Wprowadzenie do Perla

Tadeusz Sośnierz

Slajdy: `https://tadzik.net/bioperl.pdf`

Czym jest Perl?

Perl jest dynamicznie typowanym językiem programowania – czasem nazywany językiem skryptowym.

Perl skupia się na tym, aby nam było łatwo napisać nasz program i rozwiązać nasz problem – niekoniecznie na tym, żeby komputerowi było łatwo go uruchomić.

```
1 use 5.028;
2 say "Hello, world!"
```

Powyższy przykład, choć trywialny mówi nam nieco o naturze samego języka.

Wersja Perla

Na początku większości programów w Perlu znajdziemy deklarację wersji Perla, pod którą pisany był program. Np dla `5.28`:
`use 5.028;`

Perl bardzo stara się nie łamać *kompatybilności wstecznej* – istotne jest, żeby program napisany 10 (albo 20) lat temu nadal uruchamiał się bez błędów.

Dlatego też żadne nowe elementy języka (wprowadzone po wersji 5.8 z 2002 roku) nie będą dla nas dostępne, dopóki nie powiemy, że chcemy ich używać.

Możemy zatem powiedzieć, że Perl jest językiem konserwatywnym.

`say` i inne funkcje

`say` z naszego przykładu będzie pierwszą funkcją, którą poznamy: służy do wypisania tekstu na ekranie.

Funkcje w Perlu możemy pisać z nawiasami albo bez – w zależności od tego, co nam bardziej odpowiada:

```
1 use 5.028;
2
3 say "Hello, world!"; # albo...
4 say("Hello, world!");
```

Utworzony przez Larry'ego Walla – programistę, ale też lingwistę – Perl stara się być nieco jak język naturalny, który ma swoje dialekty, i można w nim mówić zarówno prostymi słowami jak i w bardziej wysublimowany sposób.

`say` a wersje Perla

Funkcja `say` to nic innego jak funkcja **`print`** która dodaje na końcu tekstu znak nowej linii:

```
1 print "Hej\n";
2 say "Hej";
```

Powyższy program jednak uruchomi się z błędem:

```
1 String found where operator expected at -e
    line 1, near "say "hej""
2     (Do you need to predeclare say?)
```

Nawet tak trywialna rzecz jak `say`, dodane w wersji 5.10, wymaga włączenia za pomocą **`use`** `<wersja>`.

Krótkie podsumowanie

Co zatem możemy powiedzieć o Perlu po tych dwóch linijkach kodu?

- Jest to stary język (powstał w 1987), ale nadal rozwijany (nowe stabilne wersje wychodzą co roku w czerwcu, najnowsza jest 5.30)
- Perl rozwija się bardzo konserwatywnie, i nowe rzeczy dodaje do języka bardzo ostrożnie (często przez lata pozostają oznaczone jako eksperymentalne)
- Perl jest językiem, gdzie możemy wyrobić sobie własny styl i mieć pewną dowoloność w tym, w jakim stylu chcemy pisać: co nie zawsze jest dobrą cechą przy pracy w dużym zespole

Gdzie się go stosuje?

Perl, jako język dynamiczny nie jest demonem wydajności – często stosuje się go jednak tam, gdzie wydajnośćnie jest kluczowa, oraz tam, gdzie czas pracy programisty jest cenniejszy niż czas pracy komputera:

- W aplikacjach webowych
- W administracji systemami
- W jednorazowych programach do doraźnych potrzeb
- Również w bioinformatyce :)

Perl 5 a Perl 6

Perl 6 miał w założeniu być następcą Perla 5, jednak z biegiem czasu wyewoluował w kompletnie niezależny język, w zeszłym roku przemianowany na Raku. Perl 5 jest rozwijany niezależnie, i to na nim będziemy się dziś skupiać.

Szybkie wprowadzenie

- Typy danych w Perlu
- Podstawowe instrukcje (**`if`**, **`for`** etc.)
- Funkcje, moduły i obiekty
- Wykorzystanie w praktyce: CPAN, BioPerl

Podstawy składni

Białe znaki (spacje, tabulacje, nowe linie etc.) nie mają w Perlu znaczenia.

Listy mogą kończyć się przecinkami:

```
1 my @liczby = (4, 8, 15, 16, 23, 42,); # OK
```

Każdą instrukcję (zwykle: linię) kończymy średnikiem, ale możemy go pominąć dla ostatniej instrukcji w bloku:

```
1 if (...) {
2     say "a";
3     say "b"  # brak przecinka: OK
4 }
```

Kodowanie znaków

Domyślnie kod Perlowy powinien składać się ze znaków ASCII. Możemy jednak, za pomocą **use** utf8; ustawić kodowanie kodu na UTF-8.

