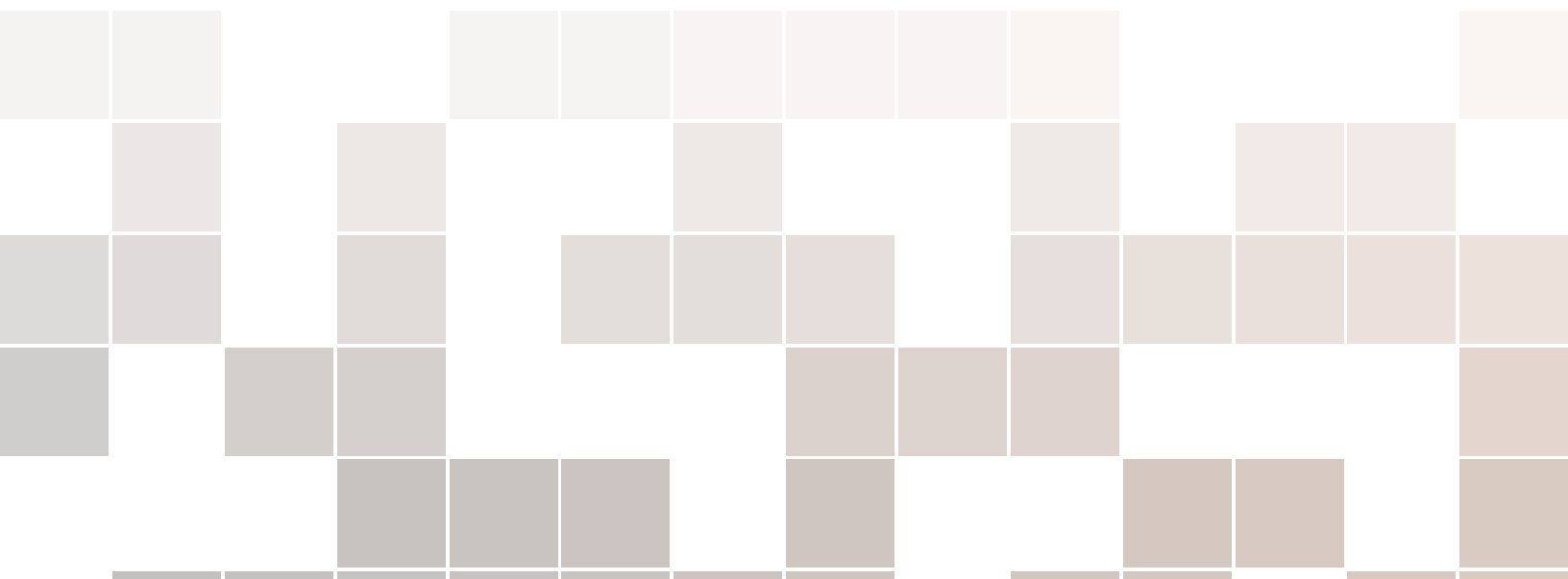


Programowanie funkcyjne

Materiały z wykładu i laboratorium

Łukasz Bartczuk



5. Wzorce programowania funkcyjnego

Na tych zajęciach będziemy poznawać podstawowe wzorce programowania funkcyjnego:

1. częściową aplikację i currying
2. potoki
3. złożenie funkcji
4. monady

5.1 Aplikacja częściowa i currying

Bardzo przydatnym narzędziem podczas tworzenia programów funkcyjnych jest częściowa aplikacja funkcji do jej argumentów oraz mechanizm pozwalający traktować funkcje wielu argumentów jako zestaw funkcji jednoargumentowych.

5.1.1 Aplikacja częściowa

Aplikacja jest procesem zastosowania funkcji do jej argumentów w celu uzyskania wartości. Jest to proces który wszyscy bardzo dobrze rozumiemy - przynajmniej na wysokim poziomie.

Wróćmy do programu, który był prezentowany na poprzednich zajęciach (zobacz punkt ??). Poniżej pokazuję tylko jego funkcje `wSrodku` oraz `main` ponieważ są one najważniejsze dla naszych obecnych rozważań:

```
let wSrodku okrag punkt =  
  (odleglosc okrag.srodek punkt) < okrag.promien
```

```
let pokazPunkt (x,y) = printfn "%f %f" x y

[<EntryPoint>]
let main argv =

    let okrag = wczytajOkrag ()
    let punkty = wczytajPunkty "punkty.txt"

    let punktyWSrodku =
        Seq.filter (fun p -> wSrodku okrag p) punkty
    Seq.iter pokazPunkt punktyWSrodku
    0
```

Zwróćmy szczególną uwagę, na fragment:

```
let punktyWSrodku =
    Seq.filter (fun p -> wSrodku okrag p) punkty
```

Przekazuje w nim do funkcji `Seq.filter` funkcję anonimową, wewnątrz której aplikuję funkcję `wSrodku` do wartości `okrag` oraz `p`. Przedstawiona funkcja anonimowa jest nam w tym przypadku potrzebna ponieważ operacja filtrowania oczekuje funkcji jednego parametru nie dwóch. Zauważmy jednak, że w przypadku tego programu wartość `okrag` pozostaje nie zmieniona we wszystkich wywołaniach tej funkcji. Z rozważań na temat funkcji wyższych rzędów wiemy, że są to funkcje, które przyjmują inne funkcje jako swoje argumenty (jak w przypadku `filter`) lub funkcje zwracają (stają się wtedy generatorami funkcji). Moglibyśmy stworzyć taki generator również w tym przypadku:

```
let wSrodkuOkregu okrag =
    fun punkt -> (odleglosc okrag.srodek punkt) < okrag.promien
```

Jego zastosowanie mogłoby wyglądać następująco:

```
let sprawdz = wSrodkuOkregu okrag
let punktyWSrodku = Seq.filter (fun p -> sprawdz p) punkty
```

Dostaniemy ten sam rezultat, co poprzednio, ale teraz robimy to w dwóch krokach. Najpierw generujemy funkcję pozwalającą na sprawdzenie czy dany punkt leży wewnątrz konkretnego (nie dowolnego) okręgu, a następnie wykorzystujemy tę funkcję przetestowania punktów z kolekcji. Ponieważ teraz funkcja anonimowa tylko i wyłącznie przekazuje swój parametr do utworzonej funkcji `sprawdz`, możemy uprościć ten zapis do postaci:

```
let sprawdz = wSrodkuOkregu okrag
let punktyWSrodku = Seq.filter sprawdz punkty
```

Ponieważ funkcja `sprawdz` wykonuje te same działania co funkcja `wSrodku`, ale dla konkretnego okręgu możemy powiedzieć że jest to ta sama funkcja z częściowo zaaplikowanymi argumentami. Dochodzimy więc do pierwszego wzorca, bardzo często wykorzystywanego w programowaniu funkcyjnym - częściowej aplikacji.

Częściowa aplikacja - jest to aplikacja funkcji tylko do niektórych jej argumentów. W jej rezultacie otrzymujemy funkcję z mniejszą ilością parametrów.

Jest to mechanizm bardzo użyteczny, gdyż pozwala wcześniej ustawić pewne - niezmiennie argumenty funkcji, tak aby później nie trzeba było już o nich myśleć.

5.1.2 Currying

Jeżeli zdefiniujemy dowolną funkcję jednego parametru, to jej sygnaturę będziemy mogli zapisać jako:

$$f : \text{typParametru} \rightarrow \text{typFunkcji}$$

Pokazana strzałka informuje nas że jest to funkcja. Z podobnego zapisu korzysta również interpreter F#. Przykładowo dla funkcji:

```
let inkrementuj x = x+1
```

określi on następującą sygnaturę:

```
val inkrementuj : x:int -> int
```

Najpierw podał typ parametru następnie strzałkę oraz typ funkcji. Mówi nam to, że symbol `inkrementuj` jest przypisany do funkcji przekształcającej wartość typu `int` na inną wartość typu `int`.

Przeanalizujmy teraz dokładnie pokazaną wcześniej funkcję `wSrodkuOkregu`. Jest to funkcja jednego parametru `Okrag`, która zwraca kolejną funkcję jednego parametru `float*float`. Jej sygnatura będzie następująca:

```
val wSrodkuOkregu : okrag:Okrag -> float * float -> bool
```

Uzyskana teraz sygnatura mówi nam, że funkcja `wSrodkuOkregu` jest funkcją jednego parametru typu `Okrag`, która zwraca kolejną funkcję. Ta funkcja z kolei przyjmuje wartość typu `float * float` i zwraca wartość typu `bool`.

Wyraźniej byłoby to widać, gdybyśmy tę sygnaturę zapisali w następujący sposób:

```
val wSrodkuOkregu : okrag:Okrag -> (float * float -> bool)
```

Jeżeli troszkę zmienimy ciało tej funkcji do postaci:

```
let wSrodku okrag punkt =
    (odleglosc okrag.srodek punkt) < okrag.promien

let wSrodkuOkregu okrag = fun punkt -> wSrodku okrag punkt
```

W tym przypadku sygnatura naszej funkcji nie zmieni się, ale możemy teraz powiedzieć, że funkcja `wSrodkuOkregu` zamienia dwuargumentową funkcję `wSrodku` na dwie funkcje jednoargumentowe. Mechanizm ten określa się mianem **curring-u**¹.

Curring - mechanizm pozwalający na zamianę funkcji wieloargumentowej w funkcję jednego argumentu.

Przyjrzyjmy się jednak sygnaturze funkcji `wSrodku`:

```
val wSrodku : okrag:Okrag -> float * float -> bool
```

Zwróćmy uwagę, że jest ona dokładnie taka sama jak sygnatura funkcji `wSrodkuOkregu`. Oznacza to, że F# automatycznie poddaje każdą funkcję wieloargumentową procesowi curringu i pozwala je traktować jako cały zestaw funkcji jednoargumentowych. Korzystając z funkcji anonimowych to samo moglibyśmy zapisać w formie:

```
let wSrodku =
    fun okrag ->
        fun punkt ->
            (odleglosc okrag.srodek punkt) < okrag.promien
```

co najlepiej obrazuje jak F# traktuje funkcje wieloargumentowe.

Wracając do programu, który był pokazywany na początku tego punktu, wykorzystując automatyczny curring, moglibyśmy go zapisać w postaci:

```
let wSrodku okrag punkt =
    (odleglosc okrag.srodek punkt) < okrag.promien

let pokazPunkt (x,y) = printfn "%f %f" x y

[<EntryPoint>]
let main argv =

    let okrag = wczytajOkrag ()
    let punkty = wczytajPunkty "punkty.txt"

    let punktyWSrodku = Seq.filter (wSrodku okrag) punkty
```

¹Nazwa ta powstała od nazwiska Haskell'a Curry-iego, który miał bardzo duży wpływ na teoretyczne podstawy języków funkcyjnych

```
Seq.iter pokazPunkt punktyWSrodku
0
```

Zwróćmy jeszcze na chwilę uwagę na funkcję `pokazPunkt`. Choć z punktu widzenia jej ciała możemy ją traktować jako dwuargumentową (x i y), to jest to funkcja jednego argumentu, który jest krotką. Obie wartości musimy podać w tym momencie jednocześnie i nie możemy tego rozdzielić na kolejne kroki. Dlatego jeżeli będziemy wymagali, aby wszystkie (lub niektóre) parametry naszej funkcji były podane jednocześnie musimy wymagać aby były one przekazane jako krotka.

5.2 Potoki

Podstawową jednostką kodu w językach funkcyjnych jest wyrażenie. Nad wyrażeniami możemy budować abstrakcję w postaci funkcji, które później możemy aplikować do argumentów. Tworzenie programów polega na łączeniu ze sobą małych funkcji rozwiązujących dobrze pojedynczy mały problem, w większą całość rozwiązującą duże, złożone zagadnienie.

Przykładowo na poprzednich zajęciach jedno z zadań polegało na wczytaniu z pliku zestawu wartości opisujących parametry równania kwadratowego, określeniu dla każdego zestawu ile rozwiązań ma dany wielomian, podzieleniu współczynników względem tej wartości na grupy i określenie ile zestawów jest w każdej grupie.

Zadanie to może być rozwiązane w następujący sposób:

```
type RozwiazanieRownaniaKwadratowego =
  | BrakRozwiazan
  | JednoRozwiazanie of float
  | DwaRozwiazania of float*float
  | RownanieLiniowe

let rownanieKwadratowe (a:float,b:float,c:float) =
  if a = 0.0 then
    RownanieLiniowe
  else
    let delta = b**2.0-4.0*a*c
    if delta<0.0 then
      BrakRozwiazan
    elif delta=0.0 then
      JednoRozwiazanie (-b/(2.0*a))
    else
      DwaRozwiazania ((-b-(sqrt delta)/(2.0*a)),
                      (-b+(sqrt delta)/(2.0*a)))
```

```
type LiczbaRozwiazan =
  | Brak
  | Jedno
  | Dwa
  | Liniowe

let okreslLiczbeRozwiazan = function
  | DwaRozwiazania _ -> Dwa
  | JednoRozwiazanie _ -> Jedno
  | RownanieLiniowe -> Liniowe
  | BrakRozwiazan -> Brak

let wczytajWspolczynniki nazwaPliku =
  let linie = System.IO.File.ReadLines nazwaPliku
  let linie = Seq.map
    (fun (linia:string) ->
      List.ofArray (linia.Split(' ')))
    linie
  let wartosci = Seq.map (List.map float) linie
  Seq.map (fun (wsp:float list) -> (wsp.[0], wsp.[1], wsp.[2])) <->
    wartosci

[<EntryPoint>]
let main argv =
  let nazwaPliku = "d:\przyklady\zad8.txt"
  let wspolczynniki = wczytajWspolczynniki nazwaPliku
  let rozwiazania = Seq.map (fun wsp -> rownanieKwadratowe wsp)
    wspolczynniki
  let grupy = Seq.groupBy okreslLiczbeRozwiazan rozwiazania
  let grupy = Seq.toList grupy
  List.iter
    (fun (k,g)-> printfn "%A %d" k (Seq.length g))
    grupy

0
```

Zwróćmy uwagę na funkcję `main`. Zauważmy, że realizuje ona następujący przepływ danych:

1. zamienia nazwę pliku na kolekcję współczynników,
2. zamienia kolekcję współczynników na kolekcję rozwiązań równania kwadratowego,

3. dzieli kolekcję rozwiązań na sekwencję grup,
4. zamienia sekwencję grup na listę grup,
5. do każdego elementu grupy aplikuje akcję.

Rezultaty poszczególnych kroków są przypisywane do symboli, które są wykorzystywane tylko po to aby przekazać wartości z jednego kroku do drugiego. Możemy z nich zrezygnować jednak w takim przypadku funkcja `main` przyjmie następującą postać:

```
[<EntryPoint>]
let main argv =

    List.iter
        (fun (k,g)-> printfn "%A %d" k (Seq.length g))
        (Seq.toList
            (Seq.groupBy okreslLiczbeRozwiazan
                (Seq.map (fun wsp -> rownanieKwadratowe wsp)
                    (wczytajWspolczynniki "d:\przyklady\zad8.txt")))))
    0
```

W powyższym przykładzie zrezygnowałem z symboli pośrednich, co skutkuje jednak koniecznością zapisania całego przepływu danych w odwrotnej kolejności. Zauważmy, że w tym momencie nazwa pliku od której zaczyna się nasz przepływ jest umieszczona na samym końcu. Można więc powiedzieć, że czytelność naszego programu znacznie spadła.

W F# mamy jednak jeszcze jedną opcję, możemy wykorzystać operator przetwarzania potokowego `|>`.

Jest on zdefiniowany w bardzo prosty sposób:

```
let (|>) x f = f x
```

Operator ten jest zdefiniowany jako funkcja, która przyjmuje dwa argumenty i aplikuje drugi argument do pierwszego. Zwróćmy uwagę, że pozwala on na odwrócenie kolejności w jakiej aplikujemy funkcje. Normalnie najpierw podajemy funkcje, a dopiero wyrażenie określające jej argument. Dzięki zastosowaniu operatora potokowego możemy najpierw podać wyrażenie, a potem funkcje np:

```
// normalnie napiszemy:
Console.WriteLine 1

// z operatorem potokowym:
1 |> Console.WriteLine
```

Zastosowanie tego operatora do programu, który pokazałem wcześniej, będzie skutkowało uzyskaniem następującego kodu:

```
[<EntryPoint>]
let main argv =
    "d:\przyklady\zad8.txt"
    |> wczytajWspolczynniki
    |> Seq.map (fun wsp -> rownanieKwadratowe wsp)
    |> Seq.groupBy okreslLiczbeRozwiazan
    |> Seq.toList
    |> List.iter (fun (k,g)-> printfn "%A %d" k (Seq.length g))

0
```

Otrzymujemy określony wcześniej przepływ danych jednak bez konieczności definiowania dodatkowych symboli. Po raz kolejny składnia języka F# pokazała swoje piękno. Żadnych zbędnych elementów.

5.3 Złożenie funkcji

Przepisując funkcję `wczytajWspolczynniki`, tak aby korzystała z operatora potokowego uzyskamy następujący kod:

```
let wczytajWspolczynniki nazwaPliku =
    nazwaPliku
    |> System.IO.File.ReadLines
    |> Seq.map (fun linia -> List.ofArray (linia.Split(' ')))
    |> Seq.map (List.map float)
    |> Seq.map (fun wsp -> (wsp.[0], wsp.[1], wsp.[2]))
```

Zauważmy, że w tym wypadku znowu parametr funkcji wykorzystujemy tylko i wyłącznie do rozpoczęcia potoku - nigdzie indziej w tej funkcji z niego nie korzystamy. Również w tym przypadku F# ma składnię (a dokładniej operator), która umożliwia nam uniknięcie zbędnego pisania. Jest nim złożenie funkcji, zdefiniowane w następujący sposób:

```
f >> g = g (f x)
```

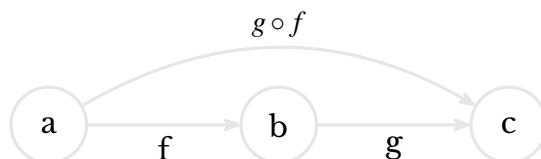
Dzięki niemu poniższą funkcję możemy przedstawić w:

```
let wczytajWspolczynniki =
    System.IO.File.ReadLines
    >> Seq.map (fun linia -> List.ofArray (linia.Split(' ')))
    >> Seq.map (List.map float)
    >> Seq.map (fun wsp -> (wsp.[0], wsp.[1], wsp.[2]))
```

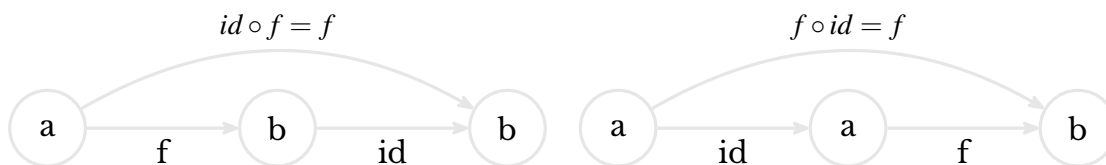
5.4 Funktory



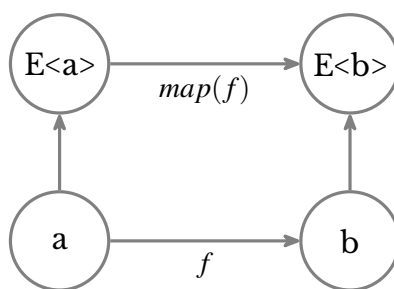
Rysunek 5.1: Funkcja tożsamościowa



Rysunek 5.2: Złożenie funkcji



Rysunek 5.3: Złożenie funkcji

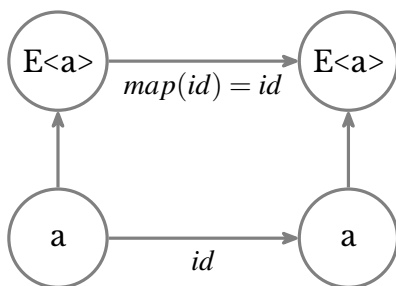


Rysunek 5.4: Funktor mapowanie

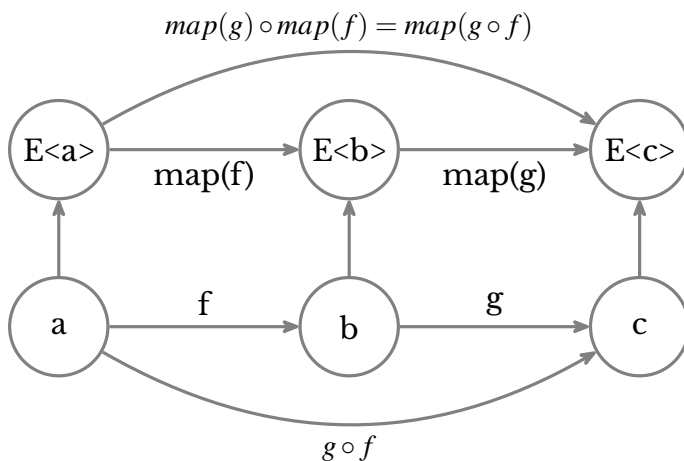
5.5 Opcje

Wróćmy na chwilę do programu służącego do sprawdzania czy punkty wczytane z pliku znajdują się wewnątrz okręgu. Umieściłem tam m.in. funkcję pozwalającą na tworzenie okręgu na podstawie informacji dostarczonych przez użytkownika z klawiatury:

```
let wczytajOkrag () =  
    let x = float (wczytajWartosc "Podaj współrzędna x: ")
```



