Margaret Haley and Apurva Gandhi

Programming Languages: Design and Implementation

Professor King

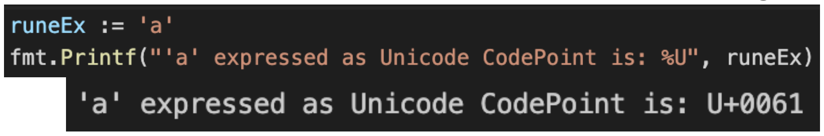
27 April 2021

Go

Robert Griesemer, Rob Pike, and Ken Thompson designed Go to be a clean, new language that would do more to help the programmer and account for issues likely to dominate the industry in the future. Dissatisfied with the complexity of languages at the time, they designed Go to provide ease of programming and efficient compilation and execution. The goals of Go included simplicity, support for concurrency and parallelism, ease of scalability, and automation of mundane tasks ("FAQ" 2-3). Originally developed from C, Go is a multi-paradigm language that is both imperative and concurrent. Additionally, Go provides some object-oriented features, such as types and methods (Griesemer slide 6, 20-28).

Go has 25 reserved words, including the common if, else, return, break, and continue. Some of Go's more unique reserved words are fallthrough, interface, and defer ("Keywords" 5). Less common primitive data types include rune [Image 1], an integer value that identifies a Unicode code point, and complex [Image 2], a floating type that represents real and imaginary numbers. Go also has iota [Image 3], which represents successive integer constants. In addition to basic structured types such as boolean, string, and array, Go also has struct,

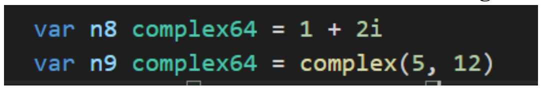
Image 1



```
runeEx := 'a'
fmt.Printf("'a' expressed as Unicode CodePoint is: %U", runeEx)
```

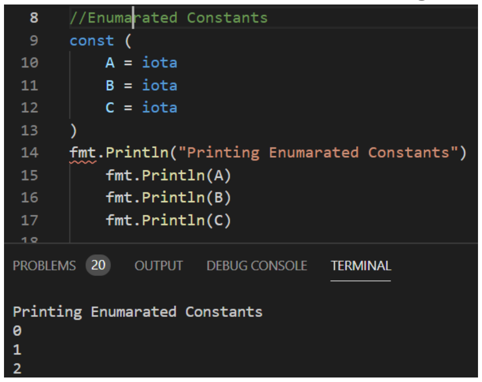
```
'a' expressed as Unicode CodePoint is: U+0061
```

Image 2



```
var n8 complex64 = 1 + 2i
var n9 complex64 = complex(5, 12)
```

Image 3

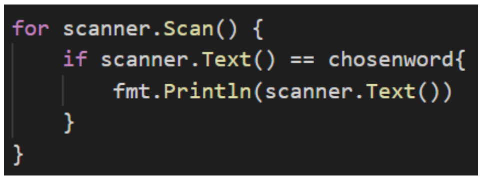


```
//Enumarated Constants
const (
    A = iota
    B = iota
    C = iota
)
fmt.Println("Printing Enumarated Constants")
    fmt.Println(A)
    fmt.Println(B)
    fmt.Println(C)
```

```
Printing Enumarated Constants
0
1
2
```

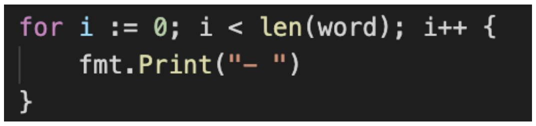
slice, and channel types. Structs are heterogenous data types composed of named fields of types, slices are flexible arrays with a variable-length sequence, and channels allow for concurrently executing functions to communicate ("Types" 12-16, 20-21). Variables in Go are declared in three ways: using the var keyword followed by the name and type; the same declaration followed by an = with the desired value; or the name followed by := and the value ("Declarations and Scope" 29-31). Go has a blank identifier, the _, which assigns and discards the value of a variable without throwing a compile error. This feature is useful because Go does not allow for unused variables. If a function returns multiple values and the programmer will not use them all, the programmer can assign them to _ ("The Blank Identifier" 35-36).

