Język Go
Poznaj i programuj

Alan A.A. Donovan
Brian W. Kernighan

Helion

Tytuł oryginału: The Go Programming Language

Tłumaczenie: Lech Lachowski

ISBN: 978-83-283-2467-1

Authorized translation from the English language edition, entitled: THE GO PROGRAMMING LANGUAGE; ISBN 0134190440; by Alan A. A. Donovan; and Brian W. Kernighan; published by Pearson Education, Inc, publishing as Addison Wesley Professional.
Copyright © 2016 Alan A. A. Donovan & Brian W. Kernighan.

All rights reserved. No part of this book may by reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
Polish language edition published by HELION SA. Copyright © 2016.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jgopop
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jgopop.zip

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Rozdział 7

Interfejsy

Typy interfejsowe wyrażają uogólnienia lub abstrakcje dotyczące zachowań innych typów. Dzięki uogólnianiu interfejsy pozwalają pisać funkcje, które są bardziej elastyczne i adaptowalne, ponieważ nie są związane ze szczegółami jednej konkretnej implementacji.

Wiele języków obiektowych ma jakąś koncepcję interfejsów, ale interfejsy języka Go wyróżnia to, że ich warunki są **spełniane pośrednio**. Innymi słowy: nie ma potrzeby deklarowania wszystkich interfejsów, których warunki spełnia konkretny typ. Wystarczy po prostu posiadanie niezbędnych metod. Taka konstrukcja pozwala na tworzenie nowych interfejsów, których warunki są spełniane przez istniejące konkretne typy bez ich zmieniania, co jest szczególnie przydatne dla typów zdefiniowanych w niekontrolowanych przez Ciebie pakietach.

Ten rozdział rozpoczniemy od przyjrzenia się podstawowym mechanizmom typów interfejsowych i ich wartościom. Potem przestudiujemy kilka ważnych interfejsów ze standardowej biblioteki — wielu programistów języka Go korzysta ze standardowych interfejsów w równym stopniu jak ze swoich własnych. Na koniec przyjrzymy się **asercjom typów** (zob. podrozdział 7.10) oraz **przełącznikom typów** (zob. podrozdział 7.13) i zobaczymy, w jaki sposób umożliwiają stosowanie innego rodzaju ogólności.

7.1. Interfejsy jako kontrakty

Wszystkie omawiane do tej pory typy były **typami konkretnymi**. Typ konkretny określa dokładną reprezentację swoich wartości i udostępnia wewnętrzne operacje tej reprezentacji, takie jak arytmetyka dla liczb albo indeksowanie, `append` i `range` dla wycinków. Konkretny typ może również zapewniać dodatkowe zachowania poprzez swoje metody. Gdy masz wartość konkretnego typu, wiesz dokładnie, czym ona **jest** i co możesz z nią **zrobić**.

W języku Go istnieje jeszcze inny rodzaj typu, zwany **typem interfejsowym**. Interfejs jest **typem abstrakcyjnym**. Nie udostępnia reprezentacji czy wewnętrznej struktury swoich wartości ani też zbioru podstawowych obsługiwanych operacji. Ujawnia tylko niektóre swoje metody. Gdy masz wartość typu interfejsowego, nie wiesz nic na temat tego, czym ona **jest**. Wiesz tylko, co może **robić**, lub, mówiąc bardziej precyzyjnie, jakie zachowania są dostarczane przez jej metody.

Dotychczas używaliśmy dwóch podobnych funkcji do formatowania łańcuchów znaków: `fmt.Printf`, która zapisuje wynik do standardowego strumienia wyjściowego (pliku), oraz `fmt.Sprintf`, która zwraca wynik jako `string`. Byłoby niedobrze, gdyby ta trudna część, jaką jest formatowanie wyniku,

Kup książkę Poleć książkę

musiała być duplikowana ze względu na te drobne różnice w sposobie wykorzystywania wyniku. Dzięki interfejsom tak nie jest. Obie te funkcje są w istocie funkcjami opakowującymi trzecią funkcję, `fmt.Fprintf`, która jest neutralna w kwestii tego, co się dzieje z obliczanym przez nią wynikiem:

```
package fmt

func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)

func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}

func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

Prefiks `F` w nazwie `Fprintf` oznacza **plik** (ang. *file*) i wskazuje, że sformatowane dane wyjściowe powinny zostać zapisane w pliku dostarczonym jako pierwszy argument. W przypadku `Printf` tym argumentem (`os.Stdout`) jest `*os.File`. Jednak w przypadku `Sprintf` tym argumentem nie jest plik, chociaż pozornie go przypomina: `&buf` jest wskaźnikiem do bufora pamięci, w którym mogą być zapisywane bajty.

Pierwszym parametrem funkcji `Fprintf` również nie jest plik. Jest nim `io.Writer`, czyli typ interfejsowy z następującą deklaracją:

```
package io

// Writer jest interfejsem, który opakowuje podstawową metodę Write.
type Writer interface {
    // Write zapisuje len(p) bajtów ze zmiennej p do bazowego strumienia danych.
    // Zwraca liczbę bajtów zapisanych z p (0 <= n <= len(p))
    // oraz każdy napotkany błąd, który spowodował przedwczesne zatrzymanie zapisywania.
    // Metoda Write musi zwracać błąd niebędący nil, jeśli zwraca n < len(p).
    // Metoda Write nie może modyfikować danych wycinka, nawet tymczasowo.
    //
    // Implementacje nie mogą przechowywać w pamięci zmiennej p.
    Write(p []byte) (n int, err error)
}
```

Interfejs `io.Writer` definiuje kontrakt między funkcją `Fprintf` a wywołującymi ją podmiotami. Z jednej strony, kontrakt wymaga, aby podmiot wywołujący zapewnił wartość konkretnego typu, takiego jak `*os.File` lub `*bytes.Buffer`, który ma metodę o nazwie `Write` z odpowiednimi sygnaturą i zachowaniem. Z drugiej strony, kontrakt gwarantuje, że funkcja `Fprintf` będzie wykonywać swoje zadania, mając daną dowolną wartość, która spełnia warunki interfejsu `io.Writer`. Funkcja `Fprintf` nie może zakładać, że zapisuje w pliku lub pamięci, tylko że może wywołać metodę `Write`.

Ponieważ funkcja `fmt.Fprintf` nie zakłada niczego w kwestii reprezentacji wartości i opiera się tylko na zachowaniach gwarantowanych przez kontrakt `io.Writer`, możemy do tej funkcji bezpiecznie przekazywać jako pierwszy argument wartość dowolnego typu konkretnego, który spełnia warunki interfejsu `io.Writer`. Ta swoboda zastępowania jednego typu innym typem, który spełnia warunki tego samego interfejsu, nazywa się **podstawialnością** i jest charakterystyczną cechą programowania obiektowego.

Kup książkę Poleć książkę

Sprawdźmy to przy użyciu nowego typu. Przedstawiona poniżej metoda `Write` typu `*ByteCounter` jedynie zlicza zapisane w nim bajty przed ich porzuceniem. (Wymagana jest konwersja, aby zapewnić dopasowanie typów dla `len(p)` i `*c` w instrukcji przypisania `+=`).

code/r07/bytecounter

```
type ByteCounter int

func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // konwersja int na ByteCounter
    return len(p), nil
}
```

Ponieważ `*ByteCounter` spełnia warunki kontraktu `io.Writer`, możemy przekazać go do funkcji `Fprintf`, która przeprowadza formatowanie swojego łańcucha znaków bez świadomości tej zmiany. `ByteCounter` poprawnie akumuluje długość wyniku.

```
var c ByteCounter
c.Write([]byte("witaj"))
fmt.Println(c) // "5", = len("witaj")

c = 0 // resetowanie licznika
var name = "Marta"
fmt.Fprintf(&c, "witaj, %s", name)
fmt.Println(c) // "12", = len("witaj, Marta")
```

Poza `io.Writer` istnieje jeszcze inny bardzo ważny interfejs dla pakietu `fmt`. Funkcje `Fprintf` i `Fprintln` zapewniają typom możliwość kontrolowania sposobu, w jaki wyświetlane są ich wartości. W podrozdziale 2.5 zdefiniowaliśmy metodę `String` dla typu `Celsius`, aby temperatury były wyświetlane jako `"100 °C"`, a w podrozdziale 6.5 wyposażyliśmy typ `*IntSet` w metodę `String`, aby zbiory były przedstawiane z wykorzystaniem tradycyjnej notacji, np. `"{1 2 3}"`. Zadeklarowanie metody `String` sprawia, że dany typ spełnia warunki jednego z najszerzej stosowanych interfejsów, czyli interfejsu `fmt.Stringer`:

```
package fmt

// String jest wykorzystywana do wyświetlania wartości przekazywanych
// jako operand do dowolnego formatu, który akceptuje łańcuch znaków,
// lub do funkcji wypisywania bez ustawionego formatu, takiej jak Print.
type Stringer interface {
    String() string
}
```

W podrozdziale 7.10 wyjaśnimy, w jaki sposób pakiet `fmt` wykrywa, które wartości spełniają warunki tego interfejsu.

Ćwiczenie 7.1. Wykorzystując pomysły z `ByteCounter`, zaimplementuj liczniki dla słów i linii. Przydać Ci się może funkcja `bufio.ScanWords`.

Ćwiczenie 7.2. Napisz funkcję `CountingWriter` (z przedstawioną poniżej sygnaturą), która mając dany interfejs `io.Writer`, zwraca nowy `Writer`, opakowujący oryginalny, oraz wskaźnik do zmiennej `int64`, która w każdym momencie zawiera liczbę bajtów aktualnie zapisanych do nowego interfejsu `Writer`.

```
func CountingWriter(w io.Writer) (io.Writer, *int64)
```

Ćwiczenie 7.3. Napisz metodę `String` dla typu `*tree` z programu `code/r04/treesort` (zob. podrozdział 4.4), która ujawnia sekwencję wartości w drzewie.

Kup książkę Poleć książkę

7.2. Typy interfejsowe

Typ interfejsowy określa zestaw metod, które musi posiadać typ konkretny, aby został uznany za instancję danego interfejsu.

Typ `io.Writer` jest jednym z najszerzej stosowanych interfejsów, ponieważ zapewnia abstrakcję wszystkich typów, do których mogą być zapisywane bajty, co obejmuje: pliki, bufory pamięci, połączenia sieciowe, klienty HTTP, programy archiwizujące, programy szyfrujące itd. Wiele innych użytecznych interfejsów definiuje pakiet `io`. Interfejs `Reader` reprezentuje dowolny typ, z którego można odczytywać bajty, a `Closer` jest dowolną wartością, którą można zamknąć, taką jak plik lub połączenie sieciowe. (Prawdopodobnie dostrzegłeś już konwencję nazewnictwa stosowaną dla wielu posiadających jedną metodę interfejsów języka Go).

```
package io

type Reader interface {
    Read(p []byte) (n int, err error)
}

type Closer interface {
    Close() error
}
```

Jeśli poszukamy dalej, znajdziemy deklaracje nowych typów interfejsów jako kombinacje istniejących. Oto dwa przykłady:

```
type ReadWriter interface {
    Reader
    Writer
}

type ReadWriteCloser interface {
    Reader
    Writer
    Closer
}
```

Użyta powyżej składnia, przypominająca osadzanie struktur, pozwala nam określać kolejny interfejs jako skrót służący do wypisywania wszystkich jego metod. Nazywa się to **osadzaniem** interfejsu. Moglibyśmy zapisać interfejs `io.ReadWriter` bez osadzania (chociaż mniej zwięźle) w następujący sposób:

```
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Write(p []byte) (n int, err error)
}
```

Moglibyśmy nawet pomieszać te dwa style:

```
type ReadWriter interface {
    Read(p []byte) (n int, err error)
    Writer
}
```

Wszystkie trzy deklaracje mają ten sam efekt. Kolejność, w jakiej pojawiają się metody, jest nieistotna. Znaczenie ma jedynie zestaw metod.

Kup książkę Poleć książkę

Ćwiczenie 7.4. Funkcja `strings.NewReader` zwraca wartość, która spełnia warunki interfejsu `io.Reader` (i innych), odczytując z jego argumentu, czyli łańcucha znaków. Zaimplementuj prostą wersję funkcji `NewReader` i użyj jej, aby umożliwić przyjmowanie przez parser HTML (zob. podrozdział 5.2) danych wejściowych z łańcucha znaków.

Ćwiczenie 7.5. Funkcja `LimitReader` z pakietu `io` akceptuje argument `r` interfejsu `io.Reader` oraz liczbę bajtów `n`, a następnie zwraca kolejny `Reader`, który odczytuje z argumentu `r`, ale zgłasza stan końca pliku po `n` bajtach. Zaimplementuj ją.

```
func LimitReader(r io.Reader, n int64) io.Reader
```

7.3. Spełnianie warunków interfejsu

Typ **spełnia** warunki interfejsu, jeśli ma wszystkie metody, których wymaga dany interfejs. Przykładowo: typ `*os.File` spełnia warunki interfejsów `io.Reader`, `Writer`, `Closer` i `ReadWriter`. Typ `*bytes.Buffer` spełnia warunki interfejsów `Reader`, `Write` i `ReadWriter`, ale nie spełnia warunków interfejsu `Closer`, ponieważ nie ma metody `Close`. Programiści języka Go często stosują skrót myślowy, mówiąc, że konkretny typ „jest" określonym typem interfejsowym, co oznacza, że spełnia warunki tego interfejsu, np. `*bytes.Buffer` „jest" typem `io.Writer`, a `*os.File` „jest" typem `io.ReadWriter`.

Reguła przypisywalności (zob. punkt 2.4.2) dla interfejsów jest bardzo prosta: wyrażenie może być przypisane do interfejsu tylko wtedy, gdy spełnia warunki tego interfejsu. Więc:

```
var w io.Writer
w = os.Stdout           // OK: *os.File ma metodę Write
w = new(bytes.Buffer)   // OK: *bytes.Buffer ma metodę Write
w = time.Second         // błąd kompilacji: time.Duration nie ma metody Write

var rwc io.ReadWriteCloser
rwc = os.Stdout         // OK: *os.File ma metody Read, Write i Close
rwc = new(bytes.Buffer) // błąd kompilacji: *bytes.Buffer nie ma metody Close
```

Ta reguła ma zastosowanie nawet wtedy, gdy prawa strona sama jest interfejsem:

```
w = rwc                 // OK: io.ReadWriteCloser ma metodę Write
rwc = w                 // błąd kompilacji: io.Writer nie ma metody Close
```

Ponieważ `ReadWriter` i `ReadWriteCloser` obejmują wszystkie metody interfejsu `Writer`, każdy typ spełniający warunki interfejsu `ReadWriter` lub `ReadWriteCloser` siłą rzeczy spełnia warunki interfejsu `Writer`.

Zanim przejdziemy dalej, powinniśmy wyjaśnić kwestię tego, co znaczy, że jakiś typ ma metodę. Przypomnijmy z punktu 6.2, że dla każdego nazwanego typu konkretnego `T` niektóre jego metody same posiadają odbiornik typu `T`, podczas gdy inne wymagają wskaźnika `*T`. Przypomnijmy również, że prawidłowe jest wywołanie metody `*T` na argumencie typu `T`, pod warunkiem że ten argument jest **zmienną**. Kompilator pośrednio pobiera jej adres. Ale to jest zwykły lukier składniowy: wartość typu `T` nie posiada wszystkich metod, które ma wskaźnik `*T`, w wyniku czego może spełniać wymagania mniejszej liczby interfejsów.

Wyjaśnimy to na przykładzie. Metoda `String` typu `IntSet` z podrozdziału 6.5 wymaga odbiornika wskaźnikowego, więc nie możemy wywołać tej metody na nieadresowalnej wartości `IntSet`:

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string

var _ = IntSet{}.String() // błąd kompilacji: String wymaga odbiornika *IntSet
```

Kup książkę Poleć książkę

Możemy jednak wywołać ją na zmiennej `IntSet`:

```
var s IntSet
var _ = s.String() // OK: s jest zmienną, a &s ma metodę String
```

Ponieważ jednak tylko `*IntSet` ma metodę `String`, tylko `*IntSet` spełnia warunki interfejsu `fmt.Stringer`:

```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s  // błąd kompilacji: IntSet nie ma metody String
```

Podrozdział 12.8 zawiera program wyświetlający metody dowolnej wartości, a narzędzie `godoc -analysis=type` (zob. punkt 10.7.4) wyświetla metody każdego typu i relacje pomiędzy interfejsami i typami konkretnymi.

