LABORATORIUM 4

TREŚCI KSZTAŁCENIA:

- PROGRAMOWANIE W F#
- NADAWANIE WARTOŚCI I KOMUNIKACJA Z UŻYTKOWNIKIEM. LICZBY I ŁAŃCUCHY ZNAKÓW. INSTRUKCJA LET. STOSOWANIE INSTRUKCJI WARUNKOWYCH I OPERATORÓW
- NIEZMIENNE STRUKTURY DANYCH

Spis treści

Wprowadzenie do F# 2
Środowiska programistyczne 2
Konfiguracja środowiska VSC 2
Konfiguracja środowiska Visual Studio 4
Wstęp do programowania w F# 5
Niezmienne Struktury Danych w F# 8
Przykłady Użycia Niezmiennych Struktur Danych 10
Przykład Kompleksowego Użycia Niezmiennych Struktur Danych 11
Zadania do samodzielnego rozwiązania 13

Wprowadzenie do F#

F# to funkcjonalny, obiektowy i imperatywny język programowania opracowany przez Microsoft, w ramach platformy .NET. Jego główne cechy to:

Kluczowe cechy F#:

1. Funkcjonalność: F# jest językiem funkcyjnym, co oznacza, że centralną rolę odgrywają funkcje i ich zastosowania. Programowanie funkcyjne pozwala na tworzenie programów, które są bardziej czytelne, mniej podatne na błędy i łatwiejsze do testowania.
2. Niezmienność: F# promuje korzystanie z wartości niezmiennych (immutable), co zmniejsza ryzyko błędów związanych z równoległością czy zmianami stanu.
3. Wysoka produktywność: Dzięki silnemu systemowi typów, automatycznej inferencji typów oraz idiomatycznej, zwięzłej składni, programy w F# mogą być pisane szybciej i z mniejszą liczbą błędów.
4. Obsługa imperatywna i obiektowa: Choć F# jest językiem funkcyjnym, wspiera również programowanie obiektowe i imperatywne. Można w nim definiować klasy, interfejsy, dziedziczenie i inne cechy programowania obiektowego.
5. Łatwość współpracy z .NET: Jako część platformy .NET, F# ma dostęp do szerokiego zestawu bibliotek, narzędzi i frameworków, takich jak ASP.NET, Entity Framework i wiele innych.
6. Asynchroniczność i współbieżność: F# wspiera asynchroniczność i współbieżność, co jest niezbędne do budowania wydajnych i skalowalnych aplikacji, zwłaszcza w środowiskach serwerowych.

Zastosowania F#:

- Obliczenia naukowe i analityczne: Dzięki łatwej pracy z danymi oraz bogatemu wsparciu dla obliczeń matematycznych, F# jest popularny w analizie danych.
- Finanse: F# często jest wykorzystywany w branży finansowej do budowania modeli obliczeniowych, analiz ryzyka oraz do złożonych symulacji.
- Aplikacje serwerowe: Można w nim tworzyć skalowalne aplikacje serwerowe, z wykorzystaniem takich technologii jak ASP.NET czy Azure Functions.

Środowiska programistyczne

Do programowania w F# można wykorzystać różne środowiska programistyczne, które oferują pełne wsparcie dla tego języka, szczególnie dzięki jego integracji z platformą .NET. Wśród najczęściej używanych środowisk wyróżnia się:

- Visual Studio – dla dużych, rozbudowanych projektów.
- Visual Studio Code z Ionide – dla lekkich, cross-platformowych projektów.
- JetBrains Rider – dla programistów poszukujących zaawansowanych narzędzi deweloperskich.
- LINQPad – dla szybkiego prototypowania i testów kodu.
- MonoDevelop – na macOS i Linux.

Konfiguracja środowiska VSC

W celu koniuracji środowiska należy:

1. Zainstalować Visual Studio Code.
2. Zainstalować .NET SDK.

3. Zainstalować Ionide (rozszerzenie do F#).
4. Stworzyć nowy projekt F# i uruchom go.
5. Dostosować środowisko, korzystając z dodatkowych narzędzi i konfiguracji.

Po skonfigurowaniu tego środowiska, posiada się pełne wsparcie dla programowania w F# z lekkim i szybkim edytorem, który doskonale integruje się z ekosystemem .NET.

Poniżej przedstawiono szczegółowy dotyczące w/w opisu.

1. Zainstaluj Visual Studio Code
2. Zainstaluj .NET SDK, F# jest częścią platformy .NET, więc należy zainstalować .NET SDK, aby móc tworzyć i kompilować projekty F#. W celu instalacji .NET SDK, przejdź na stronę oficjalną .NET a następnie pobierz i zainstaluj najnowszą wersję .NET SDK odpowiednią dla Twojego systemu operacyjnego. Po instalacji możesz sprawdzić, czy wszystko działa poprawnie, otwierając terminal/wiersz poleceń i wpisując: dotnet –version.
3. Zainstaluj Ionide w Visual Studio Code, w zakładce Extensions wpisz Ionide-fsharp i zainstaluj rozszerzenie. Ionide oferuje: podpowiedzi kodu i uzupełnianie składni (IntelliSense), debugowanie i analizę kodu, zarządzanie projektami F#, integrację z systemami budowania (np. FAKE) i testowania.

Tworzenie nowego projektu F#

Nowy projekt F# można bezpośrednio utworzyć z Visual Studio Code za pomocą interfejsu wiersza poleceń .NET. w tym celu należy postępować zgodnie z poniższym:

1. Otwórz terminal w Visual Studio Code (Ctrl+~).
2. Przejdź do folderu, w którym chcesz utworzyć projekt: cd /ścieżka/do/twojego/folderu
3. Utwórz nowy projekt F#: dotnet new console -lang "F#" -o MyFSharpApp. To polecenie utworzy nowy projekt typu konsolowego w F# w folderze o nazwie MyFSharpApp.
4. Przejdź do folderu projektu: cd MyFSharpApp
5. Otwórz projekt w Visual Studio Code: code

Kompilowanie i uruchamianie projektu

1. Aby skompilować i uruchomić projekt, wpisz w terminalu: dotnet run. Visual Studio Code automatycznie rozpozna projekt F#, a Ionide zapewni wsparcie dla podpowiedzi kodu i analizę składni.

Konfiguracja dodatkowych funkcji

- FAKE (F# Make): Jeśli chcesz zautomatyzować procesy budowania projektu, możesz skorzystać z narzędzia FAKE, które jest popularne w ekosystemie F#. Można je zainstalować za pomocą polecenia: dotnet new tool-manifest, dotnet tool install fake-cli
- Paket: Paket to menedżer zależności dla F#, alternatywa dla NuGet. Aby go zainstalować: dotnet tool install paket.

Debugowanie

Ionide obsługuje debugowanie, ale aby skonfigurować je w Visual Studio Code, musisz utworzyć plik konfiguracyjny launch.json:

- Przejdź do zakładki Run and Debug po lewej stronie (lub naciśnij Ctrl+Shift+D).
- Kliknij create a launch.json file, wybierz .NET Core i skonfiguruj debugowanie dla swojego projektu F#.

Praca z projektami F#

- Twój kod F# będzie znajdował się w pliku Program.fs.
- Dodawaj nowe pliki źródłowe do projektu w zakładce Explorer, a Visual Studio Code z Ionide automatycznie rozpozna i zintegruje je w projekcie

Dodatkowe konfiguracje i rozszerzenia

- Możesz zainstalować inne rozszerzenia poprawiające produktywność, takie jak C# for Visual Studio Code (do lepszej integracji z projektami .NET) lub NuGet Package Manager do zarządzania paczkami NuGet.

Testowanie projektów F#

- Aby dodać testy do swojego projektu, możesz skorzystać z bibliotek takich jak Expecto lub xUnit. Aby zainstalować Expecto, dodaj zależność w pliku .fsproj: <PackageReference Include="Expecto" Version="X.X.X" />. Następnie można pisać i uruchamiać testy za pomocą dotnet test.

Konfiguracja środowiska Visual Studio

Aby utworzyć projekt F# w Visual Studio (na przykładzie wersji dla systemu Windows), należy przejść przez kilka kroków:

1. Instalacja Visual Studio - podczas instalacji wybierz zestaw funkcji „.NET Desktop Development” oraz „.NET Core cross-platform development” (lub analogiczne opcje zależnie od wersji Visual Studio). Upewnij się, że opcje związane z językiem F# są zaznaczone, aby Visual Studio obsługiwało F#.
2. Tworzenie nowego projektu F#:
    - Otwórz Visual Studio.
    - Na ekranie startowym wybierz opcję Create a new project.
    - W oknie wyboru typu projektu, w polu wyszukiwania wpisz F#, aby filtrować dostępne szablony projektów.
    - Jeśli chcesz utworzyć aplikację konsolową w F#, wybierz szablon Console App (F#).
    - Jeśli chcesz utworzyć aplikację ASP.NET Core lub inne typy aplikacji, wybierz odpowiedni szablon (np. ASP.NET Core Web Application z F# jako językiem programowania).
    - Kliknij Next po wybraniu szablonu.
3. Konfiguracja projektu:
    - W kolejnym oknie nazwij swój projekt i wybierz lokalizację na dysku, gdzie projekt ma zostać zapisany.
    - W polu Solution name możesz określić nazwę dla całego rozwiązania, które może zawierać więcej niż jeden projekt.
    - Kliknij Create, aby utworzyć projekt.
4. Kompilacja i uruchomienie projektu
    - Aby skompilować i uruchomić projekt, kliknij przycisk Start (lub naciśnij F5).
    - Visual Studio skompiluje aplikację i uruchomi ją w konsoli. W przypadku aplikacji konsolowej powinieneś zobaczyć wynik programu w oknie konsoli.

Wstęp do programowania w F#

F# to funkcjonalny język programowania oparty na platformie .NET, który łączy paradygmaty programowania funkcjonalnego, obiektowego i imperatywnego. Jego największą zaletą jest prostota i moc funkcji, które pomagają tworzyć wydajny, łatwy do utrzymania kod. F# został zaprojektowany z myślą o łatwym pisaniu kodu deklaratywnego, co sprawia, że jest popularny w dziedzinach takich jak obliczenia naukowe, analiza danych czy algorytmy finansowe.

Podstawy F#

F# jest językiem silnie typowanym i funkcyjnym, co oznacza, że:

- Funkcje są podstawowymi elementami programowania.
- F# obsługuje także programowanie obiektowe i imperatywne.
- Typy są określane automatycznie (wspierane jest wnioskowanie typów).

Nadawanie wartości i komunikacja z użytkownikiem

Instrukcja let jest podstawową konstrukcją w F#, służącą do nadawania wartości zmiennym.

```
let x = 5  // Zmienna o nazwie `x` z wartością 5
let y = "Hello"  // Zmienna `y` z łańcuchem znaków
```

Wartości przypisywane za pomocą let są niemodyfikowalne (immutable), co oznacza, że po przypisaniu nie mogą być zmienione. W celu stworzenia zmiennej, którą można modyfikować, należy użyć słowa kluczowego mutable:

```
let mutable counter = 0
counter <- counter + 1  // Zmienna `counter` może być zmieniana
```

Komunikacja z użytkownikiem

F# obsługuje standardowe wejście/wyjście (I/O), umożliwiając komunikację z użytkownikiem.

- Wyświetlanie tekstu: do wyświetlania informacji na konsoli używamy funkcji printfn (znak n oznacza nową linię po wydrukowaniu).

```
let x = 5  // Zmienna o nazwie `x` z wartością 5
let y = "Hello"  // Zmienna `y` z łańcuchem znaków

printfn "Witaj, świecie!"
printfn "Wartość zmiennej x: %d" x
printfn "Pod wartością y mamy: %s" y
```

Odczyt danych od użytkownika:

Aby odczytać dane od użytkownika, można skorzystać z Console.ReadLine(). Jednak dane są zwracane jako ciąg znaków (string), więc musimy je skonwertować, jeśli potrzebujemy innego typu.

```
printfn "Podaj swoje imię: "
let name = System.Console.ReadLine()
printfn "Witaj, %s!" name  // %s - oznacza wstawienie łańcucha znaków
printfn "Podaj liczbę: "
let input = System.Console.ReadLine()
let number = int input  // Konwersja tekstu na liczbę całkowitą
```

```
printfn "Wpisałeś liczbę: %d" number
```

Liczby i łańcuchy znaków

Operacje na liczbach - w F# można wykonywać typowe operacje matematyczne na liczbach.

```
let a = 10
let b = 20
let sum = a + b  // Dodawanie
let diff = a - b  // Odejmowanie
let product = a * b  // Mnożenie
let quotient = b / a  // Dzielenie
```

Operacje na łańcuchach znaków

F# obsługuje typ string do pracy z tekstem (łańcuchami znaków). Możemy łączyć, porównywać i przetwarzać łańcuchy znaków.

```
let str1 = "Hello"
let str2 = "World"
let combined = str1 + " " + str2  // Łączenie łańcuchów znaków
printfn "%s" combined  // Wyświetli "Hello World"
```

F# wspiera interpolację łańcuchów znaków poprzez używanie specyfikatorów formatu, np. %d dla liczb całkowitych i %s dla łańcuchów.

Podstawowe Specyfikatory Formatu w F#

Oto najczęściej używane specyfikatory formatu w F#:

Specyfikator	Opis	Przykład	Wynik
%d	Liczba całkowita (integer)	`printfn "Wartość: %d" 42`	Wartość: 42
%i	Liczba całkowita (integer)	`printfn "Wartość: %i" 42`	Wartość: 42
%u	Liczba całkowita bez znaku (unsigned integer)	`printfn "Wartość: %u" 42u`	Wartość: 42
%f	Liczba zmiennoprzecinkowa (float)	`printfn "Wartość: %f" 3.14`	Wartość: 3.140000
%F	Liczba zmiennoprzecinkowa (float)	`printfn "Wartość: %F" 3.14`	Wartość: 3.14
%e	Notacja naukowa (float)	`printfn "Wartość: %e" 3.14`	Wartość: 3.140000e+00
%E	Notacja naukowa (float)	`printfn "Wartość: %E" 3.14`	Wartość: 3.140000E+00
%g	Notacja ogólna (float)	`printfn "Wartość: %g" 3.14`	Wartość: 3.14
%G	Notacja ogólna (float)	`printfn "Wartość: %G" 3.14`	Wartość: 3.14
%s	Łańcuch znaków (string)	`printfn "Witaj, %s!" "Świecie"`	Witaj, Świecie!
%c	Pojedynczy znak (char)	`printfn "Litera: %c" 'A'`	Litera: A
%b	Wartość logiczna (bool)	`printfn "Prawda: %b" true`	Prawda: true
%A	Dowolny typ (używa ToString lub niestandardowego formatowania)	`printfn "Obiekt: %A" [1; 2; 3]`	Obiekt: [1; 2; 3]
%O	Obiekt (wywołuje ToString)	printfn "Obiekt: %O" (System.DateTime.Now)	Obiekt: 27.04.2024 14:35:22

%x	Liczba szesnastkowa (hexadecimal)	printfn "Wartość: %x" 255	Wartość: ff
%X	Liczba szesnastkowa (hexadecimal, wielkie litery)	printfn "Wartość: %X" 255	Wartość: FF

Instrukcja warunkowa if..then..else

Instrukcja warunkowa w F# umożliwia podejmowanie decyzji w zależności od warunku logicznego.

Przykład instrukcji warunkowej:

```
let x = 10
let y = 5

if x > y then
    printfn "x jest większe niż y"
else
    printfn "x jest mniejsze lub równe y"
```

W F# blok instrukcji if zwraca wartość. Możesz więc używać instrukcji warunkowej bezpośrednio w przypisaniu:

```
let isEven =
    if x % 2 = 0 then
        "Liczba parzysta"
    else
        "Liczba nieparzysta"

printfn "%s" isEven
```

Operatory warunkowe i arytmetyczne

F# obsługuje pełen zestaw operatorów arytmetycznych i logicznych.

Operatory arytmetyczne:

- + – Dodawanie
- - – Odejmowanie
- – Mnożenie
- / – Dzielenie
- % – Reszta z dzielenia

Operatory porównania:

- = – Równość
- <>– Nierówność
- < – Mniejsze niż
- > – Większe niż
- <= – Mniejsze lub równe
- >= – Większe lub równe

Operatory logiczne:

- && – AND (koniunkcja)

- || – OR (alternatywa)
- not – Negacja

Korzystanie z match do bardziej rozbudowanego warunkowania

Zamiast wielokrotnych instrukcji if, możemy używać dopasowywania wzorców (match), co pozwala na bardziej elegancką obsługę różnych przypadków.

Niezmienne Struktury Danych w F#

W programowaniu funkcjonalnym, a w szczególności w F#, niezmienne struktury danych (ang. immutable data structures) odgrywają kluczową rolę. Niezmienność oznacza, że po utworzeniu struktury danych nie można jej zmienić. Zamiast tego, wszelkie operacje modyfikujące tworzą nowe struktury danych, pozostawiając oryginał bez zmian. Niezmienne struktury danych to takie, które nie mogą być zmienione po ich utworzeniu. Każda operacja, która wydaje się modyfikować taką strukturę, w rzeczywistości tworzy nową wersję z odpowiednimi zmianami, pozostawiając oryginalną strukturę nietkniętą.

Dlaczego Niezmienność jest Ważna?

- Bezpieczeństwo w Równoległym Programowaniu: Niezmienność eliminuje problemy związane z równoczesnym dostępem do danych, ponieważ dane nie mogą być modyfikowane po ich utworzeniu.
- Łatwiejsze Rozumienie i Debugowanie: Kiedy struktury danych są niezmienne, można łatwiej przewidzieć ich stan w dowolnym momencie programu, co ułatwia śledzenie błędów.
- Optymalizacja Wydajności: Niezmienne struktury danych często wykorzystują techniki takie jak dzielenie struktury (structural sharing), co pozwala na efektywne zarządzanie pamięcią i szybkie operacje tworzenia nowych wersji danych.
- Referencyjna Transparentność: Funkcje operujące na niezmiennych strukturach danych są bardziej przewidywalne i łatwiejsze do testowania, ponieważ zawsze zwracają ten sam wynik dla tych samych wejść.

F# oferuje szereg wbudowanych niezmiennych struktur danych, które są fundamentem programowania w tym języku:

a) Listy w F# są jednymi z najbardziej podstawowych niezmiennych struktur danych. Są one implementowane jako łańcuchy elementów (linked lists), co umożliwia szybkie dodawanie elementów na początku listy.

```
let lista1 = [1; 2; 3]
let lista2 = 0 :: lista1  // [0; 1; 2; 3]
```

Operacje na Listach:

```
//Dodawanie Elementu: Operator :: dodaje element na początku listy.
let nowaLista = 4 :: lista1  // [4; 1; 2; 3]
//Łączenie List: Operator @ łączy dwie listy.
let lista3 = lista1 @ [4; 5]  // [1; 2; 3; 4; 5]
//Mapowanie: List.map stosuje funkcję do każdego elementu listy.
let podwojonaLista = List.map (fun x -> x * 2) lista1  // [2; 4; 6]
```

b) Rekordy (Records) - Rekordy to niezmienne typy danych, które grupują powiązane wartości pod jedną nazwą.

```
type Osoba = {
    Imie: string
    Wiek: int
}

let osoba1 = { Imie = "Anna"; Wiek = 30 }
let osoba2 = { osoba1 with Wiek = 31 }  // Tworzy nowy rekord z
zaktualizowanym wiekiem
```

Cechy Rekordów:

- Syntaktyka with: Umożliwia tworzenie nowych rekordów na podstawie istniejących, z modyfikacją wybranych pól.
- Brak Zmian: osoba1 pozostaje niezmieniona, nawet po utworzeniu osoba2.

c) **Krotki (Tuples)** to proste, niezmienne struktury danych, które mogą przechowywać różne typy wartości.

```
let krotka = (1, "F#", true)
let (liczba, tekst, flaga) = krotka  // Destrukturyzacja krotki
```

Cechy Krotek:

- Różnorodność Typów: Każdy element krotki może być innego typu.
- Szybkość: Krotki są lekkie i efektywne do przechowywania niewielkich zbiorów danych.

d) **Zbiory Unii (Discriminated Unions)** pozwalają na definiowanie typów, które mogą mieć różne formy (konstrukcje).

```
type Ksztalt =
    | Kolo of promien: float
    | Prostokat of szerokosc: float * wysokosc: float

let obliczPole ksztalt =
    match ksztalt with
    | Kolo r -> System.Math.PI * r * r
    | Prostokat (s, w) -> s * w

let poleKola = obliczPole (Kolo 5.0)  // ~78.54
let poleProstokata = obliczPole (Prostokat (4.0, 5.0))  // 20.0
```

Cechy Zbiorów Unii:

- Elastyczność: Typ może przyjmować różne formy, co umożliwia modelowanie złożonych struktur danych.
- Bezpieczeństwo Typów: Dopasowywanie wzorców (match) zapewnia, że wszystkie możliwe przypadki są obsłużone.

Chociaż podstawowe kolekcje w F# (takie jak listy) są niezmienne, istnieją również niezmienne implementacje słowników i innych struktur danych:

- Immutable Dictionary: Dostępne w przestrzeni nazw System.Collections.Immutable.

```
open System.Collections.Immutable

let slownik1 = ImmutableDictionary<int, string>.Empty
let slownik2 = slownik1.Add(1, "jeden")
let slownik3 = slownik2.Add(2, "dwa")
```

- Inne Kolekcje: F# wspiera także inne niezmienne kolekcje, takie jak ImmutableList, ImmutableArray itp.

Przykłady Użycia Niezmiennych Struktur Danych

- Praca z Listami

```
let lista = [1; 2; 3; 4; 5]

// Dodawanie elementu
let nowaLista = 0 :: lista  // [0; 1; 2; 3; 4; 5]

// Modyfikacja elementów
let podwojonaLista = List.map (fun x -> x * 2) lista  // [2; 4; 6; 8; 10]

// Filtrowanie
let parzyste = List.filter (fun x -> x % 2 = 0) lista  // [2; 4]
```

- Praca z Rekordami

```
type Student = {
    Imie: string
    Nazwisko: string
    Ocena: float
}

let student1 = { Imie = "Jan"; Nazwisko = "Kowalski"; Ocena = 4.5 }

// Aktualizacja oceny
let student2 = { student1 with Ocena = 5.0 }
```

- Praca ze Zbiorami Unii

```
type Komunikat =
    | Powitanie of string
    | Rozlaczenie
    | KomunikatBlad of string

let wyswietlKomunikat komunikat =
    match komunikat with
    | Powitanie imie -> printfn "Witaj, %s!" imie
    | Rozlaczenie -> printfn "Do widzenia!"
    | KomunikatBlad blad -> printfn "Błąd: %s" blad

wyswietlKomunikat (Powitanie "Anna")  // Witaj, Anna!
wyswietlKomunikat Rozlaczenie  // Do widzenia!
wyswietlKomunikat (KomunikatBlad "Nieznany błąd")  // Błąd: Nieznany błąd
```

Korzyści z Używania Niezmiennych Struktur Danych

- Łatwiejsze Równoległe Przetwarzanie: Niezmienność eliminuje potrzebę stosowania blokad czy mechanizmów synchronizacji, ponieważ dane nie mogą być modyfikowane po ich utworzeniu.
- Bezpieczeństwo Typów: Kompilator F# może efektywnie optymalizować kod, wiedząc, że dane nie ulegną zmianie.
- Referencyjna Transparentność: Funkcje operujące na niezmiennych strukturach danych są bardziej przewidywalne i łatwiejsze do testowania, ponieważ ich wyniki zależą tylko od ich wejść.

Chociaż F# promuje niezmienność, język ten pozwala również na używanie mutowalnych struktur danych, gdy jest to konieczne. Mutowalne dane są oznaczane słowem kluczowym mutable i mogą być zmieniane po ich utworzeniu.

Przykład Mutowalnej Zmiennej:

```
let mutable licznik = 0
licznik <- licznik + 1  // licznik teraz wynosi 1
```

Kiedy Używać Mutowalności?

- Wydajność: W niektórych przypadkach mutowalność może przynieść korzyści wydajnościowe, zwłaszcza gdy często modyfikujesz duże struktury danych.
- Interakcje z API: Kiedy pracujesz z bibliotekami lub interfejsami, które wymagają mutowalnych danych.

Uwagi:

- Ostrożność: Używanie mutowalnych struktur danych wymaga większej ostrożności, aby unikać błędów związanych z niekontrolowanymi zmianami stanu.
- Preferowanie Niezmienności: W miarę możliwości, zaleca się preferowanie niezmiennych struktur danych dla lepszej czytelności, bezpieczeństwa i łatwiejszego debugowania kodu.

Przykład Kompleksowego Użycia Niezmiennych Struktur Danych

Przykładowa aplikacja, która zarządza listą zadań (To-Do List) używając niezmiennych struktur danych.

```
type Zadanie = {
    Id: int
    Opis: string
    CzyWykonane: bool
}

type Stan = {
    Zadania: Zadanie list
    NextId: int
}

let inicjalnyStan = { Zadania = []; NextId = 1 }

let dodajZadanie opis stan =
    let noweZadanie = { Id = stan.NextId; Opis = opis; CzyWykonane = false
}
    { stan with Zadania = noweZadanie :: stan.Zadania; NextId = stan.NextId
+ 1 }

let oznaczWykonane id stan =
    let zaktualizowaneZadania =
```

```
        stan.Zadania
        |> List.map (fun zadanie -> if zadanie.Id = id then { zadanie with
CzyWykonane = true } else zadanie)
    { stan with Zadania = zaktualizowaneZadania }

let usunZadanie id stan =
    let pozostaleZadania = List.filter (fun zadanie -> zadanie.Id <> id)
stan.Zadania
    { stan with Zadania = pozostaleZadania }

// Przykładowe użycie:
let stan1 = dodajZadanie "Kup mleko" inicjalnyStan
let stan2 = dodajZadanie "Zrobić pranie" stan1
let stan3 = oznaczWykonane 1 stan2
let stan4 = usunZadanie 2 stan3

// Wyświetlanie stanu
stan4.Zadania |> List.iter (fun z -> printfn "Zadanie %d: %s - Wykonane:
%b" z.Id z.Opis z.CzyWykonane)
// Wynik:
// Zadanie 1: Kup mleko - Wykonane: true
```

Wyjaśnienie:

- Typ Zadanie: Reprezentuje pojedyncze zadanie z unikalnym identyfikatorem, opisem i statusem wykonania.
- Typ Stan: Przechowuje listę zadań oraz NextId do nadawania unikalnych identyfikatorów nowym zadaniom.
- Funkcje dodajZadanie, oznaczWykonane, usunZadanie: Każda z tych funkcji przyjmuje aktualny stan i zwraca nowy stan z odpowiednimi modyfikacjami, bez zmieniania oryginalnego stanu.
- Korzyści: Dzięki niezmienności, każdy stan jest niezależny i nie wpływa na inne, co ułatwia śledzenie zmian i testowanie aplikacji.

Zadania do samodzielnego rozwiązania

Rozwiązania w postaci niezbędnych plików źródłowych należy przesłać do utworzonego zadania na platformie e-learningowej zgodnie ze zdefiniowanymi instrukcjami oraz w nieprzekraczalnym wyznaczonym terminie.

Zadanie 1. Stwórz aplikację konsolową, która oblicza wskaźnik masy ciała (BMI) użytkownika na podstawie wprowadzonych przez niego wagi (kg) i wzrostu (cm). Program powinien komunikować się z użytkownikiem, odczytywać dane wejściowe, przeliczać BMI i wyświetlać wynik wraz z kategorią BMI.

Wymagane funkcje:

- Komunikacja z użytkownikiem: Odczyt danych za pomocą Console.ReadLine() i wyświetlanie wyników za pomocą printfn.
- Przekształcanie typów: Konwersja danych wejściowych z string na float.
- Instrukcje warunkowe: Określenie kategorii BMI na podstawie wartości obliczonej.
- Niezmienne struktury danych: Możesz użyć rekordów do przechowywania danych użytkownika.

Zadanie 2. Napisz program, który pozwala użytkownikowi na konwersję kwoty z jednej waluty na inną. Program powinien pobierać od użytkownika:

- Kwotę do przeliczenia.
- Walutę źródłową (np. USD, EUR, GBP).
- Walutę docelową.

Następnie program powinien obliczyć i wyświetlić przeliczoną kwotę na podstawie zdefiniowanych kursów wymiany.

Wymagane funkcje:

- Instrukcje let: Przechowywanie kursów wymiany jako stałe.
- Instrukcje warunkowe: Obsługa różnych kombinacji walut.
- Operatory arytmetyczne: Obliczanie przeliczonej kwoty.
- Niezmienne struktury danych: Użycie mapy (Map) do przechowywania kursów wymiany.

Zadanie 3. Stwórz program, który analizuje wprowadzony przez użytkownika tekst. Program powinien:

- Liczyć liczbę słów.
- Liczyć liczbę znaków (bez spacji).
- Znajdować najczęściej występujące słowo.

Wymagane funkcje:

- Nadawanie wartości: Przechowywanie danych tekstowych.
- Łańcuchy znaków: Operacje na stringach, takie jak dzielenie na słowa.
- Instrukcje warunkowe: Obsługa różnych scenariuszy analizy.
- Niezmienne struktury danych: Użycie list i map do przechowywania i analizowania słów.

Zadanie 4. Stwórz aplikację, która symuluje podstawowe operacje bankowe. Użytkownik powinien móc:

- Tworzyć nowe konto.
- Depozytować środki na konto.

- Wypłacać środki z konta.
- Wyświetlać saldo konta.

Wymagane funkcje:

- Niezmienne struktury danych: Użycie mapy (Map) do przechowywania kont z unikalnymi identyfikatorami.
- Rekordy: Definiowanie struktury konta z numerem konta i saldem.
- Instrukcje warunkowe i operatorów: Obsługa różnych akcji i walidacja operacji (np. brak środków).
- Komunikacja z użytkownikiem: Interakcja poprzez menu tekstowe.

Zadanie 5. Zaprojektuj i zaimplementuj grę „Kółko i Krzyżyk” w konsoli, gdzie użytkownik gra przeciwko komputerowi. Gra powinna:

- Wyświetlać planszę po każdej turze.
- Umożliwiać graczowi wybór pozycji na planszy.
- Komputer powinien wykonywać ruchy losowo lub według prostego algorytmu.
- Określać zwycięzcę lub remis.

Wymagane funkcje:

- Niezmienne struktury danych: Reprezentacja planszy jako list lub tablica.
- Instrukcje warunkowe i operatorów: Sprawdzanie stanu gry, wygrywających kombinacji.
- Operacje na łańcuchach znaków: Wyświetlanie planszy.
- Instrukcje let i rekursja: Zarządzanie turami gry.