Paradygmaty Programowania

Programowanie funkcyjne

dr inż. Marcin Bączyk

Wykład 4

21 marca 2022

Treść dzisiejszego wykładu

- programowanie deklaratywne
- charakterystyka
- funkcje w ujęciu matematycznym
- funkcje w ujęciu programistycznym
- funkcje wyższych rzędów
- rachunek λ i domknięcia
- rekurencja

Literatura

- Mark Richards, Neal Ford: “Podstawy architektury oprogramowania dla inżynierów”
- Robert C. Martin: “Czysta architektura. Struktura i design oprogramowania. Przewodnik dla profesjonalistów”
- Joshua Backfield: “Programowanie funkcyjne. Krok po kroku”
- Čukić Ivan: “Programowanie funkcyjne w języku C++. Tworzenie lepszych aplikacji”

Zmienna a funkcja

Zmienna

- przechowuje wartość,
- posiada **tożsamość** (unikatowy adres),

```
x = 5
hex(id(x))
```

```
'0x953ec0'
```

- może być argumentem przypisania,
- może być argumentem funkcji,
- może być zwracana przez funkcję

```
def kwadrat(x):
    y = x*x
    return y
```

Funkcja

- przetwarza dane,
- posiada tożsamość,

```
hex(id(kwadrat))
```

```
'0x7f5aed4b7940'
```

```
print(kwadrat)
```

```
<function kwadrat at 0x7f5aed4b7940>
```

- może być argumentem przypisania,

```
from math import sin, pi
mój_sinus = sin
mój_sinus(pi/4)
```

```
0.7071067811865475
```

- może być argumentem innej funkcji,
- może być zwracana przez inną funkcję

Funkcje wyższego rzędu

- **Funkcja wyższego rzędu**: jej argumentami są inne funkcje
- Przykład: funkcja `map(f, k)`
  - wywołuje funkcję `f` dla kolejnych elementów `x` kolekcji `k`

```
def kwadrat(x):
    return x*x

list(map(kwadrat, [2, 3, 4]))
```

```
[4, 9, 16]
```

- Przykład: funkcja `filter(f, k)`
  - zwraca te elementy `x` kolekcji `k`, dla których `f(x)` zwraca `True`

```
def parzysta(x):
    return not x%2

list(filter(parzysta, [2, 3, 4]))
```

```
[2, 4]
```

Wyrażenia lambda

- Funkcję definiujemy, ponieważ:
  1. zamierzamy wielokrotnie wykonywać tę samą operację i/lub
  2. funkcja wyższego rzędu wymaga argumentu w postaci funkcji
- A jeśli zachodzi ②, ale nie ①?
  - Warto użyć **wyrażenia lambda** – rodzaju funkcji anonimowej
- Przykład: bez wyrażenia lambda

```
def kwadrat(x):
    return x*x

list(map(kwadrat, [2, 3, 4]))
```

```
[4, 9, 16]
```

- To samo z wyrażeniem lambda

```
list(map(lambda x: x*x, [2, 3, 4]))
```

```
[4, 9, 16]
```

Czyste funkcje

- Wynik działania zależy wyłącznie od przekazanych argumentów
  - `time.time()` nie jest czystą funkcją
- Brak **skutków ubocznych**:
  - nie modyfikują swoich argumentów (w taki sposób, by było to widoczne poza ich ciałem),
  - nie modyfikują zmiennych globalnych,
  - nie alokują zasobów (np. pamięci).
- Zatem dla czystej funkcji:
  - jedynym wynikiem działania jest zwrócenie nowego obiektu,
  - wielokrotne wywołanie z tym samym argumentem da identyczny wynik
- Stosowanie wyłącznie czystych funkcji ułatwia:
  - zrozumienie kodu,
  - wykazanie poprawości działania,
  - szukanie przyczyn złego działania
- Kolejność wywołania kilku funkcji operujących na tych samych *niemutowalnych* danych nie ma znaczenia
  - łatwiejsza optymalizacja i/lub zrównoleglenie obliczeń

Programowanie funkcyjne – dlaczego warto je poznać?

- Rozwija intelektualnie
- Języki funkcyjne są niszowe/elitarne
  - znajomość Haskella, Erlanga, Elixira itd. to dodatkowy atut
- Najważniejsze techniki "funkcyjne" są stosowane w większości współczesnych języków programowania:
  - rekurencja,
  - rachunek lambda,
  - funkcje wyższego rzędu,
  - zmienne niemutowalne
- Języki "wbudowane" w pewne narzędzia miewają charakter funkcyjny:
  - np. język Scheme do opisu geometrii w programie do modelowania struktur półprzewodnikowych (tranzystory, lasery itp.) Sentaurus Structure Editor

Programowanie funkcyjne

Programowanie deklaratywne

- Programista opisuje (deklaruje) co ma być zrobione (jaki ma być efekt działania programu), a nie w jaki sposób ma to być zrobione (jaka jest sekwencja działań prowadząca do celu).
- Charakteryzuje się minimalną ilością skutków ubocznych (co w znaczący sposób upraszcza opracowywanie programów współbieżnych).

Program deklaratywny

- Jest niezależny – wynik końcowy nie zależy od żadnego zewnętrznego stanu.
- Jest bezstanowy – nie posiada stanu wewnętrznego, który który mógłby się zmieniać między wywołaniami.
- Jest deterministyczny – dla takich samych argumentów wejściowych **zawsze** zwraca ten sam wynik.

Programowanie funkcyjne

Programowanie funkcyjne jest przykładem programowania deklaratywnego. Problem (lub ogólniej świat) modelowany jest przy pomocy pojęć matematycznych. Wraz z zaawansowaniem modelu wykorzystywane struktury matematyczne, przyjmujące postać stałych i funkcji również stają się coraz bardziej zaawansowane.

Czyste programowanie funkcyjne polega tworzeniu coraz bardziej złożonych funkcji i stałych oraz wyznaczaniu wartości zmiennych.

Charakterystyka programowania funkcyjnego

- brak przypisań $\longrightarrow$ stałe i funkcje, brak zmiennych
- brak pętli $\longrightarrow$ rekurencja
- relatywnie łatwo napisać poprawny program
- łatwiej wykazać poprawność programu
- czyste funkcje (bez skutków ubocznych)
- funkcje wyższych rzędów

Programowanie funkcyjne

Deklaratywny charakter

- Nacisk kładziony jest na opisanie wyniku działania każdego fragmentu programu.
- Często konkretny problem przedstawiany jest jako szczególny przypadek bardziej ogólnego zadania.

Przykład – wyznaczanie sumy elementów w kolekcji

Podejście imperatywne:

- Dodawanie kolejnych elementów do wyniku
- Zgodne ze sposobem w jaki działają współczesne procesory, które wykorzystują głównie proste operacje arytmetyczne oraz instrukcje skoków.

Podejście deklaratywne:

- Suma dowolnego elementu zbioru oraz sumy pozostałych jego elementów.
- Mimo że wydaje się być mniej intuicyjna, to de facto jest bliższa rzeczywistości.

Programowanie funkcyjne

Języki funkcyjne i programowanie funkcyjne

- Posiadają instrukcje pozwalające w łatwy sposób wyrażać idee wysokiego poziomu w oderwaniu od konkretnej architektury systemu.
- Programy ze względu na wykorzystywane konstrukcje zazwyczaj są wykonują się wolniej niż ich imperatywne odpowiedniki.
- Pierwotnie programowanie funkcyjne nie cieszyło się one dużą popularnością i stanowiło raczej ciekawe zagadnienie do badania dla naukowców.
- Obecnie coraz częściej stosowane do wielu zagadnień biznesowych (np. w architekturze bazującej na mikrousługach).
- Na przestrzeni lat powstało wiele udanych implementacji języków funkcyjnych.
- Programy napisane fukcyjnie charakteryzują się dużą czytelnością i efektywnością powstawania kodu.
- Czas wykonywania programu i pamięć niezbędna do jego uruchomienia, ze względu na olbrzymi rozwój systemów komputerowych, nie jest głównym ograniczeniem, jakie należy rozważać.

