

Web scraping for fun and profit

Ekstrakcja danych ze stron WWW



Marcin Hłybin

e-mail: marcin@hlybin.com

web: marcinhlybin.com

Tekst omawia zagadnienia związane z wyodrębnianiem danych ze stron WWW do automatyzacji pewnych zadań, czy analiz statystycznych. Do tego celu wykorzystany został język programowania Perl z modułami HTML::TreeBuilder do budowy struktury drzewiastej z kodu HTML oraz HTML::TableExtract do łatwej ekstrakcji danych z tabel.

III Dni Wolnego Oprogramowania
Bielsko-Biała

6 Marzec 2010

Spis treści

1 Wstęp	3
2 Przykład z życia wzięty.....	5
2.1 Pobranie zawartości strony	6
2.2 Budowa drzewa HTML.....	6
2.3 Parsowanie danych.....	6
2.4 Zapisanie wyników	7
3 HTML jako struktura drzewiasta.....	9
3.1 Przeszukiwanie drzewa.....	9
3.2 Atrybuty węzła.....	10
3.3 Wyświetlanie zawartości drzewa	10
3.4 Zastosowanie.....	11
4 Wyodrębnianie danych z tabel HTML	13
4.1 Nagłówki tabel.....	13
4.2 Wyszukiwanie konkretnej tabeli	14
4.3 Struktura drzewiasta	15
5 Podsumowanie	15
6 Bibliografia.....	15

1 Wstęp

W dzisiejszych czasach w Internecie można znaleźć wszystko. Niestety natłok informacji sprawia, że analizowanie danych nie jest wcale zadaniem łatwym. Strony napisane w języku HTML, pomimo standaryzacji znaczników, nie są ustandaryzowane po względem logicznym. Strony są tworzone z wykorzystaniem tabeli reprezentowanych znacznikami `<table></table>` lub elementów blokowych reprezentowanych znacznikami `<div></div>`. Pojawienie się metajęzyka XML oddzielającego wygląd stron WWW od treści było znaczącym krokiem naprzód, ponieważ współczesne języki programowania bez problemu radzą sobie z parsowaniem¹ i obróbką tak zapisanych danych. Problemem jest to, że nie każdy serwis WWW udostępnia dane zapisane w postaci XML i dane należy wyodrębnić bezpośrednio z plików HTML. Można tego dokonać na wiele sposób. Na przykład za pomocą:

- wyrażen regularnych, poszukując określonych ciągów znaków,
- budując strukturę drzewiastą znaczników HTML i poruszając się po niej wyodrębnić interesujące dane,
- używając metod ekstrakcji tabel HTML zagłębiając się w kolejne poziomy w celu pobrania danych ze wskazanych komórek tabeli.

Oczywiście można wykorzystywać te trzy metody jednocześnie, tak aby wyciągnięcie interesującego nas kawałka danych stało się zadaniem jak najprostszym.

Pojawia się naturalne pytanie: po co wyodrębnić dane ze stron WWW? Praktycznych przykładów jest mnóstwo. Oto kilka z nich:

1. Analiza wzrostu i spadku notowań funduszy inwestycyjnych. Na podstawie trendów można określić dogodny moment zakupu lub sprzedaży funduszu.
2. Śledzenie sprzedaży użytkownika serwisu Allegro, wyliczanie miesięcznych obrotów, stosunek towarów sprzedanych do niesprzedanych itp.
3. Obserwowanie zmian wersji oprogramowania na podstawie numeru wersji zamieszczanego na stronie WWW.

Do ekstrakcji danych ze stron WWW posłużymy się znakomitym, darmowym, skryptowym językiem programowania PERL². Cytując współtwórców, język ten pozwala, by rzeczy łatwe pozostały łatwymi,

¹ Analizowanie składni kodu źródłowego i jego reguł gramatycznych.

