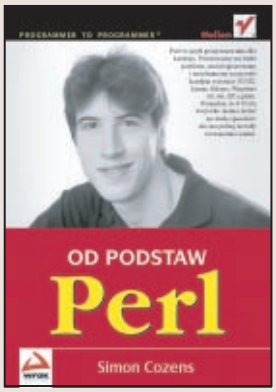
Perl. Od podstaw



Autor: Simon Cozens
Tłumaczenie: Rafał Bielec, Adam Osuchowski, Rafał Szpoton
ISBN: 83-7197-496-5
Tytuł oryginału: Beginning Perl
Format: B5, stron: 650
Przykłady na ftp: 114 kB

Perl to uniwersalny, wygodny i niezwykle elastyczny język programowania. Jego rozwój, wspierany przez rzeszę programistów, doprowadził do stworzenia narzędzia o ogromnych możliwościach dostępnego dla prawie wszystkich systemów operacyjnych. Jest to język wyjątkowy, gdyż pozwala programiście na wybór własnego stylu pisania kodu, nie narzucając „jedynie słusznych" rozwiązań. Dzięki tej zalecie Perl cieszy się wielką popularnością wśród osób nie zajmujących się zawodowo programowaniem (np. administratorów czy webmasterów). Kilka tysięcy darmowych modułów dodatkowo poszerza potencjał Perla.

Książka „Perl. Od podstaw" przedstawia szeroki zakres zastosowań tego języka. Nauczysz się z niej instalować Perla w różnych systemach operacyjnych, poznasz podstawowe konstrukcje języka: instrukcje, wyrażenia regularne i moduły, dowiesz się jak korzystać z baz danych za pomocą Perla i jak pisać skrypty CGI.

Dla kogo adresowana jest ta książka?
Książka przeznaczona jest dla wszystkich, którzy chcą poznać język Perl. Jeśli masz już doświadczenie programistyczne, będzie Ci łatwiej ją czytać; jeśli Perl jest Twoim pierwszym językiem programowania, bez trudu przebrniesz przez tą lekturę.

Książka zawiera:

- Pełen kurs języka Perl (zarówno dla Windows jak i dla Uniksa)
- Opis korzystania z dodatkowych modułów dostępnych w sieci
- Opis składni Perla
- Sposoby wykorzystania plików i baz danych w Perlu
- Omówienie programowania skryptów CGI
- Wykorzystanie Perla jako języka zorientowanego obiektowo

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

Spis treści

O Autorach ... 13

Wstęp ... 15

Krótka historia ... 15
Dlaczego Perl? ... 16
Perl jest darmowy ... 17
Do czego Perl jest używany? ... 18
Windows, Unix i inne systemy operacyjne ... 19
Znak zachęty ... 20
Co jest potrzebne, by korzystać z tej książki? ... 20
Jak mogę zdobyć Perla? ... 21
Instalacja Perla w Linuksie i Uniksie ... 21
Instalacja w Windows ... 23
Problemy w Windows ... 24
Jak uzyskać pomoc ... 26
Perldoc ... 26
Strony podręcznika ... 27
Zasoby Perla ... 29
Witryny WWW ... 29
Grupy dyskusyjne ... 30
IRC ... 30
Książki ... 30
Konwencje ... 31
Pobieranie kodu źródłowego ... 32
Przykłady ... 32

Rozdział 1. Pierwsze kroki w Perlu ... 33

Języki programowania ... 33
Kod źródłowy interpretowany i kompilowany ... 35
Biblioteki, moduły i pakiety ... 36
Dlaczego Perl jest tak potężnym językiem? ... 37
Jest naprawdę prosty ... 37
Naszym hasłem jest elastyczność ... 37
Perl w witrynach WWW ... 37
Próba darmowych źródeł ... 38
Wersje rozwojowe i Topaz ... 38
Nasz pierwszy program w Perlu ... 40
Struktura programu ... 44
Dokumentowanie swoich programów ... 44
Słowa kluczowe ... 45
Instrukcje i bloki instrukcji ... 45

ASCII i Unikod ... 47
Sekwencje specjalne ... 48
Białe spacje ... 48
Systemy numeryczne ... 48
Program uruchomieniowy Perla ... 50
Ćwiczenia ... 50

Rozdział 2. Praca z prostymi wartościami ... 51

Typy danych ... 51
Liczby ... 52
Liczby dwójkowe, szesnastkowe i ósemkowe ... 54
Łańcuchy znakowe ... 56
Pojedynczo i podwójnie cytowane łańcuchy ... 56
Alternatywne ograniczniki ... 58
Dokumenty w miejscu ... 59
Konwersja pomiędzy liczbami i łańcuchami ... 60
Operatory ... 61
Operatory numeryczne ... 61
Operatory arytmetyczne ... 61
Operatory bitowe ... 64
Prawda i fałsz ... 66
Operatory logiczne ... 69
Operatory łańcuchowe (napisowe) ... 71
Porównywanie napisów ... 73
Operatory, które poznamy później ... 74
Priorytety operatorów ... 75
Zmienne ... 76
Zmiana zawartości zmiennych ... 76
Jednoczesne operacje z przypisaniem ... 78
Autoinkrementacja i autodekrementacja ... 78
Wielokrotne przypisania ... 80
Zasięg zmiennych ... 80
Nazwy zmiennych ... 83
Interpolacja zmiennych ... 83
Konwerter walut ... 85
Wstęp do <STDIN> ... 86
Ćwiczenia ... 87

Rozdział 3. Listy i tablice asocjacyjne ... 89

Listy ... 89
Proste listy ... 90
Bardziej złożone listy ... 91
Dostęp do wartości na listach ... 94
Dzielenie listy ... 96
Przedziały ... 97
Używanie zakresów podczas podziału listy ... 99
Tablice ... 100
Przypisywanie wartości do tablic ... 100
Kontekst listy i kontekst skalarny ... 102
Dodawanie elementów do tablicy ... 103
Dostęp do elementów w tablicy ... 104
Dostęp do pojedynczych elementów ... 104
Dostęp do wielu elementów tablicy ... 107
Operacje na tablicach ... 109
Funkcje związane z tablicami ... 112

Tablice asocjacyjne ... 116
Tworzenie tablicy asocjacyjnej ... 117
Operacje na wartościach tablicy asocjacyjnej ... 118
Dodawanie wartości do tablicy asocjacyjnej oraz jej zmiana i usuwanie ... 120
Dostęp do większej liczby wartości w tablicach asocjacyjnych ... 121
Ćwiczenia ... 122

Rozdział 4. Pętle i decyzje ... 123

Decydowanie, czyli użycie „if” ... 124
Powtórzenie wiadomości o operacjach logicznych ... 128
Porównywanie liczb ... 129
Porównywanie ciągów ... 130
Inne sprawdzenie ... 131
Łączenia logiczne ... 132
Uruchom jeśli nie ... 133
Modyfikatory wyrażenia ... 133
Konstrukcje logiczne ... 134
Wielokrotny wybór ... 134
If elsif else ... 135
Rozwiązanie bardziej eleganckie ... 137
1, 2, przeskocz ile się da, 99,100 ... 137
Pętle for ... 138
Wybieranie zmiennej iteracyjnej ... 139
Co można zapętlić ... 140
Aliasy i wartości ... 140
Modyfikatory wyrażenia ... 141
Pętle while ... 143
While (<STDIN>) ... 144
Nieskończone pętle ... 145
Uruchamianie przynajmniej jeden raz ... 146
Zmiana wyrażenia ... 147
Pętla until ... 147
Kontrola nad pętlą ... 148
Wychodzenie ... 148
Omijanie pętli ... 149
Goto ... 152
Ćwiczenia ... 152

Rozdział 5. Wyrażenia regularne ... 153

Czym one są? ... 154
Wzorce ... 154
Interpolacja ... 157
Omijanie znaków specjalnych ... 159
Kotwice ... 160
Skróty i opcje ... 161
Posix i Unicode ... 164
Alternatywy ... 164
Powtórzenia ... 165
Tabelka podsumowująca ... 167
Odwołania wsteczne ... 168
Jak działa mechanizm RE ... 169
Praca z wyrażeniami regularnymi ... 171
Podstawienie ... 171
Zmiana ograniczników ... 173

Modyfikatory ... 174
Split ... 175
Join ... 176
Transformacja ... 176
Podstawowe pomyłki ... 177
Bardziej zaawansowane zagadnienia ... 177
Komentarze wewnętrzne ... 178
Modyfikatory wewnętrzne ... 178
Grupowanie bez odwołań wstecznych ... 179
Patrzenie w przód i w tył ... 179
Odwołania wsteczne (ponownie) ... 181
Ćwiczenia ... 181

Rozdział 6. Pliki i dane ... 183

Uchwyty plików ... 183
Odczytywanie linii ... 185
Tworzenie filtrów ... 186
Odczytywanie więcej niż jednej linii ... 189
Jaki jest mój separator linii? ... 190
Odczytywanie akapitów ... 192
Wczytywanie całych plików ... 193
Wpisywanie do plików ... 193
Otwieranie pliku do zapisu ... 193
Zapisywanie do uchwytu pliku ... 194
Dostęp do uchwytów plików ... 198
Zapisywanie plików binarnych ... 200
Wybieranie uchwytu ... 201
Buforowanie ... 203
Prawa dostępu ... 204
Otwieranie potoków ... 205
Wejście potokowe ... 205
Wyjście potokowe ... 208
Sprawdzanie plików ... 210
Katalogi ... 214
Glob ... 214
Odczytywanie katalogów ... 215
Ćwiczenia ... 216

Rozdział 7. Odwołania ... 217

Czym jest odwołanie? ... 217
Anonimowość ... 218
„Czas życia” odwołania ... 219
Tworzenie odwołania ... 219
Odwołania anonimowe ... 220
Wykorzystywanie odwołań ... 222
Elementy tablicy ... 224
Zmiana danych wskazywanych przez odwołanie ... 225
Odwołania do tablic asocjacyjnych ... 226
Sposób użycia skrótów ... 227
Zliczanie oraz usuwanie odwołań ... 230
Zliczanie odwołań anonimowych ... 231
Zastosowanie odwołań do struktur złożonych ... 231
Macierze ... 232
Automatyzacja ... 232

Drzewa ... 236
Listy powiązane ... 239
Ćwiczenia ... 240

Rozdział 8. Procedury ... 241

Różnica pomiędzy funkcjami i procedurami ... 242
Zazwyczaj ... 242
W Perlu ... 242
Wprowadzenie do procedur ... 243
Tworzenie procedur ... 243
Kolejność deklarowania procedur ... 245
Procedury obliczeniowe ... 248
Parametry i argumenty ... 248
Zwracanie wartości ... 249
Instrukcja return ... 251
Zachowywanie wartości ... 251
Kontekst wywołania procedury ... 252
Prototypy procedur ... 253
Zasięg widoczności zmiennych ... 255
Zmienne globalne ... 255
Zmienne leksykalne ... 257
Zakres uruchomieniowy ... 258
Kiedy używać my(), a kiedy local? ... 260
Przekazywanie bardziej złożonych parametrów ... 260
@_ zawiera aliasy ... 260
Listy są zawsze jednowymiarowe ... 261
Przekazywanie odwołań do procedury ... 261
Przekazywanie tablic zwykłych oraz asocjacyjnych do procedury ... 262
Przekazywanie uchwytów do plików ... 264
Domyślne wartości parametrów ... 265
Parametry nazwane ... 266
Odwołania do procedur ... 266
Deklaracja odwołań do procedur ... 266
Wywoływanie procedury przy użyciu odwołań ... 266
Odwołania zwrotne ... 267
Tablice zawierające odwołania do procedur ... 268
Rekurencja ... 268
Tworzenie dużych programów ... 276
Ćwiczenia ... 278

Rozdział 9. Uruchamianie i testowanie programów w Perlu ... 279

Komunikaty o błędach ... 280
Wskazówki przy wykrywaniu błędów składniowych ... 281
Brakujące średniki ... 281
Brakujące nawiasy ... 282
Niedomknięte łańcuchy ... 283
Brakujący średnik ... 283
Nawiasy wokół wyrażeń warunkowych ... 283
Słowa nierozpoznane ... 283
Moduły diagnostyczne ... 284
Dyrektywa warnings ... 284
Dyrektywa strict ... 287
Dyrektywa diagnostic ... 290

