helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

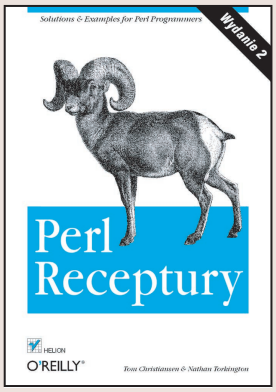
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Perl. Receptury. Wydanie II



Autorzy: Tom Christiansen, Nathan Torkington
Tłumaczenie: Mateusz Michalski (wstęp, rozdz. 1 – 10),
Rafał Szpoton (rozdz. 11 – 16, 22),
Sławomir Dzieniszewski (rozdz. 17 – 21)
ISBN: 83-7361-328-5
Tytuł oryginału: Perl Cookbook, 2nd Edition
Format: B5, stron: 1056

Książka „Perl. Receptury. Wydanie II" to wyczerpujący zbiór problemów, ich rozwiązań oraz praktycznych przykładów przydatnych dla wszystkich programujących w Perlu. Pierwsze wydanie książki cieszyło się ogromną popularnością, uznane zostało nie tylko za jedną z najlepszych książek o Perlu, lecz również za najlepszą książkę dotyczącą programowania w dowolnym języku. Ukazanie się pierwszego wydania tej książki to zarazem data powstania nowego rodzaju podręcznika programowania, nie jest to tylko zbiór różnego rodzaju sztuczek i wskazówek, ale przede wszystkim książka, która przedstawia niuanse programowania za pomocą zaczerpniętych z praktyki programistycznej problemów oraz przykładów.

Drugie wydanie książki „Perl. Receptury" zostało rozszerzone w taki sposób, aby opisać nie tylko nowe cechy samego Perla, lecz również nowe technologie powstałe od czasu opublikowania pierwszego wydania. Dodane zostały dwa całkiem nowe rozdziały, a wiele innych zostało poszerzonych: pojawiło się 80 nowych receptur, zaś 100 zostało uaktualnionych.

Książka zawiera omówienie obróbki danych (łańcuchów znakowych, wartości liczbowych, dat, tablic zwykłych oraz asocjacyjnych), obsługi operacji wejścia-wyjścia, wyrażeń regularnych, modułów, odwołań, obiektów, struktur danych, sygnałów, wykorzystania baz danych, tworzenia aplikacji graficznych, stosowania komunikacji międzyprocesowej, bezpieczeństwa, programowania aplikacji internetowych, wykorzystywania CGI oraz LWP. Tę edycję uzupełniono także o:

- Opis obsługi standardu kodowania Unicode w Perlu z uwzględnieniem obsługi łańcuchów znakowych, wyrażeń regularnych oraz operacji wejścia-wyjścia.
- Specjalny rozdział poświęcony programowaniu przy użyciu mod_perla, będącego modułem Apache osadzającym język Perl w tym popularnym serwerze HTTP, co ogromnie przyspiesza wykonywanie zadań w porównaniu z tradycyjnym interfejsem CGI.
- Nowe oraz uaktualnione receptury opisujące zastosowanie modułów dołączanych do standardowej dystrybucji Perla.
- Nowy rozdział dotyczący przetwarzania dokumentów XML, światowego standardu wykorzystywanego w procesie tworzenia oraz wymiany dokumentów.

Książka „Perl. Receptury. Wydanie II" została określona mianem najbardziej przydatnej książki napisanej dotychczas o Perlu. Uczy ona programowania w najszybszy sposób: przedstawiając sposób rozwiązania danego zadania przez ekspertów, a następnie jego objaśnienie. Chodź książka ta nie jest podręcznikiem języka Perl, pokazuje ona, jak należy programować w tym języku.

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

Spis treści

Przedmowa ***17***

Wstęp ***19***

Rozdział 1. Łańcuchy ***31***

1.0. Wprowadzenie 31
1.1. Dostęp do wybranej części łańcucha 37
1.2. Ustalanie wartości domyślnych 40
1.3. Zamiana wartości bez korzystania ze zmiennych tymczasowych 44
1.4. Konwersja między znakami a liczbami 45
1.5. Stosowanie nazwanych znaków Unicode 47
1.6. Przetwarzanie łańcucha znak po znaku 48
1.7. Odwracanie kolejności słów lub znaków w łańcuchu 51
1.8. Traktowanie dołączonych znaków Unicode jako pojedynczych znaków 52
1.9. Sprowadzanie łańcuchów zawierających znaki dołączone Unicode do postaci kanonicznej 54
1.10. Traktowanie łańcuchów w Unicode jako oktetów 56
1.11. Rozwijanie i kompresowanie tabulatorów 57
1.12. Rozwijanie zmiennych we wprowadzanych łańcuchach 59
1.13. Zmiana wielkości liter 61
1.14. Formatowanie tytułów i nagłówków 63
1.15. Interpolacja funkcji i wyrażeń w łańcuchach 66
1.16. Tworzenie wcięć w dokumentach w miejscu 68
1.17. Zmiana formatu akapitów 71
1.18. Wyświetlanie znaków ucieczki 74
1.19. Usuwanie odstępów z końca łańcucha 76
1.20. Analizowanie danych oddzielonych przecinkami 78
1.21. Zmienne niemodyfikowalne 81

1.22. Dopasowywanie fonetyczne 83
1.23. Program fixstyle 85
1.24. Program psgrep 88

Rozdział 2. Liczby 93
2.0. Wprowadzenie 93
2.1. Sprawdzanie czy łańcuch jest poprawną liczbą 95
2.2. Zaokrąglanie liczb zmiennoprzecinkowych 98
2.3. Porównywanie liczb zmiennoprzecinkowych 101
2.4. Działania na ciągach liczb całkowitych 103
2.5. Obsługa liczb rzymskich 105
2.6. Generowanie liczb losowych 106
2.7. Generowanie powtarzalnych sekwencji liczb losowych 107
2.8. Generowanie liczb jeszcze bardziej losowych 109
2.9. Generowanie liczb losowych z nierównomiernym rozkładem prawdopodobieństwa 110
2.10. Działania trygonometryczne wykonywane w stopniach, nie w radianach 112
2.11. Obliczanie bardziej zaawansowanych funkcji trygonometrycznych 113
2.12. Obliczanie logarytmów 114
2.13. Mnożenie macierzy 115
2.14. Używanie liczb zespolonych 117
2.15. Konwersja liczb binarnych, ósemkowych oraz szesnastkowych 118
2.16. Umieszczanie kropek w liczbach 120
2.17. Poprawne pisanie liczby mnogiej 121
2.18. Program: obliczanie czynników pierwszych 123

Rozdział 3. Data i czas 125
3.0. Wprowadzenie 125
3.1. Uzyskanie bieżącej daty 128
3.2. Konwertowanie DMRGMS na liczbę sekund od początku Epoki 129
3.3. Konwertowanie liczby sekund od początku Epoki na DMRGMS 131
3.4. Dodawanie lub odejmowanie wartości od dat 132
3.5. Obliczanie różnicy między dwiema datami 133
3.6. Obliczanie dnia w tygodniu (miesiącu, roku) lub numeru tygodnia w roku 135
3.7. Uzyskiwanie daty i czasu z łańcuchów 136
3.8. Wyświetlanie daty 137
3.9. Zegary wysokiej rozdzielczości 139
3.10. Krótkie usypianie systemu 142
3.11. Program: hopdelta 143

Rozdział 4. Tablice 147

4.0. Wprowadzenie 147
4.1. Określanie listy w programie 149
4.2. Wyświetlanie listy z przecinkami 150
4.3. Zmiana rozmiaru tablicy 152
4.4. Tworzenie rzadkich tablic 154
4.5. Wykonywanie operacji na wszystkich elementach tablicy 157
4.6. Wykonywanie operacji na wszystkich elementach tablicy poprzez odwołanie 161
4.7. Usuwanie powtarzających się elementów z listy 162
4.8. Wyszukiwanie elementów występujących tylko w jednej tablicy 164
4.9. Obliczanie sumy, części wspólnej i różnicy list zawierających niepowtarzające się elementy 167
4.10. Dołączanie jednej tablicy do drugiej 170
4.11. Odwracanie tablicy 171
4.12. Przetwarzanie wielu elementów tablicy jednocześnie 172
4.13. Wyszukanie pierwszego elementu z listy, który spełnia określone kryteria 173
4.14. Wyszukanie w tablicy wszystkich elementów spełniających określone kryterium 176
4.15. Sortowanie numeryczne tablicy 178
4.16. Sortowanie listy według obliczanego pola 179
4.17. Implementacja list cyklicznych 183
4.18. Ustawianie elementów tablicy w losowej kolejności 184
4.19. Program: words 185
4.20. Program: permute 187

Rozdział 5. Tablice asocjacyjne 191

5.0. Wprowadzenie 191
5.1. Dodawanie elementów do tablicy asocjacyjnej 193
5.2. Sprawdzanie obecności klucza w tablicy asocjacyjnej 194
5.3. Tworzenie tablic asocjacyjnych z niezmiennymi kluczami lub wartościami 196
5.4. Usuwanie elementów z tablicy asocjacyjnej 197
5.5. Przeglądanie tablicy asocjacyjnej 199
5.6. Wypisywanie zawartości tablicy asocjacyjnej 202
5.7. Pobieranie elementów tablicy asocjacyjnej w kolejności ich wstawiania 204
5.8. Tablice asocjacyjne z wieloma wartościami na klucz 205
5.9. Odwracanie tablicy asocjacyjnej 207
5.10. Sortowanie tablicy asocjacyjnej 209
5.11. Łączenie tablic asocjacyjnych 210
5.12. Wyszukiwanie wspólnych lub różniących się kluczy w dwóch tablicach asocjacyjnych 212

5.13. Tablice asocjacyjne z odwołaniami 213
5.14. Wstępne ustalanie rozmiaru tablicy asocjacyjnej 214
5.15. Wyszukiwanie najczęściej występujących elementów 215
5.16. Przedstawianie relacji pomiędzy danymi 216
5.17. Program dutree 218

Rozdział 6. Dopasowywanie wzorców 223
6.0. Wprowadzenie 223
6.1. Jednoczesne kopiowanie i podstawianie 229
6.2. Dopasowywanie liter 231
6.3. Dopasowywanie słów 233
6.4. Komentowanie wyrażeń regularnych 234
6.5. Wyszukiwanie n-tego wystąpienia dopasowania 237
6.6. Dopasowywanie w obrębie wielu wierszy 240
6.7. Odczytywanie rekordów z separatorem 243
6.8. Wyodrębnianie linii z określonego zakresu 245
6.9. Wykorzystanie znaków uniwersalnych powłoki jako wyrażeń regularnych 248
6.10. Przyspieszanie dopasowań interpolowanych 249
6.11. Sprawdzanie poprawności wzorca 252
6.12. Uwzględnianie ustawień regionalnych we wzorcach 254
6.13. Dopasowywanie przybliżone 255
6.14. Dopasowywanie od miejsca, do którego poprzednio pasował wzorzec 257
6.15. Zachłanne i niezachłanne dopasowania 259
6.16. Wykrywanie powtarzających się wyrazów 262
6.17. Dopasowywanie wzorców zagnieżdżonych 266
6.18. Operacje AND, OR i NOT w pojedynczym wzorcu 267
6.19. Dopasowywanie poprawnego adresu e-mail 272
6.20. Dopasowywanie skrótów 274
6.21. Program urlify 276
6.22. Program tcgrep 277
6.23. Przegląd interesujących wyrażeń regularnych 283

Rozdział 7. Dostęp do plików 287
7.0. Wprowadzenie 287
7.1. Otwieranie pliku 297
7.2. Otwieranie plików o nietypowych nazwach 300
7.3. Rozwijanie znaku tyldy w nazwach plików 302
7.4. Uwzględnianie nazw plików w komunikatach o błędach 304

7.5. Przechowywanie uchwytów plików w zmiennych 305
7.6. Tworzenie procedury przyjmującej uchwyty tak jak funkcje wbudowane 308
7.7. Buforowanie otwartych wyjściowych uchwytów plików 309
7.8. Jednoczesny zapis do wielu uchwytów plików 311
7.9. Otwieranie i zamykanie deskryptorów plików przez ich numery 312
7.10. Kopiowanie uchwytów plików 314
7.11. Tworzenie plików tymczasowych 315
7.12. Przechowywanie pliku w tekście programu 317
7.13. Przechowywanie wielu plików w polu DATA 319
7.14. Program filtra w stylu uniksowym 321
7.15. Modyfikowanie pliku w miejscu z wykorzystaniem pliku tymczasowego 325
7.16. Modyfikowanie pliku w miejscu za pomocą opcji –i 327
7.17. Modyfikowanie pliku w miejscu bez pliku tymczasowego 329
7.18. Blokowanie pliku 330
7.19. Opróżnianie wyjścia 332
7.20. Przeprowadzanie nie blokujących operacji wejścia-wyjścia 336
7.21. Ustalanie liczby nie odczytanych bajtów 337
7.22. Odczytywanie z wielu uchwytów plików bez blokowania 339
7.23. Odczytywanie całego wiersza bez blokowania 341
7.24. Program netlock 343
7.25. Program lockarea 346

Rozdział 8. Zawartość plików 351

8.0. Wprowadzenie 351
8.1. Odczytywanie linii ze znakami kontynuacji 358
8.2. Zliczanie linii (paragrafów, rekordów) w pliku 360
8.3. Przetwarzanie każdego słowa w pliku 361
8.4. Odczytywanie linii lub paragrafów od końca pliku 363
8.5. Odczytywanie pliku zwiększającego rozmiar 365
8.6. Pobieranie losowej linii z pliku 367
8.7. Losowa zmiana kolejności linii 368
8.8. Odczytywanie wybranej linii z pliku 369
8.9. Obsługa pól tekstowych o zmiennej długości 372
8.10. Usuwanie ostatniej linii pliku 373
8.11. Operacje na plikach binarnych 374
8.12. Dostęp do dowolnego miejsca w pliku 375
8.13. Uaktualnianie rekordu wewnątrz pliku 376
8.14. Odczyt łańcucha z pliku binarnego 378

8.15. Odczytywanie rekordów o stałej długości 379
8.16. Odczytywanie plików konfiguracyjnych 381
8.17. Sprawdzanie zabezpieczeń pliku 384
8.18. Traktowanie pliku jak tablicy 386
8.19. Określanie domyślnych warstw wejścia-wyjścia 387
8.20. Czytanie i zapisywanie w formacie Unicode 388
8.21. Konwersja plików tekstowych Microsoft do formatu Unicode 391
8.22. Porównywanie zawartości dwóch plików 393
8.23. Traktowanie łańcucha znakowego jak pliku 395
8.24. Program tailwtmp 396
8.25. Program tctee 396
8.26. Program laston 398
8.27. Program: proste indeksy plików 399

Rozdział 9. Katalogi 401

9.0. Wprowadzenie 401
9.1. Odczytywanie i ustawianie znaczników czasowych 407
9.2. Usuwanie pliku 408
9.3. Kopiowanie lub przenoszenie pliku 409
9.4. Wykrywanie dwóch nazw tego samego pliku 411
9.5. Przetwarzanie wszystkich plików z katalogu 412
9.6. Globbing, czyli pobieranie listy nazw plików zgodnych z wzorcem 414
9.7. Rekursywne przetwarzanie wszystkich plików z katalogu 416
9.8. Usuwanie katalogu wraz z zawartością 418
9.9. Zmiana nazw plików 420
9.10. Podział nazwy pliku na składowe 422
9.11. Symboliczna reprezentacja praw dostępu do pliku 423
9.12. Program symirror 426
9.13. Program lst 426

Rozdział 10. Procedury 431

10.0. Wprowadzenie 431
10.1. Dostęp do argumentów procedury 432
10.2. Przekształcanie zmiennych w prywatne dla funkcji 434
10.3. Tworzenie trwałych zmiennych prywatnych 436
10.4. Określanie nazwy bieżącej funkcji 438
10.5. Przekazywanie tablic i tablic asocjacyjnych przez odwołanie 440
10.6. Wykrywanie kontekstu powrotnego 441

10.7. Przekazywanie nazwanego parametru 442
10.8. Pomijanie wybranych wartości zwracanych 444
10.9. Zwracanie więcej niż jednej tablicy 445
10.10. Zwracanie informacji o wystąpieniu błędu 446
10.11. Prototypowanie funkcji 447
10.12. Obsługa wyjątków 451
10.13. Zapisywanie wartości globalnych 453
10.14. Redefinicja funkcji 456
10.15. Przechwytywanie wywołań niezdefiniowanych funkcji za pomocą AUTOLOAD 459
10.16. Zagnieżdżanie procedur 460
10.17. Tworzenie konstrukcji switch 461
10.18. Program do sortowania poczty 464

Rozdział 11. Odwołania oraz rekordy 469

11.0. Wprowadzenie 469
11.1. Tworzenie odwołań do tablic zwykłych 476
11.2. Tworzenie tablic asocjacyjnych zawierających tablice zwykłe 479
11.3. Tworzenie odwołań do tablic asocjacyjnych 480
11.4. Tworzenie odwołań do funkcji 481
11.5. Tworzenie odwołań do skalarów 484
11.6. Tworzenie tablic zawierających odwołania do wartości skalarnych 485
11.7. Używanie domknięć zamiast obiektów 487
11.8. Tworzenie odwołań do metod 489
11.9. Tworzenie rekordów 490
11.10. Odczytywanie oraz zapisywanie rekordów z tablicy asocjacyjnej do plików tekstowych 492
11.11. Wyświetlanie struktur danych 494
11.12. Kopiowanie struktur danych 496
11.13. Zapisywanie struktur danych na dysku 498
11.14. Trwałe struktury danych 499
11.15. Kopiowanie cyklicznych struktur danych przy użyciu słabych odwołań 501
11.16. Program. Szkice 504
11.17. Program. Drzewa binarne 507

Rozdział 12. Pakiety, biblioteki oraz moduły 511

12.0. Wprowadzenie 511
12.1. Definiowanie interfejsu modułu 517
12.2. Przechwytywanie błędów podczas stosowania require oraz use 520
12.3. Opóźnianie wykonania instrukcji use do czasu uruchomienia programu 522

12.4. Tworzenie zmiennych prywatnych w module 525
12.5. Tworzenie funkcji prywatnych w module 527
12.6. Określanie pakietu, z którego nastąpiło wywołanie 529
12.7. Automatyzacja czyszczenia modułu 531
12.8. Wykorzystywanie własnego katalogu z modułami 533
12.9. Przygotowywanie modułu do rozpowszechniania 536
12.10. Przyspieszanie wczytywania modułu przy użyciu SelfLoadera 539
12.11. Przyspieszanie wczytywania modułu przy użyciu Autoloadera 540
12.12. Ponowne definiowanie funkcji wbudowanych 541
12.13. Ponowne definiowanie funkcji wbudowanych we wszystkich pakietach 544
12.14. Informowanie o błędach oraz ostrzeżeniach w sposób podobny do funkcji wbudowanych 546
12.15. Personalizacja komunikatów z ostrzeżeniami 548
12.16. Niejawne odwołania do pakietów 552
12.17. Stosowanie programu h2ph w celu przetłumaczenia plików nagłówkowych w języku C 554
12.18. Używanie programu h2xs w celu tworzenia modułu korzystającego z kodu w języku C 557
12.19. Tworzenie rozszerzeń w języku C przy użyciu modułu Inline::C 560
12.20. Dokumentacja modułu przy użyciu programu Pod 562
12.21. Budowanie oraz instalacja modułu CPAN 564
12.22. Przykład. Szablon modułu 567
12.23. Program. Odczytywanie wersji oraz opisów zainstalowanych modułów 568

Rozdział 13. Klasy, obiekty oraz wiązania 573

13.0. Wprowadzenie 573
13.1. Tworzenie obiektu 583
13.2. Usuwanie obiektu 585
13.3. Zarządzanie danymi egzemplarza 587
13.4. Zarządzanie danymi klasy 590
13.5. Stosowanie klas w charakterze struktur 592
13.6. Konstruktory klonujące 596
13.7. Konstruktory kopiujące 598
13.8. Pośrednie wywoływanie metod 599
13.9. Określanie przynależności podklasy 602
13.10. Tworzenie klasy używanej do dziedziczenia 604
13.11. Dostęp do metod przesłoniętych 606
13.12. Tworzenie metod atrybutów przy użyciu AUTOLOAD 608

13.13. Kopiowanie cyklicznych struktur danych przy użyciu obiektów 611
13.14. Przeciążanie operatorów 614
13.15. Tworzenie magicznych zmiennych przy użyciu dowiązań 619

Rozdział 14. Dostęp do bazy danych 627

14.0. Wprowadzenie 627
14.1. Tworzenie oraz używanie pliku DBM 630
14.2. Usuwanie zawartości pliku DBM 632
14.3. Konwersja pomiędzy plikami DBM 633
14.4. Łączenie plików DBM 635
14.5. Sortowanie dużych plików DBM 636
14.6. Umieszczanie w pliku DBM złożonych danych 638
14.7. Dane trwałe 640
14.8. Zapisywanie wyników zapytania w pliku programu Excel lub pliku CSV 642
14.9. Wykonywanie polecenia SQL przy użyciu DBI 643
14.10. Zmiana znaczenia cudzysłowów 646
14.11. Obsługa błędów bazy danych 647
14.12. Wydajne powtarzanie zapytań 649
14.13. Tworzenie zapytań w sposób programistyczny 651
14.14. Odczytywanie liczby wierszy zwróconych przez zapytanie 653
14.15. Stosowanie transakcji 654
14.16. Wyświetlanie danych strona po stronie 656
14.17. Wykonywanie zapytań do pliku CSV przy użyciu instrukcji SQL 658
14.18. Wykorzystywanie poleceń SQL bez serwera bazy danych 659
14.19. Program. ggh — program wyświetlający zawartość pliku historii programu Netscape ...661

Rozdział 15. Interakcja z użytkownikiem 665

15.0. Wprowadzenie 665
15.1. Analiza argumentów programu 667
15.2. Sprawdzanie czy program został uruchomiony interaktywnie 670
15.3. Czyszczenie zawartości ekranu 672
15.4. Określanie rozmiaru terminala lub okna 673
15.5. Zmiana koloru tekstu 674
15.6. Odczytywanie z klawiatury pojedynczych znaków 676
15.7. Użycie sygnału dźwiękowego terminala 677
15.8. Stosowanie interfejsu POSIX termios 679
15.9. Sprawdzanie oczekujących danych wejściowych 681
15.10. Odczytywanie haseł 682
15.11. Edycja danych wejściowych 683

15.12. Zarządzanie wyglądem ekranu 684
15.13. Nadzorowanie innego programu przy użyciu modułu Expect 687
15.14. Tworzenie menu przy użyciu Tk 689
15.15. Tworzenie okien dialogowych przy użyciu Tk 692
15.16. Reagowanie na zdarzenia zmiany rozmiaru okna 695
15.17. Usuwanie okna powłoki systemu DOS przy użyciu Perl/Tk dla środowiska Windows 697
15.18. Tworzenie reprezentacji graficznej danych 698
15.19. Tworzenie miniatur obrazów 699
15.20. Dodawanie tekstu do obrazu 700
15.21. Program: mały program korzystający z modułu Term::Cap 701
15.22. Program: tkshufflepod 703
15.23. Program: graphbox 705

