helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

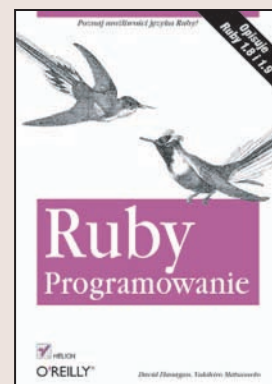
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Ruby. Programowanie



Autor: David Flanagan, Yukihiro Matsumoto
Tłumaczenie: Łukasz Piwko
ISBN: 978-83-246-1767-8
Tytuł oryginału: The Ruby Programming Language
Format: 168x237, stron: 408

Poznaj możliwości Ruby!

Zaprojektowany i stworzony w 1995 roku język Ruby dzięki swym unikalnym możliwościom zdobywa sobie coraz większe uznanie programistów na całym świecie. Jak udało mu się wkupić w łaski tego nieufnego środowiska? Przyczyniła się do tego między innymi prosta składnia z wbudowanymi w nią wyrażeniami regularnymi, automatyczne oczyszczanie pamięci i przeciążanie operatorów. Dodatkowo ogromna i chętna do pomocy społeczność sprawia, że to rozwiązanie staje się jeszcze bardziej atrakcyjne. Uwaga! Jednym z autorów tej książki jest sam Yukihiro Matsumoto – twórca języka Ruby!

Książka stanowi kompletny zbiór informacji na temat języka Ruby. Jeśli naprawdę chcesz zrozumieć ten język, oto obowiązkowa pozycja do przeczytania! W trakcie lektury zapoznasz się z bogatym API, pozwalającym na przetwarzanie tekstu; zrozumiesz techniki związane z wykonywaniem działań na liczbach, implementacją kolekcji, operacjami wejścia-wyjścia oraz pracą współbieżną i operacjami sieciowymi. Ponadto znajdziesz tu elementy dostępne powszechnie w językach programowania, takie jak instrukcje warunkowe, pętle czy też operatory logiczne. Dzięki książce „Ruby. Programowanie" wykorzystanie metod i obiektów klasy Proc oraz stosowanie platformy Ruby nie będzie stanowiło dla Ciebie najmniejszego problemu!

- Wprowadzenie do języka Ruby
- Sposoby uruchamiania programów napisanych w Ruby
- Dostępne typy danych
- Zastosowanie wyrażeń i operatorów
- Sterowanie przepływem
- Wykorzystanie iteratorów oraz enumeratorów
- Obsługa wyjątków
- Zastosowanie współbieżności
- Użycie domknięć
- Cykl życia obiektów
- Refleksje oraz metaprogramowanie
- Liczby w Ruby
- Używanie wyrażeń regularnych
- Kolekcje
- Operacje na dacie i godzinie
- Tablice jedno- oraz wielowymiarowe
- Obsługa plików oraz katalogów
- Programowanie sieciowe
- Obsługa środowiska Ruby
- Gwarancja bezpieczeństwa

Wykorzystaj elastyczność i możliwości języka Ruby!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Wstęp 7

1. Wprowadzenie 11

1.1. Krótki kurs języka Ruby 12
1.2. Wypróbuj język Ruby 20
1.3. Książka — informacje 24
1.4. Program rozwiązujący sudoku 25

2. Struktura i uruchamianie programów Ruby 31

2.1. Struktura leksykalna 32
2.2. Struktura syntaktyczna 39
2.3. Struktura plików 40
2.4. Kodowanie znaków 41
2.5. Wykonywanie programu 43

3. Typy danych i obiekty 45

3.1. Liczby 46
3.2. Tekst 50
3.3. Tablice 66
3.4. Tablice asocjacyjne 68
3.5. Zakresy 70
3.6. Symbole 72
3.7. Słowa kluczowe true, false i nil 73
3.8. Obiekty 73

4. Wyrażenia i operatory 85

4.1. Literały i literały słów kluczowych 86
4.2. Odwołania do zmiennych 87
4.3. Odwołania do stałych 88

4.4. Wywołania metod 89
4.5. Przypisania 91
4.6. Operatory 99

5. Instrukcje i przepływ sterowania ... 115
5.1. Instrukcje warunkowe 116
5.2. Pętle 124
5.3. Iteratory i obiekty przeliczalne 126
5.4. Bloki 136
5.5. Kontrola przepływu sterowania 141
5.6. Wyjątki i ich obsługa 148
5.7. Instrukcje BEGIN i END 158
5.8. Wątki, włókna i kontynuacje 159

6. Metody, obiekty klasy Proc, lambdy i domknięcia ... 165
6.1. Definiowanie prostych metod 167
6.2. Nazwy metod 170
6.3. Nawiasy w metodach 172
6.4. Argumenty metod 174
6.5. Obiekty proc i lambda 181
6.6. Domknięcia 188
6.7. Obiekty klasy Method 191
6.8. Programowanie funkcyjne 192

7. Klasy i moduły ... 201
7.1. Definiowanie prostej klasy 202
7.2. Widoczność metod — publiczne, chronione i prywatne 218
7.3. Tworzenie podklas i dziedziczenie 220
7.4. Tworzenie i inicjacja obiektów 227
7.5. Moduły 232
7.6. Funkcje load i require 236
7.7. Metody singletonowe i klasa eigenclass 240
7.8. Wyszukiwanie metod 242
7.9. Wyszukiwanie stałych 244

8. Refleksja i metaprogramowanie ... 247
8.1. Typy, klasy i moduły 248
8.2. Wykonywanie łańcuchów i bloków 250
8.3. Zmienne i stałe 252
8.4. Metody 254
8.5. Metody zwrotne 258
8.6. Śledzenie 259

8.7. Moduły ObjectSpace i GC 261
8.8. Niestandardowe struktury sterujące 262
8.9. Brakujące metody i stałe 264
8.10. Dynamiczne tworzenie metod 267
8.11. Tworzenie łańcuchów aliasów 269
8.12. Języki do wyspecjalizowanych zastosowań 274

9. Platforma Ruby 279
9.1. Łańcuchy 280
9.2. Wyrażenia regularne 285
9.3. Liczby i matematyka 295
9.4. Data i godzina 299
9.5. Kolekcje 300
9.6. Pliki i katalogi 320
9.7. Wejście i wyjście 325
9.8. Programowanie sieciowe 335
9.9. Wątki i współbieżność 340

10. Środowisko Ruby 355
10.1. Uruchamianie interpretera Ruby 356
10.2. Środowisko najwyższego poziomu 360
10.3. Praktyczne skróty do ekstrakcji i raportowania 368
10.4. Wywoływanie systemu operacyjnego 370
10.5. Bezpieczeństwo 374

Skorowidz 379

ROZDZIAŁ 2.