² *Practical Extraction and Reporting Language* – dosłownie *Praktyczny Język Wydobywania I Raportów*

a trudne były możliwe. Ponadto dostępnych jest mnóstwo modułów rozszerzających ten język zebranych w ramach repozytorium CPAN³. Do celów wyodrębniania danych posłużymy się modułami

1. HTML::TreeBuilder do budowania i parsowania drzewa znaczników HTML.
2. HTML::TableExtract do wyodrębniania danych z tabel HTML.
3. LWP, który służy jako API dla WWW wykorzystywane do przeprowadzania komunikacji protokołem HTTP, w szczególności do pobierania źródła stron WWW, parsowania formularzy HTML, obsługi ciasteczek (ang. *cookies*) itp.

³ Comprehensive Perl Archive Network – źródło dokumentacji oraz modułów języka PERL. Wyszukiwarka dostępna pod adresem <http://search.cpan.org/>.

2 Przykład z życia wzięty

Kilka dni temu znajomy poprosił mnie o wyciągnięcie danych na temat utworów granych w amerykańskich stacjach radiowych. Chodziło o rozgłośnie radiowe, które udostępniają swój sygnał także za pośrednictwem Internetu, za pomocą serwerów mediów strumieniowych, a konkretnie za pomocą programu *icecast*. Oczywiście prosząc mnie o to przy okazji przesłał URL, gdzie mogę dane na ten temat znaleźć: <http://cmn-ice.spacialnet.com/status.xml>.

Na stronie tej znajdują się ramki wyglądające tak jak poniżej:

Mount Point /1021
Stream Title: KPRI
Stream Description: KPRI
Content Type: audio/mpeg
Mount Start: Sun, 14 Feb 2010 10:26:07 -0800
Bitrate: 64
Current Listeners: 0
Peak Listeners: 1
Stream Genre: rock
Stream URL: www.kprifm.com
Current Song: Rod Stewart - Maggie May

Interesujące fragmenty to nazwa *Mount Point* oraz pole *Current Song*. Po obejrzeniu źródła strony można zobaczyć, że jedna ramka (a jest ich kilkadziesiąt) znajdują się w ramach znaczników `<div class='newscontent'></div>`, a poszczególne pola są zawarte w standardowej tabeli - w ramach znaczników `<table></table>` - składającej się z wierszy `<tr><</tr>` i kolumn `<td></td>`.

Wyciągnięcie interesujących nas danych składa się z kilku kroków:

1. Pobranie całej zawartości strony.
2. Zbudowanie drzewa HTML i znalezienie fragmentu w którym zawarta jest ramka z interesującymi nas informacjami.
3. Lokalizacja w znalezionej ramce fragmentu *Mount Point*, wyciągnięcie nazwy znajdującej się po znaku slash (/) oraz znalezienie fragmentu *Current Song*, w którym znajduje się wykonawca i tytuł obecnie odtwarzanego utworu.
4. Otworzenie pliku o nazwie odpowiadającej nazwie po znaku slash oraz zapisanie do niego wykonawcy i tytułu obecnie odtwarzanego utworu. Jeśli utwór już został wcześniej zapisany do pliku to go nie dodajemy kolejny raz.

2.1 Pobranie zawartości strony

Realizacja punktu pierwszego jest bardzo prosta. Wykorzystamy wcześniej wspomniany moduł LWP, a nawet jego uproszczoną wersję `LWP::Simple`, która implementuje metodę `get()` służącą do pobierania źródła strony WWW. Kod odpowiedzialny za tę operację:

```
use LWP::Simple;
my $xml = get("http://cmn-ice.spacialnet.com/status.xml") or die;
```

W zmiennej `$xml` zostanie zapisana zawartość strony. W przypadku niepowodzenia, skrypt zakończy swoje działanie.

2.2 Budowa drzewa HTML

Mając już źródło strony WWW należy zbudować drzewo, po którym będziemy się poruszać w celu znalezienia początku ramki z daną rozgłośnią radiową. Najpierw inicjalizujemy drzewo za pomocą:

```
use HTML::TreeBuilder;
my $tree = HTML::TreeBuilder->new_from_content($xml);
```

Gdzie pod zmienną `$xml` znajduje się wcześniej zapisane źródło strony WWW. Mając zbudowane drzewo będziemy się poruszać z góry na dół w celu znalezienia znacznika `<div>` z klasą `newscontent`, które oznacza początek interesującej nas ramki.

```
my @div = $tree->look_down(_tag => 'div', class => 'newscontent');
```

Wyniki zapisujemy w tablicy `@div`. Jest to tablica z tego względu, że ramek z rozgłościami jest kilkadziesiąt.

