Paradygmaty Programowania

mgr inż. Marcin Bączyk (prowadzący) dr inż. Marek Niewiński(autor prezentacji)

Wykład 4

15 marca 2021

Treść dzisiejszego wykładu

- wprowadzenie do programowania strukturalnego
- programowanie proceduralne
- instrukcja “goto“
- programowanie strukturalne
- przykład

Język programowania

Przyjęty umowny sposób zapisu symboli, dzięki któremu można komunikować się z maszyną obliczeniową zlecając jej wykonanie zadania.

W drugiej połowie lat 40 (ubiegłego wieku), kiedy John von Neumann zaproponował nowy sposób budowy maszyn obliczeniowych - charakteryzujący się między innymi tym, że maszyny te posiadają skończoną listę podstawowych rozkazów - operatorzy mogli wprowadzać sekwencje tych rozkazów by rozwiązywać specyficzne problemy obliczeniowe i stali się pierwszymi **programistami**.

Programowanie - język maszynowy

Pierwsze programy pisane były w językach maszynowych:

- program składał się z sekwencji: **kodów rozkazów** i **adresów** ich argumentów

Programy w kodzie maszynowym

1. absolutnie nieprzenośne między różnymi typami maszyn
2. bardzo trudne w modyfikacji
3. nieczytelne a więc i wyszukanie błędów często awykonalne

Przykład: x86-64

```
1000:   f3 0f 1e fa
1004:   48 83 ec 08
1008:   48 8b 05 d9 2f 00 00
100f:   48 85 c0
1012:   74 02
1014:   ff d0
```

Programowanie - asembler

Asembler

Program tłumaczący programy zapisane w postaci symbolicznej na kod maszynowy

Idea:

- przypisać każdemu kodowi instrukcji **mnemonik** w sposób jednoznacznie kojarzący się z wykonywaną czynnością
- nadać **nazwę symboliczną** każdej z danych, by uniknąć operowania jawnymi adresami pamięci

Przykład: x86-64

```
push {r7}
add r7, sp, #0
movs r3, #0
mov r0, r3
mov sp, r7
ldr.w r7, [sp], #4
```

Programowanie - pierwsze języki wysokiego poziomu

W drugiej połowie lat 50 powstały języki kompilowalne:

- **FORTRAN** - *formula translation* - stworzony przez zespół pracujący w **IBM**
- **COBOL** - *common business-oriented language* - stworzony przez komitet CODASYL (Conference on Data Systems Languages)
- **ALGOL** - *algorithmic language* - stworzony przez zespół pracujący w Swiss Federal Institute of Technology in Zurich (ETH Zurich)

Wszystkie te języki udostępniały możliwość definiowania **podprogramów**

Podprogram (subroutine)

Sekwencja instrukcji wykonująca konkretne zadanie - zdefiniowana w postaci jednostki, którą można wywołać (callable unit).
W różnych językach programowania może być nazywany:

- funkcją
- procedurą
- metodą

Programowanie proceduralne

Metodyka według której kod programu powinien być podzielony na **podprogramy**.

- każdy **podprogram** może być wywołany w dowolnym miejscu wykonywanego programu:
  - przez inny podprogram,
  - przez siebie samego
- **podprogramy** powinny pobierać wszystkie dane jako argumenty wywołania a wynik(i) działania zwracać poprzez tzw "return values" lub też wykorzystując argumenty wejściowe.

Podprogram jest podmiotem wykonywanych działań a dane ich przedmiotem

Programowanie strukturalne - wprowadzenie historyczne

Pierwszy kryzys oprogramowania

W latach 60 i 70 ubiegłego wieku po raz pierwszy zauważono, że tworzenie coraz to bardziej złożonego oprogramowania, działającego efektywnie i bezbłędnie jest problemem **trudnym** a w związku z tym coraz bardziej kosztownym.

