Perl. Wprowadzenie. Wydanie IV

Autorzy: Randal L. Schwartz, Tom Phoenix, Brian d foy
Tłumaczenie: Rafal Szpoton
ISBN: 83-246-0268-2
Tytuł oryginału: Learning Perl, 4th Edition
Format: B5, stron: 280

Perl to jeden z pierwszych języków służących do tworzenia skryptów wykonywanych po stronie serwera internetowego. Był i jest wykorzystywany przez programistów aplikacji internetowych do realizacji zadań związanych z przetwarzaniem danych tekstowych. Mimo rozwoju innych technologii Perl nadal zachowuje swoją popularność. Grono użytkowników Perla powiększa się, twórcy języka wciąż pracują nad jego udoskonalaniem, a ilość materiałów dostępnych w sieci gwarantuje, że żadne pytanie programisty nie pozostanie bez odpowiedzi.

Książka „Perl. Wprowadzenie. Wydanie IV" to przewodnik dla początkujących programistów aplikacji internetowych, zawierający wszystkie informacje niezbędne do rozpoczęcia pracy i tworzenia programów w języku Perl. Przedstawia zarówno zagadnienia podstawowe – typy danych i struktury języka, jak i metody przetwarzania danych tekstowych za pomocą wyrażeń regularnych, sposoby korzystania z tablic asocjacyjnych i manipulowania plikami znajdującymi się na dysku. Opisuje również zasady pracy z modułami zawierającymi dodatkowe funkcje, z których można korzystać podczas pisania aplikacji.

- Skalarne typy danych
- Wyrażenia warunkowe
- Przypisywanie wartości do zmiennych
- Korzystanie z tablic i list
- Definiowanie i stosowanie procedur
- Operacje wejścia i wyjścia
- Korzystanie z wyrażeń regularnych do dopasowywania wzorców i przetwarzania tekstów
- Sortowanie danych
- Instalowanie i stosowanie modułów

Ta książka to doskonały początek przygody z Perlem

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 2.

Dane skalarne

W języku angielskim, podobnie jak w wielu innych językach występuje rozróżnienie pomiędzy liczbą pojedynczą a liczbą mnogą. Podobnie jest również w Perlu, będącym językiem komputerowym, zaprojektowanym przez lingwistów. Można przyjąć ogólną zasadę, iż w przypadku, gdy Perl dysponuje jakąś pojedynczą jednostką, będzie ona określana jako *dana skalarna*[1]. Jest ona najprostszym rodzajem danych wykorzystywanym w tym języku. Większość skalarów stanowią liczby (na przykład 255 lub 3.25e20) bądź łańcuchy znakowe (na przykład `witaj`[2] lub też adres w Warszawie). Dlatego też, chociaż liczby oraz łańcuchy można rozpatrywać niezależnie, Perl używa ich niemal wymiennie.

Wartość typu skalarnego może być wykorzystywana w połączeniu z operatorami (jak na przykład z operatorem dodawania lub łączenia), dając w rezultacie wynik skalarny. Wartość tego typu może być przechowywana w zmiennej skalarnej, jak również odczytywana z plików i urządzeń, a także przy ich użyciu zapisywana.

Liczby

Chociaż dane skalarne są reprezentowane najczęściej przez liczbę lub łańcuch znakowy, to jednak w tej chwili pomocne okaże się ich oddzielne omówienie. W pierwszej kolejności opisane zostaną liczby, a następnie łańcuchy znakowe.

Wszystkie liczby mają ten sam format wewnętrzny

Jak przekonamy się w kolejnych kilku podrozdziałach, w Perlu możliwe jest zdefiniowanie liczb całkowitych (jak na przykład 255 lub 2001) oraz zmiennoprzecinkowych (liczby rzeczywiste z częścią dziesiętną, jak na przykład 3,14159 lub $1{,}35 \times 10^{25}$). Niemniej jednak wewnętrzne operacje w tym języku wykonywane są przy użyciu wartości zmiennoprzecinkowych

[1] W skrócie skalar — *przyp. tłum.* Termin ten ma niewiele wspólnego z podobnym określeniem używanym w matematyce i fizyce, poza tym, że określa on pojedynczą rzecz. W Perlu nie występują wektory.

[2] Dysponując doświadczeniami wyniesionymi z używania innych języków programowania, można sądzić, że łańcuch znakowy `witaj` jest zbiorem pięciu znaków, a nie pojedynczą całością. Niemniej jednak w Perlu łańcuch znakowy jest pojedynczą wartością skalarną. W przypadku istnienia takiej potrzeby można oczywiście odwoływać się do pojedynczych znaków. Sposób ten zostanie omówiony w kolejnych rozdziałach.

o podwójnej precyzji[3]. Oznacza to, że Perl nie dysponuje wewnętrznie wartościami całkowitymi. Stała typu całkowitego występująca w programie traktowana jest jak równoważna jej wartość zmiennoprzecinkowa[4]. Programista prawdopodobnie nie zauważy tej konwersji (lub nie będzie się nią przejmować). Powinien jednak zaprzestać poszukiwania osobnych operacji całkowitoliczbowych (w przeciwieństwie do operacji *zmiennoprzecinkowych*), ponieważ one nie istnieją[5].

Literały zmiennoprzecinkowe

Literał określa sposób, w jaki wartość reprezentowana jest w kodzie źródłowym programu w języku Perl. Nie jest on wynikiem działania lub też operacji wejścia-wyjścia. Literał jest daną zapisaną bezpośrednio w kodzie źródłowym.

Literały zmiennoprzecinkowe występujące w Perlu powinny wyglądać znajomo. Dozwolone jest stosowanie liczb z częścią dziesiętną lub bez niej (również z opcjonalnym przedrostkiem plus lub minus, określającym znak liczby), jak też w notacji naukowej o podstawie 10 (notacji wykładniczej).

```
1.25
255.000
255.0
7.25e45      # 7,25 razy 10 do potęgi 45 (dość duża liczba)
-6.5e24      # -6,5 razy 10 do potęgi 24 (duża liczba ujemna)
-12e-24      # -12 razy 10 do potęgi -24 (bardzo mała liczba ujemna)
-1.2E-23     # Inny sposób zapisu liczby powyżej — litera E może być wielka
```

Literały całkowitoliczbowe

Literały tego typu są całkiem proste:

```
0
2001
-40
255
61298040283768
```

Ostatni z przykładów może być trudny do odczytania. Perl pozwala dla zwiększenia przejrzystości stosować znaki podkreślenia. Umożliwia to zapis poprzedniego przykładu w następujący sposób:

```
61_298_040_283_768
```

[3] Wartość zmiennoprzecinkowa o podwójnej precyzji oznacza jakąkolwiek wartość użytą dla potrzeb deklaracji typu `double` przez kompilator języka C, wykorzystany do kompilacji Perla. Chociaż jej rozmiar może różnić się pomiędzy komputerami, to jednak większość nowoczesnych systemów używa standardu IEEE-754, zalecającego stosowanie 15 cyfr precyzji oraz zakresu przynajmniej od `1e-100` do `1e100`.

[4] Mówiąc ściśle, Perl używa niekiedy wewnętrznie liczb całkowitych w sposób niewidoczny dla programisty. Oznacza to, że jedyną różnicą, którą programista jest w stanie zauważyć, jest szybsze działanie jego programu. A któż by się na to uskarżał?

[5] Istnieje wprawdzie dyrektywa `integer`, jednak opis jej zastosowania wykracza poza zakres tej książki. Również niektóre operacje powodują obliczenie liczby całkowitej w oparciu o podane liczby zmiennoprzecinkowe, lecz nie zostaną one omówione w tym miejscu.

Chociaż jest to ta sama wartość liczbowa, dla ludzi wygląda ona inaczej. Można zastanawiać się, czy w tym celu nie powinny zostać użyte znaki przecinka[6], jednak znaki te w Perlu używane są do ważniejszych celów (jak przekonamy się o tym w kolejnym rozdziale).