Struktura i uruchamianie programów Ruby

Niniejszy rozdział opisuje strukturę programów w języku Ruby. Na początku opisano strukturę leksykalną, w tym tokeny i znaki, z których składa się program. Dalej znajduje się opis struktury syntaktycznej programu Ruby z wyjaśnieniem sposobów pisania wyrażeń, instrukcji sterujących, metod, klas itd. jako sekwencji tokenów. Na końcu zamieszczony jest opis plików zawierających kod Ruby z objaśnieniem sposobów dzielenia programów w tym języku na kilka plików oraz wykonywania pliku z kodem Ruby przez interpreter Ruby.

2.1. Struktura leksykalna

Interpreter Ruby analizuje program jako sekwencję **tokenów**. Należą do nich: komentarze, literały, znaki interpunkcyjne, identyfikatory i słowa kluczowe. Niniejszy rozdział wprowadza wymienione typy tokenów oraz zawiera ważne informacje na temat znaków, z których się one składają, w tym rozdzielających białych znaków.

2.1.1. Komentarze

Komentarze zaczynają się od znaku # i mają zasięg do końca wiersza. Interpreter Ruby ignoruje znaki # i wszystko, co znajduje się za nimi w tym samym wierszu (nie ignoruje znaku nowego wiersza, który jest białym znakiem mogącym oznaczać zakończenie instrukcji).

Jeśli # znajduje się wewnątrz łańcucha lub wyrażenia regularnego (zobacz rozdział 3.), to stanowi jego część i nie wprowadza komentarza.

```
# Ten cały wiersz jest komentarzem.
x = "#To jest łańcuch."                  # A to jest komentarz.
y = /#To jest wyrażenie regularne./      # Tu jest jeszcze jeden komentarz.
```

Komentarze zajmujące wiele wierszy są tworzone przez wstawienie znaku # na początku każdego z nich.

```
#
# Niniejsza klasa reprezentuje liczbę typu Complex.
# Mimo nazwy (złożona) nie jest ona wcale skomplikowana.
#
```

Należy zauważyć, że w języku Ruby nie ma komentarzy w stylu C `/* */`. Nie ma sposobu na wstawienie komentarza w środku wiersza kodu.

2.1.1.1. Dokumenty osadzone

W języku Ruby dostępny jest jeszcze inny sposób wstawiania komentarzy wielowierszowych. Są to tak zwane **dokumenty osadzone** (ang. *embedded document*). Pierwszy wiersz takiego komentarza zaczyna się od ciągu znaków `=begin`, a ostatni od ciągu `=end`. Tekst znajdujący się pomiędzy `=begin` a `=end` jest ignorowany. Należy tylko pamiętać, że między tekstem a ciągami `=begin` i `=end` musi być przynajmniej jedna spacja.

Dokumenty osadzone są wygodnym sposobem na tworzenie długich komentarzy bez wstawiania na początku każdego wiersza znaku #.

```
=begin Ktoś musi naprawić poniższy kod!
  Kod znajdujący się w tym miejscu jest w komentarzu.
=end
```

Warto zauważyć, że dokumenty osadzone działają tylko wówczas, gdy wiersze zaczynają się od znaku =.

```
# =begin Ten wiersz był komentarzem, a teraz sam jest w komentarzu!
    Kod znajdujący się w tym miejscu nie jest w komentarzu.
# =end
```

Jak sama nazwa wskazuje, dokumenty osadzone mogą służyć do wstawiania w programie Ruby długich bloków dokumentacji lub kodu źródłowego w innym języku (na przykład HTML lub SQL). Są one często przeznaczone do użytku przez różnego rodzaju narzędzia przetwarzania końcowego, które są uruchamiane na rzecz kodu Ruby. Po ciągu `=begin` z reguły umieszczany jest identyfikator określający, dla jakiego narzędzia przeznaczony jest dany komentarz.

2.1.1.2. Komentarze dokumentacyjne

W programach Ruby można osadzać dokumentację API w specjalnych, przeznaczonych do tego celu komentarzach poprzedzających definicje metod, klas i modułów. Do przeglądania tej dokumentacji służy omówione w rozdziale 1.2.4 narzędzie o nazwie ri. Narzędzie rdoc pobiera komentarze dokumentacyjne z kodu źródłowego Ruby i konwertuje je na format HTML lub przygotowuje do wyświetlenia przez narzędzie ri. Opis narzędzia rdoc wykracza poza tematykę tej książki. Szczegółowe informacje na jego temat znajdują się w pliku *lib/rdoc/README* w kodzie źródłowym Ruby.

Komentarz dokumentacyjny musi znajdować się bezpośrednio przed modułem, klasą lub metodą, które API dokumentuje. Z reguły zajmuje kilka wierszy zaczynających się od znaku #, ale może też mieć formę dokumentu osadzonego rozpoczynającego się ciągiem `=begin rdoc` (jeśli słowo `rdoc` zostanie pominięte, narzędzie rdoc nie zauważy tego komentarza).

Poniższy komentarz demonstruje najważniejsze elementy formatujące gramatyki oznaczania komentarzy dokumentacyjnych w języku Ruby. Szczegółowy opis tej gramatyki znajduje się w wymienionym wcześniej pliku *README*:

```
#
# Gramatyka oznaczania komentarzy rdoc jest prosta jak w wiki.
#
# Akapity oddziela się pustym wierszem.
#
# = Nagłówki.
#
# Nagłówki zaczynają się od znaku równości.
#
# == Podnagłówki.
# Powyższy wiersz tworzy podnagłówek.
# === Podpodnagłówek.
# I tak dalej.
#
# = Przykłady.
#
#   Wcięte wiersze są wyświetlane dosłownie pismem o stałej szerokości znaków.
#     Należy uważać, aby nie wciąć nagłówków lub list.
#
# = Listy i rodzaje pisma.
#
# Elementy listy zaczynają się od znaku * lub -. Rodzaj pisma określa się za pomocą interpunkcji lub kodu HTML:
# * _italic_ lub <i>kursywa</i>
# * *bold* lub <b>pogrubienie</b>
# * +code+ lub <tt>pismo o stałej szerokości znaków</tt>
#
```

```
# 1. Listy numerowane zaczynają się od liczb.
# 99. Można używać dowolnych liczb; nie muszą być kolejne.
# 1. Nie ma sposobu na zagnieżdżanie list.
#
# Terminy list opisowych są umieszczane w nawiasach kwadratowych:
# [element 1]  Opis elementu 1.
# [element 2]  Opis elementu 2.
#
```

2.1.2. Literały

Literały to wartości, które znajdują się bezpośrednio w kodzie źródłowym Ruby. Zaliczają się do nich liczby, łańcuchy tekstowe i wyrażenia regularne (inne literały, takie jak wartości tablic jednowymiarowych i asocjacyjnych, nie są pojedynczymi tokenami, a bardziej złożonymi wyrażeniami). Składnia literałów liczbowych i łańcuchowych w języku Ruby jest skomplikowana, a została szczegółowo opisana w rozdziale 3. Na razie wystarczy tylko krótki przykład demonstrujący, jak wyglądają literały w języku Ruby:

```
1                  # Literał całkowitoliczbowy.
1.0                # Literał liczby zmiennoprzecinkowej.
'one'              # Literał łańcuchowy.
"two"              # Inny literał łańcuchowy.
/trzy/             # Literał wyrażenia regularnego.
```

2.1.3. Znaki interpunkcyjne

Znaki interpunkcyjne spełniają w języku Ruby różne zadania. Ich postać ma większość operatorów w tym języku, na przykład operator dodawania to +, mnożenia *, a logiczne LUB to `||`. Pełna lista operatorów w języku Ruby znajduje się w rozdziale 4.6. Znaki specjalne służą także do oddzielania łańcuchów, wyrażeń regularnych, literałów tablic jednowymiarowych i asocjacyjnych oraz grupowania wyrażeń, argumentów metod oraz indeksów tablic. W składni języka Ruby znaki interpunkcyjne mają także wiele innych zastosowań, o których będzie jeszcze mowa.

2.1.4. Identyfikatory

Identyfikator to inaczej nazwa. W języku Ruby wykorzystuje się go do nazywania zmiennych, metod, klas itd. Może składać się z liter, liczb i znaków podkreślenia, ale nie może zaczynać się od liczby. Nie może zawierać znaków: białych, niedrukowalnych oraz interpunkcyjnych z wyjątkiem wymienionych tutaj.

Identyfikatory zaczynające się od wielkiej litery A-Z oznaczają stałe. Jeśli wartość takiego identyfikatora zostanie zmieniona w programie, interpreter zgłosi ostrzeżenie (ale nie błąd). Nazwy klas i modułów muszą zaczynać się od wielkiej litery. Poniżej znajduje się kilka przykładowych identyfikatorów:

```
i
x2
old_value
_internal    # Identyfikatory mogą zaczynać się od znaku podkreślenia.
PI           # Stała.
```

Zgodnie z konwencją identyfikatory składające się z kilku słów niebędących stałymi pisane są ze znakiem podkreślenia `w_taki_sposób`, podczas gdy identyfikatory stałych, które składają się z kilku wyrazów, pisze się `WTakiSposób` lub `W_TAKI_SPOSÓB`.

2.1.4.1. Rozróżnianie wielkich i małych liter

Wielkość liter w języku Ruby ma znaczenie. Mała litera i wielka litera to nie to samo. Na przykład słowo kluczowe `end` jest czymś innym niż słowo kluczowe `END`.

2.1.4.2. Znaki Unicode w identyfikatorach

Reguły języka Ruby dotyczące tworzenia identyfikatorów są zdefiniowane w kategoriach znaków ASCII, które są zabronione. Ogólnie mówiąc, wszystkie znaki spoza zestawu ASCII mogą być używane w identyfikatorach, wliczając te wyglądające na znaki interpunkcyjne. Na przykład w pliku UTF-8 poniższy kod Ruby jest poprawny:

```
def ×(x,y)    # Nazwą tej metody jest znak mnożenia Unicode.
  x*y         # Metoda ta mnoży podane argumenty.
end
```

Podobnie japoński programista może w swoim programie zakodowanym w systemie SJIS lub EUC używać w identyfikatorach znaków Kanji. Więcej informacji na temat pisania programów kodowanych w innych systemach niż ASCII znajduje się w podrozdziale 2.4.1.

Specjalne reguły dotyczące tworzenia identyfikatorów są oparte na znakach ASCII i nie dotyczą znaków spoza tego zestawu. Na przykład identyfikator nie może zaczynać się od cyfry z zestawu ASCII, ale może zaczynać się od cyfry z alfabetu innego niż typu Latin. Podobnie, aby być stałą, musi zaczynać się od wielkiej litery z zestawu ASCII. Na przykład identyfikator Ĺ nie jest stałą.

Dwa identyfikatory są takie same tylko wówczas, gdy są reprezentowane przez taki sam ciąg bajtów. Niektóre zestawy znaków, na przykład Unicode, posiadają więcej niż jeden punkt kodowy do reprezentacji tego samego znaku. W języku Ruby nie ma żadnej normalizacji Unicode, a więc różne punkty kodowe są uznawane za różne znaki, nawet jeśli mają identyczne znaczenie lub są reprezentowane przez taki sam glif czcionki.

2.1.4.3. Znaki interpunkcyjne w identyfikatorach

Znaki interpunkcyjne mogą występować na początku i końcu identyfikatorów. Ich znaczenie jest następujące:

`$`
: Znak ten przed nazwą zmiennej oznacza, że jest ona globalna. Podążając za przykładem Perla, w języku Ruby istnieje grupa zmiennych globalnych, które w nazwach zawierają inne znaki interpunkcyjne, takie jak `$_` i `$-K`. Lista tych specjalnych zmiennych znajduje się w rozdziale 10.