Funkcje w ujęciu matematycznym

- funkcja prosta

$$f(x) = ax^2 + bx + c$$

- funkcja złożona

$$f(x, g, h) = g(x) \cdot h(x)$$

- funkcja rekurencyjna, instrukcja warunkowa

$$n! := \begin{cases} 1, & \text{dla } n = 0 \\ n \cdot (n-1)!, & \text{dla } n >= 1. \end{cases}$$

- suma / iloczyn, pętla

$$n! = \prod_{k=1}^{n} k$$

Funkcje w ujęciu programistycznym

Funkcje (lub procedury) to inaczej podprogramy odpowiedzialne za wyodrębniony element programu. Podobnie do matematycznych odpowiedników w programowaniu również wyróżnia się kilka rodzajów funkcji:

- funkcja prosta - funkcja złożona z instrukcji wykonywanych sekwencyjnie.
- funkcja złożona - funkcja wywołująca inne funkcje w celu wyznaczenia niezbędnych wartości
- funkcja rekurencyjna - funkcja, wywołująca samą siebie, niezbędna jest instrukcja warunkowa w celu zakończenia ciągu wywołań

- Wiele funkcji, nie tylko matematycznych, daje się przedstawić zarówno w postaci rekurencyjnej jak i sumy lub iloczynu wyrażeń.
- W programowaniu funkcyjnym ze względu na prostotę wyrażenia i łatwość dowodzenia poprawności kodu preferowana jest postać rekurencyjna.

$$n! = \prod_{k=1}^{n} k$$

Przedstawienie wzoru na obliczenie silni liczby naturalnej ($n!$) w postaci iloczynu kolejnych liczb dodatnich mniejszych lub równych n jest bliższe imperatywnemu podejściu do programowania. Definicja w tej postaci dokładnie prezentuje w jaki sposób należy wykonać obliczenia w celu wyznaczenia wyrażenia. W trakcie jej obliczania aktualizowana jest wartość iloczynu, co jest sprzeczne z założeniami programowania funkcyjnego (bezstanowy). Stanem w tym przypadku można określić wartość zmiennej przechowującej kolejne wartości iloczynu.

$$n! := \begin{cases} 1, & \text{dla } n = 0 \\ n \cdot (n-1)!, & \text{dla } n >= 1. \end{cases}$$

Z kolei przedstawienie wzoru na obliczenie silni liczby naturalnej ($n!$) w postaci iloczynu wartości $(n-1)!$ oraz n jest bliższe deklaratywnemu podejściu do programowania. Wyrażenie tej postaci odwołuje się do matematycznej definicji czym jest silnia. Warto zwrócić uwagę, że bezpośrednio w definicji nie jest przedstawiony sposób obliczenia wyważenia $(n-1)!$. Program nie ma też zmiennej, która mogłaby być aktualizowana.

funkcje wyższego rzędu

W programowaniu funkcyjnym funkcje są takimi samymi wartościami jak wszystkie inne. Oznacza to, że funkcje mogą być:

- argumentami innych funkcji oraz
- wynikami zwracanymi przez inne funkcje.

Funkcje, które operują na innych funkcjach nazywamy funkcjami wyższych rzędów.

polimorfizm funkcji

Zarówno funkcje przekazywane jako argumenty do innych funkcji jak i funkcje przez nie zwracane mogą mieć różną postać. Na przykład istnieje nieskończenie wiele funkcji mapujących zbiór liczb rzeczywistych w zbiór liczb zespolonych ($g : \mathbb{R} \mapsto \mathbb{C}$). Jeżeli dana funkcja $f(x, g) : \mathbb{R} \mapsto \mathbb{R}$ oczekuje takiej funkcji przekazanej jako argument to może ona być dowolnej postaci. Taką sytuację nazywamy **polimorfizmem** – wielopostaciowością.