Wówczas możemy w kodzie używać np. Polskich znaków – nawet w nazwach zmiennych.

```
1 use utf8;
2 my $tekst = "Witaj świecie";
3 my $bąk = 'bąk';
```

Podstawowe typy danych – `$@%`

Podstawowe typy danych w perlu to **skalary** (`$`), **listy** (`@`) i **hashe** (`%`):

```
1 my $miasto = "Wrocław";
2 my $wiek   = 28;
3
4 my @liczby = (4, 8, 15, 16, 23, 42);
5 # listy mogą zawierać dowolne skalary,
6 # również referencje do innych zmiennych
7 my @mieszane = ('kot', 33, \@liczby);
8
9 my %rozszerzenia = (
10     perl   => 'pl',
11     python => 'py',
12     rust   => 'rs',
13 );
```

Przyjrzyjmy się im dokładnie po kolei.

Skalary

Skalary to zmienne które mogą przetrzymywać dowolne pojedyńcze wartości – liczby, tekst (*stringi*) lub np. referencje do innych zmiennych.
Perl jest *dynamicznie typowany*, więc każdy skalary może przechowywać różne wartości w ramach swojego istnienia.

```
1 my $zmienna = 3;
2 $zmienna = 3.1415;
3 $zmienna = 'kot';
```

Prawda i fałsz nie mają własnego typu ani słów kluczowych na poziomie kodu: zwykle używa się zamiast nich po prostu 0 i 1.

Deklaracje zmiennych

Domyślnie Perl pozwoli nam używać zmiennych, których jeszcze nie uworzyliśmy:

```
1 $x = 17;
2 say $x + $y; # 17
```

Potrafi to być źródłem wielu błędów; dużo bezpieczniej jest pracować w trybie *strict*, który wymusza na nas deklaracje zmiennych przy pomocy **my**:

```
1 use strict;
2 $x = 17;
3 # Global symbol "$x" requires explicit package
4 # name (did you forget to declare "my $x"?)
```

undef

Niezainicjalizowane (albo niezadeklarowane) zmienne mają wartość **undef** – niezdefiniowane.
Perlowych funkcje i operacje generalnie dobrze sobie radzą z wartościami **undef** – są traktowane jak puste.

```
1 my $x;
2 say $x + 5; # 5, gdyż $x jest puste
3 my $czy_x = defined($x); # false
```

Ostrzeżenia

Oprócz `strict`, warto włączyć sobie w kodzie też moduł `warnings`: pozwoli nam łatwo wychwycić typowe błędy:

```
1 use warnings;
2 my $wynik = 2 + 'trzy';
3 # Argument "trzy" isn't numeric
4 # in addition (+)
```

Dobrą praktyką jest zaczynać każdy nasz program od:

```
1 use 5.028; # albo inna wersja
2 use strict;
3 use warnings;
```

Operacje na skalarach

Dla skalarów mamy dostępne znane nam operacje matematyczne (+, −, * itp.). Tekst łączymy ze sobą za pomocą operatora `.`. To od operatora będzie zależeć typ wyniku:

```
1 my $suma = 3 + 4;       # 7
2 my $suma = '3' + '4'; # nadal 7
3
4 my $wiek = 25;
5 my $tekst = 'Mam ' . $wiek . ' lat';
6
7 # zmienne możemy zawierać w stringach,
8 # ale tylko tych w cudzysłowach:
9 my $tekst = "Mam $wiek lat"; # Mam 25 lat
10 $tekst = 'Mam $wiek lat';     # Mam $wiek lat
```

Szybkie ćwiczenie

Za pomocą `<STDIN>` możesz wczytać tekst wpisany przez użytkownika:

```
1 my $rok_urodzenia = <STDIN>;
```

Spróbuj napisać program, który wczyta od użytkownika jego rok urodzenia, a następnie wypisze jego wiek (zakładając, że jest 2020).

Udało się? Zajrzyj do dokumentacji (`https://p3rl.org/localtime`) i spróbuj napisać program który będzie działał poprawnie nawet za rok.

Instrukcje decyzyjne

Aby podejmować decyzje wewnątrz naszego programu możemy użyć instrukcji decyzyjnych: **`if`**, **`elsif`** i **`else`**:

```
1 my $wiek = <STDIN>;
2 if ($wiek < 15) {
3     say "Wstęp wzbroniony";
4 } elsif ($wiek < 18) {
5     say "Wstęp tylko z rodzicem";
6 } else {
7     say "Wstęp wolny";
8 }
```

Wcięcia wewnątrz klamer nie są wymagane, ale warto je stosować dla zachowania czytelności.