Rozdział 16. Zarządzanie procesami i komunikacja między nimi 707

16.0. Wprowadzenie 707
16.1. Zbieranie danych wyjściowych z programu 711
16.2. Uruchamianie innego programu 713
16.3. Zastępowanie wykonywanego programu innym 716
16.4. Odczytywanie danych z innego programu oraz zapisywanie ich do innego programu 717
16.5. Filtrowanie danych wyjściowych bieżącego programu 720
16.6. Wstępne przetwarzanie danych wejściowych programu 722
16.7. Odczytywanie standardowego wyjścia diagnostycznego (STDERR) 724
16.8. Kontrolowanie wejścia oraz wyjścia danych innego programu 727
16.9. Kontrolowanie wejścia, wyjścia oraz błędów innego programu 729
16.10. Komunikacja pomiędzy powiązanymi procesami 731
16.11. Tworzenie procesu przypominającego plik przy użyciu nazwanych potoków 737
16.12. Współdzielenie zmiennych przez różne procesy 741
16.13. Wypisywanie dostępnych sygnałów 743
16.14. Wysyłanie sygnału 744
16.15. Instalowanie procedury obsługi sygnału 745
16.16. Tymczasowe przesłonięcie procedury obsługi sygnału 747
16.17. Tworzenie procedury obsługi sygnału 748
16.18. Przechwytywanie kombinacji Ctrl+C 751
16.19. Zapobieganie występowaniu procesów zombie 752
16.20. Blokowanie sygnałów 755
16.21. Obsługa przekroczenia czasu operacji 757
16.22. Przekształcanie sygnałów do postaci błędów krytycznych 759
16.23. Program sigrand 760

Rozdział 17. Gniazda *765*
17.0. Wprowadzenie 765
17.1. Pisanie klienta TCP 768
17.2. Pisanie serwera TCP 770
17.3. Komunikacja za pośrednictwem protokołu TCP 773
17.4. Tworzenie klienta UDP 777
17.5. Tworzenie serwera UDP 779
17.6. Korzystanie z gniazd w domenie uniksowej 781
17.7. Identyfikowanie rozmówcy po drugiej stronie gniazda 783
17.8. Ustalanie własnej nazwy i adresu 785
17.9. Zamykanie gniazda po rozwidleniu procesu 786
17.10. Pisanie dwukierunkowych klientów 788
17.11. Rozwidlanie serwerów 790
17.12. Rozwidlanie serwera z wyprzedzeniem 791
17.13. Nie rozwidlające się serwery 794
17.14. Serwer wielozadaniowy korzystający z wątkowości 798
17.15. Jak POE pomaga pisać serwer wielowątkowy 799
17.16. Pisanie serwera działającego pod kilkoma adresami IP 802
17.17. Przygotowywanie serwera działającego jako demon 803
17.18. Ponowne uruchamianie serwera na życzenie 806
17.19. Zarządzanie wieloma strumieniami nadchodzących danych na raz 808
17.20. Przykładowy program: backsniff 811
17.21. Przykładowy program: fwdport 812

Rozdział 18. Usługi internetowe *817*
18.0. Wprowadzenie 817
18.1. Podstawy przeglądania informacji przechowywanych na serwerach DNS 819
18.2. Klient FTP 823
18.3. Wysyłanie poczty 826
18.4. Odczytywanie i wysyłanie nowych wiadomości w sieci Usenet 830
18.5. Odczytywanie poczty za pośrednictwem serwera POP3 832
18.6. Symulacja polecenia telnet wewnątrz programu 835
18.7. Sprawdzanie działania komputera za pomocą programu ping 838
18.8. Sięganie do serwera LDAP 840
18.9. Wysyłanie poczty z załącznikami 843
18.10. Wydobywanie załączników z poczty 847
18.11. Pisanie serwera XML-RPC 849

18.12. Pisanie klienta XML-RPC 851
18.13. Pisanie serwera SOAP 853
18.14. Pisanie klienta SOAP 854
18.15. Przykładowy program: rfrm 855
18.16. Przykładowy program: expn i vrfy 857

Rozdział 19. Programowanie CGI 861
19.0. Wprowadzenie 861
19.1. Pisanie skryptu CGI 866
19.2. Przekierowywanie wiadomości o błędach 869
19.3. Naprawianie błędu 500 Server Error 871
19.4. Pisanie bezpiecznego programu CGI 875
19.5. Unikanie sekwencji sterujących powłoki podczas wykonywania poleceń 880
19.6. Formatowanie list i tabel za pomocą skrótów HTML 883
19.7. Kierowanie do innego adresu 885
19.8. Wykrywanie błędów w kodzie komunikacji HTTP 887
19.9. Zarządzanie cookies 889
19.10. Tworzenie kontrolek przechowujących wprowadzone wartości 892
19.11. Pisanie skryptu CGI obsługującego wiele stron WWW 893
19.12. Zapisywanie formularza w pliku lub potoku pocztowym 896
19.13. Przykładowy program: chemiserie 898

Rozdział 20. Sieć WWW od strony klienta 903
20.0. Wprowadzenie 903
20.1. Pobieranie zasobu o określonym adresie URL za pomocą skryptu Perla 905
20.2. Automatyzacja zatwierdzania formularzy 907
20.3. Wydobywanie adresów URL 909
20.4. Konwertowanie tekstu ASCII na HTML 912
20.5. Konwertowanie dokumentu HTML na tekst ASCII 913
20.6. Wydobywanie lub usuwanie znaczników HTML 914
20.7. Odnajdywanie w dokumencie HTML łączy, które już nie działają 917
20.8. Odnajdywanie łączy do ostatnio aktualizowanych stron 918
20.9. Generowanie kodu HTML przy użyciu szablonów 920
20.10. Tworzenie zwierciadlanych kopii stron WWW 923
20.11. Tworzenie robota 924
20.12. Rozkładanie pliku dziennika serwera WWW na rekordy 925
20.13. Analizowanie dzienników serwera WWW 927

20.14. Korzystanie z cookies 930
20.15. Pobieranie stron chronionych hasłem 931
20.16. Pobieranie stron WWW w protokole https:// 932
20.17. Wznawianie żądania GET protokołu HTTP 932
20.18. Analiza kodu HTML 934
20.19. Wydobywanie danych z tabel HTML 937
20.20. Przykładowy program htmlsub 940
20.21. Przykładowy program: hrefsub 941

Rozdział 21. mod_perl 943

21.0. Wprowadzenie 943
21.1. Uwierzytelnianie 949
21.2. Ustawianie Cookies 951
21.3. Sięganie do wartości cookie 952
21.4. Kierowanie przeglądarki pod inny adres 954
21.5. Badanie nagłówków 955
21.6. Sięganie do parametrów formularza 956
21.7. Odbieranie plików ładowanych na serwer 957
21.8. Przyśpieszanie dostępu do baz danych 959
21.9. Dostosowanie działania dzienników Apache do własnych potrzeb 960
21.10. Przezroczyste przechowywanie informacji w adresach URL 962
21.11. Komunikacja między mod_perl a PHP 964
21.12. Przerabianie kodu skryptów CGI na kod mod_perl 965
21.13. Wspólne korzystanie z informacji przez różne procedury obsługi 966
21.14. Ponowne ładowanie zmienionych modułów 968
21.15. Ocena wydajności aplikacji mod_perl 969
21.16. Korzystanie z szablonów z pomocą modułu HTML::Mason 970
21.17. Korzystanie z szablonów z pomocą zestawu Template Toolkit 976

Rozdział 22. XML 983

22.0. Wprowadzenie 983
22.1. Przekształcanie dokumentów XML do postaci struktur danych 993
22.2. Analiza składniowa dokumentów XML przy użyciu reprezentacji drzewiastej DOM 995
22.3. Analiza składniowa dokumentów XML przy użyciu zdarzeń interfejsu SAX 998
22.4. Wprowadzanie prostych zmian do elementów lub ich zawartości 1002
22.5. Sprawdzanie poprawności dokumentu XML 1004
22.6. Odszukiwanie elementów oraz ich zawartości w dokumencie XML 1008

22.7. Przetwarzanie transformacji arkuszy stylów XML 1010
22.8. Analiza dokumentów o rozmiarze przekraczającym ilość dostępnej pamięci systemowej 1013
22.9. Odczytywanie oraz zapisywanie plików RSS 1015
22.10. Tworzenie dokumentu XML 1018

Skorowidz *1021*

1

Łańcuchy

„Hiob usta otworzył niemądrze i mnoży słowa bezmyślnie".
— Księga Hioba 35,16

1.0. Wprowadzenie

Wiele języków programowania zmusza do pracy na niewygodnie niskim poziomie. Programista myśli o poszczególnych liniach w pliku, język zmusza go do zajmowania się wskaźnikami. Programista operuje łańcuchami, język wymaga obsługi bajtów. Taki język może naprawdę wyprowadzić z równowagi. Jednak nie można tracić nadziei. Perl nie jest językiem niskiego poziomu, obsługa wierszy i łańcuchów jest bezproblemowa.

Perl został *zaprojektowany* z myślą o łatwej, ale zaawansowanej obróbce tekstu. Tak naprawdę potrafi on operować tekstem na tak wiele sposobów, że nie sposób opisać ich wszystkich w jednym rozdziale. Receptury związane z przetwarzaniem tekstu znajdują się także w innych rozdziałach. W szczególności rozdziały 6. i 8. omawiają ciekawe techniki, które nie zostały opisane w tym rozdziale.

Podstawowym typem danych używanym w Perlu jest skalar, co oznacza pojedyncze wartości przechowywane w pojedynczych (skalarnych) zmiennych. Zmienne skalarne zawierają łańcuchy, wartości liczbowe oraz odwołania. Tablice zwykłe i tablice asocjacyjne zawierają odpowiednio listy i powiązania wartości skalarnych. Odwołania wykorzystywane są do sięgania się do wartości w sposób pośredni, podobnie jak wskaźniki w językach niskiego poziomu. Wartości liczbowe przechowywane są z reguły w postaci zmiennoprzecinkowej o podwójnej precyzji. W Perlu łańcuchy mogą mieć dowolną długość, a jedynym ograniczeniem jest dostępna pamięć wirtualna komputera; łańcuchy mogą przechowywać dowolne dane — nawet binarne zawierające bajty zerowe.

Łańcuch w Perlu nie jest tablicą znaków czy też tablicą bajtów. Aby zaadresować wybrany znak w łańcuchu, nie można posłużyć się indeksowaniem tablicowym; do tego służy funkcja `substr`. Łańcuchy, tak jak inne typy danych w Perlu, zwiększają się w miarę potrzeb. System odzyskiwania pamięci Perla dba o zwalnianie tych miejsc, które nie są więcej potrzebne, co z reguły ma miejsce wtedy, gdy zmienna wykracza poza zakres widoczności lub kiedy wyrażenie wykorzystujące te zmienne zostanie obliczone. Innymi słowy, użytkownik nie musi się tym zajmować, ponieważ dba o to system zarządzania pamięcią.

Wartość skalarna może być zdefiniowana lub niezdefiniowana. Wartość zdefiniowana może przechowywać łańcuchy, wartości liczbowe lub odwołania. Jedyną wartością niezdefiniowaną jest `undef`. Wszystkie inne wartości, nawet wartość 0 lub pusty łańcuch, są zdefiniowane. Zdefiniowanie nie jest jednak równoważne z prawdą logiczną (`true`); aby sprawdzić czy wartość jest zdefiniowana, należy użyć funkcji `defined`. Prawda logiczna ma znaczenie specjalne, sprawdzane za pomocą takich operatorów jak `&&` i `||` w wyrażeniach warunkowych instrukcji `if` lub `while`.

Dwa zdefiniowane łańcuchy oznaczają logiczny fałsz (`false`): łańcuch pusty (`""`) oraz jednoznakowy łańcuch zawierający cyfrę zero (`"0"`). Wszystkie inne zdefiniowane wartości (np. `"false"`, `15` i `\$x`) są logiczną prawdą. Może się wydawać dziwne, że `"0"` jest fałszem, lecz wynika to z wykonywanej domyślnie przez Perla konwersji między łańcuchem a wartością liczbową. Wszystkie wartości takie jak `0.`, `0.00` czy `0.0000000` są liczbami, a przez to jeśli nie są zapisane w cudzysłowach oznaczają logiczny fałsz, ponieważ wartość zero w każdej swojej postaci jest zawsze logicznym fałszem. Jednak te trzy wartości (`"0."`, `"0.00"` oraz `"0.0000000"`) będą oznaczały logiczną *prawdę*, kiedy w kodzie programu zostaną wstawione między znakami cudzysłowu, albo gdy zostaną odczytane z wiersza poleceń, zmiennej środowiskowej bądź z pliku wsadowego.

Takie przypadki są rzadkością, ponieważ zamiana następuje automatycznie, kiedy tylko wartość jest używana numerycznie. Jeżeli jednak nie byłaby używana w ten sposób, a dokonano by jedynie sprawdzenia wartości logicznej, to można otrzymać zaskakujące wyniki — testy logiczne nigdy nie oznaczają wykonywania żadnej konwersji. Dodanie 0 do zmiennej jawnie zmusza Perla do zamiany łańcucha na wartość liczbową:

```
print "Podaj wartość: ";
0.00000
chomp($n = <STDIN>);     # zmienna $n zawiera teraz "0.00000";

print "Wartość $n jest ", $n ? "TRUE" : "FALSE", "\n";
Wartość 0.00000 jest TRUE

$n += 0;
print "Teraz wartość $n jest ", $n ? "TRUE" : "FALSE", "\n";
Teraz wartość 0 jest FALSE
```

Wartość `undef` zachowuje się jak pusty łańcuch (`""`), kiedy zostanie wykorzystana jako łańcuch; jak `0`, jeżeli zostanie użyta jako wartość liczbowa oraz jak puste odwołanie, jeżeli będzie zastosowana w tym kontekście. We wszystkich tych przypadkach jest logicznym fałszem. Jeżeli aktywowane jest wyświetlanie ostrzeżeń, to zastosowanie niezdefiniowanej

wartości wszędzie tam, gdzie Perl oczekuje wartości zdefiniowanej, spowoduje wypisanie ostrzeżenia na `STDERR`. Jedynie pytanie czy coś jest prawdą lub fałszem nie wymaga podania konkretnej wartości, stąd operacja ta nie generuje ostrzeżeń. Niektóre działania nie wywołują ostrzeżeń, gdy użyte zostaną w odniesieniu do zmiennej zawierającej wartość niezdefiniowaną. Dotyczy to operatorów inkrementacji oraz dekrementacji, `++` oraz `--`, i operatorów przypisania z dodawaniem i z łączeniem `+=` i `.=` („plus-równa się" oraz „kropka-równa się").

W kodzie programu wartości łańcuchów możemy określać, stosując cudzysłowy, apostrofy, operatory cytowania `q//` lub `qq//` oraz dokumenty w miejscu (ang. *here documents*). Bez względu na to, jaki rodzaj notacji zostanie zastosowany, literały w łańcuchu mogą być albo interpolowane, albo nieinterpolowane. Interpolacja decyduje czy odwołania do zmiennych i specjalne sekwencje znaków umieszczone w łańcuchu będą rozwijane. Większość z nich jest domyślnie interpolowana, tak jak to ma miejsce we wzorcach (`/regex/`) lub w poleceniach wykonawczych (`$x = `cmd``).

W okolicznościach, w których rozpoznawane są znaki specjalne, poprzedzenie takiego znaku lewym ukośnikiem powoduje wyłączenie jego specjalnego znaczenia — staje się on wówczas zwykłym literałem. Często określa się to mianem „wyłączania" znaku czy dodawania znaku „ucieczki".

Zastosowanie apostrofu jest kanonicznym sposobem zapisu nieinterpolowanych literałów łańcucha. Rozpoznawane są w takim przypadku jedynie trzy sekwencje specjalne: znak `'` oznaczający zakończenie łańcucha, `\'` reprezentująca apostrof i `\\` oznaczająca lewy ukośnik w łańcuchu.

```
$string = '\n';                          # dwa znaki, pierwszy to \, drugi to n
$string = 'Jon \'Maddog\' Orwant';       # apostrofy
```

Cudzysłowy dokonują interpolacji zmiennych (ale nie wywołań funkcji — receptura 1.15 opisuje jak to zrobić) oraz rozwijania znaków ucieczki, takich jak `"\n"` (nowy wiersz), `"\033"` (oznaczający ósemkową wartość 33), `"\cJ"` (*Ctrl+J*), `"\x1B"` (oznaczający heksadecymalną wartość 0x1B) i tym podobne. Kompletna lista takich sekwencji znajduje się na stronie podręcznika *perlop*(1) oraz w podrozdziale „Metaznaki i metasymbole" rozdziału 5. książki *Perl. Programowanie*[1].

```
$string = "\n";                          # znak przejścia do nowej linii
$string = "Jon \"Maddog\" Orwant";       # cudzysłowy
```

Jeżeli w łańcuchu nie ma żadnych znaków ucieczki czy zmiennych do rozwinięcia, nie ma znaczenia czy użyjemy apostrofów, czy cudzysłowów. Niektórzy programiści, wybierając między `'takim'` a `"takim"` zapisem, wolą zastosować cudzysłowy, ponieważ powodują one, że łańcuchy są łatwiej dostrzegalne. W ten sposób można też uniknąć niewielkiego

[1] Wyd. oryg. — *Programming Perl*, Larry Wall, Tom Christiansen, John Orwant, Randal R. Schwartz, wydawnictwo O'Reilly.

ryzyka pomylenia przez osobę czytającą kodu apostrofu z odwrotnym apostrofem. Dla Perla nie ma żadnej różnicy czy użyjemy apostrofów, czy cudzysłowów, a czytelnikowi może to pomóc.

Operatory `q//` i `qq//` odpowiadają, odpowiednio, apostrofom i cudzysłowom, a umożliwiają zastosowanie dowolnych znaków ograniczających łańcuchy interpolowane i nieinterpolowane. Jeżeli niepoddawany interpolacji łańcuch zawiera apostrofy, łatwiej jest użyć operatora `q//` niż poprzedzać każdy apostrof lewym ukośnikiem:

```
$string = 'Jon \'Maddog\' Orwant';    # wstawione apostrofy
$string = q/Jon 'Maddog' Orwant'/;    # to samo, ale bardziej czytelnie
```

Ogranicznikiem początkowym i końcowym powinien być ten sam znak (na przykład / powyżej) lub jedna z czterech par odpowiadających sobie nawiasów:

```
$string = q[Jon 'Maddog' Orwant'];    # ciąg znaków w apostrofach
$string = q{Jon 'Maddog' Orwant'};    # ciąg znaków w apostrofach
$string = q(Jon 'Maddog' Orwant');    # ciąg znaków w apostrofach
$string = q<Jon 'Maddog' Orwant'>;    # ciąg znaków w apostrofach
```

Dokumenty w miejscu to notacja zapożyczona z powłoki, wykorzystywana do cytowania dużych porcji tekstu. Tekst może być interpretowany jako zawarty w apostrofach, cudzysłowach bądź nawet jako polecenia do wykonania, w zależności od tego, w jaki sposób zapisze się końcowy identyfikator. Dokumenty w miejscu, które nie są interpolowane, nie rozwijają trzech sekwencji specjalnych rozpoznawanych przez apostrofy. Poniżej dwie linie tekstu podano w postaci ujmowanego w cudzysłów dokumentu w miejscu:

```
$a = <<"EOF";
To jest wieloliniowy dokument w miejscu
Zakończony przez EOF występujący po ostatniej linii
EOF
```

Warto zauważyć, że po końcowym `EOF` nie ma średnika. Bardziej szczegółowo dokumenty w miejscu omówione są w recepturze 1.16.

Uniwersalne kodowanie znaków

W kontekście komputerów wszystkie dane są tylko szeregiem pojedynczych wartości liczbowych, reprezentowanych przez ciągi bitów. Nawet łańcuchy tekstowe są jedynie ciągami kodów numerycznych interpretowanymi jako znaki przez takie programy jak przeglądarki stron internetowych, programy obsługujące pocztę elektroniczną, programy drukujące czy edytory.

W przeszłości, kiedy rozmiary pamięci były dużo mniejsze, a same pamięci dużo droższe, programiści nie ustawali w wysiłkach, aby jak najwięcej zaoszczędzić. Powszechne były takie działania jak upychanie sześciu znaków w jedno 36-bitowe słowo czy trzech znaków w jedno słowo 16-bitowe. Nawet dzisiaj numeryczne kody znaków zazwyczaj nie są dłuższe niż 7 lub 8 bitów, tak jak w, odpowiednio, kodach ASCII oraz Latin1.

Nie ma w nich zbyt wielu bitów na reprezentację znaku — a przez to reprezentowanych jest niewiele znaków. Rozważmy przykładowo plik graficzny, z kolorem zakodowanym na 8 bitach. Paleta jest w nim ograniczona do 256 różnych kolorów. Podobnie jest w przypadku znaków przechowywanych jako pojedyncze *oktety* (jeden oktet składa się z 8 bitów) — wówczas dokument ma zazwyczaj nie więcej niż 256 różnych liter, znaków interpunkcyjnych i innych symboli.

ASCII, jako *amerykański* standard kodowania dla wymiany informacji (*American Standard Code for Information Interchange*), był w ograniczonym zakresie wykorzystywany poza granicami USA, ponieważ zawierał jedynie znaki występujące w odrobinę ograniczonym amerykańskim dialekcie języka angielskiego. W rezultacie wiele państw wymyślało swoje własne, niekompatybilne sposoby kodowania oparte na 7-bitowym kodzie ASCII. Powstawało wiele kolidujących ze sobą schematów kodowania, używających tego samego ograniczonego zakresu kodów. Ta sama wartość w różnych systemach mogła reprezentować różne znaki, a określony znak mógł być przypisany w różnych systemach do różnych wartości.

Ustawienia lokalne były pierwszą próbą rozwiązania kwestii różnych języków i parametrów specyficznych dla danego kraju, jednak nie sprawdziły się zbyt dobrze jako mechanizm wyboru systemu kodowania znaków. Można je z powodzeniem stosować do celów nie związanych ze zbiorami znaków, takich jak lokalne ustawienia jednostek monetarnych, formatu daty i czasu czy zapisu liczb. Mało przydatne są jednak do kontrolowania wykorzystania tej samej 8-bitowej przestrzeni kodowej dla różnych zestawów znaków.

Główny problem to dokumenty wielojęzyczne. Stworzenie dokumentu używającego łaciny, greki i cyrylicy było bardzo kłopotliwe, ponieważ ten sam kod numeryczny mógł odpowiadać innemu znakowi w systemach odpowiadających tym językom. Na przykład kod numer 196 w ISO 8859-1 (Latin1) jest łacińską wielką literą A z dierezą; w ISO 8859-7 ten sam kod przedstawia grecką wielką literę delta. Program interpretujący kody numeryczne według ISO 8859-1 widziałby jeden znak, ale według ISO 8859-7 zupełnie co innego.

Trudno było łączyć różne zestawy znaków w tym samym dokumencie. Nawet jeżeli jakimś sposobem udało się połączyć coś ze sobą, to i tak niewiele programów było w stanie sobie z tym poradzić. Aby zobaczyć odpowiedni znak, należało wiedzieć w jakim systemie został zakodowany tekst, a mieszanie różnych systemów nie było łatwe. Mylne dobranie zestawu znaków kończyło się w najlepszym razie „papką" na ekranie.

Obsługa Unicode w Perlu

Unicode jest próbą ujednolicenia wszystkich zestawów znaków z całego świata, zawierających wiele różnych symboli, a nawet zbiorów fikcyjnych. W Unicode różne znaki mają własne kody numeryczne nazywane *punktami kodowymi* (ang. *code points*).

Dokumenty wielojęzyczne nie stwarzają w nim problemów, podczas gdy wcześniej ich tworzenie było wręcz niemożliwe. Do dyspozycji jest więcej niż 128 lub 256 znaków przypadających na jeden dokument. Gdy stosujemy Unicode, możliwe jest umieszczenie dziesiątek tysięcy zmieszanych ze sobą różnych znaków w jednym dokumencie bez wprowadzania zamieszania.

Nie ma żadnego problemu, gdy trzeba razem napisać, na przykład Ä lub Δ. Pierwszemu z tych znaków, w Unicode oficjalnie nazywanemu „LATIN CAPITAL LETTER A WITH DIAERSIS", przypisany jest punkt kodowy U+00C4 (jest to notacja stosowana w Unicode). Drugi znak, nazywany „GREEK CAPITAL LETTER DELTA" ma punkt kodowy U+0394. Różnym znakom przypisane są różne punkty kodowe, przez co nie ma pomyłek.

Perl obsługuje Unicode mniej więcej od wersji v5.6, ale uważa się, że dopiero w wersji 5.8 jego obsługa jest naprawdę solidna i w pełni funkcjonalna. Nie przez przypadek pokrywało się to z wprowadzeniem do Perla warstw wejścia-wyjścia obsługujących kodowanie. Bardziej szczegółowo temat ten został przedstawiony w rozdziale 8.

Obecnie wszystkie funkcje łańcuchowe i operatory w Perlu, łącznie z służącymi do dopasowania wzorców, działają na znakach, a nie na oktetach. Gdy zadamy w Perlu pytanie o długość łańcucha (funkcja `length`), zwrócona zostanie informacja ile jest znaków w łańcuchu, a nie ile bajtów ma dany łańcuch. Pobierając pierwsze trzy znaki z łańcucha za pomocą `substr`, w wyniku możemy otrzymać trzy lub więcej bajtów. Nie wiadomo, ale też nie ma potrzeby wiedzieć, ile ich rzeczywiście będzie. Nie warto troszczyć się o dokładne odwzorowanie poziomu bajtowego, ponieważ można się zbyt zagłębić w szczegóły. To naprawdę nie powinno mieć znaczenia — jeżeli jednak będzie miało, oznacza to, że implementacja Perla wciąż ma pewne braki. Prace nad tym wciąż trwają.

Ponieważ obsługiwane są znaki o punktach kodowych powyżej 256, argument funkcji `chr` nie jest już ograniczony do 256, podobnie jak funkcja `ord` nie ma ograniczenia do zwracania wartości mniejszych niż 256. Zapis `chr(0x394)` oznacza grecką literę Δ:

```
$char = chr(0x394);
$code = ord($char);
printf "znak %s ma kod %d, %#04x\n", $char, $code, $code;
```

```
znak Δ ma kod 916, 0x394
```

Sprawdzając długość łańcucha, otrzymamy w wyniku 1, ponieważ łańcuch ten ma tylko jeden znak. Warto zauważyć, że jest to liczba znaków, nie długość wyrażona w bajtach. Oczywiście wewnętrzna reprezentacja wymaga więcej niż 8 bitów dla tak dużego kodu. Jednak programista posługuje się znakami w ujęciu abstrakcyjnym, nie jako fizycznymi oktetami. Tego typu niskopoziomowe zagadnienia lepiej pozostawiać samemu Perlowi.

Programista nie powinien utożsamiać znaków z bajtami. Ten, kto operuje raz znakami, raz bajtami, popełnia ten sam błąd, co programista C, który beztrosko posługuje się zamiennie liczbami całkowitymi i wskaźnikami. Pomimo że na niektórych platformach wewnętrzna reprezentacja może być przypadkiem jednakowa, jest to tylko przypadek, a mieszanie interfejsów abstrakcyjnych z implementacją fizyczną na pewno przysporzy problemów w przyszłości.

Istnieje kilka sposobów na wprowadzenie znaków Unicode do kodu w Perlu. Jeżeli wykorzystywany edytor tekstowy umożliwia bezpośrednie korzystanie z Unicode, można poinformować o tym Perla przy użyciu dyrektywy `use utf8`. Innym sposobem jest zastosowanie sekwencji `\x` w łańcuchach interpolowanym do oznaczenia znaku za pomocą

jego punktu kodowego zapisanego szesnastkowo, np. `\xC4`. Znaki, których punkty kodowe przekraczają wartość 0xFF wymagają więcej niż dwóch cyfr heksadecymalnych, dlatego muszą być zapisywane w nawiasach.

```
print "\xC4 oraz \x{0394} wyglądają inaczej\n";
Ä oraz Δ wyglądają inaczej\n
```

Receptura 1.5 opisuje sposób użycia dyrektywy `use charnames` do wstawiania do łańcucha sekwencji `\N{NAZWA}`, na przykład w celu oznaczenia litery Δ można napisać `\N{GREEK CAPITAL LETTER DELTA}`, `\N{greek:Delta}` lub po prostu `\N{Delta}`.

Tyle informacji wystarczy, aby zacząć wykorzystywać Unicode w samym Perlu, jednak współpraca z innymi programami wymaga czegoś więcej.

Podczas korzystania ze starego, jednobajtowego sposobu kodowania, takiego jak ASCII lub ISO 8859-*n*, po wypisaniu znaku, którego kod numeryczny miał wartość *NN*, pojawiał się pojedynczy bajt o kodzie *NN*. To, co w rzeczywistości się ukazało, zależało od dostępnych czcionek, bieżących ustawień regionalnych i od kilku innych czynników. W Unicode nie obowiązuje bezpośrednie przekładanie logicznego numeru znaku (punktu kodowego) na fizyczne bajty. Zamiast tego muszą być one zakodowane w jednym z kilku dostępnych formatów wyjściowych.

Wewnętrznie Perl korzysta z formatu zwanego UTF-8, ale obsługuje także wiele innych formatów kodowania obowiązujących dla Unicode. Dyrektywa `use encoding` informuje Perla jakim sposobem został zakodowany dany skrypt oraz jakiego kodowania powinny używać standardowe uchwyty plików. Dyrektywa `use open` ustawia domyślne kodowanie dla wszystkich uchwytów. Specjalne argumenty funkcji `open` lub `binmode` określają format kodowania dla poszczególnych uchwytów. Opcja **–C** wiersza poleceń jest skrótem do ustawienia kodowania dla wszystkich (lub tylko standardowych) uchwytów oraz argumentów programu. Zmienne środowiskowe `PERLIO`, `PERL_ENCODING` oraz `PERL_UNICODE` zawierają różne związane z tym wskazówki.

1.1. Dostęp do wybranej części łańcucha

Problem

Chcemy odczytać lub zmienić jedynie pewien fragment wartości zmiennej łańcuchowej. Na przykład po odczytaniu rekordu o ustalonej długości, chcemy wyodrębnić poszczególne pola.

Rozwiązanie

Funkcja `substr` umożliwia odczytywanie i zapisywanie znaków w określonym miejscu w łańcuchu.

```
$value = substr($string, $offset, $count);
$value = substr($string, $offset);

substr($string, $offset, $count) = $newstring;
substr($string, $offset, $count, newstring);    # to samo co powyżej
substr($string, $offset)         = $newtail;
```

Funkcja `unpack` umożliwia jedynie odczyt, jednak działa szybciej podczas ekstrahowania wielu podłańcuchów.

```
# pobranie 5-bajtowego łańcucha z pominięciem 3 bajtów,
# pobranie dwóch 8-bajtowych łańcuchów i pozostałych
# (uwaga: działa to tylko na danych ASCII, nie działa na danych zakodowanych w Unicode)
($leading, $s1, $s2, $trailing) = unpack("A5 x3 A8 A8 A*", $data);

# podział na obszary 5-bajtowe
@fivers = unpack("A5" x (length($string)/5), $string);

# zamiana łańcucha na pojedyncze jednobajtowe znaki
@chars = unpack("A1" x length($string), $string);
```

Analiza

Łańcuchy są podstawowymi typami danych; nie są tablicami typów podstawowych. W Perlu dostęp do pojedynczych znaków lub części znaków odbywa się przy użyciu funkcji `unpack` lub `substr`, a nie poprzez indeksowanie, jak w niektórych językach programowania.

Argument `$offset` funkcji `substr` wskazuje początek poszukiwanego podzbioru znaków, licząc od początku łańcucha — jeżeli jest dodatni lub licząc od końca — jeżeli jest ujemny. Jeżeli wartością tego argumentu będzie 0, to poszukiwany podzbiór zaczyna się na początku łańcucha. Argument `$count` oznacza długość podzbioru.

```
$string = "Dany jest łańcuch znaków";
#         +012345678901234567890123     indeksowanie rosnące   (od lewej do prawej)
#         432109876543210987654321-     indeksowanie malejące  (od prawej do lewej)
#                                       uwaga: powyżej 0 oznacza 10, 20, itd.
$first = substr($string, 0, 1);     # "D"
$start = substr($string, 5, 4);     # "jest"
$rest =  substr($string, 10);       # "łańcuch znaków"
$last =  substr($string, -1);       # "w"
$end =   substr($string, -6);       # "znaków"
$piece = substr($string, -14, 7);   # "łańcuch"
```

Za pomocą funkcji `substr` można zrobić więcej niż tylko odczytać fragment łańcucha; można go nawet zmodyfikować. Wynika to z tego, że `substr` jest szczególnym rodzajem funkcji — funkcją *l-wartościową*, a więc taką, której wartości zwracanej można przypisać wartość (dla porządku, pozostałe tego rodzaju funkcje to: `vec`, `pos` oraz `keys`. W pewnym sensie `local`, `my` oraz `our` można również traktować jak funkcje l-wartościowe).

```
$string = "I dany jest łańcuch znaków";
print $string;
I dany jest łańcuch znaków
```

```
substr($string, 7, 4)= "był";          # zmiana "jest" na "był"
I dany był łańcuch znaków
substr($string, -12)= "twy problem";   # "I dany był łatwy problem"
I dany był łatwy problem
substr($string, 0, 1)= "";             # usuwa pierwszy znak
dany był łatwy problem
substr($string, -14)= "";              # usuwa ostatnie 14 znaków
dany był
```

Użycie operatora =~ oraz operatorów `s///`, `m//` lub `tr///` w połączeniu z funkcją `substr` powoduje, że modyfikują one jedynie część znaków z łańcucha.

```
# można przetestować podzbiory znaków za pomocą =~
if (substr($string, -10)=~/pattern/){
   print "Dopasowano wzorzec do ostatnich 10 znaków\n";
}

# zamiana "op" na "er" w pierwszych pięciu znakach
substr($strings, 0, 5) =~ s/op/er/g;
```

Można nawet przestawiać wartości przez zastosowanie kilku funkcji `substr` po każdej stronie przypisania:

```
# wymiana pierwszej litery z ostatnią literą w łańcuchu
$a = "abcdefgh";
(substr($a,0,1), substr($a,-1))=
(substr($a,-1), substr($a,0,1));
print $a;
hbcdefga
```

Pomimo że `unpack` nie jest funkcją l-wartościową, podczas jednoczesnego ekstrahowania wielu wartości jest szybsza niż `substr`. Należy przekazać do niej ciąg formatujący, opisujący kształt przetwarzanego rekordu. Do pozycjonowania służą trzy symbole: `"x"` z liczbą bajtów, które należy pominąć w kierunku końca łańcucha, `"X"` z liczbą bajtów do opuszczenia w kierunku początku łańcucha oraz `"@"` do przejścia do bezwzględnej (wyrażonej w bajtach) pozycji w danym rekordzie (symbole te trzeba stosować z duża rozwagą, jeżeli dane zawierają łańcuchy w Unicode, ponieważ operują ściśle na bajtach, a poruszanie się bajtowo po wielobajtowych danych jest co najmniej bardzo ryzykowne).

```
# ekstrahowanie kolumny przy użyciu funkcji unpack
$a = "To be or not to be";
$b = unpack("x6 A6", $a);              # pominięcie 6, przechwycenie 6
print $b;
or not

($b, $c) = unpack("x6 A2 X5 A2", $a);  # w przód 6, przechwycenie 2; wstecz 5, przechwycenie 2
print "$b\n$c\n";
or
be
```

Czasami wygodnie jest wyobrażać sobie dane tak, jakby były poprzecinane w określonych kolumnach. Na przykład załóżmy, że chcemy umieścić punkty przecięcia bezpośrednio przed pozycjami: 8, 14, 20, 26 oraz 30. Są to numery kolumn, w których rozpoczynają się

poszczególne pola. Można oczywiście obliczyć, że odpowiedni format dla funkcji `unpack` będzie wynosił `"A7 A6 A6 A6 A4 A*"`, ale dla twórczo leniwego programisty byłby to zbyt wielki wysiłek. Lepiej, by Perl obliczył to za nas. Wykorzystajmy funkcję `cut2fmt`:

```
sub cut2fmt {
   my(@positions) = @_;
   my $template = '';
   my $lastpos = 1;
   foreach $place (@positions) {
      $template .= "A" . ($place - $lastpos) . " ";
      $lastpos = $place;
   }
   $template .= "A*";
   return $template;
}

$fmt = cut2fmt(8, 14, 20, 26, 30);
print "$fmt\n";
```

```
A7 A6 A6 A6 A4 A*
```

Możliwości funkcji `unpack` wykraczają daleko poza zwykłe przetwarzanie tekstu. Stanowi ona swego rodzaju bramę między danymi tekstowymi a binarnymi.

W recepturze tej założyliśmy, że wszystkie dane znakowe są 7- lub 8-bitowe, w związku z tym operacje bajtowe z wykorzystaniem funkcji `pack` działają prawidłowo.

Zobacz również

Opis funkcji `pack`, `unpack` oraz `substr` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*; wykorzystanie procedury *cut2fmt* z receptury 1.24; zastosowanie funkcji `unpack` do przetwarzania danych binarnych w recepturze 8.24.

1.2. Ustalanie wartości domyślnych

Problem

Chcielibyśmy nadać wartość domyślną pewnej zmiennej skalarnej, ale tylko wtedy, gdy nie została ona określona w inny sposób. Często potrzebna jest pewna zakodowana wartość domyślna, która może być nadpisana z wiersza poleceń lub poprzez zmienną środowiskową.

Rozwiązanie

Należy użyć operatorów `||` lub `||=`, które działają zarówno na łańcuchach, jak i na wartościach liczbowych:

```
# użyj zmiennej $b, jeżeli jej wartość odpowiada logicznej prawdzie, w przeciwnym razie użyj $c
$a = $b || $c;

# jeżeli zmienna $x ma wartość logicznie nieprawdziwą, przypisz jej wartość $y
$x ||= $y;
```

Jeżeli `0`, `"0"` oraz `""` należą do dopuszczalnych wartości wykorzystywanej zmiennej, można użyć polecenia `defined`:

```
# użyj zmiennej $b, jeżeli została wcześniej zdefiniowana, w przeciwnym razie użyj zmiennej $c
$a = defined($b) ? $b : $c

# „nowo" zdefiniowany operator 'lub', który pojawi się w przyszłej wersji Perla
use v5.9;
$a = $b // $c;
```

Analiza

Różnica między dwoma powyższymi sposobami (`defined` i `||`) polega na warunku, który testują: zdefiniowanie i logiczna prawda. W Perlu trzy zdefiniowane wartości są logicznym fałszem: `0`, `"0"` oraz `""`. Jeżeli dana zmienna przechowuje taką wartość i chcielibyśmy ją zachować, to operator `||` nie zadziała poprawnie. Zamiast tego trzeba będzie użyć bardziej złożonego testu z operatorem `defined`. Często wygodnie jest tak zorganizować program, by nie operował na wartościach zdefiniowanych czy niezdefiniowanych, ale tylko na logicznej prawdzie i fałszu.

Zbiór wartości zwracanych przez operator `||` w Perlu nie jest ograniczony tylko do 0 lub 1, jak to ma miejsce w innych językach programowania. Zwraca on pierwszy operand (znajdujący się po jego lewej stronie), jeżeli jest on prawdą; w przeciwnym razie zwraca drugi operand. Operator `&&` także zwraca ostatnio obliczone wyrażenie, jednak właściwość ta jest rzadziej używana. Dla operatorów tych nie ma znaczenia czy ich operandy są łańcuchami, liczbami czy odwołaniami — mogą być każdym rodzajem skalara. Po prostu zwracają ten argument, który powoduje, że całe wyrażenie jest albo prawdą, albo fałszem. Nie ma to wpływu na logiczny sens zwracanej wartości, jednak sprawia, iż zwracane wartości stają się bardziej użyteczne.

Ta właściwość pozwala na przypisanie domyślnej wartości do zmiennej, funkcji lub dłuższego wyrażenia, jeżeli pierwsza część wyrażenia z operatorem nie jest poprawną wartością. Oto przykład wykorzystania operatora `||`, który przypisze zmiennej `$foo` zmienną `$bar` lub, w przypadku gdy zmienna `$bar` będzie miała wartość logicznego fałszu, `"WARTOŚĆ DOMYŚLNĄ"`:

```
$foo = $bar || "WARTOSC DOMYSLNA";
```

Oto kolejny przykład, w którym zmiennej `$dir` zostanie przypisany pierwszy argument, z którym program został uruchomiony lub w przypadku braku argumentów, wartość `"/tmp"`.

```
$dir = shift(@ARGV) || "/tmp";
```

To samo można zrobić bez zmieniania @ARGV:

```
$dir = $ARGV[0] || "/tmp";
```

Jeżeli 0 jest poprawną wartością dla $ARGV[0], to nie można użyć operatora ||, ponieważ dla niego jest to logiczny fałsz, mimo że jest to wartość dopuszczalna. Należy skorzystać z jedynego w Perlu operatora trójargumentowego: ?::

```
$dir = defined($ARGV[0]) ? shift(@ARGV) : "/tmp";
```

Można to również zapisać następująco — ale zmieniamy lekko semantykę:

```
$dir = @ARGV ? $ARGV[0] : "/tmp";
```

Wyrażenie to sprawdza liczbę elementów w tablicy @ARGV, ponieważ pierwszy argument (tutaj @ARGV) obliczany jest w kontekście skalarnym. Przyjmie on wartość fałszu tylko wtedy, gdy w zmiennej @ARGV będzie 0 elementów; w takim przypadku zmienna $dir przyjmie wartość "/tmp". W przeciwnym razie (gdy użytkownik poda argument) wykorzystany zostanie pierwszy argument programu.

Poniższa linia kodu inkrementuje wartość elementu w tablicy %count, używając jako klucza zmiennej $shell lub, jeżeli $shell jest wartością fałszywą, wartości "/bin/sh".

```
$count{ $shell || "/bin/sh" }++;
```

Można także połączyć kilka wariantów, tak jak w kolejnym przykładzie. Wykorzystane zostanie pierwsze wyrażenie, które zwróci prawdę.

```
# ustalenie nazwy użytkownika w systemach uniksowych
$user = $ENV{USER}
   || $ENV{LOGNAME}
   || getlogin()
   || (getpwuid($<))[0]
   || "Nieznany numer uid $<";
```

Analogicznie działa operator &&: zwraca pierwszy argument, jeżeli jest on fałszem; w przeciwnym razie zwraca drugi argument. Właściwość ta jest rzadko używana, ponieważ interesujących wartości oznaczających logiczny fałsz nie ma tak wiele jak wartości oznaczających logiczną prawdę. Przykład wykorzystania tej własności operatora && przedstawiono w recepturze 13.12 oraz w recepturze 14.19.

Operator przypisania ||= wygląda dość osobliwie, jednak działa dokładnie tak samo jak pozostałe binarne operatory przypisania. W przypadku prawie wszystkich operatorów binarnych w Perlu zapis: $ZMIENNA OP= WARTOSC znaczy to samo, co $ZMIENNA = $ZMIENNA OP WARTOSC; przykładowo $a +=$b znaczy to samo, co $a = $a + $b. Tak więc operator ||= używany jest do ustawienia zmiennej, gdy jest ona logicznym fałszem. Operator || jest zwykłym operatorem boolowskim — sprawdzającym logiczną prawdę — nie „przeszkadzają" mu wartości niezdefiniowane, nawet wtedy, gdy aktywowane jest generowanie ostrzeżeń.

Oto przykład wykorzystania `||=`, który ustawia zmienną `$starting_point` na wartość `"Greenwich"`, chyba że jej wartość została już wcześniej ustalona. Zakładamy, że zmienna `$starting_point` nie będzie miała wartości `0` lub `"0"` — a jeżeli tak, to może być ona zmieniona.

```
$starting_point ||= "Greenwich";
```

Przy przypisaniach nie można stosować operatora `or` zamiast `||`, ponieważ ten pierwszy jest za nisko w hierarchii pierwszeństwa. Zapis `$a = $b or $c` jest ekwiwalentny `($a = $b) or $c`. W ten sposób zawsze następuje przypisanie `$b` do `$a`, a nie o to nam w tym przypadku chodzi.

Nie należy stosować tych, odnoszących się do skalarów, własności operatorów `||` i `||=` do tablic zwykłych lub asocjacyjnych. Nie będzie to działać poprawnie, ponieważ operatory te stosują wobec lewego operandu kontekst skalarny. Zamiast tego trzeba zastosować zapis podobny do poniższych:

```
@a = @b unless @a;     # skopiuj, jeżeli puste
@a = @b ? @b : @c;     # przypisz @b, jeżeli nie jest puste, w przeciwnym razie przypisz @c
```

W przyszłych wersjach Perla mają być dostępne nowe operatory: `//`, `//=` oraz `err`. Te operatory „zdefiniowanego-lub", będą działały tak samo jak operatory „logicznego-lub", `||`, z tą różnicą, że będą sprawdzały zdefiniowanie, a nie prawdę. Wówczas dwie poniższe pary wyrażeń będą równoważne:

```
$a = defined($b) ? $b : $c;
$a = $b // $c;

$x = defined($x) ? $x : $y;
$x //= $y;

defined(read(FH, $buf, $count)) or die "nieudany odczyt: $!";
read(FH, $buf, $count)         err die "nieudany odczyt: $!";
```

Te trzy operatory istnieją już w wydaniu 5.9 Perla, ale, jak wynika z nieparzystego numeru wersji, jest to wydanie eksperymentalne, nieprzygotowane do działania w środowisku produkcyjnym. Operatory te będą prawdopodobnie obsługiwane w stabilnej wersji 5.10, a na pewno w wersji 6, której data wydania nie jest jeszcze znana.

Zobacz również

Opis operatora `||` na stronie *perlop*(1) oraz w rozdziale 3. książki *Perl. Programowanie*; opis funkcji `defined` oraz `exists` na stronie *perlop*(1) oraz w rozdziale 29. książki *Perl. Programowanie*.

1.3. Zamiana wartości bez korzystania ze zmiennych tymczasowych

Problem

Trzeba wymienić wartości pomiędzy dwiema zmiennymi skalarnymi bez korzystania ze zmiennych tymczasowych.

Rozwiązanie

Należy użyć przypisania listowego ze zmianą kolejności zmiennych.

```
($VAR1, $VAR2) = ($VAR2, $VAR1);
```

Analiza

W większości języków programowania zamiana wartości dwóch zmiennych wymaga wykonania kroków pośrednich:

```
$temp   = $a;
$a      = $b;
$b      = $temp;
```

W Perlu tak nie jest. Analizuje on obie strony przypisania, gwarantując, że nie utraci się przypadkowo wartości którejś ze zmiennych. W ten sposób znika konieczność stosowania zmiennej tymczasowej:

```
$a      = "alfa";
$b      = "omega";
($a, $b) = ($b, $a);     # ostatni będą pierwszymi – i odwrotnie
```

Można też zamieniać więcej niż jedną zmienną:

```
($alfa, $beta, $teta) = qw(Styczeń Marzec Sierpień);
# przenieś beta do alfa,
# przenieś teta do beta,
# przenieś alfa do teta
($alfa, $beta, $teta) = ($beta, $teta, $alfa);
```

Po wykonaniu tego kodu, zmienne `$alfa`, `$beta` i `$teta` będą miały wartości `"Marzec"`, `"Sierpień"`, `"Styczeń"`.

Zobacz również

Sekcja „List value constructors" strony *perldata*(1) oraz podrozdział „Listy i tablice" w rozdziale 2. książki *Perl. Programowanie*.

1.4. Konwersja między znakami a liczbami

Problem

Chcielibyśmy wypisać liczbę reprezentowaną przez dany znak lub odwrotnie: wypisać znak reprezentowany przez daną liczbę.

Rozwiązanie

Aby zamienić znak na liczbę, należy użyć funkcji `ord`; aby zamienić liczbę na odpowiadający jej znak, należy skorzystać z funkcji `chr`:

```
$num = ord($char);
$char = chr($num);
```

Znak formatujący `%c` używany w funkcjach `printf` i `sprintf` również zamienia liczbę na znak:

```
$num=101;
$char = sprintf("%c", $num);    # wolniejsze niż chr($num)
printf("Liczba %d to znak %c\n", $num, $num);
Liczba 101 to znak e
```

Stosując w funkcjach `pack` i `unpack`, za pomocą szablonu `C*` możemy szybko przekonwertować wiele 8-bitowych bajtów; w podobny sposób można stosować szablon `U*` w odniesieniu do znaków w Unicode.

```
@bytes = unpack("C*", $string);
%string = pack("C*", @bytes);

$unistr = pack("U4", 0x24b6, 0x24b7, 0x24b8, 0x24b9);
@unichars = unpack("U*", $unistr);
```

Analiza

W przeciwieństwie do języków niskiego poziomu, takich jak asembler, Perl nie traktuje znaków i liczb wymiennie; tutaj wymiennie traktowane są *łańcuchy* i liczby. Oznacza to, że nie ma w nim możliwości przypisywania liczb znakom i odwrotnie. Do konwersji znaku na odpowiadającą mu liczbę i odwrotnie służą, zaczerpnięte z Pascala, funkcje `chr` i `ord`:

```
$value      = ord("e");     # teraz wartość 101
$character  = chr(101);     # teraz znak "e"
```

Znak jest w rzeczywistości reprezentowany przez łańcuch o długości jeden, można go więc wypisać bezpośrednio przy użyciu funkcji `print` lub symbolu formatującego `%s` w funkcjach `printf` lub `sprintf`. Format `%c` wymusza na tych funkcjach przeprowadzenie konwersji liczby na znak; nie jest to stosowane do wypisywania znaków, które są już zapisane w formacie znakowym (czyli łańcuchowym).

```
printf("Liczba %d jest znakiem %c\n", 101, 101);
```

Funkcje `pack`, `unpack`, `chr` oraz `ord` są szybsze niż funkcja `sprintf`. Oto przykład zastosowania funkcji `pack` i `unpack`:

```
@ascii_character_numbers = unpack("C*", "demo");
print "@ascii_character_numbers\n";
100 101 109 111
$word = pack("C*", @ascii_character_numbers);
$word = pack("C*", 100,101, 109, 111);    # to samo
print "$word\n";
demo
```

A to sposób konwersji słowa `HAL` na `IBM`:

```
$hal = "HAL";
@byte = unpack("C*", $hal);
foreach $val (@byte) {
   $val++;          # dodaj jeden do każdego bajta
}
$ibm = pack("C*",@byte);
print "$ibm\n";    # wypisze "IBM"
```

Przy jednobajtowych danych znakowych, takich jak w zwykłym kodzie ASCII czy jednym z wielu zestawów znaków ISO 8859, funkcja `ord` zwraca wartości liczbowe z zakresu od 0 do 255. Odpowiada to typowi `unsigned char` z języka C.

Jednak Perl może więcej — posiada zintegrowaną obsługę uniwersalnego kodowania znaków: Unicode. Jeżeli do funkcji `chr`, `sprintf "%c"` lub `pack` przekaże się wartość większą niż 255, to w wyniku otrzyma się łańcuch w Unicode.

Oto podobne do powyższych operacje z wykorzystaniem Unicode:

```
@unicode_points = unpack("U*", "fac\x{0327}ade");
print "@unicode_points\n";
102 97 99 807 97 100 101

$word = pack("U*", @unicode_points);
print "$word\n";
façade
```

Jeśli chcemy tylko wypisać wartości liczbowe odpowiadające znakom, nie musimy koniecznie stosować funkcji `unpack`. Funkcje `printf` i `sprintf` rozumieją modyfikator `v`, działający w następujący sposób:

```
printf "%vd\n", "fac\x{0327}ade";
102.97.99.807.97.100.101

printf "%vx\n", "fac\x{0327}ade";
66.61.63.327.61.64.65
```

Wypisywane są wartości liczbowe (czyli „punkty kodowe” w nomenklaturze Unicode) poszczególnych znaków z łańcucha rozdzielone kropkami.

Zobacz również

Opis funkcji `chr`, `ord`, `printf`, `sprintf`, `pack` oraz `unpack` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*.

1.5. Stosowanie nazwanych znaków Unicode

Problem

Chcielibyśmy zastosować nazwy Unicode do nietypowych znaków stosowanych w kodzie programu bez konieczności poznawania wartości ich punktów kodowych.

Rozwiązanie

Na początku pliku należy umieścić dyrektywę `use charnames`. Dzięki temu w łańcuchach będzie można stosować sekwencje `"\N{NAZWA}"`.

Analiza

Dyrektywa `use charnames` pozwala na stosowanie symbolicznych nazw znaków Unicode. Są to wartości stałe, do których dostęp uzyskuje się za pomocą ujętej w cudzysłów sekwencji `\N{NAZWA}`. Obsługiwanych jest kilka subdyrektyw. Subdyrektywa `:full` daje dostęp do wszystkich nazw znaków, ale z koniecznością zapisywania ich dokładnie w taki sposób, w jaki występują w bazie znaków Unicode, uwzględniając przy tym notację wielkich liter. Subdyrektywa `:short` umożliwia stosowanie wygodnych skrótów. Każdy import zapisany bez dwukropka będzie potraktowany jako nazwa skryptu, co skutkuje utworzeniem, niezależnego od wielkości liter, skrótu do tego (tych) skryptu.

```
use charnames ':ful';
print "\N{GREEK CAPITAL LETTER DELTA} jest literą delta.\n";
Δ jest literą delta.

use charnames ':short';
print "\N{greek:Delta} jest wielką literą delta.\n";
Δ jest wielką literą delta.

use charnames qw(cyrilic greek);
print "\N{Sigma} oraz \N{sigma} są greckimi literami sigma.\n";
print "\N{Be} oraz \N{be} są literami beta w cyrylicy.\n";
Σ oraz σ są greckimi literami sigma.
Б oraz б są literami beta w cyrylicy.
```

Dwie funkcje dokonują translacji między liczbowymi punktami kodowymi a długimi nazwami, są to: `charnames::viacode` oraz `charnames::vianame`. Dokumenty Unicode używają notacji U+*XXXX* do oznaczenia znaku, którego punkt kodowy jest równy *XXXX*. Wykorzystajmy tę notację przy generowaniu danych wyjściowych:

```
use charnames qw(:full);
for $code (0xC4, 0x394){
   printf "Nazwa znaku U+%04X (%s) to: %s\n",
      $code, chr($code), charnames::viacode($code);
}
Nazwa znaku U+00C4 (Ä) to: LATIN CAPITAL LETTER A WITH DIARESIS
Nazwa znaku U+0394 (Δ) to: GREEK CAPITAL LETTER DELTA

use charnames qw(:full);
$name = "MUSIC SHARP SIGN";
$code = charnames::vianame($name);
printf "%s to znak U+%04X (%s)\n",
   $name, $code, chr($code);
MUSIC SHARP SIGN to znak U+266F ( # )
```

Poniższe polecenie umożliwi lokalizację kopii bazy danych ze znakami Unicode stosowanej w Perlu:

```
% perl -MConfig -le 'print "$Config{privlib}/unicore/NamesList.txt"'
/usr/local/lib/perl5/5.8.1/unicore/NamesList.txt
```

W tym pliku można znaleźć zestawienie dostępnych nazw znaków.

Zobacz również

Strona *charnames*(3) podręcznika systemowego oraz rozdział 31. książki *Perl. Programowanie*; baza danych znaków Unicode na stronie *http://www.unicode.org/*.

1.6. Przetwarzanie łańcucha znak po znaku

Problem

Trzeba przetworzyć łańcuch znak po znaku.

Rozwiązanie

Aby rozbić łańcuch na pojedyncze znaki, należy użyć funkcji `split` w połączeniu z pustym wzorcem lub, jeżeli istnieje potrzeba poznania wartości odpowiadających znakom, skorzystać z funkcji `unpack`:

```
@array = split(//, $string);       # każdy element jest pojedynczym znakiem
@array = unpack("U*", $string);    # każdy element jest punktem kodowym (liczbą)
```

Można też odczytywać kolejne znaki w pętli:

```
while (/(.)/g) {    # . nie oznacza znaku przejścia do nowej linii
      # zmienna $1 zawiera znak, ord($1) - odpowiadającą mu liczbę
   }
```

Analiza

Jak wspominaliśmy wcześniej, podstawową jednostką w Perlu jest łańcuch, nie znak. Potrzeba przetwarzania łańcucha znak po znaku występuje stosunkowo rzadko. Zazwyczaj problemy da się wygodniej rozwiązać za pomocą pewnej operacji wyższego poziomu — dopasowania wzorców. Przykładowo, w recepturze 7.14 zestaw podstawień używany jest do odczytywania argumentów wiersza poleceń.

Funkcja `split` z wzorcem pasującym do łańcucha pustego zwraca listę poszczególnych znaków w łańcuchu. Jest to wygodna właściwość przy stosowaniu zamierzonym, jednak łatwo doprowadzić do tego w sposób niezamierzony. Na przykład: wzorzec `/X*/` pasuje do wszystkich możliwych, w tym również pustych, łańcuchów.

Oto przykład wypisujący znaki użyte w łańcuchu `"an apple a day"` uporządkowane w kolejności rosnącej:

```
%seen = ();
$string = "an apple a day";
foreach $char (split //, $string){
   $seen{$char}++;
}
print "Użyte litery to: ", sort(keys %seen), "\n";
```
Użyte litery to: adelnpy

Podane w rozwiązaniu działania funkcji `split` i `unpack` zwracają tablice znaków. Jeżeli nie jest nam potrzebna tablica, powinniśmy użyć wzorca z modyfikatorem `/g` w pętli `while`, pobierając znak po znaku z łańcucha:

```
%seen = ();
$string = "an apple a day";
while ($string =~ /(.)/g) {
   $seen{$1}++;
}
print " Użyte litery to: ", sort(keys %seen), "\n";
```
Użyte litery to: adelnpy

Ogólnie mówiąc: każde przetwarzanie poszczególnych znaków łańcucha da się najprawdopodobniej zastąpić innym rozwiązaniem. Łatwiejsze może być zastosowanie wzorca niż używanie funkcji `index` i `substr` czy `split` i `unpack`. Zamiast ręcznie obliczać 32-bitową sumę kontrolną, jak w poniższym przykładzie, można wykorzystać do tego funkcję `unpack`, która zrobi to zdecydowanie efektywniej.

W poniższym przykładzie obliczana jest suma kontrolna dla zmiennej `$string` za pomocą pętli `foreach`. Istnieją lepsze sumy kontrolne; ta użyta tutaj jest tradycyjnie stosowana i prosta obliczeniowo. Tworzenie dużo lepszych sum kontrolnych umożliwia standardowy[2] moduł `Digest::MD5`.

[2] Jest on standardowo dołączany do Perla w wersji 5.8; gdy używamy innej, trzeba go pobrać z sieci CPAN.

```
$sum = 0;
foreach $byteval (unpack("C*", $string)) {
   $sum += $byteval;
}
print "suma wynosi $sum\n";
# w przypadku łańcucha "an apple a day" suma wyniesie "1248"
```

Poniższa linia robi to samo, ale dużo szybciej:

```
$sum = unpack("%32C*", $string);
```

Oto program emulujący dostępny w systemie SysV program do obliczania sum kontrolnych:

```
#!/usr/bin/perl
# sum — obliczanie 16-bitowej sumy kontrolnej dla wszystkich plików wejściowych
$checksum = 0;
while (<>) { $checksum += unpack("%16C*", $_) }
$checksum %= (2 ** 16) - 1;
print "$checksum\n";
```

A to przykład jego użycia:

```
%perl sum /etc/termcap
1510
```

Gdy dysponujemy programem *sum* w wersji GNU, trzeba wywołać go z opcją **--sysv**, aby otrzymać taką samą odpowiedź dla tego samego pliku.

```
% sum --sysv /etc/termcap
1510 851 /etc/termcap
```

Kolejny krótki program, *slowcat* z przykładu 1.1, przetwarza dane wejściowe znak po znaku, wstrzymując na krótko wyświetlanie tekstu po każdym znaku, tak by przesuwał się na tyle wolno, by wygodnie go było odczytać.

Przykład 1.1. Program slowcat

```
#!/usr/bin/perl
# slowcat — symulować p o w o l n e wyświetlanie linii
# użycie: slowcat [-DELAY] [pliki ...]
$DELAY = ($ARGV[0] =~ /^-([.\d]+)/) ? (shift, $1) : 1;
$| = 1;
while (<>) {
   for (split(//)){
      print;
      select(undef, undef, undef, 0.005 * $DELAY);
   }
}
```

Zobacz również

Opis funkcji `split` oraz `unpack` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*; wykorzystanie funkcji `select` do odmierzania czasu omówiono w recepturze 3.10.

1.7. Odwracanie kolejności słów lub znaków w łańcuchu

Problem

Trzeba odwrócić kolejność słów lub znaków w łańcuchu.

Rozwiązanie

Do odwrócenia kolejności znaków należy użyć funkcji `reverse` w kontekście skalarnym:

```
$revchars = reverse($string);
```

Do przestawienia wyrazów należy użyć funkcji `reverse` w kontekście listowym w połączeniu z funkcjami `split` i `join`:

```
$revwords = join(" ", reverse split(" ", $string));
```

Analiza

Funkcja `reverse` to w rzeczywistości dwie różne funkcje w jednej. Wywołana w kontekście skalarnym łączy swoje argumenty i zwraca jeden łańcuch z odwróconą kolejnością znaków. Wywołana w kontekście listowym, zwraca w odwrotnej kolejności podane argumenty. Wykorzystując funkcję `reverse` do operowania na znakach, musimy użyć słowa kluczowego `scalar`, aby wymusić kontekst skalarny, chyba że jest to oczywiste.

```
$gnirts    = reverse($string);    # odwrócenie kolejności liter w zmiennej $string
@sdrow     = reverse(@words);     # odwrócenie kolejności elementów w tablicy @words
$confused = reverse(@words);      # odwrócenie liter w join("", @words)
```

Oto przykład zamiany kolejności słów w łańcuchu. Użycie pojedynczej spacji, `" "`, jako wzorca dla funkcji `split` ma szczególne znaczenie. Powoduje, że funkcja ta będzie traktowała obszary znaków spacji jako separatory słów, a przy tym usuwała początkowe puste pola, podobnie jak *awk*. Normalnie `split` usuwa puste pola tylko z końca.

```
# odwrócenie kolejności wyrazów
$string = 'Yoda rzekł, "czy widzisz to?"';
@allwords = split(" ", $string);
$revwords = join(" ", reverse @allwords);
print $revwords, "\n";
to?" widzisz "czy rzekł, Yoda
```

Można się pozbyć tablicy tymczasowej `@allwords` i to samo napisać w jednej linii:

```
$revwords = join(" ", reverse split(" ", $string));
```

Wszelkie (wielokrotnie) powtórzone spacje w zmiennej `$string` stają się pojedynczymi spacjami w zmiennej `$revwords`. Jeżeli zależy nam na zachowaniu oryginalnej liczby spacji, należy użyć konstrukcji:

```
$revwords = join(" ", reverse split(/(\s+)/, $string));
```

Jednym z zastosowań funkcji `reverse` jest sprawdzanie czy dany wyraz jest palindromem (palindrom — słowo, które czytane wprost i wspak brzmi jednakowo i oznacza to samo):

```
$word = "kajak";
$is_palindrome = ($word eq reverse($word));
```

Powyższe można zamienić na jednoliniowe wyrażenie wyszukujące długie palindromy w pliku */usr/dict/words*:

```
%perl -nle 'print if $_ eq reverse && length >5' /usr/dict/words
deedeed
degged
deified
denned
hallah
kakkak
murdrum
redder
repaper
retter
reviver
rotator
sooloos
tebbet
terret
tut-tut
```

Zobacz również

Opis funkcji `split`, `reverse` i `scalar` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*; receptura 1.8.

1.8. Traktowanie dołączonych znaków Unicode jako pojedynczych znaków

Problem

Dany jest łańcuch w Unicode, zawierający znaki dołączone. Chcielibyśmy każdą taką sekwencję potraktować jako pojedynczy znak logiczny.

Rozwiązanie

Musimy przetworzyć ten łańcuch, korzystając w wyrażeniach regularnych z metaznaku `\X`.

```
$string = "fac\x{0327}ade";     # "façade"
$string =~ /fa.ade/;            # źle
$string =~ /fa\Xade/;           # dobrze

@chars = split(//, $string);    # 7 liter w zmiennej @chars
@chars = $string =~ /(.)/g;     # jak wyżej
@chars = $string =~ /(\X)/g;    # 6 „liter" w zmiennej @chars
```

Analiza

W Unicode można łączyć znak bazowy z jednym lub kilkoma następującymi po nim znakami nie oznaczającymi spacji; są to z reguły znaki diakrytyczne, takie jak oznaczenia akcentowania, cedille czy tyldy. Ze względu na fakt, iż występuje wiele takich *znaków dołączonych* (ang. *combined characters*), w głównej mierze po to, by uwzględnić znaki z narodowych systemów kodowania, mogą występować sytuacje, w których tę samą treść da się zapisać na kilka różnych sposobów.

Na przykład w słowie `"façade"` litera występująca pomiędzy „a" może zostać zapisana jako jeden znak: `"\x{E7}"`, pochodzący wprost z Latin1 (ISO 8859-1). Ciąg ten może być zakodowany jako dwubajtowa sekwencja w wewnętrznej reprezentacji UTF-8 Perla, jednak będą się one nadal liczyły jako pojedynczy znak. Mechanizm ten działa bezproblemowo.

Wiąże się z tym jednak pewna drażliwa kwestia. Innym sposobem na zapisanie U+00E7 jest wykorzystanie dwóch różnych punktów kodowych: zwykłego „c" i następującego po nim `"\x{0327}"`. Punkt kodowy U+0327 jest nie będącym spacją znakiem dołączonym, oznaczającym „cofnięcie się i dodanie cedilli pod poprzedzającym go znakiem bazowym".

Czasami zachodzi potrzeba, aby Perl traktował każdą taką sekwencję znaków dołączonych jako jeden znak logiczny. Ponieważ jednak są to dwa różne punkty kodowe, operacje znakowe Perla, wliczając w to funkcje `substr`, `length` oraz metaznaki wyrażeń regularnych, takie jak `/./` czy `/[^abc]/`, traktują znaki dołączone jako oddzielne jednostki.

W wyrażeniach regularnych metaznak `\X` odpowiada rozszerzonym sekwencjom znaków dołączonych Unicode i jest dokładnie ekwiwalentny `(?:\PM\pM*)`, co, gdy zapiszemy w dłuższej formie, oznacza:

```
(?x:        # początek nieprzechwytującej grupy znaków
    \PM     # jeden znak bez własności M (mark),
            #    taki jak litera
    \pM     # jeden znak, który ma własność M (mark),
            #    taki jak znak akcentu
    *       # dowolna liczba wystąpień takich znaków
)
```

Gdyby nie istniał ten metaznak, wykonanie prostych operacji na łańcuchach zawierających takie sekwencje byłoby utrudnione. Rozważmy na przykład sposoby odwracania kolejności słów i znaków podane w poprzedniej recepturze. Słowa „annče" oraz „niño" zapisane w Perlu z użyciem znaków dołączonych przybierają postać ciągów: `"anne\x{301}e"` i `"nin\x{303}o"`.

```
for $word ("anne\x{301}e", "nin\x{303}o") {
   printf "%s odwrócone w prosty sposób: %s\n", $word,
      scalar reverse $word;
   printf "%s odwrócone w lepszy sposób: %s\n", $word,
      join("", reverse $word =~ /\X/g);
}
```

Co w wyniku daje:

```
annèe odwrócone w prosty sposób: èenna
annèe odwrócone w lepszy sposób: eènna
niño odwrócone w prosty sposób: õnin
niño odwrócone w lepszy sposób: oñin
```

W ciągach „odwróconych w prosty sposób", oznaczenia diakrytyczne „przeskoczyły" z jednego znaku bazowego na drugi. Wynika to z tego, że znaki dołączone występują zawsze po znaku bazowym, a przecież odwróciliśmy kolejność znaków. Sposób na rozwiązanie tej sytuacji to pobieranie całych sekwencji znaku bazowego z wszystkimi znakami dołączonymi i wykonywanie odwrócenia dopiero w następnym kroku.

Zobacz również

Strony *perlre*(1) oraz *perluniintro*(1) podręcznika elektronicznego; rozdział 15. książki *Perl. Programowanie*; receptura 1.9.

1.9. Sprowadzanie łańcuchów zawierających znaki dołączone Unicode do postaci kanonicznej

Problem

Dwa łańcuchy po wyświetleniu wyglądają identycznie, ale przy sprawdzaniu nie są sobie równe, czasami nawet mają różną długość. W jaki sposób zapewnić, by Perl traktował je jako takie same łańcuchy?

Rozwiązanie

Zamiast bezpośredniego porównania odpowiadających sobie łańcuchów, z których przynajmniej niektóre zawierają sekwencje znaków połączonych Unicode, musimy wywołać funkcję `NFD()` (z modułu `Unicode::Normalize`), przekazując te łańcuchy jako parametry i porównać wyniki jej działania.

```
use Unicode::Normalize;
$s1 = "fa\x{E7}ade";
$s2 = "fac\x{0327}ade";
if (NFD($s1) eq NFD($s2)) {print "Równe!\n"}
```

Analiza

Czasami ta sama sekwencja znaków może być określona na różne sposoby. Czasami jest to skutek stosowania systemów kodowania zawierających litery ze znakami diakrytycznymi. Litery te można określić bezpośrednio, za pomocą jednego znaku (np. U+00E7, LATIN SMALL LETTER C WITH CEDILLA) lub pośrednio, za pomocą znaku podstawowego (np. U+0063, LATIN SMALL LETTER C) i następującego po nim znaku dołączonego (U+0327, COMBINING CEDILLA).

Inna sytuacja to na przykład stosowanie w tekście znaku podstawowego, po którym występują dwa lub więcej znaków dołączonych, z tym, że kolejność tych znaków może być różna w miejscach tekstu. Przypuśćmy, że używaliśmy litery „c", która miała zarówno cedillę, jak i karon na górze, tak by powstawała litera ç̌. Można ją określić na trzy różne sposoby:

```
$string = v231.780;
#    LATIN SMALL LETTER C WITH CEDILLA
#    COMBINING CARON

$string = v99.807.780;
#    LATIN SMALL LETTER C
#    COMBINING CARON
#    COMBINING CEDILLA

$string = v99.780.807;
#    LATIN SMALL LETTER C
#    COMBINING CEDILLA
#    COMBINING CARON
```

Funkcje normalizacyjne ustawiają je we właściwej kolejności. Dostępnych jest kilka takich funkcji, między innymi `NFD()` do dekompozycji kanonicznej i `NFC()` do kompozycji kanonicznej poprzedzonej dekompozycją kanoniczną. Bez względu na to, który z trzech powyższych sposobów zostanie zastosowany do określenia litery ç̌, funkcja `NFD()` zwróci `v99.807.780`, a `NFC()` `v231.780`.

Czasami lepiej jest zastosować funkcje `NFKD()` i `NFKC()`, które działają podobnie do poprzednich, z tą różnicą, że wykonują dekompozycję *kompatybilną*, po której, w przypadku funkcji `NFKC()`, następuje kanoniczna kompozycja. Na przykład punkt kodowy `\x{FB00}` oznacza podwójne f. Funkcje `NFD()` i `NFC()` dla takiego parametru zwracają to samo: `"\x{FB00}"`, ale `NFKD()` i `NFKC()` dają w wyniku łańcuchy dwuznakowe: `"\x{66}\x{66}"`.

Zobacz również

Punkt „Uniwersalne kodowanie znaków" na początku tego rozdziału; dokumentacja modułu `Unicode::Normalize`; receptura 8.20.

1.10. Traktowanie łańcuchów w Unicode jako oktetów

Problem

Dysponujemy łańcuchem zakodowanym w Unicode, ale chcielibyśmy, by Perl traktował go jako ciąg oktetów (np. w celu obliczenia jego długości lub przeprowadzenia operacji wejścia-wyjścia).

Rozwiązanie

Dyrektywa `use bytes` powoduje, iż wszystkie operacje Perla w obrębie jej zakresu leksykalnego traktują łańcuchy jako grupy oktetów. Należy jej użyć, gdy w kodzie programu następuje bezpośrednie wywołanie funkcji operujących na znakach:

```
$ff = "\x{FB00}";           # znaki 'ff'
$chars = length($ff);       # długość: jeden znak
{
   use bytes;               # wymuszenie semantyki bajtowej
   $octets = length($ff);   # długość: dwa oktety
}
$chars = length($ff);       # powrót do semantyki znakowej
```

Moduł `Encode` pozwala zmieniać łańcuchy Unicode na łańcuchy oktetów i odwrotnie. Należy go użyć, gdy kod operujący na znakach nie znajduje się w aktualnym zakresie leksykalnym:

```
use Encode qw(encode_utf8);
sub somefunc;                  # funkcja zdefiniowana gdzie indziej
$ff = "\x{FB00}";              # znaki 'ff'
$ff_oct = encode_utf8($ff);    # zamiana na oktety
$chars = somefunc($ff);        # działanie na łańcuchu znaków
$chars = somefunc($ff_oct);    # działanie na łańcuchu oktetów
```

Analiza

Jak wspomniano we wprowadzeniu do tego rozdziału, Perl rozróżnia dwa rodzaje łańcuchów: składające się z nieinterpretowanych oktetów oraz składające się ze znaków Unicode, których reprezentacja UTF-8 może wymagać więcej niż jednego oktetu. Z każdym łańcuchem związany jest znacznik, identyfikujący go jako łańcuch UTF-8 albo oktetów. Operacje wejścia-wyjścia oraz łańcuchowe (takie jak `length`) sprawdzają ten znacznik i odpowiednio stosują semantykę znakową lub oktetową.

Czasami występuje potrzeba działania na bajtach, nie na znakach. Przykładowo, w wielu protokołach stosowane są nagłówki `Content-Length`, które określają rozmiar treści komunikatu w oktetach. Do obliczenia tego rozmiaru nie można po prostu użyć funkcji `length`, ponieważ jeżeli łańcuch, dla którego wywoływana jest ta funkcja, oznaczony jest jako UTF-8, to zwrócony rozmiar będzie wyrażony w liczbie znaków.

Dyrektywa `use bytes` powoduje, iż wszystkie operacje Perla w obrębie jej zakresu leksykalnego wykorzystują odnośnie do łańcuchów semantykę oktetową, a nie znakową. Po zastosowaniu tej dyrektywy funkcja `length` zawsze zwraca liczbę oktetów, natomiast funkcja `read` zawsze informuje ile oktetów odczytano. Ponieważ jednak zasięg tej dyrektywy jest leksykalny, nie można jej użyć do zmiany zachowania kodu znajdującego się w innym zakresie (na przykład funkcji napisanej przez kogoś innego).

Aby to osiągnąć, należy utworzyć oktetową kopię łańcucha w formacie UTF-8. Oczywiście w pamięci używana jest ta sama sekwencja bajtów dla obu łańcuchów. Różnica polega na tym, że wykonana kopia łańcucha będzie miała wyzerowany znacznik UTF-8. Funkcje działające na kopii oktetowej będą zawsze stosowały semantykę oktetową, bez względu na to, w jakim zakresie się znajdują.

Istnieje także dyrektywa `no bytes`, która wymusza semantykę znakową, oraz funkcja `decode_utf8`, która zamienia łańcuchy oktetowe na łańcuchy UTF-8. Funkcje te są jednak mniej przydatne, ponieważ nie wszystkie łańcuchy oktetowe dają się zmienić na poprawne łańcuchy UTF-8, podczas gdy każdy łańcuch UTF-8 może być zamieniony na łańcuch oktetów.

Zobacz również

Dokumentacja dyrektywy `bytes`; dokumentacja standardowego modułu `Encode`.

1.11. Rozwijanie i kompresowanie tabulatorów

Problem

Chcielibyśmy zamienić znaki tabulacji występujące w łańcuchu na odpowiadającą im liczbę spacji i na odwrót. Zamianę spacji na tabulacje można wykorzystać do zmniejszenia rozmiaru pliku, który zawiera wiele spacji. Zamiana tabulacji na spacje może być wymagana przy generowaniu danych wyjściowych dla urządzeń, które nie rozpoznają tabulatorów lub rozwijają je inaczej niż w naszych zamierzeniach.

Rozwiązanie

Można skorzystać z dziwnie wyglądającego podstawienia:

```
while ($string =~ s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e) {
    # wykonywanie pustej pętli, aż do wykonania wszystkich podstawień
}
```

lub ze standardowego modułu `Text::Tabs`:

```
use Text::Tabs;
@expanded_lines = expand(@lines_with_tabs);
@tabulated_lines = unexpand(@lines_without_tabs);
```

Analiza

Zakładając, że tabulatory ustawione są na *N*-tej pozycji (gdzie *N* wynosi najczęściej osiem), łatwo jest zamienić tabulacje na spacje. Pierwsza z podanych metod nie używa modułu `Text::Tabs` i jest nieco trudna do zrozumienia. Poza tym wykorzystuje zmienną `$'`, co znacznie spowalnia dopasowanie wzorców w programie. Zostało to wyjaśnione w punkcie „Zmienne specjalne" we wprowadzeniu do rozdziału 6. Można wykorzystać ten algorytm do utworzenia filtra zamieniającego każdy tabulator w danych wejściowych na osiem pojedynczych spacji:

```
while (<>) {
   1 while s/\t+/' ' x (length($&) * 8 - length($`) % 8)/e;
   print;
}
```

Aby uniknąć stosowania zmiennej `$`, można użyć nieco bardziej skomplikowanej wersji podstawiania, wykorzystującej numerowane zmienne z jawnych przechwyceń; przedstawiony poniżej sposób zamienia każdy tabulator na cztery spacje (nie na osiem):

```
1 while s/^(.*?)(\t+)/$1 . ' ' x (length($2) * 4 - length($1) % 4)/e;
```

Jeszcze inne podejście to wykorzystanie bezpośrednich offsetów z tablic `@+` oraz `@-`. Poniższa linia także generuje pola czterech spacji:

```
1 while s/\t+/' ' x (($+[0] - $-[0]) * 4 - $-[0] % 4)/e;
```

W powyższych przykładach pętle `1 while` nie mogły być zapisane jako część operatora `s///g`, ponieważ po dokonaniu każdej zamiany trzeba od nowa obliczyć odległość od początku linii, a nie po prostu od miejsca, w którym ostatnio następowało podstawienie.

Konstrukcja `1 while WARUNEK` znaczy to samo co `while (WARUNEK) {}`, ale jest krótsza. Pochodzi ona jeszcze z czasów, w których Perl wykonywał ją zdecydowanie szybciej niż drugi wariant. Teraz druga konstrukcja wykonywana jest już niemal tak samo szybko jak pierwsza, ale używamy tej pierwszej dla wygody i z przyzwyczajenia.

Standardowy moduł `Text::Tabs` udostępnia funkcje do konwersji w obie strony, eksportując zmienną `$tabstop` do określania liczby spacji przypadających na jeden tabulator. Nie zmniejsza przy tym wydajności, ponieważ wykorzystuje zmienne `$1` i `$2`, a nie `$&` i `$`.

```
use Text::Tabs;
$tabstop = 4;
while (<>) { print expand($_) }
```

Można również użyć modułu `Text::Tabs` do „zwinięcia" tabulatorów. W poniższym przykładzie wykorzystywana jest domyślna wartość zmiennej `$tabstop` (8):

```
use Text::Tabs;
while (<>) { print unexpand($_) }
```

Zobacz również

Dokumentacja modułu `Text::Tabs`; opis operatora `s///` na stronach *perlre*(1) oraz *perlop*(1); opis zmiennych `@-` oraz `@+` (`@LAST_MATCH_START` oraz `@LAST_MATCH_END`) w rozdziale 28. książki *Perl. Programowanie*; podrozdział „Operatory dopasowywania wzorca" rozdziału 5. książki *Perl. Programowanie*.

1.12. Rozwijanie zmiennych we wprowadzanych łańcuchach

Problem

Odczytany został łańcuch z wbudowanym odwołaniem do zmiennej, na przykład:

```
Jesteś mi winny $kwota złotych
```

Trzeba zamienić ciąg `$kwota` na wartość odpowiadającej mu zmiennej.

Rozwiązanie

Jeżeli rozważane zmienne są globalne, należy użyć podstawienia z symbolicznym odwołaniem:

```
$text =~ s/\$(\w+)/${$1}/g;
```

Jeżeli jednak mogą być to zmienne leksykalne (`my`) należy użyć podwójnego `/ee`:

```
$text =~ s/(\$\w+)/$1/gee;
```

Analiza

Pierwszy sposób polega zasadniczo na wyszukaniu fragmentu przypominającego nazwę zmiennej i użyciu dereferencji symbolicznej do interpolowania jej zawartości. Jeżeli zmienna `$1` zawiera łańcuch `somevar`, to `${$1}` oznacza to wszystko, co zawiera zmienna `$somevar`. Metody tej nie da się jednak zastosować, jeżeli użyta była wcześniej dyrektywa `use strict 'refs'`, która zabrania dokonywania dereferencji symbolicznych.

Oto przykład:

```
our ($rows, $cols);
no strict 'refs';                                    # dla poniższego ${$1}/g
my $text;

($rows, $cols) = (24, 180);
$text = q(Moje dane: $rows lata i $cols cm wzrostu);    # tak jak apostrofy!
```

```
$text =~ s/\$(\w+)/${$1}/g;
print $text;
Moje dane: 24 lata i 180 cm wzrostu
```

Modyfikator operatora podstawienia `/e` służy do oznaczenia, iż część zastępująca wzorca ma być potraktowana jako kod, a nie łańcuch. Nadaje się on do takich miejsc, w których nie znamy dokładnej wartości zastępującej, ale wiemy jak ją obliczyć. Na przykład: podwojenie każdej liczby znajdującej się w łańcuchu:

```
$text = "Mam 17 lat"
$text =~ s/(\d+)/2 * $1/eg;
```

Perl, wykrywając modyfikator `/e` przy operatorze podstawiania, kompiluje kod programu znajdujący się w części zastępującej razem z resztą programu dużo wcześniej, niż ma miejsce rzeczywiste podstawienie. W trakcie jego wykonywania zmienna `$1` jest zamieniana na dopasowany łańcuch. Kod do wykonania będzie więc następujący:

```
2 * 17
```

Jeżeli napiszemy:

```
$text = 'Mam $AGE lat';      # uwaga - apostrofy
$text =~ s/(\$\w+)/$1/eg;    # ŹLE
```

to zakładając, że zmienna `$text` zawiera wzmiankę o zmiennej `$AGE`, Perl posłusznie zamieni `$1` na `$AGE`, a następnie wykona kod następującej treści:

```
'$AGE'
```

co w wyniku da znowu łańcuch pierwotny. Aby otrzymać wartość zmiennej, trzeba *ponownie* wykonać wynik wykonania kodu. W tym celu trzeba dodać jeszcze jedno `/e`:

```
$text =~ s/(\$\w+)/$1/eeg;    # wyszukuje zmienne my()
```

Można zastosować tyle modyfikatorów `/e`, ile potrzeba. Kompilacja i sprawdzanie poprawności składni przeprowadzane są tylko dla pierwszego z nich. Działa on więc jak konstrukcja `eval {BLOK}`, z tym, że nie przechwytuje wyjątków. Trzeba to postrzegać bardziej jako `do {BLOK}`.

Dodatkowe modyfikatory `/e` mają już zupełnie inny charakter. Są one bardziej podobne do konstrukcji `eval "ŁAŃCUCH"`. Do czasu uruchomienia programu nie są kompilowane. Niewielką zaletą tego podejścia jest fakt, iż w takim bloku nie jest wymagana dyrektywa `no strict 'refs'`. Natomiast jego ogromną zaletą jest to, że, w odróżnieniu od dereferencji symbolicznych, ten mechanizm wyszukuje zmienne leksykalne, utworzone za pomocą `my()`, czyli coś, czego dereferencje symboliczne nigdy nie będą w stanie zapewnić.

W kolejnym przykładzie wykorzystano modyfikator `/x`, aby dopuścić wstawianie wielokrotnych spacji oraz komentarzy we wzorcu stosowanym w podstawieniu oraz modyfikator `/e` do potraktowania części zastępującej jako kodu. Modyfikator `/e` daje większą kontrolę w przypadku wystąpienia błędów lub innych nadzwyczajnych okoliczności.

```
# rozwiń zmienne w $text, ale umieść w nim komunikat o błędzie
# jeżeli jakaś zmienna nie będzie zdefiniowana
$text =~ s{
   \$                              # znajdź znak dolara
   (\w+)                           # znajdź „wyraz" i zapisz go w zmiennej $1
}{
   no strict 'refs';               # dla poniższego $$1
   if (defined ${$1}) {
      ${$1};                       # rozwiń tylko zmienne globalne
   } else {
      "[BRAK ZMIENNEJ: \$$1";      # komunikat o błędzie
   }
}egx;
```

Dawno, dawno temu zapis `$$1` w łańcuchu oznaczał `${$}1`; to znaczy — zmienną `$$`, a po niej `1`. To przedawnione podejście ułatwiało rozwijanie zmiennej `$$` jako ID procesu przy tworzeniu nazw zmiennych tymczasowych. Obecnie `$$1` zawsze oznacza `${$1}`, czyli dereferencję zawartości zmiennej `$1`. Powyżej stosowaliśmy dłuższy zapis, aby zapewnić jasność intencji, nie ze względu na większą poprawność takiej notacji.

Zobacz również

Opis operatora `s//` na stronach podręcznika *perlre*(1) oraz *perlop*(1) i w rozdziale 5. książki *Perl. Programowanie*; opis funkcji `eval` na stronie *perlfunc*(1) i w rozdziale 29. książki *Perl. Programowanie*; podobne wykorzystanie podstawień w recepturze 20.9.

1.13. Zmiana wielkości liter

Problem

Trzeba zamienić wielkie litery na małe i odwrotnie.

Rozwiązanie

Należy skorzystać z funkcji `lc` oraz `uc` lub z sekwencji `\L` i `\U`.

```
$big = uc($little);    # "bo peep" -> "BO PEEP"
$little = lc($big);    # "JOHN"   -> "john"
$big = "\U$little";    # "bo peep" -> "BO PEEP"
$little = "\L$big";    # "JOHN"   -> "john"
```

Aby zmienić tylko jeden znak, należy użyć funkcji `lcfirst` oraz `ucfirst` lub sekwencji `\l` i `\u`.

```
$big = "\u$little";    # "bo"    -> "Bo"
$little = "\l$big";    # "BoPeep" -> "boPeep"
```

Analiza

Funkcje i sekwencje sterujące wyglądają inaczej, ale robią to samo. Można zmienić wielkość albo tylko pierwszej litery, albo całego łańcucha. Można nawet wykonać te dwie operacje łącznie, aby otrzymać wielką pierwszą literę i małe pozostałe.

```
$beast = "dromadery";
# różne części zmiennej $beast pisane wielkimi literami
$capit = ucfirst($beast);          # Dromadery
$capit = "\u\L$beast";             # (to samo)
$capall = uc($beast);              # DROMADERY
$capall = "\U$beast";              # (to samo)
$caprest = lcfirst(uc($beast));    # dROMADERY
$caprest = "\l\U$beast";           # (to samo)
```

Znaki ucieczki zmieniające wielkości liter są zazwyczaj używane do zapewnienia konsekwencji w stosowanej wielkości liter:

```
# zmiana pierwszych liter w wyrazach na wielkie, reszty na małe
$text = "otO jeST dłUGI wIErsZ";
$text =~ s/(\w+)/\u\L$1/g;
print $text;
Oto Jest Długi Wiersz
```

Można ich również użyć do nieuwzględniających wielkości liter porównań:

```
if (uc($a) eq uc($b)) {    # lub: "\U$a" eq "\U$b"
   print "a i b są takie same\n";
}
```

Pokazany w przykładzie 1.2 program *randcap* zmienia w sposób losowy 20 procent liter w podanym tekście z małych na wielkie.

Przykład 1.2. Program randcap

```
# !/usr/bin/perl –p
# randcap: filtr do losowej zamiany wielkości 20% liter
# wywołanie srand() jest niepotrzebne w wersji v5.4
BEGIN { srand(time() ^ ($$+($$<<15))) }
sub randcase { rand(100) < 20 ? "\u$_[0]" : "\l$_[0]" }
s/(\w)/randcase($1)/ge;

% randcap < genesis | head -9
boOk 01 genesis
001:001 in the Beginning goD created the heaven and tHe earTH.

001:002 and the earth wAS without ForM, aND void; AnD darkneSS was
        uppon The Face of the dEEp. And the spIrit of God moved upOn
        tHe face of the Waters.
001:003 and god Said, let there be ligHt: and therE wAs LigHt.
```

W tych językach, w których odróżnia się zapis wielkimi literami od zapisu rozpoczynanego od wielkiej litery, można wykorzystać funkcję `ucfirst()` (i odpowiadającą jej sekwencję `\u`) do zamiany pierwszej litery na wielką. Na przykład w języku węgierskim

występuje sekwencja znaków „dz". Zapisana wielkimi literami wygląda tak: „DZ", zapisana wielką literą — „Dz", natomiast małymi literami — „dz". W Unicode zdefiniowane są trzy różne znaki odpowiadające tym sytuacjom:

```
Punkt kodowy     Zapis     Znaczenie
01F1             DZ        LATIN CAPITAL LETTER DZ
01F2             Dz        LATIN CAPITAL LETTER D WITH SMALL LETTER Z
01F3             dz        LATIN SMALL LETTER DZ
```

Zmiana wielkości liter przy prostym użyciu operatora `tr[a-z][A-Z]` lub jemu podobnych jest wizją kuszącą, ale niezalecaną. Problem w tym przypadku polega na tym, że pomijane są wszystkie litery ze znakami diakrytycznymi, takimi jak dierezy, cedille czy oznaczenia akcentowania, wykorzystywane w wielu językach (także w angielskim). Poprawna obsługa zmiany wielkości liter ze znakami diakrytycznymi może być dość skomplikowana. Nie ma na to prostego rozwiązania — ale jeżeli wszystkie dane zapisane są w Unicode, to nie jest tak źle, ponieważ funkcje Perla zmieniające wielkości liter działają w takim przypadku bez problemów. Więcej informacji na ten temat znajduje się w punkcie „Uniwersalne kodowanie znaków" we wstępie do tego rozdziału.

Zobacz również

Opis funkcji `uc`, `lc`, `ucfirst`, `lcfirst` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*; objaśnienia znaków ucieczki `\L`, `\U`, `\l`, oraz `\u` w podrozdziale „Quote and Quote-like Operators" na stronie *perlop*(1) i w rozdziale 5. książki *Perl. Programowanie*.

1.14. Formatowanie tytułów i nagłówków[3]

Problem

Trzeba właściwie ustawić wielkość liter w łańcuchach reprezentujących nagłówki, tytuły książek, itp.

Rozwiązanie

Należy skorzystać z odpowiedniego wariantu przedstawionej poniżej funkcji `tc()`:

```
INIT {
   our %nocap;
   for (qw(
         a an the
         and but or
         as at but by for from in into of off on onto per to with
      ))
   {
      $nocap{$_}++;
   }
}
```

[3] Formaty zapisu tytułów i nagłówków rozważane w tej recepturze odnoszą się w głównej mierze do języka angielskiego — *przyp. tłum.*

```
sub tc {
   local $_ = shift;

   # zmień na małą literę, jeśli wyraz znajduje się na liście, w przeciwnym razie pierwszy znak wielką literą
   s/(\pL[\pL']*)/$nocap{$1} ls($1) : ucfirst(lc($1))/ge;

   s/^(\pL[\pL']*) /\u\L$1/x;     # ostatni wyraz wielkimi literami
   s/ (\pL[\pL']*)$/\u\L$1/x;     # pierwszy wyraz wielkimi literami

   # część w cudzysłowie traktowana jak tytuł
   s/\( (\pL[\pL']*) /(\u\L$1/x;
   s/(\pL[\pL']*) \) /\u\L$1)/x;

   # pierwsze słowo po średniku lub dwukropku wielką literą
   s/ ( [:;] \s+ ) (\pL[pL']*) /$1\u\L$2/x;

   return $_;
}
```

Analiza

W języku angielskim zasady poprawnego pisania nagłówków i tytułów są bardziej skomplikowane niż prosta zmiana pierwszej litery w każdym wyrazie na wielką. Taki efekt możemy uzyskać, stosując następujące podstawienie:

```
s/(\w+\S*\w*)/\u\L$1/g;
```

Większość poradników w tym zakresie podpowiada, iż pierwszy i ostatni wyraz powinno się zapisywać wielką literą, podobnie wszystkie pozostałe, które nie są przedimkami, partykułą „to" w konstrukcjach bezokolicznikowych, spójnikami parataktycznymi czy przyimkami.

Poniży program demonstruje formatowanie treści tytułów. Zakładamy, że funkcja `tc` jest zdefiniowana tak jak podano w rozwiązaniu.

```
# z przeprosinami dla Stephena Brusta, PFJ,
# i JRRT, jak zwykle.
@data = (
           "the enchantress of \x{01F3}ur mountain",
   "meeting the enchantress of \x{01F3}ur mountain",
   "the lord of the rings: the fellowship of the ring",
);
$mask = "%-30s: %s\n";
sub tc_lame {
   local $_ = shift;
   s/(\w+\S*\w*)/\u\L$1/g;
   return $_;
}
for $datum (@data) {
   printf $mask, "WSZYSTKO WIELKIMI",         uc($datum);
   printf $mask, "bez wielkich",              lc($datum);
   printf $mask, "jak w tytułach (prosto)",   tc_lame($datum);
   printf $mask, "jak w tytułach (lepiej)",   tc($datum);
   print "\n";
}
```

```
WSZYSTKO WIELKIMI          : THE ENCHANTRESS OF DZUR MOUNTAIN
bez wielkich               : the enchantress of dzur mountain
jak w tytułach (prosto)    : The Enchantress Of Dzur Mountain
jak w tytułach (lepiej)    : The Enchantress of Dzur Mountain

WSZYSTKO WIELKIMI          : MEETING THE ENCHANTRESS OF DZUR MOUNTAIN
bez wielkich               : meeting the enchantress of dzur mountain
jak w tytułach (prosto)    : Meeting The Enchantress Of Dzur Mountain
jak w tytułach (lepiej)    : Meeting the Enchantress of Dzur Mountain

WSZYSTKO WIELKIMI          : THE LORD OF THE RINGS: THE FELLOWSHIP OF THE RING
bez wielkich               : the lord of the rings: the fellowship of the ring
jak w tytułach (prosto)    : The Lord Of The Rings: The Fellowship Of The Ring
jak w tytułach (lepiej)    : The Lord of the Rings: The Fellowship of the Ring
```

Niektóre podręczniki zalecają zmianę wielkości tylko tych przyimków, które mają więcej niż trzy, cztery znaki czy czasami nawet pięć znaków. Przykładowo, w wydawnictwie O'Reilly przyjęło się, że przyimki czteroliterowe i mniejsze zapisywane są tylko małymi literami. Oto dłuższa lista przyimków, którą można modyfikować w zależności od potrzeb:

```
@all_prepositions = qw{
   about above absent across after against along amid amidst
   among amongest around as at athwart before behind below
   beneath beside besides between betwixt beyond but by circa
   down during ere except for from in into near of off on onto
   out over past per since than through till to toward towards
   undeer until unto up upon versus via with within without
};
```

Na tym kończą się możliwości tej metody, ponieważ nie zastosowano w niej mechanizmu rozróżniającego poszczególne części mowy. Niektóre z podanych wyżej przyimków mogą być w pewnych sytuacjach zaliczone do słów, które powinny być zapisane od wielkiej litery, gdyż pełnią w nich funkcję spójników hipotaktycznych, przysłówków czy nawet przymiotników. Oto przykłady takich zdań: „Down by the Riverside", ale już „Getting By on Just $30 a Day", czy „A Ringing in My Ears", ale „Bringing In the Sheaves".

Może się także zdarzyć, że funkcji `ucfirst` lub sekwencji `\u` będziemy chcieli użyć tylko do zamiany pierwszej litery na wielką, bez jednoczesnej zamiany pozostałych na małe. W ten sposób wyrazy pisane wielkimi literami, np. skrótowce, nie tracą swojej cechy. Prawdopodobnie niezbyt przydatna byłaby zamiana „FBI" czy „CIA" na „Fbi" i „Cia".

Zobacz również

Opis funkcji `uc`, `lc`, `ucfirst`, `lcfirst` na stronie podręcznika *perlfunc*(1) i w rozdziale 29. książki *Perl. Programowanie*; znaki ucieczki `\L`, `\U`, `\l` oraz `\u` w podrozdziale „Quote and Quote-like Operators" na stronie *perlop*(1) oraz w rozdziale 5. książki *Perl. Programowanie*.

1.15. Interpolacja funkcji i wyrażeń w łańcuchach

Problem

Chcielibyśmy, aby wywołanie funkcji lub wyrażenie zostało rozwinięte w łańcuchu. Pozwoli to tworzyć bardziej złożone szablony niż prosta interpolacja zmiennych skalarnych.

Rozwiązanie

Należy podzielić wyrażenie na kilka oddzielnych części poddanych konkatenacji:

```
$answer = $var1 . func() . $var2;     # tylko skalary
```

lub skorzystać z sekwencji `@{[WYRAZENIE LISTOWE]}` lub `${ \(WYRAZENIE SKALARNE)}`:

```
$answer = "LANCUCH @{[ WYRAZENIE LISTOWE ]} RESZTA LANCUCHA";
$answer = "LANCUCH ${\( WYRAZENIE SKALARNE )} RESZTA LANCUCHA";
```

Analiza

Poniższe linie kodu demonstrują wykorzystanie obu rozwiązań. W pierwszym wykorzystano konkatenację, w drugim — rozwijaną sekwencję:

```
$phrase = "Mam " . ($n + 1) . " jabłek.";
$phrase = "Mam ${\($n + 1)} jabłek.";
```

Pierwsza technika polega na tym, że tworzymy dłuższy łańcuch przez połączenie krótszych fragmentów, unikając interpolacji, ale uzyskując taki sam efekt. Ponieważ funkcja `print` dokonuje konkatenacji swoich argumentów, jeżeli chodzi nam tylko o wyświetlenie zmiennej `$phrase`, wystarczyłoby napisać:

```
print "Mam ", $n + 1, " jabłek.\n";
```

Jeżeli niezbędne jest przeprowadzanie interpolowania, skorzystać trzeba z dość złożonej metody zaproponowanej w rozwiązaniu. W cudzysłowach specjalne znaczenie mają jedynie znaki `@`, `$` oraz `\` (podobnie jak w przypadku operatorów `m//` i `s///`, także przy konstrukcji `qx()` nie jest wykonywane rozwijanie cudzysłowowe, jeżeli w roli ograniczników zastosowane zostaną apostrofy! Wyrażenie `$home = qx'katalog domowy to $HOME';` podstawi zmienną środowiskową `$HOME`, a nie zmienną Perla). Dlatego jedynym sposobem na uzyskanie rozwijania dowolnych wyrażeń jest użycie bloków `${}` lub `@{}` zawierających odwołania.

W tym przykładzie:

```
$phase = "Mam ${\( count_em() )} jabłek.";
```

wywołanie funkcji objęte nawiasami nie jest przeprowadzane w kontekście skalarnym; obowiązuje nadal kontekst listowy. Aby to zmienić, napiszemy:

```
$phase = "Mam ${\( scalar count_em() )} jabłek.";
```

Wynik interpolacji nie musi być tylko i wyłącznie przypisywany do zmiennej. Jest to ogólny mechanizm, który można zastosować do dowolnego łańcucha w cudzysłowach. Przykładowo, poniżej tworzony jest łańcuch z interpolowanym wyrażeniem, który po rozwinięciu przekazywany jest jako argument do funkcji:

```
some_func("Wszystko czego oczekujesz to @{[ split /:/, $rec]} elementów danych");
```

Interpolowanie można też przeprowadzać w dokumentach w miejscu, na przykład:

```
die "Nie udało się wysłać listu" unless send_mail(<<"EOTEXT", $target);
To:$naughty
From: Bank Najlepszy
Cc: @{ get_manager_list($naughty) }
Date: @{[ do {my $now = `date`; chomp $now; $now}]}(dzisiaj)

Droga Pani/Panie $naughty,

Dzisiejszego dnia limit został przekroczony o @{[ 500 + int rand(100)]} PLN.
W związku z powyższym zablokowaliśmy Pani/Pańskie konto.

Z poważaniem,
Zarząd
EOTEXT
```

Rozwijanie odwróconych apostrofów (``) jest szczególnym wyzwaniem, gdyż normalnie zakończyłoby się wprowadzeniem niepotrzebnych znaków przejścia do nowej linii. Poprzez utworzenie wewnątrz dereferencji anonimowej tablicy (`@{[]}`) dodatkowego bloku, tak jak w ostatnim przykładzie, możliwe staje się ustanawianie zmiennych prywatnych.

Powyższe techniki dobrze nadają się do rozwiązania postawionego problemu, ale proste rozłożenie wykonywanych operacji na kilka kroków i wykorzystanie zmiennych tymczasowych jest prawie zawsze bardziej zrozumiałe dla czytającego kod programu.

Moduł `Interpolation` z sieci CPAN oferuje rozwiązanie syntaktycznie przyjemniejsze. Przykładowo, aby elementy tablicy asocjacyjnej `%E` były obliczane i zwracały swoje indeksy, zapiszemy:

```
use Interpolation E   => 'eval';
print "Zrezygnowała Pani z konta numer $E{500 + int rand(100)}\n";
```

Podobnie, aby tablica asocjacyjna `%money` wywoływała dowolnie wybraną funkcję:

```
use Interpolation money => \&currency_commify;
print "To daje na dzień dzisiejszy kwotę $money{ 4 * $payment }.\n";
```

W wyniku otrzymamy na przykład:

```
To daje na dzień dzisiejszy kwotę $3,232.421.04.
```

Zobacz również

Strona *perlref*(1) oraz podrozdział „Korzystanie z odwołań stałych" rozdziału 8. książki *Perl. Programowanie*; moduł `Interpolation` z sieci CPAN.

1.16. Tworzenie wcięć w dokumentach w miejscu

Problem

Podczas używania wielowierszowego mechanizmu zapisywania danych zwanego *dokumentem w miejscu*, umieszczany tekst musi być zrównany z marginesem, co w kodzie programu wygląda nienaturalnie. Chcielibyśmy zastosować wcięcia w tekście dokumentu w miejscu, ale nie powinny się one pojawiać w łańcuchu wynikowym.

Rozwiązanie

Należy zastosować operator `s///`, aby pozbyć się odstępów z początków linii.

```
# wykonanie operacji w jednym kroku
($var = << "HERE_TARGET") =~ s/^\s+//gm;
   tutaj należy
   wstawić tekst
HERE_TARGET

# wykonanie operacji w dwóch krokach
$var = << "HERE_TARGET";
   tutaj należy
   wstawić tekst
HERE_TARGET
$var =~ s/^\s+//gm;
```

Analiza

Wykorzystane podstawienie jest bardzo proste. Usuwa początkowe spacje z tekstu dokumentu w miejscu. Modyfikator `/m` powoduje, że znak `^` oznacza dopasowywanie na początku każdej linii w łańcuchu, a modyfikator `/g` wymusza powtarzanie dopasowywania tak długo, jak długo to możliwe (w tym przypadku, dla wszystkich linii dokumentu w miejscu).

```
($definition = << 'FINIS') =~ s/^\s+//gm;
   The five varietes of camelids
   are the familiar camel, his friends
   the lamma and the alpaca, and the
   rather less well-known guanaco
   and vicuña.
FINIS
```

Uwaga: wszystkie wzorce w tej recepturze używają klasy `\s`, oznaczającej jeden znak odstępu, co powoduje, że dopasowywane będą także znaki przejścia do nowej linii. Spowoduje to usunięcie wszystkich pustych linii z dokumentu w miejscu. Jeżeli takie zachowanie nie jest pożądane, trzeba we wzorcu zastąpić `\s` ciągiem `[^\S\n]`.

Zastosowane podstawianie wykorzystuje fakt, że wynik przypisania może zostać użyty jako lewostronny argument operatora =~. Dzięki temu wszystko można zapisać w jednym wierszu, jednak działa to tylko przy przypisywaniu do zmiennej. Przy bezpośrednim wykorzystaniu dokumentu w miejscu, byłby on traktowany jako wartość stała, której nie można modyfikować. Treść dokumentu w miejscu można zmieniać *tylko* po uprzednim wpisaniu jej do zmiennej.

Istnieje jednak prosty sposób rozwiązania tego ograniczenia, szczególnie gdy operację taką zamierzamy przeprowadzać wielokrotnie w programie. Wystarczy napisać procedurę:

```
sub fix {
   my $string = shift;
   $string =~s/^\s+//gm;
   return $string;
}
print fix( << "END");
   Tutaj odpowiedni tekst
END
# w przypadku wcześniej zadeklarowanej funkcji można pominąć nawiasy
print fix << "END";
   Tutaj odpowiedni tekst
END
```

Podobnie jak we wszystkich innych dokumentach w miejscu, znacznik ograniczający wstawiany tekst (w tym przypadku `END`) trzeba zapisywać tuż przy lewym marginesie. Aby go też zapisać z odpowiednim wcięciem, przed liniami tekstu trzeba dodać tyle samo znaków spacji, ile zastosowano przed znacznikiem.

```
($quote = << '    FINIS') =~s/^\s+//gm;
      ...we will have peace, when you and all your works have
      perished-- and the works of your dark master to whom you would
      deliver us. You are a liar, Saruman, and a corrupter of men's
      hearts.  --Theoden in /usr/src/perl/taint.c
    FINIS
$quote =~ s/\s+--/\n--/;     # przenieś przypisanie do oddzielnej linii
```

Jeżeli opisywane działania mają być wykonane na łańcuchu zawierającym kod programu, który będzie użyty w funkcji `eval`, lub zwykły tekst do wyświetlenia, to może się zdarzyć, że niewskazane byłoby „ślepe" usuwanie wszystkich wcięć, ponieważ niszczyłoby to układ tekstu. Funkcji `eval` jest to pewnie obojętne, ale nie użytkownikowi.

Innym udogodnieniem jest zastosowanie dodatkowych znaków na początku linii zawierających kod, umożliwiających odpowiednie ułożenie tekstu. W poniższym przykładzie każdą taką linię poprzedzamy znakami `@@@` i odpowiednio wykonujemy zaplanowane wcięcia:

```
if ($REMEMBER_THE_MAIN) {
   $perl_main_C = dequote << '   MAIN_INTERPRETER_LOOP';
      @@@ int
      @@@ runops() {
      @@@    SAVEI32(runlevel);
      @@@    runlevel++;
      @@@    while ( op = (*op->op_ppaddr)() );
      @@@    TAINT_NOT;
```

```
      @@@    return 0;
      @@@ }
     MAIN_INTERPRETER_LOOP
   # można dodać więcej kodu
}
```

Usuwanie wcięć jest też niewskazane w przypadku wierszy:

```
Sub dequote;
$poem = dequote << "EVER_ON_AND_ON";
   Now far ahead the Road has gone,
      And I must follow, if I can
   Pursing it with eager feet,
      Until it joins some larger way
   Where many paths and errands meet.
      And whither then? I cannot say.
         --Bilbo in /usr/scr/perl/pp_ctl.c
EVER_ON_AND_ON
print "Oto wiersz:\n\n$poem\n";
```

Oto wyświetlony tekst:

```
Oto wiersz:

Now far ahead the Road has gone,
   And I must follow, if I can
Pursing it with eager feet,
   Until it joins some larger way
Where many paths and errands meet.
   And whither then? I cannot say.
      --Bilbo in /usr/scr/perl/pp_ctl.c
```

Wszystkie te przypadki poprawnie obsługuje podana niżej funkcja `dequote`. Jako argument należy przekazać jej dokument w miejscu. Funkcja ta sprawdza czy każda linia zaczyna się tymi samymi znakami i jeśli tak, usuwa te znaki. W przeciwnym razie odczytuje liczbę spacji występujących przed pierwszą linią i właśnie tyle usuwa z początku wszystkich pozostałych.

```
sub dequote {
   local $_ = shift;
   my ($white, $leader);    # wspólne spacje i wspólny ciąg początkowy
   if (/^\s*(?:([^\w\s]+)(\s*).*\n)(?:\s*\1\2?.*\n)+$/) {
      ($white, $leader) = ($2, quotemeta($1));
   } else {
      ($white, $leader) = (/^(\s+)/, '');
   }
   s/^\s*?$leader(?:$white)?//gm;
   return $_;
}
```

Dodanie modyfikatora `/x` do wzorców umożliwi zapisanie ich w wielu liniach z komentarzami:

```
if (m{
      ^                 # początek linii
      \s *              # 0 lub więcej znaków spacji
      (?:               # początek pierwszej nieprzechwytującej grupy
```

```
            (               #   początek bufora $1
               [^\w\s]      #       jeden znak nie będący ani spacją, ani niedopuszczalnym
                            #       w wyrazach
               +            #       jeden lub więcej takich
            )               #   koniec bufora $1
            ( \s* )         #   wstaw 0 lub więcej spacji do bufora $2
            .* \n           #   dopasowanie do końca pierwszej linii
         )                  # koniec pierwszej grupy
         (?:                # początek drugiej grupy nieprzechwytującej
            \s *            #       0 lub więcej znaków spacji
            \1              #       to wszystko co zawiera bufora $1
            \2 ?            #       to co będzie w $2, ale opcjonalnie
            .* \n           #       dopasowanie do końca linii
         ) +                # co najmniej jednokrotne wystąpienie tej grupy
           $                # aż do końca wiersza
       }x
    )
{
    ($white, $leader) = ($2, quotemeta($1));
} else {
    ($white, $leader) = (/^(\s+)/, '');
}
s{
    ^                   # początek każdej linii (z powodu operatora /m)
    \s *                # dowolna liczba początkowych spacji
       ?                #   z minimalnym dopasowaniem
    $leader             # umieszczony w cudzysłowie, przechwycony ciąg początkowy
    (?:                 # początek grupy nieprzechwytującej
       $white           #   ta sama liczba
    ) ?                 # opcja na wypadek wystąpienia znaku EOL po ciągu początkowym
}{}xgm;
```

Prawdopodobnie nawet taki zapis niewiele pomaga. Czasami po prostu nic nie daje doprawienie kodu nudnymi komentarzami opisującymi kod. Prawdopodobnie to jest jeden z takich przypadków.

Zobacz również

Sekcja „Scalar Value Constructors" strony *perldata*(1) oraz podrozdział „Wartości skalarne" rozdziału 2. książki *Perl. Programowanie*; opis operatora `s///` na stronach *perlre*(1) oraz *perlop*(1), a także podrozdział „Operatory dopasowywania wzorca" w rozdziale 5. książki *ProgrammingPerl*.

1.17. Zmiana formatu akapitów

Problem

Stosowany łańcuch jest zbyt długi i nie mieści się na ekranie. Chcielibyśmy podzielić go na kilka linii bez dzielenia wyrazów. Załóżmy przykładowo, że skrypt korygujący styl odczytuje poszczególne akapity z pliku tekstowego i zastępuje błędne frazy poprawnymi.

Zamiana wyrażenia *wykorzystanie inherentnej funkcjonalności* na *użycie* spowoduje zmianę długości linii, dlatego konieczne będzie przeformatowanie akapitu przed wysłaniem na wyjście.

Rozwiązanie

W celu prawidłowego podziału linii należy użyć standardowego modułu `Text::Wrap`:

```
use Text::Wrap
@output = wrap($leadtab, $nexttab, @para);
```

lub wykorzystać bardziej „wnikliwy" moduł `Text::Autoformat` z sieci CPAN:

```
use Text::Autoformat;
$formatted = autoformat $rawtext;
```

Analiza

Moduł `Text::Wrap` udostępnia, zastosowaną w przykładzie 1.3, funkcję `wrap`, która pobiera listę linii i formatuje je do postaci akapitu, w którym długość linii nie przekracza `$Text::Wrap::columns` znaków. W przykładzie zmiennej `$columns` przypisano wartość 20, co gwarantuje, że żadna z linii nie będzie miała więcej niż 20 znaków. Funkcji `wrap` trzeba przekazać dodatkowe dwa argumenty przed listą linii: pierwszy z nich to wcięcie dla pierwszego wiersza akapitu; drugi określa wcięcie pozostałych wierszy.

Przykład 1.3. Program wrapdemo

```
#!/usr/bin/perl –w
# wrapdemo — prezentuje działanie modułu Text::Wrap
@input = ("Edytor składa i łączy tekst, ",
          "nie jest tylko kawałkiem krzemu",
          "i",
          "wolnych elektronów!");
use Text::Wrap qw($columns &wrap);
$columns = 20;
print "0123456789" x 2, "\n";
print wrap("    ", "  ", @input), "\n";
```

Rezultatem działania tego programu jest:

```
01234567890123456789
    Edytor składa i
  łączy tekst, nie
  jest tylko
  kawałkiem krzemu
  i wolnych
  elektronów!
```

Aby otrzymać z powrotem jeden łańcuch, ze znakami nowej linii na końcu każdego wiersza z wyjątkiem ostatniego:

```
# połączenie kilku wierszy w jeden, po czym odpowiednio go zawijamy
use  Text::Wrap;
undef $/;
print wrap('', '', split(/\s*\n\s*/, <>));
```

Dysponując modułem `Term::ReadKey` (dostępny w sieci CPAN), możemy określić rozmiar okna i dzięki temu podzielić wiersze tak, by pasowały do szerokości ekranu. Jeżeli ten moduł nie jest dostępny, rozmiar ekranu możemy w niektórych systemach odczytać ze zmiennej środowiskowej `$ENV{COLUMNS}` lub analizując wynik polecenia *stty*(1).

Kolejny program stara się dopasować do akapitu zarówno krótkie, jak i długie linie, podobnie jak program *fmt*(1), ustawiając separator rekordu wejściowego `$/` na pusty łańcuch (powodując w ten sposób, że operator <> będzie odczytywał całe akapity) oraz separator rekordu wyjściowego `$\` na dwa znaki przejścia do nowej linii. Akapit jest następnie zamieniany na jeden długi łańcuch poprzez zastąpienie wszystkich znaków nowej linii i przyległych do nich odstępów pojedynczymi znakami spacji. Ostatecznie wywoływana jest funkcja `wrap` z wcięciami dla pierwszego i kolejnych wierszy ustawionymi na puste łańcuchy, dzięki czemu powstają akapity blokowe.

```
use Text::Wrap          qw(&wrap $columns);
use Term::ReadKey       qw(GetTerminalSize);
($columns) = GetTerminalSize();
($/, $\) = ('', "\n\n");       # odczytywanie akapitami, 2 nowe linie na wyjściu
while (<>) {                   # pobranie całego akapitu
   s/\s*\n\s*/ /g;             # zamiana znaków nowej linii na spacje
    print wrap('', '', $_);    # i formatowanie
}
```

Moduł `Text::Autoformat` z sieci CPAN jest dużo inteligentniejszy. Po pierwsze, stara się unikać tworzenia „wdów", czyli bardzo krótkich linii na końcu. Co bardziej istotne: poprawnie obsługuje akapity zawierające wielokrotne, głęboko zagnieżdżone cytaty. Przykład zamieszczony w dokumentacji tego modułu pokazuje, że przy użyciu tylko prostego wywołania `print autoformat($badparagraph)` możliwe jest przekonwertowanie tekstu w następującej postaci:

```
In comp.lang.perl.misc you wrote:
: > <CN = Clooless Noobie> writes:
: > CN> PERL sux because:
: > CN>    * It doesn't have a switch statement and you have to put $
: > CN>signs in front of everything
: > CN>    * There are too many OR operators: having |, || and 'or'
: > CN>operators is confusing
: > CN>    * VB rools, yeah!!!!!!!!!
: > CN> So anyway, how can I stop reloads on a web page?
: > CN> Email replies only, thanks - I don't read this newsgroup.
: >
: > Begone, sirrah! You are a pathetic, Bill-loving, microcephalic
: > script-infant.
: Sheesh, what's with this group - ask a question, get toasted! And how
: *dare* you accuse me of Ianuphilia!
```

na:

```
In comp.lang.perl.misc you wrote:
: > <CN = Clooless Noobie> writes:
: > CN> PERL sux because:
: > CN>     * It doesn't have a switch statement and you
: > CN>       have to put $ signs in front of everything
: > CN>     * There are too many OR operators: having |, ||
: > CN>       and 'or' operators is confusing
: > CN>     * VB rools, yeah!!!!!!!!! So anyway, how can I
: > CN>       stop reloads on a web page? Email replies
: > CN>       only, thanks - I don't read this newsgroup.
: >
: > Begone, sirrah! Tou are a pathetic, Bill-loving,
: > microcephatic script-infant.
: Sheesh, what's with this group - ask a question, get toasted!
: And how *dare* you accuse me of Ianuphilia!
```

Oto miniprogram wykorzystujący ten moduł do przeformatowania każdego akapitu ze strumienia wejściowego:

```
use Text::Autoformat;
$/ = '';
while (<>) {
   print autoformat($_, {squeeze => 0, all => 1}), "\n";
}
```

Zobacz również

Opis funkcji `split` oraz `join` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*; dokumentacja standardowego modułu `Text::Wrap`; moduł `Term::ReadKey` w sieci CPAN wraz z wykorzystaniem tego modułu w recepturze 15.6, moduł `Text::Autoformat` z CPAN.

1.18. Wyświetlanie znaków ucieczki

Problem

Trzeba utworzyć łańcuch, w którym przed niektórymi znakami (cudzysłowami, przecinkami, itp.) występują znaki ucieczki. Przykładowo, podczas tworzenia łańcucha formatującego dla funkcji `sprintf` należy przekonwertować literalne znaki `%` na `%%`.

Rozwiązanie

Należy wykorzystać lewy ukośnik lub podwoić każdy znak ucieczki:

```
# lewy ukośnik
$var =~ s/([LISTAZNAKOW])/\\$1/g;

# podwojenie
$var =~ s/([LISTAZNAKOW])/$1$1/g;
```

Analiza

`$var` jest zmienną, która ma być zmodyfikowana. `LISTAZNAKOW` jest listą znaków, które mają być poprzedzone znakami ucieczki. Jeżeli formatowany w ten sposób ma być tylko jeden znak, można pominąć nawiasy we wzorcu:

```
$string =~ s/%/%%/g;
```

Poniższy kod pozwala na odpowiednie przygotowanie łańcucha przed wysłaniem do powłoki (w praktyce znaki ucieczki trzeba stawiać częściej niż tylko przed ' i ", by móc bezpiecznie przekazywać dowolne łańcuchy do powłoki. Uzyskanie wszystkich właściwych znaków jest na tyle trudne, a zasięg potencjalnych szkód na tyle rozległy, że do uruchamiania programów zewnętrznych lepiej wykorzystywać listową formę poleceń `system` i `exec`, pokazaną w recepturze 16.2. Oba polecenia omijają powłokę).

```
$string = q(Mama powiedziała, "Nie rób tego.");
$string =~ s/(['"])/\\$1/g;
```

Ponieważ część zastępująca w podstawieniu jest interpretowana jak łańcuch w cudzysłowach, aby otrzymać jeden lewy ukośnik, konieczne było napisanie dwóch. A oto podobny przykład dla VMS DCL, w którym należy podwoić każdy cudzysłów, aby wyświetlony został jeden:

```
$string = q(Mam powiedziała, "Nie rób tego.");
$string =~ s/(['"])/$1$1/g;
```

Gorzej współpracuje się z interpreterami poleceń firmy Microsoft. *COMMAND.COM* z systemu Windows dopuszcza stosowanie cudzysłowów, ale nie apostrofów, odwrotne apostrofy nie służą do wykonywania poleceń, a aby zamienić cudzysłów na literał, trzeba poprzedzić go lewym ukośnikiem. Dowolna z wielu darmowych i komercyjnych powłok uniksopodobnych dostępnych dla Windows, powinna jednak działać bez większych problemów.

Ponieważ w wyrażeniu regularnym wykorzystujemy klasy znaków, możemy zastosować – do określenia zakresu oraz ^ na początku do zanegowania jego znaczenia. Poniższa linia spowoduje poprzedzenie znakiem ucieczki każdego znaku nie należącego do przedziału od A do Z.

```
$string =~ s/([^A-Z])/\\$1/g;
```

W praktyce raczej nie zajdzie taka potrzeba, gdyż spowodowałoby to, na przykład, zamianę ciągu `"a"` na `"\a"`, co w kodzie ASCII oznacza znak specjalny: BEL (zazwyczaj do oznaczenia znaków spoza alfabetu lepiej jest zastosować operator `\PL`).

Aby znakami ucieczki poprzedzić wszystkie znaki nie będące częścią słów, należy zastosować metaznaki `\Q` oraz `\E` lub skorzystać z funkcji `quometa`. Poniższe instrukcje są równoważne:

```
$string = "to \Qjest test!\E";
$string = "to jest\\ test\\!";
$string = "to " . quotemeta("jest test!");
```

Zobacz również

Opis operatora `s///` na stronie *perlre*(1) oraz *perlop*(1), rozdział 5. książki *Perl. Programowanie*; opis funkcji `quotemeta` na stronie *perlfunc*(1) oraz w rozdziale 29. książki *Perl. Programowanie*; receptura 19.1 omawiająca sposoby wyświetlania znaków HTML; receptura 19.5 omawiająca przesyłanie znaków ucieczki do powłoki.

1.19. Usuwanie odstępów z końca łańcucha

Problem

Chcielibyśmy usunąć ewentualne znaki spacji z początku lub końca odczytanej linii.

Rozwiązanie

Aby usunąć spacje, należy zastosować parę wzorców:

```
$string =~ s/^\s+//;
$string =~s/\s+$//;
```

lub napisać funkcję zwracającą nową wartość:

```
$string = trim($string);
@many = trim(@many);

sub trim {
   my @out = @_;
   for (@out) {
      s/^\s+//;        # usunięcie z początku
      s/\+$//;         # usunięcie z końca
   }
   return @out == 1
         ? $out[0]     # zwrócenie tylko jednej wartości
         : @out;       # lub więcej
}
```

Analiza

Istnieje wiele rozwiązań tego problemu, ale w zwykłym przypadku najbardziej wydajne jest podane powyżej. Funkcja ta zwraca nową wersję łańcucha przekazanego na wejście, z usuniętymi spacjami z początku i końca wiersza. Działa zarówno na pojedynczych łańcuchach, jak i na listach.

W celu usunięcia ostatniego znaku z łańcucha należy użyć funkcji `chop`. Nie należy jej mylić z podobną do niej funkcją `chomp`, która usuwa ostatnią część łańcucha tylko wtedy, gdy jest zgodna z zawartością zmiennej `$/`, domyślnie `"\n"`. Funkcje te są często używane na wejściu do usunięcia znaku przejścia do nowej linii:

```
# wyświetlenie wpisanych znaków ujętych w symbole > <
while (<STDIN>) {
   chomp;
   print ">$_<\n";
}
```

Funkcję tę można rozbudować na wiele sposobów.

Po pierwsze, co należy zrobić, jeżeli przekazanych zostanie kilka łańcuchów, ale kontekst zwrotny wskazuje, że wynik ma być pojedynczym skalarem? Funkcja pokazana w rozwiązaniu zachowuje się dość dziwnie w takiej sytuacji: zwraca wartość skalarną przedstawiającą liczbę przekazanych łańcuchów. Nie jest to zbyt przydatne. *Można* by w takiej sytuacji wypisywać komunikat ostrzegawczy lub generować wyjątek. W innej opcji można by łączyć ze sobą listę wartości zwracanych.

Dla łańcuchów zawierających dodatkowe spacje nie tylko na końcach, funkcja ta mogłaby zamieniać wszystkie ciągi znaków spacji wewnątrz łańcucha na spacje pojedyncze — uzyskamy to, dodając następującą linię na końcu pętli:

```
s/\s+/ /g;      # ostatecznie zredukuj środkowe
```

Tym sposobem łańcuch `" ale\t\tnie tutaj\n"` zostałby zamieniony na `"ale nie tutaj"`. Wydajniejszą alternatywą dla trzech podstawień:

```
s/^\s+//;
s/\s+$//;
s/\s+/ /g;
```

byłoby następujące wyrażenie:

```
$_ = join(' ', split(' '));
```

Jeżeli do funkcji nie byłyby przekazane żadne argumenty, to mogłaby zachowywać się tak jak `chop` lub `chomp`, domyślnie działając na zmiennej `$_`. Połączenie wszystkich powyższych udoskonaleń prowadzi do funkcji o następującej postaci:

```
# 1. usunięcie początkowych i końcowych spacji
# 2. zamiana wielu spacji między wyrazami na pojedyncze spacje
# 3. przy braku argumentów pobranie wejścia ze zmiennej $_
# 4. jeżeli wynik działania funkcji jest w kontekście skalarnym - połączenie listy wynikowej
#    w pojedynczy skalar z poszczególnymi wartościami oddzielonymi spacjami

sub trim {
   my @out = @_ ? @_ : $_;
   $_ = join(' ', split(' ')) for @out;
   return wantarray ? @out : "@out";
}
```

Zobacz również

Opis operatora `s///` na stronie *perlre*(1) oraz *perlop*(1), rozdział 5. książki *Perl. Programowanie*; opis funkcji `chomp` oraz `chop` na stronie *perlfunc*(1) oraz rozdział 29. książki *Perl. Programowanie*; usuwanie początkowych i końcowych spacji za pomocą funkcji `getnum` w recepturze 2.1.

1.20. Analizowanie danych oddzielonych przecinkami

Problem

Chcemy odczytać plik z danymi, w którym poszczególne wartości oddzielone są przecinkami, z tym, że w odczytywanych polach danych mogą pojawić się przecinki umieszczone w cudzysłowach lub znaki cudzysłowów poprzedzone znakami ucieczki. Większość arkuszy kalkulacyjnych oraz programów bazodanowych stosuje taki format jako standard wymiany danych.

Rozwiązanie

Jeżeli w odczytywanym pliku zastosowano uniksowe konwencje cytowania i umieszczania znaków specjalnych w łańcuchach (zgodnie z którymi pojawiające się w polu danych cudzysłowy i znaki specjalne poprzedzane są lewym ukośnikiem `"jak w \"tym\" przykładzie"`), należy użyć modułu `Text::ParseWords` oraz takiej prostej funkcji:

```
use Text::ParseWords;
sub parse_csv0 {
   return quotewords("," => 0, $_[0]);
}
```

Jednak jeżeli cudzysłowy w polach danych są podwajane `"w ""ten"" sposób"`, można zastosować poniższą procedurę, zaczerpniętą z drugiego wydania książki *Wyrażenia regularne*[4]:

```
sub parse_csv1 {
   my $text = shift;     # rekord zawierający wartości oddzielone przecinkami
   my @fields = ();

   while ($text =~ m{
      # tekst bez cudzysłowów, bez przecinków
      ( [^"',] + )
      # ...lub...
        |
      # ...pole z podwojonym cudzysłowem: (z dopuszczalnym "" w środku)
      "                     # cudzysłów rozpoczynający pole; nie przechwytujemy go
      (                     # teraz w polu pojawi się albo
       (?:     [^"]         # coś, co nie jest znakiem cudzysłowu, albo
           |
             ""             # parą powtórzonych cudzysłowów
       ) *                  # dowolną liczbę razy
      )
     "                      # cudzysłów zamykający pole ; nie przechwytywany
   }gx)
```

[4] Wyd. oryg. — *Mastering Regular Expressions*, Jeffrey E. F. Friedl, wydawnictwo O'Reilly.

```
    {
       if (defined $1) {
          $field = $1;
       } else {
          ($field = $2) =~ s/""/"/g;
       }
       push @fields, $field;
    }
    return @fields;
}
```

lub skorzystać z modułu `Text::CSV` z sieci CPAN:

```
use Text::CSV;
sub parse_csv1 {
   my $line = shift;
   my $csv = Text::CSV->new();
   return $csv->parse($line) && $csv->fields();
}
```

Można też wykorzystać inny moduł z archiwum CPAN — `Tie::CSV_File`:

```
Tie @data, "Tie::CSV_File", "data.csv";

for ($i = 0; $i < @data; $i++) {
   printf "Wiersz %d (Linia %d) to %s\n", $i, $i+1, "@{$data[$i]}";
   for($j = 0; $j < @{$data[$i]}; $j++) {
      print "Kolumna $j to <$data[$i][$j]>\n";
   }
}
```

Analiza

Dane oddzielone przecinkami są zawodną i mylącą strukturą danych. Wymaga ona dość skomplikowanego systemu obsługi, ponieważ obszary danych same w sobie mogą zawierać przecinki. Dopasowanie wzorców staje się skomplikowane i wyklucza stosowanie prostych operacji typu `split /,/`. Co gorsza, konwencje stosowania cudzysłowów i znaków ucieczki różnią się między systemami. Ta niekompatybilność powoduje, że nie da się stworzyć jednego algorytmu do wszystkich plików zawierających wartości rozdzielone przecinkami, w skrócie nazywanych danymi *CSV* (ang. *Comma-Separated Values* — wartości rozdzielone przecinkami).

Standardowy moduł `Text::ParseWords` umożliwia obsługę tych danych, w których konwencje stosowania cudzysłowów i znaków ucieczki są zgodne z używanymi w plikach uniksowych. Nadaje się więc szczególnie do analizowania wielu plików w systemach uniksowych, w których wartości oddzielone są od siebie dwukropkami, w tym *disktab*(5), *gettytab*(5), *printcap*(5) oraz *termcap*(5). Funkcji `quotewords` z tego modułu należy przekazać dwa argumenty oraz łańcuch CSV. Pierwszym argumentem jest separator (tutaj przecinek, ale często także dwukropek), drugim jest wartość prawda lub fałsz określająca czy zwracane dane mają być ujmowane w cudzysłowy.

W tego rodzaju plikach, znaki cudzysłowów wewnątrz obszarów ograniczonych cudzysłowami oznacza się przez poprzedzenie ich lewym ukośnikiem `"w \"ten\" sposób"`. Lewy ukośnik ma swoje specjalne znaczenie tylko przed znakami cudzysłowów i lewym ukośnikiem. Każde inne użycie lewego ukośnika w łańcuchu wyjściowym pozostanie niezmienione. Standardowa funkcja `quotewords()` modułu `Text::ParseWords` potrafi obsługiwać tego typu dane.

Jednak moduł ten nie przyda się na nic w systemach, w których do oznaczenia cudzysłowów wewnątrz obszarów ograniczonych cudzysłowami stosuje się ich podwojenie `"w ""ten"" sposób"`. Dla tego typu danych trzeba zastosować jedno z pozostałych rozwiązań. Pierwsze z nich oparte jest na wyrażeniu regularnym z drugiego wydania książki *Wyrażenia regularne* autorstwa Jeffreya Friedla. Zaletą tego rozwiązania jest to, że będzie działać w każdym systemie i nie trzeba instalować żadnych dodatkowych modułów. Rozwiązanie to w ogóle nie wykorzystuje żadnego modułu. Jego małą wadą jest trudność analizy kodu, pomimo dość obrazowych komentarzy.

Prezentowany w kolejnym rozwiązaniu obiektowy moduł `Text::CSV` z sieci CPAN ukrywa całą złożoność za o wiele prostszym do analizy interfejsem. Jeszcze bardziej eleganckie rozwiązanie oferuje moduł `Tie::CSV_File` z CPAN, który udostępnia coś na kształt dwuwymiarowej tablicy. Pierwszy wymiar reprezentuje poszczególne wiersze z pliku, drugi kolumny w poszczególnych wierszach.

Poniżej pokazano przykład wykorzystania podanych dwóch rodzajów procedur `parse_csv`. Operator `q()` zastępuje cudzysłowy, dzięki czemu nie musieliśmy poprzedzać wszystkiego lewymi ukośnikami.

```
$line = q(XYZZY,"","O'Reilly, Inc","Wall, Larry","a \"glug\" bit,",5,"Error,
Core Dumped");
@fields = parse_csv0($line);
for ($i = 0; $i < @fields; $i++) {
   print "$i : $fields[$i]\n";
}
0 : XYZZY
1 :
2 : O'Reilly, Inc
3 : Wall, Larry
4 : a "glug" bit,
5 : 5
6 : Error, Core Dumped
```

Jeżeli drugi argument funkcji `quotewords` miałby wartość 1, a nie 0, to cudzysłowy zostałyby zachowane, co dałoby na wyjściu następujący wynik:

```
0 : XYZZY
1 : ""
2 : "O'Reilly, Inc"
3 : "Wall, Larry"
4 : "a "glug" bit, "
5 : 5
6 : "Error, Core Dumped"
```

W ten sam sposób możemy operować na innych rodzajach plików danych, ale używając funkcji `parse_csv1` zamiast `parse_csv0`. Warto zwrócić uwagę, w jaki sposób wbudowane cudzysłowy są podwajane, a nie poprzedzane znakiem ucieczki.

```
$line = q(Dziesięć tysięcy,10000, 2710 ,,"10,000","To ""10 Kawałków""", mała",10K);
@fields = parse_csv1($line);
for ($i = 0; $i < @fields; $i++) {
   print "$i : $fields[$i]\n";
}
0 : Dziesięć tysięcy
1 : 10000
2 :  2710
3 :
4 : 10,000
5 : To "10 Kawałków", mała
6 : 10K
```

Zobacz również

Wyjaśnienie składni wyrażeń regularnych na stronie *perlre*(1) oraz w rozdziale 5. książki *Perl. Programowanie*; dokumentacja standardowego modułu `Text::ParseWords`; podrozdział „Parsing CSV Files" rozdziału 5. książki *Mastering Regular Expressions, 2nd Edition*.

1.21. Zmienne niemodyfikowalne

Problem

Potrzebujemy zmiennej, której wartość po ustawieniu nie może być modyfikowana.

Rozwiązanie

Jeżeli wystarczy nam stała, a nie zmienna skalarna, która będzie interpolowana, wystarczy zastosować dyrektywę `use constant`:

```
use constant AVOGADRO => 6.02252e23;
printf "Potrzebuujesz ich aż %g\n", AVOGADRO;
```

Jeżeli jednak musi być to zmienna, należy do typegloba przypisać odwołanie do literalnego łańcucha lub liczby, a następnie używać zmiennej skalarnej:

```
*AVOGADRO = \6.02252e23;
print "Potrzebujesz ich aż $AVOGADRO\n";
```

Jednak najbardziej niezawodny sposób polega na użyciu małej klasy, której metoda `STORE` generuje wyjątek:

```
package Tie::Constvar;
use Carp;
sub TIESCALAR {
```

```
    my ($class, $initial) = @_;
    my $var = $initcal;
    return bless \$var => $class;
}
sub FETCH {
    my $selfref = shift;
    return $$selfref;
}
sub STORE {
    confess "Nie należy zadzierać ze stałymi Wszechświata";
}
```

Analiza

Dyrektywa `use constant` jest najprostsza w użyciu, ale posiada kilka wad. Największą z nich jest fakt, iż w efekcie nie uzyskujemy zmiennej, którą można by interpolować. Inna niedogodność polega na tym, że nie stosuje zakresów widoczności; wprowadza ona procedurę o tej nazwie do przestrzeni nazw bieżącego pakietu.

Działanie tej dyrektywy polega tak naprawdę na tym, że tworzona jest procedura o podanej nazwie, nie pobierająca żadnych argumentów i zawsze zwracająca tę samą wartość (lub wartości w przypadku podania listy). Oznacza to, iż trafia ona do przestrzeni nazw bieżącego pakietu i nie ma określonego zakresu widoczności. To samo można zrobić własnoręcznie w następujący sposób:

```
sub AVOGADRO() { 6.02252e23 }
```

Chcąc ograniczyć jej widoczność tylko do bieżącego bloku, możemy utworzyć procedurę tymczasową przez przypisanie anonimowej procedury do typegloba o podanej nazwie:

```
use subs qw(AVOGADRO);
local *AVOGADRO = sub () { 6.02252e23 };
```

Ze względu na enigmatyczność tego zapisu, jeżeli nie planuje się wykorzystania tej dyrektywy, lepiej jest dodać komentarze opisujące ten fragment programu.

Jeżeli, zamiast odwołania do procedury, do typegloba przypisane zostanie odwołanie do stałej wartości skalarnej, będzie można używać zmiennej o tej nazwie. Jest to drugi sposób podany w rozwiązaniu. Wadą takiego rozwiązania jest to, że typeglobów można używać tylko dla zmiennych pakietowych, a nie leksykalnych, tworzonych za pomocą słowa kluczowego `my`. Przy stosowaniu dyrektywy `use strict` niezadeklarowane zmienne pakietowe także stanowią błąd, jednak możliwe jest ich zdeklarowanie za pomocą słowa kluczowego `our`:

```
our $AVOGADRO;
local *AVOGADRO = \6.02252E23;
```

Trzecie z podanych rozwiązań, utworzenie własnej klasy, może wydawać się najbardziej skomplikowane, zapewnia jednak największą elastyczność. Dodatkowo można deklarować zmienne leksykalne:

```
tie my $AVOGADRO, Tie::Constvar, 6.02252e23;
```

Bez przeszkód będzie więc można stosować na przykład taki zapis:

```
print "Potrzebujesz ich aż $AVOGADRO\n";
```

Ale coś takiego będzie błędem:

```
$AVOGADRO = 6.6256e-34;     # nie uda się!
```

Zobacz również

Receptura 1.15; receptura 5.3; omówienie składania stałych podprocedur pod koniec podrozdziału „Kompilacja kodu" rozdziału 18. książki *Perl. Programowanie*; moduł `Tie::Scalar: :RestrictUpdates`.

1.22. Dopasowywanie fonetyczne

Problem

Chcielibyśmy dowiedzieć się, czy dane dwa nazwiska angielskie brzmią podobnie, niezależnie od sposobu ich zapisu. Umożliwiłoby to na przykład przeprowadzenie „wyszukiwania rozmytego" podobnie brzmiących nazwisk, np. „Smith" i „Smythe", w książce telefonicznej.