Rachunek λ

Mając daną funkcję złożoną

$$f(x, g, h) = g(x) \cdot h(x)$$

możemy chcieć w miejsce argumentów g lub h przekazać funkcję bez jej formalnego definiowania W tym celu wykorzystywany jest rachunek lambda . Funkcje f, g oraz h są formalnie zdefiniowane, mają swoją nazwę i prawdopodobnie adres w pamięci fizycznej. Często w przypadku bardzo prostych wyrażeń, takich jak na przykład wielomiany niewielkich stopni, lepiej jest nie definiować funkcji a utworzyć są ad hoc i przekazać jako argument wprost w wywołaniu. Matematycznie można zapisać to w następujący sposób:

$$f(10, y \mapsto y^2, y \mapsto 1 - \frac{1}{y}) = (y \mapsto y^2)(10) \cdot (y \mapsto 1 - \frac{1}{y})(10)$$

Domknięcia

Domknięciem nazywamy funkcję λ, która odwołuje się do zmiennych spoza zakresu funkcji:

$$f(10, y \mapsto y^2, y \mapsto 1 - \frac{1}{y} + c) = (y \mapsto y^2)(10) \cdot (y \mapsto 1 - \frac{1}{y} + c)(10).$$

W powyższym przykładzie zmienna c nie występuje jako argument funkcji λ. Nie została też zdefiniowana wewnątrz jej ciała. Zmienna ta została przechwycona, a utworzona funkcja wywoła się w sposób poprawny.

Funkcje λ oraz domknięcia nie są implementowane we wszystkich językach, zwłaszcza niefunkcyjnych. W niektórych językach programowania funkcjonalności te mogą być mocno ograniczone zwłaszcza w przypadku zwracania funkcji jako argumentu.

Należy zwrócić uwagę czy odwołanie do przechwytywanej zmiennej (lub funkcji) w danej funkcji λ odbywa się przez kopiowanie czy referencję. Nie w każdym języku możliwe jest utrzymanie zmiennej poza ciałem funkcji w przypadku istniejących do niej innych odwołać (np. przez domknięcie).

Zastosowania procedur wyższych rzędów

- Niektóre pojęcia matematyczne w sposób naturalny operują na funkcja wyższych rzędów.
  Przykład : Funkcja wyznaczająca pochodną funkcji jako argument przyjmuje funkcję i zwraca funkcję w innej postaci.
- W wielu miejscach w programie powtarzany jest ten sam kod, różniący się jedynie fragmentami. Fragmenty te mogą być ujęte w postaci procedur, które następnie będą parametrami funkcji bardziej ogólnych.
  Przykład : Procedura sortowania rekordów może się różnić fragmentem dotyczącym porównania dwóch elementów, zaś sama implementacja algorytmu będzie taka sama we wszystkich przypadkach. Mechanizm porównywania wartości można przekazać do procedury w postaci osobnej funkcji.

Skutki uboczne

Jednym z podstawowych założeń programowania funkcyjnego jest minimalizacja skutków ubocznych działania wszystkich funkcji. Z punktu widzenia programowania funkcyjnego niepożądane skutki uboczne jakie mogą się zdarzyć to między innymi:

- zmiana wartości zmienne globalnej,
- zmiana wartości argumentu,
- przydzielenie zasobu, i jego późniejsze niezwrócenie.

Wszystkie powyższe przypadki, to niekorzystne skutki uboczne, utrudniające czytanie i rozumienie kodu programu. Odczytując instrukcję zakładamy, że nie zmienia ona stanu programu, który mógłby zaburzyć działanie kolejnych instrukcji.

Skutki uboczne, jakimi są zmiany stanu systemu powodują, również, że dwukrotne wywołanie tej samej funkcji nie musi zakończyć się tym samym wynikiem dla tych samych danych wejściowych.

Skutki uboczne

Jednak programy z definicji nie mogą istnieć bez skutków ubocznych. Nie wszystkie skutki uboczne są niepożądane. Poniżej kilka przykładów „przydatnych" skutków ubocznych działania funkcji:

- wyświetlenie informacji na ekranie lub wyczyszczenie ekranu,
- zapisanie danych do pliku, bazy danych lub wysłanie przez sieć,
- zmiana pola w rekordzie danych.