Image 4



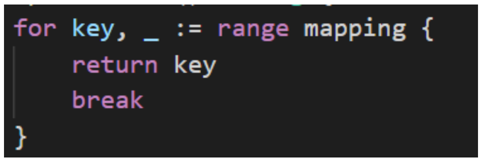
```
for scanner.Scan() {
    if scanner.Text() == chosenword{
        fmt.Println(scanner.Text())
    }
}
```

Image 5



```
for i := 0; i < len(word); i++ {
    fmt.Print("- ")
}
```

Image 6

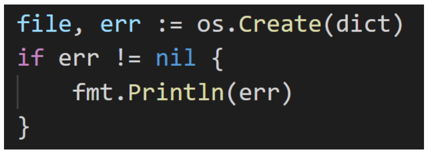


```
for key, _ := range mapping {
    return key
    break
}
```

Loops in Go can be controlled by a single condition [Image 4], a for clause [Image 5], or a range clause [Image 6]. A for loop with a for clause includes a variable declaration and a post statement, while a for loop controlled by a range clause iterates through all entries of a structured type. Break, continue, and goto statements alter the flow of control in Go loops. Conditional controls include if, switch, and select. A select statement is similar to that of a switch, except that cases refer to communication operations ("Statements" 63, 66-68, 70, 73-74). Functions in Go are declared with the keyword func followed by the function name, its parameters in parentheses, the function's return type if applicable, and the body enclosed in squiggle brackets. Although there are no classes in Go, there are functions with receivers called methods. The receiver is given in an argument list between the func keyword and the method name. Functions are called with the name of the function and any parameters, while methods are invoked with the receiver followed by a period and the method name, again with any parameters ("Declarations and Scope" 31-32). The parameters and return values of a function are passed by value by default, but call by reference may be used for parameters. Call by reference is indicated with an * before the type in a definition, and an & before the parameter in a call ("Pointers vs. Values" 28-29).

According to Robert Sebesta, languages should be evaluated according to their readability, writability, and reliability. Go achieves overall simplicity and orthogonality. It offers the necessary primitive data types for clarity and reserves only 25 words, striking a good balance between clear meaning and simplicity. Additionally, Go does not allow reserved words to be used as variable names. Go also does not offer a large variety of constructs, so programmers can learn them fairly easily as well as combine them in different ways. Although constructs allow for expression in languages, an overabundance may lead to misuse or disuse by programmers and ultimately decrease writability. According to Sebesta, expressivity refers to having "convenient, rather than cumbersome, ways of specifying computations," (Sebesta 37). Go offers convenient ways to perform common computations, such as using ++ to increment a variable. Go allows the programmer to assign variables without declaring types, which provides more flexibility. At the same time, it decreases reliability because variables may be assigned to a type other than that intended by the programmer (Sebesta 31-39). In fact, this exact issue was encountered during implementation of the hangman game. Aliasing also decreases Go's reliability, as Go allows for two variables to access the same cell in memory (Griesemer and Pike 1-2). Finally, Go provides solid exception handling [Image 7]; errors are treated as first-class citizens, and exceptions do not go undetected ("Errors" 47-48).

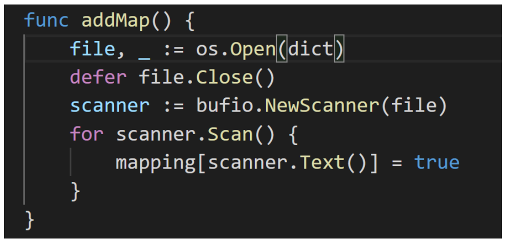
Image 7



```
file, err := os.Create(dict)
if err != nil {
    fmt.Println(err)
}
```

One of Go's strengths is the ease with which a programmer can use it; the simplicity of Go makes it painless to learn and understand. Go offers static analysis tools, which aids the programmer in identifying problems without running the code. One example of such tools is gofmt, which automates the task of code formatting with the command gofmt -w yourcode.go (Gerrand 1). The Go programming language provides native support for concurrent programming through built-in concurrency primitives, including channels that allow for constructing pipelines. Goroutines, which are Go's lightweight threads, use these channels to communicate in concurrent programs ("Goroutines" 41-42). Go programs compile to a native executable, which greatly increases speed of compilation as well as the volume of requests a server can handle concurrently ("Documentation" 1). Finally, Go's defer statement allows a programmer to execute some code at the end of the current function, usually as some sort of clean-up [Image 8]. The defer statement can appear anywhere in the code block, but it will always be executed at the end of the function ("Defer" 14-16).

