

## Narzędzia informatyczne w językoznawstwie

## Perl - Struktury kontrolne i zmienne

Marcin Junczys-Dowmunt  
junczys@amu.edu.pl

Zakład Logiki Stosowanej  
<http://www.logic.amu.edu.pl>

28. listopada 2007

- ▶ Dziś wprowadzimy różne typy danych oraz różne rodzaje zmiennych zawierające takie dane
- ▶ Omówimy podstawowe struktury kontrolne
- ▶ Na końcu przejdziemy do małych programików nadających się np. do eksperymentów psycholingwistycznych

## Liczby i łańcuchy znaków

Na razie będziemy korzystać z dwóch podstawowych typów danych: liczb i łańcuchów znakowych.

## Liczby

- ▶ Dopuszczalne są wartości całkowite, zmiennoprzecinkowe, ujemne, wykładnicze itp.
- ▶ Perl nie rozróżnia tych rodzajów liczb, jak to się dzieje np. w C

## Łańcuchy znakowe

- ▶ Wszystko co jest ujęte w podwójny lub pojedynczy cudzysłów nazywamy łańcuchem znakowym
- ▶ W Perlu nie wszystkie łańcuchy znakowe możemy nazwać stałymi (te w pojedynczym cudzysłowie na pewno)

## Liczby i łańcuchy znakowe

```

1  print(3);
   print(" ");
   print(3 + 4);
   print(" ");
5  print("3 + 4");
   print("\n");
   print(5 * 2);
   print(" ");
   print(9/3);
10 print(" ");
   print(2 ** 8);
   print("\n");

```

Proszę opisać, co się dzieje w każdym wierszu.

Jaki działa cudzysłów podwójny a jak cudzysłów pojedynczy?

Istnieją też operatory dla łańcuchów znakowych oraz operatory mieszane np.

```
1 print("--" x 50);  
  print("\n");  
  print("konkatenacja" . " " . "łańcuchów\n");  
  print("--" x 50);  
5 print("\n");
```

Na zmiennych możemy wykonywać takie same operacje jak na stałych plus operację przypisania (=):

```
1 $one = 2;  
  $two = 3;  
  $three = $one + $two;  
  print($three . "\n");  
  
1 $hat = "chair";  
  $chair = "hat";  
  print($chair . " " . $hat . "\n");
```

Jak widać, nazwa i treść zmiennej nie muszą być ze sobą powiązane.

Zmienne zawierające liczby, znaki łańcuchowe i referencje<sup>1</sup> nazywamy zmiennymi skalarnymi.

Struktura nazwy zmiennej skalarnej w Perlu:

- ▶ Pierwszy znak to przedrostek \$ (podobny do s jak *scalar*)
- ▶ Drugi znak to dowolna litera lub znak podkreślenia \_
- ▶ Kolejne znaki to dowolne litery lub liczby lub znak podkreślenia

Przykłady:

\$\_, \$a, \$abc, \$a12, \$to\_jest\_zmienna, \$a\_to\_zmienna\_o\_numerze\_2

<sup>1</sup>o referencjach więcej innym razem

Zamiast łączyć kilka zmiennych w większy łańcuch znakowy za pomocą operatora konkatenacji, możemy wykorzystać możliwości Perla w zakresie interpolacji zmiennych.

Interpolacja to operacja, która zamienia nazwę zmiennej w łańcuchu znakowym na jej wartość.<sup>2</sup>

```
1 $hat = "chair";  
  $chair = "hat";  
  print("$chair $hat\n");
```

Pytanie: Co stanie się, gdy zastąpimy podwójny cudzysłów pojedynczym cudzysłowem?

<sup>2</sup>To przez interpolację zmiennych nie możemy mówić, że wszystkie łańcuchy znakowe to stałe.

- ▶ Zmienne tablicowe to zmienne złożone
- ▶ Dokładniej: to listy zmiennych skalarnych
- ▶ Każda z tych zmiennych skalarnych w tablicy ma jednoznacznie przyporządkowany indeks (numer)
- ▶ Tablica zawierająca  $n$  elementów jest indeksowana liczbami od 0 do  $n - 1$
- ▶ Przedrostek tablicy w Perlu to @ (podobne do **a** jak *array*), np. @tablica
- ▶ Nie ma górnego ograniczenia liczby elementów tablicy (tylko pamięć danego komputera)
- ▶ Tablice są strukturami dynamicznymi (długość nie jest stała)

- ```
1 @rok = ("styczen", "luty", "marzec");
   print "Trzeci miesiac roku: ".$rok[2]."\n";
   print "Pierwszy miesiac roku: $rok[0]\n";
```
- ▶ Kanoniczny sposób zapisywania wartości do tablicy
  - ▶ Elementy zostaną ponumerowane od 0 do 2
  - ▶ Odwołując się do wartości elementów tablicy korzystamy z \$, ponieważ wartości tablicy to *skalary*!
  - ▶ Wartości tablicy też podlegają interpolacji zmiennych

```
4 $rok[3] = "kwiecien";
5 $rok[4] = "maj";
  $rok[5] = "czerwiec";
```

- ▶ Możemy zapisywać wartości bezpośrednio do elementów skalarnych tablicy
- ▶ Elementy tablicy nie różnią się niczym od normalnych zmiennych
- ▶ Jeśli taki indeks wcześniej nie istniał, zostanie stworzony
- ▶ Ewentualne wartości pośrednie zostaną wypełnione wartością `undef` (niezdefiniowana)

```
7 @kwartal3 = qw( lipiec sierpien wrzesien );
  @rok[6 .. 8] = @kwartal3;
  print join(" - ",@rok[6 .. 8])."\n";
```

- ▶ Skrótowy sposób inicjalizacji tablicy (dla listy łańcuchów znakowych)
- ▶ Białe znaki funkcjonują jak separatory
- ▶ Możemy też zapisywać do podprzedziału tablicy
- ▶ Wycinek tablicy jest tablicą, stąd przedrostek @
- ▶ Funkcja `join` łączy daną tablicę w łańcuch znakowy z podanym separatorem wstawionym między elementami

## Ustawianie zmiennych tablicowych

```
10 @rok[9,10,11] = qw( październik listopad grudzien );
    print "Następujące miesiące mają 31 dni: \n";
    print join("\n",@rok[0,2,4,6,7,9,11])."\n";

    print "Następujące miesiące mają 30 dni: \n";
15 @m30 = (3,5,8,10);
    print join("\n",@rok[@m30])."\n";

    print "A $rok[1] ma 28 dni\n";
```

- ▶ Wycinki tablic to nie tylko przedziały liczb. Możemy korzystać z list dowolnych liczb
- ▶ Lub nawet z innych tablic

## Struktury kontrolne - warunki

```
1  if (wyrażenie) { blok } else { blok }

    unless (wyrażenie) { blok } else { blok }

5  if (wyrażenie1) { blok }
    elsif (wyrażenie2) { blok }
    ...
    elsif (ostatniwyrażenie) { blok }
    else { blok }
```

- ▶ W Perlu nie ma typów logicznych jako takich, fałszywe są: liczba zero, łańcuch zerowy, wartość undef, puste tablice (zwykłe i asocjacyjne). Pozostałe są prawdziwe
- ▶ W blokach możemy umieścić dowolną liczbę instrukcji

## Operatory porównania

| Numeryczne | Łańcuchowe | Znaczenie             |
|------------|------------|-----------------------|
| ==         | eq         | jest równe            |
| !=         | ne         | nie jest równe        |
| <=>        | cmp        | porównanie ze znakiem |

  

| Numeryczne | Łańcuchowe | Znaczenie          |
|------------|------------|--------------------|
| >          | gt         | większe niż        |
| >=         | ge         | większe-równe niż  |
| <          | lt         | mniejsze niż       |
| <=         | le         | mniejsze-równe niż |

Porównując liczby korzystamy z operatorów numerycznych.  
Porównując łańcuchy korzystamy z operatorów łańcuchowych.

## Przykład if

```
1  @tab1 = qw(Antek Bartek);
    @tab2 = ("Edek", "Dudek");

    if(@tab1 == @tab2) {
5      if($tab1[0] lt $tab2[0]) {
            print "$tab1[0] poprzedza $tab2[0]\n";
        }
        elsif($tab1[0] gt $tab2[0]) {
            print "$tab2[0] poprzedza $tab1[0]\n";
10        }
        else {
            print "$tab1[0] jest równe $tab2[0]\n";
        }
    }
}
```

## Pętle

W Perlu mamy trzy główne rodzaje pętli:

- ▶ pętlę typu `while` znamy z innych programów (np. Pascal)

```
1 while (wyrażenie) {
    block
}
```

- ▶ pętla typu `for` w takiej postaci jest znana z języków podobnych do C

```
1 for (pierwsza i.; warunek; i. iterowana) {
    block
}
```

- ▶ pętla typu `foreach` jest konstrukcją typową dla Perla

```
1 foreach $element (lista) {
    block
}
```

## Pętla `while` - przykład

```
1 @pokoj = qw(krzeslo stol lozko szafa);

    print "Pokoj zawiera nast. meble: ".join(", ", @pokoj);

5 while (@pokoj > 2) {
    $mebel = pop @pokoj;
    print "Wyrzucilem przez okno: $mebel\n";

    print "Pokoj zawiera nast. meble: ";
10    print join(", ", @pokoj)."\n";
}
```

- ▶ Zmienna `@pokoj` użyta w kontekście numerycznym jest interpretowana jako liczba elementów
- ▶ funkcja `pop` usuwa ostatni element z podanej tablicy i zwraca ten element
- ▶ Pętla `while` działa dopóki warunek jest prawdziwy, tzn. póki tablica `@pokoj` zawiera więcej niż dwa elementy

## Pętla `for` - przykład

```
1 for ($i = 0; $i < 10; $i++) {
    print "Wartość \ $i to: $i\n";
}
```

Program podobny do `while.pl` :

```
1 @pokoj = qw(krzeslo stol lozko szafa);
    print "Pokoj zawiera nast. meble: ";
    print join(", ", @pokoj)."\n";

5 for ($i = pop @pokoj; @pokoj > 1; $i = pop @pokoj) {
    print "Wyrzucilem przez okno: $i\n";

    print "Pokoj zawiera nast. meble: ";
    print join(", ", @pokoj)."\n";
10 }
```

## Pętla `foreach` - eksperymenty psycholingwistyczne

Dla przykładu pętli `foreach` posłużymy się pewnym eksperymentem psycholingwistycznym. Napiszemy program generujący wszystkie sylaby pewnego polinezyjskiego języka o nazwie *Tulu*

W tym języku wszystkie sylaby mają postać CV z ograniczonego zestawu spółgłosek i samogłosek:

{a, e, i, k, l, m, o, p, t, u}

Sylaby będą wyglądały np. tak: *po, ta, ku, me*

## Pętla foreach - eksperymenty psycholingwistyczne

```
1 @consonants = qw(k l m p t);  
  
@vowels = qw(a e i o u);  
  
5 foreach $c (@consonants) {  
    foreach $v (@vowels) {  
        print("$c$v\n");  
    }  
}
```

## Pętla foreach - eksperymenty psycholingwistyczne

Teraz napiszemy program, który generuje wszystkie wyrazy *Tulu*.  
Każdy wyraz ma postać CVCV. Jak tego dokonać?

```
1 @consonants = qw(k l m p t);  
  
@vowels = qw(a e i o u);  
  
5 foreach $c1 (@consonants) {  
    foreach $v1 (@vowels) {  
        foreach $c2 (@consonants) {  
            foreach $v2 (@vowels) {  
                print("$c1$v1$c2$v2\n");  
            }  
        }  
    }  
}
```

## Pętla foreach - eksperymenty psycholingwistyczne

Zasady morfonologiczne *Tulu* narzucają nam dodatkowe ograniczenie. Wszystkie samogłoski danego wyrazu muszą być takie same, np. *paka*, *tulu*

```
1 @consonants = qw(k l m p t);  
  
@vowels = qw(a e i o u);  
  
5 foreach $c1 (@consonants) {  
    foreach $v (@vowels) {  
        foreach $c2 (@consonants) {  
            print("$c1$v$c2$v\n");  
        }  
    }  
}
```

## Pętla foreach - eksperymenty psycholingwistyczne

Kolejna reguła morfonologiczna języka *Tulu* mówi, że spółgłoski muszą być różne. Tzn. ciągi typu *tutu*, *papa* itd. nie są poprawnymi słowami języka *Tulu*

```
1 @consonants = qw(k l m p t);  
  
@vowels = qw(a e i o u);  
  
5 foreach $c1 (@consonants) {  
    foreach $v (@vowels) {  
        foreach $c2 (@consonants) {  
            print("$c1$v$c2$v\n") if($c1 ne $c2);  
        }  
    }  
}
```