Rozwiązanie

Należy wykorzystać standardowy moduł `Text::Soundex`:

```
use Text::Soundex;
$CODE  = soundex($STRING);
@CODES = soundex(@LIST);
```

lub `Text::Metaphone` z sieci CPAN:

```
use Text::Metaphone;
$phoned_words = Metaphone('Schwern');
```

Analiza

Algorytm `soundex` przeprowadza haszowanie słów (w szczególności nazwisk angielskich) do małych przestrzeni, używając prostego modelu aproksymującego angielską wymowę tych słów. Mówiąc w dużym skrócie, każde słowo jest redukowane do łańcucha czterech znaków. Pierwszy znak to wielka litera; pozostałe trzy znaki są cyframi. Porównując wartości dwóch łańcuchów, możemy stwierdzić czy podobnie brzmią słowa, które te łańcuchy reprezentują.

Poniższy program pyta o nazwisko i wyszukuje podobnie brzmiące nazwiska w pliku z hasłami. Tę samą metodę można zastosować do dowolnej bazy danych przechowującej nazwiska. Można by wobec tego kody fonetyczne z algorytmu `soundex` wykorzystać jako wartości kluczy w takiej bazie. Tak utworzone klucze nie byłyby, rzecz jasna, unikatowe.

```
use Text::Soundex;
use User::pwent;

print "Lookup user: ";
chomp($user =<STDIN>);
exit unless defined $user;
$name_code = soundex($user);

while ($uent = getpwent()) {
   ($firstname, $lastname) = $uent->gecos =~ /(\w+)[^,]*\b(\w+)/;

   if   ($name_code eq soundex($uent=>name)   ||
         $name_code eq soundex($lastname)     ||
         $name_code eq soundex($firstname)    )
   {
      printf "%s: %s %s\n", $uent->name, $firstname, $lastname;
   }
}
```

Moduł `Text::Metaphone` z CPAN rozwiązuje ten sam problem, ale w inny, lepszy sposób. Funkcja `soundex` zwraca literę i trzycyfrowy kod, który tworzony jest na podstawie jedynie początku podanego łańcucha, natomiast `Metaphone` zwraca kody o różnej długości. Na przykład:

```
                      soundex        Metaphone
Christiansen          C623           KRSXNSN
Kris Jenson           K625           KRSJNSN

Kyrie Eleison         K642           KRLSN
Curious Liaison       C624           KRSLSN
```

Aby w pełni wykorzystać możliwości modułu `Metaphone`, trzeba dodatkowo użyć modułu `String::Approx` z sieci CPAN, opisanego dokładniej w recepturze 6.13. Pozwala on na przeprowadzanie z powodzeniem dopasowań, w których przeszukiwany ciąg nie odpowiada dokładnie wzorcowi. *Odległość edycyjna* (ang. *edit distance*) to liczba zmian niezbędnych do zamiany jednego łańcucha na drugi. Poniższe wyrażenie dopasowuje parę łańcuchów, dla których odległość edycyjna wynosi dwa:

```
if (amatch("łańcuch1", [2], "łańcuch2")) {}
```

Poza tym dostępna jest jeszcze funkcja `adist`, zwracająca odległość edycyjną między podanymi łańcuchami. Odległość edycyjna między „Kris Jenson" a „Christiansen" wynosi 6, ale między ich kodami obliczonymi za pomocą modułu `Metaphone` tylko 1. Podobnie dla drugiej pary: odległość ta wynosi 8 dla ciągów oryginalnych, ale także tylko 1 dla ich kodów z `Metaphone`.

```
use Text::Metaphone qw(Metaphone);
use String::Approx qw(amatch);
```

```
if (amatch(Metaphone($s1), [1], Metaphone($s1)) {
   print "Wystarczająco podobne!\n";
}
```

Powyższy kod dopasowałby pomyślnie obie używane wyżej pary łańcuchów.

Zobacz również

Dokumentacja standardowych modułów `Text::Standard` oraz `User::pwent`; moduły `Text::Metaphone` oraz `String::Aprox` w sieci CPAN; strona *passwd*(5); rozdział 6. trzeciego tomu książki *Sztuka programowania. Sortowanie i wyszukiwanie*[5].

1.23. Program fixstyle

Załóżmy, że dana jest tabela ze starymi i nowymi łańcuchami:

Stare słowo	Nowe słowo
bonnet	Hood
rubber	eraser
lorry	truck
trousers	pants

Program z przykładu 1.4 jest filtrem, który zamienia każde wystąpienie elementu pierwszego zbioru na odpowiadający mu element z drugiego zbioru.

Przykład 1.4. Program fixstyle

```
#!/usr/bin/perl –w
# fixstyle — zamiana pierwszego zbioru <DANYCH> na drugi zbiór
#   użycie: $0 [-v] [pliki ...]
use strict;
my $verbose = ($ARGV && $ARGV[0] eq '-v' && shift);
if (@ARGV) {
   $^I = ".orig";    # zachowanie starego pliku
} else {
   warn "$0: Odczyt ze stdin\n" if –t STDIN;
}
my $code = "while (<>) {\n";
# odczytywanie konfiguracji, tworzenie kodu dla funkcji eval
while (<DATA>) {
   chomp;
   my ($in, $out) = split /\s*=>\s*/;
   next unless $in && $out;
   $code .= "s{\\Q$in\\E}{$out}g";
   $code .= "&& printf STDERR qq($in => $out w \$ARGV linia \$.\\n)" if $verbose;
   $code .= ";\n";
}
```

[5] Wyd. oryg. — *The Art Of Computer Programming*, Donald E. Knuth, wydawnictwo Addison-Wesley.

```
$code .= "print;\n}\n";
eval "{ $code } 1" || die;
__END__
analysed        => analyzed
built-in        => builtin
chastized       => chastised
commandline     => command-line
de-allocate     => deallocate
dropin          => drop-in
hardcore        => hard-core
meta-data       => metadata
multicharacter  => multi-character
multiway        => multi-way
non-empty       => nonempty
non-profit      => nonprofit
non-trappable   => nontrappable
pre-define      => predefine
preextend       => pre-extend
re-compiling    => recompiling
reenter         => re-enter
turnkey         => turn-key
```

Program ten wywołany bez podania nazwy pliku jest zwykłym filtrem. Jeżeli nazwy plików zostaną podane w wierszu poleceń, poddane zostaną edycji zmiany w nich zapisane, natomiast ich oryginalne wersje zostaną zapisane w plikach z rozszerzeniem *„.orig”*. Opis znajduje się w recepturze 7.16. Gdy umieścimy w wierszu poleceń opcję **–v**, informacja o każdej zmianie zapisana zostanie do standardowego wyjścia błędu.

Tabela ze starymi oraz nowymi łańcuchami przechowywana jest w głównym programie pod słowem `__END__`, co opisano w recepturze 7.12. Tak jak w programie *popgrep2* z receptury 6.10, każda para łańcuchów jest konwertowana na odpowiednie zamienniki i przechowywana w zmiennej `$code`.

Opcja **–t**, sprawdzająca interaktywne uruchomienie, wskazuje czy odczyt ma się odbywać z klawiatury, jeżeli nie podano żadnych argumentów. Dzięki temu jeżeli użytkownik zapomni o podaniu argumentu, nie będzie się zastanawiał czy program się zawiesił.

Jedna uwaga: ten program jest szybki, ale nie skaluje się dobrze w przypadku dokonywania tysięcy zamian. Im większa jest ilość danych w części `DATA`, tym więcej czasu zajmie konwersja. Kilkadziesiąt zmian nie zajmie wiele czasu, w takim przypadku program z przykładu 1.4 będzie wręcz działał szybciej niż poniższy. Jednak zadając mu wykonanie setek zmian, skutecznie go przyblokujemy.

Przykład 1.5 jest wersją działającą wolniej dla kilku podstawień, za to działa szybciej, mając zadane wykonanie wielu zamian.

Przykład 1.5. Program fixstyle2

```
#!usr/bin/perl –w
# fixstyle2 — taki sam jak fixstyle, jednak działający szybciej dla wielu zmian
use strict;
my $verbose = (@ARGV && $ARGV[0] eq '-v' && shift);
my %change = ();
```

```
while (<DATA>) {
   chomp;
   my ($in, $out) = split /\s*=>\s*/;
   next unless $in && $out;
   $change{$in} = $out;
}
if (@ARGV) {
   $^I = ".orig";
} else {
   warn "$0: Odczyt z stdin\n" if -t STDIN;
}
while (<>) {
   my $i = 0;
   s/^(\s+)// && print $1;          # wyświetlić początkowe spacje
   for (split /(\s+)/, $_, -1) {    # zachowanie końcowych spacji
      print( ($i++ & 1) ? $_ : ($change{$_} || $_));
   }
}
__END__
analysed        => analyzed
built-in        => builtin
chastized       => chastised
commandline     => command-line
de-allocate     => deallocate
dropin          => drop-in
hardcore        => hard-core
meta-data       => metadata
multicharacter  => multi-character
multiway        => multi-way
non-empty       => nonempty
non-profit      => nonprofit
non-trappable   => nontrappable
pre-define      => predefine
preextend       => pre-extend
re-compiling    => recompiling
reenter         => re-enter
turnkey         => turn-key
```

Ta wersja dzieli każdy wiersz na spacje i wyrazy, co działa wolniej. Następnie wyrazy te wykorzystane są do wyszukania ich odpowiedników w tablicy asocjacyjnej, co działa szybciej niż zastępowanie. Tak więc pierwsza część jest wolniejsza, druga szybsza. Różnice w szybkości zależą od liczby dopasowań.

Jeżeli nie ma potrzeby pozostawienia między wyrazami tej samej liczby spacji, druga wersja może działać tak samo szybko jak pierwsza, nawet dla kilku zmian. Znając wystarczająco dobrze dane wejściowe, możemy zamienić odstępy między wyrazami na pojedyncze spacje poprzez wstawienie następującej pętli:

```
# działa bardzo szybko, lecz usuwa odstępy
while (<>) {
   for (split) {
      print $change{$_} || $_, " ";
   }
   print "\n";
}
```

Pętla ta pozostawia dodatkowy odstęp na końcu wiersza. Jeżeli stanowi to problem, możemy zainstalować filtr wyjściowy, stosując technikę z receptury 16.5. Wystarczy wstawić następujący kod do pętli `while` usuwającej odstępy:

```
my $pid = open(STDOUT, "|-");
die "nie można wykonać fork: $!" unless defined $pid;
unless ($pid) {    # proces potomny
   while (<STDIN>) {
   s/ $//;
   print;
}
exit
}
```

1.24. Program psgrep

Wiele programów, w tym *ps*, *netstat*, *lsof*, *ls -l*, *find -ls* oraz *tcpdump*, może wyprodukować więcej danych wyjściowych niż jesteśmy w stanie wygodnie analizować. Pliki dzienników są często za duże, aby łatwo można było je przeglądać. Przepuszczając je przez filtr taki jak *grep*, możemy wybrać tylko niektóre wiersze, jednak mieszanie wyrażeń regularnych oraz złożonej logiki jest kłopotliwe; warto przejrzeć problemy napotykane w recepturze 6.18.

To, co tak naprawdę by się przydało, to możliwość stosowania pełnych zapytań do wyjścia programu lub pliku dziennika. Przykładowo, możliwość zapytania programu *ps* „Pokaż wszystkie procesy o rozmiarze większym niż 10K, ale te, które nie zostały uruchomione przez administratora" lub „Które polecenia są uruchomione na pseudoterminalach?".

Wszystko to, a nawet więcej, robi program *psgrep*, ponieważ podawane kryteria wyboru nie są zwykłymi wyrażeniami regularnymi; są pełnoprawnym kodem Perla. Każde kryterium jest po kolei stosowane do każdego pojawiającego się na wyjściu wiersza. Wyświetlane są tylko te linie, które spełniają wszystkie warunki. Poniżej podano przykładową listę rzeczy, których można szukać i sposób ich wyszukania:

Linie zawierające „sh" na końcu wyrazu:

```
% psgrep '/sh\b/'
```

Procesy, których nazwy poleceń kończą się na „sh":

```
% psgrep 'command =~ /sh$/'
```

Procesy uruchomione przez użytkowników o numerze ID mniejszym od 10:

```
% psgrep 'uid < 10'
```

Powłoki logowania w aktywnych tty:

```
% psgrep 'command =~ /^-/' 'tty ne "?"'
```

Procesy uruchomione na pseudoterminalach:

```
% psgrep 'tty =~ /^[p-t]/'
```

Odłączone procesy, które nie zostały uruchomione przez administratora:

```
% psgrep 'uid && tty eq "?"'
```

Wielkie procesy nie należące do administratora:

```
% psgrep 'size > 10 * 2**10' 'uid != 0'
```

Ostatni sposób wywołania programu *psgrep* w naszym systemie dał następujące wyniki. Jak można było przypuszczać, jedynie *netscape* i jego pochodne spełniły wszystkie kryteria.

```
 FLAGS UID  PID PPID PRI  NI  SIZE  RSS WCHAN     STA TTY TIME COMMAND
     0 101 9751    1   0   0 14932 9652 do_select S   p1  0:25 netscape
100000 101 9752 9751   0   0 10636  812 do_select S   p1  0:00 (dns helper)
```

Przykład 1.6 przedstawia program *psgrep*.

Przykład 1.6. Program psgrep

```
#!/usr/bis/perl –w
# psgrep — wyświetla wybrane wiersze z wyjścia programu ps,
#          kompilując zapytania użytkownika do kodu programu
use strict;
# każde pole z nagłówka PS
my @fieldnames =qw( FLAGS UID PID PPID PRI NICE SIZE
                    RSS WCHAN STAT TTY TIME COMMAND);
# potrzebne jest określenie formatu unpack (poniższe dotyczy ps w Linuksie)
my $fmt = cut2fmt(8, 14, 20, 26, 30, 34, 41, 47, 59, 63, 67, 72);
my %fields;    # miejsce przechowywania danych
die << "Thanatos" unless @ARGV;
użycie: $0 kryterium ...
   Każde kryterium jest wyrażeniem Perla zawierającym:
    @fieldnames
   Wszystkie kryteria muszą być spełnione, aby wyświetlić wiersz.
Thanatos
# Stworzenie aliasów funkcji dla uid, size, UID, SIZE, itd.
# Puste nawiasy w zamkniętych argumentach potrzebne do prototypów void.
for my $name (@fieldnames) {
   no strict 'refs';
   *$name = *{lc $name} = sub () { $fields{$name} };
}
my $code = "sub is_desirable { " . join(" and ", @ARGV) . " } ";
unless (eval $code.1) {
   die "Błąd w kodzie: $@\n\t$code\n";
}
open(PS, "ps wwax1 |")        || die "nie można wykonać funkcji fork: $!";
print scalar <PS>;            # wysłanie nagłówka wiersza
while (<PS>) {
   @fields{@fieldnames} = trim(unpack($fmt, $_));
   print if is_desirable();   # wiersz spełniający ich kryteria
}
close(PS)                     || die "nie udało się wykonać ps!";
# konwertowanie pozycji wcięć do formatu unpack
sub cut2fmt {
   my(@positions) = @_;
   my $template = '';
   my $lastpos = 1;
   for my $place (@positions) {
      $template .= "A" . ($place - $lastpos) . " ";
      $lastpos = $place;
   }
   $template .= "A*";
   return $template;
}
```

```
sub trim {
   my @strings = @_;
   for (@strings) {
      s/^\s+//;
      s/\s+$//;
   }
   return wantarray ? @strings : $strings[0];
}
# poniższych danych użyto do określenia punktów rozbicia kolumn.
# dalej podano też próbkę danych wejściowych
#12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345
#         1         2         3         4         5         6         7
# Pozycjonowanie
#       8     14    20    26  30  34   39    45          57  61  65     72
#       |     |     |     |   |   |    |     |           |   |   |      |
__END__
   FLAGS UID   PID  PPID PRI  NI  SIZE  RSS WCHAN       STA TTY TIME COMMAND
     100   0     1     0   0   0   760  432 do_select   S   ?   0:02 init
     140   0   187     1   0   0   784  452 do_select   S   ?   0:02 syslogd
  100100 101   428     1   0   0  1436  944 do_exit     S     1 0:00 /bin/login
  100140  99 30217   402   0   0  1552 1008 posix_lock_ S   ?   0:00 httpd
       0 101   593   428   0   0  1780 1260 copy_thread S     1 0:00 -tcsh
  100000 101 30639  9562  17   0   924  496             R    p1 0:00 ps axl
       0 101 25145  9563   0   0  2964 2360 idetape_rea S    p2 0:06 trn
  100100   0 10116  9564   0   0  1412  928 setup_frame T    p3 0:00 ssh -C www
  100100   0 26560 26554   0   0  1076  572 setup_frame T    p2 0:00 less
  100000 101 19058  9562   0   0  1396  900 setup_frame T    p1 0:02 nvi /tmp/a
```

Program *psgrep* łączy w sobie wiele technik przedstawionych w tej książce. Usuwanie początkowych i końcowych spacji z łańcucha znaków można znaleźć w recepturze 1.19. Konwertowanie wcięć do formatu `unpack` po to, by rozwinąć stałe długości pól znajduje się w recepturze 1.1. Dopasowywanie łańcuchów do wyrażeń regularnych jest podstawowym tematem rozdziału 6.

Przesyłanie wielowierszowych łańcuchów w postaci dokumentów w miejscu do funkcji `die` omówione zostało w recepturach 1.15 oraz 1.16. Przypisanie do `@fields{@fieldnames}` ustawia jednocześnie wiele wartości w tablicy asocjacyjnej o nazwie `%fields`. Wycinki tablic asocjacyjnych omówione zostały w recepturze 4.8 oraz 5.11.

Zamieszczanie przykładowych danych wejściowych poniżej `__END__` opisane zostało w recepturze 7.12. Do wykonywania testów w trakcie tworzenia programu używane było wejście z uchwytu `DATA`. Gdy program pracował właściwie, zmieniliśmy to na potok z programu *ps*, pozostawiając jednak pozostałości z oryginalnego filtru wejściowego, aby ułatwić przyszłe przenoszenie kodu i konserwację programu. Uruchamianie innych programów przez potok zostało omówione w rozdziale 16., a w szczególności w recepturach 16.10 i 16.13.

Siła i wyrazistość programu *psgrep* wynika z używania argumentów łańcuchowych nie jako zwykłych łańcuchów, lecz jako bezpośredniego kodu Perla. Podobną technikę wykorzystano w recepturze 9.9, z tą różnicą, że w programie *psgrep* argumenty użytkownika są umieszczane w procedurze o nazwie `is_desirable`. W ten sposób kompilacja łańcuchów na kod Perla następuje tylko raz, zanim program, którego dane wyjściowe będą analizowane

w ogóle rozpocznie swoje działanie. Na przykład żądanie wyświetlenia użytkowników o numerze UID mniejszym niż 10 spowoduje wygenerowanie następującego łańcucha dla funkcji `eval`:

```
eval "sub is_desirable { uid < 10 } " . 1;
```

Tajemniczy zapis `" . 1"` na końcu wiersza oznacza, iż funkcja `eval` zwróci wartość prawda, jeżeli zostanie skompilowany kod użytkownika. W ten sposób nie trzeba nawet sprawdzać zmiennej `$@`, aby przekonać się czy wystąpiły błędy kompilacji, tak jak to było robione w recepturze 10.12.

Określanie dowolnego kodu programu dla filtra wybierającego rekordy jest niesamowicie skutecznym sposobem, ale nie wywodzi się z Perla. Perl wiele zawdzięcza językowi programowania *awk*, który jest często wykorzystywany do filtracji danych. Jedyny problem tego języka polega na tym, iż nie potrafi on w prosty sposób traktować danych wejściowych jako pól o stałym rozmiarze zamiast pól oddzielonych pewnym separatorem. Poza tym pola te nie są nazywane mnemonicznie: *awk* używa oznaczeń $1, $2, itd. Perl potrafi dużo więcej.

Kryteria użytkownika nie muszą być nawet prostymi wyrażeniami. Na przykład następujące wywołanie inicjalizuje zmienną `$id` na numer użytkownika `"nobody"`, który może być później wykorzystany w dalszej części wyrażenia:

```
% psgrep 'no strict "vars";
   BEGIN { $id = getpwnam("nobody") }
   uid == $id '
```

W jaki sposób można używać wyrazów nie umieszczonych w cudzysłowach i nie poprzedzonych nawet znakiem dolara, np. `sign`, `like uid`, `command` oraz `size` do oznaczenia poszczególnych pól w każdym rekordzie wejściowym? Bezpośrednio operujemy tabelą symboli, przypisując ograniczniki do pośrednich typeglobów, co tworzy funkcje o podanych nazwach. Nazwy funkcji powstają przy użyciu zarówno wielkich, jak i małych liter, dopuszczalny jest więc zapis `"UID < 10"`, a także `"uid < 10"`. Ograniczniki opisane zostały w recepturze 11.4, a przypisywanie ich do typeglobów w celu utworzenia aliasów funkcji pokazano w recepturze 10.14.

Jedną sztuczką nie pokazaną w recepturach jest używanie ograniczników w pustych nawiasach. Pozwala to na używanie funkcji w wyrażeniach wszędzie tam, gdzie normalnie użyta byłaby pojedyncza wielkość, np. łańcuch lub stała numeryczna. Tworzy to pusty prototyp, dzięki czemu funkcja dostępu do pola `uid` nie przyjmuje żadnych argumentów, podobnie jak wbudowana funkcja `time`. Jeżeli funkcje te nie miałyby takich prototypów, to wyrażenia `"uid < 10"` lub `"size/2 > rss"` wprowadzałyby w błąd kompilator, ponieważ „widziałby" on początek niezamkniętego typegloba wieloznacznego i dopasowania wzorca. Prototypy zostały omówione w recepturze 10.11.

Wersja programu *psgrep* przedstawiona tutaj jest przystosowana do danych zwracanych z wersji programu *ps* z systemu Red Hat Linux. Dokonując przeniesienia do innych systemów, musimy sprawdzić, w których kolumnach zaczynają się poszczególne nagłówki.

Metoda ta nie dotyczy tylko programu *ps* czy systemów uniksowych; jest to ogólna technika używana do filtracji rekordów wejściowych przy użyciu wyrażeń Perla, w prosty sposób adaptowalna do innych formatów rekordów. Formatem wejściowym mogą być kolumny, rozdzielanie spacjami, przecinkami, czy też wyniki dopasowania wzorca z nawiasami przechwytującymi.

Program ten może być zmodyfikowany tak, by obsługiwał definiowaną przez użytkownika bazę danych, poprzez wprowadzenie niewielkich zmian do funkcji wybierających. Dysponując tablicą rekordów, jak opisano w recepturze 11.9, możemy pozwolić użytkownikom na określanie dowolnych kryteriów wyboru, np.:

```
sub id()        { $_->{ID} }
sub title()     { $_->{TITLE} }
sub executive() { title =~ /(?:vice-)?president/i }

# kryteria wyszukiwania określone przez użytkownika umieszczone w klauzuli grep
@slowburners =grep { id<10 && !executive } @employees;
```

Ze względów bezpieczeństwa i wydajności, rozwiązanie to jest rzadko używane w mechanizmach baz danych opisanych w rozdziale 14. Język SQL tego nie obsługuje, ale mając Perla i trochę pomysłów, możemy łatwo je samemu wprowadzić.