Niedziesiętne literały całkowitoliczbowe

Podobnie jak wiele innych języków programowania, również Perl umożliwia stosowanie liczb w systemach o podstawach innych niż 10 (system dziesiętny). Literały zapisane w systemie ósemkowym (o podstawie 8) rozpoczynają się cyfrą `0`, w systemie szesnastkowym (o podstawie 16) ciągiem `0x`, zaś w systemie dwójkowym (o podstawie 2) ciągiem `0b`[7]. Cyfry szesnastkowe od `A` do `F` (lub od `a` do `f`) reprezentują standardowe wartości liczbowe od 10 do 15:

```
0377            # Liczba 377 w systemie ósemkowym, równoważna liczbie 255 w systemie dziesiętnym
0xff            # Liczba FF w systemie szesnastkowym, równoważna liczbie 255 w systemie dziesiętnym
0b11111111      # Również liczba 255 w systemie dziesiętnym
```

Chociaż dla człowieka wartości te wydają się różnić, to w Perlu wszystkie trzy oznaczają tę samą liczbę. Nie ma znaczenia, czy zostanie zapisana liczba `0xFF`, czy też `255.000`, dlatego też należy wybierać reprezentację mającą największy sens dla programisty oraz osoby odpowiedzialnej za utrzymanie kodu (przez którą rozumiemy biedaka usiłującego bezskutecznie zrozumieć, co autor kodu miał na myśli. Najczęściej tym biednym człowiekiem jest sam twórca programu, który nie może sobie przypomnieć, co robił trzy miesiące wcześniej).

W przypadku gdy literał niedziesiętny jest dłuższy niż cztery znaki, może być trudny do odczytania. W celu zwiększenia przejrzystości takich literałów Perl umożliwia stosowanie znaków podkreślenia:

```
0x1377_0B77
0x50_65_72_7C
```

Operatory numeryczne

Perl obsługuje typowe zwykłe operatory dodawania, odejmowania, mnożenia oraz dzielenia:

```
2 + 3           # 2 plus 3 lub inaczej 5
5.1 - 2.4       # 5,1 minus 2,4 lub inaczej 2,7
3 * 12          # 3 razy 12 równa się 36
14 / 2          # 14 dzielone przez 2 lub inaczej 7
10.2 / 0.3      # 10,2 dzielone przez 0,3 lub inaczej 34
10 / 3          # Zawsze dzielenie zmiennoprzecinkowe, więc daje w rezultacie 3,3333333...
```

Obsługuje również operator dzielenia modulo (`%`). Wartością wyrażenia `10 % 3` jest reszta z dzielenia 10 przez 3, równa 1. Oba argumenty operatora są najpierw konwertowane do wartości całkowitych, dlatego też wyrażenie `10.5 % 3.2` jest obliczane jako `10 % 3`[8]. Dodatkowo

[6] W językach anglosaskich do rozdzielania części tysięcznych używany jest znak przecinka, zaś do oddzielania części dziesiętnej — znak kropki, odwrotnie niż w języku polskim — *przyp. tłum.*

[7] Zapis przy użyciu początkowej cyfry zero działa jedynie w przypadku literałów, nie działa natomiast w przypadku automatycznej konwersji łańcucha znakowego do liczby, opisanej w dalszej części tego rozdziału. Przekonwertowanie łańcucha danych przypominającego wartość ósemkową lub szesnastkową do postaci liczbowej możliwe jest przy użyciu funkcji `oct()` lub `hex()`. Niemniej jednak nie istnieje odpowiednik funkcji `"bin"`, służący do konwersji wartości dwójkowych. Łańcuchy znakowe rozpoczynające się od `0b` mogą zostać przekonwertowane przy użyciu funkcji `oct()`.

[8] Wynik działania dzielenia modulo w przypadku, gdy w charakterze argumentu(ów) została użyta liczba ujemna, może różnić się w różnych implementacjach Perla. Należy zachować więc ostrożność.

Perl udostępnia podobny do zdefiniowanego w języku Fortran operator potęgowania, którego brakowało w Pascalu oraz C. Operator ten jest reprezentowany przez podwójną gwiazdkę, jak na przykład w wyrażeniu `2**3`, oznaczającym dwa do potęgi trzeciej, czyli osiem[9]. Inne operatory numeryczne zostaną przedstawione w miarę potrzeb.

Łańcuchy

Łańcuchy stanowią sekwencje znaków (na przykład `witaj`). Mogą one zawierać dowolną kombinację dowolnych znaków[10]. Najkrótszy możliwy łańcuch nie zawiera żadnych znaków, natomiast najdłuższy wypełnia cały dostępny obszar pamięci, dlatego też nie będzie można z nim już nic więcej zrobić. Takie działanie jest zgodne z zasadą „braku ograniczeń", przestrzeganą przez Perla przy każdej okazji. Typowe łańcuchy zbudowane są z możliwych do wydrukowania sekwencji znaków, cyfr oraz znaków przestankowych, należących do zakresu kodów w tabeli ASCII od 32 do 126. Niemniej jednak możliwość umieszczania w łańcuchach Perla dowolnych znaków oznacza, że programista jest w stanie utworzyć łańcuchy zawierające nieprzetworzone dane binarne, a następnie takie łańcuchy przeszukiwać i obrabiać. Wiele innych narzędzi miałoby duże problemy z wykonaniem podobnej czynności. I tak na przykład Perl umożliwia modyfikację obrazka lub skompilowanego programu przez wczytanie go do łańcucha, wykonanie zmiany, a następnie ponowne zapisanie wyniku operacji.

Podobnie do liczb, również łańcuchy dysponują reprezentacją literalną, używaną do ich zapisywania w programie w Perlu. Literały łańcuchowe dzielą się na dwa rodzaje: *literały łańcuchowe w pojedynczych cudzysłowach* oraz *literały łańcuchowe w podwójnych cudzysłowach*.

Literały łańcuchowe w pojedynczych cudzysłowach

Literał łańcuchowy tego rodzaju jest sekwencją znaków umieszczonych w pojedynczych cudzysłowach. Cudzysłowy te nie stanowią części samego łańcucha, lecz umożliwiają Perlowi określenie początku oraz końca łańcucha. Jakikolwiek znak różny od znaku pojedynczego cudzysłowu lub znaku lewego ukośnika umieszczony pomiędzy znakami cudzysłowów (włączając również znaki nowego wiersza w przypadku gdy łańcuch rozciąga się na kilka kolejnych wierszy) należy do łańcucha. Aby umieścić w łańcuchu znak lewego ukośnika, należy użyć dwóch takich znaków umieszczonych kolejno po sobie w tym samym wierszu. Aby natomiast dodać do łańcucha znak pojedynczego cudzysłowu, należy poprzedzić go znakiem lewego ukośnika.

```
'adam'      # Łańcuch zawierający cztery znaki: a, d, a oraz m
'krzysztof' # Łańcuch zawierający dziewięć znaków
''          # Łańcuch pusty (brak znaków)
'Nie pozwól Comte\'owi przerwać zdania apostrofem!'
'Ostatnim znakiem tego łańcucha jest lewy ukośnik: \\'
'witaj\n'   # Witaj, a za nim lewy ukośnik i n
'witaj
wiec'       # Witaj i znak nowego wiersza (razem 10 znaków)
'\'\\'      # Pojedynczy znak cudzysłowu, a za nim znak lewego ukośnika
```

[9] W zwyczajny sposób nie można podnieść liczby ujemnej do potęgi niecałkowitej. Matematycy wiedzą, że wynik byłby liczbą złożoną. W celu umożliwienia takich obliczeń konieczne jest użycie modułu *Math::Complex*.

[10] W przeciwieństwie do języków C lub C++ w języku Perl nie występuje specjalny problem znaku NUL, ponieważ w celu określenia końca łańcucha Perl, zamiast poszukiwać bajtu null, zlicza jego długość.

Ciąg `\n` umieszczony wewnątrz łańcucha w pojedynczych cudzysłowach nie jest interpretowany jako znak nowego wiersza, lecz jako dwa oddzielne znaki: lewego ukośnika oraz `n`. W tego rodzaju łańcuchach specjalne znaczenie ma jedynie znak lewego ukośnika, jeżeli występuje za nim drugi znak lewego ukośnika lub pojedynczy cudzysłów.

Literały łańcuchowe w podwójnych cudzysłowach

Literał tego rodzaju przypomina łańcuchy występujące w innych językach programowania. Jest on również sekwencją znaków, chociaż w tym przypadku umieszczoną w podwójnych cudzysłowach. Tym razem jednak znak lewego ukośnika odsłania swoje potężne możliwości pod względem definiowania określonych znaków sterujących lub dowolnych innych znaków przez użycie ich reprezentacji w systemie ósemkowym lub szesnastkowym. Oto kilka przykładów łańcuchów w podwójnych cudzysłowach:

```
"krzysztoƒ"          # Łańcuch identyczny z 'krzysztof'
"witaj świecie\n"    # Łańcuch witaj świecie oraz znak nowego wiersza
"Ostatni znak w tym łańcuchu jest cudzysłowem: \""
"cola\tsprite"       # Cola, znak tabulacji oraz sprite
```

Literał `"krzysztoƒ"` w podwójnych cudzysłowach oznacza w Perlu dokładnie ten sam sześcioznakowy łańcuch co literał `'krzysztoƒ'` w pojedynczych cudzysłowach. Przypomina to literały liczbowe, w przypadku których literał `0377` był jedynie innym sposobem zapisania liczby `255.0`. Perl umożliwia zapis literałów w sposób najbardziej logiczny dla programisty. Jeżeli jednak zamierza on używać znaku lewego ukośnika (na przykład w celu traktowania ciągu `\n` jako znaku nowego wiersza), będzie zmuszony użyć cudzysłowów podwójnych.