Tak jak koperta opakowuje i ukrywa przechowywany list, tak interfejs opakowuje i ukrywa typ konkretny i przechowywaną przez niego wartość. Wywoływane mogą być tylko metody ujawnione przez typ interfejsowy, nawet jeśli dany typ konkretny ma jeszcze inne:

```
os.Stdout.Write([]byte("witaj")) // OK: *os.File ma metodę Write
os.Stdout.Close()                // OK: *os.File ma metodę Close

var w io.Writer
w = os.Stdout
w.Write([]byte("witaj")) // OK: io.Writer ma metodę Write
w.Close()                // błąd kompilacji: io.Writer nie ma metody Close
```

Interfejs z większą liczbą metod (taki jak `io.ReadWriter`) daje nam więcej informacji o wartościach, jakie zawiera, i stawia większe wymagania dotyczące typów, które go implementują, niż interfejs z mniejszą liczbą metod (taki jak `io.Reader`). Jakie więc informacje daje nam typ `interface{}`, który w ogóle nie ma żadnych metod, na temat typów konkretnych spełniających jego warunki?

Zgadza się: nie daje żadnych. Może się to wydawać bezużyteczne, ale w rzeczywistości typ `interface{}`, zwany **pustym typem interfejsowym**, jest nieodzowny. Ponieważ pusty typ interfejsowy nie stawia żadnych wymagań dotyczących typów spełniających jego warunki, możemy do niego przypisać **dowolną** wartość.

```
var any interface{}
any = true
any = 12.34
any = "witaj"
any = map[string]int{"jeden": 1}
any = new(bytes.Buffer)
```

Chociaż nie było oczywiste, używaliśmy typu pustego interfejsu już od pierwszego przykładu przedstawionego w tej książce, ponieważ pozwala on funkcjom takim jak `fmt.Println` lub `errorf` z podrozdziału 5.7 akceptować argumenty dowolnego typu.

Oczywiście gdy utworzymy wartość `interface{}` zawierającą wartość logiczną, liczbę zmiennoprzecinkową, łańcuch znaków, mapę, wskaźnik lub dowolny inny typ, nie możemy bezpośrednio nic zrobić z tą przechowywaną przez niego wartością, ponieważ ten interfejs nie ma metod. Musimy znaleźć sposób, aby ponownie wydobyć tę wartość. W podrozdziale 7.10 zobaczymy, jak to zrobić przy użyciu **asercji typów**.

Ponieważ spełnienie warunków interfejsu zależy tylko od metod dwóch zaangażowanych w to typów, nie ma potrzeby deklarowania relacji między typem konkretnym a interfejsem, którego warunki on spełnia. Mimo to czasami przydatne jest udokumentowanie i założenie danej relacji, gdy jest ona zamierzona, ale w żaden inny sposób nie jest egzekwowana przez program. Poniższa

Kup książkę Poleć książkę

deklaracja stwierdza w czasie kompilacji, że wartość typu `*bytes.Buffer` spełnia warunki interfejsu `io.Writer`:

```
// Typ *bytes.Buffer musi spełniać warunki interfejsu io.Writer.
var w io.Writer = new(bytes.Buffer)
```

Nie musimy alokować nowej zmiennej, ponieważ nada się każda wartość typu `*bytes.Buffer`, nawet `nil`, którą zapisujemy jako `(*bytes.Buffer)(nil)`, używając konwersji bezpośredniej. A ponieważ nigdy nie zamierzamy odwoływać się do zmiennej `w`, możemy zastąpić ją pustym identyfikatorem. Wszystkie te zmiany zebrane razem dają nam ten bardziej oszczędny wariant:

```
// Typ *bytes.Buffer musi spełniać warunki interfejsu io.Writer.
var _ io.Writer = (*bytes.Buffer)(nil)
```

Warunki niepustych typów interfejsowych, takich jak `io.Writer`, są najczęściej spełniane przez typ wskaźnika, w szczególności gdy jedna z metod tego interfejsu lub kilka z nich zakłada pewien rodzaj mutacji odbiornika, tak jak metoda `Write`. Wskaźnik do struktury jest szczególnie popularnym typem przenoszącym metody.

Jednak typy wskaźników wcale nie są jedynymi typami, które spełniają warunki interfejsów, i nawet warunki interfejsów z metodami modyfikującymi mogą być spełnione przez jeden z innych typów referencyjnych języka Go. Widzieliśmy przykłady typów wycinka z metodami (`geometry.Path`, podrozdział 6.1) i typów map z metodami (`url.Values`, punkt 6.2.1), a później zobaczymy typ funkcji z metodami (`http.HandlerFunc`, podrozdział 7.7). Nawet podstawowe typy mogą spełniać warunki interfejsów. Jak zobaczymy w podrozdziale 7.4, typ `time.Duration` spełnia warunki interfejsu `fmt.Stringer`.

Typ konkretny może spełniać warunki wielu niepowiązanych interfejsów. Rozważmy program, który organizuje lub sprzedaje cyfrowe artefakty kulturowe, takie jak muzyka, filmy i książki. Może on definiować następujący zestaw typów konkretnych:

```
Album
Book
Movie
Magazine
Podcast
TVEpisode
Track
```

Każdą interesującą nas abstrakcję możemy wyrazić jako interfejs. Niektóre właściwości są wspólne dla wszystkich artefaktów, np.: tytuł, data utworzenia oraz lista twórców (autorów lub artystów).

```
type Artifact interface {
    Title() string
    Creators() []string
    Created() time.Time
}
```

Inne właściwości są ograniczone do określonych typów artefaktów. Właściwości słowa drukowanego dotyczą tylko książek i czasopism, podczas gdy rozdzielczość mają tylko filmy i seriale telewizyjne.

```
type Text interface {
    Pages() int
    Words() int
    PageSize() int
}

type Audio interface {
    Stream() (io.ReadCloser, error)
```

Kup książkę Poleć książkę

```
    RunningTime() time.Duration
    Format() string // np. "MP3", "WAV"
}
type Video interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string // np. "MP4", "WMV"
    Resolution() (x, y int)
}
```

Interfejsy są tylko jednym z wielu sposobów grupowania powiązanych typów konkretnych i wyrażania ich wspólnych aspektów. Inne sposoby grupowania poznamy później. Jeśli okazałoby się np., że potrzebujemy obsługiwać elementy `Audio` i `Video` w taki sam sposób, moglibyśmy zdefiniować interfejs `Streamer` służący do reprezentowania ich wspólnych aspektów bez zmiany istniejących deklaracji typów.

```
type Streamer interface {
    Stream() (io.ReadCloser, error)
    RunningTime() time.Duration
    Format() string
}
```

Każde grupowanie typów konkretnych oparte na ich wspólnych zachowaniach może być wyrażone jako typ interfejsowy. W przeciwieństwie do języków opartych na klasach, w których zestaw interfejsów o warunkach spełnianych przez jakąś klasę jest wyraźnie określony, w języku Go możemy definiować nowe abstrakcje lub grupy interesów w razie potrzeby, bez modyfikacji deklaracji typu konkretnego. Jest to szczególnie przydatne, gdy dany typ konkretny pochodzi z pakietu napisanego przez innego autora. Oczywiście muszą istnieć bazowe podobieństwa w tych typach konkretnych.

7.4. Parsowanie flag za pomocą interfejsu flag.Value

W tym podrozdziale zobaczymy, jak inny standardowy interfejs, `flag.Value`, pomaga nam definiować nowe notacje dla flag wiersza poleceń. Rozważmy poniższy program, który zostaje uśpiony na określony czas.

code/r07/sleep

```
var period = flag.Duration("period", 1*time.Second, "sleep period")

func main() {
    flag.Parse()
    fmt.Printf("Śpi przez %v...", *period)
    time.Sleep(*period)
    fmt.Println()
}
```

Zanim program zostanie uśpiony, wyświetla przedział czasu. Pakiet `fmt` wywołuje metodę `String` typu `time.Duration`, żeby wyświetlić przedział czasu, ale nie w nanosekundach, tylko w sposób przyjazny dla użytkownika:

```
$ go build code/r07/sleep
$ ./sleep
Śpi przez 1s...
```

Domyślnym okresem uśpienia jest jedna sekunda, ale można go kontrolować za pomocą flagi wiersza poleceń `-period`. Funkcja `flag.Duration` tworzy zmienną flagi o typie `time.Duration` i pozwala użytkownikowi określać czas trwania uśpienia w różnych przyjaznych dla użytkownika

Kup książkę Poleć książkę

formatach, również w tej samej notacji, która jest wyświetlana przez metodę `String`. Ta symetria rozwiązania pozwala uzyskać miły interfejs użytkownika.

```
$ ./sleep -period 50ms
Śpi przez 50ms...
$ ./sleep -period 2m30s
Śpi przez 2m30s...
$ ./sleep -period 1.5h
Śpi przez 1h30m0s...
$ ./sleep -period "1 dzień"
invalid value "1 dzień" for flag -period: time: unknown unit dzień in duration 1 dzień
```

Ponieważ flagi czasu trwania są tak przydatne, ta funkcja jest wbudowana w pakiet `flag`, ale łatwo jest zdefiniować nową notację flag dla własnych typów danych. Trzeba tylko zdefiniować typ spełniający warunki interfejsu `flag.Value`, którego deklaracja została przedstawiona poniżej:

```
package flag

// Value jest interfejsem dla wartości przechowywanej we fladze.
type Value interface {
    String() string
    Set(string) error
}
```

Metoda `String` formatuje wartość flagi do wykorzystywania w komunikatach pomocy wiersza poleceń. Zatem każdy interfejs `flag.Value` jest również interfejsem `fmt.Stringer`. Metoda `Set` parsuje swój argument w postaci łańcucha znaków i aktualizuje wartość flagi. W efekcie metoda `Set` stanowi odwrotność metody `String` i jest dobrą praktyką, aby obie te metody korzystały z tej samej notacji.

Zdefiniujmy typ `celsiusFlag`, który pozwala określać temperaturę w stopniach Celsjusza lub Fahrenheita z odpowiednią konwersją. Należy zwrócić uwagę, że `celsiusFlag` osadza typ `Celsius` (zob. podrozdział 2.5), a tym samym uzyskuje za darmo metodę `String`. Aby spełnić warunki interfejsu `flag.Value`, trzeba tylko zadeklarować metodę `Set`:

code/r07/tempconv

```
// Typ *celsiusFlag spełnia warunki interfejsu flag.Value.
type celsiusFlag struct{ Celsius }

func (f *celsiusFlag) Set(s string) error {
    var unit string
    var value float64
    fmt.Sscanf(s, "%f%s", &value, &unit) // nie potrzeba kontroli błędów
    switch unit {
    case "C", "°C":
        f.Celsius = Celsius(value)
        return nil
    case "F", "°F":
        f.Celsius = FToC(Fahrenheit(value))
        return nil
    }
    return fmt.Errorf("nieprawidłowa temperatura %q", s)
}
```

Wywołanie funkcji `fmt.Sscanf` parsuje liczbę zmiennoprzecinkową (`value`) i łańcuch znaków (`unit`) z danych wejściowych `s`. Chociaż zwykle trzeba sprawdzać wynik błędu funkcji `Sscanf`, w tym przypadku nie musimy tego robić, ponieważ jeśli wystąpiłby problem, nie zostałby dopasowany żaden przypadek instrukcji `switch`.

Kup książkę Poleć książkę

To wszystko opakowuje przedstawiona poniżej funkcja `CelsiusFlag`. Zwraca ona podmiotowi wywołującemu wskaźnik do pola `Celsius` osadzonego w zmiennej `f` typu `celsiusFlag`. Pole `Celsius` jest zmienną, która będzie aktualizowana przez metodę `Set` w trakcie przetwarzania flag. Wywołanie `Var` dodaje daną flagę do zestawu flag wiersza poleceń aplikacji, czyli zmiennej globalnej `flag.CommandLine`. Programy z niezwykle skomplikowanymi interfejsami wiersza poleceń mogą mieć kilka zmiennych tego typu. Wywołanie `Var` przypisuje argument `*celsiusFlag` do parametru `flag.Value`, powodując, że kompilator sprawdza, czy `*celsiusFlag` posiada niezbędne metody.

```
// CelsiusFlag definiuje flagę Celsius z określoną nazwą, domyślną wartością
// oraz informacją o sposobie wykorzystania i zwraca adres zmiennej flagi.
// Argument flagi musi podawać ilość i jednostkę, np. "100C".
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}
```

Teraz możemy zacząć używać tej nowej flagi w naszych programach:

code/r07/tempflag

```
var temp = tempconv.CelsiusFlag("temp", 20.0, "temperatura")

func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

Oto typowa sesja:

```
$ go build code/r07/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: nieprawidłowa temperatura "273.15K"
Usage of ./tempflag:
  -temp value
        temperatura (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
  -temp value
        temperatura (default 20°C)
```

Ćwiczenie 7.6. Dodaj do programu `tempflag` wsparcie dla temperatury w skali Kelvina.

Ćwiczenie 7.7. Wyjaśnij, dlaczego komunikat pomocy zawiera symbol `°C`, podczas gdy domyślna wartość `20.0` go nie zawiera.

7.5. Wartości interfejsów

Koncepcyjnie wartość typu interfejsowego, czyli inaczej **wartość interfejsu**, obejmuje dwa komponenty: typ konkretny i wartość tego typu. Są one nazywane **dynamicznym typem** i **dynamiczną wartością** interfejsu.

Kup książkę Poleć książkę

W przypadku języka typowanego statycznie, takiego jak Go, typy są pojęciem czasu kompilacji, więc typ nie jest wartością. W naszym modelu koncepcyjnym zestaw wartości, zwanych **deskryptorami typów**, dostarcza na temat każdego typu informacji takich jak jego nazwa i metody. W wartości interfejsu komponent typu jest reprezentowany przez odpowiedni deskryptor typu.

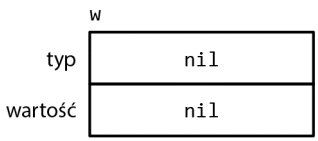
W czterech poniższych instrukcjach zmienna `w` przyjmuje trzy różne wartości. (Wartość początkowa jest taka sama jak końcowa).

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

Przyjrzyjmy się bliżej wartości i dynamicznemu zachowaniu zmiennej `w` po każdej instrukcji. Pierwsza instrukcja deklaruje `w`:

```
var w io.Writer
```

W języku Go zmienne są zawsze inicjowane do wyraźnie określonej wartości, a interfejsy nie są wyjątkiem. Wartość zerowa dla interfejsu ma ustawione na `nil` oba jego komponenty: typ i wartość (rysunek 7.1).



Rysunek 7.1. *Wartość nil interfejsu*

Wartość interfejsu jest opisywana jako `nil` lub różna od `nil` na podstawie jego dynamicznego typu, więc to jest wartość `nil` interfejsu. Możesz sprawdzić, czy wartością interfejsu jest `nil`, za pomocą `w == nil` lub `w != nil`. Wywołanie jakiejkolwiek metody wartości `nil` interfejsu wywołuje procedurę *panic*:

```
w.Write([]byte("witaj")) // panic: wyłuskanie wskaźnika nil
```

Druga instrukcja przypisuje do zmiennej `w` wartość typu `*os.File`:

```
w = os.Stdout
```

To przypisanie obejmuje pośrednią konwersję z typu konkretnego na typ interfejsowy i jest równoważne z bezpośrednią konwersją `io.Writer(os.Stdout)`. Konwersja tego rodzaju, pośrednia lub bezpośrednia, przechwytuje typ i wartość jego operandu. Typ dynamiczny interfejsu jest ustawiany na deskryptor typu dla typu wskaźnika `*os.File`, a jego wartość dynamiczna przechowuje kopię `os.Stdout`, która jest wskaźnikiem do zmiennej `os.File` reprezentującej standardowy strumień wyjściowy procesu (rysunek 7.2).

Rysunek 7.2. *Wartość interfejsu zawierająca wskaźnik *os.File*

Kup książkę Poleć książkę

Wywołanie metody `Write` na wartości interfejsu zawierającej wskaźnik `*os.File` powoduje wywołanie metody `(*os.File).Write`. To wywołanie wyświetla `"witaj"`.

