Perl w przetwarzaniu tekstów

Adam Dawidziuk
i Piotr Bolek

Wprowadzenie

Perl należy do tych języków programowania, których można się nauczyć szybko. Cechą decydującą o jego atrakcyjności jest łatwość pisania programów. Perl nie wymaga deklarowania typów zmiennych przed ich użyciem. Wystarczy „po prostu" napisać co ma być zrobione. Warto więc już na początku zapamiętać, że Perl nie jest najlepszy do wszystkiego – w szczególności nie należy rozwiązywać przy jego pomocy skomplikowanych problemów wymagających użycia złożonych struktur danych.

W rzeczywistości najczęściej mamy jednak do czynienia z zagadnieniami prostymi, których rozwiązanie przy pomocy programów np. w języku C, czy Pascalu jest wysoce nieefektywne. Natomiast zastosowanie skryptów pisanych w shellu jest skomplikowane lub wręcz niemożliwe, albo prowadzi do rozwiązań nieprzenośnych. Perl jest narzędziem o bardzo dużych możliwościach (w porównaniu z awkiem czy shellem), którego główne pole zastosowań to zarządzanie plikami i procesami oraz przetwarzanie tekstów. Typowy przykład, w którym użycie Perla jest zasadne, to tworzenie ładnie sformatowanych raportów z danych umieszczonych w różnych plikach, czy instalacja oprogramowania. Dobrze napisany program instalacyjny ma szansę działać pod niemal każdym systemem operacyjnym – Perl jest dostępny na wszystkie popularne platformy.

Niniejsza praca jest bardzo skróconym omówieniem Perla i koncentruje się na wykorzystaniu tego języka w zagadnieniach związanych z obróbką tekstów. Szukając punktu odniesienia warto dokonać porównania Perla z popularnymi edytorami strumieniowymi, takimi jak sed czy awk. Programy napisane dla seda czy awka na ogół dają się przekształcić na skrypt perlowy po dokonaniu niemal kosmetycznych zmian (można to zrobić automatycznie). Zazwyczaj jednak program w Perlu da się zapisać prościej i krócej. Oczywiście bywają też sytuacje odwrotne, ale na tyle rzadko, że *ad hoc* trudno wymyślić konkretny przykład.

Niewątpliwą zaletą Perla jest fakt, że programy w nim napisane działają dużo szybciej niż ich odpowiedniki w sedzie czy awku (często nawet szybciej niż dedykowany program napisany w C przez osobę, która nie zgłębiła wszystkich arkanów programowania). Jeżeli pojęcie „tekst" rozumieć nieco szerzej, to Perl staje się bezkonkurencyjny – tylko przy jego pomocy możemy łatwo przetwarzać dane binarne (np. przeczytać nagłówek pliku PCX, czy „zajrzeć" do bazy danych o znanym formacie).

Mimo iż Perl powstał i jest rozwijany w środowisku systemu UNIX, to w dziedzinie przetwarzania tektów można go używać także pod DOSem. Wszystko więc, o czym będzie mowa powinno funkcjonować w DOSowych implementacjach Perla*. Z używaniem DOSa i Perla wiąże się jednak pewne niebezpieczeństwo. W Perlu dostępna jest opcja `-i` (skądinąd bardzo pożyteczna), ale tak się nieszczęśliwie składa, że pod DOSem nazwy `dyplom.tex` i `dyplom.tex.bak` oznaczają ten sam plik. Próba wykonania polecenia takiego jak:

```
perl -i.bak skrypt.pl dyplom.tex
```

dla wielu DOSowych implementacji Perla jest jedną z metod usunięcia zawartości pliku `dyplom.tex` (w czasie próby utworzenia pliku `dyplom.tex.bak`).

Krótki opis języka

Poniżej przedstawiamy bardzo skondensowany opis języka. Nie będziemy omawiać wszystkich jego elementów, pominiemy te, które mają małe zastosowanie przy wykorzystywaniu Perla do przetwarzania tekstów.

Opcje wywołania. W wywołaniu Perla można podawać różne opcje. Poniżej wymieniamy najbardziej pożyteczne z nich.

- *–a* włącza tryb automatycznego dzielenia rekordów wejściowych (ang. *autosplit*). Pola zapisywane są do tablicy `@F`.
- *–e SKRYPT* umożliwia podanie jednego wiersza skryptu „z palca". Można używać tej opcji wielokrotnie do podania kilku linii skryptu.
- *–i [EXT]* powoduje modyfikację pliku wejściowego przetwarzanego przy użyciu konstrukcji `<>` (ang. *in-place editing*). Jeżeli podane

*: Tak naprawdę przykłady były testowane wyłącznie w systemie UNIX.

jest opcjonalne rozszerzenie EXT, to poprzednia zawartość pliku zostanie zachowana w pliku o nazwie z dodanym przyrostkiem EXT.

Uwaga! Użycie tej opcji w przypadku korzystania z DOSowych implementacji Perla grozi utratą zawartości pliku źródłowego.

- *−n* umieszcza skrypt w domyślnej pętli czytającej kolejne rekordy z wejścia. Brak domyślnego wyprowadzania wyników.

```
while (<>) {
   ... # Tutaj skrypt
}
```

- *−p* umieszcza skrypt w domyślnej pętli czytającej kolejne rekordy z wejścia. Każdy rekord jest po przetworzeniu wypisywany na wyjście. Użycie tej opcji przekształca Perla w edytor potokowy podobny do programu sed.

```
while (<>) {
   ... # Tutaj skrypt
} continue {
   print
}
```

- *−w* wypisuje ostrzeżenia o możliwych błędach oraz użytych w skrypcie konstrukcjach, które mogą być źródłem błędów.

Wczytywanie danych. Notacja używana w Perlu do wczytywania kolejnych rekordów wejściowych jest dosyć specyficzna i początkujący użytkownicy mają czasem problemy z jej przyswojeniem.

Aby wczytać rekord danych z wejścia, należy w Perlu napisać nazwę deskryptora pliku, którego chcemy czytać, objętą nawiasami kątowymi. Wyrażenie `<STDIN>` – oznacza polecenie wczytania jednej linii ze standardowego strumienia wejściowego. Można czytać oczywiście także z innych strumieni bądź plików, ale w takim przypadku należy je wcześniej otworzyć funkcją open. Często używaną notację <STDIN> można skrócić do postaci <> np.:

```
$line = <>;
<>;
```

Pierwszy wiersz oznacza wczytanie jednej linii ze standardowego wejścia do zmiennej `$line`, w drugim linia wczytywana jest do domyślnej zmiennej `$_`.

Typy danych. Typ zmiennej w Perlu zależy od kontekstu. Ta sama zmienna może być raz traktowana jako tekstowa, a innym razem jako numeryczna. Istnieje natomiast ścisły podział na zmienne proste (skalarne) i zmienne złożone (tablicowe). Rodzaj zmiennej określa przedrostek.

- `$var` prosta zmienna skalarna.
- `$var[13]` trzynasty element tablicy `@var`.
- `$var{'Feb'}` wartość jednego z elementów tablicy asocjacyjnej `%var`.
- `$#var` ostatni indeks tablicy `@var`.
- `@var` cała tablica (wektor); w kontekście skalarnym: liczba elementów tablicy.
- `@var[3,4,5]` przekrój tablicy `@var`.
- `%var{'a','b'}` przekrój tablicy `%var`.
- `INPUT` deskryptor pliku lub katalogu (zwyczajowo pisany wielkimi literami).
- `*var` cokolwiek (przekazane przez nazwę).

Instrukcje. Instrukcją jest każde podstawienie, z opcjonalnym modyfikatorem, zakończone średnikiem. Wykonanie instrukcji może zależeć od innego wyrażenia podanego przy użyciu jednego z modyfikatorów: `if`, `unless`, `while` albo `until` np.:

```
WYR1 if WYR2;
WYR1 until WYR2;
```

W pierwszym przypadku `WYR1` zostanie wykonane jeżeli `WYR2` zwróci wartość niezerową (podobnie jak w C interpretowaną w warunkach jako prawda). W drugim przypadku wyrażenie `WYR1` będzie wykonywane dopóki `WYR2` będzie fałszywe.

W wyrażeniach mogą być także używane operatory logiczne `||`, `&&` i `?` np.:

```
WYR1 || WYR2;
WYR1 ? WYR2 : WYR3;
```

Pierwsza linia w powyższym przykładzie jest równoważna instrukcji:

```
WYR2 unless WYR1;
```

druga jest znanym z C operatorem warunkowym. Możliwość używania w taki sposób operatorów logicznych pozwala pewne działania kodować w bardzo naturalny sposób, np.:

```
open (IN, "input.tex")
          || die "can't open file\n";
```

W powyższym wierszu dokonywana jest próba otwarcia pliku, a jeśli się ona nie powiedzie – to program jest przerywany (*open or die*).

Przy pomocy nawiasów klamrowych {} można łączyć instrukcje w bloki.

