



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

***MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH***

Podstawy paradygmatów programowania

Autor:
dr hab. Tomasz Zientarski

Lublin 2021

INFORMACJA O PRZEDMIOCIE

Cele przedmiotu:

Cel 1. Zapoznanie z głównymi paradygmatami programowania: programowanie imperatywne, obiektowe, funkcyjne oraz logiczne. Zapoznanie z dodatkowymi paradygmatami współczesnego programowania.

Cel 2. Uporządkowanie podstawowej wiedzy z zakresu stosowania współczesnych języków programowania.

Cel 3. Poznanie charakterystycznych cech i konstrukcji w językach programowania na przykładzie Prologa, Haskellu, Pythona.

Cel 4. Zdobycie umiejętności do przenoszenia algorytmów pomiędzy różnymi językami programowania.

Efekty kształcenia w zakresie umiejętności:

Efekt 1. Potrafi zapisać proste algorytmy w Pythonie, Prologu, Haskellu oraz je uruchomić.

Efekt 2. Ma umiejętność samokształcenia się i potrafi określić kierunek dalszego kształcenia.

Efekt 3. Potrafi wybrać i zastosować w praktyce właściwy język programowania do prac programistycznych.

Literatura do zajęć:

Literatura podstawowa

1. Lipovaca M., Learn you a haskell for great good!: a beginner's guide, No Starch Press, 2011.
2. Hutton G., Programming in Haskell, Cambridge University Press 2007.
3. Bramer M., Logic programming with Prolog, Secaucus: Springer, 2005.
4. Wielemaker J., SWI-Prolog 7.6-0 Reference Manual,
<https://www.swi-prolog.org/download/stable/doc/SWI-Prolog-7.6.0.pdf> (10.01.2021)
5. Lutz M., Python Wprowadzenie, Wydanie IV, Helion, 2011.
6. GHC Haskell <http://www.haskell.org/haskellwiki/GHC> (13.01.2021)

Literatura uzupełniająca

1. Ullie Endriss, An Introduction to Prolog Programming,
<http://www.worldcolleges.info/sites/default/files/pro2.pdf> (10.01.2021)
2. W.F. Clocksin, Programming in Prolog,
http://math.uni.lodz.pl/~kowalczyk/2020L_PP/PrologBook.pdf (10.01.2021)
3. Haskell tutorials <http://www.haskell.org/haskellwiki/Tutorials> (13.01.2021)
4. λ= Try Haskell online <http://tryhaskell.org/> (13.01.2021)



Metody i kryteria oceny:

Oceny cząstkowe:

- Ocena 1 Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
- Ocena 2 Zaliczenie dwóch kolokwii z tematyki omawianej w czasie zajęć laboratoryjnych.

Ocena końcowa - zaliczenie przedmiotu:

- Pozytywne oceny cząstkowe.

Plan zajęć laboratoryjnych:

Lab1.	Programowania logiczne. Prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach. Prolog – wprowadzenie do programowania logicznego
Lab2.	Programowania logiczne. Prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach. Prolog – nawroty, odcięcia, unifikacja, rezolucja
Lab3.	Programowania logiczne. prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach. Prolog – „imperatywnie”
Lab4.	Programowania logiczne. prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach. Prolog – listy
Lab5.	Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda. Haskell – wprowadzenie do programowania funkcyjnego.
Lab6.	Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda. Haskell – podstawowe konstrukcje.
Lab7.	Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda. Haskell – rekurencja i leniwe wartościowanie.
Lab8.	Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka,

	funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda. Haskell – listy.
Lab9.	Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego. Python – wprowadzenie.
Lab10	Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego. Python – łańcuchy znaków oraz programowanie funkcyjne.
Lab11.	Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego. Python – programowanie obiektowe.
Lab12.	Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego. Python – adnotacje typów i klasy danych.
Lab13.	Zagadnienia przenoszenia algorytmów pomiędzy różnymi językami programowania



LABORATORIUM 1. PROLOG – WPROWADZENIE DO PROGRAMOWANIA LOGICZNEGO.

Programowania logiczne. Prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach.

Cel laboratorium:

Zdobycie umiejętności przygotowania środowiska pracy dla interpretera Prologa po kontrolą systemu operacyjnego Windows i Linux. Nabycie umiejętności pisanie prostych programów z wykorzystaniem faktów, reguł i zmiennych w Prologu oraz ich uruchamiania pod kontrolą systemu.

Zakres tematyczny zajęć:

- instalacja Windows Subsystem for Linux w Windows 10,
- przygotowanie środowiska pracy po kontrolą systemu operacyjnego Windows lub Linux,
- zapoznanie z podstawami języka Prolog,
- zapoznanie z faktami i regułami w Prologu,
- zmienne i zmienne anonimowe w Prologu.

Pytania kontrolne:

1. Co to jest WSL?
2. W jaki sposób uruchamiamy interpreter Prologa w środowisku Linux lub Windows?
3. Jakie zasady określają nazewnictwo termów w Prologu?
4. Jak definiuje się fakt oraz regułę?
5. Czym różni się zwykła zmienna od zmiennej anonimowej?

Zadanie 1.1. Przygotowanie środowiska pracy w systemie Windows w oparciu o WSL2 (ang. *Windows Subsystem for Linux*) oraz Linux (Kubuntu 20.04 LTS).

Windows Subsystem for Linux (WSL) pozwala użytkownikom Windowsa na uruchomienie środowiska Linuxa bez użycia wirtualnej maszyny (VirtualBox). Skutkuje to mniejszym zużyciem zasobów i lepszą integracją pomiędzy Windowsem a Linuxem.

Instalacja Prologa w przypadku WSL2 oraz Linuxa

Opis instalacji WSL2 znajduje się tutaj:

<https://docs.microsoft.com/en-us/windows/wsl/install-win10>

Opis instalacji

1. Uruchom Windows PowerShell jako Administrator
2. Włączenie WSL
`dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart`



3. Zalecany restart komputera
4. Włączenie wirtualnej Maszyny Microsoftu
`dism.exe /online /enable-feature
/featurename:VirtualMachinePlatform /all /norestart`
5. Zalecany restart komputera
6. W tym kroku powinniśmy zaktualizować jądro Linux
https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi
7. Ciągłe mamy WSL1, więc aktywujemy domyślną wersję na WSL2
`wsl --set-default-version 2`
8. W tym kroku wybieramy ulubioną wersję Linuxa ze sklepu <https://aka.ms/wslstore>
Jest to (ubuntu 20.04 LTS) <https://www.microsoft.com/store/apps/9n6svws3rx71>
9. Po instalacji zakładamy konto i nadajemy hasło
10. Ikona nowego systemu powinna pojawić się w Menu Start
11. Po uruchomieniu Linuxa obowiązkowo aktualizacja systemu, a potem możemy już zainstalować Prologa. Dla ubuntu wygląda to następująco:

```
sudo apt update  
sudo apt upgrade  
sudo apt install swi-prolog
```

Instalacja Prologa w Linuxie

W przypadku Kubuntu 20.04 wykonujemy jedno polecenie lub kilka w zależności czy jest dodane repozytorium Prologa w systemie. Jak sprawdzić? Najlepiej po prostu spróbować zainstalować pisząc:

```
sudo apt-get install swi-prolog
```

Jeżeli instalator pakietów nie znajdzie Prologa, to wtedy dodajemy repozytorium pisząc:

```
sudo apt-add-repository ppa:swi-prolog/stable  
sudo apt-get update
```

Następnie instalujemy Prologa pierwszym poleceniem.

Instalacja Prologa w środowisku Windows10

Aby zainstalować Prologa należy pobrać instalatora ze strony:

<https://www.swi-prolog.org/download/stable>

Zalecana wersja 64 bit:

<https://www.swi-prolog.org/download/stable/bin/swipl-8.2.3-1.x64.exe.envelope>

Zadanie 1.2. Prolog – podstawy: składnia, semantyka, fakty i reguły

Paradygmat logiczny. Opisujemy **cel**, który wykonawca (komputer) ma osiągnąć, w tym rodzaju programowaniu programista opisuje, **co** komputer ma osiągnąć.



W Prologu zamiast opisywać algorytm wykonania pewnego zadania, opisujemy obiekty (ale nie w sensie programowania obiektowego) związane z problemem i relacje pomiędzy nimi za pomocą zestawów faktów i reguł. Prolog potrafi wywnioskować odpowiedź na podane pytanie dotyczące takiego opisu świata na podstawie podanych informacji. Działanie programu prologowego objawia się możliwością stawiania pytań związanych z uprzednio opisanym światem.

Podstawowymi elementami bazy wiedzy są termy. Wyróżniamy: **atomy**, **liczby**, **zmienne** i **termy złożone**. Każdy term zapisywany jest jako ciąg znaków pochodzących z:

- duże litery: A-Z,
- małe litery: a-z,
- cyfry: 0-9,
- znaki specjalne: % + - * / \ ~ ^ < > : . ? @ # \$ &
- brak polskich liter w kodzie

Ponadto:

- * / + -
- , (przecinek) oznacza operator and
- ; (średnik) oznacza operator or
- \+ negacja
- > < >= <=
- \= nie równe
- !s, =, :=
- % oznacza komentarz jednolinijkowy
- /* to komentarz wielolinijkowy */

Termy to podstawowe elementy bazy wiedzy w języku Prolog oraz zapytań. Dzielą się na:

- atomy (obiekty opisujące świat), nazwy własne ogólnego przeznaczenia, nazywane przez programistę. Atom jest ciągiem znaków utworzonym z małych i dużych liter, cyfr i znaku podkreślenia z zastrzeżeniem, że pierwszym znakiem musi być mała litera, np. `jas`, `a`, `aLA`, `x_y_z`, `abc`. Ponadto może składać się z dowolnego ciągu znaków ujętego w apostrofy, np. `'To też jest atom'` a także z symboli, np. `?- lub :-`,
- liczby (całkowite i rzeczywiste),
- zmienne (termy do których zostaną podstawione atomy w trakcie odpowiedzi na pytania), zawsze nazwane z **wielkiej** litery,
- termy złożone: atomy z argumentami, np. `kot(X)`, `wlasciciel_psa(ja,azor)`, `posiada(marcin,auto(fiat,punto))`.
- listy: kolekcje termów w nawiasach kwadratowych,
- łańcuchy: listy liczb lub jednoelementowych atomów,



Na końcu każdej linii w Prologu (zarówno definicji reguły, faktu, jak i w zapytaniu) jest **kropka!**

Na program w Prologu składają się fakty i reguły, które tworzą klauzule inaczej bazę wiedzy, świat. Fakty i reguły oparte są o funkcje predykatu, tj. funkcje, które na podstawie swoich argumentów zwracają tylko wartość logiczną prawda lub fałsz.

Przykładowy program składający się faktu i reguły i ma następującą postać:

```
ciezszy(pomarancza, jablko).  
ciezszy(X,Y) :- ciezszy(X,Z),ciezszy(Z,Y).  
%głowa    : ciało reguły  
%lewa część zachodzi gdy prawa część jest prawdziwa
```

gdzie

`ciezszy(pomarancza, jablko).` oznacza fakt (prawda lub fałsz),

zaś

`ciezszy(X,Y) :- ciezszy(X,Z),ciezszy(Z,Y).` oznacza regułę

Głowa reguły jest predykatem, zaś ciało reguły to nic innego jak predykaty, które mogą być połączone operatorami logicznymi `,` (logiczne AND) lub `;` (logiczne OR).

Reguły w Prologu nie są równoważne funkcjom, nie zwracają wyniku, więc nie można ich wyniku wykorzystać od razu do innych celów. Wykorzystywane są za to do odpowiedzi na zapytania.

```
?- ciezszy(pomarancza, jablko).  
true .
```

```
?- ciezszy(pomarancza, X).  
X = jablko ;  
X = mandarynka .
```

Zmienna anonimowa

Czasami istotne jest tylko, aby była jakaś zmienna, natomiast jej konkretna wartość jest nieistotna. Wykorzystujemy wtedy zmienną anonimową, oznaczaną znakiem `_`. Przykładowo:

```
ciezszy(_, winogrono). % wszystko jest cięższe od winogrona
```

Termy złożone

Mogą posłużyć do opisu zestawu cech obiektu. Na przykład:




```
posiada(piotr, auto(fiat, punto)). % Piotr posiada auto o cechach  
Fiat oraz Punto  
posiada(adam, dom). % Adam posiada dom
```

Teraz reguła o takiej postaci:

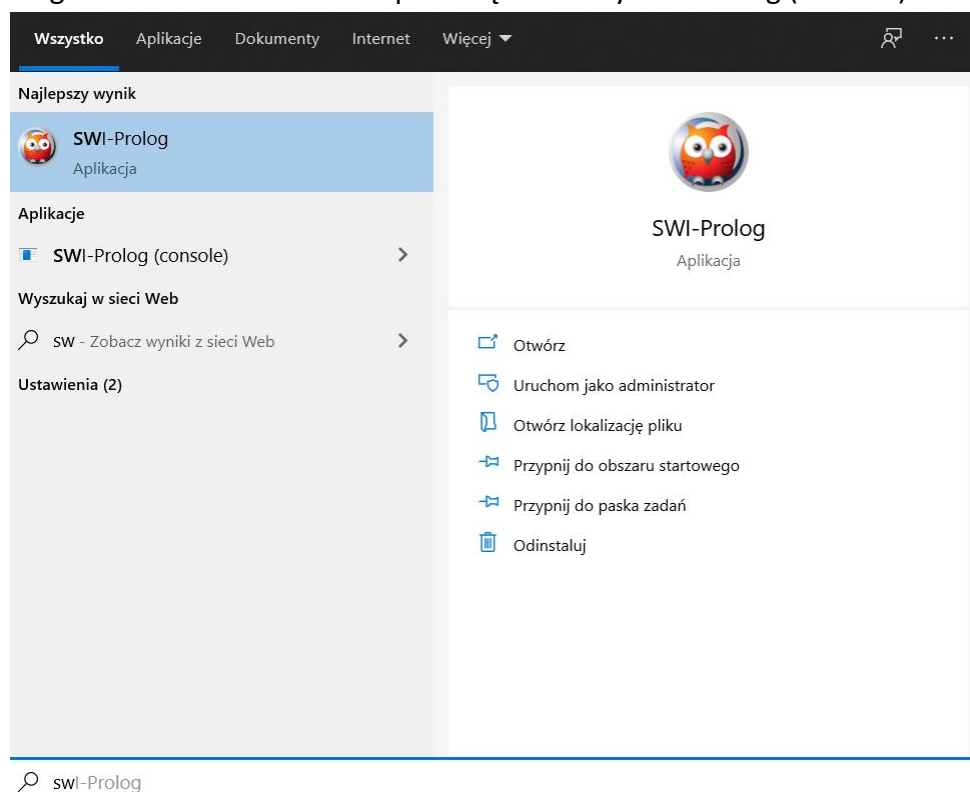
```
czy_ma_auto(X) :- posiada(X, _).
```

Jest błędna, ponieważ będzie działać dla wszystkiego – także dla domu. Ale możemy wykonać dekonstrukcję termu złożonego:

```
czy_ma_auto(X) :- posiada(X, auto(_, _)).
```

Zadanie 1.3. Prolog – praca z interpreterem

Instalacja środowiska była opisana poprzednio, więc teraz możemy uruchomić interpreter Prologa. W systemie Windows będziemy używać narzędzia SWI-Prolog, można je uruchamiać graficznie lub w konsoli za pomocą komendy SWI-Prolog (console).



Rys. 1.1. Wygląd MenuStart Windows po wpisaniu z klawiatury słowa *swi*.

Po uruchomieniu powinien ukazać się nam następujący widok.



Rys. 1.2. Widok uruchomionego programu.

W środowisku Linuxa po uruchomieniu konsoli piszemy po prostu `swipl` i otrzymujemy:



Rys. 1.3. Widok uruchomionego programu w konsoli Ubuntu

Napisane programy zapisane w pliku tekstowym z rozszerzeniem `.pl` interpretujemy z linii komend pisząc:

```
swipl mój plik.pl
```

Podczas trwającej sesji Prologa możliwe jest również załadowanie pliku poprzez wydanie komendy w nawiasach kwadratowych:

```
?- [nazwa_pliku_bez_rozszerzenia].
```

Która załaduje plik o podanej nazwie (z rozszerzeniem `.pl`) z aktualnego katalogu roboczego.

Zadanie 1.4. Prolog – pierwszy program

Chcemy napisać program określający czy coś jest cięższe od czegoś innego. Zanim zaczniemy opisywać świat musimy się zastanowić co to znaczy: **cięższy**.

Prolog nie wie co to cięższy czy lżejszy. My ludzie uczymy się tego w trakcie życia, jest to nasze



doświadczenie. Patrząc na przykłady z życia małe dziecko nie patrząc na worek pierza powie, że jest to cięższe od opakowania cukru i odwrotnie powie że maleńka z wyglądu fiołka srebrnego płynu będzie lżejsza od tego samego opakowania cukru nie wiedząc że to rtęć. Idąc dalej nie będzie mogło uszeregować tych trzech przedmiotów w kolejności rosnącej wagi. Dziecko musi poznać te przedmioty.

Tak samo w Prologu.

Weźmy pod uwagę najprostszy świat składający się z czterech faktów

```
ciezszy(pomaranecz,jablko). % pomarańcz jest cięższa od jabłka
ciezszy(jablko,mandarynka).
ciezszy(arbuz,pomaranecz).
ciezszy(jablko,winogrono).
```

Powyższe fakty zapisujemy do pliku tekstowego (owoce1.pl) z dowolnego edytora

Następnie wczytujemy plik do Prologa. Z powyższego możemy wywnioskować, że:

```
arbuz > pomaranecz > jablko > winogrono
jablko > mandarynka
```

Zapytajmy Prologa.

```
?- [owoce1].
true.
?- ciezszy(pomaranecz,jablko).
true.
```

Zadajmy kolejne pytanie.

Czy pomarańcz jest cięższa od jabłka?

W świecie Prologa lepiej zadać pytanie o brzmieniu:

Czy wiadomo coś na temat tego, że pomarańcz jest cięższa od jabłka?

Jest to istotna różnica. Dlaczego?

```
?- ciezszy(winogrono,arbuz).
false.
```

Nic nie wiadomo na temat tego, że winogrono jest cięższe od arbuza.

Nie oznacza to jednak, że tak nie jest.

```
?- ciezszy(arbuz,winogrono).  
false.
```

Zadajmy pytanie, co jest cięższe od czegoś? ENTER kończy wyszukiwanie, średnik pozwala na dalsze odpowiedzi

```
?- ciezszy(X,Y).  
X = pomarancz,  
Y = jablko ;  
X = jablko,  
Y = mandarynka ;  
X = arbuz,  
Y = pomarancz ;  
X = jablko,  
Y = winogrono.
```

na razie nie ma problemów z wariantami odpowiedzi, ale czy na pewno?

```
?-ciezszy(arbuz,winogrono).  
false.
```

Czy powyższa odpowiedź Prologa jest prawdziwa?

Nie.

Czy arbuz jest cięższy od winogrona?

Dlatego, bo prolog nie zna relacji. Dlatego też jak wspomniano powinniśmy zapytać:

Czy wiadomo coś na temat tego, że arbuz jest cięższy od winogrona?.

Wtedy odpowiedź `false` oznacza **Nie wiadomo mi nic na ten temat**.

Co już ma większy sens. Ale dlaczego tak jest? Wróćmy do relacji.

```
arbuz > pomarancz > jablko > winogrono  
jablko > mandarynka
```

Musimy zdefiniować co to jest `cięższe`, dlatego należy uzupełnić świat o dodatkowy wpis i zapisać go pod nazwą `owoce2.pl`

```
ciezszy(pomarancz,jablko).  
ciezszy(jablko,mandarynka).  
ciezszy(arbuz,pomarancz).  
ciezszy(jablko,winogrono).  
ciezszy(X,Y) :- ciezszy(X,Z), ciezszy(Z,Y).
```

Odświeżamy bazę wiedzy o świecie.

```
?- [owoce2].  
true.
```

Ponówmy pytanie: **Czy wiadomo coś na temat tego, że arbuz jest cięższy od winogrona?**.

```
?- ciezszy(arbuz,winogrono)  
true.
```

Tym razem otrzymaliśmy odpowiedź zgodną z oczekiwaniem.