Kryzys oprogramowania przejawiał się między innymi w tym że:
stworzone oprogramowanie było:

- nieefektywne
- często nie spełniało postawionych wymagań
- jego kod źródłowy był niskiej jakości a w związku z tym trudny w "utrzymaniu"

i dlatego wiele projektów kończyło się:

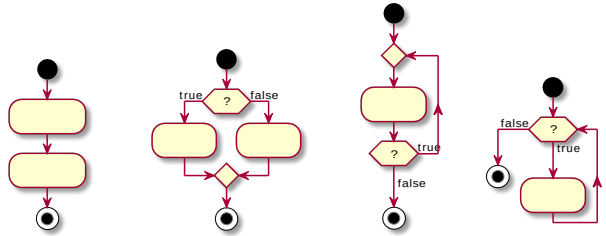
- przekroczeniem przyjętego budżetu
- przekroczeniem założonych ram czasowych wykonania

W celu poprawienia "jakości" tworzonego oprogramowania wprowadzono metodykę **programowania strukturalnego**

Twierdzenie Böhma–Jacopini'ego

Twierdzenie z obszaru teorii języków programowania mówiące że:
każdy graf przepływu (flowchart) reprezentujący realizację wybranego algorytmu może być zbudowany tylko z trzech typów struktur kontrolnych:

- sekwencji
- selekcji
- iteracji



sekwencja selekcja iteracja

Edsger Dijkstra - prace

Holenderski naukowiec, który uważany jest za jednego z "ojców" **programowania strukturalnego**.

W swoich pracach postulował wprowadzenie **matematycznego dowodzenia** poprawności implementacji algorytmów poprzez:

- ich rekursywną dekompozycję na coraz to mniejsze elementy do poziomu podstawowych jednostek
- matematycznego dowodzenia poprawności działania tych podstawowych jednostek

Podczas swoich prac zauważył, że:

- istnieją pewne sposoby stosowania instrukcji **goto** które **uniemożliwiają** rekursywną dekompozycję problemu
- istnieją także takie sposoby stosowania instrukcji **goto** które nie wprowadzają tego problemu (odpowiadają one operacjom: selekcji i iteracji).

Edsger Dijkstra - prace cd

Wykazał że:

Poprawność działania **instrukcji sekwencyjnych** oraz **selekcji** może być dowiedziona przy użyciu enumeracji.

Poprawność działania **instrukcji iteracyjnych** może być dowiedziona przy użyciu techniki indukcji.

Niestety dowodzenie tego typu było **bardzo pracochłonne** i nie przyjęło się w praktycznych zastosowaniach.

Proszę zauważyć:

Struktury kontrolne programów, które pozwalają na matematyczne wyprowadzenie dowodu poprawności działania są **tymi samymi** strukturami kontrolnymi dzięki którym można skonstruować implementację dowolnego algorytmu zgodnie z twierdzeniem Böhma–Jacopini'ego.

Instrukcja skoku **goto**

Instrukcja występująca w wielu starszych językach programowania, umożliwiająca przekazanie sterowania do dowolnej innej instrukcji w ramach działającego procesu. Miejsce przekazania sterowania oznaczone jest poprzez etykietę (label:)

Przykład w języku C++

```
int main() {
  bool condition = true;
START:
  if (condition)
    goto START;
  else
    goto END;
END:
  return 0;
}
```

Przykład w języku PHP

```
<?php
$condition=True;
START:
if($condition==True) goto START;
if($condition==False) goto END;
END:
echo "End"
?>
```

Instrukcja skoku **goto** - cd

Go To Statement Considered Harmful

W 1968 r. Edsger Dijkstra napisał list do redakcji czasopisma "CACM" w którym zasugerował, że używanie instrukcji **goto** jest szkodliwe i prowadzi to tworzenia nieczytelnego[1] i trudno-testowalnego kodu.
Wywołał on szeroką dyskusję w środowisku programistów, która trwała ponad 10 lat.

Ostatecznie jego stanowisko zwyciężyło i nowoczesne języki programowania takie jak: *python*, *java*, *javascript*, *rust* nie udostępniają tej instrukcji.
Dodatkowo języki, które mają wbudowaną tą instrukcję zazwyczaj **ograniczają** zakres celu skoku (np: do ciała aktualnej funkcji)

[1] Spaghetti code

Termin oznaczający skomplikowany i trudny do zrozumienia kod źródłowy, powstały np. z powodu nadużywania instrukcji goto.