Dostępne są następujące instrukcje strukturalne:

```
if (WYR) BLOK [[elsif (WYR) BLOK ...]
        else BLOK]
unless (WYR) BLOK [ else BLOK ]
[ETYKIETA:] while (WYR) BLOK [continue
        BLOK]
[ETYKIETA:] until (WYR) BLOK [continue
        BLOK]
[ETYKIETA:] for (WYR;WYR;WYR) BLOK
[ETYKIETA:] foreach ZMIENNA (TABLICA)
        BLOK
[ETYKIETA:] BLOK [continue BLOK]
do BLOK while WYR;
do BLOK until WYR;
```

Do zmiany przebiegu programu wewnątrz bloków służą instrukcje `next`, `last` i `redo`. Instrukcja `next` powoduje przejście do następnej iteracji pętli (działa tak jak `continue` w języku C), `last` powoduje natychmiastowe wyjście z pętli (tak jak `break` w C), a `redo` wykonuje tę samą iterację pętli raz jeszcze, bez obliczania warunku zakończenia. Każdej z instrukcji `next`, `last` i `redo`, można podać opcjonalnie etykietę, dla określenia o zmianę przebiegu której pętli chodzi (pożyteczne w przypadku pętli zagnieżdżonych).

Procedury. W Perlu można tworzyć także procedury. Deklaruje się je przy pomocy instrukcji `sub`.

```
sub procedura {
... # tutaj instrukcje procedury
}
```

Procedurę wywołuje się przez podanie jej nazwy poprzedzonej znakiem & – np. `&procedura($a, "tekst")`. Jeżeli procedura nie wymaga podania parametrów, to w wywołaniu można opuścić nawiasy.

W definicji procedury nie specyfikuje się żadnych parametrów. Parametry są przekazywane do procedury w specjalnej tablicy `@_`. Do kolejnych argumentów wywołania można odwoływać się poprzez `$_[1]`, `$_[2]` itd.

Wartością zwracaną przez procedurę jest wartość ostatniego obliczonego w niej wyrażenia.

Operatory. W Perlu dostępne są wszystkie operatory dostępne w języku C. Istnieje także kilka operatorów specyficznych dla tego języka, oto one:

- `**` – potęgowanie.
- `eq`, `ne` – równość, nierówność ciągów znaków.
- `lt`, `gt` – znakowe mniejszy niż, większy niż.
- `le`, `ge` – znakowe mniejszy (większy) lub równy.
- `<=>` – porównywanie liczb, zwraca −1, 0 lub 1.
- `cmp` – porównywanie ciągów znaków, zwraca −1, 0 lub 1.
- `=~`, `!~` – specyfikacja wzorca wyszukiwania, zamiany lub tłumaczenia znaków (druga forma – zanegowana).
- `..` – zakres (indeksów, lini wejściowych, itp.).
- `x` – operator powtórzenia ciągu znaków.

Wyrażenia regularne. Każdy znak z wyjątkiem znaków specjalnych `+?.*()[]{}|\` oznacza samego siebie.

- `.` – oznacza dowolny znak oprócz znaku nowego wiersza.
- `(...)` – nawiasy okrągłe grupują kilka elementów wzorca w jedno podwyrażenie.
- `+` – oznacza co najmniej jedno wystąpienie poprzedzającego elementu.
- `?` – co najwyżej jedno wystąpienie poprzedniego elementu.
- `*` – dowolna liczba wystąpień elementu (łącznie z brakiem wystąpienia).
- `{N,M}` – co najmniej `N` wystąpień, ale nie więcej niż `M`, `{N}` oznacza dokładnie `N` wystąpień, `{N,}` oznacza co najmniej `N` wystąpień.
- `[...]` – oznacza klasę znaków (dowolny znak z podanego zbioru), `[^...]` także klasa znaków, ale spoza podanego zbioru.
- `(...|...|...)` – jedna z alternatyw.
- `\w` – znak alfanumeryczny (wraz z „_”), `\W` – znak niealfanumeryczny.
- `\b` – granica słowa, `\B` – wnętrze słowa.
- `\s` – spacja (itd.), `\S` – nie spacja.
- `\d` – cyfra, `\D` – nie cyfra.
- `\n`,`\r`,`\t`,`\f` – CR, LF, TAB, FF.
- `\1...\9` – odwołania do kolejnych podwyrażeń zgrupowanych nawiasami `()` (wewnątrz wyrażenia).

