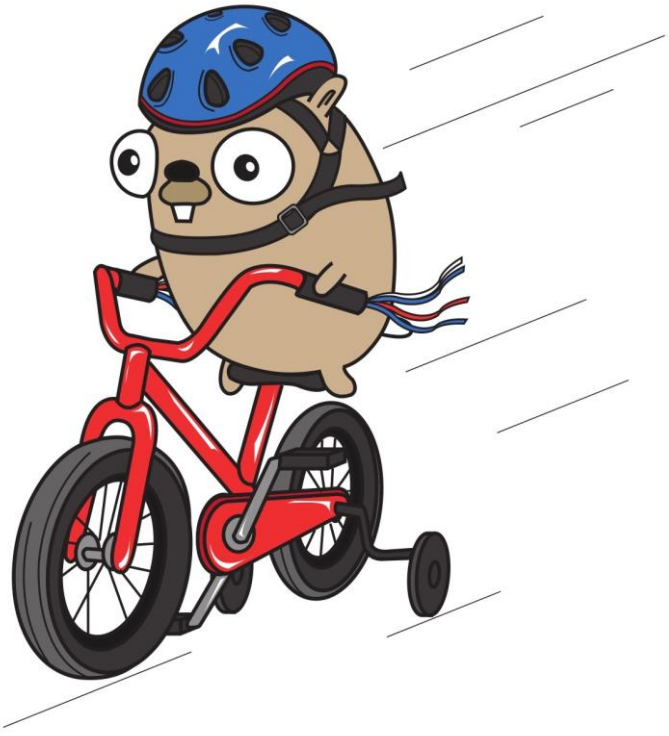
ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΝΙΚΩΝ ΜΗΧΑΝΙΚΩΝ

Βασικές συναρτήσεις της γλώσσας προγραμματισμού GO



Εισηγητής:
Δ. Πλιάκης

Συγγραφέας:
Μουτζούρης Γεώργιος
Ζαχαράκης Κων/νος

ΧΑΝΙΑ 2013

Ευχαριστίες

Ευχαριστούμε τον επιβλέποντα κ. Πλιάκη Δημήτριο, καθηγητή του τμήματος Ηλεκτρονικής του Τεχνολογικού Εκπαιδευτικού Ιδρύματος Κρήτης για την άψογη συνεργασία που είχαμε.

Χανιά 2013,
Μουτζούρης Γεώργιος
Ζαχαράκης Κωνσταντίνος

Περίληψη

Στη παρούσα πτυχιακή εργασία παρουσιάζεται μια ανοικτού κώδικα γλώσσα προγραμματισμού της Google με το όνομα Go. Με αυτή τη γλώσσα προγραμματισμού γίνεται η ανάπτυξη κώδικα για τον υπολογισμό της αναδρομικής εξίσωσης μη-γραμμικής διάχυσης.

Abstract

This thesis presents Google's open source programming language named Go. With this programming language is being the development of code for calculating the recurrence nonlinear diffusion.

Εισαγωγή

Η Go είναι μια ανοιχτού κώδικα γλώσσα προγραμματισμού που σχεδιάστηκε και αναπτύχθηκε για πρώτη φορά από τη Google Inc. το Σεπτέμβριο του 2007 από τους Robert Griesemer, Rob Pike και Ken Thompson. Προσπαθεί να συνδυάσει την ταχύτητα μιας δυναμικής γλώσσας, όπως η Python, με την ασφάλεια και την καλή απόδοση μιας compiled (σ.σ. μεταγλωττιζόμενης) γλώσσας, όπως η C ή η C++. Αν και σε δοκιμαστικό στάδιο η Go έχει σχεδιαστεί ειδικά για την ανάπτυξη προγραμμάτων και web apps που θα τρέχουν σε συστήματα με πολυπύρηνους επεξεργαστές
Ακόμα και μεγάλες βιβλιοθήκες μπορούν, σύμφωνα με την Google, να μεταγλωττιστούν σε λίγα δευτερόλεπτα, με τον κώδικα να εκτελείται με την ταχύτητα της C.Η σύνταξη της Go είναι σε γενικές γραμμές παρόμοια με εκείνη της C: Τα μπλοκ κώδικα είναι μέσα σε άγκιστρα και οι κοινές δομές ελέγχου ροής περιλαμβάνουν for, switch και if. Σε αντίθεση με τη C, η γραμμή που τελειώνει με ερωτηματικά είναι προαιρετική, οι δηλώσεις μεταβλητών γράφονται διαφορετικά και είναι συνήθως προαιρετικές και οι λέξεις-κλειδιά go και select έχουν εισαχθεί για να υποστηρίζετε ο ταυτόχρονος προγραμματισμός. Νέοι ενσωματωμένοι τύποι περιλαμβάνουν πίνακες, slices, χάρτες και κανάλια για την επικοινωνία μεταξύ των νημάτων.
Σκοπός της πτυχιακής αυτής είναι για αρχή να δώσει στον αναγνώστη τις θεωρητικές γνώσεις βάση του εγχειριδίου “An Introduction in Programming in GO” by Caleb Doxsey. Ύστερα κατόπιν παραδειγμάτων (κώδικες) να μπορέσει να δημιουργήσει το δικό του πρόγραμμα. Τέλος, δημιουργούμε και αναλύουμε ένα πρόγραμμα το οποίο επιλύει την αναδρομική εξίσωση της μη-γραμμικής διάχυσης.

Περιεχόμενα

1 Ξεκινώντας

Προγραμματισμός είναι η τέχνη, η δεξιότητα και η επιστήμη του να γράφεις προγράμματα τα οποία προσδιορίζουν το πώς οι ηλεκτρονικοί υπολογιστές λειτουργούν.
Το βιβλίο αυτό θα σας διδάξει πως να γράφετε προγράμματα χρησιμοποιώντας μια γλώσσα προγραμματισμού σχεδιασμένη από την Google οι οποία ονομάζεται Go.

Η Go είναι μια γλώσσα προγραμματισμού γενικής χρήσης με προηγμένα χαρακτηριστικά και «καθαρή» σύνταξη. Λόγω της μεγάλης της διαθεσιμότητας σε μια ποικιλία από πλατφόρμες, έχει μια καλά τεκμηριωμένη βιβλιοθήκη και εστιάζει σε καλές αρχές προγραμματιστικού σχεδιασμού.
Η Go είναι ιδανική για να μάθετε σαν πρώτη γλώσσα προγραμματισμού.
Η διαδικασία που χρησιμοποιούμε για να γράψουμε ένα πρόγραμμα χρησιμοποιώντας την Go (και τις περισσότερες γλώσσες προγραμματισμού) είναι αρκετά απλή:

1. Συγκεντρώστε τις απαιτήσεις
2. Βρείτε μια λύση
3. Γράψτε πηγιαίο κώδικα για την υλοποίηση της λύσης
4. Μεταγλωττίστε τον πηγιαίο κώδικα σε ένα εκτελέσιμο
5. Τρέξτε και δοκιμάστε το πρόγραμμα για να σιγουρευτείτε ότι λειτουργεί

Η διαδικασία αυτή είναι επαναληπτική και τα βήματα συνήθως επικαλύπτονται. Αλλά πριν γράψουμε το πρώτο μας πρόγραμμα στην Go υπάρχουν κάποιες προαπαιτούμενες έννοιες που χρειάζεται να κατανοήσουμε.

1.1 Αρχεία και Φάκελοι

Αρχείο είναι μια συλλογή από δεδομένα αποθηκευμένα ως μια μονάδα με ένα όνομα.
Τα μοντέρνα λειτουργικά συστήματα (όπως Windows ή
Mac OSX) περιέχουν εκατομμύρια αρχεία που αποθηκεύουν μια μεγάλη ποικιλία διαφορετικών τύπων πληροφοριών – τα πάντα, από έγγραφα κειμένου, εκτελέσιμα αρχεία έως αρχεία πολυμέσων.

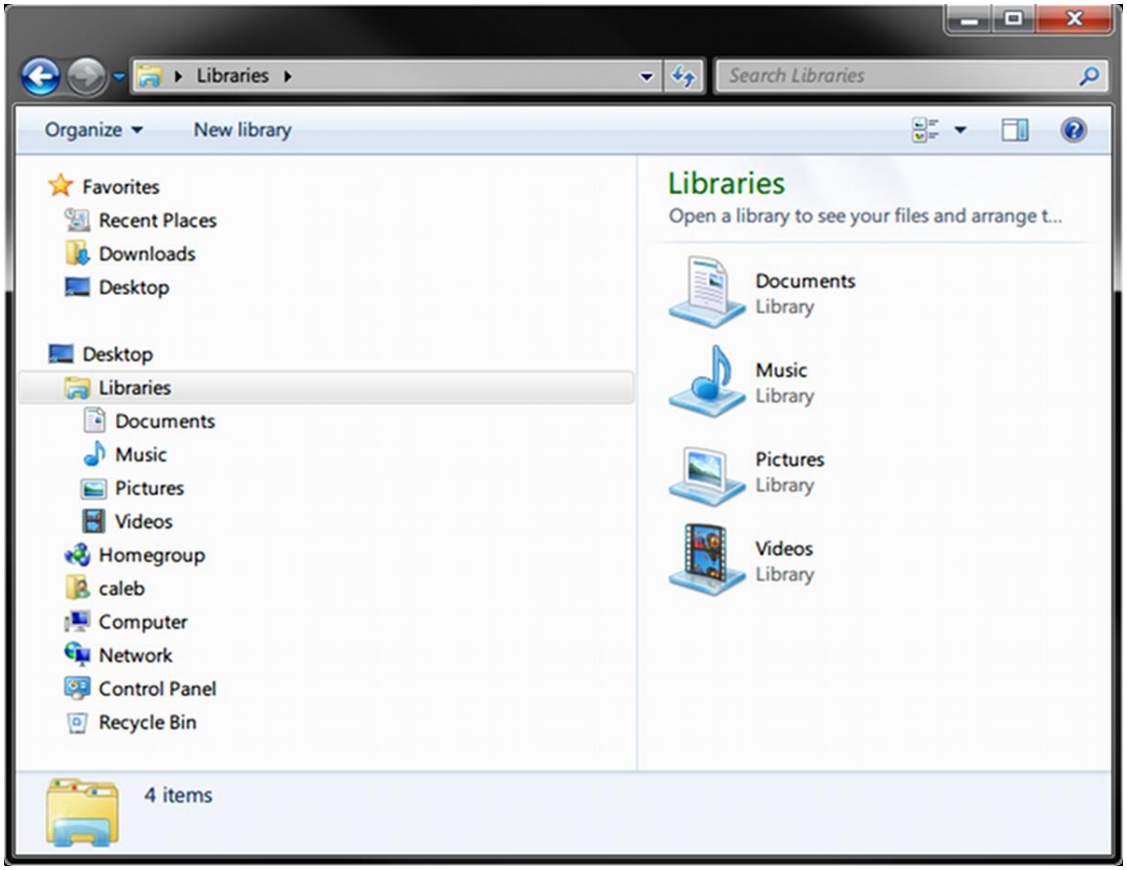
Όλα τα αρχεία αποθηκεύονται με τον ίδιο τρόπο σε ένα ηλεκτρονικό υπολογιστή:
όλα έχουν ένα όνομα, ένα καθορισμένο μέγεθος (μετρούμενο σε bytes) και ένα τύπο.
Τυπικά, ο τύπος των αρχείων καθορίζετε από την κατάληξή τους. – το τμήμα του ονόματος του αρχείου που υπάρχει μετά την τελευταία **.**. Για παράδειγμα, ένα αρχείο με το όνομα hello.txt έχει την κατάληξη txt η οποία χρησιμοποιείται για να

εκπροσωπήσει αρχεία κειμένου.

Οι φάκελοι (folders) (καλούνται επίσης κατάλογοι) χρησιμοποιούνται για να ομαδοποιήσουν αρχεία. Μπορούν επίσης να περιέχουν και άλλους φακέλους.
Στα Windows, οι προορισμοί (τοποθεσίες) των αρχείων και φακέλων αντιπροσωπεύονται με το χαρακτήρα \ (backslash), για παράδειγμα: C:\Users\john\example.txt. example.txt είναι το όνομα του αρχείου, που εμπεριέχεται στο φάκελο john, ο οποίος εμπεριέχεται στο φάκελο Users ο οποίος είναι αποθηκευμένος στον οδηγό C (ο οποίος αντιπροσωπεύει τον κύριο σκληρό δίσκο των Windows). Στα OSX (και στα περισσότερα λειτουργικά συστήματα) ο προορισμός των αρχείων και φακέλων αντιπροσωπεύεται με το χαρακτήρα / (forward slash), για παράδειγμα: /Users/john/example.txt. Όπως στα Windows, το:example.txt είναι το όνομα του αρχείου το οποίο εμπεριέχεται στο φάκελο john,ο οποίος βρίσκεται στο φάκελο Users. Αντίθετα με τα Windows,τα OSX δεν προσδιορίζουν το γράμμα του οδηγού (σκληρού δίσκου) στον οποίο είναι αποθηκευμένο το αρχείο.

Windows

Στα Windows, τα αρχεία και οι φάκελοι μπορούν να πλοηγηθούν χρησιμοποιώντας την "Εξερεύνηση των Windows" (Windows Explorer) (προσβάσημο κάνοντας διπλό κλίκ "Ο Υπολογιστής μου" (My computer) ή πληκτρολογώντας win+e)



OSX

Στα OSX, τα αρχεία και οι φάκελοι μπορούν να πλοηγηθούν χρησιμοποιώντας την "Αναζήτηση" (Finder) (προσβάσιμο κάνοντας κλίκ στο εικονίδιό της – στο εικονίδιο με το πρόσωπο στην κάτω αριστερή μπάρα).

1.2 Το Τερματικό

Οι περισσότερες από τις αλληλεπιδράσεις που έχουμε σήμερα με τους Η/Υ είναι μέσα από εξελιγμένα γραφικά περιβάλλοντα χρήστη (GUIs). Χρησιμοποιούμε πληκτρολόγιο, ποντίκι και οθόνες αφής για να επιδράσουμε σε οπτικά κουμπιά ή άλλου είδους χειρισμούς που εμφανίζονται στην οθόνη.

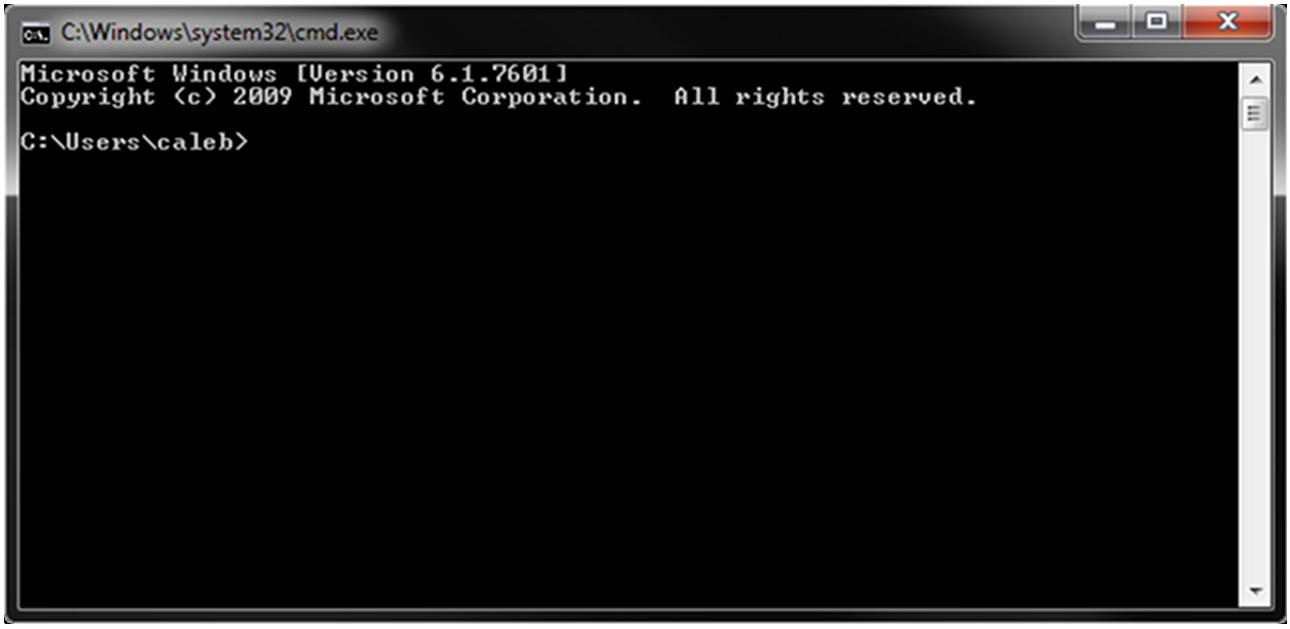
Δεν ήταν πάντα το ίδιο. Πριν το GUI είχαμε το τερματικό – μια απλούστερη διεπαφή κειμένου στον υπολογιστή όπου αντί για το χειρισμό κουμπιών στην οθόνη δίναμε εντολές και λαμβάναμε απαντήσεις. Είχαμε μια συνομιλία με τον Η/Υ.

Και παρόλο που μπορεί να φαίνεται ότι το μεγαλύτερο μέρος του κόσμου της πληροφορικής έχει αφήσει πίσω το τερματικό σαν ένα απομεινάρι του παρελθόντος , η αλήθεια είναι ότι εξακολουθεί να είναι η θεμελιώδης διεπαφή χρήστη, χρησιμοποιούμενη από τις περισσότερες γλώσσες προγραμματισμού στους

περισσότερους Η/Υ. Η γλώσσα προγραμματισμού Go δε διαφέρει, έτσι, πριν γράψουμε ένα πρόγραμμα στη Go πρέπει να έχουμε μια στοιχειώδη κατανόηση της λειτουργίας του τερματικού.

Windows

Στα Windows το τερματικό (γνωστό και ως γραμμή εντολών) μπορεί να ενεργοποιηθεί πληκτρολογώντας τα πλήκτρα “windows key + r” (κρατήστε πατημένο το windows key και πατήστε r), πληκτρολογώντας cmd.exe και πατώντας enter.
Πρέπει να δείτε ένα μαύρο παράθυρο να εμφανίζεται που μοιάζει με αυτό:



Από προεπιλογή η γραμμή εντολών ξεκινάει στο κατάλογο home.
(Στη περίπτωσή μου αυτό είναι C:\Users\caleb) Μπορείτε να εκδώσετε εντολές πληκτρολογώντας τες και πατώντας enter. Δοκιμάστε να εισάγετε την εντολή dir, η οποία εμφανίζει τα περιεχόμενα του καταλόγου.
Πρέπει να δείτε κάτι σαν αυτό:

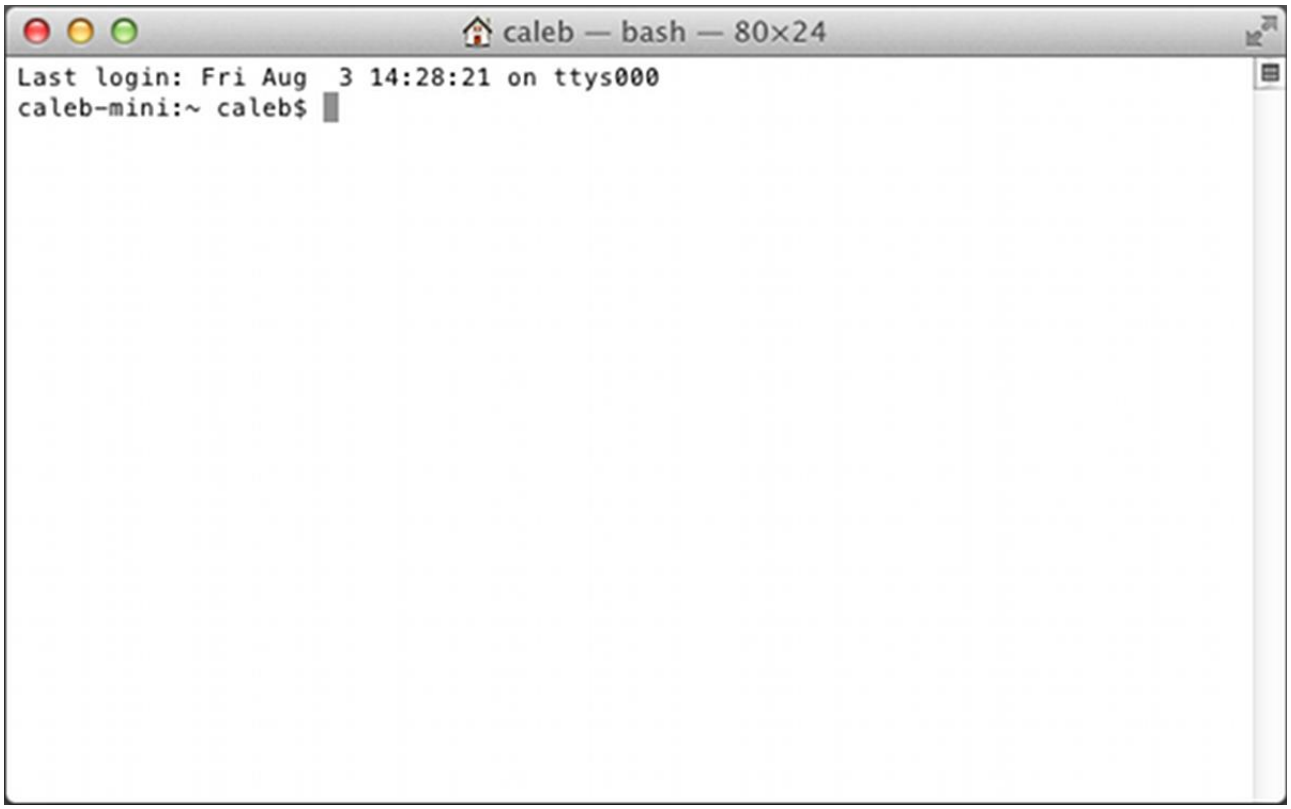
```
C:\Users\caleb>dir
 Volume in drive C has no label.
 Volume Serial Number is B2F5-F125
```

Ακολουθούμενο έπο μια λίστα με τα αρχεία και φακέλους που εμπεριέχονται στον κατάλογο home. Μπορείτε να αλλάξετε καταλόγους χρησιμοποιώντας την εντολή cd. Για παράδειγμα, υποθέτουμε ότι έχετε ένα φάκελο με το όνομα Desktop. Μπορείτε

να δείτε τα περιεχόμενά του εισάγοντας cd Desktop και μετά dir. Για να επιστρέψετε στο κατάλογο home μπορείτε να χρησιμοποιήσετε το όνομα του ειδικού καταλόγου .. (δυο τελείες η μια δίπλα στην άλλη): cd .. Μια τελεία αναπαριστά τον τρέχον φάκελο έτσι η cd. δεν κάνει τίποτα. Υπάρχουν ακόμα πολλές εντολές που μπορείτε να χρησιμοποιήσετε, αλλά αυτές είναι αρκετές για να ξεκινήσετε.

OSX

Στα OSX το τερματικό μπορεί να προσπελαστεί πηγαίνοντας στο:
Finder → Applications → Utilities → Terminal.
Πρέπει να δείτε ένα παράθυρο όπως αυτό:



Από προεπιλογή το τερματικό ξεκινάει στο κατάλογο home .
(Στη περίπτωσή μου είναι: /Users/caleb). Εκδίδεται εντολές πληκτρολογώντας τες και πατώντας "enter". Δοκιμάστε να εισάγετε την εντολή ls, που εμφανίζει τα περιεχόμενα του καταλόγου.
Πρέπει να δείτε κάτι σαν αυτό:

```
caleb-min:~ caleb$ ls
Desktop      Downloads      Movies     Pictures
Documents    Library        Music      Public
```

Αυτά είναι τα αρχεία και οι φάκελοι που εμπεριέχονται στον κατάλογο home (σε αυτή τη περίπτωση δεν υπάρχουν αρχεία). Μπορείτε να αλλάξετε καταλόγους χρησιμοποιώντας την εντολή cd. Για παράδειγμα υποθέτουμε ότι έχετε ένα φάκελο με όνομα Desktop. Μπορείτε να δείτε τα περιεχόμενά του εισάγοντας cd Desktop και κατόπιν εισάγοντας ls. Για να επιστρέψετε στο κατάλογο home μπορείτε να χρησιμοποιήσετε το όνομα ειδικού καταλόγου .. (δυο τελείες η μια δίπλα στην άλλη): cd ... Η μονή τελεία αντιπροσωπεύει το τρέχων φάκελο, έτσι, cd. δεν κάνει τίποτα. Υπάρχουν ακόμα πολλές εντολές που μπορείτε να χρησιμοποιήσετε, αλλά αυτές είναι αρκετές για να ξεκινήσετε.

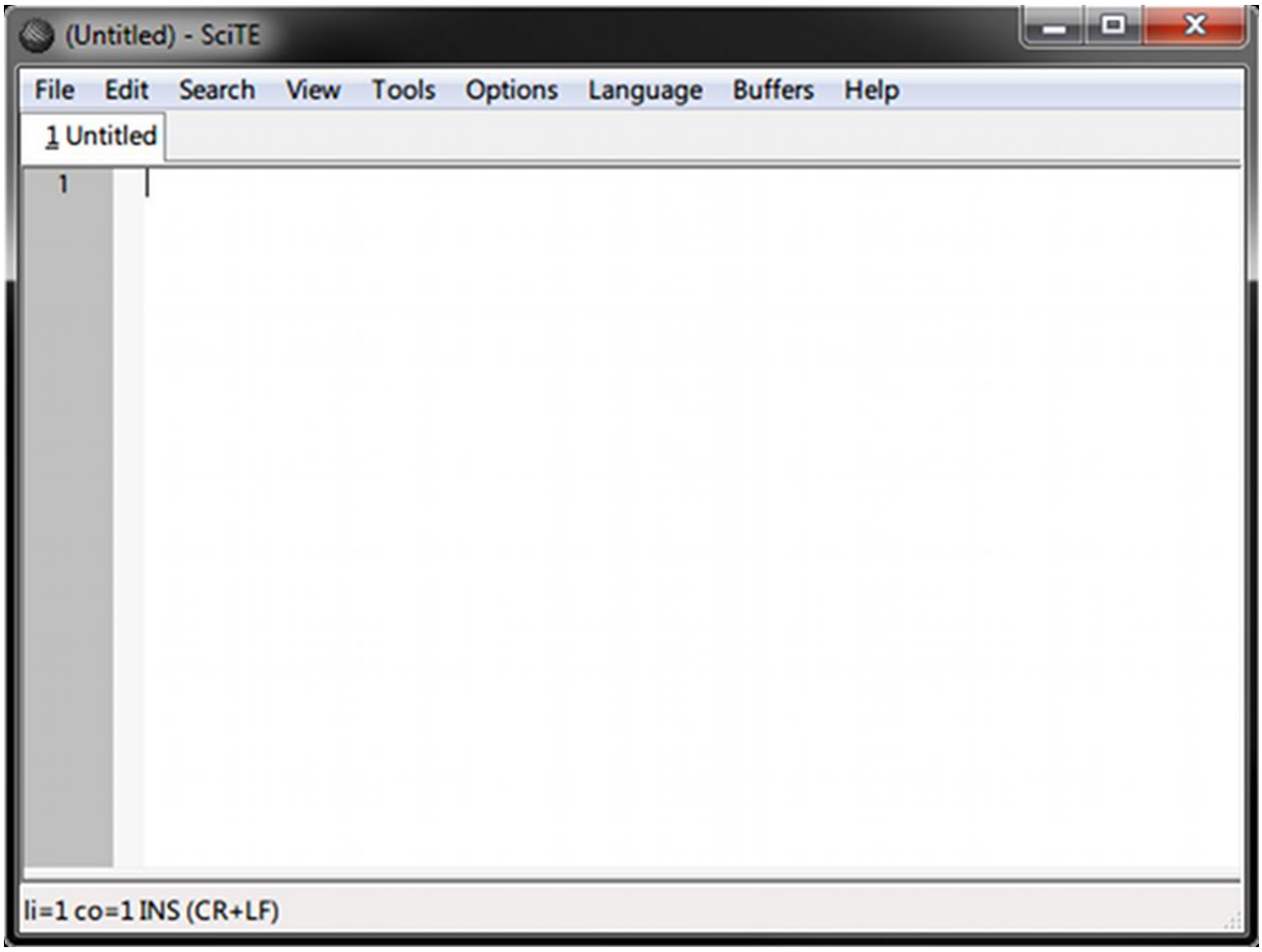
1.3 Κειμενογράφοι

Το κύριο εργαλείο που χρησιμοποιούν οι προγραμματιστές για να γράφουν λειτουργικό είναι ο κειμενογράφος. Οι κειμενογράφοι είναι παρόμοιοι των προγραμμάτων επεξεργασίας κειμένου (Microsoft Word, Open Office, …) αλλά σε αντίθεση με τέτοια προγράμματα δεν κάνουν οποιαδήποτε μορφοποίηση,(¨οχι έντονα, πλάγια, κτλ. γράμματα) αντί αυτού λειτουργούν μόνο σε απλό κείμενο.
Και τα δυο, OSX και Windows έρχονται με κειμενογράφους αλλά είναι αυστηρά περιορισμένοι και προτείνω να εγκαταστήσετε ένα καλύτερο.

Για να κάνετε την εγκατάσταση αυτού του προγράμματος ευκολότερη ένα πρόγραμμα εγκατάστασης είναι διαθέσιμο στον ιστοχώρο του βιβλίου: http://www.golang-book.com/. Αυτό το πρόγραμμα εγκατάστασης θα εγκαταστήσει τη σουίτα εργαλείων της Go, περιβαλλοντικές μεταβλητές και ένα κειμενογράφο.

Windows

Για τα Windows το πρόγραμμα εγκατάστασης θα εγκαταστήσει τον κειμενογράφο Scite.
Μπορείτε να τον ανοίξετε πηγαίνοντας: Έναρξη → Προγράμματα → Go → Scite (Start → All Programs → Go → Scite).Πρέπει να δείτε κάτι σαν αυτό:



Ο κειμενογράφος έχει μια μεγάλη λευκή περιοχή όπου μπορεί να εισαχθεί κείμενο. Στα αριστερά αυτής της περιοχής μπορείτε να δείτε τους αριθμούς γραμμής. Στο κάτω μέρος του παραθύρου υπάρχει μια μπάρα κατάστασης που εμφανίζει πληροφορίες σχετικά με το αρχείο και την τρέχουσα τοποθεσία σε αυτό (τώρα δείχνει ότι είμαστε στη γραμμή 1, στη στήλη 1, το κείμενο έχει εισαχθεί κανονικά, και ότι χρησιμοποιούμε windows-style newlines).

Μπορείτε να ανοίξετε αρχεία πηγαίνοντας: Αρχείο → Άνοιγμα (File → Open) και να περιηγηθείτε στο επιθυμητό αρχείο. Τα αρχεία μπορούν να αποθηκευτούν πηγαίνοντας: Αρχείο→Αποθήκευση ή Αρχείο→Αποθήκευση ως (File → Save or File → Save As.

Καθώς εργάζεστε σε ένα κειμενογράφο είναι χρήσιμο να μάθετε συντομεύσεις πληκτρολογίου.
Τα μενού εμφανίζουν τις συντομεύσεις στα δεξιά.
Εδώ είναι μερικές από τις πιο συνηθισμένες:

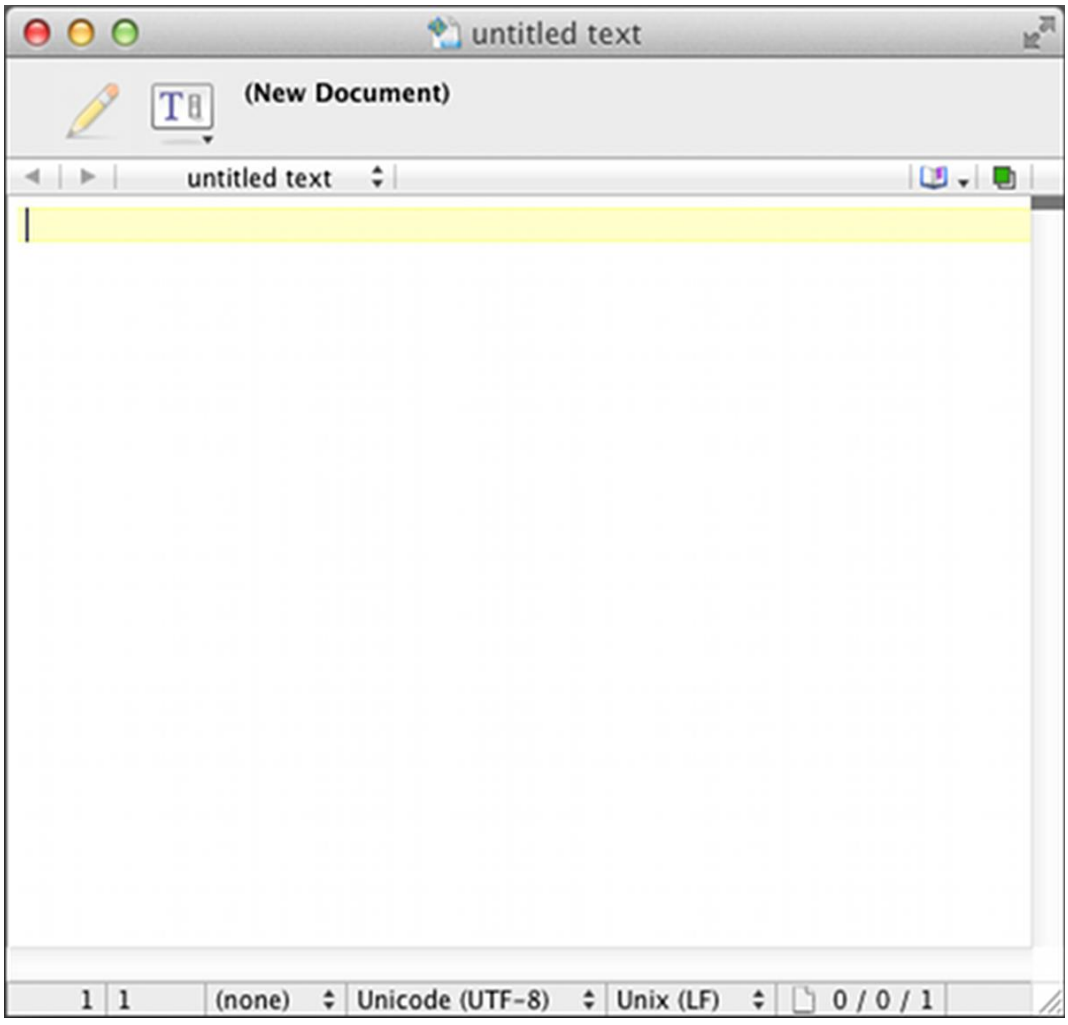
• Ctrl + S – σώζει το τρέχων αρχείο
• Ctrl + X – κόβει το επιλεγμένο κείμενο (το αποκόπτει και το τοποθετεί στα

πρόχειρα για να μπορείτε να το επικολλήσετε αργότερα)
• Ctrl + C – αντιγράφει το επιλεγμένο κείμενο
• Ctrl + V – επικολά το κείμενο στα πρόχειρα

• Χρησιμοποιήστε τα πλήκτρα με τα βέλη για να περιηγηθείτε.
Home για να πάτε στην αρχή της γραμμής και End για να πάτε στο τέλος της γραμμής.
• Κρατήστε πατημένο το Shift όσο χρησιμοποιείτε τα πλήκτρα με τα βέλη (ή Home και End) για να επιλέξετε κείμενο χωρίς τη χρήση του ποντικιού
• Ctrl + F – φέρνει μια γραμμή εύρεσης που μπορείτε να χρησιμοποιήσετε για να ψάξετε τα περιεχόμενα ενός αρχείου.

OSX

Στα OSX το πρόγραμμα εγκατάστασης εγκαθιστά τον κειμενογράφο "Text Wrangler":



Όπως ο κειμενογράφος Scite στα Windows, ο Text Wrangler περιέχει μια μεγάλη

κενή περιοχή όπου εισάγετε το κείμενο. Τα αρχεία μπορούν να ανοιχτούν πηγαίνοντας: File → Open. Τα αρχεία μπορούν να αποθηκευτούν πηγαίνοντας: File → Save ή File → Save As. Εδώ είναι μερικές χρήσιμες συντομεύσεις πληκτρολογίου:(Command είναι το πλήκτρο ⌘)

• Command + S – σώζει το τρέχων αρχείο
• Command + X – κόβει το επιλεγμένο κείμενο (το αποκόπτει και το τοποθετεί στα πρόχειρα για να μπορείτε να το επικολλήσετε αργότερα)
• Command + C – αντιγράφει το επιλεγμένο κείμενο
• Command + V – επικολλά το κείμενο στα πρόχειρα
• Χρησιμοποιήστε τα πλήκτρα με τα βέλη για να περιηγηθείτε
• Command + F – φέρνει μια γραμμή εύρεσης που μπορείτε να χρησιμοποιήσετε για να ψάξετε τα περιεχόμενα ενός αρχείου.

1.4 Εργαλεία της Go

Η Go είναι μια μεταγλωττισμένη γλώσσα προγραμματισμού , που σημαίνει ότι ο πηγιαίος κώδικας (ο κώδικας που γράφετε) είναι μεταφρασμένος σε μια γλώσσα που ο Η/Υ μπορεί να καταλάβει. Επομένως, πριν αρχίσουμε να γράφουμε ένα πρόγραμμα Go, χρειαζόμαστε τον μεταγλωττιστή.

Το πρόγραμμα εγκατάστασης θα εγκαταστήσει αυτόματα την Go. Θα χρησιμοποιήσουμε την version 1 της γλώσσας. (Περισσότερες πληροφορίες μπορούν να βρεθούν στο: http://www.golang.org)

Πάμε να σιγουρευτούμε ότι όλα δουλεύουν. Ανοίξτε ένα τερματικό και πληκτρολογήστε τα ακόλουθα:

```
go version
```

Πρέπει να δείτε το παρακάτω:

```
go version go1.0.2
```

Ο αριθμός έκδοσης μπορεί να είναι διαφορετικός. Αν εμφανιστεί λάθος ότι η εντολή δεν αναγνωρίζετε, δοκιμάστε να επανεκκινήσετε τον υπολογιστή σας,

Η σουίτα εργαλείων της Go αποτελείτε από μερικές διαφορετικές εντολές και υπο-εντολές . Μια λίστα με αυτές τις εντολές είναι διαθέσιμη πληκτρολογώντας:

```
go help
```

Θα δούμε πως χρησιμοποιούνται στα επόμενα κεφάλαια.

2. Το πρώτο σας Πρόγραμμα

Παραδοσιακά το πρώτο πρόγραμμα που γράφεται σε οποιαδήποτε γλώσσα προγραμματισμού ονομάζεται «Hello World» - Ένα πρόγραμμα που εξάγει απλά Hello World στο τερματικό σας. Ας γράψουμε ένα χρησιμοποιώντας τη Go!

Πρώτα δημιουργούμε ένα νέο φάκελο όπου μπορούμε να αποθηκεύσουμε το πρόγραμμα μας. Στο πρόγραμμα εγκατάστασης που χρησιμοποιείται στο κεφάλαιο 1 δημιουργεί ένα φάκελο στο κατάλογο Home με το όνομα Go. Δημιουργήστε το φάκελο με το όνομα ~ / Go/src/golang-book/chapter2. (Όπου ~ σημαίνει ότι είναι στον κατάλογο Home). Από το τερματικό σας μπορείτε να το κάνετε πληκτρολογώντας τις ακόλουθες εντολές:

```
mkdir Go/src/golang-book
mkdir Go/src/golang-book/chapter2
```

Χρησιμοποιώντας τον text editor γράψτε τα ακόλουθα:

```
package main

import "fmt"

// this is a comment

func main() {
    fmt.Println("Hello World")
}
```

Βεβαιωθείτε ότι το αρχείο σας είναι ίδιο με αυτό που παρουσιάζεται εδώ και αποθηκεύστε το ως main.go στο φάκελο που μόλις δημιουργήσαμε.
Ανοίξτε ένα νέο τερματικό και πληκτρολογήστε το εξής:

```
cd Go/src/golang-book/chapter2
go run main.go
```

Θα πρέπει να δείτε Hello World να εμφανίζεται στο τερματικό σας. Η εντολή go run

παίρνει τα επόμενα αρχεία (χωρισμένα με κενά), τα μεταγλωττίζει σε εκτελέσιμα αρχεία αποθηκευμένα σε ένα προσωρινό κατάλογο και στη συνέχεια τρέχει το πρόγραμμα. Αν δεν δείτε να εμφανιστεί το Hello World μπορεί να έχετε κάνει κάποιο λάθος κατά την πληκτρολόγηση του προγράμματος. Ο μεταγλωττιστής της Go θα σας δώσει συμβουλές σχετικά με το πού βρίσκεται το λάθος. Όπως και οι περισσότεροι μεταγλωττιστές, ο μεταγλωττιστής της Go είναι εξαιρετικά σχολαστικός και δεν έχει καμία ανοχή στα λάθη.

2.1 Πώς να διαβάσετε ένα πρόγραμμα Go

Ας δούμε το πρόγραμμα πιο λεπτομερώς. Τα προγράμματα Go διαβάζονται από πάνω προς τα κάτω, αριστερά προς τα δεξιά (όπως ένα βιβλίο). Η πρώτη γραμμή είναι :

```
package main
```

Αυτό είναι γνωστό ως «δήλωση πακέτο». Κάθε Go πρόγραμμα πρέπει να ξεκινάει με μια δήλωση πακέτο. Τα πακέτα είναι ο τρόπος οργάνωσης και επαναχρησιμοποίησης κώδικα της Go. Υπάρχουν δύο τύποι προγραμμάτων Go: τα εκτελέσιμα και οι βιβλιοθήκες.
Εκτελέσιμες εφαρμογές είναι τα είδη των προγραμμάτων που μπορούμε να τρέξουμε άμεσα από το τερματικό (στα Windows τελειώνουν με .exe). Οι βιβλιοθήκες είναι συλλογές από κώδικα τις οποίες πακετάρουμε μαζί, έτσι ώστε να μπορούμε να τις χρησιμοποιήσουμε σε άλλα προγράμματα. Θα μελετήσουμε τις βιβλιοθήκες λεπτομερώς αργότερα, προς το παρόν απλά φροντίστε να συμπεριλαμβάνετε αυτή τη γραμμή σε κάθε πρόγραμμα που γράφετε.

Η επόμενη γραμμή είναι μια κενή γραμμή. Οι Η/Υ συμβολίζουν τις νέες γραμμές με ειδικούς χαρακτήρες (ή πολλαπλούς χαρακτήρες). Οι νέες γραμμές, τα κενά και οι καρτέλες είναι γνωστά ως κενοί χαρακτήρες (επειδή δεν μπορείτε να τους δείτε). Η Go κυρίως δεν υπολογίζει τους κενούς χαρακτήρες, τους χρησιμοποιούμε για να κάνουμε τα προγράμματα πιο εύκολα να διαβαστούν. (Μπορείτε να αφαιρέσετε αυτή τη γραμμή και το πρόγραμμα θα συμπεριφέρεται με τον ίδιο ακριβώς τρόπο)

Μετά βλέπουμε:

```
import "fmt"
```

Η λέξη-κλειδί import είναι για να περιλαμβάνουμε κώδικα από άλλα πακέτα ώστε να τα χρησιμοποιήσουμε στο πρόγραμμα μας. Το fmt πακέτο (συντομογραφία του

format) εφαρμόζει μορφοποίηση για είσοδο και έξοδο. Λαμβάνοντας υπόψη τα όσα μόλις μάθαμε για τα πακέτα τι νομίζετε ότι τα αρχεία του fmt πακέτου θα περιέχουν;

Παρατηρήστε ότι το fmt παραπάνω περιβάλλεται από διπλά εισαγωγικά. Η χρήση των διπλών εισαγωγικών όπως παραπάνω που είναι γνωστό ως «αλφαριθμιτική ακολουθία» (string literal) είναι ένα είδος «έκφρασης». Στην Go τα strings αντιπροσωπεύουν μια ακολουθία χαρακτήρων (γραμμάτων, αριθμών,συμβόλων, κλπ) καθορισμένου μήκους. Τα strings περιγράφονται με περισσότερες λεπτομέρειες στο επόμενο κεφάλαιο, αλλά προς το παρόν το σημαντικό πράγμα που πρέπει να θυμάστε είναι ότι ένα άνοιγμα με τον χαρακτήρα " πρέπει να κλείνει από έναν άλλο χαρακτήρα " ακόμα και τίποτα μεταξύ των δύο να μην περιλαμβάνεται. (Ο χαρακτήρας " δεν είναι μέρος του string)

Η γραμμή που ξεκινά με / / είναι γνωστή ως σχόλιο.
Τα σχόλια αγνοούνται από τον compiler της Go και είναι για την δική σας διευκόλυνση (ή οποιοδήποτε άλλον που χρησιμοποιεί το πρόγραμμά σας). Η GO υποστηρίζει δύο διαφορετικούς τύπους σχολίων: ο τύπος / / στον οποίο όλο το κείμενο μεταξύ των / / και του τέλους της γραμμής είναι μέρος του σχολίου και ο τύπος / ** / όπου τα πάντα μεταξύ των
* είναι μέρος του σχολίου. (Και μπορεί να περιλαμβάνει πολλές γραμμές)

Μετά από αυτό βλέπετε μια δήλωση συνάρτησης:

```
func main() {
    fmt.Println("Hello World")
}
```

Οι συναρτήσεις (functions) είναι τα δομικά στοιχεία ενός προγράμματος Go. Έχουν εισόδους, εξόδους και μια σειρά από βήματα που ονομάζονται καταστάσεις οι οποίες εκτελούνται σε σειρά. Όλες οι συναρτήσεις ξεκινούν με την λέξη-κλειδί func ακολουθούμενη από το όνομα της συνάρτησης (main σε αυτή την περίπτωση), μια λίστα από μηδέν ή περισσότερες «παραμέτρους» που περιβάλλονται από παρενθέσεις, ένα προαιρετικό τύπο επιστροφής και ένα «σώμα» το οποίο περικλείεται από αγκύλες. Αυτή η συνάρτηση δεν έχει παραμέτρους, δεν επιστρέφει τίποτα και έχει μόνο μία δήλωση. Το όνομα main είναι ξεχωριστό γιατί είναι η συνάρτηση που καλείται όταν εκτελείτε το πρόγραμμα.

Το τελικό κομμάτι του προγράμματός μας είναι αυτή η γραμμή:

```
    fmt.Println("Hello World")
```

Αυτή η εντολή αποτελείτε από τρεις συνιστώσες. Πρώτα έχουμε πρόσβαση σε μια άλλη συνάρτηση του πακέτου fmt που ονομάζεται println (αυτή είναι fmt.Println, το κομμάτι Println
σημαίνει Print Line (Εκτύπωση)). Στη συνέχεια, δημιουργούμε ένα καινούργιο string που περιέχει το Hello World και επικαλείται (επίσης γνωστή ως κλήση ή εκτέλεση) τη συνάρτηση με το string.

Σε αυτό το σημείο έχουμε ήδη δει αρκετή νέα ορολογία και ίσως να είσαι λίγο συγκλονισμένος. Μερικές φορές είναι χρήσιμο να διαβάσετε σκόπιμα το πρόγραμμά σας δυνατά. Μια ανάγνωση του προγράμματος που μόλις γράψαμε μπορεί να είναι κάπως έτσι:

Δημιουργήστε ένα νέο εκτελέσιμο πρόγραμμα, το οποίο να παραπέμπει στη fmt βιβλιοθήκη και περιλαμβάνει μια συνάρτηση που ονομάζεται main. Αυτή η συνάρτηση δεν παίρνει ορίσματα, δεν επιστρέφει τίποτα και κάνει τα εξής: Έχει πρόσβαση στη συνάρτηση println που περιέχεται στο εσωτερικό του fmt πακέτου και επικαλείται ένα όρισμα – το string Hello World.

Η συνάρτηση Println κάνει τη πραγματική δουλειά στο πρόγραμμα. Μπορείτε να μάθετε περισσότερα γι 'αυτήν, πληκτρολογώντας το ακόλουθο στο τερματικό σας:

```
godoc fmt Println
```

Μεταξύ άλλων θα πρέπει να δείτε αυτό:

```
Println formats using the default formats for
its operands and writes to standard output.
Spaces are always added between operands and a
newline is appended. It returns the number of
bytes written and any write error encountered.
```

Η Go είναι μια γλώσσα προγραμματισμού με πολύ καλή τεκμηρίωση αλλά μπορεί να είναι δύσκολη να κατανοηθεί εκτός και αν είστε ήδη εξοικειωμένοι με γλώσσες προγραμματισμού. Παρ 'όλα αυτά η εντολή godoc είναι εξαιρετικά χρήσιμη και είναι μια καλή αρχή για να λύνεται τις τυχόν απορίες σας.

Επιστρέφοντας στη συνάρτηση , η τεκμηρίωση σας λέει ότι η συνάρτηση println θα στείλει οτιδήποτε της δώσετε στην standard έξοδο - ένα όνομα για την έξοδο του

τερματικού που εργάζεστε.Η συνάρτηση αυτή εμφανίζει το Hello World στο τερματικό.

Στο επόμενο κεφάλαιο θα εξετάσουμε πώς η Go αποθηκεύει και απεικονίζει μηνύματα σαν το Hello World μαθαίνοντας για τους τύπους δεδομένων.

Προβλήματα:

1. Τι είναι οι κενοί χαρακτήρες;

2. Τι είναι το σχόλιο; Ποιοί είναι οι δύο τρόποι που γράφουμε ένα σχόλιο;

3. Το πρόγραμμά μας ξεκίνησε με το package main. Με τι θα αρχίσουν τα αρχεία στο πακέτο fmt;

4. Χρησιμοποιήσαμε τη συνάρτηση println που ορίζεται στο fmt πακέτο. Αν θέλαμε να χρησιμοποιήσουμε τη συνάρτηση Exit από το os πακέτο τι θα πρέπει να κάνουμε;

5. Τροποποιήστε το πρόγραμμα που γράψαμε έτσι ώστε αντί της εκτύπωσης Hello World να εκτυπώσει Hello, my name is ακολουθούμενο από το όνομα σας.

3. Τύποι Δεδομένων

Στο προηγούμενο κεφάλαιο χρησιμοποιήσαμε τον τύπο δεδομένων string για την αποθήκευση του Hello World. Οι τύποι δεδομένων κατηγοριοποιούν ένα σύνολο σχετικών τιμών, περιγράφουν τις εργασίες που μπορούν να γίνουν σε αυτά και καθορίζουν τον τρόπο που αποθηκεύονται. Επειδή οι τύποι δεδομένων μπορεί να είναι μια δύσκολη έννοια για να τους καταλάβουμε, θα τους δούμε από διαφορετικές οπτικές γωνίες προτού δούμε πώς εφαρμόζονται στη Go.

Οι φιλόσοφοι κάνουν μερικές φορές μια διάκριση μεταξύ στους τύπους και στα λεγόμενα. Για παράδειγμα, ας υποθέσουμε ότι έχετε ένα σκυλί με το όνομα Max. Max είναι το σύμβολο (ένα συγκεκριμένο παράδειγμα ή μέλος) και ο σκύλος είναι ο τύπος (η γενική έννοια).
Η λέξη «σκυλί» περιγράφει μια σειρά από ιδιότητες που όλα τα σκυλιά έχουν από κοινού. Αν και θα μπορούσαμε να το υπεραπλουστεύσουμε κάπως έτσι: Όλα τα σκυλιά έχουν 4 πόδια, ο Max είναι ένα σκύλος, ως εκ τούτου ο Max έχει 4 πόδια. Οι τύποι στις γλώσσες προγραμματισμού λειτουργούν με παρόμοιο τρόπο: Όλα τα strings έχουν ένα μήκος, x είναι ένα string, επομένως το x έχει ένα μήκος.

Στα μαθηματικά μιλάμε συχνά για σύνολα. Για παράδειγμα:
$\mathbb{R}$ (το σύνολο όλων των πραγματικών αριθμών) ή $\mathbb{N}$ (το σύνολο όλων των φυσικών αριθμών). Κάθε μέλος από αυτά τα σύνολα έχουν τις ίδιες ιδιότητες με όλα τα άλλα μέλη του συνόλου. Για παράδειγμα, όλοι οι φυσικοί αριθμοί είναι προσεταιριστικοί: «για όλους τους φυσικούς αριθμούς a, b, και c, a + (b + c) = (a + b) + c και a × (b × c) = (a × b) × c ». Με αυτό τον τρόπο τα σύνολα είναι παρόμοια με τους τύπους στις γλώσσες προγραμματισμού από τι στιγμή που όλες οι τιμές ενός συγκεκριμένου τύπου έχουν τις ίδιες ιδιότητες.

Η Go είναι μια στατικού τύπου γλώσσα προγραμματισμού. Αυτό σημαίνει ότι οι μεταβλητές έχουν πάντα ένα συγκεκριμένο τύπο και αυτός ο τύπος δεν μπορεί να αλλάξει. Αρχικά οι στατικοί τύποι μπορεί να φαίνονται περίπλοκοι. Θα αφιερώσετε αρκετό από το χρόνο σας προσπαθώντας να διορθώσετε το πρόγραμμά σας έτσι ώστε τελικά να μεταγλωττιστεί. Αλλά οι τύποι θα σας βοηθήσουν να καταλάβετε τι κάνει το πρόγραμμα σας και να δείτε μια μεγάλη ποικιλία από κοινά λάθη.

Η Go έρχεται με διάφορους ενσωματωμένους τύπους δεδομένων τους οποίους θα εξετάσουμε τώρα πιο αναλυτικά.

3.1 Αριθμοί

Η Go έχει πολλούς διαφορετικούς τύπους για να αντιπροσωπεύσει τους αριθμούς.

Γενικά χωρίζουμε τους αριθμούς σε δύο διαφορετικά είδη: ακέραιους αριθμούς και αριθμούς κινητής υποδιαστολής.

Ακέραιοι

Ακέραιοι - όπως οι αντίστοιχοί τους στα μαθηματικά - είναι αριθμοί χωρίς δεκαδικό στοιχείο. (..., -3, -2, -1, 0, 1, ...) Σε αντίθεση με το δεκαδικό σύστημα με βάση το 10 που χρησιμοποιούμε για να αντιπροσωπεύουμε τους αριθμούς, οι Η/Υ χρησιμοποιούν το δυαδικό σύστημα με βάση το 2.

Το σύστημά μας αποτελείται από 10 διαφορετικά ψηφία. Μόλις έχουμε εξαντλήσει τα διαθέσιμα ψηφία μας τα αντιπροσωπεύουμε με μεγαλύτερους αριθμούς με τη χρήση 2 (ή 3, 4, 5, ...) ψηφίων που τίθενται το ένα δίπλα στο άλλο. Για παράδειγμα, ο αριθμός μετά το 9 είναι το 10, ο αριθμός μετά το 99 είναι το 100 και ούτω καθεξής. Οι υπολογιστές κάνουν το ίδιο, αλλά έχουν μόνο 2 ψηφία αντί για 10. Δηλαδή μετράτε κάπως έτσι: 0, 1, 10, 11, 100, 101, 110, 111 και ούτω καθεξής. Η άλλη διαφορά μεταξύ του αριθμητικού συστήματος που χρησιμοποιούμε και του αντίστοιχου των Η/Υ είναι ότι όλοι οι ακέραιοι έχουν ένα καθορισμένο μέγεθος . Έχουν περιθώριο για έναν ορισμένο αριθμό ψηφίων. Έτσι, ένας 4 bit ακέραιος μπορεί να μοιάζει κάπως έτσι: 0000, 0001, 0010, 0011, 0100. Τελικά θα εξαντληθεί ο χώρος και οι περισσότεροι Η/Υ απλά ξεκινούν από την αρχή. (Που μπορεί να οδηγήσει σε κάποια πολύ περίεργη συμπεριφορά)

Οι τύποι ακέραιων της Go είναι: uint8, uint16, uint32, uint64, int8, int16, int32 και int64. Οι αριθμοί 8, 16, 32 και 64 μας λένε πόσα bits χρησιμοποιεί ο καθένας από τους τύπους. Το uint σημαίνει “unsigned integer” (ακέραιος χωρίς πρόσημο), ενώ το int σημαίνει “signed integer”(ακέραιος με πρόσημο). Οι ακέραιοι χωρίς πρόσημο περιέχουν μόνο θετικούς αριθμούς (ή
μηδέν). Επιπλέον υπάρχουν δύο γνωστοί τύποι: το byte που είναι το ίδιο όπως το uint8 και το rune που είναι το ίδιο όπως το Int32. Τα Bytes είναι μια εξαιρετικά κοινή μονάδα μέτρησης που χρησιμοποιείται στους Η/Υ (1 byte = 8 bits, 1024 bytes = 1 kilobyte, 1024 kilobytes = 1 megabyte, ...) αντιθέτως ο τύπος δεδομένων byte της Go συνήθως χρησιμοποιείται για τον ορισμό άλλων τύπων. Υπάρχουν επίσης 3 τύποι ακεραίου machine-dependent: uint, int και uintptr. Είναι machine-dependent επειδή το μέγεθός τους εξαρτάται από το είδος της αρχιτεκτονικής που χρησιμοποιείτε.

Γενικά, εάν εργάζεστε με ακέραιους αριθμούς θα πρέπει να χρησιμοποιήσετε μόνο τον int τύπο.

Αριθμοί κινητής υποδιαστολής

Αριθμοί κινητής υποδιαστολής είναι αριθμοί που περιέχουν ένα δεκαδικό στοιχείο (πραγματικοί αριθμοί). (1.234, 123.4, 0.00001234, 12340000) Η πραγματική αντιπροσώπευσή τους σε έναν Η/Υ είναι αρκετά περίπλοκη και δεν είναι απαραίτητο να γνωρίζουν πώς να τους χρησιμοποιούν. Οπότε για τώρα χρειαζόμαστε μόνο να έχουμε κατά νου τα ακόλουθα:

1. Οι αριθμοί κινητής υποδιαστολής είναι ανακριβής. Περιστασιακά, δεν είναι δυνατόν να αντιπροσωπεύουν έναν αριθμό. Για παράδειγμα ο υπολογισμός του 1.01 – 0.99 έχει αποτέλεσμα 0,.020000000000000018 - ένας αριθμός πολύ κοντά σε αυτό που θα περίμενε κανείς, αλλά δεν είναι ακριβώς το ίδιο.

2. Όπως οι ακέραιοι έτσι και οι αριθμοί κινητής υποδιαστολής έχουν ένα ορισμένο μέγεθος (32 bit ή 64 bit). Χρησιμοποιώντας ένα μεγαλύτερου μεγέθους αριθμό κινητής υποδιαστολής αυξάνεται η ακρίβεια του. (πόσα ψηφία μπορεί να αναπαριστά)

3. Εκτός από τους αριθμούς υπάρχουν αρκετές άλλες τιμές που μπορούν να αναπαρασταθούν: «μη αριθμός» (NaN, για πράγματα όπως 0/0) και θετικό και αρνητικό άπειρο. (+ ∞ και - ∞)

Η Go έχει δύο τύπους κινητής υποδιαστολής: float32 και float64 (επίσης συχνά αναφέροντε ως μονής ακρίβειας και διπλής ακρίβειας αντίστοιχα), καθώς και δύο επιπλέον τύπους για την αναπαράσταση μιγαδικών αριθμών (αριθμοί με φανταστικό μέρος): complex64 και complex128. Σε γενικές γραμμές θα πρέπει να χρησιμοποιείτε το float64 όταν εργάζεστε με αριθμούς κινητής υποδιαστολής.

Παράδειγμα

Ας γράψουμε ένα πρόγραμμα με τη χρήση αριθμών. Πρώτα δημιουργήστε ένα φάκελο που ονομάζεται chapter3 και δημιουργήστε ένα αρχείο main.go που θα περιέχει τα ακόλουθα:

```
package main

import "fmt"

func main() {
    fmt.Println("1 + 1 =", 1 + 1)
}
```

Εάν εκτελέσετε το πρόγραμμα και θα πρέπει να δείτε αυτό:

```
$ go run main.go
1 + 1 = 2
```

Παρατηρήστε ότι αυτό το πρόγραμμα είναι πολύ παρόμοιο με το πρόγραμμα που γράψαμε στο κεφάλαιο 2. Περιέχει την ίδια γραμμή πακέτου, την ίδια γραμμή εισαγωγής, η ίδια δήλωση συνάρτησης και χρησιμοποιεί την ίδια συνάρτηση Println. Αυτή τη φορά αντί να εκτυπώσει το string Hello World έχουμε την εκτύπωση του string 1 + 1 = ακολουθούμενο από το αποτέλεσμα της έκφρασης 1 + 1. Η έκφραση αυτή αποτελείται από τρία μέρη: το αριθμητικό 1 (το οποίο είναι του τύπου int), τον τελεστή + (το οποίο αντιπροσωπεύει τη πρόσθεση) και άλλο ένα αριθμητικό 1. Ας προσπαθήσουμε το ίδιο πράγμα χρησιμοποιώντας αριθμούς κινητής υποδιαστολής:

```
    fmt.Println("1 + 1 =", 1.0 + 1.0)
```

Παρατηρήστε ότι χρησιμοποιούμε το .0 για να πούμε στην Go ότι αυτός είναι ένας αριθμός κινητής υποδιαστολής αντί του ακεραίου. Τρέχοντας το πρόγραμμα αυτό θα σας δώσει το ίδιο αποτέλεσμα όπως και πριν.

Εκτός από τον τελεστή της πρόσθεσης η Go έχει πολλούς άλλους τελεστές:

+	πρόσθεση
-	αφαίρεση
*	πολλαπλασιασμό
/	διαίρεση
%	ακέραιο υπόλοιπο

3.2 Αλφαριθμητικές ακολουθίες (Strings)

Όπως είδαμε στο κεφάλαιο 2 ένα string είναι μια ακολουθία χαρακτήρων με ένα συγκεκριμένο μήκος που χρησιμοποιούνται για να αντιπροσωπεύσουν ένα κείμενο. Τα strings της Go αποτελούνται από μεμονωμένα bytes, συνήθως ένα για κάθε χαρακτήρα. (Οι χαρακτήρες από άλλες γλώσσες όπως τα Κινέζικα εκπροσωπούνται από περισσότερα του ενός byte).

Τα Strings μπορούν να δημιουργηθούν χρησιμοποιώντας διπλά εισαγωγικά "Hello

World" ή μονά εισαγωγικά `Hello World`. Η διαφορά μεταξύ αυτών είναι ότι τα διπλά εισαγωγικά δεν μπορούν να περιέχουν νέες γραμμές και επιτρέπουν μια ειδική ακολουθία διαφυγής. Για παράδειγμα το \ n θα αντικατασταθεί με μια νέα γραμμή και το \ t θα αντικατασταθεί με μια οριζόντια στηλοθέτηση.

Πολλές κοινές εντολές στα strings περιλαμβάνουν: τρόπους εύρεσης του μήκους του string len("Hello World"), την πρόσβαση σε ένα μεμονωμένο χαρακτήρα του string: "Hello
World"[1], και την συνένωση δύο ξεχωριστών strings μαζί: "Hello " + "World". Ας τροποποιήσουμε το πρόγραμμα που δημιουργήσαμε νωρίτερα και να το δοκιμάσουμε:

```
package main

import "fmt"

func main() {
    fmt.Println(len("Hello World"))
    fmt.Println("Hello World"[1])
    fmt.Println("Hello " + "World")
}
```

Μερικά πράγματα που πρέπει να προσέξετε:

1. Ένα κενό διάστημα θεωρείται επίσης ένας χαρακτήρας, έτσι το μήκος του string είναι 11 και όχι 10 και η 3η γραμμή έχει "Hello " αντί του "Hello".

2. Η διευθυνσιοδότηση των strings ξεκινάει από το 0 και όχι από το 1. [1] σας δίνει το 2ο στοιχείο όχι το 1ο. Επίσης, βλέπετε 101 αντί του e όταν εκτελείτε αυτό το πρόγραμμα. Αυτό συμβαίνει επειδή ο χαρακτήρας αντιπροσωπεύεται από ένα byte (θυμηθείτε ένα byte είναι ένας ακέραιος αριθμός).

Ένας τρόπος να καταλάβετε την διευθυνσιοδότηση θα ήταν να δείτε αυτό εδώ: "Hello World" [1]. Θα το διαβάζατε σαν "το string Hello World υπο το 1," "το string Hello World στην 1" ή "Ο δεύτερος χαρακτήρας του string Hello World".

