Годишник на секция "Информатика"
Съюз на учените в България
Том 6, 2013, 76–85

Annual of "Informatics" Section
Union of Scientists in Bulgaria
Volume 6, 2013, 76–85

THE GO PROGRAMMING LANGUAGE – CHARACTERISTICS AND CAPABILITIES

Magdalina Todorova, Maria Nisheva-Pavlova, Georgi Penchev, Trifon Trifonov, Petar Armyanov, Atanas Semerdzhiev

Faculty of Mathematics and Informatics, Sofia University
5 James Bourchier blvd., Sofia 1164, Bulgaria

Abstract: This study presents the main ideas and capabilities of the Go programming language – a modern language developed by Google that integrates a few different conceptual paradigms. A comparative analysis is made with other languages in terms of concurrent programming, approach to OOP, typing and memory management - areas that were on focus during the inception of the Go language. Some typical examples of domain application of Go are reviewed and compared to similar tasks approached from other technologies.
Keywords: Programming languages, Imperative languages, Concurrent programming.

1. Design goals

The Go programming language was created in 2007 by Robert Griesmer, Robert Pike and Kenneth Thompson from Google Inc. It was officially published in 2009.
Go is a C-style language with some traits typical for the dynamic script languages. It integrates similar approaches to concurrency and object orientedness. A brief timeline of the development of this language is given in Fig. 1 [1]

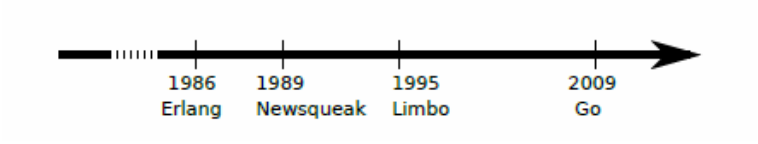


Figure 1. Timeline of Go

Some notable influences in the development of Go include:
- from C – simple structure and syntax;
- from Java – inheritance by means of "Interface";
- from C#, Java, etc – package definition;
- from Oberon – the concept of extensible structs and their attachable procedures;

- from Limbo, Newsqueak, Erlang – concurrency mechanism based on Tony Hoare's Communicating Sequential Process theory;
- from Javascript, Ruby and other dynamic languages – polymorphism, independent of inheritance.

The language is targeted towards system programming and this is why comparisons in this article are drawn mostly with C-style, as opposed to script languages.
The main goal of Go's development, according to its authors, is "to provide the efficiency of a statically typed compiled language with the ease of programming of a dynamic language" [2]. Other goals of the authors are: safety (type-safe and memory-safe); intuitive concurrency by providing „goroutines" and channels to communicate between them; efficient garbage collection "with low enough overhead and no significant latency"; high-speed compilation [3].

2. Language features

Go is a hybrid statically typed programming language. It integrates some concurrency mechanisms into the imperative style language framework. It is also possible to include machine-dependent code in Go programs, in order to increase performance and efficacy. Go is strongly typed and does not allow implicit type conversions, but it also exhibits some traits typical for the dynamically typed languages. It supports cross-compilation.

Go is not a classical object oriented language in the sense of C++ and Java because it does not feature a concept for classes and inheritance. However, the object oriented style of programming is implemented via the mechanism of interfaces that allows polymorphism to a significant degree. There is a clear and expressive type system without class hierarchies.
Go is also not a functional programming language but it has some of the main aspects of functional style. Functions are a fundamental building block of Go programs and it features some agile mechanisms for their use. The functional type in Go allows for the definition of function literals (or anonymous functions) that can be assigned to variables, be formal parameters to routines and to be returned as a result of those routines.

3. A brief review of Go's capabilities

R. Pike, one of the language's authors, defines its main features as follows [4]:
- a feel of a dynamic language with the safety of a static type system;
- compile to machine language so it runs fast;
- real run-time that supports garbage collection;
- concurrency;
- lightweight, flexible type system;
- has methods but is not a conventional object-oriented language.

A full language reference of Go can be found on its webpage [5].

3.1. Basics

For brevity's sake, some essential language constructs and its syntax constraints will be omitted from this exposition. We will present only some elements of particular importance among the flow control, functional and package constructs.

Flow control

Go implements some of the traditional C-style operators, among them: assignment, conditionals (if), loops (for), multivariant conditionals (switch), flow alterations such as break, continue, etc.

An interesting feature of the language is the ability to assign many variables simultaneously, as also seen in Ruby or Python:

```
a, b := 3, 6
a, b, c, d := 1, a+7, b+11, 29
```

Following these lines a, b, c and d are assigned 1, 10, 17, 29 respectively, i.e. the assignment is parallel and not consecutive. This mechanism also allows for value swapping:

```
a, b := b, a
```

The conditional operators (if-else и if) are semantically identical to their traditional counterparts in C-style languages. Their syntax however is more versatile. It is possible to initialize along with the condition:

```
if a, b := 1, 2; a+b > 0 {
    fmt.Println(a + b)
} else {
   fmt.Println(a - b)
}
```

The looping construct has a compact structure. It is only implemented via the operator *for*, with its three forms as follows:

```
- as in C
for init; condition; post { }
```

```
- as while in C
for condition { }
```

```
- as infinite loop (for(;;) or while(1)) in C
for { }
```

As with many other languages (Java, C++11, Python, etc), the Go language has an iterator version of the *for* loop which allows walking data structures. The syntax construct that is being used is *range*.

The *switch* operator in Go is very agile. Despite the similarities with its counterpart in C, it is also quite different and rather resembles *case* from Pascal. In Go it is not necessary for the comparison expressions to be constant – they can be any sort of conditions. This equates the *switch* operator with a series of nested *if's*. It is also not necessary to write many case labels for the same action and not necessary to use *break* to prevent falling through. The *switch* operator can also use run-time reflection and – much like *if* – can have initializer expressions in its conditions.

Functions

Functions are fundamental building blocks in Go programs. Their implementation is influenced by the modern programming language practices.

Following is an incomplete syntax schema of their definition:

func (p mytype) funcname(param type ***{****, param type* ***})*** *(ret_param type* ***{****, ret_param type* ***})***
{
… return …
}

The bold braces in this definition signify repetition in the sense of the Backus-Naur form.

Thus, *func* is a keyword; *(p mytype)* denotes a receiver and is used with method definition; *funcname* points to the function name; while *param* and *type* are a parameter and its type. Parameters are passed by value. *ret_param* and *type* are the return value and its type. As in C, curly braces frame the function body.

In Go functions can have multiple return values, as it is in Python or PHP. Also, return values can be bound to symbols.

There are no exceptions in Go. Error conditions can be signified by including error codes among the possible return values.

Packages

Go supports a simple but efficient module management system, similar to the one used in Java. A package is a set of functions and data. The keyword *package* is used for defining it. Every program consists of packages and execution starts always in the *main* package.

3.2. Types, methods, and interfaces

Types

The unusual type system is one of Go's most distinguishing traits. It avoids making use of some typical object-oriented concepts, e.g. inheritance. It is possible to define structured types and then create a set of methods for them. It is also possible to define interfaces, similar to those in Java. The interface mechanism allows for both duck-typing behavior and compile-time type checking.

The Go language has a lot of intrinsic types: booleans (bool), strings (string), numbers (int, int8, int16, int32 int64, uint, uint8, uint16, uint32, uint64, uintptr, byte – alias for uint8, rune – alias for int32 representing a Unicode code point, float32, float64, complex64, complex128); array types; slice types; struct types; pointer types; function types; interface types; map types; channel types.

For the sake of brevity, this paper will concentrate only on arrays, slices and maps.

Arrays

Arrays in Go are distinct from the ones in C and more closely resemble those in Pascal. Their features can be summarized as follows:

- The array size is a part of the array type;

- They are implemented as a sequence of values (as opposed to pointers). Assigning one array to another has copy semantics;
- If an array is passed as a function parameter, then the function operates on a copy of that array. Passing by reference can be achieved by use of pointers;

Slices

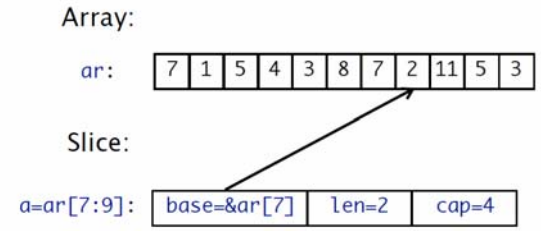
Slices are inspired by a similar construct in the D language. They are a pointer to a contiguous part of an array. Slices can be declared the same ways as arrays are but without a predefined size. So for example *var a []int* denotes a slice of an integer array.

