Perl w zarządzaniu witrynami WWW

Autor: John Callender
Tłumaczenie: Wojciech Derechowski
ISBN: 83-7197-676-3
Tytuł oryginału: Perl for Web Site Management
Format: B5, stron: 477

Książka mówi o tym, jak osiągać wyniki. Ściślej, mówi o tym, jak osiągać je szybciej, skuteczniej i z większą przyjemnością, zwłaszcza, gdy mamy pracę, która polega na tworzeniu i utrzymaniu użytecznych kolekcji informacji World Wide Web. Co ważniejsze jednak, książka pokazuje, jak dokonać tych rzeczy przy użyciu języka programowania Perl.

W najbardziej ogólnym sensie książka dotyczy tego, jak opuścić świat użytkowników i wyruszyć do nowego świata programistów komputerowych. Droga, która tam wiedzie, składa się z wielu kolejnych etapów lecz niniejsza książka nie ma zamiaru prowadzić Was do samego końca. Skupia się tylko na pierwszych kilku krokach, starając się Was przeprowadzić jak najbezpieczniej przez najbardziej oczywiste pułapki. To, jak daleko dotrzecie, zależy od Waszych potrzeb i zdolności. Książka na pewno pomaga rozpocząć tę podróż.

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

Spis treści

11

Sprawdzamy odnośniki

W tym rozdziale przedstawiamy pewne ważne własności języka Perl, które będziemy wykorzystywać w dalszej części książki. Pokazujemy najpierw, jak używa się odniesień (ang. *references*) do tworzenia wielopoziomowej struktury danych. Wyjaśniamy też, jak pobrać i zainstalować moduły CPAN. Wreszcie przedstawiamy krótkie wprowadzenie do Perla zorientowanego obiektowo, aby pomóc Wam przy korzystaniu z wielu obiektowo zorientowanych modułów, które są dostępne w CPAN. Nawet jeżeli nie sądzicie, że program do sprawdzania odnośników jest Wam potrzebny, powinniście zapoznać się z pojęciami przedstawianymi w tym rozdziale.

Utrzymanie odnośników

Jedną najważniejszych przyczyn sukcesu WWW była decyzja, którą podjął Tim Barnes-Lee na samym wstępie, że odnośniki mają być jednokierunkowe. Oznacza to, że każdy może utworzyć odnośnik, który łączy pewną stronę z dowolną inną stroną, i może to zrobić z łatwością. Nie ma potrzeby rejestrowania odnośnika w żadnym centralnym repozytorium, nie wymaga się, by ustanowić i utrzymywać odwrotny odnośnik, nie trzeba nawet zawiadamiać ludzi po drugiej stronie, że odnośnik prowadzi do nich. Po prostu umieszczamy znacznik `<A HREF=...>` na stronie HTML — i mamy odnośnik. Nadal cieszy mnie, gdy mogę to pokazać komuś, kto zaczyna się uczyć HTML. Po chwili osłupienia pada zwykle pytanie „I to wszystko?”.

Taki cel wyznaczył sobie Barnes-Lee, kiedy wymyślał WWW. Chciał, żeby ludzie mogli za pomocą komputerów robić to, co z łatwością potrafią zdziałać dzięki swym umysłom: łączyć rzeczy na pozór ze sobą niezwiązane. W świecie komputerów swoboda tego rodzaju ma jednak ujemną stronę, gdyż łatwo utworzyć popsuty odnośnik, który przez literówkę lub inny błąd staje się bezużyteczny. Co gorsza, nawet jeżeli odnośniki działają na początku poprawnie, to z czasem zaczynają murszeć. Kiedy zasoby przestają istnieć lub są przenoszone pod nowy adres, strony pełne fascynujących połączeń zmieniają się w strony, w których jest pełno popsutych odnośników. Skoro sprawujemy opiekę nad naszymi zasobami, musimy przede wszystkim dbać o swoje własne odnośniki. Zadanie to jest trudne, nawet gdy mówimy o garstce odnośników w osobistej witrynie, a kiedy dotyczy witryn zawierających setki i tysiące stron, rozrasta się do gigantycznych rozmiarów.

Perl idzie nam na pomoc. W tym rozdziale przedstawiamy trzy różne wersje programu do sprawdzania odnośników. Pierwsza wersja działa bardzo szybko, ale przetwarza tylko odnośniki lokalne (czyli odnośniki do stron w lokalnym systemie plików) i jest dość prymitywna, jeśli chodzi o interpretację

HTML. Druga wersja jest wzbogacona o możliwość sprawdzania zdalnych odnośników (wskazujących poza witrynę) i nadal działa dość szybko, ale także można ją uznać za dość prymitywną. Trzecia wersja jest dużo bardziej wyrafinowana, jednak działa znacznie wolniej.

Znajdujemy pliki z File::Find

Pierwszym krokiem przy budowie pierwszego z programów do sprawdzania odnośników jest wymyślenie dla skryptu metody uzyskiwania listy wszystkich plików HTML w witrynie. Wcześniej, w rozdziale 4., podawaliśmy do skryptu listę nazw plików z wiersza poleceń, wykorzystując zdolność shella do rozwijania znaków dzikich kart. Obecnie zastosujemy inne podejście, używając standardowego modułu `File::Find`. Zastosujemy go, wstawiając `use File::Find` do skryptu i wywołując następnie funkcję `find` tego modułu. Ułatwi nam to budowę skryptu, który przetwarza wszystkie pliki w danej kartotece początkowej, włączając w to pliki w podkartotekach, które znajdują się głębiej.

Zaczniemy od prostego skryptu demonstracyjnego, `find_files.plx`, który jest pokazany w przykładzie 11.1 (jak wszystkie przykłady w niniejszej książce również i ten można pobrać z witryny książki, pod adresem *http://www.elanus.net/book/*).

Przykład 11.1. find_files.plx

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

# find_files.plx

# skrypt demonstruje użycie modułu File::Find.

use strict;
use File::Find;

my $start_dir = shift
    or die "Usage: $0 <start_dir>\n";

unless (-d $start_dir) {
    die "Start directory '$start_dir' is not a directory.\n";
}

find(\&process, $start_dir);

sub process {

    # wołana przez funkcję find w File::Find, dla każdego
    # pliku, znalezionego przez find rekurencyjnie.

    print "Found $File::Find::name\n";
}
```

Większa część skryptu powinna obecnie wyglądać jasno. Skrypt zaczyna działanie od zdjęcia pierwszego elementu z początku tablicy `@ARGV` (czyli pierwszego elementu dostarczonego jako argument, gdy skrypt był wywołany z wiersza poleceń) i wstawienia go do zmiennej skalarnej `$start_dir`.

Jeżeli `$start_dir` otrzymuje wartość fałszywą, skrypt zakłada, że użytkownik nie dostarczył nazwy początkowej kartoteki i ginie z komunikatem *sposób użycia*. Należy wyjaśnić, że *sposób użycia* jest krótkim komunikatem, który podaje użytkownikowi instrukcję, jak skrypt powinien być uruchomiony;

obecnie zawiera zmienną specjalną `$0`, która podaje nazwę uruchomionego skryptu. (Można było zaszyć "`find_files.plx`" w *sposobie użycia*, jednak teraz komunikat potrafi automatycznie dostosować się do zmian, jeżeli ten kod trafi do skryptu o innej nazwie).

Skrypt stosuje następnie operator sprawdzania pliku `-d` (o którym była mowa w rozdziale 6.), aby sprawdzić, czy `$start_dir` reprezentuje kartotekę, i ginie — znów — z komunikatem o błędzie, jeżeli tak nie jest.

Następnie mamy kluczowe miejsce skryptu:

```
find(\&process, $start_dir);
```

Funkcja `find` została wciągnięta wcześnie przez wiersz `use File::Find`. Jak wyjaśnia dokumentacja `File::Find` (dostępna przez `man File::Find` lub `perldoc File::Find`), funkcja `find` ma co najmniej dwa argumenty, z których drugi (i każdy następny) jest nazwą kartoteki, gdzie chcemy zacząć poszukiwanie plików. Proces znajdowania plików odbywa się *rekurencyjnie*, co znaczy po prostu, że funkcja `find` „przekopuje się" przez podkartoteki, które leżą coraz głębiej (lub zawierają się — zależnie od punku widzenia) w kartotece początkowej.

Magia odniesień

To tyle, jeżeli chodzi o drugi argument funkcji `find`. A co z pierwszym argumentem? To tu zaczyna się świat magii.

Tak, czas na więcej magii. Przyjrzyjcie się uważnie pierwszemu argumentowi funkcji `find`: `\&process`. Wygląda trochę jak wywołanie procedury, ze znakiem ampersand (`&`) na początku `&process`, ale co tam robi wiodący znak *backslash* (`\`)?

Znak *backslash* tworzy odniesienie (ang. *reference*). Ściślej mówiąc, tworzy odniesienie do procedury `&process`. Dodajmy, że odniesienia są kluczem do wielu bardzo użytecznych i ważnych własności języka Perl. Nie musieliśmy przejmować się nimi dotychczas, ale skoro `File::Find` używa odniesień, to zapewne nadszedł czas, żeby o nich pomówić. Nieco później w obecnym rozdziale trzeba by, tak czy inaczej, zapoznać się z odniesieniami, ponieważ będziemy je stosować przy tworzeniu naprawdę ciekawej, wielopoziomowej struktury danych.

Na razie postarajmy się zapamiętać tylko tyle, że odniesienie jest specjalnym sposobem odwoływania się (i uzyskiwania dostępu) do pewnej innej rzeczy, o której wie Perl. Tą rzeczą może być wartość umieszczona w zmiennej skalarnej lub lista wartości znajdująca się w tablicy, lub lista par klucz-wartość zawarta w hashu, lub (jak obecnie) kod umieszczony w procedurze. Odniesienie jest użyteczne dzięki temu, że zawsze jest *skalarem* (czymś *pojedynczym*), nawet jeżeli rzecz, do której się odnosi, taka nie jest. Oznacza to, że na odniesieniach można wykonywać operacje skalarne: przechowywać odniesienia jako elementy w tablicy lub jako wartości w hashu, czy — podobnie jak w tym przypadku — przekazywać odniesienia jako argumenty do funkcji.

Odniesienia są niezwykle użyteczne w Perlu, ale żeby je wykorzystywać, musimy nauczyć się nowej składni języka Perl. Jak Mark-Jason Dominus wyjaśnia na swojej świetnej stronie podręcznikowej `perlreftut`, są dokładnie cztery nowe części składni Perla, których trzeba się nauczyć, aby używać odniesień: dwa sposoby tworzenia odniesień i dwa sposoby rozbioru odniesień (ang. *dereference*), by uzyskać dostęp do pierwotnej rzeczy, do której się odnoszą.

Znacie już pierwszą spośród tych nowych części składni odniesień: użycie znaku *backslash* przed zmienną lub nazwą procedury, aby zwrócone zostało odniesienie do zawartości tej zmiennej (lub procedury). Dwóch spośród pozostałych trzech typów składni odniesień, nauczycie się później w tym rozdziale. (Aby zapoznać się z czwartym, będziecie musieli zaczekać aż do rozdziału 14.).

Może wydawać się, że mówię teraz o czymś, co wcale nie jest takie ważne lub ciekawe. W porządku, wolno Wam tak myśleć. Zapamiętajcie tylko, co dzieje się w pierwszym argumencie funkcji `find`: przekazujemy *odniesienie* do procedury `&process`. Jest to zupełnie co innego, niż użycie `&process` (to jest — wywołania procedury) jako pierwszego argumentu tej funkcji. To spowodowałoby wartościowanie procedury ze względu na zwracaną wartość (lub wartości) tej procedury — wartość przekazywaną z kolei do funkcji `find`. Przekazanie odniesienia sprawia natomiast, że dla funkcji `find` dostępna jest sama procedura.

Czego funkcja `find` może chcieć od procedury? Jak wyjaśnia dokumentacja `File::Find`, funkcja `find` uruchomi tę procedurę jeden raz dla każdego pliku, który rekurencyjnie odszuka. Procedura będzie wywołana dla każdego odszukanego pliku i otrzyma nazwę bieżącego pliku przez przekazanie w specjalnej zmiennej `$_`. Procedura będzie także mieć dostęp do *zmiennych pakietowych*: `$File::Find::dir`, która podaje nazwę bieżąco przetwarzanej kartoteki, oraz `$File::Find::name`, która jest bezwzględną nazwą ścieżki przetwarzanego pliku. Można właściwie powiedzieć, że `$File::Find::name` oznacza to samo, co `$File::Find::dir/$_` — przynajmniej w systemach, w których separatorem ścieżki jest znak *slash*.

Pozostawia nam to swobodę definiowana procedury, jak nam się podoba. W tym skrypcie procedura `&process` jest napisana tak, żeby po prostu wydrukować bezwzględną ścieżkę każdego pliku, który przetwarza. Jeżeli wykonamy ten skrypt, wskazując odpowiednią nazwę kartoteki jako argument, powinniśmy zobaczyć coś w tym rodzaju:

```
[jbc@andros ora]$ find_files.plx /home/jbc/ora
found /home/jbc/ora
found /home/jbc/ora/find_files.plx
found /home/jbc/ora/walnuts
found /home/jbc/ora/walnuts/rutabagas
found /home/jbc/ora/walnuts/apples.html
found /home/jbc/ora/walnuts/oranges.txt
```

Jedną z ciekawszych rzeczy, które ten wynik ujawnia, jest fakt, że dla systemu Unix kartoteka jest jeszcze jednym plikiem. Zarówno kartoteka `ora`, jak i zawarta w niej kartoteka `walnuts` występują na liście jako znalezione pliki.

Znajdujemy tylko pliki HTML

Powiedzmy, że życzymy sobie, aby moduł `File::Find` ignorował wszystkie pliki, oprócz tych, które kończą się na `.html`. Nic łatwiejszego: po prostu użyjemy wyrażenia regularnego, żeby wyskoczyć z procedury `&process`, jeżeli bieżąca nazwa pliku nie kończy się w ten sposób. Można to zrobić, wstawiając na początku procedury następujący wiersz:

```
return unless /\.html$/;
```

Ten wiersz spowoduje, że procedura odda sterowanie natychmiast, jeżeli nazwa pliku (która — jak pamiętacie — jest umieszczana przez `File::Find` w `$_` dla każdego wywołania procedury) nie kończy się na `.html`.

Zauważmy, jak ten wiersz korzysta z faktu, że `$_` jest domyślną zmienną, na której wyrażenia regularne próbują swojego dopasowania. Wskutek tego wiersz jest bardzo zwięzły i jasny (przy założeniu, że wiemy o `$_`). Nie ulega wątpliwości, że jest to bardzo w stylu Perla.

Zauważmy również, że wewnątrz wyrażenia regularnego `$_` służy uniknięciu kropki (`.`), przez co nie pozwala, by została zinterpretowana jako *metaznak*, który dopasowuje „dowolny pojedynczy znak". Wreszcie zwróćmy uwagę na znak `$` na końcu wyrażenia, który je zakotwicza na końcu łańcucha, w jakim jest dopasowywane, więc `walnuts.html.bak` nie będzie pasować. Czyż wyrażenia regularne nie są wspaniałe?

Zmieniona procedura powinna wyglądać tak:

```
sub process {

    # wołana przez funkcję find w File::Find, dla każdego
    # pliku, znalezionego przez find rekurencyjnie.

    return unless /\.html$/;
    print "Found $File::Find::name\n";
}
```

Jeżeli teraz wykonamy skrypt, powinniśmy dostać następującą informację wyjścia:

```
[jbc@andros ora]$ find_files.plx /home/jbc/ora
found /home/jbc/ora/walnuts/apples.html
```

Mamy więc skrypt, który będzie rekurencyjnie kroczyć w dół systemu plików, przetwarzając wszystkie znalezione pliki, których nazwy kończą się na `.html`.

Szukamy odnośników

Użyjmy teraz `file_find.plx` jako punktu wyjścia dla nowego skryptu, `link_check.plx`, który w dość prosty sposób będzie sprawdzać popsute odnośniki w plikach HTML, jakie przetwarza. Pierwszy krok będzie polegać na zmianie procedury `&process` tak, by zamiast drukować nazwy plików HTML, które przetwarza, otwierała każdy z nich i czytała jego treść. Możemy to uzyskać, zmieniając procedurę `process` w następujący sposób:

```
sub process {

    # wołana przez funkcję find w File::Find, dla każdego
    # pliku, znalezionego przez find rekurencyjnie.

    return unless /\.html$/;
    my $file = $File::Find::name;
    unless (open IN, $file) {
        warn "can't open $file for reading: $!, continuing...\n";
        return;
    }
    my $data = join '', <IN>; # wszystkie dane na raz
    close IN;
    return unless $data;
    print "found $file, read the following data:\n\n$data\n";
}
```

Przeglądając nowe wiersze, widzimy, że *zmienna pakietowa* `$File::Find::name` jest przypisywana do zmiennej `my` o nazwie `$file`. Dzieje się tak wyłącznie dla wygody. Będziemy używać tej zmiennej wielokrotnie, a łatwiej jest pisać `$file`, niż przepisywać za każdym razem `$File::Find::name`.

Otwieramy następnie plik do czytania, przyporządkowując mu uchwyt pliku `IN` w instrukcji `open`. Zauważcie, że używamy `warn` zamiast `die` po sprawdzeniu, że operacja otwarcia pliku jest niepomyślna. Chodzi o to, że prawdopodobnie będziemy chcieli, by skrypt kontynuował przetwarzanie plików, nawet gdy stanie się coś dziwnego i któregoś z nich nie da się otworzyć do czytania.

Następnie stosujemy ciekawy trik: pobieramy jednocześnie wszystkie dane z uchwytu pliku i umieszczamy je w zmiennej skalarnej za pomocą tego wiersza:

```
my $data = join '', <IN>; # wszystkie dane na raz
```

Wcześniej zapisywaliśmy `<IN>` w nawiasach okrągłych pętli `while`, przez co `<IN>` za każdym razem zwraca pojedynczy wiersz. Zapisując `<IN>` w kontekście tablicy (który właściwie należałoby nazwać kontekstem *listy*), powodujemy, że `<IN>` zwraca wszystkie swoje wiersze jednocześnie.

W rozdziale 10. uzyskaliśmy to, przypisując do zmiennej tablicowej wartość, którą zwraca `<UCHWYT-PLIKU>` w taki sposób:

```
@walnuts = <IN>;
```

W obecnym przypadku można było użyć dwóch wierszy kodu — zamiast jednego:

```
my @data = <IN>;
my $data = join '', @data;
```

Jednak skoro (z definicji) drugi argument funkcji `join` jest listą, możemy po prostu napisać tam `<IN>`, nadając `<IN>` kontekst listy i powodując, że zwróci wszystkie swoje wiersze na raz, bez potrzeby użycia pośredniej tablicy `@data`. Zauważcie przy sposobności, że specyfikacja pustego łańcucha (za pomocą konstrukcji `''`) w pierwszym argumencie `join` jest zupełnie do przyjęcia. Powoduje to, że łańcuchy łączone przez `join` tłoczą się jeden za drugim, bez żadnych znaków pomiędzy nimi (chociaż będą rozgraniczane przez znaki nowego wiersza w `$data`, ponieważ `<IN>` zwraca znak nowego wiersza na zakończenie każdego z wierszy[1]).

Jedyny problem ze stosowaniem tego triku polega na tym, że kiedy plik jest bardzo duży, zużyjemy wiele pamięci, gdy całość tego pliku przypiszemy do zmiennej skalarnej w jednym wielkim kawałku. Jednak wielkość stron HTML raczej nie stwarza trudności (biorąc pod uwagę wielkość RAM dostępną we współczesnych komputerach), więc — jeżeli chcemy manipulować plikiem jako całością — jest to dogodny sposób, by go wczytać.

Gdy ta wersja skryptu działa poprawnie, otrzymujemy wydruk kompletnego tekstu każdego ze znalezionych plików HTML.

---

[1] Gdy mówimy o rozgraniczaniu, mamy na myśli *separator*; jeśli o ograniczaniu — *delimiter*; jeżeli o zakańczaniu — *terminator* — *przyp. tłum.*

Ekstrakcja

W tej chwili jesteśmy gotowi, by wejść na następny poziom: skrypt powinien wyciągnąć z tych plików jedynie odnośniki, a mówiąc ściślej — tylko wartości każdego atrybutu `SRC` lub `HREF`.

Jak mówiliśmy w rozdziale 4., *parsing* pliku HTML za pomocą prostego dopasowania wzorca jest przedsięwzięciem podatnym na błąd. Poniższy przykład nie sprawdza się wobec kilku rodzajów znaczników, które są poprawne jako HTML, lecz nie spełniają uproszczonych założeń skryptu. W sprawie „właściwego" programu do sprawdzania odnośników, przetwarzającego takie konstrukcje HTML bardziej płynnie, odwołaj się do przykładu na końcu tego rozdziału.

Zaczniemy od końca procedury `&process`, skąd usuwamy wiersz, który drukuje bieżącą nazwę pliku, `$file`, i całą zawartość zmiennej `$data`, po czym w to miejsce wstawiamy następującą porcję kodu:

```
my @targets = ($data =~ /(?:href|src)\s*=\s*"([^"]+)"/gi);
print "In file $file, found the following targets:\n";
foreach (@targets) {
    print " $_\n";
}
```

Skupmy się na pierwszym wierszu. Wiersz ten wygląda poważnie, ale przy założeniu, że odrobiliście zadania domowe z wyrażeń regularnych, nie jest szczególnie trudny.

Pierwszą rzeczą, której powinniśmy się przyjrzeć, jest sam wzorzec wyrażenia regularnego: `/(?:href|src)\s*=\s*"([^"]+)"/gi`. W porządku od lewej do prawej ten wzorzec mówi, że należy dopasować łańcuch, który zaczyna się albo od `href`, albo od `src`, następnie ma zero lub więcej znaków odstępu, znak równości (=), zero lub więcej znaków odstępu, podwójny cudzysłów ("), jeden lub więcej znaków innych, niż podwójny cudzysłów, a następnie jeszcze jeden podwójny cudzysłów. Po zakończeniu wzorca, końcowe modyfikatory `/g` oraz `/i` powodują, że wzorzec dopasowuje się globalnie (czyli po pierwszym dopasowaniu dopasowuje się nadal) i — odpowiednio — bez rozróżniania wielkości liter.

Znak pomiędzy podwójnymi cudzysłowami jest przechwytywany do zmiennej specjalnej `$1` dzięki nawiasom okrągłym, które okalają tę część wyrażenia. Natomiast nawiasy, użyte na początku wzorca do grupowania czynników alternatywy `href|src`, zostały pozbawione roli przechwytujących nawiasów rozmyślnie — przez specjalną sekwencję `?:` umieszczoną bezpośrednio po otwierającym nawiasie okrągłym; dzięki temu ta para nawiasów może służyć do grupowania, nie wykonując żadnego przechwytywania.

Popatrzmy teraz na całość wiersza, gdzie dzieje się coś ciekawego: jest to przypisanie wartości, którą zwraca wyrażenie regularne, do zmiennej tablicowej `@targets`.[2] Przypisanie czegoś do zmiennej tablicowej powoduje, że prawa strona przypisania ulega wartościowaniu w kontekście listy. Wyrażeń regularnych używaliśmy dotychczas zwykle w kontekście skalarnym[3] (a dokładniej — w kontekście

[2] Nazwa ta znaczy tyle, co „miejsca, do których prowadzą odnośniki" — *przyp. tłum.*

[3] Por. jednak: „Monumentalne wyrażenie regularne", rozdział 8. — *przyp. tłum.*

logicznym prawdy lub fałszu, który Perl uznaje za kontekst skalarny). Inaczej mówiąc, wykonywaliśmy coś takiego:

```
if (/wzorzec/) { coś zrób; }
```

W tym wierszu wyrażenie regularne `/wzorzec/` zwraca wartość prawdziwą lub fałszywą na podstawie tego, czy dopasowało się pomyślnie. Zapisanie wyrażenia regularnego w kontekście listy powoduje jednak, że to wyrażenie zachowuje się inaczej. Wciąż wedle tych samych reguł dopasowuje się lub nie, lecz teraz zwraca wartość, która jest listą wszystkich części łańcucha, jakie pasują do tego, co znajduje się w przechwytujących nawiasach wyrażenia. (Jeżeli wyrażenie nie zawiera nawiasów przechwytujących, otrzymujemy listę wszystkich części łańcucha, które pasują do całego wyrażenia, jakby to wyrażenie zawierało przechwytujące nawiasy wokół całego wzorca).

Powtórzmy, że obecne wyrażenie regularne ma na końcu modyfikator `/g` i dopasowuje wzorzec w łańcuchu — tyle razy, ile się da. Za każdym razem, gdy wzorzec dopasowuje się w łańcuchu, część łańcucha, która pasuje we wnętrzu nawiasów przechwytujących, jest dodawana do wartości zwracanej przez wyrażenie. Gdy wyrażenie skończyło dopasowywać, otrzymujemy listę wszystkich ograniczanych przez podwójny cudzysłów wartości `HREF` i `SRC`, zawartych w łańcuchu, który jest umieszczony w zmiennej `$content`.[4]

Chodzi więc o to, żeby wydobyć wszystkie miejsca, na które wskazują znaczniki `<A HREF="...">` dokumentu HTML oraz znaczniki `<IMG SRC="...">` wyświetlanych przez ten dokument obrazów, tak że będziemy mogli sprawdzić, czy miejsca te rzeczywiście istnieją. Jednak pamiętajmy, że ten sposób nadaje się tylko dla prostych przykładów HTML.

Konwersja

Wartości atrybutów `HREF` i `SRC`, które należą do stron WWW, same nie są zbyt przydatne do sprawdzania odnośników. Aby zobaczyć, czy wskazują na pliki, które faktycznie istnieją, musimy przekształcić je na bezwzględne ścieżki systemu plików.

Ponownie będziemy tu czynić pewne (być może nietrafne) założenia, tym razem o tym, jak skonfigurowany jest serwer HTTP. W szczególności założymy, że struktura kartotek serwera WWW jest w rzeczywistości tylko prostą gałęzią większego systemu plików. Mówiąc inaczej, zakładamy, że jeżeli drzewo dokumentów serwera WWW ma na szczycie kartotekę */walnuts/rutabagas/* widzianą z shella, to ścieżki, które wskazują na zasoby lokalne z punku widzenia serwera WWW, można przekształcić na ścieżki systemu plików dzięki wstawieniu przed nimi `'/walnuts/rutabagas/'`.

Zaczniemy od dodania u góry skryptu `link_check.plx` następującej sekcji konfiguracyjnej:

```
# sekcja konfiguracyjna:

# uwaga: pierwsze cztery zmienne konfiguracyjne *nie* powinny zawierać
# końcowego znaku slash (/)

my $start_dir   = '/w1/s/socalsail/expo'; # gdzie zacząć odnajdywanie
my $hostname    = 'www.socalsail.com';    # nazwa stacji dla witryny
my $web_root    = '/w1/s/socalsail';      # root dokumentów www
my $web_path    = '/expo';                # ścieżka www do $start_dir
```

[4] Raczej: `$data` — *przyp. tłum.*

Teraz dodajemy następującą procedurę `&convert` u dołu skryptu:

```
sub convert {

    # Przyjmuje (w pierwszym argumencie) nazwę kartoteki pliku,
    # z którego wyciągnięto listę URLi, i listę URLi, wyciągniętą
    # z tego pliku (w pozostałych argumentach). Zwraca listę
    # wszystkich URLi, które nie wskazują na zewnątrz lokalnego
    # środowiska (ang. site), różnych od ftp:, mailto:, https:,
    # lub news:, gdzie te URLe są przekształcone na nazwy
    # plików lokalnego systemu plików.

    my($dir, @urls) = @_;
    my @return_urls;
    my $escaped_hostname = quotemeta $hostname;
    foreach (@urls) {
        next if /^(ftp|mailto|https|news):/i;
        if (/^http:/i) {
            # URL zaczyna się od 'http:'
            next unless /^http:\/\/$escaped_hostname/io;
            s/^http:\/\/$escaped_hostname//io;
        }
        if (/^\//) {
            # URL zaczyna się od '/'
            $_ = $web_root . $_;
        } else {
            # URL jest ścieżką względną
            $_ = $dir . '/' . $_;
        }
        s/#.*//;  # utnij końcowe #kotwiczki
        s/\?.*//; # utnij końcowe ?argumenty
        push @return_urls, $_;
    }
    @return_urls;
}
```

Jeśli chodzi o sam Perl, nie dzieje się w tej procedurze nic nowego. Podajemy do niej nazwę kartoteki, gdzie znaleźliśmy plik HTML, a potem listę wszystkich odnośników wyciągniętych z tego pliku. Dla każdego odnośnika skrypt sprawdza, czy ten odnośnik zaczyna się następująco: `ftp:`, `mailto:`, `https:` lub `news:`. Jeżeli tak jest, skrypt zapomina o tym odnośniku i przechodzi do następnego.

Przy sposobności zauważmy, jak użycie domyślnej zmiennej indeksowej (`$_`) dla pętli `foreach` pozwala nam bardzo zwięźle zapisać w procedurze różne dopasowania wyrażeń regularnych, gdyż wszystkie dopasowują się domyślnie w tej samej zmiennej `$_`.

Następnie procedura sprawdza, czy odnośnik zaczyna się od `http:`. Jeśli tak jest, procedura przejdzie do następnego odnośnika poprzez `next` (jeżeli nazwa stacji nie jest taka sama, jak nazwa w zmiennej konfiguracyjnej `$hostname`, zdefiniowanej na szczycie skryptu) lub użyje wyrażenia regularnego, by wyszukać i zastąpić łańcuch `http://$hostname` niczym.

Zanim jednak procedura będzie mogła to zrobić, musi przygotować łańcuch `$hostname` do użytku we wzorcu *regex*, wykonując na tym łańcuchu funkcję `quotemeta`. Funkcja `quotemeta` w łańcuchu, który jest do niej przekazywany, wstawia znak *backslash* przed każdym znakiem niesłowa, po czym zwraca zmodyfikowany łańcuch. („Znak niesłowa” w tym przypadku znaczy to samo, co specjalna sekwencja wyrażeń regularnych `\W` czyli dowolny znak, który nie jest wielką lub małą literą, cyfrą lub znakiem podkreślenia). Ponieważ wyrażenia regularne Perla są pomyślane tak dogodnie,

że dla celów dopasowania zawsze można uzyskać literalne znaczenie dowolnego znaku niealfanumerycznego, poprzedzając go znakiem *backslash*, wykonanie `quotemeta` na łańcuchu `$hostname` zmieni ten łańcuch w coś, co bezpiecznie dopasowuje samo siebie. Jeżeli spróbujemy użyć `$hostname` we wzorcu *regex* bez przygotowania przez `quotemeta`, możemy mieć kłopoty z powodu takich rzeczy, jak choćby ta, że kropki w nazwie stacji będą interpretowane jako metaznaki *regex*.

Przy sposobności zwróćcie uwagę, że użyliśmy modyfikatora `/o` na końcach obu wyrażeń regularnych: `/^http:\/\/$escaped_hostname/io` oraz `s/^http:\/\/$escaped_hostname//io`. Dzięki temu skrypt działa sprawnie. Jak wspomnieliśmy w rozdziale 7., modyfikator `/o` zawiadamia Perl, że żadna zmienna zawarta we wzorcu wyszukiwania nie będzie ulegać zmianie w czasie życia skryptu. Jeżeli wyrażenie zawierające zmienną będzie podlegać wielokrotnym wartościowaniom, użycie `/o` spowoduje, że skrypt zadziała szybciej, gdyż Perl nie będzie musiał rekompilować wyrażenia, ilekroć ją napotka.

Wyskakujemy z pętli przy odnośnikach, które wskazują na inne nazwy stacji, a nazwę naszej stacji usuwamy z początku tych odnośników, które ją zawierają. Następnie, jeżeli odnośnik rozpoczyna się wiodącym znakiem *slash* (co oznacza, że reprezentuje absolutną nazwę ścieżki w serwerze WWW), jest przekształcany na rzeczywistą ścieżkę systemu plików przez wstawienie zawartości zmiennej konfiguracyjnej `$web_root` na początku. W przeciwnym razie (gdy odnośnik jest względny) na początku odnośnika jest wstawiana absolutna nazwa ścieżki do kartoteki zawierającej plik HTML, z którego ten odnośnik został wyciągnięty.

Składamy wszystkie części w całość

Podsumujmy to, co zrobiliśmy dotychczas. Napisaliśmy skrypt, który będzie rekurencyjnie schodzić w dół systemu plików, wczytując treść wszystkich napotkanych plików HTML i wyciągając z tych plików wszystkie atrybuty `<A HREF="...">` i `<IMG SRC="...">`. Utworzyliśmy również procedurę, która weźmie nazwę kartoteki i listę odnośników wyciągniętych z pliku w tej kartotece, rozstrzygnie, które z nich wskazują na lokalny system plików, i te odnośniki przekształci na pełne (czyli bezwzględne) nazwy ścieżek systemu plików.

Szybka, ale niezbyt ciekawa wersja programu do sprawdzania odnośników jest prawie gotowa. Podstawowa rzecz, jaka nam pozostała, to definicja struktury danych, która będzie utrzymywać informacje o popsutych odnośnikach wykrytych przez program.

Wracamy więc na początek skryptu — tuż pod sekcję konfiguracyjną — i dodajemy następujący fragment programu:

```
my %bad_links;     # Tzw. "hash tablic" z kluczami tworzonymi przez URLe,
                   # począwszy od $start_base, i wartościami tworzonymi
                   # przez listy popsutych odnośników w tych stronach.

my %good;          # Hash, który odwzorowuje ścieżki systemu plików na
                   # 0 lub 1 (dobre lub złe). Zapamiętuje wyniki badań,
                   # wykonanych poprzednio, tak że nie muszą być
                   # powtarzane dla następnych stron.
```

Deklarujemy tutaj dwa nowe hashe, które będą użyte w skrypcie: `%bad_links` i `%good`. Hash `%good` jest dość prosty — użyjemy go, by utrzymywać wyniki sprawdzania odnośników, które przetwarza skrypt. Klucze hashu `%good` są ścieżkami lokalnego systemu plików dla plików, które

sprawdzamy (np. `/w1/s/socalsail/index.html`). Odnośnik, który okazuje się popsuty (to znaczy — wskazuje na nieistniejący plik lokalnego systemu plików) będzie umieszczony w hashu `%good` z wartością zero (`0`). Odnośnik wskazujący na plik, który istnieje, zostanie zapamiętany z wartością jeden (`1`). Odnośnik jeszcze nieprzetwarzany nie będzie mieć klucza w hashu.

W ten sposób możemy szybko ustalić w hashu `%good`, czy pewien odnośnik został sprawdzony i z jakim wynikiem. W przeciwnym razie musielibyśmy ciągle sprawdzać system plików (co jest procesem stosunkowo powolnym), ilekroć pojawiałby się wciąż ten odnośnik. Ponieważ witryny z reguły mają wiele odnośników nawigacyjnych, które wskazują na ten sam plik, użycie hashu do zapamiętywania tych ustaleń znacznie przyspieszy działanie skryptu.

Tworzymy hash tablic

A co z `%bad_links`, drugim spośród zdefiniowanych hashy? W tym hashu będziemy umieszczać wyniki sprawdzania odnośników. Przy tej okazji zetkniemy się po raz pierwszy z niewiarygodnie ostrą i mocną własnością Perla, czyli z wielopoziomowymi strukturami danych. *Wielopoziomowa struktura danych* jest czymś w rodzaju „tablicy tablic" lub „hashu hashy". Czasem jest to „hash tablic" — jak w przypadku `%bad_links`.

Co mam na myśli, gdy mówię „hash tablic"? Chodzi mi o zwykły hash ze zwykłymi kluczami hashu, ale o wartościach, które są tablicami, a nie zwyczajnymi wartościami skalarnymi (więcej o tym za chwilę). Znaczy to, że możemy użyć tego hashu, by powiedzieć „Daj mi wartość odpowiadającą kluczowi `walnuts`", a otrzymamy całą tablicę zamiast pojedynczej wartości. Zupełnie inną tablicą może być ta, którą otrzymamy, mówiąc „Daj mi wartość odpowiadającą kluczowi `rutabagas`". I tak dalej.

W obecnym przypadku kluczami `%bad_links` będą bezwzględne ścieżki każdego z plików HTML, który zawiera popsute odnośniki. Odpowiednią wartością w każdym przypadku będzie lista zawierająca wszystkie popsute odnośniki na tej stronie.

Jak się okazuje, faktycznie nie można przechowywać tablic jako wartości „hashu tablic", gdyż poszczególne wartości hashu mogą być tylko skalarami. Jednak w tych wartościach możemy przechować *odniesienia* do tablic.

Pamiętacie, jak odniesienie do procedury przekazaliśmy na początku rozdziału do funkcji `find` modułu `Find::File`? Zamiast wstawiać do argumentów funkcji wywołanie procedury (tak: `funkcja (&procedura)`), umieściliśmy w nich odniesienie do procedury, stosując znak *backslash* przed nazwą procedury (w taki sposób: `funkcja (\&procedura)`). Wyjaśniłem wtedy, że odniesienie jest sposobem tworzenia czegoś skalarnego, co odnosi się do treści czegoś innego (znanego już przez Perl). W tym drugim przypadku chodzi o skalar, tablicę, hash, procedurę, ale samo odniesienie zawsze jest skalarem.

Tak więc, wracając do tematu hashu `%bad_links`, możemy powiedzieć, że to, co określamy jako „hash tablic", nie jest naprawdę niczym takim, jak hash tablic. Jest to hash odniesień. Kiedy jednak pobierzemy odniesienie, łatwo będzie uzyskać dostęp do tablicy, na którą ono wskazuje.

Musimy nauczyć się specjalnej składni, by opanować tę magię, ale bez wątpienia warto się ją przestudiować. Gdy opanujemy tworzenie i użycie wielopoziomowych struktur danych, wiele skądinąd trudnych zadań programistycznych zamieni się w przysłowiową bułkę z masłem.

Wzbogacamy &process o obsługę popsutych odnośników

Jednak dość reklamowania. Zobaczmy, jak to działa. Musimy tylko wrócić do procedury `&process` i zmodyfikować ją w taki sposób:

```
sub process {

    # wołana przez funkcję find w File::Find, dla każdego
    # pliku, znalezionego przez find rekurencyjnie. wyciąga
    # listę atrybutów HREF i SRC z pliku HTML, przekształca
    # je na ścieżki lokalnego systemu plików, wykorzystując
    # procedurę convert, sprawdza ich "zepsucie", po czym
    # składuje błędne ścieżki w "hashu tablic" %bad_links.

    return unless /\.html$/;
    my $file = $File::Find::name;
    unless (open IN, $file) {
        warn "can't open $file for reading: $!, continuing...\n";
        return;
    }
    my $data = join '', <IN>; # wszystkie dane na raz
    close IN;
    return unless $data;

    my @targets = ($data =~ /(?:href|src)\s*=\s*"([^"]+)"/gi);

    @targets = &convert($File::Find::dir, @targets);

    foreach my $target (@targets) {

        if (exists $good{$target}) {

            # tę już widzieliśmy

            if ($good{$target}) {
                # wiadomo, że jest dobra
                next;
            } else {
                # wiadomo, że jest zła
                push @{ $bad_links{$file} }, $target;
            }

        } else {

            # tej jeszcze nie widzieliśmy

            if (-e $target) {
                $good{$target} = 1;
            } else {
                $good{$target} = 0;
                push @{ $bad_links{$file} }, $target;
            }
        }
    }
}
```

Zwróćcie uwagę, jak stosujemy teraz procedurę `&convert`, by zamienić listę wyciągniętych odnośników na bezwzględne ścieżki sytemu plików:

```
@targets = &convert($File::Find::dir, @targets);
```

Nawiasem mówiąc, bez problemu można użyć zmiennej `@targets` po obu stronach przypisania. Perl najpierw wartościuje prawą stronę przypisania, przekazując pierwotną zawartość tablicy `@targets` do procedury `&convert`, a później wstawia wartość zwracaną z procedury `&covert` z powrotem do zmiennej `@targets`, zastępując to, co w niej było.

Następną rzeczą godną uwagi jest pomysłowy sposób, w jaki używamy hashu `%good`, by sterować przepływem programu w trakcie przetwarzania każdej pozycji `@targets`. Po pierwsze — za pomocą funkcji `exists` sprawdzamy, czy dana wartość `$target` była już przetwarzana. Jak pamiętacie, funkcja `exists` sprawdza, czy w hashu istnieje pewien klucz — zwracając wartość prawdziwą, jeżeli ten klucz istnieje, i fałszywą w przeciwnym razie. (Pozwala to zwrócić wartość prawdziwą nawet dla tych kluczy, którym jest przyporządkowana wartość fałszywa, jak `0` lub `undef`).

Jeżeli funkcja `exists` dla danego klucza zwraca wartość `true`, uznajemy, że widzieliśmy już (oraz przebadaliśmy) tę ścieżkę, możemy zatem sięgnąć po prostu po wyniki poprzedniego badania, pobierając odpowiednią wartość przez `$good{target}`. Jeśli wartość `$target` wcześniej okazała się dobra, przeskakujemy w pętli `foreach` do następnej pozycji za pomocą `next` (ponieważ nie przejmujemy się dobrymi odnośnikami, tylko złymi). W przeciwnym razie (to jest, czyli wtedy, gdy dana ścieżka okazała się zła) dodajemy tę wartość do tablicy złych odnośników dla tego oto pliku, używając następującego, niezwykle ciekawego wiersza:

```
push @{ $bad_links{$file} }, $target;
```

W tym wierszu funkcja `push`, którą już dobrze znacie, wstawia wartość `$target` na koniec tablicy Ale jakże dziwnej tablicy!

Przez chwilę przypatrzcie się tej dziwnej tablicy. Widzicie coś, co określamy jako rozbieranie odniesień (ang. *dereferencing*), ponieważ jest to przekształcenie odwrotne — z odniesienia na pierwotną rzecz, na którą odniesienie to wskazuje. W obecnym przypadku tym, co zostanie zwrócone, jeżeli pobierzemy wartość przez klucz `$file` z hashu `%bad_links`, będzie *odniesienie do tablicy* (za chwilę zobaczycie, jak tam trafiło). Umieszczając odniesienie do tablicy w parze nawiasów klamrowych z symbolem tablicy (to jest — `@`) przed otwierającym nawiasem klamrowym, uzyskujemy dostęp do pierwotnej tablicy, na którą to odniesienie wskazuje, więc możemy wykonywać takie rzeczy, jak wstawianie nowych pozycji na koniec tej tablicy.

Powtórzmy to jeszcze raz: jeżeli mamy odniesienie do tablicy, to możemy je rozebrać, uzyskując dostęp do samej tablicy, na którą wskazuje:

```
@{odniesienie_do_tablicy}
```

Tak samo jest w przypadku rozbierania odniesień do hashy. Musimy tylko podstawić odpowiedni symbol (`%`) przed parą nawiasów klamrowych:

```
@{odniesienie_do_hashu}
```

Jest to druga część składni odniesień w języku Perl, której będziemy się uczyć. Znamy teraz jeden sposób tworzenia odniesień (przez wstawienie znaku *backslash* przed zmienną lub nazwą procedury, jak w `\@walnuts` lub `\&process`) i jeden sposób rozbierania odniesień, jeżeli chcemy dostać się do rzeczy, na którą odniesienie wskazuje (przez wstawienie odpowiedniego symbolu przed parą klamrowych nawiasów, zawierającą odniesienie, jak w `@{ref}`).

Wróćmy do procedury `&process` i spójrzmy na ramię `else` pętli. To ramię, które wyzwala się dla jeszcze niesprawdzonych plików. Ramię to sprawdza, czy plik istnieje w lokalnym systemie plików, używając operatora `-e` sprawdzania plików. Jeżeli plik istnieje, to w `$good{$target}` umieszczamy wartość prawdziwą (`1`). Natomiast jeżeli nie istnieje, to bieżąca wartość `$target` pochodzi od popsutego odnośnika. W tym przypadku umieszczamy wartość fałszywą (`0`) w `$good{$target}` i znów wstawiamy tę ścieżkę na koniec tablicy, do której odniesienie jest umieszczane w `$bad_links{$file}`, w taki sposób:

```
push @{ $bad_links{$file} }, $target;
```

I to wszystko. Gdy funkcja `find` modułu `File::Find` kończy zejście poprzez kartoteki poniżej `$start_dir`, wtedy dostępny w całym skrypcie hash pod nazwą `%bad_links`, — z kluczami tworzonymi przez nazwy wszystkich plików HTML, które zawierają popsute odnośniki — jest zapełniony przez procedurę `&process`. Wartości odpowiadające tym kluczom są odniesieniami do tablic, z których każda zawiera listę popsutych odnośników w pewnym określonym pliku HTML.

Przy sposobności zwróćcie uwagę, że wszystkie te tablice właściwie nie mają nazw. Do ich zawartości można się dostać tylko przez rozebranie odniesień, które na nie wskazują. Z tego powodu będziecie słyszeć czasem, jak o takich tablicach mówi się jako o tzw. *tablicach anonimowych*. Fakt, że tablice bywają anonimowe, może wywoływać Wasze zdumienie, dopóki nie zrozumiecie, że indywidualne tablice (lub hashe, lub skalary, lub procedury) mają w programie Perl swoje własne, niezależne istnienie, całkowicie oddzielne od jakichkolwiek nazw, które są do nich przyporządkowane, albo nie.

To, jak właściwie zaczęły istnieć odniesienia, również może wydać się Wam mętne. Faktycznie nie „tworzyliśmy" ich przez wstawianie znaku *backslash* przed konwencjonalną, nazwaną tablicą (jak w `\@array`), a także nie używaliśmy drugiej z metod jawnego tworzenia i zwracania odniesień (nie uczyliście się jeszcze tej metody, ale poznacie ją w rozdziale 14.). Skąd się więc wzięły? Po prostu wyłoniły się z niebytu w gotowej postaci, jak Atena z głowy Zeusa, gdy po raz pierwszy rozbieraliście te wartości hashu za pomocą `@{ $bad_links{$file} }`.

Drukujemy raport o popsutych odnośnikach

Wiem, że użycie odniesień wygląda na pracochłonne zajęcie. Jednak zobaczmy, jak łatwo można wytworzyć raport, który stanowi ostateczny wynik skryptu `link_check.plx`. Dodamy pod koniec skryptu, tuż przed definicjami procedur, następujący fragment kodu:

```
my $time = localtime;

print "$hostname$web_path link_check report\n";
print "Report created at $time\n\n";

foreach my $file (sort keys %bad_links) {
    print "$file:\n";
    foreach my $target (sort @{ $bad_links{$file} }) {
        print "  $target\n";
    }
    print "\n";
}
```

Widzicie, jak łatwo poszło!

Kiedy uruchomimy tę wersję skryptu z shella, otrzymamy informację wyjścia, która wygląda mniej więcej tak:

```
[jbc@andros ora]$ ./link_check.plx | more
www.socalsail.com/expo link_check report
Report created at Sun Jan  9 13:14:11 2000

/w1/s/socalsail/expo/join/index.html:
  /w1/s/socalsail/search/search.gif
  /w1/s/socalsail/talk/
  /w1/s/socalsail/user_info/

/w1/s/socalsail/expo/nav/index.html:
  /w1/s/socalsail/search/search.gif
  /w1/s/socalsail/talk/

/w1/s/socalsail/expo/wx/buoy.html:
  /w1/s/socalsail/search/search.gif
  /w1/s/socalsail/talk/
```

Dodajemy wyjście HTML

Dodajmy do skryptu jeszcze jedną własność: wyjście HTML. Tworząc raport w postaci strony HTML, ułatwimy sobie ocenę wyników w przeglądarce, w której możemy kliknąć odnośnik, aby odwiedzić stronę, gdzie według raportu występują problemy.

Zaczynamy od wstawienia następującego wiersza u góry skryptu — w sekcji konfiguracyjnej:

```
my $webify      = 1;                    # wytworzyć wyjście www?
```

Część, która drukuje raport, modyfikujemy w następujący sposób:

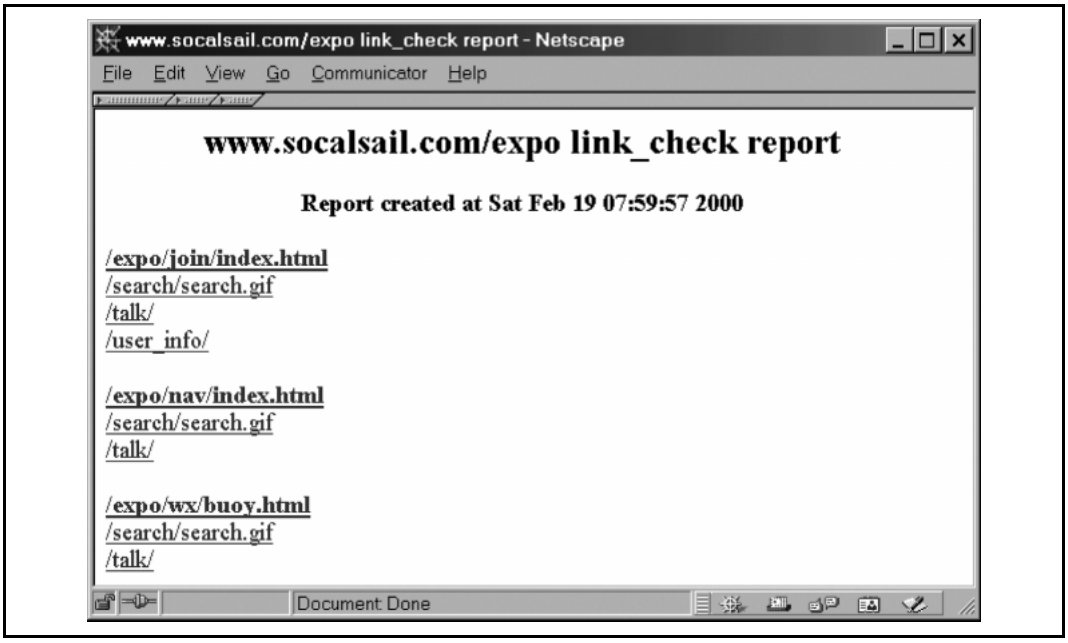
```
if ($webify) {

    # drukuj wersję HTML tego raportu

    print <<EndOfText;
<HTML>
<HEAD>
<TITLE>$hostname$web_path link_check report</TITLE>
</HEAD>
<BODY>

<H2 ALIGN="center">$hostname$web_path link_check report</H2>

<P ALIGN="center"><STRONG>Report created at $time</STRONG></P>

<P>
EndOfText

    foreach my $file (sort keys %bad_links) {
        my $pretty_file      = $file;
        my $escaped_web_root = quotemeta $web_root;
        $pretty_file         =~ s/$escaped_web_root//o;
        $pretty_file         = "<P><STRONG><A
HREF=\"$pretty_file\">$pretty_file</A></STRONG><BR>\n";
        print $pretty_file;
        foreach my $target (sort @{ $bad_links{$file} }) {
            $target =~ s/$escaped_web_root//o;
```

```
            print "<A HREF=\"$target\">$target</A><BR>\n";
        }
        print "\n</P>\n\n";
    }
    print "</BODY></HTML>\n";

} else {

    # drukuj tylko czysto tekstową wersję raportu

    print "$hostname$web_path link_check report\n";
    print "Report created at $time\n\n";

    foreach my $file (sort keys %bad_links) {
        print "$file:\n";
        foreach my $target (sort @{ $bad_links{$file} }) {
               print "  $target\n";
        }
           print "\n";
    }
}
```

Nowy kod powinien obecnie być całkowicie jasny. Jeżeli teraz uruchomimy skrypt po nastawieniu zmiennej konfiguracyjnej `$webify` na wartość prawdziwą, to jako informację wyjścia skrypt wytworzy stronę HTML. Wobec tego możemy wykonać ten skrypt z wiersza poleceń:

```
[jbc@andros ora]$ link_check.plx > report.html
```

a jego wyniki obejrzeć w przeglądarce, tak jak na rysunku 11.1.



Rysunek 11.1. Raport HTML wytworzony przez link_check.plx

Cały skrypt — w postaci, jaką powinien mieć obecnie — przedstawia przykład 11.2.

Przykład 11.2. Pierwsza wersja skryptu do sprawdzania odnośników

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

# link_check.plx

# pierwsza wersja programu do sprawdzania odnośników HTML.
# program schodzi rekurencyjnie ze $start_dir, przetwarza
# wszystkie pliki .htm lub .html, wyciąga atrybuty HREF
# i SRC, a następnie sprawdza wszystkie, które wskazują
# na plik lokalny, by potwierdzić, że ten plik istnieje.

use strict;
use File::Find;

# sekcja konfiguracyjna:

# uwaga: pierwsze cztery zmienne konfiguracyjne *nie* powinny zawierać
# końcowego znaku slash (/)

my $start_dir   = '/w1/s/socalsail/expo'; # gdzie zacząć odnajdywanie
my $hostname    = 'www.socalsail.com';    # nazwa stacji dla witryny
my $web_root    = '/w1/s/socalsail';      # root dokumentów www
my $web_path    = '/expo';                # ścieżka www do $start_dir
my $webify      = 1;                      # wytworzyć wyjście www?

# koniec sekcji konfiguracyjnej

my %bad_links;    # Tzw. "hash tablic" z kluczami tworzonymi przez URLe,
                  # począwszy od $start_base, i wartościami tworzonymi
                  # przez listy popsutych odnośników w tych stronach.

my %good;         # Hash, który odwzorowuje ścieżki systemu plików na
                  # 0 lub 1 (dobre lub złe). Zapamiętuje wyniki badań,
                  # wykonanych poprzednio, tak że nie muszą być
                  # powtarzane dla następnych stron.

find(\&process, $start_dir); # ten wiersz zapełnia oba hashe

my $time = localtime;

if ($webify) {

    # drukuj wersję HTML tego raportu

    print <<EndOfText;
<HTML>
<HEAD>
<TITLE>$hostname$web_path link_check report</TITLE>
</HEAD>
<BODY>

<H2 ALIGN="center">$hostname$web_path link_check report</H2>

<P ALIGN="center"><STRONG>Report created at $time</STRONG></P>

<P>
EndOfText

    foreach my $file (sort keys %bad_links) {
        my $pretty_file      = $file;
        my $escaped_web_root = quotemeta $web_root;
```

```
        $pretty_file         =~ s/$escaped_web_root//o;
        $pretty_file         = "<P><STRONG><A
HREF=\"$pretty_file\">$pretty_file</A></STRONG><BR>\n";
        print $pretty_file;
        foreach my $target (sort @{ $bad_links{$file} }) {
            $target =~ s/$escaped_web_root//o;
            print "<A HREF=\"$target\">$target</A><BR>\n";
        }
        print "\n</P>\n\n";
    }
    print "</BODY></HTML>\n";

} else {

    # drukuj tylko czysto tekstową wersję raportu

    print "$hostname$web_path link_check report\n";
    print "Report created at $time\n\n";

    foreach my $file (sort keys %bad_links) {
        print "$file:\n";
        foreach my $target (sort @{ $bad_links{$file} }) {
               print "  $target\n";
        }
           print "\n";
    }
}

sub process {

    # wołana przez funkcję find w File::Find, dla każdego
    # pliku, znalezionego przez find rekurencyjnie. wyciąga
    # listę atrybutów HREF i SRC z pliku HTML, przekształca
    # je na ścieżki lokalnego systemu plików, wykorzystując
    # procedurę convert, sprawdza ich "zepsucie", po czym
    # składuje błędne ścieżki w "hashu tablic" %bad_links.

    return unless /\.html$/;
    my $file = $File::Find::name;
    unless (open IN, $file) {
        warn "can't open $file for reading: $!, continuing...\n";
        return;
    }
    my $data = join '', <IN>; # wszystkie dane na raz
    close IN;
    return unless $data;

    my @targets = ($data =~ /(?:href|src)\s*=\s*"([^"]+)"/gi);

    @targets = &convert($File::Find::dir, @targets);

    foreach my $target (@targets) {

        if (exists $good{$target}) {

            # tę już widzieliśmy

            if ($good{$target}) {
                # wiadomo, że jest dobra
                next;
            } else {
                # wiadomo, że jest zła
```

```
                push @{ $bad_links{$file} }, $target;
            }

        } else {

            # tej jeszcze nie widzieliśmy

            if (-e $target) {
                $good{$target} = 1;
            } else {
                $good{$target} = 0;
                push @{ $bad_links{$file} }, $target;
            }
        }
    }
}

sub convert {

    # Przyjmuje (w pierwszym argumencie) nazwę kartoteki pliku,
    # z którego wyciągnięto listę URLi, i listę URLi, wyciągniętą
    # z tego pliku (w pozostałych argumentach). Zwraca listę
    # wszystkich URLi, które nie wskazują na zewnątrz lokalnego
    # środowiska (ang. site), różnych od ftp:, mailto:, https:,
    # lub news:, gdzie te URLe są przekształcone na nazwy
    # plików lokalnego systemu plików.

    my($dir, @urls) = @_;
    my @return_urls;
    my $escaped_hostname = quotemeta $hostname;
    foreach (@urls) {
        next if /^(ftp|mailto|https|news):/i;
        if (/^http:/i) {
            # URL zaczyna się od 'http:'
            next unless /^http:\/\/$escaped_hostname/io;
            s/^http:\/\/$escaped_hostname//io;
        }
        if (/^\//) {
            # URL zaczyna się od '/'
            $_ = $web_root . $_;
        } else {
            # URL jest ścieżką względną
            $_ = $dir . '/' . $_;
        }
        s/#.*//;  # utnij końcowe #kotwiczki
        s/\?.*//; # utnij końcowe ?argumenty
        push @return_urls, $_;
    }
    @return_urls;
}
```

Korzystamy z CPAN

Jak poprzednio wspomniałem, pierwsza wersja skryptu do sprawdzania odnośników jest bardzo ograniczona. Wersja ta sprawdza tylko odnośniki wskazujące na pliki w lokalnym systemie plików i będzie bezradna wobec takich rzeczy, jak znacznik `<BASE HREF="...">`, który modyfikuje sposób ustalania przez przeglądarkę odnośników względnych. Niemniej ta wersja działa szybko i w wielkiej witrynie, która nie narusza przyjętych założeń, jest w stanie sprawdzić przynajmniej bardziej oczywiste przypadki popsutych odnośników.

Warto byłoby udoskonalić tę wersję tak, by mogła sprawdzać odnośniki wskazujące poza witrynę, używając HTTP przy żądaniu stron, zupełnie jak przeglądarka. Można by było napisać własny kod Perl do przeglądania WWW, ale na szczęście ta praca została już wykonana — i to lepiej, niż moglibyście to zrobić Wy lub ja. Osobą, która za to odpowiada, jest Gisle Aas, ceniony w społeczności Perla autor modułu `LWP` (skrót od `libwww-perl`).

Użycie LWP zaoszczędzi nam wielkiej ilości czasu i silnych bólów głowy. Ponieważ moduł ten obecnie nie jest zawarty w standardowej dystrybucji Perla, więc będziemy musieli pobrać go z CPAN (Comprehensive Perl Archive Network, *http://www.cpan.org/*) i zainstalować (zakładając, że nie jest zainstalowany jako część Waszej kopii Perla). Opanowanie tej umiejętności będzie wymagać od nas pewnego wstępnego wysiłku, lecz wierzcie mi — inwestując w ten sposób swój czas, tylko na tym skorzystamy.

Sprawdzamy obecność LWP

Zanim weźmiemy się do pobierania i instalacji LWP, zróbmy następującą próbę, by sprawdzić, czy został on już zainstalowany w naszej konfiguracji Perla:

```
[jbc@andros jbc]$ perl -MLWP -e 'print "LWP is installed!\n"'
```

Opcja wiersza poleceń `-M`, po której następuje nazwa modułu (bez spacji pomiędzy jednym i drugim), spowoduje, że ten jednowierszy skrypt załaduje wskazany moduł. Jeżeli moduł LWP jest zainstalowany, otrzymamy następującą informację wyjścia:

```
LWP is installed!
```

W przypadku, gdy moduł LWP nie jest zainstalowany, otrzymamy coś takiego:

```
Can't locate LWP.pm in @INC (@INC contains:
/usr/lib/perl5/i386-linux/5.00405 /usr/lib/perl5
/usr/lib/perl5/site_perl/i386-linux /usr/lib/perl5/site_perl .).
BEGIN failed--compilation aborted.
```

Jeżeli w Waszej konfiguracji moduł ten nie jest zainstalowany, w następnej sekcji zobaczycie, jak poradzić sobie z jego pobraniem i instalacją. Jeśli LWP jest zainstalowany, możecie pominąć tę sekcję (ale przygotujcie się na to, że trzeba będzie do niej wrócić, gdy jakiś inny kod z CPAN — niezainstalowany w Waszej konfiguracji Perla — okaże się potrzebny).

Instalujemy LWP z CPAN

Jak wspomniałem w rozdziale 8., archiwa CPAN są tak ogromne, że pierwsze zetknięcie z nimi może trochę zniechęcać do pracy z CPAN. Po pewnym czasie jednak, zaczną być bardziej przejrzyste, a czas poświęcony przyzwyczajaniu się do tej pracy zwróci się po wielokroć.

Jest kilka sposobów na odszukanie danego modułu w CPAN. Można między innymi rozpocząć od *http://www.cpan.org/README.html*.

Tu możemy kliknąć na odnośnik *Modules*, który powinien nas przenieść do *http://www.cpan.org/modules/index.html*. Kliknięcie na odnośnik *All Modules* powinno z kolei przenieść nas stąd do *http://www.cpan.org/modules/01modules.index.html*.