Proszę samodzielnie wykonać następujące zadania

Polecenie 1.

Zadaj pytanie: czy wiadomo coś na temat tego, że pomarańcza jest cięższa od mandarynki?.

Polecenie 2.

Możemy dowiedzieć się znacznie więcej, zadając pytanie od jakich obiektów jest cięższy arbuz. Otrzymamy wynik:

```
X = pomarancz ;  
X = jablko ;  
X = mandarynka ;  
X = winogrono ;
```

ENTER kończy wyszukiwanie, średnik umożliwia dalsze wyszukiwanie.

Komunikat pojawiający się na końcu (Out of local stack) należy w tym przypadku odczytać jako: **nie wiadomo nic o innych możliwościach (obiektach)** i zazwyczaj jest spowodowany niepoprawnie zdefiniowanym światem.

Polecenie 3.

Zadaj pytanie: czy istnieją takie owoce X, Y, dla których arbuz jest cięższy od X i jednocześnie X jest cięższy od Y. Otrzymamy wynik:

```
% arbuz > pomarancz > jablko > winogrono, mandarynka  
  
X = pomarancz,  
Y = jablko ;  
X = pomarancz,  
Y = mandarynka ;  
X = pomarancz,  
Y = winogrono ;
```

Polecenie 4.

Zadaj pytanie: Czy istnieją owoce X, Y cięższe od siebie

```
arbuz > pomarancz > jablko > winogrono, mandarynka
```

Polecenie 5.

A co z określeniem relacji `lzejszy`, czy możemy zapytać. Samodzielnie zdefiniuj regułę `lzejszy(X,Y)`

```
lzejszy(X,Y):-
```

Polecenie 6.

Mamy zdefiniowany świat, zapisz go w pliku o nazwie `piernik.pl`

```
lubi(jas,piernik).  
lubi(jas,malgosia).  
lubi(malgosia,cukierek).  
lubi(malgosia,piernik).
```

Zadaj pytanie: czy prawda jest, że Jaś lubi Małgosię i Małgosia lubi Jasia?

```
false.
```

znaczy nie **NIE** ale **Nie wiem**



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Polecenie 7.

Zadaj pytanie: pokaż wszystko co lubi zarówno Jaś jak i Małgosia.

```
X=piernik.  
false.
```

Polecenie 8.

Zadaj pytanie: co lubi Jaś lub Małgosia?

```
X = piernik ;  
X = malgosia ;  
X = cukierek ;  
X = piernik.
```

Zmienna anonimowa (_).

Zadaj pytanie: czy coś lubi Jaś?

```
?-true;  
?-true.
```

Polecenie 9.

Zadaj pytanie: czy ktoś lubi piernik?

```
?-true;  
?-true.
```

Polecenie 10.

Kolejny świat.

```
posiada(piotr,auto).  
posiada(marcin,auto).
```

Trudno powiedzieć jakie to jest auto i czy przypadkiem to nie jest to samo auto.

Zapisując te fakty inaczej, lepiej opiszemy problem z użyciem termów złożonych.

```
posiada(piotr,auto(nissan,almera)).  
posiada(marcin,auto(fiat,punto)).  
maAuto(X) :- posiada(X,auto(_,_)).
```

Zadaj pytanie: czy Piotr ma auto?

```
true.
```

Polecenie 11.

Zadaj pytanie: jaki samochód posiada Piotr a jaki Marcin?

```
X = nissan,  
Y = almera
```

```
G = fiat.  
F=punto
```


LABORATORIUM 2. PROLOG – NAWROTY, ODCIĘCIE, UNIFIKACJA, REZOLUCJA.

Programowania logiczne. Prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcie, rekursja, operacje na listach.

Cel laboratorium:

Zdobycie umiejętności wykorzystania nawrotów, odcięć oraz mechanizmów przekonstruowywania wyrażeń w programach. Zrozumienie mechanizmu unifikacji w zapytaniach elementarnych i nieelementarnych.

Zakres tematyczny zajęć:

- zapoznanie z mechanizmem nawrotów,
- wykorzystanie odcięć w programach,
- zapoznanie z rezolucją i unifikacją,
- zapoznanie studenta z zapytaniami elementarnymi i nieelementarnymi.

Pytania kontrolne:

1. Do czego służą nawroty w Prologu?
2. Do czego wykorzystuje się odcięcie w Prologu?
3. Na czym polega unifikacja w Prologu?
4. Na czym polega rezolucja w Prologu?
5. Kiedy dwa termy są identyczne?
6. Co to są zapytania elementarne i nieelementarne?
7. Jaki mechanizm w Prologu umożliwia znajdowanie wszystkich odpowiedzi?
8. Do czego służą i jak jest różnica w działaniu: `is`, `==` oraz `=` ?

Zadanie 2.1. Prolog – nawroty, odcięcie, rezolucja, unifikacja

Wielokrotnie w Prologu zdarza się, że cel może zostać spełniony na wiele alternatywnych sposobów. Za każdym razem, gdy zachodzi konieczność wybrania jednej z wielu możliwości, Prolog wybiera pierwszą z nich (w kolejności występowania w pliku) zapamiętując przy okazji miejsce, w którym wybór ten został dokonany. Jeśli w jakimś momencie nie powiedzie się próba obliczenia celu, system ma możliwość powrotu do miejsca ostatnio dokonanego wyboru i zastąpienia go wyborem alternatywnym. Umożliwia to znalezienie choć jednego rozwiązania. Ten sam mechanizm, zwany **nawrotami**, działa także, gdy powiedzie się obliczanie celu. Pozwala znaleźć rozwiązania alternatywne. Jeśli Prolog natrafi na **operator odcięcie (!)** w regule, nie będzie nawracał z wyborem do wcześniejszych możliwości.

Proces dopasowywania nazywamy **unifikacją**. Na przykład **Term(a,b)=Term(a,X)** wtedy gdy **X = b**. Dla dwóch atomów poszukiwane jest takie podstawienie, że staną się identyczne.



Proces zastępowania wyrażenia przez inne wyrażenie nazywamy **rezolucją**. Przykładowo w formule **a(X) :- b(X)** zapytanie **a(X)** zastępowane jest zapytaniem **b(X)**.

Zapytania **elementarne** to takie gdzie nie ma zmiennych, fakt np. mezczyzna(tomek).

Zapytania **nienielementarne** to takie zapytania ze zmiennymi np. rodzic(ola,X). Odpowiedzią oczekiwaną na takie zapytanie jest znalezienie właściwego podstawienia dla zmiennych.

Nawroty, rezolucja, unifikacja

```
lubi(jan, tatry).
lubi(jan, beskidy).
lubi(jerzy, beskidy).
lubi(jerzy, bieszczady).
lubi(jozef, bieszczady).
lubi(karol, beskidy).
lubi(justyna, swietokrzyskie).
bratniadusza(X, Y) :- lubi(X, S), lubi(Y, S), X \= Y.
```

?- bratniadusza(jan,X). nawroty do ostatnich decyzji

1 1 podcel lubi(jan,S) ---> **tatry**

 2 podcel **lubi(Y,tatry)** ---> jan

 3 podcel nie da się udowodnić, nawrót do 2podcelu (brak rozwiązania), nawrót do 1 podcelu

2 1 podcel lubi(jan,S) ---> **beskidy**

 2 podcel **lubi(Y,beskidy)** ---> jan

 3 podcel nie da się udowodnić, nawrót do 2podcelu i ten wystarcza

lubi(Y,beskidy) ---> jerzy, potem znowu 3podcel, który da się udowodnić **jerzy**

 nawrót do 2 podcelu **lubi(Y, beskidy)** i tak dalej do konca podcelu 2 (**karol**)

3 1 podcel lubi(jan,S) nie da się spełnić i koniec

jerzy, karol

Przepisz podany wyżej program dotyczący bratniej duszy i zapisz go pod nazwą **gory1.pl**

Następnie sprawdź wynik poniższego polecenia

```
?- bratniadusza(jan,X).
```



Odcięcia

```
a(c).  
a(X) :- b(X).  
b(d).  
b(e).  
a(f).  
a(g).  
b(h).  
a(i).
```

Jaki będzie wynik dla zapytania `a(X)`.

`X = c ; X = d ; X = e ; X = h ; X = f ; X = g ; X = i.`

```
a(X,Y) :- b(X), c(Y).  
b(d).  
b(e).  
b(f).  
c(g).  
c(h).  
c(i).
```

Jaki będzie wynik dla `a(X,Y)`.

Zastosujmy teraz operator odcięcia: `!`.

```
a(X, Y) :- b(X), !, c(Y).  
b(d).  
b(e).  
b(f).  
c(g).  
c(h).  
c(i).
```

Jaki będzie teraz wynik dla `a(X,Y)`.

Operator odcięcia powoduje, że jedyne rozwiązanie będzie dla `X=d`. Prolog nie będzie próbował ponownie ustalać celów po lewej stronie, a tylko po prawej stronie.

Dużą różnicą między Prologiem, a innymi językami programowania jest to, że Prolog wylicza wyrażenia arytmetyczne nie zawsze, ale tylko w konkretnych miejscach i przypadkach. Wbudowany predykat `is` bierze wyrażenie arytmetyczne z jego prawej strony, oblicza je i unifikuje je z wyrażeniem po jego lewej stronie. Inaczej zachowuje się predykat `=` – wylicza dwa wyrażenia i porównuje wyniki, zaś `=` unifikuje dwa wyrażenia, bez ich wyliczania.

Porównaj.

is

```
A is 2+3.  
A = 5
```

ale

```
?- 3 is 2+1.  
true  
?- 3+1 is 2+2.  
false
```

W przypadku `==`

```
?- 4+1 == 2+3.  
true.  
?- 4+1 == 2+2.  
false.
```

ale

```
?- A == 2+3.  
error
```

W przypadku `=`

```
?- A = 2+3.  
A = 2+3.
```

oraz

```
?- A = kotek.  
A = kotek.
```

a także

```
?- A = A.  
true.
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Przeanalizuj wynik w sposób tak jak w analogicznym, poprzednim przykładzie (bez uruchamiania kodu).

```
?- bratniadusza(jerzy,X).
```

Polecenie 2.

Dane są czasy i przebyte dystansy.

```
czas(t1, 120).  
czas(t2, 90) .  
czas(t3, 135).  
dystans(t1, 9600).  
dystans(t2, 8100).  
dystans(t3, 13500).  
predkosc(X, Ac) :- czas(X, C),dystans(X, D),Ac is D/C.
```

Na podstawie zadania powyżej, bez użycia interpretera, oszacuj co otrzymamy po wykonaniu:

```
?- predkosc(t1, X).  
?- predkosc(t2, 65).  
?- predkosc(X, 100).
```

Sprawdź wynik w interpreterze.

Jakie wyniki uzyskamy zamieniając `is` na `=` w poniższym przykładzie i wykonując zapytania: `predkosc1(t1, X)`, `predkosc1(t2, 65)` oraz `predkosc1(X, 100)`. Porównaj wyniki i odpowiedz dlaczego tak się stało.

```
czas(t1, 120).  
czas(t2, 90) .  
czas(t3, 135).  
dystans(t1, 9600).  
dystans(t2, 8100).  
dystans(t3, 13500).  
predkosc1(X, Ac) :- czas(X, C),dystans(X, D),Ac = D/C.
```

LABORATORIUM 3. PROLOG – „IMPERATYWNI”.

Programowania logiczne. Prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach.

Cel laboratorium:

Nabycie umiejętności wczytywania danych ze standardowego wejścia oraz wypisywania danych na standardowym wyjściu. Umiejętność definiowania konstrukcji warunkowych w postaci jednolinijkowego jak i wielolinijkowego kodu. Zdobycie umiejętności wykorzystania rekurencji w kodzie oraz optymalizacji czasu jej wykonania. Nabycie umiejętności tworzenia i wykorzystania liczników i akumulatorów.

Zakres tematyczny zajęć:

- wykorzystanie predykatu `is` w programach,
- wykorzystanie predykatu `read` oraz `write`,
- mechanizmy przekazywania wyników obliczeń poprzez reguły,
- operatory logiczne w Prologu,
- instrukcje warunkowe w Prologu,
- rekurencja w Prologu, liczniki i akumulatory.

Pytania kontrolne:

1. Do czego służy predykat `read`?
2. Do czego służy predykat `write`?
3. Na czym polega metoda obliczeniowa zwana „z góry na dół” oraz „z dołu do góry”?
4. Do czego są używane w Prologu akumulatory?

Zadanie 3.1. „Imperatywność” w Prologu

Do obliczeń w Prologu możemy użyć predykatu `is` a także predykatów `read`, `write`. Należy pamiętać, że reguła musi mieć możliwość na zwrócenie wyniku za pomocą zmiennej, która przechowuje wynik, odpowiedź na zapytanie. Reguły w Prologu nie są równoważne funkcjom, nie zwracają wyniku, więc nie można ich wyniku wykorzystać od razu do innych celów.

Najprościej można wykonywać obliczenia w samym interpreterze Prologa

```
?- Z is 5+18.  
?- Z is 2*2.5 +5.  
?- Z is 2*5/5 *5.
```

Jednak większe możliwości daje zapisanie programów w pliku i późniejsza interpretacja.



```
suma(X,Y,Z):- Z is X*Y.  
?- suma(2,6,Wyn).
```

```
suma1(Z,X):- Z is X*X.  
?-suma1(W,5).
```

Użyjmy wbudowanego predykatu `read()`, oraz `write()` pamiętajmy o tym że każdą linię kończymy kropką, uwaga na `|`:

```
suma2(Z):- read(X),Z is X+6.  
?-suma2(Wyn).
```

```
suma3(X,Y):- Z is X+Y, write(Z).  
?- suma3(2,3).
```

```
suma4:- read(X),read(Y), Z is sqrt(X+Y), write(Z).  
?- suma4.
```

Teraz coś bardziej skomplikowanego

```
ppr(X,_,_):- X<0,write('bok<0').  
ppr(_,Y,_):- Y<0,write('bok<0').  
ppr(X,Y,W):- X>0,Y>0,W is X*Y.  
  
?- ppr(2,4,W).  
W = 8.
```

Jeśli mamy regułę wykorzystującą operację logiczną `and`, to część reguły po `and` nie jest wykonywana, jeżeli operacja `and` zwróciła `false`. Pozwala to na traktowanie operacji `and` jako równoważnej operacji `if...then`.

Jednak, aby stworzyć odpowiednik `if...then...else` niezbędne jest stworzenie kolejnej reguły, dla alternatywnego przypadku. Na przykład:

```
jesli(X, Y) :- X=Y, ... % jeśli X = Y to wtedy coś  
jesli(X, Y) :- X \= Y, ... % jeśli X nie jest równe Y, to wtedy  
coś innego
```

Co można też zapisać w wersji skróconej:



```
jesli(X, Y) :- X=Y, ... % jeśli X = Y to wtedy coś  
jesli(X, Y) :- ... % jeśli X nie jest równe Y, to wtedy coś innego
```

Ponieważ jeżeli nie uda się dopasowanie do pierwszej reguły, Prolog spróbuje szukać alternatyw w kolejnych liniach kodu.

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Oblicz sumę dwóch liczb, jeśli pierwsza z nich jest mniejsza lub równa od 6.

```
?- suma5(20,3,W).  
false.  
?- suma5(3,20,W).  
W = 23.
```

Polecenie 2.

Ponownie napisz ten sam program ale użyj składni dwulinijkowej. Sprawdź działania dla X=5, X=6, X=7. Samodzielnie popraw niepoprawnie działający program.

```
?- suma6(20,3,W).  
false.  
?- suma5(5,10,W).  
W = 15.  
?- suma5(6,10,W).  
W = 16;  
false.
```

Wersja poprawna daje w wyniku:

```
?- suma6(20,3,W).  
false.  
?- suma5(5,10,W).  
W = 15.  
?- suma5(6,10,W).  
W = 16.
```



Polecenie 3.

Oblicz sumę dwóch liczb, jeśli pierwsza z nich jest mniejsza od 6, w przeciwnym przypadku wynik powinien mieć taką samą wartość jak druga liczba. Prawidłowe wyniki:

?- suma8(4,10,W).

W = 14.

?- suma8(6,10,W).

W = 10.

?- suma8(7,10,W).

W = 10.

Zadanie 3.2. Rekurencja w Prologu.

Skoro możliwe jest wykorzystanie kolejnych definicji reguł jako kolejnych alternatyw, to czy możliwe jest, aby reguła wykorzystywała samą siebie? Oczywiście, przypadkiem okazało się to w przykładzie z owocami.

Napiszmy reguły pozwalające na wymnożenie dwóch liczb przez siebie, ale z wykorzystaniem rekurencyjnego wzoru na mnożenie:

$$x \cdot y = y + (x - 1) \cdot y$$

Do czego sprowadza się wymnożenie $4 \cdot 3$

$$3 \cdot 4 = 3 + 3 \cdot 3 = 3 + 3 + 2 \cdot 3 = 3 + 3 + 3 + 1 \cdot 3 = 3 + 3 + 3 + 3$$

mul(4,3,R1)

mul(4,3,X)

|
3 + mul(3,3,X)

|
3 + mul(2,3,X)

|
3 + mul(1,3,X)

Przykładowy kod obliczający iloczyn dwóch liczb.



```
mul(0,_,0). % warunki graniczne  
mul(1,X,X). % warunki graniczne  
mul(X,Y,R) :- X>1,X1 is X-1, mul(X1,Y,R1), R is R1 + Y.  
?- mul(3,40,W).  
W = 120 ;  
false.
```

Jak usunąć niedogodność z brakiem odcięcia

```
?- mul1(3,40,W).  
W = 120.
```

Co otrzymamy w wyniku poniższych wywołań.

```
mul(4,3,12).  
mul(1,3,X). %problem? Jak go rozwiązać?
```

Metoda obliczeniowa „z góry na dół” (ang. *top down computation*), typowa dla Prologu, rozpoczyna od problemu wyjściowego a następnie rozkłada go na podproblemy prostsze a te z kolei na jeszcze prostsze itd., aż dojdzie do przykładu trywialnego.

Metoda obliczeniowa „z dołu do góry” (ang. *bottom up computation*) rozpoczyna od znanych faktów a następnie rozszerza je w oparciu o posiadane reguły i fakty tak długo aż nie zostanie rozwiązany problem wyjściowy.

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Samodzielnie napisz program obliczający silnie.



```
?- silnia2(5,D).
D = 120.

?- silnia(5,120).
true.
```

Sprawdźmy, jak działa program, zamieniając `is` na `=` w końcowej instrukcji w poprzednim przykładzie. Dla przypomnienia: `is` oblicza i unifikuje, `=` unifikuje dwa wyrażenia, bez ich wyliczania.

Polecenie 2.

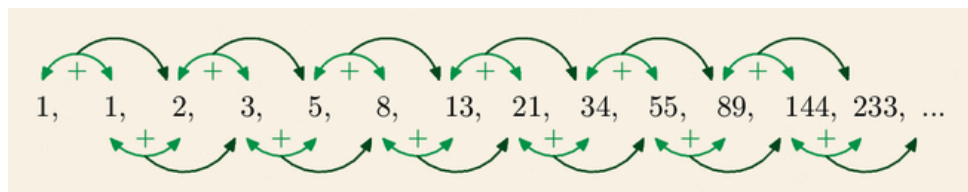
Samodzielnie napisz program obliczający silnie (użyj licznika rosnącego lub malejącego zależnie od poprzedniego rozwiązania).

```
silnia4(5,D).
```

Polecenie 3.

Napisz program w prologu obliczający wybrany element ciągu Fibonacciego (FB). Ogólny wzór oraz sposób obliczania przedstawiono na poniższych rysunkach (rysunki zaczerpnięte z : <http://matematykadlastudenta.pl/strona/762.html>).

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_{n+2} = F_{n+1} + F_n \end{cases}$$



```
?- fib(11,W).  
W = 89.
```

lub

```
?- fib(10,W).  
W = 89.
```

Zastosowana metoda obliczeniowa „z góry na dół” (ang. *top down computation*) jest dość nieefektywna, dużo większą w tym przypadku wydajnością cechuje się metoda „z dołu do góry” (ang. *bottom up computation*). Poniżej propozycja tego algorytmu. Proszę sprawdzić czas obliczeń dla 35 elementu ciągu.

