Adam Wołk

Anatomia języka Go

Go jest kompilowanym, współbieżnym, statycznie typowanym językiem programowania z automatycznym odśmiecaniem pamięci. Powstał pod szyldem firmy Google. Autorzy projektu są między innymi odpowiedzialni za powstanie UTF-8, systemu Unix, plan9 oraz wielu narzędzi, z których większość z nas korzysta do dzisiaj.

Go wypełnia niszę pomiędzy C, Javą oraz Pythonem, zdecydowanie przyciągając zwłaszcza programistów z tej ostatniej grupy językowej. Opublikowany w 2009 roku szybko stał się popularnym wyborem dla aplikacji back-endowych – serwerów HTTP, API czy prostych narzędzi konsolowych. Bogata biblioteka standardowa, zawierająca wysokiej jakości implementacje protokołów sieciowych (tcp, udp, http, mail, smtp, rpc), serializacji (json, xml, gob), sterowników do baz danych (sql) oraz kryptografii (wszystko napisane w czystym Go, bez odwoływania się do bibliotek napisanych w C) zdecydowanie sprawiają, że język jest konkurencyjny w tej dziedzinie aplikacji. Kompilator tworzy pojedynczy, statycznie linkowany program, który bardzo ułatwia wdrażanie aplikacji na serwerach, pomijając wszelakie problemy ze spełnianiem zależności. Pisać w Go można na systemach Linux, Mac OS X, FreeBSD, NetBSD, OpenBSD, MS Windows oraz Plan 9 dla platform i386, amd64, ARM i IBM PowerPC.

Część osób zapewne nie zdaje sobie sprawy z tego, że w ich codziennym życiu kod napisany w Go przewija się dość często. Serwis dl.google.com został przepisany z C++ na Go około roku 2013. Odpowiada on za dystrybucje aktualizacji Chrome, SDK Androida, Google Earth i wielu innych danych hostowanych na serwerach Google. W 2011 roku powstał również projekt Vitess, który odpowiada za skalowanie instancji MySQL dla YouTube. Zatem za każdym razem, gdy oglądacie film na YouTube, w tle pracę wykonuje aplikacja napisana w Go. Popularna platforma konteneryzacji – Docker również w całości została napisana w Go. Przykłady można mnożyć prawie w nieskończoność. Go nabiera tempa. Jest używany produkcyjnie przez duże firmy do niebanalnych zadań.

Listing 1. Hello world github.com/mulander/goexamples/hello

```
1 package main
2
3 import "fmt"
4
5 func main() {
6    fmt.Println("Hello, world")
7 }
```

Tradycyjnie zacznijmy przygodę z Go od *Hello world* (Listing 1).

Go to proceduralny język programowania o składni mocno zbliżonej do C. Program zapisujemy jako plik *hello*.go.

Każdy program w Go składa się z pakietów. W linii pierwszej definiujemy pakiet o nazwie `main`. Jest to specjalny pakiet, który informuje kompilator o konieczności wyprodukowania pliku wykonywalnego. W linii trzeciej pojawia się słowo kluczowe `import`. Pozwala ono wczytać inne, istniejące paczki, a potem używać ich w ramach naszego programu. Paczka `fmt` odpowiada za obsługę wejścia/wyjścia programu. Linia piąta definiuje funkcję `main`. Jest to punkt wejściowy naszego programu. W momencie uruchomienia kod funkcji `main` zostanie wykonany. Linia szósta odwołuje się do funkcji `Println` z pakietu `fmt`. Duża litera na początku wywoływanej funkcji nie jest przypadkiem.

Widocznością identyfikatorów w języku Go sterujemy za pomocą nazw. W języku nie ma plików nagłówkowych ani specjalnych instrukcji sterujących eksportem jednostek programowych. Go traktuje wszystkie identyfikatory rozpoczynające się od dużej litery jako eksportowane. Dzięki temu możemy wywołać funkcję `fmt.Println`. Nazwanie funkcji małą literą spowodowałoby, że byłaby ona prywatna dla pakietu `fmt`, więc nie mogłaby być wykorzystywana przez inne pakiety (w tym również i nasz pakiet `main`).

Pracując z Go, dysponujemy prostym, lecz niesamowicie elastycznym narzędziem go. Posiada ono szereg wbudowanych poleceń, które będziemy stopniowo poznawać.

Uruchomić nasz program możemy na kilka sposobów, np. wydając polecenie `go run hello.go`, które skompiluje nasz program, a następnie go uruchomi. Możemy również wyprodukować plik wykonywalny hello poleceniem `go build hello.go`. W przypadku tego drugiego polecenia program musimy uruchomić sami z konsoli poprzez wpisanie `./hello`. Obydwa sposoby są wygodne, jednak docelowo powinniśmy zacząć prawidłowo lokować naszą aplikację w systemie plików. W tym celu należy ustawić zmienną środowiskową `GOPATH` oraz dodanie katalogu `$GOPATH/bin` do naszej ścieżki. Na systemach z rodziny Unix możemy to wykonać trzema prostymi instrukcjami:

```
mkdir $HOME/go
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

`GOPATH` określa katalog bazowy, w którym narzędzie go będzie szukać źródeł pakietów, zbudowanych bibliotek, dokumentacji oraz miejsca, w którym umieści zainstalowane programy. Dodanie katalogu `$GOPATH/bin` do zmiennej środowiskowej `PATH` pozwala nam wygodnie odpalić nasze programy bez podawania pełnych ścieżek do katalogu bin. Cała ta przestrzeń robocza składa się z trzech istotnych katalogów:

- `$GOPATH/bin/` – zawiera programy wykonywalne, zbudowane z zainstalowanych aplikacji napisanych w Go
- `$GOPATH/pkg/` – zawiera zbudowany kod bibliotek/programów, gotowy do linkowania z naszym kodem
- `$GOPATH/src/` – zawiera kod źródłowy programów i bibliotek

Struktura ta wynika z przystosowania narzędzia go do pracy z publicznie udostępnianym kodem na popularnych serwisach typu github, code.google.com oraz bitbucket. Umożliwia też wykorzystanie własnych stron hostujących kod źródłowy. Warto stosować tę strukturę katalogów, nawet jeśli nie mamy zamiaru publikować własnych źródeł.

Przykłady z artykułu umieścimy w repozytorium https://github.com/mulander/goexamples. Utwórzmy zatem katalog `$GOPATH/src/github.com/mulander/goexamples`, natomiast pierwszy program umieśćmy w katalogu `$GOPATH/src/github.com/mulander/goexamples/hello/hello.go`. Dzięki tej strukturze możemy pobrać i zainstalować program jednym poleceniem:

```
go get github.com/mulander/goexamples/hello
```

Narzędzie go pobierze kod z repozytorium za pomocą git, skompiluje źródła z katalogu hello, czyli nasz plik `hello.go` z Listingu 1, a następnie powstały plik wykonywalny hello umieści w katalogu `$GOPATH/bin`. Pobrane i zbudowane zostaną również wszelkie zależności naszego programu.

```
$ go get github.com/mulander/goexamples/hello
$ hello
Hello, world
$ which hello
/home/mulander/go/bin/hello
```

Nie musimy oczywiście korzystać z githuba. Równie dobrze nasz program możemy umieścić w katalogu `$GOPATH/src/hello/hello.go`, a następnie budować poleceniem go `get hello`.

Zanim przejdziemy dalej warto wspomnieć o dodatkowym narzędziu w postaci go `fmt`.

Twórcy Go zdecydowali, że debaty na temat rozmieszczenia nawiasów, ilości wcięć etc. są jałowe, zatem wraz z językiem otrzymujemy go `fmt`, który formatuje cały kod pod jeden standard. Swój kod możemy sformatować w prosty sposób zakładając, iż znajduje się on w katalogu `$GOPATH/src/hello`. Wydając polecenie go `fmt hello`, cały kod znajdujący się w pakiecie zostanie poddany standardowemu formatowaniu.

W trakcie rozmów kwalifikacyjnych często zadawanym problemem jest tak zwany `FizzBuzz`. Osoba ubiegająca się o pracę ma zlecone napisanie programu, który dla każdej liczby podzielnej przez 3 drukuje `Fizz`, podzielnej przez 5 drukuje `Buzz`, a dla liczb zarówno podzielnych przez 3, jak i 5 drukuje `FizzBuzz`. W pozostałych przypadkach drukuje liczbę. Utwórzmy więc nowy katalog `github.com/mulander/goexamples/fizzbuzz`, który będzie biblioteką implementującą ów problem. Listing 2 prezentuje przykładową implementację:

Listing 2. github.com/mulander/goexamples/fizzbuzz

```
1    package fizzbuzz
2
3    import "fmt"
4
5    func Print(count int) {
6       for i := 1; i <= count; i++ {
7          switch {
8          case i%15 == 0:
9             fmt.Println("FizzBuzz")
10      case i%3 == 0:
11         fmt.Println("Fizz")
12      case i%5 == 0:
13         fmt.Println("Buzz")
14      default:
15         fmt.Println(i)
16      }
17    }
18   }
```

Napiszemy również program wykorzystujący nową bibliotekę (Listing 3):

Listing 3. github.com/mulander/goexamples/fizzuser

```
1  package main
2
3  import "github.com/mulander/goexamples/fizzbuzz"
4
5  func main() {
6    fizzbuzz.Print(30)
7  }
8
```

Zacznijmy od listingu numer dwa.

Wspominaliśmy już, że każdy program w Go posiada pakiet. W linii pierwszej zadeklarowaliśmy pakiet o nazwie `fizzbuzz`. W linii trzeciej importujemy pakiet `fmt`, aby mieć dostęp do funkcji `Println`, drukującej tekst, kończąc go znakiem nowej linii. Następnie w linii 5 deklarujemy funkcję o nazwie `Print`. Zwracam uwagę na dużą literę na początku nazwy. Ta funkcja jest eksportowana poza pakiet `fizzbuzz`.

Funkcja `fizzbuzz.Print` przyjmuje jeden parametr wejściowy `count` o typie `int`. W przeciwieństwie do C typ zmiennej pojawia się po deklaracji jej nazwy, a nie przed. Funkcja nie zwraca żadnej wartości. Jeżeli istniałaby wartość zwrotna, to pojawiłaby się ona po deklaracji listy parametrów, co również jest przeciwieństwem do języka C.

W linii 6 programu pojawia się pętla `for`. W Go nie znajdziemy innego rodzaju pętli. Podstawowa forma jest analogiczna do C. Pierwszy parametr jest inicjalizacją zmiennej, drugi wykonywanym testem, trzeci zaś krokiem wykonywanym po przebiegu pętli. Możemy pomijać poszczególne kroki pozostawiając je puste. Przykładowo pętla z Listingu 4 wykona się tylko raz, a pętla z Listingu 5 będzie wykonywana w nieskończoność.

Listing 4. Pętla z jednym przebiegiem

```
1    for i := 0; i < 10; {
2       fmt.Println("Hello")
3       i = 10
4    }
```

Listing 5. Pętla nieskończona

```
1    for  {
2       fmt.Println("Hello")
3    }
```

Inicjalizacja zasługuje na specjalną uwagę. Składnia `:=` oznacza inferencje typu. Oznacza to, że Go na podstawie wartości podanej po prawej stronie wyrażenia określi typ zmiennej po jego lewej stronie. Ponieważ wartość `1` zostaje określona jako typ `int`, całą deklarację można czytać jako `var i int = 1`. Tak więc na podstawie naszego przykładowego programu z Listingu 3 nasza pętla wykona się 30 razy.

W linii 7 listingu trzeciego pojawia się instrukcja `switch`. Zastosowana wersja bez wskazania zmiennej jest wygodnym sposobem zaprogramowania typowego ciągu `if/else if/else if/else` w bardziej skompresowany sposób. Nasze warunki w instrukcji `switch` są dość proste. Testują wartość zmiennej, wykonując na niej operacje dzielenia modulo, gdzie warunek zostaje spełniony, jeżeli dzielenie zostało wykonane bez reszty. Tym sposobem najmniejszy wspólny dzielnik dla liczb 3 i 5 (15) powoduje wydrukowanie ciągu znaków `FizzBuzz`, dzielnik liczby 3 wydrukowanie `Fizz`, zaś dzielnik liczby 5 wydrukowanie `Buzz`. W pozostałych przypadkach zadziała `default`, drukując po prostu samą liczbę. Listing 6 przedstawia inną formę instrukcji `switch`, która pokazuje dodatkowo, że przełączenia można dokonywać nie tylko na typach numerycznych.

Listing 6. Instrukcja switch

```
1    x := "golang"
2    switch x {
3    case "golang":
4       fmt.Println("X was golang")
5    default:
6       fmt.Println("X was not golang")
7    }
```

Listing trzeci demonstruje prosty program wykorzystujący naszą bibliotekę. Zdecydowanie najbardziej ciekawa jest linia 3, która tę bibliotekę importuje. Wprawdzie nasz program również znajduje się w katalogu `github.com/mulander/goexamples`, lecz nie ma to większego znaczenia. Dzięki zastosowanej strukturze katalogów narzędzie go wie, że ma szukać naszego programu w katalogu `$GOPATH/src/github.com/mulander/goexamples/fizzbuzz`. Jeżeli katalog ten nie zostanie odnaleziony na dysku, go pobierze repozytorium za pomocą git. Jest to niesamowicie wygodne rozwiązanie, które znacząco ułatwia zarówno publikowanie małych bibliotek z naszych większych projektów, jak i wykorzystywanie kodu napisanego przez innych.

Linia 6 z kolei demonstruje, jak wywołać funkcję przez nas wyeksportowaną. Domyślnie do zaimportowanego pakietu możemy się odwoływać za pomocą ostatniej nazwy w ścieżce, w naszym przykładzie `fizzbuzz`. Jeżeli nie pasuje nam nazwa biblioteki lub powoduje ona konflikt z pakietami w naszym kodzie, to możemy dokonać jej przemianowania na wybraną przez nas nazwę.

```
import fb "github.com/mulander/goexamples/fizzbuzz"
```

Po podaniu własnej nazwy przed ścieżką importu możemy za jej pomocą odwoływać się do eksportowanych jednostek pakietu, przykładowo: `fb.Print(30)`.

Na tym etapie warto zacząć wykorzystywać kolejne narzędzie, jakim jest go vet. Badając źródła naszego programu, go vet jest w stanie wskazać podejrzane konstrukcje językowe, takie jak wywołania funkcji Printf, gdzie argumenty nie są dopasowane do maski formatu. Warto jednak pamiętać, że narzędzie wykorzystuje heurystykę, czyli nie mamy gwarancji, że wszystkie zgłoszone problemy są faktycznymi błędami, choć dzięki temu podejściu narzędzie może znaleźć błędy, jakich nie zgłasza sam kompilator języka.

Zbudujmy więc program z listingu trzeciego, wykonując polecenie go get github.com/mulander/goexamples/fizzuser. Wynik programu możemy zaobserwować na Listingu 7:

Listing 7. Uruchomienie fizzuser

```
$ fizzuser
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
```

Warto zauważyć, że nie musimy osobno budować biblioteki. Narzędzie go doskonale wie, gdzie je znaleźć i zrobi to za nas.

Przyjęte rozwiązanie ma pewne wady. Trudno mówić o reużywalności kodu, jeżeli jego jedynym efektem jest wydrukowanie tekstu na ekran. Testowanie takiego programu nie należy również do najwygodniejszych. Dokonajmy więc drobnej modyfikacji biblioteki z listingu drugiego.

Listing 8. github.com/mulander/goexamples/fizzbuzz

```
1   package fizzbuzz
2
3   import (
4    "fmt"
5    "log"
6   )
7
8   func Generate(count int) ([]string, error) {
9    if count <= 0 {
10          return nil, fmt.Errorf("fizzbuzz: Negative fizzbuzz
count provided")
11      }
12
13      fizzbuzz := make([]string, count)
14
15      var output string
16      for i := 1; i <= count; i++ {
17         switch {
18         case i%15 == 0:
19            output = "FizzBuzz"
20         case i%3 == 0:
21            output = "Fizz"
22         case i%5 == 0:
23            output = "Buzz"
24         default:
25            output = fmt.Sprintf("%d", i)
26         }
27         fizzbuzz[i-1] = output
28      }
29
30      return fizzbuzz, nil
31  }
32
33  func Print(count int) {
34     fizzbuzz, err := Generate(count)
35     if err != nil {
36        log.Fatal(err)
37     }
38
39     for _, entry := range fizzbuzz {
40        fmt.Println(entry)
41     }
42  }
```

Listing 8 prezentuje zaktualizowany kod. Wydzielamy osobną funkcję o nazwie Generate. Jest ona eksportowana, ponieważ jej nazwa zaczyna się od dużej litery. Wraz ze zmianą kodu pojawiają się pewne nowe struktury językowe. Pierwsza zmiana pojawia się już w linii 3. Jeżeli mamy potrzebę zaimportowania więcej niż jednej biblioteki, to zamykając ich nazwy w nawiasach, możemy podać ich większą liczbę. Możemy oczywiście polegać na go fmt, które za nas posortuje alfabetycznie wszystkie importy. Dobrym zwyczajem jest jednak oddzielenie prywatnych od wbudowanych bibliotek języka. W przypadku kodu z Listingu 8 takiej potrzeby nie ma. Jeżeli jednak takowa się pojawi, to nasze biblioteki oddzielimy od systemowych pustą linią tekstu.

W linii 8 deklarujemy funkcję Generate przyjmującą wartość count int i zwracającą dwie wartości. Pierwszą []string, która będzie zawierać sekwencję FizzBuzz dla podanego parametru, oraz drugą error. Zatrzymajmy się na chwilę. Go pozwala na definicje funkcji, które zwracają więcej niż jedną wartość. Jest to niesamowicie wygodny mechanizm, który zapewne rozpoznają programiści z bagażem doświadczenia z innych języków.

Definicja funkcji w Go ma dość prostą składnię:

```
func nazwa(parametr typ, parametr typ) (a typ, b typ)
```

Możemy stosować skrócenie definicji, pisząc parametr1, parametr2 typ, jeżeli obydwa parametry są tego samego typu. Taka sama zasada stosowana jest dla wartości zwracanych przez funkcję. Podobnie jak w poprzednich przykładach (definicja zmiennych), Go stosuje odwrotną notację w stosunku do języka C. Wartości zwracane przez funkcję są definiowane po jej liście parametrów, a nie tak jak w przypadku C – przed nazwą samej funkcji.

Sygnatura funkcji Generate z linii 8 nie definiuje tymczasem nazw dla parametrów zwrotnych:

```
func Generate(count int) ([]string, error) {
```

Są one opcjonalnym, ale bardzo wygodnym mechanizmem. Jeżeli przy definicji podamy nazwę dla parametrów, przykładowo err error, to nie będziemy zmuszani do definiowania jej wewnątrz ciała funkcji, a same instrukcje return będą przyjmować, że zwracamy nazwane parametry. Na Listingu 9 możemy zobaczyć funkcję korzystającą z tego mechanizmu:

Listing 9. Nazwane wartości zwrotne

```
1   package main
2
3   import "fmt"
4
5   func test() (a, b int) {
6      a = 1
7      b = 2
8      return 9    }
10
11  func main() {
12     a, b := test()
13     fmt.Printf("a=%d b=%d\n", a, b)
14  }
```

Dzięki takiej konstrukcji funkcja nadal zwraca dwie wartości a i b, lecz nigdzie