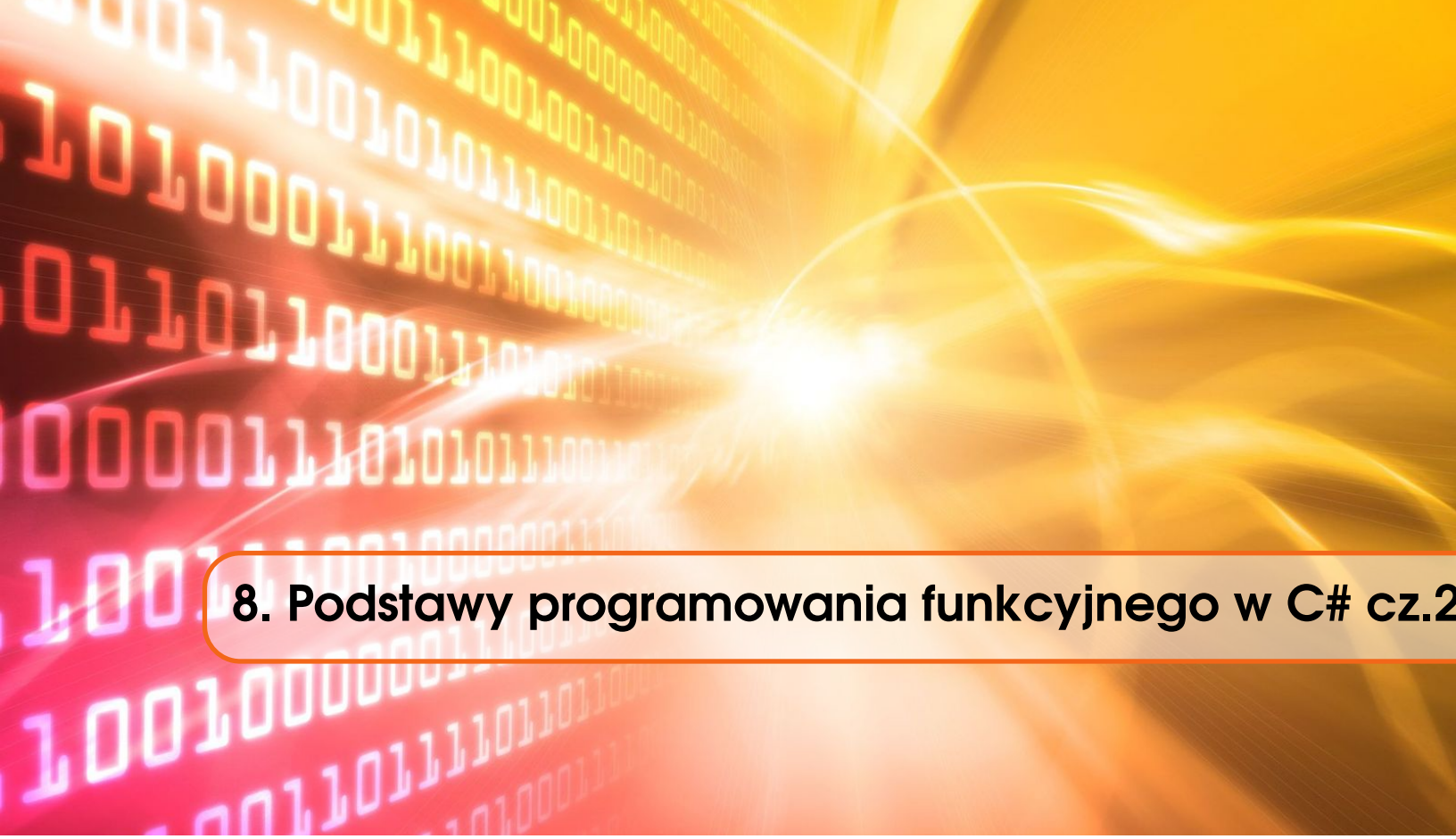
Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk



8. Podstawy programowania funkcyjnego w C# cz.2

8.1 Problem do rozwiązania

Załóżmy, że chcemy napisać program, który pozwoli nam z listy elementów wybierać te, które spełniają określone przez nas warunki. Najprostsze możliwe rozwiązanie przedstawia się następująco:

```
class OperacjeNaLiscie {
  public static List<string> TylkoDlugieNazwy(List<string> lista)
  {
    List<string> rezultat = new List<string>();
    for(int i=0; i<lista.Count; i++)
      if(lista[i].Length > 18)
        rezultat.Add(lista[i]);
    return rezultat;
  }

  public static List<string> TylkoNaLitereP(List<string> lista)
  {
    List<string> rezultat = new List<string>();
    for(int i=0; i<lista.Count; i++)
      if(lista[i][0] == 'P')
        rezultat.Add(lista[i]);
    return rezultat;
  }
}

class Program
{
  static void Main(string[] args)
```

```
    {
      List<string> przedmioty = new List<string>() {
        "Programowanie funkcyjne",
        "Programowanie stron internetowych",
        "Algebra",
        "Logika dla informatykow",
        "Sieci komputerowe"
      };

      var dlugieNazwy = OperacjeNaLiscie.TylkoDlugieNazwy(przedmioty);
      var przedmiotyNaLitereP = OperacjeNaLiscie.TylkoNaLitereP(przedmioty);
    }
  }
```

Jak łatwo zauważyć, znowu mamy bardzo dużą nadmiarowość w klasie OperacjeNaLiscie. Zauważmy, że dotyczy ona sposobu poruszania się po liście, który jest taki sam i niezmienny w obu przypadkach. To co różni funkcje w tej klasie to warunek wyboru elementu. Jeżeli go odseparujemy uzyskamy ogólny algorytm filtrowania listy. Będzie to jednak realizowane inaczej w podejściu obiektowym i funkcyjnym. Przyjrzyjmy się obu.

8.2 Podejście obiektowe

Uzyskamy to przenosząc sprawdzanie warunków do osobnych klas:

```
class TylkoDlugieNazwy {

  public bool Sprawdz(string wartosc)
     => wartosc.Length > 18;
}

class TylkoNaLitereP {

  public bool Sprawdz(string wartosc)
     => wartosc[0] == 'P';
}
```

Możemy teraz utworzyć instancje tych klas i poprzez wywołanie metody Sprawdz sprawdzać spełnienie zdefiniowanych warunków. Jeżeli jednak będziemy chcieli przekazać te obiekty do innej metody powyższe klasy muszą mieć jakiś element, który je połączy. Ponieważ jedyną częścią wspólną tych klas jest deklaracja metody Sprawdz, najwygodniej dla nas będzie tę część wspólną zdefiniować jako interfejs:

```
interface IWarunek<TWartosc> {
  bool Sprawdz(TWartosc wartosc);
}
```

Dzięki niemu ogólny algorytm filtrowania listy możemy zapisać w następujący sposób:

```
class OperacjeNaLiscie {
  public static List<T> Filtruj<T>(
```

```
        List<T> lista,
        IWarunek<T> warunek)
    {
      List<T> rezultat = new List<T>();
      for(int i=0; i<lista.Count; i++)
        if(warunek.Sprawdz(lista[i]))
          rezultat.Add(lista[i]);
        return rezultat;
    }
  }
```

Powyższe klasy możemy wykorzystać w następujący sposób:

```
class Program
{
  static void Main(string[] args)
  {
    List<string> przedmioty = new List<string>() {
      //zawartosc kolekcji jest taka
      //sama jak we wczesniej
    };

    var sprawdzDlugie = new TylkoDlugieNazwy();
    var sprawdzNaLitereP = new TylkoNaLitereP();

    var dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzDlugie);
    var przedmiotyNaLitereP
      = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzNaLitereP);
  }
}
```

Dzięki zastosowaniu strategii, jeżeli będziemy chcieli dołączyć do naszej aplikacji nowy sposób filtrowania listy, to musimy tylko utworzyć nową klasę, która będzie definiowała odpowiedni warunek. Przykładowo:

```
class TylkoNaLitereA : IWarunek<string> {

  public bool Sprawdz(string wartosc)
      => wartosc[0] == 'A';
}
```