2.3 Parsowanie danych

W kolejnym kroku będziemy iterować w pętli po każdej szukając ciągu *Mount Point* oraz *Current Song*. Odpowiedzialny kod:

```
foreach my $div (@div) {
    $div->look_down(_tag => 'h3')->as_text() =~ m#Mount Point /(.+)#;
    my $name = $1 or next;
```

Tutaj także wykorzystamy metodę `look_down()` w celu przeszukiwania drzewa z góry na dół w poszukiwaniu znacznika `<h3>` w którym znajduje się *Mount Point*. Tym razem mamy do czynienia tylko z wyselekcjonowanym fragmentem drzewa HTML, w którym znajduje się konkretna, wcześniej znaleziona ramka. Dodatkowo korzystamy z metody `as_text()`, która wyświetla tekst zawarty między znacznikami HTML oraz przepuszczamy to przez wyrażenie regularne, którego zadaniem jest zapamiętać w standardowej zmiennej `$1` ciąg znaków znajdujący się po znaku slash. W kolejnej linii,

jeśli ciąg znaków został znaleziony, przypisujemy do zmiennej \$name, jeśli nie został znaleziony przechodzimy do kolejnej ramki.

Aby znaleźć nazwę utworu wyświetlamy całą zawartość ramki i za pomocą wyrażenia regularnego szukamy w niej konkretnej części tabeli HTML. Oczywiście można tutaj skorzystać z modułu HTML::TableExtract, ale wyrażenie regularne wydaje się sposobem łatwiejszym:

```
$div->as_HTML() =~ m#<td>Current Song:</td>
                <td class="streamdata">(.*?)</td></tr></table>#
```

W standardowej zmiennej \$1 zostanie zapisana nazwa wykonawcy i tytuł utworu. Przypisując te wartości do zmiennej \$song należy pamiętać, że mogą wystąpić znaki specjalne typu „&” lub „<”, które są znacznikami HTMLa. Wtedy, znaki takie są zamieniane na specjalny kod – w tym przypadku jest to „&” oraz „<”. Dlatego też należy skorzystać z modułu HTML::Entities udostępniającego metodę *decode_entities()*, która zamienia dokonuje konwersji specjalnych kodów HTMLa na normalne znaki.

```
use HTML::Entities;
my $song = decode_entities($1) or next;
```

2.4 Zapisanie wyników

Ostatnim zadaniem jest zapisanie nazwy wykonawcy i utworu do pliku. W tym celu należy utworzyć plik (jeśli jeszcze nie istnieje), sprawdzić czy utwór już nie istnieje w pliku i w końcu zapisać do niego interesujące nas dane.

Nazwy utworów mogą być zapisywane w różny sposób np. wielkimi i małymi literami. Pragnę przypomnieć, że systemy uniksowe rozróżniają wielkość liter, więc należy zestandaryzować nazewnictwo np. pierwsze litery wszystkich wyrazów pisane wielkimi literami, reszta pisana małymi literami. Oznacz to, że nazwa „WYKONAWCA – TUTAJ TYTUŁ UTWORU” zostanie zamieniona na „Wykonawca – Tutaj Tytuł Utworu”. Możemy to szybko osiągnąć za pomocą jednej linii kodu:

```
$song =~ s/\b(\w)(\w*)/\U$1\L$2/g;
```

Następnie otwieramy plik i jeśli wykonawca i utwór nie istnieją zapisujemy do niego dane:

```
`touch $name` unless -f $name;
open my $fh, "+<", $name;
print $fh "$song\n" unless grep(/\Q$song\E/i,<$fh>);
close $fh;
```

Wykorzystujemy systemową funkcję *touch*, która utworzy nam plik, jeśli jeszcze nie istnieje. Następnie tworzymy uchwyt do pliku ukryty pod zmienną \$fh i z pomocą funkcji *print* zapisujemy

dane. Przy okazji wykorzystujemy funkcję *grep*, która skanuje cały plik w poszukiwaniu wykonawcy i nazwy utworu. Jeśli już istnieje to *print* nie zostanie wykonany.

Tym oto sposobem w kilku liniach Perla utworzyliśmy parser stacji radiowych, który uruchamiany co 2 minuty, na przykład za pośrednictwem *crontab*⁴, będzie zbierał informacje o nadawanych utworach. Cały kod prezentuje się następująco:

```
#!/usr/bin/perl -w
use strict;
use LWP::Simple;
use HTML::TreeBuilder;
use HTML::Entities;

my $xml = get("http://cmn-ice.spacialnet.com/status.xml") or die;
my $tree = HTML::TreeBuilder->new_from_content($xml);
my @div = $tree->look_down(_tag => 'div', class => 'newscontent');

foreach my $div (@div) {
    $div->look_down(_tag => 'h3')->as_text() =~ m#Mount Point /(.)#;
    my $name = $1 or next;
    $div->as_HTML() =~ m#<td>Current Song:</td>
        <td class="streamdata">(.*?)</td></tr></table>#;
    my $song = decode_entities($1) or next;
    $song =~ s/\b(\w)(\w*)/\U$1\L$2/g;
    `touch $name` unless -f $name;
    open my $fh, "+<", $name;
    print $fh "$song\n" unless grep(/Q$song\E/i,<$fh>);
    close $fh;
}
```

Wyniki wyglądają następująco. Listing utworzonych plików:

```
$ ls
1021 kbfo kcxx kgnt kkdv kkma kmva koky kpri kqnu krkt ksux kvfx kznu
test wbuk whyb wkgo wkxa wmfq wnvw wqlz wtfm wusj wxjz brze kblq kddx
khit kkex kkyk kmvr kony kpzk kqpi ksdn kuic kxpz nvw2 wakr wdjr winn
wkkg wlce wmlv wogf wqtx wtrs wvic katy kblx kgim kkbz kkiq klzx knbz
kpld kqeo krdj ksna kupi kzhk wbls whtg wjkk wkoa wlst wnns wokk wsfq
wtyd wwwy
```

Zawartość jednego z plików:

```
$ cat winn
Doobie Brothers - Takin' It To The Streets
Genesis - Tonight, Tonight, Tonight
```

⁴ Usługa służąca do cyklicznego uruchamiania zadań.

3 HTML jako struktura drzewiasta

Języki znacznikowe mają z natury strukturę drzewiastą. Nie trudno sobie wyobrazić kod HTML w postaci:

```
<html>
  <head>
    <title>Tutaj tytuł strony</title>
  </head>
  <body>
    <ul>
      <li>Pierwszy na liście</li>
      <li>Drugi na liście</li>
      <li>Trzeci na liście</li>
    </ul>
  </body>
</html>
```

Widać, że korzeniem drzewa jest znacznik *<html>*, następnie rozgałęzia się na znaczniki *<head>* oraz *<body>*, który z kolei zawiera w sobie znacznik **, a ten rozgałęzia się na trzy znaczniki **. Istnieje oczywiście możliwość „płaskiego” przetwarzania kodu HTML, czy to za pomocą wyrażeń regularnych, czy też za pomocą tokenów, które nie zostały omówione w tej pracy. Budowanie i poruszanie się po drzewie HTML jest dużo ciekawsze.

Aby utworzyć drzewo wystarczy wywołać metodę *new_from_content()*:

```
my $root = HTML::TreeBuilder->new_from_content($html);
```

Mamy do dyspozycji szereg metod służących przeszukiwaniu drzewa i odczytywaniu atrybutów danego węzła (ang. *node*). Istnieje także możliwość modyfikacji pobranego kodu HTML, dzięki czemu możemy zmienić np. nazwę domeny we wszystkich linkach na stronie.

3.1 Przeszukiwanie drzewa

Wszystkie prezentowane metody można traktować zarówno w kontekście skalarnym jak i kontekście listy. W przypadku pierwszym zwracany jest pierwszy węzeł, który spełnia zadane kryteria. W drugim przypadku zwracana jest lista wszystkich węzłów spełniających zadane kryteria. Metody mogą być wywoływane zarówno na korzeniu drzewa (główny węzeł) jak i na dowolnym węźle.