Znak lewego ukośnika może poprzedzać różne znaki, modyfikując tym samym ich znaczenie (jest więc ogólnie nazywany znakiem sterującym). Prawie kompletna[11] lista znaków sterujących, możliwych do wykorzystania w łańcuchach znakowych w podwójnych cudzysłowach umieszczona jest w tabeli 2.1.

Kolejną cechą łańcuchów w podwójnych cudzysłowach jest *interpolacja zmiennych* (ang. *variable interpolation*), oznaczająca zastępowanie w chwili użycia łańcucha umieszczonych wewnątrz niego nazw zmiennych za pomocą ich rzeczywistych wartości. Ponieważ zmienne nie zostały jeszcze formalnie przedstawione, do zagadnienia tego powrócimy później.

Operatory łańcuchowe

Wartości łańcuchów mogą być ze sobą łączone za pomocą operatora `.` (pojedynczej kropki). Nie powoduje to zmiany żadnego z łańcuchów, podobnie jak `2+3` nie zmienia ani wartości `2`, ani `3`. Wynikowy łańcuch (dłuższy) może być następnie poddany dalszym obliczeniom lub przypisany do zmiennej:

```
"witaj" . "świecie"        # Wynik identyczny z "witajświecie"
"witaj" . ' ' . "świecie"  # Wynik identyczny z 'witaj świecie'
'witaj świecie' . "\n"     # Wynik identyczny z "witaj świecie\n"
```

W przeciwieństwie do innych języków programowania, w których w celu połączenia łańcuchów wystarczy jedynie umieścić dwie wartości obok siebie, w Perlu operacja taka musi zostać wyraźnie oznaczona za pomocą operatora `.`.

[11] W ostatnich wersjach języka Perl zostały wprowadzone znaki sterujące Unicode, które nie zostaną w tym miejscu omówione.

Tabela 2.1. Znaki sterujące do użycia w łańcuchach w podwójnych cudzysłowach

Znak	Znaczenie
`\n`	znak nowego wiersza
`\r`	znak powrotu karetki
`\t`	znak tabulacji
`\f`	znak wysunięcia strony (ang. *formfeed*)
`\b`	znak sterujący *backspace*
`\a`	dzwonek
`\e`	znak sterujący *escape*
`\OO7`	dowolna wartość ASCII zapisana w systemie ósemkowym (w tym przypadku `OO7` — dzwonek)
`\x7f`	dowolna wartość ASCII zapisana w systemie szesnastkowym (w tym przypadku `7f` — znak sterujący *delete*)
`\cC`	znak sterujący *Ctrl* (w tym przypadku *Ctrl+C*)
`\\`	znak lewego ukośnika
`\"`	znak podwójnego cudzysłowu
`\l`	kolejny znak będzie małą literą
`\L`	wszystkie kolejne znaki aż do `\E` będą małymi literami
`\u`	kolejny znak będzie wielką literą
`\U`	wszystkie kolejne znaki aż do `\E` będą wielkimi literami
`\Q`	poprzedza wszystkie kolejne znaki lewym ukośnikiem aż do wystąpienia `\E`
`\E`	kończy działanie `\L`, `\U` lub `\Q`

Specjalnym operatorem łańcuchowym jest *operator zwielokrotnienia łańcucha*, oznaczany za pomocą pojedynczej małej litery `x`. Operator ten pobiera swój lewy argument (będący łańcuchem) i tworzy tyle złączeń tego łańcucha, na ile wskazuje argument umieszczony po prawej stronie (liczba):

```
"adam" x 3           # "adamadamadam"
"krzysztof" x (4+1)  # "krzysztof" x 5 czyli "krzysztofkrzysztofkrzysztofkrzysztofkrzysztof"
5 x 4                # jest w rzeczywistości interpretowany jako "5" x 4 co daje "5555"
```

Ostatni przykład wart jest wyjaśnienia. Operator zwielokrotnienia łańcucha w charakterze lewego argumentu wymaga podania łańcucha, dlatego też liczba `5` jest konwertowana do postaci `"5"` (przy użyciu reguł, które zostaną opisane w dalszej części rozdziału), dając w wyniku łańcuch jednoznakowy. Ten nowy łańcuch jest następnie kopiowany cztery razy do postaci `5555`. W przypadku zamiany argumentów miejscami, to znaczy użycia wyrażenia `4 x 5`, wynikiem operacji byłby łańcuch złożony z pięciu kopii łańcucha `4`, to znaczy `44444`. Przykład ten dowodzi, że operacja zwielokrotniania łańcucha nie jest operacją przestawną.

Argument określający liczbę kopii łańcucha (prawy argument) jest w pierwszej kolejności obcinany do liczby całkowitej (4.8 staje się 4). Jeżeli natomiast argument ten ma wartość mniejszą niż jeden, wtedy wynikiem operacji jest łańcuch pusty (o długości zerowej).

Automatyczna konwersja pomiędzy liczbami a łańcuchami

W przypadku istnienia takiej konieczności Perl w większości przypadków dokonuje automatycznej konwersji pomiędzy liczbami a łańcuchami. Skąd wie on zatem, czy wymagana jest

liczba, czy łańcuch? Wszystko zależy od operatora użytego z wartością skalarną. Jeżeli operator ten oczekuje liczby (jak chociażby operator +), wtedy Perl potraktuje wartość jak liczbę. Jeżeli natomiast oczekuje on łańcucha (jak operator .), wtedy Perl potraktuje wartość jak łańcuch. Programista nie musi martwić się o różnicę pomiędzy liczbami a łańcuchami. Wystarczy, że użyje właściwych operatorów, a Perl wykona całą pracę.

W sytuacji gdy wraz z operatorem oczekującym podania liczby (weźmy na przykład operator mnożenia) użyta zostanie wartość będąca łańcuchem, Perl dokona automatycznej konwersji łańcucha do postaci odpowiadającej mu wartości liczbowej, traktując łańcuch jak zmiennoprzecinkową wartość liczbową, zapisaną w systemie dziesiętnym[12]. Dlatego też `"12" * "3"` daje w wyniku wartość `36`. Część łańcucha począwszy od pierwszego znaku niebędącego cyfrą oraz poprzedzające łańcuch odstępy są ignorowane. Dlatego też wyrażenie `"12adam34" * " 3"` będzie mieć wartość `36` i zostanie obliczone przez Perla bez żadnych protestów z jego strony[13]. W ekstremalnym przypadku łańcuch, który nie zawiera liczby, jest konwertowany do wartości zero. Taka sytuacja miałaby miejsce w przypadku użycia w charakterze liczby łańcucha `"adam"`.

Analogicznie, jeżeli wartość liczbowa zostanie użyta w miejscu, w którym oczekiwany jest łańcuch (na przykład w przypadku łączenia łańcuchów), wartość numeryczna zamieniana jest w łańcuch, który daną liczbę reprezentuje. Na przykład w przypadku, gdybyśmy chcieli połączyć łańcuch `"Z"` z wynikiem operacji mnożenia 5 i 7[14], można byłoby to zapisać w następujący sposób:

```
"Z" . 5 * 7 # Zapis równoważny "Z" . 35 lub "Z35"
```

Inaczej mówiąc, programista nie musi martwić się, czy używa liczby, czy łańcucha (przynajmniej przez większość czasu). Perl wykonuje bowiem wszystkie konwersje za niego[15].

Wbudowane ostrzeżenia Perla

W przypadku podejrzanego zachowania się programu programista może nakazać Perlowi wyświetlanie komunikatów ostrzeżeń. Aby uruchomić program z włączonymi komunikatami tego rodzaju, należy w wierszu poleceń użyć opcji `-w`:

```
$ perl -w moj_program
```

Innym sposobem jest stałe nakazanie wyświetlania komunikatów ostrzeżeń w pierwszym wierszu programu rozpoczynającym się od `#!`:

```
#!/usr/bin/perl -w
```

Ten zapis działa poprawnie nawet w przypadku systemów innych niż Unix, w których wiersz ten jest podawany jedynie z przyzwyczajenia, jako że ścieżka do Perla nie jest istotna:

```
#!perl -w
```

---

[12] Sztuczka z wykorzystaniem początkowego zera w celu zdefiniowania wartości niedziesiętnej działa w przypadku literałów, jednak zawodzi w przypadku konwersji automatycznej. Z takimi łańcuchami należy użyć funkcji `hex()` lub `oct()`.

[13] O ile programista nie zażyczy sobie wyświetlania komunikatów ostrzeżeń, które za chwilę zostaną omówione.

[14] Wkrótce omówione zostaną zasady pierwszeństwa operatorów oraz nawiasów.

[15] Nie należy również martwić się wydajnością takiej konwersji. Perl zazwyczaj zapamiętuje wynik konwersji, co sprawia, że taka konwersja wykonywana jest jedynie raz.

W przypadku wersji 5.6 Perla lub nowszych komunikaty z ostrzeżeniami można włączyć za pomocą dyrektywy. Należy jednak zachować ostrożność, ponieważ ten sposób nie zadziała u osób używających wcześniejszych wersji Perla[16].

```
#!/usr/bin/perl
use warnings;
```

Od tej chwili w przypadku próby użycia łańcucha `'12adam34'` w charakterze liczby Perl wyświetli komunikat z ostrzeżeniem.

```
Argument '12adam34' isn't numeric
```

Oczywiście komunikaty z ostrzeżeniami przeznaczone są dla programistów, a nie użytkowników programu. Dlatego też jeżeli programista nie chce ujrzeć ostrzeżenia, nie zda się ono prawdopodobnie na nic. Ostrzeżenia te nie zmienią również działania programu, tyle że od czasu do czasu będzie pojawiać się tego rodzaju komunikat. Jeśli pojawi się komunikat z niezrozumiałym ostrzeżeniem, dłuższy opis problemu można uzyskać, używając dyrektywy `diagnostic`. Strona z dokumentacją do `perldiag` zawiera krótki komunikat z ostrzeżeniem oraz dłuższy opis diagnostyczny.

```
#!/usr/bin/perl
use diagnostic;
```

Po dodaniu do programu dyrektywy `use diagnostic` może wydawać się, że po uruchomieniu programu wstrzymuje on na chwilę swoje działanie. Dzieje się tak, ponieważ wykonuje on w tym czasie wiele innych czynności (i zużywa mnóstwo pamięci), przygotowując się na wypadek sytuacji, w której programista chciałby po wystąpieniu błędu zapoznać się z jego dokumentacją. W przypadku gdy programista nie potrzebuje już dłużej otrzymywać informacji o komunikatach ostrzeżeń wygenerowanych przez program, należy usunąć dyrektywę `use diagnostic`. Spowoduje to zoptymalizowanie programu i przyspieszenie jego uruchamiania (oraz zmniejszenie zużycia pamięci) bez negatywnego wpływu na użytkowników. Chociaż lepiej byłoby, gdyby program został poprawiony w ten sposób, aby nie powodował już więcej generowania komunikatów o błędach, to jednak samo usunięcie dyrektywy wystarczy, aby Perl przestał wyświetlać na wyjściu komunikaty diagnostyczne.

Dodatkową optymalizacją może być wykorzystanie jednej z opcji stosowanej w wierszu poleceń Perla: `-M`. Powoduje ona automatyczne załadowanie dyrektywy `diagnostic` wtedy, gdy jest ona potrzebna, co pozwala uniknąć edytowania za każdym razem kodu źródłowego:

```
$perl -mdiagnostics ./moj_program
Argument "12adam34" isn't numeric in addition (+) at ./moj_program line 17 (#1)
(W numeric) The indicated string was fed as an argument to
an operator that expected a numeric value instead. If you're
fortunate the message will identify which operator was so unfortunate.
```

W momencie gdy w książce omawiane będą sytuacje, w których Perl zwyczajowo ostrzega programistę o pomyłkach w kodzie, zostanie to dodatkowo zasygnalizowane Czytelnikowi. Niemniej jednak nie powinno się liczyć na to, że tekst lub działanie dowolnego z ostrzeżeń nie ulegnie zmianie w kolejnych wersjach Perla.

[16] Użycie dyrektywy `warnings` umożliwia wyświetlenie komunikatów o błędach składniowych. Więcej informacji na ten temat zawiera strona z dokumentacją do *perllexwarn*.

Zmienne skalarne

Określenie *zmienna* używane jest w celu nazwania kontenera przechowującego jedną lub więcej wartości[17]. Nazwa zmiennej jest stała w całym programie, lecz wartość lub wartości w niej przechowywane zmieniają się zazwyczaj cyklicznie w trakcie wykonywania programu.

Zmienna skalarna, jak można tego oczekiwać, przechowuje pojedynczą wartość skalarną. Nazwy tego rodzaju zmiennych zaczynają się od znaku dolara, po którym następuje tzw. *identyfikator*, rozpoczynający się od litery lub znaku podkreślenia, po którym może występować więcej liter, cyfr lub znaków podkreślenia. Inaczej mówiąc, identyfikator złożony jest ze znaków alfanumerycznych oraz znaków podkreślenia, lecz nie może rozpoczynać się od cyfry. Litery wielkie i małe są rozróżniane. Zmienna `$Adam` różni się od zmiennej `$adam`. Co więcej, wszystkie litery, cyfry oraz znaki podkreślenia są znaczące:

```
$bardzo_dluga_zmienna_zakonczona_1
```

Zapis w powyższym wierszu różni się od poniższego:

```
$bardzo_dluga_zmienna_zakonczona_2
```

Do zmiennych skalarnych w Perlu należy zawsze odwoływać się za pomocą poprzedzającego je znaku `$`.. W programie powłoki systemu znak `$` służy do pobierania wartości zmiennej, lecz w przypadku jej przypisywania należy ten znak pominąć. W programach *awk* lub *C* znak `$` pomijany jest całkowicie. Dlatego też jeżeli programista używa tych języków wymiennie, może złapać się na przypadkowym używaniu błędnego zapisu. Należy się tego spodziewać. (Większość programistów Perla zaleciłaby całkowite zaprzestanie używania skryptów powłoki, *awk* lub języka *C*, lecz nie musi to odpowiadać Czytelnikowi).

Wybór dobrych nazw zmiennych

Ogólnie rzecz biorąc, należy wybierać takie nazwy zmiennych, aby znaczyły coś wskazującego na cel ich utworzenia. Na przykład nazwa `$r` nie jest prawdopodobnie tak bardzo opisowa jak `$dlugosc_wiersza`. Zmienna używana jedynie w dwóch lub trzech wierszach może nosić nazwę `$n` lub podobną, lecz zmienna stosowana w całym programie powinna prawdopodobnie być nazwana bardziej opisowo.

Analogicznie, również poprawnie stosowane znaki podkreślenia mogą uczynić nazwę zmiennej łatwiejszą do odczytania oraz zrozumienia, szczególnie jeżeli programista, który zajmuje się utrzymaniem kodu programu, używa innego języka ojczystego niż programista, który ten program tworzy. Na przykład `$ta_tamarka` jest lepszą nazwą niż `$tatamarka`, gdyż ta ostatnia może wyglądać jak `$tata_marka`. Czy nazwa `$stopid` oznacza `$sto_pid` (sto identyfikatorów procesów `pid`), `$stop_id` (identyfikator ID dla jakiegoś obiektu `"stop"`), czy też może powinna być interpretowana jako słowo innego rodzaju?

Większość nazw zmiennych w zaprezentowanych w tej książce programach w Perlu złożona jest z samych małych liter. W niewielu wyjątkowych przypadkach używane są wielkie litery. Stosowanie samych wielkich liter w nazwie (na przykład `$ARGV`) ogólnie mówiąc wskazuje,

[17] Jak zobaczymy wkrótce, zmienna skalarna może przechowywać jedynie jedną wartość. Jednak inne rodzaje zmiennych, takie jak tablice lub tablice asocjacyjne, mogą przechowywać wiele wartości.

że zmienna ma znaczenie specjalne. Gdy nazwa zmiennej składa się z więcej niż jednego wyrazu, niektórzy zapisują ją jako `$podkreslenia_sa_fajne`, podczas gdy inni zapisują ją jako `$uzyjWielkichLiterPoczatkowych`. Należy jedynie być konsekwentnym.

Oczywiście wybór dobrych lub złych nazw nie ma dla Perla znaczenia. Można na przykład nazwać trzy najważniejsze zmienne programu jako `$OOOoooOOO`, `$OOooOOoo` oraz `$OoOoOoOoO` i nie wywoła to protestów ze strony Perla. Niemniej jednak w takiej sytuacji nie należy prosić nikogo o zajmowanie się późniejszym utrzymaniem kodu programu.

Przypisania skalarne

Najczęstszą operacją wykonywaną przy użyciu zmiennych skalarnych jest przypisanie, będące sposobem nadania zmiennej wartości. Operatorem przypisania w Perlu jest znak równości (podobnie jak w innych językach programowania). Powoduje on przypisanie do nazwy zmiennej umieszczonej po lewej stronie operatora wartości wyrażenia znajdującego się po prawej stronie:

```
$fred   = 17;            # Nadaje zmiennej $fred wartość 17
$barney = 'witaj';       # Nadaje zmiennej $barney wartość równą pięcioznakowemu łańcuchowi 'witaj'
$barney = $fred + 3;     # Nadaje zmiennej $barney bieżącą wartość zmiennej $fred plus 3 (20)
$barney = $barney * 2;   # Zmienna $barney ma teraz poprzednią wartość zmiennej $barney pomnożoną przez 2 (40)
```

Warto zauważyć, że zmienna `$barney` w ostatnim wierszu użyta jest dwukrotnie: raz w celu pobrania jej wartości (po prawej stronie znaku równości) oraz drugi raz w celu zdefiniowania, gdzie obliczona wartość ma zostać umieszczona (po lewej stronie znaku równości). Takie przypisanie jest dozwolone, bezpieczne i raczej często spotykane. W rzeczywistości jest ono tak powszechne, że do jego zapisu można użyć wygodnego skrótu przedstawionego w kolejnym podrozdziale.

Dwuargumentowe operatory przypisania

Wyrażenia w rodzaju `$fred = $fred + 5` (w których ta sama zmienna pojawia się po obu stronach przypisania) pojawiają się w Perlu na tyle często, aby dysponował on (podobnie jak język C i Java) skrótem służącym do zmiany wartości zmiennej. Jest nim *dwuargumentowy operator przypisania*. Prawie wszystkie operatory dwuargumentowe używane do obliczeń wartości dysponują odpowiednikiem dwuargumentowego przypisania z dołączonym znakiem równości. Na przykład poniższe dwa wiersze są równoważne:

```
$fred = $fred + 5;      # Zapis bez dwuargumentowego operatora przypisania
$fred += 5;             # Zapis z dwuargumentowym operatorem przypisania
```

Poniższe zapisy także są równoważne:

```
$barney = $barney * 3;
$barney *= 3;
```

W każdym przypadku operator powoduje w pewien sposób zmianę wartości zmiennej, a nie nadpisuje jej wartości za pomocą wyniku nowego wyrażenia.

Kolejny popularny operator przypisania utworzony jest z operatora łączenia łańcuchów (`.`). Jest nim operator dołączania (`.=`):

```
$str = $str . " ";   # Dołącza spacje do $str
$str .= " ";         # Ten sam wynik uzyskany przy użyciu operatora przypisania
```

Ten zapis jest poprawny dla prawie wszystkich operatorów. Na przykład operator podniesienia do potęgi jest zdefiniowany następująco: **=. Dlatego też `$fred **= 3` oznacza: „podnieś liczbę znajdującą się w zmiennej `$fred` do potęgi trzeciej i wynik umieść ponownie w zmiennej `$fred`".

Wypisywanie danych za pomocą print

Wypisywanie danych wyjściowych przez program jest ogólnie mówiąc dobrym pomysłem. W przeciwnym razie ktoś mógłby pomyśleć sobie, że nie robi on nic. Czynność tę umożliwia operator `print()`. Pobiera on argument skalarny i umieszcza go w niezmienionej postaci na standardowym wyjściu programu. Jeżeli programista nie zmieni standardowych ustawień, będzie nim ekran komputera:

```
print "witaj świecie\n"; # Wyświetla napis witaj świecie i znak nowego wiersza
print "Odpowiedź brzmi ";
print 6 * 7;
print ".\n";
```

Możliwe jest również wypisanie serii wartości oddzielonych przecinkami:

```
print "Odpowiedź brzmi ",6 * 7,".\n";
```

Taka seria danych jest listą. Zostaną one omówione później.

Interpolacja zmiennych skalarnych w łańcuchach

Literały łańcuchowe w podwójnych cudzysłowach, oprócz obsługi znaków sterujących, podlegają również *interpolacji zmiennych*[18]. Oznacza to, że dowolna nazwa zmiennej skalarnej[19] wewnątrz łańcucha zastępowana jest za pomocą jej aktualnej wartości:

```
$meal = "stek z brontozaura";
$barney = "fred zjadł $meal";     # Zmienna $barney ma teraz wartość "fred zjadł stek z brontozaura"
$barney = 'fred zjadł ' . $meal; # Inny sposób, aby zapisać powyższy przykład
```

Jak widać w ostatnim wierszu, możliwe jest otrzymanie identycznych rezultatów bez używania podwójnych cudzysłowów. Niemniej jednak łańcuch w podwójnych cudzysłowach jest często wygodniejszym sposobem zapisu.

Jeżeli zmiennej skalarnej nie została nigdy nadana żadna wartość[20], zamiast niej zostanie użyty pusty łańcuch:

```
$barney = "fred zjadł $meal";     # Zmienna $barney ma teraz wartość "fred zjadł "
```

Jeżeli w łańcuchu występuje pojedyncza samotna zmienna, wtedy nie należy przejmować się interpolacją:

```
print "$fred"   # Znaki cudzysłowu nie są potrzebne
print $fred;    # Lepszy styl zapisu
```

[18] Nie ma to nic wspólnego z interpolacją matematyczną lub statystyczną.

[19] Oraz kilku innych rodzajów zmiennych, o których powiemy później.

[20] Jest to wtedy specjalna niezdefiniowana wartość `undef`, która zostanie opisana w dalszej części rozdziału. Jeżeli wyświetlanie komunikatów z ostrzeżeniami zostanie włączone, wtedy w przypadku próby interpolacji wartości niezdefiniowanej Perl wyświetli ostrzeżenie.

Nie ma nic złego w umieszczaniu pojedynczej zmiennej w cudzysłowach, lecz wtedy programista narazi się na śmiech innych programistów za swoimi plecami[21]. *Interpolacja zmiennych* jest również znana pod nazwą *interpolacji w podwójnych cudzysłowach*, ponieważ występuje w sytuacji, gdy użyte zostaną znaki podwójnych, a nie pojedynczych cudzysłowów. Interpolacja ta występuje również w przypadku innych łańcuchów w Perlu, ale wspomnimy o nich później.

Aby w łańcuchu w podwójnych cudzysłowach umieścić zwykły znak dolara, należy poprzedzić go lewym ukośnikiem, który spowoduje wyłączenie specjalnego znaczenia znaku dolara:

```
$fred = 'witaj';
print "Nazwa zmiennej to \$fred.\n";       # Wyświetla znak dolara
print 'Nazwa zmiennej to $fred'   . "\n"; # podobnie jak i ten przykład
```

Jako nazwę zmiennej po znaku dolara Perl będzie traktować najdłuższą możliwą nazwę, mającą sens w tej części łańcucha. Może to stanowić problem, jeżeli programista chce natychmiast po zastępowanej wartości umieścić inny tekst zaczynający się od litery, cyfry lub znaku podkreślenia[22]. W chwili gdy Perl będzie szukać nazwy zmiennej, potraktuje te znaki jako dodatkowe znaki z *nazwy* zmiennej, co nie jest zgodne z oczekiwaniami programisty. Aby zapobiec takiej sytuacji, Perl udostępnia symbol ograniczający nazwę zmiennej, podobny do rozwiązania znanego z programu powłoki systemu. W tym celu należy umieścić nazwę zmiennej w parze nawiasów klamrowych. Innym sposobem jest zakończenie bieżącej części łańcucha i rozpoczęcie innej, a następnie połączenie obu za pomocą operatora łączenia:

```
$meal = "stek";
$n=3;
print "Fred zjadł  $n $meali.\n";
                                  # Nie zostanie wyświetlony napis steki, lecz Perl będzie szukać zmiennej $meali
print "Fred zjadł  $n ${meal}i.\n";          # Teraz użyje zmiennej $meal
print "Fred zjadł  $n $meal" . "i.\n";       # Inny sposób
print 'Fred zjadł ' . $n . ' ' . $meal ."i.\n"; # Szczególnie trudny sposób
```

Priorytet oraz łączenie operatorów

Priorytet operatorów określa, które operacje w złożonej ich grupie zostaną wykonane jako pierwsze, na przykład czy w wyrażeniu `2+3*4` należy najpierw wykonać dodawanie, czy mnożenie. W przypadku wykonania w pierwszej kolejności mnożenia (jak uczono na zajęciach z matematyki), otrzymamy wynik `2+12` lub `14`. Na szczęście Perl używa właśnie tej wspólnej definicji matematycznej, wykonując najpierw mnożenie. Z tego powodu można powiedzieć, że operacja mnożenia ma wyższy priorytet od operacji dodawania.

Domyślny priorytet operatorów może zostać zmieniony przy użyciu nawiasów. Wszystkie operacje w nawiasach są obliczane całkowicie w pierwszej kolejności przed zastosowaniem operatora umieszczonego na zewnątrz nawiasów (tak jak uczono tego na zajęciach z matematyki). Jeżeli więc istnieje konieczność wykonania dodawania przed operacją mnożenia, można zapisać `(2+3)*4`, co w przypadku wykonania tej operacji spowoduje uzyskanie wartości

[21] No cóż, taki zapis może służyć do interpretacji wartości w charakterze łańcucha, a nie liczby. W rzadkich przypadkach takie cudzysłowy mogą być konieczne, lecz zwykle ich używanie to niepotrzebna strata czasu.

[22] Istnieją również inne znaki, które mogą sprawiać problem. Jeżeli konieczne jest umieszczenie po nazwie zmiennej skalarnej znaku lewego nawiasu kwadratowego lub klamrowego, należy poprzedzić je oba znakiem lewego ukośnika. Można to zrobić również, jeżeli po nazwie zmiennej występuje apostrof lub para dwukropków. Innym sposobem jest użycie metody wykorzystującej nawiasy klamrowe, opisanej w tym rozdziale.

20. Jeżeli jednak Czytelnik chciałby jawnie wskazać, że operacja mnożenia wykonywana jest przed operacją dodawania, może dodać ozdobny, lecz zbędny zestaw nawiasów, jak na przykład `2+(3*4)`.

Chociaż priorytet operatorów w przypadku dodawania oraz mnożenia jest prosty, to łatwo można napotkać trudności, zestawiając operator łączenia łańcuchów z operatorem potęgowania. Właściwym sposobem rozwiązania tego problemu jest odwołanie się w tym momencie do oficjalnego i jednoznacznego spisu kolejności operatorów przedstawionego w tabeli 2.2[23]. Niektóre z operatorów umieszczonych w tej tabeli mogły nie zostać jeszcze opisane, a nawet mogą wcale nie pojawić się w tej książce, jednak nie powinno to zniechęcać użytkownika do zapoznania się z informacjami o nich w dokumentacji *perlop*.

Tabela 2.2. Strona łączenia oraz priorytet operatorów (od najwyższego do najniższego)

Łączenie	Operatory
lewostronne	Nawiasy oraz argumenty operatorów listowych
lewostronne	`->`
	`++ --` (autoinkrementacja oraz autodekrementacja)
prawostronne	`**`
prawostronne	`\ ! ~ + –` (operatory jednoargumentowe)
lewostronne	`=~ !~`
lewostronne	`* / % x`
lewostronne	`+ - .` (operatory dwuargumentowe — binarne)
lewostronne	`<< >>`
	Nazwane operatory jednoargumentowe (`-X` test_pliku, `rand`)
	`< <= > >= lt le gt ge` (operatory „nierówności”)
	`== != <=> eq ne cmp` (operatory „równości”)
lewostronne	`&`
lewostronne	`| ^`
lewostronne	`&&`
lewostronne	`||`
	`.. ...`
prawostronne	`? :` (operator trójargumentowy)
prawostronne	`= += -= .=` (oraz podobne operatory przypisania)
lewostronne	`, =>` (operatory listowe prawostronne)
prawostronne	`not`
lewostronne	`and`
lewostronne	`or xor`

W przedstawionej tabeli każdy operator ma wyższy priorytet od wszystkich operatorów wymienionych po nim oraz niższy priorytet od wszystkich operatorów wymienionych przed nim. Operatory z tym samym priorytetem rozwiązywane są zgodnie z zasadami łączenia.

[23] Programiści używający języka C mają powody do zadowolenia. Priorytet oraz łączenie operatorów dostępnych zarówno w Perlu, jak i w C jest identyczny w obu tych językach.

Podobnie do priorytetu, również łączenie operatorów określa kolejność wykonywania operacji w sytuacji, gdy dwa operatory o tym samym priorytecie rywalizują o trzy argumenty:

```
4 ** 3 ** 2 # 4 ** (3 ** 2) lub 4 ** 9 (łączenie prawostronne)
72 / 12 / # (72 / 12) / 3 lub 6/3 lub 2 (łączenie lewostronne)
36 / 6 * 3 # (36/6)*3 lub 18
```

W pierwszym przykładzie operator ** łączy się prawostronnie, dlatego też nawiasy są wymuszane niejawnie po prawej jego stronie. Dla porównania operatory * oraz / mają łączenie lewostronne, dlatego też zestaw niejawnych nawiasów pojawia się po ich lewej stronie.

Czy w takim razie należy nauczyć się tej tabeli na pamięć? Nie! Nikt tego nie robi. Gdy nie pamięta się kolejności operacji lub gdy nie ma czasu na sprawdzenie jej w tabeli, należy dla bezpieczeństwa użyć nawiasów. Ważniejsze jest jednak, że jeżeli nawet programista nie jest w stanie zapamiętać priorytetu operatorów i poradzić sobie bez użycia nawiasów, podobny problem napotka z pewością również osoba, która będzie zajmować się utrzymaniem kodu. Dlatego też należy być dla niej życzliwym, gdyż w pewnej chwili można znaleźć się na jej miejscu.

Operatory porównania

Perl dysponuje logicznymi operatorami porównania, służącymi do porównywania liczb. Przypominają one algebrę: `< <= == >= > !=`. Każdy z tych operatorów zwraca wartość `true` (prawda) lub `false` (fałsz). Więcej o tych operatorach dowiemy się w kolejnym podrozdziale. Niektóre z nich mogą różnić się od operatorów używanych w innych językach programowania. Na przykład operator == jest operatorem równości. Natomiast pojedynczy znak = używany jest do operacji przypisania. Operator `!=` używany jest do testowania nierówności, ponieważ operator <> w Perlu używany jest do innych celów. Również w celu zapisania relacji „większe bądź równe" używany jest operator >=, a nie =>, ponieważ ten ostatni ma w Perlu inne znaczenie. W rzeczywistości prawie każda sekwencja znaków przestankowych jest w Perlu stosowana w jakimś celu. Dlatego też można utworzyć program, wybierając losowo znaki na klawiaturze i przetestować, co on wykonuje.

Do porównywania łańcuchów Perl dysponuje równoważnym zestawem operatorów, wyglądających podobnie do krótkich zabawnych słówek: `lt le eq ge gt ne`. Operatory te porównują dwa łańcuchy znak po znaku, sprawdzając ich identyczność lub też pierwszeństwo któregoś z nich zgodnie ze standardowymi regułami sortowania łańcuchów. W standardzie ASCII wielkie litery występują przed literami małymi, należy więc zwrócić na to uwagę.

Operatory porównania (zarówno dla liczb, jak też dla łańcuchów) przedstawione są w tabeli 2.3.

Tabela 2.3. Operatory porównania liczb oraz łańcuchów

Porównanie	Liczby	Łańcuchy
równe	==	eq
różne	!=	ne
mniejsze niż	<	lt
większe niż	>	gt
mniejsze niż lub równe	<=	le
większe niż lub równe	>=	ge

Poniżej przedstawionych jest kilka przykładów wyrażeń korzystających z tych operatorów porównania:

```
35 != 30 + 5              # Fałsz
35 == 35.0                # Prawda
'35' eq '35.0'            # Fałsz (porównywane jako łańcuchy)
'fred' lt 'adam'          # Fałsz
'fred' lt 'abrakadabra'   # Fałsz
'fred' eq 'fred'          # Prawda
'fred' eq 'Fred'          # Fałsz
'    '   gt ' '           # Prawda
```

Struktura kontrolna if

W chwili gdy dysponujemy możliwością porównania dwóch wartości, programista będzie prawdopodobnie chciał, aby program podejmował decyzje w zależności od rezultatu tego porównania. Podobnie jak wszystkie zbliżone języki programowania, Perl dysponuje strukturą kontrolną `if`:

```
if ($name gt 'fred') {
     print "Wyraz '$name' jest większy od 'fred' w przyjętej kolejności sortowania.\n";
}
```

W przypadku konieczności wykonania działania alternatywnego umożliwia je słowo kluczowe `else`:

```
if ($name gt 'fred') {
     print "Wyraz '$name' jest większy od 'fred' w przyjętej kolejności sortowania.\n";
} else {
     print "Wyraz '$name' nie jest większy od 'fred'.\n";
     print "W rzeczywistości może to być ten sam łańcuch. \n";
}
```

W konstrukcji warunkowej konieczne jest ograniczenie bloków kodu przy użyciu nawiasów klamrowych (w przeciwieństwie do języka C, abstrahując od tego, czy Czytelnik zna ten język, czy nie). Dobrym pomysłem jest dodawanie do zawartości bloku wcięcia w sposób przedstawiony w przykładzie. Dzięki temu łatwiej jest zorientować się w jego układzie. Jeżeli Czytelnik używa edytora tekstowego przeznaczonego dla programistów, program ten wykona większość pracy związanej z formatowaniem za użytkownika.

Wartości logiczne

W charakterze wyrażenia warunkowego struktury kontrolnej `if` możliwe jest wykorzystanie dowolnej wartości skalarnej. Jest to bardzo praktyczne w sytuacji, gdy programista pragnie zachować wartość logiczną prawda lub fałsz w zmiennej, jak chociażby w następującym przykładzie:

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

W jakiś sposób jednak Perl decyduje, czy podana wartość reprezentuje prawdę, czy fałsz. Perl nie dysponuje oddzielnym typem danych boolean, który posiadają innej języki programowania. W zamian używa kilku prostych reguł[24]:

[24] Nie są to wprawdzie reguły używane przez samego Perla, lecz reguły, których można użyć, aby osiągnąć ten sam wynik.

- jeżeli wartość jest liczbą, wtedy `0` oznacza fałsz, a wszystkie inne liczby oznaczają prawdę,
- jeżeli wartość jest łańcuchem, wtedy pusty łańcuch (`''`) oznacza fałsz, zaś każdy inny prawdę,
- jeżeli wartość jest innym rodzajem wartości skalarnej niż liczba lub łańcuch, wtedy należy przekonwertować ją do liczby lub łańcucha i spróbować porównać ponownie[25].

W przedstawionych regułach ukryty jest pewien podstęp. Ponieważ łańcuch postaci `'0'` jest tą samą wartością skalarną co liczba `0`, Perl musi traktować je identycznie. Oznacza to, że łańcuch `'0'` jest jedynym niepustym łańcuchem, który reprezentuje wartość fałsz.

Jeżeli istnieje konieczność uzyskania wartości przeciwnej do dowolnej z wartości logicznych, należy wykorzystać operator `not`, czyli znak `'!'`. Jeżeli następujące po tym znaku wyrażenie ma wartość prawdziwą, wtedy zwrócona zostanie wartość fałsz. Jeśli natomiast operator poprzedza wartość fałszywą, wtedy zostanie zwrócona wartość prawda.

```
if (! $is_bigger) {
  # Wykonaj czynność gdy wyrażenie $is_bigger nie jest prawdziwe
}
```

Pobieranie danych od użytkownika

W tym momencie Czytelnik zainteresowany jest prawdopodobnie tym, w jaki sposób w języku Perl pobrać wartość wprowadzoną z klawiatury w programie. Oto najprostszy sposób: należy skorzystać z operatora wczytywania wiersza: `<STDIN>`[26].

Za każdym razem gdy operator `<STDIN>` używany jest w miejscu, w którym oczekiwana jest wartość skalarna, Perl pobiera ze standardowego wejścia kolejny kompletny tekst (aż do pierwszego znaku nowego wiersza) i używa tego łańcucha w charakterze wartości `<STDIN>`. Standardowe wejście może oznaczać wiele różnych rzeczy. Do czasu gdy programista nie zdefiniuje go inaczej, oznacza ono klawiaturę wykorzystywaną przez użytkownika, który uruchomił program. Jeżeli żaden łańcuch nie oczekuje w buforze na pobranie przez operator `<STDIN>` (jest to typowy przypadek, o ile użytkownik nie wprowadzi wcześniej całego wiersza), program Perla zostanie zatrzymany i będzie oczekiwać na wprowadzenie dowolnych znaków, zakończonych znakiem nowego wiersza (`return`)[27].

Wartość łańcucha zwróconego przez `<STDIN>` zakończona jest zazwyczaj za pomocą znaku nowego wiersza[28]. Dlatego też w celu odczytania danych można użyć czegoś podobnego do następującego programu:

---

[25] Oznacza to, że wartość niezdefiniowana `undef` (która zostanie wkrótce omówiona) reprezentuje fałsz, zaś wszystkie odwołania (opisane w książce z alpaką na okładce) reprezentują wartość prawdziwą.

[26] Ten operator działa przy użyciu uchwytu pliku `STDIN`. Zostanie on omówiony przy okazji opisu uchwytów plików (w rozdziale 5.).

[27] Szczerze mówiąc, to system użytkownika oczekuje na dane wejściowe. Perl oczekuje natomiast na system. Chociaż szczegóły implementacji zależą od używanego systemu oraz jego konfiguracji, to ogólnie biorąc przed wciśnięciem klawisza *return* możliwe jest poprawienie wprowadzanego tekstu przy użyciu klawisza *backspace*, ponieważ to system użytkownika, a nie Perl obsługuje wprowadzane dane. Jeżeli potrzebna jest większa kontrola nad danymi wejściowymi, należy skorzystać z modułu `Term::ReadLine` dostępnego w CPAN.

[28] Wyjątkiem jest sytuacja, gdy strumień ze standardowego wejścia kończy się w połowie wiersza. Lecz nie jest on wtedy oczywiście poprawnym plikiem tekstowym.

```
$line = <STDIN>;
if ($line eq "\n") {
   print "Został wprowadzony pusty wiersz!\n";
} else {
   print "Wprowadzony wiersz: $line";
}
```

W praktyce programiście niezbyt często zależy na zatrzymaniu wewnątrz wprowadzanego wiersza znaku nowego wiersza, dlatego też należy skorzystać z operatora `chomp`.

Operator chomp

W pierwszej chwili po zapoznaniu się z operatorem `chomp` wydaje się, że jest on zbyt mocno specjalizowany. Operator ten działa na zmiennej przechowującej łańcuch. Jeżeli zakończony jest znakiem nowego wiersza, operator `chomp` może się go pozbyć. To prawie wszystko, do czego został on użyty w następującym przykładzie:

```
$text = "wiersz tekstu\n";     # lub też dane pochodzące z <STDIN>
chomp($text);                  # Pozbywa się znaku nowego wiersza
```

Operator ten wydaje się być tak przydatny, że jest umieszczany w prawie każdym nowo tworzonym programie. Jak widać na przykładzie, jego użycie jest najlepszym sposobem na usunięcie końcowego znaku nowego wiersza z łańcucha umieszczonego w zmiennej. W rzeczywistości istnieje jeszcze łatwiejszy sposób skorzystania z operatora `chomp`, wynikający z prostej zasady: wszędzie tam, gdzie w Perlu potrzebna jest zmienna, zamiast niej może zostać użyte przypisanie. Perl wykona przypisanie, a następnie użyje zmiennej w żądany sposób. Najczęstszy sposób użycia operatora `chomp` wygląda następująco:

```
chomp($text = <STDIN>);   # Pobiera tekst bez znaku nowego wiersza
$text = <STDIN>;          # Wykonuje tę samą czynność co powyżej
chomp($text);             # lecz w dwóch krokach
```

Na pierwszy rzut oka połączenie operatora `chomp` z operatorem przypisania może nie wydawać się najprostszym sposobem. Szczególnie jeżeli wygląda to zbyt zawile. Jeśli programista rozumie go jako dwie operacje, czyli pobranie wiersza oraz użycie na nim operatora `chomp`, wtedy bardziej naturalne jest użycie składni korzystającej z dwóch instrukcji. Jeżeli natomiast myśli on o tym jako o jednej operacji, czyli wczytaniu tekstu bez znaku nowego wiersza, wtedy bardziej naturalne wydaje się użycie jednej instrukcji. Ponieważ większość innych programistów Perla będzie używać krótszego zapisu, można się do niego przyzwyczaić już w tym momencie.