Jeżeli teraz porównamy klasy TylkoNaLiterA i TylkoNaLitereP zobaczymy, że znowu są one bardzo do siebie podobne. W tym przypadku różnią się tylko i wyłącznie literą, która jest porównywana z pierwszym znakiem z łańcucha. Oczywiście i ten problem możemy łatwo rozwiązać. Musimy to zrobić poprzez wprowadzenie pola klasy i konstruktora[1]:

```
class TylkoNaLitere : IWarunek<string> {

  char znak;
```

[1]Z oczywistych względów nie możemy tego zrobić poprzez parametr metody Sprawdz

```
  public TylkoNaLitere(char znak)
  {
    this.znak = znak;
  }

  public bool Sprawdz(string wartosc)
      => wartosc[0] == znak;
}
```

Całość kodu przedstawia się następująco:
Powyższą klasę możemy wykorzystać w następujący sposób:

```
class Program
{
  static void Main(string[] args)
  {
    List<string> przedmioty = new List<string>() {
      //zawartosc kolekcji jest taka
      //sama jak we wczesniej
    };

    var sprawdzDlugie = new TylkoDlugieNazwy();
    var sprawdzNaLitereA = new TylkoNaLitere('A');
    var sprawdzNaLitereP = new TylkoNaLitere('P');

    var dlugieNazwy
        = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzDlugie);
    var przedmiotyNaLitereP
        = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzNaLitereP);
    var przedmiotyNaLitereA
        = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzNaLitereA);
  }
}
```

Zbierając wszystkie fragmenty razem uzyskamy następujący kod:

```
interface IWarunek<TWartosc> {
  bool Sprawdz(TWartosc wartosc);
}

class OperacjeNaLiscie {
  public static List<T> Filtruj<T>(
    List<T> lista,
    IWarunek<T> warunek)
  {
    List<T> rezultat = new List<T>();
    for(int i=0; i<lista.Count; i++)
      if(warunek.Sprawdz(lista[i]))
        rezultat.Add(lista[i]);
    return rezultat;
  }
}
```

```
class TylkoNaLitere : IWarunek<string> {
  char znak;

  public TylkoNaLitere(char znak) {
    this.znak = znak;
  }

  public bool Sprawdz(string wartosc)
    => wartosc[0] == znak;
}

class TylkoDlugieNazwy {
  public bool Sprawdz(string wartosc)
    => wartosc.Length > 18;
}

class Program
{
  static void Main(string[] args) {
    List<string> przedmioty = new List<string>() {
      "Programowanie funkcyjne",
      "Programowanie stron internetowych",
      "Algebra",
      "Logika dla informatykow",
      "Sieci komputerowe"
    };

    var sprawdzDlugie = new TylkoDlugieNazwy();
    var sprawdzNaLitereA = new TylkoNaLitere('A');
    var sprawdzNaLitereP = new TylkoNaLitere('P');

    var dlugieNazwy
       = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzDlugie);
    var przedmiotyNaLitereP
       = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzNaLitereP);
    var przedmiotyNaLitereA
       = OperacjeNaLiscie.Filtruj(przedmioty, sprawdzNaLitereA);
  }
}
```

8.3 Podejście funkcyjne

Jak mam nadzieję pamiętacie, w podejściu funkcyjnym, na problemy z nadmiarowością mieliśmy jedno podstawowe rozwiązanie - funkcje. Teraz również możemy je zastosować. Ponieważ jednak C# jest językiem o statycznym systemie typów, najpierw musimy poznać typ reprezentujący funkcje. Jest to trochę bardziej skomplikowane niż w przypadku F# , gdyż C# dopuszcza istnienie komend, czyli funkcji typu void. Dlatego typ funkcyjny jest w C# zdefiniowany osobno dla funkcji oraz komend. Te pierwsze są

reprezentowane poprzez delegaty Func<>, a drugie przez delegaty Action<>. Domyślnie są one zdefiniowane dla podprogramów przyjmujących do 16 parametrów: Chodź

Funkcje	Komendy
Func<out TResult>	
Func<in T1, out TResult>	Action<in T1>
Func<in T1, in T2, out TResult>	Action<in T1,in T2>
Func<in T1, in T2,...,in T16, out TResult>	Action<in T1,in T2,...,in T16>

możliwe jest stworzenie własnej wersji takiego delegata dla większej liczby parametrów, to z oczywistych względów nie jest to zalecane.