`@`
: Na początku zmiennych obiektowych znajduje się symbol `@`. Zmienne klasowe mają przedrostek złożony z dwóch takich znaków. Zmienne obiektowe i klasowe zostały opisane w rozdziale 7.

`?`
: Bardzo pomocną konwencją jest umieszczanie na końcu nazw metod zwracających wartości logiczne znaku `?`.

`!`
: Nazwy metod kończące się znakiem wykrzyknika wskazują, że należy ostrożnie ich używać. Konwencja ta jest z reguły stosowana do wyróżnienia metod mutacyjnych modyfikujących obiekty bezpośrednio w porównaniu z metodami, które zwracają zmodyfikowaną kopię obiektu.

`=`
: Metody posiadające nazwy kończące się znakiem równości można wywoływać poprzez wstawienie ich nazw bez znaku równości po lewej stronie operatora przypisania (więcej na ten temat można przeczytać w podrozdziałach 4.5.3 i 7.1.5).

Poniżej znajduje się kilka identyfikatorów z przedrostkami lub przyrostkami:

```
$files          # Zmienna globalna.
@data           # Zmienna obiektowa.
@@counter       # Zmienna klasowa.
empty?          # Metoda zwracająca wartość logiczną, czyli predykat.
sort!           # Wersja metody sort modyfikująca obiekty bezpośrednio.
timeout=        # Metoda wywoływana przez przypisanie.
```

Niektóre operatory języka Ruby są zaimplementowane jako metody, dzięki czemu można je przedefiniowywać w różnych klasach zgodnie z potrzebą. Dlatego też istnieje możliwość używania niektórych operatorów jako nazw metod. W tym kontekście znaki interpunkcyjne lub znaki operatorów są traktowane jako identyfikatory, a nie operatory. Więcej informacji na temat operatorów w języku Ruby znajduje się w podrozdziale 4.6.

2.1.5. Słowa kluczowe

Poniższe słowa kluczowe mają w języku Ruby specjalną funkcję i tak też są traktowane przez analizator składni Ruby:

```
__LINE__      case        ensure      not         then
__ENCODING__  class       false       or          true
__FILE__      def         for         redo        undef
BEGIN         defined?    if          rescue      unless
END           do          in          retry       until
alias         else        module      return      when
and           elsif       next        self        while
begin         end         nil         super       yield
break
```

Poza powyższymi słowami istnieją jeszcze trzy tokeny przypominające słowa kluczowe. Są one specjalnie traktowane przez analizator składni tylko wówczas, gdy znajdują się na początku wiersza.

```
=begin    =end      __END__
```

Jak wiadomo, tokeny `=begin` i `=end` na początku wiersza oznaczają początek i koniec komentarza wielowierszowego. Token `_END_` oznacza koniec programu (i początek sekcji danych), jeśli znajduje się sam w wierszu (ani przed nim, ani za nim nie może być żadnych białych znaków).

W większości języków słowa kluczowe byłyby tak zwanymi słowami zarezerwowanymi i nigdy nie można by było używać ich jako identyfikatorów. Analizator składni Ruby jest bardzo elastyczny. Nie zgłasza żadnych błędów, jeśli któreś z tych słów zostanie opatrzone przedrostkiem `@`, `@@` czy `$` i użyte jako identyfikator zmiennej obiektowej, klasowej lub globalnej. Ponadto słów kluczowych można używać jako nazw metod, z tym, że metoda musi zawsze być wywoływana jawnie poprzez obiekt. Należy jednak pamiętać, że użycie tych słów jako identyfikatorów spowoduje zaciemnienie kodu źródłowego. Najlepiej jest traktować je jako słowa zarezerwowane.

Wiele ważnych własności języka Ruby jest zaimplementowanych jako metody klas `Kernel`, `Module`, `Class` i `Object`. Dlatego też dobrze jest poniższe słowa również traktować jako zarezerwowane:

```
# To są metody, które wyglądają na instrukcje lub słowa kluczowe.
at_exit         catch       private     require
attr            include     proc        throw
attr_accessor   lambda      protected
attr_reader     load        public
attr_writer     loop        raise
```

```
# To są często używane funkcje globalne.
Array          chomp!         gsub!          select
Float          chop           iterator?      sleep
Integer        chop!          load           split
String         eval           open           sprintf
URI            exec           p              srand
abort          exit           print          sub
autoload       exit!          printf         sub!
autoload?      fail           putc           syscall
binding        fork           puts           system
block_given?   format         rand           test
callcc         getc           readline       trap
caller         gets           readlines      warn
chomp          gsub           scan

# To są często używane metody obiektowe.
allocate       freeze         kind_of?       superclass
clone          frozen?        method         taint
display        hash           methods        tainted?
dup            id             new            to_a
enum_for       inherited      nil?           to_enum
eql?           inspect        object_id      to_s
equal?         instance_of?   respond_to?    untaint
extend         is_a?          send
```

2.1.6. Białe znaki

Spacje, tabulatory i znaki nowego wiersza nie są same w sobie tokenami, ale służą do ich oddzielania, gdyż te w przeciwnym razie zlałyby się w jeden. Poza tą podstawową funkcją rozdzielającą większość białych znaków jest ignorowana przez interpreter Ruby. Są one stosowane głównie do formatowania programów, aby łatwiej się je czytało. Jednak nie wszystkie białe znaki są ignorowane. Niektóre są wymagane, a niektóre nawet zabronione. Gramatyka języka Ruby jest ekspresywna, ale i skomplikowana. Istnieje kilka przypadków, w których wstawienie lub usunięcie białego znaku może spowodować zmianę działania programu. Mimo iż sytuacje takie nie są częste, należy o nich wiedzieć.

2.1.6.1. Znaki nowego wiersza jako znaki kończące instrukcje

Najczęstsze zastosowanie białych znaków ma związek ze znakami nowego wiersza, które służą jako znaki kończące instrukcje. W językach takich jak Java i C każda instrukcja musi być zakończona średnikiem. W języku Ruby także można kończyć instrukcje tym znakiem, ale jest to wymagane tylko wówczas, gdy w jednym wierszu znajduje się więcej niż jedna instrukcja. Zgodnie z konwencją we wszystkich innych tego typu sytuacjach średnik nie jest używany.

