Języki programowania na platformie .NET

2025/26

# Instrukcja laboratoryjna cz.6 i 7

## Język funkcyjny F#


Prowadzący: Tomasz Goluch

Wersja: 1.0

# I. Słowa kluczowe: „let”, „fun” i operator „|>” w F#

*Cel: Zapoznanie z najważniejszymi słowami kluczowymi języka F#.*

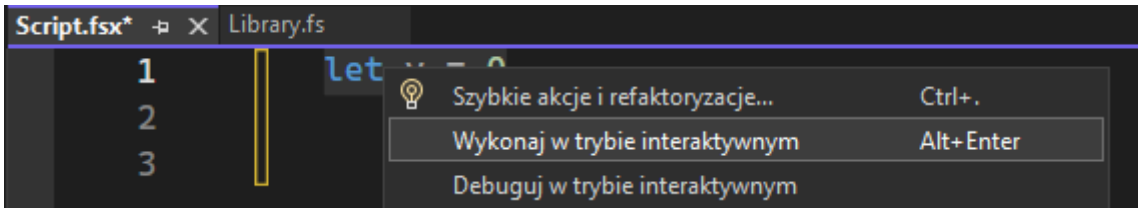
Uruchom Visual Studio i utwórz nowy *Library(.Net lub .NET Standard)* projekt – **MyProject** (Visual F#).

Dodaj nowy plik sktyptu ***Script.fsx*** i wpisz polecenie:

```
let y = 0
```

Wiąże ono identyfikator `y` z wartością `0`. Słowo kluczowe `let` jest jednym z najważniejszych w F#, pozwala również na wiązanie identyfikatora z nazwą funkcji[1].

W celu aktywacji polecenia należy zaznaczyć interesujące nas linie skryptu i z menu kontekstowego wybrać: Execute In Interactive/Wykonaj w trybie interaktywnym albo (Alt + Enter).




Wynik: `> val y: int = 0` powinien pojawić się w oknie F# Interactive[2]. Informuje on o typie wiązanej wartości `y` – value, integer.

Wiązanie z listą float’ów:

```
let data = [1.;2.;3.;4.]
```

Wiązanie funkcji `sqr`:

```
let sqr x = x * x
```

Domyślnie funkcja `sqr` będzie obsługiwać typy `int`. Dzieje się tak ponieważ operator `*` jest przeciążony dla wielu typów: `int`, `float`, `decimal`, `int64`, itd. a ponieważ nie ma żadnych dodatkowych informacji wybrany zostanie typ domyślny, którym jest `int`.

```
val sqr: x: int -> int
```

Wołanie zdefiniowanej wcześniej funkcji należy (w oknie F# Interactive) zakończyć parą średników:

[1] Więcej informacji na: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/let-bindings

[2] Okno F# Interactive można uruchomić wybierając: VIEW → Other Windows → F# Interactive.

```
> sqr 3;;
val it: int = 9
```

Rozważmy poniższe przykłady różnych implementacji tej samej funkcji, liczącej sumę kwadratów (styl imperatywny, iteracyjny):

```
let sumOfSquaresI nums =
    let mutable acc = 0 // change from 0 to 0. see behaviour of input data type
    for x in nums do
        acc <- acc + sqr x
    acc
```

- `mutable` – tworzy zmienną której wartość może ulec zmianie (tak jak w językach imperatywnych). W czystym programowaniu funkcjonalnym zmienne w ogóle nie powinny być używane[3].
- `<-` – operator ten pozwala na przypisanie nowej wartości do zmiennej.

Wołanie funkcji `sumOfSquaresI` obsługującej typy całkowite:

```
> sumOfSquaresI [1;2;3;4];;
val it: int = 30
```

Wprowadzając zmiany odpisane w komentarzu zmieniamy działanie funkcji na typy zmiennoprzecinkowe. Niestety ponowne wykonanie tego kodu w trybie interaktywnym spowoduje wygenerowanie błędu niezgodności typów:

```
error FS0001: Typ „int” nie jest zgodny z typem „float”
```

Dzieje się tak ponieważ F# nie wykonuje automatycznej konwersji typów (np. z `int` na `float`) podczas operacji arytmetycznych i jeśli będziemy musieli dostosować funkcję do obsługi typu `float`. Możemy sobie poradzić z tym problemem na kilka sposobów:

- możemy zdefiniować `sqr` od nowa funkcję razem z `sumOfSquaresI` wtedy F# mając te nowe informacje wywnioskuje, że `sqr` ma obsługiwać typ `float` :
  ```
  val sqr: x: float -> float
  val sumOfSquaresI: nums: float seq -> float
  ```
- jawne określenie typu na którym ma działać funkcja:
  ```
  let sqr (x: float) = x * x
  ```
  Wadą jest (jak i poprzedniego) ograniczenie działania tej funkcji do określonego jawnie typu.
- jawne rzutowanie w funkcji `sumOfSquaresI` :
  ```
  acc <- acc + float (sqr (int x))
  ```
  Najgorsze rozwiązanie, wartości wejściowe nie będą całkowite to nastąpią błędy spowodowane zignorowaniem części ułamkowej.
- metoda inline (najbardziej idiomatyczne i elastyczne rozwiązanie):

[3] Więcej informacji na: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/values/#why-immutablemkjyrew

```
let inline sqr x = x * x
```

Funkcja w tym przypadku nie jest kompilowana od razu tylko jest traktowana jak „szablon”. W momencie gdy wywnioskowany zostanie odpowiedni typ funkcji to powstanie dla niego skompilowana „wyspecjalizowana” wersja. Nie należy używać tego rozwiązania dla dużych funkcji ponieważ prowadzi to do problemów związanych z wydajnością kompilacji i rozmiarem wynikowego programu.