Skrócona forma **if**a

Przy **if**, **else** itd wymagane są zarówno nawiasy wokół warunku jak i klamry wokół kody. Obu tych rzeczy możemy jednak uniknąć zapisując **if**a w skróconej, odwrotnej formie:

```
1 say "Hasło zbyt krótkie" if length($pass) < 8;
```

Porównania

Podobnie jak przy operacjach matematycznych, Perl ma inne operatory do porównywania tekstu i liczb (wszystkie działania wypisują "OK"):

```
1 say "OK" if 5 == 5.0;
2 say "OK" if 55 eq '5' . '5';
3 say "OK" if 55 eq '44' + 11;
4
5 say "OK" if 5 != 6;
6 say "OK" if 'kot' ne 'pies';
7 say "OK" if 'aaa' lt 'abc';
```

Pętle

Możemy wielokrotnie wykonać kawałek kodu za pomocą pętli – **`while`** lub **`for`** (o niej za chwilę).

```
1 my $liczba = 10;
2 while ($liczba > 0) {
3     say "$liczba...";
4     # skrót od $liczba = $liczba - 1;
5     $liczba -= 1;
6 }
```

Przykład: zgadywanie liczby

Napiszmy prostą grę, w której gracz musi zgadnąć jaką liczbę wylosował program.

```
1 my $liczba = int(rand(10)) + 1; # od 1 do 10
2
```

Przykład: zgadywanie liczby

Napiszmy prostą grę, w której gracz musi zgadnąć jaką liczbę wylosował program.

```
1 my $liczba = int(rand(10)) + 1; # od 1 do 10
2
3 while(1) { # w nieskończoność
4     print "Zgadnij, jaka to liczba: ";
5     my $input = <STDIN>;
```

Przykład: zgadywanie liczby

Napiszmy prostą grę, w której gracz musi zgadnąć jaką liczbę wylosował program.

```
1 my $liczba = int(rand(10)) + 1; # od 1 do 10
2
3 while(1) { # w nieskończoność
4     print "Zgadnij, jaka to liczba: ";
5     my $input = <STDIN>;
6     if ($input < $liczba) {
7         say "To za mało";
8     }
```

Przykład: zgadywanie liczby

Napiszmy prostą grę, w której gracz musi zgadnąć jaką liczbę wylosował program.

```
1  my $liczba = int(rand(10)) + 1; # od 1 do 10
2
3  while(1) { # w nieskończoność
4      print "Zgadnij, jaka to liczba: ";
5      my $input = <STDIN>;
6      if ($input < $liczba) {
7          say "To za mało";
8      } elsif ($input > $liczba) {
9          say "To za dużo";
10     }
```

Przykład: zgadywanie liczby

Napiszmy prostą grę, w której gracz musi zgadnąć jaką liczbę wylosował program.

```
1  my $liczba = int(rand(10)) + 1; # od 1 do 10
2
3  while(1) { # w nieskończoność
4      print "Zgadnij, jaka to liczba: ";
5      my $input = <STDIN>;
6      if ($input < $liczba) {
7          say "To za mało";
8      } elsif ($input > $liczba) {
9          say "To za dużo";
10     } else {
11         say "Gratulacje!";
```

Przykład: zgadywanie liczby

Napiszmy prostą grę, w której gracz musi zgadnąć jaką liczbę wylosował program.

```
1  my $liczba = int(rand(10)) + 1; # od 1 do 10
2
3  while(1) { # w nieskończoność
4      print "Zgadnij, jaka to liczba: ";
5      my $input = <STDIN>;
6      if ($input < $liczba) {
7          say "To za mało";
8      } elsif ($input > $liczba) {
9          say "To za dużo";
10     } else {
11         say "Gratulacje!";
12         last; # momentalnie kończy pętlę
13     }
14 }
```

Obcinanie białych znaków

Jeśli wczytujemy od użytkownika tekst, a nie liczby, musimy pamiętać, że znaki wczytane z `<STDIN>` będą zawierać też znak końca linii (`\n`). Możemy go zlikwidować funkcją **chomp**:

```
1 my $imie = <STDIN>;
2 say "Przed chomp: '$imie'";
3
4 chomp $imie;
5 say "Po chomp: '$imie'";
```

```
1 Przed chomp: 'jan
2 '
3 Po chomp: 'jan'
```

Listy

Do przetwarzania wielu rzeczy na raz przydadzą nam się listy. Listy przechowujemy z zmiennych z małpą (`@`), i mogą one zawierać dowolne skalary:

```
1 my @liczby = (1, 2, 3, 4);
2 my @zakupy = ('chleb', 'jajka', 'piwo');
3
4 # "spłaszczanie" (flattening) list
5 my @wszystko = (@liczby, @zakupy);
6 # 1, 2, 3, 4, 'chleb', 'jajka', 'piwo'
```

Listy – **qw**

Listy składające się ze słów możemy wygodnie tworzyć operatorem **qw** (*quote words*). Wszystkie białe znaki są wtedy uznane za separatory i ignorowane.

```
1 my @dni_tygodnia = (
2     'poniedziałek', 'wtorek', 'środa', ...
3 );
4 # albo prościej...
5 my @dni_tygodnia = qw(
6     poniedziałek wtorek środa ...
7 );
```

Operator **qw** możemy stosować z niemal dowolnymi ogranicznikami: **qw**`(...)`, **qw**`/.../`, **qw**`[...]`, **qw**`|...|` itp.