Opcje programu Perl ... 291
–e ... 292
–n oraz –p ... 293
–c ... 295
–i ... 296
–M ... 297
–s ... 297
–I oraz @INC ... 299
–a oraz –F ... 300
–l oraz –O ... 301
–T ... 301
Techniki wykrywania błędów ... 302
Zanim uruchomimy Debuggera ... 302
Komunikaty diagnostyczne ... 302
Minimalizacja ilości kodu ... 302
Kontekst ... 303
Zakres ... 303
Priorytet operacji ... 303
Korzystanie z Debuggera ... 303
Poprawne programowanie ... 304
Strategia ... 304
Sprawdź poprawność zwracanych wartości ... 305
Bądź przygotowany na rzeczy niemożliwe ... 305
Nigdy nie ufaj użytkownikowi ... 306
Istnienie danych ... 306
Dodaj pomocne komentarze ... 306
Utrzymuj porządek w kodzie programu ... 306
Ćwiczenia ... 307

Rozdział 10. Moduły ... 309

Rodzaje modułów ... 309
Dlaczego potrzebujemy modułów? ... 310
Dołączanie innych plików ... 311
Instrukcja do ... 311
Instrukcja require ... 311
Instrukcja use ... 312
Zmiana tablicy @INC ... 313
Hierarchia pakietów ... 313
Moduł eksportujący ... 314
Moduły standardowe Perla ... 315
File::Find ... 315
Getopt::Std ... 317
Getopt::Long ... 318
File::Spec ... 319
Benchmark ... 320
Win32 ... 321
Archiwum CPAN ... 323
Instalowanie modułów przy użyciu PPM ... 325
Samodzielna instalacja modułów ... 325
Moduł CPAN ... 327
Paczki ... 331
Bundle::LWP ... 332
Bundle::libnet ... 333
Umieszczanie własnych modułów w archiwum CPAN ... 333

Rozdział 11. Programowanie obiektowe w Perlu ... 335
Praca z obiektami ... 335
Przekształcanie zadań w programy obiektowe ... 336
Czy Twoje procedury opisują zadania? ... 336
Czy dane mają być trwałe? ... 336
Czy potrzebujesz sesji? ... 337
Czy potrzebujesz obiektowości? ... 337
Czy chcesz ukryć obiekty przed użytkownikiem? ... 337
Czy wciąż nie jesteś pewien? ... 337
Poszerzanie zasobu słownictwa ... 337
Obiekty ... 338
Właściwości ... 338
Metody ... 339
Klasy ... 339
Polimorfizm ... 340
Hermetyzacja ... 341
Dziedziczenie ... 341
Konstruktory ... 342
Destruktory ... 343
Tworzenie własnych obiektów ... 346
Stosowanie operatora bless ... 347
Przechowywanie właściwości ... 349
Konstruktor ... 350
Używanie dziedziczenia ... 351
Inicjalizowanie właściwości ... 351
Tworzenie metod ... 353
Rozróżnianie metod klasy oraz obiektu ... 355
Metody zmieniające wartość właściwości ... 356
Właściwości klasy ... 357
Tworzenie metod prywatnych ... 360
Metody użytkowe ... 362
„Śmierć" obiektu ... 364
Nasza ukończona klasa ... 364
Dziedziczenie ... 366
Co to jest ... 367
Dodawanie nowych metod ... 368
Nadpisywanie metod ... 369
Wiązania ... 372
Ćwiczenia ... 377

Rozdział 12. Wprowadzenie do CGI ... 379
Jak zabrać się do pracy? ... 379
Konfiguracja CGI w systemie UNIX ... 380
Apache ... 380
Uruchamianie i zatrzymywanie Apache'a ... 380
Katalogi DocumentRoot oraz cgi-bin ... 381
Konfiguracja CGI w systemie Windows ... 382
Internet Information Server ... 382
PersonalWebServer ... 383
Używanie serwerów WWW w systemie Windows ... 384
Tworzenie skryptów CGI ... 384
Podstawy CGI ... 384
Zwykły tekst ... 385
Kod HTML ... 385
Środowisko CGI ... 388

Polecenia HTTP ... 390
Metoda GET ... 391
Metoda POST ... 392
Tworzenie interakcyjnych skryptów CGI ... 392
Przykład oparty na formularzu ... 393
Przekazywanie parametrów przy użyciu CGI.pm ... 393
Określanie metody HTTP ... 395
Określanie środowiska wykonania ... 395
Tworzenie kodu HTML ... 396
Kolejne podejście do problemu zmiennych środowiskowych ... 401
Tworzenie nagłówka HTTP ... 402
Tworzenie nagłówka dokumentu ... 404
Tworzenie czytelnego kodu HTML ... 406
Tworzenie formularzy HTML ... 408
Tworzenie adresów URL odwołujących się do siebie samych ... 409
Tworzenie i przetwarzanie formularzy przy użyciu jednego skryptu ... 411
Zachowywanie i odtwarzanie stanu CGI ... 413
Przeadresowywanie ze skryptu CGI ... 416
Odświeżanie zawartości stron przy wykorzystaniu metody push ... 416
Ciasteczka oraz śledzenie sesji ... 420
Testowanie skryptów CGI ... 425
Zastosowanie CGI.pm do testowania skryptów w wierszu poleceń ... 426
Bezpieczeństwo CGI ... 427
Przykład niebezpiecznego skryptu CGI ... 427
Uruchamianie zewnętrznych programów ... 429
Komunikacja z zewnętrznymi programami ... 430
Sprawdzanie potencjalnie niebezpiecznych zmiennych ... 431
Przykład bardziej bezpiecznego skryptu CGI ... 433
Separacja programów CGI ... 434
Najważniejsze zasady bezpieczeństwa ... 435

Rozdział 13. Współpraca Perla z bazami danych ... 437

Moduł DBM ... 438
Którą implementację DBM zastosować? ... 438
Korzystanie z baz danych DBM ... 439
Otwieranie bazy danych ... 440
Sprawdzanie stanu bazy danych DBM ... 441
Tworzenie baz danych typu DBM ... 441
Usuwanie zawartości bazy danych typu DBM ... 442
Zamykanie bazy danych DBM ... 442
Dodawanie wpisów do bazy DBM i ich zmienianie ... 442
Odczytywanie wpisów z bazy danych ... 443
Usuwanie danych z bazy ... 443
Tworzenie przenośnych programów z wykorzystaniem modułu AnyDB ... 448
Przenoszenie danych pomiędzy różnymi formatami ... 450
Przechowywanie złożonych danych ... 450
Wielopoziomowe bazy DBM (MLDBM) ... 451
Wykraczając poza płaskie pliki oraz bazy DBM ... 455
Wprowadzenie do relacyjnych baz danych ... 455
Wprowadzenie do DBI ... 456
Czego potrzebujemy? ... 457
Instalacja DBI ... 458
Dostępne sterowniki ... 460

Nasz wybór — MySQL ... 463
Instalacja w systemie Windows ... 464
Instalacja w systemie Linux ... 464
Konfiguracja konta administratora ... 466
Testowanie serwera MySQL ... 466
Instalacja DBD::MySQL ... 467
Ponownie dostępne sterowniki? ... 467
Pierwsze kroki — cykl używania bazy danych ... 467
Nawiązywanie połączenia z bazą danych ... 468
Nawiązywanie połączenia ze zdalną bazą danych ... 469
Nawiązywanie połączenia przy użyciu środowiska ... 470
Czwarty parametr — Znaczniki połączenia ... 471
Rozłączanie z bazą danych ... 472
Wykorzystywanie bazy danych ... 473
Tworzenie tabeli ... 475
Dodawanie informacji do tabeli ... 479
Krótka uwaga na temat używania cudzysłowów ... 481
Uaktualnianie danych w tabeli ... 483
Odczytywanie informacji z bazy danych ... 485
Gdzie umieszczane są rekordy? ... 487
Pobieranie pojedynczej wartości ... 489
Dowiązywanie kolumn ... 490
Pobieranie wszystkich wyników ... 491
Pobieranie z instrukcji informacji o kolumnach ... 492
Usuwanie informacji z tabeli ... 494
Rzeczy, o których nie wspomnieliśmy ... 494

Rozdział 14. Świat Perla ... 497

IPC oraz zagadnienia sieciowe ... 498
Uruchamianie programów ... 499
System ... 499
Procesy oraz IPC ... 502
Sygnały ... 502
Przechwytywanie sygnałów ... 504
Funkcje fork, wait oraz exec ... 505
Zagadnienia sieciowe ... 506
Adresy IP ... 507
Gniazda i porty ... 507
System nazw domen (DNS) ... 508
Klienty sieciowe ... 509
Tworzenie klientów ... 510
IO::Socket ... 510
Blokowanie gniazd oraz moduł IO::Select ... 511
Serwery używające IO::Socket ... 512
Interfejsy graficzne ... 515
Widgety ... 516
Perl/Tk ... 516
Perl/GTK+ oraz Perl/GNOME ... 517
Glade ... 517
Perl/Qt ... 519
Moduł Perla Win32 ... 519
Moduł Math ... 519
BigInt oraz BigFloat ... 520
Język danych Perla (PDL) ... 524

Prosta trygonometria ... 524
Dołączanie obsługi liczb zespolonych ... 526
Bezpieczeństwo oraz kryptografia ... 527
Crypt — bezpieczeństwo haseł ... 527
Kryptografia z wykorzystaniem klucza publicznego ... 529
Praca z danymi ... 532
LDAP ... 532
Różne typy danych — sposób ich prezentacji ... 533
Praca w sieci ... 534
Pliki dzienników ... 534
PerlScript ... 534
Komunikacja z C ... 535
Stosowanie C w programach Perla ... 535
Osadzanie kodu Perla ... 536
To dopiero początek ... 536

Dodatek A Wyrażenia regularne ... 537

Dodatek B Zmienne specjalne ... 545

Dodatek C Wykaz funkcji ... 551

Dodatek D Moduły standardowe ... 575

Dodatek E Wykaz opcji wiersza poleceń ... 585

Dodatek F Zestaw znaków ASCII ... 589

Dodatek G Licencje ... 597

Dodatek H Rozwiązania ćwiczeń ... 607

Skorowidz ... 627

9

Uruchamianie i testowanie programów w Perlu

Poznaliśmy do tej pory zasadnicze zagadnienia Perla. Zobaczymy jeszcze, jak używać bibliotek modułów w celu wykonania wielu częstych zadań, włączając w to tworzenie aplikacji sieciowych, CGI oraz operacje na bazach danych. W tym momencie jednak znasz już Perla na tyle dobrze, aby móc wykonać wszystko, o czym pomyślisz. Gratuluję Ci dotarcia tak daleko!

Będziesz jeszcze musiał przyzwyczaić się do analizowania problemu, który chcesz rozwiązać, rozbijania go na mniejsze części i zapisywania ich w języku, który komputer mógłby zrozumieć. Jednak to nie wszystko.

Każdy popełnia błędy. To prosta prawda odnosząca się do życia. W programowaniu jest dokładnie tak samo. Kiedy będziesz pisać programy, będziesz popełniać błędy. Po przeanalizowaniu swojego pomysłu i napisaniu kodu programu przejdziesz do kolejnych dwóch etapów tworzenia oprogramowania: testowania i usuwania błędów.

W tym rozdziale przekonamy się, jak Perl pomaga nam na tych etapach. W szczególności zajmiemy się następującymi zagadnieniami:

- **Komunikaty o błędach.**
  W jaki sposób interpreter Perla sygnalizuje niepoprawne użycie języka?
- **Moduły diagnostyczne.**
  Jakie moduły mogą Ci pomóc w wykryciu i zrozumieniu błędu w kodzie?
- **Opcje Perla wywoływanego z wiersza poleceń.**
  Stworzymy program testowy, używając wiersza poleceń.
- **Techniki wykrywania błędów i opis debuggera Perla.**
  Jak usunąć wykryte błędy?

Pod koniec tego rozdziału powinieneś być w stanie rozpoznać, wykryć i — mam nadzieję — również poprawić wszystkie błędy, jakie zrobisz. Zobaczymy również, jak stworzyć procedury testowania i krótkie, jednowierszowe programy uruchamiane z wiersza poleceń.

Komunikaty o błędach

Podczas pisania programów możesz popełnić dwa rodzaje błędów: błąd składniowy oraz błąd logiczny. **Błąd składniowy** spowodowany jest najczęściej pomyłką podczas pisania programu lub wynika z niezrozumienia składni języka. Twój kod nie ma wtedy sensu i ponieważ jest niepoprawnie napisany, niezgodnie z regułami Perla, interpreter tego języka nie może go zrozumieć, wyświetlając w efekcie komunikat o błędzie.

Błąd logiczny występuje natomiast w przypadku, gdy kod napisany jest poprawnie, jednak nie działa zgodnie z Twoimi oczekiwaniami. Tego rodzaju błędy są znacznie bardziej trudne do wyśledzenia, jednakże są pewne środki i sposoby na poradzenie sobie z nimi. Na razie jednak zobaczmy sposób, w jaki Perl wykrywa błędy składniowe i sygnalizuje nam ich wystąpienie.

Ćwiczenie. Analiza błędów składniowych

Utwórzmy kod z błędami składniowymi i spójrzmy, jak Perl nas o nich powiadomi. Posłużmy się dla przykładu następującym programem:

```
#!/usr/bin/perl
# errors.plx
use warnings;
use strict;

my $a;
print "Witaj świecie."
$a=1;
if ($a == 1 {
print "\n";
}
```

```
>perl errors.plx
Scalar found where operator expected at errors.plx line 8, near "$a"
(Missing semicolon on previous line?)
syntax error at errors.plx line 8, near "$a"
syntax error at errors.plx line 9, near "1 {"
Execution of errors.plx aborted due to compilation errors
>
```

Jak to działa

Co sygnalizuje nam Perl? Najpierw interpreter zauważył coś w wierszu 8.:

```
$a=1;
```

Cóż, wygląda na to, że wszystko jest w porządku. To całkiem poprawny kod w języku Perl. Kiedy staramy się wyśledzić i zrozumieć błędy składniowe, najważniejszą rzeczą, jaką musimy zapamiętać, jest numer wiersza zwrócony przez Perl w komunikacie o błędzie. Oznacza on ostatni numer wiersza, który został przeanalizowany przed zauważeniem błędu. Nie zawsze oznacza to jednak, że błąd wystąpił w wierszu o tym numerze. Jeżeli na przykład przeoczymy nawias zamykający, Perl może starać się przeanalizować kod aż do końca pliku, zanim nas powiadomi o wystąpieniu błędu. W tym przypadku jednak otrzymaliśmy jeszcze dodatkową podpowiedź:

```
(Missing semicolon on previous line?)
```

czyli „Brakujący średnik w poprzednim wierszu" (*przyp. tłum.*).

W rzeczywistości to właśnie jest przyczyną problemu:

```
print "Witaj świecie"
```

W wierszu o numerze 7. brakuje średnika. Co jednak w takim razie oznacza komunikat: `'Scalar found where operator expected'` („Oczekiwano operatora zamiast wartości skalarnej" — *przyp. tłum.*). Jak w przypadku wszystkich komunikatów Perla, oznacza to dokładnie to, co zostało wyświetlone. Perl znalazł wartość skalarną w miejscu, w którym oczekiwał operatora. Dlaczego? Cóż, Perl zakończył przetwarzanie łańcucha, który stanowił dane do wyświetlenia przez instrukcję `print`. Ponieważ jednak nie znalazł średnika, szukał sposobu, w jaki mógłby dokończyć instrukcję. Jedynym poprawnym znakiem byłby operator, który mógłby połączyć łańcuch z czymś innym (np. operator konkatenacji, który połączyłby go z innym skalarem). Zamiast niego jednakże Perl znalazł wielkość skalarną `$a`. Ponieważ nie możesz umieścić łańcucha tuż przy zmiennej, Perl poinformował nas o wystąpieniu błędu składniowego.

Następny problem występuje w wierszu o numerze 9:

```
if ($a ==1 {
```

W tym wypadku nie mamy już dodatkowej podpowiedzi, która pomogłaby nam wyśledzić błąd. Jest on prosty i widoczny „na pierwszy rzut oka", przez co możemy go łatwo naprawić poprzez dostawienie brakującego nawiasu zamykającego. Kod powinien oczywiście wyglądać następująco:

```
if ($a ==1) {
```

Wskazówki przy wykrywaniu błędów składniowych

Wykrywanie błędów tego rodzaju może być uciążliwe. Jest to umiejętność, którą nabywa się wraz z praktyką. Większość błędów, które prawdopodobnie napotkasz, można zaliczyć do jednej z sześciu wymienionych tu kategorii:

Brakujące średniki

Jak już zauważyliśmy, jest to najczęstszy błąd składniowy. Każda instrukcja Perla powinna kończyć się średnikiem, chyba że jest ostatnią instrukcją w bloku. Czasem otrzymasz pomocną wskazówkę, taką samą, jak w poprzednim przykładzie:

```
(Missing semicolon on previous line?)
```

W innym przypadku jednak będziesz musiał domyśleć się tego sam. Pamiętaj, że numer wiersza, który dostajesz w komunikacie o błędzie, może równie dobrze nie wskazywać na wiersz, w którym wystąpił błąd, lecz jedynie na ten wiersz, przy analizie którego został wykryty.

Brakujące nawiasy

Kolejnym najczęściej występującym rodzajem błędu jest brak otwierających lub zamykających nawiasów. Jest on bardzo uciążliwy, jako że Perl czasem zwleka z zasygnalizowaniem błędu aż do końca analizowania całego pliku. Weźmy na przykład kod:

```
#!/usr/bin/perl
# braces.plx
use warnings;
use strict;
if (1) {
 print "Witaj";
my $file = shift;
if (-e $file) {
 print "Plik istnieje. \n";
}
```

Po uruchomieniu go otrzymamy:

```
>perl braces.plx
syntax error at braces.plx line 8, near "(."
Missing right curly or square bracket at braces.plx line 11, at end of line
Execution of braces.plx aborted due to compilation errors.
>
```

Jak wiemy, problem spowodowany jest przez brakujący nawias zamykający w wierszu 7., jednak Perl nie poinformował nas o tym. Aby znaleźć miejsce wystąpienia błędu tego typu w dużym pliku, możesz zastosować kilka następujących metod:

- Zastosuj odpowiednie formatowanie tekstu, używając wcięć w celu wyróżnienia bloków programu, tak jak uczyniliśmy to w naszym przykładzie. Nie wpłynie to w żaden sposób na wykonanie programu, uczyni natomiast kod programu łatwiejszym do czytania i analizowania w przypadku, gdy wystąpi błąd tego rodzaju.
- Opuszczaj celowo średniki wszędzie tam, gdzie powinien kończyć się blok programu. Wywołasz wtedy szybciej błąd składniowy[1]. Musisz jednakże pamiętać o dostawianiu ich w momencie, gdy dodasz jeszcze jakąś instrukcję na końcu bloku.
- Używaj edytora, który będzie Ci pomocny. Edytory takie jak `vi` czy `emacs` automatycznie podświetlają pasujące do siebie pary nawiasów. Są one dostępne bezpłatnie, zarówno dla systemu Unix, jak i dla Windows.

W dalszej części rozdziału przyjrzymy się bliżej ogólnym technikom wykrywania błędów.

[1] Perl nie będzie analizować całego pliku — *przyp. tłum.*

Niedomknięte łańcuchy

Nie zapominaj również o poprawnym zakończeniu łańcuchów znaków oraz wyrażeń regularnych. Ich niepoprawne zakończenie powoduje kaskadę błędów, ponieważ dalszy kod wygląda jak łańcuch znakowy, zaś dalej umieszczone łańcuchy wyglądają dla Perla jak kod programu. Jeżeli masz odrobinę szczęścia, Perl wykryje ten błąd bardzo szybko i poinformuję Cię, w którym miejscu on wystąpił. Możesz na przykład pominąć zamykający znak ″ w wierszu o numerze 6. w poprzednim przykładzie. W konsekwencji Perl pośród innych komunikatów wyświetli:

```
(Might be a runaway multi-line "" string starting on line 6)
```

Ten rodzaj błędu występuje szczególnie często, kiedy pracujesz z dokumentami w miejscu. Spójrzmy ponownie na przykład, który widzieliśmy w rozdziale 2.:

```
#!/usr/bin/perl
#heredoc.plx
use warnings;
print<<EOF;
To jest dokument w miejscu. Rozpoczyna się w kolejnym wierszu po dwóch strzałkach
➥i kończy, kiedy tekst następujący po nich zostanie znaleziony na początku wiersza,
➥tak jak poniżej:
EOF
```

Ponieważ Perl traktuje wszystko pomiędzy `print<<EOF` a `EOF` jako zwykły tekst, wystarczy pominięcie tego ostatniego, aby Perl potraktował w ten sam sposób resztę programu.

Brakujący średnik

Jeśli zapomnisz postawić średnik w miejscu, gdzie powinien się on znajdować, prawie zawsze otrzymasz komunikat: `'Scalar found where operator expected'` („Oczekiwano operatora zamiast wartości skalarnej” — *przyp. tłum.*). Dzieje się tak dlatego, że Perl próbuje połączyć dwie instrukcje w całość i nie może sobie z tym poradzić.

Nawiasy wokół wyrażeń warunkowych

Musisz używać nawiasów wokół wyrażeń warunkowych w takich instrukcjach, jak: `if`, `for`, `while`, `unless` oraz `until`. Nie potrzebujesz ich jednak, jeśli używasz ich jako modyfikatorów instrukcji.

Słowa nierozpoznane

Komunikat o błędzie zawierający słowo `'bareword'` oznacza, że Perl nie mógł odgadnąć znaczenia danego słowa. Może miałeś na myśli zmienną skalarną i zapomniałeś zadeklarować jej typ? A może w złym kontekście użyłeś uchwytu do pliku lub napisałeś z błędem nazwę procedury? Jeśli na przykład uruchomimy następujący program:

```
#!/usr/bin/perl
#bareword.plx
```

```
use warnings;
use strict;
Witaj;
```

otrzymamy komunikat:

```
>perl bareword.plx
Bareword "Witaj" not allowed while "strict subs" in use at bareword.plx line 5.
Execution of bareword.plx aborted due to compilation errors.
>
```

Takim błędom przyjrzymy się nieco dalej, podczas omawiania instrukcji `strict`.

Moduły diagnostyczne

Wspomniałem już o konieczności używania w programie instrukcji `use strict` oraz `use warnings`. Nadszedł czas, aby dokładnie wyjaśnić, co robią te oraz inne im podobne moduły.

Jak zobaczymy w następnym rozdziale, instrukcja `use` pozwala na włączenie do programu zewnętrznych modułów. Moduły `warnings` oraz `strict` są dołączane do standardowej dystrybucji Perla. Stanowią one po prostu zwykły kod Perla. Jedyną specjalną ich cechą jest fakt, iż mogą zmieniać wewnętrzne zmienne Perla, wpływając tym samym na zachowanie się interpretera.

Ściślej mówiąc, są one bardziej dyrektywami niż modułami. Wszystkie posiadają nazwy złożone z małych liter i zamiast dostarczać jakiś gotowy kod do wykorzystania przez Ciebie, wpływają na zachowanie się samego Perla.

Dyrektywa warnings

Dyrektywa ta wpływa na sposób, w jaki Perl generuje ostrzeżenia. Zwykle dostępna jest pewna ilość ostrzeżeń, których wyświetlanie może być przez Ciebie włączane i wyłączane. Można je pogrupować w kilka kategorii: ostrzeżenia syntaktyczne, niezalecane metody programowania, problemy z wyrażeniami regularnymi, operacjami wejścia-wyjścia itd.

Powtórzona deklaracja zmiennej

Domyślnie wyświetlanie wszystkich ostrzeżeń jest wyłączone. Jeżeli jednak tylko napiszesz: `use warnings`, zostanie ono włączone. Ten program na przykład uruchomi się bez żadnej reakcji, jeśli nie użyjesz omawianej instrukcji:

```
#!/usr/bin/perl
#warntest.plx
# tutaj dodaj instrukcję 'use warnings'

my $a = 0;
my $a = 4;
```

Jeśli jednak jej użyjesz, Perl zasygnalizuje:

```
>perl warntest.plx
"my" variable $a masks earlier declaration in same scope at warntest.plx line 6.
>
```

Co to oznacza? W wierszu 6. zadeklarowaliśmy nową zmienną `$a`. Jak pamiętasz, użycie `my` powoduje utworzenie całkiem nowej zmiennej. Jednak mamy ją już zadeklarowaną wcześniej — w wierszu 5. Poprzez powtórzenie deklaracji w wierszu 6. tracimy jej starą wartość, czyli 0. Ostrzeżenie to należy do kategorii ostrzeżeń różnych.

Błąd w nazwie zmiennej

Przyjrzyjmy się innej częstej przyczynie błędów — błędnemu zapisowi nazwy zmiennej:

```
#!/usr/bin/perl
#warntest2.plx
# tutaj dodaj instrukcje 'use warnings'
my $total = 30;
print "Wartość zmiennej total wynosi $total\n";
$total += 10;
print "Wartość zmiennej total wynosi $tutal\n";
```

Bez użycia instrukcji `use warning` ujrzymy:

```
>perl warntest2.plx
Wartość zmiennej total wynosi 30
Wartość zmiennej total wynosi
```

Dlaczego straciliśmy wartość zapisaną w zmiennej? Włącz wyświetlanie ostrzeżeń i uruchom program ponownie. Oto, co zobaczysz:

```
>perl warntest2.plx
Name "main::tutal" used only once: possible typo at warntest2.plx line 7.
Wartość zmiennej total wynosi 30
Use of unitialized value in concatenation (.) at warntest2.plx line 7.
Wartość zmiennej total wynosi
```

Wyświetlony został komunikat mówiący o tym, że jedynie raz użyliśmy zmiennej `tutal`. Oczywiście pomyliliśmy się w tym miejscu; w rzeczywistości bowiem chcieliśmy wpisać `total`.

To wystarcza, abyśmy wyśledzili i poprawili błąd, więc po co jeszcze drugi komunikat? Oczywiście zmienna `$tutal` nie ma przypisanej wartości, lecz gdzie występuje operator konkatenacji? Nie użyliśmy go jawnie. Zrobił to za nas sam Perl, który łańcuch `"cokolwiek $a"` rozumie jako `"cokolwiek ".$a`. Ponieważ nie zdefiniowaliśmy `$a`, Perl wyświetlił ostrzeżenie.

Zakres widoczności dyrektywy use warnings

Zakres widoczności tej dyrektywy jest taki sam, jaki miałaby zmienna `my`, tzn. jest nim ten sam blok programu ograniczony nawiasami lub koniec aktualnego pliku. Nasz kolejny przykładowy program wyświetla ostrzeżenia występujące w całym kodzie programu:

```
#!/usr/bin/perl
#warntest3.plx
use warnings;
{
        my @a = qw(raz , dwa , trzy , cztery);
}
my @b = qw(raz , dwa , trzy , cztery);
```

W związku z tym Perl odpowie, wyświetlając następujące ostrzeżenia:

```
>perl warntest3.plx
Possible attempt to separate words with commas at warntest3.plx line 5.
Possible attempt to separate words with commas at warntest3.plx line 7.
>
```

i przypominając nam tym samym, że — ponieważ `qw()` zmienia automatycznie rozdzielone słowa w oddzielne elementy — nie musimy już rozdzielać ich przecinkami.

Jeżeli naprawdę chcesz użyć przecinków jako elementów w tablicy, możesz wyłączyć wyświetlanie ostrzeżeń poprzez użycie `no warnings`. W kolejnym programie wyświetlanie ostrzeżeń zostało wyłączone jedynie wewnątrz nawiasów:

```
#!/usr/bin.perl
#warntest4.plx
use warnings;
{
 no warnings;
my @a = qw(raz , dwa , trzy , cztery);
}
my @b = qw(raz , dwa , trzy , cztery);
```

Teraz Perl wyświetli jedynie jedno ostrzeżenie dla drugiej tablicy:

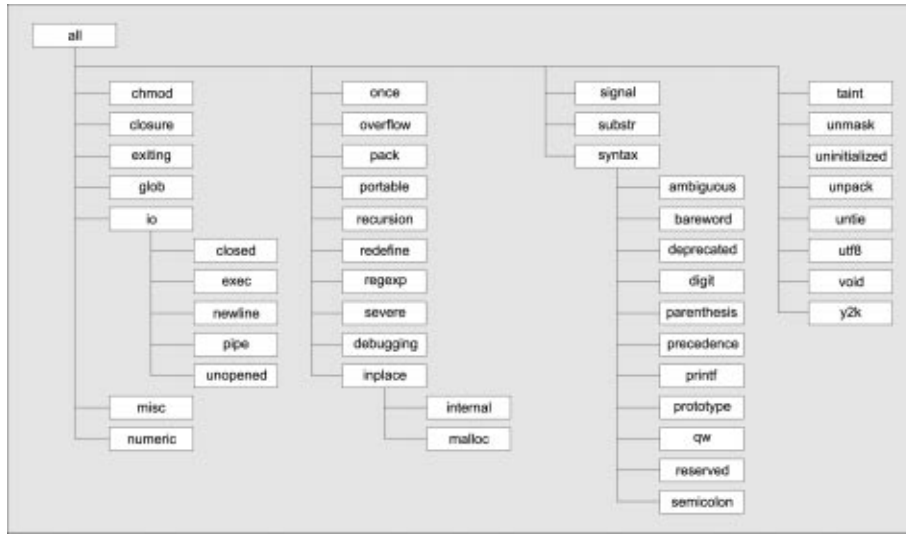
```
>perl warntest3.plx
Possible attempt to separate words with commas at warntest3.plx line 8.
>
```

czyli „Przypuszczalna próba oddzielenia słów przecinkami w `warntest3.plx`, wiersz 8” (*przyp. tłum.*).

W celu włączenia lub wyłączenia wyświetlania ostrzeżeń należących jedynie do określonej grupy, podaj jej nazwę tuż po instrukcji `use warnings`. W tym przypadku więc, aby wyłączyć wyświetlanie ostrzeżeń należących dotyczących `qw`, napisałbyś taki kod:

```
#!/usr/bin.perl
#warntest4.plx
use warnings;
{
 no warnings "qw";
my @a = qw(raz , dwa , trzy , cztery);
}
my @b = qw(raz , dwa , trzy , cztery);
```

Rodzaje ostrzeżeń, które możesz włączać i wyłączać (opisane dokładnie w dokumentacji `perldiag`), są zorganizowane schematycznie, jak na następującym schemacie:



Dyrektywa strict

Powinieneś już także wiedzieć, że instrukcja `use strict` zmusza Cię do deklarowania zmiennych przed ich użyciem. W rzeczywistości nadzoruje trzy obszary programowania: zmienne, odwołania i procedury.

Zmienne

Przyjrzyjmy się najpierw zmiennym. Kiedy używasz dyrektywy `use strict`, każda zmienna musi być najpierw zadeklarowana lokalnie (poprzez użycie instrukcji `my $var`) w bloku lub pliku, lub globalnie, tak aby była dostępna w całym programie. W tym drugim przypadku możesz zadeklarować zmienną, używając `our $var` lub pełnej nazwy `$main::var`. W następnym rozdziale zobaczymy, skąd wzięło się owo `main`.

Możesz również spotkać inny, starszy sposób deklarowania zmiennej globalnej: `use vars '$var'`. W efekcie otrzymujemy dokładnie to samo, co po użyciu `our`. Ten ostatni sposób wprowadzony został wraz z wersją Perla o numerze 5.6.0. Jego używanie zalecane jest wszędzie tam, gdzie nie jest wymagane zachowanie zgodności z poprzednimi wersjami.

Jeśli użyłeś dyrektywy `use strict` i nie zadeklarowałeś zmiennej w jeden z przedstawionych tu sposobów, Perl nie pozwoli na uruchomienie programu.

```
#!/usr/bin/perl
# strictvar.plx
use warnings;
use strict;
my $a = 5;
$main::b = "DOBRZE";
our $c = 10;
$d = "ZLE";
```

Pierwsze trzy zmienne zostały zadeklarowane poprawnie, lecz Perl wyświetli komunikat:

```
>perl strictvar.plx
Global symbol "$d" requires explicit package name at strictvar.plx line 8;
Execution of strictvar.plx aborted due to compilation errors
>
```

Komunikat ten oznacza: „Symbol globalny `"$d"` wymaga jawnego podania nazwy pakietu w *strictvar.plx* wiersz 9." (*przyp. tłum.*).

W celu poprawienia błędu powinieneś użyć jednego z pokazanych tu sposobów deklarowania zmiennych. To kolejna ważna lekcja w nauce wykrywania błędów. Nigdy nie wyłączaj wyświetlania ostrzeżeń o błędach, lecz je naprawiaj. Stanie się to szczególnie ważne, kiedy przejdziemy do kolejnej przyczyny problemów, czyli odwołań.

Odwołania

Rzeczą, którą najczęściej chcą wykonać początkujący programiści, jest stworzenie zmiennej, której nazwa generowana jest na podstawie zawartości innej zmiennej. Mógłbyś na przykład zliczać liczby w pliku składającym się z kilku sekcji. Zawsze, gdy przechodzisz do kolejnej, chciałbyś przechowywać sumę w oddzielnej zmiennej. Mógłbyś tworzyć zmienne o nazwach `$total1`, `$total2`, `$total3` itd. oraz trzymać numer aktualnej sekcji w zmiennej `$section`. Problemem jest utworzenie zmiennej, której nazwa składałaby się z wyrazu `"total"` oraz aktualnej wartości `$section`. Jak możesz go rozwiązać?

- Szczera odpowiedź:

  Możesz napisać `${"total".$section}`.

- Lepsza odpowiedź:

  Nie rób tego. W takich przypadkach zawsze lepiej użyć tablicy — zwykłej lub asocjacyjnej. W tym przypadku, ponieważ kolejne nazwy sekcji są numerami, mógłbyś użyć tablicy z sumami. Znacznie prościej napisać `$total[$section]`. Bardziej ogólnie, gdyby nazwy sekcji były łańcuchami, użyłbyś tablicy asocjacyjnej `$total{$section}`.

Dlaczego? Najbardziej oczywistym wytłumaczeniem jest: ponieważ wiesz już, jak używać tablic, a w momencie zadania pytania nie wiedziałeś jeszcze, jak skonstruować nazwę zmiennej w inny sposób. Używaj metod, które już znasz! Nie staraj się być zbyt przebiegły, jeśli jest prostsze rozwiązanie. Tworzenie takich **odwołań symbolicznych**[2] może spowodować niepotrzebne zamieszanie w zmiennych.

Przypuśćmy, że tworzysz zmienną nie jako `${"total".$section}`, lecz `${$section}`, gdzie zmienna `$section` jest czytana z pliku. Gdyby podczas odczytywania nazwy sekcji wystąpił błąd, mógłbyś otrzymać w zmiennej `$section` jedną ze zmiennych specjalnych Perla. Mogłoby to spowodować błąd lub dziwne zachowanie w dalszej części programu — tablice mogłyby przestać działać, zachowanie wyrażeń regularnych stałoby się nieprzewidywalne itp. Taki rodzaj błędów jest niezwykle trudny do wykrycia.

[2] *Symbolic references — przyp. tłum.*

Nawet gdyby wszystko poszło dobrze, nie ma gwarancji, że `$section` nie będzie zawierać nazwy, której używasz gdzieś w innej części programu. Nie jest to dobra sytuacja. Użycie dyrektywy `use strict` zapobiega jej wystąpieniu poprzez niedopuszczenie do użycia odwołań symbolicznych.

Podprogramy

Rzeczą nie mniej istotną jest fakt, iż użycie dyrektywy `use strict` zapobiega użyciu przed ich zadeklarowaniem nazw podprogramów bez nawiasów[3]. Na przykład próba uruchomienia programu:

```
#!/usr/bin/perl
# strictsubs1.plx
use warnings;
use strict;

$a = twelve;
sub twelve { return 12; }
```

spowoduje wyświetlenie komunikatu o błędzie:

```
>perl strictsubs1.plx
Bareword "twelve" not allowed while "strict subs in use at strictsubs1.plx line 6.
Execution of strictsubs1.plx aborted due to compilation errors.
>
```

Kolejny przykład jest już poprawny. Zostanie wyświetlony komunikat `'Name "main::a" used only once: possible typo'`(„Nazwa `"main::a"` użyta tylko raz: możliwy błąd typograficzny” — *przyp. tłum.*). Jest on spowodowany faktem, iż nie używamy zadeklarowanej zmiennej `$a`. Wrócimy do tego błędu za chwilkę:

```
#!/usr/bin/perl
# strictsubs2.plx
use warnings;
use strict;
sub twelve { return 12; }
$a = twelve;
```

Oczywiście możesz zawsze obejść to ograniczenie poprzez użycie nazwy podprogramu wraz nawiasami. Takie wywołanie jest zawsze poprawne.

```
#!/usr/bin/perl
# strictsubs3.plx
use warnings;
use strict;
sub twelve { return 12; }
$a = twelve();
```

Te trzy obszary — zmienne, odwołania oraz procedury są podzielone na kategorie, podobnie jak ostrzeżenia. Są to odpowiednio `vars`, `refs` i `subs`.

[3] Perl zakłada, że pomyliłeś się w tym momencie i wyświetla komunikat o nierozpoznanej instrukcji — *przyp. tłum.*

Tak jak poprzednio, użycie dyrektywy `use strict` włącza sprawdzanie wszystkiego. Możesz jednak wyłączyć (lub włączyć) sprawdzanie każdej z kategorii, tak jak w przypadku dyrektywy `use warnings`:

```
#!/usr/bin/perl
# nostrict.plx
use warnings;
use strict;
our $first = "this";
our $second = "first";
our $third;
{
 no strict ('refs');
 $third = ${$second};
}
print "$third\n";
```

```
>perl nostrict.plx
Name "main::first" used only once: possible typo at nostrict.plx line 5.
This
>
```

Ostrzeżenia zostały wyłączone w momencie deklarowania odwołania symbolicznego. Dostaliśmy jednak ostrzeżenie o jednokrotnym użyciu zmiennej `$first`, nawet pomimo że została ona później użyta w sposób niejawny[4]. W ten sposób właśnie działają ostrzeżenia: Perl sprawdza strukturalną poprawność kodu, lecz nie oblicza wartości zmiennych, jakie będą mieć w momencie uruchomienia programu. Gdyby było inaczej, z pewnością zmienna `${$second}` zostałaby rozwinięta jako `$first`.

> Nie wyłączaj sprawdzania poprawności, nawet jeśli nie pozwala ono na uruchomienie programu. Powinieneś zawsze znaleźć sposób na poprawienie błędów w programie.

Dyrektywa diagnostic

Jest jeszcze jedna dyrektywa, która może Ci pomóc podczas wykrywania błędów. Użycie dyrektywy `use diagnostics` spowoduje nie tylko wyświetlenie komunikatu o błędzie lub ostrzeżenia, lecz także tekstu wyjaśnienia z dokumentacji `perldiag`. Na przykład uruchomienie programu:

```
#!/usr/bin/perl
# diagtest.plx
use warnings;
use strict;
use diagnostics;

my $a,  $b = 6;
$a = $b;
```

powinno spowodować wyświetlenie komunikatu:

[4] Podczas przypisania `$third = ${$second}` — *przyp. tłum.*

```
>perl diagtest.plx
Parentheses missing around "my" list at diagtest.plx line 7 (#1)
  (W parenthesis) You said something like
        my $foo, $bar = @_;
  when you meant
        my ($foo, $bar) = @_;
  Remember that "my", "our", and "local" bind tighter than comma.
>
```

Jest to bardzo przydatne podczas testowania programu. Pamiętaj jednak, że użycie dyrektywy `use diagnostics` spowoduje konieczność załadowania do pamięci całej strony z programu `perldiag`, co zabiera chwilę czasu. Dobrą metodą jest używanie jej podczas pisania programu i usunięcie po jego zakończeniu.

Możesz ewentualnie użyć oddzielnego programu `splain`, który wyjaśnia ostrzeżenia Perla i komunikaty o błędach w dokładnie ten sam sposób. Przenieś je zwyczajnie do programu `splain`. Jeśli zamierzasz użyć strumienia, pamiętaj o tym, że ostrzeżenia pojawiają się na standardowym wyjściu z błędami `err`, tak więc musisz napisać `perl myprogram.plx | 2>&1 | splain`, aby je przekierować. Pamiętaj, iż program `splain` nie będzie działać w systemie Windows.

Opcje programu Perl

Wszystkie nasze dotychczasowe programy rozpoczynały się wierszem:

```
#!/usr/bin/perl
```

Natomiast nasz program uruchamialiśmy poprzez:

```
>perl program.plx
```

lub ewentualnie (w systemie Unix):

```
>./program.plx
```

Podstawowym celem umieszczenia tego wiersza na początku programu jest przekazanie systemowi Unix informacji o tym, co ma zrobić z danym plikiem. Jeśli napiszemy: `./program.plx`, oznacza to: „**uruchom program zawarty w pliku** `program.plx`”. Pierwszy wiersz wskazuje na to, w jaki sposób program ma być uruchomiony. Według zawartych w nim informacji powinien być przekazany do programu `/usr/bin/perl`, gdzie zazwyczaj umieszczany jest interpreter Perla.

Jednak nie jest to jedyne zadanie tego wiersza. Perl czyta go i szuka jakiegokolwiek dodatkowego tekstu zawierającego opcje, które informują go o szczególnym zachowaniu podczas przetwarzania pliku. Jeśli uruchamiamy interpreter Perla jawnie — w wierszu poleceń, poprzez użycie składni: `perl program.plx`, możemy również zdefiniować opcje, które zostaną zinterpretowane przed wykonaniem programu zawartego w pliku.

Wszystkie opcje są poprzedzone znakiem minus, po którym następuje znak alfanumeryczny. Muszą być również umieszczone po słowie `perl`, lecz przed nazwą programu, który ma zostać uruchomiony. Na przykład opcja `-w`, która jest równoważna (mniej więcej) z użyciem dyrektywy `use warnings`, może być użyta w pliku z programem w następujący sposób:

```
#!/usr/bin/perl -w
# program.plx
...
```

lub też w wierszu poleceń:

```
>perl -w program.plx
```

Pozwala nam to na zmianę zachowania Perla zarówno podczas pisania programu, jak i podczas jego uruchamiania. Niektóre opcje mogą zostać użyte jedynie w wierszu poleceń. Kiedy Perl otworzy i przeczyta plik, może być już zbyt późno na zmianę jego zachowania. Będzie to bardzo jasno zobrazowane na przykładzie opcji `-e`, której przyjrzymy się wkrótce.

Istnieją dwa rodzaje opcji: te, które wymagają argumentu i te, które go nie potrzebują. Opcja `-w` nie potrzebuje argumentu, podobnie jak `-c` (jak działa opcja `-c` zobaczymy już niedługo). Jeśli chcesz użyć obu tych opcji, możesz umieścić je kolejno po sobie: `-w -c` lub nawet zgrupować je do postaci `-wc`.

W przypadku opcji, które wymagają argumentu (na przykład `-i`), musi on następować zaraz po opcji. Możesz więc zgrupować opcje `-w`, `-c` oraz `-i00` jako `-wci00`; nie możesz jednak użyć zapisu `-i00wc`, ponieważ w tym przypadku `wc` zostanie zinterpretowane jako część argumentu opcji `-i`. Musisz zgrupować opcje tak, aby argument umieszczony był na końcu, lub — jeśli nie chcesz tego robić — powinieneś całkowicie je rozdzielić.

-e

Jest to najczęściej używana opcja. Ponieważ każe ona interpreterowi Perla uruchomić — zamiast programu zawartego w pliku — program zawarty w następującym po niej tekście, może być użyta jedynie w wierszu poleceń. Pozwala przez to na bardzo szybkie pisanie programów Perla. Na przykład pierwszy program, który napisaliśmy w tej książce, mógłby zostać umieszczony w wierszu poleceń w następujący sposób:

```
>perl -e 'print "Witaj świecie\n";'
Witaj świecie
>
```

Zauważ, iż umieściliśmy nasz cały program w apostrofach. Zrobiliśmy tak, ponieważ — jak dowiedzieliśmy się już wcześniej, przyglądając się tablicy `@ARGV` — program powłoki rozdziela parametry przekazane w wierszu poleceń na oddzielne słowa. Bez apostrofów, nasz program byłby złożony jedynie z instrukcji `print`, zaś `"Witaj świecie\n"` byłoby pierwszym elementem tablicy `@ARGV`.

Wiążą się z tym — niestety — dwa problemy. Pierwszy polega na tym, iż nie możemy umieścić apostrofów wewnątrz innych apostrofów; drugim jest to, że w przypadku programów powłoki niektórych systemów operacyjnych byłoby lepiej, abyś umieszczał swoje programy raczej w cudzysłowach.

Na przykład DOS, Windows itp. preferują zapis:

```
>perl -e "print \"Witaj świecie\";"
```

Najczęściej możesz uniknąć tej niedogodności przez rozważne stosowanie operatorów q/ / oraz qq/ /. Mógłbyś na przykład napisać: perl -e 'print qq/Witaj świecie\n/;', co można będzie łatwo przekształcić do postaci akceptowalnej przez DOS: perl -e "print qq/Witaj świecie\n/;". Zauważ, że w systemach UNIX preferowane są apostrofy, gdyż zapobiegają interpretowaniu zmiennych.

W kolejnych przykładach będziemy używać zapisu z apostrofami. Jeśli zatem używasz systemu Windows, przekształć je do formatu z cudzysłowami w sposób, jaki przed chwilą opisaliśmy.

Technika ta używana jest najczęściej w celu:

- szybkiego tworzenia programów działających z innymi opcjami Perla (jak to zobaczymy dalej);
- testowania małych fragmentów kodu programu w celu sprawdzenia, jak działają w Perlu.

Jeśli na przykład nie byłbym przekonany, czy znak podkreślenia zostanie dopasowany do \w w momencie użycia wyrażenia regularnego, w celu sprawdzenia tego napisałbym następujący programik:

```
>perl -e 'print qq/Tak jest dołączony\n/ if q/_/ =~ /\w/;'
Tak jest dołączony
>
```

Taki sposób jest często szybszy od przeszukiwania książek lub dokumentacji w celu sprawdzenia tego, co nas interesuje. Jak mówi Larry Wall: „Programowanie w Perlu jest nauką doświadczalną". Uczysz się poprzez wykonywanie zadań. Jeśli nie jesteś pewien niektórych elementów Perla, po prostu sprawdź je, korzystając z wiersza poleceń.

–n oraz –p

Jak już wspomniałem, możesz połączyć opcję -e z innymi w celu stworzenia w wierszu poleceń przydatnych programów. Najczęściej używanymi w ten sposób opcjami są -n oraz -p. Obie korzystają z tablicy <ARGV>. W rzeczywistości opcja -n jest równoważna z następującym zapisem:

```
while (<>) {"a tutaj twój kod"}
```

Opcji tych możemy użyć do napisania programów przeszukujących pliki, szukających zgodnych linii, zmieniających tekst itp. Poniżej na przykład przedstawiony jest jednowierszowy program wyświetlający temat oraz nadawcę nowych listów w mojej skrzynce pocztowej.

Ćwiczenie. Sprawdzanie nowej poczty

Wszystkie przychodzące do mnie listy umieszczane są w pliku o nazwie Mailbox. Każdy z nich zawiera nagłówek z informacjami o tymże liście. Na przykład tu zamieściłem fragment nagłówka listu, który wysłałem do perl5-porters:

```
Date: Mon, 3 Apr 2000 14:22:03 +0900
From: Simon Cozens <simon@cozens.net>
To:  perl5-porters@perl.org
Subject: [PATCH] t/lib/b.t
Message-ID: <20000403142203.A1437@SCOZENS>
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
X-Mailer: Mutt 1.0.1i
```

Jak widzisz, każdy wiersz nagłówka składa się z pewnego tekstu, dwukropka, spacji, po nich zaś występuje znów pewna ilość tekstu. Gdybyśmy mogli wydobyć wiersze rozpoczynające się od `Subject:` oraz `From:`, moglibyśmy streścić zawartość skrzynki pocztowej.