Image 8



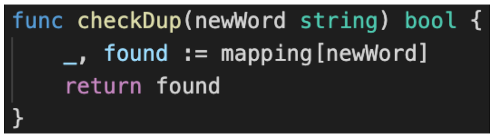
```go
func addMap() {
    file, _ := os.Open(dict)
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        mapping[scanner.Text()] = true
    }
}
```

One weakness of Go is that it does not allow for user-defined generics, which means that programmers cannot write abstractions that can be reused (Westrup and Pettersson 24). The only way to define generic data structures is through interface{}, which requires the programmer to cast values into the desired type and allows for type errors ("Types" 17-19). This disadvantage in Go is made worse by Go's limited built-in data structures. In an effort to make up for this, recent versions of Go include a containers package that offers heap and circular list operations, as well as a doubly linked list implementation ("Packages" 1-6). Go uses a garbage collector to automatically manage memory and does not offer any form of manual memory management.

While this design decision restricts some flexibility, it also removes added responsibility from the programmer and reduces potential errors (Kayyum Shaikh and Borde 4). Finally, although concurrency in Go is easily implemented, the programmer is still responsible for avoiding race conditions (Westrup and Pettersson 39).

The Go hangman program allows the user to play a game of hangman in the terminal. The program displays an ASCII art depiction of the man on the gallows, an outline of the word to be guessed, and a list of previously-guessed letters. Since Go lacks a native GUI, the program instead allows the user to permanently add new words to the dictionary file. The hangman game highlights Go's map data structure, which is an unordered group of elements indexed by a set of unique keys ("Map Types" 20-21). Words are stored from the dictionary file into the keys of a map, and the elements in the map are a boolean type set to true. To select a random word for each game, the program takes advantage of the map's random ordering by starting an iteration of it and breaking out immediately. Hangman.go also demonstrates the use of the blank identifier with the map structure [Image 9]. In this example, mapping[newWord] returns two values: the actual value (or 0, if not present) of the map element associated with the newWord key, and a boolean value indicating if the key is present. The use of Go as a language made the program easier to implement because the map structure provided simple and easy ways to store words, check for duplication, and select random words.

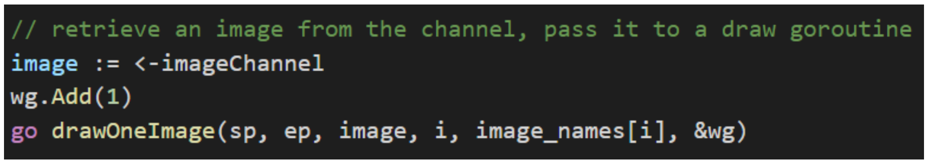
Image 9



```go
func checkDup(newWord string) bool {
    _, found := mapping[newWord]
    return found
}
```

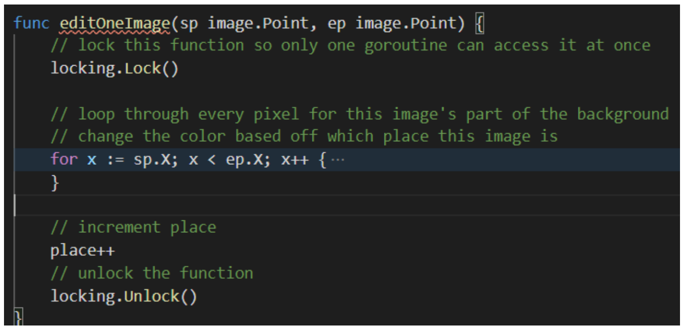
The collage.go creative program prompts the user for four images and returns a collage of these images in an output file. The program crops each image to a circle, changes its background to a unique color, and places the image on the collage. The background colors of the images correspond to the order in which they were drawn (red is first, then green, blue, and purple). Due to the program's concurrent nature, the four images are not always drawn in the same order, causing the placement of the colors to vary. In our comparison.go program, which builds a collage without concurrency, images are read and drawn one at a time. Unlike collage.go, the background colors always appear in the same position. The collage.go program demonstrates goroutines and channels, two built-in concurrency primitives in Go. The readImage() and drawOneImage() functions are called with the word go preceding them, which initiates a new goroutine [Image 10]. Additionally, a channel is used to pass images from the reading function to the drawing one [Image 10]. In collage.go, the image package is used for image processing and manipulation, and the sync package is used to create a WaitGroup that ensures concurrent tasks are completed. Sync is also used to create a Mutex locking struct that prevents more than one goroutine from editing the colors in the collage at once [Image 11]. The program also highlights Go's ability to declare functions with multiple return values through the getCoords() function. Finally, our function getNames() makes use of Go's interesting slicing feature to remove an extra newline [Image 12]. Overall, Go's concurrency features made the program easier to implement, as we simply had to add a WaitGroup(), channel, and the word go before the function calls.