3. Η συνένωση χρησιμοποιεί το ίδιο σύμβολο όπως η πρόσθεση. Ο μεταγλωττιστής της Go καταλαβαίνει τι να κάνει βασιζόμενος στους τύπους των ορισμάτων. Δεδομένου ότι και στις δύο πλευρές του + είναι strings ο μεταγλωττιστής

αναλαμβάνει να κάνει συνένωση και όχι πρόσθεση. (Η πρόσθεση δεν έχει νόημα για τα strings)

3.3 Τύποι δεδομένων αληθείας (Booleans)

Η τιμή boolean (ονομάστηκε από τον George Boole) είναι ένας ειδικός τύπος ακέραιου του 1 bit και χρησιμοποιείται για την αναπαράσταση των εκφράσεων αληθές (True) και ψευδές(False). Οι τρεις λογικοί τελεστές που χρησιμοποιούνται στην boolean είναι:

&&	and
||	or
!	not

Εδώ είναι ένα παράδειγμα προγράμματος που δείχνει το πώς μπορούν να χρησιμοποιηθούν:

```
func main() {
    fmt.Println(true && true)
    fmt.Println(true && false)
    fmt.Println(true || true)
    fmt.Println(true || false)
    fmt.Println(!true)
}
```

Η εκτέλεση αυτού του προγράμματος θα πρέπει να σας δώσει:

```
$ go run main.go
true
false
true
true
false
```

Συνήθως χρησιμοποιούμε πίνακες αληθείας για να καθορίσουμε πώς λειτουργούν αυτοί οι τελεστές:

Expression	Value
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Expression	Value
`true || true`	`true`
`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Expression	Value
`!true`	`false`
`!false`	`true`

Αυτοί είναι οι απλούστεροι τύποι που υπάρχουν στην γλώσσα Go και αποτελούν τη βάση από την οποία θα φτιαχτούν και όλοι οι άλλοι τύποι αργότερα.

Προβλήματα:

1. Πώς οι ακέραιοι αποθηκεύονται σε έναν Η/Υ;

2. Γνωρίζουμε ότι στο δεκαδικό σύστημα ο μεγαλύτερος μονοψήφιος αριθμός είναι το 9 και ο μεγαλύτερος διψήφιος αριθμός είναι το 99. Δεδομένου ότι στο δυαδικό ο μεγαλύτερος διψήφιος αριθμός είναι το 11 (3), ο μεγαλύτερος τριψήφιος αριθμός είναι το 111 (7) και ο μεγαλύτερος τετραψήφιος αριθμός είναι το 1111 (15) ποιος είναι ο μεγαλύτερος οκταψήφιος αριθμός;
(hint: $10^1-1 = 9$ and $10^2-1 = 99$)

3. Γράψτε ένα πρόγραμμα που να υπολογίζει 321325 × 424521 και να εμφανίζει το αποτέλεσμα στο τερματικό. (Χρησιμοποιήστε τον τελεστή * για τον πολλαπλασιασμό)

4. Τι είναι το string; Πώς μπορείτε να βρείτε το μήκος του;

5. Ποια είναι η τιμή της έκφρασης (true && false) || (false && true) || !(false && false);

4 Μεταβλητές

Μέχρι τώρα είχαμε δει μόνο προγράμματα που χρησιμοποιούν διακριτές τιμές (αριθμοί, strings, κτλ.) αλλά τέτοια προγράμματα δεν είναι ιδιαίτερα χρήσιμα. Για να φτιάξουμε πραγματικά χρήσιμα προγράμματα πρέπει να μάθουμε δυο νέες έννοιες: τις μεταβλητές και τις εντολές ελέγχου ροής. Αυτό το κεφάλαιο εξερευνά τις μεταβλητές πιο λεπτομερώς.

Μεταβλητή είναι μια θέση αποθήκευσης συγκεκριμένου τύπου και κατάλληλου ονόματος. Ας αλλάξουμε το πρόγραμμα που γράψαμε στο κεφάλαιο 2 ώστε να χρησιμοποιεί μια μεταβλητή:

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

Παρατηρήστε ότι το string απ' το πρωτότυπο πρόγραμμα εξακολουθεί να εμφανίζετε σε αυτό το πρόγραμμα αλλά αντί να το στέλνει κατευθείαν στη συνάρτηση Println το αναθέτει σε μια μεταβλητή. Οι μεταβλητές στη Go δημιουργούνται χρησιμοποιώντας τη λέξη-κλειδί var, στη συνέχεια προσδιορίζοντας το όνομα της μεταβλητής (x),τον τύπο (string) και τελικά εκχωρώντας μια τιμή στη μεταβλητή (Hello World).
Το τελευταίο βήμα είναι προαιρετικό, έτσι ένας εναλλακτικός τρόπος γραφής του προγράμματος θα είναι αυτός:

```
package main

import "fmt"

func main() {
    var x string
    x = "Hello World"
    fmt.Println(x)
}
```

Οι μεταβλητές στη Go είναι όμοιες με τις μεταβλητές στην άλγεβρα αλλά έχουν κάποιες μικρές διαφορές:

Πρώτον, όταν βλέπουμε το σύμβολο = έχουμε την τάση να διαβάζουμε "το x ισούται με το string Hello World". Δεν υπάρχει λάθος διαβάζοντας το πρόγραμμά μας με αυτό το τρόπο, αλλά είναι καλύτερα να το διαβάζουμε ως : "το x παίρνει το string Hello World " ή "στο x εκχωρείτε το string Hello World".

Αυτή η διαφοροποίηση είναι σημαντική επειδή (όπως προτείνει το όνομα) οι μεταβλητές μπορούν να αλλάξουν την τιμή τους καθ 'όλη τη διάρκεια ζωής του προγράμματος. Προσπαθήστε να τρέξετε το παρακάτω:

```
package main

import "fmt"

func main() {
    var x string
    x = "first"
    fmt.Println(x)
    x = "second"
    fmt.Println(x)
}
```

Στην πραγματικότητα, μπορείτε ακόμη και να κάνετε αυτό :

```
var x string
x = "first "
fmt.Println(x)
x = x + "second"
fmt.Println(x)
```

Αυτό δε θα έβγαζε κανένα νόημα αν το διαβάζατε σαν ένα αλγεβρικό θεώρημα.
Αλλά θα είχε νόημα αν διαβάζατε το πρόγραμμα σαν μια σειρά από εντολές,
Όταν βλέπουμε `x = x + "second"` πρέπει να το διαβάζουμε ως "αναθέτω στη μεταβλητή x, τη συνένωση της τιμής της μεταβλητής x και του second" . Το δεξί μέρος του = εκτελείτε πρώτα και το αποτέλεσμα καταχωρείτε στο αριστερό.

Ο τύπος `x = x + y` είναι τόσο κοινός στον προγραμματισμό που η Go έχει μια ειδική εντολή εκχώρησης: +=. Θα μπορούσαμε να είχαμε γράψει το `x = x + "second"` ως `x += "second"` και θα είχαμε το ίδιο αποτέλεσμα.

Μια άλλη διαφορά μεταξύ της Go και της άλγεβρας είναι ότι χρησιμοποιούν διαφορετικό σύμβολο για την ισότητα: ==. (δύο σύμβολα ίσον το ένα δίπλα στο άλλο). Το == είναι ένα σύμβολο όπως το + και επιστρέφει τύπους δεδομένων αληθείας. Για παράδειγμα:

```
var x string = "hello"
var y string = "world"
fmt.Println(x == y)
```

Αυτό το πρόγραμμα θα τυπώσει false επειδή το hello δεν είναι ίδιο με το world.
Αντίθετα:

```
var x string = "hello"
var y string = "hello"
fmt.Println(x == y)
```

Αυτό θα τυπώσει true επειδή τα δυο strings είναι τα ίδια.

Δεδομένου ότι η δημιουργία μιας νέας μεταβλητής με μια αρχική τιμή είναι πολύ κοινή, η Go υποστηρίζει μια μικρότερη δήλωση :

```
x := "Hello World"
```

Προσέξτε το : πριν το = και ότι δεν διευκρινίζετε ο τύπος της.
Ο τύπος δεν είναι απαραίτητος επειδή ο μεταγλωττιστής της Go είναι σε θέση να συμπεράνει τον τύπο, με βάση την τιμή που εκχωρήσατε στη μεταβλητή. (Απ' τη στιγμή που εκχωρήσατε ένα string, το x δίνει τον τύπο του string). Ο μεταγλωττιστής μπορεί επίσης να συμπεράνει την κατάσταση var:

```
var x = "Hello World"
```

Το ίδιο δουλεύει και για άλλους τύπους:

```
x := 5
fmt.Println(x)
```

Σε γενικές γραμμές θα πρέπει να χρησιμοποιείτε τη μικρότερη αυτή μορφή όποτε είναι δυνατό.

4.1 Πως να ονομάσετε μια μεταβλητή.

Το να ονομάσετε μια μεταβλητή σωστά είναι ένα σημαντικό μέρος στην ανάπτυξη λογισμικού. Τα ονόματα πρέπει να ξεκινούν με γράμμα και ενδέχεται να περιέχουν γράμματα, αριθμούς ή το χαρακτήρα _. Ο μεταγλωττιστής της Go δε νοιάζεται για το όνομα που δώσατε, αυτό αφορά και είναι όφελος για εσάς (και τους άλλους). Διαλέξτε ονόματα που περιγράφουν ξεκάθαρα το σκοπό της μεταβλητής.
Υποθέτουμε ότι έχουμε την ακόλουθη:

```
x := "Max"
fmt.Println("My dog's name is", x)
```

Σε αυτή τη περίπτωση το x δεν είναι πολύ καλό όνομα για μια μεταβλητή.

Ένα καλύτερο όνομα θα μπορούσε να είναι:

```
    name := "Max"
    fmt.Println("My dog's name is", name)
```

ή ακόμα:

```
    dogsName := "Max"
    fmt.Println("My dog's name is", dogsName)
```

Σε αυτή τη περίπτωση χρησιμοποιούμε ένα ειδικό τρόπο για να συμβολίσουμε πολλαπλές λέξεις με ένα μεταβλητό όνομα γνωστό και ως lower camel case (επίσης γνωστό ως mixed case, bumpy caps, camel back or hump back). Το πρώτο γράμμα της πρώτης λέξης είναι πεζό, το πρώτο γράμμα των επόμενων λέξεων είναι κεφαλαίο και όλα τα άλλα είναι πεζά.

4.2 Scope

Επιστρέφοντας στο πρόγραμμα που είδαμε στην αρχή του κεφαλαίου:

```
package main

import "fmt"

func main() {
    var x string = "Hello World"
    fmt.Println(x)
}
```

Ένας άλλος τρόπος γραφής του προγράμματος θα ήταν αυτός :

```
package main

import "fmt"

var x string = "Hello World"

func main() {
    fmt.Println(x)
}
```

Παρατηρήστε ότι μετακινήσαμε τη μεταβλητή έξω από τη συνάρτηση main.
Αυτό σημαίνει ότι και άλλες συναρτήσεις μπορούν να έχουν πρόσβαση σε αυτή τη μεταβλητή:

```
var x string = "Hello World"

func main() {
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

Η συνάρτηση `f` τώρα, έχει πρόσβαση στη μεταβλητή `x`.
Τώρα, ας υποθέσουμε ότι γράψαμε αυτό:

```
func main() {
    var x string = "Hello World"
    fmt.Println(x)
}

func f() {
    fmt.Println(x)
}
```

Αν τρέξετε αυτό το πρόγραμμα θα δείτε ένα λάθος:

```
.\main.go:11: undefined: x
```

Ο μεταγλωττιστής σας λέει ότι δεν υπάρχει η μεταβλητή x μέσα στη συνάρτηση f. Υπάρχει μόνο στη συνάρτηση main. Το εύρος των τοποθεσιών που σας επιτρέπετε να χρησιμοποιείτε την x ονομάζετε scope της μεταβλητής. Βασικά, αυτό σημαίνει ότι η μεταβλητή υπάρχει εντός των πλησιέστερων αγκύλων { } (ένα μπλόκ) συμπεριλαμβανομένων και των εμφωλιασμένων { } αλλά όχι εκτός αυτών. Το scope μπορεί να είναι λίγο μπερδεμένο για αρχή αλλά όσο βλέπουμε προγράμματα θα γίνετε πιο σαφές.

4.3 Σταθερές

Η Go υποστηρίζει επίσης σταθερές . Οι σταθερές είναι στην ουσία μεταβλητές των οποίων η τιμή δε μπορεί να αλλάξει αργότερα. Δημιουργούνται με τον ίδιο τρόπο με τις μεταβλητές αλλά αντί να χρησιμοποιούμε τη λέξη-κλειδί var χρησιμοποιούμε την const.

```
package main

import "fmt"

func main() {
    const x string = "Hello World"
    fmt.Println(x)
}
```

Αυτό:

```
    const x string = "Hello World"
    x = "Some other string"
```

Οδηγεί σε ένα σφάλμα μεταγλώττισης :

```
.\main.go:7: cannot assign to x
```

Οι σταθερές είναι ένας καλός τρόπος για να επαναχρησιμοποιούμε κοινές τιμές σε ένα πρόγραμμα χωρίς να χρειάζεται να τις γράφουμε κάθε φορά. Για παράδειγμα, το Pi στο πακέτο math ορίζετε ως σταθερά.

4.4 Ορισμός πολλαπλών μεταβλητών.

Η Go έχει μια άλλη συντομογραφία όταν χρειάζεται να ορίσετε πολλές μεταβλητές:

```
var (
    a = 5
    b = 10
    c = 15
)
```

Χρησιμοποιήστε τη λέξη-κλειδί var (ή const) ακολουθούμενη από παρενθέσεις, με κάθε μεταβλητή σε διαφορετική γραμμή.

4.5 Παράδειγμα Προγράμματος.

Εδώ είναι ένα πρόγραμμα το οποίο παίρνει ένα αριθμό που εισάγετε από το χρήστη και τον διπλασιάζει:

```
package main

import "fmt"

func main() {
    fmt.Print("Enter a number: ")
    var input float64
    fmt.Scanf("%f", &input)

    output := input * 2

    fmt.Println(output)
}
```

Χρησιμοποιούμε μια άλλη συνάρτηση από το πακέτο fmt για να διαβάσουμε την είσοδο του χρήστη (Scanf). Η &input θα επεξηγηθεί σε επόμενο κεφάλαιο, προς το παρών χρειάζεται να ξέρουμε ότι η Scanf βάζει στην είσοδο τον αριθμό που εισάγαμε.

Προβλήματα:

1. Ποιοι είναι οι δυο τρόποι δημιουργίας μιας νέας μεταβλητής;

2. Ποια είναι η τιμή της x αφού τρέξουμε την: x := 5; x += 1 ;

3. Τι είναι scope και πώς προσδιορίζετε το scope μιας μεταβλητής στην Go;

4. Ποιες είναι οι διαφορές μεταξύ var και const;

5. Χρησιμοποιώντας το πρόγραμμα του παραδείγματος ως βάση, γράψτε ένα πρόγραμμα το οποίο θα μετατρέπει τους βαθμούς από Fahrenheit σε Celsius. (C = (F - 32) * 5/9)

6. Γράψτε ένα άλλο πρόγραμμα το οποίο να μετατρέπει από πόδια σε μέτρα (1 ft =

0.3048 m).

5 Δομές Ελέγχου

Τώρα που ξέρουμε πώς να χρησιμοποιούμε τις μεταβλητές, είναι ώρα να αρχίσουμε να γράφουμε μερικά χρήσιμα προγράμματα. Πρώτα, ας γράψουμε ένα πρόγραμμα που μετράει μέχρι το 10, αρχίζοντας από το 1, με τον κάθε αριθμό στη δική του σειρά. Χρησιμοποιώντας ότι έχουμε μάθει μέχρι τώρα, μπορούμε να γράψουμε αυτό:

```
package main

import "fmt"

func main() {
    fmt.Println(1)
    fmt.Println(2)
    fmt.Println(3)
    fmt.Println(4)
    fmt.Println(5)
    fmt.Println(6)
    fmt.Println(7)
    fmt.Println(8)
    fmt.Println(9)
    fmt.Println(10)
}
```

Ή αυτό:

```
package main
import "fmt"

func main() {
    fmt.Println(`1
2
3
4
5
6
7
8
9
10`)
}
```

Αλλά και τα δύο αυτά προγράμματα είναι αρκετά κουραστικό να γραφτούν.
Αυτό που χρειαζόμαστε είναι ένας τρόπος ώστε να επαναλάβουμε κάτι πολλές φορές.

5.1 For

Η εντολή for μας επιτρέπει να επαναλαμβάνουμε μια σειρά από εντολές πολλές φορές. Ξαναγράφοντας το προηγούμενο πρόγραμμα χρησιμοποιώντας την for, θα δείχνει κάπως έτσι:

```
package main

import "fmt"

func main() {
    i := 1
    for i <= 10 {
        fmt.Println(i)
        i = i + 1
    }
}
```

Πρώτα, δημιουργήσαμε μια μεταβλητή ονόματι i την οποία χρησιμοποιούμε για να αποθηκεύσουμε τον αριθμό που θέλουμε να εμφανίσουμε. Κατόπιν, δημιουργήσαμε ένα βρόγχο for χρησιμοποιώντας τη λέξη-κλειδί for, ακολουθούμενη από μια συνθήκη που είναι πότε αληθής πότε ψευδής η οποία τελικά τροφοδοτεί μια σειρά εντολών για να εκτελεστούν. Ο βρόγχος for λειτουργεί έτσι:

1. Αξιολογούμε (τρέχουμε) την έκφραση i <= 10 (“i μικρότερο ή ίσο
του 10”). Αν αυτό αξιολογηθεί ως αληθές τότε εκτελούνται οι εντολές μέσα στο βρόγχο.Αλλιώς, περνάμε στην επόμενη σειρά του προγράμματός μας μετά το βρόγχο (στη περίπτωσή μας δεν υπάρχει τίποτα μετά το βρόχο έτσι βγαίνουμε από το πρόγραμμα).

2.Αφού τρέξουμε τις εντολές μέσα στο βρόγχο, επιστρέφουμε στην αρχή του βρόγχου for και επαναλαμβάνουμε το πρώτο βήμα.

Η γραμμή i = i + 1 είναι εξαιρετικά σημαντική επειδή χωρίς αυτή το i <= 10 θα ήταν πάντα αληθές και το πρόγραμμά μας δε θα σταματούσε ποτέ. (όταν συμβαίνει αυτό τότε αναφερόμαστε σε ατέρμον βρόγχο).
Σαν εξάσκηση ας τρέξουμε το πρόγραμμα όπως θα έκανε ένας υπολογιστής:

• Δημιουργούμαι μια μεταβλητή ονόματι i με την τιμή 1
• Είναι το i <= 10; Ναι
• Εμφάνισε i
• Θέσε στο i το i + 1 (το i είναι τώρα ίσο με 2)
• Είναι το i <= 10; Ναι
• Εμφάνισε i
• Θέσε στο i το i + 1 (το i είναι τώρα ίσο με 3)

• …
• Θέσε στο i το i + 1 (το i είναι τώρα ίσο με 11)
• Είναι το i <= 10; Όχι
• Δεν έμεινε τίποτα να εκτελεστεί, άρα έξοδος.
Άλλες γλώσσες προγραμματισμού έχουν αρκετούς διαφορετικούς τύπους βρόγχων επανάληψης (while, do, until, foreach, …) αλλά η Go έχει μόνο ένα που μπορεί να χρησιμοποιηθεί με πολλούς διαφορετικούς τρόπους.
Το προηγούμενο πρόγραμμα θα μπορούσε να έχει γραφτεί ως:

```
func main() {
    for i := 1; i <= 10; i++ {
        fmt.Println(i)
    }
}
```

Τώρα η συνθήκη περιέχει και άλλες δυο καταστάσεις έχοντας ερωτηματικά μεταξύ τους. Πρώτα, έχουμε την αρχικοποίηση της μεταβλητής, κατόπιν, τη συνθήκη που θα ελέγχουμε κάθε φορά και τελευταία την αύξηση της μεταβλητής. (το να προσθέτουμε 1 σε μια μεταβλητή είναι πολύ κοινό, έτσι έχουμε μια ξεχωριστή εντολή: ++. Ομοίως, η μείωση κατά 1 γίνετε με: --).

Θα δούμε επιπλέον τρόπους χρήσης του βρόγχου επανάληψης for σε επόμενα κεφάλαια.

5.2 If

Ας τροποποιήσουμε το πρόγραμμα που γράψαμε και αντί να εμφανίζουμε τους αριθμούς 1-10 σε ξεχωριστές γραμμές να διευκρινίζουμε και ποιοι είναι άρτιοι και ποιοι περιττοί.
Όπως εδώ:

```
1 odd
2 even
3 odd
4 even
5 odd
6 even
7 odd
8 even
9 odd
10 even
```

Πρώτα, χρειαζόμαστε ένα τρόπο να καθορίζουμε πότε ένας αριθμός είναι άρτιος και πότε περιττός. Ένας εύκολος τρόπος είναι να διαιρέσουμε τον αριθμό με το 2. Αν δεν έχουμε υπόλοιπο τότε ο αριθμός είναι άρτιος, αλλιώς περιττός. Επομένως, πώς θα βρούμε το υπόλοιπο μετά τη διαίρεση στη Go; Θα χρησιμοποιήσουμε τον τελεστή %. 1 % 2 ισούται με 1, 2 % 2 ισούται με 0, 3 % 2 ισούται με 1 κοκ.

Μετά, χρειαζόμαστε ένα τρόπο να εκτελούμε διαφορετικά πράγματα σύμφωνα με μια συνθήκη. Γι' αυτό χρησιμοποιούμε την εντολή if:

```
if i % 2 == 0 {
  // even
} else {
  // odd
}
```

Η εντολή if είναι παρόμοια με τη for όσον αφορά ότι η συνθήκη ακολουθείτε από ένα βρόγχο. Η εντολή if μπορεί να ακολουθείτε από μια else. Αν η συνθήκη είναι αληθής τότε ο βρόγχος μετά τη συνθήκη εκτελείτε, αλλιώς παραλείπετε και εκτελείτε ο βρόγχος μετά την else.

Η εντολή if μπορεί να περιέχει και την εντολή else if:

```
if i % 2 == 0 {
    // divisible by 2
} else if i % 3 == 0 {
    // divisible by 3
} else if i % 4 == 0 {
    // divisible by 4
}
```

Οι συνθήκες ελέγχονται από πάνω προς τα κάτω και η πρώτη στη σειρά που θα βρεθεί αληθής θα εκτελέσει τις εντολές που βρίσκονται στο βρόγχο της. Καμιά απ' τις εντολές των άλλων βρόγχων δε θα εκτελεστεί ακόμα και αν η συνθήκη τους είναι αληθής. (Για παράδειγμα, ο αριθμός 8 διαιρείτε και με το 4 και με το 2, αλλά η `// divisible by 4` δε θα εκτελεστεί ποτέ επειδή η `// divisible by 2` εκτελείτε πρώτα.

Βλέποντάς τα όλα μαζί έχουμε:

```
func main() {
    for i := 1; i <= 10; i++ {
        if i % 2 == 0 {
            fmt.Println(i, "even")
        } else {
            fmt.Println(i, "odd")
        }
    }
}
```

Ας τρέξουμε το πρόγραμμα:

• Δημιουργούμε μια μεταβλητή `i` του τύπου ακέραιου `int` και της δίνουμε την τιμή `1`
• Είναι η `i` μικρότερη ή ίση του `10`; Ναι: Πήδα στο βρόγχο
• Είναι το υπόλοιπο της `i ÷ 2` ίσο του `0`; Όχι: πήδα στο βρόγχο `else`
• Εμφάνισε `i` ακολουθούμενο από `odd`
• Αύξησε την `i` (στη κατάσταση πριν τη συνθήκη)
• Είναι η `i` μικρότερη ή ίση του `10`; Ναι: Πήδα στο βρόγχο
• Είναι το υπόλοιπο της `i ÷ 2` ίσο του `0`; Ναι: Πήδα στο βρόγχο `if`
• Εμφάνισε `i` ακολουθούμενο από `even`

• …

Ο τελεστής του ακέραιου υπολοίπου γενικά χρησιμοποιείτε σπάνια αλλά στον προγραμματισμό είναι πραγματικά χρήσιμος. Συναντάτε παντού, από τους zebra striping πίνακες εώς το διαχωρισμό των συνόλων δεδομένων.

5.3 Switch

Υποθέτουμε ότι θέλουμε να γράψουμε ένα πρόγραμμα όπου εμφανίζει τα Αγγλικά ονόματα των αριθμών. Χρησιμοποιώντας ότι έχουμε μάθει μπορούμε να ξεκινήσουμε ως:

```
if i == 0 {
    fmt.Println("Zero")
} else if i == 1 {
    fmt.Println("One")
} else if i == 2 {
    fmt.Println("Two")
} else if i == 3 {
    fmt.Println("Three")
} else if i == 4 {
    fmt.Println("Four")
} else if i == 5 {
    fmt.Println("Five")
}
```

Για να γράψουμε ένα πρόγραμμα με αυτό το τρόπο θα ήταν πολύ κουραστικό.
Η Go παρέχει ένα άλλο τρόπο για να το κάνει ευκολότερο. Την εντολή switch. Μπορούμε να ξαναγράψουμε το πρόγραμμά μας έτσι:

```
switch i {
case 0: fmt.Println("Zero")
case 1: fmt.Println("One")
case 2: fmt.Println("Two")
case 3: fmt.Println("Three")
case 4: fmt.Println("Four")
case 5: fmt.Println("Five")
default: fmt.Println("Unknown Number")
}
```

Η εντολή switch ξεκινάει με τη λέξη-κλειδί switch ακολουθούμενη από μια παράσταση (σε αυτή τη περίπτωση την i) και ύστερα με μια σειρά από περιπτώσεις (cases). Η τιμή της παράστασης συγκρίνετε με τις μεταβλητές που ακολουθούν μετά τη λέξη-κλειδί case. Αν είναι ισοδύναμες τότε εκτελούνται οι εντολές που βρίσκονται μετά την :.

Όπως στην εντολή if, κάθε case ελέγχετε από πάνω προς τα κάτω και η πρώτη στη σειρά που χαρακτηριστεί αληθής εκτελείτε. Η switch υποστηρίζει επίσης και μία default case η οποία θα εκτελεστεί αν καμία από τις παραπάνω δεν ταιριάζει τις τιμής που δόθηκε.. (όπως η else σε ένα βρόγχο if).

Αυτές είναι οι βασικές εντολές ελέγχου ροής. Επιπλέον εντολές θα διερευνηθούν σε επόμενα κεφάλαια.

Προβλήματα:

1. Τι εμφανίζει το επόμενο πρόγραμμα:

```
i := 10
if i > 10 {
    fmt.Println("Big")
} else {
    fmt.Println("Small")
}
```

2. Γράψτε ένα πρόγραμμα που να εμφανίζει όλους τους αριθμούς που διαιρούνται με το 3 μεταξύ 1 και 100. (3,6,9,κτλ.).

3. Γράψτε ένα πρόγραμμα που να εμφανίζει τους αριθμούς από το 1 εως το 100. Αλλά για τα πολλαπλάσια του 3 να εμφανίζει "Fizz" αντί του αριθμού και για τα πολλαπλάσια του 5 να εμφανίζει "Buzz". Για τους αριθμούς που είναι πολλαπλάσια και των δύο να εμφανίζει "FizzBuzz".

6 Πίνακες, Slices και Χάρτες

Στο κεφάλαιο 3 μάθαμε για τους βασικούς τύπους της Go. Σε αυτό το κεφάλαιο θα εξετάσουμε τρεις ακόμα ενσωματωμένους τύπους: πίνακες, slices και χάρτες.

6.1 Πίνακες

Ένας πίνακας είναι μια αριθμημένη ακολουθία στοιχείων ίδιου τύπου με σταθερό μήκος. Στη Go είναι:

```
var x [5]int
```

x είναι ένα παράδειγμα ενός πίνακα που αποτελείται από 5 int. Δοκιμάστε να τρέξετε το ακόλουθο πρόγραμμα:

```
package main

import "fmt"

func main() {
     var x [5]int
     x[4] = 100
     fmt.Println(x)
}
```

Θα πρέπει να δείτε:

```
[0 0 0 0 100]
```

x[4] = 100 πρέπει να διαβαστεί «βάλε στο πέμπτο στοιχείο του πίνακα x την τιμή 100». Μπορεί να φαίνεται περίεργο ότι x[4] αντιπροσωπεύει το πέμπτο στοιχείο αντί του τετάρτου αλλά όπως στα strings έτσι και στους πίνακες οι δείκτες θέσης ξεκινούν από το 0. Οι πίνακες είναι προσβάσιμοι με παρόμοιο τρόπο. Θα μπορούσαμε να αλλάξουμε το fmt.Println(x) σε fmt.Println(x [4]) και να λάβουμε το 100.

Εδώ είναι ένα παράδειγμα προγράμματος που χρησιμοποιεί πίνακα:

```
func main() {
	var x [5]float64
	x[0] = 98
	x[1] = 93
	x[2] = 77
	x[3] = 82
	x[4] = 83

	var total float64 = 0
	for i := 0; i < 5; i++ {
		total += x[i]
	}
	fmt.Println(total / 5)
}
```

Αυτό το πρόγραμμα υπολογίζει τον μέσο όρο μιας σειράς βαθμολογιών. Αν το τρέξετε θα πρέπει να δείτε 86.6. Ας δούμε το πρόγραμμα:

• Πρώτα θα δημιουργήσουμε ένα πίνακα μήκους 5 στοιχείων για να κρατήσει τα βαθμολογίες μας, μετά θα γεμίσουμε κάθε στοιχείο με ένα βαθμό.
• Στη συνέχεια θα δημιουργήσουμε ένα βρόχο for για να υπολογιστεί το συνολικό σκορ.
• Τέλος θα διαιρέσουμε το συνολικό σκορ με τον αριθμό των στοιχείων για να βρούμε το μέσο όρο

Το πρόγραμμα αυτό λειτουργεί, αλλά η Go παρέχει ορισμένες δυνατότητες που μπορούμε να χρησιμοποιήσουμε για να το βελτιώσουμε. Πρώτα, αυτά τα 2 μέρη: i <5 και total / 5 θα πρέπει να εμφανίσει ένα σφάλμα. Ας πούμε ότι αλλάξαμε τον αριθμό των στοιχείων από 5 σε 6. Θα πρέπει επίσης να αλλάξουμε και τα δύο από αυτά τα μέρη. Θα ήταν καλύτερο να χρησιμοποιήσουμε το μήκος του πίνακα, αντί:

```
	var total float64 = 0
	for i := 0; i < len(x); i++ {
		total += x[i]
	}
	fmt.Println(total / len(x))
```

Προχωρήστε, κάνετε αυτές τις αλλαγές και τρέξτε το πρόγραμμα. Θα πρέπει να

πάρετε ένα σφάλμα:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:19: invalid operation: total / 5
(mismatched types float64 and int)
```

Το πρόβλημα εδώ είναι ότι len(x) και total έχουν διαφορετικούς τύπους. Το total είναι ένα float64 ενώ το len(x) είναι ένα int. Γι 'αυτό και πρέπει να μετατρέψουμε το len(x) σε ένα float64:

```
fmt.Println(total / float64(len(x)))
```

Αυτό είναι ένα παράδειγμα ενός τύπου μετατροπής. Γενικά για τη μετατροπή μεταξύ των διαφόρων τύπων χρησιμοποιήστε το όνομα του τύπου σαν συνάρτηση.

Μια άλλη αλλαγή στο πρόγραμμα που μπορούμε να κάνουμε είναι να χρησιμοποιήσουμε μια ειδική φόρμα για τον βρόγχο for:

```
var total float64 = 0
for i, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

Σε αυτό το βρόχο for το i αντιπροσωπεύει την τρέχουσα θέση στο πίνακα και η value είναι η ίδια με το x [i]. Χρησιμοποιούμε τη λέξη-κλειδί range που ακολουθείται από το όνομα της μεταβλητής που θέλουμε να επιστρέφει ο βρόχος.
Η εκτέλεση αυτού του προγράμματος θα οδηγήσει σε ένα άλλο σφάλμα:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:16: i declared and not used
```

Ο μεταγλωττιστής της Go δεν θα σας επιτρέψει να δημιουργήσετε μεταβλητές που δεν χρησιμοποιήσατε ποτέ. Από τη στιγμή που δεν χρησιμοποιήσαμε το i στο

εσωτερικό του βρόχου μας, θα πρέπει να το αλλάξουμε σε αυτό:

```
var total float64 = 0
for _, value := range x {
    total += value
}
fmt.Println(total / float64(len(x)))
```

Μία μόνο _ (underscore) χρησιμοποιείται για να πει στον μεταγλωττιστή ότι δεν το χρειαζόμαστε αυτό. (Σε αυτή την περίπτωση δεν χρειαζόμαστε την επαναληπτική μεταβλητή.