Popatrzmy, jak można tego dokonać z wiersza poleceń:

```
>perl -ne 'print if /^(Subject|From): /' Mailbox
From: Simon Cozens <simon@brecon.co.uk>
Subject: [PATCH] t/lib/b.t
>
```

Jak to działa

W celu wydobycia interesujących nas wierszy moglibyśmy napisać następujący program:

```
#!/usr/bin/perl
use warnings;
use strict;
open INPUT, "Mailbox" or die $!;
while (<INPUT>) {
 print if /^(Subject|From): /;
}
```

To jednak zbyt dużo pracy jak na tak proste zadanie. Perl został stworzony po to, by ułatwić rozwiązywanie tego typu problemów. Zamiast pisać cały program, użyliśmy opcji `-n`, zastępującej pętlę `while(<>)` oraz opcji `-e`, która dostarcza nam pozostałej linii. Nasz jednowierszowy program został przekształcony wewnętrznie przez Perl do postaci:

```
LINE: while (defined($_ = <ARGV>)) {
  Print $_ if /^(Subject|From): /;
}
```

Jak możesz się domyślać, nie jesteśmy ograniczeni jedynie do wyświetlania tekstu. W rzeczywistości możemy używać naszych jednowierszowych programów do modyfikowania części pliku. Załóżmy, że w pliku `newyear.txt` mamy stary list, zawierający taki tekst:

```
Dziękujemy Ci za zakupy dokonane w poprzednim roku. Oczekujemy z niecierpliwością
➥nowych wyzwań, które przyniesie nam nowy rok 1999. Mamy nadzieję, iż również w tym
➥roku będziemy mogli Ci służyć.
Przesyłamy życzenia radości i pomyślności  w nowym, 1999 roku!
```

Moglibyśmy użyć Perla w celu wyświetlenia nowej, zaktualizowanej wersji w następujący sposób:

```
>perl -ne 's/1999/2000/g; print' newyear.txt
Dziękujemy Ci za zakupy dokonane w poprzednim roku. Oczekujemy z niecierpliwością
➥nowych wyzwań, które przyniesie nam nowy rok 2000. Mamy nadzieję, iż również w tym
➥roku będziemy mogli Ci służyć.
Przesyłamy życzenia radości i pomyślności  w nowym, 2000 roku!
>
```

Oczywiście wyświetliliśmy zmienioną wersję na wyjściu STDOUT. Moglibyśmy pokusić się o kolejną zmianę i przekierować wyjście z pliku w celu zapisania go do pliku, w sposób, jaki poznaliśmy już w rozdziale 6.

```
>perl -ne 's/1999/2000/g; print' newyear.txt >changedfile.txt
>
```

Ponieważ operacja typu „**zrób coś z danymi wejściowymi, a następnie wyświetl je ponownie**" jest bardzo częsta, Perl pozwala nam na użycie (zamiast -n) opcji -p, która powoduje automatyczne wyświetlenie każdego wiersza po skończonym przetwarzaniu. Możemy zatem oszczędzić sobie kilku cennych uderzeń w klawiaturę poprzez napisanie:

```
>perl -pe 's/1999/2000/g' newyear.txt
```

Jak widzieliśmy już wcześniej, Perl rozwija te jednowierszowe programy w zwykłe pętle while. Możemy więc również w zwyczajny sposób użyć instrukcji next oraz last. W celu wyświetlenia tylko tych linii, które nie zaczynają się od znaku krzyżyka (#), możemy napisać:

```
>perl -ne 'next if /^#/; print' strictvar.plx
use warnings;
use strict;
my $a =5;
$main::b = "DOBRZE";
our $c = 10;
$d = "ZLE";
>
```

Zauważ, iż nie napisaliśmy (i w rzeczywistości nie możemy napisać):

```
>perl -pe 'next if /^#/'
```

Jest to spowodowane tym, że Perl wywołany z opcją -p używa specjalnej instrukcji continue, która tłumaczona jest wewnętrznie do postaci:

```
LINE: while (defined($_ = <ARGV>)) {
  "Tutaj twój kod";
} continue { print $_;}
```

Wszystko, co jest zawarte w bloku continue { }, zostanie zawsze wykonane pod koniec iteracji — nawet w przypadku użycia next.

-c

Opcja ta nie powoduje uruchomienia programu. Zamiast tego Perl sprawdza, czy kod może być poprawnie skompilowany. Jest to dobry sposób na szybkie sprawdzenie, czy w kodzie nie występują rażące błędy składniowe. Perl ładuje również i sprawdza wszystkie moduły używane przez program, możesz więc używać jej w celu sprawdzenia, czy program ma wszystko, czego potrzebuje.

```
>perl -ce 'print "Witaj świecie\n";'
-e syntax OK.
>perl -ce 'print ("Witaj świecie\n"));'
syntax error at -e line 1, near "))"
-e had compilation errors
>
```

Bądź jednak zawsze ostrożny. Pozytywny test nie zawsze oznacza, że program uruchomi się poprawnie. Perl sprawdza jedynie, czy jest on poprawny gramatycznie, lecz nie sprawdza tego, czy ma jakikolwiek sens. Ten program wygląda poprawnie:

```
>perl -ce 'if (1) {next}'
-e syntax OK.
>
```

lecz jeśli spróbujesz go normalnie uruchomić, otrzymasz komunikat o błędzie:

```
>perl -ce 'if (1) {next}'
Can't "next" outside loop block at -e line 1.
>
```

Istnieje pewna różnica pomiędzy czasem kompilacji a czasem wykonywania programu. Błędy czasu kompilacji mogą być wykryte wcześniej i oznaczają, że Perl nie rozumie tego, co napisałeś. Błędy czasu wykonania natomiast oznaczają, że to, co napisałeś, zostało zrozumiane, lecz z pewnych powodów nie może zostać wykonane. Opcja `-c` powoduje sprawdzenie jedynie błędów czasu kompilacji.

-i

Podczas wykonywania operacji zmiany pliku najczęściej nie chcemy uzyskiwać jego zmienionej postaci na standardowym wyjściu, lecz raczej dokonywać zmian wewnątrz pliku. Moglibyśmy spróbować wprowadzić je w następujący sposób:

```
>perl -pe 's/raz/dwa/g' textfile.txt > textfile.txt
```

Wiąże się z tym jednakże pewien problem, o którym dowiedziałbyś się podczas próby wykonania tej komendy. Jest bardzo prawdopodobne, że straciłbyś w przypadku jej zastosowania całą zawartość pliku. Jeżeli nie używasz na tyle inteligentnej powłoki shella, aby mogła ochronić Cię przed błędami, najprawdopodobniej otworzyłaby ona plik do zapisu przed przekazaniem jego uchwytu do perla. W tym momencie poprzednia zawartość pliku byłaby już wymazana.

Aby obejść tę niedogodność, musiałbyś spróbować wykonać coś takiego:

```
>perl -pe 's/raz/dwa/g' textfile.txt > textfile.new
>mv textfile.new textfile.txt
```

Komenda `mv` w systemie Unix działa identycznie z komendą `ren` w systemie Windows i jest używana do zmiany nazwy pliku.

Perl pomaga Ci w rozwiązaniu przedstawionego tu problemu. Opcja `-i` powoduje utworzenie pliku tymczasowego i automatyczną zamianę edytowanego pliku nowym po zakończeniu jego przetwarzania. Teraz możesz zrobić to, co chciałeś, w jednym wierszu:

```
>perl -pi -e 's/raz/dwa/g' textfile.txt
```

No cóż, może nie jest to do końca prawda. W tej postaci polecenia perl zwróci komunikat:

```
Can't do inplace edit without backup
```

Dzieje się tak, ponieważ perl nie wie, jak nazwać plik tymczasowy. Zauważ, iż oddzieliłem opcję `-i` od `-e`. Zrobiłem tak, gdyż opcja `-i` wymaga podania dodatkowego argumentu. Wszystko, co następuje bezpośrednio po `-i`, zostanie potraktowane jako rozszerzenie, które musi być dodane do oryginalnej nazwy pliku w celu utworzenia jego tymczasowej kopii. Oto przykład:

```
>perl -pi.old -e 's/raz/dwa/g' textfile.txt
```

Perl odczyta zawartość pliku `textfile.txt`, zapisze ją w tymczasowym pliku `textfile.txt.old`, a następnie zamieni każde wystąpienie wyrazu `'raz'` na `'dwa'`.

-M

Opcji tej możesz użyć, jeśli potrzebujesz załadować z wiersza poleceń jakikolwiek moduł. Na przykład, aby zachować w naszych jednowierszowych programach całkowitą poprawność, powinniśmy tak naprawdę napisać:

```
>perl -Mstrict -Mwarnings -e ...
```

W rzeczywistości kod, który umieszczamy zazwyczaj w wierszu poleceń, nie wymaga aż takiej dokładności. Opcja `-M` jest jednak czasem przydatna: na przykład w przeszłości utworzona została jednowierszowa przeglądarka WWW, opracowana z wykorzystaniem modułów `LWP::Simple`, `Tk` oraz `HTML::Parser`.

-s

Niekiedy mógłbyś również chcieć, aby Twój program miał możliwość odczytu swoich opcji, podobnie do tego, jak odczytujesz je teraz, przekazując je do Perla. Opcja `-s` powoduje, że Perl traktuje wszystkie opcje umieszczone w wierszu poleceń po nazwie pliku jako dostępne wewnątrz programu zmienne o tych samych nazwach. Zostają one jednocześnie wymazane z tablicy `@ARGV`. Oznacza to, że możesz je przetwarzać w dowolny sposób.

Na przykład wiele programów wywołanych w wierszu poleceń z opcją `-h` wyświetla pomoc, wyjaśniającą sposób ich poprawnego użycia. Podobnie wywołane z opcją `-v` podają swój numer wersji. Spróbujmy utworzyć nasz program, działający na podobnych zasadach.

Ćwiczenie. Odczytywanie opcji przekazanych w wierszu poleceń

Dodamy opcje wyświetlające pomoc i numer wersji do programu numerującego wiersze, który napisaliśmy w poprzednim rozdziale. Zauważ, iż ten przykład używa operatora `our` i dlatego też będzie działać jedynie z wersją Perla o numerze 5.6 lub wyższym.

```
#!/usr/bin/perl -s
# nl3.plx
use warnings;
use strict;
my $lineno;
my $current = "";
our ($v,$h);
if (defined $v) {
  print "$0 - program numerujący wiersze, wersja 3\n";
  exit;
}
if (defined $h) {
  print   <<EOF;
$0 -  Program numerujący wiersze w pliku

Sposób użycia: $0 [-h|-v] [nazwapliku nazwapliku...]
Ten program wyświetla na standardowym wyjściu każdy wiersz pliku z dodanym numerem
➥wiersza.
EOF
  exit;
}
while (<>) {
  if ($current ne $ARGV) {
  $current = $ARGV;
  print "\n\t\tPlik:  $ARGV\n\n";
  $lineno=1;
}
print $lineno++;
print ":  $_";
}
```

Jeżeli przekażemy teraz opcję `-h`, nie będzie ona potraktowana jako nazwa pliku, lecz raczej jako prośba o pomoc.

```
>perl -s nl3.plx -h
nl3.plx -  Program numerujący wiersze w pliku

Sposób użycia: $0 [-h|-v] [nazwapliku nazwapliku...]
Ten program wyświetla na standardowym wyjściu każdy wiersz pliku z dodanym numerem
➥wiersza.
>
```

Jeśli używasz systemu operacyjnego, który pozwala Ci na bezpośrednie wywołanie programów napisanych w Perlu, pierwsza linia naszego programu zatroszczy się o wywołanie go z opcją `-s`. Nie będziesz zatem musiał powtarzać jej w wierszu poleceń.

W przypadku systemu Unix wywołamy więc program w następujący sposób:

```
>nl3.plx -v
nl3.plx - program numerujący wiersze, wersja 3
```

zaś w przypadku Windows najprawdopodobniej będziemy zmuszeni napisać:

```
>perl -s nl3.plx -v
nl3.plx - program numerujący wiersze, wersja 3
```

Jak to działa

Opcja `-s` w pierwszym wierszu programu lub w wierszu poleceń informuje Perl o tym, że każda opcja umieszczona po nazwie programu będzie wymagać zdefiniowania zmiennej o tej samej nazwie. Napisanie następującego wywołania w wierszu poleceń:

```
>perl -s cokolwiek -v -abc
```

spowoduje utworzenie zmiennych `$v` oraz `$abc`. Musimy być zatem przygotowani na ich odczytanie wewnątrz programu; w innym przypadku używając dyrektywy `strict` otrzymamy ostrzeżenie o błędzie. Dlatego też umieścimy w naszym programie wpis:

```
our ($v,$h)
```

Jeśli zmienne są już zdefiniowane, możemy coś z nimi zrobić:

```
if (defined $v) {
  print "$0 - program numerujący wiersze, wersja 3\n";
  exit;
}
```

Zmienna `$0` zawiera nazwę właśnie wykonywanego programu. Dobrą praktyką jest umieszczanie jej w każdym komunikacie informującym o programie.

Chociaż opcja `-s` jest bardzo poręczna w przypadku małych programów, w dużych dwie rzeczy czynią ją niezbyt funkcjonalną. Oto one:

- Nie masz kontroli nad tym, które opcje powinny być rozpoznawane. Perl utworzy zmienną dla każdej opcji, niezależnie od tego, czy tego chcesz, czy nie. Jeśli jej nie używasz, spowoduje to wyświetlenie ostrzeżenia o błędzie.

  ```
  >perl -s nl3.plx -v -foobar
  Name "main::foobar" used only once: possible typo.
  nl3.plx - program numerujący wiersze, wersja 3
  >
  ```

- Zapis `-abc` potraktowany zostanie raczej jako jedna opcja, a nie jako trzy: `-a`, `-b` oraz `-c`. Spowoduje zatem utworzenie jednej zmiennej: `$abc`.

Z tych powodów zalecanie jest stosowanie zamiast opisanej opcji standardowych modułów `Getopt::Std` oraz `Getopt::Long`. Standardowe moduły Perla zostały opisane w załączniku D. W celu uzyskania szczegółowych informacji możesz sięgnąć do strony `perlmod` w programie `man`.

–I oraz @INC

Perl domyślnie szuka wszystkich modułów oraz plików dołączonych poprzez użycie `do` lub `require`, w miejscu zdefiniowanym w specjalnej zmiennej — tablicy `@INC`. Możesz dodać do niej własne katalogi, używając w wierszu poleceń opcji `-I`.

Wywołanie:

```
perl -I/private/perl program
```

spowoduje przeszukanie przez Perl wszystkich katalogów zdefiniowanych w tablicy @INC oraz dodatkowo katalogu /private/perl, w celu odnalezienia potrzebnych plików. Aby uzyskać dodatkowe informacje na temat wykorzystania tablicy @INC, spójrz do następnego rozdziału.

–a oraz –F

Jednym z następców Perla w systemie Unix jest program awk. Ma on jedną wielką zaletę. Podczas czytania danych rozdzielonych znakami tabulacji może automatycznie przydzielić każdą kolumnę do oddzielnej zmiennej. Kod wykonujący podobną rzecz w Perlu korzystać będzie z tablicy; oto on:

```
while (<>) {
My @array = split;
...
}
```

Opcja -a użyta razem z -n oraz -p działa identycznie. Rozdziela dane do tablicy o nazwie @F, tak więc wywołanie:

```
>perl -an '...'
```

jest równoważne z fragmentem programu:

```
LINE: while (defined($_ = <ARGV>)) {\
  @F = split;
  'Tutaj twój kod'
}
```

W celu pobrania pierwszego wyrazu z każdego wiersza mógłbyś zatem napisać:

```
>perl -ane 'print $F[0],"\n"' chapter9.txt
Uruchamianie
Poznaliśmy
Będziesz
Każdy
...
>
```

Domyślnie opcja -a powoduje podział w miejscu wystąpienia spacji, możesz jednakże zmienić ten znak na dowolny inny, poprzez użycie opcji -F z następującym po niej żądanym znakiem. Na przykład pola w pliku z hasłami systemu Unix rozdzielone są dwukropkiem (jak widzieliśmy to w rozdziale 5.). Odwołując się do piątego elementu tablicy, możemy wydobyć katalog domowy. Jeśli twój plik passwd zawiera wiersz:

```
Simon:x:10018:10020::/home/simon:/bin/bash
```

w wyniku otrzymamy:

```
>perl -F: -ane 'print $F[5],"\n" if /^simon/' passwd
/home/simon
>
```

-l oraz -0

Konieczność dodawania sekwencji "\n" na końcu każdego wyświetlanego wiersza w celu przejścia do nowego może być bardzo denerwująca, szczególnie jeśli korzystamy z wiersza poleceń. Opcja -l spowoduje identyczne traktowanie separatora rekordu wyjściowego ($\) oraz wejściowego ($/). Pierwszy jest dodawany automatycznie po każdej instrukcji print. Natomiast — ponieważ drugim jest zazwyczaj znak nowego wiersza, czyli \n — użycie opcji -l spowoduje automatyczne dodawanie go do każdego wyświetlanego wiersza. Dodatkowo użycie opcji -n lub -p będzie identyczne z zastosowaniem chomp na danych wejściowych. Znów możemy uprościć nasz program:

```
>perl -F: -lane 'print $F[5] if /^simon/' passwd
/home/simon
>
```

Niekiedy występuje konieczność użycia jako separatora rekordu wejściowego innego niż \n znaku. W takim wypadku możemy bezpośrednio po opcji -l podać jego kod ASCII (zobacz załącznik F) jako liczby, zapisanej w kodzie ósemkowym.

Używając opcji -0 możesz w podobny sposób zmienić również separator rekordu wejściowego. Na przykład zapis -0100 spowoduje w rzeczywistości wykonanie na początku programu przypisania $/="A";. Wywołanie programu z opcją -0 bez argumentu lub z błędną liczbą ósemkową spowoduje, iż $/ pozostanie niezdefiniowany, w konsekwencji czego cały plik zostanie wczytany od razu.

> Możesz również umieścić opcję -l w pierwszym wierszu programu; nie jest to jednak zbyt dobry pomysł. Wielu ludzi z pewnością jej nie zauważy i będzie zastanawiać się, skąd biorą się nowe wiersze. Opcja ta może również stać się problemem dla Ciebie samego, jeśli będziesz chciał użyć instrukcji print bez przechodzenia do następnego wiersza.

-T

Jeśli pracujesz na danych, które pochodzą z niezbyt zaufanego źródła, będziesz z pewnością chciał zachować ostrożność podczas ich przetwarzania. Kiedy na przykład prosisz użytkownika o nazwę pliku do otwarcia i przekazujesz ją bezpośrednio do open, umożliwiasz mu wykonanie potencjalnie niebezpiecznych operacji. Możesz zamiast nazwy pliku otrzymać np. łańcuch wejściowy rm -rf /| i użyć jej w takiej postaci (**NIE RÓB TEGO!**). Chwilę później mógłbyś stwierdzić, iż niektóre pliki zniknęły z Twojego dysku. W celu zmuszenia Cię do wcześniejszego sprawdzenia danych Perl oferuje opcję -T, włączającą tryb bezpieczeństwa. Kiedy jest on włączony, Perl nie pozwala na wykorzystanie danych w potencjalnie niebezpiecznych operacjach. Co więcej, podobnie traktowana jest każda dana wywodząca się z takiej „niebezpiecznej" danej. Jedynym sposobem na użycie takich danych jest wykonanie na nich wyrażenia regularnego:

```
$tained  = ~/([\w.]+)/;
$untained =$1;
```

Przyjrzymy się temu bliżej w rozdziale 13.

Techniki wykrywania błędów

Wcześniej w tym rozdziale przyjrzeliśmy się błędom, które mogą zostać łatwo wykryte przez Perl — błędom, które pojawiają się, gdy napiszesz coś, co nie ma sensu. Wiele razy jednak napiszesz coś, co jest poprawne, jednak nie działa w oczekiwany sposób. Ponieważ nie ma jednego sposobu na rozwiązanie problemu, można polecić kilka technik, których możesz użyć w celu jego wyśledzenia. Perl zawiera kilka narzędzi, które mogą Ci w tym pomóc.

Zanim uruchomimy Debuggera

Zanim wyjaśnię, jak działa ten program, muszę wyjawić, iż jestem programistą starego stylu i niezbyt wierzę w pomoc tego typu programów. Ludzie traktują je jako panaceum, zastępujące dokładne zrozumienie problemu — uważają, że wystarczy zwyczajnie uruchomić program w debuggerze, a on odkryje błąd. Chociaż byłoby to cudowne, w rzeczywistości nie ma nigdy miejsca. Debugger może Ci jedynie pomóc, ale jest jeszcze kilka innych sposobów wykrywania błędów, które mogą być znacznie bardziej efektywne niż on.

Komunikaty diagnostyczne

Jest takie stare programistyczne zalecenie: „Kiedy nie wiesz, co się stanie, wyświetl wszystko na ekranie". Czy jesteś pewien, że dane wchodzące do programu, są takie jak myślisz? Wyświetl je! Czy jesteś pewien, że wyrażenie regularne zmieniło zawartość zmiennej we właściwy sposób? Wyświetl ją przed i po zmianie. Czy wiesz, ile razy Perl wykonał daną pętlę? Zajmuje mu to zdecydowanie zbyt wiele czasu? Zwyczajnie wyświetl krótki komunikat, który powie Ci, w którym miejscu programu się znajdujesz. Instrukcja `print` jest jak dotąd najbardziej potężnym i użytecznym narzędziem do wykrywania błędów, znajdującym się w Twojej dyspozycji!

Minimalizacja ilości kodu

Jeśli nie jesteś pewien, gdzie pojawia się błąd, spróbuj go wyizolować. Usuń lub otocz znakiem komentarza wszystkie niezwiązane wiersze programu. Sprawdź, czy problem wciąż występuje. Staraj się postępować tak aż do chwili, gdy problem zniknie, a następnie spójrz na to, co zmieniłeś.

Podobna technika może być zastosowana, jeśli program zachowuje się w nieprzewidywalny sposób. Znacznie łatwiej znaleźć błąd w pięciu wierszach programu niż w pięćdziesięciu. Możesz zatem spróbować stworzyć zupełnie nowy program, zawierający jedynie tę część logiki starego programu, która sprawia kłopoty i sprawdzić, czy potrafisz znaleźć w nim cokolwiek dziwnego. Takie postępowanie pozwoli również na sprawdzenie, czy nie ma czegoś złego w danych przekazywanych do programu.

W każdym razie, im mniejszy fragment kodu możesz przetestować, tym lepiej — szczególnie jeśli zamierzasz poprosić o pomoc kogoś innego. Im mniejszy jest stóg siana, tym większą masz szansę na znalezienie w nim igły. Co więcej, jeśli będziesz w stanie zademonstrować

swój problem w dwóch wierszach, wielu ludzi może zechcieć Ci pomóc; na pewno będzie ich znacznie mniej, jeśli będziesz oczekiwać od nich przeanalizowania całego programu.

Opiszemy teraz jeszcze kilka innych problemów, które mogą spowodować dziwne zachowanie programu bez powodowania błędu.

Kontekst

A może masz problem z właściwym kontekstem użycia? Zawsze upewnij się, czego oczekujesz od funkcji — czy chcesz, aby zwróciła tablicę, czy wielkość skalarną? Zapewnij, aby wynik był odbierany we właściwy sposób i przypisywany do zmiennej odpowiedniego typu.

Zakres

A może próbujesz użyć zmiennej zadeklarowanej za pomocą `my` poza zakresem jej widoczności? Pamiętaj że zadeklarowanie zmiennej za pomocą `my` wewnątrz pętli lub bloku oznacza, iż nie będziesz w stanie odczytać jej wartości na zewnątrz.

Priorytet operacji

Czy napisałeś coś podobnego do `print (2+3)*5`? Kod ten powinien spowodować dodanie 2 do 3, wyświetlenie wyniku i pomnożenie go przez 5. A może zapomniałeś użyć w nim nawiasów? Większość tego typu błędów możesz wykryć dzięki użyciu dyrektywy `warnings`, lecz bądź ostrożny. Zawsze, kiedy masz wątpliwości, użyj większej ilości nawiasów niż potrzebujesz w rzeczywistości.

Korzystanie z Debuggera

W przypadku Perla debugger nie jest oddzielnym programem, lecz specjalnym trybem pracy programu `perl`. W celu włączenia debuggera użyj opcji `-d`. Ponieważ jest to szczególny tryb uruchamiania programu, nie będziesz mógł nigdzie się dostać, zanim Twój program nie skompiluje się poprawnie. Debugger może pomóc Ci w śledzeniu przebiegu wykonywania programu oraz sprawdzaniu wartości zmiennych w różnych momentach jego działania.

W momencie uruchomienia debuggera powinieneś ujrzeć:

```
>perl -d nl3.plx
Default die handler restored.

Loading DB routines from perl5db.pl version 1.07
Editor support available.
Enter h or `h h' for help or `perldoc perldebug' for more help
Main::nl3.plx:6):       my $lineno;
 DB<1>
```

Wiersz `'DB<1>'` jest znakiem zachęty debuggera. Oto lista czynności, jakie możesz wykonać w tym momencie:

Komenda	Opis
`T`	Wyświetla „historię wywołania” wszystkich podprogramów, które perl w danej chwili wykonuje. Pozwala Ci dowiedzieć się, w jaki sposób dostałeś się w dane miejsce programu.
`s`	Jeśli wykonujesz program krokowo, przechodzi do następnego wiersza programu.
`n`	Powoduje wykonanie podprogramu. Zatrzymuje wykonanie zaraz po zwróceniu kontroli.
`Return`	Powtarza ostatnią komendę dotyczącą krokowego wykonania programu.
`r`	Wykonuje instrukcje aż do powrotu z aktualnie wykonywanego podprogramu.
`c`	Kontynuuje wykonanie programu aż do wystąpienia zdarzenia zatrzymującego debuggera.
`l`	Wyświetla kilka kolejnych wierszy do wykonania.
`-`	Wyświetla poprzednio wykonane wiersze.
`w`	Wyświetla wiersze otaczające aktualnie przetwarzany.
`/wzór/`	Przeszukuje kod programu aż do wystąpienia podanego wzoru.
`t`	Włącza oraz wyłącza tryb śledzenia. Powoduje on wyświetlenie każdej instrukcji przed wykonaniem.
`b`	Ustawia pułapkę. Zatrzymuje wykonywanie programu i przekazuje kontrolę do debuggera w momencie osiągnięcia wiersza o danym numerze lub spełnienia danego warunku.
`x`	Wyznacza dowolną wartość i zwraca wynikową strukturę danych w postaci drzewa.
`!`	Ponownie wywołuje poprzednią komendę.
`p`	Powoduje wyświetlenie argumentu.
`h`	Wyświetla dokładniejszą pomoc.

Nie będziemy się więcej przyglądać debuggerowi. Będzie on przydatny dopiero wtedy, gdy zaczniesz tworzyć w Perlu poważniejsze programy. Na razie lepszym wyjściem będzie zdobycie doświadczenia w testowaniu kodu i wykrywaniu błędów za pomocą wskazówek i technik pokazanych w dalszej części tego rozdziału. W ten sposób rzeczywiście poznasz, jak działa Perl.

Poprawne programowanie

Zdecydowanie najlepszym sposobem na wykrywanie błędów jest uniknięcie konieczności ich szukania. Ponieważ nigdy nie ma gwarancji na to, że Twój program nie będzie zawierać błędów, istnieje kilka rzeczy, które pomogą Ci zminimalizować ich liczbę i spowodować, że będą łatwe do zlokalizowania.

Strategia

Przed napisaniem kolejnego wiersza upewnij się, że masz plan. Aby efektywnie wykrywać błędy, musisz używać takiej samej metodologii, jakiej używałeś podczas pisania kodu. Pamiętaj o następujących punktach:

- Nie próbuj nigdy pisać dużego programu bez wcześniejszego wypróbowania jego mniejszych fragmentów. Podziel zadania na mniejsze części, które mogą być przetestowane po napisaniu.
- Spróbuj wyśledzić pierwszy błąd, a po jego usunięciu uruchom program ponownie — kolejne mogą być konsekwencją pierwszego.
- Po naprawieniu pierwszego błędu wypatruj dodatkowych, które mogły być ukryte.

Sprawdź poprawność zwracanych wartości

Nie można usprawiedliwić zaniedbywania sprawdzania wartości zwracanych przez dowolny operator, jeśli wpływa on znacząco na działanie systemu. Dzięki zwracanej wartości możesz sprawdzić, czy działa on poprawnie, zatem zawsze wykorzystuj tę możliwość. Co więcej, możesz zapobiec ewentualnym problemem poprzez sprawdzenie, co mogłoby pójść źle. Taki sposób programowania przedstawiony był w rozdziale 6., w którym sprawdzaliśmy, czy możemy czytać dane z plików oraz zapisywać do nich.

Bądź przygotowany na rzeczy niemożliwe

Czasem coś dzieje się tak, jak nie powinno. Dane mogą zostać pomieszane lub wymazane przez fragment kodu, w sposób, którego nie jesteś w stanie wyjaśnić. Aby jak najszybciej naprawić błąd, sprawdź, czy nie wystąpiła jedna z takich sytuacji. Jeśli wiesz na przykład, że jakaś zmienna powinna zawierać jedynie wartość 1, 2 lub 3, spróbuj wykonać następujące sprawdzenie:

```
if ($var == 1) {
# Wykonaj pierwszą czynność
} elseif ($var == 2) {
# a teraz drugą
} elseif ($var == 3) {
# i trzecią
} else {
  die "Hej! To nie powinno się wyświetlić!\n";
}
```

Jeśli będziesz mieć odrobinę szczęścia, nie powinieneś ujrzeć żadnego komunikatu. Jeśli jednak zostanie wyświetlone jakieś ostrzeżenie, będziesz wiedzieć, że zawartość zmiennej została wcześniej wymazana. Jest to pułapka, która należy do tak zwanych **pułapek asercyjnych**[5] lub — mówiąc mniej formalnie — **błędów, które nie mogą się wydarzyć**. Eric Raymond mówi o nich:

> „Błędy tego rodzaju są prawdziwą rzadkością w kodzie produkcyjnym. Jednak programiści mądrzy na tyle, aby z przyzwyczajenia sprawdzać ich wystąpienie, bywają nierzadko zaskoczeni tym, jak często mogą się one pojawiać podczas pisania kodu. Wiedzą również, że ich szukanie może często przyprawić o silny ból głowy".

[5] *Assertions* nie ma jednoznacznego odpowiednika w jęz. polskim — *przyp. tłum.*

Nigdy nie ufaj użytkownikowi

Użytkownicy są „niezawodnym" źródłem błędnych danych. Nie pozwól im stać się przyczyną błędów. Upewnij się, iż dostajesz ten rodzaj danych, którego oczekujesz. Czy czekasz na podanie znaku końca linii? Czy jesteś przygotowany na tekst wprowadzony przy użyciu wielkich, czy małych liter, a może mieszanych? Być może wreszcie nie martwisz się o to? Jeśli tak, to zacznij się martwić i staraj się być elastyczny, wszędzie, gdzie to możliwe, ponieważ jest bardziej niż prawdopodobne, iż użytkownik nie zrozumie dokładnie, o co Ci chodziło. Przede wszystkim, zanim zaczniesz używać wprowadzonych danych, upewnij się, czy są one poprawne.

Istnienie danych

Czy dane, które wpisujesz do tablicy, powinny wcześniej istnieć? A może nie? Sprawdź, czy nie wymazujesz danych, na których zachowaniu Ci zależy, a jeśli tak, to zadaj sobie pytanie, jak doszło do takiej sytuacji. Czy jesteś pewien, że otrzymałeś dane do zapisania? Sprawdź, czy umieszczasz je we właściwym miejscu. Czy jesteś pewien, że cokolwiek istnieje w zmiennej, której zawartość chcesz odczytać? Upewnij się, że dane są obecne w tablicy, do której się odwołujesz.

Dodaj pomocne komentarze

Komentarze są bardzo przydatnym środkiem do śledzenia tego, co dzieje się w programie. Używaj ich zatem we właściwy sposób. Komentarze, które wyjaśniają przepływ danych a także to, co one oznaczają i skąd pochodzą, są znacznie bardziej pomocne od tych, które objaśniają, co robisz w danej chwili. Porównaj użyteczność tych dwóch komentarzy:

```
$a = 6.28318; # przypisz 6.28318 do $a
$a = 6.28318; # pi*2
```

Niedogodnością związaną z komentarzami jest konieczność ich uaktualniania w momencie zmiany kodu programu. Upewnij się, czy nie wprowadzają Cię one w błąd (może się tak zdarzyć w najgorszym przypadku). Stare powiedzenie mówi: „Jeśli kod programu i komentarze nie zgadzają się, najprawdopodobniej zarówno jedno, jak i drugie jest złe".

Utrzymuj porządek w kodzie programu

Uporządkowany kod programu jest znacznie prostszy do zrozumienia i wykrycia błędów od tego, w którym jest bałagan. Ułatwisz sobie późniejsze znajdowanie problemu, jeśli będziesz zawsze przestrzegać następujących reguł:

- umieszczaj analogiczne fragmenty kodu w kolumnach,
- regularnie stosuj odpowiednie wcięcia,
- w każdym wierszu umieszczaj jedną instrukcję,
- rozdzielaj długie instrukcje pomiędzy większą liczbę wierszy,
- używaj znaków spacji i tabulacji w celu zwiększenia przejrzystości kodu.

Porównaj następujący fragment programu:

```
while (<>) {
if ( /^From:\s+(.*)/    ) {  $from    =  $1  }
if ( /^Subject:\s+(.*)/ ) {  $subject =  $1  }
if ( /^Date:\s+(.*)/    ) {  $date    =  $1  }

print "Poczta od $from z dnia $date dotyczaca $subject\n"
  unless /\S+/;

next until /^From/;
}
```

z tym samym programem, zapisanym w inny sposób:

```
while (<>) {if ( /^From:\s+(.*)/) {  $from=$1  }
if ( /^Subject:\s+(.*)/) { $subject=$1  }
if ( /^Date:\s+(.*)/) { $date=$1  }
print "Poczta od $from z dnia $date dotyczaca $subject\n"unless /\S+/;
next until /^From/;}
```

W którym z nich chciałbyś szukać błędów?

Ćwiczenia

Przyjrzyj się pokazanemu tu plikowi, zastosuj do niego wszystko to, o czym tu czytałeś i sprawdź, czy będziesz w stanie nadać mu właściwą formę.

```
#!/usr/bin/perl
#buggy.plx

my %hash;
until (/^q/i) {

print " Co chciałbyś zrobić? (naciśnij 'o' aby wyświetlić opcje):"
$_  =  STDIN;

if ($ eq "o") {options}elseif($ eq "r"){read}elseif($ eq "l") { list }elseif ($_ eq
➥"w"){ write }elseif ($_ eq "d") { delete } elseif ($_  eq "x") { clear } else {
➥print "Przykro mi, lecz nie rozpoznałem opcji.\n";}

sub options {
  print<<EOF
Dostępne opcje:
o - wyświetla opcje
r - czytanie wpisu
l - wyświetlenie wszystkich wpisów
w - zapisanie wpisu
d - usunięcie wpisu
x - usunięcie wszystkich wpisów
EOF;
}

sub read {
my $keyname  =  getkey();
```

```
if (exists $hash{"$keyname"}) {
print "Element '$keyname' ma wartość $hash{$keyname}";
} else {
print " Przykro mi, lecz ten element nie istnieje.\n"}}

sub list {foreach (sort keys(%hash))  {print  "$  => $hash{$ }\n";}}

sub write {
my $keyname  = getkey();
my $keyval  = getvalue();
if (exists $hash{$keyname}) { print "Przykro mi, lecz ten element już istnieje.\n"
}else {$hash{$keyname}=$keyval;}}

sub delete {
my $keyname  =  getkey();
if (exists $hash{$keyname})  {
print "Usuwam wpis $keyname. \n";
delete $hash{$keyname};}}

sub clear {undef %hash;}
sub getkey {print " Podaj nazwę klucza dla elementu: "; chomp($_  =  <STDIN>);)
sub getval {print " Podaj nazwę wartości dla elementu: "; chomp($_  =  <STDIN>);}
```