|  |
| --- |
|  |
| The Go Programming Language |
| C:\Users\Eric\Pictures\go_gopher_color_logo_250x249.png |
|  |
| **Eric Nelson** |
| **4/24/2011** |

|  |
| --- |
|  |

Contents

[Introduction 1](#_Toc291434431)

[History 1](#_Toc291434432)

[Applications 1](#_Toc291434433)

[Data Abstractions 1](#_Toc291434434)

[Arrays, Maps and Slices 1](#_Toc291434435)

[Channels 2](#_Toc291434436)

[Control Abstractions 2](#_Toc291434437)

[Parameter Passing 2](#_Toc291434438)

[Object Orientation 2](#_Toc291434439)

[Control Structures 2](#_Toc291434440)

[if 2](#_Toc291434441)

[for 3](#_Toc291434442)

[switch 3](#_Toc291434443)

[Bibliography 4](#_Toc291434444)

[Appendix 5](#_Toc291434445)

[main.go 5](#_Toc291434446)

[board.go 6](#_Toc291434447)

# Introduction

The Go programming language is a general purpose language designed primarily for systems programming. Go was created with the intention of combining the ease of programming of dynamically typed interpreted languages with the efficiency and safety of statically typed language[[1]](#endnote-1). Another major goal of the language is built in support for concurrency, which is accomplished with goroutines. The language is under very active development, and is constantly evolving and improving.

## History

Robert Griesemer, Rob Pike and Ken Thompson first started designing the language in 2007, and continued designing while working on other things as well. By 2008 Ken Thompson had begun work on a compiler. Since then, many other people have contributed to this open source language.

## Applications

Go is primarily a systems language, as with its ancestor C, as well as be a concurrent language. The concurrency built into Go

# Data Abstractions

Go includes several novel data abstractions built in to the language, such as arrays, slices, maps, and structs.

## Arrays

Arrays in Go are very similar to fixed length arrays in other languages, but there are three main differences between C arrays and Go arrays. First, arrays are values, assigning one array to other copies its values. Furthermore, if you pass an array to a function, it is copied and passed by value.

myarray = make([100]) //make an array of 100 ints

myarray[5] = 100 //sets the 6th element to 100

for i := 0; i < len(myarray); i++ {

fmt.Println(myarray[i]) //print the contents of myarray

}

## Slices

Sli

## Channels

Channels are used to facilitate communication between different goroutines and subprocesses. They can be read from and written to, and hold a type, and when they are pass

# Control Abstractions

Go is made up of packages, types, and functions

## Parameter Passing

Everything in Go is passed by value, but Go does support pointers, similar to C,

## Object Orientation

Go is not object oriented, for the most part. In a package you can define structures, and methods that act on those structures, allowing a programmer to use the language in an object oriented way, there is no type hierarchy or inheritance.

Go does support interfaces, but there is no implements keyword. A type will satisfy an interface by implementing the methods of that interface[[2]](#endnote-2)

## Concurrency

Concurrency in Go is accomplished mainly through goroutines, which are very lightweight function calls, that happen in parallel with other . Goroutines communicate with each other using channels, which

To run a goroutine, just invoke a function using the keyword go

go longCalculation()

## Control Structures

Control structures are very similar to C, but differ in that there are no do or while loops. There are also no parentheses. There are only three control structures in Go, if, for, and switch.[[3]](#endnote-3)

### if

if b.colsize[col] == b.size {

ret = false

} else {

b.pieces[col][b.colsize[col]] = player

b.colsize[col]++

ret = true

}

### for

For is somewhat generalized to allow it to have the functionality of while.

// Like a C for

for init; condition; post { }

// Like a C while

for condition { }

// Like a C for(;;)

for { }

### switch

Switch is different from a C switch in that expressions do not need to be constants or integers.

func unhex(c byte) byte {

switch {

case '0' <= c && c <= '9':

return c - '0'

case 'a' <= c && c <= 'f':

return c - 'a' + 10

case 'A' <= c && c <= 'F':

return c - 'A' + 10

}

return 0

}

# Bibliography

*Effective Go.* n.d. http://golang.org/doc/effective\_go.htm (accessed April 24, 2011).

*FAQ - The Go Programming Language.* n.d. http://golang.org/doc/go\_faq.html (accessed April 24, 2011).

*Go Programming Language Specification.* March 3, 2011. http://golang.org/doc/go\_spec.html (accessed April 24, 2011).

# Appendix

I implemented project 2 in Go.