```
w.Write([]byte("witaj")) // "witaj"
```

Zasadniczo podczas kompilacji nie możemy wiedzieć, jaki będzie dynamiczny typ wartości interfejsu, więc wywołanie poprzez interfejs musi używać **dynamicznego rozdzielania** (ang. *dynamic dispatch*). Zamiast bezpośredniego wywołania kompilator musi wygenerować kod, aby uzyskać adres metody o nazwie `Write` z deskryptora typu, a następnie wykonać pośrednie wywołanie tego adresu. Argumentem odbiornika dla tego wywołania jest kopia wartości dynamicznej interfejsu — `os.Stdout`. Efekt jest taki, jakbyśmy wykonali to wywołanie bezpośrednio:

```
os.Stdout.Write([]byte("witaj")) // "witaj"
```

Trzecia instrukcja przypisuje wartość typu `*bytes.Buffer` do wartości interfejsu:

```
w = new(bytes.Buffer)
```

Typem dynamicznym jest teraz `*bytes.Buffer`, a wartością dynamiczną jest wskaźnik do nowo alokowanego bufora (rysunek 7.3).

Rysunek 7.3. *Wartość interfejsu zawierająca wskaźnik *bytes.Buffer*

Wywołanie metody `Write` wykorzystuje ten sam mechanizm co poprzednio:

```
w.Write([]byte("witaj")) // zapisuje "witaj" w bytes.Buffer
```

Tym razem deskryptorem typu jest `*bytes.Buffer`, więc wywoływana jest metoda `(*bytes.Buffer).Write` z adresem bufora jako wartością parametru odbiornika. To wywołanie dołącza `"witaj"` do bufora.

Na koniec czwarta instrukcja przypisuje `nil` do wartości interfejsu:

```
w = nil
```

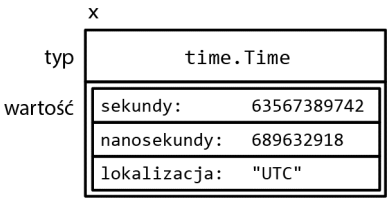
To przypisanie resetuje oba jego komponenty do wartości `nil`, przywracając zmienną `w` do tego samego stanu, w jakim była po zadeklarowaniu, czyli do stanu, który został przedstawiony na rysunku 7.1.

Wartość interfejsu może przechowywać dowolnie duże wartości dynamiczne. Przykładowo: typ `time.Time`, który reprezentuje moment w czasie, jest typem `struct` z kilkoma niewyeksportowanymi polami. Jeśli utworzymy z niego wartość interfejsu

```
var x interface{} = time.Now()
```

wynik może wyglądać tak, jak pokazano na rysunku 7.4. Koncepcyjnie wartość dynamiczna zawsze mieści się wewnątrz wartości interfejsu, bez względu na to, jak duży jest jego typ. (To tylko model koncepcyjny. Rzeczywista implementacja jest zupełnie inna)

Kup książkę Poleć książkę



Rysunek 7.4. *Wartość interfejsu przechowuje strukturę time.Time*

Wartości interfejsów mogą być porównywane za pomocą operatorów `==` i `!=`. Dwie wartości interfejsów są równe, jeśli obie są `nil` lub jeśli ich typy dynamiczne są identyczne, a ich wartości dynamiczne są równe zgodnie ze standardowym zachowaniem operatora `==` dla danego typu. Ponieważ wartości interfejsów są porównywalne, mogą być stosowane jako klucze map lub jako operandy instrukcji `switch`.

Jeśli jednak porównywane są dwie wartości interfejsów i mają one ten sam typ dynamiczny, który jednak nie jest porównywalny (np. wycinek), porównanie nie powiedzie się i wywoła panikę:

```
var x interface{} = []int{1, 2, 3}
fmt.Println(x == x) // panic: porównywanie nieporównywalnego typu []int
```

Pod tym względem typy interfejsów są ewenementem. Inne typy są bezpiecznie porównywalne (typy podstawowe i wskaźniki) albo w ogóle nieporównywalne (wycinki, mapy i funkcje), ale kiedy porównujemy wartości interfejsów lub typy złożone, które zawierają wartości interfejsów, musimy być świadomi możliwości uruchomienia procedury *panic*. Podobne ryzyko istnieje, gdy używamy interfejsów jako kluczy map lub operandów instrukcji `switch`. Wartości interfejsów należy porównywać tylko wtedy, kiedy ma się pewność, że zawierają one dynamiczne wartości porównywalnych typów.

Podczas obsługi błędów lub debugowania często pomocne jest raportowanie dynamicznego typu wartości interfejsu. W tym celu używamy czasownika `%T` pakietu `fmt`:

```
var w io.Writer
fmt.Printf("%T\n", w) // "<nil>"

w = os.Stdout
fmt.Printf("%T\n", w) // "*os.File"

w = new(bytes.Buffer)
fmt.Printf("%T\n", w) // "*bytes.Buffer"
```

Wewnętrznie pakiet `fmt` do uzyskania nazwy dynamicznego typu interfejsu wykorzystuje refleksję. Refleksji przyjrzymy się w rozdziale 12.

7.5.1. Zastrzeżenie: interfejs zawierający wskaźnik nil jest różny od nil

Wartość `nil` interfejsu, który nie zawiera w ogóle żadnej wartości, nie jest tym samym co wartość interfejsu zawierającego wskaźnik, który akurat jest `nil`. Ta subtelna różnica tworzy pułapkę, w którą wpadł chyba każdy programista Go.

Rozważmy poniższy program. Przy stałej `debug` ustawionej na wartość `true` funkcja `main` gromadzi dane wyjściowe z funkcji `f` w typie `bytes.Buffer`.

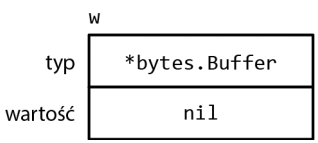
Kup książkę Poleć książkę

```
const debug = true

func main() {
    var buf *bytes.Buffer
    if debug {
        buf = new(bytes.Buffer) // włączenie gromadzenia danych wyjściowych
    }
    f(buf) // Uwaga: subtelnie nieprawidłowe!
    if debug {
        // …użycie buf…
    }
}

// Jeśli parametr out jest różny od nil, dane wyjściowe będą w nim zapisywane.
func f(out io.Writer) {
    // …coś do zrobienia…
    if out != nil {
        out.Write([]byte("zrobione!\n"))
    }
}
```

Można by się spodziewać, że zmiana `debug` na `false` wyłączy gromadzenie danych wyjściowych, ale w rzeczywistości powoduje, że program uruchamia procedurę *panic* podczas wywołania `out.Write`:

```
if out != nil {
    out.Write([]byte("zrobione!\n")) // panic: wyłuskanie wskaźnika nil
}
```

Gdy funkcja `main` wywołuje funkcję `f`, przypisuje do parametru `out` wskaźnik `nil` typu `*bytes.Buffer`, więc wartością dynamiczną parametru `out` jest `nil`. Jego typem dynamicznym jest jednak `*bytes.Buffer`, co oznacza, że `out` jest interfejsem różnym od `nil`, zawierającym wartość wskaźnika `nil` (rysunek 7.5), więc sprawdzenie defensywne `out != nil` jest nadal prawdziwe.



Rysunek 7.5. *Różny od nil interfejs zawierający wskaźnik nil*

Tak jak wcześniej dynamiczny mechanizm rozdzielania określa, że wywołana musi być metoda `(*bytes.Buffer).Write`, ale tym razem z wartością odbiornika, którą jest `nil`. Dla niektórych typów, takich jak `*os.File`, `nil` jest prawidłowym odbiornikiem (zob. punkt 6.2.1), ale `*bytes.Buffer` do nich nie należy. Ta metoda jest wywoływana, ale uruchamia procedurę *panic*, ponieważ próbuje uzyskać dostęp do bufora.

Problem polega na tym, że chociaż wskaźnik `nil` typu `*bytes.Buffer` ma metody niezbędne do spełnienia warunków tego interfejsu, to nie spełnia jego wymagań **behawioralnych**. Wywołanie to narusza w szczególności dorozumiany warunek wstępny metody `(*bytes.Buffer).Write`, który zakłada, że jej odbiornik nie będzie `nil`, więc przypisanie wskaźnika `nil` do tego interfejsu było błędem. Rozwiązaniem jest zmiana typu zmiennej `buf` w funkcji `main` na `io.Writer`, co pozwala przede wszystkim uniknąć przypisania do interfejsu dysfunkcyjnej wartości:

Kup książkę Poleć książkę

```
var buf io.Writer
if debug {
    buf = new(bytes.Buffer) // włączenie gromadzenia danych wyjściowych
}
f(buf) // OK
```

Omówiliśmy mechanizmy wartości interfejsu, więc przyjrzyjmy się teraz kilku ważniejszym interfejsom ze standardowej biblioteki języka Go. W trzech kolejnych podrozdziałach zobaczymy, w jaki sposób interfejsy są używane do sortowania, serwowania zawartości WWW i obsługi błędów.

7.6. Sortowanie za pomocą interfejsu sort.Interface

Tak jak formatowanie łańcuchów znaków sortowanie jest często używaną operacją w wielu programach. Chociaż minimalny algorytm sortowania szybkiego (ang. *quicksort*) można zmieścić w jakichś 15 liniach kodu, solidna implementacja jest znacznie dłuższa i nie jest to rodzaj kodu, który za każdym razem chcielibyśmy pisać od nowa lub kopiować, gdy jest on potrzebny.

Na szczęście pakiet `sort` zapewnia sortowanie *in situ* dowolnej sekwencji według którejkolwiek funkcji porządkującej. Jego konstrukcja jest dość niezwykła. W wielu językach algorytm sortowania jest powiązany z sekwencyjnym typem danych, podczas gdy funkcja porządkująca jest powiązana z typem elementów. Natomiast funkcja `sort.Sort` języka Go nie zakłada niczego na temat reprezentacji sekwencji lub jej elementów. Zamiast tego wykorzystuje interfejs `sort.Interface`, służący do określania kontraktu pomiędzy ogólnym algorytmem sortowania i każdym typem sekwencyjnym, który może być sortowany. Implementacja tego interfejsu określa zarówno konkretną reprezentację sekwencji (którą często jest wycinek), jak i wymagany porządek jej elementów.

Algorytm sortowania *in situ* potrzebuje trzech rzeczy (długości sekwencji, zasad porównywania dwóch elementów i sposobu zamieniania dwóch elementów), więc interfejs `sort.Interface` ma trzy metody:

```
package sort

type Interface interface {
    Len() int
    Less(i, j int) bool // i, j to indeksy elementów sekwencji
    Swap(i, j int)
}
```

Aby posortować dowolną sekwencję, musimy zdefiniować typ implementujący te trzy metody, a następnie zastosować funkcję `sort.Sort` do instancji tego typu. Rozważmy sortowanie wycinka łańcuchów znaków jako być może najprostszy przykład. Poniżej pokazano nowy typ `StringSlice` i jego metody `Len`, `Less` i `Swap`.

```
type StringSlice []string

func (p StringSlice) Len() int           { return len(p) }
func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p StringSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
```

Teraz możemy posortować wycinek łańcuchów znaków (`names`) poprzez przekonwertowanie go na typ `StringSlice` w taki sposób:

```
sort.Sort(StringSlice(names))
```

Ta konwersja daje wartość wycinka o tej samej długości, pojemności i tablicy bazowej co `names`, ale z typem, który ma trzy metody wymagane do sortowania.

Kup książkę Poleć książkę

Sortowanie wycinka łańcuchów znaków jest tak powszechne, że pakiet `sort` zapewnia typ `StringSlice` oraz funkcję o nazwie `Strings`, więc powyższe wywołanie można uprościć do postaci `sort.Strings(names)`.

Przedstawioną tu technikę można łatwo dostosować do innych porządków sortowania, np. do ignorowania wielkich liter lub znaków specjalnych. (Program Go sortujący tematy skorowidza i numery stron dla tej książki robi to przy dodatkowej logice dla cyfr rzymskich). Do skomplikowanego sortowania używamy tej samej koncepcji, ale z bardziej skomplikowanymi strukturami danych lub z bardziej skomplikowanymi implementacjami metod interfejsu `sort.Interface`.

Naszym działającym przykładem sortowania będzie lista odtwarzania utworów muzycznych wyświetlona w postaci tabeli. Każdy utwór będzie pojedynczym wierszem, a każda kolumna będzie atrybutem tego utworu, takim jak: artysta, tytuł i czas odtwarzania. Wyobraź sobie, że graficzny interfejs użytkownika prezentuje tabelę, a kliknięcie nagłówka wybranej kolumny powoduje posortowanie listy odtwarzania według tego atrybutu. Ponowne kliknięcie nagłówka tej samej kolumny odwraca kolejność. Zobaczmy, co się może zdarzyć w reakcji na każde kliknięcie.

Użyta poniżej zmienna `tracks` zawiera listę odtwarzania. (Jeden z autorów przeprasza za gust muzyczny drugiego). Każdy element jest pośredni — jest wskaźnikiem do typu `Track`. Chociaż poniższy kod działałby, gdybyśmy przechowywali elementy `Track` bezpośrednio, funkcja sortowania zamieni wiele par elementów, więc będzie działać szybciej, jeżeli każdy element będzie wskaźnikiem, czyli pojedynczym słowem maszynowym, a nie całym typem `Track`, który może mieć osiem słów lub więcej.

code/r07/sorting

```
type Track struct {
    Title  string
    Artist string
    Album  string
    Year   int
    Length time.Duration
}

var tracks = []*Track{
    {"Go", "Delilah", "From the Roots Up", 2012, length("3m38s")},
    {"Go", "Moby", "Moby", 1992, length("3m37s")},
    {"Go Ahead", "Alicia Keys", "As I Am", 2007, length("4m36s")},
    {"Ready 2 Go", "Martin Solveig", "Smash", 2011, length("4m24s")},
}

func length(s string) time.Duration {
    d, err := time.ParseDuration(s)
    if err != nil {
        panic(s)
    }
    return d
}
```

Funkcja `printTracks` wyświetla listę odtwarzania w postaci tabeli. Wyświetlenie graficzne byłoby ładniejsze, ale ta prosta procedura wykorzystuje pakiet `text/tabwriter` do wygenerowania tabeli, której kolumny są starannie wyrównane i dopełnione, tak jak pokazano nieco niżej. Zauważmy, że `*tabwriter.Writer` spełnia warunki interfejsu `io.Writer`. Gromadzi każdy zapisany w nim fragment danych. Jego metoda `Flush` formatuje całą tabelę i zapisuje ją do `os.Stdout`.

```
func printTracks(tracks []*Track) {
    const format = "%v\t%v\t%v\t%v\t%v\t\n"
```

Kup książkę Poleć książkę

```
    tw := new(tabwriter.Writer).Init(os.Stdout, 0, 8, 2, ' ', 0)
    fmt.Fprintf(tw, format, "Tytuł", "Artysta", "Album", "Rok", "Długość")
    fmt.Fprintf(tw, format, "-----", "-------", "-----", "---", "-------")
    for _, t := range tracks {
        fmt.Fprintf(tw, format, t.Title, t.Artist, t.Album, t.Year, t.Length)
    }
    tw.Flush() // oblicza szerokości kolumn i wyświetla tabelę
}
```

Aby posortować listę utworów według pola `Artist`, definiujemy nowy typ wycinka z niezbędnymi metodami `Len`, `Less` i `Swap`, analogicznie do tego, co zrobiliśmy dla `StringSlice`.

```
type byArtist []*Track

func (x byArtist) Len() int           { return len(x) }
func (x byArtist) Less(i, j int) bool { return x[i].Artist < x[j].Artist }
func (x byArtist) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
```

Aby wywołać ogólną procedurę sortowania, musimy najpierw przekonwertować `tracks` na nowy typ `byArtist`, który definiuje porządek:

```
sort.Sort(byArtist(tracks))
```

Po posortowaniu wycinka według artysty dane wyjściowe z funkcji `printTracks` są następujące:

```
Tytuł       Artysta         Album              Rok   Długość
-----       -------         -----              ---   -------
Go Ahead    Alicia Keys     As I Am            2007  4m36s
Go          Delilah         From the Roots Up  2012  3m38s
Ready 2 Go  Martin Solveig  Smash              2011  4m24s
Go          Moby            Moby               1992  3m37s
```

Jeśli użytkownik po raz drugi zażąda „sortuj według artysty”, utwory zostaną posortowane w odwrotnej kolejności. Nie musimy jednak definiować nowego typu `byReverseArtist` z odwróconą metodą `Less`, ponieważ pakiet `sort` zapewnia funkcję `Reverse`, która przekształca dowolny porządek sortowania na jego odwrotność:

```
sort.Sort(sort.Reverse(byArtist(tracks)))
```

Po odwrotnym posortowaniu wycinka według artysty dane wyjściowe z funkcji `printTracks` są następujące:

```
Tytuł       Artysta         Album              Rok   Długość
-----       -------         -----              ---   -------
Go          Moby            Moby               1992  3m37s
Ready 2 Go  Martin Solveig  Smash              2011  4m24s
Go          Delilah         From the Roots Up  2012  3m38s
Go Ahead    Alicia Keys     As I Am            2007  4m36s
```

Funkcja `sort.Reverse` zasługuje na uwagę, ponieważ wykorzystuje kompozycję (zob. podrozdział 6.3), która jest ważną koncepcją. Pakiet `sort` definiuje niewyeksportowany typ `reverse`, który jest strukturą osadzającą `sort.Interface`. Metoda `Less` dla typu `reverse` wywołuje metodę `Less` osadzonej wartości `sort.Interface`, ale z odwróconymi indeksami, co odwraca kolejność wyników sortowania.

```
package sort
type reverse struct{ Interface } // czyli sort.Interface
func (r reverse) Less(i, j int) bool { return r.Interface.Less(j, i) }
func Reverse(data Interface) Interface { return reverse{data} }
```

Kup książkę Poleć książkę

Pozostałe dwie metody typu `reverse`, czyli `Len` i `Swap`, są pośrednio dostarczane przez oryginalną wartość `sort.Interface`, ponieważ jest to osadzone pole. Wyeksportowana funkcja `Reverse` zwraca instancję typu `reverse`, która zawiera oryginalną wartość `sort.Interface`.

Aby posortować według innej kolumny, musimy zdefiniować nowy typ, np. `byYear` (według roku):

```
type byYear []*Track

func (x byYear) Len() int           { return len(x) }
func (x byYear) Less(i, j int) bool { return x[i].Year < x[j].Year }
func (x byYear) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
```

Po posortowaniu `trakcs` według roku za pomocą `sort.Sort(byYear(tracks))` funkcja `printTracks` wyświetla chronologiczny wykaz:

```
Tytuł       Artysta         Album              Rok  Długość
-----       -------         -----              ---  -------
Go          Moby            Moby               1992  3m37s
Go Ahead    Alicia Keys     As I Am            2007  4m36s
Ready 2 Go  Martin Solveig  Smash              2011  4m24s
Go          Delilah         From the Roots Up  2012  3m38s
```

Dla każdego typu elementu wycinka i każdej funkcji porządkowania, której potrzebujemy, deklarujemy nową implementację `sort.Interface`. Jak widać, metody `Len` i `Swap` mają identyczne definicje dla wszystkich typów wycinka. W następnym przykładzie typ konkretny `customSort` łączy wycinek z funkcją, pozwalając nam zdefiniować nowy porządek poprzez napisanie jedynie funkcji porównania. Nawiasem mówiąc, konkretne typy implementujące `sort.Interface` nie zawsze są wycinkami. Typ `customSort` jest typem `struct`.

```
type customSort struct {
    t [ ]*Track
    less func(x, y *Track) bool
}

func (x customSort) Len() int           { return len(x.t) }
func (x customSort) Less(i, j int) bool { return x.less(x.t[i], x.t[j]) }
func (x customSort) Swap(i, j int)      { x.t[i], x.t[j] = x.t[j], x.t[i] }
```

Zdefiniujmy wielopoziomową funkcję porządkowania, której głównym kluczem sortowania jest `Title` (tytuł), kluczem wtórnym jest `Year` (rok), a kluczem trzeciorzędowym jest `Length` (długość utworu). Oto wywołanie funkcji `Sort` wykorzystujące anonimową funkcję porządkowania:

```
sort.Sort(customSort{tracks, func(x, y *Track) bool {
    if x.Title != y.Title {
        return x.Title < y.Title
    }
    if x.Year != y.Year {
        return x.Year < y.Year
    }
    if x.Length != y.Length {
        return x.Length < y.Length
    }
    return false
}})
```

A oto wynik. Należy zwrócić uwagę, że powiązanie między dwoma utworami zatytułowanymi *Go* zostało rozerwane na korzyść utworu starszego.

Kup książkę Poleć książkę

```
Tytuł       Artysta         Album              Rok   Długość
-----       -------         -----              ---   -------
Go          Moby            Moby               1992  3m37s
Go          Delilah         From the Roots Up  2012  3m38s
Go Ahead    Alicia Keys     As I Am            2007  4m36s
Ready 2 Go  Martin Solveig  Smash              2011  4m24s
```

Chociaż sortowanie sekwencji o długości *n* wymaga O(*n* log *n*) operacji porównania, sprawdzenie, czy sekwencja została już posortowana, wymaga co najwyżej *n*–1 porównań. Funkcja `IsSorted` z pakietu `sort` sprawdza to za nas. Podobnie jak `sort.Sort`, abstrahuje zarówno sekwencję, jak i jej funkcję porządkującą za pomocą `sort.Interface`, ale nigdy nie wywołuje metody `Swap`. Poniższy kod demonstruje funkcje `IntsAreSorted` i `Ints` oraz typ `IntSlice`:

```
values := []int{3, 1, 4, 1}
fmt.Println(sort.IntsAreSorted(values)) // "false"
sort.Ints(values)
fmt.Println(values)                     // "[1 1 3 4]"
fmt.Println(sort.IntsAreSorted(values)) // "true"
sort.Sort(sort.Reverse(sort.IntSlice(values)))
fmt.Println(values)                     // "[4 3 1 1]"
fmt.Println(sort.IntsAreSorted(values)) // "false"
```

Dla wygody pakiet `sort` zapewnia wersje swoich funkcji i typów wyspecjalizowane dla `[]int`, `[]string` i `[]float64` z wykorzystaniem ich naturalnych porządkowań. W przypadku innych typów, takich jak `[]int64` lub `[]uint`, jesteśmy zdani na siebie, choć droga jest krótka.

Ćwiczenie 7.8. Wiele graficznych interfejsów użytkownika zapewnia widżet tabeli z wielopoziomowym sortowaniem stanowym: podstawowym kluczem sortowania jest ostatnio kliknięty nagłówek kolumny, wtórnym kluczem sortowania jest kliknięty przedostatnio nagłówek kolumny itd. Zdefiniuj implementację `sort.Interface` do wykorzystania przez taką tabelę. Porównaj to podejście z wielokrotnym sortowaniem przy użyciu `sort.Stable`.

Ćwiczenie 7.9. Użyj pakietu `html/template` (zob. podrozdział 4.6), aby zastąpić `printTracks` funkcją, która wyświetla utwory w postaci tabeli HTML. Użyj rozwiązania poprzedniego ćwiczenia, aby zaaranżować, że każde kliknięcie nagłówka kolumny będzie wysyłać żądanie HTTP w celu posortowania tabeli.

Ćwiczenie 7.10. Typ `sort.Interface` może być zaadaptowany do innych zastosowań. Napisz funkcję `IsPalindrome(s sort.Interface) bool`, która informuje, że sekwencja `s` jest palindromem, czyli że odwrócenie sekwencji nie zmienia jej. Zakładamy, że elementy w indeksach `i` oraz `j` są równe, jeśli `!s.Less(i, j) && !s.Less(j, i)`.

7.7. Interfejs http.Handler

W rozdziale 1. zobaczyliśmy przelotnie, jak skorzystać z pakietu `net/http` w celu zaimplementowania klientów (podrozdział 1.5) i serwerów WWW (podrozdział 1.7). W tym podrozdziale przyjrzymy się bliżej interfejsowi API serwera, którego podstawą jest interfejs `http.Handler`:

net/http

```
package http

type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}

func ListenAndServe(address string, h Handler) error
```

Kup książkę Poleć książkę

Funkcja `ListenAndServe` wymaga adresu serwera, np. `"localhost:8000"`, oraz instancji interfejsu `Handler`, do którego kierowane będą wszystkie żądania. Ta funkcja działa w nieskończoność lub do momentu awarii serwera (lub niepowodzenia jego uruchomienia) z błędem, zawsze innym niż `nil`, który jest przez tę funkcję zwracany.

Wyobraźmy sobie stronę *e-commerce* z bazą danych mapującą rzeczy na sprzedaż na ich ceny w złotych. Poniższy program pokazuje najprostszą możliwą implementację. Modeluje ona magazyn jako typ mapy (`database`), do którego doczepiamy metodę `ServeHTTP`, aby ten typ spełniał wymagania interfejsu `http.Handler`. Procedura obsługi iteruje przez mapę za pomocą pętli `range` i wyświetla jej elementy.

code/r07/http1

```
func main() {
    db := database{"buty": 50, "skarpety": 5}
    log.Fatal(http.ListenAndServe("localhost:8000", db))
}

type dollars float32

func (d dollars) String() string { return fmt.Sprintf("%.2f PLN", d) }

type database map[string]dollars

func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
```

Uruchommy serwer:

```
$ go build code/r07/http1
$ ./http1 &
```

Jeśli połączymy się z nim za pomocą programu `fetch` z podrozdziału 1.5 (lub za pomocą przeglądarki internetowej, jeśli wolisz), otrzymamy następujące dane wyjściowe:

```
$ go build code/r01/fetch
$ ./fetch http://localhost:8000
buty: 50.00 PLN
skarpety: 5.00 PLN
```

Na razie serwer może wyświetlać tylko cały inwentarz i zrobi to dla każdego żądania, niezależnie od adresu URL. W bardziej realistycznym przypadku dla serwera definiuje się wiele różnych adresów URL, z których każdy wyzwala inne zachowanie. Nazwijmy istniejące żądanie `/list` i dodajmy jeszcze jedno, o nazwie `/price`, które raportuje cenę pojedynczego elementu, określonego jako parametr żądania, np. `/price?item=skarpety`.

code/r07/http2

```
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    switch req.URL.Path {
    case "/list":
        for item, price := range db {
            fmt.Fprintf(w, "%s: %s\n", item, price)
        }
    case "/price":
        item := req.URL.Query().Get("item")
        price, ok := db[item]
```

Kup książkę Poleć książkę

```
            if !ok {
                w.WriteHeader(http.StatusNotFound) // błąd 404
                fmt.Fprintf(w, "nie ma takiej pozycji: %q\n", item)
                return
            }
            fmt.Fprintf(w, "%s\n", price)
        default:
            w.WriteHeader(http.StatusNotFound) // błąd 404
            fmt.Fprintf(w, "nie ma takiej strony: %s\n", req.URL)
        }
    }
```

Teraz procedura obsługi na podstawie komponentu ścieżki URL, `req.URL.Path`, decyduje, jaką logikę wykonać. Jeżeli procedura obsługi nie rozpozna ścieżki, zgłasza klientowi błąd HTTP poprzez wywołanie `w.WriteHeader(http.StatusNotFound)`. Należy to zrobić przed zapisaniem jakiegokolwiek tekstu w parametrze `w`. (Nawiasem mówiąc, `http.ResponseWriter` jest kolejnym interfejsem. Poszerza on interfejs `io.Writer` o metody do wysyłania nagłówków odpowiedzi HTTP). Równoważnie możemy użyć funkcji narzędziowej `http.Error`:

```
msg := fmt.Sprintf("nie ma takiej strony: %s\n", req.URL)
http.Error(w, msg, http.StatusNotFound) // błąd 404
```

Instrukcja `case` dla żądania `/price` wywołuje metodę `Query` adresu URL w celu parsowania parametrów żądania HTTP jako mapy, a dokładniej multimapy typu `url.Values` (zob. punkt 6.2.l) z pakietu `net/url`. Następnie znajduje pierwszy parametr `item` i szuka jego ceny. Jeśli element nie zostanie znaleziony, zgłasza błąd.

Oto przykładowa sesja z nowym serwerem:

```
$ go build code/r07/http2
$ go build code/r01/fetch
$ ./http2 &
$ ./fetch http://localhost:8000/list
buty: 50.00 PLN
skarpety: 5.00 PLN
$ ./fetch http://localhost:8000/price?item=skarpety
5.00 PLN
$ ./fetch http://localhost:8000/price?item=buty
50.00 PLN
$ ./fetch http://localhost:8000/price?item=kapelusz
nie ma takiej pozycji: "kapelusz"
$ ./fetch http://localhost:8000/help
nie ma takiej strony: /help
```

Oczywiście do metody `ServeHTTP` moglibyśmy dodawać kolejne przypadki (`case`), ale w realistycznej aplikacji wygodniej jest zdefiniować logikę dla każdego przypadku w osobnej funkcji lub metodzie. Ponadto powiązane adresy URL mogą wymagać podobnej logiki. Kilka plików obrazów może mieć np. adresy URL w postaci `/images/*.png`. Z tych powodów pakiet `net/http` zapewnia `ServeMux`, czyli **multiplekser żądań**, aby uprościć powiązania między adresami URL i procedurami obsługi. `ServeMux` łączy kolekcję procedur obsługi `http.Handler` w pojedynczy `http.Handler`. Ponownie widzimy, że różne typy spełniające warunki tego samego interfejsu są **podstawialne**: serwer WWW może rozsyłać żądania do dowolnej procedury obsługi `http.Handler` niezależnie od tego, jaki typ konkretny się za nią kryje.

Dla bardziej złożonych aplikacji można skomponować kilka multiplekserów `ServeMux` do obsługi bardziej zawiłych wymagań rozsyłania żądań. Język Go nie posiada kanonicznego frameworku WWW, analogicznego do Rails dla Ruby lub Django dla Pythona. To nie znaczy, że takie frameworki nie

Kup książkę Poleć książkę

istnieją, ale elementy konstrukcyjne w standardowej bibliotece języka Go są tak elastyczne, że frameworki często bywają niepotrzebne. Ponadto, chociaż frameworki są wygodne we wczesnych fazach projektu, ich dodatkowa złożoność może utrudnić długoterminowe utrzymywanie oprogramowania.

W poniższym programie utworzymy `ServeMux` i użyjemy go do skojarzenia adresów URL z odpowiednimi procedurami obsługi dla operacji `/list` i `/price`, które zostały rozdzielone na osobne metody. Następnie użyjemy multipleksera `ServeMux` jako głównej procedury obsługi w wywołaniu funkcji `ListenAndServe`.

code/r07/http3

```
func main() {
    db := database{"buty": 50, "skarpety": 5}
    mux := http.NewServeMux()
    mux.Handle("/list", http.HandlerFunc(db.list))
    mux.Handle("/price", http.HandlerFunc(db.price))
    log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound) // błąd 404
        fmt.Fprintf(w, "nie ma takiej pozycji: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
}
```

Skupmy się na dwóch wywołaniach `mux.Handle`, które rejestrują procedury obsługi. W pierwszym `db.list` jest wartością metody (zob. podrozdział 6.4), czyli wartością typu

```
func(w http.ResponseWriter, req *http.Request)
```

który po wywołaniu wywołuje metodę `database.list` z wartością odbiornika `db`. Zatem `db.list` jest funkcją, która implementuje zachowanie procedury obsługi, ale ponieważ nie ma żadnych metod, nie spełnia warunków interfejsu `http.Handler` i nie może być przekazywana bezpośrednio do `mux.Handle`.

Wyrażenie `http.HandlerFunc(db.list)` jest konwersją, a nie wywołaniem funkcji, ponieważ `http.HandlerFunc` jest typem. Posiada następującą definicję:

net/http

```
package http

type HandlerFunc func(w ResponseWriter, r *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

Kup książkę Poleć książkę

`HandlerFunc` zawiera kilka niezwykłych cech mechanizmu interfejsu języka Go. Jest to typ funkcji, który ma metody i spełnia warunki interfejsu `http.Handler`. Zachowanie jego metody `ServeHTTP` polega na wywołaniu funkcji bazowej. Dlatego `HandlerFunc` jest adapterem, który pozwala wartości funkcji spełnić warunki interfejsu, gdzie funkcja i jedyna metoda interfejsu mają tę samą sygnaturę. W efekcie ta sztuczka pozwala pojedynczemu typowi, takiemu jak `database`, spełnić warunki interfejsu `http.Handler` na kilka różnych sposobów: raz poprzez jego metodę `list`, raz poprzez jego metodę `price` itd.

Ponieważ rejestrowanie procedury obsługi w ten sposób jest tak powszechne, `ServeMux` ma złożoną metodę o nazwie `HandleFunc`, która robi to za nas, więc możemy uprościć kod rejestracji procedury obsługi do tej postaci:

code/r07/http3a

```
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

Na podstawie powyższego kodu łatwo zrozumieć, w jaki sposób można by skonstruować program, w którym istnieją dwa różne serwery WWW, nasłuchujące na różnych portach, definiujące różne adresy URL i rozsyłające żądania do różnych procedur obsługi. Należałoby po prostu skonstruować kolejny `ServeMux` i wykonać, być może równolegle, kolejne wywołanie funkcji `ListenAndServe`. Jednak w większości programów jeden serwer WWW jest wystarczający. Ponadto typowe jest definiowanie procedur obsługi HTTP w wielu plikach aplikacji i uciążliwe byłoby, gdyby wszystkie one musiały zostać bezpośrednio zarejestrowane w instancji `ServeMux` tej aplikacji.

Tak więc dla wygody pakiet `net/http` zapewnia globalną instancję `ServeMux` o nazwie `Default`
`↪ServeMux` oraz funkcje poziomu pakietu o nazwach `http.Handle` i `http.HandleFunc`. Aby użyć `DefaultServeMux` jako głównej procedury obsługi serwera, nie musimy przekazywać jej do funkcji `ListenAndServe` — zrobi to `nil`.

Funkcję `main` serwera można wtedy uprościć do tej postaci:

code/r07/http4

```
func main() {
    db := database{"buty": 50, "skarpety": 5}
    http.HandleFunc("/list", db.list)
    http.HandleFunc("/price", db.price)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

Na koniec ważne przypomnienie. Jak wspomniano w podrozdziale 1.7, serwer WWW wywołuje każdą procedurę obsługi w nowej funkcji *goroutine*. Dlatego procedury obsługi muszą podejmować środki ostrożności, takie jak blokowanie podczas uzyskiwania dostępu do zmiennych, do których dostęp mogą uzyskiwać inne funkcje *goroutine*, w tym inne żądania do tej samej procedury obsługi. O współbieżności porozmawiamy w następnych dwóch rozdziałach.

Ćwiczenie 7.11. Dodaj kolejne procedury obsługi, aby klienty mogły tworzyć, czytać, aktualizować i usuwać wpisy bazy danych. Przykładowo: żądanie w postaci `/update?item=skarpety&price=6` zaktualizuje cenę przedmiotu w inwentarzu i zgłosi błąd, jeśli element nie istnieje lub jeśli cena jest nieprawidłowa. (Uwaga: ta zmiana wprowadza równoległe aktualizacje zmiennych).

Ćwiczenie 7.12. Zmień procedurę obsługi dla operacji `/list`, aby wyświetlała swoje dane wyjściowe w postaci tabeli HTML, a nie tekstu. Przydatny może być pakiet `html/template` (zob. podrozdział 4.6).

Kup książkę Poleć książkę

7.8. Interfejs error

Od początku tej książki używaliśmy wartości tajemniczego predeklarowanego typu `error`, nie wyjaśniając, czym on właściwie jest. W rzeczywistości to po prostu typ interfejsowy z pojedynczą metodą, która zwraca komunikat o błędzie:

```
type error interface {
    Error() string
}
```

Najprostszym sposobem utworzenia instancji `error` jest wywołanie funkcji `errors.New`, która zwraca nowy `error` dla danego komunikatu o błędzie. Cały pakiet `errors` ma tylko cztery linie:

```
package errors

func New(text string) error { return &errorString{text} }

type errorString struct { text string }

func (e *errorString) Error() string { return e.text }
```

Typem bazowym `errorString` jest struktura, a nie łańcuch znaków, aby chronić jego reprezentację od przypadkowych (lub zamierzonych) aktualizacji. A powodem, dla którego to typ wskaźnika `*errorString`, a nie sam `errorString` spełnia warunki interfejsu `error`, jest to, że każde wywołanie `New` alokuje odrębną instancję `error`, która nie jest równa żadnej innej. Nie chcielibyśmy, aby taki wyróżniający się błąd jak `io.EOF` był równy z innym, który akurat ma ten sam komunikat.

```
fmt.Println(errors.New("EOF") == errors.New("EOF")) // "false"
```

Wywołania `errors.New` są stosunkowo rzadkie, ponieważ istnieje wygodna funkcja opakowująca, `fmt.Errorf`, która wykonuje również formatowanie łańcucha znaków. Użyliśmy jej kilka razy w rozdziale 5.

```
package fmt

import "errors"

func Errorf(format string, args ...interface{}) error {
    return errors.New(Sprintf(format, args...))
}
```

Chociaż `*errorString` może być najprostszym typem interfejsu `error`, na pewno nie jest jedynym. Pakiet `syscall` zapewnia np. interfejs API niskopoziomowych wywołań systemowych języka Go. Na wielu platformach definiuje on typ liczbowy `Errno` spełniający warunki interfejsu `error`, a na platformach uniksowych metoda `Error` typu `Errno` przeprowadza wyszukiwanie w tabeli łańcuchów znaków, tak jak pokazano poniżej:

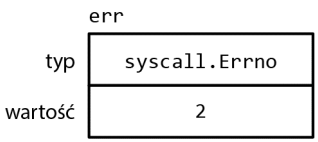
```
package syscall

type Errno uintptr // kody błędów systemu operacyjnego

var errors = [...]string{
    1:  "niedozwolona operacja",         // EPERM
    2:  "nie ma takiego pliku lub katalogu", // ENOENT
    3:  "nie ma takiego procesu",            // ESRCH
    //...
}

func (e Errno) Error() string {
    if 0 <= int(e) && int(e) < len(errors) {
```

```
        return errors[e]
    }
    return fmt.Sprintf("errno %d", e)
}
```

Poniższa instrukcja tworzy wartość interfejsu przechowującą wartość 2 typu `Errno`, oznaczającą stan `ENOENT` systemu POSIX:

```
var err error = syscall.Errno(2)
fmt.Println(err.Error()) // "nie ma takiego pliku lub katalogu"
fmt.Println(err)         // "nie ma takiego pliku lub katalogu"
```

Wartość `err` została przedstawiona graficznie na rysunku 7.6.



Rysunek 7.6. *Wartość interfejsu przechowująca wartość całkowitą typu syscall.Errno*

`Errno` jest efektywną reprezentacją błędów wywołań systemowych zaczerpniętą ze skończonego zbioru i spełnia warunki standardowego interfejsu `error`. Inne typy spełniające warunki tego interfejsu zobaczymy w podrozdziale 7.11.

7.9. Przykład: ewaluator wyrażeń

W tym podrozdziale zbudujemy ewaluator dla prostych wyrażeń arytmetycznych. Użyjemy interfejsu `Expr` do reprezentowania dowolnego wyrażenia w tym języku. Na razie ten interfejs nie potrzebuje żadnych metod, ale dodamy kilka później.

```
// Expr jest wyrażeniem arytmetycznym.
type Expr interface{}
```

Nasz język wyrażeń składa się z literałów zmiennoprzecinkowych, operatorów binarnych `+`, `-`, `*` oraz `/`, operatorów jednoargumentowych `-x` i `+x`, wywołań funkcji `pow(x,y)`, `sin(x)` i `sqrt(x)`, zmiennych takich jak `x` i `pi` oraz oczywiście nawiasów i standardowego pierwszeństwa operatorów. Wszystkie wartości są typami `float64`. Oto kilka przykładowych wyrażeń:

```
sqrt(A / pi)
pow(x, 3) + pow(y, 3)
(F - 32) * 5 / 9
```

Pięć poniższych typów konkretnych reprezentuje poszczególne rodzaje wyrażeń. Typ `Var` reprezentuje referencję do zmiennej. (Wkrótce zobaczymy, dlaczego jest wyeksportowany). Typ `literal` reprezentuje stałą zmiennoprzecinkową. Typy `unary` i `binary` reprezentują wyrażenia operatorów z jednym operandem lub dwoma, które mogą być dowolnym rodzajem `Expr`. Typ `call` reprezentuje wywołanie funkcji. Ograniczymy jej pole `fn` do `pow`, `sin` lub `sqrt`.

code/r07/eval

```
// Typ Var identyfikuje zmienną, np. x.
type Var string

// Typ literal jest stałą liczbową, np. 3.141.
type literal float64
```

Kup książkę Poleć książkę

```
// Typ unary reprezentuje wyrażenia operatora jednoargumentowego, np. –x.
type unary struct {
    op rune // możliwe wartości: '+', '–'
    x Expr
}

// Typ binary reprezentuje wyrażenie operatora binarnego, np. x+y.
type binary struct {
    op  rune // możliwe wartości: '+', '–', '*', '/'
    x, y Expr
}

// Typ call reprezentuje wyrażenie wywołania funkcji, np. sin(x).
type call struct {
    fn  string // możliwe wartości: "pow", "sin", "sqrt"
    args []Expr
}
```

Aby dokonać ewaluacji wyrażenia zawierającego zmienne, potrzebujemy **środowiska** (ang. *environment*), które mapuje nazwy zmiennych na wartości:

```
type Env map[Var]float64
```

Będziemy również potrzebować każdego rodzaju wyrażenia do zdefiniowania metody `Eval`, która zwraca wartość wyrażenia w danym środowisku. Ponieważ każde wyrażenie musi zapewniać tę metodę, dodamy ją do interfejsu `Expr`. Ten pakiet eksportuje tylko typy `Expr`, `Env` i `Var`. Klienty mogą używać ewaluatora bez dostępu do pozostałych typów wyrażeń.

```
type Expr interface {
    // Eval zwraca wartość tego wyrażenia Expr w środowisku env.
    Eval(env Env) float64
}
```

Konkretne metody `Eval` przedstawiono poniżej. Metoda dla typu `Var` wykonuje przeszukiwanie środowiska, co zwraca zero, jeśli zmienna nie jest zdefiniowana. Natomiast metoda dla typu `literal` po prostu zwraca wartość literału.

```
func (v Var) Eval(env Env) float64 {
    return env[v]
}

func (l literal) Eval(_ Env) float64 {
    return float64(l)
}
```

Metody `Eval` dla typów `unary` i `binary` rekurencyjnie ewaluują swoje operandy, a następnie stosują do nich operację `op`. Nie traktujemy dzielenia przez zero lub nieskończoność jako błędu, ponieważ te działania generują wynik, aczkolwiek nieokreślony. Wreszcie metoda dla typu `call` ewaluuje argumenty dla funkcji `pow`, `sin` lub `sqrt`, a następnie wywołuje odpowiednią funkcję z pakietu `math`.

```
func (u unary) Eval(env Env) float64 {
    switch u.op {
    case '+':
        return +u.x.Eval(env)
    case '-':
        return -u.x.Eval(env)
    }
    panic(fmt.Sprintf("nieobsługiwany operator jednoargumentowy: %q", u.op))
}
```

Kup książkę Poleć książkę

```
func (b binary) Eval(env Env) float64 {
    switch b.op {
    case '+':
        return b.x.Eval(env) + b.y.Eval(env)
    case '-':
        return b.x.Eval(env) - b.y.Eval(env)
    case '*':
        return b.x.Eval(env) * b.y.Eval(env)
    case '/':
        return b.x.Eval(env) / b.y.Eval(env)
    }
    panic(fmt.Sprintf("nieobsługiwany operator binarny: %q", b.op))
}

func (c call) Eval(env Env) float64 {
    switch c.fn {
    case "pow":
        return math.Pow(c.args[0].Eval(env), c.args[1].Eval(env))
    case "sin":
        return math.Sin(c.args[0].Eval(env))
    case "sqrt":
        return math.Sqrt(c.args[0].Eval(env))
    }
    panic(fmt.Sprintf("nieobsługiwane wywołanie funkcji: %s", c.fn))
}
```

Wykonanie niektórych z tych metod może się nie powieść. Wyrażenie `call` może mieć np. nieznaną funkcję lub niewłaściwą liczbę argumentów. Możliwe jest również skonstruowanie wyrażenia `unary` lub `binary` z nieprawidłowym operatorem, takim jak `!` lub `<` (chociaż wspomniana poniżej funkcja `Parse` nigdy tego nie zrobi). Te błędy powodują, że `Eval` uruchamia procedurę *panic*. Inne błędy, takie jak ewaluacja zmiennej `Var` nieobecnej w środowisku, powodują jedynie, że `Eval` zwraca niewłaściwy wynik. Wszystkie te błędy mogą być wykrywane poprzez sprawdzenie wyrażenia `Expr` przed jego ewaluacją. To będzie zadaniem metody `Check`, którą pokażemy wkrótce, ale najpierw przetestujmy metodę `Eval`.

Przedstawiona poniżej funkcja `TestEval` jest testem ewaluatora. Wykorzystuje pakiet `testing`, który omówimy w rozdziale 11., ale na razie wystarczy wiedzieć, że wywołanie funkcji `t.Errorf` zgłasza błąd. Funkcja wykonuje pętlę przez tablicę danych wejściowych, która definiuje trzy wyrażenia i inne środowisko dla każdego z nich. Pierwsze wyrażenie oblicza promień koła, mając dane jego pole powierzchni `A`, drugie oblicza sumę sześcianów dwóch zmiennych `x` i `y`, a trzecie przekształca temperaturę Fahrenheita `F` na stopnie Celsjusza.

```
func TestEval(t *testing.T) {
    tests := []struct {
        expr string
        env  Env
        want string
    }{
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 12, "y": 1}, "1729"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
        {"5 / 9 * (F - 32)", Env{"F": 32}, "0"},
        {"5 / 9 * (F - 32)", Env{"F": 212}, "100"},
    }
    var prevExpr string
    for _, test := range tests {
```

Kup książkę Poleć książkę

```
            // Wyświetla expr tylko wtedy, gdy się zmienia.
            if test.expr != prevExpr {
                fmt.Printf("\n%s\n", test.expr)
                prevExpr = test.expr
            }
            expr, err := Parse(test.expr)
            if err != nil {
                t.Error(err) // parsuje błąd
                continue
            }
            got := fmt.Sprintf("%.6g", expr.Eval(test.env))
            fmt.Printf("\t%v => %s\n", test.env, got)
            if got != test.want {
                t.Errorf("%s.Eval() in %s = %q, want %q\n",
                test.expr, test.env, got, test.want)
            }
        }
    }
```

Dla każdego wpisu w tablicy ten test parsuje wyrażenie, ewaluuje je w danym środowisku i wyświetla wynik. Nie mamy miejsca, aby pokazać tu funkcję `Parse`, ale znajdziesz ją, jeśli pobierzesz ten pakiet za pomocą polecenia `go get`.

Polecenie `go test` (zob. podrozdział 11.1) uruchamia testy pakietu:

```
$ go test -v code/r07/eval
```

Flaga `-v` pozwala nam zobaczyć wyświetlone dane wyjściowe z testu, które są zazwyczaj ukryte dla udanego testu, takiego jak ten. Oto dane wyjściowe z instrukcji `fmt.Printf` tego testu:

```
sqrt(A / pi)
    map[A:87616 pi:3.141592653589793] => 167

pow(x, 3) + pow(y, 3)
    map[x:12 y:1] => 1729
    map[x:9 y:10] => 1729

5 / 9 * (F - 32)
    map[F:-40] => -40
    map[F:32] => 0
    map[F:212] => 100
```

Na szczęście do tej pory wszystkie dane wejściowe były poprawne składniowo, ale nasze szczęście może nie potrwać długo. Nawet w językach interpretowanych powszechne jest sprawdzanie składni pod kątem błędów **statycznych**, czyli takich, które można wykryć bez uruchamiania programu. Dzięki oddzieleniu kontroli statycznych od dynamicznych możemy wykrywać błędy wcześniej i wykonywać wiele kontroli tylko raz, a nie przy każdej ewaluacji wyrażenia.

Dodajmy do interfejsu `Expr` kolejną metodę. Metoda `Check` sprawdza błędy statyczne w drzewie składniowym wyrażenia. Jej parametr `vars` omówimy za chwilę.

```
type Expr interface {
    Eval(env Env) float64
    // Check zgłasza błędy w tym wyrażeniu Expr i dodaje do zbioru swoje wartości Var.
    Check(vars map[Var]bool) error
}
```

Konkretne metody `Check` przedstawiono poniżej. Ewaluacja typów `literal` i `Var` nie może się nie powieść, więc metody `Check` dla tych typów zwracają `nil`. Metody dla typów `unary` i `binary` najpierw sprawdzają, czy operator jest prawidłowy, a następnie rekurencyjnie sprawdzają operandy.

Kup książkę Poleć książkę

Podobnie metoda dla typu `call` — najpierw sprawdza, czy funkcja jest znana i ma właściwą liczbę argumentów, a następnie rekurencyjnie sprawdza każdy argument.

```
func (v Var) Check(vars map[Var]bool) error {
    vars[v] = true
    return nil
}

func (literal) Check(vars map[Var]bool) error {
    return nil
}

func (u unary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-", u.op) {
        return fmt.Errorf("nieoczekiwany operator jednoargumentowy %q", u.op)
    }
    return u.x.Check(vars)
}

func (b binary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-*/", b.op) {
        return fmt.Errorf("nieoczekiwany operator binarny %q", b.op)
    }
    if err := b.x.Check(vars); err != nil {
        return err
    }
    return b.y.Check(vars)
}

func (c call) Check(vars map[Var]bool) error {
    arity, ok := numParams[c.fn]
    if !ok {
        return fmt.Errorf("nieznana funkcja %q", c.fn)
    }
    if len(c.args) != arity {
        return fmt.Errorf("wywołanie %s ma argumentów %d, wymaga %d",
            c.fn, len(c.args), arity)
    }
    for _, arg := range c.args {
        if err := arg.Check(vars); err != nil {
            return err
        }
    }
    return nil
}

var numParams = map[string]int{"pow": 2, "sin": 1, "sqrt": 1}
```

Poniżej w dwóch grupach została przedstawiona lista wadliwych danych wejściowych i wywoływanych przez nie błędów. Funkcja `Parse` (niepokazana) zgłasza błędy składniowe, a funkcja `Check` zgłasza błędy semantyczne.

```
x % 2             nieoczekiwane '%'
math.Pi           nieoczekiwane '.'
!true             nieoczekiwane '!'
"hello"           nieoczekiwane '"'

log(10)           nieznana funkcja "log"
sqrt(1, 2)        wywołanie sqrt ma argumentów 2, wymaga 1
```

Kup książkę Poleć książkę

Argument metody `Check`, czyli zbiór wartości `Var`, gromadzi zbiór nazw zmiennych znalezionych w wyrażeniu. Aby ewaluacja się powiodła, każda z tych zmiennych musi być obecna w danym środowisku. Ten zbiór jest logicznie **wynikiem** wywołania `Check`, ponieważ jednak ta metoda jest rekurencyjna, wygodniej jest dla niej zapełniać zbiór przekazywany jako parametr. W początkowym wywołaniu klient musi dostarczyć pusty zbiór.

W podrozdziale 3.2 rysowaliśmy wykres funkcji `f(x,y)`, który był definiowany w czasie kompilacji. Ponieważ możemy teraz parsować, sprawdzać i ewaluować wyrażenia w łańcuchach znaków, możemy zbudować aplikację internetową, która w trakcie działania otrzymuje od klienta wyrażenie i rysuje wykres powierzchniowy danej funkcji. Możemy użyć zbioru `vars`, żeby sprawdzić, czy wyrażenie jest funkcją tylko dwóch zmiennych: `x` i `y` — w rzeczywistości trzech, ponieważ dla wygody zapewnimy promień `r`. Użyjemy też metody `Check` do odrzucania niepoprawnych składniowo wyrażeń przed rozpoczęciem ewaluacji, aby nie powtarzać tych kontroli podczas 40 000 ewaluacji (100×100 komórek, każda z czterema rogami) poniższej funkcji.

Te etapy parsowania i sprawdzania łączy w sobie funkcja `parseAndCheck`:

code/r07/surface

```
import "code/r07/eval"

func parseAndCheck(s string) (eval.Expr, error) {
    if s == "" {
        return nil, fmt.Errorf("puste wyrażenie")
    }
    expr, err := eval.Parse(s)
    if err != nil {
        return nil, err
    }
    vars := make(map[eval.Var]bool)
    if err := expr.Check(vars); err != nil {
        return nil, err
    }
    for v := range vars {
        if v != "x" && v != "y" && v != "r" {
            return nil, fmt.Errorf("niezdefiniowana zmienna: %s", v)
        }
    }
    return expr, nil
}
```

Aby ta aplikacja stała się aplikacją internetową, potrzebujemy tylko poniższej funkcji `plot`, która ma znajomą sygnaturę funkcji `http.HandlerFunc`.

```
func plot(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    expr, err := parseAndCheck(r.Form.Get("expr"))
    if err != nil {
        http.Error(w, "nieprawidłowe wyrażenie: "+err.Error(), http.StatusBadRequest)
        return
    }
    w.Header().Set("Content-Type", "image/svg+xml")
    surface(w, func(x, y float64) float64 {
        r := math.Hypot(x, y) // odległość od punktu (0,0)
        return expr.Eval(eval.Env{"x": x, "y": y, "r": r})
    })
}
```

Kup książkę Poleć książkę

Funkcja `plot` parsuje i sprawdza wyrażenie określone w żądaniu HTTP, i używa go do utworzenia anonimowej funkcji o dwóch zmiennych. Ta anonimowa funkcja ma taką samą sygnaturę jak sztywno ustalona funkcja `f` z oryginalnego programu drukowania wykresu powierzchniowego, ale ewaluuje wyrażenie dostarczane przez użytkownika. Środowisko definiuje `x`, `y` oraz promień `r`. Na koniec funkcja `plot` wywołuje funkcję `surface`, która jest po prostu funkcją `main` z programu `code/r03/surface`, zmodyfikowaną, by przyjmować jako parametry funkcję drukowania wykresu oraz dane wyjściowe z `io.Writer`, zamiast używać sztywno ustalonej funkcji `f` i `os.Stdout`. Na rysunku 7.7 przedstawiono trzy wykresy powierzchniowe wygenerowane przez ten program.

Ćwiczenie 7.13. Dodaj do typu `Expr` metodę `String`, aby w ładny sposób formatować drzewo składniowe. Sprawdź, czy wyniki po ponownym parsowaniu dają równoważne drzewo.

Ćwiczenie 7.14. Zdefiniuj nowy typ konkretny, który spełnia warunki interfejsu `Expr` i zapewnia nową operację, taką jak obliczanie minimalnej wartości swoich operandów. Ponieważ funkcja `Parse` nie tworzy instancji tego nowego typu, aby go użyć, trzeba będzie zbudować drzewo składniowe bezpośrednio (lub rozszerzyć parser).

Ćwiczenie 7.15. Napisz program, który odczytuje pojedyncze wyrażenie ze standardowego strumienia wejściowego, prosi użytkownika o podanie wartości dla dowolnych zmiennych, a następnie ewaluuje to wyrażenie w powstałym środowisku. Obsłuż elegancko wszystkie błędy.

Ćwiczenie 7.16. Napisz internetową aplikację kalkulatora.

7.10. Asercje typów

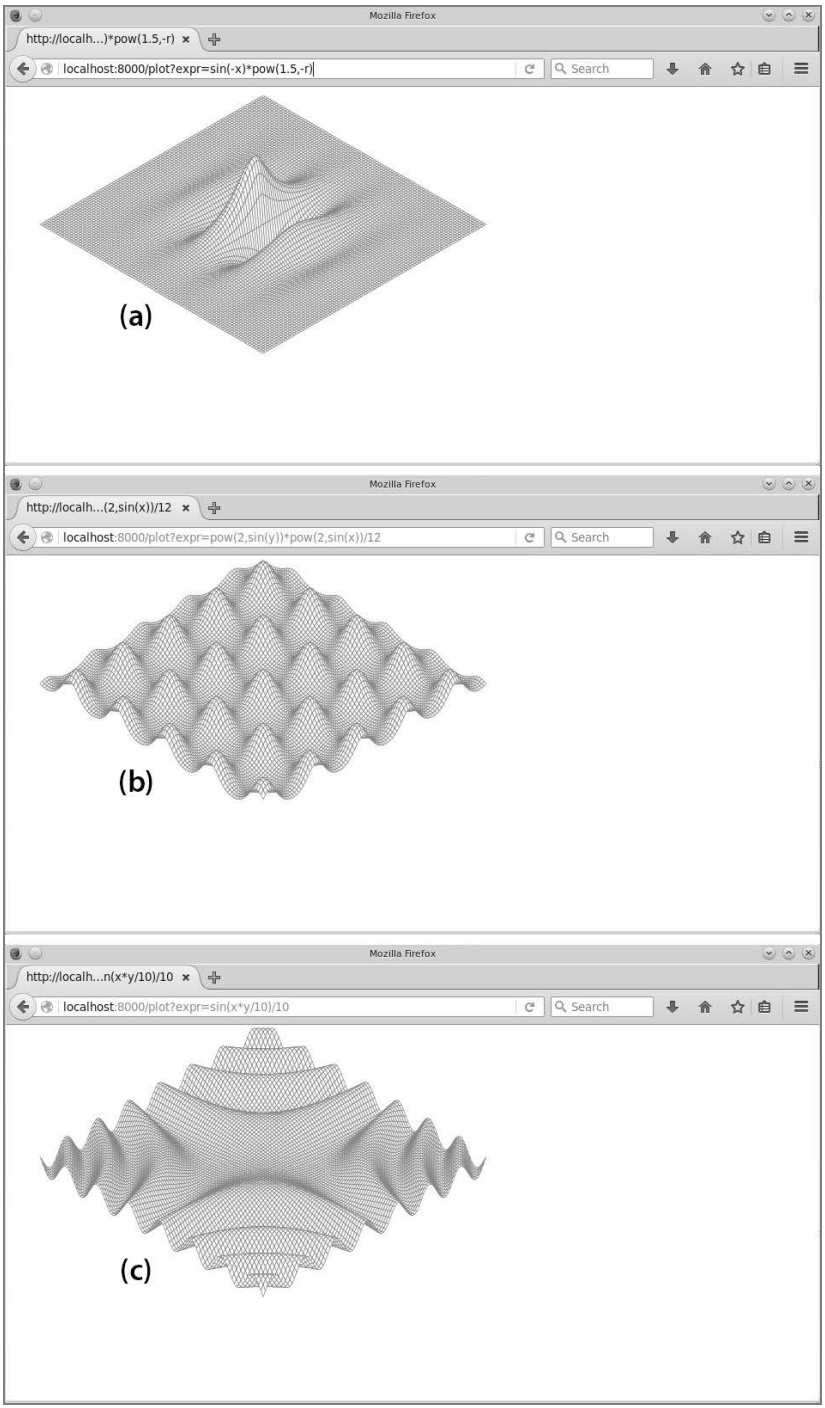
Asercja typu (ang. *type assertion*) jest operacją stosowaną do wartości interfejsu. Składniowo wygląda to jak `x.(T)`, gdzie `x` jest wyrażeniem typu interfejsu, a `T` jest typem (zwanym typem zakładanym). Asercja typu sprawdza, czy dynamiczny typ jej operandu jest zgodny z typem zakładanym.

Istnieją dwie możliwości. Po pierwsze, jeśli zakładany typ `T` jest typem konkretnym, asercja typu sprawdza, czy dynamiczny typ wyrażenia `x` jest **identyczny** z `T`. Jeżeli to sprawdzenie zakończy się pomyślnie, wynikiem asercji typu jest dynamiczna wartość wyrażenia `x`, której typem jest oczywiście `T`. Innymi słowy: asercja typu do typu konkretnego wyodrębnia konkretną wartość ze swojego operandu. Jeśli sprawdzenie się nie powiedzie, operacja uruchamia procedurę *panic*. Oto przykład:

```
var w io.Writer
w = os.Stdout
f := w.(*os.File)      // powodzenie: f == os.Stdout
c := w.(*bytes.Buffer) // panic: interfejs przechowuje typ *os.File, a nie *bytes.Buffer
```

W drugim przypadku, jeśli zakładanym typem `T` jest typ interfejsowy, asercja typu sprawdza, czy dynamiczny typ wyrażenia `x` **spełnia** warunki typu `T`. Jeżeli kontrola zakończy się powodzeniem, wartość dynamiczna nie jest wyodrębniana. Wynikiem jest nadal wartość interfejsu z tymi samymi komponentami typu i wartości, ale wynik posiada typ interfejsowy `T`. Innymi słowy: asercja typu do typu interfejsowego zmienia typ wyrażenia, udostępniając inny (i zwykle większy) zestaw metod, ale zachowuje wewnątrz wartości interfejsu komponenty, którymi są dynamiczny typ i dynamiczna wartość.

Kup książkę Poleć książkę



Rysunek 7.7. *Wykresy powierzchniowe trzech funkcji: (a) sin(–x)*pow(1.5,–r), (b) pow(2,sin(y))*pow(2,sin(x))/12, (c) sin(x*y/10)/10*

Kup książkę Poleć książkę

Po pierwszej, poniższej asercji typu zarówno w, jak i rw przechowują os.Stdout, więc każda ma typ dynamiczny *os.File, ale zmienna w typu *io.Writer udostępnia tylko metodę Write danego pliku, podczas gdy rw udostępnia również jego metodę Read.

```
var w io.Writer
w = os.Stdout
rw := w.(io.ReadWriter) // powodzenie: *os.File ma obie metody: Read i Write

w = new(ByteCounter)
rw = w.(io.ReadWriter) // panic: *ByteCounter nie ma metody Read
```

Bez względu na to, jaki był typ zakładany, asercja typu nie powiedzie się, jeśli operandem jest wartość nil interfejsu. Asercja typu do mniej restrykcyjnego typu interfejsowego (takiego z mniejszą liczbą metod) jest rzadko potrzebna, ponieważ zachowuje się jak przypisanie, z wyjątkiem przypadku nil.

```
w = rw             // io.ReadWriter jest przypisywalny do io.Writer
w = rw.(io.Writer) // nie powiedzie się tylko, jeśli rw == nil
```

Często nie jesteśmy pewni dynamicznego typu wartości interfejsu i chcielibyśmy sprawdzić, czy to jest jakiś szczególny typ. Jeśli asercja typu pojawia się w przypisaniu, w którym oczekiwane są dwa wyniki (tak jak w poniższych deklaracjach), operacja nie wywołuje paniki w przypadku niepowodzenia, ale zamiast tego zwraca dodatkowy drugi wynik, czyli wartość logiczną wskazującą powodzenie:

```
var w io.Writer = os.Stdout
f, ok := w.(*os.File)      // powodzenie: ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // niepowodzenie: !ok, b == nil
```

Ten drugi wynik jest tradycyjnie przypisywany do zmiennej o nazwie ok. Jeśli operacja się nie powiedzie, ok jest fałszem, a pierwszy wynik jest równy wartości zerowej zakładanego typu, którą w tym przykładzie jest *bytes.Buffer z wartością nil.

Wynik jest często od razu wykorzystywany do zdecydowania, co robić dalej. Rozszerzona forma instrukcji if pozwala zapisać to dość zwięźle:

```
if f, ok := w.(*os.File); ok {
    // ...użycie f...
}
```

Jeśli operand asercji typu jest zmienną, to zamiast wymyślonej kolejnej nazwy dla nowej zmiennej lokalnej można czasem zobaczyć ponownie wykorzystaną pierwotną nazwę przesłaniającą oryginał, np.:

```
if w, ok := w.(*os.File); ok {
    // ...użycie w...
}
```

7.11. Rozróżnianie błędów za pomocą asercji typów

Rozważmy zestaw błędów zwracanych przez operacje plików w pakiecie os. Operacje we-wy mogą się nie powieść z wielu różnych powodów, ale trzy rodzaje awarii często muszą być obsługiwane odmiennie: plik już istnieje (dla operacji tworzenia), nie znaleziono pliku (dla operacji odczytu) oraz odmowa dostępu. Pakiet os zapewnia te trzy funkcje pomocnicze do klasyfikowania błędu sygnalizowanego przez daną wartość error:

Kup książkę Poleć książkę

```
package os

func IsExist(err error) bool
func IsNotExist(err error) bool
func IsPermission(err error) bool
```

Naiwna implementacja jednego z tych predykatów może sprawdzać, czy komunikat o błędzie zawiera określony podłańcuch znaków:

```
func IsNotExist(err error) bool {
    // UWAGA: to nie jest solidne rozwiązanie!
    return strings.Contains(err.Error(), "plik nie istnieje")
}
```

Ponieważ jednak logika wykorzystywana do obsługi błędów we-wy może się różnić w zależności od platformy, podejście to nie jest solidne i ta sama awaria może być raportowana za pomocą wielu różnych komunikatów błędów. Sprawdzanie podłańcuchów komunikatów o błędach może być przydatne podczas testowania, które ma na celu upewnienie się, że funkcje zawodzą w oczekiwany sposób, ale jest nieodpowiednie dla kodu działającego w środowisku produkcyjnym.

Bardziej niezawodnym podejściem jest reprezentowanie ustrukturyzowanych wartości błędów za pomocą dedykowanego typu. Pakiet `os` definiuje typ o nazwie `PathError`, służący do opisywania awarii z udziałem operacji na ścieżce pliku, takich jak `Open` lub `Delete`. Definiuje też wariant o nazwie `LinkError`, opisujący awarie operacji z udziałem dwóch ścieżek plików, takie jak `Symlink` i `Rename`. Oto typ `os.PathError`:

```
package os

// Typ PathError rejestruje błąd oraz operację i ścieżkę pliku, które go wywołały.
type PathError struct {
    Op  string
    Path string
    Err  error
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

Większość klientów jest nieświadoma typu `PathError` i radzi sobie ze wszystkimi błędami w jednolity sposób, wywołując swoje metody `Error`. Chociaż metoda `Error` typu `PathError` formuje komunikat, po prostu konkatenując pola, struktura `PathError` zachowuje bazowe komponenty błędu. Klienty wymagające odróżniania jednego rodzaju awarii od innego mogą użyć asercji typu do wykrywania określonego rodzaju błędu. Taki określony rodzaj błędu zapewnia więcej szczegółów niż prosty łańcuch znaków.

```
_, err := os.Open("/plik/nie/istnieje")
fmt.Println(err) // "open /plik/nie/istnieje: nie ma takiego pliku lub katalogu"
fmt.Printf("%#v\n", err)
// Output:
// &os.PathError{Op:"open", Path:"/plik/nie/istnieje", Err:0x2}
```

Oto, jak działają te trzy funkcje pomocnicze. Przykładowo: pokazana poniżej funkcja `IsNotExist` raportuje, czy błąd jest równy `syscall.ENOENT` (zob. podrozdział 7.8), czy szczególnemu błędowi `os.ErrNotExist` (zob. `io.EOF` w punkcie 5.4.2), albo czy jest typem `*PathError`, którego błędem jest jeden z tych dwóch.

Kup książkę Poleć książkę

```
import (
    "errors"
    "syscall"
)

var ErrNotExist = errors.New("plik nie istnieje")

// IsNotExist zwraca wartość logiczną wskazującą, czy błąd jest znany,
// aby zgłosić, że plik lub katalog nie istnieją. Jej warunki są spełniane
// przez ErrNotExist oraz przez niektóre błędy wywołań systemowych.
func IsNotExist(err error) bool {
    if pe, ok := err.(*PathError); ok {
        err = pe.Err
    }
    return err == syscall.ENOENT || err == ErrNotExist
}
```

A tutaj ta funkcja w akcji:

```
_, err := os.Open("/plik/nie/istnieje")
fmt.Println(os.IsNotExist(err)) // "true"
```

Oczywiście struktura `PathError` zostaje utracona, jeśli komunikat o błędzie zostanie połączony w większy łańcuch znaków, np. poprzez wywołanie funkcji `fmt.Errorf`. Rozróżnianie błędów musi być przeprowadzane natychmiast po nieudanej operacji, zanim błąd zostanie propagowany do podmiotu wywołującego.

7.12. Kwerendowanie zachowań za pomocą interfejsowych asercji typów

Poniższa logika jest podobna do części serwera WWW `net/http` odpowiedzialnego za wypisywanie pól nagłówka HTTP takich jak `"Content-type: text/html"`. Zmienna `w` typu `io.Writer` reprezentuje odpowiedź HTTP. Zapisywane w niej bajty są ostatecznie wysłane do czyjejś przeglądarki internetowej.

```
func writeHeader(w io.Writer, contentType string) error {
    if _, err := w.Write([]byte("Content-Type: ")); err != nil {
        return err
    }
    if _, err := w.Write([]byte(contentType)); err != nil {
        return err
    }
    //...
}
```

Ponieważ metoda `Write` wymaga wycinka bajtów, a wartość, którą chcemy zapisać, to łańcuch znaków, wymagana jest konwersja `[]byte(...)`. Ta konwersja alokuje pamięć i tworzy kopię, ale kopia jest wyrzucana niemal natychmiast po utworzeniu. Udajmy, że jest to główny element serwera WWW, a nasze profilowanie wykazało, że ta alokacja pamięci go spowalnia. Czy możemy w tym przypadku uniknąć alokowania pamięci?

Interfejs `io.Writer` wskazuje nam tylko jeden fakt na temat typu konkretnego przechowywanego przez zmienną `w`: można zapisywać w nim bajty. Jeśli zajrzymy za kulisy pakietu `net/http`, zobaczymy, że dynamiczny typ przechowywany w tym programie przez `w` ma również metodę `WriteString`, która umożliwia efektywne zapisywanie w nim łańcuchów znaków, co pozwala uniknąć konieczności alokowania tymczasowej kopii. (Może to wyglądać na strzał w ciemno, ale wiele ważnych typów

Kup książkę Poleć książkę

spełniających warunki interfejsu `io.Writer` również ma metodę `WriteString`, a należą do nich m.in.: `*bytes.Buffer`, `*os.File` i `*bufio.Writer`).

Nie możemy zakładać, że dowolna zmienna `w` typu `io.Writer` ma również metodę `WriteString`. Ale możemy zdefiniować nową instancję, która ma właśnie tę metodę, i użyć asercji typu w celu sprawdzenia, czy dynamiczny typ zmiennej `w` spełnia warunki tego nowego interfejsu.

```
// writeString zapisuje s w zmiennej w.
// Jeśli w ma metodę WriteString, jest ona wywoływana zamiast w.Write.
func writeString(w io.Writer, s string) (n int, err error) {
    type stringWriter interface {
        WriteString(string) (n int, err error)
    }
    if sw, ok := w.(stringWriter); ok {
        return sw.WriteString(s) // unikanie tworzenia kopii
    }
    return w.Write([]byte(s)) // alokowanie tymczasowej kopii
}

func writeHeader(w io.Writer, contentType string) error {
    if _, err := writeString(w, "Content-Type: "); err != nil {
        return err
    }
    if _, err := writeString(w, contentType); err != nil {
        return err
    }
    //...
}
```

Aby uniknąć powtarzania się, przenieśliśmy to sprawdzanie do funkcji narzędziowej `writeString`, ale jest ona tak bardzo przydatna, że standardowa biblioteka zapewnia ją jako `io.WriteString`. Jest to rekomendowany sposób zapisywania łańcucha znaków do `io.Writer`.

W tym przypadku ciekawe jest to, że nie ma standardowego interfejsu, który definiowałby metodę `WriteString` i określał jej wymagane zachowanie. Ponadto kwestia spełniania przez konkretny typ warunków interfejsu `stringWriter` zależy wyłącznie od jego metod, a nie od jakiejkolwiek relacji między nim a typem interfejsowym. Oznacza to, że zastosowana w powyższym przykładzie technika opiera się na założeniu, że **jeśli** typ spełnia warunki poniższego interfejsu, **wtedy** `WriteString(s)` musi mieć taki sam efekt co `Write([]byte(s))`.

```
interface {
    io.Writer
    WriteString(s string) (n int, err error)
}
```

Chociaż `io.WriteString` dokumentuje swoje założenie, prawdopodobnie niewiele wywołujących ją funkcji dokumentuje, że również przyjmuje to samo założenie. Definiowanie metody określonego typu jest przyjmowane jako dorozumiane wyrażenie zgody na konkretny kontrakt behawioralny. Początkujący programiści języka Go, zwłaszcza ci z doświadczeniem w silnie typowanych językach, mogą uznać ten brak wyraźnej intencji za niepokojący, ale nieczęsto jest to problemem w praktyce. Z wyjątkiem pustego interfejsu `interface{}`, warunki typów interfejsowych są rzadko spełniane w wyniku przypadkowego zbiegu okoliczności.

Powyższa funkcja `writeString` używa asercji typu do sprawdzenia, czy wartość ogólnego typu interfejsowego spełnia również warunki bardziej szczegółowego typu interfejsowego, i jeśli tak jest, używa zachowań tego bardziej szczegółowego interfejsu. Z tej techniki można zrobić dobry użytek

Kup książkę Poleć książkę

bez względu na to, czy kwerendowany interfejs jest standardowy, jak `io.ReadWriter`, czy zdefiniowany przez użytkownika, jak `stringWriter`.

Chodzi również o to, w jaki sposób funkcja `fmt.Fprintf` odróżnia wartości spełniające warunki interfejsu `error` lub `fmt.Stringer` od wszystkich pozostałych wartości. W ramach funkcji `fmt.Fprintf` wykonywana jest czynność konwertująca pojedynczy operand na łańcuch znaków, która wygląda mniej więcej tak:

```
package fmt

func formatOneValue(x interface{}) string {
    if err, ok := x.(error); ok {
        return err.Error()
    }
    if str, ok := x.(Stringer); ok {
        return str.String()
    }
    // …wszystkie pozostałe typy…
}
```

Jeśli `x` spełnia warunki jednego z tych dwóch interfejsów, określa to sposób formatowania wartości. Jeśli nie, domyślny przypadek obsługuje wszystkie pozostałe typy mniej lub bardziej jednolicie za pomocą refleksji. Zobaczymy, jak to działa, w rozdziale 12.

Ponownie przyjęte jest założenie, że każdy typ z metodą `String` spełnia warunek behawioralnego kontraktu interfejsu `fmt.Stringer`, jakim jest zwracanie łańcucha znaków odpowiedniego do wyświetlania.

7.13. Przełączniki typów

Interfejsy są używane w dwóch różnych stylach. W pierwszym stylu, którego przykładami są: `io.Reader`, `io.Writer`, `fmt.Stringer`, `sort.Interface`, `http.Handler` i `error`, metody interfejsów wyrażają podobieństwa typów konkretnych, spełniających warunki danego interfejsu, ale ukrywają szczegóły reprezentacji i wewnętrzne operacje tych typów konkretnych. Nacisk kładzie się na metody, a nie na typy konkretne.

Drugi styl wykorzystuje zdolność wartości interfejsu do przechowywania wartości różnych typów konkretnych i traktuje interfejs jako **unię** tych typów. Asercje typów są wykorzystywane do rozróżniania tych typów dynamicznie i traktowania każdego przypadku odmiennie. W tym stylu nacisk kładzie się na typy konkretne, które spełniają warunki danego interfejsu, a nie na metody tego interfejsu (jeśli w rzeczywistości w ogóle ma jakieś), i nie ma ukrywania informacji. Używane w ten sposób interfejsy opiszemy jako **unie rozróżnialne** (ang. *discriminated unions*).

Jeśli jesteś zaznajomiony z programowaniem obiektowym, możesz rozpoznać te dwa style jako **polimorfizm podtypowy** i **polimorfizm *ad hoc***, ale nie musisz zapamiętywać tych pojęć. W pozostałej części tego rozdziału będziemy prezentować przykłady drugiego stylu.

Interfejs API języka Go do kwerendowania bazy danych SQL, tak jak interfejsy innych języków, pozwala precyzyjnie oddzielić stałą część zapytania od części zmiennych. Przykładowy klient może wyglądać tak:

```
import "database/sql"

func listTracks(db sql.DB, artist string, minYear, maxYear int) {
    result, err := db.Exec(
```

Kup książkę Poleć książkę

```
            "SELECT * FROM tracks WHERE artist = ? AND ? <= year AND year <= ?",
            artist, minYear, maxYear)
        //...
    }
```

Metoda `Exec` zastępuje w łańcuchu zapytania każdy znak `'?'` literałem SQL oznaczającym odpowiednią wartość argumentu, która może być wartością logiczną, liczbą, łańcuchem znaków lub wartością `nil`. Konstruowanie zapytań w ten sposób pomaga uniknąć ataków wstrzykiwania SQL, w których atakujący przejmuje kontrolę nad zapytaniem, wykorzystując niewłaściwe cytowanie danych wejściowych. W ramach metody `Exec` moglibyśmy znaleźć funkcję taką jak poniższa, która konwertuje każdą wartość argumentu na notację w postaci literału SQL.

```
func sqlQuote(x interface{}) string {
    if x == nil {
        return "NULL"
    } else if _, ok := x.(int); ok {
        return fmt.Sprintf("%d", x)
    } else if _, ok := x.(uint); ok {
        return fmt.Sprintf("%d", x)
    } else if b, ok := x.(bool); ok {
        if b {
            return "TRUE"
        }
        return "FALSE"
    } else if s, ok := x.(string); ok {
        return sqlQuoteString(s) // (niepokazane)
    } else {
        panic(fmt.Sprintf("nieoczekiwany typ %T: %v", x, x))
    }
}
```

Instrukcja przełącznika (`switch`) upraszcza łańcuch `if-else`, który wykonuje serię testów porównań wartości. Analogiczna instrukcja **przełącznika typów** upraszcza łańcuch `if-else` asercji typów.

W najprostszej formie przełącznik typów wygląda jak zwykła instrukcja `switch`, w której operandem jest `x.(type)` — jest to dosłownie słowo kluczowe `type` — a każdy przypadek (`case`) ma jeden typ lub kilka typów. Przełącznik typów umożliwia tworzenie wielokrotnego wyboru na podstawie dynamicznego typu wartości interfejsu. Przypadek `nil` zostaje dopasowany, jeśli `x == nil`, a przypadek `default` zostaje dopasowany, jeśli nie pasuje żaden inny. Przełącznik typów dla funkcji `sqlQuote` miałby następujące przypadki:

```
switch x.(type) {
case nil:       //...
case int, uint: //...
case bool:      //...
case string:    //...
default:        //...
}
```

Podobnie jak w zwykłej instrukcji przełącznika (zob. podrozdział 1.8), przypadki są rozpatrywane w kolejności, a gdy zostaje znalezione dopasowanie, wykonywane jest ciało danego przypadku. Porządek przypadków staje się istotny, gdy jeden typ przypadku lub kilka typów przypadku to interfejsy, ponieważ wtedy istnieje możliwość dopasowania dwóch przypadków. Pozycja przypadku `default` w stosunku do pozostałych jest nieistotna. Nie jest dozwolone wykonywanie wszystkich przypadków po kolei (`fallthrough`).

Kup książkę Poleć książkę

Należy zwrócić uwagę, że w pierwotnej funkcji logika dla przypadków `bool` i `string` wymaga dostępu do wartości wyodrębnianej przez asercję typu. Ponieważ jest to typowe, instrukcja przełączania typów ma rozszerzoną formę, która wiąże wyodrębnioną wartość z nową zmienną w obrębie każdego przypadku:

```
switch x := x.(type) { /* ... */ }
```

Tutaj również nazwaliśmy nowe zmienne `x`. Podobnie jak w przypadku asercji typów, ponowne wykorzystywanie nazw zmiennych jest powszechne. Tak jak instrukcja `switch`, przełącznik typów domyślnie tworzy blok leksykalny, więc deklaracja nowej zmiennej o nazwie `x` nie koliduje ze zmienną `x` w bloku zewnętrznym. Każda instrukcja `case` również domyślnie tworzy osobny blok leksykalny.

Przepisanie funkcji `sqlQuote` tak, aby wykorzystywała rozszerzoną formę przełącznika typów, sprawia, że staje się ona znacznie jaśniejsza:

```
func sqlQuote(x interface{}) string {
    switch x := x.(type) {
    case nil:
        return "NULL"
    case int, uint:
        return fmt.Sprintf("%d", x) // x ma tutaj typ interface{}
    case bool:
        if x {
            return "TRUE"
        }
        return "FALSE"
    case string:
        return sqlQuoteString(x) // (niepokazane)
    default:
        panic(fmt.Sprintf("nieoczekiwany typ %T: %v", x, x))
    }
}
```

W tej wersji w obrębie bloku każdego przypadku dla pojedynczego typu zmienna `x` ma ten sam typ co przypadek. Zmienna `x` ma np. typ `bool` w obrębie przypadku `bool`, a typ `string` w obrębie przypadku `string`. We wszystkich pozostałych przypadkach `x` ma (interfejsowy) typ operandu instrukcji `switch`, którym w tym przykładzie jest `interface{}`. Gdy ta sama akcja jest wymagana dla kilku przypadków, tak jak dla `int` i `uint`, przełącznik typów ułatwia ich połączenie.

Chociaż `sqlQuote` przyjmuje argument dowolnego typu, funkcja ta może być wykonana do końca tylko wtedy, kiedy typ argumentu odpowiada jednemu z przypadków umieszczonych w przełączniku typów. W przeciwnym razie uruchamiana jest procedura *panic* z komunikatem „nieoczekiwany typ”. Chociaż typem zmiennej `x` jest `interface{}`, traktujemy ją jako **unię rozróżnialną** typów `int`, `uint`, `bool`, `string` i `nil`.

7.14. Przykład: dekodowanie XML oparte na tokenach

W podrozdziale 4.5 pokazaliśmy, jak dekodować dokumenty JSON na struktury danych języka Go za pomocą funkcji `Marshal` i `Unmarshal` z pakietu `encoding/json`. Pakiet `encoding/xml` zapewnia podobny interfejs API. To podejście jest wygodne, gdy chcemy zbudować reprezentację drzewa dokumentu, ale w wielu programach jest ono zbędne. Ten pakiet zapewnia również interfejs API niskiego poziomu **oparty na tokenach**, przeznaczony do dekodowania dokumentów XML.

Kup książkę Poleć książkę

W stylu opartym na tokenach parser konsumuje dane wejściowe i wytwarza strumień tokenów, głównie czterech rodzajów (`StartElement`, `EndElement`, `CharData` i `Comment`), z których każdy jest typem konkretnym w pakiecie `encoding/xml`. Każde wywołanie `(*xml.Decoder).Token` zwraca token.

Istotne części tego interfejsu API zostały pokazane poniżej:

encoding/xml

```
package xml

type Name struct {
    Local string // np. "Tytuł" lub "id"
}

type Attr struct { // np. name="wartość"
    Name  Name
    Value string
}

// Typ Token obejmuje typy: StartElement, EndElement, CharData
// i Comment oraz kilka innych osobliwych typów (niepokazanych).
type Token interface{}
type StartElement struct { // np. <name>
    Name Name
    Attr []Attr
}
type EndElement struct { Name Name } // np. </name>
type CharData []byte                 // np. <p>CharData</p>
type Comment []byte                  // np. <!-- Comment -->

type Decoder struct{ /* ... */ }

func NewDecoder(io.Reader) *Decoder
func (*Decoder) Token() (Token, error) // zwraca następny Token w sekwencji
```

Interfejs `Token`, który nie ma metod, jest również przykładem unii rozróżnialnej. Celem tradycyjnego interfejsu, takiego jak `io.Reader`, jest ukrycie szczegółów dotyczących typów konkretnych spełniających jego warunki, aby można było tworzyć nowe implementacje. Każdy typ konkretny jest traktowany jednakowo. Natomiast zestaw typów konkretnych spełniających warunki unii rozróżnialnej jest z założenia ustalony i jest udostępniany, a nie ukryty. Typy unii rozróżnialnej mają niewiele metod. Operujące na nich funkcje są za pomocą przełącznika typów wyrażane jako zestaw przypadków, z różną logiką w każdym przypadku.

Poniższy program `xmlselect` wyodrębnia i wyświetla tekst znajdujący się pomiędzy określonymi elementami w drzewie dokumentu XML. Jeśli skorzystamy z powyższego interfejsu API, ten program może wykonać swoje zadanie w pojedynczym przejściu przez dane wejściowe w ogóle bez konieczności materializowania drzewa.

code/r07/xmlselect

```
// Xmlselect wyświetla tekst wybranych elementów dokumentu XML.
package main

import (
    "encoding/xml"
    "fmt"
    "io"
    "os"
    "strings"
)
```

Kup książkę Poleć książkę

```
func main() {
    dec := xml.NewDecoder(os.Stdin)
    var stack []string // stos nazw elementów
    for {
        tok, err := dec.Token()
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Fprintf(os.Stderr, "xmlselect: %v\n", err)
            os.Exit(1)
        }
        switch tok := tok.(type) {
        case xml.StartElement:
            stack = append(stack, tok.Name.Local) // umieszczenie na stosie
        case xml.EndElement:
            stack = stack[:len(stack)-1] // zdjęcie ze stosu
        case xml.CharData:
            if containsAll(stack, os.Args[1:]) {
                fmt.Printf("%s: %s\n", strings.Join(stack, " "), tok)
            }
        }
    }
}

// containsAll raportuje, czy x zawiera elementy y w kolejności.
func containsAll(x, y []string) bool {
    for len(y) <= len(x) {
        if len(y) == 0 {
            return true
        }
        if x[0] == y[0] {
            y = y[1:]
        }
        x = x[1:]
    }
    return false
}
```

Za każdym razem, gdy pętla w funkcji `main` napotka token `StartElement`, umieszcza nazwę danego elementu na stosie, a dla każdego tokena `EndElement` zdejmuje nazwę ze stosu. Interfejs API gwarantuje, że kolejność tokenów `StartElement` i `EndElement` zostanie prawidłowo dopasowana nawet w niepoprawnym składniowo dokumencie XML. Tokeny `Comment` są ignorowane. Gdy program `xmlselect` napotyka token `CharData`, wyświetla tekst tylko wtedy, gdy stos zawiera w kolejności wszystkie elementy nazwane przez argumenty wiersza poleceń.

Poniższe polecenie wyświetla teksty wszystkich elementów `h2` pojawiających się pod dwoma poziomami elementów `div`. Jego dane wejściowe to specyfikacja XML, która sama jest dokumentem XML.

```
$ go build code/r01/fetch
$ ./fetch http://www.w3.org/TR/2006/REC-xml11-20060816 | ./xmlselect div div h2
html body div div h2: 1 Introduction
html body div div h2: 2 Documents
html body div div h2: 3 Logical Structures
html body div div h2: 4 Physical Structures
html body div div h2: 5 Conformance
html body div div h2: 6 Notation
html body div div h2: A References
html body div div h2: B Definitions for Character Normalization
...
```

Kup książkę Poleć książkę

Ćwiczenie 7.17. Rozszerz program `xmlselect` w taki sposób, żeby elementy można było wybierać nie tylko według nazwy, ale również według ich atrybutów, tak jak w CSS, aby np. element `<div id="Page" class="wide">` można było wybrać poprzez dopasowanie zarówno atrybutu `id` lub `class`, jak i jego nazwy.

Ćwiczenie 7.18. Używając interfejsu dekodera API opartego na tokenach, napisz program, który odczyta dowolny dokument XML i zbuduje reprezentujące go drzewo węzłów generycznych. Istnieją dwa rodzaje węzłów: węzły `CharData` reprezentują tekstowe łańcuchy znaków, a węzły `Element` reprezentują nazwane elementy i ich atrybuty. Każdy węzeł `Element` ma wycinek węzłów potomnych.

```
Pomocna może się okazać następująca deklaracja.
import "encoding/xml"

type Node interface{} // CharData lub *Element

type CharData string

type Element struct {
    Type     xml.Name
    Attr     []xml.Attr
    Children []Node
}
```

7.15. Kilka porad

Przy projektowaniu nowego pakietu początkujący programiści Go często zaczynają od utworzenia zestawu interfejsów, a dopiero później definiują typy konkretne, które spełniają ich warunki. Takie podejście prowadzi do powstawania wielu interfejsów, z których każdy ma tylko jedną implementację. Nie rób tak. Takie interfejsy są niepotrzebnymi abstrakcjami. Mają też swoje koszty w czasie wykonywania programu. Wykorzystując mechanizm eksportu (zob. podrozdział 6.6), można ograniczyć, które metody typu lub pola struktury są widoczne na zewnątrz pakietu. Interfejsy są potrzebne tylko wtedy, gdy istnieją dwa konkretne typy (lub więcej), które muszą być obsługiwane w jednolity sposób.

Robimy wyjątek od tej reguły, gdy warunki interfejsu są spełniane przez pojedynczy typ konkretny, ale ten typ nie może istnieć w tym samym pakiecie co interfejs z powodu swoich zależności. W takim przypadku interfejs jest dobrym sposobem na oddzielenie dwóch pakietów.

Ponieważ interfejsy są używane w języku Go tylko wtedy, gdy ich warunki są spełniane przez dwa typy lub większą liczbę typów, z konieczności abstrahują od szczegółów jakiejkolwiek konkretnej implementacji. Rezultatem są mniejsze interfejsy z mniejszą liczbą prostszych metod, a często tylko z jedną, tak jak w przypadku `io.Writer` lub `fmt.Stringer`. Niewielkie interfejsy pozwalają łatwiej spełnić ich warunki, gdy pojawiają się nowe typy. Dobrą zasadą praktyczną przy projektowaniu interfejsów jest **prosić tylko o to, czego się potrzebuje**.

Na tym kończy się nasz przewodnik po metodach i interfejsach. Język Go zapewnia doskonałe wsparcie dla obiektowego stylu programowania, ale nie oznacza to, że należy używać wyłącznie tego stylu. Nie wszystko musi być obiektem. Samodzielne funkcje mają swoje miejsce, podobnie jak niezhermetyzowane typy danych. Należy zwrócić uwagę, że przykłady z pierwszych pięciu rozdziałów tej książki wywołują nie więcej niż dwa tuziny metod, takich jak np. `input.Scan`, w przeciwieństwie do wywołań zwykłych funkcji, takich jak np. `fmt.Printf`.

Kup książkę Poleć książkę

Skorowidz

Kup książkę Poleć książkę

G

H

I

J

K

L

Ł

M

N

Kup książkę Poleć książkę

O

P

R

S

Kup książkę Poleć książkę

Ś

T

U

V

W

Z

Kup książkę Poleć książkę

Ż

Kup książkę Poleć książkę

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Poznaj język Go
— doskonałe narzędzie dla profesjonalisty!

Język Go jest nazywany „językiem C XXI wieku". Podobnie jak C, umożliwia kompilowanie programów do wydajnego kodu maszynowego, który w natywny sposób współpracuje z poszczególnymi systemami operacyjnymi. Go jest elastycznym narzędziem pozwalającym osiągać maksymalny efekt przy minimalnych środkach. Jest wszechstronny — bardzo dobrze nadaje się do budowania infrastruktury takiej jak serwery sieciowe, do tworzenia narzędzi dla programistów, ale jest też znakomitym językiem do programowania grafiki, aplikacji mobilnych i uczenia maszynowego.

Niniejsza książka jest skierowana do osób, które chcą jak najszybciej rozpocząć tworzenie wydajnego oprogramowania w Go. Autorzy przejrzyście wyjaśnili podstawy tego języka i zasady nim rządzące, a swój wykład uzupełnili setkami interesujących i praktycznych przykładów dobrze napisanego kodu Go. Dzięki temu Czytelnik pozna wszystkie aspekty tego języka, jego najistotniejsze pakiety oraz szeroki zakres zastosowań.

W książce zostały omówione:

- podstawowe koncepcje Go, jego najważniejsze konstrukcje i elementy strukturalne programu
- proste i złożone typy danych, funkcje, metody i interfejsy
- zasady współbieżności implementowanej w Go
- kompilacja i formatowanie programu w Go
- korzystanie z pakietów oraz z bibliotek testowania
- zagadnienia zaawansowane: korzystanie z refleksji i programowanie niskiego poziomu

Alan A.A. Donovan od dwudziestu lat zajmuje się programowaniem. Jest członkiem zespołu Go firmy Google w Nowym Jorku. Od 2005 r. pracuje w firmie Google nad projektami infrastrukturalnymi. Brał udział w opracowaniu autorskiego systemu kompilacji Blaze. Zbudował wiele bibliotek i narzędzi do statycznej analizy programów Go.

Brian W. Kernighan jest profesorem Wydziału Informatyki na Uniwersytecie Princeton. W latach 1969 – 2000 pracował nad językami i narzędziami dla systemu Unix w Centrum Badań Informatycznych firmy Bell Labs. Jest współautorem kilku książek, w tym takich jak *Język ANSI C. Programowanie. Wydanie II* (Helion, 2010) i *Lekcja programowania. Najlepsze praktyki* (Helion, 2011).

Helion

45590 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

Informatyka w najlepszym wydaniu

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Addison-Wesley

ISBN 978-83-283-2467-1

cena: 59,00 zł

sięgnij po WIĘCEJ

KOD KORZYŚCI