```
fibBU(N,X) :- fibBU(0,0,1,N,X).  
fibBU(N,X,_,N,X):-!.  
fibBU(N1,X1,X2,N,X) :- N2 is N1+1, X3 is X1+X2,  
fibBU(N2,X2,X3,N,X).  
  
?- fibBU(36,W).  
W = 14930352.  
  
?- fib(35,W).  
W = 14930352.
```

Z ciągiem Fibonacciego wiąże się pojęcie tak zwanej złotej liczby, złotego podziału. Przykład obliczenia tej liczby za pomocą obu algorytmów FB. Jest to nic innego jak iloraz sąsiednich wyrazów ciągu FB. Wartość tej liczby dąży do 1.6180339887 . . .

Spróbuj samodzielnie napisać właściwy kod wykorzystujący poprzednio zdefiniowaną regułę `fibb` lub `fibBU`.

```
?- goldBU(40,E).  
E = 1.6180339887498947.
```



LABORATORIUM 4. PROLOG – LISTY.

Programowania logiczne. Prolog: składnia języka, atomy, termy, zmienne, predykaty, fakty, reguły, nawroty, odcięcia, rekursja, operacje na listach.

Cel laboratorium:

Zdobycie umiejętności konstruowania i manipulowania listami. Wykorzystanie mechanizmu unifikacji, rezolucji i operatora `|` w odniesieniu do list. Zdobycie wiedzy na temat łączności operatorów i jej typów.

Zakres tematyczny zajęć:

- definiowanie list w Prologu,
- użycie operatora `|`,
- użycie zmiennej anonimowej w listach,
- unifikacja list,
- operatory, łączność i priorytet,
- tworzenie własnych operatorów,
- przesilenia operatorów.

Pytania kontrolne:

1. Jak wygląda definicja listy w Prologu?
2. Z jakich elementów może składać się lista,
3. Do czego służy operator `|` i jaki jest jego priorytet?
4. Co to jest głowa i ogon w kontekście list w Prologu?
5. Jak sprawdzić łączność i priorytet operatora?
6. Jaki predykat tworzy w Prologu nowe operatory?
7. Jakie rodzaje łączności operatorów występują w Prologu?

Zadanie 4.1. Listy w Prologu

Lista, to podstawowa struktura danych w wielu językach, zarówno funkcyjnych, jak i imperatywnych. W Prologu lista to sekwencja zera lub więcej termów, wpisana w nawiasach kwadratowych, oddzielonych przecinkami. Na przykład:

```
[alfa, bravo, charlie].  
[1, 2, 3, cztery].  
[(2+2), posiada(marcin, auto), -4.12, X].  
[[a, b], [b, c], [d, e, f]].
```

Elementy listy mogą być termami dowolnego typu, wliczając w to inne listy. Pusta lista to `[]`. Jednoelementowa lista `[a]` nie jest równoznaczna atomowi `a`. Listy mogą być konstruowane



i rozkładane przez unifikację, ale najważniejsze jest to, że każda lista może zostać podzielona na głowę oraz ogon za pomocą operatora `|`. Operator ten wykonuje się wcześniej niż reguła. Głowa to pierwszy element listy, ogon to reszta. Głową listy jednoelementowej jest ten element, ogon to lista pusta. Lista pusta nie ma głowy i ogona. Ogon listy jest zawsze listą. Głowa może, lecz nie musi być listą.

Co otrzymamy po podaniu następujących pytań?

```
?- []=[H|T].
?- [1,2]=[H|T].
?- [1]=[H|T].
?- [1,[2,3]]=[H|T].
?- [[1,2],3]=[H|T].
?- [1,2,3,4]=[Ha,Hb|T].
?- [[1,2,3],4]=[[H1|T1]|T2].
```

Na przykład reguła sprawdzająca czy przekazany term jest listą, będzie wyglądała następująco:

```
czyLista([]).
czyLista([_|T]) :- czyLista(T).

?- czyLista([w]).
true.

?- czyLista([]).
true.

?- czyLista(w).
false.
```

Operator `|` wykonuje się przed `:-`

Jakie będą różnice w wykonaniu dwóch poniżej zapisanych programów.

```
czyLista([]).
czyLista([_|T]) :- czyLista(T),write(T).
?- czyLista([a,b,c]).
```

oraz

```
czyLista([]).
czyLista([_|T]) :- write(T),czyLista(T).
?- czyLista([a,b,c]).
```

Reguła zwracająca ostatni element listy wygląda następująco.

```
ostatni([X],X).  
ostatni([H|T],X) :- ostatni(T,X).  
?- ostatni([a,b,c,],W).  
W = c;  
false.
```

Reguła sprawdzająca czy coś należy do listy (jeżeli nie jest głową to należy do ogona).

```
isMember(X,[Y|_]) :- X=Y.  
isMember(X,[_|Y]) :- isMember(X,Y).  
?- isMember(b,[b,a,b,c,b,c]).  
true;  
false.
```

Wiele programów w Prologu może mieć rozszerzoną funkcjonalność, w tym przypadku może zapytać jakie elementy należą do listy.

```
?- isMember(X,[a,b,c]).  
X = a ;  
X = b ;  
X = c ;  
false.
```

```
?- isMember(X,[a,[b,c],d]).  
X = a ;  
X = [b, c] ;  
X = d ;  
false.
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz program obliczający ilość elementów należących do listy. Wzoruj się na programie `czyLista([a,b,c]).`

```
?- ile([a,b,c],W).  
W = 3.
```

Prawdopodobnie napisałeś program stosując metodę „z góry na dół”.



Polecenie 2.

Napisz program obliczający sumę elementów liczbowych należących do listy, zastosuj metodę „z góry na dół”

```
?- suma([1,2,3,4],W).  
W = 10.
```

Polecenie 3.

Napisz program obliczający sumę elementów należących do listy, wersja z akumulatorem „z dołu do góry”.

```
?- suma([1,2,3,4],0,W).  
W = 10.
```

Zadanie 4.2. Operatory, priorytet, łączność.

Predykat `current_op(priorytet, łączność, nazwa)` pozwala sprawdzić dane na temat dowolnego operatora, na przykład:

```
?- current_op(X, Y, +).  
X = 200,  
Y = fy ;  
X = 500,  
Y = yfx.  
  
?- current_op(X, Y, *).  
X = 400,  
Y = yfx.  
  
current_op(P, L, X).
```

Przez X tutaj określony jest priorytet (im niższy, tym jest wcześniej wykonywany), a przez Y określamy łączność operatorów.

Możliwe jest również modyfikowanie istniejących operatorów:




```
?- X is 2+3*5.  
X = 17.  
  
?- op(100,yfx,+).  
?- X is 2+3*5.  
X = 25.
```

Na zakończenie ukłon w stronę języków naturalnych. Możliwe jest stworzenie własnych operatorów za pomocą predykatu `op(priorytet, łączność, nazwa)`. Nie można jednak tego zmieniać w plikach, które są ładowane z zewnątrz, a tylko w konsoli Prologa:

```
?- lubi(jas, malgosia)=jas lubi malgosia.  
ERROR: Syntax error: Operator expected  
ERROR: lubi(jas, malgosia)=jas  
ERROR: ** here **  
ERROR: lubi malgosia .  
  
%natomiast po wykonaniu  
?- op(100, yfx, lubi).  
% a następnie  
?- lubi(jas, malgosia)=jas lubi malgosia.  
true.
```

LABORATORIUM 5. HASKELL – WPROWADZENIE DO PROGRAMOWANIA FUNKCYJNEGO.

Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda.

Cel laboratorium:

Nabycie umiejętności przygotowania środowiska pracy dla Haskell w systemie operacyjnym Windows i Linux. Zdobycie umiejętności uruchamiania funkcji z poziomu interpretera oraz ze skryptu. Poznanie podstawowych typów prostych, złożonych i polimorficznych. Nabycie umiejętności definiowania funkcji z użyciem typów prostych i polimorficznych oraz wykorzystywania wnioskowania Haskell w pisanych programach.

Zakres tematyczny zajęć:

- przygotowanie środowiska pracy po kontrolą systemu operacyjnego Windows lub Linux,
- instalacja pakietów w Windows za pomocą Chocolatey,
- zapoznanie z podstawami typami funkcji w Haskellu,
- funkcje polimorficzne,
- definiowanie i uruchamianie funkcji z poziomu interpretera,
- definiowanie i uruchamianie funkcji z poziomu interpretera po zapisaniu kodu do pliku,
- zasada wnioskowania w Haskellu,
- wpływ operatorów na wnioskowanie.

Pytania kontrolne:

1. W jaki sposób uruchamiamy interpreter Haskell w środowisku Linux lub Windows?
2. Jakie zasady określają nazewnictwo funkcji w Haskellu?
3. Co to jest typ polimorficzny?
4. Jak skonstruowana jest funkcja polimorficzna?
5. Na czym polega wnioskowanie o typie w Haskellu?
6. Wymień podstawowe klasy typów w Haskellu i związane z nimi operatory?

Zadanie 5.1. Przygotowanie środowiska pracy w systemie operacyjnym Windows10 w oparciu o PowerShell, WSL2 oraz Linux (Kubuntu 20.04 LTS).

Instalacja Haskell w środowisku Windows w oparciu o PowerShell.



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



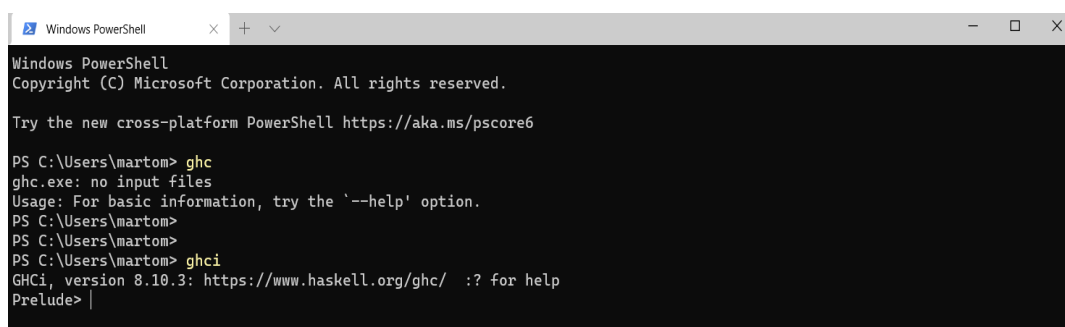
W celu instalacji w tym systemie operacyjnym należy skorzystać z tego linku (<https://www.haskell.org/platform/windows.html>) i postępować według zamieszczonego tam opisu.

1. Zaczynamy od instalacji Chocolatey. W tym celu korzystamy z linku: <https://chocolatey.org/install>. Tam znajduje się szczegółowy opis. Uruchamiamy PowerShell z uprawnieniami administratora.
2. Następnie wykonujemy polecenie: `Get-ExecutionPolicy` w efekcie otrzymamy `Restricted`.
3. Potem `Set-ExecutionPolicy Bypass -Scope Process`
4. W końcu wykonujemy polecenie:
`Set-ExecutionPolicy Bypass -Scope Process -Force;`
`[System.Net.ServicePointManager]::SecurityProtocol =`
`[System.Net.ServicePointManager]::SecurityProtocol -bor 3072;`
`ie x ((New-Object`
`System.Net.WebClient).DownloadString('https://chocolatey.org/in`
`stall.ps1'))`
5. Jeżeli nie było błędów, po chwili możemy sprawdzić poprawność instalacji pisząc w linii komend `choco`, w efekcie zobaczymy:

```
PS C:\WINDOWS\system32>
PS C:\WINDOWS\system32> choco
Chocolatey v0.10.15
Please run 'choco -?' or 'choco <command> -?' for help menu.
PS C:\WINDOWS\system32>
```

Rys. 5.1. Widok uruchomionego programu choco

6. Wszystko OK, jeżeli wcześniej zainstalowany był `cabal` to wykonujemy polecenie w celu deinstalacji starych wersji wykonując: `cabal user-config init -f`
7. Teraz została już tylko instalacja całego środowiska. Między innymi zainstalowany zostanie Haskell oraz `cabal`. Konieczny jest dostęp do internetu. Całość zajmie około 500MB i potrwa około 5 minut.
8. Na zakończenie zostało jeszcze odświeżenie środowiska: `refreshenv`
9. Zamykamy PowerShella.
10. Po instalacji możemy sprawdzić działanie Haskella wykonując polecenie `ghc` (kompilator Haskella) lub `ghci` (interpreter) w terminalu (WindowsTerminal) lub konsoli (cmd) lub w PowerShell. W efekcie zobaczymy:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\martom> ghc
ghc.exe: no input files
Usage: For basic information, try the '--help' option.
PS C:\Users\martom>
PS C:\Users\martom> ghci
GHCi, version 8.10.3: https://www.haskell.org/ghc/  :? for help
Prelude> |
```

Rys. 5.2. Widok uruchomionego w konsoli Haskell. Pierwszy uruchomiono kompilator, następnie interpreter.

Instalacja Haskell w środowisku WSL2 lub Linux (Kubuntu 20.04 LTS).

Windows Subsystem for Linux (WSL) już mamy zainstalowany (patrz pierwsze laboratorium). W tym momencie należy tylko zainstalować ulubioną wersję Linuxa. Też została zainstalowana na jednych z pierwszych zajęć. W przypadku instalacji Haskell nie ma znaczenia czy instalujemy go w samodzielnym systemie operacyjnym, czy zainstalowanym pod kontrolą Windows, czy w VirtualBox. W przypadku Kubuntu 20.04 wykonujemy jedno polecenie lub kilka w zależności czy jest dodane repozytorium Haskell w systemie. Jak sprawdzić? Najlepiej po prostu spróbować zainstalować pisząc:

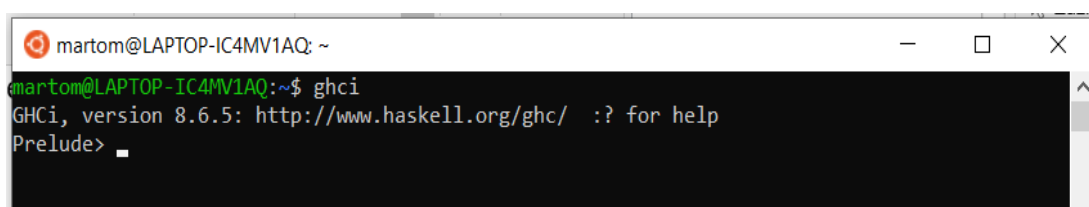
```
sudo apt-get install ghc
```

Jeżeli instalator pakietów nie znajdzie `ghc`, to wtedy dodajemy repozytorium pisząc:

```
sudo add-apt-repository -y ppa:hvr/ghc
sudo apt-get update
```

Następnie instalujemy Haskell pierwszym poleceniem.

Po uruchomieniu Kubuntu a następnie `ghci` powinniśmy zobaczyć:



```
martom@LAPTOP-IC4MV1AQ: ~
martom@LAPTOP-IC4MV1AQ:~$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> |
```

Rys. 5.3. Widok uruchomionego w konsoli Haskell

Zadanie 5.2. Haskell – podstawy języka

Paradygmat funkcyjny. Opisujemy **cel**, który wykonawca (komputer) ma osiągnąć, w tym rodzaju programowaniu programista opisuje, **co** komputer ma osiągnąć. Program to po

prostu złożona **funkcja (w sensie matematycznym)**, która otrzymawszy dane wejściowe wylicza pewien wynik.

The Glasgow Haskell Compiler (GHC) jest jednym z kompilatorów Haskellu. Do dyspozycji mamy także interpreter o nazwie `ghci`. Haskell daje możliwość tworzenia definicji złożonych funkcji, które wykorzystujemy w obliczeniach. Zbiór definicji funkcji nazywamy skryptem.

Można w nim ładować pliki z kodem programu o rozszerzeniu `.hs` poprzez komendę:

```
:load <nazwa pliku>
lub
:l <nawa pliku>
```

Wśród wielu dostępnych komend wbudowanych w interpretera, najczęściej używanymi są te przedstawione w tabeli 5.2:

Tabela 5.2. Podstawowe polecenia interpretera

Komenda	Działanie
:load name	załadowanie skryptu
:reload	przeładowanie bieżącego skryptu
:set editor name	zdefiniowanie polecenia edytora
:edit name	edycja skryptu
:edit	edycja bieżącego skryptu
:type expr	pokaż typ wyrażenia
:?	pokaż wszystkie komendy
:browse	pokaż funkcje zdefiniowane w module
:quit	wyjdź z GHCi

Interpreter `ghci` może być wykorzystywany identycznie jak interpreter Pythona – po podaniu dowolnej wartości wyrażenia matematycznego nastąpi jego obliczenie, na przykład:

```
Prelude> 2+17*12
206
```

Wykonaj samodzielnie poniższe przykłady:

Tabela 5.1. Aplikacje funkcji w interpreterze

1. Prelude> 5+4*2	15. Prelude> not False
2. Prelude> sqrt (3^2 + 4^2)	16. Prelude> mod 5 3
3. Prelude> abs (-1)	17. Prelude> div 8 3
4. Prelude> sin 1.2	18. Prelude> "ala"=="ala"
5. Prelude> sin (1.2)	19. Prelude> 7==7
6. Prelude> (sin) 1.2	20. Prelude> 7<=8
7. Prelude> sqrt (sin 1.2)	21. Prelude> 9>=8



8. Prelude> 2^200	22. Prelude> 7/=8
9. Prelude> (/) 2 3	
10. Prelude> (+) 2 3	23. Prelude> let square x = x ^ 2
11. Prelude> 2 + 3	Prelude> square 2
12. Prelude> 2 / 3	
13. Prelude> True && False	24. Prelude> ex15 = (5 > 3) && ('p' <= 'q')
14. Prelude> True False	Prelude> ex15

Charakterystyka języka

- wszystko w programie to funkcja,
- wysokopoziomowy,
- paradygmat funkcyjny,
- silnie typowany, typy wnioskowane. Niemożliwa jest więc przypadkowa (niejawna) konwersja, np. Double do Int,
- leniwy: wyznacza wartości argumentów funkcji co najwyżej raz i tylko wtedy, kiedy są potrzebne, brak wykonywania niepotrzebnych obliczeń,
- bez efektów ubocznych; funkcja może zwrócić efekt uboczny, który może być następnie wykorzystany,
- umożliwia tworzenie abstrakcyjnych struktur danych, polimorfizm i enkapsulację. Enkapsulacja jest realizowana poprzez umieszczenie każdego typu danych w osobnym module.

Podstawy języka.

Nazwy funkcji muszą zaczynać się małą literą. Dalej mogą wystąpić litery, cyfry, znaki podkreślenia (`_`) lub apostrofu (`'`).

Typy funkcji zapisujemy z dużych liter na przykład:

```
y = 5 :: Float -- jawna definicja funkcji
```

Bloki programu oddzielane są pustą linią

```
{-  
To jest komentarz  
blokowy  
-}  
-- ten komentarz rozciąga się do końca linii
```

Podstawowe typy funkcji: typy proste

Typy całkowite



Int (skończonej precyzji)	56
Integer (nieskończonej precyzji)	73214568

Typy rzeczywiste

Float/Double	3.14159265
--------------	------------

Typ logiczny

Bool	False
------	-------

Typ znakowy, napisowy

Char	'a'
String (lista znaków)	"ala"

Podstawowe typy funkcji: typy złożone

Krotka ma określony rozmiar, ale może zawierać elementy różnego typu jest heterogeniczna

(Int, Char)	(1, 'a')
(Int, Char, Float)	(1, 'a', 3.4)
((Bool, String), Int)	((True, "Ala"), 2)
([Int], Char)	([-1, 4, 2], 'c')

Lista ma nieokreślony rozmiar, ale jej elementy muszą być tego samego typu, jest homogeniczna z natury

[Int]	[1, 2, 3]
[Char]	['a', 'b']
[[Int]]	[[1], [1, 4], []]
[(String, Bool)]	[("Ala", True), ("kot", False)]

Podstawowe operatory:

Logiczne:	&& < <= > >= == /=
Matematyczne:	+ - * / ^

Używanie funkcji różni się od innych języków – nie używa się przy wywołaniu funkcji nawiasów ani przecinków, do oddzielenia kolejnych parametrów używa się spacji, na przykład:

```
Prelude> min 2 17  
2
```

Nawiasy można wykorzystywać nadal do grupowania pewnych wyrażeń matematycznych. Deklaracja własnych funkcji w pliku kodu jest z użyciem operatora `=`, na przykład:

```
Prelude> x = 2    -- definicja funkcji pierwszej
```

W identyczny sposób deklaruje się funkcje przyjmujące parametry, na przykład:

```
Prelude> funkcja x = x + 1    -- definicja funkcji drugiej
```

Użycie tych dwóch definicji w jednym pliku spowoduje błąd wykonania.

Haskell nie obsługuje definicji zmiennych (powiedzmy globalnych), które „żyją” w czasie działania programu i przechowują pewien stan – zmienne mogą istnieć tylko i wyłącznie wewnątrz funkcji.

W przypadku podwójnej deklaracji takiej samej funkcji w pliku ładowanym do środowiska wygeneruje błąd – aczkolwiek da się to zrobić w oknie interpretera, ponieważ każda późniejsza deklaracja nadpisuje poprzednią.

Typ polimorficzny

Typ polimorficzny oznacza rodzinę typów. Typy są pogrupowane w klasy typów (ang. *type classes*). Klasy typów określają wspólne właściwości typów należących do danej klasy (patrz rysunek 5.3). Niektóre z nich zawierają się w innych. Pozwalają na pisanie definicji funkcji dla całej klasy typów, a nie dla poszczególnych typów zmiennych. Funkcje zawierające zmienne typu polimorficznego są nazywane funkcjami polimorficznymi.

Parametry tych funkcji mogą posiadać zmienne o nazwach dłuższych niż jednoliterowe, ale przyjęto się, że używa się właśnie nazw o długości jednej litery np. `fst :: (a, b) -> a`. Definicja funkcji składa się z dwóch części:

- opisu typu funkcji (w tym definicji nazwy modułu), które mogą być pominięte, jeżeli nie prowadzi to do niejednoznaczności, czytamy funkcja `zwiększ` bierze jeden argument typu `Int` i zwraca jeden argument tego samego typu,

```
module Test where  
zwiększ :: Int -> Int
```

- sposobu wyliczenia wartości funkcji,

```
zwiększ x :: x + 1
```

Kompletna definicja funkcji wygląda następująco:




```
module Test where
zwiększ :: Int -> Int
zwiększ x :: x + 1
```

A tak wyglądała by ta sama definicja ale z użyciem klas typów:

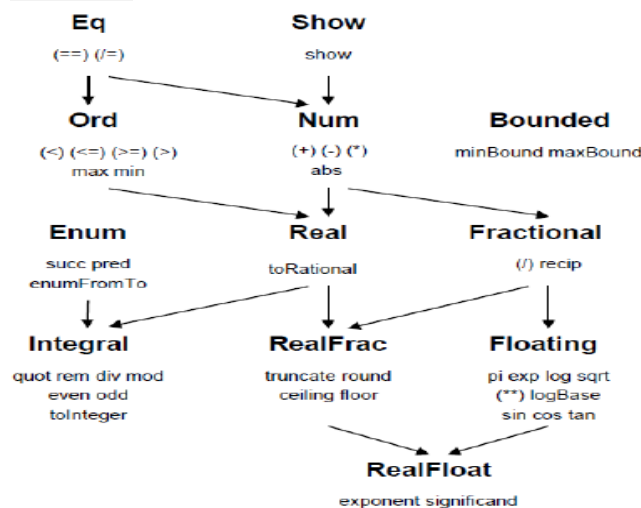
```
module Test where
zwiększ :: Num a => a -> a
zwiększ x :: x + 1
```

Z danym typem związane są dozwolone operatory i odwrotnie dany operator wymusza typ.

Przykładowo:

- **Num** jest jedną z predefiniowanych klas typów dla której zdefiniowane jest między innymi dodawanie, odejmowanie i mnożenie,
- **Eq** - klasa typów, dla których zdefiniowane jest porównywanie,
- **Enum** - klasa typów wyliczeniowych.

Typ każdego wyrażenia w języku Haskell możemy sprawdzić używając komendy `:t` lub `:type`, zaś komendą `:browse` zobaczymy funkcje dostępne w danym module (programie).



Rys. 5.3. Klasy typów i przypisane im funkcje

Zadanie 5.3. Haskell – wnioskowanie typów

W Haskellu typy funkcji wnioskowane w zależności od użytych funkcji i operatorów użytych do konstrukcji danej funkcji. Przykładowo:

Num->Fractional->Floating

Użycie operatora dodawania wymusza użycie klasy Num.

```
Prelude> :t (2+1)
(2+1) :: Num a => a
```

Użycie operatora dzielenia wymusza użycie klasy Fractional.



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
Prelude> :t (2+1)/2  
(2+1)/2 :: Fractional a => a
```

Użycie operatora funkcji sinus wymusza użycie klasy Floating.

```
Prelude> :t sin (2+1)/2  
sin (2+1)/2 :: Floating a => a
```

Kolejność deklarowania funkcji z poziomu interpretera wpływa na wynik. Brak wcześniejszej definicji `dM` uniemożliwi definicję `dUn`

```
Prelude> dU x y = x*2 + y*2  
Prelude> dUn x y = dM x +dM y  
Prelude> dM x = x + x
```

W przypadku zapisania powyższego kodu do pliku, kolejność deklaracji już nie odgrywa roli. Proszę spróbować samodzielnie zapisać powyższy kod do pliku i sprawdzić.

```
Prelude> dM 6  
Prelude> dU 6 3  
Prelude> dUn 6 3
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Zdefiniuj funkcje obliczające kwadrat podanego parametru. Funkcje powinny zwracać taki sam typ jak typ wejściowy. Po zapisaniu i załadowaniu modułu sprawdź zdefiniowane funkcje.

- a) Int
- b) Float
- c) Num
- d) pozwól na wnioskowanie typu

```
Prelude> dMe1 2  
Prelude> dMe1 2.5  
Prelude> dMe2 3  
Prelude> dMe2 2.5  
Prelude> dMe3 3  
Prelude> dMe3 2.5
```

Odpowiedz na pytanie, dlaczego po wykonaniu powyższych funkcji otrzymałeś różne rezultaty a może i błędy wykonania?

LABORATORIUM 6. HASKELL – PODSTAWOWE KONSTRUKCJE.

Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda.

Cel laboratorium:

Nabywanie umiejętności wykorzystywania podstawowych konstrukcji programistycznych podczas tworzenia kodu w Haskellu (if, case, |). Umiejętność wykorzystania właściwości operatorów. Znajomość konstrukcji `where` oraz `let ... in` oraz definiowania lokalnych funkcji w tych konstrukcjach.

Zakres tematyczny zajęć:

- priorytet operatorów oraz ich łączność,
- funkcje czyste,
- zapis infiksowy funkcji,
- konstrukcja warunkowa,
- wyrażenie `where`,
- wyrażenie `let ... in`,
- zakres widoczności funkcji w konstrukcji `where`
- zakres widoczności funkcji w konstrukcji `let ... in`.

Pytania kontrolne:

1. Co określa priorytet operatorów?
2. Na czym polega łączność operatorów?
3. Czy w Haskellu występuje konstrukcja warunkowa?
4. Jaka jest różnica pomiędzy `where` oraz `let ... in` ?
5. Jakiego typu funkcje można użyć w zapisie infiksowym?
6. Co oznacza termin **funkcje czyste**?
7. Czy funkcje czyste mogą mieć efekty uboczne?

Zadanie 6.1. Operatory i ich łączność w Haskellu

Kolejność wykonania operatorów (funkcji) jest określona przez priorytet operatora (funkcji), im wyższy tym silniejszy (patrz tabela 6.1). Dodatkowa właściwość "fixity" decyduje czy operator wiąże w lewo ("left-associative"), w prawo ("right-associative") czy równorzędnie w obu kierunkach ("non-associative"). Pamiętajmy! Aplikacja funkcji ma najwyższy priorytet (10).



Tabela 6.1. Priorytet oraz łączność operatorów w Haskellu na podstawie
<https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2>

Priorytet	łączność lewostronna	Brak łączności	łączność prawostronna
9	!!		.
8			^, ^^, **
7	*, /, 'div', 'mod', 'rem', 'quot'		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, 'elem', 'notElem'	
3			&&
2			
1	>>, >>=		
0			\$, \$!, 'seq'

Funkcje i operatory arytmetyczne w Haskellu mogą być używane zarówno w postaci prefiksowej, jak i infiksowej. Postać infiksowa (piszemy w znakach pojedynczego odwróconego apostrofu ```) jest bardziej naturalna dla operatorów arytmetycznych takich jak + - / * a postać prefiksowa dla funkcji arytmetycznych zapisanych w postaci ciągu znaków. Aplikacja prefiksowa i infiksowa funkcji

```
Prelude> min 2 7.4
Prelude> 2.7 `min` 4
Prelude> max 2.4 7
Prelude> 2.4 `max` 7
Prelude> div 92 10
Prelude> 12 `mod` 5
Prelude> 2+3
Prelude> (+) 2 3
Prelude> 2/3
Prelude> (/) 2 3
```

Operatory o wyższym poziomie pierwszeństwa będą wykonywane wcześniej. W celu wymuszenia innej kolejności wykonania musimy posłużyć się nawiasami:

```
Prelude> 2 + 2 * 2
6
Prelude> False && True || True
True
Prelude> (2 + 2) * 2
```

```
Prelude> 2*3*4
24
Prelude> 2*3^2*2
36
```

Operator `*` wiąże w lewo, więc najpierw wykonane zostało działanie $2*3$, a następnie $6*4$. Operator `^` zalicza się do grupy "right-associative" w związku z czym najpierw wykonane zostało działanie 3^2 , a następnie $2*9$, a następnie $18*2$.

Operatory infiksowe są z natury swej dwuargumentowe. Podając operatorowi jeden z argumentów możemy uzyskać funkcję jednoargumentową. Konstrukcja taka nazywa się przekrojem (ang. *section*) operatora. Przekrojów używamy przeważnie, gdy chcemy taką funkcję przekazać do innej funkcji.

```
Prelude> (+1) 2      Prelude> map (1+) [1, 2, 3]
3                  [2,3,4]
Prelude> (1+) 3      Prelude> map (>2) [1, 2, 4]
4                  [False,False,True]
Prelude> (0-) 4       Prelude> map ("ab" ++) ["cd", "ef"]
-4                 ["abcd","abef"]
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Wykonaj poniższe polecenie. Odpowiedz na pytanie: dlaczego otrzymano taki wynik? Wynik udowodnij innymi przykładami!

```
Prelude> mod 3 2^3 > 3`mod`2^3
False
```

Dowód:

Polecenie 2.

Wykonaj polecenie i odpowiedz na pytanie, dlaczego otrzymano taki wynik. Wynik udowodnij.

```
Prelude> 2^3^4
2417851639229258349412352
```



Dowód:

Zadanie 6.2. Konstrukcje programistyczne

Konstrukcja warunkowa.

Konstrukcja warunkowa nie istnieje w wersji bez else!

```
if war then
    instrukcja1
    instrukcja2
else
    instrukcja3
    instrukcja4
```

W składni Haskell'a istotne są wcięcia grupujące instrukcje. Tak więc, definicje najwyższego poziomu zaczynają się w tej samej kolumnie:

```
abs2 x = if x < 0 then -x else x      blok 1
c = 5                                   blok 2
```

Definicja może być „złamana” w dowolnym miejscu pod warunkiem, że wcięcia będą większe niż w linii początku dla danego **bloku**:

```
abs1
  x = if
    x < 0
  then -x else x
b =
  5
```

Ta definicja też jest poprawna, ale jej czytelność!

Proszę z rozważą używać tabulatora!

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz funkcję, która dla podanej liczby całkowitej zmienia jej znak na przeciwny.

A może by tak obliczać pierwiastek liczby naturalnej



Co otrzymamy dla $n < 0$.

Cechą języka funkcyjnego jest użycie tak zwanych funkcji czystych, to znaczy takich które nie mają efektów ubocznych. Co wtedy, gdy mamy te efekty uboczne. Język przewiduje formalny opis efektów ubocznych. Istnieją funkcje obsługujące te sytuacje. Oto jedna z nich:

```
error "liczba ujemna"
```

Funkcja wypisuje wyjątek z podanym tekstem `liczba ujemna`. Funkcja `error` pobiera łańcuch znaków i generuje błąd wykonania. Można jej użyć do wypisania informacji na ekranie o błędzie. Łańcuch znakowy to informacja co poszło nie tak.

Zmodyfikuj poprzednią funkcję pierwiastkującą, aby w przypadku pierwiastkowania liczby ujemnej wygenerowała stosowny komunikat. Porównaj wynik `sqrt (-2)` i ten otrzymany dla nowej funkcji.



Konstrukcja `where`

Definiuje wartości i funkcje użyte wewnątrz wyrażenia. Zasięgiem definicji w `where` — cała definicja funkcji. Jeżeli po `where` lub `let` występuje więcej niż jedna definicja lokalna, wszystkie muszą zaczynać się w tej samej kolumnie:

```
volume1 r = a * pi * cube
  where
    a = 4 / 3
    cube = r ^ 3
```

lub, również poprawnie

```
volume2 r = a * pi * cube r
  where a = 4 / 3
        cube x = x ^ 3
```

Konstrukcja `let ... in ...`

Definiuje wartości i funkcje użyte wewnątrz wyrażenia, podobnie jak `where`, lecz zasięgiem definicji w `let` jest wyrażenie po `in`.




```
volume3 r = let a = 4 / 3
           cube = r ^ 3
           in a * pi * cube
```

Różnice pomiędzy **let** a **where**. **where** dokonuje łączenia wyrażenia ze zmienną na końcu, zmienna jest wykorzystywana w funkcji wcześniej, a w **let** jest odwrotnie.

```
-- Lokalność definicji po where (let)
aa = 1
volume4 r = aa^2
           where aa = 4 / 3
fun x = aa * x
*Main> fun 5
5
```

```
-- Definicje po let przesłaniają te po where:
fun2 x = let y = x + 1
         in y
         where y = x + 2
*Main> fun2 5
6
```

Polecenie 2.

Napisz funkcję obliczającą deltę oraz pierwiastek dla równania kwadratowego postaci $y=ax^2+bx+c$. Użyj dwóch funkcji **delta** i **pdelta**.

```
pdelta :: Double -> Double -> Double -> Double
```

Polecenie 3.

Proszę uzupełnić powyższy kod o konstrukcję warunkową i funkcję **error**. Program powinien działać następująco: dla **delty > 0** wypisywany jest pierwiastek z delty, dla **delty = 0** program ma wypisać 0, a dla **delty < 0** ma być wywołana funkcja **error** ze stosownym komentarzem. Proszę użyć konstrukcję **where** lub **let ... in ...**

Polecenie 4.

Napisz program obliczający miejsca zerowe równania $y=ax^2+bx+c$. Uproszczenia: program dla delty $=0$ wypisuje dwa te same miejsca zerowe. Dla $a=0$ pisze stosowny komentarz. Ponadto proszę pamiętać, że w Haskellu funkcja zwraca zawsze jeden wynik, dlatego należy zastosować do zwracania miejsc zerowych którąś ze znanych struktur złożonych (lista, krotka). Używamy `if`, `where`, `let`, `error` i deklarujemy typ funkcji.

LABORATORIUM 7. HASKELL – REKURENCJA I LENIWE WARTOŚCIOWANIE.

Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda.

Cel laboratorium:

Nabywanie umiejętności zapisywania wywołań rekurencyjnych w Haskellu i ich optymalizowania. Wykorzystywanie w obliczeniach „leniwości” Haskellu. Nabywanie umiejętności wykorzystywania strażników w programach.

Zakres tematyczny zajęć:

- konstrukcja `case`,
- strażnicy,
- zagadnienie rekurencji w Haskellu
- leniwe wartościowanie w Haskellu,

Pytania kontrolne:

1. Co to są strażnicy?
2. Czy rekurencja jest algorytmicznie skończona, czy nieskończona w Haskellu?
3. Na czym polega leniwe wartościowanie w Haskellu?
4. Czy kolejność umieszczenia strażników ma znaczenie dla otrzymanego wyniku?

Zadanie 7. 1. Konstrukcje programistyczne c.d.

Konstrukcje `if ... then ... else if ... then ... else ...`

Można zapisać prościej za pomocą tzw. strażników `|`. Ogólnie konstrukcja tak przyjmuje postać:

```
fun x
  | war1 = akcja1
  | war2 = akcja2
  | war3 = akcja3
  | otherwise = akcja4  -- opcjonalne
```



Kolejność strażników ma znaczenie.

```
f a b | a >= b = "wieksze lub rowne"
      | a == b  = "rowne"
      | otherwise = "niewieksze"
      | a < b   = "mniejsze"
```

```
Prelude> f 1 1
"wieksze lub rowne"
```

```
Prelude> f 1 5
"niewieksze"
```

Konstrukcja case ... of ...

```
fcase1 x =
  case x of
    0 -> 1
    1 -> 5
    2 -> 2
    _ -> -1
```

W tej konstrukcji można pominąć symbol zastępczy `_`

```
fcase2 x =
  case x of
    0 -> 1
    1 -> 5
    2 -> 2
```

Przykład użycia:

```
fact n = case n of
           0 -> 1
           _ -> n * fact (n - 1)
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Proszę napisać funkcje `sign1` zwracającą znak podanej liczby używając konstrukcji warunkowej oraz `sign2` używając strażników.

$$\text{sgn}(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

Polecenie 2.

Napisz funkcję porównującą dwie liczby i zwracającą wynik typu Ordering. Użyj konstrukcji ze strażnikami lub case.

```
Prelude> 6 `compare` 7    Prelude> compare 6 7
LT                        LT
--LT, GT, EQ
```

```
--myComp :: Ord a => a -> a -> Ordering
```

Zadanie 7.2. Rekurencja

Podobnie jak w języku logicznym tak i w Haskellu nie ma zmiennych w tradycyjnie rozumianej formie, a więc nie ma także pętli, które są imperatywne z natury. W zamian króluje rekurencja.

Mnożenie dwóch liczb naturalnych, przypomnienie z Prologa

$$x \cdot y = y + (x - 1) \cdot y \text{ wzór rekurencyjny}$$

mul(1,X,X).

mul(X,Y,R) :- X1 is X-1, mul(X1,Y,R1), R is R1 + Y

```
mul(4,3,X)
|
3 + mul(3,3,X)
|
3 + mul(2,3,X)
|
3 + mul(1,3,X)
```

Zapiszmy ten sam przykład w Haskellu

```
--mul1 :: Int ->Int ->Int
mul1 a 1 = a
mul1 a b = a + mul1 a (b-1)
```

```
Prelude> mul1 4 3
Prelude> mul1 4 (-3)  -- dlaczego
Prelude> mul1 (-3) 4
```

Niby lepsza wersja, ale czy na pewno?

```
mul2 1 b = b
mul2 a b = b + mul2 (a-1) b
-- Wady???
```

```
Prelude> mul2 3 4
Prelude> mul2 (-3) 4  --dlaczego
Prelude> mul2 4 (-3)
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Na podstawie dwóch poprzednich przykładów napisz samodzielnie poprawnie działającą wersję mnożenia.

Napisz inne wersje programu pozbawione tych wad, może użyj strażników.

Polecenie 2.

Napisz funkcje obliczającą wartość silni. Użyj rekurencji, konstrukcji ze strażnikami, konstrukcji warunkowej lub case.

Pierwszy sposób obliczania silni

Zaproponuj inny sposób obliczania silni

Polecenie 3.

Napisz funkcję obliczającą kolejne elementy ciągu Fibonacciego.

Zadanie 7.3. Leniwe wartościowanie

Jedną z ważnych cech Haskell'a jest leniwe wartościowanie. Wyrażenia nie są obliczane w momencie wiązania ich do zmiennej, ale dopiero, gdy napotkane zostanie odwołanie do konkretnego wyniku.

Ten przykład już był, pierwiastek nie zostanie obliczony, bo i po co?

```
psqrt x = sqrt(if x >= 0 then x else error "liczba ujemna")
Prelude> psqrt (-2)
```

Kolejny przykład z listą, listy będą w następnym ćwiczeniu. Funkcja (`from 1`) generuje nieskończoną listę liczb `[1,2,3,4,5]`

```
from n = n:from(n+1)
const x y = x

Prelude> const 0 (from 1)
0
```

Obliczenie wartości `const 0 (from 1)` nie wymaga obliczenia `from 1`, więc nie jest obliczane.