Given that *arr* is an array, the expression *arr[n:m]* represents a pointer to the elements of *arr* from the *n-th* to the *(m-1)-th*, as in Python.

Example [4] compares a slice and an array. A custom type Slice is defined as a structure with three members – a pointer to the 0-th element, the elements count and the capacity (the total buffer size) of the slice.

Example

```
type Slice struct {
    base *elemType          // pointer to the 0-th element
    len int                 // elements count
    cap int                 // buffer size
}
```



Unlike arrays, slices can grow as new elements are appended to them. The *append* function serves this purpose. Slices can be created using the *make* function, for example:

var sl100 = make([]int, 100)

creates the slice *sl100* with a hundred integers. It is recommended that slices are created with *make*.

Maps

This data type features in many other languages, e.g. Perl (hashes), C++ (unordered_map), Java (Map), Python (dictionary). In Go it is called map and is defined as follows:

map[<from type>]<to type>

where *<from type>* denotes the key type and *<to type>* denotes the value type. Maps' keys can be of mostly any type that implements the equality operator. Usually keys are strings

and therefore maps can be thought of as associative arrays. Keys cannot be structs, arrays or slices.

Example

```
var m map[string]float64
```

This statement declares a *map* with *string* keys and *float64* values. In the C++ standard library map<string, float64> has the same semantics.

Maps support insertion, removal and updating of any key-value tuples.

Methods

Methods are defined separately from the declaration of the types they belong to, using a special function syntax that defines the receiver of a method. Every type can be a receiver and thus have a method defined for it, which allows for overloading the behavior of intrinsic types.

Interfaces

Interfaces in Go are very similar to abstract classes in C++, typeclasses in Haskell, duck typing in Python or interfaces in Java. As mentioned in [4, day 2], the word "interface" is slightly overloaded in Go: there is the concept of an interface, and there is an interface type, and then there are values of that type.

- Interface concept

The set of all methods defined for a type, describes its interface [6]. For example, in the following code fragment there are two methods (*Get* and *Put*) defined for the type *newType*.

```
type newType struct { i int }
func (p *newType) Get() int { return p.i }
func (p *newType) Put(v int) { p.i = v }
```

- Interface type

This next fragment defines an interface with the name of *I* that has two methods – *Get* and *Put*.

```
type I interface {
    Get() int
    Put (int)
}
```

Thus, *newType* is an *implementation* of the interface *I*, since it implements the two methods that *I* requires.

- Interface values

If a variable is declared as having an interface type, it can then be assigned any value that implements the interface.

Example. The function *f* as defined below, has a formal parameter *p* of the interface type *I*. Since *p* is an interface value, it has the methods *Get* and *Put*.

```
func f(p I) {
    fmt.Println(p.Get())
    p.Put(1)
}
```

Because *newType* is a correct *implementation* of the interface *I*, we can call *f* as follows:

```
var s newType;
f(&s)
```

Classes

Go does not have classes, constructors or destructors. Nevertheless, it provides the ability to write object-oriented code by first declaring a custom structure type and then assigning methods to it with the function receiver mechanism. Since any type can be a method receiver, it is possible to add behavior to intrinsic types, much like in Smalltalk. Inheritance can be emulated to a degree using interfaces.

The next example illustrates these capabilities.

Example: This fragment defines a structure type *Point2* that models a 2D point in the Cartesian space. The "class" members are the coordinates *x* and *y*, while the methods are the getters, the setters, and a display function.

```
type Point2 struct { x, y int }
func (p * Point2) Get_x() int { return p.x }
func (p * Point2) Get_y() int { return p.y }
func (p * Point2) Set_x(a int) { p.x = a }
func (p * Point2) Set_y(a int) { p.y = a }
func (p * Point2) Display() {fmt.Print(p.x, ", ", p.y}
```

3.3. Concurrency and communication

One of Go's main selling points is its focus on concurrent programming. The language features the so-called "go-routines" that allow for any function to be executed in a separate thread. At any given time only one function can access shared data. Go supplies a "channel" type that can serve as safe data access mechanism for the go-routines.