Z tego miejsca można przeglądnąć listę modułów i odszukać moduł `LWP` oraz ostatnią przeznaczoną do pobierania wersję, którą w czasie, gdy powstawał ten tekst, zawierał plik `libwww-perl-5.53.tar.gz`.

Bardzo użytecznym narzędziem jest motor CPAN do wyszukiwania, znajdujący się pod adresem *http://search.cpan.org/*. Podając do motoru wyszukiwania nazwę modułu, możemy otrzymać listę wyników, które obejmują odnośniki do dokumentacji tego modułu, jak również odnośniki do pobierania ostatniej wersji samego modułu. Ponieważ często chcę przeglądnąć dokumentację modułu, zanim zajmę się pobieraniem i instalacją, która wymaga czasu, jest to bardzo duże udogodnienie.

W taki czy inny sposób jednak w końcu uda się nam znaleźć URL, który wskazuje na ostatnią wersję `libwww-perl`. Musimy wtedy pobrać to archiwum na nasz serwer, rozpakować je, wyciągnąć pliki, które zawiera, i przeprowadzić właściwą instalację.

Pobieramy plik archiwum na serwer

Jest kilka sposobów, żeby pobrać plik CPAN na nasz serwer WWW. Osobiście — po odszukaniu odnośnika do zarchiwizowanego modułu za pomocą swojej przeglądarki — kopiuję zazwyczaj ten odnośnik do schowka swojego komputera. (Kliknięcie odnośnika prawym klawiszem myszy otwiera menu kontekstowe wersji Netscape dla Windows, pozwalające go skopiować). Wtedy przełączam się do drugiego okna, gdzie mam swoją sesję `telnet` z serwerem WWW. W sesji `telnet` przechodzę do tymczasowej kartoteki (np. do `~/tmp`, czyli kartoteki o nazwie `tmp`, którą utworzyłem we własnej kartotece użytkownika), wydaję polecenie `lynx`, a potem doklejam URL do wiersza poleceń za pomocą kombinacji klawiszy Shift-Insert. Oto przykład:

```
[jbc@andros tmp]$ lynx http://www.perl.com/CPAN/authors/id/GAAS/libwww-perl-
5.53.tar.gz
```

W ten sposób jestem w przeglądarce `lynx` i mam zgłoszenie pobierania pliku. Wybieram pobieranie, zatwierdzam domyślną nazwę `libwww-perl-5.53.tar.gz`, pod którą jest zapisywany na dysku, pobieram plik i kończę pracę z przeglądarką `lynx`.

Dekompresja pliku archiwum

Przy zgłoszeniu shella wydaję polecenie `gzip` (z opcją `-d`), by rozpakować pobrany plik:

```
[jbc@andros tmp]$ gzip -d libwww-perl-5.53.tar.gz
```

W ten sposób plik skompresowany `libwww-perl-5.53.tar.gz` zostaje zastąpiony plikiem `libwww-perl-5.53.tar`, który nie jest skompresowany. Przeczytajcie tekst w ramce „Dokończenie przez tabulację", żeby dowiedzieć się, jak uniknąć urazu od ciągłego powtarzania tych samych ruchów przy wypisywaniu takich długich nazw plików.

Ekstrakcja plików z archiwum

Teraz zastosujemy polecenie `tar` (od *tape archive*, co nawiązuje do pierwotnej funkcji tego polecenia, która polega na tworzeniu kopii zapasowych na taśmie) do ekstrakcji poszczególnych plików z pliku `tar`. (Właściwie wyciągnę całe drzewo kartotek i plików, a nie poszczególne pliki). Najtrudniej jest zapamiętać wszystkie potrzebne opcje `tar`. W tym przypadku zastosujemy opcję `x` (mówimy

Dokończenie przez tabulację

Całe lata (dosłownie) strawiłem na wypisywaniu naprawdę długich nazw plików w wierszu poleceń Unixa, aż wreszcie pewien uprzejmy guru powiedział mi o *dokończeniu przez tabulację*. W shellu `bash` nie ma potrzeby wypisywania całej nazwy pliku, gdy podaje się argument takiego polecenia, jak `gzip`. Wystarczy napisać tylko taką część nazwy pliku, po której shell będzie w stanie odróżnić ten plik od innych plików w bieżącej kartotece. Gdy napiszecie taką część, naciskacie klawisz Tab, a shell dokończy za Was resztę nazwy pliku, po czym będziecie mogli nacisnąć klawisz Enter, żeby wprowadzić polecenie.

Dokończenie przez tabulację jest jedną z wielu funkcji edycji wiersza poleceń, które udostępnia `bash`; więcej na ten temat można dowiedzieć się z sekcji poświęconej bibliotece `READLINE` na stronie `man bash`.

programowi `tar`, że ekstrakcja dotyczy istniejącego archiwum) i opcję `f` (na końcu, co przypadku tej opcji jest ważne), oznaczającą, że zaraz podamy nazwę pliku `tar`, z którego chcemy wyciągać pliki. Zbierając to w całość, uzyskujemy:

```
[jbc@andros tmp]$ tar -xf libwww-perl-5.53.tar
```

Ten wiersz powoduje, że `tar` wyciąga z archiwum wszystkie pliki i kartoteki, umieszczając je w kartotece o nazwie `libwww-perl-5.53`. Za pomocą `cd` przechodzimy do tej kartoteki (korzystając z dokończenia przez tabulację, jeżeli mamy dość rozumu — zamiast wypisywać całą nazwę kartoteki), a następnie wydajemy polecenie `ls`, żeby wylistować treść tej kartoteki:

```
[jbc@andros libwww-perl-5.53]$ ls
ChangeLog  Makefile.PL  README.SSL  bin  lwpcook.pod
MANIFEST   README       TODO        lib  t
```

Instalacja

Gdy wyciągnęliśmy pliki zawarte w pliku `tar` z modułem Perla, powinniśmy najpierw przeczytać dołączony do tego modułu plik `README`. Dowiemy się między innymi, czy istnieją warunki, które muszą być spełnione przy instalacji. W tym przypadku plik `README` mówi nam to:

We recommend that you have the following packages installed before you

```
install libwww-perl:
  URI
  MIME-Base64
  HTML-Parser
  libnet
  Digest::MD5
These packages should be available on CPAN.
```

Wydaje się, że czekają nas kolejne ćwiczenia w pobieraniu, dekompresji i ekstrakcji. Wracamy więc do CPAN, znajdujemy najświeższe pliki `*.tar.gz`, zawierające te moduły, po czym powtarzamy wymienione powyżej kroki z pobraniem `gzip -d` i `tar -xf`. Kiedy skończymy, będziemy mieć w kartotece `tmp` garść kartotek, po jednej dla każdego zalecanego modułu, a także jedną kartotekę z nadal niezainstalowanym modułem `libwww-perl`.

Najtrudniejszą częścią całego zadania będzie odszukanie modułu `libnet`, ponieważ na liście CPAN jest faktycznie wymieniony jako `Bundle-libnet`.

W tej chwili wydruk `ls` w kartotece `~/tmp` powinien wyglądać tak:

```
[jbc@andros tmp]$ ls
Bundle-libnet-1.00      HTML-Parser-3.04      URI-1.04
Bundle-libnet-1.00.tar  HTML-Parser-3.04.tar  URI-1.04.tar
Digest-MD5-2.09         MIME-Base64-2.11      libwww-perl-5.53
Digest-MD5-2.09.tar     MIME-Base64-2.11.tar  libwww-perl-5.53.tar
```

Chociaż `URI` jest pierwszym modułem na liście zależności `LWP`, okazuje się, że chce, abyśmy zainstalowali najpierw `MIME-Base64`, więc ten moduł faktycznie będzie pierwszym, który zainstalujemy. Przez `cd` przechodzimy do jego kartoteki i czytamy plik `README`. Według tego pliku musimy wykonać następujące cztery kroki:

```
perl Makefile.PL
make
make test
make install
```

Każdy z nich jest poleceniem, które powinniśmy wprowadzić w wierszu poleceń Unixa, i ta sama sekwencja czterech poleceń będzie powtarzać się przy instalacji prawie wszystkich modułów Perla (chyba że użyjemy modułu CPAN, o czym powiemy za chwilę).

Instalacja z dostępem root i przez zwykłego użytkownika

Instalacja nowego modułu Perla jest jedną z tych okazji, gdy warto mieć dostęp do uprawnień `root` (lub do uprzejmego administratora, który skorzysta dla nas z tych uprawnień). Jeżeli wykonujemy instalację (a właściwie jej ostatnie stadium, `make install`) jako użytkownik `root`, moduł, który instalujemy, może zostać zainstalowany w głównej instalacji Perla w serwerze, dzięki czemu będzie dostępny dla każdego, kto korzysta z tego serwera. Będziemy mogli też używać tego modułu w naszych skryptach, nie manipulując listą kartotek, w których Perl szuka plików należących do modułów.

Jeżeli nie mamy dostępu do konta `root`, wciąż możemy instalować własne moduły Perla, ale będziemy musieli instalować je we własnej przestrzeni na serwerze. Użycie tych modułów w skryptach także stanie się nieco trudniejsze, zmuszając nas do umieszczania dodatkowego wiersza w każdym skrypcie (za chwilę dowiemy się, co to za wiersz).

Jeżeli instalujemy jako użytkownik `root`, możemy po prostu przejść przez wspomniane wcześniej stadia (`perl Makefile.PL`, `make`, `make test` i `make install`), przestrzegając wskazówek i odpowiadając na pytania — w miarę, jak się pojawiają. (Ostatnie stadium — `make install` — faktycznie jest jedynym stadium, w którym będziemy potrzebować uprawnień `root`). Jeżeli jednak instalujemy, korzystając ze zwykłego konta użytkownika, to będziemy musieli nieco zmodyfikować wiersz `Makefile.PL`. W tym przypadku wprowadzimy ten wiersz w taki sposób:

```
[jbc@andros MIME-Base64-2.11]$ perl Makefile.PL PREFIX=/home/jbc/perl
```

gdzie `/home/jbc/perl` odpowiada naszej kartotece z prawem do pisania (w której chcemy zainstalować swoje osobiste kopie modułów Perl).

Stadia `make`, `make test` i `make install` przeprowadzamy w zwykły sposób. Po ostatnim stadium — `make install` — nowy moduł powinien być zainstalowany w wyspecyfikowanej przez parametr `PREFIX` kartotece, której ścieżka wygląda mniej więcej tak:

```
/home/jbc/perl/lib/site_perl/5.005/i386-linux/MIME/Base64.pm
```

Wtedy musimy po prostu pamiętać o umieszczeniu instrukcji `use lib` w skrypcie, zanim spróbujemy użyć modułu `MIME::Base64`. Chodzi o coś w tym rodzaju:

```
use lib '/home/jbc/lib/perl5/site_perl/5.005/i386-linux';
```

W ten sposób Perl doda tę kartotekę do listy kartotek w specjalnej tablicy `@INC`, która jest sprawdzana, ilekroć wydajemy instrukcję `use Some::Module`.

Zatem przeprowadzimy instalację każdego modułu: `Mime-Base64`, `URI`, `HTML-Parser`, `Bundle-libnet` i `Digest::MD5`. Po przeprowadzeniu wszystkich instalacji będziemy mogli wrócić do `LWP` i zainstalować ten moduł. (Instalacja `LWP` jest dość wymagająca, gdyż zadaje szereg pytań, aby skonfigurować różne usługi, z których być może zechcemy korzystać w sieci Internet. Jednak — nawet jeżeli opuścimy większość tych pytań, wciąż powinniśmy mieć wersję, która nadaje się do naszych obecnych celów).

Pobieranie, dekompresja, `make` i instalacja są dość nudne i powtarzają się. Można przypuszczać, że ktoś wymyślił metodę automatyzacji tego zadania. Jest ktoś taki — Andreas König, autor modułu `CPAN.pm` (i kilku innych). W sprawie szczegółów dotyczących tego tematu zajrzyj do ramki „Instalujemy moduły z CPAN.pm".

Sprawdzamy zdalne odnośniki

Przykład 11.3 pokazuje skrypt `link_check2.plx` — ulepszoną wersję programu do sprawdzania odnośników, która daje możliwość sprawdzania odnośników wskazujących poza środowisko (ang. *offsite links*). Części tego skryptu, które różnią się od poprzedniego, zostały wyróżnione.

Przykład 11.3. Skrypt do sprawdzania odnośników ze sprawdzaniem poza witryną

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

# link_check2.plx

# Druga wersja programu do sprawdzania odnośników HTML.
# program schodzi rekurencyjnie ze $start_dir, przetwarza
# wszystkie pliki .htm lub .html, wyciąga atrybuty HREF
# i SRC, a następnie sprawdza wszystkie, które wskazują
# na plik lokalny, by potwierdzić, że ten plik istnieje,
# a dodatkowo w tym samym celu może użyć LWP::Simple do
# sprawdzania HEAD dla zdalnych atrybutów. Następnie
# składa raport o popsutych odnośnikach.

use strict;
use File::Find;
use LWP::Simple;
# sekcja konfiguracyjna:

# uwaga: pierwsze cztery zmienne konfiguracyjne *nie* powinny zawierać
# końcowego znaku slash (/)

my $start_dir    = '/w1/s/socalsail/expo'; # gdzie zacząć odnajdywanie
my $hostname     = 'www.socalsail.com';    # nazwa stacji dla witryny
my $web_root     = '/w1/s/socalsail';      # root dokumentów www
my $web_path     = '/expo';                # ścieżka www do $start_dir
my $webify       = 1;                      # wytworzyć wyjście www?
my $check_remote = 1;                      # sprawdzać zdalne odnośniki?
```

Instalujemy moduły z CPAN.pm

Moduł `CPAN.pm` jest niezwykle użyteczny przy pobieraniu, dekompresji i instalacji modułów Perl. Dostarcza prosty shell poleceń, który przyspiesza ręczną pracę we wszystkich stadiach opisanych w tym rozdziale.

Dlaczego więc zadałem sobie tyle trudu, wyjaśniając całe postępowanie, skoro `CPAN.pm` pozwala je pominąć? Zrobiłem to, ponieważ `CPAN.pm` może jeszcze nie być zainstalowany, a w takim razie trzeba będzie przeprowadzić normalną instalację modułu, aby go zainstalować.

Aby sprawdzić, czy `CPAN.pm` jest zainstalowany w Waszej kopii Perla, napiszcie przy zgłoszeniu Unixa następujący wiersz:

```
[john@ithil john]$ perl -MCPAN -e shell
```

Jeżeli `CPAN.pm` jest dostępny, dostaniecie coś, co wygląda mniej więcej tak:

```
cpan shell -- CPAN exploration and modules installation (v1.52)
ReadLine support enabled

cpan>
```

Ostatni wiersz jest zgłoszeniem shella `CPAN.pm`. Spróbujcie wprowadzić `h`, a otrzymacie listę niektórych dostępnych poleceń. Niejednokrotnie wystarczy, jeżeli wprowadzicie:

```
cpan> install Module_name
```

(gdzie *`Module_name`* zastąpicie nazwą modułu, który chcecie zainstalować), a `CPAN.pm` przeprowadzi całą sekwencję pobrania, dekompresji, `make` i instalacji.

Jeżeli `CPAN.pm` nie jest zainstalowany, to gdy uruchomicie polecenie `perl -MCPAN -e shell`, otrzymacie komunikat, który wygląda tak:

```
[jbc@andros jbc]$ perl -MCPAN -e shell
Can't locate CPAN.pm in @INC (@INC contains: /usr/lib/perl5/alpha-
linux/5.00404 /usr/lib/perl5 /usr/lib/perl5/site_perl/alpha-linux
/usr/lib/perl5/site_perl .).
BEGIN failed--compilation aborted.
```

W takim przypadku znajdźcie moduł `CPAN.pm` w archiwach CPAN, pobierzcie go i przeprowadźcie proces ręcznej instalacji modułu, opisany wcześniej w tym rozdziale. Następnie powinniście rozpocząć shell `CPAN.pm` za pomocą polecenia `perl -MCPAN -e shell`. Kiedy ten shell rozpoczyna pracę po raz pierwszy, zadaje szereg pytań, aby przeprowadzić własną konfigurację. Można przyjąć domyślne nastawienia dla niemal wszystkich tych pytań, lecz należy uważać na to, które pyta o parametry, jakie chcecie przekazywać do `Makefile.PL`. Zakładając, że nie dokonujecie instalacji jako `root`, lecz musicie instalować swoje moduły do własnych kartotek, będziecie musieli dać mniej więcej taką odpowiedź, gdy `CPAN.pm` zapyta o parametry `Makefile.PL`:

```
PREFIX=/home/jbc
```

gdzie `/home/jbc` zastąpicie przez taką własną kartotekę z prawem do pisania, w której `CPAN.pm` ma zainstalować Waszą osobistą bibliotekę Perla.