Kolejny przykład:

```
roots (a, b, c) = if d < 0 then error "pierwiastki urojone"
                  else (r1, r2) where
                    r1 = e + sqrt d / (2 * a)
                    r2 = e - sqrt d / (2 * a)
                    d = b * b - 4 * a * c
                    e = -b / (2 * a)
```

`r1`, `r2` będzie obliczone gdy `d >= 0`, zaś `e` zostanie obliczone tylko raz gdy będzie liczone `r1` lub `r2`

LABORATORIUM 8. HASKELL – LISTY.

Wprowadzenie do programowania funkcyjnego. Haskell: składnia języka, funkcje, strażnicy, struktury danych, listy parametryczne, typy funkcji, funkcje częściowe, rekursja, dopasowanie do wzorca, funkcje wyższego rzędu i funkcje lambda.

Cel laboratorium:

Poznanie cech charakterystycznych list i krotek oraz podstawowych operatorów pozwalających na manipulowanie nimi. Nabycie umiejętności wykorzystania list zasięgowych oraz funkcji map, filter, takeWhile, fold. Aplikacja funkcji anonimowej w Haskellu.

Zakres tematyczny zajęć:

- listy i krotki w Haskellu,
- struktury homo i heterogeniczne,
- konstruktor tworzący listę,
- operator indeksowania listy,
- konkatenacja listy,
- listy nieskończone (specyfikacja zasięgu),
- listy zasięgowe (parametryczne),
- operator aplikacji zbiorowej, filtrowanie list,
- tak zwane funkcje związające fold,
- funkcja anonimowa.

Pytania kontrolne:

1. Czy listy są homo czy heterogeniczne?
2. Czy krotki (tuple) są heterogeniczne?
3. Czy krotki mogą zmieniać swoją długość?
4. Czy listy mogą zmieniać swoją długość?
5. Podaj przykład listy zasięgowej i omów zasadę działania?
6. Jak działa konstruktor tworzący listę?
7. Podaj nazwę operatora aplikacji zbiorowej?
8. Podaj nazwę funkcji filtrującej listy?
9. Do czego służą funkcje z rodziny fold?
10. Co to jest funkcja anonimowa?



Zadanie 8.1. Haskell - listy

Podstawową strukturą danych w Haskellu, jak i w Prologu oraz Pythonie, jest lista. W odróżnieniu od tych dwóch, lista jest homogeniczna, tj. może przechowywać dane tylko jednego typu. Łańcuchy znaków również są traktowane jako listy. Listy deklaruje się w zwykły sposób, na przykład:

```
Prelude> lista = [1, 2, 3]
```

Stworzy funkcję o nazwie lista, która zwraca zawsze listę elementów 1, 2 oraz 3. Ale można również zadeklarować tak:

```
Prelude> [1..10]
```

oraz:

```
Prelude> lista = [1,3..10]
```

Co wygeneruje listę od 1 do 10 w pierwszym przypadku, a listę od 1 do 10 co dwa w drugim przypadku. Działa to również dla liter, na przykład:

```
Prelude> [ 'a' .. 'z' ]
```

Łączenie list realizuje się przez operator ++, na przykład:

```
Prelude> [1, 2, 3] ++ [10, 11, 12]  
[1,2,3,10,11,12]
```

Działa to oczywiście także dla łańcuchów znaków:

```
Prelude> let s = "Hello" ++ " " ++ "world"  
Prelude> s  
"Hello world"
```

Operator : dodaje element na początek listy. Operator !! pobiera element listy o podanym indeksie, na przykład:

```
Prelude> [1, 2, 3] !! 2  
3
```

Funkcje head, tail zwracają głowę i ogon listy, last zwraca ostatni element, length zwraca długość listy, reverse ją odwraca i zwraca odwróconą.

Funkcja take pobiera liczbę oraz listę i zwraca pierwsze X elementów listy.

Funkcja drop zwraca listę bez pierwszych X elementów.

Funkcja sum zwraca sumę listy, a product zwraca iloczyn elementów listy.



Funkcja `elem` pobiera element oraz listę i zwraca czy element jest w liście. Można jej użyć na dwa sposoby:

```
Prelude> elem 1 [1, 2, 3]
```

Ale bardziej czytelny jest sposób infiksowego użycia, jako operator:

```
Prelude> 1 `elem` [1, 2, 3]
```

Funkcje: `lambda`, `map`, `filter`, `takeWhile` oraz funkcje `fold`.

Funkcje anonimowe nie mają nazwy, a zaczyna się je znakiem `\`. Potem pojawia się lista parametrów, potem `->` i ciało funkcji, na przykład:

Na przykład `\x -> x + 1` oznacza funkcję, która dla argumentu `x` daje wartość `x + 1`

```
Prelude> (\x -> x*x) 2
4
Prelude> f = \x -> x + x
Prelude> f 5
10
```

Ale funkcje anonimowe mogą oczywiście przyjmować więcej parametrów niż 1, jak normalne funkcje.

```
Prelude> (\x y -> x+y) 1 2
3
Prelude> (\x -> \y -> x+y) 1 2
3
```

Funkcja `map` pobiera funkcję i listę, a następnie uruchamia funkcję dla każdego elementu listy, tworząc nową listę

```
Prelude> map (+ 2) [1 .. 5]
Prelude> map (\x->x-2)[1..10]
Prelude> map (\x->x+2)[1..10]
Prelude> map (\a -> a * 3) [1, 2, 3]
Prelude> map silnia1 [1..5] – silnia musi być zdefiniowana
```

Funkcja `filter` to funkcja, która oczekuje predykatu (funkcji zwracającej `True` albo `False`) i listy. Zwraca listę elementów z listy, dla których predykat jest prawdą.

```
Prelude> filter (> 2) [1 .. 5]
Prelude> filter (< 3) [1, 2, 3, 4, 1, 2, 3, 4]
```



Funkcja `takeWhile` oczekuje predykatu oraz listy i zwraca wszystkie elementy listy aż do momentu, kiedy predykat zwróci pierwsze `false`. Na przykład:

```
takeWhile (/=' ') "slonie lubia imprezy"
```

Zwróci "slonie".

Funkcje `fold` przetwarzają uporządkowane kolekcje danych (zazwyczaj listy) w celu wyliczenia końcowego wyniku przy pomocy jakiejś funkcji łączącej elementy (suma, iloczyn, iloraz). Dwie najbardziej popularne funkcje z tej rodziny to **foldr** (ang. *fold right*) i **foldl** (ang. *fold left*).

Na przykład, obliczanie sumy kolejnych elementów listy.

```
product [] = 0
product (x:xs) = x + product xs
```

Za pomocą funkcji `foldr` możemy to zapisać w następujący sposób:

```
Prelude> foldr (+) 0 [1,2,3]
```

Funkcje `foldl` i `foldr` działają identycznie – w `foldl` akumulator uzyskuje wartość pierwszego elementu od razu, zaś w `foldr` na końcu.

```
foldr (+) 6 [1,2,3,4,5]
-- 1+(2+(3+(4+(5+6)))) od prawej do lewej szybciej
```

```
foldl (+) 6 [1,2,3,4,5]
-- (((6+1)+2)+3)+4)+5 od lewej do prawej wolniej
```

Kolejne wzorcowe przykłady na leniwość Haskella. Proszę sprawdzić co wyjdzie z każdego podanego poniżej przykładu.

```
Prelude> take 5 [1..]
Prelude> head [100..]
Prelude> take 2 (cycle [1..3])
Prelude> take 8 (cycle "LOL ")
Prelude> take 10 (repeat 5)
```



Zadanie 8.2. Haskell - krotki

Krotki (tuple) przechowują pary, trójki, lub więcej wartości. Używa się ich identycznie jak np. w Pythonie, używając nawiasów i przecinka dla definicji kolejnych elementów. W odróżnieniu od Haskellowych list, mogą przechowywać elementy różnych typów. Funkcje `fst` oraz `snd` pobierają tuple i zwracają odpowiednio pierwszą i drugą wartość, na przykład:

```
Prelude> fst (8,11)
8
Prelude> snd (8,11)
11
Prelude> zip "abc" [1,2,3,4]
[( 'a',1), ( 'b',2), ( 'c',3)]
```

Zadanie 8.3. Haskell - listy zasięgowe

Listy zasięgowe (ang. *list comprehensions*), identycznie jak w Pythonie, przechodzą po liście i wykonują pewną funkcję na wszystkich jej elementach, tworząc nową listę. Przykładowo:

```
Prelude> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

Każdy element z listy `[1..10]` jest zapisywany do zmiennej `x` i następnie ta zmienna przechodzi przez funkcję `x*2` i tworzona jest nowa lista. W list comprehensions dostępny jest również operator `if`, po przecinku, na przykład:

```
Prelude> [ x | x <- [50..100], x `mod` 7 == 3 ]
[52,59,66,73,80,87,94]
```

Można również przechodzić po dwóch (i więcej) listach naraz, gdzie dla każdego elementu z pierwszej uruchomi się każdy element z kolejnej, a dla każdego z kolejnej, każdy z następnej, na przykład:

```
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50 ]
[55,80,100,110]
```

Możliwe jest także użycie zmiennej anonimowej `_` (jak w Prologu), kiedy nie interesuje nas dana z listy, na przykład:

```
Prelude> sum [ 1 | _ <- [ 1 .. 3] ]
```

Funkcja obliczy długość listy 3.



Kolejny przykład:

```
Prelude> cilit xs = [ if x < 10 then "BOOM!" else "BANG!" | x <-  
xs, odd x]  
Prelude> cilit [7..13]  
["BOOM!", "BOOM!", "BANG!", "BANG!"]
```

List comprehensions mogą generować krotki, na przykład:

```
Prelude> let triangles = [ (a,b,c) | c <- [1..6], b <- [1..5], a  
<- [1..5] ]  
Prelude> triangles
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz funkcję obliczającą iloczyn elementów listy, użyj foldr.

```
6
```

Polecenie 2.

Napisz funkcję sprawdzającą czy podany znak jest dużą literą. Użyj elem.

```
Prelude> czyUp 'A'  
True
```

Polecenie 3.

Napisz funkcję usuwającą z napisu duże litery. Napisz to lista znaków.

```
rm1 "Ala ma"  
Prelude> lama
```

Polecenie 4.

Napisz kod programu, który ze wszystkich możliwych trójek (a, b, c) wygeneruje tylko te, dla których $a^2 + b^2 = c^2$, tak zwane trójki pitagorejskie. Wartości a,b,c należą do przedziału od 0 do 20. Użyj list comprehensions.

Polecenie 5.

Napisz funkcję obliczającą silnię dla liczb parzystych w zakresie od 1 do 20. Użyj funkcji `map`, `filter`, `where`, `even`, `mod` i specyfikacji zasięgu listy.

Polecenie 6.

Napisz funkcje obliczającą kwadraty kolejnych liczb naturalnych przedziale od 0 do 20. Użyj `map` i `where`. Następnie drugi program z `map` i funkcją `lambda`.

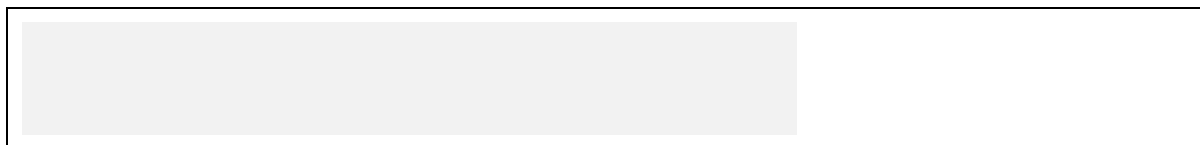
Polecenie 7.

Napisz kod, który zwróci największą wartość poniżej 1000000 podzielną przez 3829 bez reszty. Zastosuj klasyczne rozwiązanie. Napisz drugą wersję funkcji wykorzystując leniwość Haskella. Która jest szybsza.

Te przykłady pokazują także leniwość Haskella, lista jest sortowana malejąco i wystarczy pierwszy element. Nie ma znaczenia czy lista jest skończona czy też nie. Kończy gdy znajdzie pierwszy wynik.

Polecenie 8.

Napisz funkcję obliczającą liczbę wszystkich nieparzystych kwadratów liczb od 1 do 10000. Do obliczeń możesz wykorzystać `takeWhile`, `sum`, `odd`, `map`, `filter`, `length`.



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



LABORATORIUM 9. PYTHON – WPROWADZENIE.

Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego.

Cel laboratorium:

Zapoznanie ze składnią języka Python. Poznanie konsekwencji dynamicznie przypisywanych typów zmiennych. Nabycie umiejętności pracy z podstawowymi strukturami danych.

Zakres tematyczny zajęć:

- instalacja Pythona i jego tryb interaktywny
- składnia języka Python,
- dynamiczne typowanie zmiennych, inferencja typów,
- słowniki jako struktury danych,
- listy i operatory zasięgowe,
- krotki,
- pętla for-in.

Pytania kontrolne:

1. Czy język Python pozwala na programowanie zorientowane obiektowo?
2. Jaka jest różnica pomiędzy dynamicznym i statycznym typowaniem?
3. Czy w składni języka Python mają znaczenie białe znaki?
4. W jaki sposób nie można wykorzystywać parametrów nazwanych?
5. Czy pętla for-in może operować na ciągu znaków?
6. W jaki sposób wykonuje się dekonstrukcję krotki?

Zadanie 9.1. Instalacja interpretera języka Python

Język Python jest wieloparadygmatowym, skryptowym językiem programowania, na podstawie którego poznasz mechanizmy dynamicznego typowania. W podobny sposób, jak w przypadku Haskella, korzystać będzie można z dostępnego interaktywnego trybu.

Aby zainstalować interpreter Python oraz narzędzia do pracy w trybie interaktywnym na platformie Ubuntu lub podobnej możesz skorzystać z polecenia




```
sudo apt install python3
```

Dla pozostałych platform możesz skorzystać z odpowiedniego instalatora dostępnego pod <https://python.org>, odpowiedniego sklepu z aplikacjami lub narzędzia Chocolatey, wydając komendę:

```
choco install python -y
```

Ważnym szczegółem jest jednak wersja Pythona – przez długi czas równolegle rozwijana była **wersja 2 oraz wersja 3, które nie są ze sobą kompatybilne**. Ostatnia wersja Python 2.7 uzyskała status „end-of-life” 1 stycznia 2020 roku i jej używanie jest wysoce niezalecane, nie zmienia to jednak faktu, że wiele organizacji nadal musi utrzymywać produkty opracowane z wykorzystaniem wersji 2. Kolejne duże wydania wersji 3 (np. 3.6, 3.7) są kompatybilne wstecznie, ale zawierają nowe funkcje i udogodnienia dla programisty. Na zajęciach będziemy posługiwać się elementami z wersji co najmniej 3.8.

Uwaga: niektóre platformy opartych o Linuksa nadal używają Pythona w wersji 2 jako domyślnego interpretera po wydaniu polecenia `python` i należy wtedy wydać komendę `python3`. W innych komenda `python` w ogóle może nie być dostępna.

```
marcinb@DESKTOP-DMS34MK:/mnt/c/Users/marcinb/Downloads$ python3 l9.py
Hello, dear Arya
marcinb@DESKTOP-DMS34MK:/mnt/c/Users/marcinb/Downloads$ |
```

Rys. 9.1. Przykład pracy w trybie wsadowym, z komendą `python3`.

Zadanie 9.2. Środowisko IDLE i tryb interaktywny

Po zainstalowaniu środowiska Python uzyskujesz dostęp do konsolowego interpretera, narzędzia, komendy `python` lub `python3`, w zależności od systemu operacyjnego. Narzędzie to pozwala uruchamiać pliki z rozszerzeniem `.py` i obserwować rezultaty ich działania, jak również pozwala pracować w trybie interaktywnym, odpowiadając na pytania użytkownika w koncepcji REPL (*Read-Evaluate-Print-Loop*).

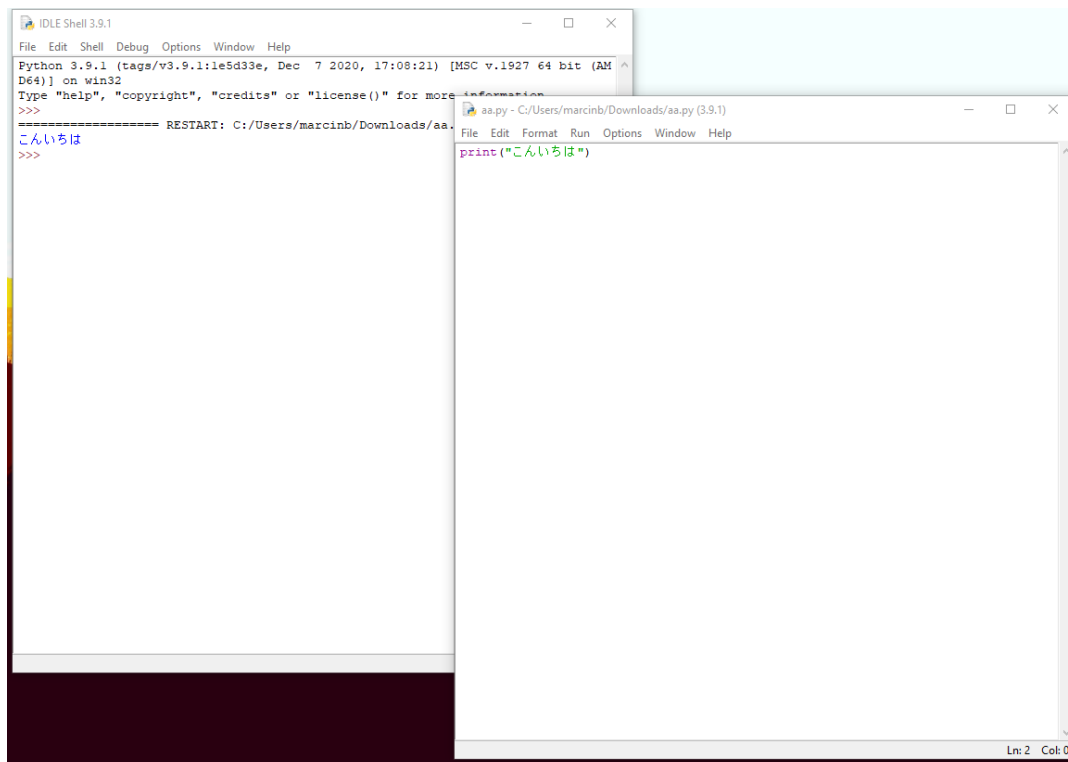
```
C:\>python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec 7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 1
>>> x
1
>>> for x in [1,2]:
...     print(x)
...
1
2
>>> exit()
```

Rys. 9.2. Przykład pracy w trybie interaktywnym.



Aby opuścić interaktywny tryb pracy, należy wydać komendę `exit()`.

W wersji dla środowisk graficznych dostępne jest również narzędzie IDLE, służące jako graficzny interpreter poprzez wykorzystanie okna „IDLE Shell” oraz prosty edytor skryptów w języku Python wraz z kolorowaniem składni i możliwością uruchomienia skryptu poprzez opcję *Run -> Run Module*.



Rys. 9.3. Środowisko IDLE.

Zadanie 9.3. Składnia języka Python

To, co w Pythonie jest jednym z najbardziej charakterystycznych wyróżników jest to, że sposób formatowania składni ma znaczenie. To znaczy fakt czy blok instrukcji jest wcięty (za pomocą spacji lub tabulatora) może oznaczać, czy jest on, czy nie, częścią innego bloku.

Oprócz tego język Python nie ma żadnego znaku końca linii poza znakiem nowej linii. To znaczy – nie ma średników ani kropek. Przykładowy kod funkcji w języku Python do rekurencyjnego obliczania silni:

```
def fac(n):  
    print('n =', n)  
    if n > 1:  
        return n * fac(n - 1)  
    else:  
        return 1
```

Jak można zaobserwować z zaprezentowanego kodu, bloki (np. `if`, `else`, `def`) zaczyna się znakiem dwukropka, a potem następuje wcięcie. W odróżnieniu od większości innych



języków programowania Python wymaga wcięć w blokach – tj. element jest traktowany jako należący do bloku wtedy i tylko wtedy, kiedy jest wcięty w stosunku do tego bloku. Oznacza to, że białe znaki mają znaczenie – ale nie dotyczy to np. spacji w wyrażeniach.

`def` oznacza definicję funkcji, jednak nie jest nigdzie zdefiniowany typ zwracany przez funkcję. Można również zaobserwować, że nie ma określonego typu zmiennej dla zmiennej `n`.

Zmienne będą miały typ wydedukowany (inferowany), ale w odróżnieniu od języka Haskell, typ nie jest dla danej zmiennej stały, ale można potem go zmienić (czyli w Pythonie jest typowanie dynamiczne), ale Python nie pozwoli np. na bezpośrednie dodanie do siebie liczby i litery (czyli jest typowanie silne), w odróżnieniu na przykład od języka JavaScript, który konwertuje typy „w locie”.

```
x = 1 # x będzie typu int

y = 1.1 # y będzie float
z = "ala" # z będzie str
a = False # a będzie bool

a = x + y # zadziała, bo da się automatycznie dodać liczby
b = y + z # nie zadziała, błąd wykonywania programu
```

Wykonując program z tego listingu możesz śmiało zauważyć, że błąd `TypeError` wystąpi dopiero w momencie, kiedy program uruchomi ostatnią linię – a co za tym idzie, mechanizm wykonywania jest typowy dla języków interpretowanych. Warto też zauważyć, że wbudowane słowa kluczowe `False` oraz `True` są wrażliwe na wielkość znaków.

Trzecia najważniejsza cecha języka Python, poza wrażliwością na wcięcia oraz dynamicznym typowaniem, to obiektowość – wszystkie typy w języku Python są obiektami, w odróżnieniu od np. języka Java i typów prymitywnych w niej obecnych.

Do zapewnienia rozgałęzień w programie używamy typowej instrukcji `if`. Można używać samego `if`, jak i `if/else`, jak i `if/elif/else` do zapewnienia różnych wersji odpowiedzi, jednak każde z nich wymaga stworzenia całego bloku.

Operator sprawdzania równości to `==`, a nierówności to `!=`, w odróżnieniu od Prologa; operatory logiczne `or`, `and` oraz `not` są zapisywane dokładnie takimi słowami, ale jedną z ciekawych cech jest możliwość porównywania kilku elementów naraz w jednym wyrażeniu, bez konieczności wykorzystania operatora `and`.

```
x = 3
1 < x < 5 # True
```

Zadanie 9.4. Konwersje pomiędzy typami zmiennych

Konwersja pomiędzy typami zmiennych może zachodzić poprzez wbudowane funkcje takie jak `str()`, `int()`, `bool()` i inne, odpowiednie dla wszystkich wymienionych typów. Po połączeniu z funkcją `print()`, można zaobserwować działanie funkcji:

```
print(str(1.1))
print(str(2))
print(int("2"))
print(float("2.1"))
```

Wyniki:

```
1.1
2
2
2.1
```

Wyniki jednak mogą nieco odbiegać od oczekiwań, zwłaszcza w przypadku konwersji na wartość logiczną:

```
print(bool(1))
print(bool(1.1))
print(bool(0))
print(bool(-1))
```

daje w wyniku:

```
True
True
False
True
```

Innymi słowy, każda liczba jest `True`, chyba że jest to liczba 0. Równie ciekawie jest w przypadku łańcuchów znaków, ponieważ następujący kod:

```
print(bool("ala ma kota"))
print(bool("True"))
print(bool("False")) # !!!
print(bool("0")) # !!!
print(bool(""))
```

Wyświetli następujące wyniki:

```
True
True
True
True
False
```

Jak widać z przedstawionych przykładów – każdy ciąg znaków, niezależnie od zawartości, jest konwertowany na wartość `True`, chyba, że jest to ciąg pusty.

Zadanie 9.5. Funkcje i parametry oraz parametry nazwane

Poznałeś również już funkcję `print()`, która pozwala na wyświetlanie tekstu na ekranie. Funkcja ta jednak może przyjmować wiele parametrów, w tym część określonych jako parametry nazwane, tj. takie, w których wymagane jest określenie nazwy parametru przy jego wywołaniu.

```
print("tekst", end=" ") # wyświetla tekst ale na zakończenie
print("tekst", end=" ") # nie znak nowej linii, ale spacja
```

Parametry nazwane można stosować do wszystkich parametrów, można również mieszać te konwencje:

```
def foo(a, b):
    return a + b

print(foo(1, 2))
print(foo(1, b=2))
print(foo(a=1, b=2))
print(foo(b=2, a=1))
```

Nie jest jednak możliwe użycie parametru nazwanego, a po nim parametrów pozycyjnych:

```
print(foo(b=2, 1))
# SyntaxError: positional argument follows keyword argument
```

Python dysponuje również bliźniaczą funkcją w stosunku do `print()`, funkcją `input()`, która pozwala pobrać od użytkownika tekst z konsoli, opcjonalnie podając komunikat prośby.

```
tekst = input("Podaj tekst: ")
```

Zadanie 9.6. Słowniki

Słownik to jedna z podstawowych struktur danych w Pythonie, odpowiadająca Hashtable w Javie albo tablicy asocjacyjnej w PHP. Pozwala na tworzenie struktur w postaci Klucz-Wartość, w których zarówno klucze, jak i wartości, nie muszą być takich samych typów w całym słowniku.

Słowniki definiuje się z wykorzystaniem operatora {}, na przykład:

```
d = { "klucz": "wartość", "innyklucz": 3 }
```

Odwołać się do określonego klucza można za pomocą prostego operatora indeksowania [], na przykład:

```
d = { "klucz": "wartość", "innyklucz": 3 }  
print(d["klucz"])
```

Można również modyfikować wartości pod określonym kluczem oraz dodawać nowe:

```
d["innyklucz"] = 33  
d["nowyklucz"] = "witaj"
```

Jeśli kluczami są łańcuchy znaków, wielkość liter ma oczywiście znaczenie, ponieważ aby operator indeksowania dopasował poszukiwany klucz do klucza w słowniku, oba obiekty muszą być dokładnie równoważne.

W języku Python większość typów złożonych, do których należą słowniki, są w gruncie rzeczy obiektami. Stąd też możliwe jest wykonywanie operacji na obiekcie słownika z wykorzystaniem operatora kwalifikowanego ., na przykład:

```
d = { "klucz": "wartość", "innyklucz": 3 }  
d.clear() # wyczyść słownik
```

Zadanie 9.7. Listy

Listy definiowane są za pomocą nawiasów kwadratowych. Są to uporządkowane struktury danych elementów, można odwoływać się do nich za pomocą indeksów. W odróżnieniu od języka Haskell, listy nie muszą być homogeniczne, tj. mogą zawierać elementy różnych typów i nie jest to żaden problem. Listy indeksowane są od 0.

```
la = [] # lista pusta  
li = ["a", "b", "Jon Snow", 3]  
print(li[3]) # wyświetli 3
```

Istotną ciekawostką jest jednak sam mechanizm indeksów, ponieważ operator [] pozwala na wybór elementu z indeksem ujemnym – zwracając element licząc „od tyłu”, gdzie ostatni element listy ma indeks -1.



```
li = ["a", "b", "Jon Snow", 3]
print(li[-3]) # wyświetli b
```

Jedną z ciekawych cech języka Python jest operator `:` wewnątrz operatora indeksowania `[]` który posłuży do tworzenia wycinków list, czyli pozwala na tzw. slicing. Slicing można używać z wykorzystaniem następującej składni:

```
Wynik = Lista[ start : koniec : krok ]
```

Wynik będzie nową listą utworzoną na podstawie istniejącej listy `Lista` począwszy od indeksu `start`, do indeksu `koniec - 1`, co `krok`.

Gdzie wszystkie z tych elementów mogą być opcjonalne, a w przypadku braku kroku można również pominąć drugi znak dwukropka.

```
li = ["a", "b", "Jon Snow", 3]

print(li[0:2]) # indeksy 0 i 1 -- ['a', 'b']
print(li[1:2]) # indeksy 1 i 1 -- ['b']

print(li[:3]) # indeksy od początku do 2 -- ['a', 'b', 'Jon Snow']
print(li[1:]) # indeksy od 1 do końca -- ['b', 'Jon Snow', 3]

print(li[0:3:2]) # indeksy 0, 2 -- ['a', 'Jon Snow']
```

Jak wspomniano wcześniej, struktury danych w Pythonie są obiektami, stąd też lista dysponuje szeregiem metod, takich jak:

- `append()` (dodaje nowy element na końcu),
- `insert()` (wstawia element we wskazanym miejscu),
- `extend()` (dodaje listę do listy),
- `remove()` (usuwa element na wskazanym indeksie z listy).
- `index()` (zwraca na którym indeksie listy występuje element podany w parametrze metody).

Używa się ich w stosunku do obiektu listy, na przykład: `li.insert(2, "test")`, jednak mogą zwrócić oczywiście błąd, na przykład funkcja `index()` zwraca `ValueError` jeżeli obiektu nie ma w liście.

```
li = ["a", "b", "Jon Snow", 3]

print(li.index("a")) # wyświetla 0

print(li.index(-3.14))
# rzucony jest błąd
# ValueError: -3.14 is not in list
```

Aby uchronić się przed tym i sprawdzić czy w ogóle obiekt istnieje w liście, można wykorzystać operator `in`.

```
li = ["a", "b", "Jon Snow", 3]

print("3" in li) # wyświetla False
print(3 in li) # wyświetla True
```

Usuwanie elementu z listy może być również realizowane przez metodę `pop()` która usuwa i zwraca pierwszy element listy. W powiązaniu z metodą `push()` można traktować dowolną listę jak stos (strukturę LIFO).

Pobieranie aktualnej długości listy realizowane jest jednak zupełnie inaczej – Python oferuje globalną funkcję `len()`, która zwraca długość dowolnej struktury danych, która umie „obliczyć” swoją długość (np. lista, słownik, generator, krotka, niektóre klasy). W podobny sposób lista również może zostać przekonwertowana na łańcuch znaków globalną funkcją `str()`. Z kolei inne struktury danych można przekonwertować do postaci listy z wykorzystaniem funkcji `list()`.

Ważnym szczegółem jest fakt, że listy są zmienne (mutowalne), a niektóre funkcje operujące na liście bezpośrednio ją modyfikują, a nie zwracają nową, zmodyfikowaną listę – w szczególności dotyczy funkcji takich jak `sort()` oraz `reverse()`.

```
lista = [2, 1, 3, 4]

lista.sort()
print(lista)

lista.reverse()
print(lista)
```

Program taki wyświetli oczywiście następujący wynik:

```
[1, 2, 3, 4]
[4, 3, 2, 1]
```

Aby posortować listę w taki sposób, aby została zwrócona nowa, posortowana lista, bez modyfikacji oryginału, należy wykorzystać funkcję `sorted()`.




```
lista = [2, 1, 3, 4]

lista2 = sorted(lista)
print(lista2)
```

Do funkcji `sorted()` można również przekazywać własną funkcję wybierającą klucz sortowania, co będzie istotne w niektórych przypadkach.

Zadanie 9.8. Krotki (tuple)

Jak się spodziewałeś, skoro wykorzystujemy operatory `{}` i `[]` do budowania odpowiednio słowników i list, również wykorzystywany jest operator `()` – tym razem, identycznie jak w przypadku języka Haskell, do budowy krotek (tuple).

```
k = (1, 2)
l = (1, "a")
m = (1, 2, False)
```

Krotki są niezmiennymi (niemutowalnymi) strukturami danych i mogą zawierać wiele elementów, różnych typów. Podstawową operacją na krotce będzie oczywiście jej dekonstrukcja na wiele zmiennych. W typowych językach funkcyjnych można to wykonywać wewnątrz mechanizmu pattern matching, w Pythonie dostępna jest również wersja imperatywna:

```
a, b = (1, "Daenerys")
print(a) # wyświetla 1
print(b) # wyświetla Daenerys
```

(można tutaj zwrócić uwagę, że ta sama konstrukcja może być również wykorzystana do dekonstrukcji listy)

Krotki przede wszystkim mogą być wykorzystywane do tworzenia funkcji, które zwracają obiekty opakowane w nie, przez co pojedyncza funkcja zwraca nie jedną, ale wiele wartości za pośrednictwem jednego `return`.

```
def f():
    return (2, 3)

x, y = f()
# funkcja efektywnie zwróciła x oraz y
```

Zadanie 9.9. Generator `range()` oraz pętla `for-in`

Funkcja `range()` zwraca generator – specjalny obiekt, po którym można iterować, którego zawartością są elementy od 0 do podanej wartości. Można go skonwertować na listę za pomocą funkcji `list()`.



```
list(range(3))  
# zwraca listę [0,1,2]
```

W jakim celu stosować funkcję `range()`? W odróżnieniu od wielu innych języków programowania, Python nie dysponuje pętlą `for`, która uruchamia się określoną liczbę razy. Można jednak wykorzystać pętlę `for-in`, służącą do przechodzenia po wszystkich elementach listy, w połączeniu z funkcją `range()`, do stworzenia pętli iterującej w określonych granicach:

```
for x in range(10):  
    print(x, '** 2 =', x*x)
```

Co przykładowo wyświetli:

```
0 ** 2 = 0  
1 ** 2 = 1  
2 ** 2 = 4  
3 ** 2 = 9  
4 ** 2 = 16  
5 ** 2 = 25  
6 ** 2 = 36  
7 ** 2 = 49  
8 ** 2 = 64  
9 ** 2 = 81
```

Dodatkowo, pętla `for-in` może być również wykorzystana do iterowania po dowolnej liście:

```
for i in [ 1, 2, 3 ]:  
    print(i)
```

Jak również po łańcuchu znaków, znak po znaku:

```
for i in "test":  
    print(i)
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz program typu FizzBuzz – program ma wyświetlać listę liczb od 1 do 100, ale zamiast liczb podzielnych przez 3 ma wyświetlić słowo „Fizz”, zamiast podzielnych przez 5 słowo „Buzz”, a zamiast podzielnych jednocześnie przez 3 oraz przez 5 – słowo „FizzBuzz”. Skorzystaj z operatora `mod` do sprawdzania podzielności.

Wynik działania programu powinien wyglądać następująco:



```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
...
97
98
Fizz
Buzz
```

Polecenie 2.

Napisz program, który iterując znak po znaku wyświetli każdą cyfrę liczby w postaci słownej, np. 110 wyświetli jako „jeden jeden zero”.

Polecenie 3.

Napisz program, który pozwoli na zamianę liczb arabskich (np. 123) na system rzymski (np. CXXIII). Skorzystaj z funkcji `input()`, która pozwala wczytać tekst od użytkownika oraz odpowiednich funkcji do konwersji. Możesz wykorzystać słowniki, aby powiązać liczby arabskie i odpowiadające im łańcuchy znaków w systemie rzymskim.

LABORATORIUM 10. PYTHON – ŁAŃCUCHY ZNAKÓW ORAZ PROGRAMOWANIE FUNKCYJNE.

Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego.

Cel laboratorium:

Zapoznanie z obsługą łańcuchów znaków w języku Python i importowaniem dodatkowych modułów. Poznanie możliwości wykorzystania operatorów list w stosunku do łańcuchów znaków.

Zakres tematyczny zajęć:

- łańcuchy znaków w języku Python, interpolacja łańcuchów,
- importowanie dodatkowych modułów i pojedynczych funkcji,
- listy zasięgowe,
- funkcje anonimowe, referencje do funkcji,
- funkcje wyższego rzędu, wykorzystanie map i filter,
- obsługa wartości losowych.

Pytania kontrolne:

1. Na czym polega zmienność i niezmiennosc łańcuchów znaków?
2. Czy operator indeksowania operuje na bajtach, czy na znakach?
3. Czy w Pythonie dostępny jest typ obsługujący pojedyncze znaki?
4. Do czego wykorzystywane są funkcje map oraz filter?
5. Czym jest funkcja lambda?

Zadanie 10.1. Obsługa łańcuchów znaków w języku Python

Łańcuchy znaków w Pythonie opatrywane są apostrofami lub cudzysłowami, i nie ma znaczenia który znak zostanie wykorzystany - nie powoduje to zmian w analizie łańcuchów, nie rozróżnia pojedynczych znaków – w Pythonie wszystko jest typu `str`, w odróżnieniu od innych języków programowania rozróżniających znaki typu `char` od łańcuchów, czyli typu `string`.

Możliwe jest użycie potrójnych apostrofów lub cudzysłowów, aby stworzyć wielolinijkowe łańcuchy znaków, na przykład:



```
test = """wielolinijkowy
string z
nowymi liniami"""
```

Ciekawostką języka Python jest nieobecność wielolinijkowych komentarzy w kodzie – w tej roli, zwłaszcza w przypadku komentarzy dokumentujących zastosowanie funkcji stosuje się właśnie wielolinijkowe łańcuchy znaków, które nie są przypisywane do żadnej zmiennej, co automatycznie powoduje, że interpreter je będzie ignorował.

```
1 def foo():
2     """
3     Ta funkcja zwraca zawsze 4
4     """
5     (function) foo: () -> Literal[4]
6     Ta funkcja zwraca zawsze 4
7     foo()
8
```

Rys. 10.1. Widok komentarza działającego jako dokumentacja w edytorze kodu.

W łańcuchach znaków dostępne są również sekwencje ucieczki takie jak `\n` lub `\t`. Począwszy od Pythona 3.6 można również korzystać z mechanizmu interpolacji łańcuchów, automatycznie wstawiając wartości zmiennych do środka łańcucha znaków, za pośrednictwem operatora `f` (tzw. `f-string`):

```
x = 1.73
y = 17
z = "Jon Snow"

s = f"{z} w Tańcu ze Smokami ma {x} m wzrostu i {y} lat"
print(s)
```

Wynikiem działania tego programu jest tekst:

```
Jon Snow w Tańcu ze Smokami ma 1.73 m wzrostu i 17 lat
```

Jest to bardzo popularny mechanizm stosowany również w języku C# za pomocą operatora `$` czy w języku JavaScript poprzez tworzenie łańcuchów z wykorzystaniem znaków ```.

Zadanie 10.2. Dodawanie modułów do skryptu języka Python

Poprzez słowo kluczowe `import` można dołączyć do programu dodatkowe moduły, będące luźnym odpowiednikiem bibliotek. Na przykład bibliotekę `string`, która dostarcza kilka istotnych stałych zawierających różnego typu znaki alfanumeryczne.



```
import string

print(string.ascii_letters) # abcdefghijklmno...IJKLMNOPQRSTUVWXYZ
print(string.ascii_lowercase) # abcdefghijklmnopqrstuvwxyz
print(string.ascii_uppercase) # ABCDEFGHIJKLMNOPQRSTUVWXYZ
print(string.digits) # 0123456789

print(string.punctuation) # '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
print(string.printable) # digits + letters + punctuation + white
print(string.whitespace) # space + tab + linefeed + return + ...
```

Jak widzisz, poprzez zaimportowanie z wykorzystaniem słowa kluczowego `import`, uzyskujemy dostęp do obiektu o nazwie identycznej z nazwą modułu. W podobny sposób będzie można zaimportować moduł `random` i korzystać z jego funkcji jakby były statycznymi składowymi klasy:

```
import random
random.randint(0, 100) # liczba losowa z zakresu 0-100
random.choice([1, 2, 3, 4]) # losowy element kolekcji
```

Czasami jednak chcielibyśmy zaimportować funkcje bezpośrednio do globalnej przestrzeni nazw, aby uruchamiać je jak dowolne inne globalne lub lokalne funkcje – do tego wykorzystać można składnię `from ... import ...`.

```
from random import randint

def foo(x):
    return x * 2

r = randint(1, 100) # uwaga - liczba losowa z zakresu od 1 do 100
print(foo(r))
```

Możliwe jest również zaimportowanie wszystkich funkcji z modułu poprzez `from ... import *`, jednak jest to niezalecane z uwagi na potencjalne konflikty nazw funkcji.

Zadanie 10.3. Operacje na łańcuchach

W podobny sposób jak w przypadku Haskella, łańcuchy znaków i listy są traktowane bardzo podobnie – co za tym idzie, możliwe jest wykorzystywanie operatora `[]` oraz mechanizmu slicing do tworzenia nowego tekstu na podstawie istniejącego tekstu.



```
s = "abcdef"
print(s[1:7:2]) # bdf
```

Ważnym szczegółem jednak będzie to, że nie istnieje, jak wspomniano wcześniej, typ danych obejmujący pojedynczy znak – stąd operacja sprawdzenia typu z wykorzystaniem operatora `type()`:

```
s = "abcdef"
x = s[3]

print(type(x))
```

wyświetli oczywiście `<class 'str'>`.

Drugim ważnym szczegółem jest próba modyfikacji zawartości łańcucha znaków:

```
s = "abcdef"
s[3] = "x"
```

Nie powiedzie się ona, wraz z komunikatem błędu:

```
TypeError: 'str' object does not support item assignment.
```

Oznacza to, że łańcuch znaków, w odróżnieniu od listy, jest strukturą niezmienną (niemutowalną). Aby móc zmienić pojedynczy znak w łańcuchu, najlepsze będzie wykorzystanie mechanizmu tworzenia łańcuchów poprzez operatory zasięgowe, na przykład:

```
s = "abcdef"

# chcemy zmienić znak pod indeksem 3 na x
nowy = s[:3] + "x" + s[4:]

print(nowy) # abcxef
```

Wspominaliśmy na początku o niekompatybilności pomiędzy językiem Python w wersji 2 oraz wersji 3. Jedną z przyczyn tej niekompatybilności jest fakt, że wszystkie łańcuchy znaków w Python 3 są oparte o kodowanie UTF-8, pozwalając na przechowywanie prawie wszystkich znaków ze sposobów pisma używanych na Ziemi.

Powoduje to pytanie – czy operator indeksowania dla łańcucha znaków odnosi się do znaków, czy do bajtów? W jaki sposób zadziała w przypadku znaków wielobajtowych, na przykład pochodzących z innych systemów pisma niż alfabet łaciński?

```
s = "平仮名"
print(s[2])
```

Taki kod wyświetli oczywiście tylko znak 名 – operator indeksowania operuje na znakach, nie na pojedynczych bajtach. Jak widzisz, nie wymaga również niczego specjalnego, aby korzystać ze wszystkich znaków UTF-8 w twoich programach, w zależności jednak od systemu kodowania znaków w konsoli, używanego systemu operacyjnego i używanego emulatora terminala wyniki mogą być nieoczekiwane. Warto tutaj zauważyć, że część z tych elementów poprawnie działa na platformie Windows dopiero od Pythona 3.6.

The image shows two terminal windows side-by-side. The top window is a Windows Command Prompt with a dark background. It shows the execution of a Python script: `C:\Users\marcinb>C:/Python3... 1 s = "平仮名" 2 y = "zażółć gęślą jaźń 🐱" 3 4 print(s[2]) 5 print(y)`. The output is `名` followed by `zażółć gęślą jaźń 🐱`. The bottom window is an Ubuntu terminal (16.04 LTS) with a dark background. It shows the same script being executed: `C:\Users\marcinb>C:/Python39/python.exe c:/Users/marcinb/Downloads/I12.py`. The output is `名` followed by `zażółć gęślą jaźń 🐱`. The difference in the second string's output is due to different terminal encodings.

Rys 10.2. Różne wyniki działania tego samego programu w różnych emulatorach terminala

Do porównywania ze sobą zawartości łańcuchów można wykorzystać operator `==`, który je ze sobą porówna pod względem treści. Wynika z tego kolejny aspekt języka Python – możliwe jest takie wykorzystanie operatorów, aby różnie działały dla różnych klas, co za tym idzie – w Pythonie dostępny jest mechanizm przeciążania operatorów.

Jak wspomniano wcześniej, łańcuchy są obiektami, co oznacza, że można na nich wykonywać metody takie jak `lower()` (zmienia wielkie litery na małe) `upper()` (zmienia małe litery na wielkie), `replace()` (zmienia jeden podciąg na inny łańcuch) albo `strip()` (usuwa białe znaki z początku i końca), zauważ jednak, że z powodu niemutowalności łańcuchów tekstu, metody te zwracają zawsze nowy łańcuch znaków, nie modyfikując

istniejącego. Odpowiednie metody można również uruchamiać bezpośrednio na stałych, nie tylko na zmiennych.

Możesz również zadać sobie pytanie – w jaki sposób zadziałają funkcje `lower()` czy `upper()` dla znaków, które nie mają swojego „małego” lub „wielkiego” odpowiednika. Oczywiście, znaki zostaną zwrócone bez modyfikacji:

```
print("カタカナ123".lower()) # wyświetla カタカナ123
```

Dostępne są również metody:

- `count()` liczy ile razy string z parametru występuje w stringu na którym jest wykonywana,
- `startswith()` sprawdza czy zaczyna się określonym tekstem,
- `endswith()` analogicznie – ale czy kończy,

Do wyszukiwania elementów w liście można było wykorzystać metodę `index()`, która działa również dla łańcuchów znaków, oraz dla poszukiwania podłańcuchów wewnątrz łańcucha znaków:

```
s = "ala ma kota"  
print(s.index('ma')) # wyświetli 4
```

W przypadku łańcuchów znaków, w odróżnieniu od list, oferowana jest również metoda `find()`, która znajduje pozycję tekstu w tekście – lub zwraca `-1`, a nie kończy się błędem w przypadku nieznaalezienia.

Zadanie 10.4. Wyrażenia generatorowe

Wyrażenia generatorowe mają ogólną postać:

```
(<funkcja> for <parametr> in <zbiór>)
```

Wyrażenie takie zwraca generator, czyli obiekt, który zwraca kolejny swój rezultat dopiero wtedy, kiedy zostanie o niego poproszona, np. przez pętlę `for-in`, tj. wykorzystywana jest leniwa ewaluacja. Jest to równoznaczne z funkcjami wykorzystującymi operator `yield`. W poprzednim laboratorium jedna taka funkcja została już wcześniej zademonstrowana – jest to oczywiście funkcja `range()`.

```
def lazyf():  
    for i in range(5):  
        yield i
```

Najbardziej istotną jednak kwestią jest możliwość wykorzystania wyrażeń generatorowych do tworzenia list, poprzez konwersję do listy z wykorzystaniem albo funkcji `list()` albo operatora tworzenia nowej listy `[]`.



```
[i for i in range(5)] # [0, 1, 2, 3, 4]
```

Możliwe jest również w ten sposób tworzenie list z wykorzystaniem bardziej skomplikowanych funkcji:

```
a = [1, 2, 3, 4]
print([i ** 2 for i in a])
# [1, 4, 9, 16]
```

Wykorzystanie instrukcji warunkowej `if` następuje jednak po `for`:

```
[x for x in range(1,5) if x % 2 == 0] # [2, 4]
```

Zadanie 10.5. Wyrażenia lambda i przypisywanie funkcji do zmiennych

Funkcje anonimowe zapisywane z wykorzystaniem wyrażenia lambda można zapisać poprzez użycie słowa kluczowego lambda:

```
lambda x: x * 2
```

i jest to równoznaczne z funkcją:

```
def funkcja(x):
    return x * 2
```

jednak oczywiście funkcja taka pozbawiona jest nazwy.

Elementy funkcyjne oczywiście mogą pozwalać na zapisywanie funkcji do zmiennych, a co za tym idzie – tworzenia funkcji wyższego rzędu:

```
def ff(f, x): # funkcja przyjmuje funkcję i liczbę
    return f(f(x)) # i uruchamia ją raz oraz na jej wyniku

funkcja = lambda x : x ** 2 # zapis funkcji lambda do zmiennej
liczba = 2

wynik = ff(funkcja, liczba)
print(wynik) # 16
```

Oczywiście należy pamiętać, że użycie `()` skutkuje wywołaniem funkcji, stąd poniższy zapis jest różny w działaniu:



```
def funkcja():
    return 4

x = funkcja
print(x)
y = funkcja()
print(y)
```

i wyświetli następujący rezultat:

```
<function funkcja at 0x000001B48717F040>
4
```

Zadanie 10.6. Funkcje map(), filter() oraz reduce() w Pythonie

Jak większość języków programowania z elementami funkcyjnymi, Python również pozwala na korzystanie z funkcji wyższego rzędu `map()` i `filter()` w celu operacji na kolekcjach. Obydwie te funkcje znajdują się w standardowej bibliotece i są dostępne w globalnej przestrzeni nazw.

Funkcja `map()` uruchamia przekazaną jej w pierwszym parametrze funkcję na każdym elemencie przekazanej jej w drugim parametrze listy.

```
map(lambda x: x * 2, [1, 2, 3, 4]) # [2, 4, 6, 8]
```

Funkcja `filter()` wybiera z kolekcji elementów tylko te dla których funkcja przekazana jako pierwszy parametr (predykat) zwróciła `True`.

```
filter(lambda x: x % 2 == 0, [1, 2, 3, 4]) # [2, 4]
```

Jak pamiętasz, Haskell dysponował również funkcją `fold()`. Tutaj jej odpowiednikiem będzie funkcja `reduce()`, jednak nie jest ona obecna w standardowej bibliotece, należy ją zaimportować, na przykład z wykorzystaniem konstrukcji `from ... import ...`:

```
from functools import reduce
reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) # (((((1+2)+3)+4)+5) = 15
```

Funkcja `reduce()` uruchamia przekazaną jej jako pierwszy argument funkcję dwuargumentową dla wszystkich elementów kolekcji, od lewej do prawej (jak `foldl`), tworząc na końcu jedną wartość.

Oczekuje ona, że pierwszy parametr będzie funkcją przyjmującą dwa parametry. Pierwszy z nich będzie wartością akumulowaną (akumulatorem), a drugi – będzie przyjmował wartości kolejnych elementów kolekcji.

Opcjonalny trzeci argument, inicjalizator, to będzie wartość początkowa obliczeń, jeżeli nie jest używany, pierwszy element kolekcji jest wykorzystywany jako wartość początkowa (jak w Haskellowej `foldl1`).

Możesz zauważyć, że funkcje `map()` i `filter()` duplikują rzeczy, które da się osiągnąć poprzez wyrażenia generatorowe i *list comprehensions*. W Pythonie preferowane jest używanie tych drugich.

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz funkcję, która odwróci podany przez użytkownika tekst i wykorzystaj ją do sprawdzenia czy tekst podany od użytkownika jest palindromem (jest taki sam od przodu, jak i od tyłu).

Polecenie 2.

Napisz pojedynczą funkcję, która zaszyfruje tekst złożony tylko z wielkich liter z wykorzystaniem szyfru Cezara. Wykorzystaj podejście funkcyjne.

Polecenie 3.

Napisz program, który wygeneruje losowy numer rejestracyjny samochodu w powiecie m. Lublin (LU XXXXX), gdzie X jest wielką literą lub cyfrą.

Polecenie 4.

Napisz program, który zamieni liczbę w postaci rzymskiej na system arabski.



LABORATORIUM 11. PYTHON – PROGRAMOWANIE OBIEKTOWE.

Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego.

Cel laboratorium:

Zapoznanie z mechanizmami obiektowości w języku Python. Nauka wykorzystania dynamicznie dodawanych atrybutów oraz obiektu self. Zademonstrowanie mechanizmu przeciążania operatorów.

Zakres tematyczny zajęć:

- klasy i obiekty w języku Python,
- obiekt self,
- składowe klasy, składowe statyczne,
- metody prywatne w języku Python,
- metody „magiczne”.

Pytania kontrolne:

1. W jaki sposób Python pozwala na obsługę składowych statycznych?
2. Czy Python pozwala na dynamiczne dodawanie atrybutów obiektu?
3. Jak w Pythonie realizowana jest enkapsulacja?
4. W jaki sposób objawia się działanie polimorfizmu w programowaniu obiektowym w Pythonie?

Zadanie 11.1. Klasy i obiekty w języku Python

Jak wspomniano wcześniej, Python jest językiem obiektowym. Co za tym idzie, możliwe jest tworzenie własnych klas i ich hierarchii dziedziczenia, polimorfizm i enkapsulacja. Przykładowa klasa może wyglądać następująco:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f():
        return 'hello world'
```

Stworzenie instancji klasy (obektu) wymaga użycia następującej konstrukcji:



```
x = MyClass()
```

Okazuje się jednak, że nawet jeżeli klasa nie mówi nic o istnieniu pola w klasie, to można takie utworzyć w obiekcie dynamicznie:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f():
        return 'hello world'

x = MyClass()
x.data = "ala"

print(x.data) # wyświetla "ala"
```

Jak widać, obiekty mogą mieć dodawane atrybuty (ang. *data attributes*) w sposób całkowicie dynamiczny, tworzone są w momencie ich pierwszego użycia, jak zmienne lokalne.

Możliwe jest również stworzenie klasy całkowicie pozbawionej jakichkolwiek cech i poleganie na atrybutach danych – wymaga to jednak użycia specjalnego słowa kluczowego `pass`, które pozwala tworzyć puste bloki.

```
class My:
    pass

x = My()
x.attr = 0
x.attr_b = "test"
```

Porównaj teraz, co się stanie w takim wypadku:

```
class MyClass:
    i = 123

x = MyClass()
print(x.i)
print(MyClass.i)
```

Program wyświetli:

```
123
123
```

W przypadku jednak, kiedy spróbujemy zmodyfikować tę wartość, modyfikuje wartości we wszystkich istniejących obiektach:

```
class MyClass:
    i = 123

x = MyClass()
print(x.i) # 123
print(MyClass.i) # 123

MyClass.i = 456
print(x.i) # 456

y = MyClass()
print(y.i) # 456
```

Modyfikacja jednak w jednym obiekcie, nie wpływa na kolejne obiekty:

```
class MyClass:
    i = 123

x = MyClass()
print(x.i) # 123
print(MyClass.i) # 123

x.i = 456
print(x.i) # 456

y = MyClass()
print(y.i) # 123
```

W pierwszym przykładzie istniała również metoda `f()`, co jednak się stanie gdy spróbujemy uruchomić taki kod:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f():
        return 'hello world'

x = MyClass()
print(MyClass.f())
print(x.f())
```

Okazuje się, że wywołanie metody `f()` jakby była metodą klasy a nie obiektu (metodą statyczną) działa, natomiast wywołanie jej samej na obiekcie – powoduje błąd:

```
TypeError: f() takes 0 positional arguments but 1 was given
```

Pojawi się zatem pytanie – jaki to argument został przekazany do funkcji, która oczekuje ich 0?

Zadanie 11.2. Obiekt `self`

Okazuje się, że Python realizuje wywołanie metod na obiektach w bardzo specjalny sposób, przekazując wskaźnik do obiektu który wywołuje metodę do niej samej. Oznacza to automatycznie, że każda składowa która ma zostać wykorzystana „na obiekcie”, a nie „na klasie” musi przyjmować jeden parametr – i nazywa go się zazwyczaj `self`.

```
class MyClass:
    def f(self):
        return 'hello world'

x = MyClass()
print(x.f()) # wyświetli "hello world"
```

Ten sam mechanizm można również wykorzystać na przykład do stworzenia konstruktora:

```
class MyClass:
    def __init__(self, data):
        self.data = data

x = MyClass(3)
print(x.data) # wyświetla 3
```

Oczywiście, metody klasowe oraz metody z konkretnych obiektów mogą być również zapisywane do zmiennych i przekazywane dalej w stylu funkcyjnym:




```
class MyClass:
    def __init__(self, seed):
        self.seed = seed

    def random(self):
        return self.seed + 4

x = MyClass(4)
y = x.random

print(y)
print(y())
```

wyświetli to rzecz jasna:

```
<bound method MyClass.random of <__main__.MyClass object at
0x00000216FA9DDBE0>>
8
```

Co oznacza, że została rzeczywiście uruchomiona metoda random z konkretnego obiektu klasy MyClass.

Dodatkowo, dostępne są dwie specjalne adnotacje (w Pythonie zwane dekoratorami): @staticmethod oraz @classmethod. Pozwalają one zmienić metodę na typową metodę statyczną oraz na tzw. metodę klasy. Metoda klasy jako swój pierwszy argument podczas wywołania dostaje automatycznie klasę, na przykład:

```
class MyClass:
    @classmethod
    def foo(cls, a, b):
        print(cls)
        print(a)
        print(b)

x = MyClass()
x.foo(1, 2)
```

Wyświetla:

```
<class '__main__.MyClass'>
1
2
```

Z kolei @staticmethod zmusza metodę do zachowywania jak się jak typowa metoda statyczna, pozwalając na uruchamianie jej zarówno na obiekcie, jak i na klasie:

```
class MyClass:
    @staticmethod
    def foo(a, b):
        print(a)
        print(b)

x = MyClass()
x.foo(1, 2)
MyClass.foo(2, 3)
```

Zadanie 11.3. Dziedziczenie

Podobnie jak w innych językach z paradygmatem programowania obiektowego, dość podstawową kwestią staje się dziedziczenie. W przypadku Pythona, zadziała to w sposób całkowicie oczekiwany:

```
class Animal:
    def brr(self):
        return "brr"

    def make_sound(self):
        return ""

class Dog(Animal):
    def make_sound(self):
        return "woof"

d = Dog()
print(d.brr()) # wyświetli "brr"
print(d.make_sound()) # wyświetli "woof"
```

Możemy również w klasach pochodnych odwoływać się do metod i atrybutów klas bazowych poprzez funkcję `super`.

```
class Animal:
    def make_sound(self):
        return "Animal does brr"

class Dog(Animal):
    def make_sound(self):
        return super().make_sound() + " & woof"

d = Dog()
print(d.make_sound()) # Animal does brr & woof
```

Jak widzisz z przedstawionych przykładów, przeciążanie metod nie wymaga specjalnego ich traktowania – wszystkie metody są automatycznie oznaczone jako wirtualne. Warto zwrócić również tutaj uwagę, że Python, podobnie jak C++, pozwala na dziedziczenie z wielu klas bazowych, inaczej niż na przykład język Java.

Zadanie 11.4. Składowe prywatne

W Pythonie nie do końca dostępna jest koncepcja prywatnych składowych – ile istnieje możliwość, aby nazwać atrybut z wykorzystaniem dwóch znaków podkreślenia jako prefiks i spowoduje to wyświetlanie błędu `AttributeError` w przypadku odwołania do niego z zewnątrz. Przykładowy program:

```
class MyClass:
    def __init__(self):
        self.__data = 3

    def show(self):
        return self.__data

x = MyClass()
print(x.show())
print(x.__data)
```

Wyświetli 3, a następnie wystąpi błąd `AttributeError`:

```
AttributeError: 'MyClass' object has no attribute '__data'
```

Jednak wystarczy wykorzystać specjalną konstrukcję `_NazwaKlasy`, aby uzyskać dostęp do „prywatnej” składowej:



```
class MyClass:
    def __init__(self):
        self.__data = 3

    def show(self):
        return self.__data

x = MyClass()
print(x.show())
print(x._MyClass__data)
```

Powyższy kod wyświetla już dwa razy liczbę 3. W jaki sposób to działa? Po prostu każda składowa oznaczona dwoma znakami podkreślenia zwyczajnie zmienia swoją nazwę na taką posiadającą wspomniany prefiks w czasie wykonania programu.

Taka sama konstrukcja jest wykorzystywana i działa w przypadku metod – można wyjść zatem z założenia, że prefiks `__` służy raczej do ochrony programisty przed przypadkowymi błędami niż przed celowymi próbami zmiany koncepcji działania programu.

Zadanie 11.5. Magiczne metody

Istnieje szereg specjalnych składowych, których nazwy zaczynają się i kończą znakami `__`. Nazywane są one magicznymi metodami. Mogą one posłużyć do stworzenia konstruktora, jak zaprezentowano w przykładach wcześniej, ale również między innymi do stworzenia przeciążania operatorów. Na przykład możliwe jest zbudowanie takich obiektów:

```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname

    def __eq__(self, o):
        return self.name == o.name and self.surname == o.surname
```

Zaprezentowana magiczna metoda `__eq__` służy do implementacji operatora równości:

```
j = Person("Jon", "Snow")
j2 = Person("Jon", "Snow")
d = Person("Daenerys", "Targaryen")

print(j == d) # zwraca False
print(j2 == j) # zwraca True
```

W podobny sposób można stworzyć własne implementacje metod takich jak `__str__` (konwersja do typu `str`), `__add__` (dodawanie obiektów), `__lt__` (mniejszy) i tym

podobne. Pozwalają na to, aby uczynić klasy zgodnymi z szeregiem wbudowanych funkcji takich jak `repr()` czy `abs()`.