Operator `chomp` jest funkcją. Jako funkcja musi zwracać wartość, która w jego przypadku oznacza liczbę usuniętych znaków. Ta liczba jest rzadko przydatna:

```
$food = <STDIN>;
$betty = chomp $food;   # Zwraca wartość 1, ale przecież jest to oczywiste!
```

Jak widać, można używać operatora `chomp` zarówno bez, jak też z nawiasami. To kolejna ogólna zasada w Perlu. Oprócz sytuacji, w których usunięcie nawiasów zmienia znaczenie wyrażenia, są one zawsze opcjonalne.

Jeżeli wiersz zakończony jest za pomocą dwóch lub więcej znaków nowego wiersza[29], operator `chomp` usuwa tylko jeden. Jeżeli łańcuch nie zawiera znaków nowego wiersza, wtedy operator `chomp` nie robi nic i zwraca wartość zero.

Struktura kontrolna while

Podobnie do większości algorytmicznych języków programowania, Perl dysponuje kilkoma strukturami pętli[30]. Pętla `while` powtarza blok kodu tak długo, jak spełniony jest warunek sterujący (ma wartość `true`):

```
$count = 0;
while ($count < 10) {
$count +=2;
print "licznik ma teraz wartość $count\n";   # Wyświetla wartości 2 4 6 8 10
}
```

Podobnie jak zwykle w Perlu, warunek logiczny działa w tym przypadku identycznie co warunek w instrukcji `if`. Podobnie również jak w tamtej instrukcji, również tu wymagane są nawiasy klamrowe. Warunek logiczny sprawdzany jest przed pierwszym wykonaniem pętli, dlatego też gdy od początku nie jest spełniony, pętla może zostać całkowicie pominięta.

Wartość undef

Co się stanie, gdy zmienna skalarna zostanie użyta przed nadaniem jej wartości? Nie zdarzy się nic poważnego, a z pewnością nie przydarzy się błąd krytyczny. Zanim do zmiennej po raz pierwszy zostanie przypisana wartość, ma ona specjalną wartość niezdefiniowaną `undef`, która w Perlu oznacza: „nie znajduje się tu nic ciekawego, przejdź dalej”. Jeżeli programista użyje tej wartości w kontekście wartości liczbowej, zostanie ona potraktowana jako wartość `0`. Jeśli natomiast zostanie ona użyta w kontekście łańcucha, zostanie potraktowana jako łańcuch pusty. Niemniej jednak wartość `undef` nie jest ani liczbą, ani łańcuchem. Jest ona całkowicie różnym rodzajem wartości skalarnej. Ponieważ wartość `undef` w przypadku jej użycia w kontekście liczbowym jest traktowana jako zero, łatwo jest utworzyć licznik startujący od wartości `0`.

```
# Dodaj kilka liczb nieparzystych
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n +=2; # Przejdź do kolejnej liczby nieparzystej
}
print "Suma wynosi: $sum.\n";
```

[29] Tego rodzaju sytuacja nie może zdarzyć się podczas wczytywania wiersza, lecz może wystąpić, gdy programista ustawił separator danych wejściowych (`$/`) na znak inny niż znak nowego wiersza, użył funkcji `read` lub może sam złączył kilka łańcuchów.

[30] Każdy programista prędzej czy później utworzy przez pomyłkę nieskończoną pętlę. Jeżeli program nie chce się zakończyć, można zmusić go do tego w ten sam sposób, co dowolny inny program w systemie. Bardzo często program zatrzymuje użycie kombinacji *Ctrl+C*. Aby upewnić się o tym, należy sprawdzić dokumentację systemu.

Program ten działa poprawnie, gdy zmienna $sum miała przed rozpoczęciem wykonywania pętli wartość undef. Podczas pierwszego wykonania pętli zmienna $n ma wartość jeden, dlatego też w pierwszym wierszu do zmiennej $sum dodawane jest 1. Przypomina to dodawanie wartości 1 do zmiennej, która już zawiera zero, ponieważ wartość undef została użyta w kontekście liczbowym. W tym momencie suma wynosi 1. Dalej pętla działa już w sposób tradycyjny, ponieważ zmienna została zainicjalizowana.

Podobnie można utworzyć zmienną przechowującą łańcuch bez jego wcześniejszej inicjalizacji:

```
$string .= "więcej tekstu\n";
```

Gdy zmienna $string ma wartość niezdefiniowaną undef, będzie zachowywać się, jakby zawierała pusty łańcuch, co spowoduje przypisanie do niej tekstu "więcej tekstu\n". Jeżeli jednak zmienna zawiera łańcuch, wtedy nowy tekst jest dołączany.

Programiści Perla często używają nowej zmiennej właśnie w ten sposób, pozwalając, aby zachowywała się w razie takiej potrzeby jak zero lub pusty łańcuch.

Wiele operatorów zwraca wartość undef,gdy argumenty wykraczają poza dozwolony zakres lub nie mają sensu. Jeżeli programista nie wykona nic specjalnego, wtedy otrzyma wartość zero lub pusty łańcuch bez żadnych dodatkowych konsekwencji. W praktyce rzadko jest to większy problem. W rzeczywistości większość programistów bazuje na takim zachowaniu języka. Należy jednak nadmienić, że w przypadku włączenia wyświetlania komunikatów ostrzeżeń Perl będzie zazwyczaj ostrzegał przed niezwykłym użyciem wartości niezdefiniowanej, ponieważ może to wskazywać na wystąpienie błędu. Na przykład kopiowanie wartości undef z jednej zmiennej do innej nie jest problemem, lecz próba wyświetlenia jej za pomocą instrukcji print spowodowałaby wyświetlenie komunikatu o błędzie.

Funkcja defined

Jednym z operatorów, który może zwrócić wartość undef jest operator <STDIN>. Zazwyczaj zwraca on wiersz tekstu. Jeżeli jednak brak jest kolejnych danych wejściowych, na przykład napotkany zostanie znak końca pliku, w celu zasygnalizowania tej sytuacji operator ten zwróci wartość undef[31]. Aby odróżnić sytuację, gdy wartość wynosi undef, a nie jest pustym łańcuchem, należy użyć funkcji defined, zwracającej wartość fałsz dla wartości niezdefiniowanej (undef) oraz prawda dla jakiejkolwiek innej wartości:

```
$madonna = <STDIN>;
if (defined($madonna)) {
   print "Wprowadzono $madonna";
} else {
    print "Nie wprowadzono danych!\n";
}
```

Gdy programista chce utworzyć własne wartości niezdefiniowane, może wykorzystać operator undef:

```
$madonna = undef;   # Zupełnie jakby to była wartość niezdefiniowana
```

[31] W normalnej sytuacji jeżeli dane wejściowe wprowadzane są z klawiatury, nie występuje znak „końca pliku”, jednak dane wejściowe mogą zostać przekierowane z pliku. Użytkownik może też nacisnąć przycisk, który system rozpozna jako znak końca pliku.

Ćwiczenia

Odpowiedzi do ćwiczeń umieszczone są w dodatku A:

1. [5] Napisz program obliczający obwód okręgu o promieniu 12,5. Wzór na obwód wynosi 2π razy średnica (około 2 razy 3,141592654). Uzyskany wynik powinien wynosić około 78,5.
2. [4] Zmodyfikuj program z poprzedniego ćwiczenia w ten sposób, aby pobierał wartość średnicy od użytkownika programu. Gdy użytkownik jako wartość średnicy wprowadzi 12,5, powinien otrzymać tę samą wartość obwodu co w poprzednim ćwiczeniu.
3. [4] Zmodyfikuj program z poprzedniego ćwiczenia, tak aby w przypadku, gdy użytkownik wprowadzi liczbę mniejszą od zera, obliczona wartość obwodu wynosiła zero, a nie była liczbą ujemną.
4. [8] Napisz program pobierający od użytkownika dwie liczby (wprowadzone w dwóch oddzielnych wierszach) i wyświetlający ich iloczyn.
5. [8] Napisz program, pobierający od użytkownika łańcuch oraz liczbę (wprowadzone w dwóch oddzielnych wierszach) i wyświetlający ten łańcuch w tylu oddzielnych wierszach, na ile wskazywała podana liczba. (Wskazówka: skorzystaj z operatora `"x"`). Jeżeli użytkownik podał łańcuch „fred” oraz liczbę „3”, w rezultacie powinien otrzymać trzy wiersze, każdy zawierający wyraz „fred”. Jeśli użytkownik wprowadzi łańcuch „fred” oraz liczbę „299792”, wyświetlonych wierszy może być znacznie więcej.