```
# koniec sekcji konfiguracyjnej

my %bad_links;     # Tzw. "hash tablic" z kluczami tworzonymi przez URLe,
                   # począwszy od $start_base, i wartościami tworzonymi
                   # przez listy popsutych odnośników w tych stronach.

my %good;          # Hash, który odwzorowuje ścieżki systemu plików na
                   # 0 lub 1 (dobre lub złe). Zapamiętuje wyniki badań,
                   # wykonanych poprzednio, tak że nie muszą być
                   # powtarzane dla następnych stron.

find(\&process, $start_dir); # ten wiersz zapełnia oba hashe

my $time = localtime;

if ($webify) {

    # drukuj wersję HTML tego raportu

    print <<EndOfText;
<HTML>
<HEAD>
<TITLE>$hostname$web_path link_check report</TITLE>
</HEAD>
<BODY>

<H2 ALIGN="center">$hostname$web_path link_check report</H2>

<P ALIGN="center"><STRONG>Report created at $time</STRONG></P>

<P>
EndOfText

    foreach my $file (sort keys %bad_links) {
        my $pretty_file      = $file;
        my $escaped_web_root = quotemeta $web_root;
        $pretty_file         =~ s/$escaped_web_root//o;
        $pretty_file         = "<P><STRONG><A
HREF=\"$pretty_file\">$pretty_file</A></STRONG><BR>\n";
        print $pretty_file;
        foreach my $target (sort @{ $bad_links{$file} }) {
            $target =~ s/$escaped_web_root//o;
            print "<A HREF=\"$target\">$target</A><BR>\n";
        }
        print "\n</P>\n\n";
    }
    print "</BODY></HTML>\n";

} else {

    # drukuj tylko czysto tekstową wersję raportu

    print "$hostname$web_path link_check report\n";
    print "Report created at $time\n\n";

    foreach my $file (sort keys %bad_links) {
        print "$file:\n";
        foreach my $target (sort @{ $bad_links{$file} }) {
               print "  $target\n";
        }
           print "\n";
    }
}
```

```
sub process {

    # wołana przez funkcję find w File::Find, dla każdego
    # pliku, znalezionego przez find rekurencyjnie. wyciąga
    # listę atrybutów HREF i SRC z pliku HTML, przekształca
    # je na ścieżki lokalnego systemu plików, wykorzystując
    # procedurę convert, sprawdza ich "zepsucie", po czym
    # składuje błędne ścieżki w "hashu list" %bad_links.

    return unless /\.html$/;
    my $file = $File::Find::name;

#    warn "processing $file...\n";

    unless (open IN, $file) {
        warn "can't open $file for reading: $!, continuing...\n";
        return;
    }
    my $data = join '', <IN>; # wszystkie dane na raz
    close IN;
    return unless $data;

    my @targets = ($data =~ /(?:href|src)\s*=\s*"([^"]+)"/gi);

    @targets = &convert($File::Find::dir, @targets);

    foreach my $target (@targets) {

        if (exists $good{$target}) {

            # tę już widzieliśmy

            if ($good{$target}) {
                # wiadomo, że jest dobra
                next;
            } else {
                # wiadomo, że jest zła
                push @{ $bad_links{$file} }, $target;
            }

        } elsif ($target =~ /^http:/) {

            # zdalny odnośnik, jeszcze niewidziany

            if (head($target)) {
                $good{$target} = 1;
            } else {
                push @{ $bad_links{$file} }, $target;
                $good{$target} = 0;
            }

        } else {

            # lokalny odnośnik, jeszcze niewidziany

            if (-e $target) {
                $good{$target} = 1;
            } else {
                $good{$target} = 0;
                push @{ $bad_links{$file} }, $target;
            }
        }
    }
}
```

```
sub convert {

    # Przyjmuje (w pierwszym argumencie) nazwę kartoteki pliku,
    # z którego wyciągnięto listę URLi, i listę URLi, wyciągniętą
    # z tego pliku (w pozostałych argumentach). Zwraca listę
    # wszystkich URLi, które nie wskazują na zewnątrz lokalnego
    # środowiska (ang. site), różnych od ftp:, mailto:, https:,
    # lub news:, gdzie te URLe są przekształcone na nazwy
    # ścieżek lokalnego systemu plików. Nieobowiązkowo, jeżeli
    # zmienna $check_remote jest nastawiona na wartość prawdziwą,
    # zwraca w pierwotnej postaci wszelkie odnośniki 'http:',
    # które wskazują *poza* lokalną witrynę.

    my($dir, @urls) = @_;
    my @return_urls;
    my $escaped_hostname = quotemeta $hostname;
    foreach (@urls) {
        next if /^(ftp|mailto|https|news):/i;
        if (/^http:/i) {
            # URL zaczyna się od 'http:'
            if (/^http:\/\/$escaped_hostname/io) {
                # lokalny odnośnik; konwertuj na lokalną nazwę
                # pliku, przygotowując konwersję poniżej
                s/^http:\/\/$escaped_hostname//io;
            } else {
                # zdalny odnośnik
                push @return_urls, $_ if $check_remote;
                next;
            }
        }
        if (/^\//) {
            # URL zaczyna się od '/'
            $_ = $web_root . $_;
        } else {
            # URL jest ścieżką względną
            $_ = $dir . '/' . $_;
        }
        s/#.*//;  # utnij końcowe #kotwiczki
        s/\?.*//; # utnij końcowe ?argumenty
        push @return_urls, $_;
    }
    @return_urls;
}
```

Dzięki pracy włożonej w instalację modułu `LWP` (wraz ze wszystkimi modułami, których on wymaga) zmiany potrzebne po to, by ten skrypt mógł sprawdzać zdalne odnośniki WWW, są dość nieznaczne.

Najpierw musieliśmy wywołać moduł `LWP::Simple`, który daje nam łatwy dostęp do transakcji WWW z wnętrza naszego skryptu:

```
use LWP::Simple;
```

Następnie dodaliśmy nową zmienną konfiguracyjną, żeby kontrolować to, czy skrypt będzie się zajmować sprawdzaniem zdalnych odnośników:

```
my $check_remote = 1;                    # sprawdzać zdalne odnośniki?
```

Wreszcie dodaliśmy nowe ramię `elsif` w procedurze `&process`, które obsługuje zdalne odnośniki (to jest — odnośniki rozpoczynające się od `http:`), jeszcze nieprzetwarzane przez skrypt:

```
} elsif ($target =~ /^http:/) {

    # zdalny odnośnik, jeszcze niewidziany

    if (head($target)) {
        $good{$target} = 1;
    } else {
        push @{ $bad_links{$file} }, $target;
        $good{$target} = 0;
    }
```

Widzicie miejsce, gdzie sprawdzamy zdalny odnośnik? Miejsce to znajduje się w następującym wierszu:

```
    if (head($target)) {
```

gdzie stosujemy funkcję `head` modułu `LWP::Simple`, by wysłać żądanie HTTP HEAD po URL, który zawiera zmienna `$target`. Jak zapewne wiecie, żądanie HEAD pyta o pewną metainformację w zdalnym dokumencie, nie prosi natomiast o sam dokument. (Jeżeli chcemy uzyskać cały dokument, wysyłamy żądanie GET). Żądanie HEAD pozwala nam jednak sprawdzić, czy dokument istnieje, bez potrzeby pobierania go w całości.

Funkcja `head` modułu `LWP::Simple` jest opisana w dokumentacji `POD`, dołączonej do modułu. Dokumentację tę możemy czytać z shella w taki sposób:

```
[jbc@andros jbc]$ man LWP::Simple
```

lub

```
[jbc@andros jbc]$ perldoc LWP::Simple
```

Jeżeli swoją kopię `LWP` zainstalowaliśmy w naszej osobistej kartotece, będziemy musieli pomóc programom `man` i `perldoc` w szukaniu dokumentacji. Jeżeli instalacja `LWP` znajduje się w kartotece:

```
/home/jbc/perl/lib/perl5/site_perl/5.005/i386-linux/
```

możemy spróbować użycia:

```
[jbc@andros jbc]$ perldoc /home/jbc/perl/lib/perl5/site_perl/5.005/i386-
linux/LWP/Simple.pm
```

wskazując programowi `perldoc` sam plik modułu i umożliwiając mu wyciągnięcie dokumentacji wprost z osadzonego opisu `POD`.

Wreszcie obecna wersja skryptu ma w procedurze `&convert` dodatkową logikę, która obsługuje zdalne odnośniki i umieszcza je tablicy odnośników do sprawdzenia, jaką zwraca procedura (przy założeniu, że zmienna `$check_remote` jest nastawiona na wartość prawdziwą):

```
if (/^http:/i) {
    # URL zaczyna się od 'http:'
    if (/^http:\/\/$escaped_hostname/io) {
        # lokalny odnośnik; konwertuj na lokalną nazwę
        # pliku, przygotowując konwersję poniżej
        s/^http:\/\/$escaped_hostname//io;
```

```
            } else {
                # zdalny odnośnik
                push @return_urls, $_ if $check_remote;
                next;
            }
        }
```

I to tyle! Ta wersja skryptu działa nieco wolniej, niż poprzednia, zwłaszcza jeżeli musi sprawdzić wiele odnośników wskazujących poza witrynę, ale wciąż wykonuje przyzwoitą pracę przy rozpoznawaniu w tej witrynie bardziej oczywistych przypadków popsutych odnośników.

Właściwy program do sprawdzania odnośników

Tyle się rozpisałem na temat rozwiniętych w tym rozdziale skryptów do sprawdzania odnośników, że w końcu trzeba się wziąć za omówienie „właściwego" programu do sprawdzania odnośników. W ostatecznej wersji skryptu sprawdzającego odnośniki porzucimy wątpliwy pomysł wyciągania atrybutów SRC i HREF ze stron HTML za pomocą prostych wzorców *regex*. Odrzucimy także testy `-e` w lokalnym systemie plików przy ustalaniu obecności lub nieobecności lokalnych obrazów i plików HTML. Ta wersja skryptu będzie wędrować przez witrynę jak pajęczy program motoru wyszukiwania, sprawdzając każdy odnośnik za pomocą żądania HTTP wydanego za pośrednictwem `LWP`.

Bez dalszych ceremonii przedstawiamy ten skrypt w przykładzie 11.4. Dzieje się tam bardzo wiele (w tym — mnóstwo magicznych wrażeń dostarczają importowane moduły), ale wszystko to omówimy szczegółowo po obejrzeniu samego skryptu.

Przykład 11.4. Skrypt sprawdzający za pomocą LWP „zepsucie" odnośników

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

# link_check3.plx

# Trzecia wersja programu do sprawdzania odnośników HTML.
# Zaczyna od URL (wymaganego jako argument wiersza poleceń),
# i wędruje przez całą witrynę (lub jej większą część, jaką
# może dosięgnąć poprzez odnośniki, dostępne rekurencyjnie
# ze strony początkowej), sprawdzając za pomocą żądań GET
# i HEAD z LWP::UserAgent, czy działają wszystkie atrybuty
# HREF i SRC. Składa raport o popsutych odnośnikach.

use strict;
use LWP::UserAgent;
use HTTP::Request;
use HTML::LinkExtor;
use URI::URL; # wymagany przez HTML::LinkExtor, przy wywołaniu z base

my $from_addr  = 'wasz@adres.tu'; # adres e-mail dla agenta użytkownika
my $agent_name = 'link_check3.plx';  # nazwa, z którą zgłasza się robot

my $delay      = 1;    # liczba sekund pomiędzy żądaniami
my $timeout    = 5;    # liczba sekund do wygaśnięcia żądania
my $max_pages  = 1000; # liczba stron do przetworzenia
my $webify     = 1;    # wyprowadzić wyjście www?
my $debug      = 1;    # wyprowadzić debugging do STDERR?

my %bad_links;    # "hash tablic" z kluczami tworzonymi przez URLe,
                  # począwszy od $start_base, i wartościami tworzonymi
                  # przez listy popsutych odnośników w tych stronach.
```

```
my %good;         # hash, który odwzorowuje URLe na 0 lub 1
                  # (dobre lub złe). Zapamiętuje wyniki badań,
                  # wykonanych poprzednio, tak że nie muszą
                  # być powtarzane dla następnych stron.

my @queue;        # tablica, która zawiera listę URLi
                  # (pod $start_url) do sprwadzenia.

my $total_pages;  # zawiera liczbę stron przetworzonych dotychczas.

# Koniec konfiguracji. Początek właściwego skryptu.

my $last_request = 0;    # czas ostatniego żądania, dla $delay

# najpierw skonstruuj agent użytkownika

my $ua = LWP::UserAgent->new;
$ua->agent("$agent_name " . $ua->agent);
$ua->from($from_addr);
$ua->timeout($timeout); # nastaw interwał wygaśnięcia

# teraz przetwórz argument wiersza poleceń

my $start_url = shift or
    die "Usage: $0 http://start.url.com/\n";

my($success, $type, $actual) = &check_url($start_url);

unless ($success and $type eq 'text/html') {
    die "The start_url isn't reachable, or isn't an HTML file.\n";
}

$good{$start_url} = 1;
push @queue, $start_url;
my $start_base         = $start_url;
$start_base            =~ s{/[^/]*$}{/}; # utnij po ostatnim '/'
my $escaped_start_base = quotemeta $start_base;

while (@queue) {
    ++$total_pages;
    if ($total_pages > $max_pages) {
        warn "stopped checking after reaching $max_pages pages.\n";
        --$total_pages; # zmniejsz, by liczba w raporcie była dobra
        last;
    }
    my $page = shift @queue;
    &process_page($page); # może dodawać nowe pozycje do @queue
}

# drukuj raport

my $time = localtime;

if ($webify) {

    # drukuj wersję HTML tego raportu

    print <<EndOfText;
<HTML>
<HEAD>
<TITLE>$start_url $0 report</TITLE>
</HEAD>
```

```
<BODY>

<H2 ALIGN="center">$start_url $0 report</H2>

<P ALIGN="center"><STRONG>Report created at $time</STRONG></P>

<P>
EndOfText

    foreach my $file (sort keys %bad_links) {
        print "<P><STRONG><A HREF=\"$file\">$file</A></STRONG><BR>\n";
        foreach my $target (sort @{ $bad_links{$file} }) {
            print " <A HREF=\"$target\">$target</A><BR>\n";
        }
        print "\n</P>\n\n";
    }
    print "</BODY></HTML>\n";

} else {

    # drukuj tylko czysto tekstową wersję raportu

    print "$start_url $0 report\n";
    print "Report created at $time\n\n";

    foreach my $file (sort keys %bad_links) {
        print "$file:\n";
        foreach my $target (sort @{ $bad_links{$file} }) {
            print " $target\n";
        }
        print "\n";
    }
}

# koniec właściwego skryptu. dalej są procedury.

sub check_url {

    # Sprawdź, że URL jest ważny, stosując metodę HEAD (i GET,
    # jeżeli HEAD nie skutkuje). Zwraca 3-elementową tablicę:
    # ($success, $type, $actual).

    my $url = shift;

    if ($debug) { warn " checking $url...\n"; }
    unless (defined $url) {
        return ('', '', '');
    }

    sleep 1 while (time - $last_request) < $delay;
    $last_request = time;
    my $response = $ua->request(HTTP::Request->new('HEAD', $url));
    my $success = $response->is_success;
    unless ($success) {
        # spróbuj żądania GET; pewnym stacjom nie podoba się HEAD
        sleep 1 while (time - $last_request) < $delay;
        $last_request = time;
        $response = $ua->request(HTTP::Request->new('GET', $url));
        $success = $response->is_success;
    }
    if ($debug) {
        if ($success) {
            warn "  ...good.\n";
```

```
        } else {
            warn "  ...bad.\n";
        }
    }
    my $type = $response->header('Content-Type');
    my $actual;
    if ($success) {
        $actual = $response->base; # przekierowano nas?
    }
    return ($success, $type, $actual);
}

sub process_page {

    # Wołana z pojedynczym argumentem, który jest stroną pod
    # $start_base, wymagającą przetworzenia. Strona ta zostanie
    # (1) pobrana przez GET, (2) przetworzona ze względu na
    # odnośniki, które zawiera, i (3) poddana sprawdzaniu samych
    # odnośników (popsute dopisuje się do %bad_links, a te, które
    # pod $start_base wskazują na ważne, niesprawdzone pliki HTML,
    # dodaje się do @queue). Procedura nie ma wartości zwracanej.

    my $page = shift;
    return unless defined $page;

    if ($debug) { warn "processing $page for links\n"; }

    sleep 1 while (time - $last_request) < $delay;
    $last_request = time;
    my $response = $ua->request(HTTP::Request->new('GET', $page));
    unless ($response->is_success
        and $response->header('Content-Type') eq 'text/html') {
        # dziwne, skoro przechodzi przez testy
        # żądania HEAD, by trafić do @queue
        $good{$page} = 0;
        return;
    }
    my $base = $response->base;
    unless ($base =~ /$escaped_start_base/o) {
        # wygląda na to, że przekierowano nas ze $start_base
        return;
    }
    my $parser = HTML::LinkExtor->new(undef, $base);
    $parser->parse($response->content);
    my @links = $parser->links;
    foreach my $linkarray (@links) {
        my ($tag, %links) = @{$linkarray};
        if ($tag =~ /^(a|img|frame)$/) {
            TARGET: while (my($attr, $target) = each %links) {
                if ($attr =~ /^(href|src|lowsrc)$/) {
                    # to są pozycje $target, o które
                    # nam chodzi.
                    next TARGET unless $target =~ /^(?:https?|ftp):/;
                    $target =~ s/#.*//; # usuń końcowe #kotwiczki
                    if (exists $good{$target}) {
                        # ten już widzieliśmy
                        if ($good{$target}) {
                            # wiadomo, że jest dobry
                            next;
                         } else {
                            # wiadomo, że jest zły
                            push @{ $bad_links{$base} }, $target;
                        }
```

```
                    } else {
                        # tego jeszcze nie widzieliśmy
                        my($success, $type, $actual)
                            = &check_url($target);
                        unless ($success) {
                            $good{$target} = 0;
                            push @{ $bad_links{$base} }, $target;
                            next TARGET;
                        }
                        $good{$target} = 1;

                        if (defined $type
                            and $type eq 'text/html'
                            and defined $actual
                            and $actual =~ /$escaped_start_base/o) {

                            push @queue, $target;
                        }
                    }
                }
            }
        }
    }
}
```

