

επιστρέφει την τιμή

```
//-> [Function]
```

Προτού δημιουργήσουμε τη συνάρτηση εκτύπωσης θα πρέπει να σκεφτούμε ποια ενναλακτική δομή της JavaScript θα ήταν κατάλληλη για μετατροπή της συναρτησιακής δενδροειδής μορφής σε αυτήν και φυσικά να έχει δυνατότητες αποτελεσματικής απεικόνισης σε αλφαριθμητικό τύπο. Τα στοιχεία αυτά τα συναντάμε σαφώς στην εγγενή δομή των αντικείμενων (object) της JavaScript, τα οποία μάλιστα μπορούν να αποτυπωθούν στην αλφαριθμητική τους διάσταση με αρκετή ευκολία μέσω του πρωτοκόλλου **JSON** (JavaScript Object Notation).

Αρχικά πάνε να δούμε πως θα υλοποιήσουμε τη συνάρτηση μετατροπής της δομής του δέντρου σε ένα JavaScript αντικείμενο ({}). Παράλληλα με τη λειτουργία αυτή θα αντιληφθούμε καλύτερα και τον τρόπο με τον οποίο επεμβαίνουμε εσωτερικά στη δομή που όπως είδαμε γίνεται μέσω εισόδου συναρτήσεων, αλλά και τη δυνατότητα προσπέλασης **empty** κόμβων χρησιμοποιώντας τη δυνατότητα των συναρτήσεων και αποφεύγοντας τη χρήση συνθηκών **if** στον κώδικα. Έστω η συνάρτηση

```
const objToTree = t => ?
```

Το σίγουρο είναι ότι θα δέχεται μία δομή στην είσοδο της, και θα εξάγει τη δομή σε μορφή αντικειμένου. Όποτε μία προσέγγιση θα ήταν να είχαμε μία συνάρτηση της μορφής

```
const objToTree = t => ({})
```

Παρόλα αυτά, η επιστροφή του αντικείμενου που βάζουμε ως στόχο αποδεικνύεται ότι μπορεί να πραγματοποιηθεί και εσωτερικά της συνάρτησης προσπέλασης, επιτρέποντας έτσι τη λεπτομερέστερη επέμβαση στη δομή του δέντρου όπως θα δούμε. Για παράδειγμα μπορούμε να γράψουμε

```
const objToTree = t => t((val, left, right) => ({

}));
```

πετυχαίνοντας το ίδιο αποτέλεσμα. Η σύνθεση του αντικειμένου που εξάγεται παίρνει την παρακάτω μορφή

```
const objToTree = t => t((val, left, right) => ({
    value: val,
    leftTree: ?,
    rightTree: ?
}));
```

Ποια έκφραση όμως θα πάρει τη θέση των ερωτηματικών; Εύλογα μπορούμε να αντιληφθούμε ότι θα μπορούσαμε κάλλιστα να καλέσουμε αναδρομικά την **objToTree** για τα επιμέρους δέντρα-παιδιά.

```
const objToTree = t => t((val, left, right) => ({
    value: val,
    leftTree: objToTree(left),
    rightTree: objToTree(right)
}));
```

Η λειτουργικότητα **objToTree** είναι σχεδόν έτοιμη και αυτό γιατί αν αποτιμήσουμε την έκφραση

```
objToTree(myTree);
//-> TypeError: e is not a function
```

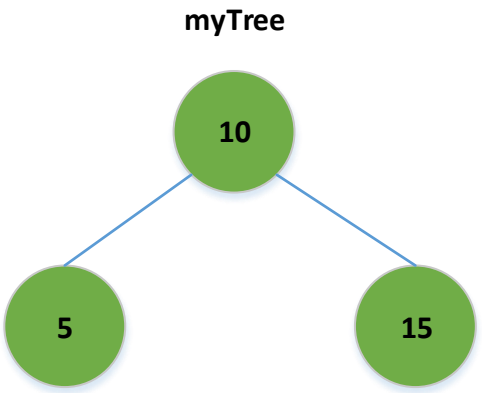
Θα πάρουμε πολύ ορθά σφάλμα καθώς δεν καλύπτουμε τις περιπτώσεις που η τιμή **t** της **ojbToTree** είναι τύπου **empty** και όχι **tree.** Για να καλύψουμε και αυτές τις περιπτώσεις μπορούμε πολύ απλά, και εδώ φαίνεται η απήχηση που προσφέρει η συναρτησιακή φύσης της δομής, να εισάγουμε και μία επιπλέον λειτουργικότητα που θα αναλαμβάνει να προσπελάσει τις τιμές **empty**, δίνοντας ταυτόχρονα πέρας στην αναδρομή και στην υλοποίηση του τελικού αντικειμένου. Μπορούμε τελικώς να γράψουμε

```
const myTree = tree(10)(tree(5)(empty)(empty))(tree(15)(empty)(empty));

// ojbToTree :: Tree a -> {a}
const objToTree = t => t((val, left, right) => ({
    value: val,
    leftTree: objToTree(left),
    rightTree: objToTree(right)
}), () => "empty");

const myTreeInObj = objToTree(myTree);

// logger :: Tree a -> ()
const logger = t => console.log(JSON.stringify(t, null, 2));

logger(myTreeInObj);
//->
{
  "value": 10,
  "leftTree": {
    "value": 5,
    "leftTree": "empty",
    "rightTree": "empty"
  },
  "rightTree": {
    "value": 15,
    "leftTree": "empty",
    "rightTree": "empty"
  }
}
```

Παρατηρούμε λοιπόν ότι έχουμε μία πολύ καλή απεικόνιση της δομής του δέντρου, χάρη στη μετατροπή του σε αντικείμενα τύπου **{value, leftTree, rightTree}.**

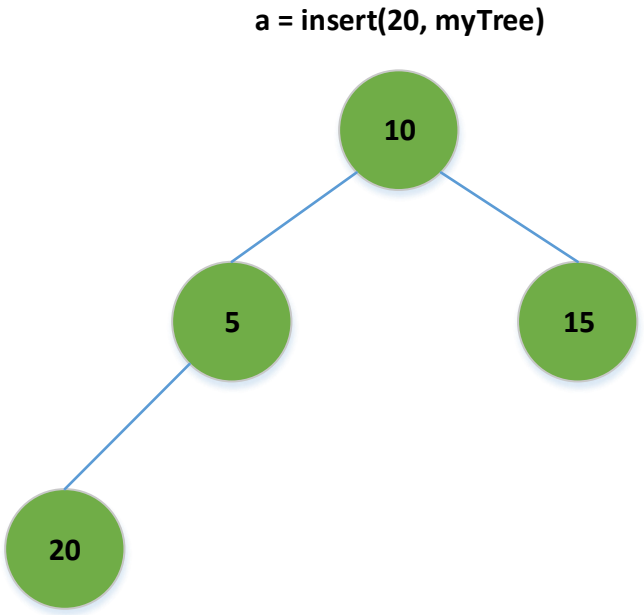
Αφού είδαμε λοιπόν πως μπορούμε να δούμε αναπαραστατικά για λόγους οπτικής ανάδρασης ένα συναρτησιακό δέντρο, πάμε να υλοποιήσουμε συναρτήσεις που μας επιτρέπουν τη δυναμική δημιουργία τους. Αρχικά όπως είπαμε, θα αναλύσουμε τη δημιουργία ενός κλασσικού ισορροπημένου δυαδικού δέντρου όπου τα στοιχεία εισέρχονται τηρώντας την προϋπόθεση της ισοδυναμίας των επιμέρους μεγεθών. Αυτό πρακτικά σημαίνει ότι κάθε δέντρο που έχει ως ρίζα το αριστερό κλαδί ενός κόμβου, δεν διαφέρει στο σύνολο του πλήθους των κόμβων που διαθέτει συνολικά, περισσότερο της μονάδας από το αντίστοιχο δέντρο που έχει ως ρίζα το δεξί κλαδί του ίδιου γονέα.

Έστω ότι έχουμε τη λειτουργικότητα **insert** η οποία δέχεται μία τιμή και ένα δέντρο και επιστρέφει ένα νέο δέντρο στο οποίο ο νέος κόμβος είναι ενσωματωμένος με τον τρόπο που προαναφέραμε. Για την καλύτερη κατανόηση παρατίθεται η παρακάτω ακολουθία σχημάτων.

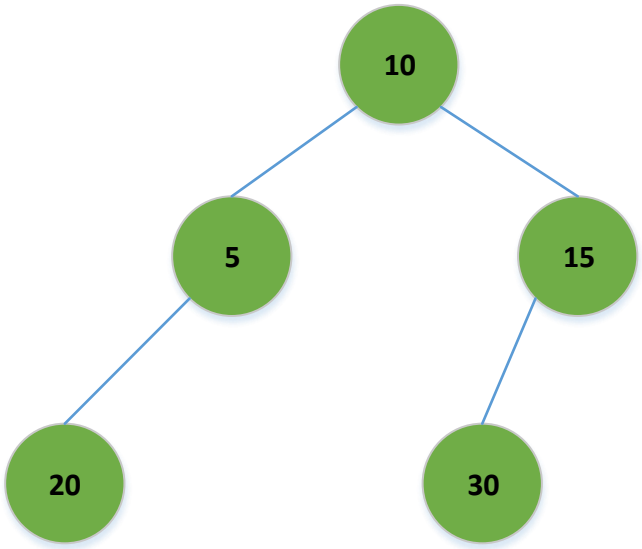


Η έκφραση **myTree** όπως είδαμε και από την κλήση **logger(myTreeInObj)** μπορεί να αναπαρασταθεί με τον παραπάνω τρόπο. Τότε για τις τέσσερις (4) επόμενες διαδοχικές κλήσεις της **insert** με την παρακάτω ακολουθία, τα στιγμιότυπα των δέντρων θα είχαν αυτή τη μορφή.

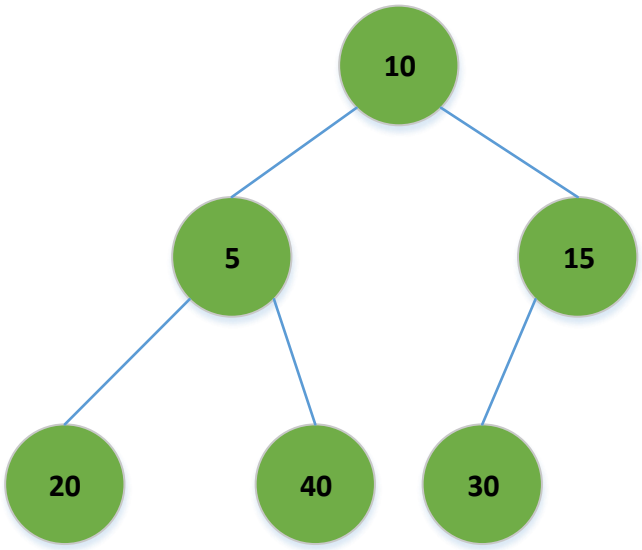
```
const a = insert(20, myTree);
const b = insert(30, a);
const c = insert(40, b);
const d = insert(50, c);
```

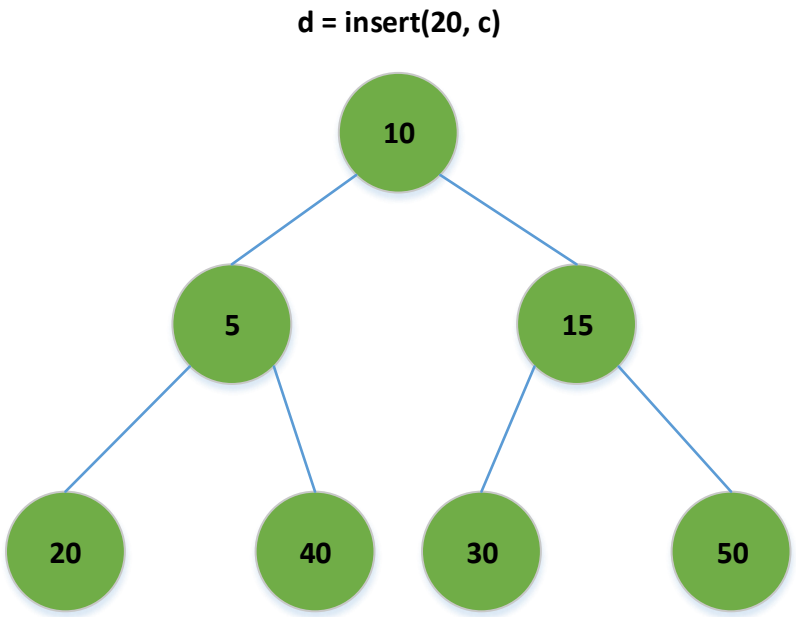


b = insert(20, a)



c = insert(20, b)





Πάμε να δούμε πως θα υλοποιούσαμε την **insert** στη γλώσσα της **JavaScript.** Βοηθητικά θα χρειαστούμε μία συνάρτηση επιστροφής του βάρους ενός δέντρου (πλήθος κόμβων) έτσι ώστε η **insert** να είναι σε θέση να αντιληφθεί την κατεύθυνση που θα ακολουθήσει κατά τη διαδικασία δημιουργίας του νέου δέντρου και εν τέλει εισαγωγής του νέου κόμβου. Για τη συνάρτηση του βάρους μπορούμε να γράψουμε

```
// size :: Tree a -> Number
const size = t => t((value, left, right) => 1 + size(left) + size(right),
() => 0);

size(myTree);
//-> 3
size(d);
//-> 7
```

Η **insert** μπορεί πλέον να γραφεί με την παρακάτω έκφραση

```
// insert :: (a, Tree a) -> Tree a
const insert = (value, t) => t(
    (val, left, right) => size(left) <= size(right) ? tree(val)(insert(value,
left))(right) : tree(val)(left)(insert(value, right)),
    () => tree(value)(empty)(empty)
);
```

Συνεπώς για τη δημιουργία του δέντρου **d** θα έχουμε

```
const a = insert(20, myTree);
const b = insert(30, a);
const c = insert(40, b);
const d = insert(50, c);

logger(objToTree(d));
//->
{
  "value": 10,
  "leftTree": {
    "value": 5,
    "leftTree": {
      "value": 20,
      "leftTree": "empty",
      "rightTree": "empty"
    },
    "rightTree": {
      "value": 40,
      "leftTree": "empty",
      "rightTree": "empty"
    }
  },
  "rightTree": {
    "value": 15,
    "leftTree": {
      "value": 30,
      "leftTree": "empty",
      "rightTree": "empty"
    },
    "rightTree": {
      "value": 50,
      "leftTree": "empty",
      "rightTree": "empty"
    }
  }
}
```

Βλέπουμε λοιπόν ότι η λειτουργικότητα **insert** καταφέρνει τη δημιουργία ενός νέου δέντρου στο οποίο είναι τοποθετημένος ένας νέος κόμβος κάνοντας χρήση της αναδρομής με κατεύθυνση που ορίζει η συνθήκη **size(left) <= size(right)** και βάθος (αριθμός αναδρομικών κλήσεων) όσο είναι το ύψος του δέντρου (ξεκινώντας από τη ρίζα του και καταλήγοντας στον πρώτο κόμβο τύπου **empty**). Πάμε να δούμε ποιες άλλες λειτουργικότητες μπορούν να εφαρμοστούν στην παραπάνω δομή. Η δομή **tree** όπως και η **pair** είναι **mappable** συνεπώς μπορούμε να υλοποιήσουμε μία συνάρτηση υψηλής τάξης

map που δέχεται μία συνάρτηση και μία δομή δέντρου και επιστρέφει ένα νέο δέντρο με στοιχεία που προκύπτουν από εφαρμογή της δοθείσας συνάρτησης επάνω τους.

Μπορούμε να γράψουμε δηλαδή

```
// map :: (a -> b, Tree a) -> Tree b
const map = (f, t) => t(
    (value, left, right) => tree(f(value))(map(f, left))(map(f, right)),
    () => empty
);
```

Διακρίνουμε χαρακτηριστικά την απλότητα της υλοποίησης χάρη στην αναδρομική φύση του προβλήματος που στηρίζεται στη συναρτησιακή προσέγγιση της δομής. Όσο έχουμε κόμβους τύπου **tree,** η **map** επιστρέφει νέους με στοιχεία που εξάγονται από το σύστημα της **f** και τα υποδέντρα που βρίσκονται αριστερά και δεξιά του κόμβου (τα παιδιά δηλαδή) επανακαλούν την **map.** Αν το δέντρο είναι τύπου **empty**, τότε απλά η συνάρτηση επιστρέφει έναν νέο κόμβο **empty** μέσω της έκφρασης **() => empty.** Αν τη χρησιμοποιήσουμε με όρισμα το δέντρο **d** και συνάρτηση εισόδου έναν αθροιστή για παράδειγμα, θα έχουμε

```
const mapped = map(x => x + 10, d);

logger(objToTree(mapped));
//->
{
  "value": 20,
  "leftTree": {
    "value": 15,
    "leftTree": {
      "value": 30,
      "leftTree": "empty",
      "rightTree": "empty"
    },
    "rightTree": {
      "value": 50,
      "leftTree": "empty",
      "rightTree": "empty"
    }
  },
  "rightTree": {
    "value": 25,
    "leftTree": {
      "value": 40,
      "leftTree": "empty",
      "rightTree": "empty"
    },
    "rightTree": {
      "value": 60,
      "leftTree": "empty",
      "rightTree": "empty"
    }
  }
}
```

Παρατηρούμε δηλαδή ότι όλες οι τιμές των κόμβων της **d** έχουν αθροιστεί με τον αριθμό 10, όπως ορίζει η συνάρτηση **x => x + 10** . Τέλος, καθαρά για λόγους εξάσκησης, ας δούμε και τις υλοποιήσεις που μας επιτρέπουν τη μετατροπή μίας δομής πίνακα σε ένα δυαδικό δέντρο. Αυτές μπορούν να υλοποιηθούν με αρκετή ευκολία κάνοντας χρήση της συνάρτησης **insert** στις δομές υψηλής τάξης **foldl** και **foldr** ακριβώς με τον ίδιο τρόπο που είχε γίνει και στην περίπτωση της δομής **pair.** Βέβαια θα συναντήσουμε μία μικρή διαφορά που έχει να κάνει με τον τρόπο που λειτουργεί η **insert** της δομής **tree** και η **prepend** της δομής **pair.** Έχουμε λοιπόν

```
// foldr :: ((a, b) -> b, b, [a]) -> b
const foldr = (f, z, [first, ...rest]) => first === undefined ? z : f(first,
foldr(f, z, rest));

const treeFrom = arr => foldr(insert, empty, arr);

logger(objToTree(treeFrom([1, 2, 3])));
//->
{
  "value": 3,
  "leftTree": {
    "value": 2,
    "leftTree": "empty",
    "rightTree": "empty"
  },
  "rightTree": {
    "value": 1,
    "leftTree": "empty",
    "rightTree": "empty"
  }
}
```

Παρατηρούμε ότι η **treeFrom** μετατρέπει τη δομή του πίνακα [1, 2, 3] σε δυαδικό δέντρο. Η πράξη γίνεται μέσω της γνωστής **foldr** που χρησιμοποιεί τη συνάρτηση **insert** διπλώνοντας πετυχημένα τα στοιχεία του πίνακα σε μία δομή τύπου **tree.** Το γεγονός όμως ότι η πράξη **insert** τοποθετεί το στοιχείο στο τέλος της δομής tree, και όχι στην κορυφή, όπως έκανε η συνάρτηση **prepend** για την υλοποίηση **pair,** έχει ως αποτέλεσμα την τοποθέτηση των στοιχείων ανάποδα (δηλαδή ρίζα του δέντρου γίνεται το τελευταίο στοιχείο του πίνακα). Αν θέλουμε να κρατήσουμε τη σειρά εισαγωγής, τότε έχουμε δύο πιθανές λύσεις για τη νέα treeFrom. Η μία πολύ απλά αντιστρέφει τον πίνακα εισόδου

μέσω μίας συνάρτησης **reverse** και η δεύτερη συνιστά τη χρήση της ενναλακτικής συνάρτησης **foldl.** Συμπερασματικά λοιπόν μπορεί να αποτυπωθεί

```
// foldr :: ((a, b) -> b, b, [a]) -> b
const foldr = (f, z, [first, ...rest]) => first === undefined ? z : f(first,
foldr(f, z, rest));

//reverse :: [a] -> [a]
const reverse = arr => arr.reverse();

const treeFrom = composeN(pcurryN(foldr)(insert)(empty), reverse);

logger(objToTree(treeFrom([1, 2, 3])));
//->
{
  "value": 1,
  "leftTree": {
    "value": 2,
    "leftTree": "empty",
    "rightTree": "empty"
  },
  "rightTree": {
    "value": 3,
    "leftTree": "empty",
    "rightTree": "empty"
  }
}
```

Χρησιμοποιώντας βοηθητικά τις συναρτήσεις **composeN** και **pcurryN** για να γραφεί η **treeFrom** σε **point-free** style. Τέλος ενναλακτικά όπως προαναφέραμε, μπορεί να γίνει χρήση της **foldl** για την εξαγωγή του ίδιου αποτελέσματος όπως απεικονίζεται παρακάτω

```
// foldl :: ((b, a) -> b, b, [a]) -> b
const foldl = (f, z, [first, ...rest]) => first === undefined ? z : foldl(f, f(z,
first), rest);

// flip :: ((a, b) -> c) -> ((b, a) -> c)
const flip = f => (x, y) => f(y, x);

const treeFrom = arr => foldl(flip(insert), empty, arr);

logger(objToTree(treeFrom([1, 2, 3])));
//->
{
  "value": 1,
  "leftTree": {
    "value": 2,
    "leftTree": "empty",
    "rightTree": "empty"
  },
  "rightTree": {
    "value": 3,
    "leftTree": "empty",
    "rightTree": "empty"
  }
}
```

Να προσθέσουμε ότι η συνάρτηση **flip** είναι απαραίτητη για την ευθυγράμμιση των παραμέτρων που δέχεται η **insert** με τις παραμέτρους που αναμένει η συνάρτηση **f** εντός της λειτουργικότητας **foldl.**

Αφού είδαμε την υλοποίηση του ισορροπημένου δέντρου, πάμε να δούμε πόσο εύκολα μπορούμε να υλοποιήσουμε μία άλλη γνωστή εκδοχή της δομής, αυτή του δυαδικού δέντρου αναζήτησης (**Binary Search Tree**). Στην πραγματικότητα το μόνο που αλλάζει είναι η λειτουργικότητα της εισαγωγής που πλέον αντί να συγκρίνει το βάρος των επιμέρους κλαδιών για να επιτευχθεί η εισαγωγή, συγκρίνει την τιμή που πρέπει να εισαχθεί με αυτή του εκάστοτε κόμβου. Αν η τιμή εισαγωγής είναι μικρότερη, τότε η κατεύθυνση αναδρομής είναι αριστερή, αλλιώς δεξιά. Παρακάτω βλέπουμε την παραλλαγμένη **insert**

```
// insert :: (Ord a, Tree a) -> Tree a
const insert = (value, t) => t(
    (val, left, right) => value <= val ? tree(val)(insert(value, left))(right) :
tree(val)(left)(insert(value, right)),
    () => tree(value)(empty)(empty)
);
```

Όπως βλέπουμε τα στοιχεία που εισάγονται στο δυαδικό δέντρο αναζήτησης πρέπει να είναι συγκρίσιμα και να υπακούνε στον τύπο **Ord** ή στην αντικειμενοστρεφή διάλεκτο να υλοποιούν τη διασύνδεση **Comparable**. Για να μην επεκταθούμε περισσότερο, αφού σκοπός είναι η ανάδειξη της δυνατότητας δημιουργίας συναρτησιακών δομών, το παράδειγμα που θα δώσουμε θα είναι και αυτό με αριθμούς όπως και στην περίπτωση του κλασσικού δυαδικού δέντρου που είδαμε παραπάνω. Καλώντας λοιπόν την **treeFrom** που μπορεί να πάρει οποιαδήποτε μορφή από τις προηγούμενες που είδαμε με τη νέα λειτουργικότητα **insert,** αποδεικνύεται ότι

```
logger(objToTree(treeFrom([5, -3, 23, 30, 1, 19, -13])));
//->
{
  "value": 5,
  "leftTree": {
    "value": -3,
    "leftTree": {
      "value": -13,
      "leftTree": "empty",
      "rightTree": "empty"
    },
    "rightTree": {
      "value": 1,
      "leftTree": "empty",
      "rightTree": "empty"
    }
  },
  "rightTree": {
    "value": 23,
    "leftTree": {
      "value": 19,
      "leftTree": "empty",
      "rightTree": "empty"
    },
    "rightTree": {
      "value": 30,
      "leftTree": "empty",
      "rightTree": "empty"
    }
  }
}
```

τα στοιχεία του πίνακα [5, -3, 23, 30, 1, 19, -13] εισάγονται σε μία δομή δυαδικού δέντρου αναζήτησης και όπως παρατηρούμε οι τιμές που βρίσκονται αριστερά, είναι μικρότερες από την τιμή του γονέα τους, ενώ αυτές που είναι δεξιά είναι μεγαλύτερες από την τιμή

του γονέα τους. Η λογική αυτή φυσικά έχει αναδρομική ισχύ με αποτέλεσμα να τηρούνται οι προϋποθέσεις για μία αποδοτική διαδικασία αναζήτησης ικανοποιώντας δηλαδή τον πρωταρχικό σκοπό της δομής. Μπορούμε δηλαδή να δημιουργήσουμε μία συνάρτηση αναζήτησης, όπως απεικονίζεται παρακάτω

```
// find :: (a, Tree a) -> Boolean
const find = (value, t) => t(
    (val, left, right) => value === val || find(value, left) || find(value,
right),
    () => false
);
```

η οποία αναζητά το στοιχείο **value** σε ένα δυαδικό συναρτησιακό δέντρο αναζήτησης, με αλγόριθμο πολυπλοκότητας **O(logn)**, σε αντίθεση με μία σειριακή αναζήτηση που έχει πολυπλοκότητα **O(n).** Έτσι για παράδειγμα η έκφραση

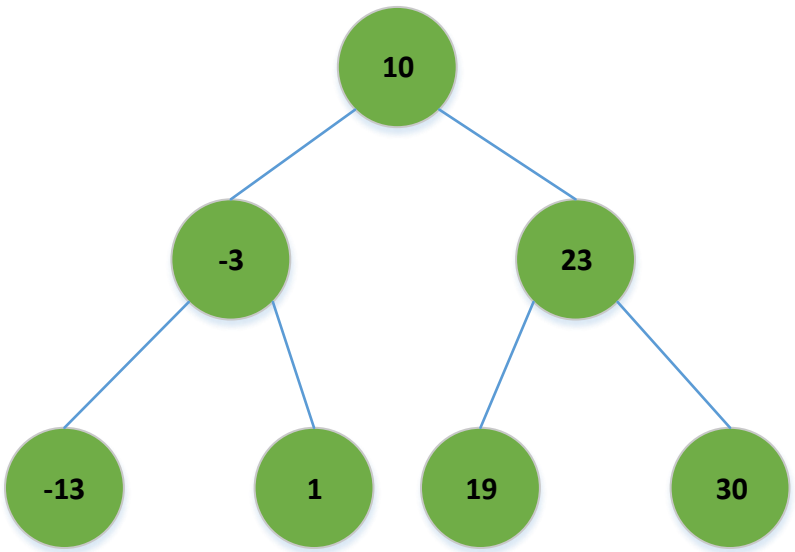
```
find(19, treeFrom([5, -3, 23, 30, 1, 19, -13]));
//-> true
```

είναι αληθής αφού το στοιχείο 19 βρίσκεται στη δομή του δέντρου, ενώ η έκφραση

```
find(-19, treeFrom([5, -3, 23, 30, 1, 19, -13]));
//-> false
```

είναι ψευδής αφού ο αριθμός 19 δεν υπάρχει. Τέλος παρακάτω παραθέτουμε σχηματικά την έκφραση **treeFrom([5, -3, 23, 30, 1, 19, -13])** για οπτική διευκόλυνση του αναγνώστη.

treeFrom([5, -3, 23, 30, 1, 19, -13])



Εν κατακλείδι, πριν κλείσουμε με τον παρόν υπόκεφάλαιο και μπούμε στη διαχείριση των ροών (**Streams**), αξίζουν να σημειωθούν τα εξής συμπεράσματα όσον αφορά στις συναρτησιακές δομές. Το πρώτο έχει να κάνει με τη θέση των δομών αυτών στο μοντέλο του συναρτησιακού προγραμματισμού. Όπως είδαμε, η υλοποίηση των δομών αυτών ήταν αμιγώς συναρτησιακή, αφού η σύνθεση και η δημιουργία τους βασίστηκε αποκλειστικά στη χρήση εκφράσεων λάμδα σε θεωρητικό αλλά και σε πρακτικό επίπεδο, καθιστώντας εφικτό το σχηματισμό της έννοιας της κατάστασης μέσω της μερικής εφαρμογής αυτών των αφαιρέσεων (**closures**). Το γεγονός αυτό, η χρήση δηλαδή αναδρομικών συναρτήσεων για την κατασκευή των δομών αυτών, έχει ως αποτέλεσμα τη συμμόρφωση τους στη βάση που ορίζει ο συναρτησιακός προγραμματισμός. Αυτή, όπως εύλογα πλέον αντιλαμβανόμαστε είναι η τήρηση των κανόνων της αγνότητας (**pureness**) των λειτουργιών σε συνδυασμό με την αποφυγή παρενεργειών (**side effects**), κανόνες στους οποίους ο βαθμός καταστρατήγησης είναι ανάλογος με το πλήθος των δυσχερειών που θα συναντήσει ένας προγραμματιστής στην απόπειρα υλοποίησης ενός προγράμματος κατά τον συναρτησιακό μοντέλο γραφής. Πράγματι, οι δομές τις οποίες αντιμετωπίζει η αμιγώς συναρτησιακή σχεδίαση (όπως είναι και η **pair**/**tree**), εντάσσονται σε ένα ευρύτερο πλαίσιο υπόστασης με την ονομασία "**Persistent Data Structures**". Οι δομές αυτές, έχουν την ιδιότητα της συντηρησιμότητας καθώς οποιαδήποτε ενέργεια τροποποίησης τους έχει ως συνέπεια την όσο τον δυνατόν αποδοτικότερη επαναδημιουργία τους, τηρώντας έτσι την αρχή της αγνότητας και καθιστώντας τις φυσικά κατάλληλες προς χρήση στον συναρτησιακό μοντέλο προγραμματισμού. Για παράδειγμα ας πάρουμε τη δομή **pair**. Η διαδικασία εισαγωγής μέσω της **prepend** στην αρχική θέση αλλά και οποιαδήποτε άλλη πιθανή συνάρτηση εισαγωγής στοιχείου σε συγκεκριμένη θέση της διασυνδεμένης λίστας, γεννά μία καινούργια δομή **pair** η οποία στη συγκεκριμένη περίπτωση βάση υλοποιήσεως συμμερίζεται έξυπνα την ουρά της προηγούμενης. Το γεγονός ότι υπάρχουν κοινά στοιχεία μεταξύ τους δεν δημιουργεί κανένα πρόβλημα από τη στιγμή που οποιαδήποτε ενέργεια τροποποίησης συνοδεύεται από το σχηματισμό νέας δομής προσομοιώνοντας έτσι πετυχημένα ένα μοντέλο **copy-on-write**. Φυσικά το ίδιο πρέπει να ισχύει και στα στοιχεία που αποθηκεύονται εντός της δομής **pair**. Πρέπει να είναι δηλαδή αγνά ούτως ώστε να μην επιτρέπεται η δυνατότητα τροποποίησης τους, πράγμα που στην JavaScript το βάρος

αυτό εναπόκειται στον προγραμματιστή και στο βαθμό προσήλωσης του στις αρχές του συναρτησιακού προγραμματισμού. Για παράδειγμα βλέπουμε ότι

```
const a = prepend({name: "random"}, null);
//-> pair({name: "random"})(null)
const b = prepend("something", a);
//-> pair("something")(pair({name: "random"})(null))

car(cdr(b))
//-> { name: 'random' }
car(cdr(b)).name = "mutated";

a;
//-> pair({name: "mutated"})(null)
b;
//-> pair("something")(pair({name: "mutaded"})(null))
```

η δομή **pair b** δημιουργείται από την εισαγωγή του στοιχείου "**something"** στην κεφαλή της δομής **pair a.** Παρά ταύτα πιθανή τροποποίηση του αντικειμένου που βρίσκεται στην ουρά της δομής **pair b** {name: 'random'} έχει ως αποτέλεσμα την τροποποίηση του αντίστοιχου αντικειμένου και στη δομή **pair a.** Αυτό οφείλεται καθαρά στο γεγονός ότι η JavaScript επιτρέπει την τροποποίηση ενός αντικειμένου δίχως το σχηματισμό νέου για λόγους αποδοτικότητας, πράγμα όμως που όπως είδαμε αντιτίθεται στις θεμιτές πρακτικές του συναρτησιακού μοντέλου. Στις αμιγώς συναρτησιακές γλώσσες όπως η **Haskell**, όλες οι δομές είναι αγνές και ακολουθούν τις προδιαγραφές που θέτει η ορολογία "**Persistent Data Structures**". Το δεύτερο συμπέρασμα έχει να κάνει, όπως προαναφέραμε, με την ευκολία και την ελαστικότητα που αντιμετωπίζει ο συναρτησιακός προγραμματισμός τις δομές αυτές αφού θα λέγαμε ότι αυτές αποτελούν φυσικό επακόλουθο των βασικών πράξεων που υποστηρίζει. Και ως απόδειξη της τελευταίας πρότασης, πάμε να δούμε πως μπορούμε να μετασχηματίσουμε τη λειτουργικότητα της δομής **pair** σε μία δομή **stream** χρησιμοποιώντας την τεχνική της καθυστερημένης αποτίμησης (**lazy evaluation**) με τη βοήθεια ασφαλώς, τι άλλο; Συναρτήσεων.