Funkcje i procedury, które zmieniają stan systemu są niezbędne do poprawnego działania programu. Bez jakiejkolwiek formy wyświetlenia wyniku, sam wynik jest bezużyteczny. Należy jednak zwrócić uwagę by funkcje, które mają swoje skutki uboczne, informowały o tym poprzez swoją nazwę, np. *wyświetl*, *wyczyśćEkran* lub *zapiszWynikNaDysku*.

Zmienne niemutowalne

Zmienne w systemie komputerowym mogę być:

- mutowalne oraz
- niemutowalne.

Zazwyczaj poprzez pojęcie zmiennej rozumie się wartość, która może ulega zmianie w trakcie swojego życia. Natomiast w programowaniu funkcyjnym przyjmuje się zmienne są niemutowalne. Oznacza to, że:

- zmienne się nie zmieniają w czasie swojego życia oraz, że
- jedynie zmienne globalne mogą zmieniać swoje referencje.

W przypadku zmiany referencji zmiennej globalnej poprzedni obiekt zostaje zniszczony, a w jego miejsce utworzony nowy.

Niemutowanie zmiennych ułatwia utrzymanie ich prawidłowego stanu programu. Zmienne utworzone w prawidłowych sposób i zawierające prawidłowe dane nie zmienią tego stanu.

Przykład

Wśród grupy studentów, ze wszystkich występujących imion i nazwisk, tylko niektóre ich kombinacje oznaczają konkretne osoby. Przez funkcję starosty rozumiemy osobę występującą w imieniu całej grupy studentów. W tym przykładzie funkcję tę należy rozumieć jako referencję do zmiennej globalnej, gdyż nie jest to cecha żadnej z osób w grupie.

Jeżeli zmianie ulegać będą kolejno imię i nazwisko starosty w pewnym etapie (dokładnie pomiędzy zmianą imienia a nazwiska) program znajdzie się w nieprawidłowym stanie, gdyż jako starosta będzie przypisana osoba, która nie istnieje – imię i nazwisko nie będzie identyfikowało żadnej osoby z grupy.

Natomiast, jeżeli utworzymy nowy rekord (wraz z imieniem i nazwiskiem nowego starosty) a następnie zostanie on wskazany jako nowy starosta program przez cały czas będzie w stanie prawidłowym.

Rekurencja

Ponieważ w programowaniu funkcyjnym zmienne są niemutowalne to konstrukcje takiej jak pętle nie mają zastosowania w prost. Dzieje się tak, że zazwyczaj pętle w trakcie kolejnych iteracji aktualizują zmienne lokalne, by następnie zwrócić wynik. Aby móc rozwiązać ten problem należy posłużyć się rekurencją.

Listy

W przypadku rekurencji i operacji na zbiorach istotnym pojęciem są listy. Z punktu widzenia programowania funkcyjnego listą może być dowolny typ zbioru, który umożliwia wyznaczenie:

- głowy listy oraz
- ogona listy.

Poprzez głowę listy będziemy rozumieli pierwszy jej element, zaś jako ogon listy wszystkie pozostałe.

Listy i rekurencja

Wykorzystując pojęcie listy, operacje wykonywane na zbiorach elementów można przedstawić w postaci ogólnego algorytmu:

1. Weź pierwszy element zbioru. Czasami może to być też dowolny element tego zbioru.
2. Wykonaj żądaną operację na elemencie.
3. Scal wynik operacji z wynikiem wywołania algorytmu dla pozostałych elementów ze zbioru.

Przykład

Sumę zbioru liczbowego można przedstawić jako sumę dowolnego jego elementu oraz sumę wszystkich pozostałych. Podobnie znajdowanie elementu maksymalnego można przedstawić jako wynik porównania danego, dowolnie wybranego elementy z wartością maksymalną pozostałej części zbioru.

Przykład

Filtrowanie zbioru elementów, ze względu na funkcję celu można przedstawić jako scalenie wyniku zwracanego dla pojedynczego wyniku, który może być zbiorem pustym, z wynikiem zwracanym dla pozostałych elementów zbioru, który również może być pusty.

Przykładowy problem programistyczny

Wyznaczyć średnią ilość godzin w tygodniu poświęconych na naukę wśród studentów, którzy mają średnią ważoną ze wszystkich przedmiotów wyższą lub równą 4.0.