## main.go

package main

import (

board "board"

os "os"

fmt "fmt"

strconv "strconv"

)

func main() {

var size = 7

var connects = 4

if len(os.Args) >= 3 {

size\_r := os.Args[1]

connects\_r := os.Args[2]

size\_i, err\_s := strconv.Atoi(size\_r)

con\_i, err\_c := strconv.Atoi(connects\_r)

if err\_s != nil {

// handle error

fmt.Println(err\_s)

} else if err\_c != nil {

// handle error

fmt.Println(err\_c)

} else {

if size\_i >= 5 && size\_i <= 20 && con\_i >= 4 && con\_i <= 8 {

size = size\_i

connects = con\_i

} else {

fmt.Println("Size or number of connects out of bounds (size 5-20, connects 4-8)")

}

}

} else {

fmt.Println("Usage: connectfour <size of board> <number of connects>\nUsing default size 7x7 with 4 connects.")

}

fmt.Printf("%d size board, %d connects\n", size, connects)

b := board.NewBoard(size, connects)

playing := true

turn := 1

for playing {

b.Print()

fmt.Printf("%s's turn. Please select a column to play.\n", (\*b.Players())[turn])

var col int

\_, err := fmt.Scanf("%d", &col)

if err != nil || col <= 0 || col > b.Size() {

fmt.Println("Please enter a valid number.")

} else {

fmt.Printf("Playing %d\n", col)

b.Place(col-1, turn)

if turn == 1 {

turn = 2

} else {

turn = 1

}

}

won, winner := b.Check()

if won {

b.Print()

fmt.Printf("Congrations %s. You Won!\n", (\*b.Players())[winner])

playing = false

}

}

}

## board.go

package board

import fmt "fmt"

type Board struct {

pieces [][]int

colsize []int

players map[int]string

size int

connects int

}

func NewBoard(size, connects int) \*Board {

b := new(Board)

b.connects = connects

b.pieces = make([][]int, size)

b.colsize = make([]int, size)

for i := 0; i < size; i++ {

b.pieces[i] = make([]int, size)

}

b.players = map[int]string{

0: "\_",

1: "X",

2: "O",

}

b.size = size

return b

}

func (b \*Board) Size() int {

return b.size

}

func (b \*Board) Players() \*map[int]string {

return &b.players

}

func (b \*Board) Pieces() \*[][]int {

return &b.pieces

}

func (b \*Board) Print() {

fmt.Print("┌─")

for i := 1; i < b.size; i++ {

fmt.Print("┬─")

}

fmt.Println("┐")

for i := b.size - 1; i >= 0; i-- {

for j := 0; j < b.size; j++ {

fmt.Printf("│%s", b.players[b.pieces[j][i]])

}

fmt.Println("│")

}

fmt.Print("└─")

for i := 1; i < b.size; i++ {

fmt.Print("┴─")

}

fmt.Println("┘")

}

func (b \*Board) Place(col, player int) bool {

var ret bool

if b.colsize[col] == b.size {

ret = false

} else {

b.pieces[col][b.colsize[col]] = player

b.colsize[col]++

ret = true

}

return ret

}

func (b \*Board) Check() (is\_win bool, winner int) {

//check vertically.

consecutive := 1

prev := 0

for row := 0; row < b.size; row++ {

for col := 0; col < b.size; col++ {

if b.pieces[col][row] == prev && prev != 0 {

consecutive++

} else {

consecutive = 1

}

if consecutive >= b.connects {

return true, b.pieces[col][row]

}

prev = b.pieces[col][row]

}

}

//check horizontally.

consecutive = 1

prev = 0

for col := 0; col < b.size; col++ {

for row := 0; row < b.size; row++ {

if b.pieces[col][row] == prev && prev != 0 {

consecutive++

} else {

consecutive = 1

}

if consecutive >= b.connects {

return true, b.pieces[col][row]

}

prev = b.pieces[col][row]

}

}

// checck diagonally /

for col := 0; col <= b.size-b.connects; col++ {

for row := 0; row <= b.size-b.connects; row++ {

if b.pieces[col][row] != 0 {

complete := true

for dm := 1; dm < b.connects; dm++ {

row\_d := row + dm

col\_d := col + dm

if b.pieces[col][row] != b.pieces[row\_d][col\_d] {

complete = false

break

}

}

if complete {

return true, b.pieces[col][row]

}

}

}

}

// checck diagonally \

for col := 0; col <= b.size-b.connects; col++ {

for row := b.size - 1; row >= b.connects-1; row-- {

if b.pieces[col][row] != 0 {

complete := true

for dm := 1; dm < b.connects; dm++ {

row\_d := row - dm

col\_d := col + dm

if b.pieces[col][row] != b.pieces[col\_d][row\_d] {

complete = false

break

}

}

if complete {

return true, b.pieces[col][row]

}

}

}

}

return false, 0

}

1. (FAQ - The Go Programming Language n.d.) [↑](#endnote-ref-1)
2. (FAQ - The Go Programming Language n.d.) [↑](#endnote-ref-2)
3. (Effective Go n.d.) [↑](#endnote-ref-3)