4.10.3 Ροές (Streams)

Στην επιστήμη της πληροφορικής, ως ροή ορίζουμε μία ακολουθία δεδομένων που γίνονται διαθέσιμα στον χρήστη (**καταναλωτή/consumer**) επιμεριζόμενα στη διάσταση του χρόνου. Πρόκειται για μία τεχνική που επιτρέπει στους προγραμματιστές τη διαίρεση

των δεδομένων τους σε αυτοτελής μονάδες τις οποίες λαμβάνει και επεξεργάζεται σταδιακά ένα πρόγραμμα πελάτης δημιουργώντας έτσι ένα αντιστάθμισμα μεταξύ της πολυπλοκότητας που προκύπτει από την άμεση προσπέλαση των δεδομένων με την πολυπλοκότητα που θα προκύψει από την προσπέλαση σε χρονικά διάσπαρτα δεδομένα. Η εναλλαγή αυτή που πηγάζει από τη μετακύλιση της πολυπλοκότητας στη χρονική διάσταση, αποδεικνύεται ότι συνοδεύεται από ορισμένα οφέλη. Αυτά είναι η αποδοτικότερη αξιοποίηση των πόρων ενός συστήματος αποφεύγοντας περιττά υπολογιστικά κόστη τα οποία κρίνονται μη αναγκαία κατά το στάδιο της κατανάλωσης μίας ροής και δεύτερον, το γεγονός ότι η εισαγωγή του χρονικού παράγοντα επιτρέπει στο μοντέλο τη μεγαλύτερη ταύτιση του με την ανθρώπινη προσέγγιση αποτελώντας έτσι ένα δομικό φυσικό στοιχείο για τη δημιουργία μίας πληθώρας εφαρμογών. Το εύρος των εφαρμογών αυτών, μπορεί να αρχίσει από τη μεταφορά πολυμεσικού περιεχομένου (e-radio, voip, iptv) και να καταλήξει μέχρι και στο σχηματισμό μοντέλων για την επεξεργασία των σημάτων συσκευών εισόδου-εξόδου (IO devices) σε επίπεδο λειτουργικού συστήματος. Πώς όμως μπορεί να υλοποιηθεί προγραμματιστικά μία ροή; Για να απαντηθεί αυτό το ερώτημα είναι ωφέλιμο να γίνει αναφορά στο εισαγωγικό κεφάλαιο στο σημείο όπου γίνεται ανάλυση των διαφορετικών στρατηγικών αποτίμησης που ακολουθούνε οι σύγχρονες γλώσσες προγραμματισμού. Η JavaScript, όπως και οι περισσότερες γλώσσες που είναι κατά κύριο λόγο στραμμένες στην αντικειμενοστρέφεια, υποστηρίζουν τη μέθοδο αποτίμησης **applicative order/call by sharing.** Από την άλλη γλώσσες που είναι περισσότερο στραμμένες στον συναρτησιακό τρόπο γραφής συνηθώς υποστηρίζουν επιπρόσθετα τη μέθοδο αποτίμησης **normal order/call by need**. Η κύρια διαφορά ανάμεσα στις δύο αυτές τεχνικές είναι ότι στην πρώτη περίπτωση η αποτίμηση των εκφράσεων που ορίζονται ως παράμετροι εντός συναρτήσεων είναι αυστηρή (**strict evaluation**), ενώ στη δεύτερη η αποτίμηση γίνεται σε δεύτερο χρόνο και μόνο αν αυτή απαιτηθεί (**lazy evaluation**). Ανάλογα με το είδος των εκφράσεων καθεμία από τις προαναφερόμενες τεχνικές παρουσιάζει πλεονεκτήματα ή μειονεκτήματα. Επίσης ένα θετικό στοιχείο είναι ότι διά της χρήσης κατάλληλων αφαιρέσεων δύναται η προσομοίωση του ενός μοντέλου με χρήση του άλλου ακόμα και αν η γλώσσα δεν το υποστηρίζει. Αυτό είναι άλλωστε που θα κάνουμε και εμείς στο παρόν υποκεφάλαιο με σκοπό τη δημιουργία της δομής των ροών, κάνοντας χρήση δηλαδή τεχνικών καθυστερημένης αποτίμησης

(**lazy**) σε ένα περιβάλλον (αυτό της JavaScript) που υποστηρίζει μόνο αυστηρές αποτιμήσεις (**strict/eager**).

Η έννοια της ροής αποτελεί περισσότερο μία ιδιότητα που συνοδεύει κάποια οργανωμένα δεδομένα παρά μία αυτοτελή δομή δεδομένων. Αν υποθέσουμε δηλαδή ότι έχουμε οργανώσει τα δεδομένα μας σε κάποιο διατεταγμένο τρόπο, τότε μπορούμε με τις κατάλληλες τροποποιήσεις να κατασκευάσουμε μία ροή που θα μας επιτρέπει τη σταδιακή κατανάλωση της οργανωμένης δομής αντί για μαζική προγραμματιστική προσπέλαση της. Στο παρόν υποκεφάλαιο θα δούμε πως μπορούμε να μετατρέψουμε τη συναρτησιακή δομή της διασυνδεμένης λίστας **pair** σε μία ροή, εκμεταλλευόμενοι κατά κύριο λόγο την ευελιξία που προσφέρει η συναρτησιακή προσέγγιση υλοποίησης της. Ας θυμηθούμε τον ορισμό της δομής στη διάσταση του λογισμού λάμδα και της αντίστοιχης προγραμματιστικής της απεικόνισης στη γλώσσα JavaScript. Η παρακάτω αφαίρεση λάμδα λοιπόν έχουμε αποδείξει ότι

```
pair = λx.λy.λs.(s x y)
```

αντιπροσωπεύει πετυχημένα τη δομή μίας διασυνδεμένης λίστας. Η κεφαλή και η ουρά γίνονται διαθέσιμες με τις παρακάτω εκφράσεις

```
K = λx.λy.x
I = λx.x

car = λp.p K
cdr = λp.p (K I)

car = λp.p (λx.λy.x)
cdr = λp.p (λx.λy.y)
```

Αντίστοιχα στο προγραμματιστικό πεδίο έχουμε

```
const pair = x => y => s => s(x)(y);

const K = x => y => x;
const I = x => x;

const car = p => p(K);
const cdr = p => p(K(I));
```

Έτσι για το σχηματισμό μία διασυνδεμένης λίστας που περιέχει τους ακέραιους αριθμούς στο εύρος [0, 5) θα γράφαμε

```
const zeroToFour = pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(null)))));
//-> pair(0)(pair(1)(pair(2)(pair(3)(pair(4)(null)))))
```

Παρατηρούμε ότι η δημιουργία της δομής πραγματοποιείται απευθείας λόγω της αυστηρής αποτίμησης των συναρτησιακών εκφράσεων. Έτσι ο παράγοντας **y=pair(1(…))** που ορίζεται ως δεύτερο όρισμα στη μητρική κλήση **pair(0)(y)** ή το όρισμα **z=pair(2(…))** στην κλήση **pair(1)(z),** υπολογίζονται άμεσα δίνοντας έτσι υπόσταση στη δομή της διασυνδεμένης λίστας. Η λογική αυτή δεν αντιμετωπίζει κάποια ευπάθεια και όπως είδαμε είναι εξ' ολοκλήρου λειτουργική και μάλιστα στην πλειονότητα των περιπτώσεων είναι η επιθυμητή. Το μαγικό με αυτού του είδους τις αναδρομικές συναρτησιακές δομές είναι όμως η δυνατότητα σχηματισμού τους δίχως την ολοκληρωμένη αποτίμηση της έκφρασης του ορισμού τους, σε αντίθεση δηλαδή με τη δομή **zeroToFour**. Πώς όμως μπορεί να γίνει κάτι τέτοιο;

Αποδεικνύεται ότι οι δομές που ορίζονται αναδρομικά ακολουθώντας τη δεξιά-προσεταιριστική ιδιότητα (right-associative) μπορούν ορισθούν σταδιακά σε μία λογική αποτίμησης ένας μέρους της δομής (που εξαρτάται από τον ορισμό της) συναθροιζόμενη με την καθυστερημένη αποτίμηση της δομής του υπόλοιπου μέρους της. Για να αντιληφθούμε καλύτερα τα λεγόμενα, ας υποθέσουμε ότι η γλώσσα JavaScript προσφέρει στον προγραμματιστή τη δυνατότητα της καθυστερημένης αποτίμησης μίας έκφρασης προσάπτοντας στην έκφραση τον τελεστή **lazy.** Η συναρτησιακή αναδρομική φύσης της δομής **pair** επιτρέπει στην έκφραση **zeroToFour** να πάρει την παρακάτω μορφή

```
const zeroToFour = pair(0)(lazy pair(1)(lazy pair(2)(lazy pair(3)(lazy pair(4)(lazy null)))));
//-> pair(0)(...ToBeEvaluated)
```

Εισάγοντας τον τελεστή της καθυστερημένης αποτίμησης η διασυνδεμένη λίστα, αποκτά τις ιδιότητες μίας ροής, καθώς όπως παρατηρούμε ο παράγοντας **x** στην έκφραση pair(0)(**x**) δεν έχει αποτιμηθεί και βρίσκεται σε ένα καθεστώς αδράνειας. Ο υπολογισμός της παραμέτρου **x** πραγματοποιείται μόνο όταν αυτό απαιτηθεί, όταν δηλαδή ένας καταναλωτής της ροής-δομής ζητήσει την προγραμματιστική προσπέλαση της αντίστοιχης πληροφορίας.

Η ιδιότητα αυτή πηγάζει, όπως προαναφέραμε, από τη δυνατότητα σχηματισμού της δομής μέσω της συνάρτησης υψηλής τάξης **foldr.** Έτσι, αν υποθέσουμε ότι η καθυστερημένη αποτίμηση πραγματοποιείται διά μέσου του τελεστή **lazy**, η κλασσική συνάρτηση μετατροπής ενός πίνακα σε μία δομή ροής θα πάρει την παρακάτω μορφή

```
// foldr :: ((a, b) -> b, b, [a]) -> b
const foldr = (f, z, [first, ...rest]) => first === undefined ? z : f(first,
foldr(f, z, rest));

// lazyPrepend :: (a, Pair a) -> Lazy Pair a
const lazyPrepend = (x, lazy y) => pair(x)(y);

// lazyPairFrom :: [a] -> Lazy Pair a
const lazyPairFrom = arr => foldr(lazyPrepend, null, arr);

lazyPairFrom([1, 2, 3, 4, 5]);
//-> pair(0)(...ToBeEvaluated)
```

Και για να δείξουμε την υλοποίηση ενός καταναλωτή. Έστω ότι θέλουμε να εκτυπώσουμε τα στοιχεία μίας ροής, τότε θα γράφαμε μία συνάρτηση **logger** για την οποία ισχύει

```
const logger = lp => lp === null ?
        console.log(null) :
        (console.log(car(lp)), logger(force cdr(lp)));

logger(lazyPairFrom([1, 2, 3, 4, 5]))
//-> 1
//-> 2
//-> 3
//-> 4
//-> 5
//-> null
```

Ο τελεστής **force** εξαναγκάζει την αποτίμηση των εκφράσεων που σχηματίστηκαν μέσω του τελεστή **lazy**, επιτρέποντας έτσι την ομαλή συνέχεια της κατανάλωσης σε ένα αναδρομικό μοτίβο με βήματα την εκτύπωση της κεφαλής και την αποτίμηση της ουράς.

Ωραία όλα αυτά αλλά ως γνωστόν οι τελεστές **lazy** και **force** δεν είναι διαθέσιμοι στη γλώσσα που μελετάμε. Για ακόμη μια φορά ο συναρτησιακός προγραμματισμός έχει τη λύση, καθώς το μοντέλο **lazy-force** μπορεί να προσομοιωθεί πετυχημένα διά της χρήσης συναρτησιακών αφαιρέσεων και εφαρμογών. Έστω λοιπόν ότι έχουμε μία έκφραση **x**, πώς θα μπορούσαμε να καθυστερήσουμε την αποτίμηση της έκφρασης από τη μηχανή της JavaScript; Πολύ απλά προσθέτοντας ένα αφαιρετικό επίπεδο γύρω από τη συγκεκριμένη έκφραση. Και όπως εύλογα μπορούμε να μαντέψουμε το αφαιρετικό αυτό επίπεδο είναι μία συνάρτηση. Μπορούμε να γράψουμε δηλαδή

```
x;
//-> x

() => x;
//-> Function
```

ή

```
3;
//-> 3

() => 3;
//-> Function
```

ή

```
pair(1)(null);
//-> pair(1)(null)

() => pair(1)(null);
//-> Function
```

Παρατηρούμε ότι ενθυλακώνοντας τη ζητούμενη έκφραση εντός μίας συναρτησιακής έκφρασης καταφέρνουμε την καθυστέρηση της αποτίμησης της ζητούμενης αυτής έκφρασης. Πρακτικά αυτό που γίνεται μίας και η γλώσσα ακολουθεί αυστηρή στρατηγική αποτίμησης (**strict evaluation strategy**), είναι η ανταλλαγή εκφράσεων αποτίμησης. Αντί να δώσουμε στη μηχανή τη ζητούμενη έκφραση προς αποτίμηση, της δίνουμε μία άλλη (συναρτησιακή), η άμεση αποτίμηση της οποίας δεν μας ενοχλεί καθότι επιστρέφει μία συνάρτηση. Εάν επιδιώξουμε την αποτίμηση της ζητούμενης έκφρασης από την άλλη, το μόνο που έχουμε να κάνουμε είναι η απαλοιφή της συναρτησιακής αφαίρεσης μέσω της εφαρμογής της. Έτσι το μοντέλο lazy-force περιγράφεται από τις παρακάτω συμπληρωματικές ενέργειες

```
x;
//-> x

() => x; // lazy x
//-> Function

(() => x)() // force (lazy x)
//-> x
```

ή

```
3;
//-> 3

() => 3; // lazy 3
//-> Function

(() => 3)() // force (lazy 3)
//-> 3
```

ή

```
pair(1)(null);
//-> pair(1)(null)

() => pair(1)(null); // lazy pair(1)(null)
//-> Function

(() => pair(1)(null))() // force (lazy pair(1)(null))
//-> pair(1)(null)
```

Προτού αντικαταστήσουμε τις ισοδύναμες εκφράσεις των τελεστών lazy-force στο θεωρητικό μοντέλο που μόλις αναλύσαμε με σκοπό το σχηματισμό μίας ροής, αξίζει να σημειωθούν δύο διευκρινήσεις. Πρώτον, ο τελεστής **lazy** σε αντίθεση με τον τελεστή **force**, εφαρμόζεται στο αριστερό τμήμα μίας έκφρασης ή καλύτερα μίας δήλωσης (**lhs** - left hand side) και όχι στο δεξί (**rhs** - right hand side) όπως γίνεται με τον δεύτερο. Επειδή ως γνωστόν κατά τη δήλωση των παραμέτρων μίας συνάρτησης στην JavaScript είναι αδύνατη η αποφυγή της άμεσης αποτίμησης, η συναρτησιακή έκφραση που αντικαθιστά τον τελεστή **lazy** θα δηλώνεται στο δεξί μέρος μίας δήλωσης (rhs) διαφοροποιώντας ελάχιστα το μοντέλο δήλωσης καθυστερημένης αποτίμησης. Το δεύτερο στοιχείο που χρήζει αναφοράς είναι το γεγονός ότι οι τρόποι υλοποίησης μίας ροής ποικίλουν. Σε κάθε όμως περίπτωση η βασική αρχή πάνω στην οποία στηρίζονται είναι η αξιοποίηση των συναρτησιακών εκφράσεων. Παρακάτω θα αναδείξουμε αρχικά την υλοποίηση των ροών με ένα μοντέλο που αντικατοπτρίζει όσο το δυνατόν καλύτερα τη θεωρία της συνάρτησης **lazyPrepend** και στη συνέχεια θα εμβαθύνουμε σε μία υλοποίηση των ροών και των λειτουργιών που τη συνοδεύουν με τη βοήθεια συναρτήσεων υψηλής τάξης (**higher order functions**).

Πάμε λοιπόν να δούμε πως μπορούμε να δημιουργήσουμε μία διασυνδεμένη λίστα τύπου **pair** με την ιδιότητα της ροής στην JavaScript. Η συνάρτηση μετατροπής ενός πίνακα σε ροή θα έπαιρνε την παρακάτω μορφή

```
// foldr :: ((a, b) -> b, b, [a]) -> (a, () -> b)
const foldr = (f, z, [first, ...rest]) => first === undefined ? z : f(first,
/* lazy */ () => foldr(f, z, rest));

// prepend :: (a, Pair a) -> Pair a
const prepend = (x, y) => pair(x)(y);

// lazyPairFrom :: [a] -> Lazy Pair a
const lazyPairFrom = arr => foldr(prepend, null, arr);

const logger = lp => lp === null ?
        console.log(null) :
        (console.log(car(lp)), logger((/* force */cdr(lp)())));

const a = lazyPairFrom([1, 2, 3, 4, 5]);
//-> pair(0)(Function)
//   or
//-> pair(0)(() => pair(1)(() => pair(2)(() => pair(3)(() => pair(4)(() =>
null)))));

logger(a);
//-> 1
//-> 2
//-> 3
//-> 4
//-> 5
//-> null
```

Παρατηρούμε ότι ο τελεστής **force** επιτυγχάνεται πολύ εύκολα στη συνάρτηση καταναλωτή **logger** μέσω της εφαρμογής της συνάρτησης ουράς της δομής **lazy pair, cdr.** Από την άλλη η φύση της αυστηρής αποτίμησης που ακολουθεί η JavaScript δεν μας επιτρέπει τη χρήση της συναρτησιακής έκφρασης στη θέση του τελεστή **lazy**, στη συνάρτηση prepend, όπως έγινε αντίστοιχα με τη συνάρτηση **lazyPrepend**. Αντί αυτού αναγκαζόμαστε στη μεταβίβαση της συμπεριφοράς στις συναρτήσεις δυναμικής παραγωγής της δομής χρησιμοποιώντας έτσι τη συναρτησιακή έκφραση καθυστέρησης της αποτίμησης εντός της λειτουργικότητας **foldr,** πετυχαίνοντας έτσι το επιθυμητό αποτέλεσμα. Φυσικά η λογική αυτή επεκτείνεται και στις υπόλοιπες λειτουργικότητες που συνοδεύουν τη δομή **lazy pair,** όπως για παράδειγμα οι συναρτήσεις **map**, **filter**, στις οποίες η συμπεριφορά του τελεστή **lazy** πρέπει να ενσωματωθεί εντός της υλοποίησης. Παρακάτω απεικονίζεται ο τρόπος

```
//map :: (a -> b, Lazy Pair a) -> Lazy Pair b
const map = (f, lp) => lp === null ? null : pair(f(car(lp)))(/* lazy */() =>
map(f, (/* force */cdr(lp)())));

// filter :: (a -> Boolean, Lazy Pair a) -> Lazy Pair a
const filter = (f, lp) => lp === null ? null : f(car(lp)) ? pair(car(lp))(/* lazy
*/() => filter(f, (/* force */cdr(lp)()))) : filter(f, (/* force */cdr(lp)()));

const a = lazyPairFrom([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
const b = filter(x => x % 2 === 0, a);
const c = map(x => x * 20, b);

logger(c);
//-> 0
//-> 40
//-> 80
//-> 120
//-> 160
//-> 200
//-> null
```

Βλέπουμε λοιπόν ότι οι λειτουργικότητες υψηλής τάξης **map** και **filter** διατηρούν ίδιο κορμό υλοποίησης με αυτές της δομής pair που γνωρίσαμε στο υποκεφάλαιο 4.10.1, με μοναδική διαφορά την εισαγωγή των τελεστών lazy και force υπό τη μορφή συναρτησιακών εκφράσεων και εφαρμογών αντίστοιχα. Επίσης παρατηρούμε ότι η δομή διατηρεί τις συνθετικές της ιδιότητες επιτρέποντας έτσι την ορθή διασύνδεση των εισόδων-εξόδων σε μία σύνθετη διάταξη. Πρακτικά δηλαδή ο κώδικας του παραπάνω σχήματος μπορεί να γραφεί ισοδύναμα με τη χρήση των βοηθητικών συναρτήσεων **curryN**, **composeN** και **pipeN** με τη μορφή που φαίνεται παρακάτω

```
const a = lazyPairFrom([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
const takeEvenAndMul20 = composeN(curryN(map)(x => x * 20), curryN(filter)(x => x
% 2 === 0));
const b = takeEvenAndMul20(a);

logger(b);
//-> 0
//-> 40
//-> 80
//-> 120
//-> 160
//-> 200
//-> null
```

ή

```
const a = lazyPairFrom([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
const takeEvenAndMul20 = pipeN(curryN(filter)(x => x % 2 === 0), curryN(map)(x =>
x * 20));
const b = takeEvenAndMul20(a);

logger(b);
//-> 0
//-> 40
//-> 80
//-> 120
//-> 160
//-> 200
//-> null
```

Οι συναρτήσεις **compose** και **pipe** αναλαμβάνουν τη δημιουργία σύνθετης λειτουργικότητας που απαρτίζεται από τα συστήματα **filter** και **map**, όπως έχουμε μάθει, σε μία σειριακή διάταξη που αναμένει ως είσοδο μία δομή **lazy pair** και εξάγει μία νέα που είναι έτοιμη προς κατανάλωση.

Ωραία όλα αυτά, αλλά ποια είναι τα πραγματικά οφέλη μίας διασυνδεμένης λίστας που χρησιμοποιεί την τεχνική της καθυστερημένης αποτίμησης (lazy evaluation); Στην εισαγωγή του παρόντος υποκεφαλαίου, αναφέραμε ότι οι ροές μπορούν χρησιμοποιηθούν με σκοπό την αποδοτικότερη αξιοποίηση των πόρων ενός συστήματος αποφεύγοντας περιττά υπολογιστικά κόστη. Πράγματι το γεγονός ότι μία ροή έχει τη δυνατότητα να συγκροτηθεί δίχως να γίνει άμεση προσπέλαση των στοιχείων της, επιτρέπει στην ίδια να στοιβάξει υπό τη μορφή δηλωτικών συνήθως εκφράσεων ένα πλήθος από τροποποιήσεις που πυροδοτούνται μαζικά και βέλτιστα από ένα πρόγραμμα-πελάτης (καταναλωτής). Για την καλύτερη κατανόηση της παραπάνω πρότασης, πάμε να δούμε ένα παράδειγμα στο οποίο γίνεται εμφανής η διαφορά μεταξύ των δομών **pair** και **lazy pair** αντίστοιχα. Έστω οι συναρτήσεις **rangePair** και **rangeLazyPair** που παράγουν τις αντίστοιχες δομές με στοιχεία τους ακέραιους αριθμούς που ορίζονται στο δηλωμένο εύρος.

```
// rangePair :: (Number, Number) -> Pair Number
const rangePair = (x, y) => x < y ? prepend(x, rangePair(x + 1, y)) : null;

// rangeLazyPair :: (Number, Number) -> Lazy Pair Number
const rangeLazyPair = (x, y) => x < y ? prepend(x, /* lazy */() =>
rangeLazyPair(x + 1, y)) : null;
```

Οι παρακάτω εκφράσεις έχουν μία ουσιαστική διαφορά

```
const a = rangePair(0, 1000);
const b = rangeLazyPair(0, 1000);
```

Η δομή **pair** που αποθηκεύεται στη μεταβλητή **a**, σχηματίζεται άμεσα λόγω της αυστηρής αποτίμησης (eager evaluation) που ακολουθεί η υλοποίηση της. Αυτό πρακτικά σημαίνει ότι η **rangePair** θα κληθεί αναδρομικά 1000 (y - x) φορές, ώστε τελικά να σχηματιστεί η διασυνδεμένη λίστα και να εκχωρηθεί στη μεταβλητή **a**. Η δομή lazy pair της μεταβλητής **b** από την άλλη, θα κληθεί μόνο μία φορά ακριβώς λόγο του ότι η υλοποίηση στηρίζεται στην καθυστερημένη αποτίμηση (lazy evaluation) χρησιμοποιώντας την έκφραση **() => rangeLazyPair**, η συνέχεια της οποία εξασφαλίζεται από έναν καταναλωτή μέσω του τελεστή **force**. Η λογική αυτή επεκτείνεται παραπέρα (**cascading**) και σε όλες τις υπόλοιπες λειτουργικότητες των αντίστοιχων δομών οδηγώντας την ανάλυση σε δύο βασικά συμπεράσματα-οφέλη.

Πρώτον, όπως είπαμε η αποφυγή περιττών υπολογισμών. Έστω ότι έχουμε τις δύο αντίστοιχες υλοποιήσεις απεικόνισης **map**, **mapp** για τη δομή **pair** και **maplp** για τη δομή **lazy pair**

```
//mapp :: (a -> b, Pair a) -> Pair b
const mapp = (f, p) => p === null ? null : prepend(f(car(p)), mapp(f, cdr(p)));

//maplp :: (a -> b, Lazy Pair a) -> Lazy Pair b
const maplp = (f, lp) => lp === null ? null : prepend(f(car(lp)), /* lazy */() =>
maplp(f, (/* force */cdr(lp)()));
```

Τότε για το σχηματισμό των εκφράσεων c, d

```
const a = rangePair(0, 1000);
const b = rangeLazyPair(0, 1000);

const c = mapp(x => x * 10, mapp(x => x + 1000, a));
const d = maplp(x => x * 10, maplp(x => x + 1000, b));
```

Απαιτούνται στην πρώτη περίπτωση προσπέλαση αρχική της δομής **a** (1000 επαναλήψεις). Στη συνέχεια εφαρμογή της πρώτης **mapp** με παράμετρο την αφαίρεση λάμδα λx. (x + 1000) (άλλες 1000 επαναλήψεις) και τέλος εφαρμογή της δεύτερης **mapp** με παράμετρο τη λx. (x * 10) (άλλες 1000 επαναλήψεις). Αθροιστικά δηλαδή η αποτίμηση της έκφρασης **c** υποκρύπτει 3000 επαναλήψεις που διεξάγονται άμεσα για το σχηματισμό της νέας διασυνδεμένης λίστας. Από την άλλη πλευρά η αποτίμηση της έκφρασης d, περιλαμβάνει τρεις μόνο υπολογισμούς που αφορούν μόνο τις κεφαλές. Δηλαδή η έκφραση **b** σχηματίζει τη ροή, δημιουργώντας το πρώτο στοιχείο της ροής, και την υπόσχεση (**promise**), να υπολογίσει τα υπόλοιπα αργότερα (μέσω της καθυστερημένης

αποτίμησης). Η πρώτη κλήση **maplp** επεμβαίνει στο πρώτο στοιχείο της δομής **b** εφαρμόζοντας την αφαίρεση λx. (x + 1000) και μελλοντικά υπόσχεται να επέμβει και στα υπόλοιπα. Το ίδιο συμβαίνει και με την επόμενη κλήση της **maplp.** Στο σύνολο δηλαδή πραγματοποιούνται 3 συναρτησιακές εφαρμογές, σε αντίθεση με τις 3000 που εκτελούνται άμεσα στην περίπτωση της δομής **pair**. Φανταστείτε λοιπόν τώρα, ότι έχοντας στην κατοχή μας τις δομές που δείχνουν οι εκφράσεις a,b θέλουμε, αφού εφαρμόσουμε σε αυτές τα συστήματα άθροισης και πολλαπλασιασμού μέσω των συναρτήσεων απεικόνισης **map**, να εξάγουμε μόνο τα πρώτα δέκα στοιχεία της κάθε δομής. Ποια δομή θα ήταν καταλληλότερη για την υλοποίηση της παραπάνω λογικής; Η απάντηση έχει να κάνει με την απόδοση που υποκρύπτει κάθε υλοποίηση και μπορεί να μεταφραστεί σε όρους πολυπλοκότητας ανάλογα με το πλήθος των υπολογισμών. Έστω λοιπόν ότι έχουμε τις συναρτήσεις **takeNfromPair** και **takeNfromLazyPair** που δέχονται ως είσοδο έναν αριθμό που αντιπροσωπεύει το πλήθος των στοιχείων που θέλουμε να εξάγουμε, την αντίστοιχη δομή, και μία θεωρητική συνάρτηση που επεμβαίνει στα στοιχεία.

```
// takeNfromPair :: (Number, Pair a, () -> ()) -> ()
const takeNfromPair = (n, p, f) => n < 1 || p === null ? undefined : (f(car(p)),
takeNfromPair(n - 1, cdr(p), f));

// takeNfromLazyPair :: (Number, Lazy Pair a, () -> ()) -> ()
const takeNfromLazyPair = (n, lp, f) => n < 1 || lp === null ? undefined :
(f(car(lp)), takeNfromLazyPair(n - 1, (/* force */cdr(lp)()), f));
```

Έτσι έχουμε, για την περίπτωση της δομής **pair**, η έκφραση **a** για να μετατραπεί σε **c,** απαιτούνται όπως είδαμε 3000 υπολογισμοί (1000 για τη δημιουργία, 2000 για τις κλήσεις map) και επιπρόσθετα άλλοι 10 για την τελική προσπέλαση των δέκα πρώτων στοιχείων της δομής

```
const a = rangePair(0, 1000); //-> 1000 calculations
const c = mapp(x => x * 10, mapp(x => x + 1000, a)); //-> 2000 calculations
takeNfromPair(10, c, x => console.log(x)); //-> 10 calculations
//-> 10000
//-> 10010
//-> 10020
//-> 10030
//-> 10040
//-> 10050
//-> 10060
//-> 10070
//-> 10080
//-> 10090
```

Ενώ για τη περίπτωση της δομής **lazy pair**, η μετατροπή της έκφρασης **b** σε **d** προϋποθέτει την κλήση 3 υπολογισμών (1 για τη δημιουργία, 2 για τις κλήσεις map) και επιπρόσθετα άλλοι 10 για την τελική προσπέλαση των δέκα πρώτων στοιχείων της δομής.

```
const b = rangeLazyPair(0, 1000); //-> 1 calculation
const d = maplp(x => x * 10, maplp(x => x + 1000, b)); //-> 2 calculations
takeNfromLazyPair(10, d, x => console.log(x)); //-> 10 calculations
//-> 10000
//-> 10010
//-> 10020
//-> 10030
//-> 10040
//-> 10050
//-> 10060
//-> 10070
//-> 10080
//-> 10090
```

Βλέπουμε λοιπόν ότι χρησιμοποιώντας τη δομή της ροής γλιτώνουμε ένα τεράστιο υπολογιστικό κόστος το οποίο μπορεί να αποφευχθεί χάρη στην τεχνική της καθυστερημένης αποτίμησης από τη μία, και στις ιδιότητες που εισάγει ο συναρτησιακός προγραμματισμός από την άλλη, αρχίζοντας από τον ορισμό της αναδρομικής δομής και τελειώνοντας στις γενικές λειτουργικότητες που δένουν με μαθηματική ακρίβεια μεταξύ τους προσφέροντας ισχυρές προγραμματιστικές δυνατότητες.

Το δεύτερο συμπέρασμα που προκύπτει και αποτελεί άμεση απόρροια του προηγούμενου παραδείγματος, είναι η δυνατότητα ορισμού μία άπειρης ροής (**Infinite Stream**). Έστω ότι θέλουμε να εξάγουμε όλους τους περιττούς αριθμούς που βρίσκονται στο εύρος $[0, 10^9)$. Πάμε να δούμε πως θα υλοποιούσαμε τη λύση κάνοντας χρήση της δομής **pair**.

```
// rangePair :: (Number, Number) -> Pair Number
const rangePair = (x, y) => x < y ? prepend(x, rangePair(x + 1, y)) : null;

// filterp :: (a -> Boolean, Pair a) -> Pair a
const filterp = (f, p) => p === null ? null : f(car(p)) ? prepend(car(p),
filterp(f, cdr(p))) : filterp(f, cdr(p));

const a = rangePair(0, 1000000000);
//-> RangeError: Maximum call stack size exceeded
const b = filterp(x => x % 2 !== 0, a);
```

Παρατηρούμε ότι η αποτίμηση της έκφρασης **a** δεν ολοκληρώνεται καθώς παρουσιάζεται σφάλμα που αναφέρει ότι το όριο των εγγραφών της στοίβας έχει ξεπεραστεί. Πράγματι αν δούμε προσεκτικά την υλοποίηση της **rangePair,** θα δούμε ότι η δομή pair σχηματίζεται όπως έχουμε μάθει αναδρομικά εμφωλεύοντας κάθε φορά στην ουρά της μία νέα υπόσταση του εαυτού της. Η φύση της κλήσης αυτής, δεν επιδέχεται βελτιστοποίηση της μορφής **tco** (tail-case-optimization) και ως εκ τούτου είναι ευάλωτη στο σφάλμα της υπερχείλισης της στοίβας (για την ακρίβεια η βελτιστοποίηση **tco** που είδαμε στο υποκεφάλαιο της αναδρομής δεν υποστηρίζεται, ακόμα τουλάχιστον, από τις γνωστές μηχανές JavaScript). Πάμε να δούμε ποιο θα ήταν το αποτέλεσμα αν χρησιμοποιούσαμε τη δομή **lazy pair.**

```
// rangeLazyPair :: (Number, Number) -> Lazy Pair Number
const rangeLazyPair = (x, y) => x < y ? prepend(x, /* lazy */() =>
rangeLazyPair(x + 1, y)) : null;

// filterlp :: (a -> Boolean, Lazy Pair a) -> Lazy Pair a
const filterlp = (f, lp) => lp === null ? null : f(car(lp)) ? pair(car(lp))(/*
lazy */() => filterlp(f, (/* force */cdr(lp)()))) : filterlp(f, (/* force
*/cdr(lp)()));

const a = rangeLazyPair(0, 1000000000);
const b = filterlp(x => x % 2 !== 0, a);
```

Ο κώδικας τρέχει δίχως κανένα σφάλμα. Το πρόβλημα είναι ότι χρειαζόμαστε έναν καταναλωτή ώστε να μπορέσουμε να οπτικοποιήσουμε το αποτέλεσμα. Θα μπορούσαμε να χρησιμοποιήσουμε τη συνάρτηση **logger.** Όμως

```
const logger = lp => lp === null ?
    console.log(null) :
    (console.log(car(lp)), logger((/* force */cdr(lp)())));

logger(b);
//-> RangeError: Maximum call stack size exceeded
```

παρόλο που η υλοποίηση της **logger** είναι σε μορφή tail-cased optimized, το σφάλμα της υπερχείλισης διατηρείται. Στις περιπτώσεις των άπειρων ροών χρησιμοποιούμε συνήθως έναν καταναλωτή που ονομάζεται **trampoline** (εφαρμόζεται σε δομές καθυστερημένης αποτίμησης γνωστές και ως **thunks**) και στην περίπτωση μας μπορεί να πάρει την παρακάτω μορφή αξιοποιώντας έναν βρόχο τύπου **while**

```
const trampoline = (lp, f) => {
    while (typeof lp === "function" && lp !== null) {
        f(car(lp));
        lp = cdr(lp)();
    }
};

trampoline(b, console.log.bind(console));
//-> 1
//-> 3
//-> ...
//-> 999999997
//-> 999999999
```

Παρατηρούμε ότι μέσω του βρόχου **while** δύναται η κατανάλωση ολόκληρης της ροής. Φυσικά στο παρόν παράδειγμα χρησιμοποιήσαμε ως άνω όριο τον αριθμό 1000000000. Μπορούμε κάλλιστα να ορίσουμε μία άπειρη ροή χωρίς άνω όριο, συγκλίνοντας ακόμα πιο πολύ στο πραγματικό ορισμό της. Για παράδειγμα μπορούμε να ορίσουμε το σύνολο των φυσικών αριθμών ℕ με τον παρακάτω τρόπο

```
const N = (n => {
    const rangeFromN = n => prepend(n, /* lazy */() => rangeFromN(n + 1));
    return rangeFromN(n);
})(0);
```

και

```
trampoline(N, console.log.bind(console));
//-> 0
//-> 1
//-> 2
//-> ...
//-> infinity
```

Αφού πήραμε μια γεύση για το τι ακριβώς είναι μία ροή και αντιληφθήκαμε τις προγραμματιστικές δυνατότητες που ξεκλειδώνει η χρήση της, πάμε να δούμε άλλον έναν τρόπο υλοποίησης τους που γίνεται εφικτός χάρη στον συναρτησιακό προγραμματισμό και στις συναρτήσεις υψηλής τάξης (higher order functions). Αποδεικνύεται λοιπόν ότι επεκτείνοντας τη λογική της καθυστερημένης αποτίμησης σε ένα υψηλότερο επίπεδο δύναται η δημιουργία μίας αφαίρεσης λάμδα που δέχεται οποιαδήποτε συνάρτηση, συμπεριλαμβανομένης της συναρτησιακής δομής μας, και τη μετατρέπει σε μία μορφή στην οποία το στάδιο της εφαρμογής καθυστερείτε (λόγω της πρόσθετης αφαίρεσης, όπως

ο τελεστής **lazy**). Η συνάρτηση υψηλής τάξης φαίνεται παρακάτω και έστω ότι την ονομάζουμε **delay**

```
// delay :: (a -> b) -> a -> () -> b
const delay = fn => (...args) => /* lazy */ () => fn(...args);
```

Παρατηρούμε ότι η **delay** δέχεται μία συνάρτηση **fn**, επιστρέφει μία νέα συνάρτηση που αναμένει κάποιες παραμέτρους **args**, και επιστρέφει τελικώς μία καθυστερημένη έκφραση (συναρτησιακή) που κατόπιν εφαρμογής της, καλεί την αρχική συνάρτηση **fn** με τις δηλωμένες παραμέτρους **args**. Πάμε να δούμε πως μπορούμε να τη χρησιμοποιήσουμε.

Για τη συνάρτηση **range** που επιστρέφει μία ροή αριθμών ορισμένους σε έναν εύρος που καθορίζεται από δύο αριθμητικές παραμέτρους, έχουμε

```
// delay :: (a -> b) -> a -> () -> b
const delay = fn => (...args) => /* lazy */ () => fn(...args);

// range :: (Number, Number) -> Lazy Pair Number
const range = (x, y) => x < y ? prepend(x, /* lazy */ delay(range)(x + 1, y)) :
null;

const a = range(0, 10);

// logger :: (Lazy Pair a) -> ()
const logger = lp => lp === null ? console.log(null) : (console.log(car(lp)),
logger((/* force */cdr(lp)()));

logger(a);
//-> 0
//-> 1
//-> 2
//-> 3
//-> 4
//-> 5
//-> 6
//-> 7
//-> 8
//-> 9
//-> null
```

Βλέπουμε ότι η συμπεριφορά είναι ακριβώς η ίδια με τις προηγούμενες υλοποιήσεις. Για τη συνάρτηση **map** που είναι και καταναλωτής αλλά και παραγωγός μίας ροής, θα γράφαμε

```
// delay :: (a -> b) -> a -> () -> b
const delay = fn => (...args) => /* lazy */ () => fn(...args);

// range :: (Number, Number) -> Lazy Pair Number
const range = (x, y) => x < y ? prepend(x, /* lazy */ delay(range)(x + 1, y)) :
null;

const a = range(0, 10);

//map :: (a -> b, Lazy Pair a) -> Lazy Pair b
const map = (f, lp) => lp === null ? null : prepend(f(car(lp)), /* lazy */
delay(map)(f, (/* force */cdr(lp)())));

// logger :: (Lazy Pair a) -> ()
const logger = lp => lp === null ? console.log(null) : (console.log(car(lp)),
logger((/* force */cdr(lp)())));

logger(map(x => x + 10, a));
//-> 10
//-> 11
//-> 12
//-> 13
//-> 14
//-> 15
//-> 16
//-> 17
//-> 18
//-> 19
//-> null
```

Τέλος υπάρχει και η δυνατότητα προσθήκης της **delay** ένα στάδιο αρχύτερα, συμπεριλαμβάνοντας έτσι στην καθυστερημένη αποτίμηση και την αρχική κλήση της συνάρτησης που δημιουργεί τη ροή. Για παράδειγμα, η κλήση **range** θα μπορούσε να μετασχηματιστεί σε μία **lazy** μορφή της δηλώνοντας την ως παράμετρο στη συνάρτηση μετασχηματισμού **delay.** Φυσικά οι συναρτήσεις κατανάλωσης της ροής θα πρέπει να διαμορφωθούν αναλόγως ούτως ώστε να υπάρχει πλήρη αντιστοιχία μεταξύ των κλήσεων **lazy** και **force** για την αποφυγή ανεπιθύμητων συμπεριφορών. Για παράδειγμα η συνάρτηση range θα πάρει τη μορφή που απεικονίζεται παρακάτω

```
// delay :: (a -> b) -> a -> () -> b
const delay = fn => (...args) => /* lazy */ () => fn(...args);

// range:: (Number, Number) -> Lazy Pair Number
const range = /* lazy */ delay((x, y) => x < y ? prepend(x, range(x + 1, y)) :
null);


const a = range(0, 10);

// logger :: (Lazy Pair a) -> ()
const logger = lp => {
    const evaluated = /* force */ lp();
    return evaluated === null ? console.log(null) :
        (console.log(car(evaluated)), logger(cdr(evaluated)));
};

logger(a);
//-> 0
//-> 1
//-> 2
//-> 3
//-> 4
//-> 5
//-> 6
//-> 7
//-> 8
//-> 9
//-> null
```

Παρατηρούμε ότι ο τελεστής **force** στη συνάρτηση κατανάλωσης της ροής **logger** εφαρμόζεται στην τιμή evaluated, για να πραγματοποιηθεί αποτίμηση της αρχικής έκφρασης **range** (μέσω της κλήσης a()). Η λογική αυτή επεκτείνεται αναδρομικά και σε κάθε κλήση **cdr**(evaluated) τοποθετείται η αμέσως επόμενη ουρά τύπου **pair**, η αποτίμηση της οποίας πραγματοποιείται μέσω της έκφρασης

```
const evaluated = /* force */ lp();
```

Επιπρόσθετα οι συναρτήσεις map και filter θα σχηματιστούν ως εξής

```
//map :: (a -> b, Lazy Pair a) -> Lazy Pair b
const map = /* lazy */ delay((f, lp) => {
    const evaluated = /* force */ lp();
    return evaluated === null ? null :
        prepend(f(car(evaluated)), map(f, cdr(evaluated)))
});

// filter :: (a -> Boolean, Lazy Pair a) -> Lazy Pair a
const filter = /* lazy */ delay((f, lp) => {
    const evaluated = /* force */ lp();
    return evaluated === null ? null :
        f(car(evaluated)) ? prepend(car(evaluated), filter(f, cdr(evaluated))) :
         filter(f, cdr(evaluated))();
});


const a = range(0, 10);
const b = map(x => x + 10, a);
const c = filter(x => x % 2 === 0, b);

logger(c);
//-> 10
//-> 12
//-> 14
//-> 16
//-> 18
//-> null
```

Συνακόλουθα η συνάρτηση υψηλής τάξης **delay** μπορεί να χρησιμοποιηθεί και στις λοιπές λειτουργικότητες που αφορούν μία ροή, όπως για παράδειγμα οι συναρτήσεις **foldr**, **lazyPairFrom** κα.

Στο σημείο αυτό και κλείνοντας με το υποκεφάλαιο των ροών, μπορούμε να αντιληφθούμε ότι η ελαστικότητα και η ευρωστία που παρουσιάζει ο συναρτησιακός προγραμματισμός είναι τεράστια. Ο τρόπος δήλωσης λειτουργιών μέσω αυτοτελών συναρτησιακών αφαιρέσεων και η δυνατότητα σύνθεσης αυτών με σκοπό τη δημιουργία νέων, ξεκλειδώνουν όχι μόνο τον προγραμματιστή παρουσιάζοντας του έναν νέο κόσμο στον οποίο μπορεί να αναπτύξει νέες ιδέες, αλλά επιτρέπουν και το σχηματισμό νέων υπολογιστικών-προγραμματιστικών μοντέλων. Οι ροές είναι μόνο ένα μικρό παράδειγμα προς αυτή την κατεύθυνση. Για την ακρίβεια οι ροές μπορούν να υλοποιηθούν και στον αντικειμενοστρεφή προγραμματισμό, και μάλιστα πλέον πολύ απλά με την εγγενή υποστηρίξει της JavaScript στις δομές **generator.** Τα **generators** επιτρέπουν την καθυστερημένη αποτίμηση και στηρίζονται σε μοντέλα πεπερασμένων καταστάσεων που

φυσικά μπορούν να σχεδιαστούν και συναρτησιακά. Το πιο σημαντικό στοιχείο, και αυτό που τελικώς πρέπει να μείνει στον αναγνώστη είναι ότι ο δηλωτικός τρόπος γραφής υπό το πρίσμα του συναρτησιακού μοντέλου ανάλυσης, υποστηρίζεται από μαθηματικές αρχές και είναι βαθύτατα δομημένος. Οι υλοποιήσεις **pair, tree** και **lazy pair**, σαφώς δεν είναι αυθαίρετες. Κάνοντας μία γρήγορη ανασκόπηση στα αντίστοιχα υποκεφάλαια θα δείτε ότι οι ορισμοί και οι λειτουργικότητες ακολουθούνε μία κοινή γραμμή, σαν να υπακούνε κάποιους κρυφούς κανόνες που ανταποδοτικά προσφέρουν νέες ιδιότητες. Στα τελευταία λοιπόν υποκεφάλαια θα μελετήσουμε κάποιες νέες έννοιες από τη μαθηματική θεωρία κατηγοριών και θα δούμε πως αυτή μεταφέρεται στον συναρτησιακό προγραμματισμό για τη δημιουργία καινούργιων ισχυρών μοντέλων.

4.11 Συναρτήτες (Functors)

Κάνοντας μία ανασκόπηση στο περιεχόμενο της παρούσας διπλωματικής, ο αναγνώστης μπορεί να αντιληφθεί εύκολα ότι ο συναρτησιακός προγραμματισμός είναι ισχυρά δομημένος και ευέλικτος. Αυτό φυσικά οφείλεται στο γεγονός, όπως σαφώς ξεκαθαρίσαμε εξ αρχής, ότι η συναρτησιακή σχεδίαση δεν είναι αυθαίρετη άλλα αντιθέτως στηρίζεται βαθύτητα σε μαθηματικά μοντέλα και έννοιες. Ο συναρτησιακός προγραμματισμός ακριβώς όπως και τα μαθηματικά, αντιμετωπίζει τα δεδομένα που διαχειρίζεται στα πλαίσια κάποιας αφηρημένης κατηγορίας συνόλου (**Set**) η οποία ακολουθεί κάποιους νόμους που της επιτρέπουν να σχηματίσει μαθηματικές λειτουργικότητες γύρω από τα στοιχεία της (Milewski, 2018). Η αλληλεπίδραση ανάμεσα στα καθορισμένα αυτά σύνολα μπορεί να μελετηθεί υπό τη μαθηματική θεωρία των κατηγοριών και συγκεκριμένα των κατηγοριών των συνόλων (**Category of Sets**). Η θεωρία κατηγοριών όπως αναφέραμε και στο δεύτερο κεφάλαιο, αποτελεί ένα αφηρημένο μοντέλο που επιτρέπει την ενιαιοποίηση διαφόρων μαθηματικών δομών αντιμετωπίζοντας τες ως ένα σύνολο αντικειμένων που σχετίζονται μεταξύ τους δια μέσω των λεγόμενων μορφισμών. Για παράδειγμα στην κατηγορία των συνόλων, τα σύνολα ορίζονται ως αντικείμενα της κατηγορίας, ενώ οι μορφισμοί αποτελούν αλγεβρικές συναρτήσεις που στην προκειμένη περίπτωση επιτρέπουν την απεικόνιση των στοιχείων ενός συνόλου σε ένα άλλο σύνολο. Τον αφηρημένο αυτό ορισμό της κατηγορίας εκμεταλλεύεται και ο συναρτησιακός προγραμματισμός αφού προσπαθεί να ερμηνεύσει τα δεδομένα τα οποία

διαχειρίζεται υπό το πρίσμα μίας τέτοιας κατηγορίας. Φυσικά στην πλειονότητα των περιπτώσεων ο ορισμός αυτός έχει υπόσταση και στηρίζεται σε γνωστές κατηγορίες όπως είναι τα σύνολα. Πολλές φορές όμως υπάρχει η δυνατότητα σχηματισμού μία νέας κατηγορίας που προκύπτει από τον μετασχηματισμό μίας υπάρχουσας. Το γεγονός αυτό ενεργοποιεί τον προγραμματιστή με νέες εγγενής ιδιότητες. Η απεικόνιση αυτή μεταξύ των αντικειμένων των δύο κατηγοριών υλοποιείται μέσω των συναρτήτων (**Functors**). Εκ πρώτης όψεως και λαμβάνοντας υπόψιν τη θεωρητική τους σημασία, ο σκοπός των συναρτήτων θα λέγαμε ότι δεν είναι διακριτός. Όπως θα δούμε όμως παρακάτω στην πραγματικότητα οι συναρτήτες αποτελούνε τη θεμελιώδη έννοια μέσω της οποίας μπορούμε να ορίσουμε μία συνθετική δομή. Ας δούμε όμως τα πράγματα ένα ένα.

Αναφέραμε ότι ένας συναρτήτης απεικονίζει τα αντικείμενα μίας κατηγορίας σε μία άλλη. Η ενέργεια αυτή πρακτικά μεταφράζεται στην ύπαρξη μίας συνάρτησης απεικόνισης **fmap** που επιτρέπει τον ομαλό μετασχηματισμό των κατηγοριών δίχως να παραβιάζονται οι δύο βασικοί νόμοι με βάση τους οποίους αυτή ορίζεται (Lonsdorf, 2015). Για τον λόγο αυτό η συνάρτηση **fmap** πρέπει να ακολουθεί του δύο παρακάτω νόμους

1) **Identity Law**

$$fmap(id) \equiv id$$

2) **Composition Law**

$$fmap(f) \circ fmap(g) \equiv fmap(f \circ g)$$

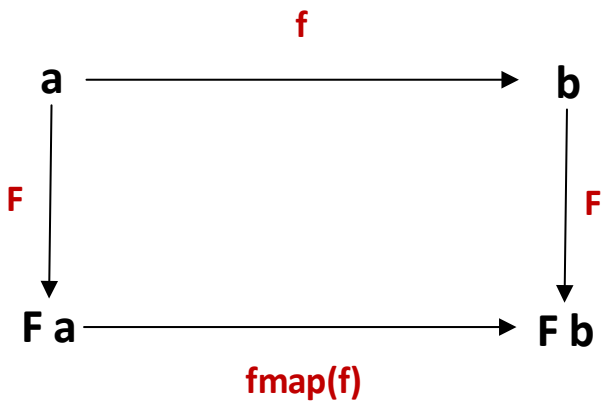
Να πούμε ότι η συνάρτηση **fmap** απεικονίζει ουσιαστικά μεταξύ τους τα αντικείμενα της παλιάς και της νέας κατηγορίας εφαρμόζοντας τη συνάρτηση εισόδου στα αντικείμενα της πρώτης. Για να γίνει καλύτερα αντιληπτό, έστω ότι ορίζουμε ως **F** έναν συναρτήτη που μετασχηματίζει ένα αντικείμενο **a** σε **F a** και ένα αντικείμενο **b** σε **F b .** Αν τα αντικείμενα **a** και **b** συνδέονται μεταξύ τους με έναν μορφισμό **f** για τον οποίο

$$f: a \to b$$

τότε αν η συνάρτηση **fmap** κληθεί με παράμετρο τον μορφιμό **f** θα επιτρέψει τον νέο μορφισμό που συνδέει τα αντικείμενα **F a** και **F b** της νεοσύστατης κατηγορίας, δηλαδή

$$fmap(f): F\ a \to Fb$$

Μπορούμε να αναπαραστήσουμε την παραπάνω πρόταση σχηματικά με την παρακάτω μορφή



Με λίγα λόγια ως συναρτήτης ορίζεται οποιαδήποτε μαθηματική δομή που προσφέρει υλοποίηση σε μία συνάρτηση **fmap** με υπογραφή

$$fmap :: (a \to b) \to F\ a \to F\ b$$

και ακολουθεί τους δύο βασικούς νόμους που προαναφέρθηκαν. Πάμε να δούμε ένα παράδειγμα συναρτήτη και συγκεκριμένα τον ταυτοτικό. Ο ταυτοτικός συναρτήτης τυλίγει μία τιμή οποιασδήποτε μορφής, προσφέροντας τη λειτουργικότητα της **fmap** ως μία απλή εφαρμογή μορφισμού. Στη γλώσσα της JavaScript θα έχουμε

```
// Identity :: a -> Identity a
const Identity = value => ({
    // fmap :: Identity a ⇝ (a -> b) -> Identity b
    fmap: f => Identity(f(value))
});
```

και αν θέλουμε να γράψουμε τη γενική μορφή της **fmap** απαλείφοντας το περιβάλλον του αντικειμένου που διαχειρίζεται μπορούμε να γράψουμε

```
// fmap :: (a -> b) -> Fa -> Fb
fmap = f => Fa => Fa.fmap(f);
```

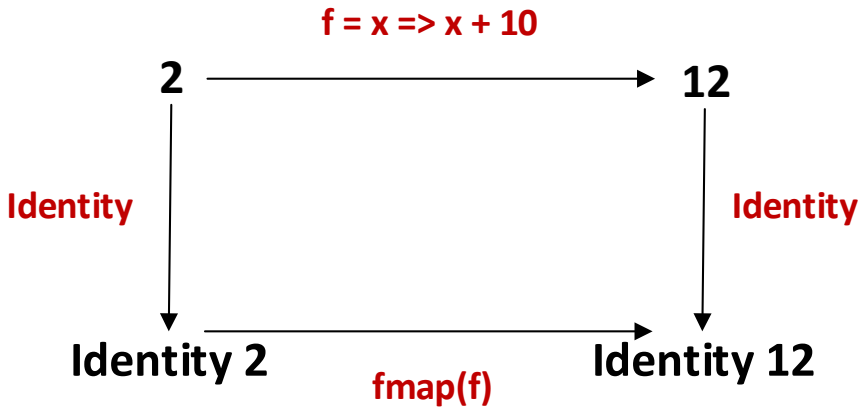
Πάμε να δούμε τώρα αν η δομή ακολουθεί του δύο νόμους που ορίζουν οι συναρτήτες. Έστω ότι καλούμε τον συναρτήτη με την τιμή 2 (δύο)

```
// id :: a -> a
const id = x => x;

const v = Identity(2);

fmap(id)(v)
//-> Identity 2
id(v)
//-> Identity 2
```

Τότε παρατηρούμε ότι ο πρώτος νόμος ισχύει αφού οι δύο τελευταίες εκφράσεις είναι ισοδύναμες και επιστρέφουν **Identity 2**. Φυσικά η λογική αυτή επεκτείνεται και ο πρώτος νόμος μπορούμε να πούμε ότι τηρείται για κάθε τιμή x που ανήκει στον τύπο **Number**. Για τον δεύτερο νόμο πρέπει να εισάγουμε τη συνάρτηση σύνθεσης **compose.** Έχουμε

```
// compose :: (b -> c, a -> b) -> a -> c
const compose = (g, f) => x => g(f(x));

const f = x => x + 10;
const g = x => x * 2

compose(fmap(g), fmap(f))(v);
//-> Identity 24
fmap(compose(g, f))(v);
//-> Identity 24
```

Παρατηρούμε ότι η σύνθεση των συναρτήσεων fmap(g) και fmap(f) είναι ισοδύναμη με την απεικόνιση της σύνθεσης συναρτήσεων $g \circ f$ στο αντικείμενο **Identity 2.** Ο δεύτερος νόμος ισχύει για κάθε x που ανήκει στον τύπο **Number.** Θα λέγαμε ότι η τιμή 2 μέσω του συναρτήτη **Identity** μεταφέρεται σε ένα άλλο περιβάλλον στο οποίο υπάρχει η δυνατότητα τροφοδότησης της τιμής σε συναρτήσεις που υλοποιούν διάφορες λειτουργικότητες. Στην παρακάτω εικόνα φαίνεται καθαρά η διαδικασία μετασχηματισμού ανάμεσα στις δύο διαφορετικές κατηγορίες, την **Number** και την **Identity Number** για την τιμή Number = 2**.**



Ποια είναι όμως η πρακτική χρήση των συναρτήτων και τι απήχηση έχουν στο μοντέλο του συναρτησιακού προγραμματισμού; Το έργο που επιτελούν μπορεί να μην είναι ακόμα διακριτό στον αναγνώστη αλλά οι συναρτήτες προσφέρουν ουσιαστικά μία αμιγώς συναρτησιακή αφαίρεση γύρω από τα δεδομένα που διαχειρίζονται, ενεργοποιώντας το μοντέλο του συναρτησιακού προγραμματισμού με επιπλέον τεχνικές που του δίνουν τη δυνατότητα της αυστηρής τήρησης των νόμων που πρέπει να ακολουθεί. Ο σχηματισμός αγνών συναρτήσεων και η αποφυγή παρενεργειών είναι δύο έννοιες που ακολουθήσει πιστά το μοντέλο της απεικόνισης μέσω της **fmap** που προσφέρουν οι συναρτήτες.

Το πιο γνωστό παράδειγμα συναρτήτη που χρησιμοποιούμε κατά κόρον σε όλες τις γλώσσες προγραμματισμού συναρτησιακές ή μη είναι η δομή του πίνακα. Ναι ο πίνακας είναι ένας συναρτήτης και φυσικά η τρομερή ευελιξία και χρηστικότητα που προσφέρει στα πλαίσια που ορίζει η συναρτησιακή σχεδίαση μόνο τυχαία δεν είναι. Αν έχουμε έναν τύπο **Number**, ο συναρτήτης **List** η **Array** απεικονίζει τα αντικείμενα του τύπου **Number** σε **List Number** ή **Array Number** ή όπως συνηθίζεται να γράφεται για λόγους συντομίας **[Number]**. Η συνάρτηση απεικόνισης **fmap** στις περιπτώσεις των πίνακων όπως εύλογα μπορεί να φανταστεί ο αναγνώστης, είναι η γνωστή **map** με υπογραφή

$$fmap = map :: (a \to b) \to [a] \to [b]$$

Η δομή του πίνακα όντας ένας συναρτήτης, εγκιβωτίζει τα στοιχεία ενός απτού (**concrete type**) τύπου **a** σε ένα νέο υπολογιστικό πλαίσιο (**computational context**) ή σε μία νέα κατηγορία στην οποία μπορούμε να προσδώσουμε υπόσταση σε μία ομαδοποιημένη διάταξη των τιμών του συγκεκριμένου τύπου **a**. Επιπρόσθετα αφού ένας πίνακας αποκρύπτει αφαιρετικά την έλλειψη ντετερμινισμού, ομαδοποιώντας όπως είπαμε ένα πλήθος από ντετερμινιστικές ή μη τιμές, προσδίδει τη λειτουργικότητα **fmap** ως μία πράξη απεικόνισης όλων των τιμών του και όχι μίας συγκεκριμένης (αν φυσικά θα μας ήταν χρήσιμη μία διαφορετική λειτουργικότητα της **fmap**, θα μπορούσαμε να την υλοποιήσουμε, αρκεί να τηρούνται οι νόμοι που ακολουθεί ένας συναρτήτης). Οι δύο νόμοι των συναρτητών ισχύουν φυσικά και για τους πίνακες (Lipovaca, 2011).

```
const v = [1, 2, 3];

// fmap = map :: (a -> b) -> [a] -> [b]
const fmap = map = f => x => x.map(f);

fmap(id)(v);
//-> [1, 2, 3]
id(v);
//-> [1, 2, 3]

// compose :: (b -> c, a -> b) -> a -> c
const compose = (g, f) => x => g(f(x));

const f = x => x + 10;
const g = x => x * 2;

compose(fmap(g), fmap(f))(v);
//-> [22, 24, 26]
fmap(compose(g, f))(v);
//-> [22, 24, 26]
```

Συνεπώς η δομή του πίνακα προσφέρει μία διάταξη στον συναρτησιακό προγραμματισμό στην οποία η απεικόνιση συναρτήσεων τύπου $a \to b$ σε μία συγκεκριμένη δομή του ιδίου τύπου παράγει μία εντελώς νέα δομή, τηρώντας έτσι τους νόμους της αγνότητας (**pureness**), της μη μεταβλητότητας (**immutable**), και συνεπώς προσδίδει αναφορική διαφάνεια (**referential transparency**) στον τρόπο ανάπτυξης λογισμικού. Και φυσικά χάρη στον τύπο της **fmap** (που παίρνει τη μορφή ενός **reducer** function) δύναται η δημιουργία συνθετικών αναπαραστάσεων όπου ο συναρτήτης (είτε είναι πίνακας είτε οτιδήποτε άλλο), μετασχηματίζεται και τροφοδοτείται στο αμέσως επόμενο στάδιο μετασχηματισμού σε μία σειριακή ή αλυσιδωτή μορφή. Για παράδειγμα έστω ότι έχουμε τον παρακάτω πίνακα και τις συναρτήσεις μετασχηματισμού

```
const v = [10, 11, 12, 13, 14];
// f :: Number -> String
const f = x => `Embedded value {${x}}`;
// g :: String -> String
const g = s => `Embed again {${s}}`;
```

Μπορούμε συνεπώς να γράψουμε αλυσιδωτά την έκφραση

```
v.map(f).map(g);
```

ή οποία όμως στη συναρτησιακή της μορφή πηγάζει από την έκφραση

```
fmap(g, fmap(f, v));
```

Η αποτίμηση των δύο παραπάνω εκφράσεων φυσικά είναι ισοδύναμες και το αποτέλεσμα τους παράγει τον παρακάτω πίνακα

```
//->
// ['Embed again {Embedded value {10}}',
//  'Embed again {Embedded value {11}}',
//  'Embed again {Embedded value {12}}',
//  'Embed again {Embedded value {13}}',
//  'Embed again {Embedded value {14}}']
```

Ένα άλλο παράδειγμα συναρτήτη που υλοποιήσαμε στην παρούσα διπλωματική είναι η δομή της διασυνδεμένης λίστας **Pair.** Ακριβώς όπως και η δομή του πίνακα, ένα ζεύγος τύπου pair εγκιβωτίζει τιμές μέσα σε ένα υπολογιστικό πλαίσιο που εκφράζει μη ντετερμινισμό, και κάνει τις τιμές αυτές προσπελάσιμες μέσω της λειτουργικότητας **fmap.** Ας ξαναθυμηθούμε την υλοποίηση

```
const pair = x => y => s => s(x)(y);

const K = x => y => x;
const I = x => x;

const car = p => p(K);
const cdr = p => p(K(I));

// prepend :: (a, Pair a) -> Pair a
const prepend = (x, p) => pair(x)(p);

// map :: (a -> b, Pair a) -> Pair b
const map = (f, p) => p === null ? null : prepend(f(car(p)), map(f, cdr(p)));
```

Στην περίπτωση της **Pair** η συνάρτηση **fmap**, όπως εύλογα φαντάζεστε είναι η **map** που φαίνεται παραπάνω. Έτσι αν έχουμε ένα πλήθος τιμών εντός μία διάταξης Pair, ισχύουν τα παρακάτω

```
const v = pair(1)(pair(2)(pair(3)(null)));

// fmap :: (a -> b) -> Pair a -> Pair b
const fmap = f => p => map(f, p);

fmap(id)(v);
//-> pair(1)(pair(2)(pair(3)(null)))
id(v);
//-> pair(1)(pair(2)(pair(3)(null)))

// compose :: (b -> c, a -> b) -> a -> c
const compose = (g, f) => x => g(f(x));

const f = x => x + 10;
const g = x => x * 2;

compose(fmap(g), fmap(f))(v);
//-> pair(22)(pair(24)(pair(26)(null)))
fmap(compose(g, f))(v);
//-> pair(22)(pair(24)(pair(26)(null)))
```

Παρατηρούμε δηλαδή ότι οι νόμοι που πρέπει να ισχύουν για έναν συναρτήτη έχουν απήχηση και για τη δομή **Pair.** Αν θέλουμε να χρησιμοποιήσουμε την τεχνική της αλυσίδωσης για αυτή τη δομή, πρέπει να σχηματιστεί ένα επιπρόσθετο περιβάλλον που θα λειτουργεί ως αφαιρετική οντότητα διαχείρισης μίας διάταξης της μορφής pair(x)(pair(…)(…)). Η ανάγκη αυτή προκύπτει καθαρά από τον τρόπο λειτουργίας της JavaScript, όπου η δημιουργία ενός τέτοιου περιβάλλοντος ισοδυναμεί με τη δημιουργία ενός αντικειμένου. Μπορούμε να γράψουμε για παράδειγμα

```
// Pair :: pair a -> Pair a
const Pair = x => ({
    fmap: f => Pair(map(f, x))
});

Pair(v).fmap(f).fmap(g) // same as Pair(v).fmap(compose(g, f))
//-> pair(22)(pair(24)(pair(26)(null)))
```

Η τιμή v εκχωρείται εντός της συνάρτησης **Pair** για το σχηματισμό ενός αντικειμένου που επιτρέπει την εκτέλεση συνάρτησης **fmap** ως μέθοδο και συνακόλουθα ενεργοποιεί την τεχνική της αλυσίδωσης.

Είδαμε λοιπόν πως οι δομές του πίνακα και της διασυνδεμένης λίστας υλοποιούν τις προϋποθέσεις που ορίζει ένας συναρτήτης αφού προσδίδουν υλοποίηση στη λειτουργικότητα **fmap** τηρώντας ταυτόχρονα τους βασικούς κανόνες των συναρτητών. Είδαμε επίσης, πως από μαθηματικής πλευράς οι συναρτήτες αλλάζουν το υπολογιστικό περιβάλλον γύρω από το οποίο συνθέτονται τα αντικείμενα μίας κατηγορίας, και δημιουργούν ένα νέο (μία νέα κατηγορία δηλαδή) όπου η σύνθεση επιτελείται μέσω μίας απεικόνισης (**fmap**).

Προτού κλείσουμε με το παρόν υποκεφάλαιο πάμε να δούμε και έναν άλλο γνωστό συναρτήτη, η χρήση του οποίου είναι διαδεδομένη σε συναρτησιακές γλώσσες προγραμματισμού. Πρόκειται για τον γνωστό τύπο **Maybe a**, που εκγκιβωτίζει έναν τύπο δεδομένων **a** σε ένα υπολογιστικό πλαίσιο, εντός του οποίου ένα αντικείμενο παίρνει δύο καταστάσεις. Η μία κατάσταση δηλώνει την ύπαρξη μίας απτής τιμής μέσω μίας δομής **Just a,** ενώ η άλλη δηλώνει ανυπαρξία μέσω μίας δομής **Nothing.** Πρακτικά αυτό που επιτυγχάνεται είναι η πολυμορφική συμπεριφορά που αποκτά ένας τύπος δεδομένων, επιτρέποντας έτσι στη λειτουργικότητα απεικόνισης **fmap** να υλοποιεί δύο προσεγγίσεις. Η πρώτη, στην περίπτωση που η τιμή υπάρχει εντός της δομής **Just**, η **fmap** εφαρμόζεται στο εσωτερικό στοιχείο της κανονικά. Στη δεύτερη περίπτωση, η **fmap** επιστρέφει απλά την ίδια δομή **Nothing** αγνοώντας εντελώς τη συνάρτηση εισόδου και επεκτείνοντας το καθεστώς της ανυπαρξίας. Εκ πρώτης όψεως μπορεί να μην αντιλαμβάνεται κάποιος τη χρήση της δομής **Maybe**, αλλά αποτελεί μία σημαντική διάταξη που επιτρέπει στον συναρτησιακό προγραμματισμό την ενιαία αντιμετώπιση των δεδομένων που διαχειρίζεται απαλείφοντας την ανάγκη χρήσης εκφράσεων **if** για την αντιμετώπιση ανεπιθύμητων τιμών. Προτού δούμε ένα παράδειγμα, πάμε να μοντελοποιήσουμε τα παραπάνω στη γλώσσα της JavaScript.

```
// Maybe a -> Nothing | Just a

const Nothing = () => ({
    fmap: _ => Nothing()
});

const Just = x => ({
    fmap: f => Just(f(x))
});
```

Βλέπουμε ότι οι δύο καταστάσεις του συναρτήτη **Maybe a**, μεταφράζονται στην JavaScript, στις συναρτήσεις που παράγουν αντικείμενα τύπου **Nothing** και **Just a** με υλοποίηση της συνάρτησης απεικόνισης **fmap**. Επίσης όπως προαναφέραμε παρατηρούμε ότι στην περίπτωση της δομής **Just** η **fmap** εφαρμόζει τη συνάρτηση τύπου $a \rightarrow b$ στο εσωτερικό στοιχείο της δομής, ενώ η δομή **Nothing** αγνοεί εντελώς τη συνάρτηση και επιστρέφει τον εαυτό της.

Έστω λοιπόν ότι έχουμε την παρακάτω συνάρτηση **findStudent** που δέχεται ένα σειριακό αριθμό και επιστρέφει ένα αντικείμενο τύπου **Student**. Ως προγραμματιστές γνωρίζουμε ότι οι οριακές καταστάσεις συνήθως αποτελούν εμπόδιο στη δυνατότητα ανάπτυξης ομοιόμορφου κώδικα καθώς εισάγουν αβεβαιότητα, που στην πλειονότητα των περιπτώσεων αντιμετωπίζεται με τη χρήση δομών ελέγχου **if.** Έχουμε προς το παρόν

```
// findStudent :: String -> Student
const findStudent = sn => {
    switch (sn) {
        case "1111": return {name: "Alonzo Church", age: 36};
        case "2222": return {name: "Haskell Curry", age: 39};
        case "3333": return {name: "Alan Turing", age: 27};
        default: return undefined; // ?????????
    }
};
```

Η υπογραφή της παραπάνω συνάρτησης δέχεται ένα αλφαριθμητικό τύπο **String** και επιστρέφει ένα αντικείμενο μαθητή **Student.** Η προσέγγιση αυτή όμως δεν είναι ακριβής διότι στις περιπτώσεις που ο σειριακός αριθμός υπάγεται στην περίπτωση της **default**, η τιμή επιστροφής δεν είναι ένα αντικείμενο **Student** αλλά η τιμή **undefined**. Φανταστείτε τώρα πως ένα πρόγραμμα πελάτης χρησιμοποιεί τη συνάρτηση **findStudent**, για να κάνει κάποιους υπολογισμούς. Για παράδειγμα έστω η συνάρτηση πελάτης **processStudent** με υπογραφή

$$processStudent :: Student \rightarrow Int$$

```
// processStudent :: Student -> Int
const processStudent = st => st ? st.age : undefined // ???
```

Θεωρούμε ότι η συνάρτηση δέχεται ένα τύπο **Student** και επιστρέφει την ηλικία του. Επειδή ακριβώς η **findStudent** μπορεί να επιστρέψει την τιμή **undefined,** ο κώδικας υλοποίησης της **processStudent** πρέπει να μεριμνήσει για την περίπτωση αυτή εισάγοντας μία συνθήκη ελέγχου που θα διαφοροποιεί τη ροή που θα ακολουθήσει το πρόγραμμα. Η

λογική αυτή επεκτείνεται και στην τιμή επιστροφής της **proccessStudent**, καθώς εύλογα μπορεί να αναρωτηθεί κάποιος, ποια θα είναι η ηλικία επιστροφής μίας **undefined** τιμής; Ακριβώς για αυτές τις περιπτώσεις συνίσταται η χρήση του συναρτήτη **Maybe**, που επιτρέπει τον εμπλουτισμό μίας τιμής υποδεικνύοντας ύπαρξη (**Just x**) ή ανυπαρξία (**Nothing**). Συνεπώς μπορούμε πολύ απλά να τροποποιήσουμε την **findStudent**, έτσι ώστε αντί να επιστρέφει αντικείμενα τύπου **Student** ή την τιμή **undefined,** να φέρνει αντικείμενα τύπου **Maybe Student.**

Η νέα **findStudent** θα έχει συνεπώς την υπογραφή

$$\text{findStudent} :: String \rightarrow Maybe\ Student$$

```
// findStudent :: String -> Maybe Student
const findStudent = sn => {
    switch (sn) {
        case "1111": return Just({ name: "Alonzo Church", age: 36 });
        case "2222": return Just({ name: "Haskell Curry", age: 39 });
        case "3333": return Just({ name: "Alan Turing", age: 27 });
        default: return Nothing();
    }
};
```

Βλέπουμε ότι η συνάρτηση στις περιπτώσεις που έχουμε κάποιο έγκυρο σειριακό αριθμό επιστρέφει το αντικείμενο τύπου **Student** εντός της δομής **Just,** ενώ αντιθέτως σε όλες τις υπόλοιπες περιπτώσεις επιστρέφει τη δομή **Nothing** δηλώνοντας έλλειψη τιμής. Ποιο είναι το αποτέλεσμα; Το αποτέλεσμα είναι ότι το πρόγραμμα πελάτης γνωρίζοντας πλέον ότι η τιμή επιστροφής της συνάρτησης **findStudent** είναι ένα τύπος **Maybe Student**, μπορεί να υιοθετήσει έναν πιο ευέλικτο κώδικα αποφεύγοντας περιττούς ελέγχους. Συνεπώς η **processStudent** μπορεί να πάρει τις δύο παρακάτω μορφές.

```
// processStudent :: Student -> Int
const processStudent = st => st.age;

findStudent("2222").fmap(processStudent)
//-> Just 39
findStudent("r@nd0m").fmap(processStudent);
//-> Nothing
```

ή

```
// processStudent :: Maybe Student -> Maybe Int
const processStudent = fmap(st => st.age);

processStudent(findStudent("2222"))
//-> Just 39

processStudent(findStudent("r@nd0m"));
//-> Nothing
```

Παρατηρούμε δηλαδή ότι προσθέτοντας το υπολογιστικό πλαίσιο του συναρτήτη **Maybe**, καταφέραμε να σχηματίσουμε κατά τα πρότυπα που ορίζει ο συναρτησιακός προγραμματισμός, μία αγνή διάταξη που απεικονίζει συναρτήσεις μετασχηματισμού αδιαφορώντας για το περιεχόμενο.

Οι συναρτήτες αποτελούν μία από τις βασικές διεπαφές που χρησιμοποιούνται στη συναρτησιακή σχεδίαση αφού όπως είδαμε η δυνατότητα απεικόνισης των δομών αυτών μας βοηθάει στην οργανωμένη διαχείριση των τιμών που περικλείουν και τελικώς στην ανάπτυξη εύρωστου συναρτησιακού κώδικα. Επιπρόσθετα η κατασκευή ενός συναρτήτη, όπως αποδεικνύεται αποτελεί προπομπό για τη δημιουργία άλλων πλουσιότερων δομών, όπως είναι τα **Applicatives** (Εφαρμοστές) και τα **Monads** (Μονάδες). Οι δομές αυτές αν και διαθέτουν έναν βαθμό δυσκολίας για τη σε βάθος κατανόηση τους, επιτρέπουν στον προγραμματιστή που υιοθετεί τη συναρτησιακή σχεδίαση να χτίζει δομές με προσαρμοσμένα υπολογιστικά περιβάλλοντα που αλληλοεπιδρούν μεταξύ τους σχηματίζοντας χρήσιμες λειτουργικότητες.

Κλείνοντας, αξίζει να σημειωθεί ότι όλα τα παραπάνω (**Functors**), αλλά και αυτά που θα δούμε στη συνέχεια, αποτελούν μία απλή νύξη για την προγραμματιστική διάσταση στην οποία μπορούν να επεκταθούν. Πόσο μάλλον για τη μαθηματική. Παρά ταύτα και λαμβάνοντας υπόψιν το γεγονός ότι μία πλήρης τεκμηριωμένη μελέτη για τα (Functors Applicatives, Monoids, Monads) θα αποτελούσε από μόνη της διπλωματική εργασία, αποτελεί μία καλή αρχή για κάποιον που θέλει να πάρει μία ιδέα και να εφαρμόσει τις αρχές στη γλώσσα της JavaScript. Παρακάτω λοιπόν θα δούμε τι είναι τα **Monoids.**

4.12 Μονοειδή (Monoids)

Τα μονοειδή αποτελούν μία πανίσχυρη έννοια που έχει σημαντική απήχηση στην προγραμματιστική θεωρία, ειδικότερα στη συναρτησιακή. Φυσικά όπως και όλες οι προηγούμενες προσεγγίσεις τα μονοειδή στηρίζονται στη μαθηματική θεωρία και όπως θα δούμε παρακάτω, η πρώτη γνωριμία μας με αυτά γίνεται σε μικρή ηλικία. Ο σχηματισμός ενός μονοειδούς επιτυγχάνεται όταν διαθέτουμε μία δυαδική πράξη που επεμβαίνει σε κάποια αντικείμενα ορισμένης μορφής και παράγει αντικείμενα της ιδίας. Για παράδειγμα η πράξη της πρόσθεσης ενός σετ αριθμών, έστω $\mathbb{N}$, σχηματίζει ένα μονοειδές καθώς

$$a + b = c\,, \quad a, b, c \in \mathbb{N}$$

Τι γίνεται όμως με την πράξη της αφαίρεσης για παράδειγμα; Σύμφωνα με τον παραπάνω ορισμό η αφαίρεση σχηματίζει και αυτή ένα μονοειδές σε ένα σετ αριθμών. Η απάντηση όμως είναι αρνητική καθώς πέρα από την ύπαρξη μίας δυαδικής πράξης ή συνάρτησης που επεμβαίνει και παράγει ομοιόμορφα στοιχεία, υπάρχουν και άλλες προϋποθέσεις που όπως θα δούμε δεν καλύπτει η αφαίρεση αλλά ισχύουν για την πρόσθεση. Συνεπώς ο σχηματισμός ενός μονοειδούς υλοποιείται όταν διαθέτουμε μία δυαδική πράξη (συνάρτηση) που επεμβαίνει και παράγει ομοιόμορφα στοιχεία, αλλά επιπρόσθετα η πράξη αυτή ακολουθεί και την προσεταιριστική ιδιότητα (**associativity**). Τέλος πρέπει αυτή η πράξη πρέπει να διαθέτει και ένα ουδέτερο στοιχείο που αν εισαχθεί ως παράμετρος στην πράξη, η πράξη εξουδετερώνεται και επιστρέφεται η αρχική παράμετρος. Ας δούμε αναλυτικότερα την πρόσθεση (**Sum**). Η πρόσθεση ως γνωστόν τηρεί την προσεταιριστική ιδιότητα αφού γνωρίζουμε ότι

$$a + b + \ c = (a + b) + c = a + (b + c)\,, \quad a, b, c \in \mathbb{N}$$

Απομένει και η δήλωση ενός ουδέτερο στοιχείου. Το ουδέτερο στοιχείο της πρόσθεσης στο σύνολο $\mathbb{N}$ (και στο σύνολο των πραγματικών $\mathbb{R}$) είναι ο αριθμός 0 καθότι η πράξη της πρόσθεσης δύο ορισμάτων εκ των οποίων το ένα είναι ο αριθμός 0, έχει ως αποτέλεσμα την επιστροφή του άλλου ορίσματος.

$$\mathrm{a} + 0 = 0 + a = a\,, \quad a \in \mathbb{N}$$

Μία άλλη πράξη που σχηματίζει ένα μονοειδές στο σετ των φυσικών αριθμών είναι η πράξη του πολλαπλασιασμού καθώς αφενός τηρείται η προσεταιριστική ιδιότητα και αφετέρου ο πολλαπλασιασμός διαθέτει ως ουδέτερο στοιχείο τον αριθμό 1. Ισχύουν δηλαδή

$$a * b * \ c = (a * b) * c = a * (b * c)\,, \quad a, b, c \in \mathbb{N}$$

και

$$\mathrm{a} * 1 = 1 * a = a\,, \quad a \in \mathbb{N}$$

Ωραία όλα αυτά όμως ποια η προγραμματιστική απήχηση που έχουν οι μονοειδής σχηματισμοί; Αποδεικνύεται ότι η υλοποίηση δύο συναρτήσεων που αντιπροσωπεύουν τους δύο παραπάνω νόμους προσφέρουν τη δυνατότητα της μεταφοράς των ιδιοτήτων ενός μονοειδούς στο προγραμματιστικό πεδίο. Οι υπογραφές των συναρτήσεων αυτών στη συναρτησιακή γλώσσα της Haskell για παράδειγμα έχουν την παρακάτω μορφή

```
class Monoid a where
    mempty  :: a
    mappend :: a -> a -> a

    mconcat :: [a] -> a
    mconcat = foldr mappend mempty
```

Οι δύο συναρτήσεις (**mempty**, **mappend**) αντιπροσωπεύουν τους νόμους που είδαμε παραπάνω. Η συνάρτηση **mempty** δηλαδή επιστρέφει το ουδέτερο στοιχείο της συνάρτησης **mappend**. Η συνάρτηση **mappend** αποτελεί τη δυαδική πράξη που δέχεται δύο αντικείμενα που ανήκουν στο σύνολο του μονοειδούς που μελετάται και επιστρέφει ένα νέο. Η συνάρτηση **mconcat**, η υλοποίηση της οποίας παρέχεται, αποτελεί άμεσο παράγωγο (ιδιότητα) των δύο προηγούμενων κανόνων. Θα μελετήσουμε πιο κάτω τη λειτουργικότητα της.

Ας δούμε όμως στην πράξη ένα παράδειγμα που σχηματίζει ένα μονοειδή. Λόγος γίνεται για τον γνωστό τύπο δεδομένων **String.** Όπως αποδεικνύεται ο τύπος των αλφαριθμητικών σχηματίζει ένα μονοδειδή καθώς σε όλες τις γλώσσες, υπάρχει η δυνατότητα συνάθροισης των αλφαρηθμητικών (πράξη mappend) αλλά και η ύπαρξη αλφαρηθμητικού που λειτουργεί ως ουδέτερο στοιχείο. Έχουμε δηλαδή

$$mempty :: String$$
$$mempty = ""$$

και

$$mappend :: String \rightarrow String \rightarrow String$$
$$mappend = +$$

Η συνάρτηση **mappend** φυσικά έχει διαφορετικό συμβολισμό αναλόγως τη γλώσσα προγραμματιστικής γραφής. Στη γλώσσα της JavaScript η πράξη **mappend** είναι γνωστή και ως **concat** ή **concatenation** και επιτυγχάνεται μέσω του συμβόλου της μαθηματικής πρόσθεσης, δηλαδή (+). Αν αναλύσουμε στην πράξη τα παραπάνω θα δούμε ότι οι νόμοι ισχύουν κανονικά. Έστω ότι έχουμε

```
const a = "random1";
const b = "random2";
const c = "random3";
// mempty :: String
const mempty = "";

// mappend :: String -> String -> String
const mappend = s1 => s2 => s1 + s2;
```

όπου **mappend** η δυαδική πράξη, και **mempty**, το ουδέτερο στοιχείο των αλφαριθμητικών. Τότε για την προσεταιριστική ιδιότητα καθώς και για το ουδέτερο στοιχείο έχουμε

```
mappend(mappend(a)(b))(c) == mappend(a)(mappend(b)(c))
//-> "random1random2random3"

mappend(a)(mempty) == mappend(mempty)(a)
//-> "random1random2random3"
```

αφού

```
("random1" + "random2") + "random3" == "random1" + ("random2" + "random3") ==
"random1" + "random2" + "random3"
```

και

```
"random1" + "" == "" + "random1" == "random1"
```

Παρατηρούμε δηλαδή ότι η γνωστή πράξη της συνάθροισης (**mappend**) που χρησιμοποιούμε σε μεγάλο βαθμό οι προγραμματιστές από τα πρώτα βήματα μας στην επιστήμη των υπολογιστών είναι μια αυστηρά δομημένη μαθηματική έννοια. Ένα άλλο γνωστό παράδειγμα δομής που σχηματίζει μονοειδή κατηγορία είναι οι γνωστές σε όλους μας λίστες ή πίνακες. Η συνάθροιση πινάκων που επιτυγχάνεται συνήθως από μία μέθοδο που προσφέρει εγγενώς μία γλώσσα, συνήθως αντικειμενοστρεφής προσέγγισης, τηρεί την προσεταιριστική ιδιότητα και παράγει έναν νέο πίνακα. Το ουδέτερο στοιχείο αυτής της πράξης αποτελεί ο κενός πίνακας και συμβολίζεται συνήθως με το σύμβολο [ ]. Για παράδειγμα στην JavaScript έχουμε

```
const a = [1, 2, 3, 4];
const b = [5, 6, 7, 8];
const c = [9, 10, 11, 12];
// mempty :: [Number]
const mempty = [];

// mappend :: [Number] -> [Number] -> [Number]
const mappend = concat
```

Η πράξη **concat** αποτελεί τη μέθοδο που κληρονομούν όλοι οι πίνακες από το αντικείμενο **Array.prototype,** με τη διαφορά ότι είναι απαλλαγμένη από το περιβάλλον του αντικειμένου από το οποίο καλείται (object context). Για την υλοποίηση της μπορούμε να χρησιμοποιήσουμε τη βοηθητική συνάρτηση υψηλής τάξης **unmethodify,** που επιτελεί ακριβώς αυτό τον μετασχηματισμό. Μπορούμε να γράψουμε λοιπόν

```
const unmethodify = fn => a1 => a2 => fn.bind(a1, a2)();

const concat = unmethodify(Array.prototype.concat);
```

Που συνακόλουθα δημιουργεί την παρακάτω ισοδυναμία

```
[1, 2, 3].concat([4, 5, 6]) == concat([1, 2, 3])([4, 5, 6])
//-> [1, 2, 3, 4, 5, 6]
```

Έτσι η πράξη **mappend** υπακούει στην προσεταιριστική ιδιότητα, αφού

```
mappend(mappend(a)(b))(c) == mappend(a)(mappend(b)(c))
// -> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

mappend(a)(mempty) == mappend(mempty)(a)
// -> [1, 2, 3, 4]
```

Επιπρόσθετα παρατηρούμε ότι η συνάθροιση ενός πίνακα **a** με το ουδέτερο στοιχείο, δηλαδή τον κενό πίνακα [ ], επιστρέφει τον αρχικό πίνακα **a**.

Ένα από τα βασικά προτερήματα των μονοειδών σχηματισμών που πηγάζει από την προσεταιριστική ιδιότητα (**associativity**) είναι η δυνατότητα αναδρομικής προσέγγισης στην υλοποίηση σύνθετων (εμφωλευμένων) κλήσεων της συνάρτησης **mappend.** Μάλιστα το γεγονός ότι η πράξη **mappend** είναι πλήρως προσεταιριστική και έχει την υπογραφή ενός **reducer function** επιτρέπει τη χρήση των δομών δίπλωσης **foldl**, **foldr,** που με τη βοήθεια ενός πίνακα προσομοιώνουν πετυχημένα τον εκάστοτε προσεταιρισμό (left or right). Επιπρόσθετα, όπως και στο δομή της διασυνδεμένης λίστας **pair** (που όπως θα δούμε σχηματίζει και αυτή ένα μονοειδή), δύναται η παραγωγή δύο μορφών αναδρομής. Η πρώτη προσομοιώνει την αριστερά προσεταιριστική ιδιότητα (left

associative) παράγοντας μία διάταξη πού είναι **tail-case optimized,** ενώ η δεύτερη προσέγγιση επιτρέπει την παραγωγή μίας διάταξης που μπορεί να χαρακτηριστεί και ως **lazy**, καθυστερώντας την αποτίμηση της ολοκληρωμένης έκφρασης. Για παράδειγμα για να βρούμε το άθροισμα μίας ακολουθίας φυσικών αριθμών με αρχικό στοιχείο το 0, και τελικό το n, μπορούμε να γράψουμε τις δύο ισοδύναμες ως προς το αποτέλεσμα εκφράσεις.

```
// mempty :: Number
let mempty = 0;

// mappend :: Number -> Number -> Number
const mappend = n1 => n2 => n1 + n2;

// Sum :: Number -> Number
const Sum = n => n < 0 ? mempty : mappend(n)(Sum(n - 1));

Sum(100);
//-> 5050
```

Η παραπάνω έκφραση εκτελεί την πράξη του αθροίσματος με δεξιά προσεταιρισμό, δηλαδή

$$a = \Big(100 + \big(99 + \big(98 + \cdots \big(1 + (0)\big) \ldots\big)\big)\Big)$$

Αντιθέτως η παρακάτω έκφραση χρησιμοποιεί αριστερό προσεταιρισμό

```
// mempty :: Number
let mempty = 0;

// mappend :: Number -> Number -> Number
const mappend = n1 => n2 => n1 + n2;

// Sum :: Number -> Number
const Sum = n => n < 0 ? mempty : mappend(Sum(n-1))(n);

Sum(100);
//-> 5050
```

Δηλαδή

$$b = \Big(\big(\ldots\big(\big(\big((0) + 1\big) + 2\big) + 3\big) \ldots + 99\big) + 100\Big)$$

Η διαφορά ανάμεσα στις δύο είναι φαίνεται παρακάτω

Η πρώτη έκφραση μπορεί να χρησιμοποιήσει την τεχνική της καθυστερημένης αποτίμησης

```
const Sum = n => n < 0 ? mempty : mappend(n)(/* lazy */ () => Sum(n - 1));
```

Μεταβάλλοντας φυσικά την υπογραφή της συνάρτησης **mappend**. Η δεύτερη μορφή από την άλλη μπορεί να χρησιμοποιήσει έναν συσσωρευτή των αποτελεσμάτων της mappend, για να παράξει μία **tail-case optimized** (αποφυγή του προβλήματος της υπερχείλισης της στοίβας) υλοποίηση της ζητούμενης συνάρτησης. Μπορούμε να γράψουμε λοιπόν

```
const Sum = (n, mempty = 0) => n < 0 ? mempty : Sum(n - 1, mappend(n)(mempty));
```

Για την ακρίβεια η νέα Sum αναπαριστά την παρακάτω πρόταση

$$\Big(\big(\ldots\big(\big(\big((0)+100\big)+99\big)+98\big)\ldots+1\big)+0\Big)$$

Αλλά αυτό έχει να κάνει με το γεγονός ότι η συνθήκη ξεκινάει από τα υψηλά νούμερα και καταλήγει στα χαμηλά, με αποτέλεσμα το σταδιακό χτίσιμο της αναπαράστασης να ξεκινάει από τον αριθμό 100. Το σημαντικό είναι ότι οτιδήποτε παρουσιάζει αριστερό προσεταιρισμό, μπορεί να υλοποιηθεί ως **tail-case optimized.** Οι παραπάνω αναπαραστάσεις μπορούν να γραφούν και κάνοντας χρήση των συναρτήσεων **foldr** και **foldl** αντίστοιχα. Έστω δηλαδή ότι έχουμε τις υλοποιήσεις **folding** για τη δομή του πίνακα, όπως απεικονίζεται παρακάτω

```
// foldr :: (a -> b -> b, b, [a]) -> b
const foldr = pcurryN((f, z, [first, ...rest]) => first === undefined ? z :
f(first)(foldr(f, z, rest)));

// foldl :: (b -> a -> b) -> b -> [a] -> b
const foldl = pcurryN((f, z, [first, ...rest]) => first === undefined ? z :
foldl(f, f(z)(first), rest));
```

Οι υλοποιήσεις εισέρχονται στη συνάρτηση υψηλής τάξης **pcurryN**, έτσι ώστε να αποκτήσουν την αντίστοιχη ιδιότητα. Οι αναπαραστάσεις των αθροισμάτων λοιπόν μπορούν να γραφούν ισοδύναμα και με τους παρακάτω τρόπους. Η μορφή **a** αντιστοιχεί στην παρακάτω αναπαράσταση

```
foldr(mappend)(mempty)([100, 99, …, 1, 0]);
```

Ενώ η μαθηματική αναπαράσταση **b** ισοδυναμεί με την παρακάτω έκφραση στην JavaScript

```
foldl(mappend)(mempty)([100, 99, …, 1, 0]);
```

Μάλιστα αν ξαναδούμε τις συναρτήσεις που πρέπει να υλοποιεί ένας σχηματισμός μονοειδούς θα δούμε ότι παρατίθεται και η συνάρτηση **mconcat,** η υλοποίηση της οποίας είναι σταθερή και ισχύει για κάθε μονοειδής κατηγορία. Όπως εύλογα μπορεί να συμπεράνει ο αναγνώστης η παράσταση

```
foldr(mappend)(mempty)([100, 99, …, 1, 0]);
```

επιτελεί ακριβώς την πράξη που ορίζει η συνάρτηση **mconcat,** δηλαδή

```
mconcat = foldr mappend mempty
```

Το μαγικό της υπόθεσης έγκειται στο γεγονός ότι όλες οι παραπάνω ιδιότητες ισχύουν σε οποιοδήποτε σχηματισμό ακολουθεί τη μονοειδή κατηγορία, προσφέροντας έτσι στον συναρτησιακό προγραμματισμό ένα ισχυρό μοντέλο παραγωγής δομών που ενθαρρύνουν τη μεταξύ τους σύνθεση (Milewski, 2018) και επιτρέπουν τη δημιουργία ελαστικών διατάξεων.

Ένας προσεκτικός αναγνώστης θα παρατηρούσε ότι οι παραπάνω υλοποιήσεις μοιάζουν αρκετά με τα παραδείγματα που είδαμε στη δομή της διασυνδεμένης λίστας **Pair**. Πράγματι είχαμε αναφέρει ότι η ευελιξία που προσφέρει η δομή της διασυνδεμένης λίστας οφείλεται αφενός στο ότι μπορεί να λειτουργήσει ως συναρτήτης, και αφετέρου στο ότι ορίζεται αναδρομικά χτίζοντας έτσι έναν μονοειδή σχηματισμό. Η δομή pair άλλωστε δεν διαφέρει καθόλου από τους πίνακες αφού σε κάποιες γλώσσες οι πίνακες αναπαρίστανται μέσω της δομής **cons**, δηλαδή ακριβώς το ίδιο δομικό στοιχείο που χρησιμοποιήσαμε για τη δημιουργία της δομής **Pair.** Ας εξετάσουμε λοιπόν πως υλοποιούνται οι ζητούμενες συναρτήσεις **mempty** και **mappend** για την **Pair.** Έχουμε

```
const pair = x => y => s => s(x)(y);

const K = x => y => x;
const I = x => x;

const car = p => p(K);
const cdr = p => p(K(I));

// prepend :: Number -> Pair Number -> Pair Number
const prepend = e => p => pair(e)(p);

const a = pair(1)(pair(2)(pair(3)(pair(4)(null))));
const b = pair(5)(pair(6)(pair(7)(pair(8)(null))));
const c = pair(9)(pair(10)(pair(11)(pair(12)(null))));

// mempty :: null
const mempty = null;

// concat :: Pair Number -> Pair Number -> Pair Number
const concat = p1 => p2 => p1 === null ? p2 :
prepend(car(p1))(concat(cdr(p1))(p2));

// mappend :: Pair Number -> Pair Number -> Pair Number
const mappend = concat;

mappend(mappend(a)(b))(c) == mappend(a)(mappend(b)(c))
// ->
pair(1)(pair(2)(pair(3)(pair(4)(pair(5)(pair(6)(pair(7)(pair(8)(pair(9)(pair(10)(
pair(11)(pair(12)(null))))))))))))

mappend(a)(mempty) == mappend(mempty)(a)
// -> pair(1)(pair(2)(pair(3)(pair(4)(null))))
```

Βλέπουμε ότι η δυαδική πράξη **mappend** είναι πλήρως προσεταιριστική (associative) και ουδέτερο στοιχείο του μονοειδούς σχηματισμού ορίζουμε τον δείκτη **null.** Αυτομάτως υπάρχει η δυνατότητα εκμετάλλευσης της συνάρτησης **mconcat** που επιτρέπει τη δίπλωση ενός πίνακα με **pairs**, σε μία ολοκληρωμένη δομή της ίδιας προέλευσης, δηλαδή **pair**. Για παράδειγμα μπορεί να γραφεί

```
const a = pair(1)(pair(2)(pair(3)(pair(4)(null))));
const b = pair(5)(pair(6)(pair(7)(pair(8)(null))));
const c = pair(9)(pair(10)(pair(11)(pair(12)(null))));

// foldr :: (a -> b -> b, b, [a]) -> b
const foldr = pcurryN((f, z, [first, ...rest]) => first === undefined ? z :
f(first)(foldr(f, z, rest)));

// mconcat :: [a] -> a
const mconcat = foldr(mappend)(mempty);

mconcat([a, b, c]);
// ->
pair(1)(pair(2)(pair(3)(pair(4)(pair(5)(pair(6)(pair(7)(pair(8)(pair(9)(pair(10)(
pair(11)(pair(12)(null))))))))))))
```

Ισοδύναμα στη θέση της **foldr** θα μπορούσε να χρησιμοποιηθεί η συνάρτηση **foldl**, έχοντας το ίδιο αποτέλεσμα. Η προκαθορισμένη υλοποίηση όμως συνιστά τη χρήση της **foldr**, καθώς όπως είπαμε μπορεί να εκμεταλευτεί την καθυστερημένη αποτίμηση. Επίσης ο δεξιά προσεταιρισμός σε πολλούς μονοειδής σχηματισμούς, μεταξύ αυτών και η δομή **pair**, είναι πιο γρήγορος καθώς αποφεύγεται η επαναδημιουργία ολόκληρης της πρώτης παραμέτρου pair στη συνάρτηση mappend. Αν παρατηρήσουμε προσεκτικά η συνάρτηση mappend για τη δομή **pair**, δημιουργεί έναν κλόνο της πρώτης παραμέτρου, έστω p1, και στη συνέχεια συναθροίζει άμεσα τη δεύτερη δομή pair που εισάγεται ως παράμετρος, έστω p2. Έτσι σε κάθε επανάληψη είναι προτιμότερο να χτίζουμε τη διάταξη μας στη δεύτερη παράμετρο (συσσωρευτής) και όχι στην πρώτη, όπως πράττει δηλαδή η λειτουργικότητα **foldl.**

Προτού δούμε άλλη μια σημαντική παρατήρηση, αξίζει να σημειωθεί ότι με τον ίδιο ακριβώς τρόπο, μπορούμε να ισχυριστούμε ότι η δομή **Lazy Pair (Stream)** σχηματίζει και αυτή μία μονοειδή κατηγορία. Ο ισχυρισμός μας θα ήταν ορθός, καθώς η διάταξη **Lazy Pair** στηρίζεται για τη δημιουργία της πλήρως στη δομή **Pair** και συνεπώς κληρονομεί όλα τα χαρακτηριστικά της. Στην πραγματικότητα η **Lazy Pair** προκύπτει από την αδυναμία έκφρασης με διαφορετικό τρόπο στη γλώσσα της JavaScript της τεχνικής της καθυστερημένης αποτίμησης. Σε συναρτησιακές γλώσσες η **Lazy Pair** θα ήταν η δομή **Pair** με τη διαφορά ότι η παράμετρος που αντιπροσωπεύει την ουρά της διασυνδεμένης λίστας θα δηλώνονταν ως **lazy.**

Στην αρχή του 3ου κεφαλαίου είδαμε μία σημαντική ιδιότητα των συναρτήσεων, αυτή της σύνθεσης. Η υπογραφή της συνθετικής πράξης είδαμε ότι δέχεται ως ορίσματα δύο συναρτήσεις (με ορθή διάταξη των τύπων) και επιστρέφει μία νέα συνάρτηση. Μήπως και η πράξη της σύνθεσης σχηματίζει μονοειδής σχηματισμό στο σύνολο των συναρτήσεων; Η απάντηση είναι θετική καθώς τηρούνται όλες οι προϋποθέσεις που ορίζει η θεωρία μας. Αποδεικνύεται λοιπόν ότι η σύνθεση συναρτήσεων αποτελεί την πράξη **mappend** στον μονοειδή σχηματισμό, ενώ ως ουδέτερο στοιχείο ορίζεται, όπως πολλοί μαντεύετέ, η συνάρτηση **id** ή ο ταυτοτικός μορφισμός όπως λέμε. Έχουμε δηλαδή

$$mempty :: a \to a$$

$$mempty = id$$

και

$$mappend :: (b \to c) \to (a \to b) \to (a \to c)$$

$$mappend = \circ$$

Στην προγραμματιστική τους διάσταση η παραπάνω συνθήκες μπορούν να μεταφραστούν με τον παρακάτω τρόπο

```
const id = x => x;

const compose = f => g => x => f(g(x));

// mempty :: a -> a
const mempty = id;

// mappend :: (b -> c) -> (a -> b) -> (a -> c)
const mappend = compose;
```

Παρατηρούμε λοιπόν ότι στον συναρτησιακό προγραμματισμό οι συναρτήσεις εκφράζονται ως πράξεις ή ως δεδομένα αφού πολλές φορές υλοποιούν ίδιες λειτουργικότητες με τις παραδοσιακές, όπως έχουμε στο μυαλό μας, δομές δεδομένων όπως για παράδειγμα οι πίνακες. Στην προκειμένη περίπτωση αποδείξαμε ότι συνθέτουν έναν μονοειδή σχηματισμό διά της πράξης της σύνθεσης. Επιπρόσθετα, αν και αποφύγαμε να το αποδείξουμε στο υποκεφάλαιο των συναρτητών, αξίζει να σημειωθεί ότι οι συναρτήσεις μπορούν να εκφραστούν και ως συναρτήτες προσφέροντας επίσης μία υλοποίησης της fmap που ισοδυναμεί με τη συνθετική πράξη ∘ (**compose**).

4.13 Μονάδες (Monads)

Κατά την ανάλυση των συναρτητών (**functors**) είδαμε ότι είναι εφικτή η δημιουργία ενός υπολογιστικού πλαισίου, οι ιδιότητες του οποίου παρέχονται μέσω μίας συνάρτησης απεικόνισης **fmap.** Το ερώτημα στο οποίο απάντα αυτή η συνάρτηση τίθεται ως εξής: Όταν έχουμε μία συνάρτηση της μορφής $\boldsymbol{a \rightarrow b}$ και μία δομή τύπου $\boldsymbol{F\ a}$, τότε με ποιο τρόπο μπορούμε να απεικονίσουμε τη συνάρτηση μας στο σχηματισμό $\boldsymbol{F\ a}$ για να παραγάγουμε μία νέα δομή $\boldsymbol{F\ b}$. Η ακριβής υλοποίηση λοιπόν της συνάρτησης **fmap** είναι αυτή που δίνει υπόσταση στον συναρτήτη και ορίζει τη γενική περιγραφή του υπολογιστικού περιβάλλοντος που σχηματίζει. Οι μονάδες (**monads**), όπως και οι συναρτήτες πηγάζουν από τα μαθηματικά και συγκεκριμένα τη θεωρία κατηγοριών. Στην πραγματικότητα οι μονάδες αποτελούν ειδική κατηγορία συναρτήτων, που μπορούν να χρησιμοποιηθούν στη συναρτησιακή προγραμματιστική προσέγγιση για το σχηματισμό δομημένων προγραμμάτων που είναι σε θέση να προσδώσουν ένα στρώμα αφαίρεσης αποκρύπτοντας σε μεγάλο βαθμό την περίπλοκη λογική τους. Όπως και οι συναρτήτες, οι μονάδες δημιουργούν ένα υπολογιστικό περιβάλλον στο οποίο παρέχουν λύσεις σύνθεσης με άλλα ίδιας μορφής περιβάλλοντα (**Applicatives**) και επιπρόσθετα δίνουν τη δυνατότητα απεικόνισης τους με συναρτήσεις της μορφής $\boldsymbol{a \rightarrow M\ b}$. Το γεγονός αυτό επιτρέπει την απλοποίηση ενός μεγάλου εύρους προβλημάτων μετασχηματίζοντας μία αλληλουχία προγραμματιστικών ενεργειών σε μία σωληνοειδής διάταξη που αφαιρεί αποδοτικά την περιττή διαχείριση των δεδομένων (data-management), τον έλεγχο ροής (control flow) καθώς και πιθανές παρενέργειες (side-effects). Ως μονάδα λοιπόν ορίζουμε το υπολογιστικό πλαίσιο που περικλείει έναν τύπο δεδομένων και διαθέτει μηχανισμούς υλοποίησης δύο λειτουργιών που εκφράζονται με τη μορφή συναρτήσεων. Στη συναρτησιακή γλώσσα **Haskell**, για παράδειγμα, η δομή της μονάδας πρέπει να υλοποιεί τις εξής δύο συναρτήσεις.

```
class Monad m where
 return :: a -> M a
 (>>=)  :: M a -> (a -> M b) -> M b
```

Η πρώτη ονομάζεται **return** (δεν έχει καμία σχέση με τη δεσμευμένη λέξη που παρέχουν οι γνωστές γλώσσες προγραμματισμού για την επιστροφή μίας τιμής εντός μίας συνάρτησης), και αποτελεί τον προκαθορισμένο τρόπο εγκιβωτισμού μίας τιμής τύπου **a** εντός ενός σχηματισμού μονάδας **m**. Η δεύτερη συνάρτηση αποτελεί την πεμπτουσία των μονάδων και αποτελεί, όπως φαίνεται και από την υπογραφή της, έναν τρόπο απεικόνισης μίας δομής μονάδας σε συναρτήσεις με υπογραφή $\boldsymbol{a \rightarrow M\ b}$**.** Ο μηχανισμός αυτός είναι γνωστός ως συνάρτηση **chain** ή **flatmap** ή **bind,** και η υλοποίηση αποτυπώνει καθοριστικά την απάντηση στο εξής ερώτημα: Αν έχουμε ένα δεδομένο κάποιας μορφής **a** εντός ενός υπολογιστικού πλαισίου, δηλαδή έναν σχηματισμό $\boldsymbol{M\ a,}$ πως ακριβώς θα μετασχηματιστεί (transformation) ο σχηματισμός αυτός εάν τον τροφοδοτήσουμε με μία συνάρτηση της μορφής $\boldsymbol{a \rightarrow M\ b}$**.** Η προϋπόθεση φυσικά είναι το αποτέλεσμα της συνάρτησης αυτής να επιστρέφει μία μονάδα $\boldsymbol{M\ b}$**.** Αλλά όπως αποδεικνύεται η διαδικασία αλληλεπίδρασης των τιμών που αποκρύπτουν τα υπολογιστικά πλαίσια $\boldsymbol{M}$ καθορίζεται πλήρως από τα υπολογιστικά αυτά περιβάλλοντα δίνοντας έτσι τη δυνατότητα απόκρυψης σύνθετων υπολογιστικών διαδικασιών πίσω από την υλοποίηση αυτής της συνάρτησης. Όπως και οι συναρτήτες, οι δομές των μονάδων πρέπει να ακολουθούν ορισμένους νόμους, η τήρηση των οποίων προσφέρει ένα ενιαίο πλαίσιο συμπερασμάτων πάνω στο οποίο μπορούν να βασιστούν οι λειτουργικότητες που σχηματίζονται και εμπλέκονται με αυτές τις δομές. Πάμε λοιπόν να δούμε ποιοι είναι αυτοί.

- Ο πρώτος νομός ορίζει την αριστερά ταυτοτική ιδιότητα (**left identity**) και αναφέρει η έκφραση $return\ x \gg= f$ πρέπει να είναι ισοδύναμη με την έκφραση $f\ x$, δηλαδή

$$return\ x \gg= f \equiv f\ x$$

Με άλλα λόγια η δημιουργία ενός υπολογιστικού περιβάλλοντος γύρω από ένα δεδομένο $\boldsymbol{x}$ και η τροφοδότηση μίας συνάρτησης $\boldsymbol{f}$ στο υπολογιστικό περιβάλλον μέσω της συνάρτησης **chain**, επιστρέφει το ίδιο ακριβώς αποτέλεσμα που θα προέκυπτε από εφαρμογή του δεδομένου $\boldsymbol{x}$ στη συνάρτηση $\boldsymbol{f}$**.**

- Ο δεύτερος νόμος ορίζει τη δεξιά ταυτοτική ιδιότητα (**right identity**) και αναφέρει η έκφραση $M \gg= return$ πρέπει να είναι ισοδύναμη με την έκφραση m, δηλαδή

$$M \gg= return \equiv M$$

Με άλλα λόγια η τροφοδότηση της συνάρτησης **return** μέσω της **chain** σε μία μονάδα ***M*,** επιστρέφει ως αποτέλεσμα μία ολόιδια μονάδα ***M*.**

- Ο τρίτος νόμος ορίζει την προσεταιριστική (associativity) ιδιότητα που πρέπει να διέπει τη συνάρτηση **chain** και αναφέρει ότι η έκφραση $M \gg= f \gg= g$ πρέπει να είναι ισοδύναμη με την έκφραση $M \gg= (\lambda x. f\ x \gg= g)$, δηλαδή

$$M \gg= f \gg= g \equiv\ M \gg= (\lambda x. f\ x \gg= g)$$

Με άλλα λόγια η τροφοδότηση μίας μονάδας ***M*** μέσω της **chain** με μία συνάρτησης ***f*** και στη συνέχεια η τροφοδότηση του αποτελέσματος με τη συνάρτηση ***g*,** πρέπει να είναι ισοδύναμη με την τροφοδότηση της ***M*** με μία συνάρτηση που τροφοδοτεί το αποτέλεσμα της κλήσης ***f x*** με μία συνάρτηση ***g*.** Θα δούμε παρακάτω τις συνέπειες αυτού του νόμου.

Αφού αναλύσαμε το θεωρητικό πλαίσιο γύρω από τις μονάδες, πάμε να δούμε κάποια παραδείγματα αυτών για να αντιληφθούμε πως μπορούμε να μεταφέρουμε τα παραπάνω στη γλώσσα της JavaScript. Ένα από τα πιο γνωστά παραδείγματα δομών που υλοποιούν έναν σχηματισμό μονάδας προσφέροντας υλοποίηση στις συναρτήσεις **return** και **chain** είναι η δομή του πίνακα. Ο πίνακας εκτός της ιδιότητας **fmap** που διαθέτει αφού μπορεί να αντιμετωπιστεί ως συναρτήτης, παρέχει ένα υπολογιστικό πλαίσιο το οποίο γνωρίζει με ποιο τρόπο να εφαρμόζει συναρτήσεις της μορφής $\boldsymbol{a} \rightarrow [\boldsymbol{b}]$.

Ας υποθέσουμε ότι έχουμε έναν αυθαίρετο πίνακα με τιμές. Στο υποκεφάλαιο των συναρτητών αναφέραμε ότι η δομή ενός πίνακα αποκρύπτει τον μη ντετερμινισμό, ομαδοποιώντας αποδοτικά ισοπίθανες τιμές που μπορεί να αποδοθεί σε μία έκφραση. Έχοντας αυτό στο μυαλό μας πάμε να δούμε πως θα υλοποιήσουμε τις ζητούμενες συναρτήσεις, για τη δομή του πίνακα, στη γλώσσα της JavaScript.

Η συνάρτηση **return** (στο παρόν παράδειγμα θα την ονομάσουμε **pure** καθώς η λέξη **return** είναι δεσμευμένη. Η ονομασία **pure** προέρχεται από τη δομή των **Applicatives** και προσφέρει παρόμοια λειτουργικότητα) είπαμε ότι τοποθετεί μία τιμή εντός του υπολογιστικού πλαισίου της δομής της μονάδας που θέλουμε να σχηματίσουμε, οπότε θα μπορούσαμε να ισχυριστούμε ότι για οποιαδήποτε τιμή **c** ανεξαρτήτως τύπου, η συνάρτηση return θα επιστρέφει την τιμή **[c].** Άρα θα έχουμε

```
// return/pure :: a -> [a]
const pure = x => [x];
```

Έτσι αν έχουμε για παράδειγμα το αλφαριθμητικό "random" η συνάρτηση **pure** θα επιστρέψει την ίδια τιμή εντός ενός πίνακα. Για $a = String$ λοιπόν έχουμε

```
pure("random");
//-> ["random"]
```

Η συνάρτηση **chain** από την άλλη έχει πιο σύνθετη λειτουργικότητα. Αναφέραμε ότι η συνάρτηση αυτή προσδίδει έναν τρόπο απεικόνισης μίας δομής μονάδας σε συναρτήσεις της μορφής $\boldsymbol{a \rightarrow M\ b}$**.** Ας δούμε αρχικά την υπογραφή της συνάρτησης για τη δομή του πίνακα. Αν όπου $\boldsymbol{M}$ βάλουμε τον κατασκευαστή τύπων (type constructor) **Array** ή **List** ή **[ ]** (literals στην JavaScript) θα έχουμε την υπογραφή της chain με την παρακάτω μορφή

$$chain :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$$

Αν λάβουμε υπόψιν το γεγονός ότι ένας πίνακας παραθέτει ομαδοποιημένες ισοπίθανες τιμές που μπορούν να αποδοθούν σε μία έκφραση, τότε η απεικόνιση όλων των στοιχείων του πίνακα $[\boldsymbol{a}]$ μέσω μίας συνάρτηση map (fmap) με τη δοθείσα συνάρτηση της μορφής $\boldsymbol{a \rightarrow [b]}$ θα ήταν μία δικαιολογημένη απόπειρα. Για παράδειγμα για $a, b = Number$ έστω ότι έχουμε έναν πίνακα με τους αριθμούς 1 έως 3 και μία συνάρτηση που απεικονίζει τα στοιχεία ενός πίνακα με τους αριθμούς από 6 έως 10. Έχουμε στην Javascipt

```
// p :: [Number]
const p = [1, 2, 3];

// f :: Number -> [Number]
const f = x => [6, 7, 8, 9, 10];
```

Ποιο θα είναι το αποτέλεσμα της απεικόνισης της δομής p με τη συνάρτηση f. Αν χρησιμοποιήσουμε τη συνάρτηση **fmap** του συναρτήτη της δομής του πίνακα (όπως είπαμε όλες οι μονάδες είναι και συναρτήτες) θα έχουμε το παρακάτω αποτέλεσμα

```
// fmap :: (a -> b) -> F a -> F b
const fmap = f => Fa => Fa.map(f);

fmap(f)(p);
//-> [[6, 7, 8, 9, 10], [6, 7, 8, 9, 10], [6, 7, 8, 9, 10]]
```

Παρατηρούμε ότι στη θέση των στοιχείων $1, 2, 3$ έχουμε πλέον τις δομές $[6, 7, 8, 9, 10]$. Όλα καλά μέχρι εδώ αλλά το αποτέλεσμα της **chain** θα πρέπει να επιστρέψει έναν πίνακα της μορφής $[Number]$ και όχι $[[Number]]$. Αν συνυπολογίσουμε και την ερμηνεία που αποδώσαμε στους πίνακες όσον αφορά στην πιθανή ύπαρξη μίας

ομάδας τιμών, θα ήταν συνετή η απόφαση μας να παρουσιάσουμε το τελικό αποτέλεσμα ως $[6,7,8,9,10,6,7,8,9,10,6,7,8,9,10]$ και όχι ως $[[6,7,8,9,10],[6,7,8,9,10],[6,7,8,9,10]]$. Πράγματι το $2^{ο}$ στάδιο της συνάρτησης **chain** και αυτό που έπεται της απεικόνισης της μονάδας μας με τη βοήθεια της fmap, είναι η απαλλαγή από το περίσσιο υπολογιστικό φλοιό που παράγει η εφαρμογή της συνάρτησης $\boldsymbol{f}$ σε μία διαδικασία που ονομάζεται **join** ή **flatten** (από τους πίνακες). Η συνάρτηση **chain** εν ολίγης, όσον αφορά τους πίνακες αλλά και οποιοδήποτε άλλο σχηματισμό μονάδας αποτελεί μία διαδικασία απεικόνισης και ισοπέδωσης και για αυτό τον λόγο έχει υιοθετηθεί και ως δεύτερη ονομασία της **chain** ο όρος **flatmap.** Συνεπώς για τη δομή του πίνακα, η συνάρτηση **chain** μπορεί να αναπαρασταθεί με τον παρακάτω τρόπο

```
// chain :: [a] -> (a -> [b]) -> [b]
const chain = m => f => flatten(fmap(f)(m));
```

Απομένει να δώσουμε μία υλοποίηση στη συνάρτηση **flatten** που θα μετατρέπει ουσιαστικά έναν πίνακα από πίνακες, σε έναν μονοδιάστατο πίνακα. Το βάθος της συνάρτησης, δηλαδή ο βαθμός στον οποίο θα επεμβαίνει η **flatten** αναδρομικά για να ισοπεδώσει τα στοιχεία ενός πίνακα θα είναι ένα (1). Δηλαδή ο πίνακας από πίνακες θα μετατρέπεται σε πίνακα. Ο πίνακας από πίνακες από πίνακες, σε πίνακες από πίνακες κοκ. Έτσι θα έχουμε την παρακάτω υπογραφή

$$flatten :: [[a]] \rightarrow [a]$$

Η υλοποίηση της **flatten** στην JavaScript έχει ως εξής

```
// flatten :: [[a]] -> [a]
const flatten = ([first, ...rest]) => first === undefined ?
    [] : Array.isArray(first) ?
    first.concat(flatten(rest)) : [first].concat(flatten(rest));
```

Επίσης ένας δεύτερος τρόπος γραφής είναι η χρησιμοποίηση της **concat** στα πλαίσια που ορίζει ο μονοειδής σχηματισμός του πίνακα, δηλαδή όπως είδαμε η συνάρτηση **mconcat.** Μπορούμε να γράψουμε ισοδύναμα

```
// flatten :: [[a]] -> [a]
const flatten = foldr(mappend)([]);
```

Όπου **mappend** η συνάρτηση

```
// mappend :: [a] -> [a] -> [a]
const concat = unmethodify(Array.prototype.concat);
const mappend = concat;
```

Επίσης ένας τρίτος τρόπος, που είναι και αυτός που θα χρησιμοποιήσουμε τελικά καθώς υποστηρίζεται από την εγγενή βιβλιοθήκη, είναι μέσω της μεθόδου apply που κληρονομούν οι συναρτήσεις από το αντικείμενο **Function.prototype.** Έχουμε

```
// flatten :: [[a]] -> [a]
const flatten = a => Array.prototype.concat.apply([], a);
```

Συνεπώς τελικά η κλήση της **chain** με παραμέτρους τον πίνακα $[1, 2, 3]$ και τη συνάρτηση $\boldsymbol{f}$ που επιστρέφει τον πίνακα $[6, 7, 8, 9, 10]$ επιστρέφει το αποτέλεσμα που απεικονίζεται παρακάτω

```
// p :: [Number]
const p = [1, 2, 3];

// f :: Number -> [Number]
const f = x => [6, 7, 8, 9, 10];

// chain :: [a] -> (a -> [b]) -> [b]
const chain = m => f => flatten(fmap(f)(m));

chain(p)(f);
//-> [6, 7, 8, 9, 10, 6, 7, 8, 9, 10, 6, 7, 8, 9, 10]
```

Είδαμε λοιπόν ότι η δομή του πίνακα μπορεί να αποτελέσει σχηματισμό μονάδας προσδίδοντας έναν τρόπο υλοποίησης στη συνάρτηση **chain** που απεικονίζει τη συνάρτηση εισόδου και ισοπεδώνει τον παραγόμενο πίνακα από πίνακες, σε μία νέα μονάδα ίδιας μορφής (μονοδιάστατος πίνακας). Προτού δούμε ένα παράδειγμα όπου μας είναι χρήσιμη η λειτουργικότητα της **chain** στη δομή ενός πίνακα, πάμε να δούμε αν η συγκριμένη υλοποίηση τηρεί του νόμους που διέπουν μία μονάδα.

Η αριστερή ταυτοτική ιδιότητα (**left identity**) ορίζει όπως είδαμε την παρακάτω ισοδυναμία

$$return\ x \gg\!= f \equiv f\ x$$

Έστω ότι ορίζουμε ως $\boldsymbol{x}$ ένα αντικείμενο τύπου **Student,** τότε

```
// s :: Student
const s = { name: "Alonzo Church", age: 36 };

// [Student]
pure(s)
//-> [{ name: "Alonzo Church", age: 36 }]
```

Ως συνάρτηση τροφοδότησης της δομής **pure(s)** μέσω της **chain**, δηλαδή ως συνάρτηση $\boldsymbol{f}$ ορίζουμε την παρακάτω συνάρτηση

```
// f :: Student -> [Student]
const f = ({ name, age }) => [{ name: `${name}/Child1`, age: age - 20 }, { name:
`${name}/Child2`, age: age - 22 }];
```

Η συνάρτηση $\boldsymbol{f}$ επιστρέφει έναν νέο πίνακα με τα παιδιά του αρχικού αντικειμένου Student, τα στοιχεία των οποίων για λόγους επίδειξης κουβαλάνε και όνομα του πατέρα

Η έκφραση $return\ x \gg= f$ μεταφράζεται και επιστρέφει στην JavaScript την παρακάτω τιμή

```
chain(pure(s))(f);
//-> [ { name: 'Alonzo Church/Child1', age: 16 },
//     { name: 'Alonzo Church/Child2', age: 14 } ]
```

Ενώ η έκφραση $\boldsymbol{f}\ \boldsymbol{x}$ επιστρέφει

```
f(s);
//-> [ { name: 'Alonzo Church/Child1', age: 16 },
//     { name: 'Alonzo Church/Child2', age: 14 } ]
```

Παρατηρούμε δηλαδή ότι οι δύο εκφράσεις είναι ισοδύναμες, και πράγματι είναι για κάθε συνδυασμό $\boldsymbol{f}$ και $\boldsymbol{x}$ που θα επιλέξουμε. Η αριστερή ταυτοτική ιδιότητα λοιπόν ισχύει.

Η δεξιά ταυτοτική ιδιότητα (**right identity**), από την άλλη, ορίζει την παρακάτω ισοδυναμία

$$M \gg= return \equiv M$$

Έστω ότι ορίζουμε ως M το ίδιο το αντικείμενο τύπου **Student.** Η έκφραση $M \gg= return$ παίρνει τη μορφή που φαίνεται παρακάτω

```
const M = [{ name: "Alonzo Church", age: 36 }];

chain(M)(pure);
//-> [{ name: 'Alonzo Church', age: 36 }]
```

Παρατηρούμε ότι το αποτέλεσμα είναι ίδιο με τη μονάδα M, και πράγματι η παραπάνω ιδιότητα ισχύει για κάθε τύπο M. Η δεξιά ταυτοτική ιδιότητα λοιπόν ισχύει.

Ο προσεταιρισμός (**Associativity**) όσον αφορά τη συνάρτηση **chain** ορίζει την παρακάτω ισοδυναμία εκφράσεων

$$M \gg= f \gg= g \equiv\ M \gg= (\lambda x. f\ x \gg= g)$$

Έστω λοιπόν ότι ορίζουμε και μία άλλη συνάρτηση $\boldsymbol{g}$ με υπογραφή της μορφής

$$g :: Student \rightarrow [Student]$$

Ας υποθέσουμε ότι η υλοποίηση της g επιστρέφει τα παιδιά των παιδιών της αρχικής μονάδας M με τον παρακάτω τρόπο

```
// g :: Student -> [Student]
const g = ({name, age}) => [{name: `${name}/Gchild1`, age: age - 10}, {name:
`${name}/Gchild2`, age: age - 8}];
```

Αν μεταφέρουμε την έκφραση $M \gg= f \gg= g$ στο πεδίο της JavaScript θα έχουμε

```
chain(chain(M)(f))(g);
//-> [ { name: 'Alonzo Church/Child1/Gchild1', age: 6 },
//     { name: 'Alonzo Church/Child1/Gchild2', age: 8 },
//     { name: 'Alonzo Church/Child2/Gchild1', age: 4 },
//     { name: 'Alonzo Church/Child2/Gchild2', age: 6 } ]
```

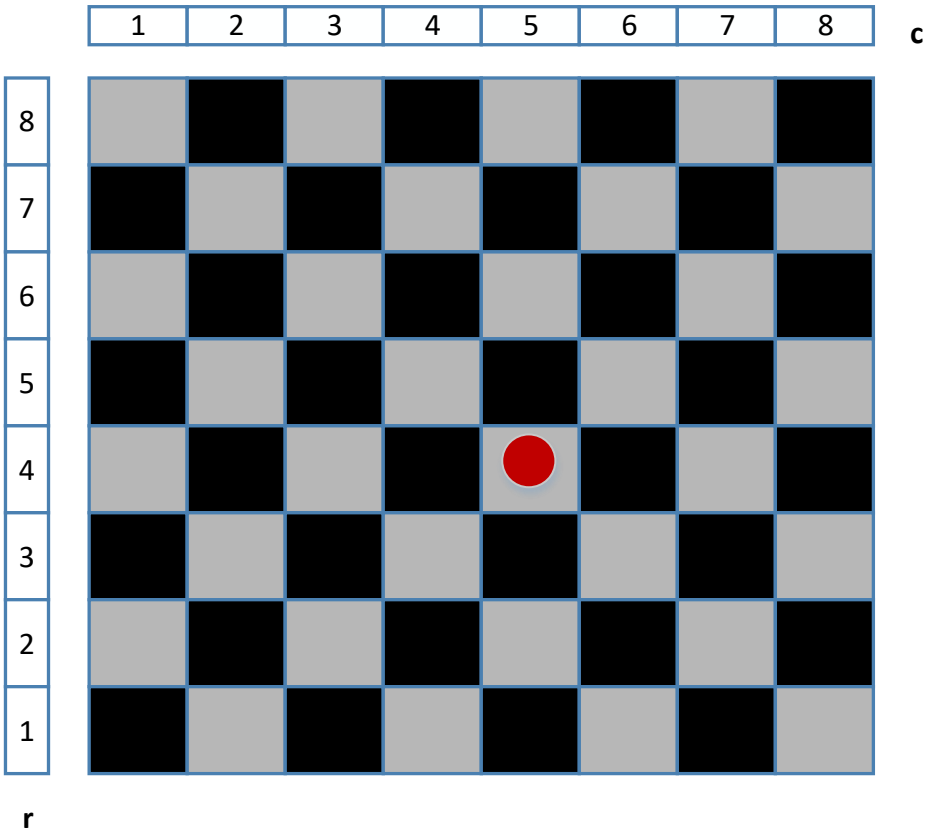
Παρατηρούμε ότι η λειτουργικότητα chain στη δομή των πινάκων ακολουθεί μία συνδυαστική λογική, αφού τα στοιχεία υποβάλλονται σε διαδοχικές απεικονίσεις (**fmap**) και ισοπεδώσεις (**flatten**). Το πλήθος των στοιχείων του πίνακα που θα δημιουργηθεί από πιθανές διαδοχικές τροφοδοτήσεις τη συνάρτησης **chain** είναι ίσο με το γινόμενο του πλήθος στων στοιχείων των πινάκων κάθε βαθμίδας.

Η έκφραση $M \gg= (\lambda x. f\ x \gg= g)$ όπως αποδεικνύεται επιστρέφει το ίδιο αποτέλεσμα. Έχουμε

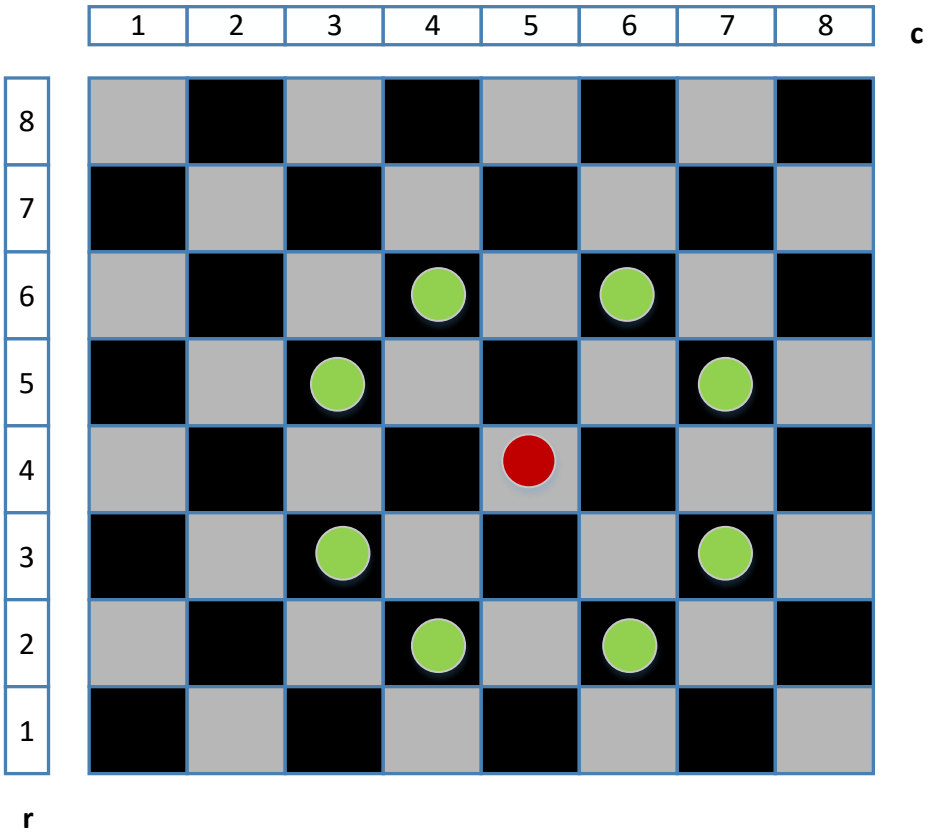
```
chain(M)(x => chain(f(x))(g));
//-> [ { name: 'Alonzo Church/Child1/Gchild1', age: 6 },
//     { name: 'Alonzo Church/Child1/Gchild2', age: 8 },
//     { name: 'Alonzo Church/Child2/Gchild1', age: 4 },
//     { name: 'Alonzo Church/Child2/Gchild2', age: 6 } ]
```

Παρατηρούμε ότι οι αποτιμήσεις των εκφράσεων είναι ολόιδιες και αυτό διότι η προσεταιριστική ιδιότητα, όσον αφορά την υλοποίηση **chain** που αξιοποιεί ο σχηματισμός μονάδας της δομής του πίνακα, τηρείται.

Πάμε να δούμε λοιπόν ένα απτό παράδειγμα στο οποίο η λειτουργικότητα των πινάκων ως μονάδες μας λύνει τα χέρια. Φανταστείτε ότι έχουμε μπροστά μας ένα σκακιστικό ταμπλό, και σε μία τυχαία θέση υπάρχει ένα πιόνι (έστω το άλογο) και θέλουμε να βρούμε τις πιθανές θέσεις που μπορεί να έχει μετά από **n** διαδοχικές κινήσεις. Ας οπτικοποίησουμε το πρόβλημα για να γίνει καλύτερα κατανοητό. Έστω το σκακιστικό ταμπλό



Αναθέτουμε τους αριθμούς από 1 έως 8 για τις στήλες (**columns**) και γραμμές (**rows**) αντίστοιχα. Αν υποθέσουμε ότι η αρχική θέση του αλόγου είναι στη θέση (col: 5, row: 4) τότε οι πιθανές θέσεις του αλόγου μετά από μία (n = 1) κίνηση θα είναι οι εξής



Πως θα προσεγγίζαμε το πρόβλημα λοιπόν; Αν θεωρήσουμε ότι η θέση του αλόγου προσομοιώνεται από τον τύπο ***Position*** τότε εύλογα αυτό που θα θέλαμε θα ήταν μία συνάρτηση που θα αντιστοιχούσε τη θέση αυτή, σε μία ομάδα θέσεων του ίδιου τύπου. Με λίγα λόγια θα χρειαστούμε μία συνάρτηση, έστω **moveHorse**, η υπογραφή της οποίας θα έχει την παρακάτω μορφή

$$moveHorse :: Position \rightarrow [Position]$$

Αυτομάτως αντιλαμβανόμαστε ότι η υπογραφή της συνάρτησης αυτής μπορεί κάλλιστα να χρησιμοποιηθεί ως τροφοδοσία στη συνάρτηση **chain** για την παραγωγή ίσως κάποιου χρήσιμου αποτελέσματος. Αν ορίσουμε τον τύπο ***Position*** ως ένα αντικείμενο που κρατά στο εσωτερικό του τον αριθμό της στήλης και της γραμμής, μπορούμε να κατασκευάσουμε μία βοηθητική συνάρτηση παραγωγής τέτοιων αντικειμένων. Έχουμε λοιπόν

```
// createPos :: (Number, Number) -> Position
const createPos = (c, r) => ({
    col: c,
    row: r
});
```

Έτσι για τη δημιουργία της αρχικής θέσης (col: 5, row: 4) αρκεί να γράψουμε την παρακάτω έκφραση

```
// init :: Position
const init = createPos(5, 4);
//-> {col: 5, row: 4}
```

Πάμε να δούμε την υλοποίηση της συνάρτησης **moveHorse.** Η λειτουργικότητα αυτή, λογικά θα πρέπει να δεχτεί ένα αντικείμενο τύπου ***Position*** και να επιστρέφει όλες τις πιθανές θέσεις του αλόγου που προκύπτουν μετά από μία ενέργεια. Επειδή οι πιθανές θέσεις είναι πολλές στον αριθμό θα πρέπει φυσικά να τις ομαδοποιήσουμε χρησιμοποιώντας τη δομή του πίνακα για την αναπαράσταση τους. Βλέποντας το σκακιστικό ταμπλό αντιλαμβανόμαστε το μοτίβο που ακολουθεί η κίνηση του αλόγου πάνω σε αυτό. Για την αρχική θέση (col: 5, row: 4), παρατηρούμε ότι οι πιθανές θέσεις που μπορεί να βρεθεί το άλογο έχουν την παρακάτω συσχέτιση με την αρχική θέση του πιονιού.

Αρχική θέση	**Πιθανή θέση (n=1)**
(col: 5, row: 4)	(col’: 7(col + 2), row’: 3(row + 1))
(col: 5, row: 4)	(col’: 7(col + 2), row’: 5(row + 1))
(col: 5, row: 4)	(col’: 3(col - 2), row’: 3(row - 1))
(col: 5, row: 4)	(col’: 3(col - 2), row’: 5(row + 1))
(col: 5, row: 4)	(col’: 6(col + 1), row’: 2(row - 2))
(col: 5, row: 4)	(col’: 6(col + 1), row’: 6(row + 2))
(col: 5, row: 4)	(col’: 4(col - 1), row’: 2(row - 2))
(col: 5, row: 4)	(col’: 4(col - 1), row’: 6(row + 2))

Συνεπώς η **moveHorse** μπορεί να γραφεί όπως φαίνεται παρακάτω

```
// moveHorse :: Position -> [Position]
const moveHorse = ({col, row}) => ([
    createPos(col + 2, row - 1),
    createPos(col + 2, row + 1),
    createPos(col - 2, row - 1),
    createPos(col - 2, row + 1),
    createPos(col + 1, row - 2),
    createPos(col + 1, row + 2),
    createPos(col - 1, row - 2),
    createPos(col - 1, row + 2)
]);
```

Δέχεται δηλαδή ως παράμετρο ένα αντικείμενο τύπου ***Position*** και επιστρέφει όλες τις πιθανές θέσεις που προκύπτουν μετά από μία ενέργεια ομαδοποιημένες σε έναν πίνακα, δηλαδή [***Position***].

Κάποιος θα έλεγε ότι θα μπορούσαμε να βρούμε τις πιθανές θέσεις με την παρακάτω έκφραση.

```
moveHorse(init);
```

Πράγματι για την περίπτωση της πρώτης ενέργειας (δηλαδή για n = 1), θα μπορούσε να πραγματοποιηθεί η παραπάνω κλήση, αλλά πως θα προσεγγίζαμε το πρόβλημα αργότερα; Θα είχαμε στην κατοχή μας έναν πίνακα. Πώς θα τον τροφοδοτούσαμε στη συνάρτηση **moveHorse** πάλι (πχ για n = 2), χωρίς να κάναμε αλλαγές στον κώδικα. Αυτό ακριβώς το πρόβλημα αντιμετωπίζει η δομή μίας μονάδας καθώς αποκρύπτει την περίπλοκη λογική που θα πρόκυπτε κατά την προσπάθεια μας να φέρουμε σε αρμονία τους συγκεκριμένους τύπους. Για αυτό τον λόγο χρησιμοποιούμε και τη συνάρτηση **chain.** Η υπογραφή της συνάρτησης όπως έχουμε δει για τη δομή του πίνακα, περιμένει μία μονάδα, μία συνάρτηση τύπου **moveHorse**, και επιστρέφει ένα αποτέλεσμα. Λαμβάνοντας υπόψιν αυτά μπορούμε να γράψουμε

```
// init :: Position
const init = createPos(5, 4);

// moveHorse :: Position -> [Position]
const moveHorse = ({col, row}) => ([
    createPos(col + 2, row - 1),
    createPos(col + 2, row + 1),
    createPos(col - 2, row - 1),
    createPos(col - 2, row + 1),
    createPos(col + 1, row - 2),
    createPos(col + 1, row + 2),
    createPos(col - 1, row - 2),
    createPos(col - 1, row + 2)
]);

chain(pure(init))(moveHorse);
//-> [ { col: 7, row: 3 },
//     { col: 7, row: 5 },
//     { col: 3, row: 3 },
//     { col: 3, row: 5 },
//     { col: 6, row: 2 },
//     { col: 6, row: 6 },
//     { col: 4, row: 2 },
//     { col: 4, row: 6 } ]
```

Παρατηρούμε ότι χρησιμοποιούμε τη συνάρτηση **return** (pure) για να προσθέσουμε το υπολογιστικό πλαίσιο που ορίζει η μονάδα γύρω από την τιμής μας (αρχική θέση), έτσι ώστε να μπορέσουμε να χρησιμοποιήσουμε τη δομή στην κλήση της **chain.** Όλα καλά μέχρι εδώ αλλά έχουμε ένα πρόβλημα λογικής. Η αρχική θέση (col: 5, row: 4) για το άλογο, έχει 8 στο σύνολο πιθανές ορθές θέσεις. Τι θα γινότανε όμως αν είχαμε ως αρχική θέση του αλόγου τη θέση (col: 7, row: 3) για παράδειγμα. Το άλογο θα έβγαινε έξω από το ταμπλό καθώς θα παρουσιάζει θέσεις με δείκτες col,row μεγαλύτερους του 8 ή μικρότερους του 1. Πρέπει συνεπώς να διορθώσουμε τη συνάρτηση **moveHorse** έτσι ώστε να φιλτράρει τις θέσεις ανάλογα με τους δείκτες που προκύπτουν. Μπορούμε να χρησιμοποιήσουμε τη συνάρτηση **filter.** Έχουμε

```
// colAndRowInBound :: Position -> Boolean
const colAndRowInBound = ({col, row}) => col > 0 && col < 9 && row > 0 && row <
9;

// moveHorse :: Position -> [Position]
const moveHorse = ({ col, row }) => ([
    createPos(col + 2, row - 1),
    createPos(col + 2, row + 1),
    createPos(col - 2, row - 1),
    createPos(col - 2, row + 1),
    createPos(col + 1, row - 2),
    createPos(col + 1, row + 2),
    createPos(col - 1, row - 2),
    createPos(col - 1, row + 2)
]).filter(colAndRowInBound);
```

Έτσι για την αρχική θέση (col: 7, row: 3) θα έχουμε πλέον ορθά

```
// init :: Position
const init = createPos(7, 3);

// colAndRowInBound :: Position -> Boolean
const colAndRowInBound = ({col, row}) => col > 0 && col < 9 && row > 0 && row <
9;

// moveHorse :: Position -> [Position]
const moveHorse = ({ col, row }) => ([
    createPos(col + 2, row - 1),
    createPos(col + 2, row + 1),
    createPos(col - 2, row - 1),
    createPos(col - 2, row + 1),
    createPos(col + 1, row - 2),
    createPos(col + 1, row + 2),
    createPos(col - 1, row - 2),
    createPos(col - 1, row + 2)
]).filter(colAndRowInBound);

chain(pure(init))(moveHorse);
//-> [ { col: 5, row: 2 },
//     { col: 5, row: 4 },
//     { col: 8, row: 1 },
//     { col: 8, row: 5 },
//     { col: 6, row: 1 },
//     { col: 6, row: 5 } ]
```

Δώσαμε λύση στο πρόβλημα, για την περίπτωση που το n μας είναι 1. Πως μπορούμε να επεκτείνουμε τη λύση μας, για n = 2, n = 3 και τελικώς για n. Εδώ φαίνεται η μαγεία του υπολογιστικού περιβάλλοντος που προσδίδει ο σχηματισμός της μονάδας

μας. Αποδεικνύεται ότι μπορούμε να επανατροφοτήσουμε το αποτέλεσμα της πρώτης κλήσης της **chain** σε μία νέα κλήση της, με παράμετρο την ίδια συνάρτηση **moveHorse**, λαμβάνοντας έτσι ως έξοδο στο σύστημα μας τις θέσεις του αλόγου που προκύπτουν μετά από δύο ενέργειες (n = 2). Για την αρχική θέση (col: 5, row: 4) έχουμε

```
// init :: Position
const init = createPos(5, 4);

const a = chain(pure(init))(moveHorse);
//-> [ { col: 7, row: 3 },
//     { col: 7, row: 5 },
//     { col: 3, row: 3 },
//     { col: 3, row: 5 },
//     { col: 6, row: 2 },
//     { col: 6, row: 6 },
//     { col: 4, row: 2 },
//     { col: 4, row: 6 } ]

const b = chain(a)(moveHorse);
```

Το αποτέλεσμα που αποθηκεύεται στη μεταβλητή **a** τροφοδοτείται σε αμέσως επόμενη κλήση της chain για την απεικόνιση των πιθανών θέσεων που προκύπτουν, αν υποθέσουμε ως αρχικές θέσεις όλες τις τιμές του πίνακα της μεταβλητή **a.** Το αποτέλεσμα **b** αντιπροσωπεύει δηλαδή όλες τις πιθανές αυτές θέσεις στη μορφή ενός πίνακα. Φυσικά το μέγεθος αυξάνεται πολλαπλασιαστικά για κάθε επιπλέον στάδιο, όποτε η απεικόνιση των δεδομένων είναι δυσχερής. Η αποτίμηση της έκφραση **b**, που αντιπροσωπεύει τη λύση του προβλήματος για n = 2 φαίνεται παρακάτω

```
const b = chain(a)(moveHorse);
//->
/**
[ { col: 5, row: 2 },
  { col: 5, row: 4 },
  { col: 8, row: 1 },
  { col: 8, row: 5 },
  { col: 6, row: 1 },
  { col: 6, row: 5 },
  { col: 5, row: 4 },
  { col: 5, row: 6 },
  { col: 8, row: 3 },
  { col: 8, row: 7 },
  { col: 6, row: 3 },
  { col: 6, row: 7 },
  { col: 5, row: 2 },
  { col: 5, row: 4 },
  { col: 1, row: 2 },
  { col: 1, row: 4 },
  { col: 4, row: 1 },
  { col: 4, row: 5 },
  { col: 2, row: 1 },
  { col: 2, row: 5 },
  { col: 5, row: 4 },
  { col: 5, row: 6 },
  { col: 1, row: 4 },
  { col: 1, row: 6 },
  { col: 4, row: 3 },
  { col: 4, row: 7 },
  { col: 2, row: 3 },
  { col: 2, row: 7 },
  { col: 8, row: 1 },
  { col: 8, row: 3 },
  { col: 4, row: 1 },
  { col: 4, row: 3 },
  { col: 7, row: 4 },
  { col: 5, row: 4 },
  { col: 8, row: 5 },
  { col: 8, row: 7 },
  { col: 4, row: 5 },
  { col: 4, row: 7 },
  { col: 7, row: 4 },
  { col: 7, row: 8 },
  { col: 5, row: 4 },
  { col: 5, row: 8 },
  { col: 6, row: 1 },
  { col: 6, row: 3 },
  { col: 2, row: 1 },
  { col: 2, row: 3 },
  { col: 5, row: 4 },
  { col: 3, row: 4 },
  { col: 6, row: 5 },
  { col: 6, row: 7 },
  { col: 2, row: 5 },
  { col: 2, row: 7 },
  { col: 5, row: 4 },
  { col: 5, row: 8 },
  { col: 3, row: 4 },
  { col: 3, row: 8 } ]
*/
```

Αφού παρατηρούμε ότι η έξοδος της μίας έκφρασης, αποτελεί είσοδο στην επόμενη κλήση της **chain**, μπορούμε να προχωρήσουμε σε αναδρομική υλοποίηση προκειμένου να

παραμετροποιήσουμε τον αριθμό των κλήσεων (που μεταφράζονται σε ενέργειες, δηλαδή τον αριθμό n). Έστω η συνάρτηση **moveHorseNTimes**

```
// moveHorseNTimes :: (Number, [Position]) -> [Position]
const moveHorseNTimes = (n, m) => n < 1 ? m : chain(moveHorseNTimes(n - 1,
m))(moveHorse);

// init :: Position
const init = createPos(5, 4);

// return/pure :: a -> [a]
const pure = x => [x];

moveHorseNTimes(2, pure(init));
//-> b
```

Παρατηρούμε ότι εμφωλεύουμε την αναδρομική κλήση εντός της **chain** για να παράξουμε την κατάλληλη διάταξη. Μπορούμε να γράψουμε την ίδια αναπαράσταση κάνοντας χρήση της συνάρτηση **foldl.** Θα έχουμε

```
// foldl :: (a -> b -> a) -> a -> [b] -> a
const foldl = f => z => ([first, ...rest]) => first === undefined ? z :
foldl(f)(f(z)(first))(rest);

// moveHorseNTimes :: (Number, [Position]) -> [Position]
const moveHorseNTimes = (n, m) => foldl(chain)(m)(Array(n).fill(moveHorse));

moveHorseNTimes(2, pure(init));
//-> b
```

Η συνάρτηση **moveHorseNTimes** τελικά λύνει το ζητούμενο αρχικό πρόβλημα.

Φανταστείτε τώρα ότι επιπρόσθετα με την τελική πιθανή θέση του αλόγου, θέλουμε να κρατάμε ως πληροφορία όλες τις προηγούμενες θέσεις του. Δηλαδή ουσιαστικά θέλουμε να απεικονίσουμε τη διαδρομή που ακολούθησε το άλογο ώστε ένα καταλήξει στην εκάστοτε τελική θέση. Η συναρτησιακή προσέγγιση που υιοθετούμε διευκολύνει σε μεγάλο βαθμό την υλοποίηση της λύσης. Το μόνο που πρέπει να κάνουμε είναι η τροποποίηση της δομής ***Position*** έτσι ώστε εσωτερικά της, να αποθηκεύεται η απαιτούμενη πληροφορία (ένα αντικείμενο δηλαδή τύπου ***Position***). Θέλουμε δηλαδή να κατασκευάσουμε μία αναδρομική δομή. Προσθέτουμε μία τρίτη παράμετρο στην **createPos** που θα αντιπροσωπεύει το προηγούμενο ***Position.*** Έχουμε

```
// createPos :: (Number, Number, Position) -> Position
const createPos = (c, r, prevPos) => ({
    col: c,
    row: r,
    prevPos,
});
```

Τώρα κατά τη δήλωση της αρχικής θέσης, η τιμή της παραμέτρου **prevPos** θα αντιπροσωπεύεται από την έκφραση **undefined**, για να δηλώσει την απώλεια προηγούμενης θέσης.

```
// init :: Position
const init = createPos(5, 4, undefined);
```

Τέλος το μόνο που απομένει, είναι η τροποποίηση της συνάρτησης **moveHorse** έτσι ώστε να ενθυλακώνει εντός των καινούργιων θέσεων και την παλιά. Η υλοποίηση φαίνεται παρακάτω

```
// colAndRowInBound :: Position -> Boolean
const colAndRowInBound = ({col, row}) => col > 0 && col < 9 && row > 0 && row <
9;

// moveHorse :: Position -> [Position]
const moveHorse = ({ col, row, prevPos }) => ([
    createPos(col + 2, row - 1, createPos(col, row, prevPos)),
    createPos(col + 2, row + 1, createPos(col, row, prevPos)),
    createPos(col - 2, row - 1, createPos(col, row, prevPos)),
    createPos(col - 2, row + 1, createPos(col, row, prevPos)),
    createPos(col + 1, row - 2, createPos(col, row, prevPos)),
    createPos(col + 1, row + 2, createPos(col, row, prevPos)),
    createPos(col - 1, row - 2, createPos(col, row, prevPos)),
    createPos(col - 1, row + 2, createPos(col, row, prevPos))
]).filter(colAndRowInBound);
```

Η συνάρτηση **moveHorseNTimes** διατηρεί την υλοποίηση της

```
// moveHorseNTimes :: (Number, [Position]) -> [Position]
const moveHorseNTimes = (n, m) => foldl(chain)(m)(Array(n).fill(moveHorse));
```

Αν δοκιμάσουμε να εκτελέσουμε τη συνάρτηση για n=1 και αρχική θέση την (col: 5, row: 4) έχουμε το εξής πλέον αποτέλεσμα

```
moveHorseNTimes(1, pure(init));
//->
/**
 *
[
    {
        "col": 7,
        "row": 3,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
    {
        "col": 7,
        "row": 5,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
    {
        "col": 3,
        "row": 3,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
    {
        "col": 3,
        "row": 5,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
    {
        "col": 6,
        "row": 2,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
```

```
{
        "col": 6,
        "row": 6,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
    {
        "col": 4,
        "row": 2,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    },
    {
        "col": 4,
        "row": 6,
        "prevPos": {
            "col": 5,
            "row": 4
        }
    }
]
 */
```

Το βάθος του αντικειμένου **prevPos** καθορίζεται από το πλήθος n ενεργειών κίνησης του αλόγου. Για n = 1, έχουμε μία προηγούμενη θεση, για n = 2 παραθέτουμε ένα δείγμα.

```
moveHorseNTimes(2, pure(init));
//->
/**
 *
 * [
  {
    "col": 5,
    "row": 2,
    "prevPos": {
      "col": 7,
      "row": 3,
      "prevPos": {
        "col": 5,
        "row": 4
      }
    }
  },
  ...
  ]
 */
```

Παρατηρούμε ότι έχουμε δύο αντικείμενα της μορφής **prevPos** εμφωλευμένα, που δηλώνουν τη διαδρομή που ακολουθήσε το πιόνι για να φτάσει στη θέση (col: 5, row 2). Η διαδρομή καθορίζεται από μέσα προς τα έξω. Στο παρόν παράδειγμα δηλαδή έχουμε

$$(col: 5, row: 4) \rightarrow (col: 7, row: 3) \rightarrow (col: 5, row: 2)$$

Αντιλαμβανόμαστε λοιπόν ότι ο σχηματισμός μίας μονάδας προσφέρει στον προγραμματιστή ένα ισχυρό εργαλείο αντιμετώπισης προβλημάτων αφαιρώντας τη σύνθετη λογική πίσω από μία συνάρτηση **chain** στα πλαίσια που ορίζει ο συναρτησιακός προγραμματισμός. Φυσικά ο πίνακας δεν αποτελεί τον μοναδικό σχηματισμό μονάδας. Υπάρχουν και άλλες γνωστές υλοποιήσεις που μας διευκολύνουν προσθέτοντας το κατάλληλο υπολογιστικό πλαίσιο γύρω από τα δεδομένα μας. Για παράδειγμα, ο συναρτήτης **Maybe a,** που όπως είπαμε υποδηλώνει την ύπαρξη ή ανυπαρξία μίας τιμής, μπορεί να λειτουργήσει και ως μονάδα, δίνοντας μία υλοποίηση στη συνάρτηση **chain** που δεν καταστρατηγεί του νόμους των **Monads.** Η δομή της διασυνδεμένης λίστας **Pair**, που όπως είδαμε συμπεριφέρεται και ως συναρτήτης αφού τελικώς αποτελεί μία λίστα , απλώς με δική μας υλοποίηση, μπορεί να μεταχειριστεί και ως σχηματισμός μονάδας. Οι μονάδες χρησιμοποιούνται και για τη συναρτησιακή προσέγγιση λύσεων προβλημάτων που εγγενώς παρουσιάζουν παρενέργειες (side effects), όπως οι λειτουργικότητες εισόδου-εξόδου (**IO Monad**). Επιπρόσθετα προσφέρουν λύση και στη διαχείριση της κατάστασης μία δομής ή του προγράμματος μας γενικά υλοποιώντας τη μονάδα κατάστασης (**State Monad**). Η λογική της μονάδας αυτής έχει να κάνει με τον εγκιβωτισμό ενός υπολογισμού της μορφής

$$State \rightarrow (a, State)$$

όπου κάθε μεταβολή της κατάστασης μίας δομής παράγει μία πιθανή τιμή της μορφής **a,** και μία νέα κατάσταση. Για παράδειγμα αν προσομοιώσουμε τη δομή μίας στοίβας ως έναν πίνακα, τότε οι γνωστές λειτουργικότητες **pop** και **push** υιοθετούν την υπογραφή

$$State \rightarrow (a, State)$$

Πετυχαίνοντας έτσι πλήρη αγνότητα στον κώδικα (**pureness**) καθώς τα αποτελέσματα των κλήσεων τους θα είναι σταθερά και δεν θα μεταβάλλουν τη δομή της στοίβας αλλά θα δημιουργούν πάντα μία νέα. Στην JavaScript θα έχουμε

```
// [Number]/State Stack Number
const a = [1, 2, 3, 4, 5];

// pop :: State -> (Number, State)
const pop = state => ({
    val: state[0],
    state: state.slice(1),
});

// push :: Number -> State -> (Number, State)
const push = n => state => ({
    val: undefined,
    state: [n].concat(state)
});

pop(a);
//-> { val: 1, state: [ 2, 3, 4, 5 ] }
push(10)(a);
//-> { val: undefined, state: [ 10, 1, 2, 3, 4, 5 ]
```

Παρατηρούμε ότι οι κλήσεις **pop(a)** και **push(10)(a)** επιστρέφουν ένα αντικείμενο στο οποίο εμπεριέχεται και η προηγούμενη κατάσταση της στοίβας. Οι κλήσεις αυτές είναι αγνές καθότι δεν μεταβάλλουν το περιεχόμενο του πίνακα **a.** Αποδεικνύεται ότι η συμπεριφορά αυτή μπορεί να τοποθετηθεί αποδοτικά εντός ενός υπολογιστικού πλαισίου σχηματίζοντας τη μονάδα της κατάστασης (**State Monad**). Η μονάδα κατάστασης ουσιαστικά εγκιβωτίζει έναν υπολογισμό της μορφής

$$State \rightarrow (a, State)$$

προσφέροντας έτσι δυνατότητες συσσώρευσης ενεργειών που καταλήγουν στην παραπάνω υπογραφή. Στη γλώσσα της JavaScript η υλοποίηση θα είχε την παρακάτω μορφή

```
// State :: (Stack -> (Number, Stack) => s) s -> State s a
const State = s => ({
    getStateComputation: () => s
});

// pop :: (Stack -> (Number, Stack) => s) State s a
const pop = State(state => ({
    val: state[0],
    state: state.slice(1),
}));

// push :: (Stack -> (Number, Stack) => s) Number -> State s a
const push = n => State(state => ({
    val: undefined,
    state: [n].concat(state)
}));


// pure :: a -> State s a
const pure = x => State(s => ({
    val: x,
    state: s
}));

// chain :: State s a -> (a -> State s b) -> State s b
const chain = m => f => State(s => {
    const {val, state} = m.getStateComputation()(s);
    const newState = f(val);
    return newState.getStateComputation()(state);
});
```

Ο κώδικας της μονάδας κατάστασης μπορεί να φαίνεται περίπλοκος, κυρίως λόγο του γεγονότος ότι εγκιβωτίζει μία συνάρτηση, άλλα πετυχαίνει την αποδοτική αφαίρεση του υπολογισμού καθιστώντας τον διαθέσιμο για ένα πλήθος καταστατικά προβλήματα. Αν καλέσουμε την **chain** με παράμετρους τη συνάρτηση **pop** και **push** για παράδειγμα χτίζουμε έναν νέο υπολογισμό της μορφής $State \rightarrow (a, State)$ που όταν δεχτεί μία στοίβα, θα πράττει πάνω της μία ενέργεια της μορφής **pop** και άλλη μία της μορφής **push,** εισάγοντας ουσιαστικά το στοιχείο που μόλις είχε εξαχθεί. Έχουμε

```
// [Number]/State Stack Number
const a = [1, 2, 3, 4, 5];

chain(pop)(push).getStateComputation()(a);
//-> { val: undefined, state: [ 1, 2, 3, 4, 5 ] }
```

Στο επόμενο παράδειγμα καλούμε τις ενέργειες **pop**, **λy. push 10**, **λx. pop,** με αποτέλεσμα να εξάγεται το πρώτο στοιχείο, να εισάγεται ο αριθμός 10, και τέλος να εξάγεται ο ίδιος αριθμός. Έχουμε

```
// [Number]/State Stack Number
const a = [1, 2, 3, 4, 5];

chain(chain(pop)(y => push(10)))(x => pop).getStateComputation()(a);
//-> { val: 10, state: [ 2, 3, 4, 5 ] }
```

Το αποτέλεσμα των διαδοχικών κλήσεων **chain** παράγει ένα **State Monad** το οποίο μπορεί να χρησιμοποιηθεί από οποιοδήποτε κώδικα πελάτη και να το διαχειριστεί κατάλληλα στα πλαίσια που ορίζει ο συναρτησιακός προγραμματισμός.

Όπως προαναφέραμε η δομή **Pair** μπορεί να αξιοποιηθεί και αυτή ως μονάδα (**Pair Monad**). Η συμπεριφορά της συνάρτησης **chain** θα είναι παρόμοια αυτή των πινάκων, θα προσφέρουν δηλαδή ένα υπολογιστικό πλαίσιο που θα προχωρά στη συνδυαστική απεικόνιση των επιμέρους δομών και ταυτόχρονη ισοπέδωση τους. Πάμε να δούμε πως υλοποιούν έναν σχηματισμό μονάδας. Ο τύπος της συνάρτησης **chain** για αυτήν τη μονάδα, αλλά και για κάθε άλλη είπαμε ότι μπορεί να αναπαρασταθεί με την πράξης της απεικόνισης και της ισοπέδωσης για αυτό και **flatmap**. Έχουμε

```
// chain :: Pair a -> (a -> Pair b) -> Pair b
const chain = p => f => flatten(map(f)(p));
```

Αρκεί λοιπόν να δώσουμε υλοποίηση στις συναρτήσεις **map** και **flatten.** Η συνάρτηση **map** έχει υλοποιηθεί στα προηγούμενα κεφάλαια και είναι αυτή που χρησιμοποιήθηκε στον συναρτήτη Pair, δηλαδή η **fmap.** Η συνάρτηση **flatten**, θα υλοποιηθεί με τη βοήθεια της αντίστοιχης **mconcat** που διέπει τη δομή Pair, αφού όπως είδαμε και αυτή σχηματίζει και μονοειδή σχηματισμό (Monoid). Για την **map** έχουμε

```
const pair = x => y => s => s(x)(y);

const K = x => y => x;
const I = x => x;

const car = p => p(K);
const cdr = p => p(K(I));

// prepend :: a -> Pair a -> Pair a
const prepend = x => y => pair(x)(y);

// fmap/map :: (a -> b) -> Pair a -> Pair b
const map = f => p => p === null ? null : prepend(f(car(p)))(map(f)(cdr(p)));
```

Για την υλοποίηση της **flatten** όπως είπαμε θα χρειαστούμε τη συνάρτηση mappend και foldr, που αξιοποιούνται για τη δημιουργία της συνάρτησης **mconcat.** Η **mconcat** θα δέχεται μία δομή **Pair** από **Pairs.** Σε αντίθεση με την περίπτωση που είδαμε στους μονοειδής σχηματισμούς όπου η αρχική δομή ομαδοποίησης των στοιχείων που θα εισαχθούν στην **mappend** είναι ένας πίνακας. Με λίγα λόγια η υπογραφή της **mconcat** θα έχει την παρακάτω μορφή

$$mconcat :: Pair(Pair\ a) \rightarrow Pair\ a$$

και όχι αυτή

$$mconcat :: [Pair\ a] \rightarrow Pair\ a$$

Η υλοποίηση επιτυγχάνεται με τη βοήθεια της **foldr** που διπλώνει δομές τύπου **Pair**. Έχουμε

```
// mempty :: null
const mempty = null;

// mappend :: Pair a -> Pair a -> Pair a
const mappend = p1 => p2 => p1 === null ? p2 :
prepend(car(p1))(mappend(cdr(p1))(p2));

// foldr :: (a -> b -> b) -> b -> Pair a -> b
const foldr = f => z => p => p === null ? z : f(car(p))(foldr(f)(z)(cdr(p)));

// mconcat/flatten = Pair (Pair a) -> Pair a
const mconcat = foldr(mappend)(mempty);
```

Η συνάρτηση **mconcat** μπορεί να χρησιμοποιηθεί για την υλοποίηση της λειτουργικότητας της **flatten,** ακριβώς όπως έγινε και με τη δομή του πίνακα. Για την ακρίβεια

$$mconcat \equiv flatten$$

Συνεπώς τελικώς για την **chain** μπορούμε να γράψουμε

```
// mconcat/flatten = Pair (Pair a) -> Pair a
const mconcat = flatten = foldr(mappend)(mempty);

// fmap/map :: (a -> b) -> Pair a -> Pair b
const map = f => p => p === null ? null : prepend(f(car(p)))(map(f)(cdr(p)));

// chain :: Pair a -> (a -> Pair b) -> Pair b
const chain = p => f => flatten(map(f)(p));
```

Η δεύτερη συνάρτηση που απαρτίζει τη μονάδα, η **return** δηλαδή, εγκιβωτίζει μία τυχαία τιμή εντός του υπολογιστικού πλαισίου που ορίζει η μονάδα **Pair**. Θα έχουμε

```
// pure/return :: a -> Pair a
const pure = x => prepend(x)(null);
```

Τέλος, οι τρεις νόμοι ισχύουν κανονικότατα για τη δομή της Pair.

Η αριστερή ταυτοτική ιδιότητα (**left identity**) αναφέρει ότι τηρείται

$$return\ x \gg= f \equiv f\ x$$

Στην JavaScript έχουμε

```
// f :: a -> Pair a
const f = x => prepend(1)(prepend(2)(null));

chain(pure(1))(f);
//-> pair(1)(pair(2)(null))

f(1);
//-> pair(1)(pair(2)(null))
```

Η δεξιά ταυτοτική ιδιότητα (**right identity**) αναφέρει ότι τηρείται

$$M \gg= return \equiv M$$

Στην JavaScript έχουμε

```
const M = prepend(1)(null);
//-> pair(1)(null)

chain(M)(pure);
//-> pair(1)(null)
```

Τέλος όσον αφορά στον προσεταιρισμό (**Associativity**) ισχύει η παρακάτω ισοδυναμία

$$M \gg= f \gg= g \equiv\ M \gg= (\lambda x. f\ x \gg= g)$$

Ένα παράδειγμα που αντιπροσωπεύει την παραπάνω ισοδυναμία απεικονίζεται από κάτω

```
// f :: a -> Pair a
const f = x => prepend(1)(prepend(2)(null));
// g :: a -> Pair a
const g = x => prepend(x)(prepend(-x)(null));

const M = prepend(1)(null);
//-> pair(1)(null)

chain(chain(M)(f))(g);
//-> pair(1)(pair(-1)(pair(2)(pair(-2)(null))))

chain(M)(x => chain(f(x))(g));
//-> pair(1)(pair(-1)(pair(2)(pair(-2)(null))))
```

Φυσικά οι παραπάνω ισοδυναμίες ισχύουν για όλους τους συνδυασμούς συναρτήσεων που τηρούν τους προκαθορισμένους τύπους. Παρατηρούμε ότι η διασυνδεμένη λίστα συμπεριφέρεται ως ένας σχηματισμός μονάδας και παρουσιάζει ακριβώς τις ίδιες λειτουργικότητες με τη δομή του πίνακα, καθώς σε αφηρημένο επίπεδο και οι δύο δομές έχουν σκοπό την ομαδοποίηση δεδομένων αποκρύπτοντας την έλλειψη ντετερμινισμού όσον αφορά τις πιθανές τιμές στις οποίες αποτιμάται μία έκφραση.

Προτού δούμε άλλο ένα παράδειγμα εφαρμογής μίας μονάδας για την επίλυση ενός προβλήματος πάμε να αναλύσουμε λίγο προσεκτικά τον τρίτο νόμο των μονάδων, δηλαδή την προσεταιριστική (**associativity**) ιδιότητα στην οποία υπακούνε. Ο νόμος αναφέρει ότι η εφαρμογή της συνάρτησης **chain** σε μία μονάδα και μία συνάρτηση **f** της μορφής **a** → ***M b,*** και στη συνέχεια η εφαρμογή του αποτελέσματος στην ίδια την **chain** και σε μία συνάρτηση **g** της μορφής ***b*** → ***M c*** **,** ισοδυναμεί με την εφαρμογή της συνάρτησης **chain** στην αρχική μονάδα και σε μία νέα συνάρτηση, που όπως θα δούμε χτίζεται από τις **f** και **g** και έχει μορφή ***a*** → ***M c.*** Αν θυμηθούμε τον τύπο βλέπουμε ότι

$$M \gg\!= f \gg\!= g \equiv M \gg\!= (\lambda x. f\ x \gg\!= g)$$

η έκφραση $\lambda x. f\ x \gg\!= g$ αποτυπώνει μία συνάρτηση που θα δεχτεί ένα δεδομένο τύπου **a,** θα παράγει έναν σχηματισμό μονάδας τύπου ***M b*** μέσω της $f\ x$, και στη συνέχεια η τιμή ***M b*** θα τροφοδοτηθεί μέσω της **chain** σε μία συνάρτηση **g** της μορφής ***b*** → ***M c*** για τη δημιουργία ενός αποτελέσματος ***M c*** **.** Με λίγα λόγια η συνάρτηση αυτή καταφέρνει να συνδυάσει δύο συναρτήσεις της μορφής **a** → ***M b,*** ***b*** → ***M c*** σε μία της μορφής ***a*** → ***M c.*** Σας θυμίζει τίποτα αυτό; Αν θυμηθούμε την πράξη της σύνθεσης συναρτήσεων **compose** (∘) , θα παρατηρήσουμε ότι η συνάρτηση μπορούσε να μετατρέψει δύο συναρτήσεις της μορφής **a** → ***b***, ***b*** → ***c*** σε μία της μορφής ***α*** → ***c.*** Αν είμαστε παρατηρητικοί,

συμπεραίνουμε ότι πρόκειται για την ίδια πράξη με τη διαφορά ότι στην πρώτη περίπτωση προσθέτουμε το υπολογιστικό πλαίσιο της μονάδας ***M*.**

Δημιουργήσαμε δηλαδή έναν τρόπο σύνθεσης συναρτήσεων που συνυπολογίζουν το υπάρχον υπολογιστικό πλαίσιο που ορίζει η εκάστοτε μονάδα, (για αυτό και χρησιμοποιείτε η συνάρτηση **chain** για την υλοποίηση της).

Η πράξης της σύνθεσης συναρτήσεων που παράγουν σχηματισμούς μονάδων, ονομάζεται σύνθεση Kleisli (**Kleisli Composition**), προς τιμήν του μαθηματικού Heinrich Kleisli. Ο ταυτοτικός μορφισμός της σύνθεσης, όπως θα δούμε είναι η συνάρτηση **return,** καθώς αν συμβολίσουμε την πράξη της σύνθεσης Kleisli με τον τελεστή (<=<, σύμβολο που χρησιμοποιεί η Haskell), τότε έχουμε για οποιαδήποτε συνάρτηση **f** της μορφής **a** → ***M b*,**

$$f <=< return \equiv return <=< f \equiv f$$

Η υλοποίηση της σύνθεση Kleisli ορίζει ότι, εάν έχουμε μία συνάρτηση **f** της μορφής ***b*** → ***M c*** και μία συνάρτηση **g** της μορφής **a** → ***M b*,** τότε το αποτέλεσμα της σύνθεσης διατηρώντας το υπολογιστικό πλαίσιο που ορίζει η μονάδα ***M*** ισούται με

$$f <=< g = \lambda x.(g(x) >>= f)$$

Αν μεταφερθούμε στο περιβάλλον της JavaScript μπορούμε να ορίσουμε τη συνάρτηση **composeKTwo** που συντελεί την παραπάνω πράξη. Έχουμε

```
// composeKTwo :: (b -> M c, a -> M b) -> (a -> M c)
const composeKTwo = (f, g) => x => chain(g(x))(f);
```

Η παραπάνω σχέση ισχύει αν θεωρήσουμε ότι η **chain** είναι ορισμένη για τη μονάδα **M,** διαφορετικά αν υποθέσουμε ότι η μονάδα M συνοδεύεται από το περιβάλλον ενός αντικειμένου στην JavaScript (σύνηθες), τότε θεωρώντας ότι προσφέρεται η λειτουργικότητα **chain** ως μέθοδος, μπορούμε να γράψουμε

```
// composeKTwo :: (b -> M c, a -> M b) -> (a -> M c)
const composeKTwo = (f, g) => x => g(x).chain(f);
```

Επιπρόσθετα, μπορούμε να υλοποιήσουμε μία γενική συνάρτηση **composeKN** η οποία δέχεται ένα πλήθος N συναρτήσεων της μορφής ***x*** → ***M y*,** και επιστρέφει την τελική συνάρτηση που προκύπτει από την πράξη της σύνθεσης Kleisli μεταξύ αυτών. Η συνάρτηση αυτή μπορεί να υλοποιηθεί ακριβώς όπως προσεγγίσαμε την πράξη της σύνθεσης συναρτήσεων της μορφής **x** → ***y*** , καθώς η σύνθεση Kleisli, όπως αποδεικνύεται σχηματίζει μονοειδή σχηματισμό αφού τηρεί την προσεταιριστική ιδιότητα και διαθέτει

ουδέτερο στοιχείο (συνάρτηση **return**). Στην περίπτωση της απλής σύνθεσης αποδείξαμε ότι

$$composeN = mconcat = foldr(mappend)(mempty)$$

όπου $mappend = \circ \ , \quad mempty = id$

Στην περίπτωση της σύνθεσης Kleisli μπορούμε να γράψουμε ισοδύναμα λοιπόν

$$composeKN = mconcat = foldr(mappend)(mempty)$$

όπου $mappend = (<=<) \ , \quad mempty = return$

Για την ακρίβεια, επειδή η συνάρτηση composeN που υλοποιήσαμε στο κεφάλαιο της σύνθεσης, δέχεται ως ορίσματα απευθείας τις συναρτήσεις που έπειτα τις τοποθετεί σε έναν πίνακα, και όχι κατευθείαν τον πίνακα με τις συναρτήσεις, οι υλοποιήσεις των συναρτήσεων στην JavaScript θα έχουν την παρακάτω μορφή

$$composeN' = (\ldots args) => composeN(args)$$

και

$$composeKN' = (\ldots args) => composeKN(args)$$

μετατρέποντας ουσιαστικά τις υπογραφές

$$\text{composeN} :: [y \to z, x \to y, \ldots b \to c, a \to b] \to a \to z$$

και

$$\text{composeKN} :: [y \to M\ z, x \to M\ y, \ldots b \to M\ c, a \to M\ b] \to a \to M\ z$$

σε

$$\text{composeN} :: (y \to z, x \to y, \ldots b \to c, a \to b) \to a \to z$$

και

$$\text{composeKN} :: (y \to M\ z, x \to M\ y, \ldots b \to M\ c, a \to M\ b) \to a \to M\ z$$

Για την **composeKN** έχουμε λοιπόν

```
// composeKTwo :: (b -> M c, a -> M b) -> (a -> M c)
const composeKTwo = (f, g) => x => chain(g(x))(f);

const mempty = pure;
const mappend = composeKTwo;

// foldr :: ((a -> b) -> b) -> b -> [a] -> b
const foldr = f => z => ([first, ...rest]) => first === undefined ? z : f(first,
foldr(f)(z)(rest));

// composeKN :: (y -> M z, x -> M y, ..., b -> M c, a -> M b) -> a -> M z
const composeKN = (...args) => foldr(mappend)(mempty)(args);
```

Φυσικά όπως είπαμε βασική προϋπόθεση ο ορισμός των συναρτήσεων **return/pure** και **chain** για το σχηματισμό της μονάδας **M** που μας ενδιαφέρει να έχει πραγματοποιηθεί.

Αφού αναλύσαμε και την πράξη της σύνθεσης Kleisli, πάμε να δούμε ένα τελευταίο παράδειγμα ενός μικρού προβλήματος στο οποίο δίνει συναρτησιακή λύση ο σχηματισμός μίας μονάδας. Φανταστείτε ότι έχουμε δύο τρύπες στις οποίες τοποθετούμε με τη σειρά έναν αριθμό από μπίλιες. Θέλουμε να γράψουμε ένα πρόγραμμα το οποίο θα δέχεται τον αριθμό των συνολικών κινήσεων, και θα τοποθετεί διαδοχικά στις δύο τρύπες έναν τυχαίο αριθμό από μπάλες. Αν η διαφορά μεταξύ του πλήθος των μπαλών των δύο τρυπών είναι μεγαλύτερη από 20 τότε η τοποθέτηση των μπαλών πρέπει να σταματάει. Επιπρόσθετα πρέπει να κρατάμε το ιστορικό των κινήσεων με τις μπίλιες που τοποθετήθηκαν στην τελική λύση.

Ας μελετήσουμε το πρόβλημα προσεκτικά. Αρχικά θα χρειαστούμε μία δομή που θα αποθηκεύει την κατάσταση των τρυπών ανά ενέργεια. Αν υποθέσουμε ότι η δομή αυτή ονομάζεται **Holes,** τότε το πρόβλημα προσομοιώνεται με μεγάλη ακρίβεια από μία ροή ενεργειών στην οποία θα εφαρμόσουμε μία συνάρτηση της μορφής (**Holes**, **action**) → **Holes**, ένα **reducer** function δηλαδή για την παραγωγή του τελικού αντικειμένου **Holes** που θα προκύπτει μετά από **n** στον αριθμό **actions.** Η προσέγγιση αυτή είναι ορθή, με εξαίρεση το γεγονός ότι δεν έχουμε τη δυνατότητα του ελέγχου της διαφοράς του πλήθους των μπαλών κατά την εκτέλεση της συνάρτησης παρά μόνο στο τέλος, όταν δηλαδή έχουμε στη διάθεση μας το τελικό αντικείμενο τύπου **Holes**. Μία λύση θα ήταν η χρησιμοποίηση της μονάδας **Maybe** που θα ενθυλακώσει ένα αντικείμενο **Holes**, και στις περιπτώσεις που η ζητούμενη διαφορά γίνει μεγαλύτερη της τιμής 20, θα επιστρέφει έναν τύπο **Nothing**

αντί για **Just Holes.** Η λύση αυτή αν και τηρεί τις προϋποθέσεις του προβλήματος δεν δίνει τη δυνατότητα της προώθησης ενός μηνύματος στις περιπτώσεις που η δομή είναι **Nothing,** πράγμα που θα ήταν θεμιτό για την παρουσίαση της διαφοράς που βγήκε εκτός ορίων. Αποδεικνύεται ότι στις περιπτώσεις αυτές υπάρχει ένας γνωστός σχηματισμός μονάδας, και συναρτήτης συνακόλουθα, που επιτρέπει τον διαχωρισμό των τιμών σε δύο κατηγορίες. Η δεξιά κατηγορία υποδηλώνει ομαλό υπολογισμό και λειτουργεί ακριβώς όπως η δομή **Just**. Η αριστερή κατηγορία υποδηλώνει συνήθως ένα σφάλμα στην προσέγγιση ή μία τιμή αναπαράστασης που υποκρύπτει ότι η ροή του προγράμματος έχει πάρει διαφορετική τροπή. Η μονάδα αυτή είναι γνωστή ως **Either Monad** και ο κατασκευαστής τύπου της έχει τη μορφή

$$Either\ a\ b :: Left\ a \mid Right\ b$$

Η συναρτήσεις **return** και **chain** που υλοποιεί η μονάδα είναι απλές και φαίνονται παρακάτω

$$return\ x = Right\ x$$

$$Right\ x \gg= f = f\ x$$

$$Left\ x \gg= f = Left\ x$$

Πάμε να δούμε πως μπορούμε να χτίσουμε τη λύση σιγά σιγά στην JavaScript

Έστω ο κατασκευαστής **holes** που δημιουργεί αντικείμενα τύπου **Holes.** Το αντικείμενο αυτό θα απεικονίζει την κατάσταση των τρυπών στην αρχή, και θα επαναδημιουργείται μετά το πέρας μίας ενέργειας. Έχουμε

```
// holes :: (Number, Number, [String]) -> Holes
const holes = (x, y, message) => ({
    left: x,
    right: y,
    m: message
});
```

Πάμε να δούμε πως θα ορίζαμε τις ενέργειες. Όπως προαναφέραμε θα είναι συναρτήσεις που δέχονται έναν αριθμό, μία κατάσταση **Holes**, και θα παράγουν μία νέα κατάσταση **Holes**. Επειδή όμως θα κάνουμε χρήση της μονάδας **Either** μπορούμε να υποθέσουμε ότι η υπογραφή των συναρτήσεων θα έχει τη μορφή

$$action :: Number \rightarrow Holes \rightarrow Either\ [String]\ Holes$$

Θα έχουμε

```
// smallerThanN :: Number -> Number -> Boolean
smallerThanN = n => x => x < n;

// addLeft :: Number -> Holes -> Either [String] Holes
const addLeft = n => ({ left, right, m }) => smallerThanN(20)(left + n - right) ?
    Right(holes(left + n, right, m.concat([`Added ${n} balls to the left hole`]))) :
    Left(m.concat([`Added ${n} balls to the left hole -> Cannot procced, ${left + n}
- ${right} >= ${20}`]));


// addRight :: Number -> Holes -> Either [String] Holes
const addRight = n => ({ left, right, m }) => smallerThanN(20)(right + n - left) ?
    Right(holes(left, right + n, m.concat([`Added ${n} balls to the right hole`]))) :
    Left(m.concat([`Added ${n} balls to the left hole -> Cannot procced, ${right + n}
- ${left} >= ${20}`]));
```

Παρατηρούμε ότι οι συναρτήσεις δέχονται έναν αριθμό, ένα αντικείμενο τύπου **Holes**, και στη συνέχεια αυξάνουν την αντίστοιχη τρυπά με τον αριθμό αυτό. Αν μετά την αύξηση η διαφορά του πλήθους των μπαλών ανάμεσα στις δύο τρύπες είναι μικρότερη από τον αριθμό 20, τότε επιστρέφουμε την τιμή του αντικειμένου **Holes** εντός μία δομής **Right**, που υποδηλώνει ομαλή κατάσταση και προσδίδει συνέχεια στους υπολογισμούς. Εάν είναι μικρότερη, τότε επιστρέφεται ένας πίνακας με το αλφαριθμητικό "Cannot procced" και τη διαφορά που είχε σαν αποτέλεσμα να τεθεί ο υπολογισμός εκτός ορίων. Η τιμή εγκιβωτίζεται εντός μία δομής **Left** που υποδηλώνει ότι κάποια συνθήκη έχει καταστρατηγηθεί και συνεπώς η μετέπειτα υπολογισμοί παύουν να πραγματοποιούνται. Παρατηρούμε επιπρόσθετα ότι και στις δύο περιπτώσεις η δομή [**String**], είτε εντός της δομής **Holes**, είτε από μόνη της στην περίπτωση της **Left** μορφής, χρησιμεύει ως ένας μονοειδής (**Monoid**) σχηματισμός που συσσωρεύει με τη μορφή αλφαριθμητικών τις ενέργειες που έχουν γίνει. Πάμε να δούμε πως γράφονται οι δομές **Left** και **Right.**

```
// Left :: a -> Left a
const Left = x => ({
    fmap: _ => Left(x),
    chain: _ => Left(x),
    getValue: () => x
});

// Right :: b -> Right b
const Right = x => ({
    fmap: f => Right(f(x)),
    chain: f => f(x),
    getValue: () => x
});
```

Βλέπουμε ότι η υλοποίηση της **chain** που υλοποιούν οι δύο μορφές διαφέρει. Η **Left** μορφή απλά δίνει συνέχεια στους υπολογισμούς επιστρέφοντας κάθε φορά το ίδιο αντικείμενο, ενώ η **Right** από την άλλη παίρνει το εσωτερικό στοιχείο που έχει και το εφαρμόζει στη συνάρτηση εισόδου **f** της **chain,** παράγοντας έτσι μία νέα δομή της μορφής **Right.** Στα πλαίσια αυτά οι συναρτήσεις **return** και **chain**, μπορούν να πάρουν τη μορφή

```
// return/pure :: a -> Right a
const pure = x => Right(x);

// chain :: Either a b -> (b -> Either a c) -> Either a c
const chain = m => f => m.chain(f);
```

Παρατηρούμε ότι η **chain** επιτρέπει την προσπέλαση αντικειμένων της μορφής **Left**, **Right** , παίρνοντας τα υπό τη μορφή παραμέτρων και μεταφράζοντας τη διάταξη σε αλυσιδωτή κλήση.

Με τα εργαλεία που δημιουργήσαμε πλέον συναρτησιακά πάμε να τρέξουμε μία απλή διάταξη ενεργειών, για να δούμε πως αυτή συμπεριφέρεται.

```
const init = holes(0, 0, []);

chain(pure(init))(addLeft(2)).getValue();
//-> { left: 2, right: 0, m: [ 'Added 2 balls to the left hole' ] }
```

και μπορούμε να δώσουμε συνέχεια στη ροή με τον παρακάτω τρόπο

```
const init = holes(0, 0, []);

const a = chain(pure(init))(addLeft(2));

chain(a)(addRight(4)).getValue();
//-> { left: 2, right: 4, m: [ 'Added 2 balls to the left hole', 'Added 4 balls
to the right hole' ] }
```

Βλέπουμε ότι η κλήση της **chain** επιστρέφει κάθε φορά την ενημερωμένη κατάσταση στην οποία βρίσκονται οι τρύπες μέσω του αντικειμένου **Holes.** Ας δούμε τώρα τι θα γίνει εάν μεγαλώσουμε τη διαφορά ανάμεσα στις δύο τρύπες. Έστω ότι βάζουμε 30 μπάλες στη αριστερή τρύπα.

```
const init = holes(0, 0, []);

const a = chain(pure(init))(addLeft(2));

const b = chain(a)(addRight(4));

chain(b)(addLeft(30)).getValue();
//-> [ 'Added 2 balls to the left hole',
//     'Added 4 balls to the right hole',
//     'Added 30 balls to the left hole -> Cannot procced, 32 - 4 >= 20' ]
```

Παρατηρούμε ότι το αποτέλεσμα που επιστράφηκε είναι ένας πίνακας τύπου [**String**], καθώς η δομή που το περικλείει είναι ένας τύπος **Left.** Στο σημείο αυτό ουσιαστικά η ροή έχει παγώσει καθώς η διαφορά ανάμεσα στις δύο τρύπες είναι μεγαλύτερη η ίση με τον αριθμό 20. Οποιαδήποτε διαδοχική κλήση της chain στη δομή Left θα επιστρέψει τον ίδιο πίνακα [**String**] αποφεύγοντας έτσι περιττούς υπολογισμούς. Για παράδειγμα

```
chain(chain(b)(addLeft(30)))(addRight(50)).getValue();
//-> [ 'Added 2 balls to the left hole',
//     'Added 4 balls to the right hole',
//     'Added 30 balls to the left hole -> Cannot procced, 32 - 4 >= 20' ]
```

Βλέπουμε ότι η ενέργεια **addRight(50)** δεν έχει κάποιο αντίκτυπο στο εσωτερικό της δομής **Left.** Ωραία, πάμε τώρα να αυτοματοποιήσουμε λίγο τη διαδικασία έτσι ώστε το πρόγραμμα να είναι σε θέση να φτιάχνει τη διάταξη από **chains** λαμβάνοντας ως είσοδο έναν πίνακα από διαδοχικά **actions.** Η συνάρτηση **chain** θα λειτουργεί ουσιαστικά ως **reducer** function επάνω σε αυτό τον πίνακα. Πάμε να δούμε πως θα παράγαγουμε έναν τέτοιο πίνακα. Έχουμε

```
// createActionTable :: Number -> [Holes -> Either [String] Holes]
const createActionTable = n => Array(n).fill(undefined).map(x =>
Math.floor(Math.random() * 11)).map((x, i) => i % 2 === 0 ? addLeft(x) :
addRight(x));
```

Η **createActionTable** επιστρέφει μία διάταξη της μορφής

$$z = [addLeft(x), addRight(y), \dots], \qquad len\, z == n$$

Όπου οι αριθμού x,y … είναι τυχαίοι αριθμού στο εύρος [0, 10]. Συνεπώς πλέον το μόνο που απομένει να κάνουμε είναι η δίπλωση (**folding**) του συγκεκριμένου πίνακα με τη συνάρτηση **chain.** Έχουμε

```
// foldl :: ((a -> b -> a) -> a -> [b]) -> a
const foldl = (f, z, [first, ...rest]) => first === undefined ? z : foldl(f,
f(z)(first), rest);

// play :: Number -> Either [String] Holes
const play = n => foldl(chain, pure(init), createActionTable(n));
```

Για n = 20, παραθέτουμε τρεις (3) διαδοχικές κλήσεις

```
play(20).getValue();
//-> { left: 40,
//     right: 47,
//     m:
//       ['Added 2 balls to the left hole',
//         'Added 5 balls to the right hole',
//         'Added 7 balls to the left hole',
//         'Added 4 balls to the right hole',
//         'Added 1 balls to the left hole',
//         'Added 6 balls to the right hole',
//         'Added 8 balls to the left hole',
//         'Added 10 balls to the right hole',
//         'Added 2 balls to the left hole',
//         'Added 0 balls to the right hole',
//          Added 8 balls to the left hole',
//         'Added 10 balls to the right hole',
//         'Added 4 balls to the left hole',
//         'Added 4 balls to the right hole',
//         'Added 2 balls to the left hole',
//         'Added 6 balls to the right hole',
//         'Added 1 balls to the left hole',
//         'Added 0 balls to the right hole',
//         'Added 5 balls to the left hole',
//         'Added 2 balls to the right hole'] }
```

```
play(20).getValue();
//-> ['Added 1 balls to the left hole',
//    'Added 7 balls to the right hole',
//    'Added 0 balls to the left hole',
//    'Added 9 balls to the right hole',
//    'Added 4 balls to the left hole',
//    'Added 5 balls to the right hole',
//    'Added 1 balls to the left hole',
//    'Added 8 balls to the left hole -> Cannot procced, 29 - 6 >= 20']
```

```
play(20).getValue();
//-> ['Added 7 balls to the left hole',
//    'Added 1 balls to the right hole',
//    'Added 8 balls to the left hole',
//    'Added 4 balls to the right hole',
//    'Added 10 balls to the left hole -> Cannot procced, 25 - 5 >= 20']
```

Παρατηρούμε ότι στην πρώτη περίπτωση ο υπολογισμός επιτυγχάνεται κανονικά και επιστρέφεται μία δομής τύπου **Right Holes**. Στις άλλες δύο περιπτώσεις όμως έχουμε αποτυχία καθώς η διαφορά των μπαλών γίνεται σε κάποια φάση μεγαλύτερη η ίση από 20 οπότε και η κλήση **play** επιστρέφει έναν τύπο **Left [String].** Τέλος μία διαφορετική προσέγγιση της λύσης του προβλήματος θα ήταν η αξιοποίηση της σύνθεσης Kleisli. Ο πίνακας που παράγει η λειτουργικότητα **createActionTable** περιέχει συναρτήσεις της μορφής $Holes \rightarrow Either\ [String]\ Holes$, πράγμα που σημαίνει ότι δύναται η δημιουργία μίας νέας συνάρτησης που θα προκύψει από τη σύνθεση (Kleisli) τους. Έχουμε

```
// play :: Number -> Either [String] Holes
const play = n => chain(pure(init))(composeKN(...createActionTable(n)));
```

Μία τυχαία κλήση επιστρέφει

```
play(20).getValue();
// -> ['Added 8 balls to the right hole',
//     'Added 3 balls to the left hole',
//     'Added 8 balls to the right hole',
//     'Added 2 balls to the left hole',
//     'Added 5 balls to the right hole',
//     'Added 1 balls to the left hole',
//     'Added 6 balls to the left hole -> Cannot procced, 27 - 6 >= 20']
```

Παρατηρούμε ότι η λειτουργικότητα της **play** είναι ακριβώς ίδια.

Σε αυτό το σημείο θα κλείσουμε με τους σχηματισμούς Μονάδων. Αυτό που πρέπει να κρατήσουμε από την παρούσα ανάλυση είναι ότι οι δομές αυτές προσφέρουν όπως είδαμε, ισχυρά αφαιρετικά πλαίσια τα οποία μπορεί να αξιοποιήσει αποδοτικά ο συναρτησιακός προγραμματισμός για την επίλυση σύνθετων προβλημάτων

5 Συμπεράσματα

Σκοπός της παρούσας διπλωματικής εργασίας ήτανε η εξοικείωση του αναγνώστη με τις βασικές αρχές που διέπουν τον συναρτησιακό προγραμματισμό υπό το πρίσμα της προγραμματιστικής γλώσσας JavaScript. Φυσικά θα ήταν λάθος να πούμε ότι όλα τα παραπάνω περιορίζονται στους κόλπους της συγκεκριμένης γλώσσας και δεν επιδέχονται ενιαία υιοθέτηση και από άλλες γλώσσες. Άλλωστε όπως είδαμε το μοντέλο του συναρτησιακού προγραμματισμού βασίζεται σε ένα ισχυρό, δομημένο μαθηματικά, υπολογιστικό πλαίσιο το οποίο εξοπλίζει τον προγραμματιστή με ένα νέο τρόπο σκέψης, καθολικής εφαρμογής. Το πραγματικό ερώτημα που θα πρέπει να γεννάται είναι το κατά πόσο ορισμένα χαρακτηριστικά που διαθέτει μία γλώσσα, ευνοούν ή δυσχεραίνουν τον προγραμματιστή στην υιοθέτηση του συναρτησιακού τρόπου γραφής σε αυτή. Στην προκειμένη περίπτωση θα πρέπει να αναρωτηθούμε το εξής "Είναι ικανό το περιβάλλον της JavaScript να φιλοξενήσει κώδικα που πηγάζει από τον συναρτησιακό τρόπο σκέψης και αντίληψης;" Η απάντηση στο παραπάνω ερώτημα είναι συνετό να αναλυθεί σε δύο συνιστώσες.

Η πρώτη είναι καθαρά τεχνική. Εκ πρώτης όψεως θα λέγαμε ότι ναι, η JavaScript είναι ικανή να υιοθετήσει τον συναρτησιακό τρόπο σύνταξης από τη στιγμή που η έννοια της συνάρτησης υφίσταται και μπορεί να μεταχειριστεί ως δεδομένο ή όπως διατυπώθηκε η συνάρτηση μπορεί να συμπεριφερθεί ως πολίτης πρώτης κατηγορίας (**first class citizen**). Εν μέρη θα είχαμε δίκαιο και απόδειξη αποτελεί η παρούσα εργασία στην οποία παρουσιάστηκαν μία πληθώρα συναρτησιακών εφαρμογών. Αλλά τα πράγματα δεν είναι τόσο μονόπλευρα και πάντοτε υπάρχει ένα κόστος. Η JavaScript είναι μία δυναμική γλώσσα, με απουσία σταδίου μεταγλώττισης (**complilation**) και αντίστοιχων σφαλμάτων (το στάδιο JIT που διαθέτει η JavaScript λαμβάνει μέρος κατά την εκτέλεση και όχι πριν (Wikipedia/Just-in-time_compilation, n.d.)), με απουσία τύπων και είναι στραμμένη στην πρωτότυπη αντικειμενοστρέφεια, ευνοώντας ή επιτρέποντας ορθότερα τις παρενέργειες (**side effects**). Όλες οι παραπάνω ιδιότητες, όπως πρέπει πλέον να γνωρίζουμε, έρχονται σε σύγκρουση με τον συναρτησιακό μοντέλο γραφής. Αυτό πρακτικά σημαίνει ότι το κόστος που θα προκύψει από την προσπάθεια της εξουδετέρωσης των αθέμιτων αυτών

χαρακτηριστικών θα μεταβιβαστεί με τον ένα ή τον άλλο τρόπο στον προγραμματιστή. Αναφέραμε ότι η απουσία τύπων για παράδειγμα μπορεί να αντιμετωπιστεί με την προσθήκη σχολίων πάνω από τον ορισμό μίας συνάρτησης. Η απουσία επιστροφής σφαλμάτων κατά τη μεταγλώττιση μπορεί να αντικατασταθεί με ελέγχους που επιτρέπουν την ανάδειξη παρόμοιας συμπεριφοράς μέσω σφαλμάτων που θα προκύπτουν κατά το στάδιο της εκτέλεσης. Άλλα χαρακτηριστικά όπως για παράδειγμα η ύπαρξη τροποποιήσεων (**mutations**) και συνακόλουθα οι παρενέργειες (**side effects**) δεν επιδέχονται κάποια λύση ελαφριάς μορφής και εναπόκεινται καθαρά στην προσοχή και συγκέντρωση του προγραμματιστή για την αποφυγή τους. Τέλος η γλώσσα δεν παρέχει εγγενής υλοποιήσεις βασικών συναρτησιακών πράξεων, όπως για παράδειγμα η σύνθεση συναρτήσεων ή η αυτόματη μετατροπή των συναρτήσεων στις **curried** εκδοχές τους. Αυτό φυσικά έχει ως συνέπεια την παραγωγή περισσότερου κώδικα από πλευράς του προγραμματιστή για την εφαρμογή των συγκεκριμένων λειτουργιών ή την εξάρτηση του από τρίτες βιβλιοθήκες συναρτησιακού σκοπού (όπως αυτή που υλοποιήθηκε στα πλαίσια της εργασίας ή άλλες γνωστές όπως **underscore.js**, **lodash.js**, **rambda.js**). Παρατηρούμε δηλαδή ότι το περιβάλλον δεν είναι και τόσο φιλικό όσον αφορά τη συναρτησιακή προγραμματιστική προσέγγιση στη γλώσσα της JavaScript και σε καμία περίπτωση δεν συγκρίνεται με περιβάλλοντα αμιγώς συναρτησιακών γλωσσών όπως είναι αυτά της **Haskell** ή της **Closure**. Παρά ταύτα υιοθέτηση του συγκεκριμένου τρόπου γραφής σε ένα τέτοιο περιβάλλον δεν είναι απαγορευτική και το κόστος που πληρώνει στο τέλος ο προγραμματιστής μειώνεται αναλογικά με το βαθμό εξοικείωσης που παρουσιάζει κατά την εφαρμογή συναρτησιακών λύσεων στο συγκεκριμένο οικοσύστημα της γλώσσας. Τέλος και για να καταλήξουμε και στη δεύτερη συνιστώσα της απάντησης, μια πιο δραστική λύση στα προβλήματα που παρουσιάζονται παραπάνω είναι η υιοθέτηση ορισμένων διαλέκτων της JavaScript. Λόγος γίνεται για τις γλώσσες **TypeScript** και **Elm** οι οποίες μεταφράζουν τον κώδικα που έχει γραφτεί για το περιβάλλον τους σε κώδικα JavaScript με τη χρήση **transpilers**. Η πρώτη αποτελεί υπερσύνολο της γλώσσας JavaScript και παρέχει τη δυνατότητα ορισμού τύπων ενώ η δεύτερη είναι αμιγώς συναρτησιακή με αρκετά όμως διαφορετική σύνταξη. Φυσικά η απήχηση που έχουν οι συγκεκριμένες διάλεκτοι δεν η ίδια με αυτή της JavaScript (**vanilla JS**) η οποία υποστηρίζεται εγγενώς από όλους τους δημοφιλής περιηγητές, όμως αξίζει να σημειωθεί

ότι σχηματίζουν ένα ελκυστικό περιβάλλον για τη συγγραφή συναρτησιακού κώδικα (η **Elm** όπως είπαμε το επιβάλλει).

Η δεύτερη συνιστώσα έχει θεωρητικό χαρακτήρα. Αυτό που πρέπει να κατανοήσει ο αναγνώστης είναι ότι το μοντέλο του συναρτησιακού προγραμματισμού είναι πάνω από τις γλώσσες. Φυσικά δεν παραβλέπουμε όσα αναφέραμε παραπάνω, ότι δηλαδή ορισμένα περιβάλλοντα είναι πιο προσιτά στην παραγωγή κώδικα συναρτησιακού χαρακτήρα από κάποια άλλα, όμως στη ζυγαριά της κρίσης πρέπει να συνυπολογιστούν και άλλοι παράγοντες. Πρώτον η συναρτησιακή σχεδίαση αποτελεί όπως είπαμε ένα διαφορετικό τρόπο αντίληψης κατά τη διαδικασία ανάπτυξης λογισμικού. Σίγουρα λοιπόν αυτό την καθιστά καταλληλότερο μοντέλο για την αντιμετώπιση ορισμένων προβλημάτων ενώ παράλληλα μπορεί να παρουσιάσει αδυναμίες στην επίλυση κάποιων άλλων. Το γεγονός αυτό ενεργοποιεί τον προγραμματιστή με τη δυνατότητα της επιλογής. Αν για παράδειγμα θεωρούμε ότι το πρόβλημα που προσπαθούμε να αντιμετωπίσουμε επιδέχεται αρκετά ευκολότερη λύση με την εφαρμογή του συναρτησιακού τρόπου σκέψης, και οποιαδήποτε άλλη προσέγγιση παρουσιάζει δυσχέρειες, τότε γιατί να μην προχωρήσουμε με τη συναρτησιακή λύση; Ακόμα και αν το περιβάλλον επιβαρύνει τον προγραμματιστή με κάποιο κόστος, όπως αυτό της JavaScript. Το δεύτερο σημείο που πρέπει να προσθέσουμε είναι ότι ο βαθμός ελευθερίας που παρουσιάζει ο συναρτησιακός προγραμματισμός κατά την εφαρμογή του δεν είναι περιορισμένος. Η κατάσταση δεν είναι ασπρόμαυρη. Το βάθος και η αυστηρότητα την οποία θα υιοθετήσει ένας προγραμματιστής κατά τη συναρτησιακή προσέγγιση είναι υποκειμενικό μέγεθος και καθορίζεται από τον ίδιο. Αυτό σημαίνει ότι το πάντρεμα του συναρτησιακού μοντέλου με κάποιο άλλο, στην περίπτωση της JavaScript της πρωτότυπης αντικειμενοστρέφειας, δεν αποκλείεται. Μάλιστα στη συγκεκριμένη γλώσσα μία τέτοια προσέγγιση είναι η πλέον ενδεδειγμένη καθώς με την κατάλληλη προσοχή τα οφέλη της πρωτότυπης αντικειμενοστρέφειας μπορούν να διατηρηθούν. Φυσικά τα οφέλη του συναρτησιακού προγραμματισμού φθίνουν όσες περισσότερες είναι οι παραδοχές που γίνονται και καταστρατηγούν τις βασικές αρχές του. Βρίσκεται στο χέρι του προγραμματιστή όμως, να εντοπίσει εκείνο το σημείο ισορροπίας στον κώδικα του, στο οποίο μεγιστοποιεί τα γενικά κέρδη που μπορεί να αποκομίσει κατά το στάδιο της ανάπτυξης ενός προγράμματος στη συγκεκριμένη γλώσσα. Πρακτικά, επιδίωξη αποτελεί ο διαχωρισμός του αγνού κώδικα (**pure code**) από το μη αγνό, αυτόν

δηλαδή που παρουσιάζει κάποιες παρενέργειες (**side effects**). Το υβριδικό αυτό μοντέλο ανάπτυξης λειτουργεί και γνωρίζει άνθηση στην JavaScript. Απόδειξη αποτελούν δημοφιλής βιβλιοθήκες (State of JavaScript, 2018) ανάπτυξης web προγραμμάτων (**React**) και διαχείρισης καταστάσεων (**Redux**) οι οποίες υιοθετούν αρχές συναρτησιακού προγραμματισμού και προσφέρουν στον προγραμματιστή παράθυρα επικοινωνίας με σκοπό τη δημιουργία νέων αγνών λειτουργικοτήτων (**pure functions**) ή τη μεταβολή κάποιων καταστάσεων, δηλαδή τη δημιουργία παρενεργειών (**side effects**). Φυσικά αν μελετήσει κανείς τον κώδικα υλοποίησης των συγκεκριμένων βιβλιοθηκών (React-GitHub, n.d.) (Redux-GitHub, n.d.) θα δει ότι αυτός δεν είναι καθαρά συναρτησιακός, στην αυστηρή του μορφή, αλλά αρκεί το γεγονός ότι στηρίζεται στις βασικές ιδέες του.

Ανακεφαλαιώνοντας μπορούμε να πούμε με βεβαιότητα ότι ο προγραμματιστής της JavaScript μόνο να κερδίσει έχει από την αξιοποίηση του μοντέλου ανάλυσης που ορίζει ο συναρτησιακός προγραμματισμός. Ως εκ τούτου, η εκπαίδευση ή έστω γνωριμία του προγραμματιστή με αυτό θα λέγαμε ότι κρίνεται επιβεβλημένη

Παράρτημα Α (Προγραμματιστική βιβλιοθήκη radiancejs)

Στο παράρτημα αυτό θα δοθούν κατευθυντήριες οδηγίες για την εύρεση και την αξιοποίηση της προγραμματιστικής βιβλιοθήκης που υλοποιήθηκε στα πλαίσια της παρούσας διπλωματικής εργασίας. Βασικός σκοπός της δημιουργίας της αποτέλεσε η ανάγκη για μία πιο οργανωμένη και αποδοτική αποτύπωση της θεωρίας που αναλύθηκε στις προηγούμενες σελίδες. Μπορούμε να ισχυριστούμε ότι το παραπάνω υλικό βρίσκεται πλέον σε μία μορφή που είναι προγραμματιστικά αξιοποιήσιμη σε μεγαλύτερο βαθμό. Η βιβλιοθήκη αυτή περιλαμβάνει έτοιμο κώδικα συναρτησιακού προσανατολισμού γραμμένο σε γλώσσα JavaScript που κύριο σκοπό έχει την αρωγή του αναγνώστη κατά την εκμάθηση του συναρτησιακού τρόπου γραφής. Πέρα από την ευκολότερη μελέτη του πηγαίου κώδικα, ο οποίος πλέον βρίσκεται οργανωμένος σε αποθετήριο, η βιβλιοθήκη μπορεί να βοηθήσει και πρακτικά με την ενσωμάτωση της συναρτησιακής λειτουργικότητας σε άλλα προγράμματα ή προγραμματιστικές εργασίες που καλούμαστε να υλοποιήσουμε.

Η ανάπτυξη της βιβλιοθήκης πραγματοποιήθηκε στην γλώσσα της JavaScript κατά τα πρότυπα που ορίζει η ES7 προδιαγραφή. Αρχιτεκτονικά η βιβλιοθήκη ακολουθεί αρθρωτό σχηματισμό (ES6 modules) με την βοήθεια του εργαλείου Webpack και με το τελικό αρχείο εξαγωγής (distribution file) να έχει UMD (universal module definition) χαρακτηριστικά για τη ταυτόχρονη υποστήριξη του σε περιηγητές αλλά και στον περιβάλλον της NodeJS. Το αποθετήριο που χρησιμοποιήθηκε είναι το **GitHub** ενώ η διανομή του πακέτου (βιβλιοθήκης) πραγματοποιείται από το **Npm registry.** Παρακάτω φαίνονται οι αντίστοιχοι σύνδεσμοι (URL) εύρεσης της βιβλιοθήκης

GitHub

https://github.com/felichio/radiancejs

Npm

https://www.npmjs.com/package/radiancejs

Συνοπτικά μπορούμε να πούμε ότι η βιβλιοθήκη προσφέρει πέντε βασικές διευκολύνσεις.

1) Περιέχει μία πληθώρα συναρτήσεων υψηλής τάξης (όπως η πράξη της σύνθεσης. της σύνθεσης Kleisli, της τεχνικής currying).

2) Περιέχει κάποιες από τις βασικές μεθόδους των πινάκων στην συναρτησιακή τους μορφή (διαχωρισμένη δηλαδή από το περιβάλλον του αντικειμένου Array.prototype).

3) Προσομοιώνει την διασυνδεμένη λίστα ως μία αναδρομική δομή δεδομένων, καθαρά συναρτησιακή (όπως στην LISP)

4) Επιτρέπει την δημιουργία ροών μέσω της τεχνικής της καθυστερημένης αποτίμησης που γίνεται διαθέσιμη χάρη στην συναρτησιακή φύση της δομής της λίστας.

5) Επιτρέπει την μεταχείριση όλων των παραπάνω ως σχηματισμούς Μονάδων (Monads) διαθέτοντας την βασική συνάρτηση chain

Για περισσότερες πληροφορίες παροτρύνουμε τον αναγνώστη να διαβάσει τις βασικές περιγραφές στο αποθετήριο GitHub. Πέρα από οδηγίες εγκατάστασης (installation), σχηματισμού (building), και ελέγχου (testing) της βιβλιοθήκης δίνονται επιπρόσθετα μία σειρά από παραδείγματα χρήσης. Στον φάκελο source όπου βρίσκεται ολόκληρος ο πηγαίος κώδικας, ο αναγνώστης μπορεί να μελετήσει μεμονωμένα τις υλοποιήσεις των συναρτήσεων που τον ενδιαφέρουν αφού όλες συνοδεύονται από μία βασική περιγραφή, από την υπογραφή τους κατά τα πρότυπα Hindley-Milner καθώς και ξεχωριστά παραδείγματα χρήσης. Τέλος να πούμε ότι ο αναγνώστης είναι σε θέση να εμπλουτίσει, να τροποποιήσει και να ελέγξει τον κώδικα όπως αυτός επιθυμεί.

Βιβλιογραφία

(n.d.). Retrieved from Wikipedia/Composition_over_inheritance: https://en.wikipedia.org/wiki/Composition_over_inheritance

(n.d.). Retrieved from compatibility table ES6: http://kangax.github.io/compat-table/es6/

(n.d.). Retrieved from Wikipedia/Functional_programming: https://en.wikipedia.org/wiki/Functional_programming

(n.d.). Retrieved from Wikipedia/Just-in-time_compilation: https://en.wikipedia.org/wiki/Just-in-time_compilation

(n.d.). Retrieved from React-GitHub: https://github.com/facebook/react

(n.d.). Retrieved from Redux-GitHub: https://github.com/reduxjs/redux

(2018). Retrieved from State of JavaScript: https://stateofjs.com/

Abelson, H., & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programms.* Cambridge: The MIT press.

Atencio, L. (2016). *Functional Programming in Javascript.* New York: Manning Publications.

Backus, J. (1978, August). Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *ACM Turing Award Lecture* , pp. 613-641.

Braithwaite, R. (2017). *JavaScript Allongé.* Lean Publishing.

Crockford, D. (2008). *JavaScript: The Good Parts.* Sebastopol: O'Reilly Media.

Hughes, J. (1990). Why functional programming matters. *Research Topics in Functional Programming, ed. D Turner*, 17-42.

Kereki, F. (2017). *Mastering JavaScript Functional Programming.* Birmingham: Packt Publishing.

Lipovaca, M. (2011). *Learn You a Haskell for Great Good.* San Francisco: No Starch Press.

Lonsdorf, B. (2015). *Mostly Adequate Guide to Functional Programming.*

Michaelson, G. (2011). *An introduction to functional programming through lambda calculus.* Edinburgh: Dover Publications.

Milewski, B. (2018). *Category Theory for Programmers.* Creative Commons.

PopularitY of Programming Language. (2019, May). Retrieved from PopularitY of Programming Language: http://pypl.github.io/PYPL.html

Simpson, K. (2014). *You Don't Know JS_ Scope & Closures.* O'Reilly Media.

Simpson, K. (2015). *You Don't Know Async And Performance.* O'Reilly Media.

Thompson, S. (1991). *Type Theory and Functional Programming.* Canterbury: Addison-Wesley.