F# 4.0
dla zaawansowanych

Wydanie IV

Don Syme
Adam Granicz
Antonio Cisternino

Helion

Apress

Tytuł oryginału: Expert F# 4.0, 4th Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-2943-0

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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/f4zaa4.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/f4zaa4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

O autorach **9**

O recenzentach technicznych **11**

Podziękowania **13**

Rozdział 1. Wprowadzenie **15**
Geneza języka F# 16
O książce 17
Dla kogo przeznaczona jest ta książka? 20

Rozdział 2. Pierwszy program w F# — wprowadzenie do języka **21**
Tworzenie pierwszego programu w F# 21
Używanie bibliotek obiektowych w F# 33
Pobieranie i używanie pakietów 35
Dostęp do zewnętrznych danych za pomocą pakietów języka F# 37
Uruchamianie serwera WWW i udostępnianie danych za pomocą pakietów języka F# 38
Podsumowanie 39

Rozdział 3. Wprowadzenie do programowania funkcyjnego **41**
Liczby i łańcuchy znaków 41
Stosowanie instrukcji warunkowych oraz operatorów && i || 44
Definiowanie funkcji rekurencyjnych 44
Listy 46
Opcje 49
Wprowadzenie do dopasowywania do wzorca 50
Wprowadzenie do wartości w postaci funkcji 54
Podsumowanie 63

Rozdział 4. Wprowadzenie do programowania imperatywnego **65**
Programowanie funkcyjne a programowanie imperatywne 65
Pętle i iteracje w programowaniu imperatywnym 66
Używanie modyfikowalnych rekordów 68

Używanie modyfikowalnych wiązań let ... 70
Praca z tablicami ... 71
Wprowadzenie do imperatywnych kolekcji platformy .NET ... 74
Wyjątki i ich kontrolowanie ... 78
Wywoływanie efektów ubocznych — podstawowe operacje wejścia – wyjścia ... 81
Łączenie programowania funkcyjnego z wydajnymi wstępnymi obliczeniami i pamięcią podręczną z programowania imperatywnego ... 84
Łączenie podejścia funkcyjnego z imperatywnym — programowanie funkcyjne z efektami ubocznymi ... 91
Podsumowanie ... 95

Rozdział 5. Typy w programowaniu funkcyjnym ... 97
Przegląd wybranych prostych definicji typów ... 97
Wprowadzenie do typów generycznych ... 102
Pisanie kodu w wersji generycznej ... 110
Więcej o typach różnego rodzaju ... 115
Wprowadzenie do tworzenia typów pochodnych ... 116
Rozwiązywanie problemów z inferencją typów ... 120
Podsumowanie ... 125

Rozdział 6. Programowanie z wykorzystaniem obiektów ... 127
Wprowadzenie do obiektów i składowych ... 127
Używanie klas ... 130
Dodawanie innych aspektów notacji obiektowej do typów ... 133
Definiowanie typów obiektowych o modyfikowalnym stanie ... 138
Wprowadzenie do interfejsowych typów obiektowych ... 141
Inne techniki implementowania obiektów ... 146
Łączenie podejścia funkcyjnego i obiektów — zwalnianie zasobów ... 151
Rozszerzanie istniejących typów i modułów ... 157
Używanie obiektów języka F# i typów platformy .NET ... 160
Podsumowanie ... 163

Rozdział 7. Hermetyzacja i porządkowanie kodu ... 165
Ukrywanie elementów ... 165
Porządkowanie kodu za pomocą przestrzeni nazw i modułów ... 170
Projekty, podzespoły i kolejność kompilacji ... 174
Posługiwanie się plikami z sygnaturami ... 179
Ponowne wykorzystanie kodu ... 181
Tworzenie i udostępnianie pakietów ... 181
Podsumowanie ... 182

Rozdział 8. Praca z danymi tekstowymi ... 183
Budowanie łańcuchów znaków i formatowanie danych ... 183
Parsowanie łańcuchów znaków i danych tekstowych ... 188
Stosowanie wyrażeń regularnych ... 190
Używanie XML-a do reprezentowania formatów konkretnych ... 195
Użycie JSON-a do reprezentowania formatów konkretnych ... 201
Rekurencyjne parsowanie zstępujące ... 204
Parsowanie i formatowanie binarne ... 207
Podsumowanie ... 211

Rozdział 9. Praca z sekwencjami i danymi o strukturze drzewiastej ... 213
Wprowadzenie do sekwencji ... 213
Więcej o pracy z sekwencjami ... 219
Struktura poza sekwencjami — modelowanie dziedzin ... 227
Wzorce aktywne — widoki danych strukturalnych ... 234
Sprawdzanie równości, haszowanie i porównywanie ... 239
Wywołania ogonowe i programowanie rekurencyjne ... 244
Podsumowanie ... 251

Rozdział 10. Programowanie z użyciem liczb i wykresy ... 253
Wprowadzenie do FsLab ... 253
Tworzenie podstawowych wykresów za pomocą biblioteki FSharp.Charting ... 254
Podstawowe literały i typy liczbowe ... 255
Sekwencje, statystyka i kody liczbowe ... 259
Statystyki, algebra liniowa i rozkłady w bibliotece Math.NET ... 266
Szeregi czasowe, ramki danych i biblioteka Deedle ... 271
Jednostki miary ... 273
Podsumowanie ... 279

Rozdział 11. Programowanie reaktywne, asynchroniczne i równoległe ... 281
Wprowadzenie do terminologii ... 282
Zdarzenia ... 283
Operacje asynchroniczne ... 286
Agenty ... 296
Przykład: asynchroniczny agent do obsługi robota internetowego ... 301
Współbieżność oparta na współużytkowanej pamięci ... 305
Podsumowanie ... 310

Rozdział 12. Programowanie symboliczne z użyciem danych strukturalnych ... 311
Sprawdzanie układów za pomocą rachunku zdań ... 311
Upraszczanie wyrażeń i różniczkowanie ... 325
Podsumowanie ... 336

Rozdział 13. Integrowanie zewnętrznych danych i usług ... 337
Podstawowe żądania REST ... 338
Wprowadzenie do kwerend ... 341
Inne możliwości związane z SQL-em ... 346
Podsumowanie ... 353

Rozdział 14. Budowanie inteligentnych aplikacji sieciowych ... 355
Bezpośrednie udostępnianie treści w sieci ... 355
Rozwijanie aplikacji sieciowych z rozbudowanym klientem za pomocą platformy WebSharper ... 360
Podsumowanie ... 406

Rozdział 15. Wizualizacje i graficzny interfejs użytkownika ... 407
Wprowadzenie do biblioteki Eto ... 407
Jak błyskawicznie zbudować aplikację „Witaj, świecie!”? ... 408
Budowa aplikacji graficznej ... 409

Łączenie kontrolek i menu ... 410
Łączenie elementów w interfejs użytkownika ... 413
Rysowanie aplikacji ... 416
Tworzenie przeglądarki fraktali ze zbioru Mandelbrota ... 420
Pisanie własnych kontrolek ... 431
Świat, wygląd i układy współrzędnych ... 435
Lekkie kontrolki ... 444
Podsumowanie ... 450

Rozdział 16. Programowanie zorientowane na język ... 451
Wyrażenia reprezentujące obliczenia ... 452
Używanie mechanizmu refleksji w F# ... 466
Cytowania w języku F# ... 470
Tworzenie dostawców typów w F# ... 474
Podsumowanie ... 476

Rozdział 17. Biblioteki i współdziałanie ... 477
Typy, pamięć i współdziałanie ... 477
Biblioteki — ogólny przegląd ... 478
Używanie typów systemowych ... 481
Inne struktury danych z języka F# i platformy .NET ... 482
Nadzorowanie i izolowanie wykonywania kodu ... 484
Dodatkowe biblioteki związane z refleksją ... 484
Na zapleczu — współdziałanie z C# i innymi językami platformy .NET ... 487
Współdziałanie z językami C i C++ za pomocą technologii PInvoke ... 490
Podsumowanie ... 500

Rozdział 18. Pisanie i testowanie kodu w F# ... 501
Pisanie kodu ... 501
Efektywne używanie F# Interactive ... 507
Używanie diagnostyki z użyciem śledzenia kodu ... 510
Debugowanie kodu z użyciem środowiska IDE ... 513
Testowanie kodu ... 516
Podsumowanie ... 521

Rozdział 19. Projektowanie bibliotek języka F# ... 523
Projektowanie standardowych bibliotek platformy .NET ... 524
Metodyka projektowania funkcyjnego ... 528
Dobry projekt bibliotek w F# ... 530
Wybrane zalecane idiomy ... 537
Podsumowanie ... 539

Ddodatek Krótki przewodnik po języku F# ... 541
Komentarze i atrybuty ... 541
Typy podstawowe i literały ... 541
Typy ... 542
Wzorce i dopasowywanie ... 542
Funkcje, kompozycja i potoki ... 543
Wiązanie i sterowanie przepływem ... 543

Wyjątki ... 544
Krotki, tablice, listy i kolekcje ... 544
Operatory ... 545
Definicje typów i obiekty ... 546
Przestrzenie nazw i moduły ... 547
Przepływy pracy i wyrażenia reprezentujące sekwencje ... 548
Kwerendy i cytowania ... 549

Skorowidz ... 551

ROZDZIAŁ 2

Pierwszy program w F# — wprowadzenie do języka

W tym rozdziale opisaliśmy proste interaktywne programowanie z użyciem F#. Najpierw pobierz F# z witryny *http://fsharp.org* i zainstaluj go (jeśli jeszcze tego nie zrobiłeś). W dalszych podrozdziałach używane jest narzędzie F# Interactive (*fsi.exe* lub *fsharpi*). Pozwala ono interaktywnie wykonywać fragmenty kodu w F#. Jest to wygodne w trakcie eksplorowania języka. W niniejszym rozdziale zobaczysz przykłady najważniejszych konstruktów języka F# i poznasz najistotniejsze biblioteki.

Tworzenie pierwszego programu w F#

Na listingu 2.1 pokazaliśmy pierwszy kompletny program w F#. Możliwe, że początkowo go nie zrozumiesz. Jednak pod listingiem kod jest objaśniony krok po kroku.

Listing 2.1. *Analizowanie łańcucha znaków pod kątem powtarzających się słów*

```
/// Podział łańcucha znaków na słowa w miejscach występowania spacji
let splitAtSpaces (text: string) =
    text.Split ' '
    |> Array.toList

/// Analizowanie łańcucha znaków pod kątem powtarzających się słów
let wordCount text =
    let words = splitAtSpaces text
    let numWords = words.Length
    let distinctWords = List.distinct words
    let numDups = numWords - distinctWords.Length
    (numWords, numDups)

/// Analizowanie łańcucha znaków pod kątem powtarzających się słów i wyświetlanie wyników
let showWordCount text =
    let numWords, numDups = wordCount text
    printfn "--> Liczba słów w tekście: %d" numWords
    printfn "--> Liczba powtórzeń: %d" numDups
```

Wklej ten program do narzędzia F# Interactive. Możesz je uruchomić w następujący sposób:

- Jeśli używasz wiersza poleceń lub terminalu w systemie Linux albo Mac OS X, wywołaj w danym narzędziu polecenie `fsharpi` i wklej kod.

- Jeżeli pracujesz w systemie Windows, wywołaj polecenie `fsi.exe` w wierszu poleceń dla programistów i wklej kod.
- Jeśli posługujesz się Emacsem, zainstaluj pakiet `fsharp-mode` z repozytorium MELPA, zastosuj kombinacje klawiszy *Ctrl+c* i *Ctrl+s*, aby uruchomić narzędzie F# Interactive, a następnie prześlij kod z bufora do tego narzędzia za pomocą kombinacji klawiszy *Ctrl+c* i *Ctrl+r*.
- Jeżeli korzystasz ze środowiska Visual Studio, otwórz skrypt w języku F# i wybierz opcję *F# Interactive* w menu *View*. Aby przenieść kod do narzędzia F# Interactive, zaznacz ten kod i wciśnij kombinację klawiszy *Alt+Enter*.
- Jeśli piszesz kod w środowisku Xamarin Studio, utwórz skrypt w języku F# i wybierz opcję *F# Interactive* w menu *Pads*. Aby przenieść kod do narzędzia F# Interactive, zaznacz ten kod i wciśnij kombinację klawiszy *Ctrl+Enter*.