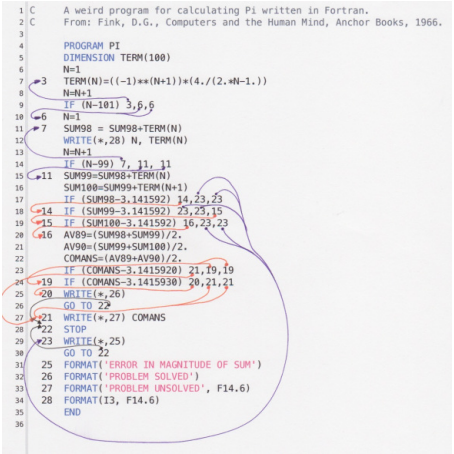
Spaghetti code - przykład

Fortran **IF**

```
IF (value)
    3, 6, 9
```

odpowiada semantycznie

```
IF (value)
    (value<0) THEN GOTO 3
    (value==0) THEN GOTO 6
    (value>0) THEN GOTO 9
```



Źródło: https://craftofcoding.files.wordpress.com/2013/10/lore_spaghetti.pdf

Programowanie strukturalne

Każda metodyka programowania narzuca programiście, pewien zestaw ograniczeń

Szeroka definicja programowania strukturalnego

Metodyka programowania, która nakłada ograniczenia na **bezpośrednie** przekazywanie sterowania programem (by Robert C. Martin).

Definicja bardziej użyteczna

Metodyka programowania, która narzuca:

- stosowanie wysokopoziomowych struktur kontrolnych: selekcji, iteracji i sekwencji oraz unikanie stosowania nisko-poziomowej instrukcji skoku
- stosowanie rekursywnej dekompozycji problemu obliczeniowego typu *top-down* w celu zdefiniowania tzw. *podstawowych jednostek obliczeniowych*
- każda *podstawowa jednostka* ma rozwiązywać pojedynczy problem programistyczny

Programowanie strukturalne - cd

Sposób rozwiązywania problemu obliczeniowego metodyką *programowania strukturalnego* można podsumować w następujący sposób:

- Pisz kod programu:
  - stosując struktury kontrolne typu: selekcja, iteracja i sekwencja
  - deklarując i wywołując podprogramy: $P_1, ..., P_n$
- definiuj implementację podprogramów $P_1, ..., P_n$ w ten sposób, że wykorzystuje ona wywołania innych podprogramów
- dekompozycję podprogramów wykonuj dopóty ich implementacja będzie na tyle prosta, że nie będzie wymagać wywołań innych podprogramów

Python a programowanie strukturalne

Instrukcja selekcji

```
if expression:
    statement(s)
else:
    statement(s)
```

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
else:
    statement(s)
```

Instrukcje iteracji

```
while expression:
    statement(s)
```

```
for iterating_var in sequence:
    statement(s)
```

Definiowanie podprogramu przy użyciu funkcji

```
def function_name( parameters ):
    statement(s)
    return [expression]
```

Python - odstępstwa od "idealnych" struktur kontrolnych

Wiele współczesnych języków programowania ma wbudowane instrukcje:

- **break**
- **continue**
- **exit**

których użycie może zmienić sposób działania głównie instrukcji iteracji

Python - break

```
for iterating_var in sequence:
    # loop code before condition
    if condition:
        break
    #loop code after condition
# code outside for-loop
```

```
while expression:
    # loop code before condition
    if condition:
        break
    #loop code after condition
# code outside while-loop
```

Po wykonaniu instrukcji **break** sterowanie wykonaniem programu zostanie przeniesione do linii *# code outside loop*

Python - continue

```
for iterating_var in sequence:
    # loop code before condition
    if condition:
        continue
    #loop code after condition
# code outside for-loop
```

```
while expression:
    # loop code before condition
    if condition:
        continue
    #loop code after condition
# code outside while-loop
```

Po wykonaniu instrukcji **continue** sterowanie wykonaniem programu zostanie przeniesione do linii *# loop code before condition*

Python sys.exit

```
import sys
sys.exit(0)
```

Wywoływanie funkcji **exit** kończy działanie programu i zwraca do systemu operacyjnego kod zakończenia

Programowanie strukturalne - przykład

Problem obliczeniowy

Znaleźć rozwiązania równania kwadratowego w dziedzinie liczb rzeczywistych i zespolonych

$$a \cdot x^2 + b \cdot x + c = 0$$

Krok 1

Definiujemy dedykowany podprogram, który będzie rozwiązywał problem obliczeniowy

```
#!/usr/bin/env python3
def quadraticEquationsSolver():
    pass

if __name__ == "__main__":
    quadraticEquationsSolver()
```

Programowanie strukturalne - przykład - cd

W ramach rozwiązywanego zadania można wyróżnić następujące pod-zadania:

- pobranie współczynników równania od użytkownika
- właściwe obliczenia znajdujące miejsca zerowe
- wyświetlenie uzyskanych wyników

Na potrzeby bieżącego zadania obliczeniowego dodatkowo przyjęto:

- współczynniki równania są wprowadzane jako argumenty wywołania skryptu (3 liczby rzeczywiste w postaci: a b c)
- współczynniki te będą przechowywane w postaci 3-elementowej krotki
- wyniki będą udostępniane jako 2-elementowa krotka liczb rzeczywistych lub zespolonych

Krok 2

Dekompozycja na pod-zadania

```
def quadraticEquationsSolver():
    status,coefficients=readCoefficients()
    if status:
        zeroes=findZeroes(coefficients)
        displayZeroes(zeroes)
    else:
        print("Error!! - wrong input format")
```

Zmienna *status* służy do raportowania o poprawności wczytanych współczynników

Krok 3

Implementacja pod - zadania: wczytanie wartości współczynników równania

```
import sys
import math

def readCoefficients():
    if len(sys.argv)-1!=3:
        return False, (0,0,0)
    else:
        a = float(sys.argv[1])
        b = float(sys.argv[2])
        c = float(sys.argv[3])
        if abs(a)!=0.0:
            return True, (a,b,c)
        else:
            return False, (0,0,0)
```

Programowanie strukturalne - przykład - cd

W ramach pod- zadania wyznaczania miejsc zerowych można wyróżnić kolejne pod - zadania:

- wyznaczenie wartości Δ
- gdy $\Delta >= 0$ wyznaczenie miejsc zerowych w dziedzinie liczb rzeczywistych
- gdy $\Delta < 0$ wyznaczenie miejsc zerowych w dziedzinie liczb zespolonych

Krok 4

Dekompozycja pod - zadań dla pod - programu *findZeroes*

```
def findZeroes(coefficients):
    a,b,c=coefficients
    delta=calculateDelta(a,b,c)
    if delta>=0:
        zeroes=findRealZeroes(a,b,c,delta)
    else:
        zeroes=findComplexZeroes(a,b,c,delta)
    return zeroes
```

Krok 5

Implementacja pod - zadania obliczania wartości Δ

```
def calculateDelta(a,b,c):
    return b*b - 4*a*c
```

Krok 6

Implementacja pod - zadania wyznaczania zer w dziedzinie liczb rzeczywistych

```
def findRealZeroes(a,b,c,delta):
    x1=(-b-math.sqrt(delta))/(2*a)
    x2=(-b+math.sqrt(delta))/(2*a)
    return (x1,x2)
```

Krok 7

Implementacja pod - zadania wyznaczania zer w dziedzinie liczb zespolonych

```
def findComplexZeroes(a,b,c,delta):
    import cmath
    x1=(-b-cmath.sqrt(delta))/(2*a)
    x2=(-b+cmath.sqrt(delta))/(2*a)
```

Krok 8

Implementacja pod - zadania wyświetlania wyników

```
def displayZeroes(zeroes):
    print(zeroes)
```

Programowanie strukturalne - przykład - wyniki działania

```
$python3 example_8.py 1 0 0
(-0.0, 0.0)

$python3 example_8.py -2 -8 10
(1.0, -5.0)

$python3 example_8.py 2 2 2
((-0.5-0.8660254037844386j), (-0.5+0.8660254037844386j))
```