Rysunek 5.5: Funktor mapowanie - prawo tożsamości



Rysunek 5.6: Funktor mapowanie - prawo złożenia

```
let y = float (wczytajWartosc "Podaj współrzędna y: ")
let r = float (wczytajWartosc "Podaj promień: ")
{ srodek = (x,y); promien = r }
```

Funkcja ta chodź dość jasna i czytelna ma ukrytą jedną bardzo poważną wadę. Co się stanie jeżeli użytkownik wprowadzi łańcuch znaków, który nie reprezentuje poprawnej liczby. Oczywiście funkcja `float` nie może poprawnie wykonać swojego zadania i zgłosi bardzo brzydki wyjątek:

```
let wczytajOkrag () =
    let x = float (wczytajWartosc "Podaj współrzędną x: ")
    let y = float (wczytajWartosc "Podaj współrzędną y: ")
    let r = float (wczytajWartosc "Podaj promień: ")
    {srodek = (x,y); promien = r}

let odleglosc (x1,y1)
    sqrt ((x1-x2)**2.

let wSrodku (okrag:Ok
```

Exception Unhandled
System.FormatException: 'Input string was not in a correct format.'

View Details | Copy Details | Start Live Share session...

Exception Settings

Rozwiązaniem naszego problemu może być zastosowanie funkcji `Double.TryParse` w miejsce `float`. Funkcja ta zwraca dwie wartości: pierwsza to wartość logiczna określająca czy operacja się powiodła, druga to liczba rzeczywista powstała w wyniku konwersji.

Zwracany znacznik moglibyśmy wykorzystać do sprawdzenia, czy wszystkie dane zostały wczytane poprawnie i tylko w takim przypadku zwracamy utworzony okrąg:

```
let wczytajOkrag () =
    let jestX, x = Double.TryParse
        (wczytajWartosc "Podaj współrzędna x: ")
    let jestY, y = Double.TryParse
        (wczytajWartosc "Podaj współrzędna y: ")
    let jestR, r = Double.TryParse
        (wczytajWartosc "Podaj promień: ")
    if jestX && jestY && jestR then
        { srodek = (x,y); promien = r }
    else
        ???
```

Oczywiście jeżeli użytkownik wprowadzi dane poprawnie, to z określeniem wyniku funkcji nie ma problemu, co jednak powinniśmy zwrócić jeżeli dane będą błędne?

W programowaniu imperatywnym odpowiedź jest prosta. Najczęściej zwracana jest wartość `null`. Decyzja ta wiąże się jednak z wieloma problemami. Spróbujmy odtworzyć ten mechanizm w F#. Jest to trochę skomplikowane, ponieważ `null` nie jest poprawną wartością dla rekordów. W tym jednym przypadku musimy wykorzystać klasy w F# ²:

```
[<AllowNullLiteral>]
type Okrag(x:float,y:float, r:float) =
    member this.srodek = (x,y)
    member this.promien = r

let wczytajOkrag () =
    let jestX, x = Double.TryParse
        (wczytajWartosc "Podaj współrzędna x: ")
    let jestY, y = Double.TryParse
        (wczytajWartosc "Podaj współrzędna y: ")
    let jestR, r = Double.TryParse
        (wczytajWartosc "Podaj promień: ")
    if jestX && jestY && jestR then
        new Okrag(x,y,r);
    else
        null

let wSrodku (okrag:Okrag) punkt =
```

²nie będę w tym miejscu opisywał składni klas w F# bo nie jest nam to potrzebne

```
(odleglosc okrag.srodek punkt) < okrag.promien
```

Zwróćmy uwagę na dwie istotne kwestie w powyższym fragmencie:

1. nawet wykorzystanie klas nie umożliwia nam bezpośrednie korzystanie z wartości `null`. Dopiero dołączenie atrybutu [`<AllowNullLiteral>`] to umożliwi, gdy powiemy F#, że na pewno wiemy co robimy.
2. musimy jawnie określić typ parametry funkcji `wSrodku` - w przeciwieństwie do rekordów F# nie określa nazw klasy na podstawie jej składowych.

Z tego wynika, że F# na prawdę nie lubi wartości `null`. Ale dlaczego? Odpowiedź znajdziemy podczas próby wykorzystania tej funkcji w funkcji `main`:

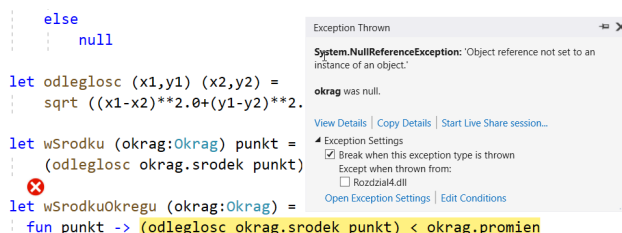
```
let pair b a = (a, b)

[<EntryPoint>]
let main argv =

    let okrag = wczytajOkrag ()
    okrag
    |> wSrodku
    |> pair (wczytajPunkty "punkty.txt")
    ||> Seq.filter
    |> Seq.iter pokazPunkt

0
```

Przy tak napisanej funkcji, jeżeli użytkownik wprowadzi złe dane, to uzyskamy słynny wyjątek:



```

else
    null

let odleglosc (x1,y1) (x2,y2) =
    sqrt ((x1-x2)**2.0+(y1-y2)**2.0)

let wSrodku (okrag:Okrag) punkt =
    (odleglosc okrag.srodek punkt)

let wSrodkuOkregu (okrag:Okrag) =
    fun punkt -> (odleglosc okrag.srodek punkt) < okrag.promien

```

Exception Thrown
System.NullReferenceException: 'Object reference not set to an instance of an object.'
okrag was null.
View Details | Copy Details | Start Live Share session...
Exception Settings
[x] Break when this exception type is thrown
Except when thrown from:
[] Rozdzial4.dll
Open Exception Settings | Edit Conditions

Jest to błąd, który bardzo często pojawia się w aplikacjach o ile nie sprawdzimy, czy rezultat funkcji nie jest wartością `null`:

```
[<EntryPoint>]
let main argv =
    let okrag = wczytajOkrag ()
    if okrag <> null then
        okrag
        |> wSrodku
```

```
|> pair (wczytajPunkty "punkty.txt")
||> Seq.filter
|> Seq.iter pokazPunkt
0
```

Taki styl programowania określany jest mianem programowania defensywnego i może prowadzić, jak to określił pomysłodawca tej wartości Tony Hoare, do błędu za milion dolarów (zobacz):

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

Zwróćmy również uwagę na sygnaturę funkcji `wczytajOkrag`: `() -> Okrag`. Na jej podstawie możemy domniemywać, że funkcja ta zawsze zwróci nam instancję typu `Okrag`. Jednak tutaj może nas spotkać niemiła niespodzianka w postaci wartości `null`, której możemy się nie spodziewać. O takich funkcjach mówimy, że są nieuczciwe.

Na szczęście w językach funkcyjnych możemy odnaleźć bardzo eleganckie rozwiązanie tego problemu. Jest nim typ `Option<'t>`, zapisywany częściej jako: `'t option`. Jest on zdefiniowany jako unia z dyskriminatorem:

```
type Option<'T> =
| None
| Some of 'T
```

Przypadek `Some` oznacza, że mamy jakiś rezultat, z kolei przypadek `None`, że funkcja nie zwraca z jakichś względów nie zwróciła wartości. Możemy teraz wykorzystać ten typ w funkcji `wczytajOkrag`:

```
let wczytajOkrag () =
    let jestX, x = Double.TryParse
        (wczytajWartosc "Podaj współrzędna x: ")
    let jestY, y = Double.TryParse
        (wczytajWartosc "Podaj współrzędna y: ")
    let jestR, r = Double.TryParse
        (wczytajWartosc "Podaj promien: ")
    if jestX && jestY && jestR then
        Some {srodek = (x,y); promien = r}
    else
        None
```

Sygnaturę tej funkcji możemy teraz zapisać następująco `wczytajOkrag: () -> Option<Okrag>`. Mówi nam ona, że wartość zwracana z funkcji jest opcjonalna, czyli może zdarzyć się przypadek, że nie będzie ona istniała. Dzięki zastosowaniu opcji możemy również wrócić do stosowania rekordów do opisu okręgu, ponieważ nie potrzebujemy już wartości `null`. Jak mam nadzieję pamiętacie, wykorzystanie unii z dyskriminatorem wymaga zastosowania mechanizmu dopasowania wzorców, przez co funkcja `main` będzie wyglądała następująco:

```
[<EntryPoint>]
let main argv =
    let okrag = wczytajOkrag ()
    match okrag with
    | Some okrag ->
        okrag
        |> wSrodku
        |> pair (wczytajPunkty "punkty.txt")
        ||> Seq.filter
        |> Seq.iter pokazPunkt
    | None -> ()
    0 // return an integer exit code
```

Powyższy kod oznacza, że jeżeli funkcja `wczytajOkrag` zwróci nam okrąg to wykonaj na nim jakieś działania, w przeciwnym przypadku jeżeli zwróci `None` nie rób nic. Na chwilę obecną nie widzimy z zastosowania opcji większego pożytku (poza zapewnieniem uczciwości funkcji). Jednak F# zawiera cały zestaw funkcji, które pozwalają na wykonywanie operacji na opcjach. Znajdują się one w module `Option`. Znajdziemy tam `m.in.` funkcję wyższych rzędów `map`, która jest zdefiniowana w następujący sposób:

```
let map mapa opcja =
    match opcja with
    | None -> None
    | Some x -> Some (mapa x)
```

Zwróćmy uwagę na sposób działania funkcji `map`. Jeżeli jako atrybut `opcja` zostanie przekazana wartość `None` to na wyjściu otrzymamy wartość `None`. Jeżeli jednak będzie to `Some x` to w wydobywamy zapisaną w tej opcji wartość, przekształcamy ją za pomocą funkcji przekazanej jako `mapa` i wynik zapisujemy z powrotem do `Some`.

Funkcję tę możemy wykorzystać w następujący sposób:

```
[<EntryPoint>]
let main argv =
    let okrag = wczytajOkrag ()
    Option.map (fun okrag ->
```



```

    okrag
    |> wSrodku
    |> pair (wczytajPunkty "punkty.txt")
    ||> Seq.filter
    |> Seq.iter pokazPunkt
  ) okrag
  |> ignore
0 // return an integer exit code

```

Jednak jest on mało interesujący. Mając na uwadze, że funkcja `map` przyjmuje mapę i mapę zwraca, możemy dokonać złożenia kolejnych wywołań tej funkcji w następujący sposób:

```

[<EntryPoint>]
let main argv =
    wczytajOkrag ()
    |> Option.map wSrodku
    |> Option.map (pair (wczytajPunkty "punkty.txt"))
    |> Option.map
        (fun (predykat,punkty) -> Seq.filter predykat punkty)
    |> Option.map (Seq.iter pokazPunkt)
    |> ignore
0 // return an integer exit code

```

Zauważmy, że w tym przypadku przekazujemy pomiędzy kolejnymi wywołaniami funkcji `Option.map` opcji, dlatego podczas aplikowania operacji filtrowania sekwencji nie możemy skorzystać z operatora `||>`. Jeżeli jednak przypomnimy sobie, że operatory w F# możemy traktować jak funkcje, oraz zastosujemy operator podwójnego odwrotnego potoku `<||`, to funkcja `main` uprości się do postaci:

```

[<EntryPoint>]
let main argv =
    wczytajOkrag ()
    |> Option.map wSrodku
    |> Option.map (pair (wczytajPunkty "punkty.txt"))
    |> Option.map ((<||) Seq.filter)
    |> Option.map (Seq.iter pokazPunkt)
    |> ignore
0

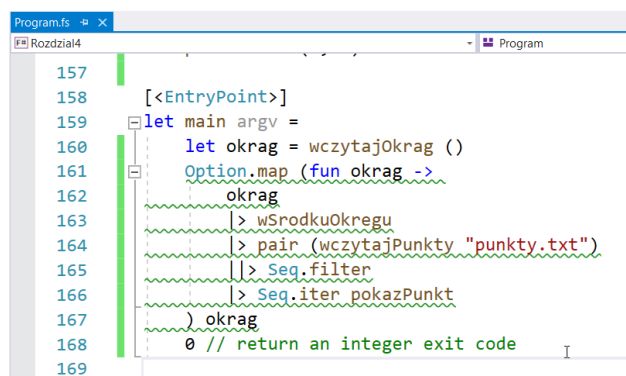
```

Idąc dalej możemy zdefiniować dla funkcji `Option.map` własny operator, co w ostatecznej wersji nam nadać funkcji `main` następujący kształt:

```
let (?|>) opcja mapa = Option.map mapa opcja
```

```
[<EntryPoint>]
let main argv =
    wczytajOkrag ()
    ?|> wSrodku
    ?|> (pair (wczytajPunkty "punkty.txt"))
    ?|> ((<|>) Seq.filter)
    ?|> (Seq.iter pokazPunkt)
    |> ignore
    0
```

We wszystkich przedstawionych powyżej postaciach funkcji `main` na końcu jest zastosowana funkcja `ignore`. Jej wykorzystanie nie jest konieczne, ale bez niej cały ciąg wyrażeń jest podświetlany w Visual Studio:



Jest to związane z tym, że wyniku działania funkcji `Option.map` uzyskujemy wartość, z której później nie korzystamy, co nie do końca odpowiada środowisku. Przystanie się ono jednak burzyć, jeżeli za pomocą funkcji `ignore` pokażemy, że świadomie ignorujemy ten rezultat.

Przejdźmy teraz do funkcji `wczytajPunkty`, która po zastosowaniu operatora potokowego może przyjąć postać:

```
let wczytajPunkty nazwaPliku =
    nazwaPliku
    |> File.ReadLines
    |> Seq.map (fun l -> l.Split(' ')) linie
    |> Seq.map zamienNaPunkt
```

Jednak w tej funkcji również czai się niebezpieczeństwo. Już sama funkcja `File.ReadLines` potrafi zgłosić aż 8 różnych wyjątków. Oczywiście wyjątki możemy przechwycić wykorzystując blok `try ... with ...` jak pokazano poniżej:

```
let wczytajPunkty nazwaPliku =
    try
        nazwaPliku
        |> File.ReadLines
        |> Seq.map (fun l -> l.Split(' '))
        |> Seq.map zamienNaPunkt
        |> Some
    with
        _ -> None
```

Działanie tej funkcji jest bardzo proste. Jeżeli nie zostanie zgłoszony żaden wyjątek, to zwróci ona sekwencję punktów opakowaną w opcję `Some`. Jeżeli jednak zgłoszony zostanie wyjątek, to będzie on przechwycony przez blok `try ... with ...` i zwrócona zostanie wartość `None`. Zmiana ta spowoduje jednak błąd w funkcji `main`:

```
[<EntryPoint>]
let main argv =
    wczytajOkrag ()
    ?|> wSrodku
    ?|> pair (wczytajPunkty "punkty.txt")
    ?|> (<||>) Seq.filter
    ?|> Seq.iter pokazPunkt
    //|> Result.mapError (fun e -> printfn "%s" e)
    |> ignore
    0
```

jest to spowodowane tym, że funkcja `pair` zwracała do tej pory wartość typu $(\text{float} * \text{float} \rightarrow \text{bool}) * \text{seq} < \text{float} * \text{float} >$, która idealnie pasowała do wymagań kolejnej funkcji `Seq.filter`. Teraz jednak zwracana wartość ma typ $(\text{float} * \text{float} \rightarrow \text{bool}) * \text{seq} < \text{float} * \text{float} > \text{option}$, który już nie pasuje do sygnatury `Seq.filter`.

Jednym z możliwych rozwiązań tego problemu jest modyfikacja funkcji `pair` do postaci:

```
let pair b a =
    match b with
    | Some y -> Some (a, y)
    | None -> None
```

Funkcja ta zakłada, że pierwszy parametr przyjmuje opcję, która jest dopasowywana za pomocą wyrażenia `match`. Operacja ta pozwala na osiągnięcie wartości o typie bardziej zbliżonym do naszych potrzeb $(\text{float} * \text{float} \rightarrow \text{bool}) * \text{seq} < \text{float} * \text{float} >$ `option`. Teraz cała para jest opcjonalna, a nie tylko jej drugi element. To jednak nie rozwiązuje w pełni naszego problemu ponieważ błąd dalej będzie występował w tej samej linii (tylko, że teraz będzie już inny):

Aby zrozumieć na czym ten błąd polega powinniśmy przeanalizować sygnatury funkcji wykorzystywanych we wcześniejszych liniach. Będzie to jednak prostsze jeżeli

```
[<EntryPoint>]
let main argv =
    wczytajOkrag ()
    ?|> wSrodku
    ?|> pair (wczytajPunkty "punkty.txt")
    ?|> (<||) Seq.filter
    ?|> Seq.iter pokazPunkt
    |> ignore
    0
```

zrezygnujemy z wykorzystania jakichkolwiek operatorów i wrócimy do rozwiniętej postaci (w tym przypadku błąd będzie podkreślany w odwołaniu do symbolu para):

```
[<EntryPoint>]
let main argv =
    let okrag = wczytajOkrag ()
    let predykat = Option.map wSrodku okrag
    let para = Option.map (pair (wczytajPunkty "punkty.txt")) ←
        predykat
    let punktyWSrodku = Option.map ((<||) Seq.filter) para
    Option.iter (Seq.iter pokazPunkt) punktyWSrodku
    0
```

Możemy teraz sprawdzić jaką postać mają poszczególne symbole pośrednie:

1. okrag: Okrag option
2. predykat: (float*float->bool)option
3. para: ((float*float -> bool)*seq<float*float>) option option
4. punktyWSrodku: seq<float*float> option

Zwróćmy uwagę zwłaszcza na symbol para. Zauważmy, że na jest to podwójna opcja (tak, to nie jest błąd). Skąd to się wzięło? Przyjrzyjmy się linii w które symbol ten jest definiowany. Częściowa aplikacja funkcji para jest przekazana do funkcji map, jako argument mapa i wewnątrz niej ostatecznie uruchomiona. Spójrzmy jeszcze raz definicję funkcji map, zwłaszcza na jej wyróżniony fragment:

```
let map mapa opcja =
    match opcja with
    | None -> None
    | Some x -> Some (mapa x)
```

Ponieważ funkcja pair zwraca wartość typu ('b*'a)option. Czyli może to być albo Some ('b*'a), albo None. Jeżeli teraz ten wynik prześlemy do konstruktora Some, to otrzymamy: Some (Some ('b*'a)), albo Some (None). W obu przypadkach będzie to opcja ukryta w kolejnej opcji.

Czyli całej tej sytuacji jest winna funkcja `map`, z której nie powinniśmy w tym przypadku korzystać. Bardzo do niej podobną, tylko bardziej odpowiednią do naszych potrzeb jest funkcja `Option.bind`, której definicja jest następująca:

```
let map mapa opcja =  
  match opcja with  
  | None -> None  
  | Some x -> mapa x
```

Zauważmy, że w tym przypadku, funkcja `Option.bind` zwraca wartość `None` jeżeli taką wartość otrzyma jako argument `opcja`. Jednak, gdy argument `opcja` ewaluuje się do wartości `Some x`, to zwraca rezultat funkcji `mapa`, już bez opakowywania w opcję.

Wykorzystując tą funkcję nasz program będzie wyglądał następująco:

```
[<EntryPoint>  
let main argv =  
  wczytajOkrag ()  
  ?|> wSrodku  
  |> Option.bind (pair (wczytajPunkty "punkty.txt"))  
  ?|> (<|)|) Seq.filter  
  ?|> Seq.iter pokazPunkt  
  |> ignore  
  0
```

Podobnie jak dla `Option.map` zdefiniowaliśmy operator `?|>`, również dla `Option.bind` możemy to zrobić:

```
let inline (>=>) wynik mapa = Option.bind mapa wynik
```

Kształt tego operatora nie jest wybrany przeze mnie przypadkowo. Identycznie operacja `bind` jest zdefiniowana w chyba najpopularniejszym języku funkcyjnym - Haskell-u.

W rezultacie pozwoli nam to zapisać naszą aplikację w następujący sposób:

```
open System  
open System.IO  
  
type Okrag = {  
  srodek: (float*float);  
  promien : float;  
}  
  
let zamienNaPunkt (ls:string[]) = (float ls.[0], float ls.[1])
```

```
let wczytajPunkty nazwaPliku =
    try
        nazwaPliku
        |> File.ReadLines
        |> Seq.map (fun l->l.Split(' '))
        |> Seq.map zamienNaPunkt
        |> Some
    with
        _ -> None

let wczytajWartosc komunikat =
    printf "%s" komunikat
    Console.ReadLine()

let wczytajOkrag () =
    let jestX, x = Double.TryParse
        (wczytajWartosc "Podaj wspolrzedna x: ")
    let jestY, y = Double.TryParse
        (wczytajWartosc "Podaj wspolrzedna y: ")
    let jestR, r = Double.TryParse
        (wczytajWartosc "Podaj promien: ")
    if jestX && jestY && jestR then
        Some {srodek = (x,y); promien = r}
    else
        None

let odleglosc (x1,y1) (x2,y2) =
    sqrt ((x1-x2)**2.0+(y1-y2)**2.0)

let wSrodku (okrag:Okrag) punkt =
    (odleglosc okrag.srodek punkt) < okrag.promien

let pokazPunkt (x,y) = printfn "%f %f" x y

let pair b a =
    match b with
    | Some y -> Some (a, y)
    | None -> None

let (?|>) wynik mapa = Option.map mapa wynik

let (>=) wynik mapa = Option.bind mapa wynik
```

```
[<EntryPoint>]
let main argv =
    wczytajOkrag ()
    ?|> wSrodku
    >>= pair (wczytajPunkty "punkty.txt")
    ?|> (<||) Seq.filter
    ?|> Seq.iter pokazPunkt
    |> ignore
    0
```

Dopisać
nie opcji

5.5.1 Funkcje do obsługi opcji

1. **Option.bind binder opcja**

sygnatura: 'a->'b option->'a option->'b option

opis: Funkcja przyjmuje opcję, odczytuje wartość typu 'a i transformuje ją do opcji zawierającej wartość typu 'b.

2. **Option.contains wartość opcja**

sygnatura: 'a->'a option->bool

opis: Zwraca `true`, gdy podana opcja jest `Some x` i `x = wartość`. W przeciwnym przypadku zwraca `false`

3. **Option.count opcje**

sygnatura: 'a option->int

opis: Zwraca 0 gdy opcja jest `None` i 1 w przeciwnym przypadku

4. **Option.defaultValue wartość opcja**

sygnatura: 'a->'a option->'a

opis: Jeżeli opcja jest `Some x`, to zwraca `x`, w przeciwnym przypadku zwraca wartość

5. **Option.defaultWith generator opcja**

sygnatura: unit->'a->'a option->'a

opis: Jeżeli opcja jest `Some x`, to zwraca `x`, w przeciwnym przypadku zwraca wynik funkcji generator

6. Option.exists predykat opcja

sygnatura: 'a->bool->'a option->bool

opis: Jeżeli opcja jest Some x, to sprawdza czy x spełnia warunek podany jako predykat, w przeciwnym przypadku zwraca **false**

7. Option.filter predykat opcja

sygnatura: 'a->bool->'a option->'a option

opis: Jeżeli opcja jest Some x, to zwraca Some x gdy x spełnia warunek podany jako predykat. W każdym przeciwnym przypadku zwraca wartość

8. Option.flatten opcja

sygnatura: 'a option option->'a option

opis: Funkcja spłaszcza hierarchię opcji. Jeżeli opcja to None, to funkcja zwraca None. Jeżeli jednak opcja to Some(x), to funkcja zwraca x, przy czym x musi być opcją

9. Option.fold funkcja stan opcja

sygnatura: 'stan->'a->'stan->'stan->'a option->'stan

opis: Funkcja wykonuje operację fold na opcji. Jeżeli opcja to None, to zwraca stan początkowy podany jako stan

10. Option.foldBack funkcja opcja stan

sygnatura: 'a->'stan->'stan->'a option->'stan->'stan

opis: Funkcja wykonuje operację foldBack na opcji. Jeżeli opcja to None, to zwraca stan początkowy podany jako stan

11. Option.forall predykat opcja

sygnatura: 'a->bool->'a option->bool

opis: Funkcja działa podobnie jak exists, ale zwraca **true**, gdy opcja jest None

12. Option.get opcja

sygnatura: 'a option->'a

opis: Zwraca wartość skojarzoną z opcją lub rzuca wyjątek, jeżeli opcja jest None

13. Option.isNone opcja

sygnatura: 'a option->bool

opis: Zwraca **true**, gdy opcja jest None i **false** w przeciwnym przypadku

14. Option.isSome opcja

sygnatura: 'a option->bool

opis: Zwraca **false**, gdy opcja jest None i **true** w przeciwnym przypadku

15. Option.iter akcja opcja

sygnatura: 'a->()->'a option->()

opis: Wykonuje akcję na opcji, jeżeli ta jest Some x. Jeżeli opcja jest None nie robi nic

16. Option.map mapa opcja

sygnatura: 'a->'b->'a option->'b option

opis: Transformuje opcję typu 'a w opcję typu 'b za pomocą funkcji mapa

17. Option.map2 mapa opcja1 opcja2

sygnatura: 'a->'b->'c->'a option->'b option->'c option

opis: Aplikuje funkcję mapa do wartości skojarzonych z opcjami, jeżeli żadna z nich nie jest None. Zwraca również opcję.

18. Option.map3 mapa opcja1 opcja2 opcja3

sygnatura: 'a->'b->'c->'d->'a option->'b option->'c option->'d option

opis: Aplikuje funkcję mapa do wartości skojarzonych z opcjami, jeżeli żadna z nich nie jest None. Zwraca również opcję.

19. Option.ofNullable

sygnatura: Nullable<'a>->'a option

opis: Tworzy opcję na podstawie wartości typu Nullable

20. Option.ofObj

sygnatura: 'a->'a option

opis: Tworzy opcję na podstawie obiektu

21. Option.orElse jeśliNie opcja

sygnatura: 'a opcja->'a opcja->'a opcja

opis: Zwraca opcję, jeżeli ta jest Some, przeciwnym przypadku zwraca opcję jeśliNie

22. Option.orElseWith jeśliNieGenerator opcja

sygnatura: unit->'a option->'a option->'a option

opis: Zwraca opcję, jeżeli ta jest Some, przeciwnym przypadku zwraca opcję będącą rezultatem funkcji jeśliNieGenerator

23. Option.toArray opcja

sygnatura: 'a option->'a[]

opis: Jeżeli opcja jest Some, to zwraca tablicę jednoelementową, w przeciwnym przypadku zwraca tablicę pustą.

24. Option.toList opcja

sygnatura: 'a option->'a list

opis: Jeżeli opcja jest Some, to zwraca listę jednoelementową, w przeciwnym przypadku zwraca listę pustą.

25. Option.toNullable opcja

sygnatura: 'a option->Nullable<'a>

opis: Konwertuje opcję do wartości typu Nullable

26. Option.toObj opcja

sygnatura: 'a option -> 'a

opis: Jeżeli opcja jest Some x, to zwraca x, w przeciwnym przypadku zwraca null

5.6 Wyniki

Opcje, choć bardzo użyteczne do określania wartości opcjonalnych, czyli takich, które mogą, ale nie muszą istnieć, nie specjalnie nadają się do modelowania sytuacji wyjątkowych. Jest to spowodowane oczywiście tym, że None mówi nam, że wartość nie istnieje, ale nie podaje powodu dla którego tak się dzieje.

Do obsługi błędów zdecydowanie lepszym wyborem będzie typ Result:

```
type Result<'T, 'E>
| Ok of 'T
| Error of 'E
```

Jak widzimy jego definicja jest bardzo zbliżona do Option, ale oba przypadki tej wartości mogą teraz przenosić wartość. Ok oznacza, że operacja zakończyła się poprawnie, a Error, że wystąpił jakiś błąd.

W naszym programie typ ten możemy wykorzystać w następujący sposób:

```
let wczytajPunkty nazwaPliku =
    try
        nazwaPliku
        |> File.ReadLines
        |> Seq.map (fun (l:string)->l.Split(' '))
        |> Seq.map zamienNaPunkt
        |> Ok
    with
        (e:exn) -> Error e.Message

let wczytajOkrag () =
    let jestX, x = Double.TryParse
        (wczytajWartosc "Podaj wspolrzedna x: ")
    let jestY, y = Double.TryParse
        (wczytajWartosc "Podaj wspolrzedna y: ")
    let jestR, r = Double.TryParse
        (wczytajWartosc "Podaj promien: ")
    if jestX && jestY && jestR then
        Ok {srodek = (x,y); promien = r}
    else
        Error "Blednie podana wartosc"
```

```

let pair b a =
  match b with
  | Ok y -> Ok (a, y)
  | Error e -> Error e

let (?|>) wynik mapa = Result.map mapa wynik

let (>=) wynik mapa = Result.bind mapa wynik

[<EntryPoint>]
let main argv =
  wczytajOkrag ()
  ?|> wSrodku
  >= pair (wczytajPunkty "punkty.txt")
  ?|> (<||) Seq.filter
  ?|> Seq.iter pokazPunkt
  |> Result.mapError (printfn "%s")
  |> ignore
0

```

Ponieważ liczba funkcji zawartych w module `Result` jest ograniczona do niezbędnego minimum wiele z nich musimy zaimplementować samodzielnie. Przykładowo moglibyśmy w naszym programie stworzyć funkcję `wyswietl`:

```

let wyswietl akcjaOk akcjaError = function
| Ok x -> akcjaOk x
| Error e -> akcjaError e

```

co pozwoli nam przekształcić funkcję `main` do postaci:

```

[<EntryPoint>]
let main argv =
  wczytajOkrag ()
  ?|> wSrodku
  >= pair (wczytajPunkty "punkty.txt")
  ?|> (<||) Seq.filter
  |> wyswietl (Seq.iter pokazPunkt) (printfn "%s")
0

```

5.7 Zadania

Zadanie 5.1 Napisz program z zadania ?? wykorzystując operator potokowy.

Zadanie 5.2 Napisz program z zadania ?? wykorzystując operator potokowy.

Zadanie 5.3 Napisz program, który wczyta od użytkownika z klawiatury liczbę całkowitą. Wartość ta może być opcjonalna (wtedy użytkownik powinien podać łańcuch pusty lub wprowadzić same białe znaki). Następnie podaną wartość należy podnieść do kwadratu i wyświetlić na ekranie lub wyświetlić komunikat: "Nie podałeś wartości, to ja nie podam wyniku". Do rozwiązania zadania wykorzystaj typ Option i wyrażenie dopasowania wzorca `match ... with`

Zadanie 5.4 Napisz program, który wczyta od użytkownika z klawiatury liczbę całkowitą. Wartość ta może być opcjonalna (wtedy użytkownik powinien podać łańcuch pusty lub wprowadzić same białe znaki). Następnie podaną wartość należy podnieść do kwadratu i wyświetlić na ekranie lub wyświetlić komunikat: "Nie podałeś wartości, to ja nie podam wyniku". Do rozwiązania zadania wykorzystaj typ Option i funkcje z modułu Option.

Zadanie 5.5 Napisz program, który wczyta od użytkownika z klawiatury liczbę całkowitą. Wartość ta może być opcjonalna (wtedy użytkownik powinien podać łańcuch pusty lub wprowadzić same białe znaki). Następnie podaną wartość należy podnieść do kwadratu i wyświetlić na ekranie. Jeżeli użytkownik nie podał liczby przyjmij domyślną wartość 100. Do rozwiązania zadania wykorzystaj typ Option i wyrażenie dopasowania wzorca `match ... with`

Zadanie 5.6 Napisz program, który wczyta od użytkownika z klawiatury wartość całkowitą. Wartość ta jest obowiązkowa. Uwzględnij fakt, że może ona jednak być błędnie wprowadzona. Następnie podaną wartość należy podnieść do kwadratu i wyświetlić na ekranie lub wyświetlić komunikat informujący o błędzie. Do rozwiązania zadania wykorzystaj typ Result.

Zadanie 5.7 Napisz program, który wczyta od użytkownika z klawiatury wartość całkowitą. Wartość ta jest obowiązkowa. Jeżeli użytkownik nie poda wartości lub poda ją błędną, to wyświetl komunikat o błędzie, a następnie poproś go o ponowne wprowadzenie tej wartości. Następnie podaną wartość należy podnieść do kwadratu i wyświetlić na ekranie.

Zadanie 5.8 Napisz program, który wczyta od użytkownika z klawiatury wartość całkowitą. Wartość ta jest obowiązkowa. Jeżeli użytkownik nie poda wartości lub poda ją błędną, to wyświetl komunikat o błędzie, a następnie poproś go o ponowne wprowadzenie tej wartości. Użytkownik może ponownie wprowadzać wartość maksymalnie 4 razy. Jeżeli uzyskamy poprawną wartość, to należy podnieść do ją do kwadratu i wyświetlić na ekranie, jeżeli nie to wyświetlić komunikat: "No ile razy można się mylić".

Zadanie 5.9 Napisz program, który wczyta od użytkownika z klawiatury liczbę całkowitą. Wartość ta może być opcjonalna (wtedy użytkownik powinien podać łańcuch pusty lub wprowadzić same białe znaki). Uwzględnij fakt, że może ona jednak być błędnie wprowadzona. Następnie podaną wartość należy podnieść do kwadratu i wyświetlić na ekranie lub wyświetlić odpowiednie komunikaty:

1. Jeżeli użytkownik nie podał wartości to: "Nie podałeś wartości, to ja nie podam wyniku".

2. Jeżeli użytkownik podał wartość ale błędną, to wyświetl komunikat o błędzie. Do rozwiązania zadania wykorzystaj zarówno typ Result jak i typ Option.

Zadanie 5.10 Wczytaj od użytkownika dwie wartości całkowite. Jeżeli użytkownik poda je poprawnie, dodaj je i wynik sumy wyświetl na ekranie. W przeciwnym przypadku wyświetl komunikat o błędzie.

Zadanie 5.11 Wczytaj od użytkownika dwie wartości całkowite. Jeżeli użytkownik poda poprawnie pierwszą wartość, to wczytaj drugą. Jeżeli obie wartości są poprawne, to dodaj je i wynik sumy wyświetl na ekranie. W przeciwnym przypadku wyświetl komunikat o błędzie, która z tych wartości była wprowadzona w sposób niepoprawny. Wykorzystaj typ Result i funkcję bind.

Zadanie 5.12 Wczytaj od użytkownika dwie wartości całkowite. Jeżeli użytkownik poda poprawnie pierwszą wartość, to wczytaj drugą. Jeżeli obie wartości są poprawne, to dodaj je i wynik sumy wyświetl na ekranie. W przeciwnym przypadku wyświetl komunikat o błędzie, która z tych wartości była wprowadzona w sposób niepoprawny i poproś o ponowne wprowadzenie obu wartości.

Zadanie 5.13 Rozszerz program z zadania 5.1. Nazwę pliku niech użytkownik poda z klawiatury. Jeżeli uda się go wczytać poprawnie, to wyświetl wyniki tak jak poprzednio. Jeżeli nie to podaj odpowiedni komunikat z informacją o błędzie.

Zadanie 5.14 Rozszerz program z zadania 5.13. Dodatkowo poproś użytkownika ile wierszy z danymi chce wczytać z pliku. Jeżeli użytkownik nie poda wartości, to wczytaj wszystkie. Podobnie, gdy użytkownik poda większą wartość niż jest wierszy w pliku.