Przy braku średników interpreter Ruby musi zgadnąć na własną rękę, gdzie kończy się dana instrukcja. Jeśli kod w wierszu stanowi pełną pod względem syntaktycznym instrukcję, znakiem ją kończącym jest znak nowego wiersza. Gdy instrukcja nie jest kompletna, Ruby kontynuuje jej analizę w kolejnym wierszu (w Ruby 1.9 jest jeden wyjątek od tej reguły, który został opisany dalej w tym podrozdziale).

Jeżeli wszystkie instrukcje mieszczą się w pojedynczym wierszu, nie ma problemu. Jednak w przypadku gdy instrukcja zajmuje więcej niż jeden wiersz, należy tak ją podzielić, aby interpreter nie wziął jej pierwszej części za kompletną instrukcję. W takiej sytuacji w grę wchodzi biały znak. Program może działać na różne sposoby w zależności od tego, gdzie się on znajduje. Na przykład poniższa procedura dodaje `x` do `y` i sumę zapisuje w zmiennej `total`:

```
total = x +         # Wyrażenie niekompletne — analiza jest kontynuowana.
  y
```

Natomiast poniższy kod przypisuje zmienną `x` do `total`, a następnie oblicza wartość `y` i nic z nią nie robi:

```
total = x   # To jest kompletne wyrażenie.
  + y       # Bezużyteczne, ale kompletne wyrażenie.
```

Kolejnym przykładem są instrukcje `return` i `break`, po których opcjonalnie może znajdować się wyrażenie określające wartość zwrotną. Znak nowego wiersza pomiędzy słowem kluczowym a wyrażeniem spowoduje zakończenie instrukcji przed tym ostatnim.

Znak nowego wiersza można wstawić bez obawy, że instrukcja zostanie zakończona zbyt wcześnie, po każdym operatorze, kropce lub przecinku w wywołaniu metody oraz literale dowolnego rodzaju tablicy.

Można go także zastąpić lewym ukośnikiem, co zapobiega automatycznemu zakończeniu instrukcji przez Ruby:

```
var total = first_long_variable_name + second_long_variable_name \
  + third_long_variable_name # Powyżej nie ma żadnego znaku kończącego instrukcję.
```

Reguły dotyczące kończenia instrukcji są nieco inne w Ruby 1.9. Jeśli pierwszym znakiem niebędącym spacją w wierszu jest kropka, wiersz ten jest traktowany jako kontynuacja poprzedniego, a więc znajdujący się wcześniej znak nowego wiersza nie kończy instrukcji. Wiersze zaczynające się od kropek są przydatne w przypadku długich łańcuchów metod, czasami nazywanych płynnymi API (ang. *fluent API*); każda wywołana w nich metoda zwraca obiekt, na rzecz którego mogą być wywołane dodatkowe metody. Na przykład:

```
animals = Array.new
  .push("pies")    # Nie działa w Ruby 1.8.
  .push("krowa")
  .push("kot")
  .sort
```

2.1.6.2. Spacje a wywoływanie metod

W niektórych sytuacjach gramatyka Ruby dopuszcza pominięcie nawiasów w wywołaniach metod, które dzięki temu mogą być używane tak, jakby były instrukcjami; w znacznym stopniu wpływa to na elegancję kodu. Niestety, możliwość ta powoduje zależność od białych znaków. Przykładowo poniższe dwa wiersze kodu różnią się tylko jedną spacją:

```
f(3+2)+1
f (3+2)+1
```

W pierwszym wierszu do funkcji `f` zostaje przekazana wartość `5`, a do zwróconego wyniku zostaje dodana jedynka. Ponieważ w drugim wierszu po nazwie funkcji znajduje się spacja, Ruby zakłada, że nawias w wywołaniu tej funkcji został pominięty. Nawias znajdujący się dalej jest traktowany jako sposób oddzielenia wyrażenia, a argumentem funkcji jest całe wyrażenie `(3+2)+1`. Jeśli włączone są ostrzeżenia (za pomocą opcji `-w`), Ruby zgłasza ostrzeżenie zawsze, gdy napotyka taki niejednoznaczny kod.

Rozwiązanie tego problemu jest proste:

- Nigdy nie należy umieszczać spacji między nazwą metody a otwierającym nawiasem.
- Jeżeli pierwszy argument metody zaczyna się od nawiasu otwierającego, wywołanie metody zawsze powinno być otoczone nawiasami. Na przykład `f((3+2)+1)`.

- Zawsze uruchamiaj interpreter z opcją `-w`, dzięki czemu będzie on zgłaszał ostrzeżenia zawsze, gdy zapomnisz o powyższych regułach!

2.2. Struktura syntaktyczna

Do tej pory zostały omówione tokeny i znaki, z których się one składają. Teraz krótko zajmiemy się tym, jak tokeny leksykalne łączą się w większe struktury syntaktyczne programu Ruby. Niniejszy podrozdział opisuje składnię programów Ruby od najprostszych wyrażeń po największe moduły. W efekcie jest on mapą prowadzącą do kolejnych rozdziałów.

Podstawową jednostką syntaktyczną w języku Ruby jest **wyrażenie** (ang. *expression*). Interpreter Ruby **oblicza** wyrażenia, zwracając ich wartości. Najprostsze wyrażenia to **wyrażenia pierwotne** (ang. *primary expression*), które bezpośrednio reprezentują wartości. Należą do nich opisywane wcześniej literały łańcuchowe i liczbowe. Inne tego typu wyrażenia to niektóre słowa kluczowe, jak `true`, `false`, `nil` i `self`. Odwołania do zmiennych również są wyrażeniami pierwotnymi. Ich wartością jest wartość zmiennej.

Bardziej skomplikowane wartości mogą być zapisane jako wyrażenia złożone:

```
[1,2,3]                # Literał tablicowy.
{1=>"one", 2=>"two"}   # Literał tablicy asocjacyjnej.
1..3                   # Literał zakresowy.
```

Operatory służą do wykonywania obliczeń na wartościach, a wyrażenia złożone zbudowane są z prostszych podwyrażeń rozdzielonych operatorami:

```
1             # Wyrażenie pierwotne.
x             # Inne wyrażenie pierwotne.
x = 1         # Wyrażenie przypisania.
x = x + 1     # Wyrażenie z dwoma operatorami.
```

Tematowi operatorów i wyrażeń, w tym zmiennych i wyrażeń przypisania, poświęcony jest rozdział 4.

