Systemy operacyjne

Laboratorium 7

Perl – podstawy

Jarosław Rudy
Politechnika Wrocławska

27 kwietnia 2017

Temat obejmuje podstawowe zrozumienie języka `Perl` – tworzenie skryptów, składnię, obsługę zmiennych, wejścia/wyjścia itp. Jako dodatkowe źródło wiedzy o `Perl`u zaleca się stronę `perldoc`. Używanie typowego manuala (`man perl`) nie jest zalecane.

1 Wstęp

`Perl` jest językiem programowania, którego użyjemy do zastąpienia praktycznie wszystkich poprzednich mechanizmów i komend. Pozwoli on nam na kopiowanie, usuwanie czy przeglądanie plików i dowiązań jak `bash`. Pozwoli na przeszukiwanie i przetwarzanie drzew katalogowych podobnie do `find`a i mechanizmu potoków. Umożliwi także przetwarzanie tekstu w sposób zbliżony do `awk`a. Co więcej, bazowa składnia `perl`a czerpie z języka `C/C++`, a sam język jest w większości przenośny (tzn. dostępny także dla Windowsa, chociaż dostępne funkcje mogą się różnić).

`Perl` ma też wady – pełni on wiele funkcji (odpowiednik powłoki systemowej, odpowiednik narzędzi typu `awk` czy `grep`, programowanie obiektowe), jest językiem ogólnego przeznaczenia i działa według filozofii "istnieje więcej niż jeden sposób by rozwiązać dany problem". W rezultacie `perl` nie jest najprostszym językiem, co obejmuje m.in. składnię. Ponadto część narzędzi/technik będzie podobna do tych, które znany z wcześniejszych laboratoriów, a część nie.

Zaczniemy od podstawowych cech `perl`a, które posłużą nam do wykonywania tych samych zadań co w `bash`u.

2 Uruchamianie skryptów

Skrypty `perl`owe zaczynamy linią `#!/usr/bin/perl` (o ile `perl` został zainstalowany w domyślnej lokalizacji) zamiast `#!/bin/bash`. Instrukcje `use strict;`

i `use warnings;` *mogą* zostać umieszczone w początkowej części skryptu, co czasami może pomóc wykryć niektóre błędy. Pierwsza z tych instrukcji powinna być jednak używana z rozwagą, gdyż wymusza restrykcyjny sposób pisania kodu (np. wymóg deklarowania zmiennych lokalnych z użyciem słów kluczowych `local` i `my`), co może sprawiać problemy bez głębszej znajomości `perla`.

3 Zmienne i ich kontekst

Zmienne w `perlu` zawsze używane są w pewnym *kontekście*. Podstawowym kontekstem jest kontekst *skalarny*. Skalary przechowują pojedynczą wartość (liczba, tekst itp.) i oznaczane są poprzez znak dolara:

```
$skalar = 15;
```

Skalarów będziemy używać przez większość czasu. Ze skalarami związanych jest kilka niuansów, ale zanim do nich przejdziemy przedstawimy kolejny kontekst: *tablicowy* (ewentualnie *listowy*)[1]. Kontekst tablicowy oznaczany jest przez znak `"@"`. Przykładowa definicja tablicy 3 elementów:

```
@tablica = ( "Element1" , "Element2" , "Element3" );
```

Należy zwrócić uwagę na to, że nawiasy są okrągłe, a nie klamrowe! Tablice same w sobie oznaczają kontekst tablicowy, ale pojedynczy element tablicy jest skalarem! Jeśli więc chcemy przypisać wartość do i-tego elementu tablicy do pewnej (skalarnej) zmienne, to powinniśmy napisać:

```
$zmienna = $tablica[i];
```

Ciekawą cechą perla jest możliwość indeksowania elementów tablicy licząc od jej *końca* z użyciem indeksów ujemnych:

```
$tab[-1]     # ostatni element
$tab[-2]     # przedostatni element
```

Tablice `perlowe` indeksowane są więc od 0 (kierunek od początku tablicy) lub od -1 (kierunek od końca tablicy).

Kolejną kwestią jest uzyskanie liczby elementów w tablicy. Pierwszą (niezalecaną) metodą jest uzyskanie indeksu ostatniego elementu tablicy (składnia `"$#"`), a potem dodanie do wyniku jedynki:

```
$liczba_elementow = $#tablica + 1;
```

Drugim sposobem uzyskania liczby elementów jest... użycie tablicy w kontekście SKALARNYM. Jeśli chcemy przypisać wynik do zmiennej skalarnej, to sprawa jest prosta:

```
$liczba_elementow = @tablica;
```

[1] Tablice i listy nie są dokładnie tym samym w `perlu`, ale dla uproszczenia możemy tak założyć.

Zauważmy, że po prawej stronie jest cała tablica (`"@"`), zaś po lewej stronie jest skalar.

Problem pojawia się, gdy potrzebujemy uzyskać liczbę elementów tablicy w sytuacji, gdy kontekst jest określany automatycznie i wynik nie jest skalarem. Przykładowo pętla `for`:

```
for( @tablica )
```

wymusi kontekst tablicowy – pętla wykona się tyle razy ile jest elementów tablicy (zwykle taki jest cel). Jednak, co jeśli chcemy w tym momencie użyć kontekstu skalarnego? Możemy albo użyć poprzedniej metody:

```
$zmienna = @tablica;
for( $zmienna )
```

Można też wymusić kontekst skalarny za pomocą operatora `scalar`:

```
for( scalar @tablica )
```

W powyższych przypadkach pętla wykona się tylko raz – z wartością będącą liczbą elementów tablicy.

Oprócz skalarów i tablic występują także inne konteksty, jak kontekst hashowy i referencja, które jednak pojawią się dopiero w następnym laboratorium.

4 Argumenty skryptu

W `bash`u argumenty przekazane do skryptu przechowywane były w zmiennych `$0`, `$1`, `$2` itp. przy czym `$0` przechowywało nazwę (ścieżkę) samego skryptu. W `perl`u `$0` ma identyczną funkcję, ale "zwykłe" argumenty przechowywane są w specjalnej tablicy `@ARGV`[2] (podobnie do języka `C`). Tak więc odpowiednikiem `bash`owego `$1` w `perl`u jest `$ARGV[0]`, zaś odpowiednikiem `$#` jest `@ARGV` w kontekście skalarnym lub `$#ARGV+1`.

5 Funkcja `print` i podstawowe separatory

`Perl` do drukowania tekstu na ekran wykorzystuje funkcję `print`. Zwykle wywołuje się ją tak jakby była komendą `bash`ową (bez nawiasów):

```
print argument;
```

zamiast

```
print (argument);
```

[2] Zmienne `$1`, `$2` itp. występują w `perl`u, ale mają inne znaczenie.

Związane jest to z faktem, że źle ustawione nawiasy mogą zostać uznane za definicję listy i wywołać trudne w znalezieniu błędy.

Perl, podobnie do awka wykorzystuje *wyjściowe* separatory pól/rekordów, które wykorzystywane są przez print. Dla awka separatory wyjściowe były zawarte w zmiennych OFS (pola) i ORS (rekordy). W przypadku perla zmiennymi tymi są $, (pola) oraz $\ (rekordy). W przeciwieństwie do awka, oba separatory wyjściowe w perlu domyślnie zawierają PUSTE łańcuchy. Jeśli chcemy, aby zawierały spację i nową linię (domyślne zachowanie awka), to należy je ustawić:

```
$, = " ";
$\ = "\n";
```

Tak więc instrukcja:

```
print "Ala", "ma", "kota";
```

Domyślnie wydrukuje tekst "Alamakota" bez wstawienia nowej linii na końcu. Jeśli jednak ustawimy separatory jak pokazano powyżej, to instrukcja da efekt "Ala ma kota" wraz ze znakiem nowej linii.

Tablice również posiadają swoje separator, które jest interpretowany przez print. Separator ten jest przechowywany w zmiennej $". Jeśli utworzymy tablicę:

```
@tab = ( "A" , "B" , "C");
```

a następnie wywołamy:

```
print @tab;
```

to otrzymamy wynik "ABC"... jeśli jednak ujmiemy odwołanie do tablicy w cudzysłowy:

```
print "@tab"
```

to wynikiem będzie "A B C". Dzieje się tak dlatego, że domyślną wartością zmiennej $" jest pojedyncza spacja, a zmienna ta działa tylko jeśli tablica jest w cudzysłowach.

W tym miejscu warto zaznaczyć, że cudzysłowy (") i apostrofy (') w perlu działają podobnie do ich wersji bashowych – apostrofy sprawiają, że tekst pomiędzy nimi jest traktowany dosłownie, podczas gdy cudzysłowy pozwalają na interpretację niektórych znaków specjalnych. Perl pozwala jednak używać obu trybów ("dosłownego" i "interpetowanego") przy użyciu w zasadzie dowolnych znaków ograniczających zamiast " czy '. Cecha ta jest nieco zaawansowana i nie będziemy je tutaj szczegółowo omawiać, ale jest przydatna, gdy pracujemy z tekstem, który posiada w sobie zarówno znaki " jak i '.

Ostatnia uwaga: łańcuchy tekstowe w perlu sklejane są za pomocą operatora kropki (". "). Jest to podobne do języka php, ale zupełnie różne od awka (gdzie sklejaniem zajmuje się operator spacji) czy basha (gdzie tekst jest sklejany bez żadnego separatora). Oczywiście możemy użyć cudzysłów/apostrofów jako sklejaczy, tak więc:

```
print "Ala " . "ma " . $zmienna . " kotow!";
```

oraz

```
print "Ala ma $zmienna kotow!";
```

są praktycznie równoważne.

6 Domyślna zmienna i cukierki składniowe

`Perl` bardzo często pozwala na wykorzystanie "magicznej" zmiennej `$_`. Zmienna ta wykorzystywana jest jako domyślny parametr wielu funkcji. Przykładowo:

```
print;     # To samo co print $_;
```

Składnia ta ma na celu zbliżenie programu `perl`a do języka naturalnego, w którym podmiot nie jest powtarzany w kolejnych zdaniach, jeśli się nie zmienił. Oczywiście zwykle nie ma wymogu korzystania ze zmiennej `$_`. Zmienna ta jest także powodem dziwnej składni pętli `for` pokazanej we wcześniejszej części instrukcji. Jest tak dlatego, że program:

```
for $_ ( @tablica )
{
   print $_;
}
```

ma takie samo znaczenie jak:

```
for ( @tablica )
{
   print;
}
```

Oczywiście możemy w pętli `for` użyć innej zmiennej zamiast `$_`.

W tym miejscu omówimy też pewne nietypowe cechy składni `perl`a. Po pierwsze, `perl` wymusza umieszczanie nawiasów klamrowych w miejscach typu instrukcja `if`, nawet jeśli "wnętrzem" `if`a jest pojedyncza instrukcja:

```
if ( warunek ) { instr; }     # poprawne w C i perlu
if ( warunek ) instr;         # poprawne w C, ale NIE w perlu
```

Uwaga ta dotyczy także pętli typu `for`, `while` i podobnych "bloków".

Po drugie, w `perl`u nawet prosta instrukcja `if` może zostać zapisana na kilka sposobów. Przykładowo, poniższe 4 instrukcje są równoważne:

```
if ( A ) { B; }
B if A;
unless ( ! A ) { B; }
B unless ! A;
```

7 Podstawowa obsługa plików

Pliki w perlu są obsługiwane poprzez tak zwane uchwyty (*ang.* filehandler), które przechowują odwołanie do pliku (podobnie do uniksowych deskryptorów plików). Należy zauważyć, że uchwyty nie są zmiennymi i nie są same w sobie poprzedzone znakiem kontekstu. Przykładowym uchwytem jest UCHWYT. Zmienne mogą zostać jednak wykorzystane do przechowywania uchwytów:

```
$zmienna_na_uchwyt = UCHWYT;
```

Plik otwierane są funkcją open:

```
open UCHWYT , "sciezka";
```

Kod ten otwiera plik tylko do odczytu i jest równoważny:

```
open UCHWYT , "<" , "sciezka";
```

lub

```
open UCHWYT , "<sciezka";
```

Symbol < może zostać zastąpiony przez inne symbole, by otworzyć plik w innym trybie niż odczyt. Pliki są zamykane funkcją close:

```
close UCHWYT;
```

W tym miejscu warto wspomnieć o funkcji die, która kończy skrypt perlowy, wypisując przy tym podaną wiadomość. Często łączy się to rozwiązanie ze zmienną $!, która przechowywyje opis ostatniego błędu:

```
open UCHWYT , "sciezka" or die "Nie udalo sie otworzyc pliku
$sciezka poniewaz: $!";
```

Pliki są odczytywane z podziałem na rekordy, w sposób nieco zbliżony do awka. Separatorem rekordów wejściowych jest zmienna $/. Szczegółowy sposób zachowania tej zmiennej zostawimy na późniejsze laboratoria. Na razie wystarczy nam wiedza, że domyślną wartością $/ jest znak nowej linii, więc perl będzie odczytywał plik liniami. Możemy odczytać pojedynczą linię:

```
$kolejna_pojedyncza_linia = <UCHWYT>;
```

lub cały plik jako tablicę linii:

```
@wszystkie_pozostale_linie = <UCHWYT>;
```

Zauważmy, że oba sposoby zaczynają odczyt od aktualnego wskaźnika pliku. Jeśli więc odczytaliśmy już pół pliku to wywołanie @tablica = <UCHWYT>; umieści w tablicy tylko drugą połowę pliku.

Powyższy kod możemy użyć wraz z pętlą for oraz "magiczną" zmienną $_:

```
for ( <UCHWYT> )
{
   print;
}
```

co wydrukuje wszystkie linie pliku.

Jedną z różnic `perla` względem `awka` jest to, że `perl` NIE usuwa separatora z końca rekordu. Innymi słowy, powyższy kod wydrukuje nowe linie dlatego, że były one w pliku. Jeśli wcześniej w programie napisaliśmy `$\ = "\n"`, to `print` doda dodatkowy znak nowej linii, co oczywiście skutkuje dodatkowymi pustymi liniami w wydruku.

Aby rozwiązać ten problem należy usunąć separator z końca linii. Służy do tego funckja `chomp`. Usuwa ona *aktualną* wartość zmiennej `$/` z końca podanego argumentu. Ponieważ domyślną zmienną dla `chomp` jest, oczwiście, `$_`, to możemy napisać:

```
for ( <UCHWYT> )
{
   chomp;
   print;
}
```

Odczyt plików można także wykonać z użyciem funkcji `read`. Zapis do pliku można wykonać z użyciem znanej nam funkcji `print`, po dodaniu do niej odpowiedniego uchwytu:

```
print UCHWYT "tekst";
```

Oczywiśce przed próbą zapisu plik należy otworzyć, a po skończeniu pracy plik należy zamknąć.

8 Odczytywanie wpisów katalogowych

Aby uzyskać podobny efekt do komendy `ls`, czyli otrzymać listę wpisów katalogowych, wykorzystujemy podobne funkcje jak przy odczycie zwykłych plików, ale z prefiksem “dir”:

```
opendir $zmienna_uchwytu, $sciezka_do_katalogu or die $!;
while ( readdir $zmienna_uchwytu )
{
   print;
}
closedir $zmienna_uchwytu;
```

Powyższy kod wydrukuje na ekran wszystkie wpisy z katalogu danego ścieżką `$sciezka_do_katalogu`, wliczając w to wpisy ukryte (czyli także wpisy `"."` oraz `".."`).

9 Testowanie plików i operatory

Perl umożliwia testowanie właściwości pliku podobnie do bashowych komend test i [. Wykorzystywane są do tego operatory perlowe. Operator sprawdzania czy $sciezka jest istniejącym plikiem regularnym:

```
if ( -f $sciezka )
```

jest dostępny tak samo jak operator większości:

```
if ( $zmienna > 5 )
```

Oczywiście operator > jest dwuargumentowy, podczas gdy -f jest jednoargumentowy. Ponadto, w powyższym przykładzie wartością zmiennej może być prawdziwa ścieżka (jak "sciezka/do/pliku") lub uchwyt do pliku. Można też użyć uchwytu wprost, bez posługiwania się zmienną:

```
if ( -f UCHWYT )
```

Operatorów tego typu można używać w dowolnym miejscu, w którym spodziewamy się wyrażenia przyjmującego wartość logiczną. W celu znalezienia opisu operatorów plikowych w perldocu należy w pasku wyszukiwania wpisać "-x".

Pozostałe operatory w perlu są podobne do tych wykorzystywanych w komendzie test, z jedną różnicą. W komendzie test do porównywania łańcuchów używaliśmy operatorów = (lub ewentualnie ==) oraz !=, zaś do porównywania liczb używaliśmy operatorów -eq, -ne itp. W perlu jest odwrotnie:

- operatory == (nie =, które oznacza przypisanie!) oraz != służą do porównywania liczb, wraz z >, <, >=, <= itp.,
- operatory eq oraz ne służą do porównywania tekstu.

Widzimy więc, że operatory perla zawierają cechy zarówno operatorów z języka C/C++ jak i tych znanych z basha.

10 Inne funkcje

Wszystkie ważne komendy znane z basha, takie jak cp, mv, rm, readlink czy ln posiadają swoje perlowe odpowiedniki. Jednakże nie zawsze funkcja perlowa nazywa się tak samo jak komenda basha. Przykładowo do tworzenia linków symbolicznych w bashu służyła komenda ln, zaś w perlu odpowiada za to funkcja symlink.