Η Go παρέχει επίσης μια μικρότερη σύνταξη για τη δημιουργία πίνακα:

```
x := [5]float64{ 98, 93, 77, 82, 83 }
```

Εμείς δεν χρειάζεται πλέον να καθορίσουμε τον τύπο επειδή η Go μπορεί να το καταλάβει. Μερικές φορές οι πίνακες όπως αυτό, μπορεί να έχουν πάρα πολύ μεγάλο μήκος και να χωρέσουν σε μια γραμμή, έτσι η Go σας επιτρέπει να το χωρίσετε όπως εδώ:

```
x := [5]float64{
    98,
    93,
    77,
    82,
    83,
}
```

Παρατηρήστε το επιπλέον κόμμα , μετά το 83. Αυτό απαιτείται από τη Go και μας επιτρέπει να αφαιρέσουμε εύκολα ένα στοιχείο από ένα πίνακα μετατρέποντάς το σε σχόλιο:

```
x := [4]float64{
    98,
    93,
    77,
    82,
    // 83,
}
```

Αυτό το παράδειγμα απεικονίζει ένα σημαντικό ζήτημα των πινάκων:
Το μήκος τους είναι σταθερό και μέρος του ονόματος του τύπου του πίνακα.
Για να αφαιρέσετε το τελευταίο στοιχείο, θα πρέπει στην πραγματικότητα να αλλάξετε και τον τύπο επίσης. Η λύση της Go σε αυτό το πρόβλημα είναι να χρησιμοποιήσουμε έναν διαφορετικό τύπο: slices.

6.2 Slices

Ένα slice είναι ένα τμήμα ενός πίνακα. Όπως τους πίνακες έτσι και τα slices έχουνε δείκτες θέσης και καθορισμένο μήκος. Σε αντίθεση με τους πίνακες, αυτό το μήκος επιτρέπεται να αλλάξει. Εδώ είναι ένα παράδειγμα από ένα slice:

```
var x []float64
```

Η μόνη διαφορά μεταξύ αυτού και ενός πίνακα είναι ότι λείπει το μήκος μεταξύ των αγκύλων. Στην περίπτωση αυτή, το x έχει δημιουργηθεί με ένα μήκος 0.

Αν θέλετε να δημιουργήσετε ένα slice θα πρέπει να χρησιμοποιήσετε την ενσωματωμένη συνάρτηση make:

```
x := make([]float64, 5)
```

Αυτή δημιουργεί ένα slice που συσχετίζεται με έναν υπο-πίνακα τύπου float64 με μήκος 5. Τα slices είναι πάντα συσχετισμένα με κάποιους και παρόλο που δεν μπορούν ποτέ να είναι μεγαλύτερα από τους πίνακες, μπορούν να είναι μικρότερα. Η συνάρτηση make επιτρέπει επίσης μια τρίτη παράμετρο:

```
x := make([]float64, 5, 10)
```

Το 10 αντιπροσωπεύει την χωρητικότητα του υπο-πίνακα στον οποίο εμπεριέχετε το slice:

Ένας άλλος τρόπος για να δημιουργήσετε τα slices είναι να χρησιμοποιήσετε την έκφραση [low : high]:

```
arr := []float64{1,2,3,4,5}
x := arr[0:5]
```

Το low είναι ο δείκτης για το πού να ξεκινήσει το slice και το high είναι ο δείκτης που να τελειώσει (αλλά δεν συμπεριλαμβάνεται ο ίδιος ο δείκτης). Για παράδειγμα ενώ το arr[0:5] επιστρέφει [1,2,3,4,5], το arr[1:4] επιστρέφει [2,3,4].

Για ευκολία επιτρέπεται επίσης να παραλείψετε το low, high ή ακόμη και τα δύο low και high. arr[0]: είναι το ίδιο με arr[0:len(arr)], arr[: 5] είναι το ίδιο με arr[0:5] και arr[:] είναι το ίδιο με arr[0:len(arr)].

Slice Functions

Η Go περιλαμβάνει δύο ενσωματωμένες συναρτήσεις για να βοηθήσει με τα slices: append και copy. Εδώ είναι ένα παράδειγμα της append:

```
func main() {
    slice1 := []int{1,2,3}
    slice2 := append(slice1, 4, 5)
    fmt.Println(slice1, slice2)
}
```

Μετά την εκτέλεση αυτού του προγράμματος το slice1 έχει [1,2,3] και το slice2 έχει [1,2,3,4,5]. Η append δημιουργεί ένα νέο slice λαμβάνοντας ένα υπάρχον slice (το πρώτο όρισμα) και προσαρτίζει όλα τα ακόλουθα ορίσματα σε αυτό.

Εδώ είναι ένα παράδειγμα της copy:

```
func main() {
     slice1 := []int{1,2,3}
     slice2 := make([]int, 2)
     copy(slice2, slice1)
     fmt.Println(slice1, slice2)
}
```

Μετά την εκτέλεση αυτού του προγράμματος το slice1 έχει [1,2,3] και το slice2 έχει [1,2]. Τα περιεχόμενα του slice1 αντιγράφονται μέσα στο slice2, αλλά από τη στιγμή που το slice2 έχει χώρο για δύο μόνο στοιχεία μόνο τα δύο πρώτα στοιχεία του slice1 αντιγράφονται.

6.3 Χάρτες

Ένας χάρτης είναι μια μη διατεταγμένη συλλογή από ζευγάρια κλειδιού-τιμής.
Επίσης γνωστό ως συσχετιστικός πίνακας, πίνακας κατατεμαχισμού ή λεξικό, οι χάρτες χρησιμοποιούνται για να αναζητήσετε μια τιμή από το αντίστοιχο κλειδί του.
Εδώ είναι ένα παράδειγμα ενός χάρτη σε Go:

```
var x map[string]int
```

Ο τύπος χάρτη αντιπροσωπεύεται από την λέξη-κλειδί map, που ακολουθείται από τον τύπο του κλειδιού σε αγκύλες και τέλος τον τύπο της τιμής. Εάν επρόκειτο να το διαβάσετε φωναχτά θα λέγατε "x είναι ένας χάρτης από string σε int."

Όπως οι πίνακες και τα slice, οι χάρτες μπορούν να είναι προσβάσιμοι με τη χρήση παρενθέσεων. Δοκιμάστε να εκτελέσετε το ακόλουθο πρόγραμμα:

```
var x map[string]int
x["key"] = 10
fmt.Println(x)
```

Θα πρέπει να δείτε ένα σφάλμα παρόμοιο με αυτό:

```
panic: runtime error: assignment to entry in nil
map

goroutine 1 [running]:
main.main()
  main.go:7 +0x4d

goroutine 2 [syscall]:
created by runtime.main

C:/Users/ADMINI~1/AppData/Local/Temp/2/bindi
t269497170/go/src/pkg/runtime/proc.c:221
exit status 2
```

Μέχρι τώρα έχουμε δει μόνο τα σφάλματα χρόνου μεταγλώττισης. Αυτό είναι ένα παράδειγμα ενός σφάλματος χρόνου εκτέλεσης. Όπως το λέει το όνομα, τα σφάλματα χρόνου εκτέλεσης συμβαίνουν όταν εκτελείτε το πρόγραμμα, ενώ τα σφάλματα χρόνου μεταγλώττισης συμβαίνουν όταν προσπαθείτε να μεταγλωττίσετε το πρόγραμμα.

Το πρόβλημα με το πρόγραμμά μας είναι ότι οι χάρτες πρέπει να αρχικοποιηθούν πριν τους χρησιμοποιήσουμε. Θα έπρεπε να είχαμε γράψει αυτό:

```
x := make(map[string]int)
x["key"] = 10
fmt.Println(x["key"])
```

Εάν εκτελέσετε αυτό το πρόγραμμα θα πρέπει να δείτε να εμφανίζεται το 10. Η εντολή `x["key"] = 10` είναι παρόμοια με αυτή που είδαμε στους πίνακες, αλλά το κλειδί αντί να είναι ένας ακέραιος, είναι ένα string επειδή ο τύπος του κλειδιού του χάρτη είναι `string`. Μπορούμε επίσης να δημιουργήσουμε ένα χάρτη με ένα τύπο κλειδιού `int`:

```
x := make(map[int]int)
x[1] = 10
fmt.Println(x[1])
```

Αυτό μοιάζει πολύ με ένα πίνακα αλλά υπάρχουν μερικές διαφορές. Πρώτον, το

μήκος του χάρτη (που βρίσκεται χρησιμοποιώντας την len(x)) μπορεί να αλλάξει καθώς θα προσθέσουμε νέα στοιχεία σε αυτό. Όταν δημιουργείται για πρώτη φορά έχει μήκος 0, μετά την x[1] = 10 έχει μήκος 1. Δεύτερον, οι χάρτες δεν είναι διαδοχικά. Έχουμε x[1], που σε ένα πίνακα θα σήμαινε ότι πρέπει να είναι ένα x[0], αλλά οι χάρτες δεν έχουν αυτή την απαραίτητη προϋπόθεση.

Μπορούμε επίσης να διαγράψουμε στοιχεία από ένα χάρτη χρησιμοποιώντας την ενσωματωμένη συνάρτηση delete:

```
delete(x, 1)
```

Ας δούμε ένα παράδειγμα προγράμματος που χρησιμοποιεί ένα χάρτη:

```
package main

import "fmt"

func main() {
    elements := make(map[string]string)
    elements["H"] = "Hydrogen"
    elements["He"] = "Helium"
    elements["Li"] = "Lithium"
    elements["Be"] = "Beryllium"
    elements["B"] = "Boron"
    elements["C"] = "Carbon"
    elements["N"] = "Nitrogen"
    elements["O"] = "Oxygen"
    elements["F"] = "Fluorine"
    elements["Ne"] = "Neon"

    fmt.Println(elements["Li"])
}
```

elements είναι ένας χάρτης που αντιπροσωπεύει τα πρώτα 10 χημικά στοιχεία που

συντάσσονται με το σύμβολο τους. Αυτός είναι ένας πολύ κοινός τρόπος χρήσης των χαρτών: σαν έναν πίνακα αναζήτησης ή λεξικό. Ας υποθέσουμε ότι προσπαθήσαμε να αναζητήσουμε ένα στοιχείο που δεν υπάρχει:

```
fmt.Println(elements["Un"])
```

Εάν εκτελέσετε αυτό θα πρέπει να μην δείτε τίποτα να εμφανίζει. Τεχνικά ένας χάρτης επιστρέφει την τιμή μηδέν για τον τύπο της τιμής (η οποία για string είναι το κενό string). Παρόλο που θα μπορούσαμε να ελέγξουμε τη μηδενική τιμή στην περίπτωση (elements["Un"] == "") η Go παρέχει έναν καλύτερο τρόπο:

```
name, ok := elements["Un"]
fmt.Println(name, ok)
```

Η πρόσβαση σε ένα στοιχείο ενός χάρτη μπορεί να επιστρέψει δύο τιμές αντί για μια. Η πρώτη τιμή είναι το αποτέλεσμα της αναζήτησης, η δεύτερη μας λέει αν ήταν ή όχι η αναζήτηση επιτυχής. Στη Go βλέπουμε συχνά κώδικα όπως αυτό:

```
if name, ok := elements["Un"]; ok {
    fmt.Println(name, ok)
}
```

Πρώτα θα προσπαθήσουμε να πάρουμε την τιμή από το χάρτη, στη συνέχεια αν είναι επιτυχής θα εκτελέσουμε τον κώδικα μέσα στο μπλοκ.
Όπως είδαμε στους πίνακες, υπάρχει επίσης ένας συντομότερος τρόπος για να δημιουργήσετε ένα χάρτη:

```
elements := map[string]string{
    "H": "Hydrogen",
    "He": "Helium",
    "Li": "Lithium",
    "Be": "Beryllium",
    "B": "Boron",
    "C": "Carbon",
    "N": "Nitrogen",
    "O": "Oxygen",
    "F": "Fluorine",
    "Ne": "Neon",
}
```

Οι χάρτες επίσης, συχνά χρησιμοποιούνται για την αποθήκευση γενικών πληροφοριών. Ας τροποποιήσουμε το πρόγραμμά μας, έτσι ώστε εκτός από την αποθήκευση του ονόματος του στοιχείου να αποθηκεύουμε και την αρχική του κατάσταση (κατάσταση σε θερμοκρασία δωματίου):

```
func main() {
    elements := map[string]map[string]string{
        "H": map[string]string{
            "name":"Hydrogen",
            "state":"gas",
        },
        "He": map[string]string{
            "name":"Helium",
            "state":"gas",
        },
        "Li": map[string]string{
            "name":"Lithium",
            "state":"solid",
        },
```

```
            "Be": map[string]string{
                  "name":"Beryllium",
                  "state":"solid",
            },
            "B":  map[string]string{
                  "name":"Boron",
                  "state":"solid",
            },
            "C":  map[string]string{
                  "name":"Carbon",
                  "state":"solid",
            },
            "N":  map[string]string{
                  "name":"Nitrogen",
                  "state":"gas",
            },
            "O":  map[string]string{
                  "name":"Oxygen",
                  "state":"gas",
            },
            "F":  map[string]string{
                  "name":"Fluorine",
                  "state":"gas",
            },
            "Ne":  map[string]string{
                  "name":"Neon",
                  "state":"gas",
            },
      }

      if el, ok := elements["Li"]; ok {
            fmt.Println(el["name"], el["state"])
      }
}
```

Παρατηρήστε ότι ο τύπος του χάρτη μας έχει αλλάξει από map[string]string σε map[string]map[string]string. Τώρα έχουμε ένα χάρτη από string σε χάρτη από string σε string. Ο εξωτερικός χάρτης χρησιμοποιείται ως ένας πίνακας αναζήτησης με βάση το σύμβολο του στοιχείου, ενώ οι εσωτερικοί χάρτες χρησιμοποιούνται για την αποθήκευση γενικών πληροφοριών σχετικά με τα στοιχεία. Αν και οι χάρτες συχνά χρησιμοποιούνται σαν αυτό, στο κεφάλαιο 9 θα δούμε έναν καλύτερο τρόπο για την αποθήκευση δομημένων πληροφοριών.

Προβλήματα:

1. Πώς μπορείτε να έχετε πρόσβαση στο τέταρτο στοιχείο ενός πίνακα ή ενός slice;

2. Ποιο είναι το μήκος ενός slice που δημιουργείται χρησιμοποιώντας: make([]int, 3, 9);

3. Δίνεται ο πίνακας:

```
x := [6]string{"a","b","c","d","e","f"}
```

τι θα σας δώσει το x[2:5];

4. Γράψτε ένα πρόγραμμα που να βρίσκει το μικρότερο αριθμό σε αυτή τη λίστα:

```
x := []int{
    48,96,86,68,
    57,82,63,70,
    37,34,83,27,
    19,97, 9,17,
}
```

7 Συναρτήσεις (Functions)

Μία συνάρτηση είναι ένα ανεξάρτητο τμήμα κώδικα που ορίζει μηδέν ή περισσότερες παραμέτρους εισόδου και μηδέν ή περισσότερες παραμέτρους εξόδου. Οι συναρτήσεις(επίσης γνωστές ως διαδικασίες (procedures) ή υπορουτίνες (subroutines) συχνά εκπροσωπούνται ως ένα μαύρο κουτί: (το μαύρο κουτί αντιπροσωπεύει τη συνάρτηση)

Μέχρι τώρα στα προγράμματα που έχουμε γράψει στη Go, έχουμε χρησιμοποιήσει μόνο μία συνάρτηση:

```
func main() {}
```

Τώρα θα αρχίσουμε να γράφουμε προγράμματα που χρησιμοποιούν περισσότερες από μία συναρτήσεις.

7.1 Η δεύτερη Συνάρτησή σας

Θυμηθείτε αυτό το πρόγραμμα από το κεφάλαιο 6:

```
func main() {
    xs := []float64{98,93,77,82,83}

    total := 0.0
    for _, v := range xs {
        total += v
    }
    fmt.Println(total / float64(len(xs)))
}
```

Αυτό το πρόγραμμα υπολογίζει τον μέσο όρο μιας σειράς αριθμών. Η εύρεση του μέσου όρου είναι ένα πολύ γενικό πρόβλημα, έτσι είναι ένα ιδανικό παράδειγμα για να το ορίσουμε ως συνάρτηση.

Η συνάρτηση average θα πρέπει να λάβει ένα slice από float64 και να επιστρέφει μια έξοδο float64. Τοποθετήστε αυτό πριν από την συνάρτηση main:

```
func average(xs []float64) float64 {
    panic("Not Implemented")
}
```

Οι συναρτήσεις ξεκινούν με την λέξη-κλειδί func και ακολουθείται από το όνομα της συνάρτησης. Οι παράμετροι (Inputs) της συνάρτησης ορίζονται ως εξής: τύπος όνομα, τύπος όνομα, ... Η συνάρτηση μας έχει μια παράμετρο (η λίστα με τις βαθμολογίες) που την ονομάσαμε xs. Μετά τις παραμέτρους βάζουμε τον τύπο επιστροφής. Συλλογικά, οι παράμετροι και ο τύπος επιστροφής είναι γνωστά ως η υπογραφή της συνάρτησης.

Τέλος, έχουμε το σώμα της συνάρτησης, το οποίο είναι μια σειρά από εντολές μεταξύ των αγκυλών. Σε αυτό το σώμα θα επικαλέσουμε μια ενσωματωμένη συνάρτηση που ονομάζεται panic που προκαλεί ένα σφάλμα χρόνου εκτέλεσης. (Θα δούμε περισσότερα για την panic σε αυτό το κεφάλαιο αργότερα)
Η σύνταξη της συνάρτησης μπορεί να είναι δύσκολη γι 'αυτό μια καλή ιδέα είναι να χωρίσετε τη διαδικασία σε μικρότερα κομμάτια, αντί να προσπαθήσετε να την κάνετε όλη σε ένα μεγάλο κομμάτι.

Τώρα ας πάρουμε τον κώδικα από την συνάρτηση main και να τον βάλουμε μέσα στην συνάρτηση του μέσου όρου:

```
func average(xs []float64) float64 {
    total := 0.0
    for _, v := range xs {
        total += v
    }
    return total / float64(len(xs))
}
```

Παρατηρούμε ότι αντικαταστήσαμε την fmt.Println με την return. Η εντολή return

σταματάει τη συνάρτηση αμέσως και μετά επιστρέφει την τιμή στην συνάρτηση που την κάλεσε αυτή. Τροποποιήστε την main μοιάζει κάπως έτσι:

```
func main() {
     xs := []float64{98,93,77,82,83}
     fmt.Println(average(xs))
}
```

Η εκτέλεση αυτού του προγράμματος θα πρέπει να σας δώσει ακριβώς τα ίδια αποτελέσματα με το αρχικό. Λίγα πράγματα που πρέπει να θυμάστε:

• Τα ονόματα των παραμέτρων δεν πρέπει να ταιριάζουν με το όνομα που καλούμε την συνάρτηση. Για παράδειγμα, θα μπορούσαμε να είχαμε κάνει αυτό:

```
func main() {
     someOtherName := []float64{98,93,77,82,83}
     fmt.Println(average(someOtherName))
}
```

Και το πρόγραμμά μας θα εξακολουθεί να λειτουργεί.

• Οι συναρτήσεις δεν έχουν πρόσβαση σε οτιδήποτε που καλεί την συνάρτηση. Αυτό δεν θα λειτουργήσει:

```
func f() {
     fmt.Println(x)
}
func main() {
     x := 5
     f()
}
```

Πρέπει να κάνουμε αυτό έναντι του άλλου:

```
func f(x int) {
     fmt.Println(x)
}
func main() {
     x := 5
     f(x)
}
```

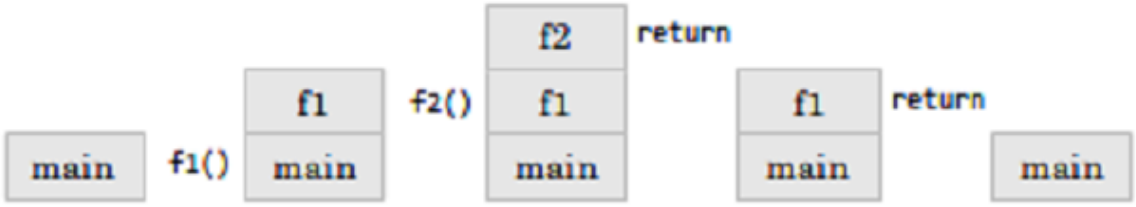
Ή αυτό:

```
var x int = 5
func f() {
     fmt.Println(x)
}
func main() {
     f()
}
```

- Οι συναρτήσεις είναι χτισμένες σε μια «στοίβα». Ας υποθέσουμε ότι είχαμε αυτό το πρόγραμμα:

```
func main() {
     fmt.Println(f1())
}
func f1() int {
     return f2()
}
func f2() int {
     return 1
}
```

Θα μπορούσαμε να το φανταστούμε κάπως έτσι:



Κάθε φορά που καλούμε τη συνάρτηση την ωθούμε στη στοίβα που την καλεί και κάθε φορά που επιστρέφουμε από μια συνάρτηση ξαφνικά πηγαίνουμε στην τελευταία συνάρτηση από την στοίβα.

• Μπορούμε επίσης να αναφέρουμε τον τύπο επιστροφής (return):

```
func f2() (r int) {
    r = 1
    return
}
```

7.2 Επιστροφή Πολλαπλών Τιμών

Η Go είναι επίσης ικανή να επιστρέφει πολλαπλές τιμές από μια συνάρτηση:

```
func f() (int, int) {
     return 5, 6
}

func main() {
     x, y := f()
}
```

Τρεις αλλαγές είναι απαραίτητες: αλλάξτε τον τύπο return έτσι ώστε να περιέχει πολλαπλούς τύπους που χωρίζονται από , , αλλάξτε την έκφραση μετά την return έτσι ώστε να περιέχει πολλές εκφράσεις που χωρίζονται από , και τέλος αλλάξτε την εντολή εκχώρησης, έτσι ώστε οι πολλαπλές τιμές να είναι στην αριστερή πλευρά του := ή =.

Οι πολλαπλές τιμές συχνά χρησιμοποιούνται για να επιστρέψουν μια τιμή σφάλματος σε συνδυασμό με το αποτέλεσμα (x, err : = f ()), ή με μια boolean για να δείξουν την επιτυχία (x, ok: = f ()).

7.3 Συναρτήσεις Variadic

Υπάρχει μια ειδική φόρμα διαθέσιμη για την τελευταία παράμετρο σε μια συνάρτηση της Go:

```
func add(args ...int) int {
     total := 0
     for _, v := range args {
          total += v
     }
     return total
}
func main() {
     fmt.Println(add(1,2,3))
}
```

Με τη χρήση ... πριν από το όνομα του τύπου της τελευταίας παραμέτρου, μπορείτε να παρατηρήσετε ότι λαμβάνει μηδέν ή πολλές από εκείνες τις παραμέτρους. Σε αυτή την περίπτωση παίρνουμε μηδέν ή πολλούς int. Καλούμε τη συνάρτηση όπως και κάθε άλλη συνάρτηση με εξαίρεση ότι μπορούμε να βάλουμε όσα int θέλουμε.

Αυτό είναι ακριβώς το πώς η λειτουργία fmt.Println υλοποιείται:

```
func Println(a ...interface{}) (n int, err
error)
```

Η συνάρτηση Println παίρνει οποιοδήποτε αριθμό τιμών του κάθε είδους. (Ο ειδικός τύπος interface {} θα συζητηθεί λεπτομερώς στο κεφάλαιο 9)

Μπορούμε επίσης να βάλουμε ένα slice ints ακολουθώντας με ... :

```
func main() {
    xs := []int{1,2,3}
    fmt.Println(add(xs...))
}
```

7.4 Εξαναγκασμός κλεισίματος (Closure)

Είναι δυνατόν να δημιουργήσετε συναρτήσεις στο εσωτερικό άλλων συναρτήσεων:

```
func main() {
     add := func(x, y int) int {
          return x + y
     }
     fmt.Println(add(1,1))
}
```

Η add είναι μια τοπική μεταβλητή που έχει τον τύπο func(int, int) int (μια λειτουργία που παίρνει δύο int και επιστρέφει ένα int). Όταν δημιουργείτε μια τοπική συνάρτηση όπως αυτή, έχει επίσης πρόσβαση σε άλλες τοπικές μεταβλητές (θυμηθείτε το scope από το κεφάλαιο 4):

```
func main() {
     x := 0
     increment := func() int {
          x++
          return x
     }
     fmt.Println(increment())
     fmt.Println(increment())
}
```

Η increment προσθέτει το 1 στη μεταβλητή x που ορίζεται στο scope της συνάρτησης main. Αυτή η μεταβλητή x μπορεί να είναι προσβάσιμη και να τροποποιηθεί από τη συνάρτηση increment. Για το λόγο αυτό τη πρώτη φορά που καλούμε την increment βλέπουμε να εμφανίζεται το 1, αλλά τη δεύτερη φορά που την καλούμε βλέπουμε να εμφανίζεται το 2 .

Μια συνάρτηση όπως αυτή μαζί με τις μη τοπικές μεταβλητές είναι γνωστή ώς closure. Στην περίπτωση αυτή η increment και η μεταβλητή x σχηματίζουν την closure.

Ένας τρόπος για να χρησιμοποιήσετε την closure είναι γράφοντας μια συνάρτηση η οποία επιστρέφει μια άλλη συνάρτηση που - όταν καλείται - μπορεί να παράγει μια ακολουθία αριθμών. Για παράδειγμα, εδώ είναι το πώς θα μπορούσαμε να

δημιουργήσουμε όλους τους ζυγούς αριθμούς:

```
func makeEvenGenerator() func() uint {
	i := uint(0)
	return func() (ret uint) {
		ret = i
		i += 2
		return
	}
}
func main() {
	nextEven := makeEvenGenerator()
	fmt.Println(nextEven()) // 0
	fmt.Println(nextEven()) // 2
	fmt.Println(nextEven()) // 4
}
```

Η makeEvenGenerator επιστρέφει μια συνάρτηση η οποία δημιουργεί ζυγούς αριθμούς. Κάθε φορά που καλείται προσθέτει 2 στην τοπική μεταβλητή i η οποία - σε αντίθεση με τις συνήθεις τοπικές μεταβλητές - εξακολουθεί να παραμένει σταθερή μεταξύ των κλήσεων.

7.5 Αναδρομή (Recursion)

Τελικά, μια συνάρτηση που είναι ικανή να καλέσει τον εαυτό της. Εδώ είναι ένας τρόπος για να υπολογιστεί το παραγοντικό ενός αριθμού:

```
func factorial(x uint) uint {
	if x == 0 {
		return 1
	}

	return x * factorial(x-1)
}
```

Η factorial καλεί τον εαυτό της, το οποίο είναι αυτό που την κάνει αναδρομική

συνάρτηση. Για να καταλάβετε καλύτερα πώς λειτουργεί αυτή η συνάρτηση, ας την δούμε μέσο του παραγοντικού 2 (factorial (2)):

• Είναι το x == 0; Όχι. (το x είναι 2)
• Βρές το παραγοντικό του x – 1
  • Είναι το x == 0; Όχι. (το x είναι 1)
  • Βρές το παραγοντικό του x – 1
    • Είναι το x == 0; Ναι, επέστρεψε 1.
  • επέστρεψε 1 * 1
• επέστρεψε 2 * 1

Η Closure και η recursion είναι ισχυρές τεχνικές προγραμματισμού που αποτελούν τη βάση ενός προτύπου γνωστό ως συναρτησιακός προγραμματισμός. Οι περισσότεροι άνθρωποι θα βρουν τον συναρτησιακό προγραμματισμό πιο δύσκολο να κατανοηθεί από τους βρόχους for, τις εντολές if, τις μεταβλητές και τις απλές συναρτήσεις.

7.6 Defer, Panic & Recover

Η Go έχει μια ειδική εντολή που ονομάζεται defer η οποία προγραμματίζει την κλήση μιας συνάρτησης για να τρέξει μετά την ολοκλήρωση της συνάρτησης. Σκεφτείτε το ακόλουθο παράδειγμα:

```
package main

import "fmt"

func first() {
     fmt.Println("1st")
}
func second() {
     fmt.Println("2nd")
}
func main() {
     defer second()
     first()
}
```

Αυτό το πρόγραμμα εμφανίζει το 1st και ακολουθεί το 2nd. Βασικά η defer μετακινεί την κλήση του second στο το τέλος της συνάρτησης:

```
func main() {
    first()
    second()
}
```

Η defer χρησιμοποιείται συχνά όταν οι πόροι θα πρέπει να ελευθερωθούν με κάποιο τρόπο. Για παράδειγμα, όταν ανοίγετε ένα αρχείο θα πρέπει να σιγουρευτείτε ότι θα το κλείσετε αργότερα. Με την defer :

```
f, _ := os.Open(filename)
defer f.Close()
```

Αυτό έχει 3 πλεονεκτήματα: (1) κρατάει την κλήση Close κοντά στην κλήση Open έτσι είναι πιο εύκολο να καταλάβουμε, (2) εάν η συνάρτησή μας είχε πολλές εντολές return (ίσως μια μέσα σε μια if και μέσα σε μια else) η close θα εκτελεστεί πριν και από τις δύο και (3) οι συναρτήσεις defer εκτελούνται ακόμη και αν συμβεί ένα run-time panic.

Panic & Recover

Νωρίτερα δημιουργήσαμε μια συνάρτηση που ονομάζεται panic για να προκαλέσει ένα σφάλμα χρόνου εκτέλεσης. Μπορούμε να διαχειριστούμε ένα χρόνο εκτέλεσης panic με την ενσωματωμένη συνάρτηση recover . Η recover σταματά την panic και επιστρέφει την τιμή που πήρε όταν καλέστηκε η panic. Θα μπορούσαμε να το χρησιμοποιήσουμε και έτσι:

```
package main

import "fmt"

func main() {
     panic("PANIC")
     str := recover()
     fmt.Println(str)
}
```

Αλλά η κλήση της recover δεν πρόκειται ποτέ να συμβεί σε αυτή την περίπτωση, επειδή η κλήση της panic σταματά αμέσως την εκτέλεση της συνάρτησης. Αντ 'αυτού θα πρέπει να το συνδυάσετε με την defer:

```
package main

import "fmt"

func main() {
    defer func() {
        str := recover()
        fmt.Println(str)
    }()
    panic("PANIC")
}
```

Μια panic υποδηλώνει ένα σφάλμα του προγραμματιστή (για παράδειγμα, προσπαθεί να αποκτήσει πρόσβαση σε ένα δείκτη θέσης στοιχείου ενός πίνακα που είναι εκτός ορίων, όταν έχει ξεχαστεί η αρχικοποίηση ενός χάρτη, κλπ.) ή μία εξαιρούμενη κατάσταση που δεν υπάρχει εύκολος τρόπος ανάκτησής της από κάπου. (Εξ ου και η ονομασία «πανικός (panic)»)

Προβλήματα:

1. Η sum είναι μια συνάρτηση η οποία παίρνει ένα slice από αριθμό και τους προσθέτει μεταξύ τους. Πώς θα έμοιαζε η υπογραφή της συνάρτηση στην Go;

2. Γράψτε μια συνάρτηση η οποία να παίρνει έναν ακέραιο, να τον διαιρεί στη μέση και να επιστρέφει true αν είναι άρτιο ή false αν είναι περιττός. Για παράδειγμα half(1) θα πρέπει να επιστρέψει (0, false) και half(2) θα πρέπει να επιστρέψει (1, true).

3. Γράψτε μια συνάρτηση με μία variadic παράμετρο που να βρίσκει το μεγαλύτερο αριθμό από μια λίστα αριθμών.

4. Χρησιμοποιώντας την makeEvenGenerator σαν παράδειγμα, γράψτε μια συνάρτηση makeOddGenerator που να παράγει περιττούς αριθμούς.

5. Η ακολουθία Fibonacci ορίζεται ως: fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2). Γράψτε μια συνάρτηση recursive που να μπορεί να βρει το fib(n).

6. Τι είναι η defer, η panic και η recover; Πώς θα κάνετε ανάκτηση από ένα run-time panic;

8 Δείκτες

Όταν καλούμε μια συνάρτηση, το όρισμα που παίρνει αντιγράφετε και στη συνάρτηση.

```
func zero(x int) {
     x = 0
}
func main() {
     x := 5
     zero(x)
     fmt.Println(x) // x is still 5
}
```

Σε αυτό το πρόγραμμα η συνάρτηση zero δεν μορφοποιεί την αρχική μεταβλητή x στη συνάρτησης main. Αλλά τί θα γινόταν αν εμείς το θέλαμε;
Ένας τρόπος είναι να χρησιμοποιήσουμε ένα ειδικό τύπο δεδομένων, γνωστό και ως δείκτη:

```
func zero(xPtr *int) {
     *xPtr = 0
}
func main() {
     x := 5
     zero(&x)
     fmt.Println(x) // x is 0
}
```

Οι δείκτες αναφέρονται σε μια τοποθεσία στη μνήμη όπου είναι αποθηκευμένη μια τιμή αντί να αναφέρονται στην ίδια την τιμή. Χρησιμοποιώντας ένα δείκτη (*int) η συνάρτηση zero μπορεί να τροποποιήσει την αρχική μεταβλητή.

8.1 Τελεστές * και &

Στη Go ένας δείκτης αναπαρίσταται χρησιμοποιώντας το χαρακτήρα * ακολουθούμενος από το τύπο της αποθηκευμένης τιμής. Στη συνάρτηση zero ο xPtr είναι ένας δείκτης μιας int.

* χρησιμοποιείτε επίσης για να αναφερθούμε στις μεταβλητές των δεικτών. Όταν αναφερόμαστε σε ένα δείκτη έχουμε πρόσβαση στην τιμή που αυτός δείχνει. Όταν γράφουμε *xPtr = 0 εννοούμε "βάλε τον int 0 στη θέση μνήμης που δείχνει ο xPtr". Αν αντίθετα γράψουμε xPtr = 0 θα πάρουμε ένα λάθος απ το μεταγλωττιστή επειδή xPtr δεν είναι int, είναι *int, που μπορεί να μας δώσει μόνο ένα άλλο *int.

Τελικά χρησιμοποιούμε τον τελεστή & για να βρούμε τη διεύθυνση μιας μεταβλητής. &x επιστρέφει *int (pointer to an int) επειδή x είναι int. Αυτό είναι που μας επιτρέπει να μετατρέψουμε την αρχική μεταβλητή.
&x στη main και xPtr στη zero αναφέρονται στη ίδια θέση μνήμης.

8.2 Συνάρτηση new

Ένας άλλος τρόπος δείκτη είναι να χρησιμοποιήσουμε την ενσωματωμένη συνάρτηση new:

```
func one(xPtr *int) {
    *xPtr = 1
}
func main() {
    xPtr := new(int)
    one(xPtr)
    fmt.Println(*xPtr) // x is 1
}
```

Η new παίρνει ένα τύπο σαν όρισμα, δεσμεύει αρκετή μνήμη για να χωρέσει μια τιμή αυτού του τύπου και επιστρέφει ένα δείκτη.
Σε μερικές γλώσσες προγραμματισμού υπάρχει μια σημαντική διαφορά μεταξύ new και &, έχοντας ιδιαίτερη προσοχή στο να σβήσουμε οτιδήποτε δημιουργήθηκε με τη new. Στη Go δεν είναι το ίδιο, είναι μια "garbage collected" γλώσσα προγραμματισμού που σημαίνει ότι η μνήμη διαγράφετε αυτόματα αν δεν αναφέρετε πια τίποτα σε αυτή.

Οι δείκτες χρησιμοποιούνται σπάνια στους ενσωματωμένους τύπους της Go, αλλά όπως θα δούμε στο επόμενο κεφάλαιο, είναι εξαιρετικά χρήσιμοι όταν συνδυάζονται με δομές.

Προβλήματα:

1. Πώς θα πάρετε τη διεύθυνση μνήμης μια μεταβλητής;

2. Πως θα αντιστοιχήσετε μια τιμή σε ένα δείκτη;

3. Πως θα δημιουργήσετε ένα νέο δείκτη;

```
func square(x *float64) {
    *x = *x * *x
}
func main() {
    x := 1.5
    square(&x)
}
```

4. Ποια είναι η τιμή της x αφού τρέξετε αυτό το πρόγραμμα:

5. Γράψτε ένα πρόγραμμα που να ανταλλάσσει δυο ακεραίους (x := 1; y := 2; swap(&x, &y) πρέπει να σας δώσει x=2 and y=1).

9 Δομές δεδομένων και διεπαφές

Αν και θα ήταν δυνατό για μας να γράφουμε τα προγράμματα μόνο με τη χρήση ενσωματωμένων τύπων δεδομένων της Go, σε κάποιο σημείο θα γίνονταν αρκετά κουραστική. Σκεφτείτε ένα πρόγραμμα που αλληλεπιδρά με τα σχήματα:

```
package main

import ("fmt"; "math")

func distance(x1, y1, x2, y2 float64) float64 {
    a := x2 - x1
    b := y2 - y1
    return math.Sqrt(a*a + b*b)
}

func rectangleArea(x1, y1, x2, y2 float64)
float64 {
    l := distance(x1, y1, x1, y2)
    w := distance(x1, y1, x2, y1)
    return l * w
}

func circleArea(x, y, r float64) float64 {
    return math.Pi * r*r
}
```

Παρακολουθώντας όλες τις συντεταγμένες, είναι δύσκολο να δούμε τι κάνει το πρόγραμμα και πιθανόν να οδηγήσει σε λάθη.

9.1 Δομές Δεδομένων (Structs)

Ένας εύκολος τρόπος για να γίνει αυτό το πρόγραμμα καλύτερο είναι να χρησιμοποιήσετε μια δομή. Μια δομή είναι ένας τύπος που περιέχει ονομασμένα

πεδία .
Για παράδειγμα, θα μπορούσατε να αντιπροσωπεύσετε ένα κύκλο (Circle), όπως αυτό:

```
type Circle struct {
    x float64
    y float64
    r float64
}
```

Η λέξη-κλειδί type εισάγει ένα νέο τύπο. Ακολουθείται από το όνομα του τύπου (Circle), τη λέξη-κλειδί struct για να δείξει ότι έχουμε ορίσει ένα τύπο δομής και μια λίστα με τα πεδία στο εσωτερικό των αγκύλων. Κάθε πεδίο έχει ένα όνομα και έναν τύπο. Όπως και με τις συναρτήσεις έτσι και εδώ μπορούμε να συμπτύξουμε πεδία που έχουν τον ίδιο τύπο:

```
type Circle struct {
    x, y, r float64
}
```

Αρχικοποίηση

Μπορούμε να δημιουργήσουμε ένα παράδειγμα του νέου μας τύπου Circle με ποικίλους τρόπους:

```
var c Circle
```

Όπως και με άλλους τύπους δεδομένων, αυτό θα δημιουργήσει μια τοπική μεταβλητή του Circle η οποία είναι προεπιλεγμένη στο μηδέν. Για μια δομή το μηδέν σημαίνει ότι κάθε ένα από τα πεδία ορίζεται στην αντίστοιχη μηδενική τιμή (0 για int, 0.0 για float, "" για string, nil για δείκτες, ...) Μπορούμε επίσης να χρησιμοποιήσουμε τη νέα συνάρτηση:

```
c := new(Circle)
```

Αυτή διαθέτει μνήμη για όλα τα πεδία, ορίζει καθένα από αυτά στην μηδενική τους τιμή και επιστρέφει έναν δείκτη. (*Circle) Πιο συχνά θέλουμε να δώσουμε μια τιμή

σε κάθε ένα από τα πεδία. Μπορούμε να το κάνουμε αυτό με δύο τρόπους. Όπως εδώ:

```
c := Circle{x: 0, y: 0, r: 5}
```

Ή μπορούμε να αφήσουμε εκτός τα ονόματα των πεδίων, αν γνωρίζουμε την σειρά με την οποία ορίζονται:

```
c := Circle{0, 0, 5}
```

Πεδία

Μπορούμε να έχουμε πρόσβαση σε πεδία χρησιμοποιώντας τον τελεστή . :

```
fmt.Println(c.x, c.y, c.r)
c.x = 10
c.y = 5
```

Ας τροποποιήσουμε τη συνάρτηση circleArea έτσι ώστε να χρησιμοποιεί ένα Circle:

```
func circleArea(c Circle) float64 {
    return math.Pi * c.r*c.r
}
```

Στη main έχουμε:

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(c))
```

Ένα πράγμα που πρέπει να θυμόμαστε είναι ότι τα ορίσματα πάντα αντιγράφονται στη Go. Αν προσπαθήσουμε να τροποποιήσουμε ένα από τα πεδία στο εσωτερικό της συνάρτησης circleArea, δεν θα τροποποιήσει την αρχική μεταβλητή. Γι 'αυτό το λόγο γράφουμε τη συνάρτηση όπως αυτή:

```
func circleArea(c *Circle) float64 {
    return math.Pi * c.r*c.r
}
```

Και αλλάζουμε στη main:

```
c := Circle{0, 0, 5}
fmt.Println(circleArea(&c))
```

9.2 Μέθοδοι

Αν και αυτό είναι καλύτερο από την πρώτη έκδοση του κώδικα, μπορούμε να τον βελτιώσουμε σημαντικά χρησιμοποιώντας έναν ειδικό τύπο συνάρτησης γνωστό και ως method:

```
func (c *Circle) area() float64 {
    return math.Pi * c.r*c.r
}
```

Μεταξύ της λέξης-κλειδί func και του ονόματος της συνάρτησης προσθέτουμε έναν "receiver". Ο δέκτης (receiver) είναι σαν μια παράμετρο – έχει ένα όνομα και ένα τύπο – άλλα όταν δημιουργούμε τη συνάρτηση με αυτό το τρόπο, μας επιτρέπει να καλέσουμε την συνάρτηση χρησιμοποιώντας τον τελεστή . :

```
fmt.Println(c.area())
```

Αυτό είναι πιο εύκολο να διαβαστεί, δεν χρειαζόμαστε πλέον τον τελεστή & (η Go αυτόματα ξέρει να βάλει ένα δείκτη στο circle για αυτή τη μέθοδο) επειδή η συνάρτηση αυτή μπορεί να χρησιμοποιηθεί με το Circle μπορούμε να την ξανα-ονομάσουμε σε συνάρτηση area.
Ας κάνουμε το ίδιο πράγμα για το τετράγωνο(rectangle):

```
type Rectangle struct {
     x1, y1, x2, y2 float64
}

func (r *Rectangle) area() float64 {
     l := distance(r.x1, r.y1, r.x1, r.y2)
     w := distance(r.x1, r.y1, r.x2, r.y1)
     return l * w
}
```

Η main έχει:

```
r := Rectangle{0, 0, 10, 10}
fmt.Println(r.area())
```

Ενσωματωμένοι Τύποι

Τα πεδία μιας δομής αντιπροσωπεύουν συνήθως την σχέση «έχει-μια». Για παράδειγμα, ένας κύκλος (Circle) έχει-μια ακτίνα (Radius). Ας υποθέσουμε ότι είχαμε μια δομή με όνομα Person:

```
type Person struct {
    Name string
}
func (p *Person) Talk() {
    fmt.Println("Hi, my name is", p.Name)
}
```

Και θέλουμε να δημιουργήσουμε μια νέα δομή με όνομα Android. Θα μπορούσαμε να το κάνουμε έτσι:

```
type Android struct {
    Person Person
    Model string
}
```

Αυτό θα μπορούσε να λειτουργήσει, αλλά θα προτιμούσαμε να πούμε ένα Android είναι-ένα Person, και όχι ένα Android έχει-μια Person. Η Go υποστηρίζει σχέσεις όπως αυτή με τη χρήση ενός ενσωματωμένου τύπου. Επίσης γνωστά ως ανώνυμα πεδία (anonymous fields), οι ενσωματωμένοι τύποι μοιάζουν με αυτό:

```
type Android struct {
    Person
    Model string
}
```

Χρησιμοποιούμε τον τύπο (Person) και δεν του δίνουμε ένα όνομα.
Όταν ορίζεται με αυτόν τον τρόπο τη δομή Person μπορεί να προσπελαστεί χρησιμοποιώντας το όνομα του τύπου:

```
a := new(Android)
a.Person.Talk()
```

Αλλά επίσης μπορούμε να καλέσουμε κάθε μέθοδο Person άμεσα στην Android:

```
a := new(Android)
a.Talk()
```

Η σχέση είναι-ένα λειτουργεί με αυτό τον τρόπο διαισθητικά: Οι άνθρωποι (person) μπορούν να μιλήσουν, ένα Android είναι ένας άνθρωπος, ως εκ τούτου το Android μπορεί να μιλήσει.

9.3 Διεπαφές

Μπορεί να είχατε παρατηρήσει ότι ήμασταν σε θέση να ονομάσουμε τη μέθοδο Rectangle area το ίδιο πράγμα με τη μέθοδο Circle area. Αυτό δεν ήταν ατύχημα. Όπως στη πραγματική ζωή έτσι και στον προγραμματισμό οι σχέσεις αυτές είναι σύνηθες φαινόμενο. Η Go έχει έναν τρόπο να κάνει σαφές αυτές τις τυχαίες ομοιότητες μέσω ενός τύπου που είναι γνωστό ως μια διεπαφή. Εδώ είναι ένα παράδειγμα της διεπαφής Shape:

```
type Shape interface {
     area() float64
}
```

Όπως μια δομή έτσι και μια διεπαφή δημιουργείται χρησιμοποιώντας τη λέξη-κλειδί type, που ακολουθείται από το όνομα και τη λέξη-κλειδί field. Αλλά αντί να ορίσουμε τα πεδία, ορίζουμε ένα «σύνολο μεθόδων». Το σύνολο μεθόδων είναι μια λίστα από μεθόδους που ένας τύπος πρέπει να έχει για να «εκτελέσει» τη διεπαφή.
Στην περίπτωση μας, τόσο το Rectangle οσο και το Circle έχουν μεθόδους area που επιστρέφουν float64 έτσι και οι δύο τύποι εκτελούν τη διεπαφή Shape. Από μόνη της αυτή δεν θα ήταν ιδιαίτερα χρήσιμη, αλλά μπορούμε να χρησιμοποιήσουμε τους τύπους της διεπαφής ως ορίσματα σε συναρτήσεις:

```
func totalArea(shapes ...Shape) float64 {
     var area float64
     for _, s := range shapes {
          area += s.area()
     }
     return area
}
```

Θα καλέσουμε τη συνάρτηση όπως εδώ:

```
fmt.Println(totalArea(&c, &r))
```

Οι διεπαφές επίσης μπορούν να χρησιμοποιηθούν ως πεδία:

```
type MultiShape struct {
     shapes []Shape
}
```

Μπορούμε να μετατρέψουμε ακόμα και το ίδιο MultiShape σε ένα Shape, δίνοντάς του μια μέθοδο area:

```
func (m *MultiShape) area() float64 {
     var area float64
     for _, s := range m.shapes {
          area += s.area()
     }
     return area
}
```

Τώρα το MultiShape μπορεί να περιέχει Circle, Rectangle ή ακόμη και άλλα MultiShape.

Προβλήματα:

1. Ποια είναι η διαφορά ανάμεσα σε μια μέθοδο και σε μια συνάρτηση;

2. Γιατί θα χρησιμοποιήσετε ένα ενσωματωμένο ανώνυμο πεδίο αντί ενός κανονικά ονομασμένου πεδίου;

3.Προσθέστε μια νέα μέθοδο στην διεπαφή Shape που να ονομάζεται perimeter και να υπολογίζει την περίμετρο ενός σχήματος. Εφαρμόστε τη μέθοδο για τον Κύκλο και το Τετράγωνο.

10 Ταυτοχρονισμός

Τα μεγάλα προγράμματα συχνά αποτελούνται από πολλά μικρότερα υπο-προγράμματα. Για παράδειγμα, ένας διακομιστής ιστοσελίδων λαμβάνει αιτήσεις που υποβάλλονται από προγράμματα περιήγησης στο διαδύκτιο και παρέχει HTML ιστοσελίδες σαν απάντηση. Κάθε αίτημα λαμβάνεται σαν ένα μικρό πρόγραμμα.

Θα ήταν ιδανικό για προγράμματα όπως αυτά να είναι σε θέση να τρέξουν μικρότερα στοιχεία τους την ίδια χρονική στιγμή (στην περίπτωση του διακομιστή ιστοσελίδων να λαμβάνει πολλαπλές αιτήσεις).Η επίτευξη προόδου σε περισσότερες από μία εργασίες ταυτόχρονα είναι γνωστό ως ταυτοχρονισμός. Η Go έχει μεγάλη υποστήριξη για τον ταυτοχρονισμό χρησιμοποιώντας go-ρουτίνες και κανάλια.

10.1 Go-ρουτίνες

Μια go-ρουτίνα είναι μία συνάρτηση που είναι ικανή να εκτελεί ταυτόχρονα και άλλες συναρτήσεις. Για να δημιουργήσουμε μια go-ρουτίνα χρησιμοποιούμε τη λέξη-κλειδί go που ακολουθείται από μια συνάρτηση επίκλησης:

```
package main

import "fmt"

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
    }
}

func main() {
    go f(0)
    var input string
    fmt.Scanln(&input)
}
```

Το πρόγραμμα αποτελείται από δύο go-ρουτίνες. Η πρώτη go-ρουτίνα είναι έμμεση και είναι η κύρια συνάρτηση από μόνη της. Η δεύτερη go-ρουτίνα δημιουργείται όταν καλούμε την go f (0). Κανονικά όταν επικαλούμαστε μια συνάρτηση, το πρόγραμμά μας θα εκτελέσει όλες τις εντολές μέσα στη συνάρτηση και στη συνέχεια επιστρέφει στην επόμενη γραμμή μετά την επίκληση. Με την go-ρουτίνα

επιστρέφουμε αμέσως στην επόμενη γραμμή και δεν περιμένουμε για την ολοκλήρωση της συνάρτησης. Για το λόγο αυτό το κάλεσμα της συνάρτησης Scanln έχει συμπεριληφθεί, χωρίς αυτή το πρόγραμμα θα έβγαινε προτού δοθεί η ευκαιρία να εμφανίσει όλα τα νούμερα.

Οι go-ρουτίνες είναι ελαφριές και μπορούμε εύκολα να δημιουργήσουμε χιλιάδες από αυτές. Μπορούμε να τροποποιήσουμε το πρόγραμμά μας για να εκτελέσει 10 go-ρουτίνες κάνοντας το παρακάτω:

```
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

Μπορεί να έχετε παρατηρήσει ότι όταν εκτελείτε αυτό το πρόγραμμα φαίνεται να εκτελεί τις go-ρουτίνες σε σειρά και όχι ταυτόχρονα. Ας προσθέσουμε κάποια καθυστέρηση στη συνάρτηση με το time.Sleep και rand.Intn:

```
package main

import (
     "fmt"
     "time"
     "math/rand"
)

func f(n int) {
     for i := 0; i < 10; i++ {
          fmt.Println(n, ":", i)
          amt := time.Duration(rand.Intn(250))
          time.Sleep(time.Millisecond * amt)
     }
}
```

```
func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

Η f εμφανίζει τους αριθμούς από 0 έως 10, με αναμονή μεταξύ 0 και 250 ms μετά από κάθε ένα. Οι go-ρουτίνες πρέπει τώρα να εκτελούνται ταυτόχρονα.

10.2 Κανάλια

Τα κανάλια παρέχουν έναν τρόπο για δύο go-ρουτίνες να επικοινωνούν μεταξύ τους και να συγχρονίζουν την εκτέλεσή τους. Εδώ είναι ένα παράδειγμα προγράμματος χρησιμοποιώντας κανάλια:

```
package main

import (
	"fmt"
	"time"
)

func pinger(c chan string) {
	for i := 0; ; i++ {
		c <- "ping"
	}
}
func printer(c chan string) {
	for {
		msg := <- c
		fmt.Println(msg)
		time.Sleep(time.Second * 1)
	}
}
func main() {
	var c chan string = make(chan string)

	go pinger(c)
	go printer(c)

	var input string
	fmt.Scanln(&input)
}
```

Αυτό το πρόγραμμα θα εκτυπώνει "ping" για πάντα (πατήστε enter για να σταματήσει). Ένας τύπος κανάλι εκπροσωπείται με την λέξη-κλειδί chan που ακολουθείται από τον τύπο των πραγμάτων που έχουν εισαχθεί στο κανάλι (σε αυτή την περίπτωση εισάγαμε string). Ο τελεστής <- (αριστερό βέλος) χρησιμοποιείται για την αποστολή και λήψη μηνυμάτων στο κανάλι. c <- "ping" σημαίνει αποστολή "ping". msg := <- c σημαίνει λήψη ενός μηνύματος και αποθήκευση στο msg. Η fmt

γραμμή θα μπορούσε επίσης να έχει γραφτεί όπως αυτό: fmt.Println(<-c) σε αυτή την περίπτωση θα μπορούσαμε να καταργήσουμε την προηγούμενη γραμμή.

Χρησιμοποιώντας ένα κανάλι σαν αυτό συγχρονίζουμε τις δύο go-ρουτίνες. Όταν ο pinger προσπαθεί να στείλει ένα μήνυμα στο κανάλι, θα περιμένει μέχρι ο printer να είναι έτοιμος να λάβει το μήνυμα. (αυτό είναι γνωστό ως blocking) Ας προσθέσουμε έναν άλλο αποστολέα στο πρόγραμμα και να δούμε τι θα συμβεί.
Προσθέστε αυτή τη συνάρτηση:

```
func ponger(c chan string) {
	for i := 0; ; i++ {
		c <- "pong"
	}
}
```

Τροποποιήστε τη main:

```
func main() {
	var c chan string = make(chan string)

	go pinger(c)
	go ponger(c)
	go printer(c)

	var input string
	fmt.Scanln(&input)
}
```

Το πρόγραμμα τώρα θα εμφανίζει εναλλάξ "ping" και "pong".

Κατεύθυνση καναλιού

Μπορούμε να καθορίσουμε μια κατεύθυνση σε έναν τύπο καναλιού περιορίζοντάς τον έτσι ώστε να κάνει αποστολή ή λήψη. Για παράδειγμα η συνάρτηση pinger μπορεί να αλλάξει σε αυτό:

```
func pinger(c chan<- string)
```

Τώρα το c μπορεί μόνο να αποσταλεί. Προσπαθώντας να λάβει από το c θα οδηγήσει σε ένα σφάλμα μεταγλώττισης. Ομοίως μπορούμε να αλλάξουμε την συνάρτηση printer σε αυτό:

```
func printer(c <-chan string)
```

Ένα κανάλι που δεν έχει αυτούς τους περιορισμούς είναι γνωστό ως διπλής κατεύθυνσης. Ένα κανάλι διπλής κατεύθυνσης μπορεί να περάσει σε μια συνάρτηση η οποία παίρνει κανάλια που μόνο στέλνουν ή μόνο λαμβάνουν, αλλά το αντίστροφο δεν ισχύει.

Select

Η Go έχει μια ειδική εντολή που ονομάζεται select η οποία λειτουργεί σαν την switch αλλά για τα κανάλια:

```
func main() {
	c1 := make(chan string)
	c2 := make(chan string)

	go func() {
		for {
			c1 <- "from 1"
			time.Sleep(time.Second * 2)
		}
	}()
	go func() {
		for {
			c2 <- "from 2"
			time.Sleep(time.Second * 3)
		}
	}()
	go func() {
		for {
			select {
			case msg1 := <- c1:
				fmt.Println(msg1)
			case msg2 := <- c2:
				fmt.Println(msg2)
			}
		}
	}()

	var input string
	fmt.Scanln(&input)
}
```

Αυτό το πρόγραμμα εμφανίζει "from 1" κάθε 2 δευτερόλεπτα και "from 2" κάθε 3 δευτερόλεπτα. Η select παίρνει το πρώτο κανάλι που είναι έτοιμο και λαμβάνει από αυτό (ή στέλνει σε αυτό). Αν περισσότερα από ένα κανάλια είναι έτοιμα τότε επιλέγει

τυχαία ένα από το οποίο θα λαμβάνει. Εάν κανένα από τα κανάλια δεν είναι έτοιμο, τότε η διαδικασία σταματάει μέχρι κάποιο να γίνει διαθέσιμο.

Η εντολή select χρησιμοποιείται συχνά για να εφαρμόσει ένα timeout:

```
select {
case msg1 := <- c1:
     fmt.Println("Message 1", msg1)
case msg2 := <- c2:
     fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
     fmt.Println("timeout")
}
```

Το time.After δημιουργεί ένα κανάλι και μετά τη δοθείσα διάρκεια, θα στείλει τον τρέχον χρόνο σε αυτό. (εμείς δεν ενδιαφερόμαστε για το χρόνο γι 'αυτό δεν τον αποθηκεύουμε σε μια μεταβλητή) Μπορούμε επίσης να καθορίσουμε μια default case:

```
select {
case msg1 := <- c1:
     fmt.Println("Message 1", msg1)
case msg2 := <- c2:
     fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
     fmt.Println("timeout")
default:
     fmt.Println("nothing ready")
}
```

Η default case θα συμβεί αμέσως αν κανένα από τα κανάλια δεν είναι έτοιμα.

Buffered Κανάλια

Είναι επίσης δυνατόν να περάσουμε μια δεύτερη παράμετρο στη συνάρτηση make όταν δημιουργούμε ένα κανάλι:

```
c := make(chan int, 1)
```

Αυτό δημιουργεί ένα buffered κανάλι με χωρητικότητα 1.
Κανονικά τα κανάλια είναι συγχρονισμένα δηλαδή οι δύο πλευρές του καναλιού θα περιμένουν έως ότου η άλλη πλευρά να είναι έτοιμη. Ένα buffered κανάλι είναι ασύγχρονο δηλαδή στην αποστολή ή στη λήψη ενός μηνύματος δεν θα περιμένει, εκτός αν το κανάλι είναι ήδη πλήρες.

Προβλήματα:

1. Πώς μπορείτε να καθορήσετε την κατεύθυνση ενός τύπου καναλιού;

2. Γράψτε τη δική σας συνάρτηση Sleep χρησιμοποιώντας την time.After.

3. Τι είναι το buffered κανάλι; Πώς θα δημιουργήσετε ένα με χωρητικότητα 20;

11 Πακέτα

Η Go σχεδιάστηκε να είναι μια γλώσσα η οποία ενθαρρύνει τις ορθές πρακτικές

τεχνολογίας λογισμικού . Ένα σημαντικό μέρος της υψηλής ποιότητας προγράμματος είναι η επαναχρησιμοποίηση του κώδικα – ενσωματωμένης στην αρχή "Don't Repeat Yourself."

Όπως είδαμε στο κεφάλαιο 7, οι συναρτήσεις είναι το πρώτο επίπεδο στο οποίο ενεργούμε για να κάνουμε επαναχρησιμοποίηση του κώδικα. Η Go επίσης παρέχει ένα άλλο μηχανισμό για την επαναχρησιμοποίηση κώδικα: τα πακέτα. Σχεδόν κάθε πρόγραμμα που έχουμε δει μέχρι στιγμής περιλαμβάνει αυτή τη γραμμή :

```
import "fmt"
```

fmt είναι το όνομα ενός πακέτου που περιλαμβάνει μια σειρά από συναρτήσεις που σχετίζονται με τη μορφοποίηση και την έξοδο στην οθόνη. Φτιάχνοντας κώδικα με αυτό το τρόπο εξυπηρετούμε 3 σκοπούς:

1. Μειώνεται η πιθανότητα να έχουμε επικαλυπτόμενα ονόματα. Αυτό κρατά τα ονόματα των συναρτήσεων μας σύντομα και περιεκτικά.
2. Οργανώνει τον κώδικα έτσι ώστε να είναι ευκολότερο να βρούμε τον κώδικα που θέλουμε να επαναχρησιμοποιήσουμε.
3. Επιταχύνει τον μεταγλωττιστή απαιτώντας μόνο επανα-μεταγλώττιση μικρότερων κομματιών ενός προγράμματος. Παρόλο που χρησιμοποιούμε το πακέτο fmt, δεν πρέπει να το ξανά μεταγλωττίσουμε κάθε φορά που αλλάζουμε το πρόγραμμά μας.

11.1 Δημιουργώντας Πακέτα

Τα πακέτα έχουν πραγματική σημασία μόνο στο πλαίσιο ενός ξεχωριστού προγράμματος που τα χρησιμοποιεί. Χωρίς αυτό το ξεχωριστό πρόγραμμα δε θα υπήρχε τρόπος να χρησιμοποιήσουμε τα πακέτα που δημιουργήσαμε. Ας φτιάξουμε μια εφαρμογή που θα χρησιμοποιεί ένα πακέτο που θα γράψουμε.
Φτιάξτε ένα φάκελο στο ~/Go/src/golangbook με όνομα chapter11. Μέσα σε αυτό το φάκελο φτιάξτε ένα αρχείο ονόματι main.go το οποίο θα περιέχει αυτό:

```
package main

import "fmt"
import "golang-book/chapter11/math"

func main() {
     xs := []float64{1,2,3,4}
     avg := math.Average(xs)
     fmt.Println(avg)
}
```

Τώρα φτιάξτε ένα άλλο φάκελο μέσα στο φάκελο chapter11 με όνομα math. Μέσα σε αυτό το φάκελο δημιουργήστε ένα αρχείο με όνομα math.go που περιέχει αυτό:

```
package math

func Average(xs []float64) float64 {
     total := float64(0)
     for _, x := range xs {
          total += x
     }
     return total / float64(len(xs))
}
```

Χρησιμοποιώντας το τερματικό στο φάκελο math που μόλις δημιουργήσατε, τρέξτε go install. Αυτό θα μεταγλωττίσει το πρόγραμμα math.go και θα δημιουργήσει ένα αρχείο-σύνδεσμο: ~/Go/pkg/os_arch/golang-book/chapter11/math.a.
(όπου os είναι κάτι σαν windows και arch κάτι σαν amd64) Τώρα, επιστρέψτε στο φάκελο chapter11 και τρέξτε go run main.go. Πρέπει να δείτε 2.5. Κάτι που πρέπει να προσέξετε:

1. math είναι το όνομα ενός πακέτου που είναι μέρος της διανομής της
Go, αλλά απ' τη στιγμή που τα πακέτα της Go μπορούν να είναι ιεραρχικά είναι ασφαλές να χρησιμοποιήσουμε το ίδιο όνομα για το πακέτο μας (Το πραγματικό πακέτο math
είναι απλά math, το δικό μας είναι golangbook/chapter11/math).
2. Μόλις εισάγουμε (import) τη βιβλιοθήκη math χρησιμοποιούμε το πλήρες όνομά

της (import "golang-book/chapter11/math"), αλλά μέσα στο αρχείο math.go χρησιμοποιούμε μόνο το τελευταίο μέρος του ονόματος (package math).
3. Επίσης, χρησιμοποιούμε μόνο το σύντομο όνομα math όταν αναφερόμαστε σε συναρτήσεις της βιβλιοθήκης μας. Αν θέλαμε να χρησιμοποιήσουμε και τις δυο βιβλιοθήκες στο ίδιο πρόγραμμα η Go μας επιτρέπει να χρησιμοποιήσουμε ένα ψευδώνυμο:

```
import m "golang-book/chapter11/math"

func main() {
    xs := []float64{1,2,3,4}
    avg := m.Average(xs)
    fmt.Println(avg)
}
```

m είναι το ψευδώνυμο.

4. Ίσως να παρατηρήσατε ότι κάθε συνάρτηση στα πακέτα που είδαμε, ξεκινά με κεφαλαίο γράμμα. Στη Go αν κάτι ξεκινά με κεφαλαίο γράμμα σημαίνει ότι άλλα πακέτα (και προγράμματα) είναι σε θέσει να το δουν. Αν είχαμε ονομάσει τη συνάρτηση average αντί για Average το πρόγραμμα main δε θα ήταν σε θέση να το δει. Είναι μια καλή πρακτική να εκθέτεται μόνο τα τμήματα του πακέτου που θέλουμε να χρησιμοποιήσουν άλλα πακέτα και να κρύβετε όλα τα άλλα.
Αυτό μας επιτρέπει να αλλάξουμε ελεύθερα αυτά τα τμήματα αργότερα χωρίς να ανησυχούμε αν θα χαλάσουμε άλλα προγράμματα και θα κάνει και το πακέτο μας ευκολότερο στη χρήση.
5. Τα ονόματα των πακέτων ταιριάζουν με τους φακέλους που υπάγονται. Υπάρχουν τρόποι γύρω από αυτό, αλλά είναι πολύ πιο εύκολο αν μείνετε σε αυτό το μοτίβο.

11.2 Τεκμηρίωση

Η Go έχει την ιδιότητα να παράγει αυτόματα τεκμηρίωση για τα πακέτα που γράψαμε με παρόμοιο τρόπο με την επίσημη τεκμηρίωση πακέτων. Στο τερματικό τρέξτε αυτή την εντολή:

```
godoc golang-book/chapter11/math Average
```

Πρέπει να δείτε πληροφορίες για τη συνάρτηση που μόλις γράψαμε. Μπορούμε να βελτιώσουμε αυτή τη τεκμηρίωση συμπληρώνοντας ένα σχόλιο πριν τη συνάρτηση:

```
// Finds the average of a series of numbers
func Average(xs []float64) float64 {
```

Αν τρέξατε go install στο φάκελο math, τότε ξανατρέξτε την εντολή godoc, πρέπει να δείτε το σχόλιό μας κάτω απ' τον ορισμό της συνάρτησης. Αυτή η τεκμηρίωση είναι επίσης διαθέσιμη στο δίκτυο τρέχοντας την εντολή:

```
godoc -http=":6060"
```

και εισάγοντας αυτή τη URL στον περιηγητή σας:

```
http://localhost:6060/pkg/
```

Θα πρέπει να μπορείτε να περιηγηθείτε σε όλα τα πακέτα που είναι εγκατεστημένα στο σύστημά σας.

Προβλήματα:

1. Γιατί χρησιμοποιούμε πακέτα;

2. Ποια η διαφορά μεταξύ ενός ονόματος που ξεκινά με κεφαλαίο γράμμα και ενός που δεν ξεκινά; (Average vs average)

3. Τι είναι ένα πακέτο με ψευδώνυμο; Πως δημιουργείτε;

4. Αντιγράψαμε τη συνάρτηση average απ' το κεφάλαιο 7 στο καινούριο μας πακέτο. Φτιάξτε τις συναρτήσεις Min και Max οι οποίες βρίσκουν τις μικρότερες και τις μεγαλύτερες τιμές σε ένα slice με float64.

5. Πως θα τεκμηριώσετε τις συναρτήσεις που δημιουργήσατε στο ερώτημα 3;

12 Έλεγχος (Testing)

Ο προγραμματισμός δεν είναι εύκολος, ακόμα και οι καλύτεροι προγραμματιστές είναι ανίκανοι να γράψουν ένα πρόγραμμα που θα δουλεύει ακριβώς όπως πρέπει κάθε φορά. Επομένως, ένα σημαντικό μέρος της διαδικασίας ανάπτυξης λογισμικού είναι ο έλεγχος . Γράφοντας ελέγχους για τον κώδικά μας είναι ένας καλός τρόπος να διασφαλίσουμε την ποιότητα και να βελτιώσουμε την αξιοπιστία .

Η Go περιλαμβάνει ένα ειδικό πρόγραμμα που κάνει τη γραφή ελέγχων ευκολότερη, έτσι ας δημιουργήσουμε μερικούς ελέγχους για τα πακέτα που φτιάξαμε σε προηγούμενο κεφάλαιο. Στο φάκελο math απ' το κεφάλαιο 11 δημιουργήσαμε ένα νέο αρχείο με όνομα math_test.go που περιέχει αυτό:

```
package math

import "testing"

func TestAverage(t *testing.T) {
     var v float64
     v = Average([]float64{1,2})
     if v != 1.5 {
          t.Error("Expected 1.5, got ", v)
     }
}
```

Τώρα τρέξτε την εντολή:

```
go test
```

Πρέπει να δείτε αυτό:

```
$ go test
PASS
ok      golang-book/chapter11/math      0.032s
```

Η εντολή go test θα ψάξει για κάθε έλεγχο σε κάθε αρχείο στο τρέχων φάκελο και θα τα τρέξει. Οι έλεγχοι ταυτοποιούνται από τις συναρτήσεις που αρχίζουν με τη λέξη

Test και παίρνουν ένα όρισμα του τύπου *testing.T. Στη περίπτωσή μας, απ΄ τη στιγμή που ελέγχουμε την συνάρτηση Average ονομάζουμε τη συνάρτηση ελέγχου TestAverage.

Μόλις έχουμε εγκατεστημένη τη συνάρτηση ελέγχου γράφουμε ελέγχους που χρησιμοποιούν το κώδικα που ελέγχουμε. Στη περίπτωση αυτή ξέρουμε ότι ο μέσος όρος του [1,2] πρέπει να είναι 1.5, αυτό είναι που ελέγχουμε. Είναι πιθανών καλή ιδέα να δοκιμάσουμε πολλούς διαφορετικούς συνδυασμούς αριθμών, έτσι ας αλλάξουμε λίγο το πρόγραμμα ελέγχου:

```
package math

import "testing"

type testpair struct {
     values []float64
     average float64
}

var tests = []testpair{
     { []float64{1,2}, 1.5 },
     { []float64{1,1,1,1,1,1}, 1 },
     { []float64{-1,1}, 0 },
}

func TestAverage(t *testing.T) {
     for _, pair := range tests {
          v := Average(pair.values)
          if v != pair.average {
               t.Error(
                    "For", pair.values,
                    "expected", pair.average,
                    "got", v,
               )
          }
     }
}
```

Αυτός είναι ένας πολύ κοινός τρόπος να εγκαθιστούμε τους ελέγχους (πολλά παραδείγματα μπορούν να βρεθούν στο πηγιαίο κώδικα των πακέτων που

περιλαμβάνονται στη Go). Δημιουργούμε μια struct να παραστήσουμε τις εισόδους και εξόδους της συνάρτησης. Ύστερα, δημιουργούμε μια λίστα αυτών των δομών (pairs). Μετά, μπαίνουμε στο βρόγχο καθεμίας και τρέχουμε τη συνάρτηση.

Προβλήματα:

1. Η γραφή μιας καλής ακολουθίας ελέγχου δεν είναι πάντα εύκολο, αλλά η διαδικασία γραφής συχνά αποκαλύπτει περισσότερα από αυτά που συνειδητοποιήσατε στην αρχή για ένα πρόβλημα. Για παράδειγμα, με την συνάρτηση Average τι θα συμβεί αν εισάγετε μια κενή λίστα ([]float64{}); Πώς μπορούμε να τροποποιήσουμε τη συνάρτηση ώστε να επιστρέφει 0 σε αυτή τη περίπτωση;

2. Γράψτε μια σειρά από ελέγχους για τις συναρτήσεις Min και Max που γράψατε στο προηγούμενο κεφάλαιο.

13 Τα πακέτα του πυρήνα

Αντί να γράφετε τα πάντα απ' την αρχή , ο πραγματικός κόσμος του προγραμματισμού εξαρτάται απ' την ικανότητά μας να συνδεόμαστε με τις υπάρχουσες βιβλιοθήκες. Σε αυτό το κεφάλαιο θα ρίξουμε μια ματιά σε μερικά απ' τα πιο συχνά χρησιμοποιούμενα πακέτα που εμπεριέχονται στη Go.

Πρώτα, μια προειδοποίηση: παρόλο που μερικές από αυτές τις βιβλιοθήκες είναι αρκετά προφανείς (ή έχουν επεξηγηθεί σε προηγούμενα κεφάλαια), πολλές απ' τις βιβλιοθήκες που περιλαμβάνονται στη Go απαιτούν εξειδικευμένες γνώσεις σε συγκεκριμένους τομείς (για παράδειγμα: κρυπτογραφία). Είναι πέρα απ' το σκοπό αυτού του βιβλίου η επεξήγηση τέτοιων τεχνολογιών.

13.1 Αλφαριθμητικές ακολουθίες (Strings)

Η Go περιλαμβάνει ένα μεγάλο αριθμό από συναρτήσεις για να εργαστείτε με strings στο πακέτο strings:

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	fmt.Println(
		// true
		strings.Contains("test", "es"),

		// 2
		strings.Count("test", "t"),

		// true
		strings.HasPrefix("test", "te"),

		// true
		strings.HasSuffix("test", "st"),

		// 1
		strings.Index("test", "e"),

		// "a-b"
		strings.Join([]string{"a","b"}, "-"),

		// == "aaaaa"
		strings.Repeat("a", 5),

		// "bbaa"
		strings.Replace("aaaa", "a", "b", 2),

		// []string{"a","b","c","d","e"}
		strings.Split("a-b-c-d-e", "-"),
```

```
            // "test"
            strings.ToLower("TEST"),

            // "TEST"
            strings.ToUpper("test"),

        )
}
```

Μερικές φορές χρειαζόμαστε να εργαστούμε με τα strings λαμβάνοντάς τα ως δυαδικά δεδομένα. Για να μετατρέψουμε ένα string σε ένα slice από bytes (και αντίστροφα) κάντε αυτό:

```
arr := []byte("test")
str := string([]byte{'t','e','s','t'})
```

13.2 Είσοδος / Έξοδος

Πριν κοιτάξουμε στα αρχεία χρειάζεται να κατανοήσουμε το πακέτο io. Το πακέτο io αποτελείται από μερικές συναρτήσεις, αλλά κυρίως διεπαφές που χρησιμοποιούνται σε άλλα πακέτα. Οι δύο κύριες διεπαφές είναι οι Reader και Writer . Οι Reader υποστηρίζουν ανάγνωση μέσω της μεθόδου Read. Οι Writer υποστιρίζουν εγγραφή μέσω της μεθόδου Write. Πολλές συναρτήσεις στη Go παίρνουν τους Reader ή Writer ως ορίσματα. Για παράδειγμα το πακέτο io έχει μια συνάρτηση Copy που αντιγράφει δεδομένα από ένα Reader σε ένα Writer:

```
func Copy(dst Writer, src Reader) (written
int64, err error)
```

Για να διαβάσετε ή να γράψετε σε ένα []byte ή σε ένα string μπορείτε να χρησιμοποιήσετε τη δομή Buffer που βρίσκετε στο πακέτο bytes:

```
var buf bytes.Buffer
buf.Write([]byte("test"))
```

Ένας Buffer δε χρειάζετε να αρχικοποιηθεί και υποστηρίζει και τις δυο διεπαφές Reader και Writer. Μπορείτε να τη μετατρέψετε σε []byte καλώντας buf.Bytes(). Αν χρειάζεστε μόνο να διαβάσετε ένα string μπορείτε επίσης να χρησιμοποιήσετε τη συνάρτηση strings.NewReader που είναι πιο αποτελεσματική απ' το να χρησιμοποιήσετε ένα buffer.

13.3 Αρχεία & Φάκελοι

Για να ανοίξετε ένα αρχείο στη Go χρησιμοποιείστε τη συνάρτηση Open απ' το πακέτο os. Εδώ είναι ένα παράδειγμα του πως να διαβάσετε τα περιεχόμενα ενός αρχείου
και να τα εμφανίσετε στο τερματικό:

```
package main

import (
	"fmt"
	"os"
)

func main() {
	file, err := os.Open("test.txt")
	if err != nil {
		// handle the error here
		return
	}
	defer file.Close()

	// get the file size
	stat, err := file.Stat()
	if err != nil {
		return
	}


	// read the file
	bs := make([]byte, stat.Size())
	_, err = file.Read(bs)
	if err != nil {
		return
	}

	str := string(bs)
	fmt.Println(str)
}
```

Χρησιμοποιούμε defer file.Close() αμέσως μετά το άνοιγμα του αρχείου για να σιγουρευτούμε ότι το αρχείο κλείνει μετά την ολοκλήρωση της συνάρτησης. Η ανάγνωση αρχείων είναι πολύ κοινή, έτσι υπάρχει ένας σύντομος τρόπος για να το κάνουμε:

```
package main

import (
	"fmt"
	"io/ioutil"
)

func main() {
	bs, err := ioutil.ReadFile("test.txt")
	if err != nil {
		return
	}
	str := string(bs)
	fmt.Println(str)
}
```

Εδώ φαίνεται πως μπορούμε να δημιουργήσουμε ένα αρχείο:

```
package main

import (
	"os"
)

func main() {
	file, err := os.Create("test.txt")
	if err != nil {
		// handle the error here
		return
	}
	defer file.Close()

	file.WriteString("test")
}
```

Για να πάρουμε τα περιεχόμενα ενός καταλόγου χρησιμοποιούμε την ίδια συνάρτηση os.Open αλλά δώστε τις ένα προορισμό καταλόγου αντί ένα όνομα αρχείου.
Έτσι, καλούμε τη μέθοδο Readdir:

```
package main

import (
	"fmt"
	"os"
)

func main() {
	dir, err := os.Open(".")
	if err != nil {
		return
	}
	defer dir.Close()

	fileInfos, err := dir.Readdir(-1)
	if err != nil {
		return
	}
	for _, fi := range fileInfos {
		fmt.Println(fi.Name())
	}
}
```

Συχνά, θέλουμε να περιηγηθούμε σε ένα φάκελο (να διαβάσουμε τα περιεχόμενα ενός φακέλου, όλους τους υπο-φακέλους, όλους τους υπο-υποφακέλους,...). Για να το κάνουμε ευκολότερο υπάρχει μια συνάρτηση Walk που παρέχετε στο πακέτο path/filepath:

```
package main

import (
     "fmt"
     "os"
     "path/filepath"
)

func main() {
     filepath.Walk(".", func(path string, info
os.FileInfo, err error) error {
          fmt.Println(path)
          return nil
     })
}
```

Η συνάρτηση Walk καλείται για κάθε αρχείο και φάκελο στο φάκελο root. (σε αυτή τη περίπτωση .)

13.4 Λάθη

Η Go έχει ένα ενσωματωμένο τύπο για λάθη που ήδη έχουμε δει (τον τύπο error). Μπορούμε να δημιουργήσουμε τα δικά μας λάθη χρησιμοποιώντας τη συνάρτηση New στο πακέτο errors:

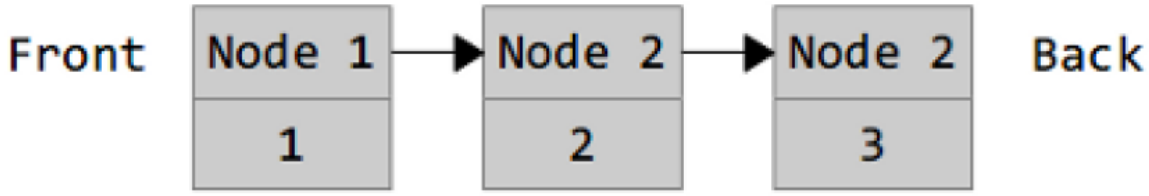
```
package main

import "errors"

func main() {
     err := errors.New("error message")
}
```

13.5 Περιεχόμενα (Containers) & Ταξινόμηση (Sort)

Επιπρόσθετα, με τις λίστες και τους χάρτες η Go έχει πολλές διαφορετικές συλλογές διαθέσιμες υπό το πακέτο container.
Θα ρίξουμε μια ματιά στο πακέτο container/list σαν παράδειγμα.

Λίστα (List)

Το πακέτο container/list υλοποιεί μια διπλά συνδεδεμένη λίστα.
Μια συνδεδεμένη λίστα είναι ένας τύπος μιας δομής δεδομένων που μοιάζει κάπως έτσι:



Κάθε κόμβος της λίστας περιέχει μια τιμή (σε αυτή τη περίπτωση 1, 2, ή 3) και ένα δείκτη στον επόμενο κόμβο. Απ' τη στιγμή που αυτή είναι μια διπλά συνδεδεμένη λίστα, κάθε κόμβος θα έχει επίσης δείκτες προς τον προηγούμενο κόμβο. Αυτή η λίστα μπορεί να δημιουργήθηκε από αυτό το πρόγραμμα:

```
package main

import ("fmt" ; "container/list")

func main() {
     var x list.List
     x.PushBack(1)
     x.PushBack(2)
     x.PushBack(3)

     for e := x.Front(); e != nil; e=e.Next() {
          fmt.Println(e.Value.(int))
     }
}
```

Η τιμή μηδέν για μια List είναι μια κενή λίστα (μια *List μπορεί επίσης να δημιουργηθεί χρησιμοποιώντας list.New). Οι τιμές επισημαίνονται στη λίστα χρησιμοποιώντας PushBack. Ανατρέχουμε σε κάθε κομμάτι της λίστας παίρνοντας το πρώτο στοιχείο και ακολουθώντας όλους τους συνδέσμους μέχρι να φτάσουμε στο μηδενικό δείκτη (nil).

Ταξινόμηση (Sort)

Το πακέτο ταξινόμησης περιέχει συναρτήσεις για την αυθαίρετη ταξινόμηση δεδομένων. Υπάρχουν πολλές προκαθορισμένες συναρτήσεις (για slices ή ints και floats). Εδώ είναι ένα παράδειγμα του πως να ταξινομήσετε τα δικά σας δεδομένα:

```
package main

import ("fmt" ; "sort")

type Person struct {
     Name string
     Age int
}

type ByName []Person

func (this ByName) Len() int {
     return len(this)
}
func (this ByName) Less(i, j int) bool {
     return this[i].Name < this[j].Name
}
func (this ByName) Swap(i, j int) {
     this[i], this[j] = this[j], this[i]
}

func main() {
     kids := []Person{
          {"Jill",9},
          {"Jack",10},
     }
     sort.Sort(ByName(kids))
     fmt.Println(kids)
}
```

Η συνάρτηση Sort στη ταξινόμηση παίρνει sort.Interface και την ταξινομεί. Η sort.Inerface απαιτεί 3 μεθόδους: Len, Less και Swap. Για να καθορίσουμε τη δική μας ταξινόμηση δημιουργούμε ένα νέο τύπο (ByName) και τον κάνουμε ισοδύναμο σε ένα slice με το τι θέλουμε να ταξινομήσουμε.
Μπορούμε στη συνέχεια να καθορίσουμε τις 3 μεθόδους.
Η ταξινόμηση της λίστας μας από ανθρώπους (Person) είναι εύκολη όπως η εισαγωγή της στο καινούριο μας τύπο.
Μπορούμε επίσης να την ταξινομήσουμε κατά ηλικία κάνοντας αυτό:

```
type ByAge []Person
func (this ByAge) Len() int {
    return len(this)
}
func (this ByAge) Less(i, j int) bool {
    return this[i].Age < this[j].Age
}
func (this ByAge) Swap(i, j int) {
    this[i], this[j] = this[j], this[i]
}
```

13.6 Κατατεμαχισμός (Hashes) & Κρυπτογραφία (Cryptography)

Μια συνάρτηση κατατεμαχισμού παίρνει μια ομάδα από δεδομένα και τα μειώνει σε μικρότερα σταθερού μεγέθους. Οι συναρτήσεις αυτές συχνά χρησιμοποιούνται στον προγραμματισμό για την αναζήτηση δεδομένων και για την εύκολη ανίχνευση αλλαγών. Οι συναρτήσεις κατατεμαχισμού στη Go είναι χωρισμένες σε δυο κατηγορίες: κρυπτογραφικές και μη-κρυπτογραφικές. Οι μη-κρυπτογραφικές μπορούν να βρεθούν κάτω από το πακέτο hash και περιλαμβάνουν adler32, crc32, crc64 και fnv. Εδώ είναι ένα παράδειγμα χρησιμοποιώντας crc32:

```
package main

import (
     "fmt"
     "hash/crc32"
)

func main() {
     h := crc32.NewIEEE()
     h.Write([]byte("test"))
     v := h.Sum32()
     fmt.Println(v)
}
```

Το crc32 υλοποιεί τη διεπαφή Writer έτσι ώστε να μπορούμε να γράψουμε bytes σε αυτό όπως και κάθε άλλος Writer. Απ' τη στιγμή που έχουμε γράψει τα πάντα, θέλουμε να καλέσουμε Sum32() για να επιστρέψει uint32. Μια κοινή χρήση του crc32 είναι η σύγκριση δύο αρχείων. Αν η τιμή του Sum32 είναι ίδια και για τα δυο αρχεία είναι πολύ πιθανό (αν και δεν είναι 100% σίγουρο) ότι τα αρχεία είναι ίδια.
Αν οι τιμές είναι διαφορετικές τότε τα αρχεία δεν είναι απόλυτα ίδια:

```
package main

import (
	"fmt"
	"hash/crc32"
	"io/ioutil"
)

func getHash(filename string) (uint32, error) {

	bs, err := ioutil.ReadFile("test1.txt")
	if err != nil {
		return 0, err
	}
	h := crc32.NewIEEE()
	h.Write(bs)
	return h.Sum32(), nil
}

func main() {
	h1, err := getHash("test1.txt")
	if err != nil {
		return
	}
	h2, err := getHash("test2.txt")
	if err != nil {
		return
	}
	fmt.Println(h1, h2, h1 == h2)
}
```

Οι κρυπτογραφικές συναρτήσεις κατατεμαχισμού είναι παρόμοιες με τις μη-κρυπτογραφικές, αλλά έχουν την πρόσθετη ιδιότητα να είναι δύσκολο να αντιστραφούν. Δίνοντας στον κρυπτογραφικό κατατεμαχισμό ένα σύνολο δεδομένων,

είναι εξαιρετικά δύσκολο να καθορίσει τι δημιούργησε τον κατατεμαχισμό. Αυτοί οι κατατεμαχισμοί συχνά χρησιμοποιούνται σε εφαρμογές ασφαλείας.
Μία κοινή κρυπτογραφική συνάρτηση κατατεμαχισμού είναι γνωστή ως SHA-1. Εδώ φαίνετε πώς χρησιμοποιείτε:

```
package main

import (
     "fmt"
     "crypto/sha1"
)

func main() {
     h := sha1.New()
     h.Write([]byte("test"))
     bs := h.Sum([]byte{})
     fmt.Println(bs)
}
```

Αυτό το παράδειγμα είναι πολύ κοινό με το crc32, επειδή και τα δυο ορίζουν τη διεπαφή hash.Hash. Η κύρια διαφορά είναι ότι ενώ η crc32 υπολογίζει κατατεμαχισμό των 32 bit, η sha1 υπολογίζει κατατεμαχισμό των 160 bit. Δεν υπάρχει ατόφιος τύπος για να αναπαραστήσει έναν αριθμό 160 bit, έτσι χρησιμοποιούμε ένα slice από 20 bytes.

13.7 Διακομιστές (Servers)

Η κατασκευή διακομιστών δικτύου με τη Go είναι πολλή εύκολη. Πρώτα, θα δούμε πώς να δημιουργούμε ένα TCP διακομιστή:

```
package main

import (
	"encoding/gob"
	"fmt"
	"net"
)

func server() {
	// listen on a port
	ln, err := net.Listen("tcp", ":9999")
	if err != nil {
		fmt.Println(err)
		return
	}
	for {
		// accept a connection
		c, err := ln.Accept()
		if err != nil {
			fmt.Println(err)
			continue
		}
		// handle the connection
		go handleServerConnection(c)
	}
}
```

```
func handleServerConnection(c net.Conn) {
	// receive the message
	var msg string
	err := gob.NewDecoder(c).Decode(&msg)
	if err != nil {
		fmt.Println(err)
	} else {
		fmt.Println("Received", msg)
	}

	c.Close()
}

func client() {
	// connect to the server
	c, err := net.Dial("tcp", "127.0.0.1:9999")
	if err != nil {
		fmt.Println(err)
		return
	}

	// send the message
	msg := "Hello World"
	fmt.Println("Sending", msg)
	err = gob.NewEncoder(c).Encode(msg)
	if err != nil {
		fmt.Println(err)
	}

	c.Close()
}
```

```
func main() {
	go server()
	go client()

	var input string
	fmt.Scanln(&input)
}
```

Αυτό το παράδειγμα χρησιμοποιεί το πακέτο encoding/gob το οποίο κάνει εύκολη την κωδικοποίηση των τιμών της Go έτσι ώστε άλλα προγράμματά της (ή το ίδιο Go πρόγραμμα σε αυτή τη περίπτωση) να μπορούν να τα διαβάσουν.
Επιπλέον κωδικοποιήσεις είναι διαθέσιμες στα πακέτα encoding (όπως encoding/json) όπως επίσης στα πακέτα 3rd party. (για παράδειγμα μπορούμε να χρησιμοποιήσουμε labix.org/v2/mgo/bson για bson υποστήριξη)

HTTP

Οι HTTP διακομιστές είναι ακόμα πιο εύκολοι στην εγκατάσταση και χρήση:

```
package main

import ("net/http" ; "io")

func hello(res http.ResponseWriter, req
*http.Request) {
	res.Header().Set(
		"Content-Type",
		"text/html",
	)
	io.WriteString(
		res,
		`<doctype html>
<html>
	<head>
		<title>Hello World</title>
	</head>
	<body>
		Hello World!
	</body>
</html>`,
	)
}

func main() {
	http.HandleFunc("/hello", hello)
	http.ListenAndServe(":9000", nil)
}
```

Η HandleFunc χειρίζεται μια URL διεύθυνση (/hello) καλώντας τη δοθείσα συνάρτηση. Μπορούμε επίσης να χειριστούμε στατικά αρχεία χρησιμοποιώντας FileServer:

```
http.Handle(
     "/assets/",
     http.StripPrefix(
          "/assets/",
          http.FileServer(http.Dir("assets")),
     ),
)
```

RPC

Η net/rpc (κλήση απομακρυσμένης διαδικασίας) και τα net/rpc/jsonrpc πακέτα παρέχουν ένα εύκολο τρόπο έκθεσης μεθόδων ώστε να μπορούν να κληθούν από ένα δίκτυο (και όχι μόνο στο πρόγραμμα λειτουργίας τους).

```
package main

import (
	"fmt"
	"net"
	"net/rpc"
)

type Server struct {}
func (this *Server) Negate(i int64, reply
*int64) error {
	*reply = -i
	return nil
}

func server() {
	rpc.Register(new(Server))
	ln, err := net.Listen("tcp", ":9999")
	if err != nil {
		fmt.Println(err)
		return
	}
	for {
		c, err := ln.Accept()
		if err != nil {
			continue
		}
		go rpc.ServeConn(c)
	}
}
```

```
func client() {
	c, err := rpc.Dial("tcp", "127.0.0.1:9999")
	if err != nil {
		fmt.Println(err)
		return
	}
	var result int64
	err = c.Call("Server.Negate", int64(999),
&result)
	if err != nil {
		fmt.Println(err)
	} else {
		fmt.Println("Server.Negate(999) =",
result)
	}
}

func main() {
	go server()
	go client()

	var input string
	fmt.Scanln(&input)
}
```

Αυτό το πρόγραμμα είναι παρόμοιο με το παράδειγμα TCP, εκτός του ότι τώρα δημιουργήσαμε ένα αντικείμενο για να κρατήσουμε όλες τις μεθόδους που χρειάζεται να εκθέσουμε και καλέσαμε την μέθοδο Negate απ' τον πελάτη (client). Δείτε την τεκμηρίωση στο net/rpc για περισσότερες πληροφορίες.

13.8 Συντακτική ανάλυση των ορισμάτων της γραμμής εντολών

Όταν επικαλούμε μία εντολή στο τερματικό, είναι πιθανό να περάσουμε τα ορίσματα αυτής της εντολής. Το έχουμε δει με την εντολή go:

```
go run myfile.go
```

το run και myfile.go είναι ορίσματα. Μπορούμε επίσης να περάσουμε και flags σε μια εντολή:

```
go run -v myfile.go
```

Το πακέτο flag μας επιτρέπει να αναλύσουμε τα ορίσματα και τα flags που στέλνουμε στο πρόγραμμά μας. Εδώ είναι ένα παράδειγμα που δημιουργεί ένα αριθμό μεταξύ 0 και 6. Μπορούμε να αλλάξουμε τη μέγιστη τιμή στέλνοντας ένα flag (-max=100) στο πρόγραμμα:

```
package main

import ("fmt";"flag";"math/rand")

func main() {
    // Define flags
    maxp := flag.Int("max", 6, "the max value")
    // Parse
    flag.Parse()
    // Generate a number between 0 and max
    fmt.Println(rand.Intn(*maxp))
}
```

Κάθε επιπλέον μη-flag όρισμα μπορεί να ανακτηθεί με flag.Args() το οποίο επιστρέφει ένα []string.

13.9 Αρχέγονος συγχρονισμός (synchronization primitives)

Ο προτεινόμενος τρόπος για να χειριστείτε τον ταυτοχρονισμό και τον συγχρονισμό στη Go είναι μέσω των Go-ρουτινών και καναλιών που συζητήσαμε στο κεφάλαιο 10. Ωστόσο, η Go παρέχει περισσότερες ρουτίνες πολλαπλών νημάτων στα πακέτα sync και sync/atomic.

Mutexes

Ένα mutex (mutal exclusive lock) κλειδώνει ένα τμήμα του κώδικα σε ένα μονό νήμα κάθε στιγμή και χρησιμοποιείτε για να προστατέψει κοινόχρηστους πόρους από μη-ατομικές λειτουργίες. Εδώ είναι ένα παράδειγμα του mutex:

```
package main

import (
	"fmt"
	"sync"
	"time"
)
func main() {
	m := new(sync.Mutex)

	for i := 0; i < 10; i++ {
		go func(i int) {
			m.Lock()
			fmt.Println(i, "start")
			time.Sleep(time.Second)
			fmt.Println(i, "end")
			m.Unlock()
		}(i)
	}

	var input string
	fmt.Scanln(&input)
}
```

Όταν το mutex (m) είναι κλειδωμένο ότι άλλο επιχειρήσει να κλειδώσει θα μπλοκάρει μέχρι να ξεκλειδωθεί. Μεγάλη προσοχή θα πρέπει να ληφθεί υπ όψιν όταν χρησιμοποιούμε mutexes ή το synchronization primitives που παρέχετε στο πακέτο sync/atomic.
Ο κλασικός προγραμματισμός πολλαπλών νημάτων είναι δύσκολος,

Είναι εύκολο να κάνουμε λάθη και αυτά τα λάθη είναι δύσκολο να βρεθούν, απ' τη στιγμή που μπορεί να εξαρτώνται από ένα συγκεκριμένο, σχετικά σπάνιο και δύσκολο να αναπαραχθεί ένα σύνολο από περιστάσεις. Μία απ' τις μεγαλύτερες δυνατότητες της Go είναι ότι τα χαρακτηριστικά του ταυτοχρονισμού που παρέχει είναι πολύ ευκολότερα να κατανοηθούν και να χρησιμοποιηθούν κατάλληλα από ότι τα νήματα και τα κλειδώματα.

14 Επόμενα βήματα

Τώρα έχουμε όλες της πληροφορίες που χρειαζόμαστε για να γράψουμε τα περισσότερα προγράμματα στη Go. Αλλά θα ήταν επικίνδυνο να πούμε ότι είμαστε ικανοί προγραμματιστές. Ο προγραμματισμός είναι τόσο τέχνη όσο του απλά να έχουμε γνώση. Αυτό το κεφάλαιο θα σας δώσει μερικές προτάσεις σχετικά με τον καλύτερο τρόπο να κυριαρχήσετε στην τέχνη του προγραμματισμού .

14.1 Μελετήστε τους καλύτερους

Ένα μέρος του να γίνετε καλός καλλιτέχνης ή συγγραφέας είναι να μελετάτε τη δουλειά των καλύτερων. Δεν υπάρχει διαφορά στον προγραμματισμό. Ένας απ' τους καλύτερους τρόπους να γίνετε ένας ικανός προγραμματιστής είναι να μελετάτε πηγιαίους κώδικες γραμμένους από άλλους. Η Go είναι κατάλληλη για αυτό επειδή ο πηγιαίος κώδικας είναι ελεύθερα διαθέσιμος.

Για παράδειγμα ας ρίξουμε μια ματιά στο πηγιαίο κώδικα της βιβλιοθήκης io/ioutil που είναι διαθέσιμος στο: http://golang.org/src/pkg/io/ioutil/ioutil.go

Διαβάστε το κώδικα αργά και προσεκτικά. Προσπαθήστε να καταλάβετε κάθε γραμμή και διαβάστε τα περιεχόμενα σχόλια. Για παράδειγμα στη μέθοδο ReadFile υπάρχει ένα σχόλιο που λέει:

```
// It's a good but not certain bet that FileInfo
// will tell us exactly how much to read, so
// let's try it but be prepared for the answer
// to be wrong.
```

Αυτή η μέθοδος πιθανόν να ξεκινάει απλούστερη απ' ότι καταλήγει, έτσι είναι ένα καλό παράδειγμα του πως τα προγράμματα μπορούν να εξελίσσονται μετά τον έλεγχο και γιατί είναι σημαντικό να περιέχουμε σχόλια στις αλλαγές
Ολόκληρος ο πηγιαίος κώδικας για όλα τα πακέτα είναι διαθέσιμα στο:

http://golang.org/src/pkg/

14.2 Φτιάξτε κάτι

Ένας απ' τους καλύτερους τρόπους για να ακονίσετε της ικανότητές σας είναι να εξασκηθείτε στους κώδικες. Υπάρχουν πολλοί τρόποι για να το κάνετε: Μπορείτε να εργαστείτε σε προβλήματα προγραμματισμού που υπάρχουν σε σελίδες όπως: http://projecteuler.net/ ή δοκιμάζοντας μόνοι σας σε ένα μεγαλύτερο πρόγραμμα. Μπορείτε επίσης να δοκιμάσετε να φτιάξετε ένα δικομιστή ιστοσελίδων (web server) ή να γράψετε ένα απλό παιχνίδι.