Wyrażenia w połączeniu ze słowami kluczowymi tworzą instrukcje, jak na przykład instrukcja `if`, która warunkowo wykonuje kod, lub instrukcja `while` wykonująca kod wielokrotnie:

```
if x < 10 then     # Jeśli to wyrażenie ma wartość true,
  x = x + 1        # należy wykonać tę instrukcję.
end                # Oznacza koniec instrukcji warunkowej.
while x < 10 do    # Dopóki wyrażenie to ma wartość true...
  print x          # należy wykonywać tę instrukcję.
  x = x + 1        # Następnie należy wykonać tę instrukcję.
end                # Oznacza koniec pętli.
```

W języku Ruby instrukcje te są z technicznego punktu widzenia wyrażeniami, ale nadal istnieje przydatne rozróżnienie pomiędzy wyrażeniami na wpływające na przepływ sterowania w programie i na te, które tego nie robią. Instrukcje sterujące zostały opisane w rozdziale 5.

We wszystkich nieprymitywnych programach wyrażenia i instrukcje grupowane są w parametryzowane jednostki, dzięki czemu mogą być wielokrotnie wywoływane przy użyciu różnych danych wejściowych. Jednostki te są nazywane funkcjami, procedurami lub podprocedurami. Ponieważ język Ruby jest zorientowany obiektowo, jednostki te to **metody**. Metody i związane z nimi struktury nazywane proc i lambda są tematem rozdziału 6.

Zestawy metod zaprojektowane, aby wzajemnie oddziaływały między sobą, można łączyć w klasy, a grupy wzajemnie powiązanych klas i metod, które są od nich niezależne, tworzą **moduły**. Klasy i moduły są tematem rozdziału 7.

2.2.1. Struktura bloku

Programy Ruby mają strukturę blokową. Moduły, klasy i definicje metod, a także większość instrukcji zawiera bloki zagnieżdżonego kodu. Są one oznaczane słowami kluczowymi lub specjalnymi znakami i zgodnie z konwencją powinny być wcięte na głębokość dwóch spacji względem swoich ograniczników. W programach w języku Ruby mogą występować dwa rodzaje bloków. Jeden z nich jest formalnie nazywany blokiem. Jest to fragment kodu związany z metodą iteracyjną lub do niej przekazywany:

```
3.times { print "Ruby! " }
```

W powyższym przykładzie klamry wraz ze znajdującym się między nimi kodem stanowią blok związany z wywołaniem metody iteracyjnej `3.times`. Formalne bloki tego typu mogą być oznaczane klamrami lub słowami kluczowymi `do` i `end`:

```
1.upto(10) do |x|
  print x
end
```

Ograniczniki `do` i `end` są z reguły używane w przypadkach, gdy blok zajmuje więcej niż jeden wiersz kodu. Zwróć uwagę na wcięcie wielkości dwóch spacji kodu w bloku. Bloki zostały opisane w podrozdziale 5.4.

Aby uniknąć mylenia prawdziwych bloków, drugi ich rodzaj można nazwać **ciałem** (w praktyce jednak termin „blok” jest używany w obu przypadkach). Ciało jest listą instrukcji, które składają się na ciało definicji klasy, metody, pętli `while` lub czegokolwiek innego. Nigdy nie jest oznaczane klamrami — w tym przypadku ogranicznikami są słowa kluczowe. Szczegóły dotyczące składni ciał instrukcji, metod oraz klas i modułów znajdują się odpowiednio w rozdziałach 5., 6. i 7.

Ciała i bloki można zagnieżdżać jedne w drugich. Programy języka Ruby zazwyczaj zawierają kilka poziomów zagnieżdżonego kodu, czytelnego dzięki wcięciom. Poniżej znajduje się schematyczny przykład:

```
module Stats                          # Moduł.
  class Dataset                       # Klasa w module.
    def initialize(filename)          # Metoda w klasie.
      IO.foreach(filename) do |line|  # Blok w metodzie.
        if line[0,1] == "#"           # Instrukcja if w bloku.
          next                        # Prosta instrukcja w instrukcji if.
        end                           # Koniec ciała instrukcji if.
      end                             # Koniec bloku.
    end                               # Koniec ciała metody.
  end                                 # Koniec ciała klasy.
end                                   # Koniec ciała modułu.
```

2.3. Struktura plików

Zasad dotyczących struktury kodu języka Ruby w pliku jest kilka. Dotyczą one przygotowywania programów do użytku i nie dotyczą bezpośrednio samego języka.

Po pierwsze, jeśli w programie Ruby zawarty jest komentarz shebang (`#!`) informujący systemy operacyjne typu Unix, jak go uruchomić, musi on znajdować się w pierwszej linijce.

Po drugie, w sytuacji gdy w programie Ruby znajduje się komentarz określający kodowanie znaków (opisane w podrozdziale 2.4.1), musi on znajdować się w pierwszej linijce lub w drugiej, jeśli w pierwszej jest komentarz shebang.

Po trzecie, jeżeli plik zawiera linijkę, w której znajduje się tylko token `_END_` bez żadnych białych znaków przed nim i za nim, interpreter kończy przetwarzanie w tym miejscu. W dalszej części pliku mogą znajdować się dowolne dane, które program może odczytywać za pomocą stałej `DATA` obiektu `IO` (więcej informacji na temat tej stałej globalnej można znaleźć w podrozdziale 9.7 i rozdziale 10.).

Program Ruby nie musi mieścić się w jednym pliku. Na przykład wiele programów ładuje kod Ruby z dodatkowych plików bibliotecznych. Do ładowania kodu z innych plików służy metoda `require`. Szuka ona określonych modułów na ścieżce wyszukiwania i uniemożliwia załadowanie danego modułu więcej niż jeden raz. Szczegóły na ten temat znajdują się w podrozdziale 7.6.