Narzędzia do edycji kodu w F# są też dostępne w innych edytorach, takich jak Atom i Sublime Text 3. Jeśli pracujesz w wierszu poleceń, pamiętaj, aby wpisać `;;` w celu zakończenia trybu interaktywnego wprowadzania instrukcji. W Visual Studio i Xamarin Studio oraz innych środowiskach interaktywnych nie jest to konieczne — można w nich przesyłać tekst za pomocą podanych wcześniej kombinacji klawiszy.

■ **Wskazówka** Gdy korzystasz ze środowiska **IDE** (ang. *Integrated Development Environment*), zwykle możesz uruchomić narzędzie F# Interactive za pomocą skrótu. Na przykład w środowisku Visual Studio możesz wybrać opcję *F# Interactive* w menu *View* lub wcisnąć kombinację klawiszy *Ctrl+Alt+F*, gdy otwarty jest plik lub skrypt w języku F#. Pojawi się wtedy okno narzędzia F# Interactive i będziesz mógł rozpocząć przesyłanie tekstu do niego; wymaga to zaznaczenia tekstu i wciśnięcia kombinacji *Alt+Enter*.

```
> <Uruchom narzędzie F# Interactive w opisany wcześniej sposób>
F# Interactive version xx.x.xxxxx.x
(Tu wyświetlane są zależne od wersji informacje o narzędziu)
For help type #help;;

> <zastosuj polecenie wysyłania kodu do narzędzia lub
   wklej pokazany wcześniej program> ;;

val splitAtSpaces : text:string -> string list
val wordCount : text:string -> int * int
val showWordCount : text:string -> unit
```

Tu F# Interactive informuje o typach funkcji `splitAtSpaces`, `wordCount` i `showWordCount` (więcej o typach dowiesz się już wkrótce). Słowo kluczowe `val` to skrót od *value* (czyli „wartość”). W programowaniu w F# funkcje są traktowane jak wartości, co opisaliśmy w rozdziale 3. Ponadto narzędzie F# Interactive czasem wyświetla więcej informacji, niż pokazaliśmy w tej książce (na przykład wewnętrzne szczegóły dotyczące wygenerowanych wartości). Jeśli chcesz wypróbować przedstawione fragmenty kodu, możesz pominąć dodatkowe informacje. Na razie zobacz, jak zastosować interaktywnie funkcję `wordCount`:

```
> let (numWords,numDups) = wordCount "All the king's horses and all the king's men";;

val numWords : int = 9
val numDups : int = 2
```

W tym kodzie pokazaliśmy efekt wykonania funkcji `wordCount` i powiązania dwóch zwracanych przez nią wyników z nazwami `numWords` i `numDups`. Analiza wyników pozwala zobaczyć, że dany tekst

zawiera dziewięć słów. Dwa z nich się powtarzają, a siedem występuje tylko raz. Funkcja `showWordCount` wyświetla wyniki jako efekt uboczny działania zamiast zwracać je jako wartość:

```
> showWordCount "Couldn't put Humpty together again";;

--> Liczba słów w tekście: 5
--> Liczba powtórzeń: 0
```

Na podstawie danych wyjściowych można w przybliżeniu stwierdzić, jak kod działa. Teraz przyjrzyjmy się programowi dokładnie.

Dokumentowanie kodu

Zacznijmy od definicji funkcji `wordCount` z listingu 2.1. Pierwszy wiersz definicji nie jest kodem, lecz komentarzem:

```
/// Analizowanie łańcucha znaków pod kątem powtarzających się słów
```

Komentarze to wiersze rozpoczynające się od znaków // lub bloki umieszczone między sekwencjami (* i *). Wiersze rozpoczynające się od trzech ukośników (///) to komentarze XMLDoc, w których w razie potrzeby można umieszczać dodatkowe znaczniki XML-owe. Komentarze z programu można zapisać w jednym pliku *.xml* i przetwarzać za pomocą dodatkowych narzędzi.

Używanie instrukcji let

Przyjrzyj się teraz dwóm pierwszym wierszom funkcji `wordCount` z listingu 2.1. Te wiersze definiują funkcję `wordCount` i lokalną wartość `words`. W obu przypadkach używane jest słowo kluczowe `let`:

```
let wordCount text=
    let words = ...
```

Słowo kluczowe `let` to najważniejsza instrukcja, jaką będziesz stosować, programując w F#. Służy ona do definiowania danych, obliczanych wartości i funkcji. Po lewej stronie instrukcji `let` często znajduje się prosty identyfikator, można tam jednak umieszczać także wzorce (przykłady znajdziesz w punkcie „Używanie krotek” w dalszej części rozdziału) lub nazwę funkcji z listą nazw argumentów — tak jak w przypadku funkcji `wordCount` przyjmującej jeden argument, `text`. Po prawej stronie instrukcji `let`, czyli po znaku `=`, znajduje się wyrażenie.

Wartości i niemodyfikowalność

W innych językach **wartość** (ang. *value*) lokalna jest nazywana **zmienną** (ang. *variable*) lokalną. W F# nie można jednak modyfikować wartości lokalnych po ich zainicjowaniu, chyba że dana wartość lokalna jest jawnie opatrzona modyfikatorem `mutable` (do tego tematu wrócimy w rozdziale 4.). Dlatego programiści języka F# i autorzy jego specyfikacji wolą zwykle posługiwać pojęciem „wartość” zamiast „zmienna”.

W rozdziale 4. zobaczysz, że dane pośrednio wskazywane przez wartości lokalne mogą być modyfikowalne, nawet jeśli dana wartość lokalna taka nie jest. Na przykład wartość lokalna będąca uchwytem do tabeli z haszowaniem nie może zostać zmodyfikowana tak, by wskazywała inną tabelę. Natomiast zawartość samej tabeli można modyfikować, wywołując operacje, które dodają i usuwają elementy tej tabeli. Liczne wartości i struktury danych w F# są jednak całkowicie **niemodyfikowalne** (ang. *immutable*). Oznacza to,

że ani określona wartość lokalna, ani dane, które ona wskazuje, nie mogą zostać zmienione za pomocą zewnętrznych modyfikacji. Takie dane nazywane są zwykle **wartościami niemodyfikowalnymi**. Na przykład wszystkie podstawowe typy z platformy .NET (m.in. liczby całkowite, łańcuchy znaków i wartości typu `System.DateTime`) są niemodyfikowalne. Ponadto w bibliotece języka F# zdefiniowane są różne niemodyfikowalne struktury danych, na przykład `Set` i `Map`, oparte na drzewach binarnych. Pakiet `System.Collections.Immutable` z repozytorium NuGet obejmuje definicje znacznie większej liczby niemodyfikowalnych typów.

Wartości niemodyfikowalne mają wiele zalet. Początkowo definiowanie wartości, których nie da się zmienić, może wydawać się dziwne. Jednak wiedza o tym, że wartość jest niemodyfikowalna, sprawia, iż rzadko trzeba zastanawiać się nad **tożsamością obiektu** (ang. *object identity*). Niemodyfikowalne wartości można przekazywać do funkcji i mieć przy tym pewność, że nie zostaną zmienione. Ponadto można przekazywać je między różnymi wątkami bez obaw o niebezpieczny współbieżny dostęp do nich (to zagadnienie opisaliśmy w rozdziale 11.).

Omówienie typów

F# to język z kontrolą typów. Dlatego uzasadnione jest pytanie o **typ** funkcji `wordCount`. Narzędzie F# Interactive już go wyświetliło:

```
val wordCount : text:string -> int * int
```

Wynika z tego, że funkcja `wordCount` przyjmuje jeden argument typu `string` i zwraca wartości typu `int * int`, co w F# oznacza parę liczb całkowitych. Symbol `->` informuje, że `wordCount` to **funkcja**. W programie nie podano jawnie typu argumentu `text`, ponieważ typ funkcji `wordCount` jest określany na podstawie definicji w wyniku **inferencji**. Inferencję typów opisaliśmy w ramce „Co to jest inferencja typów?”, a także szczegółowo w dalszych rozdziałach.

Typy w F# są istotne z różnych powodów — od wydajności, przez produktywność pisania kodu, po współdziałanie z innymi technologiami. Typy pomagają strukturyzować biblioteki, radzić sobie ze złożonością interfejsów API i dodawać ograniczenia w kodzie, które zagwarantują, że będzie on prawidłowy i będzie działał wydajnie. System typów w F# (inaczej niż w wielu innych językach z kontrolą typów) jest jednocześnie prosty i daje duże możliwości. Pod wieloma względami jest też piękny. Jest tak, ponieważ stosowane są w nim jednolite i umożliwiające łączenie konstrukty, na przykład krotki i funkcje, pozwalające tworzyć zwięzłe i opisowe typy. Ponadto inferencja typów sprawia, że prawie nigdy nie trzeba zapisywać typów w programie, choć czasem ich podawanie jest przydatne.

W tabeli 2.1 wymieniliśmy niektóre spośród najważniejszych sposobów tworzenia typów. Są to dostępne w F# techniki definiowania nowych typów. W rozdziałach 3. i 4. szczegółowo opisaliśmy wszystkie te techniki.

Niektóre konstruktory typów, na przykład `list` i `option`, są **generyczne**. Oznacza to, że można ich używać do budowania grup typów, tworząc zmienne generyczne (na przykład typów `int list`, `string list`, `int list list` itd.). Do tworzenia typów generycznych można stosować zarówno notację przedrostkową (`int list`), jak i notację przyrostkową (`list<int>`). Zwyczajowo notację przedrostkową (`int list`) stosuje się tylko dla typów z rodzin `list` i `option`. Zmienne określające typ, na przykład `'a` i `'T`, to miejsca, które można zastąpić dowolnym typem. Typy generyczne i zmienne określające typ opisaliśmy szczegółowo w rozdziałach 3. i 5.

Tabela 2.1. *Wybrane ważne typy i konstruktory oraz odpowiadające im wartości (int to typ reprezentujący liczby całkowite)*

Rodzina typów	Przykłady	Opis
typ option	int option, option<int>	Wartość danego typu lub wartość specjalna None. Np.: Some 3, Some "3", None.
typ list	int list, list<int>	Niemodyfikowalna lista powiązana wartości danego typu. Wszystkie elementy listy muszą być tego samego typu. Np.: [], [3;2;1].
typ1 -> *typ2*	int -> string	Typ funkcji reprezentujący wartość w postaci funkcji, która przyjmuje wartości pierwszego typu i oblicza wyniki drugiego typu. Np.: (fun x -> x+1).
typ1 * ... * *typN*	int * string	Typ krotki — pary, trójki lub większej liczby typów. Np.: (1,"3"), (3,2,1).
typ[]	int[]	Typ tablicowy do tworzenia „płaskich” i modyfikowalnych kolekcji o stałej wielkości.
unit	unit	Typ obejmujący jedną wartość (), podobny do wartości void z wielu języków imperatywnych.
'T	'T, 'a, 'Klucz, 'Wartość	Zmienna określająca typ stosowana w kodzie generycznym.

Co to jest inferencja typów?

Inferencja typów polega na analizowaniu kodu w celu ustalenia ograniczeń na podstawie sposobu używania różnych nazw. Ograniczenia są ustalane dla każdego pliku (w uruchamianym w wierszu poleceń kompilatorze języka F#) i dla każdej wprowadzonej porcji kodu (w narzędziu F# Interactive). Te ograniczenia muszą być spójne. Gwarantuje to, że w programie typy są odpowiednio stosowane. Jeśli tak nie jest, zgłaszany jest błąd typów. Ograniczenia są ustalane od góry do dołu, od lewej do prawej i od zewnątrz do wewnątrz. Jest to istotne, ponieważ w niektórych sytuacjach kolejność inferencji wpływa na jej wynik.

Inferencja typów skutkuje **automatycznym uogólnieniem** kodu, dzięki czemu można go ponownie wykorzystać i jest w oczywisty sposób generyczny. Stosowane są odpowiednie typy generyczne, a programista nie musi ich jawnie zapisywać. Automatyczne uogólnienie to ważny czynnik pozwalający pisać zwięzły i jednocześnie możliwy do ponownego wykorzystania kod z kontrolą typów. Więcej o automatycznym uogólnianiu dowiesz się z rozdziału 5.