Go-routines

In [7] go-routines are described as follows – "A go-routine has a simple model: it is a function executing in parallel with other go-routines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small,

so they are cheap, and grow by allocating (and freeing) heap storage as required." Any function can be executed as go-routine by using the keyword *go* when calling it.

Channels

Channels in Go are modeled after ideas from "Communicating Sequential Processes (CSP)", by Tony Hoare [8] and can be viewed as a type safe implementation of Unix pipes. In Go, they serve as both data store and synchronization mechanism between go-routines. Channels are also created with the *make* function.

3.4. Garbage collection

Go has a garbage collection mechanism. Heap space can be allocated without the necessity of manually releasing it, as in most modern programming languages.

4. Some missing features

There are key object-oriented concepts that are missing from Go. Overloading operators and implicit conversions are among them but the absence of clearly defined class hierarchy is the most notorious. There are no variant types though interfaces compensate for this omission. Go does not support dynamic code loading, dynamic libraries and does not feature any generic programming construct. There are no exceptions, although the *recover* and *panic* functions serve much the same purpose. Run-time assertions are also not supported [9].

5. Application

Go as a system programming language

It has been successfully applied by Google Inc. for the development of server software and other internal developments. A part of Google Maps is powered by Go. It has showed good results in various distributed systems that require efficiency. The concurrency model, the high-level abstractions and the efficient implementation make Go a prime candidate for the implementation of Complex Event Processing (CEP) [10].

Go as a general purpose language

Any general purpose tasks can be solved by Go. It can be successfully used for problems that require text processing, script programs, etc. Because of the garbage collection mechanism Go is not ideally suited for real-time software.

The Chrome browser supports the so-called NativeClient (NaCl) Go compiler in the browser core. This allow for writing machine-dependent code in web-applications on Chrome OS. Many organizations use Go for a variety of software projects [11].

6. Conclusion

Go is a young language but with much potential. While not yet in the top 10 programming languages charts, there is significant data that Go is gathering momentum. Its simplicity for creating multi-threaded and network applications is an important selling point. It meets its main design goals – Go is powerful, agile, effective and successfully combining static typing safety with dynamic languages ease of use. It seems only a matter of time that Go takes its place among the most popular programming languages, most likely in the server-side world [12].

Beside its wide application palette, Go is a good choice for computer science academic purposes and allows for illustrating a number of programming language paradigms – compiled, concurrent, imperative, structured [3].

Acknowledgments

The presented study has been supported by the Bulgarian National Science Fund within the project titled "Contemporary programming languages, environments and technologies and their application in building up software developers", Grant No. DFNI-I01/12.

References

[1] Gieben, M., Learning Go. http://www.miek.nl/files/go/ (date of last visit: April 28, 2013).

[2] Pike, R., The Go Programming Language. YouTube, Retrieved 1 Jul 2011. (date of last visit: April 28, 2013).

[3] Go (programming language). http://en.wikipedia.org/wiki/Go_(programming_language). (date of last visit: April 28, 2013).

[4] Pike R., The Go Programming Language. day 1. http://golang.org/doc/{G}oCourseDay1.pdf, day 2. http://golang.org/doc/{G}oCourseDay2.pdf, day 3. http://golang.org/doc/{G}oCourseDay3.pdf, 2010. (date of last visit: April 28, 2013).

[5] Go Authors. The Go Programming Language. http://golang.org/ (date of last visit: April 28, 2013).

[6] Taylor, I., Go interfaces. http://www.airs.com/blog/archives/277, 2009. (date of last visit: April 28, 2013).

[7] Go Authors. Effective Go. http://golang.org/doc/effective_go.html, 2010. (date of last visit: April 28, 2013).

[8] Hoare, C. A. R., Communicating Sequential Processes. 2004, http://www.usingcsp.com/cspbook.pdf (date of last visit: April 28, 2013).

[9] Go Authors. The Go Programming Language. FAQ. http://golang.org/doc/go_faq.html (date of last visit: April 28, 2013).

[10] Complex event processing. http://en.wikipedia.org/wiki/Complex_event_processing. (date of last visit: April 28, 2013).

[11] Go Language Resources. Organizations Using Go. http://go-lang.cat-v.org/organizations-using-go. (date of last visit: April 28, 2013).

[12] Darrow, B., Will Go be the new go-to programming language? http://gigaom.com/2012/09/13/will-go-be-the-new-go-to-programming-language/ (date of last visit: April 28, 2013).