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Stwórz klasę `Geopoint`, która będzie przechowywać długość i szerokość geograficzną. Jej konstruktor powinien przyjmować te współrzędne i zapisywać do odpowiednich atrybutów obiektu. Zaimplementuj operator równości dla tej klasy oraz konwersję do string, która powinna zwracać tekst w następującym formacie:

```
{ lat: szerokość, lon: długość }
```

Polecenie 2.

Napisz program, który pozwoli na wyświetlenie posortowanej listy punktów geograficznych z wykorzystaniem klasy `Geopoint`. Posortuj punkty względem odległości od punktu o współrzędnych 51.21167 (szerokość geograficzna, *latitude*), 22.52222 (długość geograficzna, *longitude*).

Aby obliczyć odległość między dwoma punktami geograficznymi możesz wykorzystać następującą metodę:

```
def haversine(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2,
lat2])
    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon /
2)**2
    c = 2 * asin(sqrt(a))
    # Radius of earth in kilometers is 6371
    meters = 6371000 * c
    return meters
```

Aby móc skorzystać z tej funkcji musisz zaimportować funkcje z modułu `math`.

```
from math import radians, cos, sin, asin, sqrt
```

Uwaga: funkcja przyjmuje parametry w kolejności: szerokość1, długość1, szerokość2, długość2.



Przykładowe dane:

51.21782, 22.54583
51.21353, 22.54142
51.21483, 22.52527
51.22352, 22.55640

Poprawny wynik:

51.21483, 22.52527
51.21353, 22.54142
51.21782, 22.54583
51.22352, 22.55640

LABORATORIUM 12. PYTHON – ADNOTACJE TYPÓW I KLASY DANYCH.

Elementy programowania w języku Python. Składnia języka, podprogramy, typowanie dynamiczne, struktury danych (słownik, lista, krotka), operacje na łańcuchach znaków, listy zasięgowe. Elementy programowania obiektowego i funkcyjnego.

Cel laboratorium:

Zapoznanie studentów z nowymi elementami dostępnymi w najnowszych wersjach języka Python. Przygotowanie do stosowania adnotacji typów w tworzonych aplikacjach. Przedstawienie koncepcji klas danych i zamrożonych instancji.

Zakres tematyczny zajęć:

- adnotacje typów w języku Python,
- klasy danych i zamrożone instancje,
- wyrażenia przypisujące (ang. *walrus operator*).

Pytania kontrolne:

1. Czy Python obsługuje statyczne typowanie?
2. Jak adnotuje się typy zmiennych?
3. W jaki sposób adnotuje się zmienne jako niezmiennie?
4. Na czym polega niezmiennosc (niemutowalność) zamrożonych instancji?
5. Jakimi narzędziami można statycznie sprawdzać poprawność kodu Pythona pod względem typów?

Zadanie 12.1. Adnotacje typów

Jak wspomniano wcześniej, Python pozwala na dynamiczną zmianę typów zmiennych, także tych które są parametrami lub wynikami działania funkcji. Typowanie statyczne ma jednak przewagę nad typowaniem dynamicznym, zazwyczaj w mechanizmie sprawdzania poprawności typów przed uruchomieniem lub kompilacją programu, aby zapobiec błędom czasu wykonywania.

W Pythonie 3.5 wprowadzone zostały adnotacje typów pozwalające na określenie, jakiego typu będą parametry funkcji oraz jej wynik. Ma to ogólną postać:

```
def foo(parametr : typ) -> typ_zwracany:  
    pass
```

Przykładowo, można zadeklarować funkcję, która ma przyjąć dwie liczby całkowite i zwrócić liczbę całkowitą:



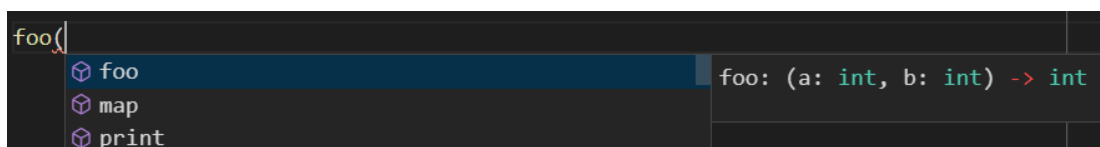
```
def foo(a : int, b : int) -> int:  
    return a + b  
  
print(foo(1, 2)) # wyświetli 3
```

Co jednak się stanie, kiedy spróbujemy uruchomić taką funkcję z parametrami o typach nie pasujących?

```
print(foo("a", "b")) # wyświetli "ab"
```

Środowisko uruchomieniowe nie wymaga i nie sprawdza adnotacji typów i nie będzie generować błędów ani ostrzeżeń z nimi związanych.

Niemniej mogą one być wykorzystywane przez narzędzia firm trzecich, w szczególności środowiska programistyczne oraz lintery.



Rys. 12.1. Przykład podpowiedzi typów w edytorze kodu, na podstawie adnotacji typów.

Jednym z narzędzi wykorzystywanych do sprawdzenia poprawności programu pod względem typów jest `mypy`, który możesz zainstalować w swoim systemie korzystając z menedżera pakietów Pythona, `pip`, wydając komendę:

```
pip install mypy
```

Od tej pory możesz wydać komendę `mypy` w konsoli, przekazując jej nazwę pliku `.py`, aby sprawdzić jej poprawność pod względem wykorzystania typów. Uruchamiając test na pliku z poprzedniego przykładu zobaczysz komunikaty o błędach:

```
l12.py:5: error: Argument 1 to "foo" has incompatible type "str"; expected "int"  
l12.py:5: error: Argument 2 to "foo" has incompatible type "str"; expected "int"  
Found 2 errors in 1 file (checked 1 source file)
```



```

File Edit Selection View Go Run Terminal Help
l12.py - Visual Studio Code

C: > Users > marcinb > Downloads > l12.py > foo
1 def foo(a : int, b : int) -> int:
2     return a + b
3
4 print(foo(1, 2)) # wyświetli 3
5 print(foo("a", "b")) # wyświetli "ab"

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: cmd
C:\Users\marcinb\Downloads>mypy l12.py
l12.py:5: error: Argument 1 to "foo" has incompatible type "str"; expected "int"
l12.py:5: error: Argument 2 to "foo" has incompatible type "str"; expected "int"
Found 2 errors in 1 file (checked 1 source file)

C:\Users\marcinb\Downloads>

Python 3.9.1 64-bit 0 0 Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python

```

Rys 12.2. Przykład działania narzędzia mypy.

Moduł `typing` jest wykorzystywany do lepszego opisywania typów, zwłaszcza typów złożonych. Możesz zaimportować z niej takie klasy jak `List`, `Dict` czy `Tuple`, aby opisać, że funkcja przyjmuje lub zwraca listy, słowniki lub krotki.

```

from typing import List

def howmany2(xs: List[int]) -> int:
    """Funkcja zwraca ile jest elementów równych
    2 w przekazanej liście xs"""
    return len(filter(lambda x : x == 2, xs))

```

W podobny sposób można opisywać słowniki:

```

from typing import Dict

def print_name_and_grade(grades: Dict[str, float]) -> None:
    for student, grade in grades.items():
        print(student, grade)

```

Możliwe jest wykorzystanie `Optional` aby opisać, że pewna zmienna może być pewnego określonego typu lub być równa `None`, z kolei `Union` pozwoli na określenie, że zmienna będzie mogła być jednego typu spośród kilku.

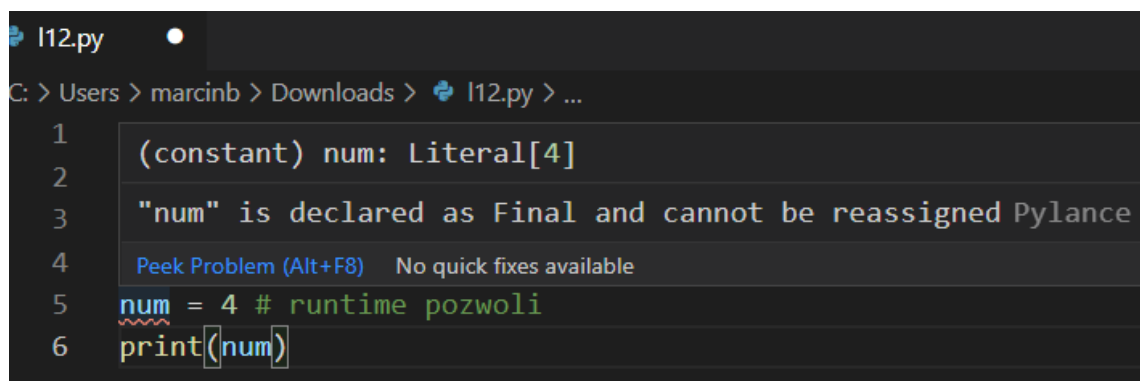
Wreszcie ten sam mechanizm może być stosowany także do opisu pojedynczych zmiennych:

```
x : int = "3"
print(x)
```

Jedną z najistotniejszych cech modułu `typing` wprowadzoną w Pythonie 3.8 jest także typ `Final`. Zmienne oznaczone z jego wykorzystaniem stają się niezmiennie po ustaleniu początkowej wartości – i automatyczne narzędzia takie jak `mypy` czy `PyLance` będą respektować ten zapis.

```
from typing import Final

num : Final[int] = 3
print(num)
num = 4 # runtime nie sprawdza czy wolno wykonać taką operację
print(num)
```



Rys 12.3. Przykład komunikatu narzędzia PyLance.

```
C:\Users\marcinb>mypy Downloads\l12.py
Downloads\l12.py:5: error: Cannot assign to final name "num"
Found 1 error in 1 file (checked 1 source file)
```

Rys 12.4. Przykład komunikatu narzędzia mypy.

Mechanizm `Final` jest bardzo podobny do stałych tworzonych z wykorzystaniem konstrukcji `const` w JavaScript lub różnicy pomiędzy `val` i `var` np. w języku Kotlin.

Zadanie 12.2. Klasy danych (data classes)

W Pythonie w wersji 3.7 zostały wprowadzone adnotacje (dekoratory) `@dataclass`, które ułatwiają tworzenie obiektów, automatycznie generując magiczne metody do tworzonych przez użytkownika klas. Wymaga to zaimportowania modułu `dataclasses` i wykorzystania dekoratora.



```
from dataclasses import dataclass

@dataclass
class Person:
    name : str
    surname : str
```

W przedstawionym powyżej przykładzie nie jest konieczne tworzenie oddzielnego konstruktora, zostanie on utworzony automatycznie, między innymi razem z magicznymi metodami `__repr__` oraz `__str__`, a także operatorami porównań, pozwalając na uproszczenie tworzonego kodu.

```
x = Person("Kiana", "Kaslana")
print(x) # wyświetli Person(name='Kiana', surname='Kaslana')
```

Mechanizm `@dataclass` działa wyszukując w opisanej nim klasie zmienne o jawnie podanym typie. Sam moduł `dataclasses` udostępnia również szereg funkcji, takich jak `astuple()` lub `asdict()` pozwalając konwertować pomiędzy obiektami a krotkami lub słownikami.

```
x = Person("Kiana", "Kaslana")
print(dataclasses.astuple(x)) # wyświetli ('Kiana', 'Kaslana')
```

W innych językach programowania pojęcia *data class* bardzo często związane jest nie tylko z automatycznym generowaniem konstruktorów czy użytecznych operatorów, ale również z niezmiennością obiektów. W Pythonie nie jest możliwe stworzenie całkowicie niezmiennych obiektów, ale można to zasymulować przekazując do dekoratora parametr `frozen` o wartości `True`, tworząc tzw. zamrożone instancje klasy.

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Person:
    name : str
    surname : str

x = Person("Kiana", "Kaslana")
x.name = "Kallen"
```

Próba modyfikacji w ten sposób „zamrożonego” obiektu skończy się błędem:

```
dataclasses.FrozenInstanceError: cannot assign to field 'name'.
```

Mechanizm ten wykorzystuje magiczne funkcje `__setattr__` oraz `__delattr__`, które są zawsze wywoływane w momencie próby zapisu do atrybutów obiektu.

Ostatni szczegół na który warto zwrócić uwagę polega na tym, że choć magiczne funkcje takie jak `__str__()` są uruchamiane automatycznie na przykład przy konwersji na typ `str`, na przykład przez wbudowaną funkcję `str()`, to można je również uruchamiać bezpośrednio, przykładowo:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Person:
    name : str
    surname : str

x = Person("Mei", "Raiden")
print(str(x))
print(x.__str__())
```

Wyświetlony zostanie tutaj dwa razy dokładnie ten sam wynik.

Zadanie 12.3. Wyrażenia przypisujące (ang. walrus operator)

Wyrażenia przypisujące są nowym elementem obecnym dopiero w Pythonie 3.8. Wprowadzony został nowy operator, nazywany „morsem” (ang. *walrus operator*) - `:=`.

Operator ten pozwala na przypisywanie wartości zmiennym wewnątrz wyrażeń. Podstawowym celem stosowania tej koncepcji jest oszczędność pisania i zwięzłość tworzonego kodu.

```
num = [1,2,3,4,5]

if size := len(num) < 10:
    print(f"Lista ma mało elementów, a konkretnie to {size}")
```

W zaprezentowanym przykładzie zmienna `size` tworzona jest od razu wewnątrz instrukcji porównania `if`.

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Przygotuj klasę z atrybutem `@dataclass`, która będzie przechowywać informacje o osobie (imię, wiek, waga). Stwórz te obiekty jako instancje zamrożone.

Polecenie 2.

Przygotuj funkcje `withName`, `withAge` oraz `withWeight`, które pozwolą na stworzenie nowej zamrożonej instancji na podstawie istniejącego obiektu, ale ze zmienionymi cechami – odpowiednio imieniem, wiekiem i wagą.

Sprawdź, czy tworząc takie funkcje jako składowe klasy możesz korzystać z tzw. *fluent interface* (np. `x.withName("Himeko").withAge(28)` powinno z obiektu `x` stworzyć obiekt o imieniu Himeko i wieku 28, zostawiając wagę bez zmian).

Polecenie 3.

Dodaj do swojej klasy danych metodę, która pozwoli na serializację danych do postaci JSON – format JSON ma postać:

```
{ "klucz": "wartość", "klucz2": "wartość2", "klucz3": 0 }
```

Możesz skorzystać z gotowych funkcji do serializacji lub opracować własną, specjalistyczną dla twojej klasy danych.

LABORATORIUM 13. ZAGADNIENIA PRZENOSZENIA ALGORYTMÓW POMIĘDZY RÓŻNYMI JĘZYKAMI PROGRAMOWANIA

Cel laboratorium:

Zapoznanie się z podobieństwami i różnicami pomiędzy paradygmatami programowania podczas tworzenia identycznego oprogramowania w różnych podejściach.

Zakres tematyczny zajęć:

- funkcja „silnia” w wersji iteracyjnej i rekurencyjnej, w paradygmacie logicznym, funkcyjnym i imperatywnym,
- program „FizzBuzz” w paradygmatach imperatywnym i funkcyjnym,
- implementacja algorytmu sortowania szybkiego z wykorzystaniem wbudowanych cech języka Haskell.

Pytania kontrolne:

5. Czy podejście rekurencyjne do obliczania silni jest bardziej czy mniej efektywne niż podejście iteracyjne?
6. W jaki sposób można zastąpić pętle w językach programowania ich pozbawionych?
7. Czy funkcja `map` może być stosowana tak samo w Pythonie oraz Haskellu?
8. Czy algorytm sortowania szybkiego jest bardziej czytelny w wersji funkcyjnej czy też imperatywnej? Czy czytelność ma wpływ na wydajność?

Zadanie 13.1. Silnia

Jak już wiesz z poprzednich zajęć, silnia to typowy przykład funkcji, która może zostać zrealizowana na kilka sposobów, w tym z wykorzystaniem rekurencji, która jest typowym podejściem w językach funkcyjnych (w szczególności rekurencji ogonowej).

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz funkcję obliczającą silnię w języku Prolog.

Polecenie 2.

Napisz funkcję obliczającą silnię w języku Haskell. Czy możesz zastosować takie same podejście jak w przypadku języka Prolog?



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Polecenie 3.

Napisz funkcję obliczającą silnię w języku Python. Czy możesz zastosować takie same podejście jak w języku Haskell? Przetestuj, czy bardziej efektywna w języku Python byłaby wersja korzystająca z rekurencji czy też wersja korzystająca z listy lub generatora.

Zadanie 13.2. FizzBuzz

FizzBuzz to program, który jest bardzo często wykorzystywany jako zadanie rekrutacyjne – sprawdza, czy kandydat zna składnię języka programowania oraz czy potrafi czytać poprawnie wymagania dotyczącego postawionego przed nim projektu.

Program FizzBuzz ma za zadanie wyświetlić liczby od 1 do 100 w kolejnych liniach, ale:

- Zamiast liczb podzielnych przez 3 wyświetla słowo „Fizz”,
- Zamiast liczb podzielnych przez 5 wyświetla słowo „Buzz”,
- Zamiast liczb podzielnych przez 3 i 5 jednocześnie wyświetla słowo „FizzBuzz”.

Oznacza to, że program wyświetli wartości:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
...
...
97
98
Fizz
Buzz
```

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz program FizzBuzz w wybranym przez siebie imperatywnym języku programowania (np. C, Java, PHP, JavaScript).

Polecenie 2.

Napisz program FizzBuzz w wybranym przez siebie języku programowania, ale **bez użycia instrukcji pętli**.

Polecenie 3.

Napisz program FizzBuzz w języku Python, w taki jednak sposób, aby wykorzystać listy – spróbuj przygotować program, który najpierw wygeneruje listę odpowiednich wartości i dopiero tę listę wyświetli z wykorzystaniem pętli `for-in`. Możesz skorzystać z funkcji `map`.

Polecenie 4.

Napisz program FizzBuzz w języku Haskell. Wykorzystaj instrukcję `map` do przekształcenia wartości 1..100 na odpowiednie łańcuchy tekstu. Musisz uwzględnić fakt homogeniczności listy w Haskellu, więc w celu zamiany liczby na łańcuch znaków skorzystaj z funkcji `show`. Do wyświetlenia tekstu z dodaniem znaku nowej linii możesz skorzystać z funkcji `putStrLn`, jednak jak zauważysz – niezbędny będzie mechanizm, aby móc tę funkcję uruchomić dla każdego elementu listy – a funkcja `map` nie pozwoli na to, ponieważ `putStrLn` jest typu `String -> IO ()`, a `map` oczekuje funkcji typu `(a -> b)`. Aby obejść ten problem zaimportuj moduł `Control.Monad` i wykorzystaj z niego funkcję `forM_`, która jest typu `(Foldable t, Monad m) => t a -> (a -> m b) -> m ()` i pozwala uruchomić funkcję taką jak `putStrLn` dla każdego elementu listy.



Zadanie 13.3. Sortowanie szybkie

Jak wiadomo ci z zajęć dotyczących algorytmów i struktur danych, sortowanie szybkie jest dość efektywnym algorytmem sortowania. Jego ogólna zasada polega na podziale listy na dwie części wedle pewnego wyróżnionego elementu i ułożenia elementów mniejszych od wyróżnionego po jednej stronie, a większych – po drugiej stronie, po czym funkcja uruchamia się rekurencyjnie dla obydwu połówek listy.

Jak możesz zauważyć, skoro algorytm opiera się o rekurencję oraz wybieranie elementów z listy, można stwierdzić, że będzie dobrym algorytmem do implementacji w językach funkcyjnych.

Proszę samodzielnie wykonać następujące polecenia.

Polecenie 1.

Napisz implementację algorytmu QuickSort (sortowania szybkiego) w języku Python, korzystając z typowego podejścia imperatywnego, wykorzystując pętlę do układania elementów po dwóch stronach elementu wyróżnionego.

Polecenie 2.

Napisz prostą implementację algorytmu QuickSort w języku Haskell, korzystając z operatora ++ do sklejania list oraz funkcji filter do wybierania elementów z listy.



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego