Systemy operacyjne 07

Skrypty w perlu

1. Wyrażenia warunkowe

Zadaniem wyrażeń warunkowych jest kierowanie do właściwego miejsca w programie i wykonanie go w zależności od wyniku przeprowadzonego testu. Wyrażenia te, w przeciwieństwie do pętli wykonywane są tylko raz. Konstrukcja wygląda następująco

```
if (test1) {
    # blok1;
  } elsif (test2) {
    # blok2;
} elsif (test3) {
    # blok3;
}
...
else {
    # blokN;
}
```

test1, test2, ... są to dowolne wyrażenia, których wartością może być prawda lub fałsz. Grupa poleceń elsif oraz else jest opcjonalna. Zasada działania jest prosta - przypadku, gdy test1 zwróci wartość prawdziwą, wykonane zostaną polecenia z blok1, pozostała część wyrażenia zostanie zignorowana. W przypadku gdyby test1 okazał się fałszem, sprawdzane są kolejno test2, test3, itd., i w zależności od zwracanych wartości wykona się blok2, blok3, itd. Gdyby wszystkie testy zwróciły wartość fałszywą, wykonany zostanie blok po else.

Innym wyrażeniem jest unless, które jest przeciwieństwem wyrażenia if:

```
unless (test) {
    # blok1;
} else {
    # blok2;
}
```

Instrukcje z blok1 wykonają się w przypadku zwrócenia fałszu przez test. W przeciwnym razie wykona się blok2.

2. Pętle w perlu

1. Pętla while

Zadaniem pętli while jest wykonywanie określonego bloku instrukcji tak długo, jak długo test będzie zwracał prawdę. Budowa pętli:

```
while (test) {
    # blok poleceń;
}
```

Jeśli pominięty zostanie test, pętla stanie się pętlą nieskończoną. Programista musi zadbać o to, aby wyrażenie testowe w pewnym momencie zwróciło wartość fałszywą, w przeciwnym razie nie będzie możliwe zatrzymanie działania pętli

2. Pętla until

Pętla ta jest podobna w działaniu do pętli while. Jedyna różnica polega na tym, że wykonuje się ona tak długo, jak długo test zwraca fałsz:

```
 until (test) {
    # blok poleceń;
}
```

3. Pętla do

Jest to trzecia odmiana pętli while. Składnia:

```
do {
    # blok poleceń;
} while (test); lub do {
    # blok poleceń;
} until (test);
```

W pętli tej najpierw wykonywany jest blok poleceń, a dopiero potem następuje sprawdzenie warunku kontynuacji pętli. Powoduje to, że taka pętla w przeciwieństwie do while i until wykona się co najmniej jeden raz.

4. Pętla for

Podstawowym wykorzystaniem pętli while jest wykonywanie bloku instrukcji przez cały czas spełnienia jakiegoś warunku. Pętla for znajduje zastosowanie wtedy, kiedy dany blok ma być wykonany określoną ilość razy. Składnia:

```
for (warunek_początkowy; test; zmiana_indeksu) {
    # blok poleceń;
}
```

warunek_początkowy określa początkowy stan licznika pętli, test jest wyrażeniem sprawdzającym, czy kontynuować lub zakończyć pętlę, natomiast zmiana_indeksu jest wyrażeniem służącym do modyfikowania licznika pętli. Po uruchomieniu pętli następuje inicjalizacja licznika pętli, po czym sprawdzany jest warunek testowy. Jeśli zwróci on prawdę, wykona się blok poleceń pętli. Po jego zakończeniu następuje modyfikacja licznika pętli i ponowne sprawdzenie warunku. Cały cykl powtarza się tak długo, dopóki test będzie zwracał prawdę. Przykład pętli for wyświetlający 5 kolejnych liczb naturalnych:

```
for ($i = 1; $i < 6; ++$i) {
    print $i;
}
```

5. Pętla foreach

Pętla foreach pobiera argument w postaci listy i wykonuje na nim blok instrukcji. Warunkiem zakończenia pętli jest przejście poprzez wszystkie elementy danej listy. Budowa:

```
foreach $zmienna (@tablica) {
    # blok poleceń;
}
```

Zmienna $zmienna będzie przyjmować kolejne wartości z tablicy @tablica, i na niej będą wykonywane wszystkie polecenia bloku. W przypadku tablicy rozproszonej:

```
foreach $zmienna (keys %tablica_rozpr) {
    # bok poleceń;
}
```

należy dodać słowo kluczowe keys, które spowoduje, że $zmienna będzie przyjmować wartości kolejnych kluczy tablicy rozproszonej i dzięki nim uzyskiwać dostęp do wartości.

Argumentem pętli foreach może być także operator zakresu:

```
foreach $zmienna (1..5) { ... }
foreach $zmienna ('a'..'z') { ... }
```

a także lista zmiennych:

```
foreach $zmienna (2, 'wtorek', 'Ola', $wiek) { ... }
```

6. Wyrażenia kontrolujące wykonywanie pętli

Mimo iż pętle przerywane są w momencie niekorzystnego wyniku testu, czasem zachodzi potrzeba wcześniejszego przerwania lub ponowienia jej wykonania. W Perlu istnieją trzy polecenia służące do kontrolowania pętli:

• **last** - powoduje natychmiastowe przerwanie wykonania pętli. Dalej wykonywane są polecenia poza nią,
• **next** - przerywa wykonanie bieżącej iteracji pętli. Następuje powrót na jej początek do miejsca sprawdzania warunku,
• **redo** - powoduje ponowne wykonanie bieżącej iteracji pętli. Różnica między redo i next polega na tym, że redo nie sprawdza warunku pętli ani nie modyfikuje zawartości licznika.

Warto dodać, że wyrażenia te nie mogą być stosowane w pętlach typu do. Ważną kwestią jest to, że polecenia te działają tylko w otaczającej je pętli. Jeśli za ich pomocą chcieć by przerwać wykonanie kilku zagnieżdżonych pętli, można posłużyć się etykietami:

```
ETYKIETA: while (test1) {
    # instrukcje;
    while (test2) {
        # instrukcje;
        if (test3) {
            last ETYKIETA;
        }
    }
}
```

W tym przypadku polecenie last ETYKIETA powoduje wyjście z obu pętli do ponownego sprawdzenia testu pętli zewnętrznej.

7. Sortowanie list

Do przeprowadzania operacji sortowania list służy funkcja sort. Domyślnie funkcja ta sortuje według kodów ASCII, a wynik sortowania zwraca jako nową listę, którą da się przypisać do zmiennej tablicowej. Można też elementy sortować według liczb - była o tym mowa przy omawianiu tablic. Za pomocą tej funkcji można także sortować elementy tablic rozproszonych. Jedną z metod jest sortowanie według kluczy:

```
@klucze = sort keys %tablica_rozproszona;
@klucze = sort {$a <=> $b} keys %tablica_rozproszona;
```

W pierwszym przypadku w tablicy @klucze znajdą się posortowane według ASCII klucze tablicy %tablica_rozproszona; w drugim przypadku klucze zostaną przypisane w porządku numerycznym. Wyrażenia umieszczone w nawiasach klamrowych za słowem sort określają, w jaki sposób dana lista ma być sortowana. Wyrażenie to ma do dyspozycji dwa argumenty: $a i $b. Zwraca ono wartość ujemną, kiedy pierwszy z nich jest mniejszy od drugiego, zero - gdy oba argumenty są równe oraz wartość dodatnią, gdy pierwszy jest większy od drugiego. Tworzenie procedur sortujących umożliwiają operatory porównania: cmp - dla łańcuchów oraz <=> dla liczb. Zmienne $a i $b występują tylko wewnątrz procedury sortującej i znikają w momencie jej zakończenia; ich zadaniem jest przechowywanie elementów listy podlegającym sortowaniu. Tworząc własne procedury sortujące można uzyskać niemal dowolne rezultaty. Przykład poniżej sortuje elementy listy @liczby w kolejności malejącej:

```
@malejaco = sort {$b <=> $a} @liczby;
```

Procedurę sortującą warto wykorzystać, kiedy konieczne jest posortowanie elementów tablicy %tablica według ich wartości:

```
@klucze = sort {$tablica{$a} cmp $tablica{$b}} keys %tablica;
```

W tym przypadku w tablicy @klucze znajdą się klucze uporządkowane według kodów ASCII ich wartości.

8. Przeszukiwanie list

Przeszukiwanie listy ma na celu odnalezienie określonych elementów. Chcąc odnaleźć właściwy element listy można korzystając z pętli foreach przejść przez wszystkie elementy danej listy i każdy z nich porównać ze wzorcem. Z kolei celu znalezienia podłańcucha w innym łańcuchu można skorzystać z funkcji index - funkcja ta zwraca jego pozycję lub wartość ujemną, kiedy podłańcuch nie występuje w łańcuchu głównym:

```
$lancuch = 'poniedziałek';
$szukany = 'nie';
$pozycja = index $lancuch, $szukany;
```

W tym konkretnym przypadku funkcja index zwróci wartość 2 i przypisze ją do zmiennej $pozycja. Do wydobycia wszystkich elementów listy pasujących do wzorca służy funkcja grep. Zwraca ona listę tych elementów:

```
@nowa = grep {$_ < 50} @liczby;
```

Polecenie to zapisze do tablicy @nowa wszystkie liczby z tablicy @liczby mniejsze od 50. Działanie tej funkcji polega na pobieraniu kolejnych elementów listy, przypisywaniu ich do zmiennej specjalnej $_, po czym następuje porównanie ich z wzorcem. Wyrażenie w nawiasach klamrowych powinno zwracać prawdę lub fałsz w zależności od tego, czy dany warunek jest spełniony czy też nie. Na podstawie wyniku tego wyrażenia element dodawany jest do nowej listy lub jest ignorowany. Wyrażeniem tym może być także wyrażenie regularne.

9. Modyfikowanie elementów list

W Perlu istnieją następujące funkcje służące do modyfikacji list:

- push: dodaje element (listę) na koniec listy,
- pop: usuwa element z końca listy,
- unshift: dodaje element (listę) na początek listy,
- shift: usuwa element z początku listy,
- splice: dodaje lub usuwa dowolny element listy.

Funkcja push pobiera 2 argumenty: listę do zmodyfikowania oraz listę argumentów do dodania:

```
push @modyfikowana, @inna_tablica;
push @modyfikowana, (element1, element2, ...);
```

Funkcja zwraca ilość elementów w nowej tablicy.
Funkcja pop działa odwrotnie - usuwa element z końca listy zwracając go jako wartość:

```
pop @tablica;
```

Funkcja unshift jest odpowiednikiem funkcji push tyle że operuje ona na początkowych elementach listy. Operacja:

```
unshift @modyfikowana, @elementy;
unshift @modykifowana, (element1, element2, ...);
```

spowoduje, że na początek tablicy @modyfikowana zostanie wstawiona lista elementów podana jako drugi argument. Wartością zwracaną przez funkcję jest ilość elementów nowej tablicy.

Funkcja shift działa odwrotnie - powoduje zdjęcie pierwszego elementu listy i zwrócenie go jako wynik działania:

```
shift @tablica;
```

O wiele więcej możliwości posiada funkcja splice. Pobiera ona 4 argumenty: tablicę do zmodyfikowania, miejsce w tablicy za którym ma być dodany (usunięty) element, liczbę elementów do usunięcia (zastąpienia) oraz liczba elementów do dodania. Jej działanie najlepiej zilustrować na przykładzie:

```
@liczby = 0..9;
splice(@liczby, 5, 3);
```

W trakcie tej operacji z tablicy @liczby usunięte zostaną 5, 6 oraz 7. Gdyby pominąć trzeci argument (3), usunięte zostałyby elementy: 5, 6, 7, 8 i 9, tzn. od 5 aż do końca tablicy. Dzięki tej funkcji można także zastępować elementy innymi:

```
splice(@liczby, 5, 3, (el1, el2, el3));
```

Elementy 6, 7 i 8 tablicy @liczby zostaną zastąpione przez el1, el2 i el3. Lista (el1, el2, el3) przeznaczonych do dodania elementów może być dowolna (nie musi być równa ilości usuwanych elementów) - w takim przypadku tablica @liczby zostałaby odpowiednio powiększona lub pomniejszona. Gdyby jako trzeci argument tej funkcji wstawić zero, wtedy żaden z elementów tablicy @liczby nie zostałby usunięty.

10. Budowa procedur

Procedury są grupą poleceń, które umieszczone w jednym miejscu programu mogą być wywoływane dowolną ilość razy. Wykonują one określone operacje na przekazanych danych, po czym zwracają wyliczoną wartość. Dane przekazane do procedury nazywamy argumentami. Procedury należy stosować tam, gdzie zachodzi potrzeba wielokrotnego wykonania powtarzających się czynności. Podstawowa składnia procedury ma następującą postać:

```
sub nazwa {
    # instrukcje;
}
```

Jej definicja rozpoczyna się słowem sub, po którym umieszczana jest nazwa procedury. Wszelkie instrukcje, jakie mają zostać wykonane w procedurze muszą zostać umieszczone wewnątrz nawiasów klamrowych. Zasada nazewnictwa procedur jest taka sama jak w przypadku zmiennych skalarnych czy tablic.

11. Wywoływanie procedur

Najprościej procedurę można wywołać w następujący sposób:

```
&nazwa_procedury();
```

Znak & jest opcjonalny, ale w celu odróżnienia procedury od wbudowanej funkcji Perla warto go stosować. Wywołanie procedury może nastąpić w dowolnym miejscu w programie, także wewnątrz innej procedury.

12. Zwracanie wartości przez procedury

Każda procedura może zwracać jakąś wartość. Domyślnie jest nią wynik ostatniego wyliczenia wewnątrz procedury. W przykładzie poniżej:

```
sub iloczyn {
    # instrukcje;
    $liczba1 * $liczba2;
}
```

wartością zwróconą będzie iloczyn liczb w zmiennych $liczba1 i $liczba2. Wartość tę można przypisać do zmiennej:

```
$iloczyn = &iloczyn();
```

Aby mieć pełną kontrolę nad wartościami zwracanymi przez procedurę, należy w jej wnętrzu użyć funkcji return. Wyrażenie obliczone za funkcją return zostanie zwrócone jako wynik działania procedury. W przykładzie powyżej taki sam efekt można osiągnąć stosując następującą definicję procedury:

```
sub iloczyn {
    # instrukcje;
    return $liczba1 * $liczba2;
}
```

Oczywiście dzięki funkcji return procedura może zwrócić więcej niż jeden element:

```
sub lista {
    # instrukcje;
    return ($wartosc1, $wartosc2, $wartosc3);
}
```

Wartości zwrócone przez procedurę można np. przypisać do listy:

```
@lista = &lista;
```

13. Przekazywanie argumentów do procedury

Argumenty przekazuje się do procedury w momencie jej wywołania. Mogą nimi być dowolne dane: liczby, łańcuchy, zmienne skalarne oraz tablice zwykłe i rozproszone. Wszystkie te elementy, jeszcze zanim zostaną przekazane, rozwijane są do postaci jednej spójnej listy.

```
&procedura($skalar, @array, "lancuch");
```

Lista argumentów przechowywana jest w lokalnej zmiennej @_ dostępnej wewnątrz procedury. Poniższa procedura powoduje zsumowanie trzech przekazanych jej argumentów:

```
sub suma {
    return $_[0] + $_[1] + $_[2];
}

$suma = &suma($argument1, $argument2, $argument3);
```

gdzie $argument1, $argument2 i $argument3 są zmiennymi zawierającymi liczby do zsumowania. W tym konkretnym przykładzie z góry wiadomo, jaka liczba elementów zostanie przekazana do procedury. Gdyby liczba argumentów nie była z góry ustalona, w celu wykonania powyższej operacji możnaby zastosować np. pętlę foreach dla każdego elementu tablicy @_ i przy każdej iteracji zwiększać zmienną przechowującą wynik o wartość tego elementu.

14. Przykłady

1. Operacje na plikach

Wersja domyślnie używana to 5.0, nie wspiera ona wielu elementów w tym standardowego od ok. 2000 roku dostępu do plików. Przy
obsłudze plików należy użyć wersji 5.8, czyli:

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

zamieniamy na
`#!/usr/local/bin/perl`
Co umożliwi działanie wersji 5.8 (ca. 2004) a nie 5.0 (z 1999).
Otwieranie pliku:

```
open (my $FH, '<', "/etc/passwd") or die("Cannot open
file!");
```

Zmienna $FH jest naszym uchwytem. Odczytajmy plik linia po linii:

```
while (my $linia = <$FH> ) {
    print $linia;
}
```

Oczywiście możemy tutaj użyć instrukcji foreach (`foreach my $linia (<$FH>)`), ale to utworzy nam tymczasową tablicę zawierającą wszystkie linie z pliku, co nie jest korzystne pamięciowo. W pętli while za każdym razem z uchwytu odczytywana jest jedna linia. Zamiast '<' możemy otworzyć plik do zapisu ">" lub dopisać do pliku ">>". Po zakończeniu korzystania z pliku należy go zamknąć (`close $FH;`) aby wyczyścić bufor i zapisać zaległe dane.
Drugi przykład:

```
open(my $FH, '<', "/etc/passwd") or die("cannot access
file!");
open(my $OPH, ">", "test.txt") or die("cannot access
savefile!");
while (my $linia = <$FH>) {
if (index($linia,"is2014") == -1) {
next;
}
my @Q=split(':',$linia);
print "Znalazlem: ".$Q[2]." ".$Q[4]."\n";
print $OPH $Q[2]." ".$Q[4]."\n";
}
close $FH;
close $OPH;
```

Użyte funkcje:
`index($string, $substring)` – zwraca położenie $substring w $string. -1 gdy go nie znajdzie. Użyteczne w wyszukiwaniu.
`split($delimiter, $lancuch)` – zwraca tablicę ciągów po podzieleniu łańcucha wejściowego.
Uwaga: Podejście print `split(':', $linia)[1]` nie uruchomi się ze względu na nieznany typ
podczas parsowania.
`print $HANDLE string...` – drukuje ciąg tekstowy do pliku wskazanego przez $HANDLE

2. Wyrażenia regularne

Podstawowym narzędziem do modyfikowania łańcuchów tekstowych w Perlu są jednak rozbudowane mechanizmy wyrażeń regularnych. Powyższy przykład można przepisać

```
$str="Ala ma kota";
$str =~ s/ma/miala/;
print $str;
```

z takim samym wynikiem.
Zawsze wygląda do tak:
funkcja/regex/zamiennik/modyfikatory;
Najczęściej stosowane są funkcje:
s – zamień (substitute)
m – dopasuj (match) – np. if ($str =~ m/ma/) { ...
Często stosowane modyfikatory:
g – global – zamienia wszystkie wystąpienia, nie tylko pierwsze
i – ignore case – ignoruje małe/wielkie litery.
x – eXclude whitespace – wyklucza białe znaki chyba, że są escape'owane.
Modyfikatory można łączyć np. s/ma/miala/ig

3. Procedury

```
sub maks {
    my $maksimum=-65536;
    foreach my $a (@_)
    {
      if ($a>$maksimum)
      {
        $maksimum=$a;
      }
    }
return $maksimum;
}
```

Wykonać możemy taką funkcję przez np.:

```
my @k=(2, 3, 17, 8);
print maks(1, 2, 7, 6, 4, -1, @k, 10);
```

4. Uruchomienie zewnętrznych programów

```
system(”ps -u $uzytkownik”);
```

Zadania do wykonania

1. Napisz prosty kalkulator zapisujący wynik do pliku.
2. Napisz skrypt przyjmujący w parametrach ciąg liczb całkowitych, a wyprowadzający na ekran ich średnią arytmetyczną.

3. Napisz program drukujący tabliczkę mnożenia o wymiarze zadanym w pierwszym argumencie programu. Uwzględnij przypadek gdy użytkownik nie podał argumentu.
4. Napisz program, który przeszuka dany plik tekstowy pod kątem numerów dowodów osobistych (3 wielkie litery i 6 cyfr) a następnie zastąpi wszystkie ich wystąpienia 9 znakami X zapisując wynik do drugiego pliku.