Liczbowe funkcje rekurencyjne wielu zmiennych – uczymy się na błędach

Artykuł prezentuje implementację funkcji rekurencyjnych wielu zmiennych w języku F# oraz sposób wyświetlania wartości tych funkcji. W artykule zaprezentowane również zostanie podejście, które pomoże zrozumieć, w jaki można uczyć się tworzenia funkcji rekurencyjnych wielu zmiennych oraz jakich błędów należy przy tym unikać. Funkcje rekurencyjne są ważne w wytwarzaniu oprogramowania, ponieważ istnieją takie problemy, które o wiele łatwiej można zaimplementować rekurencyjnie niż w sposób jawny. Dzięki niej łatwo jest wykonać wiele zadań, w których potrzeba jest wyników cząstkowych do obliczenia całości. Istnieją problemy, które o wiele prościej można zapisać rekurencyjnie niż w sposób jawny. Rekurencja jest trudna, więc i jej testowanie nie jest proste. W artykule zostaną omówione funkcje rekurencyjne liczbowo-liczbowe, zaś język F# posłuży tylko do implementowania tych funkcji. Pokazany zostanie również sposób, w jaki można wykrywać defekty funkcji rekurencyjnych.

1. Wstęp

Języki programowania takie jak Java, C++, Python są językami, które wykonują sekwencje zadań zdefiniowane jawnie przez programistę. Język F# jest językiem funkcyjnym. Programowanie funkcyjne działa inaczej niż programowanie sekwencyjne, ponieważ aplikacje tego typu nie wykonują zadań sekwencyjnie, tylko wyznaczają jedynie wartości poszczególnych wyrażeń. Programy funkcyjne składają się głównie z funkcji, które są ich podstawowymi elementami. Główna funkcja składa się tylko i wyłącznie z innych funkcji, które z kolei składają się z jeszcze innych funkcji, a ta cecha daje możliwość tworzenia funkcji złożonych. Takie podejście jest czysto matematyczne – funkcje przyjmują pewną liczbę zmiennych i zwracają wynik (stąd pomysł na wykorzystanie języka F# w artykule).

Analogicznie jak w innych językach programowania, w F# można pisać programy przepisując po prostu założenia. Jednak nie wszystkie programy funkcyjne mogą powstać tylko za pomocą przepisania założeń. Takie podejście, w którym założenia przepisywane są w postaci kodu, nie jest efektywne w dużej liczbie przypadków. Efektywny program może być trudniejszy do napisania i nie zawsze jest tak przejrzysty jak jego nieefektywny odpowiednik. W przypadku programowania funkcyjnego nie powstają takie problemy jak kolejność wykonywania instrukcji, zmiana wartości zmiennej w nieoczekiwany sposób, co poprawia czytelność programów, ułatwia ich weryfikację i jest pewnego rodzaju ułatwieniem [1]. Projektowanie funkcji rekurencyjnych wielu zmiennych nie jest jednak rzeczą trywialną. Łatwo można popełnić błąd podczas ich implementacji, co zostanie pokazane w artykule.

2. Funkcje rekurencyjne z jedną zmienną

W tym rozdziale zostaną zaprezentowane funkcje rekurencyjne jednej zmiennej w języku F# i ich wybrane wartości.

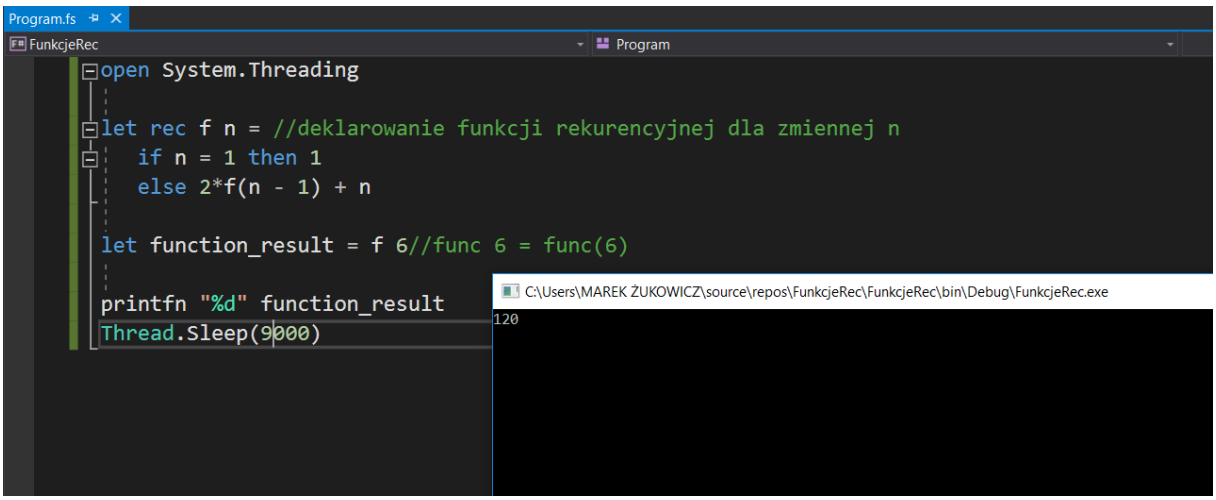
Przykład 1. Załóżmy, że chcemy obliczyć wartość funkcji f opisanej rekurencyjnie dla wartości 6:

$$f(n) = \begin{cases} 1, & n = 1 \\ 2f(n-1) + n, & n > 1 \end{cases}$$

W języku F# implementacja takiej funkcji będzie miała postać:

```
open System.Threading

let rec f n = //deklarowanie funkcji rekurencyjnej dla zmiennej n
   if n = 1 then 1
   else 2*f(n - 1) + n

let function_result = f 6//func 6 = func(6)

printfn "%d" function_result
Thread.Sleep(3000)
```

Wartość funkcji f dla liczby 6 jest równa 120, co widać po wywołaniu powyższego kodu na konsoli:

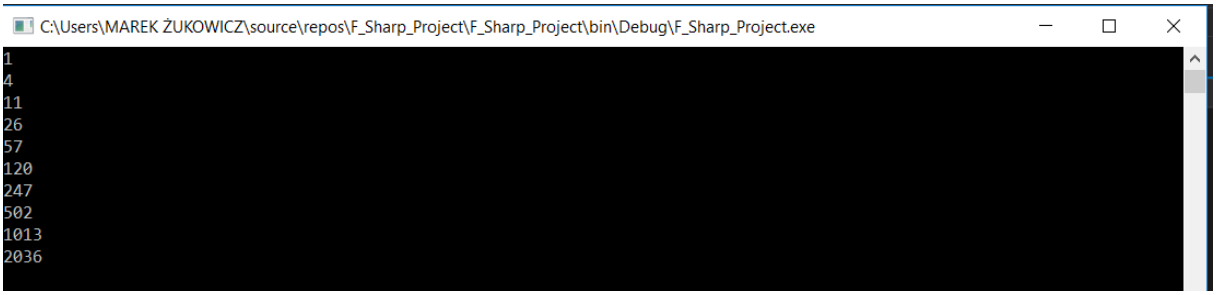


Rysunek 1: Wynik działania programu funkcji f dla wartości 6

W języku F# **wcięcia są istotne**, analogicznie jak w języku Python – traktujemy je jak ciało metody (np. {} w języku C#). Jeżeli chcemy otrzymać wartości funkcji f dla kilku liczb, to można to zrobić za pomocą pętli `for`:

```
let rec  function_f n =
   if n = 1 then 1
   else 2*function_f(n - 1) + n

let function_result = 0

let list = [1;2;3;4;5;6;7;8;9;10] //Deklarowanie listy argumentów.
for i in list do //Deklarowanie pętli for
    printfn "%d" function_f(i) //wyświetlanie wartości na konsoli

Thread.Sleep(9000)
```

Po wywołaniu wyżej przedstawionego kodu otrzymamy kolejno liczby 1, 4, 11, 26, 57, 120, 247, 502, 1013, 2036, co również zostało wyświetlone na konsoli:



Rysunek 1: Wynik działania programu funkcji f dla tablicy argumentów tab

W języku F# przetwarzanie danych można osiągnąć poprzez użycie pipeline'ów [3]:

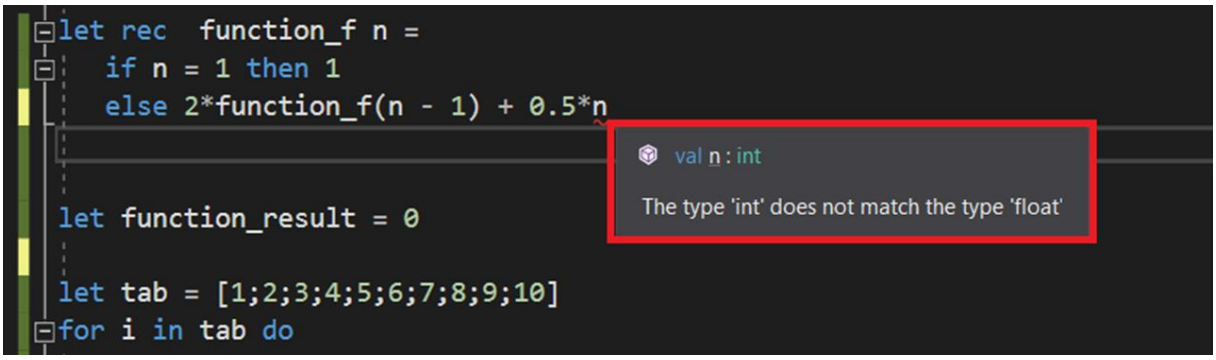
```
tab |> List.map f |> List.iter (printfn "%d" )
```

Język F# na bieżąco sprawdza, czy funkcja będzie mogła osiągnąć niedozwolone wartości ze względu na jej charakter.

Przykład 2. Jeśli zmienimy funkcję f w następujący sposób:

$$f(n) = \begin{cases} 1, & n = 1 \\ 2f(n-1) + \boldsymbol{\frac{n}{2}}, & n > 1 \end{cases} \quad (*)$$

to matematycznie, jest to niepoprawny zapis, ponieważ wartościami funkcji rekurencyjnej f mogą być liczby naturalne. Wyrażenie $2f(n-1) + \frac{\boldsymbol{n}}{\boldsymbol{2}}$ spowoduje, że co druga wartość będzie ułamkiem. Język F# wyłapie taki błąd, od razu bez skompilowania programu, co widać na obrazku:



Rysunek 2: Walidacja typów typu wartości funkcji f

Jest to bardzo cenna uwaga. Nie możemy mnożyć funkcji z pomniejszonym argumentem przez ułamki oraz nie możemy dodać do niej ułamka. Z jakościowego punktu widzenia obsługa wyjątków jest konieczna – użytkownik, który wprowadził niepoprawne dane będzie wiedział, że błąd jest po jego stronie a nie po stronie programu.

3. Funkcje rekurencyjne wielu zmiennych

Funkcje rekurencyjne mogą być zdefiniowane za pomocą kilku parametrów. Jednak zdefiniowanie funkcji rekurencyjnej wielu zmiennych nie jest zagadnieniem trywialnym i wymaga

doświadczenia. Jeżeli chcemy zaimplementować funkcję rekurencyjną wielu zmiennych, to należy ją tak zdefiniować, aby zawsze istniała dla niej możliwość osiągnięcia warunku początkowego, do którego może się cofnąć. Jest to niezbędny krok podczas obliczania wartości funkcji rekurencyjnych. Język F# jest bardzo dobrym narzędziem, które pozwoli zrozumieć sposób definiowania takich funkcji. W przykładzie 2 pokazany został błąd dla przeciwdziedziny funkcji (zbioru wartości), o którym poinformowało nas środowisko do programowania.

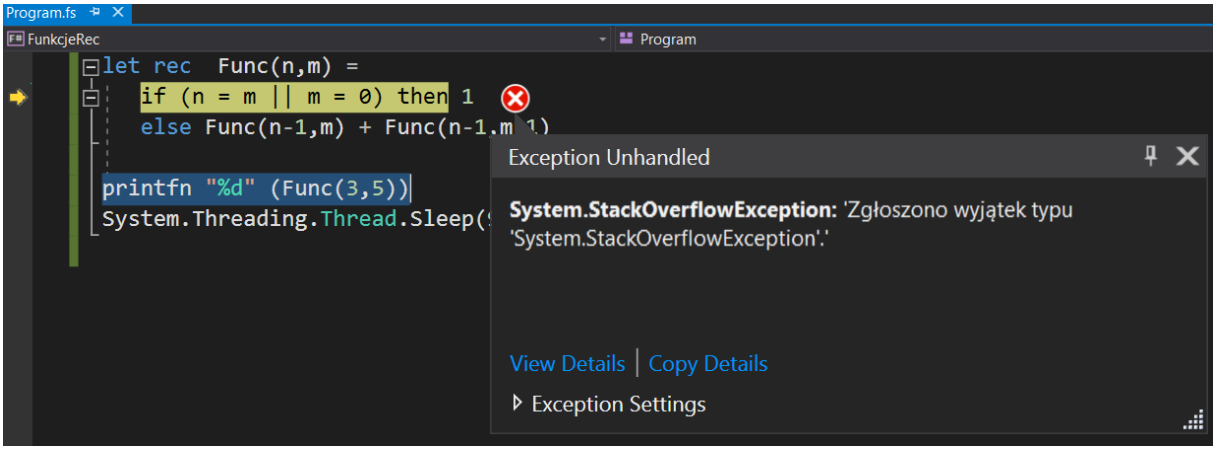
Przykład 3. Załóżmy, że mamy zdefiniowaną następującą funkcję:

$$f(n,m) = \begin{cases} 1, & n = m \vee m = 0 \\ f(n-1,m) + f(n-1,m-1), & n > m \end{cases}$$

Funkcja narzuca warunek, że $n \geq m$. Ponieważ jest to funkcja rekurencyjna, to $m, n \in N$. Łatwo sprawdzić, co się stanie, jeśli funkcja nie będzie zawierała takiego warunku dla pary liczb $(3,5)$:

```
let rec  Func(n,m) =
   if (n = m || m = 0) then 1
   else Func(n-1,m) + Func(n-1,m-1)

printfn "%d" (Func(3,5)
Thread.Sleep(9000)
```

Po uruchomieniu VS dostaniemy komunikat: **System.StackOverflowException:** 'Exception of type 'System.StackOverflowException' was thrown.' co widać na obrazku:



Rysunek 3: Przykład wyjątku funkcji Func dla argumentu (3,5)

Wobec tego taki przypadek musi zostać uwzględniony w definicji funkcji. Można to zrobić w następujący sposób [2]:

Przykład 4.

```
let rec Func(n,m) =
 if n < m then
     raise (System.Exception "n < m") // funkcja, która zwraca wyjątek
     elif (n = m || m = 0) then 1
     else Func(n-1,m) + Func(n-1,m-1)
```

```
let Func_result = Func(5,3)
System.Console.WriteLine(Func_result)
System.Threading.Thread.Sleep(5000))
```

albo w następujący sposób:

Przykład 5.

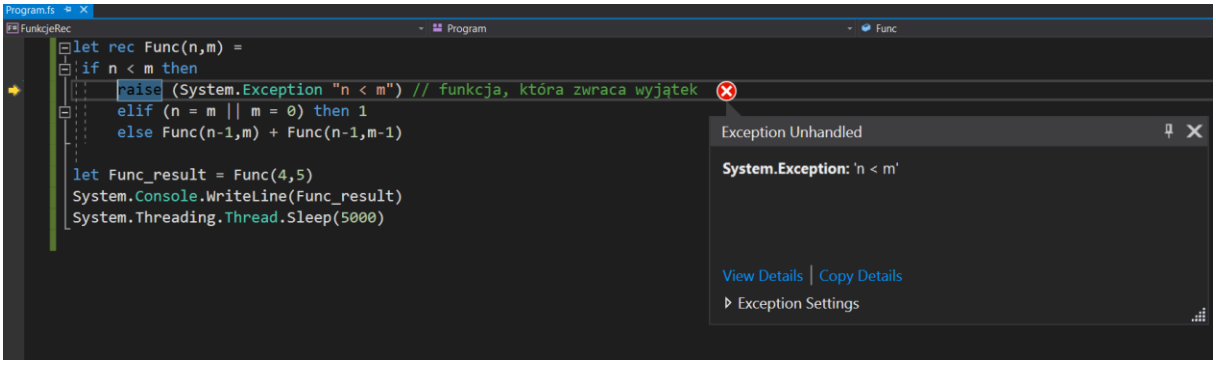
```
let rec Func(n,m) =
    if n < m then
    failwith "Niepoprawne argumenty funkcji" // funkcja, która zwraca wyjątek
    else if (n = m || m = 0) then 1
    else Func(n-1,m) + Func(n-1,m-1)

let Func_result = Func(5,3)
System.Console.WriteLine(Func_result)

System.Threading.Thread.Sleep(5000)
```

W języku F# istnieje możliwość rozwiązania problemu z niepoprawnymi argumentami za pomocą użycia bloku `try … with`, jednak ten blok nie zawsze zadziała, ponieważ nie każdy rodzaj wyjątku może zostać przechwycony. Kod programu mógłby wyglądać następująco:

Przykład 6.

```
let rec  Func(n,m) =
    try
        if (n = m || m = 0) then 1
        else Func(n-1,m) + Func(n-1,m-1)
    with | :? System.SystemException -> printfn "Niepoprawne argumenty"; 0
    // klauzula, która rzuca wyjątek, jednak wymaga podania wartości, którą funkcja ma
    // zwrócić w przypadku wyjątku, co nie jest najlepszym rozwiązaniem.

let Func_result = Func(5,3)
System.Console.WriteLine(Func_result)

System.Threading.Thread.Sleep(5000)
```

Jednak w tym przypadku blok `try … with` zostanie pominięty, ponieważ wyjątek typu **System.StackOverflowException** to jest jeden z dwóch rodzajów wyjątków, których nie można przechwycić. Taki wyjątek powstanie np. dla argumentu (4,5). Wykorzystując funkcję przedstawioną w przykładzie 4 otrzymamy taki komunikat w przypadku podania niepoprawnych argumentów:



Rysunek 4: Przykład użycia klauzuli raise

Widzimy, że został zwrócony wyjątek $n < m$. Chcąc wyświetlić wartości naszej funkcji dla zbioru argumentów, możemy użyć tabeli i pętli `for`, tak jak niżej:

```
let rec Func(n,m) =
 if n < m then
     raise (System.Exception "n < m")
     else if (n = m || m = 0) then 1
     else Func(n-1,m) + Func(n-1,m-1)

let list =
[(5,5);(5,4);(5,3);(5,2);(5,1);(5,0);(6,6);(6,5);(6,4);(6,3);(6,2);(6,1);(6,0)]

for i in list do printfn "%d" (Func(i))
System.Threading.Thread.Sleep(5000)
```

Konsola wyświetli takie wyniki:

Rysunek 5: Zbiór wartości funkcji Func dla tablicy tab

Wyniki naszej rekurencyjnej funkcji są, jak widać, symetryczne. Dzieje się tak, ponieważ nasz „tajemniczy” rekurencyjny wzór to **dwumian Newtona:**

$$\binom{n}{m},$$

dla którego ma miejsce własność:

$$\binom{n}{n-m} = \binom{n}{m}$$

Język F# pozwala uczyć się metodą prób i błędów, a przez to wyciągać wnioski na przyszłość (analogicznie jak inne języki programowania). Spróbujemy, w celu przeprowadzania doświadczenia, zmodyfikować wzór funkcji:

$$f(n,m) = \begin{cases} 1, & n = m \vee m = 0 \\ f(n-1,m) + f(n-1,m-1), & n > m \end{cases}$$

Nowy wzór będzie miał postać:

$$g(n,m) = \begin{cases} 1, & n = m \vee m = 0 \\ 2g(n-1,m) + 2g(n-1,m-1), & n > m \end{cases}$$

Dla argumentów `{(5,5);(5,4);(5,3);(5,2);(5,1);(5,0);(6,6);(6,5);(6,4);(6,3);(6,2);(6,1);(6,0)}`, funkcja g zwróci wartości:

Rysunek 6: Zbiór wartości funkcji g dla tablicy tab

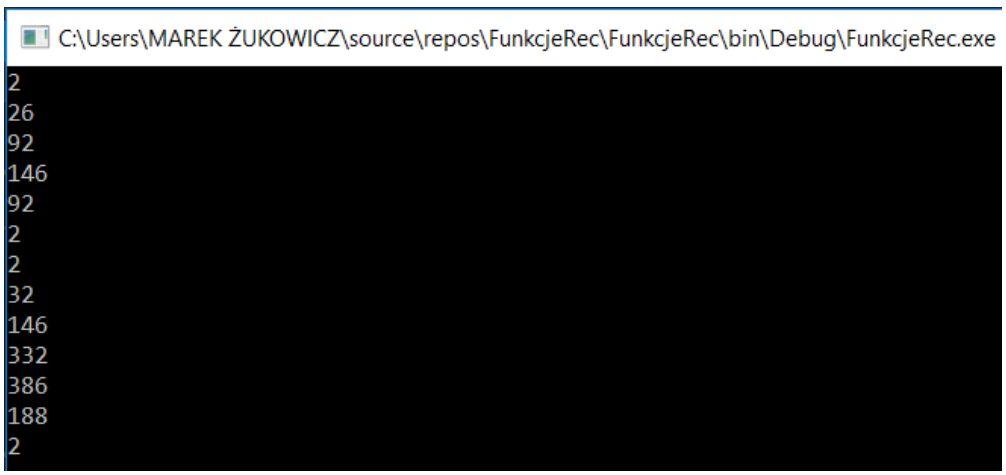
Funkcję f można zmodyfikować na wiele sposobów. Znacznie ciekawszy jest następny przykład funkcji:

$$h(n,m) = \begin{cases} 2, & n = m \vee m = 0 \\ 2h(n-1,m) + h(n-1,m-1) + 2, & n > m \end{cases}$$

dla której kod programu może być następujący:

```
let rec Func(n,m) =
   if n < m then
      raise (System.Exception "n < m")
      else if (n = m || m = 0) then 2
      else 2*Func(n-1,m) + Func(n-1,m-1) + 2
```

Dla tablicy argumentów takich, jak w poprzedniej funkcji, zwracane są następujące wartości:



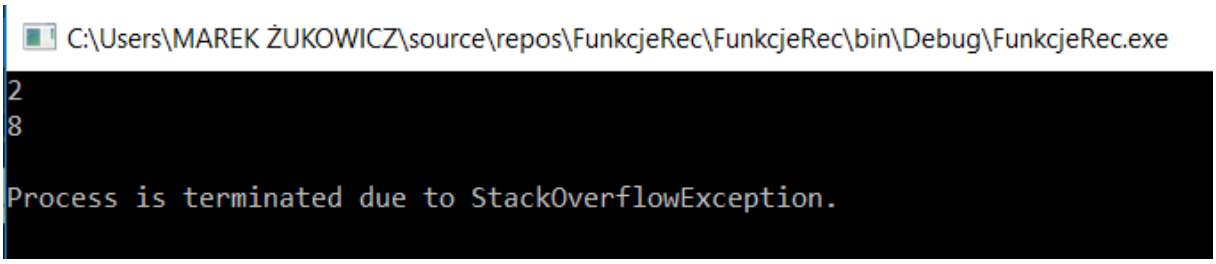
Rysunek 7: Zbiór wartości funkcji h dla tablicy argumentów tab

Z modyfikowaniem funkcji należy jednak postępować ostrożnie. Jedna mała zmiana może spowodować, że funkcja jest niepoprawna, np.:

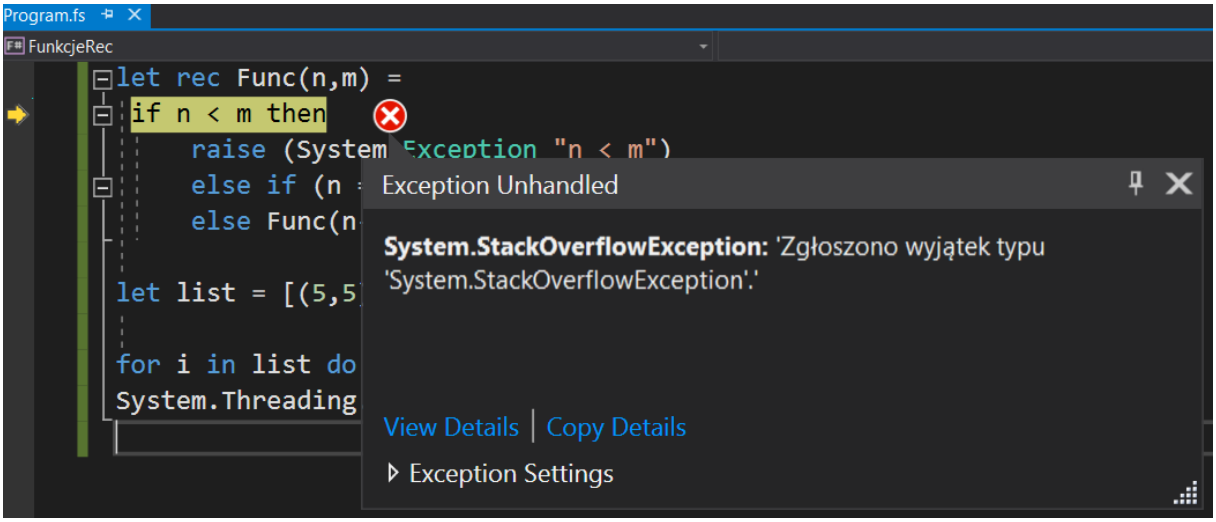
```
let rec Func(n,m) =
 if n < m then
     raise (System.Exception "n < m")
     else if (n = m || m = 0) then 2
```

```
     else Func(n-1,m) + Func(n-1, m-2) // zmiana na m-2

let list = [(5,5);(5,4);(5,3);(5,2);(5,1);(5,0)];

for i in list do printfn "%d" (Func(i));
System.Threading.Thread.Sleep(5000);
```

Tak zdefiniowana funkcja zwróci wyjątek dla argumentu (5,3):



Rysunek 8: Przykład wyjątku wyżej opisanej funkcji



Aby dowiedzieć się co poszło źle, należy wykonać analizę wywołania funkcji dla argumentu (5,3). Niech funkcja $j(n,m)$ będzie dana wzorem:

$$j(n,m) = \begin{cases} 2, & n = m \vee m = 0 \\ j(n-1,m) + j(n-1,m-2), & n > m \end{cases}$$

zatem:

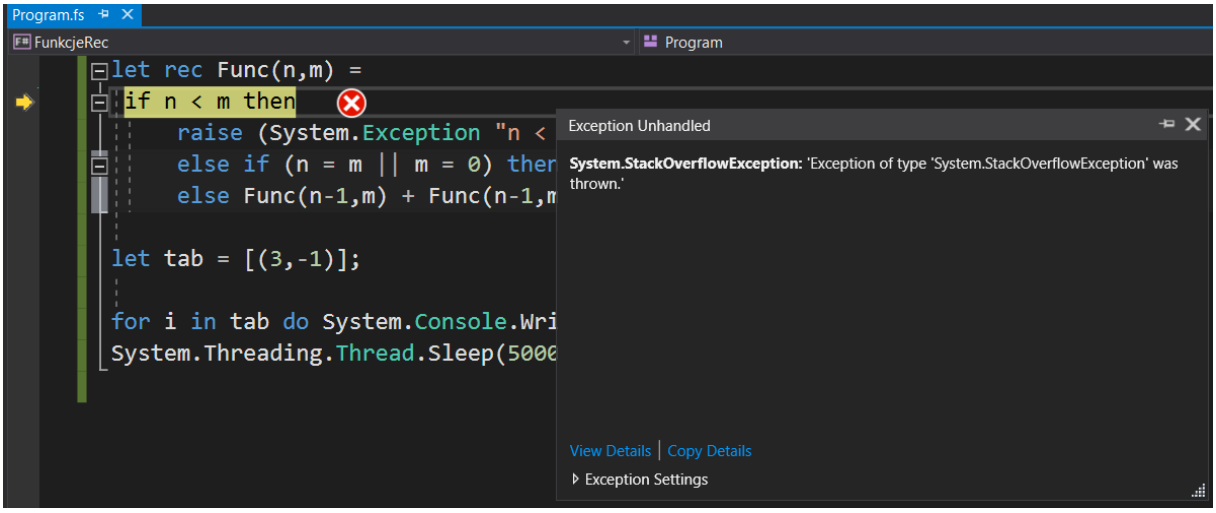
$$j(5,3) = j(4,3) + j(4,1) = j(3,3) + j(3,1) + j(3,1) + \boldsymbol{j(3,-1)}$$

Spróbujemy teraz obliczyć wartość funkcji $j(3,-1)$, czyli uruchomić funkcję:

```
let rec Func(n,m) =
 if n < m then
     raise (System.Exception "n < m")
     else if (n = m || m = 0) then 2
```

```
     else Func(n-1,m) + Func(n-1,m-2)

let list = [(2,-1)];

for i in list do printfn "%d" (Func(i));
System.Threading.Thread.Sleep(5000);
```

Po uruchomieniu od razu otrzymujemy wyjątek:

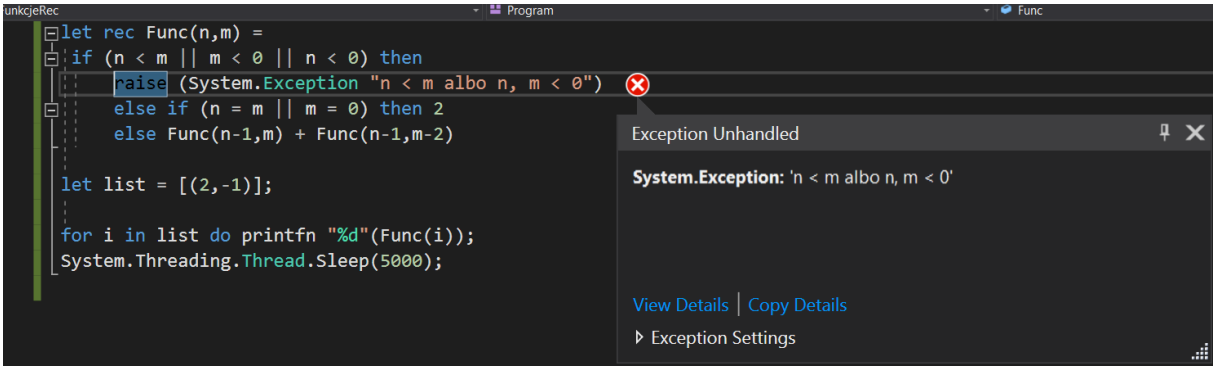


Rysunek 9: Przykład wyjątku funkcji j dla argumentu (3,-1)

Nasuwa się wniosek, że funkcja, którą wywołujemy będzie działać tylko dla liczb nieujemnych. Wobec tego wprowadzimy dodatkową walidację w kodzie, a mianowicie:

```
let rec Func(n,m) =
 if (n < m || m < 0 || n < 0) then
     raise (System.Exception "n < m albo n, m < 0")
     else if (n = m || m = 0) then 2
     else Func(n-1,m) + Func(n-1,m-2)

let list = [(2,-1)];

for i in list do printfn "%d"(Func(i));
System.Threading.Thread.Sleep(5000);
```

Po wywołaniu otrzymujemy wyjątek z treścią:



Rysunek 10: Zwrócenie wyjątku przez zmodyfikowaną funckję j

Teraz już mamy zabezpieczenia przed niepoprawnymi argumentami dla naszej funkcji.

Przykład 7. Załóżmy, że chcemy sprawdzić, czy poprawna jest definicja funkcji rekurencyjnej dwóch zmiennych postaci:

$$f(n,m) = \begin{cases} 1, & n * m = 4 \\ 2 + f(n-1, m-1), & n \geq m, n * m \neq 4 \\ 2 + f(n, m-1), & n < m, n * m \neq 4 \end{cases}$$

W języku F# taka funkcja będzie miała postać:

```
let rec Func(n,m) =
    if (n*m = 4) then 1
    else
        if (n >= m ) then 2 + Func(n-1,m-1)
        else 2 + Func(n,m - 1)

let tab = [(2,1)]
for i in tab do System.Console.WriteLine(Func(i))
System.Threading.Thread.Sleep(5000)
```

Aby dowiedzieć się, czy taka funkcja jest poprawnie zaimplementowana, możemy wykonać analizę klas równoważności, znaną w testowaniu oprogramowania. Funkcje rekurencyjne działają na liczbach całkowitych, co ułatwi naszą analizę. Klasy równoważności podzielone będą na zbiory, które utworzone zostaną na podstawie warunków funkcji $f(n,m)$:

a) $n * m = 4$, czyli będą to np. pary $\{(1,4),(2,2),(4,1),(-1,-4),(-2,-2),(-4,-1)\}$,
b) $n \geq m \wedge n * m \neq 4$, np. $\{(1,1),(2,1)\}$,
c) $n < m \wedge n * m \neq 4$, np. $\{(1,2),(2,3)\}$.

Wywołanie funkcji f dla klasy z podpunktu a) daje wynik 1. Próba obliczenia wartości funkcji f dla danych wejściowych z podpunktu b) np. $(2,1)$ kończy się wyjątkiem **System.StackOverflowException:** 'Exception of type 'System.StackOverflowException' was thrown.' Analizując błąd otrzymujemy:

$$f(2,1) = 2 + f(1,0) = 2 + 2 + f(0,-1) = 2 + 2 + 2 + f(-1,-2) = \cdots$$

W taki sposób funkcja nigdy nie osiągnie warunku początkowego, zatem jej przepis jest niepoprawny. Warunek początkowy musi być tak zdefiniowany, aby był możliwy do osiągnięcia dla każdych danych wejściowych – również przy funkcjach rekurencyjnych, które nie działają na liczbach.

Przykład 8. Rozważmy taką funkcję:

$$f(n,m) = \begin{cases} 1, & n * m = 4 \\ 2 + f(n-1,m), & n \geq m, n * m \neq 4 \\ 2 + f(n,m-1), & n < m, n * m \neq 4 \end{cases}$$

Zanim jednak zaimplementujemy ją w języku F#, sprawdzimy, czy ta funkcja jest poprawnie zdefiniowana. W tym celu należy wykonać analizę poprawności definicji dla trzech klas równoważności ze względu na dane wejściowe:

(1) $n * m = 4$, czyli $\{(1,4); (2,2); (4,1); (-1,-4); (-2,-2); (-4,-1)\}$,
(2) $n \geq m \wedge n * m \neq 4$,
(3) $n < m \wedge n * m \neq 4$.

Ad (1)

Dla danych z klasy (1) funkcja zwróci odpowiednio wartość 1.

Ad (2)

Dane wejściowe z klasy (2) możemy przedstawić następująco:

$$n \geq m \wedge (n,m) \neq (2,2) \wedge (n,m) \neq (4,1)$$

Niech zatem $n \geq m$ oraz $n, m > 2$. Wówczas funkcja f będzie będzie zmniejszała argument n podczas kolejnych wywołań, aż dojdzie do momentu że $n = m$. Jeśli $n = m$, to n zostanie zmniejszone o 1. Wtedy zadziała część $2 + f(n, m-1)$. Dzięki temu argumenty $n, m = 2$ w pewnym momencie i wynik zostanie zwrócony, ponieważ $2 * 2 = 4$. Jeśli $m = 1$, to n będzie zmniejszane dotąd aż będzie miało wartość 4 (gdy $m = 2$, to n będzie zmniejszane aż osiągnie wartość 2).

Ad (3)

Dane wejściowe z klasy (3) również należy podzielić na kilka przypadków:

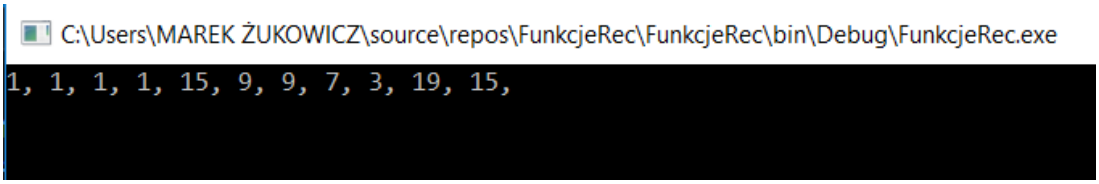
$$n < m \wedge (n,m) \neq (1,4)$$

Analizę tej klasy wykonuje się analogicznie, jak wyżej – to już pozostawiamy czytelnikowi.

Uruchamiając kod:

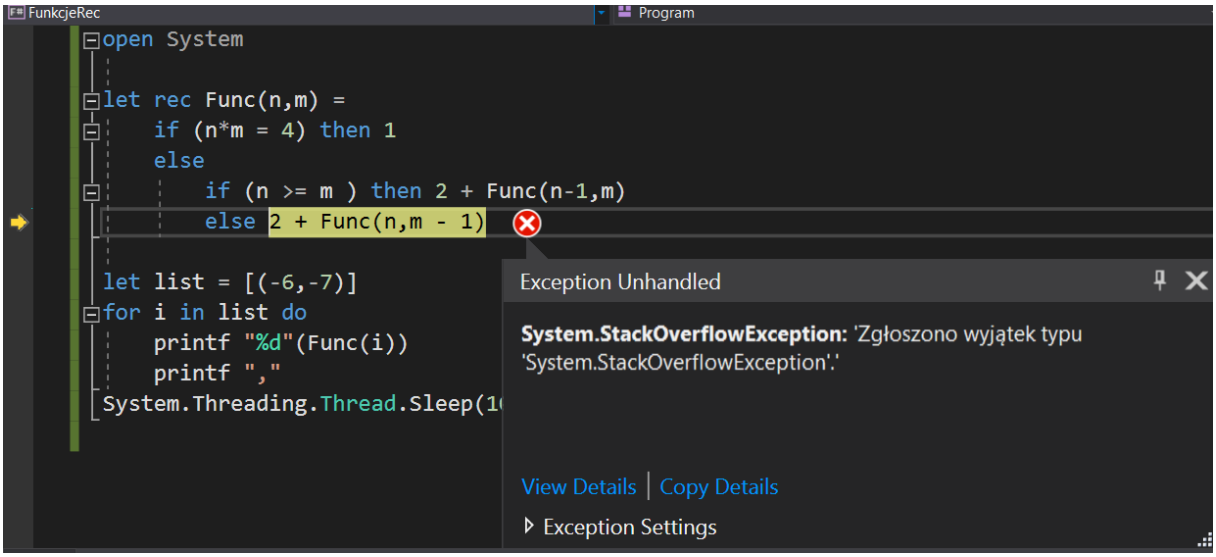
```
open System

let rec Func(n,m) =
    if (n*m = 4) then 1
    else
        if (n >= m ) then 2 + Func(n-1,m)
        else 2 + Func(n,m - 1)

let list = [(1,4);(-1,-4);(-2,-2);(-4,-1);(6,-4);(0,0);(5,3);(5,2);(5,1);(5,0);(-4,6)]
for i in list do
    printf "%d"(Func(i))
    printf ","
System.Threading.Thread.Sleep(10000)
```

otrzymujemy zestaw danych wyjściowych:



Rysunek 11: Zbiór wartości funkcji f dla tablicy argumentów tab

Na pierwszy rzut oka funkcja rekurencyjna wydaje się poprawna, ale tak niestety nie jest. Uruchamiając wyżej przedstawiony kod dla argumentu $(-6,-7)$ otrzymamy wyjątek:



Rysunek 12: Przykład wyjątku funkcji f

Taka sytuacja występuje dlatego, że nie zostanie osiągnięty warunek początkowy podczas kolejnych wywołań. Funkcja $f(n,m)$ zadziała tylko dla liczb nieujemnych (dla pary $(0,0)$ zwróci wartość 9), ponieważ nie ma zaimplementowanych ograniczeń dla liczb ujemnych – funkcja z definicji zmniejsza argumenty, a dla pary $(-6,-7)$ będzie to robić w nieskończoność, stąd pojawia się wyjątek. Tylko liczby ujemne z klasy (a) zwrócą naturalny wynik.

Przykład 9. Kolejny przykład prezentuje funkcję rekurencyjną trzech zmiennych. Załóżmy, że istnieje funkcja F trzech zmiennych, określona rekurencyjnie zgodnie z regułami:

1) *Argumenty składowe muszą być większe od 0 i muszą być liczbami naturalnymi,*
2) $F(1,1,1) = 2$,
3) *Jeżeli* $n > m, k$, *to zwracana jest wartość* $F(n-1,m,k)+1$,
4) *Jeżeli* $n \leq k, m$ *oraz* $m > k$, *to zwracana jest wartość* $F(n,m-1,k)+1$,
5) *Jeżeli* $n \leq k, m$ *oraz* $m \leq k$, *to zwracana jest wartość* $F(n,m,k-1)+1$.

Kod tej funkcji w języku F# może być zapisany następująco:

```
let rec F (n,m,k) =
    if (n < 1 || m < 1 || k < 1) then raise (System.Exception "Argument na każdej
                                            współrzędnej musi  być większy od 0")
    else
        if (n = 1 && m = 1 && k = 1) then 2
        else
```

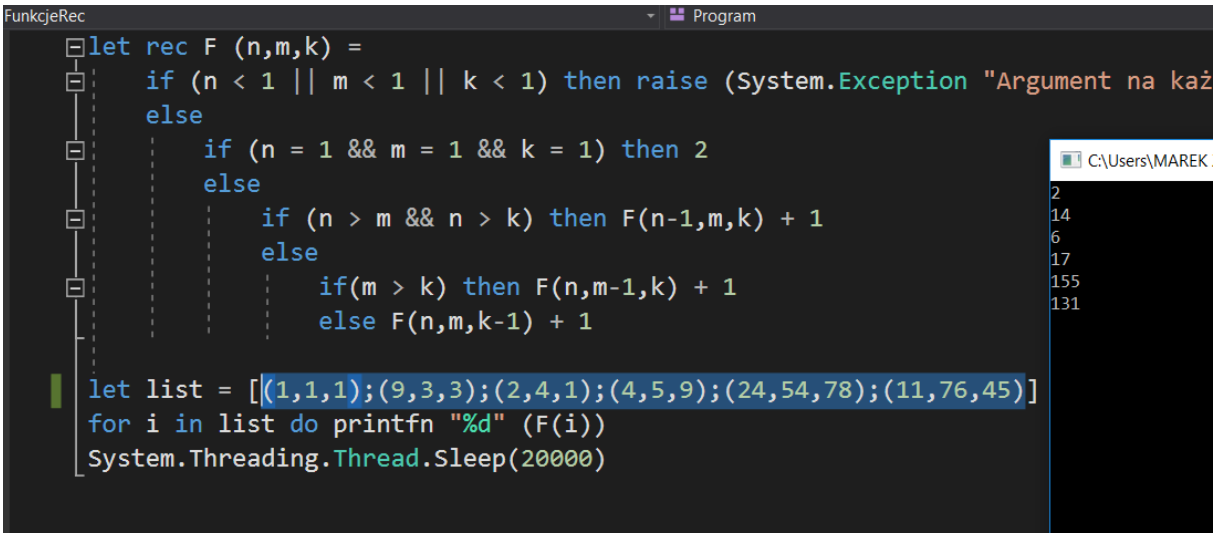
```
if (n > m && n > k) then F(n-1,m,k) + 1
else
    if(m > k) then F(n,m-1,k) + 1
    else F(n,m,k-1) + 1
```

Podanie samego wzoru funkcji to nie wszystko. Zaimplementowaną funkcję należy przetestować. Sposób, w jaki opisana została funkcja F jest na tyle przejrzysty, że można go utożsamić z opisem klas równoważności. Można zaimplementować w kodzie listę, która zawiera element każdej klasy z wyjątkiem klasy $\{(1,1,1); (9,3,3); (2,4,1); (4,5,9)\}$. Dla takich argumentów funkcja zwróci wartości odpowiednio: $2, 14, 6, 17$. Dal elementu z klasy 1 np. $(0,1,0)$ funkcja F rzuci wyjątek z treścią `"Argument na każdej współrzędnej musi być większy od 0"`.

Jedna z podstawowych zasad testowania mówi, że testowanie gruntowne jest niemożliwe, a testy przeprowadzone tylko dla jednego elementu z każdej klasy też nie są potwierdzeniem, że funkcja zawsze zadziała poprawnie. Należy wykazać wobec tego, że funkcja F zawsze osiągnie warunek początkowy oraz że nie istnieje wartość, dla której program nie zadziała – rzucenie wyjątku w tym przypadku nie oznacza niepoprawnego działania. W tym celu potrzebna będzie analiza matematyczna poprawności funkcji:

Jeśli $n > m$ i $n > k$, to zwracana jest wartość $F(n-1,m,k)+1$. Jeśli w wyniku iteracji dojdzie do sytuacji, w której $n \leq m$ lub $n \leq k$ to zwrócony zostanie wynik $F(n,m-1,k)+1$ – jeśli $m > k$ albo $F(n,m,k-1)+1$ - jeśli $m \leq k$. Wartość zmiennej n jest zmniejszana tylko w przypadku, gdy pozostałe wartości są mniejsze. Jeśli $n = k$ lub $n = m$, to funkcja zmniejszy wartość m albo k. W przypadku, gdy $n = m = k$, to również zwrócony zostanie wynik $F(n,m,k-1)+1$. Zmniejszając wartości współrzędnych w taki sposób dojdzie do sytuacji, że zwrócone zostanie wyrażenie $F(n-1,m,k)+1$. Jeżeli, w wyniku kolejnej iteracji, zmienne m albo k osiągną wartość mniejszą lub równą wartości n, to ich wartości zostaną odpowiednio zmniejszone. W ten sposób dojdziemy do sytuacji, że $(n,m,k) = (1,1,1)$, a wobec tego funkcja zwróci wynik dla każdej trójki (n,m,k) przy założeniu, że $m,n,k > 0$.

Niżej na obrazku pokazane są wartości funkcji F dla danych wejściowych z listy $(1,1,1); (9,3,3); (2,4,1); (4,5,9); (24,54,78); (11,76,45)$:



Rysunek 13: Wartości funkcji F dla wyżej wymienionych argumentów

4. Podsumowanie

Celem napisania artykułu było pokazanie, jakie problemy mogą występować podczas implementowania funkcji rekurencyjnych wielu zmiennych. Programowanie jest dobrą praktyką do uczenia się implementacji wspomnianych funkcji metodą prób i błędów. Wyjątki zwracane przez uruchomiony kod skłaniają do analizy i wyciągania pewnych wniosków, co rozwija umiejętności analityczne, matematyczne, a to z kolei jest dobrą praktyką dla własnego rozwoju w branży IT. Testerzy oprogramowania znając źródło wyjątków, mogą wyciągnąć wiele wniosków na przyszłość, co z pewnością będzie dobrą praktyką przy rozwiązaniu podobnych problemów. Mogą przewidzieć defekty czy napisać przypadki testowe dla problemów, które są analogiczne do tych omawianych w artykule. Jest to jedna ze strategii testowania stosowana w praktyce. Przewidywanie defektów jest również czynnością, która zapobiega powstawaniu defektów oraz poprawia jakość na etapie analizy, co jest bardzo pożądaną umiejętnością wśród testerów oprogramowania.

Referencje:

[1] https://pl.wikibooks.org/wiki/Programowanie/Programowanie_funkcyjne
[2] https://docs.microsoft.com/pl-pl/previous-versions/visualstudio/visual-studio-2013/dd233239(v=vs.120)
[3] https://theburningmonk.com/2011/09/fsharp-pipe-forward-and-pipe-backward/

Recenzenci:

Anna Justyna Rejrat
Cezary Piątek