

Baza Programisty

Programowanie w języku Clojure



Stuart Halloway
Aaron Bedra

Książka z przedmową

Richa Hickeya – twórcy języka Clojure

Helion



Tytuł oryginału: Programming Clojure

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-246-5372-0

© Helion 2013.
All rights reserved.

Copyright © 2012 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek 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.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi 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)

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

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

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to!» Nasza społeczność](#)

Spis treści

Podziękowania	10
Przedmowa do wydania drugiego	11
Przedmowa do wydania pierwszego	13
Wstęp	15
Rozdział 1. Wprowadzenie	23
1.1. Dlaczego Clojure?	24
1.2. Szybkie wprowadzenie do programowania w Clojure	34
1.3. Biblioteki języka Clojure	40
1.4. Podsumowanie	44
Rozdział 2. Przegląd języka Clojure	45
2.1. Konstrukcje składniowe	46
2.2. Makra odczytu	55
2.3. Funkcje	56
2.4. Zmienne, wiązania i przestrzenie nazw	61
2.5. Wywoływanie kodu Javy	68
2.6. Przepływ sterowania	70
2.7. Gdzie się podziela pętla for?	74
2.8. Metadane	77
2.9. Podsumowanie	79

Rozdział 3. Ujednolicanie danych za pomocą sekwencji	81
3.1. Wszystko jest sekwencją	83
3.2. Stosowanie biblioteki sekwencji	87
3.3. Sekwencje nieskończone i „leniwe”	96
3.4. W Clojure Java jest sekwencyjna	98
3.5. Funkcje przeznaczone dla konkretnych struktur	104
3.6. Podsumowanie	113
Rozdział 4. Programowanie funkcyjne	115
4.1. Zagadnienia z obszaru programowania funkcyjnego	116
4.2. Jak stosować „leniwe” podejście?	121
4.3. Leniwsze niż leniwe	130
4.4. Jeszcze o rekurencji	136
4.5. Podsumowanie	146
Rozdział 5. Stan	147
5.1. Współbieżność, równoległość i blokady	148
5.2. Referencje i pamięć STM	150
5.3. Nieskoordynowane i synchroniczne aktualizacje za pomocą atomów	157
5.4. Stosowanie agentów do asynchronicznego aktualizowania danych	158
5.5. Zarządzanie stanem specyficznym dla wątku za pomocą zmiennych	163
5.6. Gra Snake w języku Clojure	168
5.7. Podsumowanie	178
Rozdział 6. Protokoły i typy danych	179
6.1. Programowanie z wykorzystaniem abstrakcji	180
6.2. Interfejsy	183
6.3. Protokoły	184
6.4. Typy danych	188
6.5. Rekordy	193
6.6. Makro reify	198
6.7. Podsumowanie	199

Rozdział 7. Makra	201
7.1. Kiedy należy stosować makra?	202
7.2. Makro do sterowania przebiegiem programu	202
7.3. Upraszczenie makr	209
7.4. Taksonomia makr	214
7.5. Podsumowanie	224
Rozdział 8. Wielometody	225
8.1. Życie bez wielometod	226
8.2. Definiowanie wielometod	228
8.3. Więcej niż proste wybieranie metod	231
8.4. Tworzenie doraźnych taksonomii	233
8.5. Kiedy należy korzystać z wielometod?	237
8.6. Podsumowanie	241
Rozdział 9. Sztuczki z Javą	243
9.1. Obsługa wyjątków	244
9.2. Zmagania z liczbami całkowitymi	248
9.3. Optymalizowanie wydajności	250
9.4. Tworzenie klas Javy w języku Clojure	255
9.5. Praktyczny przykład	261
9.6. Podsumowanie	268
Rozdział 10. Tworzenie aplikacji	269
10.1. Wynik w grze Clojurebreaker	270
10.2. Testowanie kodu zwracającego wynik	274
10.3. Biblioteka test.generative	278
10.4. Tworzenie interfejsu	287
10.5. Instalowanie kodu	292
10.6. Pożegnanie	295
Dodatek A. Edytory kodu	297
Dodatek B. Bibliografia	299
Skorowidz	301

Rozdział 1.

Wprowadzenie

Szybki wzrost popularności języka Clojure wynika z wielu przyczyn. Po krótkich poszukiwaniach w sieci WWW dowiesz się, że Clojure:

- ◆ jest językiem funkcyjnym;
- ◆ jest Lisphem na maszynie JVM;
- ◆ ma specjalne mechanizmy do obsługi współbieżności.

Wszystkie te cechy są ważne, jednak żadna z nich nie odgrywa najważniejszej roli. Naszym zdaniem najistotniejszymi aspektami języka Clojure są jego prostota i możliwości.

Prostota w kontekście oprogramowania jest ważna z kilku względów, tu jednak mamy na myśli pierwotne i najważniejsze znaczenie tego słowa — proste jest to, co nie jest złożone. Proste komponenty umożliwiają systemowi przeprowadzanie operacji zaplanowanych przez projektantów i nie wykonują czynności niepowiązanych z danym zadaniem. Z naszych doświadczeń wynika, że niewielka złożoność zwykle szybko przekształca się w niebezpiecznie poważną.

Także słowo *możliwości* ma wiele znaczeń. Tu mamy na myśli zdolność do wykonywania stawianych aplikacji zadań. Aby programista miał odpowiednie możliwości, musi wykorzystać platformę, która sama je posiada i jest powszechnie dostępna. Taką platformą jest na przykład maszyna JVM. Ponadto używane narzędzia muszą zapewniać pełny, nieograniczony dostęp do oferowanych możliwości. Dostęp do możliwości jest często podstawowym wymogiem w projektach, w których trzeba w pełni wykorzystać platformę.

Przez lata tolerowaliśmy bardzo skomplikowane narzędzia, które były jedynym sposobem na uzyskanie potrzebnych możliwości. Czasem akceptowaliśmy też ograniczone możliwości w celu uproszczenia modelu programowania. Niekiedy nie da się uniknąć pewnych kompromisów, jednak w obszarze możliwości i prostoty nie trzeba się z nimi godzić. Język Clojure to dowód na to, że cechy te można połączyć.

1.1. Dlaczego Clojure?

Wszystkie charakterystyczne cechy języka Clojure mają zapewniać prostotę, możliwości lub i jedno, i drugie. Oto kilka przykładów:

- ◆ Programowanie funkcyjne jest proste, ponieważ pozwala oddzielić obliczenia od stanu i tożsamości. Zaletą jest to, że programy funkcyjne są łatwiejsze do zrozumienia, pisania, testowania, optymalizowania i równoległego wykonywania.
- ◆ Konstrukcje umożliwiające współdziałanie języków Clojure i Java dają duże możliwości, ponieważ zapewniają bezpośredni dostęp do składni Javy. Zaletą jest to, że można uzyskać wydajność na poziomie Javy i stosować składnię charakterystyczną dla tego języka. Co ważniejsze, nie trzeba uciekać się do języka niższego poziomu, aby zapewnić sobie dodatkowe możliwości.
- ◆ Lisp jest prosty w dwóch bardzo ważnych aspektach — oddziela wczytywanie od wykonania, a jego składnia obejmuje niewielką liczbę niezależnych elementów. Zaletą jest to, że wzorce projektowe są ujęte w abstrakcyjne konstrukcje składniowe, a S-wyrażenia obejmują kod w językach XML, JSON i SQL.
- ◆ Lisp daje też duże możliwości, ponieważ udostępnia kompilator i system makr działający w czasie wykonywania programu. Zaletą jest to, że w Lispie występuje późne wiązanie i można łatwo tworzyć języki DSL.
- ◆ Model czasu w języku Clojure jest prosty. Oddzielono w nim wartości, tożsamość, stan i czas. Zaletą jest to, że w programach można sprawdzać i zapamiętywać informacje bez obaw o to, że starsze dane zostaną nadpisane.
- ◆ Protokoły są proste. W Clojure polimorfizm jest niezależny od dziedzinienia. Zaletą jest to, że można bezpiecznie i w konkretnym celu rozszerzać typy oraz abstrakcje. Nie wymaga to stosowania zawiłych wzorców projektowych lub wprowadzania zmian w cudzym kodzie (co często prowadzi do błędów).

Ta lista cech stanowi „mapę” pomocną w dalszych rozdziałach książki. Nie martw się, jeśli na razie nie rozumiesz wszystkich szczegółów. Każdej z tych cech poświęcamy cały rozdział.

Zobaczmy, jak niektóre z tych cech sprawdzają się w praktyce. Zbudujmy w tym celu prostą aplikację. Przy okazji dowiesz się, jak wczytywać i wykonywać większe przykładowe programy, które prezentujemy dalej w książce.

Język Clojure jest elegancki

W języku Clojure stosunek sygnału do szumu jest wysoki. Dlatego programy pisane w tym języku są krótkie. Takie programy są tańsze w tworzeniu, instalowaniu i konserwacji¹. Jest to prawdą zwłaszcza wtedy, gdy programy są zwięzłe, a nie tylko treściwe. Przyjrzyj się na przykład poniższemu kodowi w Javie (pochodzi on z projektu Apache Commons):

```
data/snippets/isBlank.java
public class StringUtils {
    public static boolean isBlank(String str) {
        int strLen;
        if (str == null || (strLen = str.length()) == 0) {
            return true;
        }
        for (int i = 0; i < strLen; i++) {
            if ((Character.isWhitespace(str.charAt(i)) == false)) {
                return false;
            }
        }
        return true;
    }
}
```

Metoda `isBlank()` sprawdza, czy łańcuch znaków jest *pusty* (nie zawiera żadnych znaków lub obejmuje same odstępny). Oto kod podobnej metody w języku Clojure:

```
src/examples/introduction.clj
(defn blank? [str]
  (every? #(Character/isWhitespace %) str))
```

Wersja w języku Clojure jest krótsza, a co ważniejsze — *prostsza*. Nie występują tu zmienne, modyfikowalny stan ani rozgałęzienia. Efekt ten jest możliwy dzięki *funkcjom wyższego rzędu*. Funkcje tego rodzaju przyjmują inne funkcje

¹ *Software Estimation: Demystifying the Black Art* [McC06] to znakomita książka. Jej autorzy dowodzą, że krótsze jest tańsze.

jako argumenty i (lub) zwracają funkcje. Funkcja `every?` przyjmuje funkcję i kolekcję, a zwraca wartość `true`, jeśli otrzymana funkcja zwraca `true` dla każdego elementu z kolekcji.

Ponieważ w wersji w języku Clojure nie ma rozgałęzień, kod jest bardziej czytelny i łatwiejszy do przetestowania. Zalety te są jeszcze wyraźniejsze w większych programach. Ponadto, choć kod jest zwięzły, można go łatwo zrozumieć. Program w języku Clojure można potraktować jak *definicję* pustego łańcucha znaków — jest to łańcuch, w którym każdy znak jest odstępem. Kod ten jest znacznie lepszy od metody z projektu Commons, w którym definicja pustego łańcucha znaków jest ukryta za szczegółowym kodem pętli i instrukcji `if`.

Oto inny przykład. Przyjrzyj się banalnej klasie `Person` napisanej w języku Java:

```
data/snippets/Person.java
```

```
public class Person {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

W języku Clojure rekord `Person` można zdefiniować w jednym wierszu:

```
(defrecord Person [first-name last-name])
```

Korzystać z tego rekordu można tak:

```
(def foo (->Person "Aaron" "Bedra"))
-> #'user/foo
foo
-> #:user.Person{:first-name "Aaron", :last-name "Bedra"}
```

Instrukcję `defrecord` i powiązane funkcje omawiamy w podrozdziale 6.3, „Protokoły”.

Kod w języku Clojure nie tylko jest znacznie krótszy; rekord `Person` jest tu *niezmienny*. Niezmiennie struktury danych są z natury bezpieczne ze względu na wątki, a mechanizmy modyfikowania danych można utworzyć w odrębnej warstwie za pomocą referencji, agentów i atomów języka Clojure. Techniki te omawiamy w rozdziale 5., „Stan”. Ponieważ rekordy są niezmiennie, język Clojure automatycznie udostępnia poprawne implementacje funkcji `hashCode()` i `equals()`.

Język Clojure ma wbudowanych wiele eleganckich rozwiązań, jeśli jednak stwierdzisz, że czegoś Ci w nim brakuje, możesz samodzielnie dodać potrzebne mechanizmy. Umożliwiają to cechy Lispa.

Clojure to odświeżony Lisp

Clojure jest Lispem. Od dziesięcioleci zwolennicy Lispa mówią o przewagach, jakie język ten ma w porównaniu z właściwie wszystkimi innymi językami. Plan opanowania świata przez Lispa jest jednak realizowany dość powoli.

Twórcy języka Clojure, podobnie jak autorzy każdej odmiany Lispa, musieli zmierzyć się z dwoma problemami. Oto one:

- ◆ Clojure ma odnieść sukces jako odmiana Lispa. Wymaga to przekonania użytkowników Lispa, że Clojure obejmuje najważniejsze mechanizmy swojego poprzednika.
- ◆ Jednocześnie Clojure ma odnieść sukces tam, gdzie *wcześniejsze wersje Lispa się nie przyjęły*. Wymaga to zdobycia poparcia większej społeczności programistów.

Autorzy języka Clojure radzą sobie z tymi problemami przez udostępnienie mechanizmów metaprogramowania (charakterystycznych dla Lispa) i wprowadzenie zestawu uprawnień składniowych ułatwiających stosowanie języka Clojure programistom, którzy nie znają Lispa.

Dlaczego Lisp?

Wersje Lispa mają mały rdzeń, są prawie pozbawione składni i mają rozbudowane mechanizmy do obsługi makr. Z uwagi na te cechy można zmodyfikować Lispa pod kątem projektu, zamiast dostosowywać projekt do Lispa. Przyjrzyj się poniższemu fragmentowi kodu w Javie:

```
public class Person {
    private String firstName;
    public String getFirstName() {
        // Ciąg dalszy.
    }
}
```

W kodzie występuje metoda `getFirstName()`. Metody są polimorficzne i można je dostosować do potrzeb. Jednak znaczenie *każdego innego słowa* w przykładowym kodzie jest *określane przez język*. Czasem wygodna jest możliwość zmiany znaczenia poszczególnych słów. Pozwala to na przykład:

- ◆ zdefiniować słowo `private` jako „prywatne w kodzie produkcyjnym, ale publiczne na potrzeby serializacji i testów jednostkowych”;
- ◆ zdefiniować słowo `class` w taki sposób, aby platforma automatycznie generowała metody pobierające i ustawiające dla pól prywatnych (o ile programista nie zarządzi inaczej);
- ◆ utworzyć podklasę dla `class` i umieścić w niej wywoływane zwrotnie uchwyt dla zdarzeń cyklu życia; klasa z obsługą cyklu życia może na przykład zgłaszać zdarzenie utworzenia egzemplarza tej klasy.

Natykaliśmy się na programy, w których potrzebne były wszystkie te cechy. Bez potrzebnych mechanizmów programiści musieli uciekać się do powtarzalnych i podatnych na błędy sztuczek. Aby poradzić sobie z brakiem niezbędnych rozwiązań, napisano dosłownie *miliony* wierszy kodu.

W większości języków trzeba poprosić osoby odpowiedzialne za implementację, aby dodały wspomniane wcześniej mechanizmy. W Clojure możesz samodzielnie dodać potrzebne cechy, pisząc *makra* (rozdział 7., „Makra”). Nawet sam język Clojure jest zbudowany za pomocą makr, takich jak `defrecord`:

```
(defrecord name [arg1 arg2 arg3])
```

Jeśli chcesz zmienić znaczenie słowa, możesz napisać własne makro. Jeżeli potrzebujesz rekordów o ścisłej kontroli typów i z opcjonalnym sprawdzaniem, czy pola nie mają wartości `null`, wystarczy utworzyć własne makro `defrecord`. Powinno ono wyglądać tak:

```
(defrecord name [Type :arg1 Type :arg2 Type :arg3]
  :allow-nulls false)
```

Możliwość zmiany działania języka w nim samym jest wyjątkową zaletą Lispa. Można znaleźć wiele opisów różnych aspektów tego podejścia:

- ◆ Lisp jest homoikoniczny². Oznacza to, że kod w Lispie to dane. Dlatego łatwo jest tworzyć programy za pomocą innych programów.

² <http://pl.wikipedia.org/wiki/homoikoniczność>

- ◆ Cały język jest dostępny w każdym momencie. Paul Graham w artykule *Revenge of the Nerds*³ wyjaśnia, dlaczego jest to ważne.

Składnia Lispa eliminuje też problemy z priorytetami operatorów i łącznością operacji. W książce tej nie znajdziesz tabel dotyczących tych zagadnień. Z uwagi na wymóg stosowania nawiasów wyrażenia są jednoznaczne.

Wadą prostej, jednolitej składni Lispa (przynajmniej dla początkujących) jest nacisk na nawiasy i listy (listy to główny typ danych w Lispie). Clojure udostępnia ciekawe połączenie cech, które ułatwiają używanie Lispa programistom znającym inne języki.

Lisp z mniejszą liczbą nawiasów

Clojure zapewnia istotne zalety programistom używającym innych odmian Lispa. Oto te korzyści:

- ◆ W języku Clojure fizyczne listy z Lispa są uogólnione do abstrakcyjnej postaci — *sekwencji*. Pozwala to zachować możliwości, jakie dają listy, a przy tym wykorzystać ich zalety w innych strukturach danych.
- ◆ Wykorzystanie maszyn JVM jako podstawy dla kodu w języku Clojure daje dostęp do standardowej biblioteki i bardzo popularnej platformy.
- ◆ Sposób analizowania symboli i cudzysłówów sprawia, że w języku Clojure pisanie standardowych makr jest stosunkowo proste.

Wielu użytkowników języka Clojure nie zna Lispa, za to prawdopodobnie słyszało niepochlebne opinie na temat stosowania w nim nawiasów. W języku Clojure zachowano nawiasy (i możliwości Lispa!), jednak pod kilkoma względami usprawniono tradycyjną składnię Lispa.

- ◆ Clojure zapewnia wygodną, opartą na literałach składnię dla różnych struktur danych (nie tylko dla list), takich jak wyrażenia regularne, odwzorowania, zbiory, wektory i metadane. Dlatego kod w języku Clojure jest mniej zależny od list niż w większości innych odmian Lispa. Między innymi parametry funkcji są podawane w wektorach, `[]`, a nie w listach, `()`.

```
src/examples/introduction.cj
```

```
(defn hello-world [username]  
  (println (format "Witaj, %s" username)))
```

Zastosowanie wektora sprawia, że lista argumentów jest lepiej widoczna, co ułatwia czytanie definicji funkcji w języku Clojure.

³ <http://www.paulgraham.com/icad.html>

- ◆ W języku Clojure, inaczej niż w większości odmian Lispa, przecinki są traktowane jak odstępy.

```
; Wektory przypominają tablice z innych języków.
[1, 2, 3, 4]
-> [1 2 3 4]
```

- ◆ Język Clojure jest idiomatyczny i nie wymaga niepotrzebnego zagnieżdżenia nawiasów. Przyjrzyj się makru `cond`, które występuje zarówno w Common Lispie, jak i w Clojure. Makro `cond` sprawdza zestaw par test-wynik i zwraca pierwszy wynik, dla którego wartość wyrażenia testowego to `true`. Każda para test-wynik znajduje się w nawiasach:

```
; Makro cond w Common Lispie.
(cond ((= x 10) "equal")
      (> x 10) "more"))
```

W języku Clojure dodatkowe nawiasy są niepotrzebne.

```
; Makro cond w Clojure.
(cond (= x 10) "equal"
      (> x 10) "more")
```

Jest to kwestia czysto estetyczna i oba podejścia mają swoich zwolenników. Ważne jest to, że język Clojure pozbawiono uciążliwych aspektów Lispa, jeśli było to możliwe bez utraty zalet tego ostatniego.

Clojure jest świetną odmianą Lispa zarówno dla ekspertów, jak i dla początkujących programistów Lispa.

Clojure to język funkcyjny

Clojure jest językiem funkcyjnym, natomiast nie jest (w odróżnieniu od Haskell) czysto funkcyjny. Oto cechy języków funkcyjnych:

- ◆ Funkcje to *pełnoprawne obiekty*. Oznacza to, że funkcje można tworzyć w czasie wykonywania programu, przekazywać, zwracać i ogólnie używać ich jak wszelkich innych typów danych.
- ◆ Dane są *niezmienne*.
- ◆ Funkcje są *czyste*, czyli nie mają efektów ubocznych.

W wielu obszarach programy funkcyjne są łatwiejsze do zrozumienia, mniej podatne na błędy i *znacznie* łatwiejsze do wielokrotnego użytku. Na przykład poniższy krótki program wyszukuje w bazie danych utwory każdego kompozytora, który napisał dzieło pod tytułem „Requiem”:

```
(for [c compositions :when (= "Requiem" (:name c))] (:composer c))
-> ("W. A. Mozart" "Giuseppe Verdi")
```

Słowo `for` nie jest początkiem pętli, ale *wyrażenia listowego* (ang. *list comprehension*). Kod ten oznacza „dla każdego `c` z kolekcji `compositions`, gdzie nazwą `c` jest „Requiem”, podaj kompozytora `c`”. Wyrażenia listowe omawiamy w punkcie „Przekształcanie sekwencji”.

Przykładowy kod ma cztery pożądane cechy:

- ◆ Jest *prosty*. Nie obejmuje pętli, zmiennych ani zmiennego stanu.
- ◆ Jest *bezpieczny ze względu na wątki*. Nie wymaga stosowania blokad.
- ◆ Jest *możliwy do równoległego wykonywania*. Każdy krok można przydzielić do odrębnego wątku bez konieczności modyfikowania kodu poszczególnych kroków.
- ◆ Jest *uniwersalny*. Kolekcją `compositions` może być zwykły zbiór, kod w XML-u lub zbiór wyników z bazy danych.

Warto porównać programy funkcyjne z programami *imperatywnymi*, w których instrukcje zmieniają stan programu. Większość programów obiektowych pisze się w stylu imperatywnym, dlatego nie mają one *żadnych* z wymienionych wcześniej zalet. Są niepotrzebnie skomplikowane, niebezpieczne ze względu na wątki, nie umożliwiają równoległego działania, a kod jest trudny do uogólnienia. Bezpośrednie porównanie podejścia funkcyjnego i imperatywnego znajdziesz w podrozdziale 2.7, „Gdzie się podziela pętla `for`?”.

Programiści znają zalety języków funkcyjnych już od długiego czasu. Jednak języki funkcyjne, na przykład Haskell, nie zdobyły dominującej pozycji. Wynika to z tego, że nie wszystkie operacje można wygodnie wykonać w podejściu czysto funkcyjnym.

Są cztery powody, dla których język Clojure może zyskać większe zainteresowanie niż dawne języki funkcyjne:

- ◆ Podejście funkcyjne jest dziś bardziej przydatne niż kiedykolwiek wcześniej. Pojawiają się maszyny o coraz większej liczbie rdzeni, a języki funkcyjne pozwalają łatwo wykorzystać możliwości takiego sprzętu. Programowanie funkcyjne omawiamy w rozdziale 4., „Programowanie funkcyjne”.
- ◆ W językach funkcyjnych niewygodnie zarządza się stanem, jeśli musi się on zmieniać. Clojure udostępnia mechanizmy do obsługi zmiennego stanu za pomocą programowej pamięci transakcyjnej i referencji, agentów, atomów i wiązania dynamicznego.

- ◆ W wielu językach funkcyjnych typy są określane statycznie. W języku Clojure stosuje się dynamiczne określanie typów, co ułatwia zadanie programistom poznającym dopiero programowanie funkcyjne.
- ◆ Wywoływanie w języku Clojure kodu Javy *nie* odbywa się w modelu funkcyjnym. Przy korzystaniu z Javy wkraczamy w znany świat zmiennych obiektów. Zapewnia to wygodę początkującym, którzy uczą się programowania funkcyjnego, a także pozwala w razie potrzeby zrezygnować z podejścia funkcyjnego. Wywoływanie kodu Javy opisujemy w rozdziale 9., „Sztuczki z Javą”.

Mechanizmy zarządzania zmianą stanu w języku Clojure umożliwiają pisanie programów współbieżnych bez bezpośredniego korzystania z blokad i uzupełniają podstawowy funkcjonalny rdzeń języka.

Clojure upraszcza programowanie współbieżne

Mechanizmy programowania funkcyjnego dostępne w Clojure ułatwiają pisanie kodu bezpiecznego ze względu na wątki. Ponieważ niezmiennie struktury danych *nigdy* się nie zmieniają, nie występuje zagrożenie uszkodzeniem danych w wyniku działania innych wątków.

Jednak obsługa współbieżności w Clojure wykracza poza mechanizmy programowania funkcyjnego. Jeśli potrzebne są referencje do zmiennych danych, Clojure zabezpiecza je za pomocą programowej pamięci transakcyjnej (ang. *software transactional memory* — STM). STM służy do tworzenia kodu bezpiecznego ze względu na wątki i jest rozwiązaniem wyższego poziomu niż blokady z Javy. Zamiast stosować podatne na błędy strategie blokowania danych, można zabezpieczyć współużytkowany stan za pomocą transakcji. Jest to dużo lepsze podejście, ponieważ wielu programistów dobrze zna transakcje z uwagi na doświadczenie w korzystaniu z baz danych.

Poniższy kod tworzy działającą, bezpieczną ze względu na wątki bazę danych z kontami przechowywaną w pamięci:

```
(def accounts (ref #{}))  
(defrecord Account [id balance])
```

Funkcja `ref` tworzy zabezpieczoną za pomocą transakcji referencję do bieżącego stanu bazy danych. Aktualizowanie stanu jest niezwykle proste. Poniżej pokazujemy, jak dodać do bazy nowe konto:

```
(dosync  
  (alter accounts conj (->Account "CLJ" 1000.00)))
```


Instrukcja `dosync` powoduje aktualizowanie bazy `accounts` w transakcji. Rozwiązanie to zapewnia bezpieczeństwo ze względu na wątki i jest łatwiejsze w stosowaniu od blokad. Dzięki transakcjom nigdy nie trzeba martwić się o to, które obiekty i w jakiej kolejności należy zablokować. Podejście transakcyjne sprawdza się lepiej także w niektórych standardowych zastosowaniach. Przykładowo: w modelu tym wątki wczytujące dane nigdy nie muszą blokować danych.

Choć przedstawiony przykład jest bardzo prosty, technika jest ogólna i sprawdza się także w praktyce. Więcej informacji o współbieżności i pamięci STM w języku Clojure znajdziesz w rozdziale 5., „Stan”.

Wykorzystanie maszyny JVM w Clojure

Clojure zapewnia przejrzysty, prosty i bezpośredni dostęp do Javy. W kodzie można bezpośrednio wywołać metody z dowolnego interfejsu API Javy:

```
(System/getProperties)
-> {java.runtime.name=Java(TM) SE Runtime Environment
... i wiele innych ...}
```

Clojure obejmuje wiele składniowych mechanizmów do wywoływania kodu w Javie. Nie omawiamy tu ich szczegółowo (zobacz podrozdział 2.5, „Wywoływanie kodu w Javie”), warto jednak wspomnieć, że w Clojure występuje mniej kropek i *mniej nawiasów* niż w analogicznym kodzie Javy:

```
// Java
"hello".getClass().getProtectionDomain()

; Clojure
(.. "hello" getClass getProtectionDomain)
```

Clojure udostępnia proste funkcje do implementowania interfejsów Javy i tworzenia klas pochodnych od klas Javy. Ponadto wszystkie funkcje języka Clojure obejmują implementację interfejsów `Callable` i `Runnable`. Dlatego można łatwo przekazać poniższą funkcję anonimową do konstruktora klasy `Thread` Javy:

```
(.start (new Thread (fn [] (println "Witaj" (Thread/currentThread))))
-> Witaj #<Thread Thread[Thread-0,5,main]>
```

Dziwne dane wyjściowe wynikają ze sposobu wyświetlania informacji o obiektach Javy w języku Clojure. `Thread` to nazwa klasy danego obiektu, a człon `Thread[Thread-0,5,main]` to efekt wywołania metody `toString` obiektu.

(Zauważ, że w przedstawionym kodzie nowy wątek działa aż do zakończenia pracy, natomiast jego dane wyjściowe mogą w dziwny sposób przeplatać się z wierszami zachęty środowiska REPL. Nie jest to problem z językiem Clojure, a jedynie wynik zapisywania danych do strumienia wyjścia przez więcej niż jeden wątek).

Ponieważ składnia do wywoływania kodu Javy w Clojure jest przejrzysta i prosta, zwykle używa się Javy bezpośrednio, zamiast ukrywać kod w tym języku za charakterystycznymi dla Lispa nakładkami.

Poznałeś już kilka powodów do stosowania języka Clojure. Pora przystąpić do pisania kodu.

1.2. Szybkie wprowadzenie do programowania w Clojure

Aby uruchomić środowisko języka Clojure i kod z tej książki, potrzebujesz dwóch rzeczy. Oto one:

- ◆ *Środowisko uruchomieniowe Javy*. Pobierz⁴ i zainstaluj Javę w wersji 5. lub nowszej. W wersji 6. znacznie poprawiono wydajność i system informowania o wyjątkach, dlatego warto stosować właśnie ją.
- ◆ *Leiningen*⁵. Leiningen to narzędzie do zarządzania zależnościami i uruchamiania operacji na kodzie. Jest to także najpopularniejsze w świecie języka Clojure narzędzie do wykonywania tych zadań.

Leiningen posłuży do zainstalowania języka Clojure i wszystkich elementów wymaganych w przykładowym kodzie. Jeśli masz już zainstalowane narzędzie Leiningen, wiesz zapewne, jak z niego korzystać. Jeżeli jeszcze nie masz potrzebnej wiedzy, zapoznaj się z krótkim wprowadzeniem ze strony narzędzia w serwisie GitHub⁶. Znajdziesz tam instrukcje dotyczące instalacji, a także podstawowe informacje na temat użytkowania Leiningena. Nie musisz jednak uczyć się wszystkiego, ponieważ w książce opisujemy polecenia potrzebne do uruchomienia przykładów.

W trakcie pracy z książką korzystaj z języka Clojure w wersji właściwej dla przykładowego kodu. Po przeczytaniu książki możesz zastosować się do instrukcji z ramki „Samodzielne budowanie języka Clojure” i zbudować aktualną wersję języka.

⁴ <http://www.oracle.com/technetwork/java/javase/downloads/index.html>

⁵ <http://github.com/technomancy/leiningen>

⁶ <http://github.com/technomancy/leiningen>

W punkcie „Pobieranie przykładowego kodu”, znajdziesz instrukcje dotyczące pobierania przykładowego kodu. Po ściągnięciu przykładowego kodu trzeba użyć Leiningena do pobrania zależności. W katalogu głównym z kodem wywołaj następującą instrukcję:

```
lein deps
```

Samodzielne budowanie języka Clojure

Możliwe, że chcesz zbudować język Clojure na podstawie kodu źródłowego, aby uzyskać nowe funkcje i wprowadzić poprawki błędów. Można zrobić to w następujący sposób:

```
git clone git://github.com/clojure/clojure.git
cd clojure
mvm package
```

Przykładowy kod jest regularnie aktualizowany pod kątem nowych rozwiązań wprowadzanych w języku Clojure. Zapoznaj się z plikiem *README* w przykładowym kodzie, aby sprawdzić numer najnowszej wersji, dla której sprawdzono kod.

Zależności są pobierane i umieszczane w odpowiednim miejscu. Możesz przetestować zainstalowane narzędzia przez przejście do katalogu z przykładowym kodem i uruchomienie środowiska REPL języka Clojure. Leiningen obejmuje skrypt uruchomieniowy środowiska REPL, który wczytuje język Clojure wraz z zależnościami potrzebnymi w dalszych rozdziałach.

```
lein repl
```

Po udanym uruchomieniu środowiska REPL powinien pojawić się wiersz zachęty z tekstem `user=>`:

```
Clojure
user=>
```

Teraz jesteś gotowy do wyświetlenia tekstu „Witaj, świecie”.

Korzystanie ze środowiska REPL

Aby pokazać, jak korzystać ze środowiska REPL, tworzymy kilka wersji kodu wyświetlającego tekst „Witaj, świecie”. Najpierw wpisz kod (`println "Witaj, świecie"`) w wierszu zachęty środowiska REPL.

```
user=> (println "Witaj, świecie")
```

```
-> Witaj, świecie
```

Drugi wiersz, Witaj, świecie, to żądane dane wyjściowe z konsoli.

Teraz umieścimy kod w funkcji, która potrafi „zwracać się” do użytkownika po imieniu.

```
(defn hello [name] (str "Witaj, " name))
-> #'user/hello
```

Rozłożmy ten kod na fragmenty. Oto one:

- ◆ `defn` służy do definiowania funkcji;
- ◆ `hello` to nazwa funkcji;
- ◆ funkcja `hello` przyjmuje jeden argument, `name`;
- ◆ `str` to wywołanie funkcji łączącej dowolną listę argumentów w łańcuch znaków;
- ◆ `defn`, `hello`, `name` i `str` to *symbole*, czyli nazwy prowadzące do różnych elementów; dozwolone symbole opisujemy w punkcie „Symbole”.

Przyjrzyj się zwracanej wartości, `#'user/hello`. Przedrostek `#'` oznacza, że funkcję zapisano w zmiennej języka Clojure, a `user` to *przestrzeń nazw*, w której znajduje się ta funkcja. Jest to domyślna przestrzeń nazw w środowisku REPL, odpowiadająca domyślnemu pakietowi w Javie. Na razie zmienne i przestrzenie nazw nie mają znaczenia. Omawiamy je w podrozdziale 2.4, „Zmienne, wiązanie i przestrzenie nazw”.

Teraz można wywołać funkcję `hello` i przekazać do niej imię.

```
user=> (hello "Janku")
-> "Witaj, Janku"
```

Jeśli środowisko REPL znajduje się w dziwnym stanie, najłatwiej zamknąć je za pomocą kombinacji klawiszy `Ctrl+C` w systemie Windows lub `Ctrl+D` w systemach uniksowych, a następnie ponownie uruchomić.

Specjalne zmienne

Środowisko REPL obejmuje szereg przydatnych zmiennych specjalnych. W czasie pracy w środowisku REPL wyniki obliczania trzech ostatnich wyrażeń znajdują się w specjalnych zmiennych `*1`, `*2` i `*3`. Pozwala to na wygodną pracę w modelu iteracyjnym. Spróbujmy połączyć kilka powitań.

```
user=> (hello "Janku")
-> "Witaj, Janku"
```

```
user=> (hello "Clojure")
-> "Witaj, Clojure"
```

Teraz można zastosować specjalne zmienne do połączenia wyników ostatnich instrukcji.

```
(str *1 " i " *2)
-> "Witaj, Clojure i Witaj, Janku"
```

Pełnienie błędu w środowisku REPL prowadzi do zgłoszenia wyjątku Javy (z uwagi na zwięzłość szczegóły pomijamy). Niedozwolone jest na przykład dzielenie przez zero.

```
user=> (/ 1 0)
-> ArithmeticException Divide by zero clojure.lang.Numbers.divide
```

Tu problem jest oczywisty, jednak czasem jest bardziej skomplikowany i potrzebujemy szczegółowego stosu wywołań. W specjalnej zmiennej `*e` znajdują się informacje o ostatnim wyjątku. Ponieważ wyjątki w Clojure są wyjątkami Javy, można wyświetlić stos wywołań za pomocą instrukcji `pst` (od ang. *print stacktrace*, czyli wyświetl stos wywołań)⁷.

```
user=> (pst)
-> ArithmeticException Divide by zero
| clojure.lang.Numbers.divide
| sun.reflect.NativeMethodAccessorImpl.invoke0
| sun.reflect.NativeMethodAccessorImpl.invoke
| sun.reflect.DelegatingMethodAccessorImpl.invoke
| java.lang.reflect.Method.invoke
| clojure.lang.Reflector.invokeMatchingMethod
| clojure.lang.Reflector.invokeStaticMethod
| user/eval1677
| clojure.lang.Compiler.eval
| clojure.lang.Compiler.eval
| clojure.core/eval
```

Współdziałanie z Javą omawiamy w rozdziale 9., „Sztuczki z Javą”.

Jeśli blok kodu jest zbyt długi, aby można go wygodnie wpisać w środowisku REPL, umieść kod w pliku, a następnie wczytaj ten plik w środowisku. Możesz użyć ścieżki bezwzględnej lub podać ją względem miejsca uruchomienia środowiska REPL.