Indeksowanie

Listy indeksujemy za pomocą nawiasów kwadratowych. Nieco mylący może być fakt, że zapisujemy je wtedy z dolarem: to dlatego, że „wyciągamy” z niej skalar:

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2 say $zakupy[0]; # pierwszy element
3 $zakupy[1] = 'ser';
4
5 # push() dodaje element na koniec
6 push @zakupy, 'ketchup';
```

Rozbijanie list

Podobnie jak możemy przypisywac do list listę wartości możemy też przypisać listę do listy zmiennych (skalarów lub list) – trzeba tylko pamiętać o nawiasach wokół tych zmiennych. Lista jest wtedy rozkładana pomiędzy podane zmienne.

```
1 my @liczby = (1, 2, 3, 4, 5);
2
3 # 3, 4, 5 są ignorowane
4 my ($pierwszy, $drugi) = @liczby;
5
6 # 3, 4, 5 trafiają do @reszta
7 my ($pierwszy, $drugi, @reszta) = @liczby;
```

Długość listy

Długość listy możemy uzyskać przypisując listę do skalara w tym *kontekście*, lista zwraca wartość równą liczbie jej elementów.

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2 my $ilosc = @zakupy;
3 # bardziej jawnie:
4 # my $ilosc = scalar(@zakupy);
```

Dla każdej listy `@x` mamy też dostępną zmienną `$#x`, zawierającą ostatni indeks listy (czyli jej długość - 1).

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2 say $zakupy[$#zakupy]; # piwo
```

Referencje

Aby zawrzeć listę w liście możemy użyć *referencji* – odniesienia do innej zmiennej. Uzyskujemy ją przy pomocy `\`. Możemy również utworzyć referencję do listy za pomocą nawiasów kwadratowych:

```
1 my @wiersz1 = (1, 2, 3);
2 my $wiersz2 = [4, 5, 6]; # skalar!
3 my @macierz = (
4     \@wiersz1,
5     $wiersz2,
6 );
```

Indeksowanie referencji

Indeksowanie referencji do list jest nieco kłopotliwe: musimy albo pomóc sobie strzałką (->), albo *zdereferować* zmienną otaczając ją małpą:

```
1 my $zakupy = ['chleb', 'jajka', 'piwo'];
2 $zakupy->[1] = 'ser';
3 say @{$zakupy}[0];
4
5 # skrócona forma dereferencji
6 push @$zakupy, 'ketchup';
```

Referencje a kopie

Przypisanie do zmiennej referencji sprawia, że obie odnoszą się do tej samej listy: zmiana jednej modyfikuje drugą. Jeśli potrzebujemy kopii musimy przypisać do siebie faktyczne listy:

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2 my $lista  = \@zakupy;
3 $lista->[1] = 'ser';
4 say $zakupy[1]; # ser
```

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2 my @lista  = @zakupy;
3 $lista[1] = 'ser';
4 say $zakupy[1]; # jajka
```

Pętla **for**

Listy wygodnie jest przetwarzać za pomocą pętli **for** (albo **foreach** – robią dokładnie to samo):

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2
3 for my $produkt (@zakupy) {
4     say $produkt;
5 }
6
7 # chleb
8 # jajka
9 # piwo
```

Pętla **for** w stylu C

Pętli **for** możemy również używać w „klasycznym” stylu:

```
1 for (my $i = 10; $i > 0; $i--) {
2     say "$i...";
3 }
```

W wielu przypadkach łatwiej jest to zapisać operatorem zakresu:

```
1 # to samo co for powyżej
2 say "$_..." for reverse 0..9
```

Zmienna $_: „to”

Nie musimy tworzyć nowej zmiennej dla każdego elementu pętli – możemy skorzystać z automatycznej zmiennej $_:

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2
3 for (@zakupy) {
4     say $_; # "say it"
5 }
```

Wiele funkcji w Perlu domyślnie używa zmiennej $_ jeśli nie podamy im żadnej innej:

```
1 my @zakupy = ('chleb', 'jajka', 'piwo');
2
3 say for @zakupy;
4 # to samo co:
5 say $_ for @zakupy;
```

Operacje na listach

Oprócz znanego nam **push**(), Perl posiada wiele wbudowanych funkcji do operacji na listach:

- operator (**..**) tworzący listę liczb z zadanego zakresu
- **pop**() usuwa element z końca listy i zwraca jego wartość
- **shift**() i **unshift**()
  usuwają/dodają element na początku
- **sort**() i **reverse**()
  sortują/odwracają listy (nie w miejscu!)
- **splice**() pozwala zastąpić część jednej listy inną

```
1 my @liczby = (1..10);
2
3 # usuwa 5 liczb licząc od indeksu 3,
4 # i zastępuje je listą ('a', 'b', 'c')
5     splice(@liczby, 3, 5, qw(a b c));
6
7 # @liczby = (1, 2, 3, 'a', 'b', 'c', 9, 10)
```

Ćwiczenie: Lista TODO

Napisz program, który:

- Utworzy pustą listę zadań (**my** `@todo = ();`)
- Będzie wczytywał od użytkownika zadania i dodawał je do listy (użyj **push**)...
- ...dopóki użytkownik nie zostawi pustego wpisu (porównaj przez `eq`, zakończ pętlę **last**. Pamiętaj o **chomp**)
- Na koniec wypisz całą listę na ekran (**for** i `say`)

Udało się? Spróbuj dodać do programu menu:

```
1 Co chcesz zrobić?
2 1) Dodaj nowe zadanie
3 2) Wypisz wszystkie zadania
4 3) Zakończ
5 Twój wybor: ...
```

`split()` i `join()`

`split()` i `join()` to funkcje, które pozwalają nam łatwo przetwarzać tekst przy użyciu list.

```
1 my $tekst = "ala ma kota";
2 my @slowa = split ' ', $tekst;
3 $tekst = join "\n", @slowa;
4 say $tekst;
5 # ala
6 # ma
7 # kota
```

Do `split()` możemy również przekazać jako separator *wyrażenie regularne* – więcej o nich poźniej.

Listy są w Perlu wszechobecne i mamy wiele wygodnych sposobów pracy z nimi – w dalszej części kursu poznamy jeszcze wiele z nich.

Hashe

Hashe to struktury danych przechowywujące dane jako klucze i wartości.

Hashe tworzymy w zmiennych z procentem (`%`), a w nich często używamy operatora strzałki, która automatycznie zamienia swoją lewą stronę w string.

Hashe indeksujemy za pomocą nawiasów klamrowych (`{ }`).

```
1 my %priorytety = (
2     praca => 'wysoki',
3     'rozrywka', 'niski',
4 );
5 $priorytety{rozrywka} = 'najwyższy';
6 # apostrofy przy indeksowaniu są opcjonalne
7 say $priorytety{'rozrywka'};
```

Referencje do hashy

Hashe w swoich wartościach mogą przechowywać dowolne skalary – nawet referencje do innych hashy. Rereferencje do hashy tworzymy przy pomocy nawiasów klamrowych:

```
my $duzy = {
    liczby => [1, 2, 3],
    koszyk => { chleb => 1, sok => '2l' },
};
```

Przy wyciąganiu danych z takich zagnieżdżonych referencji, strzałka potrzebna będzie nam tylko przy pierwszym indeksowaniu:

```
say $duzy->{koszyk}->{chleb}; # zbędne
say $duzy->{koszyk}{chleb};   # prościej
```

Przetwarzanie hashy

Oprócz indeksowania możemy odwołać się do samych kluczy lub samych wartości hasha za pomocą funkcji **keys**() i **values**(). Obie te funkcje zwracają listy.

```
1 my %numery_uczestnikow = (
2     Agata  => '555 12 34',
3     Bogdan => '555 32 33',
4 );
5
6 my @telefony = values %numery_uczestnikow;
7
8 for (keys %numery_uczestnikow) {
9     say "Numer do $_ to " .
          $numery_uczestnikow{$_};
10 }
```

Operacje na hashach

Do hashy również mamy dostępne kilka użytecznych funkcji:

- **exists**() mówi nam, czy dany klucz istnieje w hashu
- **delete**() usuwa klucz z hasha (wraz z wartością)
- **each**(), wołane wielokrotnie, zwraca pary (klucz, wartość)

```
1 # usuwa wszystkie elementy z hasha,
2 # efektywnie to samo co %hash = ()
3 while (my ($key, $value) = each %hash) {
4     delete $hash{$key};
5 }
```

Ćwiczenie: hashe i referencje

Napisz program, który wczyta plik zawierający listę wpisów od różnych użytkowników i pokategoryzuje je w hashu. Z pliku:

```
admin login
beata obliczenia
beata raport
admin logout
```

Ma powstać hash list:

```
my %zdarzenia = (
    admin => ['login', 'logout'],
    beata => ['obliczenia', 'raport'],
);
```

Na początek spróbuj utworzyć hasha który zawiera tylko ostatnie wykryte zdarzenie dla danego użytkownika (bez list).

Inne typy danych w Perlu

Perl dostarcza nam również wiele innych typów danych – zazwyczaj będą jednak one schowane w zwykłych skalarach. Tak działa np. obsługa plików:

```
1 # Otwiera plik log.txt do zapisu
2 open(my $log, '>', 'log.txt');
3
4 say $log "coś się wydarzyło";
5
6 close $log;
```

Obiekty w Perlu

W przeciwieństwie np. do Pythona, nie każda wartość w Perlu jest obiektem – ale niektóre, np. IO::File nimi są:

```
1  use IO::File; # opcjonalny moduł
2
3  my $plik = IO::File->new('dane.txt');
4
5  while (<$plik>) {
6      chomp; # operuje na $_
7      say "Wczytano dane: $_";
8  }
9
10 $plik->close; # wywołanie metody na obiekcie
```

Funkcje

Funkcje to niejako małe programy, które otrzymują jakieś dane, produkują jakieś wyniki i nierzadko mają jakieś efekty uboczne. Niektóre już znamy: `say()`, **`open`**`()` czy **`rand`**`()`.

Funkcje w Perlu tworzymy słowem **`sub`** (od *subroutine*).

```
1 sub hello {
2     say "Hello world"
3 }
4
5 hello(); # nawiasy opcjonalne
```

Argumenty do funkcji

Argumenty (dane wejściowe) do funkcji w Perlu otrzymujemy – nieco archaicznie, i niezbyt wygodnie – w formie listy `@_`:

Podobnie jak w przypadku `$_`, funkcje operujące na listach (takie jak **`shift`**) domyślnie operują na `@_`.

```
1 sub hello {
2     my ($kto) = @_;
3     # albo: my $kto = shift;
4
5     say "Witaj, $kto";
6 }
7
8 hello "świecie";
```

Na szczęście od wersji 5.20 mamy dostępne również sygnatury funkcji :)

Sygnatury funkcji

Sygnatury funkcji (wciąż eksperymentalne!) pozwalają nam odczytywać argumenty funkcji w sposób znany z innych języków programowania.

```
1 use 5.020;
2 use experimental 'signatures';
3
4 sub hello($kto) {
5     say "Witaj $kto"
6 }
7
8 hello "świecie";
```

Pozwalają również uniknąć błędów wynikających z przekazania złej liczby argumentów do funkcji – w klasycznej metodzie ani `hello()` ani `hello(1, 2, 3)` nie byłoby błędem.

Wartości zwracane

Funkcje mogą zwracać wartości do programu za pomocą instrukcji **`return`**. W przypadku jej braku, wynikiem działania funkcji jest ostatnie wyrażenie wewnątrz funkcji.

Możemy wymusić brak zwracanych wartości przez **`return;`**.

```
1 sub suma($x, $y) { return $x + $y }
2
3 # albo prościej:
4 sub suma($x, $y) { $x + $y }
5
6 my $wynik = suma(4, 5);
```

Przykład: funkcja `prompt()`

Napiszmy funkcję `prompt()`, która wczyta dane od użytkownika wyświetlając zadany komunikat lub znak zachęty.

```
1  sub prompt {
2      if (@_) {
3          print @_, ' ';
4      } else {
5          print '> ';
6      }
7      my $input = <STDIN>;
8      chomp $input;
9      return $input;
10 }
11 my $dane = prompt;
12 my $wiek = prompt "Ile masz lat?"
```

`prompt()` z sygnaturą

Zobaczmy jak zmieni się prompt z użyciem sygnatur funkcji.

```
1  use 5.020;
2  use experimental 'signatures';
3
4  # parametr z wartością domyślną
5  sub prompt($komunikat = '>') {
6      print "$komunikat ";
7      my $input = <STDIN>;
8      chomp $input;
9      return $input;
10 }
11 my $dane = prompt;
12 my $wiek = prompt "Ile masz lat?"
```

Programowanie funkcyjne

Funkcje w Perlu możemy również zapisać w zmiennych, np. aby przekazać je do innej funkcji.

Taki sposób programowania, gdzie funkcje traktujemy jak dane w programie nazywamy *programowaniem funkcyjnym*.

Perl ma wiele mechanizmów które pozwalają nam wygodnie wykorzystać takie funkcje.

```
1 # zapisanie funkcji w zmiennej
2 my $funkcja = sub { say @_ };
3
4 # alternatywnie: referencja do funkcji
5 sub wypisz { say @_ }
6 my $funkcja = \&wypisz;
7
8 # wywołanie
9 $funkcja->("Hello world");
```

map() i **grep**()

Niektóre funkcje w Perlu przyjmują funkcje jako argumenty: częściej jednak zdarza się przekazywać kod do funkcji po prostu jako blok kodu.

Użytecznymi przykładami są **map**() i **grep**(), które pozwalają nam przekształcać i filtrować listy za pomocą zadanej funkcji/bloku:

```
1 my @liczby = (1, 2, 3, 40, 50);
2
3 # 40, 50
4 my @duze = grep { $_ > 10 } @liczby;
5
6 # 2, 4, 6, 80, 100
7 my @podwojone = map { $_ * 2 } @liczby;
```

Moduły

Funkcje często będą dostępne w zewnętrznych modułach, takich jak `List::Util` czy `Data::Dumper`. Moduły ładujemy za pomocą **`use`**, ale nie wszystkie funkcje będą dostępne automatycznie.

```
1 use List::Util;
2 use Data::Dumper;
3 my @lista = (1, 2, 3, 2);
4
5 # uniq() nie jest domyślnie eksportowane
6 my @unikalne = List::Util::uniq(@lista);
7
8 # do Dumper() przekazujemy referencje (lub
    skalary)
9 say Dumper \@unikalne;
```

Importowanie z modułów

Dla łatwiejszego dostępu do funkcji z modułów możemy je importować do naszego programu podając ich nazwy w **use**:

```
1 use List::Util qw(sum uniq);
2
3 my @lista = (1, 2, 3, 2);
4
5 say sum uniq (1, 2, 3, 2); # 6
```

Tworzenie własnych modułów

Moduły Perlowe tworzymy jako pliki `.pm`: samo umieszczenie w nich funkcji wystarcza do tego, by mieć do nich dostęp w innych programach – choć jeszcze nie będzie się dało ich importować.

```
1 # plik MyModule.pm
2 package MyModule;
3
4 sub prompt { ... }
5
6 1; # każdy moduł musi kończyć się "prawdą"
```

```
1 # szukaj modułu w bieżącym katalogu
2 use lib '.';
3 use MyModule;
4
5 my $input = MyModule::prompt('Podaj dane:');
```

Eksportowanie

Aby umożliwić importowanie funkcji z naszych modułów używamy modułu `Exporter`.

```
1 package MyModule;
2 use Exporter 'import';
3 our @EXPORT_OK = qw(prompt);
4 # albo samo @EXPORT żeby eksportować zawsze
5
6 sub prompt { ... }
```

```
1 use MyModule 'prompt';
2
3 my $input = prompt('Podaj dane:');
```

Ćwiczenie: funkcje

- Napisz funkcję która, otrzymawszy nazwę pliku i tekst zapisze ten tekst do pliku:
  `zapisz_plik('log.txt', 'Program działa')`
  Spróbuj napisać ją zarówno używając `@_` jak i sygnatur funkcji.
- Przenieś swoją funkcję do osobnego modułu (np. `Pliki.pm`) i uruchom ją z poziomu innego programu.
- Za pomocą (1..100) wygeneruj listę 100 liczb. Spróbuj za pomocą **`map`**`()` i **`grep`**`()` znaleźć wszystkie takie liczby z tego zakresu, które podniesione do drugiej potęgi są palindromami. Przyda się funkcja **`reverse`**`()` i operator potęgowania (`**`)

Obiekty w Perlu

Perl wspiera programowanie obiektowe: choć wbudowana składnia do niego jest nieco toporna.

Każdy obiekt w Perlu to po prostu „wzbogacony” skalar – posiadający jako swoje metody funkcje z wybranego modułu. Takim skalarem zwykle jest referencja do hasha – klucze i wartości tworzą wówczas atrybuty tego obiektu.

```
1 package Plik {
2     sub new($class, $nazwa) {
3         bless { nazwa => $nazwa }, $class;
4     }
5
6     ...
7 }
8
9 my $plik = Plik->new('obj.pl');
```

Obiekty w Perlu (cd.)

```
1  package Plik {
2      use IO::File;
3
4      sub new($class, $nazwa) { ... }
5
6      sub wczytaj($self) {
7          my $fh = IO::File->new($self->{nazwa});
8          my @linie;
9          while (<$fh>) {
10             chomp;
11             push @linie, $_;
12         }
13         return @linie;
14     }
15 }
16
17 my $plik = Plik->new('obj.pl');
18 say for $plik->wczytaj;
```

Obiekty z Moo

Jako że tworzenie obiektów ręcznie przy pomocy **bless** nie jest zbyt wygodne, powstało wiele modułów ułatwiających ten proces. Najpopularniejszym z nich jest `Moo`:

```
1  package Plik {
2      use IO::File; use Moo;
3
4      # atrybut tylko do odczytu
5      has nazwa => (is => 'ro');
6
7      sub wczytaj($self) {
8          my $fh = IO::File->new($self->nazwa);
9          ...
10     }
11 }
12
13 # new() wygenerowane automatycznie
14 my $plik = Plik->new(nazwa => 'obj.pl');
```

Obsługa błędów

Niektóre z błędów w trakcie działania programu, nieobsłużone, crashują nasz program. Jest to mechanizm *wyjątków*, znany z innych języków programowania.

```
1 sub podziel($x, $y) { $x / $y }
2 my $wynik = podziel(5, 0);
3 say $wynik;
4
5 # Illegal division by zero at test.pl line 4.
```

Błędy mogą być dowolnymi obiektami, jednak większość standardowych błędów to stringi.

Obsługa błędów – **eval**

Standardowy mechanizm obsługi błędów jest dość toporny: kod możemy umieścić w bloku **eval** { }, i jeśli wystąpił w nim wyjątek to zostanie on umieszczony w zmiennej $@:

```
1 sub podziel($x, $y) { $x / $y }
2
3 my $wynik = eval { podziel(5, 0) };
4
5 if ($@) {
6     say "Błąd: $@";
7 } else {
8     say $wynik;
9 }
```

Obsługa błędów – **die**

Jeśli sami chcemy zasygnalizować w kodzie jakiś błąd używamy do tego trafnie nazwanej funkcji **die**():

```
1 sub podziel($x, $y) {
2     if ($y == 0) {
3         die "Nie wolno dzielić przez zero"
4     }
5     $x / $y
6 }
```

Obsługa błędów – `Try::Tiny`

Aby uprościć sobie nieco obsługę błędów (i upodobnić ją do innych języków) możemy użyć popularnego modułu `Try::Tiny`:

```
1  use Try::Tiny;
2
3  sub podziel($x, $y) { $x / $y }
4
5  try {
6      say podziel(5, 0);
7  } catch {
8      # błąd tym razem w $_
9      say "Błąd: $_";
10 }; # uwaga: ważny średnik
```

Moduły dodatkowe

`Moo` i `Try::Tiny` które widzieliśmy przed chwilą nie są dostępne w bibliotece standardowej Perla – musimy je sobie doinstalować.

Moduły instalujemy z repozytorium CPAN – Comprehensive Perl Archive Network.

Moduły potrafią być prostymi bibliotekami (jak np. moduł `JSON`),
implementacjami obiektów (np. popularny `Path::Tiny`)
frameworkami (jak `Mojolicious` czy `Catalyst`)
albo nawet rozszerzeniami składni
(od `Moops` aż po `Lingua::Romana::Perligata`).

Sporo modułów powstaje też dla żartu, jak np.
`Acme::EyeDrops` potrafiący zamienić dowolny kod w Ascii-Art
(który nadal działa).

Instalowanie modułów

Moduły instalujemy narzędziem `cpan` (standardowo dostępnym) lub `cpanm` (prostszym i szybszym):

```
$ cpan i Moo
$ cpanm Try::Tiny
```

Większość praktycznego pisania kodu w Perlu to klejenie ze sobą gotowych modułów – mówi się czasem, że to CPAN jest tym językiem, w którym się programuje.

CPAN i MetaCPAN

CPAN to samo archiwum modułów. MetaCPAN (`https://metacpan.org`) to strona, na której możemy je odnaleźć, ocenić czy przejrzeć dokumentację.

Dokumentacje modułów Perlowych zawierają zwykle kilka przykładów na samym początku, więc szybko możemy ustalić czy moduł jest tym, czego szukamy.

Spójrzmy na dokumentację kilku popularnych modułów i zobaczmy, co możemy z nimi zrobić.

Ćwiczenie: wyjście na świat

Zainstaluj jakiś moduł z CPANa (np. Mojolicious albo BioPerl) i spróbuj uruchomić kilka przykładów kodu z dokumentacji. Upewnij się, czy rozumiesz kod, który piszesz :)

Zakończenie

Nikt nie uczy się programowania słuchając, jak ktoś inny mówi: trzeba do tego praktyki.

Aby nie zapomnieć za szybko nowej wiedzy (i poszerzyć ją) warto porobić ćwiczenia z `https://adventofcode.com` – każdy rok zaczyna się od bardzo prostych zadań, a kończy na zupełnie życiowych problemach.

Dobre i przystępne tutoriale (artykuły, książki i wideokursy) można znaleźć na `https://perlmaven.com`.