Zastosowanie typu Func<> pozwoli nam na zapisanie zdefiniowanej wcześniej funkcji Filtruj w następujący sposób:

```
class OperacjeNaLiscie {
  public static List<T> Filtruj<T>(
    List<T> lista, Func<T, bool> warunek)
  {
    List<T> rezultat = new List<T>();
    for(int i=0; i<lista.Count; i++)
      if(warunek(lista[i]))
        rezultat.Add(lista[i]);
    return rezultat;
  }
}
```

Dzięki temu nie trzeba już definiować żadnego dodatkowego interfejsu, czy klasy bazowej. Musimy jedynie zdefiniować funkcje, które będą określały nasz warunek:

```
class Program {
  static bool TylkoDlugie(string wartosc) => wartosc.Length > 18;
  static bool TylkoNaP(string wartosc) => wartosc[0] == 'P';

  static void Main(string[] args)  {
    List<string> przedmioty = new List<string>() {
      "Programowanie funkcyjne",
    ...
    };

    List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoDlugie);
    List<string> przedmiotyNaLitereP
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaP);
  }
}
```

Zauważmy, że podczas przekazywania funkcji jako parametr podajemy tylko jej nazwę.

Podobnie jak w poprzednim punkcje, będziemy chcieli zapisać bardziej ogólną wersję funkcji TylkoNaP, która pozwalali nam porównać pierwszą literę łańcucha znaków

z dowolnym znakiem. Oczywiście podobnie jak poprzednio nie możemy dołożyć dodatkowego parametru do tej funkcji, ponieważ będzie to niezgodne z typem delegatu, jaki zdefiniowaliśmy w metodzie Filtruj. Możemy jednak wykorzystać fakt, że funkcja wyższych rzędów może nie tylko przyjmować inną funkcję jako parametr, ale również funkcje zwracać. Pozwoli nam to rozwiązać ten problem w następujący sposób:

```
class Program {
  static Func<string,bool> TylkoNaLitere(char litera)
    => (string wartosc) => wartosc[0] == litera;

  static void Main(string[] args)  {
    List<string> przedmioty = new List<string>() {
      "Programowanie funkcyjne",
      ...
    };

    List<string> przedmiotyNaLitereP
        = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaLitere('P'));
    List<string> przedmiotyNaLitereA
        = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaLitere('A'));
  }
}
```

Całość aplikacji będzie przedstawiała się w następujący sposób:

```
class OperacjeNaLiscie {
  public static List<T> Filtruj<T>(
    List<T> lista, Func<T, bool> warunek)
  {
    List<T> rezultat = new List<T>();
    for(int i=0; i<lista.Count; i++)
      if(warunek(lista[i]))
        rezultat.Add(lista[i]);
    return rezultat;
  }
}

class Program {
  static bool TylkoDlugie(string wartosc) => wartosc.Length > 18;

  static Func<string,bool> TylkoNaLitere(char litera)
    => (string wartosc) => wartosc[0] == litera;

  static void Main(string[] args)  {
    List<string> przedmioty = new List<string>() {
      "Programowanie funkcyjne",
      ...
    };

    List<string> dlugieNazwy
      = OperacjeNaLiscie.Filtruj(przedmioty, TylkoDlugie);
    List<string> przedmiotyNaLitereP
```

```
        = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaLitere('P'));
      List<string> przedmiotyNaLitereA
        = OperacjeNaLiscie.Filtruj(przedmioty, TylkoNaLitere('A'));
    }
  }
```

R W przeciwieństwie do F# w C# nie istnieje mechanizm automatycznego Curringu. Dlatego zamiany funkcji wieloargumentowej na ciąg funkcji jednego argumentu musimy dokonywać ręcznie. Możemy co prawda napisać funkcje (a raczej cały zestaw funkcji), które będą ten mechanizm ukrywały np:

```
public static Func<T1, Func<T2, TWynik>>
   Curry<T1, T2, TWynik>(Func<T1, T2, TWynik> funkcja)
      => (T1 v1) => (T2 v2) => funkcja(v1, v2);
```

Jednak ich wykorzystanie jest trochę kłopotliwe w przypadku funkcji nazwanych. Gdybyśmy chcieli wykorzystać powyższą funkcję do rozwiązania problemu sparametryzowania funkcji TylkoNaLitereA, w następujący sposób:

```
class Program {
  static bool TylkoNaLitere(char znak, string wyraz)
    => wyraz[0] == znak;

  static void Main(string[] args) {
    var curried = Curry(TylkoNaLitere);
  }
```

uzyskamy błąd, ponieważ C# nie jest wstanie rozpoznać automatycznie wartości zmiennych typów, jakie należy wykorzystać:

```
class Program
{
    static bool TylkoNaLitere(char znak, string wyraz)
        => wyraz[0] == znak;

    public static Func<T1, Func<T2, TWynik>> Curry<T1, T2, TWynik>(Func<T1, T2, TWynik> funkcja)
        => (T1 v1) => (T2 v2) => funkcja(v1, v2);

    static void Main(string[] args)
    {

        var curried = Curry(TylkoNaLitere);

    }
}
```

Func<T1, Func<T2, TWynik>> Program.Curry<T1, T2, TWynik>(Func<T1, T2, TWynik> funkcja)

CS0411: The type arguments for method 'Program.Curry<T1, T2, TWynik>(Func<T1, T2, TWynik>)' cannot be inferred from the usage. Try specifying the type arguments explicitly.

Problem ten możemy rozwiązać w dwojaki sposób. Albo podając jawnie argumenty typu przy wywołaniu metody Curry, albo przypisać najpierw funkcję do zmiennej:

```
static void Main(string[] args)
{
  var curried1 = Curry<char, string, bool>(TylkoNaLitere);
```

```
    Func<char, string, bool> funkcja = TylkoNaLitere;
    var curried2 = Curry(funkcja);
  }
```

8.4 Obliczenia leniwe w C#

Wywołując metodę w C# najpierw muszą być wyznaczone wartości wszystkich jej argumentów. Oznacza to, że C# domyślnie wykonuje obliczenia w sposób zachłanny. Przykładowo po wykonaniu programu:

```
class Program
{
  static int Funkcja1() {
    Console.WriteLine(nameof(Funkcja1));
    return 1;
  }

  static int Funkcja2() {
    Console.WriteLine(nameof(Funkcja2));
    return 2;
  }

  static int Funkcja3(bool warunek, int wartosc1, int wartosc2)
  {
    return warunek ? wartosc1 : wartosc2;
  }

  static void Main(string[] args) {
    var wynik = Funkcja3(true, Funkcja1(), Funkcja2());
    Console.WriteLine(wynik);
    var wynik = Funkcja3(false, Funkcja1(), Funkcja2());
    Console.WriteLine(wynik);
  }
}
```

Na ekranie zobaczymy:

```
Funkcja1
Funkcja2
1
Funkcja1
Funkcja2
2
```

Zwróćmy uwagę, że podczas obu wywołań metody Funkcja3 uruchamiane są obie metody Funkcja1 i Funkcja2. Jeżeli jednak przyjrzymy się definicji metody Funkcja3

zauważymy, że w przypadku pierwszego jej wykonania wartość przekazana jako parametr wartosc2 będzie pominięta. Podobnie w drugim wywołaniu pomięta będzie wartość parametru wartosc1. Można więc powiedzieć, że w pierwszym wywołaniu metoda Funkcja2, a w drugim Funkcja1 są uruchamiane niepotrzebnie. W pokazanym przykładzie nie ma to większego znaczenia, ale będzie już miało, w przypadku gdy metoda, która jest zbędnie uruchamiana zakończy się błędnie lub jej wykonanie będzie trwało bardzo długo.

Rozwiązaniem może być zastosowanie w naszej aplikacji obliczeń leniwych. Obliczenia takie wykonywane są dopiero w momencie, gdy ich wartość jest rzeczywiście konieczna. W C# są one reprezentowane jako obiekty typu Lazy<>:

```
static int Funkcja6(int liczbaPowtorzen, Lazy<int> wartosc) {

  int suma = 0;
  for(int i=0; i<liczbaPowtorzen; i++) {
    suma += wartosc.value;
  }

  return suma;

}
```

8.5 Zadania do samodzielnego wykonania

Zadanie 8.1 Napisz funkcję Odwroc, która tworzy nową funkcję o odwrotnej kolejności argumentów. Poniższy kod powinien wyświetlić w konsoli "OK".

```
Func<double, double, double> funkcja1 = (a, b) => a / b;
var x1 = 10;
var x2 = 2;

var wynik1 = funkcja1(x1, x2);
var funkcja2 = Odwroc(funkcja1);
var wynik2 = funkcja2(x2, x1);
if (wynik1 == wynik2)
  Console.WriteLine("OK");
else
Console.WriteLine("Blad");
```

Zadanie 8.2 Napisz funkcję Zloz, która tworzy nową funkcję stanowiącą złożenie dwóch funkcji. Poniższy kod powinien wyświetlić w konsoli "OK":

```
Func<double, double> funkcja1 = a => a * a;
Func<double, double> funkcja2 = b => b * 2;

var funkcja3 = Zloz(funkcja1, funkcja2);
var wynik = funkcja3(10);
if (wynik == 200)
  Console.WriteLine("OK");
else
```

```
    Console.WriteLine("Blad");
```

Zadanie 8.3 Napisz funkcję, która przyjmuje dowolną wartość oraz zestaw dowolnej liczby funkcji, które ją przetworzą i zwrócą listę z wynikami. Przykładowe zastosowanie:

```
var wynik1 = Przetworz(1.0, Math.Sin, Math.Cos );
var wynik2 = Przetworz(2.0, x => $"{x}", x => $"Wynik {x}");
```

Zadanie 8.4 Napisz funkcję, która przyjmuje bezparametrową akcję (funkcję typu void) i wykonuje ją tyle razy ile wskazuje drugi parametr.

Zadanie 8.5 Napisz funkcję, która przyjmie akcję o jednym parametrze oraz liczbę powtórzeń. W jej rezultacie powinieneś dostać akcją, która pozwoli podać argument i wykona odpowiednie działania. Np. poniższy kod

```
var akcja = Powtorz<string>(10, Console.WriteLine);
akcja("Ala ma kota");
```

powinien wyświetlić na 10 razy na ekranie tekst ”Ala ma kota”.

Zadanie 8.6 Napisz program, w którym użytkownik będzie mógł wybrać jak chce wyświetlić listę dowolnych elementów. 1 - każdy element w osobnej linii rozpoczynający się od *, 2 - lista HTML, 3 - lista Markdown. Wykorzystaj wzorzec strategii.

Zadanie 8.7 Napisz program, w którym użytkownik będzie mógł wybrać jak chce wyświetlić listę dowolnych elementów. 1 - każdy element w osobnej linii rozpoczynający się od *, 2 - lista HTML, 3 - lista Markdown. Wykorzystaj funkcje wyższych rzędów.

Zadanie 8.8 Napisz klasę pozwalającą na utworzenie niemodyfikowalnej listy łączonej (nie wolno korzystać z istniejącej listy) przechowującej elementy dowolnego typu

Zadanie 8.9 Napisz funkcje realizujące mapowanie, filtrowanie utworzonej w poprzednim zadaniu listy.

Zadanie 8.10 Napisz funkcję realizującą wyszukiwanie wartości spełniającej określony warunek w utworzonej w zadaniu 3.8 liście.

Zadanie 8.11 Napisz funkcję, która zwróci nową funkcję realizująca memoryzację. Przykładowo:

```
Func<int, int> funkcja1 = x => x + 1;

var funkcja2 = Memoryzacja(funkcja1);

funkcja2(1);
funkcja2(1);
```

W obu przypadkach funkcja2 powinna zwrócić wynik wywołania funkcji, którą otrzymała jako parametr, ale funkcja ta powinna być wywołana tylko za pierwszym razem.