14.3 Συνεργαστείτε

Τα περισσότερα προγράμματα λογισμικού φτιάχνονται από ομάδες, επομένως το να μάθετε να δουλεύετε σε μια ομάδα είναι κρίσιμο, αν μπορείτε, βρείτε ένα φίλο -ίσως ένα συμμαθητή- και συνεργαστείτε σε ένα πρόγραμμα. Μάθετε πως να χωρίζετε ένα πρόγραμμα σε κομμάτια έτσι ώστε να μπορείτε να εργαστείτε και οι δυο ταυτόχρονα.

Μια άλλη επιλογή είναι να εργαστείτε σε ένα ανοιχτού κώδικα πρόγραμμα. Βρείτε μια βιβλιοθήκη ελεύθερης διανομής και γράψτε κώδικα (ίσως να φτιάξετε ένα bug), και υποβάλετε το στο συντηρητή . Η Go έχει μια αυξανόμενη κοινότητα που μπορείτε να επικοινωνήσετε μέσω mail στο (http://groups.google.com/group/golang-nuts).

15 Πρόγραμμα Μη-Γραμμικής Διάχυσης

Να γίνει πρόγραμμα που να δημιουργεί ένα πίνακα πραγματικών αριθμών με δύο δείκτες: T[i][j] όπου i -χρόνος και j –διάστημα και να υπολογίζει την ακόλουθη αναδρομική εξίσωση της μη-γραμμικής διάχυσης:

$$T[i+1][j] = T[i][j] + 2\frac{t}{2\,s^2}\,T[i][j]^{\,p(T[i][j+1] + T[i][j-1] - 2T[i][j])}$$

Τα t, s, p $\in \mathbb{R}$,να δίνονται από τον χρήστη.

Κώδικας Προγράμματος

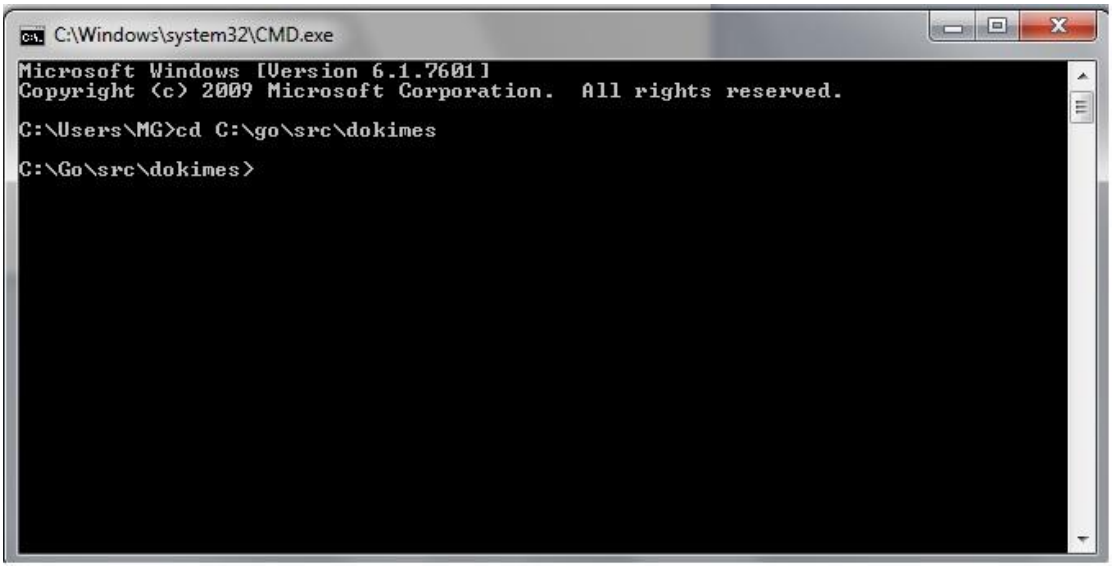
```
package main
import (                    //Δήλωση των πακέτων  που περιέχουν
"fmt"                       //τις βιβλιοθήκες fmt, math.
"math"
)

const (       //Δήλωση σταθερών
N=100         //Ορίζεται το μέγεθος του πίνακα T όπου οι τιμές που παίρνουν τα N, M
M=100 )       //αντιστοιχούν στις γραμμές και στήλες.

var Time, Space int              //Δήλωση των μεταβλητών του προγράμματος
var t, s, p float64
var T[N][M] float64


func main(){
                                 //Με την εντολή fmt.Println εκτυπώνεται στο
fmt.Println ("\nDwste ta t, p, s:")   //τερματικό το μήνυμα:  Dwste ta t, p, s:
fmt.Scanf ("%f %f %f" ,&t, &p, &s)    //Με την εντολή fmt.Scanf καταχωρούνται οι
//τιμές
elenxos(t, p, s)                 //Καλείται η συνάρτηση ελέγχου

}
```

```
//Η συνάρτηση αυτή ελέγχει αν ισχύει η ανισότητα t *p / s^2 < 1. Αν είναι αληθής
//ζητάει από τον χρήστη το χρόνο και το διάστημα και καλεί την συνάρτηση
//Ipologismos_Pinaka αλλιώς  ξαναζητάει τα στοιχεία και  ξανά-ελέγχει. Αφού
//τελειώσει ο υπολογισμός του πίνακα καλείται η συνάρτηση Ektipwsi_Pinaka.

func elenxos (t,p,s float64) {
        if ((t*p)/math.Pow(s,2))<1{       //Με την if γίνεται ο έλεγχος της συνθήκης

                fmt.Println ("\nDwste to xrono se deuterolepta:")
                fmt.Scanf ("%d", &Time)
                fmt.Scanf ("%d", &Time)//???
                fmt.Println ("\nDwste to diastima se ekatosta:")
                fmt.Scanf ("%d", &Space)
                fmt.Scanf ("%d", &Space)//???
                Ipologismos_Pinaka(Time, Space,t, p, s)

        }else{
fmt.Println ("\nTa t, p, s pou dwsate den epalitheuoun tin eksiswsi t * p /   s^2 < 1 !
\nKsanadwste ta t, p, s:")
                fmt.Scanf ("%f %f %f", &t, &p, &s)
                fmt.Scanf ("%f %f %f", &t, &p, &s) //???
                elenxos(t, p, s)              //Καλείται η συνάρτηση ελέγχου
                }

Ektipwsi_Pinaka(Time, Space, T)       //Καλείται η συνάρτηση εκτύπωσης

}

//Η συνάρτηση αυτή ζητάει από τον χρήστη τα στοιχεία για την πρώτη γραμμή του
//πίνακα και μόλις τελειώσει η καταχώρηση των στοιχείων υπολογίζει την εξίσωση
//και καταχωρεί τις τιμές στο πίνακα.

func Ipologismos_Pinaka (time,space int, t, p, s float64) {
        for i:=0; i<time; i++{
for j:=1; j<space-1;j++{   //Το j=1για να αφήσει την πρώτη στήλη με //μηδενικά και
το space-1 για να αφήσει την //τελευταία στήλη με μηδενικά

                        if i==0{       //Για να δοθούν οι τιμές της πρώτης γραμμής του
//πίνακα
                                fmt.Printf("\nDwste tin timi tou stoixeiou T[0][%d] tou
pinaka:\n",j)
                                fmt.Scanf("%f",&T[i][j])
                                fmt.Scanf("%f",&T[i][j]) //Χρησιμοποιείτε λόγο σφάλμα
```

```
                                                        //μεταγλωττιστή
if j==space-2{       // Για να υπολογιστεί η εξίσωση αφού //πρώτα τελειώσει η
καταχώρηση των
//στοιχείων της πρώτης γραμμής.


for j=1; j<space-1; j++{

        T[i+1][j]=T[i][j]+2*(t/(2*math.Pow(s,2)))*math.Pow(T[i][j],
p*(T[i][j+1]+T[i][j-1]-2*T[i][j]))

                                        }
                                }

                        } else {


        T[i+1][j]=T[i][j]+2*(t/(2*math.Pow(s,2)))*math.Pow(T[i][j],
p*(T[i][j+1]+T[i][j-1]-2*T[i][j]))

                                }
                }
        }

}
//Η συνάρτηση αυτή εκτυπώνει τον τελικό πίνακα.

func Ektipwsi_Pinaka(x,y int, Pinakas[N][M]float64) {
        fmt.Printf ("\n\n\n Oi times tou Pinaka T einai:\n\n")
                for i:=0; i<x;i++{
                                for j:=0; j<y;j++{
fmt.Printf(" %6.3f", Pinakas[i][j])  //Το 6.3f είναι για
                                                //το σύνολο των
  //ψηφίων (6) και τα //δεκαδικά ψηφία (.3) //που θα εμφανίζει.
                                }
                        fmt.Printf("\n")
                }

}
```

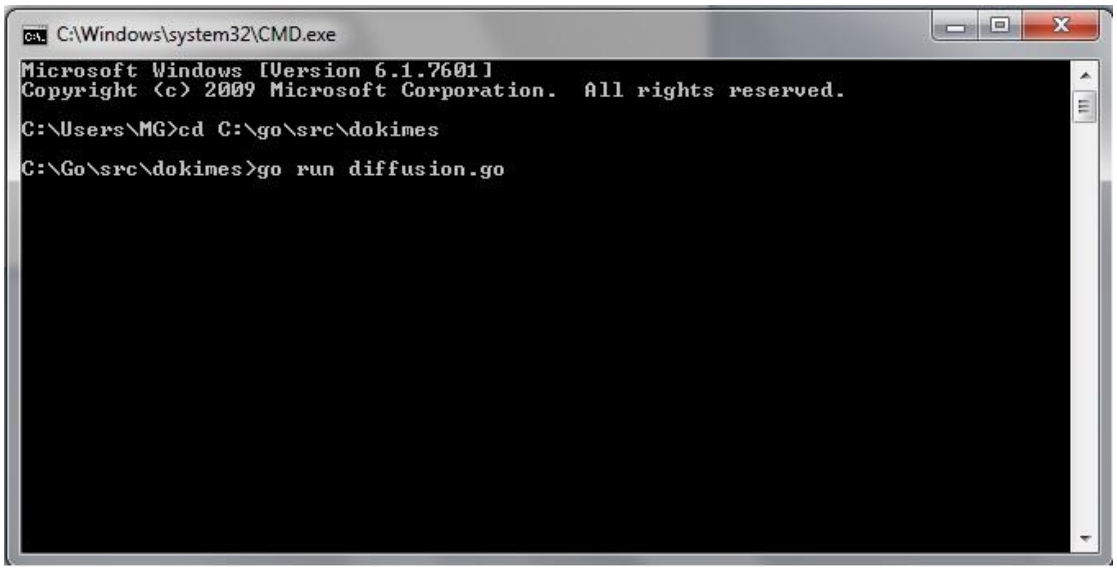
Για να κατανοήσουμε πως λειτουργεί ο κώδικας θα τον τρέξουμε βήμα-βήμα Ξεκινώντας , ανοίγουμε το τερματικό μας

και εισάγουμε με την εντολή cd τον κατάλογο στον οποίο βρίσκετε το αρχείο του κώδικά μας και πατάμε enter

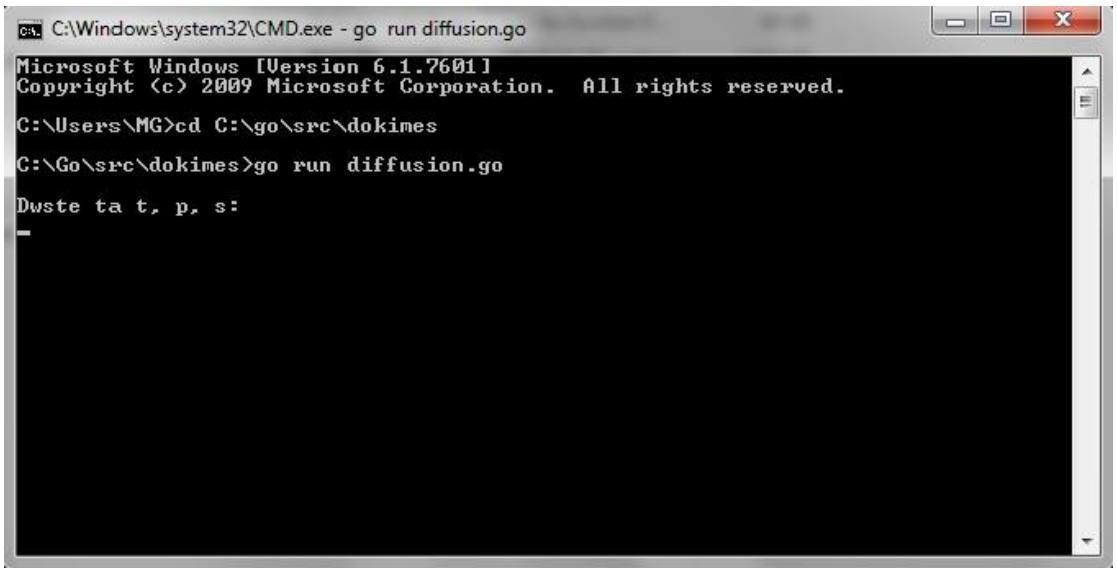


εμάς ο κατάλογός μας είναι ο C:\go\src\dokimes.

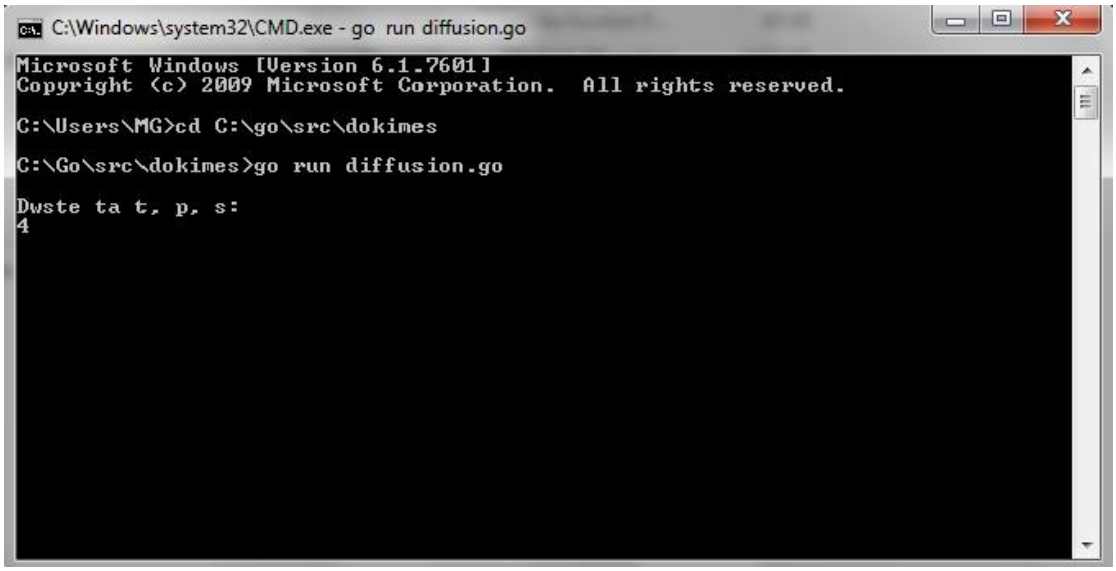
Στη συνέχεια δίνουμε την εντολή go run ακολουθούμενη από το όνομα του αρχείου μας με την κατάληξη .go. *Στη περίπτωσή μας το όνομα είναι diffusion.go.*



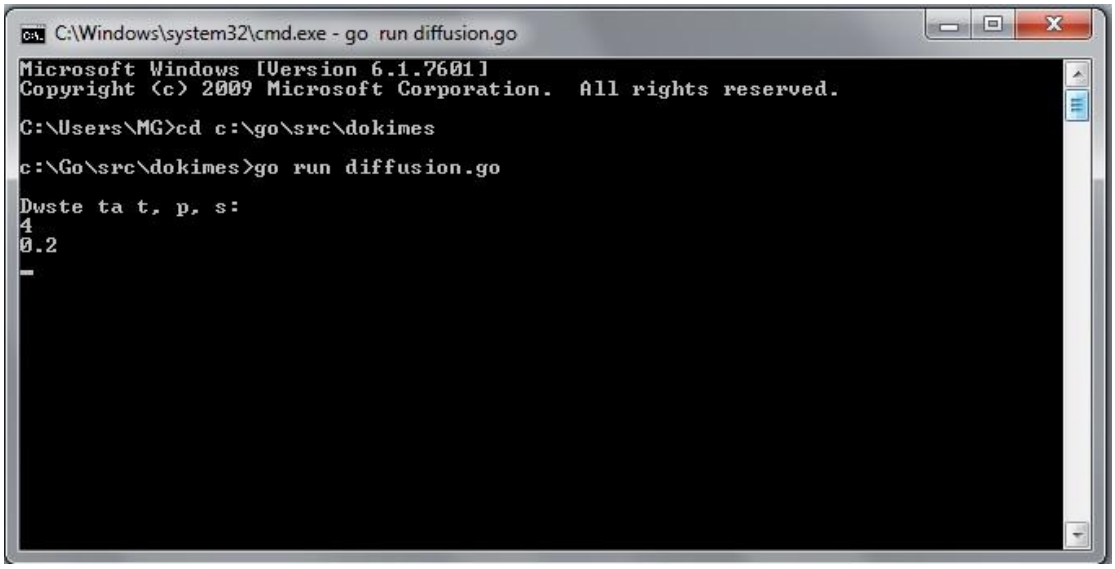
Αφού πατήσουμε enter αρχίζει ο κώδικάς μας να τρέχει και στο πρώτο στάδιο θα μας ζητήσει να εισάγουμε τα δεδομένα.



Δίνουμε το πρώτο ζητούμενο t ίσον με 4

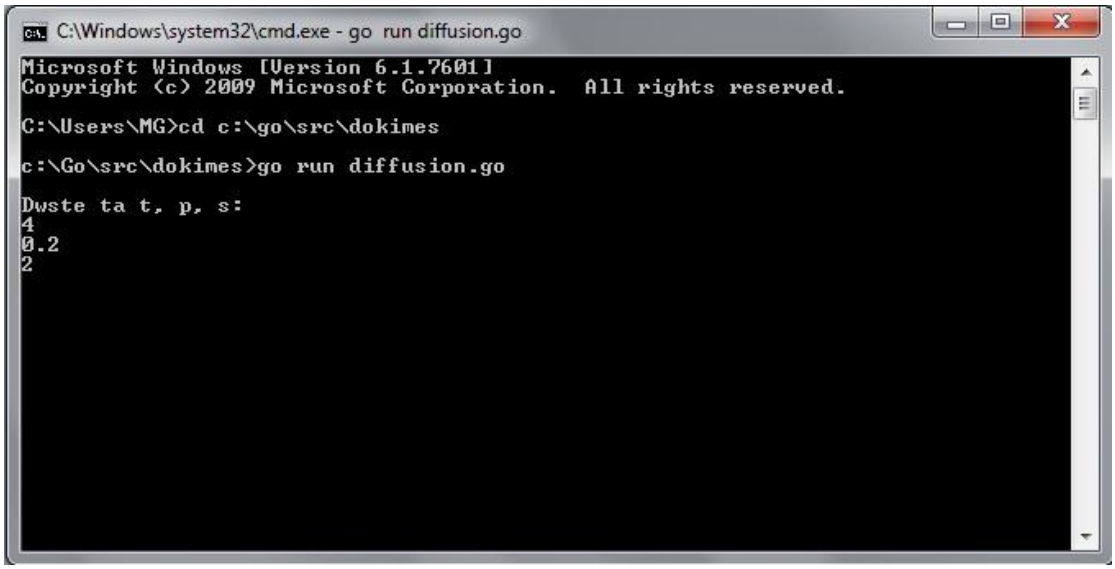


και πατάμε enter.
Δίνουμε το επόμενο ζητούμενο p ίσον με 0.2

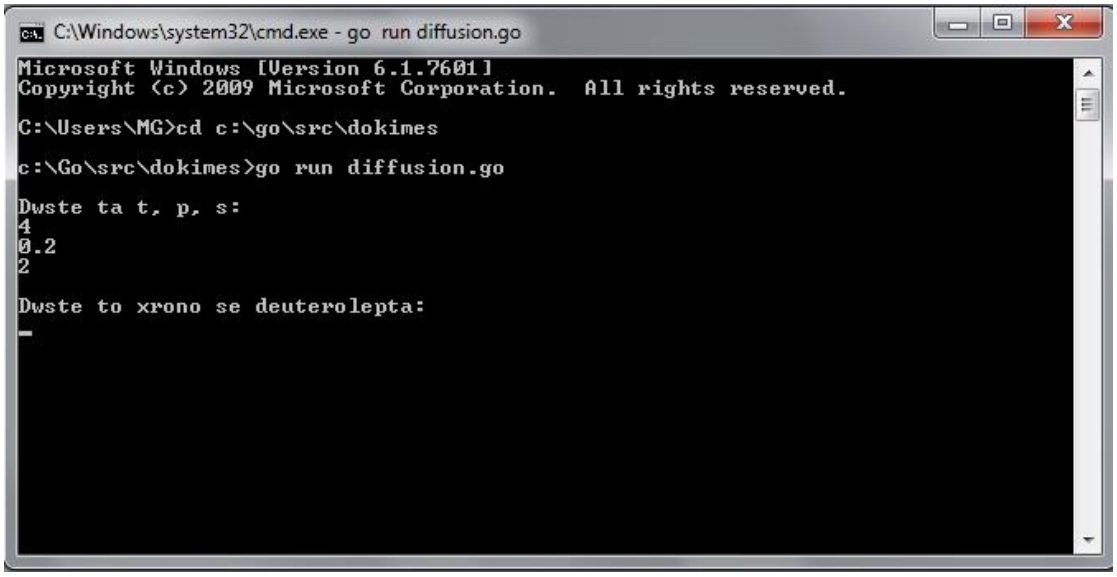


και πατάμε enter.

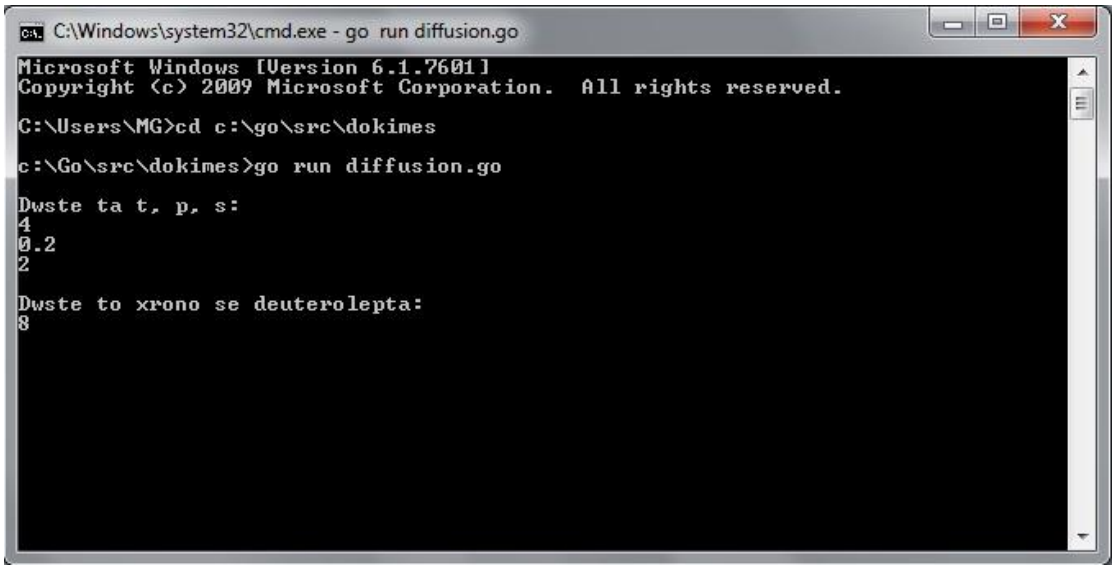
Δίνουμε και το τρίτο ζητούμενο s ίσον με 2

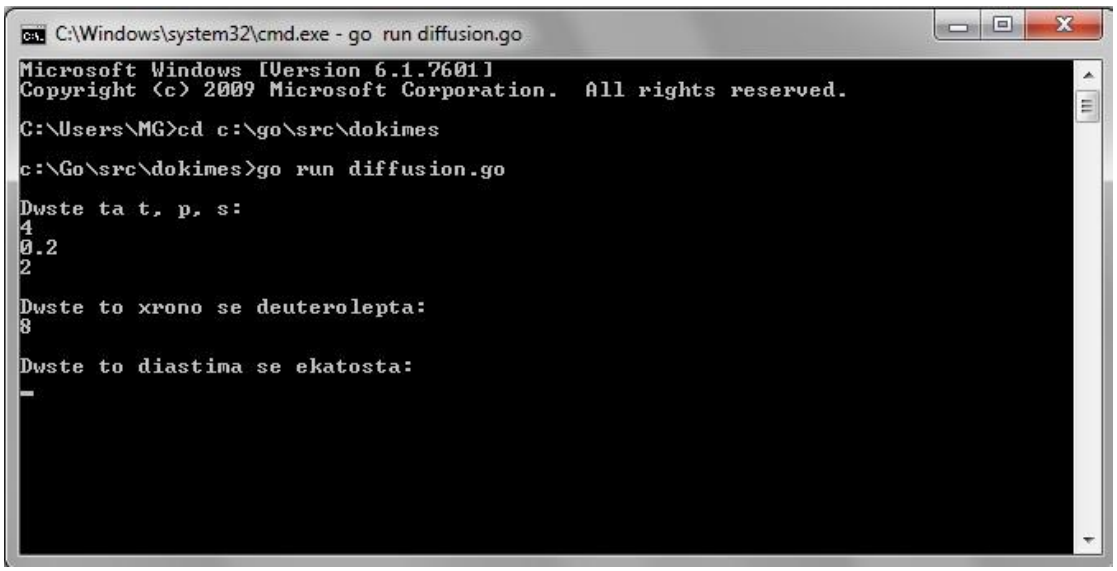


Πατώντας enter το πρόγραμμα μας ζητάει το χρόνο,

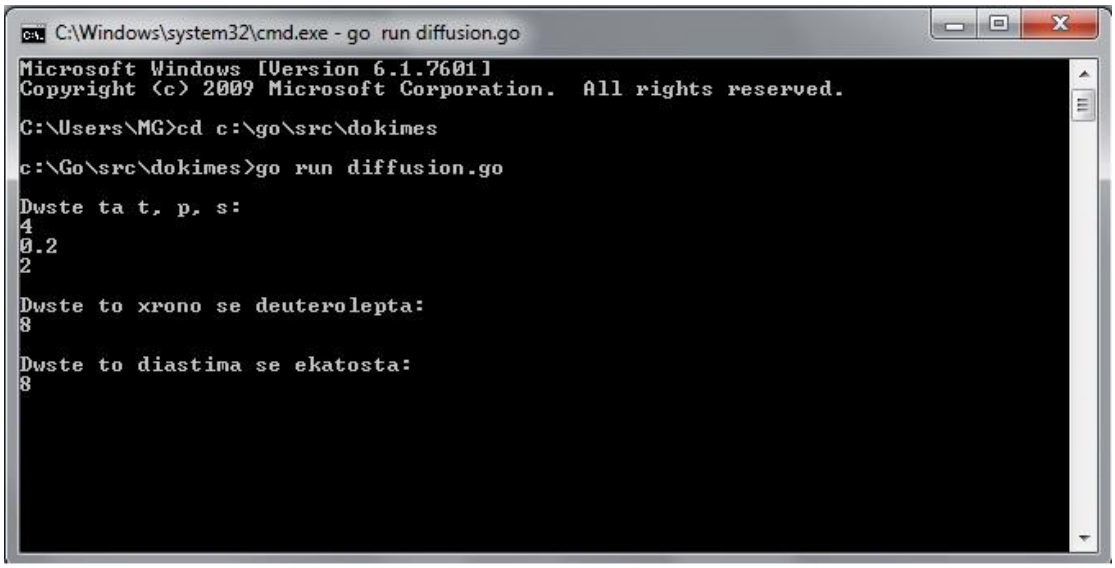


του δίνουμε 8 και πατάμε enter για να μας ζητήσει το διάστημα.

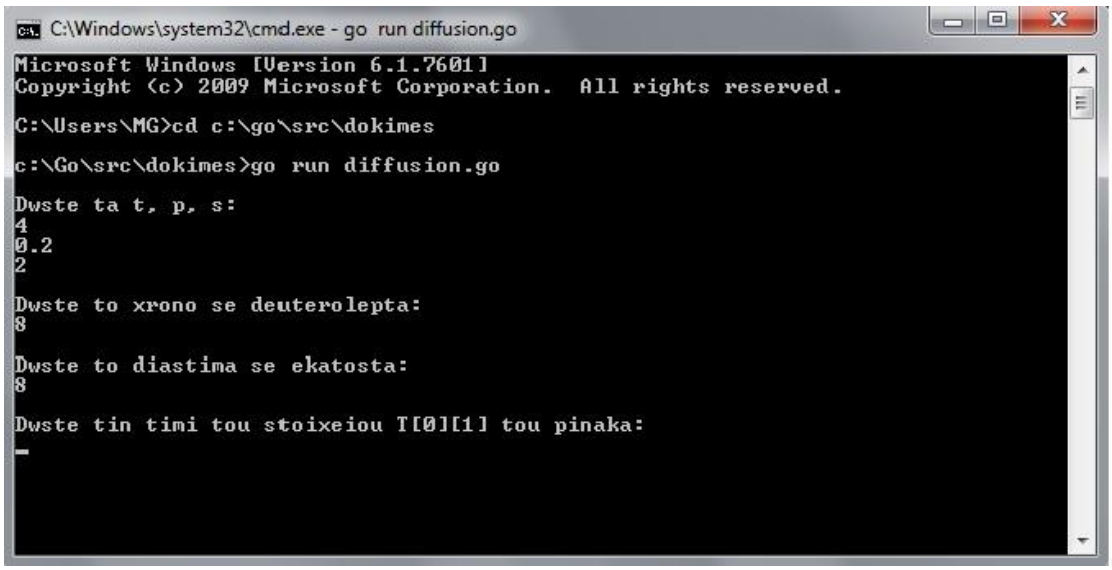




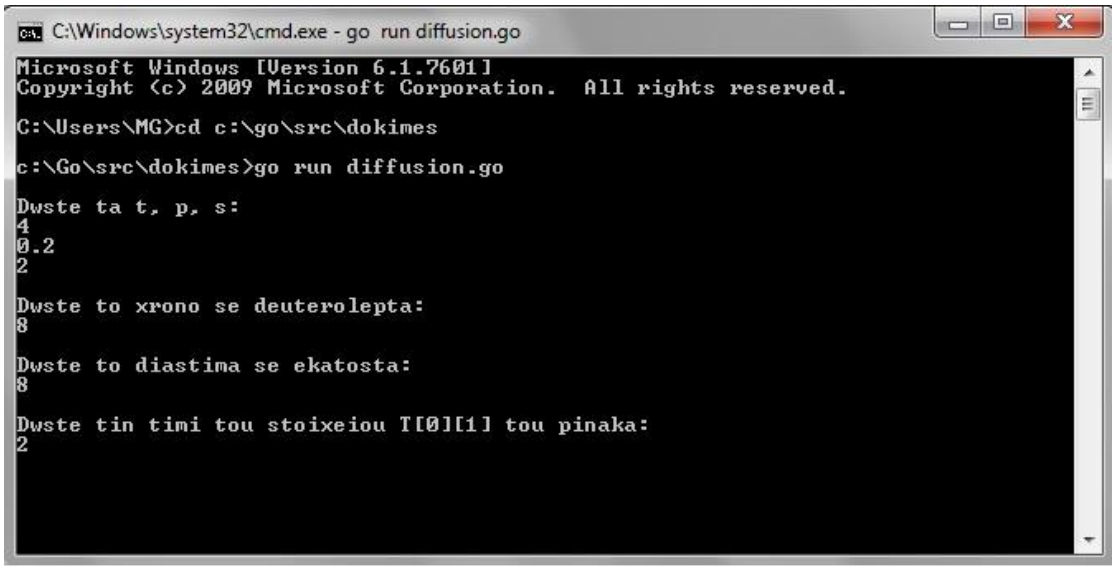
Δίνουμε ξανά 8 και πατάμε enter.