Image 10



```
// retrieve an image from the channel, pass it to a draw goroutine
image := <-imageChannel
wg.Add(1)
go drawOneImage(sp, ep, image, i, image_names[i], &wg)
```

Image 11

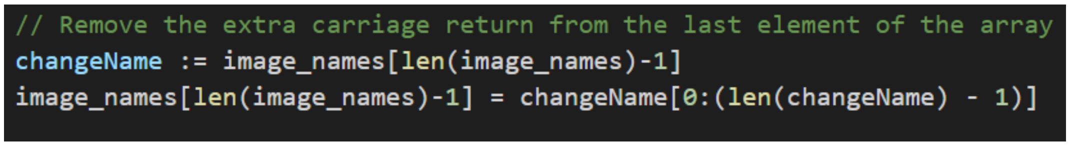


```
func editOneImage(sp image.Point, ep image.Point) {
    // lock this function so only one goroutine can access it at once
    locking.Lock()

    // loop through every pixel for this image's part of the background
    // change the color based off which place this image is
    for x := sp.X; x < ep.X; x++ {···
    }

    // increment place
    place++
    // unlock the function
    locking.Unlock()
}
```

Image 12



```
// Remove the extra carriage return from the last element of the array
changeName := image_names[len(image_names)-1]
image_names[len(image_names)-1] = changeName[0:(len(changeName) - 1)]
```

References

1. “Declarations and Scope.” *Go*, https://golang.org/ref/spec#Declarations_and_scope. Accessed 2021 March 21.
2. “Defer.” *Go,* https://golang.org/doc/effective_go#defer. Accessed 2021 March 22.
3. “Documentation.” *Go*, https://golang.org/doc/. Accessed 2021 March 20.
4. “Errors.” *Go,* https://golang.org/ref/spec#Errors. Accessed 2021 March 23.
5. “Frequently Asked Questions (FAQ).” *Go*, https://golang.org/doc/faq#What_is_the_purpose_of_the_project. Accessed 2021 March 20.
6. Gerrard, A. “go fmt your code.” *The Go Blog*. https://blog.golang.org/gofmt. Accessed 2021 April 23.
7. “Goroutines.” *Go*, https://golang.org/doc/effective_go#goroutines. Accessed 2021 April 23.
8. Griesemer, R. "The Evolution of Go." Talks.golang.org, July 9 2015, https://talks.golang.org/2015/gophercon-goevolution.slide#1. Accessed 2021 April 21.
9. Griesemer, Rob, and Rob Pike. *Proposal: Alias Declarations for Go*, 31 June 2016, https://go.googlesource.com/proposal/+/1487446b91599daa695905dc51a77d1bcc7086d8/design/16339-alias-decls.md. Accessed 2021 April 26.
10. Kayyum Shaikh, B. and Borde, S. Quantitative Evaluation by Applying Metrics to Existing ‘GO’ with Other Programming Languages in *International Conference on Ongoing Research in Management and IT - Volume 1.* https://www.researchgate.net/publication/285581592_Quantitative_Evaluation_by_applying_metrics_to_existing_GO_with_other_programming_language. Accessed 2021 April 26.
11. “Keywords.” *Go*, https://golang.org/ref/spec#Keywords. Accessed 2021 March 20.
12. “Pointers vs. Values” *Go*, https://golang.org/doc/effective_go#pointers_vs_values. Accessed 2021 March 22.
13. Sebesta, R.W. *Concepts of Programming Languages*. 11th ed. (Harlow: Pearson Education Limited, 2016). 30-41.
14. “Statements.” *Go*, https://golang.org/ref/spec#Statements. Accessed 2021 March 20.
15. “The Blank Identifier.” *Go*, https://golang.org/doc/effective_go#blank. Accessed 2021 March 21.
16. “Types.” *Go*, https://golang.org/ref/spec#Types. Accessed 2021 March 20.
17. “Map Types.” *Go,* https://golang.org/ref/spec#Map_types. Accessed 2021 March 24.
18. “Packages.” *Go,* https://golang.org/pkg/. Accessed 2021 March 22.
19. Westrup, E. and Pettersson, F. “Using the Go Programming Language in Practice.” *Axis Communications AB for the Department of Computer Science, Lund University,* 2014. https://www.researchgate.net/publication/312490994_Using_the_Go_Programming_Language_in_Practice. Accessed 2021 April 24.