Wywoływanie funkcji

Funkcje są istotą większości kodu pisanego w F#. Nie jest zaskoczeniem, że pierwszą rzeczą, jaką robi funkcja wordCount, jest wywołanie innej funkcji. Tu jest nią splitAtSpaces, czyli pierwsza funkcja zdefiniowana w programie:

```
let wordCount text=
    let words = splitAtSpaces text
```

Przyjrzyjmy się najpierw funkcji `splitAtSpaces` w narzędziu F# Interactive:

```
> splitAtSpaces "witaj, świecie";;

val it : string list = ["witaj,"; "świecie"]
```

Widać tu, że funkcja `splitAtSpaces` dzieli podany tekst na słowa w miejscach występowania spacji. W przykładowym kodzie występują też:

- literały znakowe, na przykład `' '` i `'a'`;
- literały łańcuchowe, na przykład `"witaj, świecie"`;
- listy literałów łańcuchowych, na przykład zwrócona wartość `["witaj,"; "świecie"]`.

Literały i listy omówiliśmy szczegółowo w rozdziale 3. Listy to ważna struktura danych w F#. W tej książce znajdziesz wiele przykładów ich zastosowań.

Prosta składnia

Kompilator języka F# i narzędzie F# Interactive uwzględniają wcięcia w kodzie w F#, aby określić, gdzie konstrukty zaczynają się i kończą. Reguły stosowania wcięć są bardzo intuicyjne. Opisaliśmy je w dodatku, który stanowi przewodnik po składni języka F#. Na listingu 2.2 pokazaliśmy wersję funkcji `wordCount`, w której za pomocą słowa kluczowego `in` jawnie określany jest zasięg wszystkich nazw.

Listing 2.2. *Wersja funkcji wordCount, w której jawnie używane są słowa kluczowe „in”*

```
/// Analizowanie łańcucha znaków pod kątem powtarzających się słów
let wordCount text =
    let words = splitAtSpaces text in
    let distinctWords = List.distinct words in
    let numWords = words.Length in
    let numDups = numWords - distinctWords.Length in
    (numWords, numDups)
```

Dwa średniki (`;;`) są konieczne do kończenia kodu wprowadzanego w F# Interactive. Jeśli używasz środowiska IDE, na przykład Visual Studio, zwykle średniki są dodawane automatycznie po zaznaczeniu i uruchomieniu kodu. W tej książce podwójne średniki są pokazywane w interaktywnie wykonywanych fragmentach kodu, ale już nie w większych przykładach.

Czasem wygodne jest zapisywanie definicji `let` w jednym wierszu. W tym celu należy oddzielić wyrażenie od samej definicji słowem kluczowym `in`. Oto przykład:

```
let powerOfFour n =
    let nSquared = n * n in nSquared * nSquared
```

A to przykład zastosowania tej funkcji:

```
> powerOfFour 3;;

val it : int = 81
```

Zapis `let` *wzo* `=` *wyr1* `in` *wyr2* to podstawowy konstrukt języka. Człon *wzo* oznacza wzorzec, a *wyr1* i *wyr2* to wyrażenia. Kompilator języka F# dodaje słowo kluczowe `in`, jeśli *wyr2* jest wyrównane w pionie względem słowa kluczowego `let` z następnego wiersza.

■ **Wskazówka** Zalecamy stosowanie w kodzie pisanym w języku F# wcięć w postaci czterech spacji. Używanie tabulacji jest niedozwolone. Narzędzia obsługujące język F# zgłaszają błąd po napotkaniu tabulacji. Większość edytorów języka F# automatycznie przekształca wystąpienia znaku tabulacji na spacje.

Zasięg

Wartości lokalne, na przykład `words` i `distinctWords`, są niedostępne poza ich **zasięgiem** (ang. *scope*). Gdy wartości są definiowane za pomocą słowa kluczowego `let`, ich zasięg to całe wyrażenie podane po definicji, ale z wyłączeniem samej definicji. Poniżej znajdziesz dwie przykładowe błędne definicje, które próbują uzyskać dostęp do wartości poza ich zasięgiem. Widać tu, że definicje `let` są przetwarzane sekwencyjnie od góry w dół. Pomaga to zagwarantować, że programy są poprawnie zbudowane i pozbawione wielu błędów związanych z niezainicjowanymi wartościami.

```
let badDefinition1 =
    let words = splitAtSpaces text
    let text = "We three kings"
    words.Length
```

Oto efekt uruchomienia kodu:

```
error FS0039: The value or constructor 'text' is not defined
```

A to następny przykład:

```
let badDefinition2 = badDefinition2 + 1
```

I efekt jego wykonania:

```
error FS0039: The value or constructor 'badDefinition2' is not defined
```

W definicjach funkcji można usunąć wartość z zasięgu, deklarując inną wartość o tej samej nazwie. Na przykład poniższa funkcja oblicza wartość `(n*n*n*n)+2`:

```
let powerOfFourPlusTwo n =
    let n = n * n
    let n = n * n
    let n = n + 2
    n
```

Ten kod to odpowiednik następującego zapisu:

```
let powerOfFourPlusTwo n =
    let n1 = n * n
    let n2 = n1 * n1
    let n3 = n2 + 2
    n3
```

Usunięcie wartości z zasięgu nie zmienia pierwotnej wartości. Sprawia jedynie, że nazwa danej wartości nie jest dostępna w bieżącym zasięgu. Nie należy nadużywać tej techniki.

Ponieważ wiązania tworzone za pomocą instrukcji `let` to odmiana wyrażenia, takie polecenia można zagnieżdżać. Oto przykład:

```
let powerOfFourPlusTwoTimesSix n =
    let n3 =
```

```
        let n1 = n * n
        let n2 = n1 * n1
        n2 + 2
    let n4 = n3 * 6
    n4
```

Tu `n1` i `n2` to wartości zdefiniowane lokalnie za pomocą instrukcji `let` w wyrażeniu definiującym wartość `n3`. Tych wartości lokalnych nie można stosować poza ich zasięgiem. Na przykład poniższy kod spowoduje zgłoszenie błędu:

```
let invalidFunction n =
    let n3 =
        let n1Inner = n + n
        let n2Inner = n1Inner * n1Inner
        n1Inner * n2Inner
    let n4 = n1Inner + n2Inner + n3 // Błąd! Wartość n3 znajduje się w zasięgu, ale
                                    // pozostałe wartości są poza zasięgiem!
    n4
```

Zasięg lokalny jest wykorzystywany w F# do wielu celów — zwłaszcza do ukrywania szczegółów implementacji, których nie chcesz ujawniać poza funkcjami lub obiektami. Zagadnienie to opisaliśmy szczegółowo w rozdziale 7.

Używanie struktur danych

Oto następny fragment kodu:

```
let wordCount text =
    let words = splitAtSpaces text
    let distinctWords = List.distinct words
    ...
```

Ten fragment pokazuje, jak struktury danych są używane w kodzie napisanym w języku F#. Ostatni wiersz ilustruje istotę obliczeń wykonywanych przez funkcję `wordCount`. Wykorzystano tu funkcję `List.distinct` z biblioteki języka F#, aby przekształcić podane słowa na listę z niepowtarzającymi się wyrazami. Wyniki działania tej funkcji można zobaczyć za pomocą narzędzia F# Interactive:

```
> List.distinct ["b"; "a"; "b"; "b"; "c" ];;

val it : string list = [ "b"; "a"; "c" ]

> List.distinct (List.distinct ["abc"; "ABC"]);;

val it : string list = [ "abc"; "ABC" ]
```

Warto zwrócić tu uwagę na kilka rzeczy:

- Narzędzie F# Interactive wyświetla zawartość ustrukturyzowanych wartości (na przykład list).
- Powtarzające się elementy są w wyniku konwersji usuwane.
- Elementy z listy zachowują kolejność, ponieważ funkcja `List.distinct` nie zmienia kolejności niepowtarzających się wartości.
- W trakcie wyznaczania kolejności i sprawdzania równości w łańcuchach znaków domyślnie uwzględniana jest wielkość liter.

Nazwa `List` oznacza moduł języka F# `FSharp.Collections.List` z podstawowej biblioteki tego języka. Moduł ten obejmuje operacje powiązane z wartościami typu `list`. Dla typów często tworzone są odrębne moduły zawierające operacje powiązane z danym typem. Wszystkie moduły z należących do przestrzeni nazw `FSharp` przestrzeni nazw `Core`, `Collections`, `Text` i `Control` można podawać w postaci jednowyrazowych przedrostków (na przykład `List.distinct`). Inne moduły z wymienionych przestrzeni nazw to m.in. `Set`, `Option`, `Map` i `Array`. Funkcję `List.distinct` dodano w wersji F# 4.0. We wcześniejszych wersjach należy stosować inne rozwiązania, na przykład `Set.ofList`.

Używanie właściwości i notacji z kropką

Dwa następne wiersze funkcji `wordCount` wyznaczają szukany wynik — liczbę powtarzających się słów. Używana jest do tego właściwość `Length` przetwarzanej wartości:

```
let numWords = words.Length
let numDups = numWords - distinctWords.Length
```

F# analizuje nazwy właściwości na etapie kompilacji (lub interaktywnie, jeśli używasz narzędzia F# Interactive; wtedy rozróżnienie na czas kompilacji i czas wykonywania programu nie obowiązuje). W tym celu wykorzystuje znane w czasie kompilacji informacje o typie wyrażenia podanym po lewej stronie kropki. Tu są to wyrażenia `words` i `distinctWords`. Niekiedy w kodzie trzeba podać typ, aby dookreślić potencjalnie wieloznaczne nazwy właściwości. Na przykład w poniższym kodzie podano typ, aby określić, że nazwa `inp` oznacza listę. Dzięki temu system typów języka F# może wywnioskować, że nazwa `Length` oznacza właściwość powiązaną z wartościami typu `list`:

```
let length (inp:'T list) = inp.Length
```

Tu `'T` oznacza, że funkcja `length` jest generyczna. Sprawia to, że można ją stosować do list dowolnego typu. Generyczny kod opisaliśmy szczegółowo w rozdziałach 3. i 5.

Zastosowanie notacji z kropką wskazuje na to, że F# jest językiem jednocześnie funkcyjnym i obiektowym. Właściwości są rodzajem **składowych** (jest to ogólna nazwa określająca dowolne cechy powiązane z typem lub wartością). Składowe wskazywane za pomocą przedrostka w postaci nazwy typu to **składowe statyczne**. Składowe powiązane z konkretną wartością danego typu to **składowe instancji**. Dostęp do składowych instancji można uzyskać, podając obiekt po lewej stronie kropki. Rozróżnienie na wartości, właściwości i metody znajdziesz dalej w tym rozdziale, a szczegółowe omówienie składowych zawiera rozdział 6.

■ **Uwaga** Adnotacje określające typ mogą okazać się przydatną dokumentacją. Zwykle należy je dodawać w miejscu deklaracji zmiennej.

Czasem jawnie nazwane funkcje działają jak składowe. Wcześniej przedstawiony kod można zapisać też tak:

```
let numWords = List.length words
let numDups = numWords - List.length wordSet
```

W kodzie napisanym w języku F# zetkniesz się z oboma stylami. W niektórych bibliotekach języka F# składowe są używane rzadko lub w ogóle się ich nie stosuje. Jednak rozsądne korzystanie ze składowych i właściwości pozwala znacznie ograniczyć konieczność tworzenia trywialnych funkcji `get` i `set` w bibliotekach, istotnie poprawić czytelność kodu klienckiego, a także umożliwić programistom, którzy stosują środowiska takie jak Visual Studio, łatwe i intuicyjne przeglądanie podstawowych funkcji rozwijanych bibliotek.

Jeśli kod nie zawiera adnotacji określających typ, które pozwalają zinterpretować notację z kropką, wystąpi błąd taki jak pokazany poniżej:

```
> let length inp = inp.Length;;

error FS0072: Lookup on object of indeterminate type based on information prior to this
program point. A type annotation may be needed prior to this program point to constrain
the type of the object. This may allow the lookup to be resolved. You can resolve this
by adding a type annotation as shown earlier.
```

Używanie krotek

Ostatni fragment funkcji `wordCount` zwraca **krotkę** składającą się z liczby słów i liczby powtarzających się słów:

```
...
(numWords, numDups)
```

Krotki to najprostsza i jednocześnie bodaj najbardziej przydatna ze wszystkich struktur danych języka F#. Wyrażenie tworzące krotkę to zestaw połączonych wyrażeń dających nowe wyrażenie:

```
let site1 = ("www.cnn.com", 10)
let site2 = ("news.bbc.com", 5)
let site3 = ("www.msnbc.com", 4)
let sites = (site1, site2, site3)
```

Oto typy określone w wyniku inferencji i obliczone wartości:

```
val site1 : string * int = ("www.cnn.com", 10)
val site2 : string * int = ("news.bbc.com", 5)
val site3 : string * int = ("www.msnbc.com", 4)
val sites : (string * int) * (string * int) * (string * int) =
  (("www.cnn.com", 10), ("news.bbc.com", 5), ("www.msnbc.com", 4))
```

Krotki można rozbić na tworzące je komponenty na dwa sposoby. W przypadku par (czyli krotek dwuelementowych) można jawnie wywołać funkcje `fst` (od ang. *first*, czyli „pierwszy”) i `snd` (od ang. *second*, czyli „drugi”). Jak wskazują nazwy, funkcje te pobierają pierwszy i drugi element pary:

```
> fst site1;;
val it : string = "www.cnn.com"

> let relevance = snd site1;;
val relevance : int = 10
```

Funkcje `fst` i `snd` są zdefiniowane w bibliotece języka F# i zawsze można z nich korzystać w programach w tym języku. Oto proste definicje tych funkcji:

```
let fst (a, _) = a
let snd (_, b) = b
```

Częściej jednak krotki są rozbijane za pomocą **wzorców**, tak jak w poniższym kodzie:

```
let url, relevance = site1
let siteA, siteB, siteC = sites
```

Tu nazwy podane po lewej stronie definicji są wiązane z odpowiednimi elementami podanej po prawej stronie wartości w postaci krotki. Tak więc `url` przyjmuje wartość `"www.cnn.com"`, a `relevance` — wartość 10.

Wartości w postaci krotek mają określony typ. Istnieje dowolna liczba rodzin typów krotek — jedna dla par (przechowujących dwie wartości), jedna dla trójek (przechowujących trzy wartości) itd. Oznacza to, że jeśli spróbujesz zastosować trójkę w miejscu, gdzie oczekiwana jest para, przed uruchomieniem kodu zgłoszony zostanie błąd związany z typami:

```
> let a, b = (1, 2, 3);;

error FS0001: Type mismatch. Expecting a
    'a * 'b
but given a
    'a * 'b * 'c
The tuples have differing lengths of 2 and 3
```

Krotki często są używane do zwracania wielu wartości przez funkcje (tak jak wcześniej w przykładowej funkcji `wordCount`). Nieraz służą też do przekazywania wielu argumentów do funkcji. Często krotka zwracana przez jedną funkcję staje się krotką przekazywaną do innej funkcji. W poniższym przykładowym kodzie została pokazana inna wersja zdefiniowanej i stosowanej wcześniej funkcji `showWordCount`:

```
let showResults (numWords, numDups) =
    printfn "--> Liczba słów w tekście: %d" numWords
    printfn "--> Liczba powtórzeń: %d" numDups

let showWordCount text = showResults (wordCount text)
```

Funkcja `showResults` przyjmuje parę wartości rozbijaną na elementy `numWords` i `numDups`. Para danych wyjściowych z funkcji `wordCount` staje się parą danych wejściowych funkcji `showResults`.

Wartości i obiekty

W F# wszystko jest **wartością**. W niektórych innych językach wszystko jest **obiektem**. W praktyce słowa te możesz często stosować wymiennie, choć programiści języka F# zwykle rezerwują określenie „obiekt" dla wartości specjalnego rodzaju:

- wartości, których obserwowalne właściwości zmieniają się w trakcie wykonywania programu (zwykle w wyniku jawnej modyfikacji przechowywanych w pamięci danych lub za pomocą zewnętrznych zmian stanu);
- wartości, które wskazują dane lub stany określające tożsamość (takie jak unikatowe identyfikatory całkowitoliczbowe) albo wyznaczają ogólną tożsamość obiektu, służącą do odróżniania obiektów od innych wartości identycznych w pozostałych aspektach;
- wartości, które można sprawdzić, by ustalić dodatkowe mechanizmy za pomocą rzutowania, konwersji i interfejsów.

F# obsługuje obiekty, ale nie wszystkie wartości są uznawane za obiekty. Programowanie w F# nie jest zorientowane obiektowo. Język ten umożliwia jednak programowanie obiektowe i pozwala korzystać z obiektów tam, gdzie są najbardziej przydatne. W rozdziale 4. omówiliśmy szczegółowo tożsamość i modyfikowanie wartości.

Używanie kodu imperatywnego

Zdefiniowane w poprzednim punkcie funkcje `showWordCount` i `showResults` wyświetlają wyniki za pomocą funkcji bibliotecznej `printfn`:

```
printfn "--> Liczba słów w tekście: %d" numWords
printfn "--> Liczba powtórzeń: %d" numDups
```

Jeśli znasz język OCaml, C lub C++, funkcja `printfn` będzie wyglądała znajomo (jak odmiana funkcji `printf`). Funkcja `printfn` dodaje też znak nowego wiersza na końcu wyświetlanego tekstu. Tu wzorzec `%d` to miejsce przeznaczone na liczbę całkowitą, a pozostała część tekstu jest wyświetlana w dosłownej postaci w konsoli.

F# obsługuje też powiązane funkcje, na przykład `printf`, `sprintf` i `fprintf` (ich omówienie znajdziesz w rozdziale 4.). Rodzina funkcji `printf` (inaczej niż w C i C++) formatuje tekst z zachowaniem bezpieczeństwa ze względu na typ. Kompilator języka F# sprawdza, czy kolejne argumenty są zgodne z wymogami poszczególnych miejsc na dane.

Przyrostek `n` w nazwie `printfn` oznacza, że po danych wyjściowych rozpoczynany jest nowy wiersz. W F# tekst można formatować także w inny sposób. Możesz na przykład bezpośrednio wykorzystać biblioteki platformy .NET:

```
System.Console.WriteLine("--> Liczba słów w tekście: {0}", numWords)
System.Console.WriteLine("--> Liczba powtórzeń: {0}", numDups)
```

Sekwencja `{0}` wyznacza miejsce na dane, przy czym kompilator nie sprawdza tu, czy argumenty pasują do tego miejsca. Za pomocą funkcji `printfn` można też pokazać, jak stosować sekwencyjnie uruchamiane wyrażenia do wywoływania efektów w świecie zewnętrznym.

Podobnie jak w wyrażeniach `let...in...` czasem wygodnie jest zapisywać sekwencyjnie wykonywany kod w jednym wierszu. W tym celu dwa wyrażenia należy rozdzielić średnikiem (`;`). Pierwsze podwyrażenie jest przetwarzane (zwykle w celu wywołania efektów ubocznych), jego wynik zostaje pominięty, a wartością całego wyrażenia zostaje wynik drugiego podwyrażenia. Oto prostszy przykład zastosowania tego konstruktu:

```
let two = (printfn "Witaj, świecie"; 1 + 1)
let four = two + two
```

Ten kod po uruchomieniu wyświetli napis `Witaj, świecie` tylko jeden raz — gdy uruchomiona zostanie definicja wartości `two`. W F# nie występują typowe instrukcje. Fragment `(printfn "Witaj, świecie"; 1 + 1)` jest wyrażeniem, jednak w trakcie jego przetwarzania pierwsza część powoduje efekt uboczny, po czym wynik tej części jest pomijany. Często wygodnie jest stosować nawiasy do wyodrębniania sekwencyjnie wykonywanego kodu. Kod ze skryptu można (w teorii) umieścić w nawiasie i dodać średnik. Dzięki temu proste konstrukty stosowane w kodzie są lepiej widoczne:

```
(printfn "--> Liczba słów w tekście: %d" numWords;
 printfn "--> Liczba powtórzeń: %d" numDups)
```

Uwaga Symbol `;` służy do zapisywania sekwencyjnie wykonywanego kodu w wyrażeniach. Znaki `;;` są używane do kończenia interakcji w sesji w narzędziu F# Interactive. Średniki są opcjonalne, gdy poszczególne fragmenty sekwencyjnie wykonywanego kodu są umieszczane w odrębnych wierszach rozpoczynających się z tym samym wcięciem.

Używanie bibliotek obiektowych w F#

Wartość F# wynika nie tylko z tego, co możesz zrobić w samym języku, ale też z komponentów spoza języka, z których można korzystać. Na przykład sam F# nie obejmuje biblioteki do tworzenia graficznego interfejsu użytkownika. Zamiast tego jest powiązany z platformą .NET, a poprzez nią może korzystać z większości znaczących technologii programistycznych dostępnych w popularnych platformach informatycznych. Zetknąłeś się już z jednym przypadkiem zastosowania bibliotek platformy .NET — w pierwszej ze zdefiniowanych wcześniej funkcji:

```
/// Podział łańcucha znaków na słowa w miejscach występowania spacji
let splitAtSpaces (text: string) =
    text.Split ' '
    |> Array.toList
```

Tu `text.Split` to wywołanie metody instancji `Split` z biblioteki platformy .NET. Metoda ta jest zdefiniowana dla wszystkich obiektów reprezentujących łańcuchy znaków.

Aby lepiej przedstawić tę technikę, w drugim przykładzie wykorzystamy dwie rozbudowane biblioteki dostępne w platformie .NET — `System.IO` i `System.Net`. Kompletny przykład pokazany na listingu 2.3 to skrypt, który można uruchomić w narzędziu F# Interactive.

Listing 2.3. *Używanie w F# bibliotek platformy .NET związanych z sieciami*

```
open System.IO
open System.Net

/// Pobieranie zawartości adresu URL za pomocą żądania sieciowego
let http (url: string) =
    let req = WebRequest.Create(url)
    let resp = req.GetResponse()
    let stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    let html = reader.ReadToEnd()
    resp.Close()
    html

http "http://news.bbc.co.uk"
```

W tym przykładzie zostało wykorzystanych kilka ważnych bibliotek platformy .NET. Pomaga to zapoznać się z ciekawymi konstruktami języka F#. W następnych punktach opisaliśmy pokazany listing.

Dostęp do przestrzeni nazw i modułów za pomocą wywołania open

Pierwszy fragment widoczny w tym kodzie to zastosowanie wywołania `open` do uzyskania dostępu do funkcji z przestrzeni nazw `System.IO` i `System.Net`:

```
open System.IO
open System.Net
```

Przestrzenie nazw omówiliśmy szczegółowo w rozdziale 7. Pokazane tu deklaracje sprawiają, że można korzystać z dowolnych komponentów dostępnych w wymienionych przestrzeniach nazw bez podawania kompletnych ścieżek. Jeśli nie zastosujesz wywołania `open`, będziesz musiał posługiwać się podanym poniżej zapisem, którego rozwlekłość jest oczywista:

```
let req = System.Net.WebRequest.Create(url)
```

Za pomocą wywołania open można też korzystać z zawartości modułów języka F# bez podawania kompletnych ścieżek. Dokładne omówienie modułów znajdziesz w rozdziale 7.

Więcej o wywołaniu open

Wywołanie `open` to łatwy sposób na dostęp do zawartości przestrzeni nazw i modułów. Należy jednak zwrócić uwagę na pewne szczegóły. Otóż wywołanie to nie wczytuje ani nie wskazuje biblioteki. Zamiast tego ujawnia funkcje z już wczytanych bibliotek. Aby wczytać bibliotekę, należy wskazać konkretny plik DLL za pomocą składni `#r` (w skrypcie) lub opcji `-r` (w wierszu poleceń).

Biblioteki i przestrzenie nazw są niezależne od siebie. Wiele bibliotek może udostępniać funkcje z tej samej przestrzeni nazw, a każda biblioteka może zawierać funkcje z wielu przestrzeni nazw. Często jedna biblioteka obejmuje większość funkcji z danej przestrzeni nazw. Na przykład większość funkcji z przestrzeni nazw `System.Net` znajduje się w bibliotece `System.Net.dll`, ale niektóre funkcje z tej przestrzeni nazw pochodzą z innych bibliotek systemowych. Aby umieścić kod w przestrzeni nazw, dodaj deklarację `namespace` w górnej części pliku (opisaliśmy to w rozdziale 7.).

Jeśli dwie przestrzenie nazw obejmują typy, podrzędne przestrzenie nazw i (lub) moduły o identycznych nazwach, to po użyciu wywołania `open` dla tych przestrzeni nazw dostęp do ich zawartości można uzyskać za pomocą tych samych skróconych ścieżek. Na przykład przestrzeń nazw `System` obejmuje typ `String`, a przestrzeń `FSharp.Core` zawiera moduł `String`. Wtedy w trakcie wyszukiwania identyfikatora takiego jak `String.map` sprawdzane są wartości i składowe obu komponentów o nazwie `String`. Jeśli pojawia się wieloznaczność, wybierany jest element z przestrzeni nazw, dla której `open` wywołano później.

Jeżeli zdarzy się kolizja nazw, możesz zdefiniować własne skrócone nazwy modułów i typów. Oto przykłady: `module MyString = My.Modules.String` i `type SysString = System.String`. Nie można jednak tworzyć aliasów dla przestrzeni nazw.

Pobieranie stron internetowych

W drugiej połowie listingu 2.3 używana jest biblioteka `System.Net`, by zdefiniować funkcję `http` wczytującą strony internetowe w HTML-u. Aby zbadać działanie tej funkcji, wpisz w narzędziu F# Interactive następujące wiersze:

```
> open System.IO;;
> open System.Net;;
> let req = WebRequest.Create("http://news.bbc.co.uk");;
val req : WebRequest
> let resp = req.GetResponse();;
val resp : WebResponse
> let stream = resp.GetResponseStream();;
val stream : Stream
> let reader = new StreamReader(stream);;
val reader : StreamReader
> let html = reader.ReadToEnd();;
val html : string =
  "<html><head><title>BBC News and Sport</title><meta http-eq"+[959 chars]
```

Pierwszy wiersz tego kodu tworzy obiekt typu `WebRequest`. Używana jest do tego statyczna metoda `Create`, będąca składową typu `System.Net.WebRequest`. Wynik tej operacji to obiekt, który działa jak uchwyt przetwarzanego żądania pobrania strony internetowej. Za pomocą tego uchwytu możesz na przykład anulować żądanie lub sprawdzić, czy zostało już przetworzone. Drugi wiersz wywołuje metodę instancji `GetResponse`. Pozostałe wiersze z przykładu pobierają strumień danych z odpowiedzi na żądanie (służy do tego wywołanie `resp.GetResponseStream()`), tworzą obiekt służący do odczytu tego strumienia (za pomocą wywołania `new StreamReader(stream)`) i wczytują kompletny tekst ze strumienia. Operacje wejścia – wyjścia z platformy .NET omówiliśmy szczegółowo w rozdziale 4. Na razie możesz za pomocą eksperymentów w narzędziu F# Interactive przekonać się, że opisane informacje rzeczywiście pobierają kod HTML strony internetowej.

Wartości, metody i właściwości

Oto różnice między wartościami, metodami i właściwościami:

- **Wartości** to parametry i elementy z najwyższego poziomu zdefiniowane za pomocą wywołania `let` lub dopasowywania do wzorca. Na przykład: `form`, `text`, `wordCount`.
- **Metody** to nazwane operacje powiązane z typami lub wartościami. Mogą być dostępne zarówno dla prostych wartości, jak i dla obiektów. Metody można przeciążać (zobacz rozdział 6.), co sprawia, że to, która metoda zostanie użyta, zależy od typów i liczby argumentów. Na przykład: `System.Net.WebRequest.Create(url)` i `resp.GetResponseStream()`.
- **Właściwości** to powiązane z typami albo wartościami nazwane operacje pobierające lub ustawiające dane. Właściwość to skrót pozwalający wywoływać metody składowe, które służą do pobierania lub ustawiania danych. Na przykład: `System.DateTime.Now` i `form.TopMost`.
- **Właściwości indeksujące** to właściwości, które przyjmują argumenty w postaci indeksu. Właściwości indeksujące o nazwie `Item` są dostępne za pomocą składni `.[_]` (kropka jest wymagana). Na przykład: `vector.[3]` i `matrix.[3,4]`.

Oto ustalony w wyniku inferencji typ funkcji `http` kończącej opisaną sekwencję:

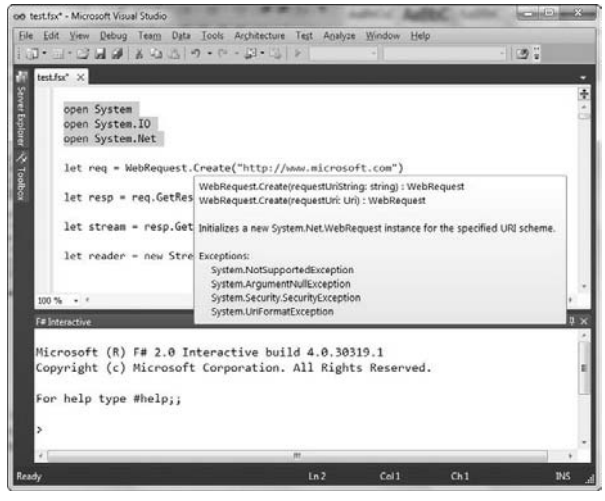
```
val http : url:string -> string
```

Pobieranie i używanie pakietów

W poprzednim podrozdziale pokazaliśmy, jak używać jednej z bibliotek bogatego ich zbioru dostępnego nawet w najprostszej instalacji F#. Język ten umożliwia jednak korzystanie także z wielu innych bibliotek. Na przykład w witrynie *http://nuget.org* dostępne jest duże repozytorium **pakietów**. W następnym przykładzie zastosujesz pakiety `Suave` i `FSharp.Data` języka F#.

Pomoc w formacie XML w środowisku IDE

W rozbudowanych interaktywnych edytorach, takich jak Visual Studio i Xamarin Studio, można łatwo dowiedzieć się więcej o funkcjach dostępnych w bibliotekach. W tym celu należy umieścić kursor na odpowiednich identyfikatorach w kodzie źródłowym. Na przykład jeśli przeniesiesz kursor na słowo `Create` w wywołaniu `WebRequest.Create`, zobaczysz pokazaną poniżej pomoc w formacie XML.



Dwa najczęściej stosowane sposoby pobierania pakietów dla języka F# to używanie programów *nuget.exe* i *paket.exe*. Możesz wykorzystać dowolny z nich.

- Jeśli chcesz używać programu *nuget.exe*, najpierw pobierz go ze strony *http://nuget.org* i zainstaluj. Następnie uruchom poniższe instrukcje w wierszu poleceń:

```
> nuget install Suave –ExcludeVersion -OutputDirectory packages
> nuget install FSharp.Data –ExcludeVersion -OutputDirectory packages
```

- Jeżeli wolisz zastosować program *paket.exe*, najpierw pobierz go ze strony *http://fsprojects.github.io/paket* i zainstaluj. Następnie umieść pokazany poniżej tekst w pliku *paket.dependencies* zapisanym w pustym katalogu (do tworzenia i edycji pliku wykorzystaj dowolny edytor tekstu):

```
source https://nuget.org/api/v2
nuget Suave
nuget FSharp.Data
```

Potem uruchom poniższą instrukcję w katalogu zawierającym plik `paket.dependencies`:

```
> paket.exe install
```

Niezależnie od tego, którą techniką się posłużysz, uzyskasz katalog *packages* zawierający podkatalogi *FSharp.Data* i *Suave*. Są to zainstalowane pakiety, których będziesz używać w dalszych podrozdziałach.

Dostęp do zewnętrznych danych za pomocą pakietów języka F#

Po pobraniu pakietu `FSharp.Data` możesz zastosować go, by uzyskać dostęp do różnorodnych zewnętrznych źródeł danych. W tym przykładzie użyjesz tabeli bezpośrednio z Wikipedii — popularnej encyklopedii internetowej. Możesz wybrać dowolną tabelę z danymi z tej lub innej witryny. Tu używana jest tabela *Table of the Top 100 Most Endangered Species*, która w czasie powstawania tej książki była dostępna pod następującym adresem URL:

```
http://en.wikipedia.org/wiki/The_world's_100_most_threatened_species
```

W razie potrzeby popraw ten adres URL lub zastosuj inny, prowadzący do innej tabeli z danymi, i odpowiednio dostosuj pokazany poniżej kod:

```
#r "packages/FSharp.Data/lib/net40/FSharp.Data.dll"

open FSharp.Data

type Species =
HtmlProvider<"http://en.wikipedia.org/wiki/The_world's_100_most_threatened_species">

let species =
    [ for x in Species.GetSample().Tables.``Species list``.Rows ->
          x.Type, x.``Common name`` ]
```

Są to prawdopodobnie trzy najbardziej magiczne wiersze z tego rozdziału. Ich uruchomienie daje następujący efekt:

```
val species : (string * string) list =
  [("Plant (tree)", "Baishan Fir");
   ("Reptile", "Leaf scaled sea-snake");
   ...
   ("Mammal", "Attenborough's echidna")]
```

Za pomocą zaledwie trzech wierszy zastosowaliśmy technikę **screen scraping** (polegającą na wczytywaniu informacji z ekranu) i pobraliśmy spory zbiór danych. Teraz możesz przetworzyć te dane za pomocą technik programowania funkcyjnego. Więcej na ich temat dowiesz się z rozdziałów 3. i 8.

```
let speciesSorted =
    species
      |> List.countBy fst
      |> List.sortByDescending snd
```

Oto wynik uruchomienia tego kodu:

```
val speciesSorted : (string * int) list =
  [("Plant", 13); ("Bird", 11); ("Fish", 10); ("Plant (tree)", 8);
   ("Amphibian (frog)", 7); ("Mammal (primate)", 6); ("Mammal", 5);
   ...
   ("Fish (shark)", 1)]
```

Ta lista wyświetla 100 podzielonych na kategorie gatunków zwierząt najbardziej zagrożonych wyginięciem.

Uruchamianie serwera WWW i udostępnianie danych za pomocą pakietów języka F#

Jeden z pakietów pobranych w poprzednim podrozdziale to Suave. Jest to prosta platforma służąca do tworzenia serwerów WWW w języku F#. Więcej na jej temat dowiesz się ze strony *http://suave.io*. Serwery WWW Suave są bardzo wydajne i skalowalne, co pozwala na równoległą obsługę wielu żądań sieciowych. Serwery te działają w modelu nieblokującym i wykorzystują mechanizmy języka F#, które poznasz w rozdziale 11.

W pokazanym poniżej kodzie uruchamiany jest lokalny serwer WWW, który udostępnia tekst w formacie HTML wygenerowany na podstawie danych pobranych w poprzednim przykładzie. Więcej o programowaniu sieciowym i pisaniu stron internetowych dowiesz się z rozdziału 14. Przyjrzyj się poniższemu kodowi:

```
#r "packages/Suave/lib/net40/Suave.dll"

open Suave
open Suave.Web

let html =
    [ yield "<html><body><ul>"
      for (category,count) in speciesSorted do
         yield sprintf "<li>Kategoria <b>%s</b>: <b>%d</b></li>" category count
      yield "</ul></body></html>" ]
    |> String.concat "\n"

startWebServer defaultConfig (Successful.OK html)
```

Ten serwer WWW działa lokalnie i jest dostępny tylko na maszynie lokalnej. Z rozdziału 14. dowiesz się, jak umożliwić dostęp do serwera z poziomu innych maszyn. Po uruchomieniu tego kodu zobaczysz informacje podobne do poniższych:

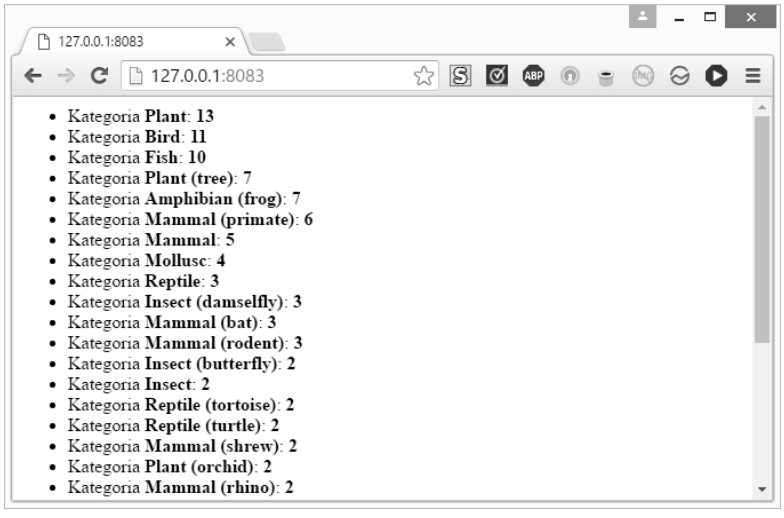
```
[I] 2015-02-12T20:15:40.6980929Z: listener started in 14.001 ms with binding
localhost:8083
```

Jeśli na tym samym komputerze uruchomisz przeglądarkę internetową, możesz wpisać w jej pasku adresu adres URL *http://localhost:8083*. Jeżeli serwer WWW działa poprawnie, zobaczysz wyświetlone informacje z rysunku 2.1.

Więcej o programowaniu rozwiązań dla sieci dowiesz się z rozdziału 14. Zobaczysz przede wszystkim, jak używać opartej na F# platformy WebSharper do pisania za pomocą F# kodu w HTML-u i JavaScripcie oraz kodu uruchamianego po stronie serwera. Niewielkim dodatkowym nakładem pracy możesz uatrakcyjnić wyświetlane treści za pomocą popularnej platformy Angular, służącej do tworzenia rozwiązań działających w sieci. Jeśli masz praktyczną wiedzę z zakresu HTML-a, rozpoznasz wiele użytych tu elementów związanych z wyświetlaniem stron:

```
let angularHeader = """<head>
<link rel="stylesheet"
href="http://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.2.26/angular.min.js"> </script>
</head>"""

let fancyText =
    [ yield """<html>"""
```



Rysunek 2.1. *Używanie przeglądarki do wyświetlania witryny utworzonej za pomocą Suave*

```
    yield angularHeader
    yield """ <body>"""
    yield """  <table class="table table-striped">"""
    yield """   <thead><tr><th>Kategoria</th><th>Liczba</th></tr></thead>"""
    yield """   <tbody>"""
    for (category,count) in speciesSorted do
       yield sprintf "<tr><td>%s</td><td>%d</td></tr>" category count
    yield """   </tbody>"""
    yield """  </table>"""
    yield """ </body>"""
    yield """</html>""" ]
  |> String.concat "\n"

startWebServer defaultConfig (Successful.OK fancyText)
```

Możliwe, że na tym etapie będziesz musiał ponownie uruchomić narzędzie F# Interactive. Nowa zawartość strony jest widoczna na rysunku 2.2.

Język F# możesz stosować także do wykonywania wielu innych zadań oprócz udostępniania stron internetowych. Przekonasz się o tym w trakcie lektury tej książki.

Podsumowanie

W tym rozdziale przedstawiliśmy proste programowanie interaktywne z użyciem języka F# i platformy .NET. Zobaczyłeś tu wiele konstruktów stosowanych w codziennej pracy z tym językiem. Nauczyłeś się pobierać pakiety używane w F# i wstępnie zapoznałeś się z tym, jak używać rozbudowanej biblioteki języka F# — `FSharp.Data`. Utworzyłeś też i uruchomiłeś serwer WWW, który udostępnia informacje pobrane z Wikipedii. Językiem używanym do wyświetlania danych był HTML.

W następnym rozdziale bliżej przyjrzysz się opisanym tutaj i nowym konstruktom służącym do opartego na kompozycji komponentów i zwięzłego programowania funkcyjnego w F#.

Rysunek 2.2. *Witryna z lepszym formatowaniem utworzona za pomocą biblioteki Suave*

■ **Uwaga** W rozdziale 3. używane są niektóre funkcje zdefiniowane w niniejszym rozdziale. Jeśli korzystasz z narzędzia F# Interactive, wygodne będzie pozostawienie otwartej sesji w trakcie lektury.

Skorowidz

A

abstrakcje, 83
abstrakcyjne ujmowanie sterowania, 60
adnotacje
  dotyczące dostępności, 168, 169
  określające typ, 120
ADO.NET, 349
  nawiązywanie połączeń, 349
  tworzenie baz danych, 350
  tworzenie tabel, 350
  używanie procedur składowanych, 352
agenty, 296
agregacje, 343
akcje, 62
aksjomat, 313
akumulowanie, 63, 229
algebra liniowa, 266
algorytm centroidów, 264
algorytmy generyczne, 112
aliasy, 69
analizowanie łańcucha znaków, 21
anulowanie operacji, 295
aplikacja wizualizująca fraktale, 424
aplikacje
  graficzne, 409
  ograniczone obliczeniami, 290
  ograniczone operacjami wejścia – wyjścia, 290
  sieciowe, 355
  typu klient – serwer, 378, 382
  współbieżne, 516
argumenty
  opcjonalne, 135
  z nazwami, 135
ASCII, 184, 185
asercje, 511
asynchroniczne operacje, 293
  platformy .NET, 294
  przetwarzanie, 290
  zadanie, 358
asynchroniczny agent, 301
atrybut ReflectedDefinition, 473
atrybuty, 541
  do sterowania, 512
automatyczne
  rzutowanie w górę, 118
  śledzenie zasobów, 403
  uogólnienie kodu, 25, 61
  właściwości, 141
automatycznie otwierane moduły, 174

B

bazy danych, 347
BDD, Binary Decision Diagrams, 311
bezpieczeństwo danych, 346
biblioteka
  Deedle, 271
  Eto, 407
  Eto Forms, 416
  Extreme Optimization, 267
  FCore, 267
  FSharp.Charting, 254
  FSharp.Core, 480, 481
  FSharp.Data, 480
  Math.NET, 266–270
  Math.NET Numerics, 267
  NMath, 267
  System.Net, 34
  WebSharper.Warp, 378
biblioteki, 477, 478
  .NET, 108
  JavaScriptu, 404
  obiektowe, 33
  platformy .NET, 481
  związane z refleksją, 484
binarne diagramy decyzyjne, 320
bit, 312
blokada, 307
  typu ReaderWriterLock, 308
blokowanie
  haszowania, 242
  porównywania dla typu, 242
  sprawdzania równości, 242
bramka, 312
budowa
  aplikacji graficznej, 409
  kontrolki, 434
budowanie
  bibliotek, 504
  łańcuchów znaków, 183

C

cechy układów, 319
chronienie wewnętrznego stanu, 169
CLR, Common Language Runtime, 477
cytowania, 470, 471, 549
czas życia zasobu, 152, 154
częściowe
  implementacje obiektów, 147
  zaimplementowane typy, 148

D

dane
  binarne, 211
  liczbowe, 253
  zewnętrzne, 337
debugowanie, 513
  aplikacji współbieżnych, 516
  kodu w języka C, 515
  kodu w języku F#, 514
definicje lokalne, 166
definiowanie
  generatora wyrażeń, 457
  interfejsowych typów obiektowych, 142
  palety kolorów, 423
  typów, 97, 546
  typów obiektowych, 138
  typów rekordowych, 98
  typów wyjątków, 80
  unii z dyskryminatorami, 100
  wielu typów, 102
deklarowanie właściwości automatycznych, 141
dekodowanie
  danych binarnych, 211
  łańcuchów znaków Unicode, 210
delegaty, 161
delegowanie, 148, 149
diagnostyka, 510
diagram BDD, 323
DLL, Dynamic Link Library, 175
dodawanie
  jednostek do algorytmów liczbowych, 274
  jednostek do definicji typów, 277
  operatorów przeciążonych, 134
  reaktywności, 370
dokumentacja, 23, 504
domknięcie, closure, 60
domyślne odwzorowania typów, 496
dopasowywanie, 542
  do wzorca, 50, 117
  ustrukturyzowanych wartości, 51
  wartości typów, 236
dostawca typów, 337, 340, 474, 486
  dla wyrażeń regularnych, 194
dostęp
  do baz danych, 348
  do modułów, 33
  do przestrzeni nazw, 33
  do zewnętrznych danych, 37
dostosowywanie generycznych typów kolekcji, 243
dowiązywanie statyczne, 505
drzewa
  niezrównoważone, 247
  składniowe, 197, 250
  zrównoważone, 48, 248
dwubitowy sumator, 316
dynamiczne
  operatory refleksji, 470
  rzutowanie w dół, 117
  szablony HTML-a, 373
dyrektywa __declspec, 492
dyrektywy narzędzia F# Interactive, 509
dyskryminator, 100
dziedziczenie
  implementacji, 149, 534
  interfejsów, 146

E

edycja kodu, 22, 502
efekty uboczne, 91, 93
elastyczne ograniczenia typu, 119
elementy aplikacji, 426
ETL, Extract, Transform, Load, 15

F

F#, 16
F# Interactive, 507
  dyrektywy, 509
finalizacja, 153
flowlet, 361, 398
format
  JSON, 195, 201, 339
  XML, 195, 197
formatowanie
  binarne, 207
  danych, 183
  łańcuchów znaków, 186, 187
  strukturalne, 186
formaty konkretne, 195, 201
formlety, 361, 392
  reaktywne, 399
  zależne, 398
formularze funkcyjne, 392
fraktale, 420, 424
FsLab, 253
funkcja, 24
  lambda, 55
  makeLayout, 411
  makeMenu, 412
  printf, 185
  Seq.choose, 222
  writeValue, 108
funkcje, 25, 130, 543
  anonimowe, 55
  formatujące, 185
  generyczne, 104, 123, 473
  kolekcji, 55, 56
  lokalne, 59, 60
  matematyczne, 258
  modułu Array, 72
  modułu WebSharper.Formlets.Enhance, 394
  określające typ, 124
  przekształcanie sekwencji, 215
  rekurencyjne, 44
  rekurencyjne ogonowo, 46
  statystyczne, 62, 267, 268
  używane jako wartości, 217
  wewnątrzwierszowe, 113
  wzajemnie rekurencyjne, 46
  z modułu List, 47

z modułu Seq, 215
zliczające, 62
zwracające klucz, 62
funkcyjne formularze sieciowe, 392

G

generator, 63
generator wyrażeń reprezentujących obliczenia, 457
generowanie
dokumentacji, 504
list, 219
nakładek, 499
tablic, 73, 219
zadań, 296
generyczna
funkcja, 87
serializacja binarna, 108
usługa memoizacji, 88
generyczne
algorytmy, 110, 111
formatowanie strukturalne, 186
formatowanie wartości, 107
haszowanie, 105
operatory, 124
pakowanie i wypakowywanie, 107
porównania, 107
typy kolekcji, 243
generyczność kodu dotyczącego liczb, 263
generyczny kod, 110
graficzne narzędzia projektowe, 416
graficzny interfejs użytkownika, GUI, 149, 407
grupowanie, 345
w sekwencjach, 223
GUI, Graphical User Interface, 149, 407

H

hash-consing, 232
haszowanie, 105, 107, 239, 240
strukturalne, 77
hermetyzacja, 140, 165
hierarchie, 113
interfejsowych typów obiektowych, 146
typów interfejsowych, 112
histogram, 268
hosting siteletów, 378
HSV, Hue, Saturation, Value, 422

I

IDE, Integrated Development Environment, 22, 513
idiomy, 537
IL, Intermediate Language, 170
imperatywne kolekcje, 74
implementowanie
interfejsowych typów obiektowych, 143
interfejsu IDisposable, 154
licznika, 297
lokalnych uproszczeń, 327
metody Async.Parallel, 305
modelowania probabilistycznego, 461
indeksowanie w sekwencjach, 223
inferencja, 24
typów, 25, 57, 120, 217
instrukcja let, 23, 26
instrukcje warunkowe, 44
integralność danych, 347
inteligentne aplikacje sieciowe, 355
interfejs
API, 335
IDisposable, 151, 153
PInvoke, 499
interfejsowe typy obiektowe, 141, 142
iteracje, 66
iterowanie
po sekwencji, 214
z wykorzystaniem funkcji, 60
izolowanie wykonywania kodu, 484

J

jawne
określanie argumentów, 123
tworzenie wątków, 306
tworzenie zadań, 306
jawny typ sygnaturowy, 179
jednostki miary, 273
język
C, 490
C++, 490
F#, 16
pośredni, IL, 170
SQL, 341
T-SQL, 348
wyrażeń algebraicznych, 327
JSON, JavaScript Object Notation, 195

K

kanał odpowiedzi, 299
kanoniczna krzywa sklejana, 419
kategorie
narzędzi, 501
wyjątków, 78
klasa, 130
Container, 410
System.Console, 84
klient rozbudowany, 360
klonowanie rekordów, 99
klucz, 261
kod imperatywny, 32
kodowanie
danych binarnych, 211
Unicode, 210
kody
do formatowania łańcuchów znaków, 186
liczbowe, 259
kolejka komunikatów, 409
kolejność kompilacji, 174, 177, 178
kolekcje, 74, 544
z platformy .NET, 483
kombinator
deszeregujący, 207
szeregujący, 207

kombinatory
HTML-a, 367
siteletów, 383
komentarze, 541
komórka cons, cons cell, 47
komórki referencji, 90
kompilacja, 177, 509
kompilowanie schematu, 466
komponenty
jednoplikowe, 181
wielokrotnego użytku, 181
kompozycja, 543
funkcji, 57
konstruktor try…finally, 79
konstruktory, 25
domyślne, 132
generyczne, 24
główne, 132
jawne, 133
konstrukty, 218, 289, 453
bibliotek, 529
pochodne, 313
konteksty udostępniania, 386
kontrola
przepełnienia, 256
typów, 24, 471
kontrolka
menedżera skryptu, 369
OwnerDrawButton, 434
kontrolki
graficzne, 410
lekkie, 444
niestandardowe, 432
kontynuacja, 249
konwencje
nazewnictwa, 531
wielkości znaków, 531
wywołań, 490
konwersja, 257
RGB na HSV, 423
konwersje, 43
krotka, 30, 544
krzywa Béziera, 419
kwantyfikowane formuły logiczne, 313
kwerendy, 220, 261, 341, 549
wewnętrzne, 344

L

leniwe
przetwarzanie, 94, 217
wartości, 89
liczby, 41
LINQ, Language Integrated Queries, 217
listy, 46, 544
literał tekstowy, 184
literały, 41, 255, 541
logiczne uporządkowanie danych, 346

Ł

łańcuch znaków, 41, 43, 183
łączenie
elementów, 413
kontrolek i menu, 410
narzędzi z debugowaniem, 520
operacji w potok, 56
operacji w potok, 57
podejścia funkcyjnego i obiektów, 151
siteletów, 383
skryptów, 502
wzorców, 53

M

macierz, 270, 440
odwrotna, 270
maszyny stanowe, 298
mechanizm
odzyskiwania pamięci, 152, 489
podziału na tokeny, 204
refleksji, 466
memoizacja, 89, 232
obliczeń, 86
unikatowa, 320
menu, 410
metoda, 35
Async.Parallel, 305
GetEnumerator, 67
GetResponse, 78
Object.Finalize, 156
Object.ToString(), 186
Regex.Matches, 67
System.IO.Path.GetExtension, 61
TryGetValue, 76
metody
instancji, 61
jako funkcje, 61
przeciążone, 137
składowe, 129
statyczne, 61
metodyka projektowania funkcyjnego, 528, 529
model
pamięci w platformie .NET, 307
REST, 387
modele
dziedzin, 227, 231
obliczenia na żądanie, 230
przekształcanie, 229
tworzenie węzłów, 232
zapisywanie w pamięci podręcznej, 231
list, 370
modelowanie
probabilistyczne, 461, 463
relacyjne, 316
moduł, 33
Array, 72
List, 47
Seq, 215, 220
funkcje, 215
WebSharper.Formlets.Enhance, 394
moduły, 170, 547
o nazwie typu, 172
otwierane automatycznie, 174
modyfikowalne
dane, 70
komórki referencji, 71, 90
rekordy, 68
struktury danych, 69, 77
wiązania let, 70
modyfikowanie liści, 229

N

nadzorowanie wykonywania kodu, 484
naiwne przetwarzanie rachunku zdań, 314
narzędzia, 501
  do edycji kodu, 22
narzędzie
  F# Interactive, 21, 508
  FsCheck, 521
  NuGet, 181
  NUnit, 519
  Paket, 182
  WebSharper, 355
  XUnit, 519
nawiązywanie połączeń, 349
nazwy pól rekordów, 99
niekontrolowane efekty uboczne, 459
niemodyfikowalne
  kolekcje, 49
  listy powiązane, 48
  odwzorowania, 49
  struktury danych, 69
  zbiory, 48
niemodyfikowalność, 23
niestandardowe
  atrybuty, 468
  operatory kwerend, 460
notacja
  Pascalowa, 531
  płynna, 56
  pobierania wycinków, 73
  z kropką, 29
  Wielbłądzia, 531

O

obiekt, 31, 85, 127, 140, 546
  typu Application, 413
  typu WebRequest, 35
obiekty
  do obsługi współbieżności, 152
  graficzne, 152
  modyfikowalne, 138
  obiekty obserwowane, 285
obliczana funkcja rekurencyjna, 87
obliczanie zbioru Mandelbrota, 421
obliczenia
  na żądanie, 230
  probabilistyczne, 461
  z efektami ubocznymi, 92
obsługa
  memoizacji, 87
  robota internetowego, 301
  SQL-a, 341
  wielu stron, 339
  współbieżności, 152
odbiorniki, 511
odchylenie standardowe, 262
odpowiedzi HTML-owe, 380
odrębność modyfikowalnych struktur danych, 92
odwzorowania
  struktur danych, 493
  typów, 496
odzyskiwanie pamięci z kopiowaniem, 489
ograniczenia
  dotyczące wartości, 121, 122
  interfejsu PInvoke, 499
  jednostek miary, 278
  parametrów określających typ, 239
ograniczenie
  comparison, 240
  equality, 240
określanie kolorów, 422
opcje, 49
opcjonalne ustawienia właściwości, 140
operacja
  choose, 217
  concat, 217
  filter, 217
  map, 217
  Seq.tryFind, 223
  Seq.tryPick, 223
  sort, 220
  truncate, 220
operacje
  arytmetyczne z kontrolą przepełnienia, 256
  asynchroniczne, 286, 288, 293
  bitowe, 258
  synchroniczne, 288
  wejścia – wyjścia, 81, 82, 83, 294
operator, 545
  |>, 57
  ->, 217
  >>, 57
  and, 44
  or, 44
operatory
  arytmetyczne, 42, 256
  binarne, 63
  generyczne, 124
  przeciążone, 134
  refleksji, 470
opóźnione obliczenia, 62
optymalizacja, 503
osadzanie pageletów, 368

P

pagelet, 361, 365
pakiet, 35
  FSharp.Data, 35, 37, 199, 338
  UI.Next, 367, 370
  Suave, 35, 38
  WebSharper.Html, 367, 373
pakowanie, 115
paleta kolorów, 423
pamięć
  alokowana na stercie, 151
  nieusuwalna, 153
  podręczna, 84, 89
parametry
  akumulujące, 246
  generyczne, 54
  określające typ, 54
  w postaci funkcji, 112
  wyjściowe, 76
parsowanie, 207, 339
  danych, 201
  łańcuchów znaków, 188
  prostych wartości, 188
  wyrażeń algebraicznych, 329
  zstępujące, 204, 205
pełnoprawne wartości, 285
pędzle, 417

pętla
for, 66
while, 66, 67
pętle
iterujące po sekwencjach, 67
sekwencyjne, 66
zdarzeń, 409
pierwsza aplikacja, 408
pierwszy program, 21
piglet, 361, 392, 400
piglety reaktywne, 401
PInvoke, 490, 491, 499
pióra, 417
pisanie kodu, 501
platforma
.NET, 74
WebSharper, 360, 362
plik CInteropDLL.h, 491
pliki
DLL, 175, 479
jako moduły, 173
projektu, 177
wykonywalne, 487
z sygnaturami, 179, 534
płynna notacja, 56
płynne ograniczenia, 125
pobieranie
danych, 339
danych z bazy, 363
elementów z sekwencji, 221
pakietów, 35
stron internetowych, 34
początkowe ustawienia właściwości, 140
podwójne buforowanie, 435
podzespół, 174
polimorfizm, 144
implementacji, 144
połączenia sieciowe, 152
połączenie z bazą, 349
pomoc, 36
ponowne wykorzystanie kodu, 181
porównania, 239
generyczne, 104
liczb, 43, 257
z użyciem atrybutów, 240
porządkowanie, 62
kodu, 159, 165, 170
postać normalna negacji, 238
pośrednik platformy .NET, 405
potok, 56, 217, 543
półsumator, 316
predykaty, 62
probabilistyka, 461
problem
z inferencją typów, 120
z ograniczeniami dotyczącymi wartości, 122
procedury składowane, 352
procesor, 304
procesy, 282
program
fsi.exe, 415
nuget.exe, 36
paket.exe, 36
programowanie
asynchroniczne, 281
funkcyjne, 41, 65, 84, 91, 97, 528
imperatywne, 65, 66, 84, 94
obiektowe, 127, 247, 528
proceduralne, 528
reaktywne, 281, 370
rekurencyjne, 244
równoległe, 281
symboliczne, 311
z użyciem liczb, 253
zorientowane na język, 451
programy
asynchroniczne, 282
reaktywne, 282
równoległe, 282
współbieżne, 282
projektowanie
bibliotek, 523
bibliotek platformy .NET, 524
funkcyjne, 528, 529
sprzętu, 312
z użyciem sygnatur, 180
projekty warstwowe, 178
propagacja ograniczeń typów, 120
przechwytywanie wyjątków, 79
przeciążanie
metod, 137
operatorów, 135
przeciążone
funkcje matematyczne, 258
operatory generyczne, 124
przeglądarka
fraktali, 420
internetowa, 414
przekazywanie
kontynuacji, 247
wskaźników, 498
przekształcanie
danych na wiele widoków, 234
modeli dziedzin, 229
sekwencji, 215
współrzędnych, 439
przepełnienie stosu, 249
przepływ pracy, 465, 548
przestrzenie kolorów, 422
przestrzeń nazw, 33, 170, 547
Eto.Forms, 413
FSharp.Core, 481
FSharp.Reflection, 485
System.Data.SqlClient, 153
System, 482
System.Collections.Generic, 483
System.IO, 153
System.Net, 153
System.Net.Sockets, 153
System.Text.RegularExpressions, 190
System.Threading, 153
System.Xml, 195
składowe, 196
typy, 196
WebSharper.UI.Next.Client, 368
z platformy .NET, 479
przetwarzanie
asynchroniczne, 288, 294
danych wejściowych, 188
drzew niezrównoważonych, 247
drzewa składniowego, 250
komunikatów, 298
liczb, 262
list, 245
skrzynki pocztowej, 297
przypisywanie do kategorii, 261
publikowanie zdarzeń, 284
punkt końcowy, 381

R

rachunek zdań, 311
ramki danych, 271
reaktywna aplikacja, 374
redundancja, 347
refleksja dotycząca typów, 466
reguły dopasowania, 53
rejestr, 312
rekordy, 99, 102
rekurencja, 91
  ogonowa, 245, 247
  prawidłowa, 46
rekurencyjne
  parsowanie zstępujące, 204, 205
  wyrażenia, 465
reprezentacje, 115
  abstrakcyjne, 451
  konkretne, 451
  obliczeniowe, 451
reprezentowanie
  formuł rachunku zdań, 320
  reprezentowanie rachunku zdań, 312
REST, 387
RGB, Red, Green, Blue, 422
rodzaje oprogramowania, 505
rodziny typów, 25
rozbudowywanie formletów, 395
rozkład macierzy, 270
rozkłady, 266, 268
rozrastanie się aplikacji, 415
rozszerzanie istniejących
  modułów, 157
  typów, 157
rozszerzenia natywne, 404
rozwijanie aplikacji sieciowych, 360
równoległe
  pobieranie, 286
  przetwarzanie plików, 290
  wykorzystanie procesorów, 304
różniczkowanie, 325
  wyrażeń algebraicznych, 333
rysowanie
  analogowego zegara, 435
  aplikacji, 416
rzutowanie
  w dół, 117
  w górę, 116, 118

S

screen scraping, 37
sekwencje, 213, 219, 259, 548
  grupowanie, 223
  indeksowanie, 223
  iterowanie, 214
  leniwe, 216
  operatory, 220
  pobieranie elementów, 221
  przekształcanie, 215
  składanie, 224
  typy, 216
  ucieczki, 185
  wyszukiwanie elementów, 222
  zwalnianie zasobów, 225
serwer WWW, 38, 356
setter, 170
sieciowy interfejs API, 335
  samowystarczalny, 377
sitelety, 361, 375
  offline, 378
  online, 378
  samowystarczalne, 378
  z jednym punktem końcowym, 379
  z uwierzytelnianiem, 384
  z wieloma punktami końcowymi, 381
składanie sekwencji, 224
składnia abstrakcyjna, 197, 323
składowe, 29, 127, 130
  generatora wyrażeń, 458
  instancji, 29
  obiektów typu Doc, 368
  obiektu fsi, 508
  rozszerzające, 157, 158
  statyczne, 29
  z typu MailboxProcessor, 300
  z typu WebSharper.Sitelets.Context, 386
skrypt, 502
skrzynka pocztowa, 297, 300
słowniki, 75
  z kluczami złożonymi, 77
słowo kluczowe
  in, 26
  let, 26
  null, 162
sortowanie, 343
specyfikatory formatowania, 187
sprawdzanie
  prostych cech układów, 319
  równości, 239, 240
  typów, 117
  układów, 311, 323
SQL, 341, 346
stabilność kodu, 194
statyczne rzutowanie w górę, 116
statystyka, 259
sterowanie
  przepływem, 543
  za pomocą funkcji, 60
sterownik, 334
sterta, 244
stos, 151, 244
stosowanie
  jednostek, 278
  operacji asynchronicznych, 304
  właściwości indeksowych, 134
  wyrażeń regularnych, 190
struktury danych, 28, 161, 482
  modyfikowalne, 48
  niemodyfikowalne, 48
strumienie w .NET, 82
sumator
  pełny, 316
  z przeniesieniem warunkowym, 323
sygnatury, 180
symboliczne
  dane, 319
  różniczkowanie wyrażeń algebraicznych, 333
sytuacja wyścigu, 307
szablony HTML-a, 371
szeregi czasowe, 271
szeregowanie
  łańcuchów znaków, 496
  parametrów, 494

Ś

śledzenie kodu, 510
środowisko uruchomieniowe
  CLR, 477
świat, 440

T

tabela wyszukiwania, 86
tablice, 71, 544
  dwuwymiarowe, 73
  o zmiennej długości, 74
tautologia, 313
technika screen scraping, 37
techniki implementowania
  obiektów, 146
technologia PInvoke, 490
testowanie, 516
testy
  jednostkowe, 518, 520
  oparte na właściwościach, 521
tokeny, 204
tożsamość, 71
  obiektu, 24
transakcje, 347
transformacje, 63
T-SQL, 348
twierdzenia podstawowe, 313
tworzenie
  baz danych, 350
  dostawców typów, 474
  funkcji generycznej, 123
  interfejsu użytkownika, 408
  modeli probabilistycznych,
    465
  niegenerycznych wartości, 122
  niestandardowych kontrolek,
    432
  pageletów, 366
  pakietów, 181
  pierwszego programu, 21
  podzespołów, 175
  projektu warstwowego, 178
  przeglądarki fraktali, 420
  reaktywnych formletów, 399
  reaktywnych pigletów, 401
  samowystarczalnych siteletów,
    377
  siteletu, 376, 383
  standardowych elementów
    aplikacji, 426
  tabel, 350
  typów pochodnych, 116
  wątków, 306
  węzłów modelu dziedziny,
    232
  wykresów, 254
  wzorców, 53
  zadań, 306
  zdarzeń, 284
typ, 25, 97, 542, 546
  FSharp.Data.JsonProvider,
    202
  funkcji, 24
  LightWeightContainerControl,
    446
  LightWeightControl, 444
  obiektowy ze stanem, 138
  option, 49
  SqlCommandProvider, 348
  sygnaturowy, 179
  System.Collections.Generic.
    ↪Dictionary, 75
  TableLayout, 414
  TimeSpan, 61
  XmlProvider, 199
typy
  atrybutów, 116
  bezpośrednie, 115, 534
  częściowo konkretne, 148
  delegatów, 115, 486
  elastyczne, 119
  generyczne, 102
  interfejsowe, 112
  liczbowe, 255
  obiektowe, 138
  ogólne, 484
  platformy .NET, 160
  pochodne, 116
  podstawowe, 482, 541
  proste, 41
  referencyjne, 115
  rekordowe, 98
  sygnaturowe, 180
  używane z siteletach, 380
  wyjątków, 78, 80, 116
  wyliczeniowe, 116, 162
  z implementacją interfejsu
    IDisposable, 153
  z platformy .NET, 485

U

uchwyty plików, 151
udostępnianie
  danych, 38
  mechanizmów jawne, 113
  mechanizmów niejawne, 113
  pakietów, 181
  treści w sieci, 355
ujednolicanie współrzędnych, 437
ujścia, 63
układ współrzędnych, 437
układy sprzętowe, 317
ukrywanie
  elementów, 165
  kodu, 166, 168
  modyfikowalnych danych, 70
umieszczanie
  kodu w module, 171
  modułów w przestrzeniach
    nazw, 171
  typów w przestrzeniach nazw,
    171
unia z dyskryminatorem, 100, 102
Unicode, 185, 210
unikanie aliasów, 69
upraszczanie wyrażeń
  algebraicznych, 325, 331
uruchamianie
  asynchronicznych operacji,
    293
  serwera WWW, 38
  siteletów, 376
usługi, 337, 482
ustawienia
  konfiguracyjne, 507
  optymalizacji, 503
  właściwości obiektów, 140
usuwanie jednostek, 278
używanie
  bibliotek obiektowych, 33
  cytowań, 471
  częściowo
    zaimplementowanych
    typów, 149

danych, 506
diagnostyki, 510
formatu JSON, 201
funkcji kolekcji, 55
funkcji lokalnych, 59
funkcji printf, 185
histogramów, 268
instrukcji let, 23
interfejsowych typów obiektowych, 145
klas, 130
klasy System.Console, 84
kodu imperatywnego, 32
konstruktu try…finally, 79
kontynuacji, 249
krotek, 30
leniwych sekwencji, 216
metody TryGetValue, 76
modyfikowalnych rekordów, 68
modyfikowalnych wiązań let, 70
notacji z kropką, 29
obiektów, 160
ogólnych typów, 484
używanie
pakietów, 35
plików jako modułów, 173
procedur składowanych, 352
przestrzeni nazw System.Text.Regular ↪Expressions, 190
przestrzeni nazw System.Xml, 195
słowników, 75
słowników z kluczami złożonymi, 77
struktur danych, 28
sygnatur, 180
typu XmlProvider, 199
typów systemowych, 481
typu FSharp.Data.JsonProvider, 202
unii z dyskryminatorami, 102
wartości null, 162
wartości w postaci funkcji, 55
wewnętrznej struktury danych, 90
właściwości, 29
wyrażeń reprezentujących sekwencje, 217
wzorców, 50
XML-a, 195

W

wariancja, 262
wartości, 23, 31, 35
niemodyfikowalne, 24
typu Async<'T>, 293
ustrukturyzowane, 51
w postaci funkcji, 48, 54, 55
wartość, value
null, 162, 344
własna macierzy, 270
wątki, 152, 282, 306
wykonania, 409
wcięcia w kodzie, 26
wektory, 270
wiązanie, 543
let, 70
widoki danych strukturalnych, 234
widżet, 410
witryna, 39
własne kontrolki, 431
właściwości, 29, 35
automatyczne, 141
indeksowe, 134
indeksujące, 35
wskaźnik, 498
współbieżność, 305, 309
współdziałanie, 477
z platformą .NET, 217
współrzędne
ujednolicone, 439
widoków, 440, 441
współużytkowana pamięć, 307
współużytkowanie danych, 346
wstępne wykonywanie obliczeń, 84, 85
wycinki tablic, 73
wydajność, 431
wyjątki, 78, 295, 533, 544
wykonywanie kodu, 484
wykres, 254
liniowy, 255
punktowy, 254
wyrażenia
algebraiczne, 329, 331
regularne, 190
znaczenie znaków, 192
reprezentujące obiekt, 143
reprezentujące obliczenia, 452, 453, 459, 460
reprezentujące przedział, 214
reprezentujące sekwencje, 217, 219, 226, 548
ustalające powodzenie, 454
wyszukiwanie elementów w sekwencjach, 222
wyścig, 307
wyświetlanie witryny, 39
wywołania
częściowe, 58, 84
ogonowe, 244
zwrotne, 62, 146
wywołanie open, 33, 34
wywoływanie
efektów ubocznych, 81
funkcji, 25
po stronie klienta, 382
wzorce, 30, 50, 53, 542
aktywne, 234, 358
częściowe, 237
parametryzowane, 237
ukrywanie reprezentacji, 237
do sprawdzania typów, 117
wzorzec projektowy
model – widok – kontroler, 434

X

XML, Extensible Markup Language, 195

Z

zabezpieczanie reguł, 53
zadania, tasks, 296, 306
platformy .NET, 296
zalecane idiomy, 537
zależności, 506
zapisywanie
operacji, 226
w pamięci podręcznej, 231

zapobieganie otwieraniu modułu, 172
zarządzane operacje asynchroniczne, 289
zarządzanie
  pamięcią, 488
  zależnościami, 506
  zasobami, 154, 403
zasięg, scope, 27
zasoby, 151
zbiór Mandelbrota, 420
zdarzenia, 283, 409
  jako pełnoprawne wartości, 285
zegar analogowy, 435
złączenia, 345
zmienna, variable, 23
  określająca typ, 61
zmienność, 140
znak
  lewego ukośnika, 184
  średnika, 26
znaki
  ASCII, 184
  Unicode, 210
  w wyrażeniach regularnych, 192
zwalnianie
  niezarządzanych obiektów, 155
  wewnętrznych obiektów, 154
  zasobów, 151, 153, 225
zwiększanie stabilności kodu, 194
zwracanie wartości, 534

Ź

źródła programowania funkcyjnego, 528

Ż

żądania REST, 338

F# jest nowoczesnym, lecz dojrzałym językiem programowania, który zyskuje coraz większą popularność. Ułatwia korzystanie z kilku paradygmatów programowania: programowania funkcyjnego, obiektowego i imperatywnego. Został zaprojektowany do pisania prostego, solidnego i wydajnego kodu, lecz przydaje się do rozwiązywania złożonych problemów. Kod w F# jest zwięzły i efektywny. Sam F# jest projektem *open source*, jednak stanowi część platformy .NET. To wszystko czyni z niego język przyszłości!

Trzymasz w ręku kompletny przewodnik po języku F#, w którym wyczerpująco przedstawiono informacje niezbędne do rozpoczęcia pracy w tej technologii. Zwięźle opisano tu paradygmaty obsługiwane przez F#, a następnie pokazano, jak można wykorzystać ten język do rozwiązywania problemów z siecią, danymi, programowaniem równoległym i analizami. W ten sposób nauczysz się stosować wspomniane paradygmaty i używać kwerend, co pozwoli Ci osiągnąć wysoką produktywność w pisaniu programów dla wielu systemów i technologii.

W tej książce znajdziesz:

- wyjaśnienie paradygmatów programowania funkcyjnego, obiektowego i imperatywnego
- kompletne informacje na temat najnowszej wersji języka F#
- instrukcje projektowania bibliotek języka F#
- wskazówki dotyczące pisania hermetycznego i uporządkowanego kodu
- zagadnienia programowania reaktywnego, asynchronicznego i równoległego
- techniki rozwiązywania problemów programistycznych za pomocą F#

Don Syme — jest architektem języka F#. Brał udział w rozwijaniu typów generycznych w C# i technologii .NET Common Language Runtime. W 2015 roku został odznaczony Srebrnym Medalem Królewskiej Akademii Inżynierii.
Adam Granicz — ma 10-letnie doświadczenie w pracy z językiem F#. Pracuje nad narzędziem WebSharper — podstawowym środowiskiem języka F#. Regularnie pisze artykuły i zabiera głos na konferencjach.
Antonio Cisternino — jest profesorem Uniwersytetu w Pizie we Włoszech. Zajmuje się głównie metaprogramowaniem i budową architektury systemów. Od kilku lat korzysta z F# i bierze udział w rozwijaniu tego języka.

ISBN 978-83-283-2943-0
cena: 99,00 zł