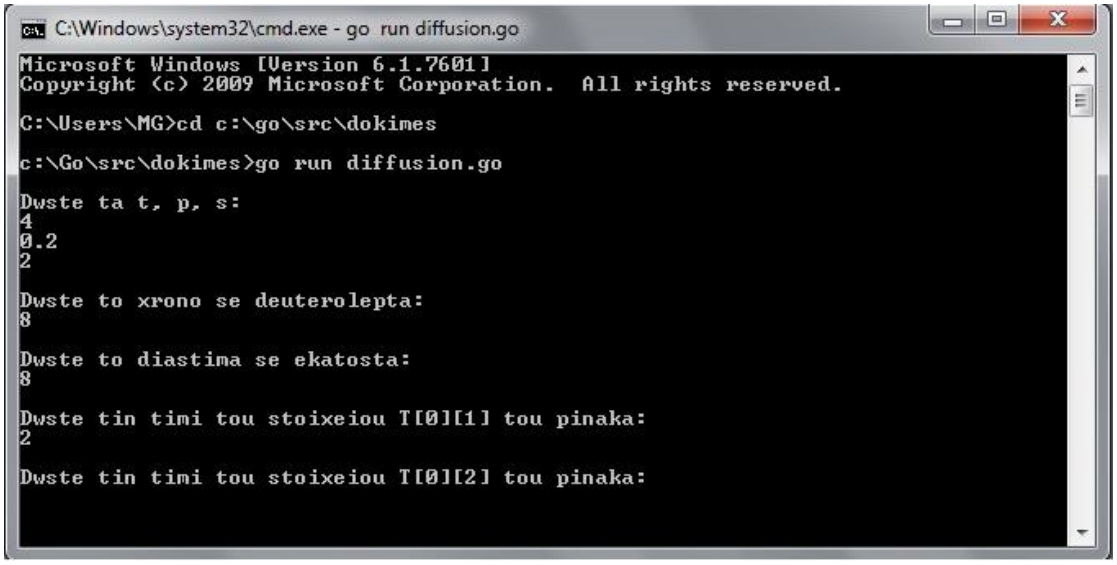
Μας ζητάει το στοιχείο της 0 σειράς και 1ης στήλης του πίνακα *(δεδομένου ότι ένας πίνακας ξεκινάει από το [0,0])*.



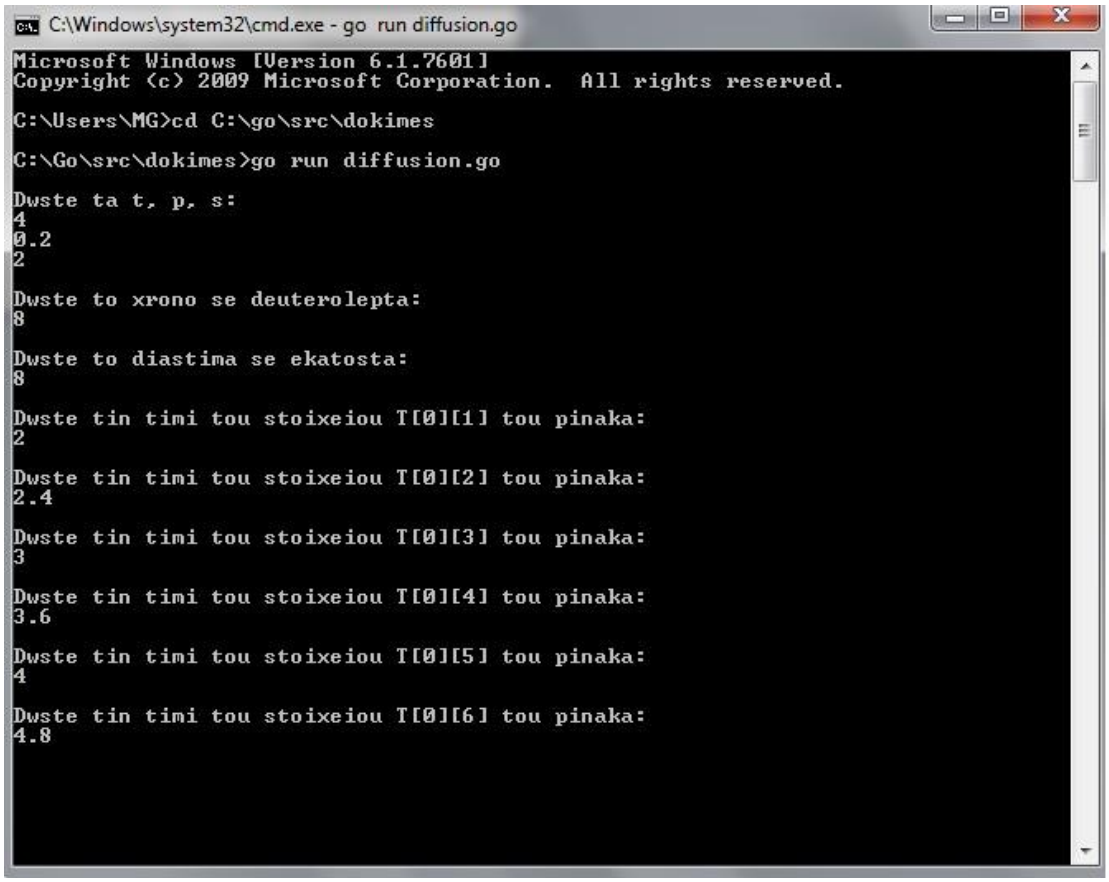
Δίνουμε το 2



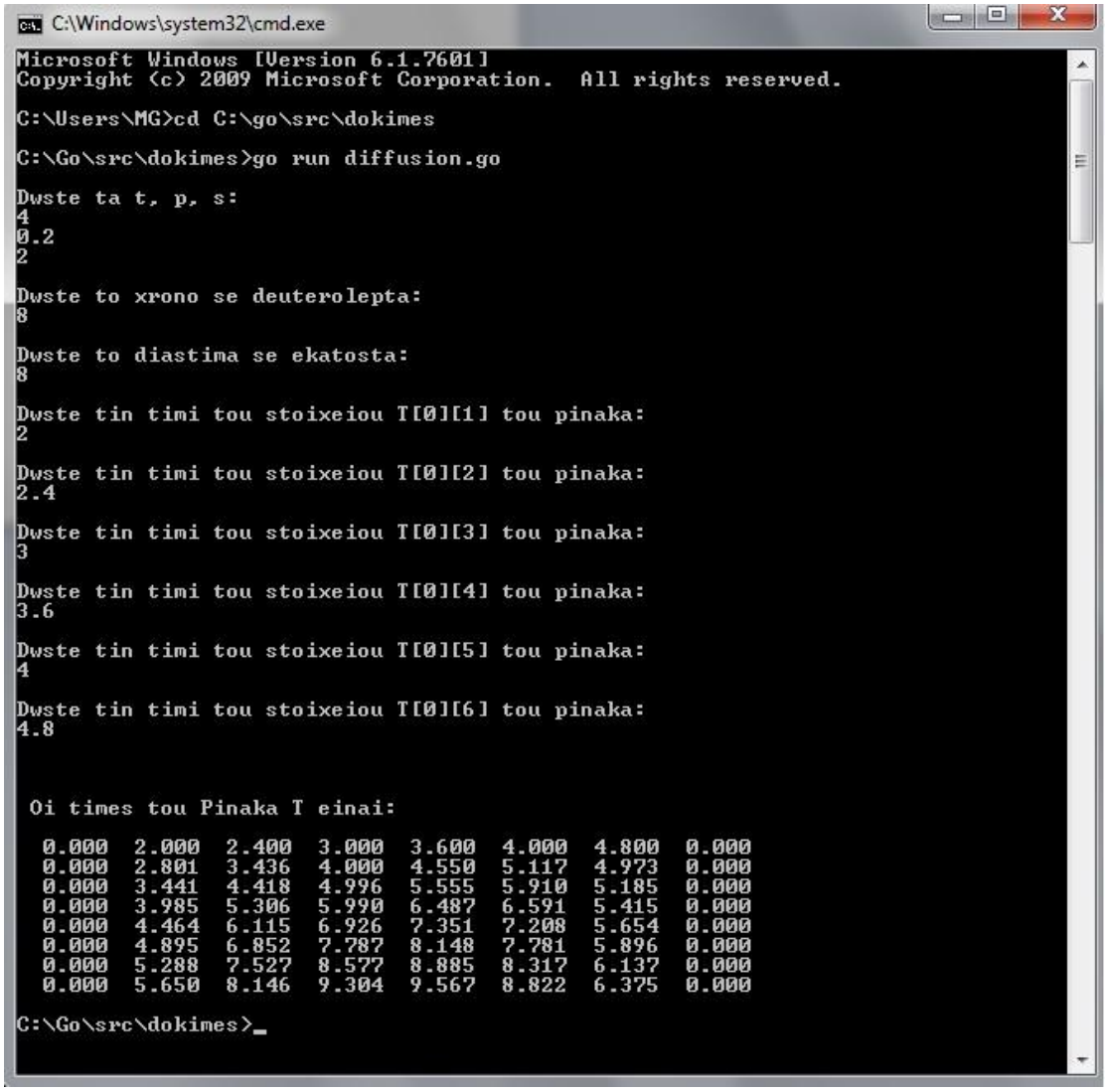
και πατώντας enter μας ζητάει το επόμενο στοιχείο [0,2] του πίνακα.



Την ίδια διαδικασία θα ακολουθήσουμε μέχρι να γεμίσουμε όλη την πρώτη σειρά του πίνακα.
Όταν δώσουμε και το τελευταίο στοιχείο της πρώτης σειράς το command window θα έχει αυτή τη μορφή:

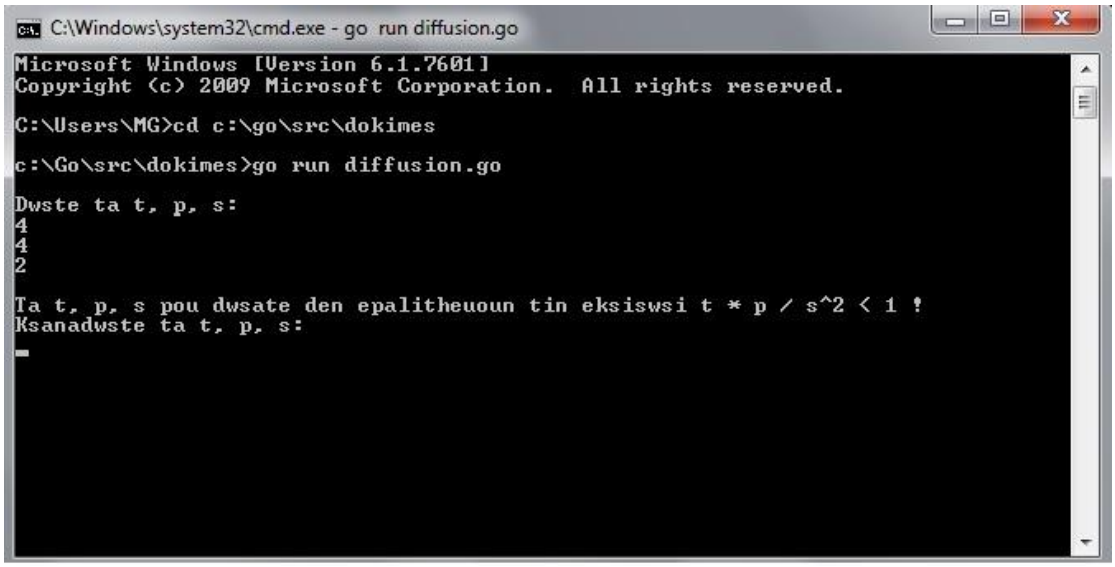


Όταν πατήσουμε enter τότε το πρόγραμμα θα έχει ολοκληρώσει τον υπολογισμό του πίνακα σύμφωνα με τα δεδομένα που του έχουμε εισάγει και θα δούμε ένα πίνακα της μορφής:



Στη περίπτωση όμως που του εισάγουμε λάθος τα δεδομένα t, p, s τότε το πρόγραμμά μας θα μας εμφανίσει ότι έχουμε εισάγει λάθος δεδομένα και θα μας ζητήσει να τα ξανά-εισάγουμε.

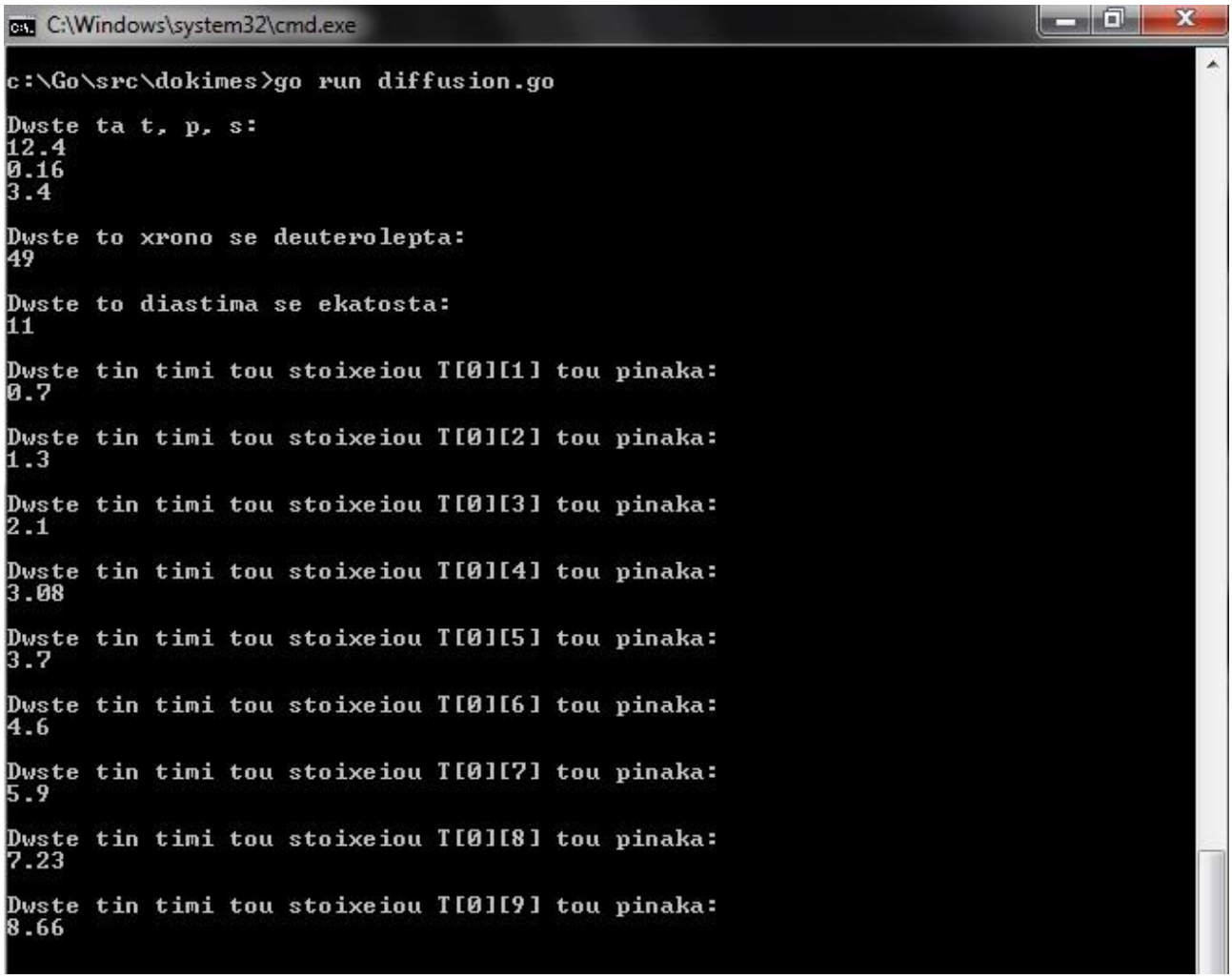
Έτσι θα δούμε αυτό:

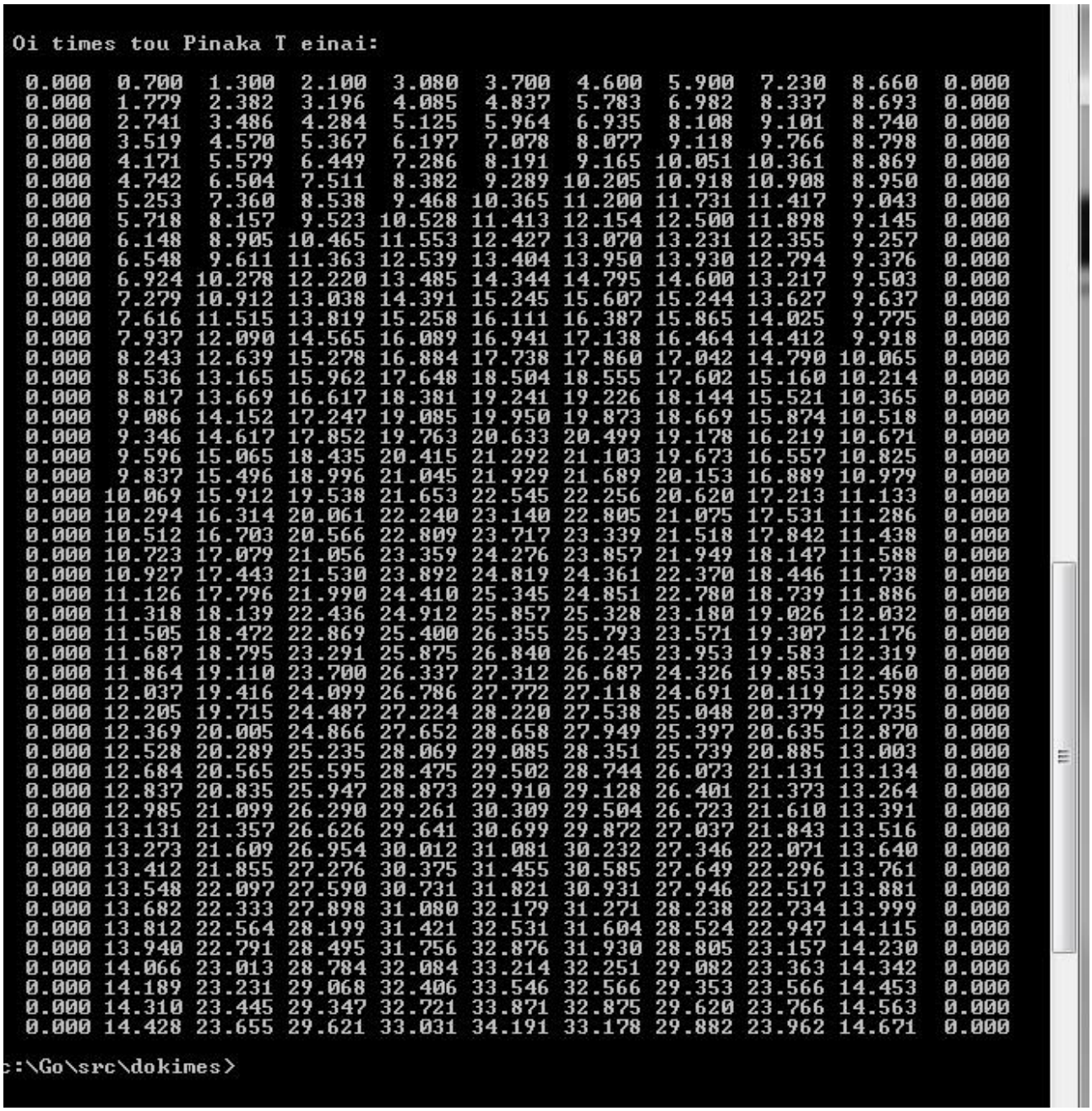


Ακολουθούν μερικά ακόμα παραδείγματα:

```
C:\Windows\system32\cmd.exe

C:\Users\MG>cd c:\go\src\dokimes

c:\Go\src\dokimes>go run diffusion.go

Dwste ta t, p, s:
0.2
0.04
0.9

Dwste to xrono se deuterolepta:
14

Dwste to diastima se ekatosta:
11

Dwste tin timi tou stoixeiou T[0][1] tou pinaka:
0.5

Dwste tin timi tou stoixeiou T[0][2] tou pinaka:
1.2

Dwste tin timi tou stoixeiou T[0][3] tou pinaka:
1.92

Dwste tin timi tou stoixeiou T[0][4] tou pinaka:
2.45

Dwste tin timi tou stoixeiou T[0][5] tou pinaka:
3.33

Dwste tin timi tou stoixeiou T[0][6] tou pinaka:
3.9

Dwste tin timi tou stoixeiou T[0][7] tou pinaka:
4.61

Dwste tin timi tou stoixeiou T[0][8] tou pinaka:
5.70

Dwste tin timi tou stoixeiou T[0][9] tou pinaka:
6.51


 Oi times tou Pinaka T einai:

  0.000  0.500  1.200  1.920  2.450  3.330  3.900  4.610  5.700  6.510  0.000
  0.000  0.746  1.447  2.166  2.700  3.573  4.149  4.863  5.942  6.653  0.000
  0.000  0.993  1.694  2.411  2.950  3.816  4.398  5.115  6.183  6.794  0.000
  0.000  1.240  1.941  2.657  3.201  4.060  4.647  5.368  6.421  6.934  0.000
  0.000  1.485  2.188  2.902  3.451  4.303  4.896  5.621  6.659  7.073  0.000
  0.000  1.729  2.435  3.147  3.702  4.546  5.145  5.873  6.894  7.210  0.000
  0.000  1.971  2.682  3.392  3.953  4.789  5.394  6.125  7.128  7.346  0.000
  0.000  2.209  2.929  3.637  4.203  5.033  5.643  6.377  7.360  7.481  0.000
  0.000  2.445  3.176  3.882  4.454  5.276  5.892  6.628  7.591  7.615  0.000
  0.000  2.677  3.422  4.127  4.705  5.520  6.141  6.879  7.819  7.748  0.000
  0.000  2.906  3.669  4.373  4.955  5.763  6.390  7.130  8.047  7.880  0.000
  0.000  3.131  3.915  4.618  5.206  6.007  6.639  7.381  8.272  8.010  0.000
  0.000  3.353  4.161  4.863  5.456  6.251  6.888  7.630  8.496  8.140  0.000
  0.000  3.571  4.406  5.108  5.706  6.495  7.137  7.880  8.719  8.268  0.000

c:\Go\src\dokimes>_
```

```
C:\Windows\system32\cmd.exe

c:\Go\src\dokimes>go run diffusion.go

Dwste ta t, p, s:
5.3
1.6
6

Dwste to xrono se deuterolepta:
25

Dwste to diastima se ekatosta:
11

Dwste tin timi tou stoixeiou T[0][1] tou pinaka:
0.3

Dwste tin timi tou stoixeiou T[0][2] tou pinaka:
0.9

Dwste tin timi tou stoixeiou T[0][3] tou pinaka:
1.6

Dwste tin timi tou stoixeiou T[0][4] tou pinaka:
2.4

Dwste tin timi tou stoixeiou T[0][5] tou pinaka:
3.3

Dwste tin timi tou stoixeiou T[0][6] tou pinaka:
4.2

Dwste tin timi tou stoixeiou T[0][7] tou pinaka:
5.5

Dwste tin timi tou stoixeiou T[0][8] tou pinaka:
6.8

Dwste tin timi tou stoixeiou T[0][9] tou pinaka:
7.9


 Oi times tou Pinaka T einai:

  0.000  0.300  0.900  1.600  2.400  3.300  4.200  5.500  6.800  7.900  0.000
  0.000  0.383  1.045  1.759  2.569  3.447  4.569  5.647  6.880  7.900  0.000
  0.000  0.478  1.193  1.919  2.732  3.686  4.701  5.873  6.956  7.900  0.000
  0.000  0.590  1.340  2.080  2.917  3.853  4.918  5.988  7.052  7.900  0.000
  0.000  0.718  1.487  2.245  3.092  4.047  5.067  6.132  7.127  7.900  0.000
  0.000  0.862  1.633  2.410  3.271  4.217  5.233  6.252  7.200  7.900  0.000
  0.000  1.012  1.781  2.576  3.444  4.390  5.382  6.372  7.267  7.900  0.000
  0.000  1.159  1.932  2.740  3.616  4.554  5.528  6.483  7.331  7.900  0.000
  0.000  1.293  2.085  2.904  3.783  4.715  5.668  6.590  7.392  7.900  0.000
  0.000  1.413  2.237  3.067  3.948  4.870  5.803  6.692  7.449  7.900  0.000
  0.000  1.519  2.385  3.228  4.109  5.021  5.933  6.791  7.504  7.900  0.000
  0.000  1.614  2.528  3.386  4.267  5.168  6.059  6.885  7.557  7.900  0.000
  0.000  1.700  2.664  3.540  4.422  5.312  6.181  6.977  7.608  7.900  0.000
  0.000  1.779  2.792  3.689  4.572  5.451  6.300  7.065  7.657  7.900  0.000
  0.000  1.852  2.914  3.832  4.718  5.587  6.415  7.151  7.704  7.900  0.000
  0.000  1.919  3.029  3.969  4.859  5.718  6.527  7.234  7.750  7.900  0.000
  0.000  1.983  3.138  4.101  4.995  5.846  6.635  7.314  7.794  7.900  0.000
  0.000  2.042  3.241  4.227  5.127  5.970  6.741  7.392  7.837  7.900  0.000
  0.000  2.098  3.340  4.348  5.254  6.090  6.843  7.468  7.879  7.900  0.000
  0.000  2.152  3.434  4.464  5.376  6.206  6.942  7.542  7.920  7.900  0.000
  0.000  2.202  3.523  4.575  5.494  6.318  7.039  7.614  7.959  7.900  0.000
  0.000  2.251  3.609  4.681  5.607  6.426  7.132  7.684  7.998  7.900  0.000
  0.000  2.297  3.691  4.784  5.717  6.532  7.223  7.752  8.035  7.900  0.000
  0.000  2.341  3.769  4.883  5.823  6.633  7.311  7.817  8.072  7.900  0.000
  0.000  2.384  3.845  4.977  5.925  6.732  7.396  7.882  8.107  7.900  0.000

c:\Go\src\dokimes>
```


C:\Windows\system32\cmd.exe
c:\Go\src\dokimes>go run diffusion.go
Dwste ta t, p, s:
12.4
0.16
3.4
Dwste to xrono se deuterolepta:
49
Dwste to diastima se ekatosta:
11
Dwste tin timi tou stoixeiou T[0][1] tou pinaka:
0.7
Dwste tin timi tou stoixeiou T[0][2] tou pinaka:
1.3
Dwste tin timi tou stoixeiou T[0][3] tou pinaka:
2.1
Dwste tin timi tou stoixeiou T[0][4] tou pinaka:
3.08
Dwste tin timi tou stoixeiou T[0][5] tou pinaka:
3.7
Dwste tin timi tou stoixeiou T[0][6] tou pinaka:
4.6
Dwste tin timi tou stoixeiou T[0][7] tou pinaka:
5.9
Dwste tin timi tou stoixeiou T[0][8] tou pinaka:
7.23
Dwste tin timi tou stoixeiou T[0][9] tou pinaka:
8.66


Oi times tou Pinaka T einai:
0.000 0.700 1.300 2.100 3.080 3.700 4.600 5.900 7.230 8.660 0.000
0.000 1.779 2.382 3.196 4.085 4.837 5.783 6.982 8.337 8.693 0.000
0.000 2.741 3.486 4.284 5.125 5.964 6.935 8.108 9.101 8.740 0.000
0.000 3.519 4.570 5.367 6.197 7.078 8.077 9.118 9.766 8.798 0.000
0.000 4.171 5.579 6.449 7.286 8.191 9.165 10.051 10.361 8.869 0.000
0.000 4.742 6.504 7.511 8.382 9.289 10.205 10.918 10.908 8.950 0.000
0.000 5.253 7.360 8.538 9.468 10.365 11.200 11.731 11.417 9.043 0.000
0.000 5.718 8.157 9.523 10.528 11.413 12.154 12.500 11.898 9.145 0.000
0.000 6.148 8.905 10.465 11.553 12.427 13.070 13.231 12.355 9.257 0.000
0.000 6.548 9.611 11.363 12.539 13.404 13.950 13.930 12.794 9.376 0.000
0.000 6.924 10.278 12.220 13.485 14.344 14.795 14.600 13.217 9.503 0.000
0.000 7.279 10.912 13.038 14.391 15.245 15.607 15.244 13.627 9.637 0.000
0.000 7.616 11.515 13.819 15.258 16.111 16.387 15.865 14.025 9.775 0.000
0.000 7.937 12.090 14.565 16.089 16.941 17.138 16.464 14.412 9.918 0.000
0.000 8.243 12.639 15.278 16.884 17.738 17.860 17.042 14.790 10.065 0.000
0.000 8.536 13.165 15.962 17.648 18.504 18.555 17.602 15.160 10.214 0.000
0.000 8.817 13.669 16.617 18.381 19.241 19.226 18.144 15.521 10.365 0.000
0.000 9.086 14.152 17.247 19.085 19.950 19.873 18.669 15.874 10.518 0.000
0.000 9.346 14.617 17.852 19.763 20.633 20.499 19.178 16.219 10.671 0.000
0.000 9.596 15.065 18.435 20.415 21.292 21.103 19.673 16.557 10.825 0.000
0.000 9.837 15.496 18.996 21.045 21.929 21.689 20.153 16.889 10.979 0.000
0.000 10.069 15.912 19.538 21.653 22.545 22.256 20.620 17.213 11.133 0.000
0.000 10.294 16.314 20.061 22.240 23.140 22.805 21.075 17.531 11.286 0.000
0.000 10.512 16.703 20.566 22.809 23.717 23.339 21.518 17.842 11.438 0.000
0.000 10.723 17.079 21.056 23.359 24.276 23.857 21.949 18.147 11.588 0.000
0.000 10.927 17.443 21.530 23.892 24.819 24.361 22.370 18.446 11.738 0.000
0.000 11.126 17.796 21.990 24.410 25.345 24.851 22.780 18.739 11.886 0.000
0.000 11.318 18.139 22.436 24.912 25.857 25.328 23.180 19.026 12.032 0.000
0.000 11.505 18.472 22.869 25.400 26.355 25.793 23.571 19.307 12.176 0.000
0.000 11.687 18.795 23.291 25.875 26.840 26.245 23.953 19.583 12.319 0.000
0.000 11.864 19.110 23.700 26.337 27.312 26.687 24.326 19.853 12.460 0.000
0.000 12.037 19.416 24.099 26.786 27.772 27.118 24.691 20.119 12.598 0.000
0.000 12.205 19.715 24.487 27.224 28.220 27.538 25.048 20.379 12.735 0.000
0.000 12.369 20.005 24.866 27.652 28.658 27.949 25.397 20.635 12.870 0.000
0.000 12.528 20.289 25.235 28.069 29.085 28.351 25.739 20.885 13.003 0.000
0.000 12.684 20.565 25.595 28.475 29.502 28.744 26.073 21.131 13.134 0.000
0.000 12.837 20.835 25.947 28.873 29.910 29.128 26.401 21.373 13.264 0.000
0.000 12.985 21.099 26.290 29.261 30.309 29.504 26.723 21.610 13.391 0.000
0.000 13.131 21.357 26.626 29.641 30.699 29.872 27.037 21.843 13.516 0.000
0.000 13.273 21.609 26.954 30.012 31.081 30.232 27.346 22.071 13.640 0.000
0.000 13.412 21.855 27.276 30.375 31.455 30.585 27.649 22.296 13.761 0.000
0.000 13.548 22.097 27.590 30.731 31.821 30.931 27.946 22.517 13.881 0.000
0.000 13.682 22.333 27.898 31.080 32.179 31.271 28.238 22.734 13.999 0.000
0.000 13.812 22.564 28.199 31.421 32.531 31.604 28.524 22.947 14.115 0.000
0.000 13.940 22.791 28.495 31.756 32.876 31.930 28.805 23.157 14.230 0.000
0.000 14.066 23.013 28.784 32.084 33.214 32.251 29.082 23.363 14.342 0.000
0.000 14.189 23.231 29.068 32.406 33.546 32.566 29.353 23.566 14.453 0.000
0.000 14.310 23.445 29.347 32.721 33.871 32.875 29.620 23.766 14.563 0.000
0.000 14.428 23.655 29.621 33.031 34.191 33.178 29.882 23.962 14.671 0.000
c:\Go\src\dokimes>

16 Πηγές – Βιβλιογραφία

http://en.wikipedia.org/wiki/Go_%28programming_language%29

http://golang.org/

C Programming for Engineering & Computer Science
H.H.Tan, T.B.D' Orazio

An Introduction to Programming in Go.
(2012) by Caleb Doxsey