Poniższy kod ilustruje każdy z wymienionych punktów struktury pliku z programem Ruby:

```
#!/usr/bin/ruby -w            Komentarz shebang.
# -*- coding: utf-8 -*-       Komentarz określający kodowanie.
require 'socket'              Załadowanie biblioteki sieciowej.
  ...                         Kod programu.
__END__                       Koniec programu.
  ...                         Dane programu.
```

2.4. Kodowanie znaków

Na najniższym poziomie program w języku Ruby jest ciągiem znaków. Reguły leksykalne tego języka zostały zdefiniowane przy użyciu znaków z zestawu ASCII. Na przykład komentarze zaczynają się od znaku # (kod ASCII 35), a dozwolone białe znaki to: tabulator poziomy (ASCII 9), znak nowego wiersza (10), tabulator pionowy (11), wysuw strony (12), powrót karetki (13) i spacja (32).Wszystkie słowa kluczowe języka Ruby zostały zapisane znakami ASCII; także wszystkie operatory i inne znaki interpunkcyjne pochodzą z tego zestawu.

Domyślnie interpreter Ruby przyjmuje, że kod źródłowy Ruby jest zakodowany w systemie ASCII. Nie jest to jednak wymóg. Interpreter może przetwarzać także pliki zakodowane w innych systemach zawierających wszystkie znaki dostępne w ASCII. Aby mógł on zinterpretować bajty pliku źródłowego jako znaki, musi wiedzieć, jakiego kodowania użyć. Kodowanie mogą określać pliki Ruby lub można poinformować o tym interpreter. Wyjaśnienie, jak to zrobić, znajduje się nieco dalej.

Interpreter Ruby jest bardzo elastyczny, jeśli chodzi o znaki występujące w programach. Niektóre znaki ASCII mają specjalne znaczenie i nie mogą być stosowane w identyfikatorach. Poza tym program Ruby może zawierać wszelkie znaki dozwolone przez kodowanie. Napisaliśmy wcześniej, że identyfikatory mogą zawierać znaki spoza zestawu ASCII. To samo dotyczy komentarzy, literałów łańcuchowych i wyrażeń regularnych — mogą zawierać dowolne znaki inne niż znak ograniczający oznaczający koniec komentarza lub literału. W plikach ASCII łańcuchy mogą zawierać dowolne bajty, także te, które reprezentują niedrukowalne znaki kontrolne (takie użycie surowych bajtów nie jest jednak zalecane; w literałach w języku Ruby można stosować symbole zastępcze, dzięki czemu dowolne znaki można wstawić przy

użyciu kodów liczbowych). Jeśli plik jest zakodowany w systemie UTF-8, komentarze, łańcuchy i wyrażenia regularne mogą zawierać dowolne znaki Unicode. Jeśli plik jest zakodowany w jednym z japońskich systemów — SJIS lub EUC — łańcuchy mogą zawierać znaki Kanji.

2.4.1. Określanie kodowania programu

Domyślnie interpreter Ruby przyjmuje, że programy są kodowane w systemie ASCII. W Ruby 1.8 kodowanie można zmienić za pomocą opcji wiersza poleceń `-K`. Aby uruchomić program Ruby zawierający znaki Unicode w UTF-8, należy uruchomić interpreter przy użyciu opcji `-Ku`. Programy zawierające japońskie znaki w kodowaniu EUC-JP lub SJIS można uruchomić, wykorzystując opcję `-Ke` i `-Ks`.

Ruby 1.9 również obsługuje opcję `-K`, ale nie jest ona już preferowanym sposobem określania kodowania pliku z programem. Zamiast zmuszać użytkownika skryptu do określenia kodowania w trakcie uruchamiania Ruby, twórca skryptu może je określić za pomocą specjalnego komentarza znajdującego się na początku pliku[1]. Na przykład:

```
# coding: utf-8
```

Komentarz musi składać się wyłącznie ze znaków ASCII i zawierać słowo `coding` z dwukropkiem lub znakiem równości, po którym znajduje się nazwa wybranego kodowania (nie może ona zawierać spacji ani znaków interpunkcyjnych z wyjątkiem myślnika i znaku podkreślenia). Białe znaki mogą znajdować się po obu stronach dwukropka lub znaku równości, a przed łańcuchem `coding` może znajdować się dowolny przedrostek, jak `en`. W całym tym komentarzu, włącznie ze słowem `coding` i nazwą kodowania, nie są rozróżniane wielkie i małe litery, a więc może on być pisany zarówno małymi, jak i wielkimi literami.

Komentarze kodowania zazwyczaj zawierają także informację o kodowaniu dla edytora tekstowego. Użytkownicy edytora Emacs mogliby napisać:

```
# -*- coding: utf-8 -*-
```

A użytkownicy programu vi:

```
# vi: set fileencoding=utf-8 :
```

Tego typu komentarz kodowania zazwyczaj może znajdować się tylko w pierwszej linijce pliku. Wyjątkiem jest sytuacja, gdy pierwsza linijka jest zajęta przez komentarz shebang (który umożliwia wykonanie skryptu w systemach uniksowych). Wówczas kodowanie może znajdować się w drugiej linijce.

```
#!/usr/bin/ruby -w
# coding: utf-8
```

W nazwach kodowania nie są rozróżniane wielkie i małe litery, a więc nazwy można pisać w dowolny sposób, także mieszając małe litery z wielkimi. Ruby 1.9 obsługuje następujące kodowania źródła: ASCII-8BIT (inna nazwa to BINARY), US-ASCII (7-bit ASCII), kodowania europejskie ISO-8859-1 do ISO-8859-15, Unicode UTF-8 oraz japońskie SHIFT_JIS (inaczej SJIS) i EUC-JP. Konkretne kompilacje lub dystrybucje Ruby mogą obsługiwać także dodatkowe kodowania.

Pliki zakodowane w systemie UTF-8 identyfikują swoje kodowanie, jeśli ich trzy pierwsze bajty to 0xEF 0xBB 0xBF. Bajty te nazywane są BOM (*Byte Order Mark* — znacznik kolejności bajtów) i nie są obowiązkowe w plikach UTF-8 (niektóre programy działające w systemie Windows dodają te bajty przy zapisywaniu plików Unicode).

---

[1] W tej kwestii Ruby wykorzystuje konwencję z języka Python; zobacz *http://www.python.org/dev/peps/pep-0263/*.

W Ruby 1.9 słowo kluczowe `__ENCODING__` (na początku i końcu znajdują się dwa znaki podkreślenia) ewaluuje do kodowania źródła aktualnie wykonywanego kodu. Powstała w wyniku tego wartość jest obiektem klasy `Encoding` (więcej na temat klasy `Encoding` znajduje się w podrozdziale 3.2.6.2).

2.4.2. Kodowanie źródła i domyślne kodowanie zewnętrzne

W Ruby 1.9 ważna jest różnica pomiędzy kodowaniem źródła pliku Ruby a **domyślnym kodowaniem zewnętrznym** procesu Ruby. Kodowanie źródła jest tym, co zostało opisane wcześniej — informuje interpreter, jak odczytywać znaki w skrypcie. Jest ono zazwyczaj ustawiane za pomocą komentarzy kodowania. Program Ruby może składać się z więcej niż jednego pliku, a każdy z nich może mieć inne kodowanie źródła. Kodowanie źródła pliku wpływa na kodowanie literałów łańcuchowych w tym pliku. Więcej informacji na temat kodowania łańcuchów znajduje się w podrozdziale 3.2.6.

Domyślne kodowanie zewnętrzne to coś innego — jest ono używane przez Ruby podczas odczytu z plików i strumieni. Obejmuje cały proces Ruby i nie może zmieniać się od pliku do pliku. W typowych warunkach jest ustawiane na podstawie lokalizacji, na którą ustawiony jest komputer. Można to jednak zmienić, używając odpowiednich opcji wiersza poleceń, o czym za chwilę. Domyślne kodowanie zewnętrzne nie wpływa na kodowanie literałów łańcuchowych, ale ma duże znaczenie dla operacji wejścia i wyjścia; więcej informacji na ten temat znajduje się w podrozdziale 9.7.2.

Wcześniej podana została opcja `-K` jako sposób na określenie kodowania źródła. W rzeczywistości ustawia ona domyślne kodowanie zewnętrzne procesu i stosuje je jako domyślne kodowanie źródła.

W Ruby 1.9 opcja `-K` jest dostępna ze względu na zgodność z Ruby 1.8, ale nie jest już zalecanym sposobem ustawiania kodowania zewnętrznego. Dwie nowe opcje `-E` i `--encoding` pozwalają na określenie kodowania za pomocą jego pełnej nazwy zamiast jednoliterowego skrótu. Na przykład:

```
ruby -E utf-8              # Nazwa kodowania po opcji -E.
ruby -Eutf-8               # Spacja jest opcjonalna.
ruby --encoding utf-8      # Nazwa kodowania po opcji --encoding ze spacją.
ruby --encoding=utf-8      # Po opcji --encoding można wstawić znak równości.
```

Wszystkie szczegóły na ten temat znajdują się w podrozdziale 10.1.

Domyślne kodowanie zewnętrzne można sprawdzić, wykorzystując klasową metodę `Encoding.default_external`. Zwraca ona obiekt typu `Encoding`. Aby sprawdzić nazwę (jako łańcuch) kodowania znaków pochodzącego od lokalizacji, należy użyć metody `Encoding.locale_charmap`; zawsze bazuje ona na ustawieniach dotyczących lokalizacji i ignoruje opcje wiersza poleceń zmieniające domyślne kodowanie zewnętrzne.

2.5. Wykonywanie programu

Ruby to język skryptowy. Oznacza to, że programy Ruby są listami lub skryptami poleceń do wykonania. Domyślnie polecenia są wykonywane sekwencyjnie w takiej kolejności, w jakiej zostały napisane. Instrukcje sterujące (opisane w rozdziale 5.) zmieniają tę domyślną kolejność, pozwalając na warunkowe wykonywanie niektórych instrukcji lub wielokrotne ich powtarzanie.

Programiści przyzwyczajeni do tradycyjnych statycznych języków kompilowanych, jak C lub Java, mogą być nieco zbici z tropu. W Ruby nie ma specjalnej metody `main`, od której zaczyna się wykonywanie. Interpreter Ruby otrzymuje skrypt instrukcji do wykonania i zaczyna je wykonywać od pierwszego do ostatniego wiersza.

Ostatnie zdanie nie jest jednak do końca prawdziwe, ponieważ interpreter najpierw przeszukuje plik w celu znalezienia instrukcji `BEGIN`, których kod wykonuje najpierw. Następnie wraca do pierwszego wiersza i zaczyna wykonywanie sekwencyjne. Więcej na temat instrukcji `BEGIN` znajduje się w podrozdziale 5.7.

Inna różnica pomiędzy językiem Ruby a językami kompilowanymi dotyczy definicji modułów, klas i metod. W językach kompilowanych są to struktury syntaktyczne przetwarzane przez kompilator. W Ruby są to instrukcje jak wszystkie inne. Kiedy interpreter Ruby napotka definicję klasy, wykonuje ją, powodując powstanie nowej klasy. Podobnie kiedy napotka definicję metody, wykonuje ją, powodując powstanie nowej metody. W dalszej części programu interpreter najprawdopodobniej znajdzie wyrażenie wywołujące tę metodę, które spowoduje wykonanie instrukcji zawartych w ciele tej metody.

Interpreter Ruby wywoływany jest w wierszu poleceń poprzez podanie mu skryptu do wykonania. Bardzo proste jednowierszowe skrypty wpisuje się czasami bezpośrednio w wierszu poleceń. Częściej jednak podaje się nazwę pliku zawierającego skrypt. Interpreter Ruby odczytuje plik i wykonuje znajdujący się w nim skrypt. Najpierw wykonuje bloki `BEGIN`. Później przechodzi do pierwszego wiersza pliku i działa, dopóki nie wystąpi jedna z poniższych sytuacji:

- Interpreter wykona polecenie, które spowoduje zakończenie programu.
- Dojdzie do końca pliku.
- Dojdzie do wiersza oznaczającego logiczny koniec pliku za pomocą tokenu `_END_`.

Przed zakończeniem działania interpreter Ruby zazwyczaj (jeśli nie została wywołana metoda `exit!`) wykonuje ciała wszystkich znalezionych instrukcji `END` oraz pozostały kod zamknięcia zarejestrowany za pomocą funkcji `at_exit`.