Uruchomienie po wprowadzeniu zmian:

```
> sumOfSquaresI data;;
val it: float = 30.0
```

Składnia języka F# jest wrażliwa na dzielenie linii oraz wcięcia. Instrukcje wykonywane wewnątrz funkcji bądź pętli – tak jak ma to miejsce w powyższym przykładzie – muszą być wcięte głębiej (rozpoczynać się bardziej od prawej strony). W momencie kiedy wymagane jest wcięcie musi ono zawierać przynajmniej jedną spację – nie można używać tabulacji[4]. Jednak jest to dopuszczalne przy domyślnych ustawieniach edytora VS, ponieważ tabulacja zamieniana jest na 4 znaki spacji (TOOLS → Options → Text Editor → F# → Tabs).

Ponownie zaimplementujmy funkcję licząca sumę kwadratów ale tym razem funkcyjnie:

```
let rec sumOfSquaresF nums =
    match nums with
    | [] -> 0 // change from 0 to 0. see behaviour of input data type
    | h::t -> sqr h + sumOfSquaresF t
```

- `rec` – funkcja może działać rekurencyjnie[5].
- `match`, `with`, `|`, `->` – wyrażenie znajdujące się po `match` jest porównywane ze wzorcami w kolejnych gałęziach rozpoczynających się od `|`. Pierwsze dopasowanie powoduje przerwanie porównywania i wykonanie wyrażenia zamieszczonego po `->`[6]
- `::` – podział listy na pierwszy element `h` (głowę) i resztę elementów `t` (ogon). Pustą listę oznaczamy za pomocą dwóch nawiasów kwadratowych `[]`.

Jeśli możemy zaobserwować uniwersalność funkcji inline. Jeśli będziemy chcieli skompilować 2 funkcje `sumOfSquaresI` i `sumOfSquaresF` wykorzystując `sqr` ale jedna z nich będzie operować na `float` a druga na `int` to jedynym rozwiązaniem będzie wykorzystanie `sqr` w wersji inline. Udana kompilacja:

```
val inline sqr: x: ^a -> 'b when ^a: (static member ( * ) : ^a * ^a -> 'b)
val sumOfSquaresI: nums: float seq -> float
val sumOfSquaresF: nums: int list -> int
```

[4] Więcej informacji na: https://docs.microsoft.com/en-us/dotnet/articles/fsharp/language-reference/code-formatting-guidelines

[5] Więcej informacji na: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/recursive-functions-the-rec-keyword

[6] Więcej informacji na: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/match-expressions

Jeszcze raz zaimplementujmy funkcję licząca sumę kwadratów, ale tym razem funkcjonalnie i bez rekurencji:

```
let sumOfSquares nums =
    Seq.sum(Seq.map(sqr) nums)
```

albo:

```
let sumOfSquares nums =
    Seq.sum(Seq.map(fun x -> x * x) nums)
```

- `Seq.map` – funkcja zwraca nową kolekcję składającą się z wyników zastosowania funkcji mapującej podanej jako pierwszy parametr (w powyższym przykładzie `sqr`) na elementach kolekcji wejściowej przekazanej jako drugi parametr (w powyższym przykładzie lista `nums`) [7].
- `Seq.sum` – sumuje wszystkie elementy kolekcji przekazanej jako parametr[8].

W celu poprawienia czytelności zagnieżdżonych funkcji możemy uszeregować je w kolejności wykonania:

```
let sumOfSquares nums =
    nums
    |> Seq.map(fun x -> x * x)
    |> Seq.sum
```

- `|>` – operator przekierowania, pozwala na użycie, wartości otrzymanych po lewej stronie, w wyrażeniu po prawej stronie tego operatora. Pozwala to uniknąć problemu zwanego piramidą zagłady (Pyramid of Doom) kiedy kolejne wywołania funkcji są głęboko zagnieżdżone jedne w drugich. W takim przypadku kod staje się nieczytelny, wizualnie tworzy "piramidę" wcięć i musi być analizowany "od środka na zewnątrz", zamiast w naturalnej, liniowej kolejności od góry do dołu.

Informacje na temat wszystkich słów kluczowych można znaleźć na: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/keyword-reference, a opisujące pozostałe funkcje modułu `Collections.Seq` tutaj: https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-seqmodule.html.

W F# wykorzystując pakiet ParallelSeq łatwo możemy przetwarzać równolegle (wielowątkowo) sekwencje (kolekcje). Pozwala to na przyspieszenie operacji na dużych zbiorach danych, które wymagają intensywnych obliczeń (tzw. CPU-bound), poprzez automatyczne rozdzielenie pracy na wiele rdzeni procesora.

[7] Więcej informacji na: http://msdn.microsoft.com/pl-pl/library/ee370346.aspx
[8] Więcej informacji na: http://msdn.microsoft.com/pl-pl/library/ee370214.aspx

Zamiast przetwarzać kolekcję element po elemencie, jeden po drugim (jak robi to Seq), pakiet ten (a konkretnie moduł PSeq, który udostępnia) wykonuje operacje na wielu elementach jednocześnie.

W pierwszym kroku należy doinstalować bibliotekę **FSharp.Collections.ParallelSeq**, najlepiej za pomocą narzędzie NuGet. W oknie **Package Manager Console**[9] uruchamiamy polecenie:

```
PM> Install-Package FSharp.Collections.ParallelSeq -ProjectName MyProject
```

To samo co dotychczas ale równolegle:

```
#r "nuget: FSharp.Collections.ParallelSeq"
open FSharp.Collections.ParallelSeq
let sumOfSquaresP nums =
    nums
    |> PSeq.map(fun x -> x * x)
    |> PSeq.sum
```

Może okazać się wymagane jawne określenie typu parametru `nums` jeśli będzie inny niż domyślny dla `PSeq.sum` czyli `int`.

```
let sumOfSquaresP (nums: seq<float>) =
```

Przewagą F# nad innymi językami funkcyjnymi jest fakt, że możemy wykorzystać całe bogactwo, które oferuje nam środowisko .NET. Oto przykład pobrania historycznych informacji finansowych o firmie Microsoft (ticker: `msft.us`) z witryny `stooq.com`:

```
open System.Net.Http
let ticker = "msft.us"
let dataStart = System.DateTime(2000, 1, 1).ToString("yyyyMMdd")
let dataEnd = System.DateTime(2023, 2, 10).ToString("yyyyMMdd")
let url =
    sprintf "https://stooq.com/q/d/l/?s=%s&d1=%s&d2=%s&i=d"
            ticker dataStart dataEnd

let client = new HttpClient()
let getDataAsync = async {
    let! csvResponse = Async.AwaitTask (client.GetStringAsync(url))
    return csvResponse
}
let csv = Async.RunSynchronously getDataAsync
```

Pobrany string ma następującą zawartość (komenda: `csv;;`):

[9] Jeśli okno nie jest dostępne (domyślnie znajduje się ono w tym samym miejscu co F# Interactive, poniżej okna edytora – jedna z dostępnych zakładek) to możemy je znaleźć w: TOOLS → NuGet Package Manager → Package Manager Console.

```
> res;;
val it: string =
  "Date,Open,High,Low,Close,Volume
2000-01-03,40.471,40.8992,38.6126,40.1865,77193325.494835
2000-01-04,39.1545,40.378,38.702,38.8342,78484901.620581
2000-01-05,38.3086,40.1274,37.7086,39.2409,92901040.975204
2000-01-06,38.6796,39.2642,37.3658,37.927,79728621.758994
2000-01-07,37.4495,38.702,36.9968,38.4194,89933866.773534
2000-01-10,39.1098,39.1932,38.3971,38.702,65207479.974728
2000-01-11,38.4398,39.3887,37.4707,37.7086,67788892.776219
2000-01-12,37.4066,37.5358,36.0055,36.4823,96487167.923791
2000-01-13,35.9812,37.4495,34.9907,37.1727,120577775.96378
```

Zawiera dane rozdzielone przecinkiem. W pierwszym kroku musimy usunąć nagłówek, następnie dzielimy dane. Ustalamy, że długość sekwencji równa jest 6 i wybieramy pierwszą i trzecią pozycję: datę i cenę zamknięcia.

```
let prices =
    csv.Split([|'\r'; '\n'|], System.StringSplitOptions.RemoveEmptyEntries)
    |> Seq.skip 1
    |> Seq.map (fun line -> line.Split(','))
    |> Seq.filter (fun values -> values.Length = 6)
    |> Seq.map (fun values -> (values.[0], values.[4]))
    |> Seq.toArray
```

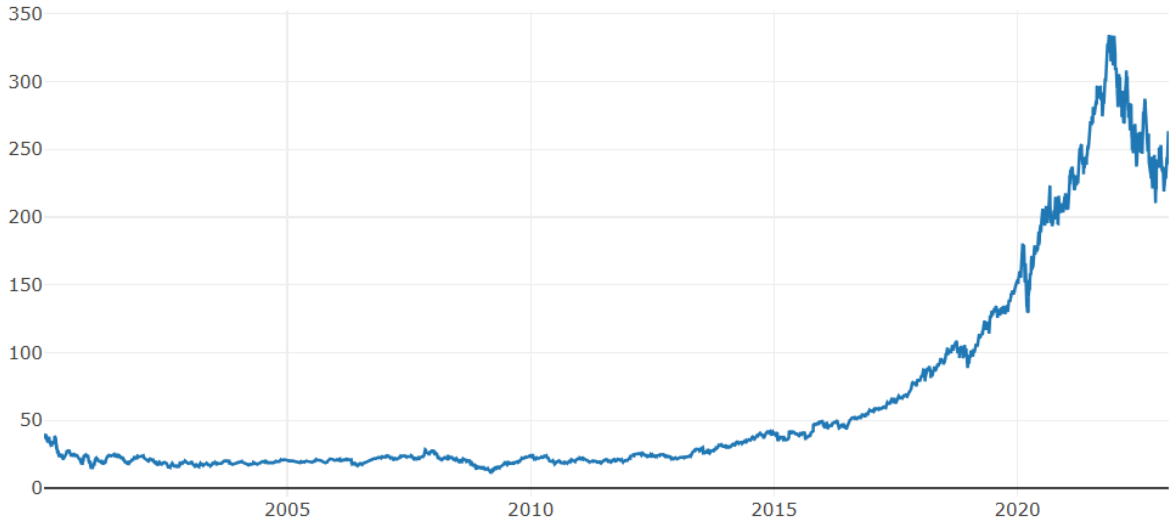
Zawartość tabeli zwracanej przez `prices` (komenda: `prices;;`):

```
> prices;;
val it: (string * string) array =
  [|("2000-01-03", "40.1865"); ("2000-01-04", "38.8342");
    ("2000-01-05", "39.2409"); ("2000-01-06", "37.927");
    ("2000-01-07", "38.4194"); ("2000-01-10", "38.702");
    ("2000-01-11", "37.7086"); ("2000-01-12", "36.4823");
```

Dane zwracane przez funkcję prices możemy zaprezentować w bardziej przyjaznej formie np. wyświetlić w formie wykresu. W tym celu musimy doinstalować pakiet `XPlot.Plotly`, polecenie: `Install-Package XPlot.Plotly`. Oraz uruchomić poniższy skrypt:

```
#r "nuget: XPlot.Plotly"
open XPlot.Plotly
Chart.Line(prices).Show()
```

Powinniśmy uzyskać następujący wykres:




Jeśli chcemy z dotychczasowego kodu utworzyć funkcję (inaczej dokonać refaktoryzacji) wystarczy podać jej nazwę (`loadPrices`) i listę parametrów (w naszym przypadku 1 parametr – `ticker`[10]), ciało funkcji należy wyekstrahować tabulatorami i podać zwracany typ (`prices`):

```
let loadPrices (ticker: string) =
    let stooqTicker = ticker.ToLower() + ".us"
    ...
    let prices =
    ...
    prices
```

Proszę zauważyć, że ticker jest przyjmowany jako parametr zatem nie należy go redefiniować w kodzie. Wywołanie funkcji jest następujące:

- `Chart.Line(loadPrices "MSFT").Show();;` – historia cen akcji dla firmy Microsoft
- `Chart.Line(loadPrices "ORCL").Show();;` – historia cen akcji dla firmy Oracle
- `Chart.Line(loadPrices "EBAY").Show();;` – historia cen akcji dla firmy Ebay

Wywołanie w pętli:

```
["MSFT"; "ORCL"; "EBAY"] |> Seq.iter (fun ticker ->
    let data = loadPrices ticker
    Chart.Line(data).Show()
)
```

[10] Ticker inaczej Stock Symbol to skrót używany do jednoznacznej identyfikacji notowanych publicznie akcji określonej korporacji.

## II. Asynchronizm i przetwarzanie równoległe

*Cel: Opanowanie wyrażeń obliczeniowych async do tworzenia nieblokujących operacji oraz wykorzystanie funkcji Async.Parallel do jednoczesnego wykonywania zadań w celu poprawy wydajności.*

F# udostępnia potężny mechanizm jakim są wyrażenie obliczeniowe (ang. Computation Expression) to specjalna składnia, która pozwala „ukryć" skomplikowaną logikę i „opakować" ją w ładny, liniowy kod. Wykorzystamy jedną z jego realizacji, która pozwala na pisanie eleganckiego kodu asynchronicznego unikając (jak to miało miejsce w starych wersjach JavaScript) problemów piekła „callbacków" lub Pyramid of Doom. Stwórzmy asychroniczną metodę `loadPricesAsync`, zgodnie z konwencją, dodając do niej suffix `Async`. Ciało funkcji definujemy w bloku `async { ... }`. W tym momencie kompilator zaczyna akceptować inny zestaw reguł dla kodu wewnątrz tych nawiasów:

- `let!` oznacza: "Uruchom tę asynchroniczną operację i poczekaj na jej wynik, nie blokując wątku. Gdy wynik będzie gotowy, przypisz go do zmiennej i przejdź dalej".
- `return` oznacza: "Zakończ ten blok asynchroniczny i zwróć tę wartość do wnętrza opakowania `Async`".
- `use!` działa jak `let!`, ale dodatkowo automatycznie zwolni zasób (jak `IDisposable`) po wyjściu z bloku.

```
let loadPricesAsync (ticker: string) = async {
...
    let! csv = Async.AwaitTask (client.GetStringAsync(url))
...
    return prices
}
```

Dysponując asynchroniczną wersją funkcji, można uruchomić wszystkie pobierania w tym samym czasie (równolegle) i poczekać na zakończenie wszystkich. Najpierw tworzymy listę „obietnic" (operacji do wykonania). Ten krok jest natychmiastowy - nic jeszcze nie jest pobierane. Następnie tworzymy jedną, dużą operację, która uruchomi wszystkie równolegle. Służy do tego funkcja `Async.Parallel`.

```
let requests =
    [
        loadPricesAsync "MSFT"
        loadPricesAsync "ORCL"
...
    ]
let parallelRequests = Async.Parallel requests
```

Uruchamiamy tę jedną dużą operację i czekamy na wszystkie wyniki. Funkcja `Async.RunSynchronously` to jest jedyny moment blokowania. Wartość `results` to tablica z gotowymi wynikami które można wyświetlić.

```
let results = Async.RunSynchronously parallelRequests
results |> Array.iter (fun data ->
    Chart.Line(data).Show()
)
```

Kolejnym krokiem jest zrównoleglenie wykonywanych operacji. Wykorzystamy znaną nam bibliotekę **FSharp.Collections.ParallelSeq**. Wprowadzenie zmian jest bardzo proste i schematyczne:

```
|> PSeq.ofArray
|> PSeq.skip 1
|> PSeq.map (fun line -> line.Split(','))
|> PSeq.filter (fun values -> values.Length = 6)
|> PSeq.map (fun values -> (values.[0], values.[4]))
|> PSeq.toArray
```

W pierwszej linijce dokonujemy konwersji na sekwencję równoległą. Następnie wprowadzamy użycie operatorów PSeq zamiast Seq. Na końcu zbieramy z powrotem wyniki do tablicy.

Należy pamiętać, że nie zawsze zrównoleglanie obliczeń będzie owocować zmniejszeniem czasu przetwarzania zadania. Powyższe podejście jest anty-wzorcem wydajnościowym. Dzielenie stringów jest tak szybką operacją (mierzoną w nanosekundach), że narzut związany z zarządzaniem wątkami i synchronizacją PSeq sprawi, że ten kod będzie znacznie wolniejszy niż oryginalna wersja Seq.

## III. F# w ujęciu obiektowym

*Cel: Zapoznanie z podstawami OOP w F#.*

W F# możemy jak w normalnym języku zorientowanym obiektowo definiować klasy. Przykład klasy analizatora giełdowego wyliczającego odchylenie standardowe i stopę zwrotu w zadanym okresie. Jako parametry do funkcji `GetAnalysers` przekazujemy `ticker` spółki giełdowej i interesujący nas okres. Kod należy umieścić w pliku `.fs` wraz z funkcją `prices` i kodem pobierający historyczne dane. Plik z rozszerzeniem `.fs`, reprezentuje kod źródłowy kompilowany do assembly, w przeciwieństwie do dotychczas wykorzystywanego pliku `.fsx` który zawiera interpretowane skrypty. Nic nie stoi na przeszkodzie aby do projektu biblioteki dołączyć wcześniej zaimplementowany przez nas plik udostępniający funkcję `loadPrices`. W tym celu należy dodać do w sekcji `ItemGroup` pliku projektu :

```
<ItemGroup>
  <Compile Include="Script.fsx" />
  <Compile Include="Library.fs" />
</ItemGroup>
```

Należy pamiętać, że nie jest to dobra praktyka ponieważ przeznaczenie tych plików jest diametralnie różne:

- **.fsx (Skrypt):** Jest przeznaczony do uruchamiania w trybie interaktywnym (FSI). To jak "notatnik" do szybkiego testowania kodu, eksploracji danych i pisania małych skryptów.
- **.fs (Źródło):** Jest przeznaczony do *kompilacji*. To plik budujący twoją właściwą aplikację lub bibliotekę (`.exe` lub `.dll`).

Ponadto dyrektywy `#r` i `#load` są ignorowane albo powodują błędy podczas kompilacji.

Przykładowy kod klasy `StockAnalyzer`:

```
open Script
open System
open System.Globalization

type public StockAnalyzer (lprices, days) =
    let prices =
        lprices
        |> Seq.map snd
        |> Seq.take days

    static member public GetAnalyzer(ticker, days) =
        new StockAnalyzer(loadPrices ticker, days)

    static member public GetAnalyzers(tickers, days) =
        tickers
        |> Seq.map loadPrices
        |> Seq.map (fun prices -> new StockAnalyzer(prices, days))

    member s.Return =
        let lastPriceString = prices |> Seq.item 0
        let startPriceString = prices |> Seq.item (days - 1)
        let lastPrice = Double.Parse(lastPriceString, CultureInfo.InvariantCulture)
        let startPrice = Double.Parse(startPriceString, CultureInfo.InvariantCulture)
        lastPrice / startPrice - 1.0
```

```
    member s.StdDev =
        let logRets =
            prices
            |> Seq.pairwise
                |> Seq.map (fun (yStr, xStr) ->
                let x = Double.Parse(xStr, CultureInfo.InvariantCulture)
                let y = Double.Parse(yStr, CultureInfo.InvariantCulture)
                log (y / x))
        let mean = logRets |> Seq.average
        let sqr x = x * x
        let var = logRets |> Seq.averageBy (fun r -> sqr (r - mean))
        sqrt var
```

Funkcja `GetAnalyzers` działa w sposób synchroniczny i każde wykonanie funkcji `loadPrices` musi się zakończyć aby mogło zostać wykonane następne. Poniżej wersja ze zrównoleglonym działaniem.

```
static member public GetAnalyzersParal (tickers, days) =
    tickers
    |> Seq.map loadPricesAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> Seq.map (fun prices -> new StockAnalyzer(prices, days))
```

Działanie można opisać w 4 podstawowych krokach:

- `Seq.map loadPricesAsync`: Błyskawicznie tworzy listę trzech "obietnic" (zadań `Async`), ale jeszcze niczego nie uruchamia.
- `Async.Parallel`: Łączy te trzy obietnice w jedną, dużą operację, która mówi "uruchom te trzy zadania równolegle".
- `Async.RunSynchronously`: Uruchamia tę dużą operację. Zapytania (MSFT, ORCL, EBAY …) są wysyłane do sieci w tym samym czasie i czeka, aż najwolniejsze z zapytań się zakończy.
- `Seq.map (fun prices ...)`: Gdy wszystkie dane są już pobrane, błyskawicznie tworzy trzy obiekty `StockAnalyzer`.

Pomimo znacznej poprawy wydajności funkcja nie jest w pełni asynchroniczna. Zawiera ona `Async.RunSynchronously`, co oznacza, że będzie blokować wątek wywołujący aż do uzyskania wyników. Poniżej wersja tej funkcji w pełni asynchroniczna zwraca „obietnicę” `Async`, pozwalając wywołującemu zdecydować, kiedy ją uruchomić (np. `Async.RunSynchronously` w funkcji main).

```
static member public GetAnalyzersParalAsync (tickers, days) = async {
    let operations =
        tickers |> Seq.map loadPricesAsync
    let parallelOperation = Async.Parallel operations
    let! allPrices = parallelOperation
    let analyzers =
        allPrices
        |> Array.map (fun prices -> new StockAnalyzer(prices, days))
    return analyzers
}
```

Działanie można opisać w poniższych krokach:

Stwórz listę „obietnic” `loadPricesAsync` zwraca `Async<prices>`

- Stwórz operację równoległą, `Async.Parallel` zwraca `Async<prices[]>`
- Uruchom ją asynchronicznie (używając `let!`), `allPrices` to teraz `prices[]`
- Zmapuj wyniki i zwróć je z bloku `async`.

Proszę wykorzystać klasę `StockAnalyzer` w projekcie VB Console Application i wyświetlić informacje (odchylenie standardowe i stopa zwrotu) dla przynajmniej 6 firm. Przykładowo mogą to być firmy o nast. tickerze: `MSFT` – Microsoft Corporation, `ORCL` – Oracle Corporation, `EBAY` – Ebay Inc, GOOG – Alphabet Inc., `AAPL` – Apple Inc., `C` – Citigroup Inc. Pomocne informacje można znaleźć w serwisie Slickcharts, który pokazuje listę wszystkich 500 komponentów S&P 500 wraz z ich nazwami, tickerami i wagą w indeksie. https://www.slickcharts.com/sp500.

Proszę w projekcie VB Console Application wyświetlić informacje (odchylenie standardowe i stopa zwrotu) dla tych samych firm w sposób asynchroniczny. Końcowy efekt powinien być podobny do załączonego poniżej:

```
Standard Deviation is: 0,663288220093315 Return is: 1,19926522343731
Standard Deviation is: 0,626222022407931 Return is: 0,633833026367939
Standard Deviation is: 0,150890646410155 Return is: 0,554779113503946
Standard Deviation is: 0,663288220093315 Return is: 1,19926522343731
Standard Deviation is: 0,626222022407931 Return is: 0,633833026367939
Standard Deviation is: 0,150890646410155 Return is: 0,554779113503946
```

## IV. Sygnatury

*Cel: Zapoznanie z sygnaturami w F#*[11].

Język F# korzysta z wnioskowania typu w celu dedukcji typów, dzięki czemu rzadko zachodzi potrzeba jawnego określania typów w kodzie, zwłaszcza w przypadku funkcji. Wyrażenia typu mają specjalną składnię, która różni się od składni używanej w normalnych wyrażeniach. Łatwo to zauważyć korzystając z sesji interaktywnej, ponieważ typ każdego wyrażenia jest wyświetlany podczas jego ewaluacji:

```
> let same x = x;;
val same: x: 'a -> 'a
```

W F# nie można wyszukiwać funkcje biblioteczne nie są powiązane z klasami i nie można ich wyszukiwać jak metod w językach obiektowych – po wpisaniu nazwy obiektu i kropki. W takim przypadku pomocne będą sygnatury funkcji, pozwalające szybko zawęzić listę kandydatów.

Za pomocą słowa kluczowego `type` można zdefiniować własne typy tak aby dopasować je do pożądanego podpisu funkcji:

```
type Adder = int -> int
type AdderGenerator = int -> Adder
```

Można je stosować aby wymóc właściwą sygnaturę podczas definiowania funkcji:

```
let a:AdderGenerator = fun x -> (fun y -> x + y)
let b:AdderGenerator = fun (x:float) -> (fun y -> x + y)
```

[11] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/signature-files

```
    let c                = fun (x:float) -> (fun y -> x + y)
```

W przypadku drugiej definicji dostaniemy błąd:

```
...error FS0001: Oczekiwany typ tego wyrażenia:
    „int”
Typ w tym miejscu:
    „float”
```

Plik *.fsi zawiera informacje o publicznych sygnaturach takich jak typy, przestrzenie nazw i moduły. Można go użyć do zmiany ich domyślnych ustawień dostępności. W celu automatycznego wygenerowania pliku z sygnaturami należy dodać flagę kompilatora: `sig:<nazwa_pliku>.fsi` w ustawieniach lub pliku projektu:

```
  <PropertyGroup>
                          ...
      <OtherFlags>--sig:<nazwa_pliku>.fsi</OtherFlags>
  </PropertyGroup>
```

Wygenerowany plik należy dodać do projektu usuwając metody i funkcje, które mają być niedostępne. Szczegółowe zasady które muszą być spełnione można znaleźć tutaj. Dodaną flagę należy usunąć aby zapobiec ponownego wygenerowania i nadpisania zmian w pliku sygnatur. W przypadku błędu podczas generowania biblioteki może okazać się konieczne usuniecie przestrzeni nazw: `namespace FSharp`. Należy też sprawdzić w pliku projektu czy plik z deklaracjami (*.fsi) dodany jest przed odpowiadającym mu plikiem z definicjami (*.fs):

```
  <ItemGroup>
    <Compile Include="Library.fsi" />
    <Compile Include="Library.fs" />
  </ItemGroup>
```

W innym przypadku dostaniemy błąd: FS0238 informujący o tym, że definicja pliku lub modułu została już podana (plik *.fs) i kolejność plików w F# jest ważna z powodu wnioskowania typów.

## V. Designing with types

*Cel: Zapoznanie z podstawami Designing with types w F#[12].*

W projektowaniu zorientowanym na domenę powinniśmy rozróżniać składowe encji na poziome podstawowych typów. Jeśli zawiera ona przykładowo: adres email, kod pocztowy oraz numer telefonu najprostszym rozwiązaniem było by reprezentowanie ich wszystkich w postaci stringów. Nie jest to jednak zgodne z duchem DDD – czy adres email to to samo co kod pocztowy? W F# najprostszym sposobem na utworzenie osobnego typu jest opakowanie podstawowego typu, np.: string wewnątrz innego typu. Oto przykład z wykorzystaniem unii dysktyminowanej[13] obsługującej jeden przypadek (single-case discriminated union):

[12] https://fsharpforfunandprofit.com/posts/designing-with-types-single-case-dus, https://jindraivanek.hashnode.dev/f-tips-weekly-1-single-case-du

[13] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions

```
type String50 = String50 of string
```

alternatywnie można użyć rekordu z jednym polem:

```
type String50 = { String50: string }
```

Utworzenie w obydwu przypadkach wartości nowego typu jest identyczne i intuicyjne:

```
let simpleString = String50 "simpleString"
```

Jak widać w F# tworzenie wielu drobnych typów jest mniej bolesne niż w C# i Java gdzie prowadzi do tzw. „obsesji prymitywów”. Na takie typy możemy nakładać różne ograniczenia. Z pomocą mogą przyjąć odpowiedzi na pytania:

- czy dopuszczamy wartość null? – tutaj już na poziomie języka wiemy, że nie
- minimalna i maksymalna długość?
- czy może przekraczać wiele linii?
- czy może mieć rozpoczynać się lub kończyć spacją/spacjami?
- czy może zawierać znaki niedrukowalne?

Te wszystkie ograniczenia powinny być sprawdzone podczas tworzenia nowej wartości, wiemy że jej wartość już nie ulegnie zmianie więc zawsze będzie poprawna. Warto od razu utworzyć moduł szczególnie, że z czasem liczba powiązanych z naszym typem funkcji zacznie się rozrastać. Moduł[14] to grupa kodu języka F#, takiego jak wartości, typy i wartości funkcyjne[15]. Grupowanie kodu w modułach pomaga zachować powiązany kod razem i pomaga uniknąć konfliktów nazw w programie.

Moduły nie działają w skryptach interaktywnych zatem należy utworzyć plik *.*fs*:

```
/// Module containing functions related to String50 type
module String50 =
    type T = String50 of string
    let create (s:string) =
        if s <> null && s.Length <= 50
        then Some (String50 s)
        else None
```

Deklaracja wartości typu `String50`, w przypadku `toolongString` powstanie `None`:

```
let notSoLongString = String50.create "notSoLongString"
let toolongString = String50.create
"tooLongLongLongLongLongLongLongLongLongLongLongString"
```

Użycie:

```
match toolongString with
| Some sbyte -> printfn "good"
```

[14] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/modules
[15] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/functions/#function-values, https://fsharpforfunandprofit.com/posts/function-values-and-simple-values/#function-values

```
| None -> printfn "bad"
```

```
bad
```

Jeśli chcemy wykonać pewną funkcję na opakowanej wartości albo ją pobrać. Możemy zaimplementować te funkcjonalności w postaci wartości funkcyjnych naszego modułu:

```
module String50 =
                ...

    /// Apply the given function to the wrapped value
    let apply f (String50 s) = f s
    /// Get the wrapped value
    let value s = apply id s
```

Wartość funkcyjna `apply` jako argument przyjmuje funkcję `f` i wykonuje ją na wartości `s` która reprezentuję w naszym przypadku wartość opakowaną czyli string. Wartość funkcyjna `value` zwraca tę opakowaną wartość `s` wykonując funkcję `id` na `s` przy pomocy `apply`, gdzie `id` to funkcja „funkcją tożsamościowa", która po prostu zwraca swój argument. Jej implementacja jako wartości funkcyjnej mogła by być następująca: `let id = fun x -> x`.

Zauważmy, że korzystanie z `apply` nie jest wymagane i wartość funkcyjna `value` mogła by wyglądać po prostu tak: `let value (String50 s) = s`.

Teraz możemy dostać się do naszej opakowanej wartości i wykorzystać jej metody, np. `ToUpper()`:

```
notSoLongString
|> Option.map String50.value
|> Option.map (fun s -> s.ToUpper())
|> Option.iter (printfn "%s")
```

Wykorzystanie funkcji pomocniczych modułu Option[16] wynika z faktu, że `String50` jest opcjonalny. `Option.map` i `Option.iter` pozwalają wykonywać funkcje na tym typie nie musząc obsługiwać osobno przypadku dla `Some` i `None`. Różnią się od siebie tym, że `Option.map` wykonuje `Some x -> Some (f x)` więc zwraca opcje, natomiast `Option.iter` wykonuje `Some x -> f x` i jest przydatny jeśli nie interesuje nas wartość zwrócona. W przypadku `Option.map` razie musielibyśmy jawnie zignorować tę wartość:

```
|> Option.map (printfn "%s")
|> ignore
```

Bez `Option.map` i `Option.iter` nich kod wyglądał by mniej przejrzyście, np. tak:

```
match notSoLongString with
| Some s -> printf "%s" ((String50.value s).ToUpper())
| None -> ()
```

[16] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/options, https://fsharpforfunandprofit.com/posts/the-option-type/, https://www.compositional-it.com/news-blog/using-the-option-module-in-f/

albo tak:

```
let result = match notSoLongString with
             | Some s ->  (String50.value s).ToUpper()
             | None -> ""
printfn "%s" result
```

NOTSOLONGSTRING

W przypadku kiedy chcielibyśmy utworzyć typ różniący się od `String50` tylko ograniczeniem na dopuszczalną długość stringu, załóżmy byłby to `String100` najprawdopodobniej dokonalibyśmy prawie całkowitej duplikacji kodu, ponadto typy te są nieporównywalne, co nie jest oczekiwanym zachowaniem. Poniższy kod zwróci błąd:

```
let s50 = String50.create "John"
let s100 = String100.create "Smith"

let s50' = s50.Value
let s100' = s100.Value

let areEqual = (s50' = s100')  // compiler error
```

Rozwiązaniem jest utworzenie wspólnego interfejsu[17] `IWrappedString` który będzie wspierał wszystkie nasze opakowane stringi, deklarującego kilka standardowych funkcji (`create`, `equals`, `compareTo`):

```
module WrappedString =

    type IWrappedString =
        abstract Value : string

    let create canonicalize isValid ctor (s:string) =
        if s = null
        then None
        else
            let s' = canonicalize s
            if isValid s'
            then Some (ctor s')
            else None

    let apply f (s:IWrappedString) =
        s.Value |> f

    let value s = apply id s

    let equals left right =
        (value left) = (value right)

    let compareTo left right =
        (value left).CompareTo (value right)
```

Kluczową funkcją jest `create`, wykorzystująca kontynuacje. Kontynuacja to po prostu funkcja przekazywana do innej funkcji w celu wskazania jej, co ma zrobić dalej. Przyjmuje ona funkcję kanonizującą, walidującą i konstruktor. Ponieważ wartości w F# są niezmienne najlepszym momentem na ich normalizację i walidację jest faza konstrukcji. Utworzenie

[17] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/interfaces

nowej wartości następuje jedynie gdy walidacja zakończy się pomyślnie. W pierwszym kroku dane wejściowe są kanonizowane. Kanonizacja to inaczej normalizacja/standaryzacja mająca na celu ujednolicenie typu danych, który może występować w wielu różnych formach do ujednoliconej postaci, np.: adresów e-mail pisane małymi literami, przycinanie początkowych/końcowych/nadmiarowych spacji itp. Następnie dokonywana jest walidacja, która zwraca typ Opcji (unia z dwoma przypadkami `Some` i `None`). Jeśli walidacja się powiedzie, zwracana jest wartość z podanego konstruktora opakowana w `Some`. W przypadku błędu walidacji zwracana jest wartość `None`. Wartości null nigdy nie będą prawidłowe w tym scenariuszu.

```
    let singleLineTrimmed s =
        System.Text.RegularExpressions.Regex.Replace(s,"\s"," ").Trim()

    let lengthValidator len (s:string) =
        s.Length <= len

    type String100 = String100 of string with
        interface IWrappedString with
            member this.Value = let (String100 s) = this in s

    let string100 = create singleLineTrimmed (lengthValidator 100) String100

    let convertTo100 s = apply string100 s

    type String50 = String50 of string with
        interface IWrappedString with
            member this.Value = let (String50 s) = this in s

    let string50 = create singleLineTrimmed (lengthValidator 50)  String50

    let convertTo50 s = apply string50 s
```

Następnie w tym samym module `WrappedString` definiujemy dwa nowe typy implementujące nasz interfejs `IWrappedString` – `String50` oraz `String100`. Tworzymy dla nich specjalizowane funkcje konstruujące **`string50`** i **`string100`** przekazując do funkcji **`create`** odpowiednie kontynuacje:

- kanonizator: **`singleLineTrimmed`**
- walidator: **`lengthValidator len`**
- konstruktor: `String50` albo `String100`.

Poniżej przykład skonstruowania oraz użycia takich opakowanych stringów:

```
let s50 = WrappedString.string50 "abc" |> Option.get
printfn "s50 is %A" s50
let bad = WrappedString.string50 null
printfn "bad is %A" bad
let s100 = WrappedString.string100 "abc" |> Option.get
printfn "s100 is %A" s100

printfn "s50 is equal to s100 using module equals? %b" (WrappedString.equals s50
s100) //true
printfn "s50 is equal to s100 using Object.Equals? %b" (s50.Equals s100) //false
//printfn "s50 is equal to s100? %b" (s50 = s100) // compiler error
```

W celu porównania wartości typów implementujących `IWrappedString` możemy wykorzystać metodę **`equals`**, metoda **`Equals`** udostępniana przez klasę **`Object`** zwróci **`false`** ponieważ porównuje referencje, natomiast użycie operatora porównania = spowoduje błąd kompilacji.

W kolejnym kroku możemy tworzyć bardziej specjalizowane typy wykorzystując moduł `WrappedString`. Przykładowo `WrappedString` będzie bazował na stringu ujednolicanym do pojedynczej linii oraz nie dłuższym niż 100 znaków ale dodatkowo będzie walidowany przy pomocy wyrażenia regularnego pod kątem poprawności z adresem email:

```
module EmailAddress =

    type T = EmailAddress of string with
        interface WrappedString.IWrappedString with
            member this.Value = let (EmailAddress s) = this in s

    let create =
        let canonicalize = WrappedString.singleLineTrimmed
        let isValid s =
            (WrappedString.lengthValidator 100 s) &&
            System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        WrappedString.create canonicalize isValid EmailAddress

    /// Converts any wrapped string to an EmailAddress
    let convert s = WrappedString.apply create s
```

Utworzenie adresów email:

```
let address1 = EmailAddress.create "x@example.com" //x@example.com
let address2 = EmailAddress.create "example.com"   //null
```

Możemy utworzyć ogólniejszą metodę `createWithCont` przyjmującą dwie funkcje, jedną wykonywaną w przypadku sukcesu (która przyjmuje nowo skonstruowany e-mail jako parametr), a drugą wykonywaną w przypadku niepowodzenia (która przyjmuje ciąg błędu jako parametr):

```
let createWithCont success failure =
    let canonicalize = WrappedString.singleLineTrimmed
    let isValid success failure s =
        if (WrappedString.lengthValidator 100 s) &&
            System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then success (EmailAddress s)
                true
            else failure "Email address must contain an @ sign and be shorter
than 100 signs"
                false
    WrappedString.create canonicalize (isValid success failure) EmailAddress
```

Teraz możemy utworzyć nową funkcję `createEmailAddress` z dodatkowym zachowaniem przekazanym w postaci kontynuacji `success` i `failure`.

```
let success (EmailAddress.EmailAddress s) = printfn "success creating email %s" s
let failure  msg = printfn "error creating email: %s" msg
let createEmailAddress = EmailAddress.createWithCont success failure
```

Konstrukcja nowych wartości typu `EmailAddress`:

```
let address3 = createEmailAddress "example.com"
//error creating email: Email address must contain an @ sign and be shorter than
100 signs
let address4 = createEmailAddress x@example.com
//success creating email x@example.com
```

Kolejnym poziomem zapewnienia poprawności naszym typom będzie wykorzystanie dyskryminowanych unii[18] dzięki którym nielegalne stany możemy uczynić niereprezentowalnymi. Przykładowo, jeśli chcemy aby nasz typ reprezentujący informacje do kontaktu zawierał przynajmniej jeden adres (klasyczny albo elektroniczny), to możemy go zaimplementować w następujący sposób:

```
type ContactInfo =
    | EmailOnly of EmailAddress.T
    | PostOnly of string
    | EmailAndPost of EmailAddress.T * string
```

Widzimy że stan w którym nie podano żadnego kontaktu jest niereprezentowany więc zabroniony. Dla uproszczenia przykładu adres klasyczny przechowywany jest w stringu, w realnym zastosowaniu utworzylibyśmy dla niego dedykowany typ podobny do `EmailAddress`.

[18] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions