ĆWICZENIE 5

SCILAB: elementy programowania.

Pisząc skrypty możemy chcieć uzyskać możliwość interaktywnego wprowadzania danych podczas działania programu. W tym celu możemy wykorzystać polecenie ***input.***

input *– polecenie monituje użytkownika o wprowadzanie danych z klawiatury* (zastosowanie w tworzeniu interaktywnych skryptów). Wprowadzane dane mogą być zmiennymi typu liczbowego lub tekstowego.

Podstawowe składnie polecenia:

zmienna = input('tekst monitu'); - tekst monitu pojawia się w nowej linii i program oczekuje na podanie wartości monitowanej zmiennej (zmienna liczbowa).

zmienna_tekstowa = input('tekst','string'); - tekst monitu pojawia się w nowej linii i program oczekuje na podanie wartości monitowanej zmiennej (zmienna tekstowa).

Przykłady:

```
--> x = input('podaj wartość x: ')
podaj wartość x:  - należy podać wartość zmiennej np. 4
x =
4
--> y = input('podaj imię','string')
podaj imię  -  należy podać zmienną tekstową, np. Marek
 y  =
 Marek
```

Inne przydatne funkcje wejścia/wyjścia

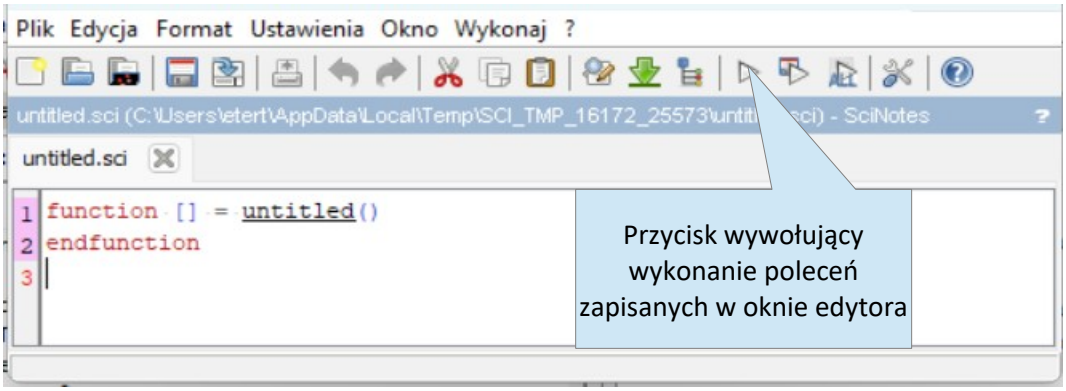
- **disp(x)** - wyświetlanie wartości zmiennej x,
- **disp('tekst')** – wyświetlenie tekstu,
- **write(%io(2),zmienna)** – zapisuje zmienną (liczby rzeczywiste lub ciągi znaków) w konsoli,
- **return /resume** – wznowienie wykonywanie skryptu,
- **abort** - przerywa (zatrzymuje) wykonywanie skryptu
- **pause** – przerwa w wykonywaniu skryptu lub funkcji do czasu wciśnięcia klawisza Enter,
- **sleep(milisekundy)** – przerywa wykonywanie wszystkich poleceń na ustalony czas.
- **break** - przerywa wykonywanie instrukcji pętli (for, while).
- **continue –** wymusza przejście do początku instrukcji pętli (for, while).

Numeryczne rozwiązywanie zadań pojawiających się w praktyce inżynierskiej wymaga przeprowadzenia numerycznej realizacji algorytmu opisującego sposób rozwiązania danego problemu. Realizacja takiego algorytmu zazwyczaj wymaga wykonania wielu działań takich jak: pozyskiwanie/wczytywania danych, obliczenia algebraiczne/logiczne, tworzenie zmiennych zawierających obliczone wartości, prezentowanie uzyskanych wyników w postaci graficznej/tekstowej, zapis wyników do pliku itp. Chcąc zrealizować te wszystkie zadania korzystając jedynie z *Konsoli,* należałoby wprowadzać wiele poleceń w określonej kolejności. Powtarzając obliczenia (np. dla zmienionych danych wejściowych) należałoby powtórzyć wprowadzanie do *Konsoli* wszystkich poleceń (wspomagając się ew. zawartością *Historii Poleceń*). Dlatego, gdy algorytm obliczeniowy ma być realizowany wielokrotnie, sekwencję poleceń służących realizacji tego algorytmu można zapisać wykorzystując ***Edytor*** programu Scilab. Tak utworzoną i zapisaną sekwencję poleceń można przechowywać w postaci pliku i wykorzystywać wielokrotnie. Edytor jest więc narzędziem do programowania zadań numerycznych realizowanych automatycznie w programie Scilab. W Scilab

wyróżniamy dwie podstawowe struktury programistyczne służące do automatyzacji zadać obliczeniowych. Są to: **funkcje** oraz **skrypty**.

Uwaga: pisząc funkcje lub skrypty korzystamy z edytora wywoływanego poleceniem: *edit*

Edytor pojawi się w nowym oknie, które podobnie jak każde okno w Scilab może zostać 'zadokowane' w głównym oknie Scilaba.



Korzystając z edytora możemy zapisywać sekwencję poleceń i następnie wywołać realizację wszystkich poleceń w kolejności w jakiej są zapisane. Poszczególne polecenia są wykonywane a ich wyniki są widoczne w odpowiednich miejscach programu: wykresy w oknach graficznych, utworzone zmienne w przeglądarce zmiennych, zmienne dla których nie wyłączono 'echa' oraz celowo wyświetlane komunikaty w konsoli. Utworzony z użyciem edytora zestaw poleceń możemy zapisać w postaci pliku z rozszerzeniem *.sci* lub *.sce* dzięki czemu możemy z tych plików korzystać w późniejszym czasie. Edytor jest więc miejscem/narzędziem umożliwiającym programowanie wszelkich zadać, do których Scilab może być wykorzystywany. Edytor Scilab-a posiada funkcjonalność edytora programistycznego: numerowanie wierszy, kolorowanie składni poleceń (na powyższym rysunku widzimy kolory zastosowane do nazw poleceń i nawiasów), oznaczanie zakresów typowych poleceń itp. Reguły zapisywania poleceń są takie same jak dla poleceń wpisywanych bezpośrednio do konsoli. Edytor umożliwia jednoczesne otwarcie i pracę w wielu zakładkach.

Funkcje:

Pojęcie funkcji (w kontekście Scilab) jest nam już znane. Program oferuje możliwość korzystania z wielu funkcji wbudowanych będących typowymi elementami programu. Np. obliczenie wartości funkcji trygonometrycznych, np. sinus dla zmiennej x, jest realizowane przez wywołanie odpowiedniej funkcji *sin(x)*, gdzie *sin* jest nazwą funkcji, zaś w nawiasie podany jest argument dla którego funkcja ma być obliczona. Innym przykładem jest funkcja *linspace* stosowana do tworzenia wektorów o równomiernie rozłożonych elementach. Do poprawnego działania funkcja *linspace* wymaga podania argumentów umieszczonych w nawiasie. Np. *linspace(0,5,100)* spowoduje utworzenie wektora o 100 elementach równomiernie rozłożonych w zakresie od 0 do 5. Takich funkcji jest bardzo wiele i dotyczą różnych elementów programu.

Scilab daje użytkownikowi możliwość tworzenia własnych funkcji, które będą działały na podobnej zasadzie jak funkcje wbudowane, tzn. będzie można je wywoływać podając nazwę oraz zmienne niezbędne do jej realizacji. Funkcje użytkownika są tworzone w edytorze Scilaba i zapisywane w postaci pliku z rozszerzeniem *.sci* lub *.sce*. **UWAGA**: nadając nazwę utworzonej funkcji **nie** należy stosować standardowych nazw funkcji wbudowanych Scilaba. Wywołanie funkcji następuje poprzez podanie jej nazwy (bez rozszerzenia i ścieżki dostępu), oraz argumentu/argumentów. Ważne aby plik danej funkcji był zapisany w katalogu bieżącym lub w jednym z katalogów przeszukiwanych przez Scilaba przy wywoływaniu poleceń.

Definicja funkcji ma następującą postać:

function[wartość_funkcji]=nazwa_funkcji(argumenty)
ciąg instrukcji używający argumentów do obliczenia wartości_funkcji
endfunction

Zarówno *wartość_funkcji* jak też *argumenty* mogą być liczbami, wektorami lub macierzami.

Należy zwrócić uwagę, że zmienną `wartość_funkcji` należy zapisywać w nawiasie kwadratowym , zaś zmienną `argumenty` w nawiasie zwykłym. Jeśli zmiennych jest więcej niż jedna należy je rozdzielić przecinkami.

Przykład 1: *(funkcja o nazwie sil, obliczająca silnię liczby naturalnej)*
Otworzyć okno edytora i wpisać poniższą funkcję.

```
function[wynik]=sil(n)
wynik=1;
for i=1:n
wynik=wynik*i;
end
endfunction
```

W powyższej funkcji użyto instrukcji *for*, o której będzie więcej w dalszej części instrukcji.
Tak utworzoną funkcję należy zapisać do pliku we własnym katalogu roboczym.
Uwaga: należy stosować zasadę , że plikowi funkcji nadaje się taką samą nazwę jak *nazwa_funkcji* – w powyższym przypadku będzie to: *sil.sce*.
Zanim funkcję będzie można wywołać w *Console* Scilaba należy wywołać wykonanie pliku z pozycji *Edytora* (rys powyżej) lub zastosować polecenie *exec('plik_funkcji',-1)*. Plik zostanie sprawdzony i jeśli nie będzie błędów stanie się funkcją, którą będzie można wywoływać podając nazwę oraz argument.

```
--> exec('C:\.....\sil.sci', -1)
--> sil(5)
 ans  =
   120.
--> a=sil(4)
 a  =
   24.
```

Z tak utworzonej funkcji możemy korzystać podczas bieżącej sesji Scilab-a. Aby móc korzystać z tej funkcji w kolejnych sesjach programu należy funkcję w danej sesji 'aktywować' wywołując ją z poziomu *Edytora* lub stosując polecenie *exec('plik_funkcji',-1)*. Podobnie postępujemy, jeśli w danej sesji chcemy skorzystać z funkcji utworzonych wcześniej.

Jeżeli funkcja będzie używana tylko w bieżącym skrypcie i nie ma potrzeby zapisywania funkcji w oddzielnym pliku, można skorzystać z polecenia *deff* służącego do tworzenia tzw. funkcji *in-line*. Polecenie *deff* służy do tworzenia nieskomplikowanych, pojedynczych funkcji.
Składnia polecenia jest następująca:

```
deff ('y = nazwa_funkcji(argumenty)',' y=wyrażenia opisujące działanie funkcji')
```

Przykład 2: *(funkcja obliczająca wartość wyrażenia $y=x^2-3x+2$)*

```
-->deff('y = f(x)','y = x^2-3*x+2')
-->f(5)
ans  =
   12.
```

Przykład 3: *(funkcja obliczająca sumę oraz różnicę dwóch zmiennych)*

```
-->deff('[S,R] = m(x,y)','S = x+y ; R=x-y')
-->[suma, roznica]=m(5,3)
 suma  =
   8.
 roznica  =
   2.
```

Funkcja utworzona poleceniem *deff* nie jest zapisywana w oddzielnym pliku, jest więc funkcją lokalną dostępną w bieżącej sesji programu.

Instrukcje programistyczne: skrypty

Pod względem programistycznym, skrypty stanowią pewną większa całość i mogą zawierać wyrażenia obliczeniowe, funkcje wbudowane Scilaba jak również funkcje stworzone przez użytkownika. Przy pisaniu skryptów stosuje się typowe instrukcje złożone spotykane we wszystkich językach programowania wysokiego poziomu: *if, for, while, select*. Składnie wymienionych instrukcji są analogiczne jak w innych znanych językach programowania. Przystępując do programowania z wykorzystaniem instrukcji złożonych będzie zachodziła konieczność stosowania operatorów porównania. W tabeli obok jest lista dostępnych operatorów realizujących działania porównywania (sprawdzania czy zachodzi określona relacja). Wynikiem sprawdzania relacji jest wartość logiczna prawda/fałsz. Wartość ta może być porównywana ze standardowymi zmiennymi logicznymi Scilaba (%t lub %T dla prawda, %f lub %F dla fałsz) Jeżeli zajdzie konieczność jednoczesnego sprawdzenia kilku relacji, przydatne mogą okazać się operatory logiczne (tabela obok).

oznaczenie	**Typ sprawdzanej relacji**
= =	równość
<	mniejsze niż
>	większe niż
< =	mniejsze lub równe
> =	większe lub równe
< > lub ~ =	nie równe

oznaczenie	Typ operatora
&	and / koniunkcja
|	or / alternatywa
~	not / negacja

Przykład 4: *(sprawdzenie relacji między liczbami)*

```
-->3==6
 ans  =
  F              // realacja nieprawdziawa, wynik sprawdzania F /false /fałsz
 --> -3<0 & 3>0
 ans  =
  T              // realacja prawdziawa, wynik sprawdzania T / true /prawda
```

Przykład 5 - skrypt: *(zapisać w nowym pliku o nazwie silnia)*

W oknie edytora otworzyć nową zakładkę i wpisać poniższe linie skryptu.

```
n=input('Podaj wartość n: ');
disp('Wartość n! wynosi:')
disp(sil(n))    (Wykorzystujemy wcześniej stworzoną funkcję sil)
```

W powyższym skrypcie wykorzystujemy wcześniej utworzoną funkcję *sil.* W bieżącej sesji funkcja *sil* była już uruchamiana i jest aktywna. Jednak przy kolejnej sesji Scilab-a należy pamiętać aby przed uruchomieniem skryptu aktywować działanie funkcji *sil*. Można to zrobić za pomocą *Edytora* lub dopisując na początku skryptu polecenie

`exec('C:\...\sil.sci', -1)` /w miejsce kropek należy wpisać ścieżkę dostępu do pliku funkcji/

w ten sposób każde uruchomienie skryptu będzie aktywowało funkcję sil.

Skrypt do uruchomienia nie wymaga podawania argumentów. Można go uruchomić z poziomu edytora (ikona -wykonaj, lub menu -wykonaj)., lub w *Console Scilaba* stosując polecenie *Exec('nazwa_pliku',0):*

```
--> exec('silnia.sci', 0)
Podaj wartość n: 6
 Wartość n! wynosi:
  720.
```

Instrukcje sterujące: pętla for („dla”). Instrukcja działa ‘w pętli’ i powtarza ściśle określoną ilość razy grupę instrukcji zawartych wewnątrz pętli.

```
for zmienna_iterowana = macierz_wartości (wart.początkowa:skok:wart.końcowa)
ciąg_instrukcji
end
```

Ilość powtórzeń (iteracji) jest określona za pomocą *macierzy_wartości,* która ma postać `wart.początkowa:skok:wart.końcowa. Skok` może zostać pominięty, wówczas przyjmowana jest wartość domyślna skoku = 1.

Przykład 6: (*dla liczb z zakresu od 1 do 7 utworzyć macierz M zawierająca: w pierwszym wierszu – liczby nieparzyste, w drugim wierszu – kwadraty tych liczb nieparzystych, w trzecim wierszu liczby nieparzyste podniesione do trzeciej potęgi - plik P_6.sci*).

W oknie edytora otworzyć nową zakładkę i wpisać poniższe linie skryptu.

```
for i=1:7
   M(1,i)=i;
   M(2,i)=i^2;
   M(3,i)=i^3;
end
```

W konsoli wywołujemy utworzoną macierz M:

```
--> M
 M  =
 1.  2.  3.    4.   5.   6.   7.
 1.  4.  9.   16.  25.  36.  49.
 1.  8.  27.  64.  125. 216. 343.
```

Instrukcje sterujące: pętla while („dopóki”). Instrukcja działa ‘w pętli’ i powtarza grupę instrukcji nieokreśloną liczbę razy. Uruchomienie każdej kolejnej pętli jest uwarunkowane spełnieniem zapisanego warunku logicznego (`wyrażenie_warunkowe`). Dopóki `wyrażenie_warunkowe` jest prawdziwe (zwraca wartość logiczną *prawda/true*) ciąg instrukcji jest wykonywany. Jeśli `wyrażenie_warunkowe` nie jest prawdziwe (zwraca wartość logiczną *fałsz/false*) wykonywany jest ciąg instrukcji zapisanych po słowie `else` lub pętla *while* kończy działanie.

```
while wyrażenie_warunkowe (logiczne)
      ciąg_instrukcji
else
      ciąg_instrukcji
end
```

Część „`else`” jest opcjonalna, instrukcje występujące w niej są wykonywane gdy „wyrażenie warunkowe” stanie się fałszywe.

Przykład 7: (*utworzenie macierzy A 4x3, której elementy są sumą numeru wiersza i numeru kolumny - plik P_7.sci*). W oknie edytora otworzyć nową zakładkę i wpisać poniższe linie skryptu.

```
x=1;
while x < 5
      A(x,1)=x+1;
      A(x,2)=x+2;
      A(x,3)=x+3;
      x=x+1;
end
```

W konsoli wywołujemy utworzoną macierz A:

```
--> A
A  =
  2.  3.  4.
  3.  4.  5.
  4.  5.  6.
  5.  6.  7.
```

UWAGA: Instrukcja *While* nie określa ilości wykonywanych iteracji a jedynie warunek jaki powinien być spełniony aby rozpocząć kolejną iterację. Z tego powodu należy uważnie formułować *wyrażenie_warunkowe* aby uniknąć ‘zapętlenia’ programu (niekończące się wykonywanie pętli).

Instrukcje sterujące: warunek if („jeżeli”). Instrukcja warunkowa *if* sprawdza wyrażenie(a) warunkowe i wykonuje odpowiednią grupę instrukcji gdy wyrażenie jest prawdziwe. Opcjonalnie stosowane słowa *elseif* i *else* umożliwiają wykonywanie alternatywnych grup instrukcji, przy czym dla każdego *elseif* sprawdzane jest oddzielne wyrażenie warunkowe. Po *else* nie podaje się (i nie

sprawdza) wyrażenia warunkowego. Grupa instrukcji po *else* jest wykonywania gdy żadne z wcześniejszych wyrażeń warunkowych nie jest prawdziwe.

Składnia instrukcji warunkowej ***if***:

```
 if wyrażenie_warunkowe1
       grupa_instrukcji1
  elseif wyrażenie_warunkowe2
       grupa_instrukcji2
  else grupa_instrukcji3
 end
```

Części „elseif" i „else" są opcjonalne.

Przykład 8: (*znajdywanie odwrotności liczby - plik P_8.sci*):
W oknie edytora otworzyć nową zakładkę i wpisać poniższe linie skryptu.

```
x=input('podaj liczbę różną od zera ');
if (x~=0)
      y=1/x;
      disp ('odwrotmość liczby');
      disp(x);
      disp('wynosi')
      disp(y);
else disp('błąd, liczba nie może być zerem')
end
```

Instrukcje sterujące: instrukcja select/case. Instrukcja sprawdza kolejne warunki i wykonuje polecenia po pierwszym przypadku, w którym sprawdzany warunek jest spełniony (prawdziwy). Jeśli nie zostanie znalezione żadne dopasowanie, wykonywane jest polecenie po *else*.

Składnia instrukcji warunkowej ***select/case***:

```
select wyrażenie ,
case wyrażenie1 then polecenia ,
case wyrażenie2 then polecenia ,
...
case wyrażenieN then polecenia ,
else polecenia ,
end
```

Przykład 9: (*zamiana numeru miesiaca na zapis słowny - plik P_9.sci*)
W oknie edytora otworzyć nową zakładkę i wpisać poniższe linie skryptu.

```
A=input('podaj numer miesiaca')
select A
case 1 then
    disp('styczen')
case 2 then
    disp('luty')
case 3 then
    disp('marzec')
case 4 then
    disp('kwiecień')
case 5 then
    disp('maj')
case 6 then
    disp('czerwiec')
case 7 then
    disp('lipiec')
case 8 then
    disp('sierpien')
case 9 then
    disp('wrzesien')
case 10 then
    disp('pazdziernik')
```

```
case 11 then
    disp('listopad')
case 12 then
    disp('grudzien')
else
    disp('błąd – należy podać liczbę całkowitą z zakresu 1 - 12')
end
```

Poszczególne instrukcje sterujące mogą być wzajemnie zagnieżdżone, tzn. np. wewnątrz instrukcji pętli *for* można umieścić instrukcję warunkową *if.* Poniższy przykład pokazuje zagnieżdżenie instrukcji warunkowej *if* wewnątrz pętli *while.*

Przykład 10: (*gra – zgadywanie numerów - plik P_10.sci*)
W oknie edytora otworzyć nową zakładkę i wpisać poniższe linie skryptu.

```
M=20;                                       // maksymalna wartość liczby
liczba=floor(1+M*rand());                   //losowa liczba całkowita z zakresu 1:M
disp('Odgadnij liczbę całkowitą z zakresu: '); //komunikat
disp([1,M]);                                // informacja o granicach zakresu
proba=input('Odgadnij liczbę: ');           // wybór liczby– zachęta do zgadywania

while (proba~=liczba)                       // początek instrukcji while
   if proba>liczba                          // początek instrukcji if-else
      disp('Liczba jest za duża');
   else
      disp('Liczba jest za mała');
   end                                      // koniec instrukcji if
   proba=input('Odgadnij liczbę: ');        // kolejny wybór liczby– zachęta do zgadywania
end                                         // koniec instrukcji while
disp('Brawo – Liczba odgadnięta!');
```

Zadanie1.
Napisać funkcję dla której argumentem będzie liczba n. Funkcja oblicza sumę kwadratów liczb całkowitych z zakresu 0 do n gdy n jest dodatnie, lub od n do 0 gdy n jest ujemne. Funkcję zapisać w pliku Z1.sci

Zadanie2.
Napisać funkcję, która będzie dokonywała przeliczenia kąta podanego w radianach na kąt wyrażony w stopniach. Funkcję zapisać w pliku Z2.sci

Zadanie 3
Napisać skrypt obliczający pole trójkąta dla podanych długości jego boków. zapisać w pliku Z3.sci
Algorytm programu:
1. Użytkownik podaje długości boków trójkąta *a, b, c* (polecenie input),
2. Obliczamy połowę obwodu trójkąta – wynik przypisujemy zmiennej *p,*
3. Obliczamy pole trójkąta wg wzoru:
$$A=(p*(p-a)*(p-b)*(p-c))^{(1/2)}$$
4. Wynik powinien zostać wyświetlony w *Konsoli*. Komunikat powinien zawierać długości boków trójkąta oraz wartość wyliczonej powierzchni.