```
; Zapisz kod w pliku temp.clj, a następnie wywołaj instrukcję:
user=> (load-file "temp.clj")
```

REPL to znakomite środowisko do wypróbowywania pomysłów i otrzymywania natychmiastowych informacji zwrotnych. Aby jak najlepiej wykorzystać książkę, w trakcie jej lektury nie zamykaj środowiska REPL.

⁷ Instrukcja `pst` jest dostępna tylko w Clojure 1.3.0 i nowszych wersjach.

Dodawanie stanu współużytkowanego

Funkcja `hello` z poprzedniego przykładu to *czysta* funkcja, czyli taka, której działanie nie ma efektów ubocznych. Czyste funkcje łatwo się pisze i testuje. Są także łatwe do zrozumienia, dlatego w wielu sytuacjach warto je stosować.

Jednak w większości programów występuje współużytkowany stan, a do zarządzania nim służą funkcje typu *impure* (czyli takie, które nie są czyste). Dodajmy do funkcji `hello` mechanizm śledzenia liczby użytkowników. Najpierw trzeba utworzyć strukturę danych do przechowywania tej liczby. Użyjmy do tego zbioru.

```
#{}
-> #{}

```

`#{}` to literał oznaczający pusty zbiór. Potrzebna jest też operacja `conj`.

```
(conj coll item)
```

Instrukcja `conj` (od ang. *conjoin*, czyli łączyć) tworzy nową kolekcję z dodanym elementem. Dołączmy element do zbioru, aby upewnić się, że powstaje nowy zbiór.

```
(conj #{} "Janku")
-> #{"Janku"}

```

Tworzenie nowych zbiorów jest już możliwe. Pora opracować sposób na sprawdzanie *aktualnego* zbioru użytkowników. Clojure udostępnia kilka służących do tego *typów referencyjnych*. Najprostszym typem referencyjnym jest `atom`.

```
(atom initial-state)
```

Aby określić nazwę atomu, należy użyć instrukcji `def`.

```
(def symbol initial-value?)
```

Instrukcja `def` przypomina polecenie `defn`, ale jest ogólniejsza. Przy jej użyciu można definiować funkcje *lub* dane. Użyjmy słowa `atom` do utworzenia atomu i instrukcji `def` do powiązania atomu z nazwą `visitors`.

```
(def visitors (atom #{}))
-> #'user/visitors

```

Aby zaktualizować referencję, trzeba użyć funkcji, na przykład `swap!`.

```
(swap! r update-fn & args)
```

Funkcja `swap!` przeprowadza operację `update-fn` na referencji `r` i w razie potrzeby używa przy tym opcjonalnych argumentów `args`. Spróbujmy wstawić użytkownika do kolekcji `visitors`, używając do aktualizowania funkcji `conj`.

```
(swap! visitors conj "Janku")
-> #{"Janku"}

```

`atom` to tylko jeden z kilku typów referencyjnych dostępnych w języku Clojure. Wybór odpowiedniego typu referencyjnego nie jest prosty (zagadnienie to omawiamy w rozdziale 5., „Stan”).

W dowolnym momencie można sprawdzić zawartość pamięci, do której prowadzi referencja. Służy do tego instrukcja `deref` lub jej krótszy odpowiednik, `@`.

```
(deref visitors)
-> #{"Janku"}
```

```
@visitors
-> #{"Janku"}
```

Teraz można zbudować nową, bardziej rozbudowaną wersję funkcji `hello`.

```
src/examples/introduction.clj
```

```
(defn hello
  "Wyświetla powitanie na wyjściu, używając nazwy użytkownika.
  Potrafi stwierdzić, że korzystałeś już z programu."
  [username]
  (swap! visitors conj username)
  (str "Witaj, " username))
```

Teraz sprawdźmy, czy użytkownicy są poprawnie zapisywani w pamięci.

```
(hello "Marku")
-> "Witaj, Marku"
```

```
@visitors
-> #{"Jacku" "Janku" "Marku"}
```

Na Twoim komputerze lista użytkowników będzie prawdopodobnie inna. Na tym właśnie polega problem ze stanem. Efekty są różne w zależności od tego, kiedy zaszły dane zdarzenia. Zrozumieć funkcję można na podstawie jej analizy. Aby zrozumieć stan, trzeba poznać całą historię działania programu.

Jeśli to możliwe, unikaj przechowywania stanu. Jeżeli jest to niemożliwe, dbaj o to, aby stanem można było zarządzać. Używaj do tego typów referencyjnych, na przykład atomów. Atomy (i wszystkie inne typy referencyjne w Clojure) są bezpieczne przy korzystaniu z wielu wątków i procesorów. Co lepsze, zapewnienie bezpieczeństwa nie wymaga stosowania blokad, które bywają skomplikowane w użyciu.

Na tym etapie powinieneś umieć już wprowadzać krótkie fragmenty kodu w środowisku REPL. Wprowadzanie większych porcji kodu odbywa się podobnie. Także biblioteki języka Clojure można wczytywać i uruchamiać z poziomu środowiska REPL. Dalej pokazujemy, jak to zrobić.

1.3. Biblioteki języka Clojure

Kod języka Clojure jest umieszczony w *bibliotekach*. Każda biblioteka języka Clojure znajduje się w *przestrzeni nazw*, która jest odpowiednikiem pakietu Javy. Bibliotekę języka Clojure można wczytać za pomocą instrukcji `require`.

```
(require quoted-namespace-symbol)
```

Jeśli programista żąda biblioteki o nazwie `clojure.java.io`, Clojure szuka pliku o nazwie `clojure/java/io.clj` w ścieżce ze zmiennej `CLASSPATH`. Zobaczmy, jaki jest tego efekt.

```
user=> (require 'clojure.java.io)
-> nil
```

Początkowy pojedynczy apostrof (`'`) jest niezbędny i służy do *dosłownego podawania* (ang. *quoting*) nazwy biblioteki (podawanie nazw omawiamy w podrozdziale 2.2, „Makra odczytu”). Zwrócona wartość `nil` oznacza powodzenie. Przy okazji sprawdź, czy możesz wczytać przykładowy kod do tego rozdziału (bibliotekę `examples.introduction`).

```
user=> (require 'examples.introduction)
-> nil
```

Biblioteka `examples.introduction` obejmuje implementację generowania liczb Fibonacciego. W językach funkcyjnych jest to tradycyjny program typu „Witaj, świecie”. Liczby Fibonacciego omawiamy szczegółowo w podrozdziale 4.2, „»Leniwe« podejście”. Na razie upewnij się, że możesz uruchomić przykładową funkcję `fibs`. Wprowadź poniższy wiersz kodu w środowisku REPL, a otrzymasz 10 pierwszych liczb Fibonacciego.

```
(take 10 examples.introduction/fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

Jeśli otrzymałeś 10 pierwszych liczb Fibonacciego (wymienionych powyżej), poprawnie zainstalowałeś przykładowy kod z książki.

Wszystkie przykłady sprawdziliśmy za pomocą testów jednostkowych (testy znajdują się w katalogu `examples/test`). Samych testów nie omawiamy w książce, jednak mogą okazać się przydatnym źródłem wiedzy. Aby uruchomić testy jednostkowe, użyj instrukcji `lein test`.

Instrukcje `require` i `use`

Po zażądaniu biblioteki języka Clojure za pomocą instrukcji `require` elementy z biblioteki trzeba wskazywać za pomocą pełnej nazwy. Zamiast nazwy `fibs` trzeba użyć określenia `examples.introduction/fibs`. Uruchom drugi egzemplarz środowiska REPL⁸ i wprowadź poniższy kod.

```
(require 'examples.introduction)
-> nil
```

```
(take 10 examples.introduction/fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

Wprowadzanie pełnych nazw szybko staje się kłopotliwe. Możesz użyć instrukcji `refer` dla przestrzeni nazw i odwzorować wszystkie nazwy z tej przestrzeni na bieżącą przestrzeń nazw.

```
(refer quoted-namespace-symbol)
```

Wywołaj instrukcję `refer` dla przestrzeni `examples.introduction` i sprawdź, czy możesz bezpośrednio wywołać funkcję `fibs`.

```
(refer 'examples.introduction)
-> nil
```

```
(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

Wygodna funkcja `use` pozwala wykonać instrukcje `require` i `refer` w jednym kroku.

```
(use quoted-namespace-symbol)
```

W nowym środowisku REPL wprowadź następujące instrukcje.

```
(use 'examples.introduction)
-> nil
```

```
(take 10 fibs)
-> (0 1 1 2 3 5 8 13 21 34)
```

W trakcie pracy z przykładowym kodem z książki możesz wywołać instrukcję `require` lub `use` z opcją `:reload`, aby wymusić ponowne wczytanie biblioteki.

```
(use :reload 'examples.introduction)
-> nil
```

Opcja `:reload` jest przydatna, jeśli wprowadzasz zmiany i chcesz sprawdzić ich efekt bez ponownego uruchamiania środowiska REPL.

⁸ Otwarcie nowego środowiska REPL zapobiega konfliktom nazw między utworzonym wcześniej kodem a funkcjami o tych samych nazwach z przykładowego kodu. W praktyce nie stanowi to problemu. Zagadnienie to omawiamy w punkcie „Przestrzeń nazw”.

Znajdowanie dokumentacji

Potrzebna dokumentacja często jest dostępna bezpośrednio w środowisku REPL. Najprostszą funkcją pomocniczą⁹ jest `doc`.

`(doc name)`

Użyjmy funkcji `doc` do wyświetlenia dokumentacji funkcji `str`.

```
user=> (doc str)
```

```
-----
clojure.core/str
([] [x] [x & ys])
With no args, returns the empty string. With one arg x, returns
x.toString(). (str nil) returns the empty string. With more than
one arg, returns the concatenation of the str values of the args.
```

Pierwszy wiersz danych wyjściowych funkcji `doc` obejmuje pełną nazwę sprawdzanej funkcji. W drugim znajdują się argumenty generowane bezpośrednio w kodzie. Wybrane często stosowane nazwy argumentów i ich zastosowanie omawiamy w ramce „Zwyczajowe nazwy parametrów”. Dalsze wiersze obejmują *łańcuch znaków dokumentacji*, jeśli jest on podany w definicji funkcji.

Zwyczajowe nazwy parametrów

Łańcuchy znaków dokumentacji w funkcjach `reduce` i `areduce` obejmują szereg krótkich nazw parametrów. Oto niektórych z tych nazw i sposoby ich stosowania.

Parametr	Zastosowanie
<code>a</code>	Tablica Javy
<code>agt</code>	Agent
<code>coll</code>	Kolekcja
<code>expr</code>	Wyrażenie
<code>f</code>	Funkcja
<code>idx</code>	Indeks
<code>r</code>	Referencja
<code>v</code>	Wektor
<code>val</code>	Wartość

Nazwy mogą wydawać się krótkie, jednak jest tak nie bez powodu — „dobre” nazwy często są już „zajęte” przez funkcje języka Clojure! Użycie dla parametrów nazw identycznych z nazwami funkcji jest dopuszczalne, ale uznaje się to za oznakę złego stylu. Parametr zastąpi wtedy funkcję, dlatego jest ona niedostępna, kiedy parametr znajduje się w zasięgu programu. Dlatego nie należy nazywać referencji `ref`, agentów `agent`, a liczników — `count`, ponieważ są to nazwy funkcji.

⁹ Tak naprawdę `doc` to makro języka Clojure.

Łańcuch znaków dokumentacji można dodać do funkcji przez umieszczenie go bezpośrednio po jej nazwie.

```
src/examples/introduction.clj
```

```
(defn hello
  "Wyświetla powitanie na wyjściu, używając nazwy użytkownika. "
  [username]
  (println (str "Witaj, " username)))
```

Czasem nie znasz nazwy elementu, którego dokumentacji potrzebujesz. Funkcja `find-doc` wyszukuje informacje o wszystkich elementach, dla których dane wyjściowe funkcji `doc` pasują do przekazanego wyrażenia regularnego lub łańcucha znaków.

```
(find-doc s)
```

Za pomocą funkcji `find-doc` można sprawdzić, w jaki sposób Clojure skraca kolekcje.

```
user=> (find-doc "reduce")
-----
clojure/areduce
([a idx ret init expr])
Macro
... Szczegóły pominięto ...
-----
clojure/reduce
([f coll] [f val coll])
... Szczegóły pominięto ...
```

Funkcja `reduce` pozwala w skrócony sposób stosować operacje do kolekcji języka Clojure. Omawiamy ją w punkcie „Przekształcanie sekwencji”. Funkcja `areduce` współdziała z tablicami Javy, a opisujemy ją w punkcie „Korzystanie z kolekcji Javy”.

Duża część języka Clojure jest napisana w nim samym, dlatego lektura jego kodu źródłowego to pouczające zadanie. Kod źródłowy funkcji języka Clojure można wyświetlić za pomocą instrukcji `source` z biblioteki `repl`.

```
(clojure.repl/source symbol)
```

Wyświetlmy kod źródłowy prostej funkcji `identity`.

```
(use 'clojure.repl)
(source identity)

-> (defn identity
    "Returns its argument."
    {:added "1.0"
     :static true}
    [x] x)
```

Oczywiście, można też używać interfejsu API Reflection Javy. Metody `class`, `ancestors`, `instance?` i podobne pozwalają sprawdzić model obiektowy Javy i informują na przykład o tym, że kolekcje języka Clojure są jednocześnie kolekcjami Javy.

```
(ancestors (class [1 2 3]))
```

```
-> #{clojure.lang.ILookup clojure.lang.Sequential
      java.lang.Object clojure.lang.Indexed
      java.lang.Iterable clojure.lang.IObj
      clojure.lang.IPersistentCollection
      clojure.lang.IPersistentVector clojure.lang.AFn
      java.lang.Comparable java.util.RandomAccess
      clojure.lang.Associative
      clojure.lang.APersistentVector clojure.lang.Counted
      clojure.lang.Reversible clojure.lang.IPersistentStack
      java.util.List clojure.lang.IEditableCollection
      clojure.lang.IFn clojure.lang.Seqable
      java.util.Collection java.util.concurrent.Callable
      clojure.lang.IMeta java.io.Serializable java.lang.Runnable}
```

Internetową dokumentację interfejsu API języka Clojure znajdziesz na stronie <http://clojure.github.com/clojure>. W ramce widocznej w prawej części tej strony znajdują się odnośniki do wszystkich funkcji i makr. Po lewej stronie umieszczono odnośniki do artykułów na temat różnych cech języka Clojure.

1.4. Podsumowanie

Właśnie zakończyłeś szybki przegląd języka Clojure. Poznałeś dającą duże możliwości składnię tego języka i jego związki z Lispem, a także zobaczyłeś, jak łatwe jest wywoływanie w Clojure kodu Javy.

Uruchomiłeś język Clojure w swoim środowisku, a także napisałeś w środowisku REPL krótkie programy ilustrujące programowanie funkcyjne i służący do obsługi stanu model referencji. Pora przyjrzeć się całemu językowi.

Skorowidz

A

agenty, 158
 bieżąca wartość, 159
 sprawdzanie poprawności, 160
 transakcje, 161
 tworzenie, 158
 wykrywanie błędów, 160
algebra relacji, 110
Apache Ant, 244
atomy, 157
 dereferencja, 157
 tworzenie, 157
 ustawienie wartości, 157

B

biblioteki, 20, 40
 dosłowne podawanie nazwy, 40
 drzewo właściwości systemowych, 238
 znajdowanie dokumentacji, 42

C

Clojure, 15, 23
 aktualizacja, 261
 aspekty, 23
 biblioteki, 20, 40
 drzewo właściwości systemowych, 238
 inspector, 238

 Lazytest, 286
 math.combinatorics, 275
 Midje, 286
 sekwencje, 87
 test, 239
 test.generative, 278
cechy języka, 24, 209
czytnik, 46
 makra odczytu, 55
edytory kodów, 297
funkcje, 56
 czyste, 38
 dodawanie sugestii typów, 253
 impure, 38
 wywołanie, 56
 wyższego rzędu, 25
Java, 15, 33, 243
 dostęp, 243
 interfejsy, 183, 255
 javadoc, 70
 kompilacja AOT, 248
 konstrukcja new, 68
 korzystanie z kolekcji, 258
 mechanizm wywołań zwrotnych, 256
 parsery SAX, 255
 środowisko uruchomieniowe, 34
 tworzenie klas, 255

Clojure

Java

- tworzenie pośredników, 255
- tworzenie tablic, 258
- wykorzystanie możliwości, 247
- wywoływanie kodu, 68
- wywoływanie metody, 69
- język funkcyjny, 30
 - cechy, 30
- konstrukcje składniowe, 24, 46
- Leiningen, 34, 261
- liczby całkowite, 248
 - operatory, 248
- Lisp, 15, 24, 27
- makra, 28, 201
 - taksonomia, 216
- maszyny JVM, 29
- metadane, 77
- model czasu, 24
- niezmienne struktury danych, 27
- obsługa wyjątków, 244
 - kontrolowane wyjątki, 245
 - porządkowanie zasobów, 245
 - reagowanie na wyjątki, 247
- optymalizowanie wydajności, 250
 - dodawanie sugestii typów, 253
 - używanie typów prostych, 250
- pobieranie zależności, 261
- połączenie z witryną, 261
- programowanie, 16
 - funkcyjne, 18, 24, 115
 - pętle, 74
 - reguły, 120
 - współbieżne, 32
- protokoły, 24, 179, 184
- przestrzeń nazw, 36, 61, 65
- rejestracja informacji, 264
- rekordy, 193
- rozkładanie struktury, 63
 - możliwości, 65
- sekwencje, 29, 81
 - biblioteka, 87
 - cechy, 83

Java, 98

- manipulowanie, 197
- wymuszanie realizacji, 97
- wyrażenia listowe, 95
- stan, 147
 - model aktualizacji, 163
 - model funkcyjny, 149
 - model referencyjny, 149
- sterowanie przepływem, 70
 - instrukcje, 70
 - makra, 71
 - rekurencja, 72
- środowisko REPL, 35
 - zmienne specjalne, 36
- tożsamość, 147
 - typy referencyjne, 147
- tworzenie aplikacji, 269
 - gra Clojurebreaker, 270
- typy danych, 179, 188
 - cechy, 188
- typy referencyjne, 38
 - atom, 38
- wartość, 147
- wiązania, 61
- wielometody, 225
- współbieżność, 18, 32, 148
 - powody stosowania, 148
 - sytuacja wyścigu, 149
 - zakleszczenie, 149
- zmienne, 36, 61
 - cechy, 62

D

- definicja pustego łańcucha znaków, 26
- duck typing, 250

F

- funkcje, 56
 - anonimowe, 58
 - konstrukcja, 59
 - powody tworzenia, 58
 - stosowanie, 61
 - unikanie, 223

bez sprawdzania przepelnienia, 251
częściowe wywołanie, 134
czyste, 38, 116
 niezmienne dane, 116
dodawanie sugestii typów, 253
efekty uboczne, 71
funkcje wyższego rzędu, 25
leniwe, 127
liczba argumentów, 57
listy, 105
łańcuchy znaków dokumentacji, 42
odwzorowania, 53, 106
 tworzenie, 108
operatory matematyczne, 47
predykaty, 52, 56
przejrzyste referencyjnie, 118
 memoizacja, 119
rozwińnięcie funkcji, 135
 implementacja, 135
sekwencje, 83
 filtrowanie, 91
 predykaty, 92
 przekształcanie, 93
 tworzenie, 88
 wyrażenia regularne, 100
słowa kluczowe, 54
wektory, 105
wiązania, 62
 zasięg leksykalny, 63
wywołanie, 47, 56
 notacja przedrostkowa, 47
 notacja wrostkowa, 47
zbiory, 109
złożone, 134

H

hermetyzacja, 119
 efekty uboczne, 120
Heroku, 292
 biblioteka, 293
 git init, 293
 git push, 294
 plik Procfile, 292

polecenie heroku, 293
rejestracja konta, 292
repozytorium git, 293

I

instrukcje
 add-points, 171
 agent-errors, 160
 alias, 233
 alter, 152
 assoc, 171
 atom, 157
 binding, 164
 clear-agent-errors, 160
 commute, 154
 concat, 210
 cond, 227
 condp, 182
 conj, 38, 86
 cons, 83, 132, 171
 def, 38, 61, 218
 defmacro, 203
 defmethod, 228
 defmulti, 175, 228
 defn, 57
 defonce, 133
 defrecord, 27, 54
 deref, 39, 150
 derive, 236
 dir, 279
 do, 71, 219
 doall, 97
 dosync, 33, 150
 faux-curry, 135
 file-seq, 101
 first, 76, 83
 fn, 58
 for, 75
 force, 221
 if, 70, 202
 import, 67, 199
 in-ns, 66, 281
 inspect-tree, 238

instrukcje

- lazy-cat, 129
- lazy-seq, 142
- lein deps, 261
- lein test, 40
- let, 212
- line-seq, 102
- loop, 72
- loop/recur, 72
- macroexpand, 207
- map, 219
- memoize, 165
- next-counter, 155
- partial, 134
- partition, 132
- prefer-method, 232
- println, 116, 203
- project, 112
- proxy, 176
- pst, 37
- recur, 72, 120, 130
- ref, 156
- refer, 41
- ref-set, 150
- require, 40, 41, 279
- reset!, 157
- rest, 83
- score print-table, 276
- select, 112
- send, 159
- send-off, 161
- seq, 101, 258
- set, 90
- set!, 167, 254
- source, 43
- start, 163
- swap!, 158
- take-while, 91
- trampoline, 139
- unless, 202
- use, 41, 66
- var, 62
- with-open, 102

J

- Java, 15, 16, 243
 - interfejsy, 183
 - wady, 183
 - zalety, 183
- klasy
 - BigDecimal, 249
 - BigInteger, 249
 - Character, 50
 - ChunkedSeq, 84
 - Person, 26
 - Random, 68
 - StringUtils, 74
 - Thread, 33
 - WidgetFactory, 205
- metody egzemplarza, 204
- obsługa, 245
- obsługa wyjątków, 244
 - kontrolowane wyjątki, 244
 - porządkowanie zasobów, 245
- sekwencje, 98
 - kolekcje sekwencyjne, 98
- wywoływanie kodu, 68

K

- kod gry Snake, 169
 - interfejs GUI, 169, 175
 - aktualizowanie, 175
 - defmulti, 175
 - fill-point, 175
 - game, 176
 - game-panel, 175
 - paint, 175
 - proxy, 176
 - tworzenie nowej gry, 176
 - wyświetlanie węża i jabłka, 175
- model funkcyjny, 169
 - add-points, 170, 171
 - assoc, 171
 - cons, 171
 - dodawanie punktów, 170
 - eats?, 172

- head-overlaps-body?, 172
 - lose?, 172
 - move, 171
 - point-to-screen-rect, 171
 - ruch węża, 171
 - sprawdzanie warunków zwycięstwa, 172
 - turn, 173
 - tworzenie nowego jabłka, 171
 - tworzenie węża, 171
 - tworzenie zestawu stałych, 169
 - win?, 172
 - wykrywanie zetknięcia, 172
 - zjadanie jabłka, 172
 - zmiana kierunku węża, 173
 - model zmiennego stanu, 169, 173
 - reset-game, 173
 - update-direction, 174
 - update-positions, 174
 - wydłużanie węża, 174
 - zmiany pozycji węża i jabłka, 173
 - konstrukcje składniowe, 24
 - liczba, 46
 - BigDecimal, 48
 - BigInt, 48
 - całkowita, 48
 - Ratio, 48
 - wektor, 47
 - zmiennoprzecinkowa, 48
 - lista, 46, 47
 - wywoływanie funkcji, 47
 - łańcuch znaków, 46, 49
 - wywołanie, 50
 - odwzorowania, 46, 52
 - sekwencje, 85
 - rekordy, 52
 - wywoływanie, 54
 - słowo kluczowe, 46, 53
 - symbol, 46, 49
 - wartość logiczna, 46
 - reguły działania, 51
 - wartość nil, 46
 - wektory, 46
 - sekwencje, 84
 - zbiory, 46
 - sekwencje, 85
 - znak, 46
- L**
- Leiningen, 34, 261
 - pobieranie zależności, 35
 - wtyczka lein-noir, 287
 - leniwe sekwencje, 121, 125
 - moment realizacji, 127
 - Lisp, 15, 24
- M**
- makra, 28, 201
 - amap, 260
 - and, 207, 216
 - areduce, 260
 - assert, 222
 - bad-unless, 207
 - bench, 212
 - binding, 164, 221
 - chain, 209
 - comment, 217
 - cond, 30
 - czas kompilacji, 203
 - czas rozwijania makra, 203
 - declare, 137, 218
 - definterface, 183
 - defpartial, 288
 - defprotocol, 185
 - defrecord, 28, 193
 - defstruct, 218
 - deftype, 189
 - delay, 220
 - dosync, 221
 - dotimes, 250
 - extend-protocol, 186
 - extend-type, 186
 - for, 95
 - import-static, 220
 - is, 239
 - język szablonów, 210

makra

- konstrukcje specjalne, 215
 - lazy-seq, 125
 - let, 221
 - letfn, 124
 - manipulowanie listą, 210
 - ns, 67
 - obsługa kilku konstrukcji, 208
 - obsługa szablonów, 211
 - przechwytywanie symboli, 213
 - splicing unquote, 211
 - unquote, 211
 - przetwarzanie, 203
 - reguły stosowania, 202
 - reify, 198
 - rozwijanie, 206
 - rekurencyjne, 207
 - tworzenie, 212
 - sprawdzanie, 206
 - sterowanie przebiegiem programu, 202
 - tablice Javy, 260
 - taksonomia, 214
 - kategorie, 216
 - time, 212, 221
 - tworzenie, 203
 - tworzenie nazw, 212
 - nazwy lokalne, 214
 - tworzenie zmiennych, 218
 - unikanie funkcji anonimowych, 223
 - unless, 203
 - upraszczanie, 209
 - wartościowanie argumentów, 203, 206
 - nakładki, 221
 - odraczanie, 220
 - warunkowe, 216
 - when, 208
 - when-not, 202, 208
 - with-open, 221, 245
 - with-out-str, 221
 - współdziałanie z Javą, 219
 - wzorzec projektowy, 205
- makra odczytu, 55
- dereferencja, 56

- funkcja anonimowa, 56
 - komentarz, 55, 56
 - metadane, 56
 - podawanie zmiennych, 56
 - przytaczanie, 56
 - syntax-quote, 56
 - unquote, 56
 - unquote-slicing, 56
 - wzorzec wyrażenia regularnego, 56
- maszyny JVM, 15, 29
- autorekurencja, 124
 - Clojure, 16
 - Java, 16
- memoizacja, 165
- metadane, 77
- standardowe klucze, 78
- model czasu, 24

N

- niezmiennie struktury danych, 27

O

- optymalizacja TCO, 73, 123

P

- pamięć STM, 32, 151
- technika MVCC, 153
 - transakcje, 151
 - ACI, 151
 - aktualizacje, 151
- polimorfizm, 231
- programowa pamięć transakcyjna, *Patrz*
- pamięć STM
- programowanie, 16, 18, 31, 97, 148
- funkcyjne, 18, 24, 30, 116
 - autorekurencja, 124
 - czyste funkcje, 116
 - definicje rekurencyjne, 121
 - leniwe podejście, 118, 121
 - leniwe sekwencje, 121, 125
 - problemy rekurencyjne, 138
 - prosta rekurencja, 122

- reguły, 120
- rekurencja, 118
- rekurencja końcowa, 123
- rekurencja wzajemna, 136
- trwale struktury danych, 117
- wielokrotne wykorzystanie kodu, 119
- współużytkowanie struktur, 117
- wyrażenie listowe, 31
- zalety, 119
- imperatywne, 31
- kod pętli, 74
- prawidłowy proces pisania kodu, 273
- sekwencyjne, 97
- testowanie kodu, 274
 - BDD, 286
 - generowanie danych testu, 279
 - podstawowe etapy, 278
 - programowe sprawdzanie poprawności, 280
 - przeprowadzanie testów, 276, 283
 - sprawdzanie poprawności danych wyjściowych, 277
 - sprawdzanie poprawności kodu, 278
 - TDD, 286
 - testy jednostkowe, 286
 - testy regresji, 277
 - tworzenie danych wejściowych, 275
 - zgłaszanie błędu, 284
- współbieżne, 148
 - powody stosowania, 148
- wykorzystanie abstrakcji, 180
 - obsługa dodatkowych typów, 181
 - odczyt, 180
 - zapis, 180
- protokoły, 24, 179, 184
 - zalety, 184
- przestrzeń nazw, 36, 65
 - clojure.core, 66
 - clojure.string, 67
 - funkcje, 68
 - modyfikacja deklaracji, 264
 - myapp, 66
 - user, 36, 65
 - wiązania, 65

R

- referencje, 150
 - sprawdzanie poprawności, 156
- transakcje, 150
 - ACID, 151
 - aktualizowanie informacji, 152
 - atomowe, 151
 - izolowane, 151
 - licznik, 155
 - pamięć STM, 151
 - skoordynowane, 151
 - spójne, 151
 - trwale, 151
 - wartość wewnętrztransakcyjna, 153
 - właściwości, 151
 - zmiana encji, 150
- tworzenie, 150
- rekordy, 193
 - dodawanie metod, 195
 - dostęp do pól, 194
 - implementacja protokołu, 195
 - Note, 193
 - odwzorowania, 193
 - Person, 26
 - tworzenie, 193
- rekurencja, 72, 118
 - autorekurencja, 124
 - definicje, 121
 - indukcja, 121
 - przypadek bazowy, 121
 - problemy rekurencyjne, 138
 - prosta rekurencja, 122
 - przyspieszanie, 143
 - rekurencja końcowa, 123
 - optymalizacja TCO, 123
 - rekurencja wzajemna, 136
 - memoizacja, 144
 - optymalizowanie, 139
 - przekształcanie na autorekurencję, 138
 - zastępowanie leniwym podejściem, 141

S

sekwencje, 29, 81
 biblioteka, 86, 87
 funkcje, 88
 cechy, 83
 filtrowanie, 91
 funkcje, 91
 funkcje, 83
 into, 86
 Java, 98
 kolekcje sekwencyjne, 82, 98
 niezmiennicze, 87
 odwzorowania, 85
 predykaty, 92
 przekształcanie, 93
 funkcje, 93
 strumienie, 102
 system plików, 101
 funkcje, 101
 tryb leniwy, 86, 96
 stosowanie, 97
 zalety, 96
 tworzenie, 88
 funkcje, 88
 wektory, 84
 wymuszanie realizacji, 97
 wyrażenia regularne, 100
 funkcje, 100
 zbiory, 85
 stan, 147
 agenty, 147, 158
 bieżąca wartość, 159
 sprawdzanie poprawności, 160
 transakcje, 161
 tworzenie, 158
 wykrywanie błędów, 160
 atomy, 147, 157
 dereferencja, 157
 tworzenie, 157
 ustawienie wartości, 157
 model aktualizacji, 163
 modele zarządzania, 168

referencje, 147, 150
 transakcje, 150
 tworzenie, 150
 wywoływane zwrótnie metody, 166
 zmienne, 147, 163

Ś

środowisko REPL, 35
 zmienne specjalne, 36

T

tworzenie aplikacji, 269
 instalowanie kodu, 292
 git push, 292, 294
 Heroku, 292
 plik Procfile, 292
 testowanie kodu wyniku, 274
 defspect, 282
 dir, 279
 failures, 284
 generate-test-data, 282
 generowanie danych testu, 279
 in-ns, 281
 programowe sprawdzanie poprawności,
 280
 random-secret, 280
 require, 279
 score print-table, 276
 score-inputs, 278
 selections, 275
 sprawdzanie poprawności danych
 wyjściowych, 277
 test dla danych wejściowych, 276
 test-dirs, 283
 test-namespaces, 283
 test-vars, 283
 testy regresji, 277
 tworzenie danych wejściowych, 275
 zgłaszanie błędów, 284
 tworzenie interfejsu, 287
 board, 290
 defpartial, 288
 dodanie stanu, 287

- framework sieciowy, 287
 - interfejs gracza, 288
 - render, 290
 - session/flash-put!, 290
 - session/get, 287
 - session/put!, 287
 - session/remove!, 290
 - wynik, 270
 - exact-matches, 271
 - frequencies, 271
 - generowanie, 270
 - merge-with, 272
 - score, 273
 - select-keys, 272
 - unordered-matches, 272
 - typy danych, 179, 188
 - anonimowe egzemplarze, 198
 - cechy, 188
 - CryptoVault, 179, 188
 - definiowanie metod, 189
 - dostęp do pól, 189
 - rekordy, 193
 - tworzenie, 189
 - typy referencyjne, 38
 - atom, 38
- W**
- wiązania, 62
 - bindings, 63
 - dynamiczne, 164
 - modyfikacje na odległość, 164
 - dynamicznie określany zasięg, 164
 - funkcje, 62
 - zasięg leksykalny, 63
 - lokalne, 164
 - mechanizm rozkładania struktury, 63
 - podstawowe, 61
 - przestrzeń nazw, 65
 - zmiennne, 62
 - wielometody, 225
 - account-level, 235
 - assert-expr, 240
 - definiowanie, 228
 - dodawanie implementacji metod, 228
 - domyślne działanie, 230
 - doraźne taksonomie, 233
 - dziedziczenie, 236
 - mechanizm wybierania implementacji, 229
 - my-print, 228, 231
 - paint, 175
 - service-charge, 235
 - stosowanie, 237
 - reguły, 241
 - wybijanie metod, 229
 - dla typu kolekcji, 231
 - na podstawie typu pierwszego argumentu, 231
 - rozwiązywanie konfliktów, 232
 - współbieżność, 32
 - wyrażenie listowe, 94
 - elementy, 94
- Z**
- zmiennne, 36, 61, 163
 - aliasy, 62
 - cechy, 62
 - mechanizm rozkładania struktury, 63
 - metadane, 62
 - rodzaje definicji, 133
 - specjalne, 164
 - modyfikacje na odległość, 164
 - tworzenie, 133
 - makra, 218
 - wiązania, 62
 - lokalne dla wątku, 164
 - podstawowe, 61

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>

Zmień myślenie na funkcyjne!

Clojure to jeden z najciekawszych współczesnych języków programowania funkcyjnego, obecny na rynku od pięciu lat. Język jest oparty na wirtualnej maszynie języka Java i zachęca użytkowników do programowania współbieżnego. W ostatnim czasie Clojure gwałtownie zdobywa popularność i uznanie wśród programistów. Dzieje się tak, ponieważ jest dobrze przemyślany i wspaniale zaprojektowany, kryje w sobie potencjał języka Lisp, a do tego jest szybki i działa na dobrze wszystkim znanej wirtualnej maszynie.

Ta książka to kompletny przewodnik po Clojure. Lekturę zaczniesz od poznania jego zalet, składni i zasad programowania. W momencie, kiedy zbudujesz solidne fundamenty, przejdziesz do nauki programowania funkcyjnego, które wymaga pewnej zmiany w sposobie myślenia. Ale nie martw się, z tą książką przyjdzie Ci to z łatwością! W kolejnych rozdziałach skupisz się na programowaniu współbieżnym, protokołach, typach danych i makrach. Zobaczysz również, jak za pomocą Clojure stworzyć klasę języka Java, oraz zaznajomisz się z procesem tworzenia kompletnej aplikacji korzystającej z Clojure. Jeżeli chcesz poszerzyć swoje horyzonty programistyczne, trafiłeś na idealną książkę. Ta inwestycja się opłaci!

Poznaj zalety Clojure:

- potencjał języka Lisp
- programowanie funkcyjne i współbieżne
- uporządkowaną i przemyślaną architekturę
- zasięg wirtualnej maszyny Java
- morze nowych możliwości!



(Nr katalogowy: 13 236)



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefonicznie:
0 801 339900
0 601 339900

helion.pl
księgarnia
internetowa

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>



Helion

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

sięgnij po WIECEJ



KOD KORZYŚCI

Cena: 54,90 zł

ISBN 978-83-246-5372-0



Informatyka w najlepszym wydaniu

9 788324 653720