Funkcje standardowe. W Perlu dostępnych jest wiele funkcji standardowych. Tutaj wymienimy tylko niektóre z nich, szczególnie przydatne przy przetwarzaniu tekstów. Znaczek „†” oznacza, że poprzedzający go argument może być opuszczony (w takim przypadku domyślnym argumentem funkcji jest zmienna `$_`). Znaczek „*” przy nawiasach oznacza, że opcjonalnie mogą one być opuszczone. W nawiasach kwadratowych – elementy opcjonalne.

- `chop(LISTA†)` – odcina ostatni znak z wszystkich elementów listy, zwraca ostatni odcięty znak. Jeżeli argument jest pojedynczą zmienną, to można opuścić nawiasy.
- `eof[(DESKR)]` – test końca pliku; bez argumentu – ostatnio czytanego pliku.
- `eval(WYRAŻENIE)` – `WYRAŻENIE` jest analizowane i wykonane tak jakby był to program w Perlu. Wartością zwracaną jest wartość ostatniego obliczonego wyrażenia. W przypadku wykrycia błędu składniowego bądź wykonania, funkcja zwraca wartość niezdefiniowaną, a w zmiennej `$@` umieszczany jest komunikat błędu.
- `grep(WYR,LISTA)` – oblicza wyrażenie `WYR` dla każdego elementu listy, podstawiając kolejne wartości z listy pod zmienną `$_`. Modyfikacja `$_` powoduje modyfikację odpowiedniego elementu z listy. Zwraca tablicę elementów, dla których wyrażenie było prawdziwe (zwróciło wartość niezerową).
- `join(WYR,LISTA)` – łączy oddzielne ciągi znaków z listy w jeden łańcuch, wstawiając między sąsiednie elementy wartość wyrażenia `WYR`.
- `length(WYR†)*` – zwraca długość wyrażenia `WYR` w znakach.
- `open(DESKR[,NAZWA])` – otwiera plik i związuje z nim deskryptor. Jeżeli `NAZWA` jest pominięta to nazwa pliku musi być zawarta w zmiennej skalarnej o takiej samej nazwie jak `DESKR`. Do oznaczenia trybu otwarcia stosowana jest następująca konwencja:
  - `"FILE"` – otwarcie pliku do czytania.
  - `"<FILE"` – jw.
  - `">FILE"` – otwarcie pliku do pisania, w razie potrzeby utworzenie.
  - `">>FILE"` – otwarcie pliku w trybie dopisywania.
  - `"+>FILE"` – otwarcie pliku do pisania i czytania.
  - `"|CMD"` – otwarcie potoku do polecenia `CMD`.
  - `"CMD|"` – otwarcie potoku z polecenia `CMD`.
- `pop(@TABLICA)*` – pobiera ostatni element z tablicy skracając ją o jeden.
- `print[([DESKR] LISTA†)*]` – wypisuje na wyjściu ciąg znaków składający się z elementów oddzielonej przecinkami listy. Przy pominięciu deskryptora wypisuje na standardowe wyjście (albo na ostatnio wybrany poleceniem `select` kanał wyjściowy).
- `printf[([DESKR] LISTA)*]` – równoważne z `print DESKR sprintf(LISTA)`.
- `push(@TABLICA,LISTA)` – dopisuje wartości elementów listy na koniec tablicy. Długość tablicy jest zwiększana o liczbę elementów na liście.
- `reverse(LISTA)*` – w kontekście wektorowym zwraca listę w odwrotnej kolejności; w kontekście skalarnym odwraca kolejność znaków w pierwszym elemencie listy.
- `scalar(%TABLICA)` – zwraca wartość prawdziwą jeżeli istnieją jakieś elementy w tablicy asocjacyjnej.
- `scalar(@TABLICA)` – zwraca liczbę elementów w tablicy.
- `select[(DESKR)]` – ustawia domyślny kanał wyjściowy; bez argumentów zwraca ostatnio wybrany kanał.
- `shift(@TABLICA)*` – pobiera pierwszy element z tablicy i przesuwa wszystkie kolejne w dół, skracając długość tablicy o jeden. Argument może być pominięty i wtedy funkcja operuje na tablicy `@ARGV` w programie głównym lub na tablicy `@_` w procedurach.
- `sort([PROC] LISTA)*` – sortuje listę i zwraca tablicę posortowanych wartości. Opcjonalnie istnieje możliwość podania procedury wartościującej (jako bloku, albo wywołania zdefiniowanej procedury).
- `split[(WZORZEC[,CIĄG†[,LIMIT]])]` – dzieli ciąg znaków na kawałki i zwraca jako tablicę. Jeżli podany jest limit, to dzieli na co najwyżej `LIMIT` pól. Przy braku wzorca, dzieli na spacjach. W przypadku użycia w kontekście skalarnym zwraca liczbę pól, a podzielony łańcuch zwraca w tablicy `@_`.
- `sprintf(FORMAT, LISTA)` – zwraca sformatowany ciąg znaków tak, jak funkcje rodziny `printf` z języka C.

- `unshift(@TABLICA,LISTA)` – dodaje listę na początek tablicy przesuwając wszystkie elementy w górę; zwraca nową długość tablicy.
- `substr(WYR,POCZ[,DŁ])` – wybiera fragment (długości `DŁ` – jeśli była podana) ciągu znaków reprezentowanego przez `WYR`, począwszy od znaku o numerze `POCZ` (znaki są oczywiście numerowane od zera).

Funkcje wyszukiwania i zamiany. Przy przetwarzaniu danych tekstowych szczególnie istotne są funkcje wyszukiwania i zamiany łańcuchów znakowych.

- `[WYR =~][m]/WZORZEC/[g][i][o]`
  Poszukuje wzorca w wyrażeniu (domyślnie w zmiennej `$_`). Po poprzedzeniu wzorca literą `m` można zamiast „ciachów” używać jako ograniczników niemal dowolnej pary znaków. W kontekście wektorowym zwraca tablicę łańcuchów pasujących do podwyrażeń ograniczonych nawiasami (tzn. `$1`, `$2`, `$3`, itd.). Opcjonalne modyfikatory: „`g`” oznacza dopasowanie wielokrotne, „`i`” dopasowanie bez rozróżnienia wielkości liter, „`o`” rozwinięcie zmiennych zawartych we wzorcu tylko raz (np. przy wielokrotnym używaniu tego samego wzorca w pętli).
- `[$ZM =~]s/WZORZEC/NOWY/[g][i][e][o]`
  Poszukuje wzorca w łańcuchu znaków i w przypadku znalezienia zastępuje go nową zawartością. Zwraca liczbę dokonanych zamian. Jeżeli nie zostanie dokonane żadne podstawienie zwraca zero. Opcjonalne modyfikatory: „`g`” zastąpienie wszystkich wystąpień wzorca; „`e`” interpretacja nowego ciągu znaków jako wyrażenia przed dokonaniem podstawienia; „`i`” oraz „`o`” mają znaczenie takie samo jak w przypadku wyszukiwania. Jeżeli wzorzec jest pusty, to do poszukiwania używany jest wzorzec z ostatniego wyszukiwania lub zamiany.
- `[$ZM =~]tr/CIĄG/NOWYCIĄG/[c][d][s]`
  Tłumaczy wszystkie wystąpienia znaków z pierwszego ciągu na odpowiednimi znakami z nowego ciągu. Zwraca liczbę zastąpionych znaków. Zamiast `tr` można używać `y`. Opcjonalne modyfikatory: „`c`” zamiana znaków, które nie występują na liście; „`d`” kasowanie wszystkich znaków z listy; „`s`” – od ang. *squeeze* – kompresja sekwencji jednakowych znaków na wyjściu (np. `tr/abc/ABC/s` przekształci łańcuch `"abaaacc"` na `"ABAC"`).

Zmienne specjalne. W Perlu dostępnych jest wiele zmiennych mających znaczenie specjalne. Oto lista najciekawszych z nich:

- `$_` – najważniejsza zmienna w Perlu, pełniąca rolę standardowego bufora wejściowego, oraz standardowej przestrzeni poszukiwań wzorców.
- `$.` – numer bieżącego wiersza ostatnio przeczytanego pliku.
- `$/` – separator rekordów – domyślnie znak nowego wiersza, odpowiednik zmiennej `RS` w awku. Zmienna ta może mieć wartość wieloznakową.
- `$,` – separator pól na wyjściu, odpowiednik `OFS` w awku.
- `$\` – separator rekordów na wyjściu, odpowiednik `ORS` w awku.
- `$0` – nazwa pliku zawierającego wykonywany właśnie skrypt perlowy, wartość tej zmiennej można modyfikować.
- `$ARGV` – nazwa bieżączego pliku czytanego przy użyciu konstrukcji `<>`.
- `$&` – ciąg znaków pasujący do ostatnio używanego wzorca wyrażenia regularnego.
- `` $` `` – ciąg znaków poprzedzających to co pasowało do ostatniego wzorca.
- `$'` – ciąg znaków poprzedzany przez to co pasowało do ostatniego wzorca.
- `$1...$9` – wzorce pasujące do podwyrażeń podanych w ostatnim wyrażeniu regularnym w nawiasach `()`.
- `@ARGV` – tablica zawierająca argumenty wiersza wywołania skryptu (bez nazwy samego skryptu).
- `@_` – tablica argumentów dla procedur.

Przykłady

Przedstawimy teraz kilka przykładów wykonywania pewnych zadań jakie mogą pojawić się w czasie przetwarzania tekstów.

Usunięcie przeniesień w pliku tekstowym. Prosty przykład: usunięcie dzielenia wyrazów między wierszami w „sformatowanym” pliku tekstowym. Należy znaleźć wiersze tekstu zakończone dywizem, usunąć dywiz i „skleić” podzielony wyraz. Wystarczy napisać:

```
perl -pe 's/-\s*\n$//' ⟨pliki⟩
```

Użycie wzorca `/-\n/` oznacza dywiz wraz ze znakiem końca wiersza. Między tymi znakami może stać spacja (tabulator itp.), ale może tam też nie być nic – odpowiedni jest więc wzorzec `/-\s*\n$/`, który po znalezieniu po prostu usuwamy. Co prawda z dwóch wierszy robimy w ten sposób jeden długi, ale TEX już sobie z tym poradzi.

Słowo `pliki` w tym przykładzie (także w następnych) oznacza nazwy plików podane wprost lub określone wzorcem (np. `*.tex`). Przy użyciu takim jak w przykładzie wszystko co powstanie w wyniku przetwarzania tych plików zostanie skierowane do standardowego strumienia wyjściowego, który można oczywiście zapisać do pliku. Jeżeli jednak po przetworzeniu nie chcemy łączyć zawartości wszystkich plików w jeden, to możemy użyć opcji `-i`.

Wycięcie wybranych kolumn z plików.

```
perl -ne 'chop;
    print substr($_,⟨kolumna⟩,⟨szer⟩),
    "\n";' pliki
```

Funkcja `substr` wykonywana kolejno na każdym wierszu (opcja `-n`) „wycina” z niego łańcuch znaków o długości `szer`, począwszy od znaku `kolumna`. Przed użyciem funkcji `substr` należy z wiersza usunąć znak nowej linii (`\n`), czyli ostatni znak wiersza, który później jawnie dopisuje funkcja `print`.

Wyciągnięcie tabelek z pliku LATEX-owego.

```
perl -ne "print
if /\\\\begin{tabular}/../\\\\end{tabular}/;"
```

Tu po prostu mówimy o co nam chodzi (jeszcze jedna miła cecha Perla). *Drukuj jeśli* (`print if`) aktualnie przetwarzany wiersz znajduje się między wierszami wzorcowymi (tutaj `\begin{tablar}` a `\end{tabular}`). Znak `\` ma dla Perla znaczenie specjalne, więc mu je odbieramy pisząc „`\\`”. Ponieważ użyliśmy tutaj znaków `"..."` dla wyodrębnienia „programu” z wiersza polecenia, a znak „`\`” jest znakiem specjalnym także dla interpretera poleceń, należy dodatkowo raz jeszcze podwoić każdy „w tył ciach”. Zapis można uprościć nie dopuszczając interpretera poleceń do zawartości naszego programu obejmując go znakami apostrofów `'...'`:

```
perl -ne 'print
if /\\begin{tabular}/../\\end{tabular}/;'
```

Usuwanie kolejnych pustych wierszy. Gdy z dwóch lub więcej pustych wierszy trzeba zrobić jeden wystarczy napisać:

```
perl -ne 'print
if /\S/ || !$s; $s=/^\s*$/;' ⟨pliki⟩
```

Wzorzec `/\S/` pasuje do każdego wiersza niepustego, a `/^\s*$/` do każdego pustego.

Zmienna `$s` służy do zapamiętania rodzaju ostatniego wiersza, wartość 1 oznacza pusty, 0 – niepusty. Program można więc przeczytać jako: *drukuj jeśli wiersz niepusty lub poprzedni wiersz nie był pusty* (`print if /\S/ || !$s;`).

Sortowanie akapitów. Przykład ten został wykorzystany przy przygotowywaniu tego tekstu do posortowania listy funkcji standardowych.

```
$/="";
@a=<>;
print sort @a;
```

W pierwszym wierszu jako separator rekordów wejściowych zostaje wybrany pusty wiersz (takie znaczenie ma przypisanie pustego łańcucha zmiennej `$/`), dzięki czemu możemy operować na całych akapitach. W drugim wierszu do tablicy `@a` zostaje wczytana cała zawartość pliku wejściowego (każdy akapit do oddzielnego elementu tablicy). Na koniec w wierszu trzecim wszystkie elementy tablicy `@a` są sortowane i zapisywane na wyjście.

„Spacjowanie” liczb naturalnych. Zadanie polega na wpisaniu spacji (lub innego znaku) między cyfry wielocyfrowej liczby naturalnej – co trzy cyfry od końca, np. tak: 10 000 000.

Operację tę najłatwiej wykonać w pętli:

```
1 while s/(.*\d)(\d\d\d)/$1 $2/;
```

Wyrażenie `1` wykonywane jest dopóki wynik zamiany jest pozytywny, czyli została ona dokonana. Zastępowany jest najdłuższy ciąg znaków zakończonych czwórką cyfr, w ten sposób, że przed ostatnie trzy cyfry wstawiana jest spacja (użycie podwyrażeń i odwołań do nich: `$1`, `$2`). Ponieważ za każdym razem do wzorca dopasowywany jest najdłuższy możliwy ciąg znaków, to pętla wykona się dokladnie tyle razy, ile trzeba.

Zamiana s/.../.../ domyślnie wykonywana jest na zmiennej $_. Dla wygody dobrze jest opisywaną pętlę zamknąć w procedurze:

```
sub spaces {
    local($_) = @_;
    1 while s/(.*\d)(\d\d\d)/$1 $2/;
    $_;
}
```

i odwoływać się do niej następująco:

```
$wynik = &spaces(123424234);
```

Funkcja local($_) powoduje, że zmiany wartości zmiennej $_ są ograniczone do procedury &spaces i nie wpływają na jej wartość na zewnątrz.

Zamiana znaków i ciągów znaków w wielu plikach. Problem zmiany kodowania polskich liter najłatwiej rozwiązać funkcją tr.

```
perl -pi.bak -e 'tr/abc/xyz/' ⟨pliki⟩
```

W tym przykładzie każda litera „a” zostanie zastąpiona literą „x”, „b” – „y”, a „c” – „z”.

Dla porównania popatrzmy na program zamieniający jeden łańcuch znaków na inny (np. „wrr” na „Bzzz...”).

```
perl -pi.bak -e 's/wrr/Bzzz.../g' ⟨pliki⟩
```

Programik taki może być szczególnie użyteczny, jeżeli wykonamy go od razu na wielu plikach.

„deTEX”. Łatwo wyobrazić sobie sytuację, gdy z pliku należy usunąć wszystkie polecenia TEX-a czy LATEX-a i zostawić tylko tekst zasadniczy. Sytuacja taka może mieć np. miejsce gdy chcemy dokonać sprawdzenia poprawności pisowni przy użyciu programu, który „nie zna” TEX-a. Programik w Perlu, który to robi może wyglądać następująco:

```
1. #! /usr/bin/perl -pi
2. s#^%.*$##g ;
3. s#([^\\])%.*$#$1#g ;
4. s#\\begin\{\w+\}##g ;
5. s#\\end\{\w+\}##g ;
6. s#\\\w+##g ;
7. s#\d##g ;
```

Wiersz pierwszy rozpoczyna się znakiem #, więc jest dla perla komentarzem. W systemie UNIX wpisanie takiego wiersza na początku jest informacją dla shella, że dany plik zawiera program dla Perla, który ma być wywołany z opcjami -pi. Dzięki temu po nadaniu atrybutu wykonywalności, plik zawierający ten program może być uruchamiany przez podanie jego nazwy.

Cała reszta programu to seria zastąpień. Zwróćmy uwagę, że ogranicznikiem wzorców w funkcji s nie musi być koniecznie „ciach” (/) – może to być dowony znak (nawet znak komentarza).

W wierszu (2) usuwane są komentarze TEX-owe zaczynające się od początku wiersza, a w wierszu (3) – pozostałe (bez ruszania jednak sekwencji oznaczającej znak procentu \%).

W wierszach (4) i (5) usuwane są ograniczniki środowisk LATEX-owych (\begin{...} i \end{...}), w (6) wszystkie słowa poprzedzone znakiem \ (czyli polecenia TEX-owe). Na koniec usuwane są wszystkie cyfry (7).

Eliminacja zawieszeń. Ostatni przykład – programik służący do eliminacji zawieszeń (patrz GUST, Zeszyt 6, tabelka na str. 7) pozostawimy bez komentarza, aby nie pozbawić czytelników przyjemności samodzielnej analizy sposobu jego działania.

```
#!/usr/bin/perl
$asciipl="a-zA-Ząćęłńóśżź ĄĆĘŁŃÓŚŻŹ";
$litery="aiozwuAIOZWU";
while (<>) {
  $match =
    s/([^$asciipl])([$litery])\s+$/$1\n$2~/g;
  if ($match) {
    $_ .= <>;
    s/([$litery]\~)(\s+)/$2$1/;
    redo;
  }
  s/([^$asciipl])([$litery])\s+/$1$2~/g;
  s/^([$litery])\s+/$1~/g;
  print;
}
```

Podsumowanie

Na tym kończymy, z konieczności bardzo przyspieszony, kurs Perla. Wszystkie przedstawione tutaj przykłady powstały jako rozwiązania rzeczywistych, praktycznych problemów, z którymi spotykaliśmy się w czasie przygotowywania różnych tekstów w TEX-u. Są one bardzo krótkie, ale zadania, które rozwiązują nie zawsze da się w tak prosty i elegancki sposób rozwiązać używając innych narzędzi.

Perl jest doskonałym uzupełnieniem dobrego edytora tekstów. Jeżeli używamy edytora (np. vi, Emacs), który umożliwia filtrowanie bloków tekstu przez zewnętrzne programy, to często wystarczy napisać jednolinijkowy skrypt, podać go Perlowi w wierszu wywołania opcją -e i „przepuścić” przez niego zaznaczony blok tekstu. Tak właśnie było w przypadku sortowania listy funkcji standardowych przedstawionej wcześniej w niniejszym tekście. Lista ta została najpierw wpisana bez zwracania uwagi na kolejność, a następnie posortowana uproszczonym (w zapisie nie działaniu) sposobem przedstawionym w przykładzie *Sortowanie akapitów*:

```
perl -e '$/=""; print sort <>;'
```

Cała operacja odbyła się bez opuszczania edytora.

Siła Perla polega właśnie na tym, że za pomocą bardzo prostych, często jednolinijkowych skryptów można dokonywać niebanalnego przetwarzania jednego lub wielu plików.

Literatura

1. Larry Wall, Randal L. Schwartz: *Programming Perl*, O'Reilly & Associates, 1991.
2. Randal L. Schwartz: *Learning Perl*, O'Reilly & Associates, 1993.
3. Johan Vromans: *Perl Reference Guide*, dostępny dla na serwerach CPAN (Comprehensive Perl Archive Network) w katalogu doc/refguide/.
4. *Perl manual pages*, dostępne w dystrybucji Perla.

◇ Adam Dawidziuk
A.Dawidziuk@elka.pw.edu.pl

◇ Piotr Bolek
P.Bolek@ia.pw.edu.pl

TeX Live CDROM

Sebastian Rahtz
Secretary, TeX Users Group

I am very glad to announce the launch today in Paris of TeX Live, a new CD published by the TeX Users Group, the UK TeX Users Group and GUTenberg (French TeX Users), with help from NTG (Dutch TeX Users) and many individuals from other groups. TeX Live's 649 megabytes contains:

- a ready to run Unix TeX setup, Thomas Esser's teTeX (based on Karl Berry's Web2c). It has binaries for: Linux on Intel and m68k platforms, Irix 5.2, 5.3 on MIPS (SGI Indy/Indigo), SunOS 4.1.3 on Sun, Solaris 2.3, 2.4, 2.5 on SPARC, HPUX 9.01, 10.01 for HP workstations, Digital Unix (OSF/1) 2.0 and 3.2 for DEC Alpha machines, FreeBSD and NetBSD on Intel platforms, Ultrix 4.3, for DEC Decstation machines, AIX 3.2, 4.1.1, for IBM RS6000 machines, NeXTStep on Intel platforms;
- a very large support tree of macros, fonts and documentation arranged according to the TeX Directory Structure layout;
- the GUTenberg Mac, DOS and Windows distributions (archived).

You can use the TeX system by running directly from the CD, installing on your hard disk, or by adding packages to your existing system.

This is not a dump of CTAN full of compressed archives. This is a *working* system. To make it useable under Unix, it uses the Rock Ridge extensions to the ISO9660 file system. Ordinary systems can still read it, but will not see the long file names or symbolic links.

If you are on any flavour of Unix, BUY this CD! There is no complicated compilation or moving of installed files around, it will just *WORK*.

More details, and ordering information, can be found at `http://www.tug.org/texlive.html`. If you dont have WWW access, mail me or tug@tug.org for details. Cost is around $20 for members of any TeX users group, or $40 for others (the prices vary, depending on postage). All profits from sales go back to fund new versions of the CD, and to TeX-related development projects.

(*Komunikat z sieci 29 maja 1996*)