Zrobiliście dostateczne postępy w nauce języka Perl, by dla zrozumienia większej części tego skryptu nie potrzebować wyjaśnień, które tłumaczą go wiersz po wierszu. Oto najważniejsze punkty.

Na szczycie skryptu wciągamy wszystkie moduły, jakie wraz z kilkoma nowymi (`HTTP::Request`, `HTML::LinkExtor` i `URI::URL`) wykorzystamy, by — jeśli można tak powiedzieć — unieść ciężar, który stanowi *parsing* plików HTML i ekstrakcja odnośników. Poprzedni program do sprawdzania odnośników posługiwał się przy tym prostym wyrażeniem regularnym, ale te moduły robią to znacznie bardziej rygorystycznie, uwzględniając również takie rzeczy, jak znacznik `<BASE>` w nagłówkach dokumentu.

W sekcji konfiguracyjnej mamy przypisania do zmiennych skalarnych, jakie wykorzystamy w skrypcie — wśród nich przypisanie do zmiennej `$delay`, którą `LWP` uwzględnia przy kolejnych żądaniach (by uniknąć bombardowania obcego serwera WWW przez wielką liczbę niemal równoczesnych żądań HTTP). Definiujemy również skalar `$timeout`, po którym skrypt rozpoznaje popsuty odnośnik i przechodzi do następnego. Zmienna `$max_pages` ma zapobiec utknięciu przez skrypt w nieskończonej pętli, co mogłoby się zdarzyć, jeżeliby skrypt CGI w serwerze generował nieskończony szereg odnośników.

Składnia zorientowana obiektowo

Pierwszą naprawdę nową częścią tego skryptu jest miejsce, gdzie konstruujemy nowy obiekt agenta użytkownika za pomocą następującego wiersza:

```
my $ua = LWP::UserAgent->new;
```

Składnia o tym dziwnym wyglądzie, którego jeszcze nie widzieliście w tej książce, jest charakterystyczna dla czegoś, co określa się jako *programowanie zorientowane obiektowo* lub *OOP* (od ang. *Object-Oriented Programming*). Moduł `LWP::Simple`, którego użyliśmy w drugiej wersji

programu do sprawdzania odnośników, pozwalał nam wykonać proste transakcje HTTP z programu Perl, bez żadnej znajomości OOP. Jednak, aby skorzystać z bogatszych własności `LWP`, musimy użyć zorientowanego obiektowo interfejsu `LWP::UserAgent`.

Nie trzeba znać programowania zorientowanego obiektowo, by wykorzystywać zorientowane obiektowo moduły, napisane przez innych. Mimo to warto poświęcić nieco czasu, żebyście chociaż trochę zapoznali się z OOP, zanim zajmiemy się innymi kwestiami. Pomoże Wam to lepiej korzystać z OOP w wykonaniu kogoś innego, a to z kolei pozwoli Wam wykorzystać wiele zorientowanych obiektowo modułów Perl (jak `LWP::UserAgent`), które znajdziecie w CPAN. Ramka „Zorientowany obiektowo Perl" jest krótkim wprowadzeniem do wspomnianego zagadnienia.

Gdy tworzymy nowy obiekt agenta użytkownika za pomocą `LWP::UserAgent->new`, musimy wstawić ten obiekt do zmiennej skalarnej, by później mieć do niego dostęp. W tym przypadku wstawiamy obiekt agenta użytkownika do zmiennej skalarnej `$ua`.

Być może dziwicie się, jak zmienna skalarna, która wedle Waszej wiedzy może zawierać pojedynczą wartość, zdolna jest pomieścić coś, co jest tak złożone i pełne własności, jak obiekt. Jeżeli jednak pomyślicie nad tym przez chwilę, to założę się, że będziecie w stanie wywnioskować, jak robi to Perl. Kiedy widzieliśmy, jak Perl zmienił coś złożonego w skalar, tak że mogliśmy to przechowywać i poddawać manipulacjom? Słusznie! Potrzebne są tu odniesienia!

Obiekt zwracany przez metodę `new` klasy `LWP::UserAgent` faktycznie jest *odniesieniem*. Aby uzyskać dostęp do *metod* tego obiektu (to znaczy — umieścić informację w obiekcie lub ją z niego odzyskać), musimy rozebrać odniesienie. W tym celu używamy trzeciej (i ostatniej, o której dowiemy się z tego rozdziału) części składni dla odniesień w języku Perl: *strzałki rozbioru odniesienia*, lub `->`. (Nie należy jej mylić z *operatorem zastępującym przecinek* `=>`, używanym do rozgraniczania kluczy i wartości hashu. Prócz przypadkowego podobieństwa, nic ich ze sobą nie łączy).

Następne trzy wiersze skryptu `link_check3.plx` dobrze pokazują zastosowanie strzałki rozbioru odniesienia:

```
$ua->agent("$agent_name " . $ua->agent);
$ua->from($from_addr);
$ua->timeout($timeout); # nastaw interwał wygaśnięcia
```

W pierwszym z tych wierszy wywołujemy metodę obiektu `$ua` o nazwie `agent`, by wyjąć zawartość ze zmiennej `$agent_name` (zdefiniowanej w sekcji konfiguracyjnej skryptu), skleić ją z domyślną wartością zwracaną przez tę samą metodę `agent` i umieścić całość z powrotem we własności `agent` obiektu. Metoda `from` umieszcza w obiekcie skalar `$from_adrr` (także zdefiniowany w sekcji konfiguracyjnej skryptu), a metoda `timeout` nastawia wartość `timeout` obiektu.

Wszystkie te metody zostały opisane w dokumentacji `LWP::UserAgent`, która powinna być dostępna przez polecenie `man LWP::UserAgent` (lub `perldoc /path/to/local/copy/LWP/UserAgent.pm`, jeżeli musieliście zainstalować `LWP` we własnej kartotece użytkownika).

Skrypt zdejmuje następnie początkowy URL z wierzchołka tablicy `@ARGV` (czyli bierze pierwszy argument dostarczony w wierszu poleceń i wstawia go do `$start_url`) lub ginie z komunikatem „sposób użycia", jeżeli argument nie został podany. Ten `$start_url` jest następnie przetwarzany

Zorientowany obiektowo Perl

Programowanie zorientowane obiektowo jest stosunkowo niedawną innowacją w świecie programowania komputerów. Jedni je lubią, inni nie, ale bez względu na to, co można o nim sądzić, wielu doświadczonych programistów Perla wyznaje religię OOP i używa takiego programowania przy tworzeniu modułów. Aby skorzystać z tych modułów, powinniście trochę wiedzieć na temat OOP.

OOP jest jeszcze jedną bronią programistów w nieustannej wojnie przeciw złożoności. Jak procedury, które zachowują swoje wewnętrzne zmienne dla siebie i całą informację przekazują do środka i na zewnątrz przez parametry wywołań i wartości zwracane, tak programowanie zorientowane obiektowo pomaga programistom ukrywać w programach złożoność szczegółów niskiego poziomu Jeżeli jednak dobrze napisana procedura jest jak czarna skrzynka, to programowanie zorientowane obiektowo przypomina czarną skrzynkę, która ma zamek wprawiony w drzwiczki i małą skrzyneczkę w bocznej ściance na wiadomości przekazywane do środka i na zewnątrz. Możemy powiedzieć, że OOP znacznie bardziej stanowczo powstrzymuje próby „grzebania we wnętrznościach programu".

OOP osiąga to przez dodatkowy poziom abstrakcji, który oddziela Was (kogoś, kto wykorzystuje pewien zorientowany obiektowo kod) od tego kodu. Aby zrobić coś za pomocą zorientowanego obiektowo modułu Perla, trzeba użyć specjalnego, zorientowanego obiektowo interfejsu, który jest starannie oddzielony od wewnętrznej implementacji, jaka wykonuje faktyczną pracę.

Związany jest z tym pewien nowy żargon. Kiedy używacie zorientowanego obiektowo programu, robicie to przez *wywołanie metod*. Wywołujecie metody *czegoś*, a jedną z rzeczy, której metody wywołujecie, jest *klasa*. Wywołanie metody tej klasy nazywa się po prostu *wywołaniem metody klasy*. Jeżeli wywołacie pewną metodę klasy, zwaną *konstruktorem*, uzyskacie od niej *obiekt*. Gdy macie ten obiekt, wywołujecie jego *metody obiektu*.

W ujęciu Perla klasy wyglądają jak nazwy modułów, które już widzieliście (np. `Some::Class`). Obiekt zwracany przez konstruktor jest umieszczany w zmiennej skalarnej. Metoda jest wywoływana za pomocą ciekawego symbolu strzałki (`->`), który łączy nazwę klasy (lub obiekt) po lewej stronie z nazwą wywoływanej metody po prawej (więcej o symbolu strzałki dowiecie się później w tym rozdziale). Metoda, która ma argumenty, otrzymuje te argumenty na liście nawiasowej po nazwie metody, podobnie jak procedura.

Oto kilka przykładów tego, jak wygląda zorientowany obiektowo Perl — wraz z komentarzami oddającymi sposób mówienia programistów:

```
# wywołaj metodę klasy, by zwrócić nowy obiekt
my $obj = Some::Class->new;

# wywołaj metodę obiektu, by przypisać atrybut
$obj->color('green');

# wywołaj metodę obiektu, by zwrócić atrybut
my $color = $obj->color;
```

przez procedurę `&check_url`, która zwraca trzyelementową listę `($success, $type, $actual)`. Jeżeli spojrzymy na procedurę `&check_url`, widzimy, że po pewnych przygotowaniach procedura ta wykona następujący ciekawy wiersz:

```
sleep 1 while (time - $last_request) < $delay;
```

Tak jak poprzednio, gdy tworzyliśmy bloki `if` w jednym wierszu (usuwając nawiasy klamrowe i stawiając instrukcję `if` na końcu wiersza), możemy teraz postąpić z pętlą `while`. Wiersz ten spowoduje, że program będzie pozostawać w uśpieniu przez jedną sekundę, dopóki wartość zwracana przez funkcję

`time` (która zwraca liczbę sekund od czasu Epoki) — pomniejszona o wartość zmiennej `$last_request` (która zawiera czas ostatniego żądania skryptu) — jest mniejsza, niż wartość zmiennej konfiguracyjnej `$delay`. Inaczej mówiąc, skrypt będzie czekać w tym miejscu, aż minie `$delay` sekund, od kiedy zmienna `$last_request` była aktualizowana przez wartość bieżącego czasu. Jeżeli stosujemy domyślną wartość `1` w `$delay`, oznacza to, że skrypt wysyła tylko jedno żądanie HTTP na sekundę, co zapobiega zalaniu obcego serwera WWW przez setki lub tysiące niemal równoczesnych żądań.

Sprawdzamy zdalne odnośniki

Gdy skrypt wychodzi z uśpienia w procedurze `&check_url` i zastępuje wartość zmiennej `$last_request` wartością bieżącego czasu, pojawia się następujący bardzo ciekawy wiersz:

```
my $response = $ua->request(HTTP::Request->new('HEAD', $url));
```

Jeżeli poświęcicie kilka sekund, żeby zastanowić się nad tym wierszem, to okaże się, że jest on dość prosty. Mając URL, który zawiera się w zmiennej `$url`, wysyłamy dla niego żądanie HEAD i odpowiedź na to żądanie umieszczamy w nowym obiekcie o nazwie `$response`. Patrząc najpierw na prawą stronę wyrażenia i wgłębiając się w zagnieżdżone nawiasy, widzicie, że faktycznie rozpoczynamy od utworzenia nowego obiektu, wywołując metodę `new` klasy `HTTP::Request`. Metoda ta otrzymuje od nas dwa argumenty: łańcuch `'HEAD'` i `$url`, który został przekazany do procedury `&check_url`. Tak postępuje się, by utworzyć obiekt `HTTP::Request`, który dla danej wartości `$url` reprezentuje żądanie HEAD (jeżeli Was to ciekawi, więcej szczegółów zawiera `man HTTP::Request`).