Metoda *find_by_tag_name()* zwraca węzły zawierające podane znaczniki HTML. Przykładowo:

```
@naglowki = $root->find_by_tag_name('h1', 'h2');
```

W tablicy `@naglowki` znajdują się wszystkie węzły zawierające znaczniki `<h1>` oraz `<h2>`. Aby pracować na danych zawartych w tablicy należy dokonać iteracji przez wszystkie elementy np. za pomocą pętli `foreach`.

Metoda `find_by_attribute()` jest podobna do poprzedniej tylko pozwala szukać znaczników HTML o konkretnych atrybutach.

```
@notowania = $root->find_by_attribute('class','notowanie');
```

Można także przeszukiwać wszystkie, od danego węzła, w dół i w górę za pomocą metod `look_down()` oraz `look_up()`. Jak się można domyślić pierwsza metoda przeszukuje drzewo od danego węzła zagłębiając się w strukturę, a druga od danego węzła cofając się. Metoda przyjmuje parametry w postaci `atrybut=>wartość`. Na przykład:

```
@temat = $root->look_down(_tag => 'h1', class=>'temat');
```

Atrybut `_tag` jest specjalnym atrybutem, który oznacza nazwę znacznika HTML.

3.2 Atrybuty węzła

Dostępne są cztery metody dostarczające podstawowych informacji o węźle:

1. `$node->tag()`

Zwraca nazwę znacznika HTML np. `img`, `h1`.

2. `$node->parent()`

Zwraca nazwę węzła nadrzędnego (rodzica). W przypadku, gdy węzeł jest korzeniem drzewa, zostanie zwrócony `undef`.

3. `$node->content_list()`

Zwraca listę węzłów podrzędnych (dzieci) danego węzła. W przypadku braku węzłów podrzędnych lista będzie pusta.

4. `$node->attr(nazwa_atrybutu)`

Zwraca wartość podanego atrybutu znacznika HTML. W przypadku braku takiego atrybutu zwraca `undef`. Dla przykładu jeśli węzłem jest znacznik `` to po wywołaniu metody `$node->attr("src")` otrzymamy `plik.jpg`.

3.3 Wyświetlanie zawartości drzewa

Jeśli zaistnieje potrzeba wyświetlenia zawartości węzła, ponieważ na przykład dalszą obróbkę danych łatwiej będzie nam prowadzić przy użyciu wyrażeń regularnych, możemy wykorzystać dwie metody:

1. `$node->as_HTML()`

Wyświetla dany węzeł i wszystkie węzły podrzędne w postaci kodu HTML.

2. `$node->as_text()`

Tak jak powyżej, tylko pomija wszystkie znaczniki HTML. Zostanie wyświetlony sam tekst.

3.4 Zastosowanie

Powracając do naszego przykładowego kodu z początku tego rozdziału, wykorzystamy drzewo HTML do znalezienia kilku interesujących nas elementów takich jak tytuł i pozycje z listy.

```
#!/usr/bin/perl
use HTML::TreeBuilder;

my $root = HTML::TreeBuilder->new_from_content(<<EOF);
<html>
    <head>
        <title>Tutaj tytuł strony</title>
    </head>
    <body>
        <ul>
            <li>Pierwszy na liście</li>
            <li>Drugi na liście</li>
            <li>Trzeci na liście</li>
        </ul>
        <ul>
            <li>Czwarty na liście</li>
            <li>Piąty na liście</li>
        </ul>
    </body>
</html>
EOF

$root->eof( );
my $tytul = $root->find_by_tag_name('title');
my @lista = $root->look_down(_tag => 'ul');
print "tytuł: ".$tytul->as_HTML()."\n";

foreach my $ul (@lista) {
    @li=$ul->content_list();
    foreach my $pozycja (@li) {
        print $pozycja->as_text()."\n";
    }
}
```

Bezpośrednio w kodzie podaliśmy źródło przykładowej strony WWW. Zastosowaliśmy dodatkową metodę `$root->eof()`, która oznacza, że nasze drzewo jest gotowe i żadne nowe elementy nie będą do niego dodawane.

Następnie za pomocą metody `find_by_tag_name()` poszukujemy znacznika `<title>`. Jako, że taki znacznik zazwyczaj występuje tylko raz w kodzie możemy traktować węzeł w kontekście skalarnym, tzn. że pierwszy znaleziony znacznik `<title>` jest tym, który nas interesuje.

Później stosujemy metodę `look_down()` w poszukiwaniu wszystkich węzłów zaczynających się od znacznika ``. Istnieją dwa takie węzły, dlatego też wynik zapisujemy w postaci tablicy `@lista`.

W kolejnej linii wyświetlamy tytuł w postaci kodu HTML za pomocą metody `as_HTML()` i przechodzimy do iterowania po tablicy `@lista` w celu wyświetlenia wszystkich punktów listy oznaczonych znacznikiem `` w ramach znacznika ``. Za pomocą metody `content_list()` schodzimy o poziom niżej tak, że w tablicy `@li` znajdują się jedynie pozycje zaczynające się od `` i kończące się na `` bez uwzględniania nadrzędnych znaczników ``.

Ostatnim krokiem jest iteracja po znalezionych wpisach `` i wyświetlenie ich w postaci czystego tekstu za pomocą metody `as_text()`.

Wynik naszego skryptu wygląda następująco:

```
$ ./parser.pl  
tytuł: <title>Tutaj tytuł&Aring;#130; strony</title>
```

```
Pierwszy na liście  
Drugi na liście  
Trzeci na liście  
Czwarty na liście  
Piąty na liście
```

Wszystko działa poprawnie. Należy zwrócić uwagę na sposób wyświetlenia węzła `<title>`. W miejscu litery „ł” w słowie „tytuł” pojawił się specjalny kod „Å#130;”. Jest to zamierzony efekt i wspominałem o tym nieco wcześniej. Jeśli chcemy wyświetlić znak w kodowaniu oryginalnym – w naszym przypadku jest to UTF-8 – musimy posłużyć się modułem `HTML::Entities` i metodą `decode_entities()`. Wtedy kod zostanie z powrotem zamieniony na literę „ł”.

4 Wyodrębnianie danych z tabel HTML

Tabele HTML mają często skomplikowane struktury. Parsowanie tabel z wykorzystaniem poprzedniej metody nie jest zadaniem łatwym, tym bardziej jeśli poszczególne kolumny nie są identyfikowane za pomocą konkretnych klas, np. `<td class="waluta">EUR</td>`. Z pomocą na szczęście przychodzi nam moduł Perla o nazwie `HTML::TableExtract`.

Tabele możemy wyodrębniać na dwa sposoby: w postaci czystego tekstu lub jako struktura drzewiasta. Pierwsza postać jest trybem domyślnym. Aby analizować tabele z wykorzystaniem struktury drzewiastej należy użyć modułu `HTML::TableExtract` z parametrem `qw(tree)`, czyli:

```
use HTML::TableExtract qw(tree);
```

Podczas ekstrakcji danych z tabel mamy do dyspozycji trzy atrybuty:

1. *headers* – atrybut ten określa nazwy nagłówek tabeli jakich szukamy. W kodzie HTML nagłówki muszą być zdefiniowane znacznikami `<th>`.
2. *depth* – określa głębokość po której się poruszamy zaczynając od 0. Dotyczy jedynie zagnieżdżonych tabel.
3. *count* – określa numer tabeli zagnieżdżonej na danej głębokości, określonej atrybutem *depth*.

4.1 Nagłówki tabel

Najprościej znaleźć interesujące nas dane wykorzystując nagłówki tabel. Wyobraźmy sobie następującą tabelę:

```
<table>
  <tr>
    <th>Lewy</th>
    <th>Srodek</th>
    <th>Prawy</th>
  </tr>
  <tr>
    <td>1</td>
    <td>2</td>
    <td>3</td>
  </tr>
  <tr>
    <td>4</td>
    <td>5</td>
    <td>6</td>
  </tr>
</table>
```

Wygląda ona następująco:

Lewy	Środek	Prawy
1	2	3
4	5	6

Przystępujemy do parsowania zakładając, że w zmiennej `$html` znajduje się kod naszej tabeli.

```
$te = HTML::TableExtract->new( headers => [qw(Lewy Srodek Prawy)] );
$te->parse($html);
```