Nie trudzimy się, żeby umieścić ten obiekt `HTTP::Request` w zmiennej, lecz natychmiast przekazujemy go do metody `request` wywołanej dla obiektu `LWP::UserAgent`, który utworzyliśmy u góry skryptu. Metoda `request` coś zwraca — i to coś (jak wspomnieliśmy poprzednio) jest innym obiektem. Aby zdobyć szczegółowe informacje na ten temat, przeczytajcie odpowiednią stronę podręcznikową (w tym przypadku `man LWP::UserAgent`).

W pojedynczym wierszu utworzyliśmy jeden obiekt, użyliśmy go jako argumentu metody wywołanej dla drugiego obiektu i stąd zwróciliśmy trzeci obiekt. Jak już zapewne widzicie, orientacja obiektowa jest jedną z tych rzeczy, nad którymi trudno zapanować. Jednak dopóki uważnie czytamy dokumentację modułu i doprowadzamy do porządku swoją składnię OOP mimo wszelkich trudności, możemy być w zasadzie pewni, że kod będzie robić to, co do niego należy.

Następnie wywołujemy metodę `is_success` naszego obiektu `$response`:

```
my $success = $response->is_success;
```

Metoda `is_success` zwraca wartość prawdziwą, jeżeli żądanie, które wytworzyło ten obiekt `$response`, generuje kod pomyślnej odpowiedzi HTTP ze zdalnego serwera, a w przeciwnym razie — fałszywą. W naszym przypadku, jeżeli `$success` ma wartość fałszywą, wykonujemy blok `unless`, w którym powtarzamy żądanie, lecz tym razem metodą `GET`, zamiast `HEAD` (ponieważ niektóre serwery są tak skonfigurowane, że odrzucają żądania `HEAD`, chociaż bez żadnych trudności obsługują żądania `GET`). Po wykonaniu żądania `GET` jeszcze raz dla uzyskanego przez nie obiektu `$response` wywołujemy metodę `is_success`, umieszczając rezultat w zmiennej `$success`. Wiemy więc już, czy można pomyślnie żądać danego URL.

Czy wykonaliśmy żądanie GET, czy też nie, wywołujemy teraz dla tego obiektu $response metodę header, podając do niej argument 'Content-Type', aby powiedzieć jej, że jest to ten nagłówek, który chcemy wydobyć z odpowiedzi serwera. Nagłówek ten wstawiamy następnie do zmiennej $type:

```
my $type = $response->header('Content-Type');
```

Nawiasem mówiąc, musimy uzyskać nagłówek 'Content-Type' i zwrócić go z procedury &check_url, tak aby kod, który woła tę procedurę, wiedział, czy dostaliśmy z powrotem stronę HTML, bo wtedy trzeba w niej odszukiwać i sprawdzić kolejne odnośniki.

Następnie wywołujemy metodę base, która zwraca bazowy URL dokumentu, jaki nam został przekazany. Jak wskazuje komentarz, mimo że znamy URL, który został podany jako argument do procedury &check_url, to jednak nie wiemy, czy jest to strona ostatecznie przez nas pobrana. Jedna z ciekawszych rzeczy w zachowaniu LWP::UserAgent polega na tym, że ten agent wędruje za przekierowaniami, tak więc strona, na którą ostatecznie patrzymy, nie musi być stroną, jakiej żądaliśmy na początku. (Jest to doskonały przykład czegoś, o czym na ogół nie myślimy, próbując samodzielnego kodowania takich funkcji, jakie posiada LWP). Umieszczamy zwrócony URL bazowy dokumentu w zmiennej $actual:

```
my $actual;
if ($success) {
    $actual = $response->base; # przekierowano nas?
}
```

Wreszcie zwracamy trzyelementową wartość przekazywaną przez procedurę przy powrocie:

```
return ($success, $type, $actual);
```

Praca odbywa się teraz w głównej części skryptu, gdzie sprawdzamy wartości $success i $type zwrócone przez URL — przetworzony przez &check_url jako pierwszy — i giniemy z komunikatem o błędzie, jeżeli wartości te nie wyglądają właściwie:

```
unless ($success and $type eq 'text/html') {
    die "The start_url isn't reachable, or isn't an HTML file.\n";
}
```

Jeśli skrypt nadal działa, umieszczamy zapis dla tej początkowej strony w %good. Następnie wstawiamy $start_url na koniec kolejki @queue, która jest tablicą stron, jakie będziemy przetwarzać:

```
push @queue, $start_url;
```

Przetwarzamy kolejkę

Przypisujemy teraz $start_url do nowej zmiennej skalarnej o nazwie $start_base i używamy podstawieniowego wyrażenia regularnego, by usunąć ze $start_base wszystko, co znajduje się po ostatnim znaku /. Można też powiedzieć, że w $start_base będzie ostatecznie znajdować się URL kartoteki, która zawiera stronę początkową. Stosujemy też quotemeta, by uniknąć wszelkich metaznaków *regex* w $start_base, a rezultat umieszczamy w $escaped_start_base:

```
my $start_base         = $start_url;
$start_base            =~ s{/[^/]*$}{/}; # utnij po ostatnim '/'
my $escaped_start_base = quotemeta $start_base;
```

Krocząc poprzez strony w `@queue`, będziemy posługiwać się zmienną `$escaped_start_base` przy rozstrzyganiu, czy powinniśmy wędrować za znalezionymi odnośnikami. Skrypt sprawdza wszystkie znalezione odnośniki pod względem „zepsucia", ale zwracane strony będzie przeglądać w poszukiwaniu kolejnych odnośników tylko wtedy, gdy te strony mają URL, który zaczyna się od `$start_base`. (Jeżeli skrypt nie wprowadzi takiego rozróżnienia, może wywędrować na wycieczkę połączoną ze sprawdzaniem odnośników w całej Sieci).

Wreszcie w kluczowym miejscu skryptu przetwarzamy każdą ze stron w kolejce:

```
while (@queue) {
    ++$total_pages;
    if ($total_pages > $max_pages) {
        warn "stopped checking after reaching $max_pages pages.\n";
        --$total_pages; # zmniejsz, by liczba w raporcie była dobra
        last;
    }
    my $page = shift @queue;
    &process_page($page); # może dodawać nowe pozycje do @queue
}
```

Pętla ta sprawdza najpierw, czy skrypt doszedł do granicy zapamiętanej w zmiennej konfiguracyjnej `$max_pages`, a jeżeli tak jest, to opuszcza on tę pętlę za pomocą polecenia `last`. W przeciwnym razie pętla zdejmuje z początku tablicy `@queue` pierwszą stronę i wykonuje na niej procedurę `&process_page`.

Procedura `&process_page` nie zwraca żadnej wartości, ale być może modyfikuje tablicę `@queue`, wstawiając do niej nowe zapisy, jak zobaczymy za chwilę. Takie „działanie na odległość" jest zapewne w porządku, gdy stosujemy je z umiarem, jednak właściwie powinno zostać zaniechane, jeżeli można go uniknąć w interesie zrozumiałości i łatwości utrzymania kodu.

Większa część procedury `&process_page` powinna wyglądać obecnie dość jasno. Procedura pozostaje w uśpieniu przez odpowiedni okres, tworzy obiekt `$response` przez wywołanie metody `request` obiektu `$ua` i sprawdza tę odpowiedź, by ustalić, czy transakcja była pomyślna. Następnie wywołuje metodę `base` obiektu `$response` i sprawdza, czy URL bazowy strony, do której doszliśmy, nie znajduje się w rzeczywistości poza `$start_base`. (Nie powinien, gdyż w przeciwnym razie URL nie jest wstawiany na koniec tablicy `@queue`, ale być może gdzieś po drodze znów przekierowano nas na inną stronę).

Przy sposobności zwróćmy uwagę na użycie modyfikatora `/o` w wyrażeniu regularnym, które należy do tego wiersza:

```
unless ($base =~ /$escaped_start_base/o) {
```

Jak wiemy, modyfikator `/o` sprawia, że Perl kompiluje to wyrażenie regularne tylko raz, mimo że wzorzec wyszukiwania zawiera zmienną, której zawartość mogłaby zmieniać się pomiędzy wywołaniami wyrażenia. W zamian za przyrzeczenie, że zawartość wzorca wyszukiwania nie ulegnie zmianie, skrypt będzie działać trochę szybciej, niż wtedy, gdy wyrażenie jest rekompilowane, ilekroć ma być wykonane.

Następnie mamy kolejną, zorientowaną obiektowo rozrywkę, którą zapewnił nam Gisle Aas, autor modułu' `HTML::LinkExtor`. Oto kod, który wykorzystamy do precyzyjnej, „właściwej" ekstrakcji odnośników ze sprawdzanych przez skrypt dokumentów HTML:

```
my $parser = HTML::LinkExtor->new(undef, $base);
$parser->parse($response->content);
my @links = $parser->links;
```

Spróbujcie nie pozostawać w tyle: dla `HTML::LinkExtor` tworzymy nowy obiekt `$parser` przez wywołanie metody `new`. Przekazujemy do niej dwa argumenty, chociaż pierwszy jest faktycznie wartością niezdefiniowaną, którą zwraca funkcja Perla `undef`. Argument ten przekazujemy wyłącznie ze względu na jego miejsce. Drugi argument to `$base` zwróconego wcześniej obiektu `$response`.

Gdy utworzyliśmy nowy obiekt `$parser`, wywołujemy jego metodę `parse`, podając do niej zawartość pobranej strony (tę zawartość uzyskaliśmy, wywołując metodę `content` dla obiektu `$response`). Następnie wywołujemy metodę `links` obiektu `$parser`, która zwraca „tablicę tablic" (w rzeczywistości jest to tablica odniesień do tablic), a w niej zawarta jest informacja o wszystkich znacznikach odnośników dokumentu (czyli wszystkich znacznikach, które przez atrybuty `SRC` lub `HREF` wskazują gdzie indziej).

Następnie przetwarzamy odniesienia do tablic, krocząc w pętli `foreach` wzdłuż szeregu elementów tablicy `@links`:

```
foreach my $linkarray (@links) {
    my ($tag, %links) = @{$linkarray};
```

Wewnątrz pętli `foreach` rozbieramy bieżące odniesienie do tablicy (zawarte w `$linkarray`) za pomocą składni rozbioru odniesień, którą widzieliście wcześniej w tym rozdziale — `@{$linkarray}`. Otrzymana w ten sposób tablica jest dość zabawną strukturą danych, która wygląda tak: (znacznik, atrybut, wartość, atrybut, wartość, ...). (Więcej informacji na ten temat można znaleźć dzięki `man HTML::LinkExtor`). Przez przypisanie tej listy do `($tag, %links)` tworzymy skalar i hash dla bieżącego znacznika, z czego możemy skorzystać przy dalszym przetwarzaniu.

Przetwarzanie to zaczyna się od użycia wyrażenia regularnego, żeby stwierdzić, czy ten znacznik (`HTML::LinkExtor` zwraca go jako znacznik zapisany małymi literami) jest znacznikiem `<a...>`, czy `<img...>`:

```
if ($tag =~ /^(a|img|frame)$/) {
```

Jeżeli jest którymś z nich, włącza się następujący kod:

```
TARGET: while (my($attr, $target) = each %links) {
```

W tym wierszu jest kilka nowych rzeczy. Przede wszystkim *etykieta* (jest nią `TARGET:`), która oznacza początek pętli `while`. To oznaczenie przyda się nam za chwilę, ponieważ pozwoli nam odwoływać się do tej pętli przez nazwę, gdy będziemy chcieli wrócić na szczyt pętli za pomocą `next`. Etykieta może być dowolna, ale na mocy konwencji powinna zawierać TYLKO WIELKIE litery.

Druga nowość w tym wierszu polega na zastosowaniu funkcji `each`, która jest podobna do funkcji `keys` — służącej do tworzenia listy wszystkich kluczy hashu — jednak działa nieco inaczej. Zamiast zwracać klucze, `each` zwraca pary klucz-wartość. Ponadto zamiast zwracać całą listę kluczy (jak robi to `keys`), `each` zwraca pojedynczą parę klucz-wartość („następną" parę klucz-wartość według mniej więcej przypadkowego, wewnętrznego uporządkowania hashu) za każdym razem, gdy ją wywołujemy. Jeżeli przypisanie wartości zwracanej przez `each` wstawimy do warunku pętli `while`, co zrobiliśmy w tym przypadku, to przejdziemy poprzez cały hash.

W końcu mamy resztę procedury `&process_page`, gdzie przetwarzamy każdy z atrybutów każdego z odnośników na przetwarzanej stronie:

```
                    if ($attr =~ /^(href|src|lowsrc)$/) {
                        # to są pozycje $target, o które
                        # nam chodzi.
                        next TARGET unless $target =~ /^(?:https?|ftp):/;
                        $target =~ s/#.*//; # usuń końcowe #kotwiczki
                        if (exists $good{$target}) {
                            # ten już widzieliśmy
                            if ($good{$target}) {
                                # wiadomo, że jest dobry
                                next;
                             } else {
                                # wiadomo, że jest zły
                                push @{ $bad_links{$base} }, $target;
                            }
                        } else {
                            # tego jeszcze nie widzieliśmy
                            my($success, $type, $actual)
                                = &check_url($target);
                            unless ($success) {
                                $good{$target} = 0;
                                push @{ $bad_links{$base} }, $target;
                                next TARGET;
                            }
                            $good{$target} = 1;

                            if (defined $type
                                and $type eq 'text/html'
                                and defined $actual
                                and $actual =~ /$escaped_start_base/o) {

                                push @queue, $target;
                            }
                        }
                    }
                }
            }
        }
    }
```

Ten fragment powinien wyglądać dość jasno — robimy w nim mniej więcej to samo, co w poprzedniej wersji naszego programu do sprawdzania odnośników. Warto zwrócić uwagę na to, jak używamy etykiety, aby wrócić na szczyt pętli w taki sposób:

```
next TARGET unless $target =~ /^(?:https?|ftp):/;
```

Ciekawy jest też sposób użycia procedury `&check_url` do przetwarzania każdego odnośnika, który nie znajduje się jeszcze w hashu `%good` (to znaczy — do przetwarzania odnośnika, który nie był wcześniej widziany i nie został sprawdzony). Ostatnia z różnic dotyczy sposobu wstawiania wartości URL, które reprezentują niewidziane wcześniej odnośniki spod `$start_base`, na koniec tablicy `@queue` do dalszego przetwarzania.

I to właściwie wszystko. Pozostała część skryptu po prostu drukuje raport w taki sam sposób, jak poprzednie programy do sprawdzania odnośników.

Gratulacje! Napisaliśmy zupełnie niezły program do sprawdzania odnośników — taki, który może szukać popsutych odnośników w witrynach znajdujących się gdziekolwiek w obrębie WWW, prowadząc przy tym *parsing* HTML w dość rygorystyczny sposób. Dowiedzieliśmy się, jak pobierać i instalować moduły z CPAN, oraz nauczyliśmy trzech spośród czterech ważnych części składni języka Perl, przeznaczonej do tworzenia i użycia odniesień. Wreszcie wiemy już dość dużo o zorientowanej obiektowo składni Perl, by należycie korzystać ze zorientowanych obiektowo modułów Perla. To naprawdę wiele jak na jeden rozdział.