Następnie będziemy iterować po wszystkich znalezionych tabelach spełniający warunek zawarty w atrybucie `headers`, czyli takie tabele, których nagłówki zawarte między znacznikami `<th></th>` nazywają się kolejno „Lewy”, „Srodek” i „Prawy”.

```
foreach my $ts ($te->tables) {
    foreach $row ($ts->rows) {
        print join(',', @$row), "\n";
    }
}
```

Znalezione wiersze traktowane są w kontekście listowym stąd użycie `@$row`. Do wyświetlenia zawartości tabeli po przecinku wykorzystujemy standardową funkcję Perla `join()`.

Jeśli interesuje nas tylko pierwsza znaleziona tabela możemy pominąć pierwszą pętlę `foreach` i `$ts->rows` zamienić na `$te->rows`. Wtedy kod będzie wyglądał następująco:

```
foreach $row ($te->rows) {
    print join(',', @$row), "\n";
}
```

Wynik naszego skryptu dla podanego kodu HTML wygląda następująco:

```
$ ./tabela.pl
1,2,3
4,5,6
```

4.2 Wyszukiwanie konkretnej tabeli

Wiedząc jak wygląda struktura tabel na danej stronie WWW możemy wykorzystać atrybuty `count` oraz `depth` w celu znalezienia konkretnej tabeli. Atrybut `count` wyznacza nam numer tabeli liczony od 0 z lewej do prawej, wiersz po wierszy, natomiast atrybut `depth` wyznacza, w przypadku zagnieżdżonych tabel, głębokość, od której rozpoczynamy liczenie.

W naszym przykładzie mamy jedną tabelę i kod

```
$te = HTML::TableExtract->new( headers => [qw(Lewy Srodek Prawy)] );
```

Możemy zastąpić:

```
$te = HTML::TableExtract->new( count => 0, depth => 0 );
```

W obu przypadkach zostanie znaleziona ta sama tabela z tą różnicą, że w drugim kolumny nagłówkowe „Lewy”, „Srodek” i „Prawy” zostaną potraktowane jako zwykłe wiersze w tabeli.

Dlatego też nasz kod zwróci wynik:

```
Lewy,Srodek,Prawy
1,2,3
4,5,6
```

4.3 Struktura drzewiasta

Struktura drzewiasta ułatwia nam wyświetlanie konkretnych komórek z tabeli z wykorzystaniem metody `cell()`. Aby zainicjalizować strukturę drzewiastą posłużymy się następującym kodem:

```
use HTML::TableExtract qw(tree);
$te = HTML::TableExtract->new( count => 0, depth => 0 );
$te->parse($html);
```

Czyli zupełnie tak samo jak w poprzednim przykładzie. Szukamy pierwszej tabeli (numer 0) na głębokości 0. Następnie przekształcimy znalezioną tabelę na strukturę drzewiastą i wyświetlimy pierwszą komórkę z drugiego wiersza – numerując pierwszy wiersz i komórkę jako 0 będzie to pozycja **1,0**.

```
my $table = $te->first_table_found;
my $table_tree = $table->tree;
print $table_tree->cell(1,0)->as_text."\n";
```

Za pomocą metody `as_text()` wyświetlamy zawartość komórki w postaci czystego tekstu, bez znaczników HTML. Wynik powyższego skryptu to:

```
$ ./drzewo.pl
1
```

5 Podsumowanie

Perl dostarcza nam rozbudowanych mechanizmów do wyodrębniania interesujących danych ze stron WWW. Dzięki temu automatyzacja zadań związanych z przeszukiwaniem zasobów Internetu staje się prosta i przyjemna, a wyodrębnienie danych z konkretnego miejsca na stronie nie zawsze musi oznaczać budowanie zaawansowanych wyrażeń regularnych. Połączenie modułów LWP, HTML::TableExtract oraz HTML::TreeBuilder tworzy uniwersalne narzędzie, dla którego żadna strona nie będzie trudna do analizy.

6 Bibliografia

Perl & LWP, Sean M. Burke, O'Reilly & Associates 2002.

Dokumentacja *perldoc* modułów HTML::TreeBuilder oraz HTML::TableExtract.