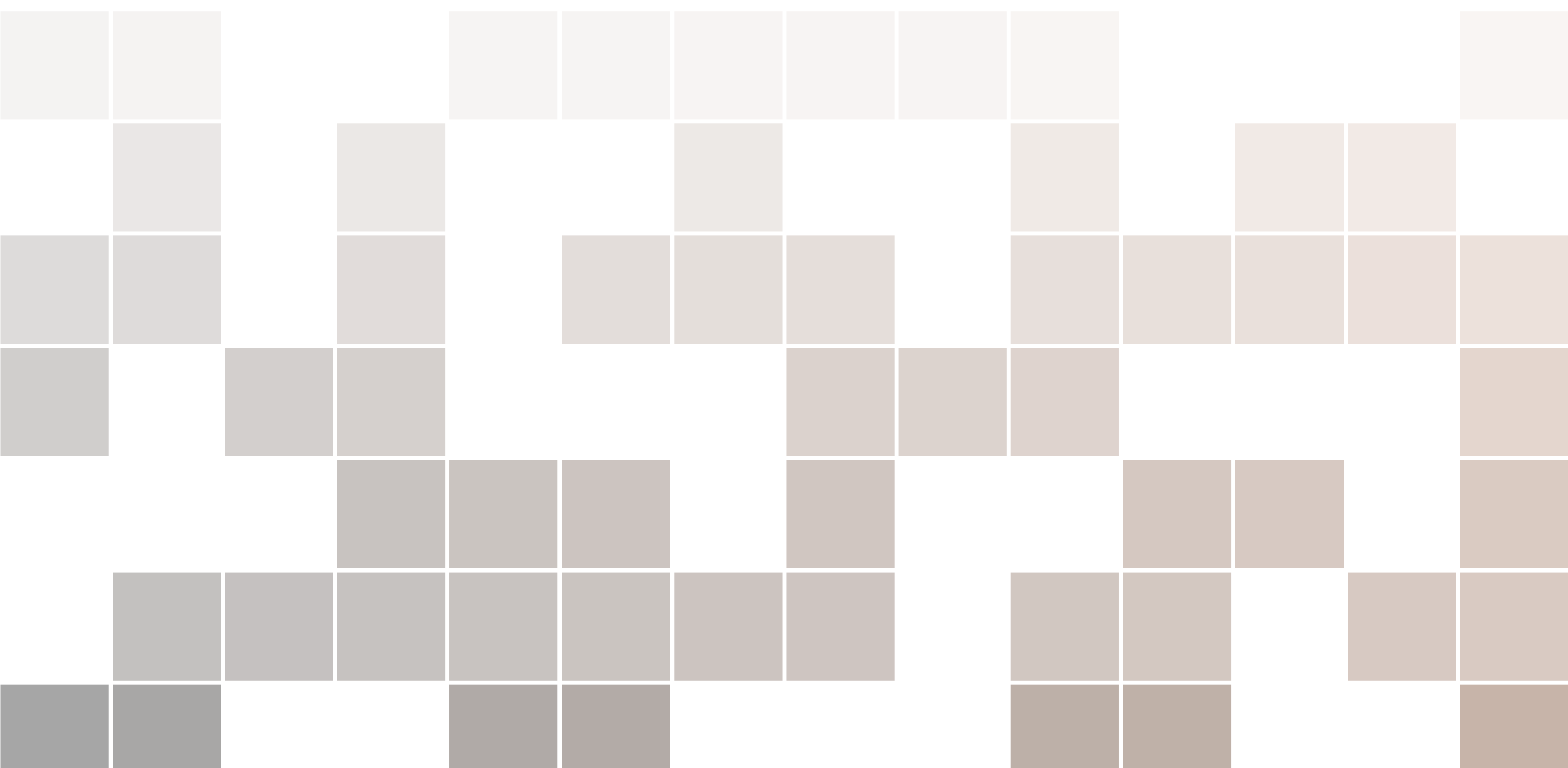
Paradygmaty programowania

Laboratorium

Łukasz Bartczuk



1. Podstawy programowania w języku F#

1.1 Programowanie funkcyjne

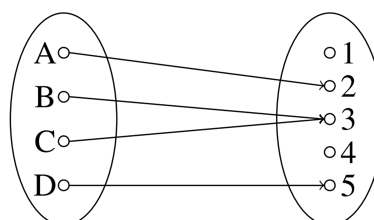
Programowanie funkcyjne, jak zresztą nazwa na to wskazuje, polega przede wszystkim na pisaniu programów, które za pomocą ciągu wywołań funkcji przetwarzają dane w celu uzyskania rezultatu, który można przedstawić użytkownikowi.

Krótkie przypomnienie informacji o funkcjach

Definicja 1.1.1 Niech dane będą dwa dowolne zbiory A i B. Funkcją f, czyli odwzorowaniem zbioru A w B nazywa się relację binarną $f \subseteq A \times B$, która każdemu elementowi ze zbioru A przyporządkowuje co najwyżej jeden element ze bioru B, tzn. jeżeli $a \in A$ i $b \in B$, to $< a, b > \in f$.

Element a nazywany jest argumentem funkcji, a element b wartością funkcji f dla argumentu a.

Graficznie funkcję można przedstawić za pomocą poniższego diagramu:



Zwróćmy uwagę, że zbiory A i B są dowolnymi zbiorami, co oznacza, że ich elementami mogą być nie tylko liczby, ale również bardziej złożone struktury jak pary, rekordy, tablice, zbiory, a nawet inne funkcje.

Funkcje można zapisać w postaci:

$$f : A \to B$$

gdzie f określa nazwę, a wyrażenie $A \to B$ typ funkcji. Sygnatury funkcji są bardzo przydatne podczas analizy programów, a zwłaszcza programów funkcyjnych.

Oczywiście funkcje mogą mieć więcej niż jeden argument. Mówimy wtedy o funkcjach wieloargumentowych. Ich sygnaturę funkcji zapisuje się następująco:

$$f : A_1 \times A_2 \times \cdots \times A_n \to B$$

Bardzo istotnym wnioskiem wynikającym z powyższej definicji jest fakt, że wartość funkcji matematycznej zależy wyłącznie od jej parametrów wejściowych.

Teoretyczne rozważania na temat funkcji będziemy kontynuować również w dalszych częściach tego opracowania.

Ktoś mógłby powiedzieć, że przecież funkcje można tworzyć w dowolnym języku programowania, więc po co wymyślać osobne języki programowania, które opierają się na funkcjach. Owszem, ale czy funkcje pisane w językach takich jak C++ są funkcjami w sensie matematycznym?

Przyjrzyjmy się następującemu programowi:

```
int c = 0;

int funkcja(int a, int b)
{
  c = a+b+c;
  return c;
}
```

Zawiera on prostą funkcję, która sumuje wartości swoich parametrów i do wyniku tej operacji, dodaje jeszcze wartość zmiennej globalnej `c`. Wynik tego dodawania zapisuje w tej samej zmiennej `c` oraz zwraca jako swój rezultat.

Przyjrzyjmy się teraz wywołaniu tej funkcji:

```
cout<<funkcja(1,2)<<endl;
cout<<funkcja(1,2)<<endl;
cout<<funkcja(1,2)<<endl;
```

W rezultacie działania tego programu na ekranie otrzymamy następujące wartości:

```
3
6
9
```

Zwróćmy uwagę, że za każdym razem gdy wywołamy tę funkcję z tymi samymi argumentami uzyskujemy inną wartość. Może to wskazywać, że funkcja ta ma pewien ukryty parametr wejściowy, który nie jest zdefiniowany. Tym parametrem jest tzw. stan, w którym funkcja ta jest wywoływana. Najprościej można stan przedstawić jako wszystkie elementy zewnętrzne (w tym pamięć, sprzęt, sieć internet), które maja wpływ na sposób działania funkcji. Stan ten może ulegać modyfikacjom. Jeżeli zmiany stanu odbywają się wewnątrz funkcji, możemy powiedzieć, że funkcja ta posiada **efekt uboczny** (w tym przypadku polegający na wprowadzeniu nowej wartości do tego samego obszaru pamięci – czyli wykonaniu operacji przypisania).

Czy wobec tego modyfikowalny stan ułatwi nam analizę programów, czy ją utrudni? Oczywiście utrudni. Jest to spowodowane tym, że nie możemy powiedzieć, jaki będzie

dokładny rezultat działania funkcji, patrząc na samą funkcję. Musimy jeszcze znać stan w jakim ta funkcja będzie wywołana.

Powiecie teraz, że przecież i tak nie korzystacie ze zmiennych globalnych, bo nie jest to dobry styl programowania, albo programujecie obiektowo w takich językach jak JAVA czy C#, gdzie nie można deklarować zmiennych poza klasą czy funkcją. Oczywiście macie rację, ale modyfikacje składowych klasy, w sposób które nie jest jawnie określony w programie również utrudnia nam analizę programu, a także tworzenie testów jednostkowych, czy aplikacji równoległych lub asynchronicznych.

Już sam fakt istnienia pewnego elementu, który może wpłynąć niejawnie na sposób działania funkcji powoduje, że funkcje tworzone w językach imperatywnych nie są funkcjami w sensie matematycznym. Jednak istnienie modyfikowalnego stanu może również spowodować, że funkcje nie będą podlegały podstawowym prawom matematyki.

Przeanalizujmy następujący program:

```
#include <iostream>

using namespace std;

int f(int a, int b) {
  return a+b;
}

int g(int a, int b) {
  return a*b;
}

int main()
{
  cout<<"f(1,2) + g(3,4) = "<<f(1,2)+g(3,4)<<endl;
  cout<<"g(3,4) + f(1,2) = "<<g(3,4)+f(1,2)<<endl;
  return 0;
}
```

W tym przypadku, na ekranie zobaczymy:

```
f(1,2) + g(3,4) = 15
g(3,4) + f(1,2) = 15
```

Oczywiście jest to w stu procentach zgodne z naszą intuicją, bo w końcu od szkoły podstawowej nauczyciele matematyki uczą nas że dodawanie jest przemienne, czyli $a+b = b+a$. Co się jednak stanie jeżeli do powyższego programu wprowadzimy modyfikowalny stan? Przykładowo zmieńmy ten program w następujący sposób:

```
#include <iostream>

using namespace std;

int c = 1;

int f(int a, int b) {
  c = a+b;
  return a+b;
}
```

```
int g(int a, int b) {
  return a*b*c;
}

int main()      {
  cout<<"f(1,2)␣+␣g(3,4)␣=␣"<<f(1,2)+g(3,4)<<endl;
  return 0;
}
```

Po jego uruchomieniu uzyskamy wartość: 39. Jeżeli jednak zamienimy linijkę na

```
cout<<"g(3,4)␣+␣f(1,2)␣=␣"<<g(3,4)+f(1,2)<<endl;
```

to wynik będzie wynosi znowu 15.

Przykład ten pokazuje, że istnienie w aplikacji modyfikowalnego stanu powoduje, że kolejność wykonywanych działań wpływa na ostateczny rezultat naszego programu i ma bardzo duże znaczenie podczas jego analizy. Potwierdza to twierdzenie, że funkcje tworzone w językach imperatywnych, mogą nie być funkcjami w sensie matematycznym.

Języki funkcyjne starają się ograniczać wpływ stanu aplikacji na działanie funkcji. Znacząco ułatwia to analizę programu, gdyż wystarczy popatrzeć na funkcję (nawet w oderwaniu od reszty aplikacji) i możemy powiedzieć, co ta funkcja robi i w jakich sytuacjach działa dobrze, a w jakich będzie działała błędnie. Umożliwia to wykorzystanie wielu ciekawych technik programistycznych, które są doceniane również przez społeczności programujące imperatywnie, a w szczególności obiektowo (i spowodowało wprowadzenie cech funkcyjnych do języków takich jak JAVA, C# czy C++).

1.2 Wprowadzenie do języka F#

Język F# jest nowoczesnym językiem funkcyjnym pozwalającym na tworzenie aplikacji dla platformy .NET. Jest to język typu "Functional First". Oznacza to, że wspiera on przede wszystkim paradygmat programowania funkcyjnego. Pozwala jednak na tworzenie oprogramowania również w stylu imperatywnym i obiektowym, choć w ramach tych zajęć nie będziemy zagłębiać się w tych cechach tego języka.

Choć język ten nie jest jeszcze tak szeroko stosowany jak inne główne języki platformy .NET - C# lub Visual Basic, to możemy w nim tworzyć dowolny rodzaj aplikacji (konsolowe, okienkowe, internetowe, czy inne).

W kolejnych punktach przyjrzymy się podstawowym elementom języka F# , a następnie zobaczymy jak możemy tworzyć i uruchamiać – w różnych środowiskach – aplikacje stworzone za jego pomocą.

1.3 Deklarowanie zmiennych

Nowe zmienne są deklarowane za pomocą słowa kluczowego `let`:

```
let x = 3+4
```

Powyższa instrukcja powoduje wyznaczenie wartości wyrażenia znajdującego się po prawej stronie znaku równości i powiązanie wyniku z nazwą x. W interpreterze zobaczymy następujący rezultat:

```
val x : int = 7
```

Oznacza to oczywiście, że utworzyliśmy wartość całkowitą 7 (typ `int`), która będzie powiązana z symbolem `x`.

Domyślnie język F# traktuje pamięć jako obszar jednokrotnego zapisu. Oznacza to, że pamięć (w tym przypadku wartość zmiennej) można ustawić tylko podczas deklaracji i później nie można jej już zmienić [1], tylko zwolnić do ponownego wykorzystania. Dlatego można powiedzieć, że wykorzystanie określania zmienna jest w tym przypadku lekkim nadużyciem, w końcu symbol `x` nie określa zmiennej w sensie innych (imperatywnych) języków programowania. Lepszym i chyba bardziej poprawnym określeniem byłoby etykieta dla wartości, ale ponieważ w innych materiałach jakie możecie odnaleźć na tego języka najczęściej korzysta się jednak z terminu źmienna", więc my również pozostaniemy przy tej nazwie.

1.4 Proste typy danych w F#

Zwróćmy uwagę, że podczas deklaracji zmiennej `x` nie określiliśmy jej typu. Choć język F# jest językiem ściśle typowanym, tzn. każdy deklarowany symbol musi mieć przypisany typ, który później nie może być zmieniony, nie ma potrzeby, aby ręcznie określać typ każdej zmiennej (jak również parametru funkcji – co zobaczymy później). Jest to możliwe ponieważ, interpreter i kompilator F# mają wbudowany bardzo rozwinięty system wnioskowania typu. Mechanizm ten jest bardzo rozbudowany i sprawdza się w większości przypadków, czasami musimy jednak wspomóc go poprzez ręczne określenie typu zmiennej. Możemy to zrobić podając po nazwie zmiennej dwukropek oraz nazwę typu:

```
let x:int = 12
```

Ponieważ F# jest językiem, który kompiluje się do platformy .NET, tworząc w nim programy mamy dostęp do wszystkich typów, które są na niej zdefiniowane. Podobnie operacje jakie możemy na nich wykonywać. Są jednak niewielkie różnice w nazewnictwie np. typ liczb zmiennoprzecinkowych podwójnej precyzji, który w C# nazywa się `double`, w F# określany jest jako `float` [2]. F# jest wyposażony w standardowy zestaw operatorów matematycznych, relacyjnych oraz logicznych. Miłym dodatkiem jest operator `**` służący jako operacja potęgowania, czyli operację x^y możemy zapisać w formie `x**y`. W języku tym podstawowe funkcje arytmetyczne tj. np. sin, cos, abs, min, max, log, log10, round, floor, ceil, sqrt i inne. Pełną listę operatorów F# można znaleźć pod adresem: https://fsharp.github.io/fsharp-core-docs/reference/fsharp-core-operators.html. Oczywiście możemy również korzystać z wszystkich funkcji i wartości zdefiniowanych w klasie `System.Math`.

1.4.1 Obsługa łańcuchów znaków w F#

Łańcuchy znaków zapisujemy w cudzysłowach:

[1] Język F# nie jest językiem czysto funkcyjnym więc umożliwia tworzenie kodu wykorzystującego efekty uboczne – między innymi umożliwiające modyfikowanie wartości zmiennych – ale jak już wspominałem, nie będzie to celem tych zajęć

[2] Dostępna jest również nazwa double lecz jest ona zdecydowanie rzadziej wykorzystywana

```
let tekst = "Ala ma kota"
```

Instrukcja ta spowoduje utworzenie niemutowalnego obiektu klasy String. Ponieważ jest to pełnoprawny obiekt, zawiera on właściwości i metody, które służą do wykonywania operacji na tym łańcuchu znaków. Przykładowo, jeżeli będziemy chcieli określić długość tego łańcucha, możemy to zrobić w następujący sposób:

```
tekst.Length
```

Podobnie będziemy postępowali, w przypadku gdy będziemy chcieli wywołać jakąś metodę działającą na stringach np. przy wycinaniu jego fragmentu:

```
tekst.Substring(4,2)
```

Powyższa instrukcja zwróci nam nowy łańcuch znaków "ma". Łańcuchy znaków sa indeksowane od zera. Opis wszystkich dostępnych funkcji możemy odnaleźć pod adresem: https://docs.microsoft.com/pl-pl/dotnet/api/system.string?view=net-5.0#properties.

W F# istnieją trzy użyteczne operatory umożliwiające pracę na stringach '+' służący do łączenia dwóch łańcuchów znaków, [] - umożliwiający dostęp do pojedynczego znaku oraz [..] - wycinający fragment. Przykładowo, jeżeli będziemy chcieli pobrać pierwszy znak z łańcucha możemy to zrobić następująco:

```
tekst.[0]
```

Z kolei zapis

```
tekst.[4..5]
```

da identyczny rezultat jak wcześniej pokazywane wywołanie funkcji `Substring`, przy czym w przypadku funkcji podawaliśmy indeks pierwszego znaku, od którego chcieliśmy skopiować fragment oraz liczbę znaków do skopiowania, a w przypadku operatora [..] podajemy indeksy pierwszego i ostatniego znaku do skopiowania.

Bardzo ciekawym rozwiązaniem, są interpolowane łańcuchy znaków zaimplementowane w F# . Załóżmy, że dokonujemy jakichś obliczeń i wyniki chcielibyśmy wyświetlić na ekranie. Możemy to zrobić np. w następujący sposób:

```
let x = 12.0
let wynik = sin x
System.Console.WriteLine("Wartoscia funkcji sin dla argumentu "
   +x.ToString()+" jest "+wynik.ToString())
```

Jak widzimy przygotowanie odpowiedniego komunikatu nie jest zbyt przyjemne. Po pierwsze należy wykonać łączenie wielu łańcuchów znaków. Po drugie nazwy x i wynik są przypisane do wartości typu float, więc aby operator '+' mógł poprawnie działać, należy je zamienić na łańcuchy znaków, co robimy za pomocą metod ToString().

Dzięki łańcuchom interpolowanym możemy to samo zapisać prościej:

```
let x = 12.0
let wynik = sin x
System.Console.WriteLine($"Wartoscia funkcji sin dla argumentu ↩
   {x} jest {wynik}")
```

Łańcuch interpolowany rozpoczynamy od znaku $ i wewnątrz niego w nawiasach { } podajemy wyrażenie, którego wartość ma być obliczona i wstawiona do łańcucha w jego miejsce.

1.4.2 Konwersja typów w F#

Podczas tworzenia aplikacji w F# należy pamiętać, że język ten, w przeciwieństwie do innych języków programowania, **nie wspiera niejawnych konwersji** wobec tego typ wartości oraz typ przypisany nazwy muszą być zgodne. Wykonanie poniższej instrukcji

```
let x:float = 12
```

Zakończy się zwróceniem komunikatu o błędzie:
error FS0001: This expression was expected to have type ’float’ but here has type ’int’.

Nie ma również możliwości wykonywania operacji na wartościach różnych typów. Przykładowo poniższa instrukcja, która w innych językach programowania wykona się bez problemu, tutaj spowoduje wyświetlenie komunikatu o błędzie.

```
let x = 12.0 + 3
```

Aby powyższa instrukcja była poprawna, należy dokonać jawnej konwersji typu, której dokonujemy – w przypadku typów prostych – za pomocą funkcji o takiej samej nazwie jak typ docelowy:

```
let x = 12.0 + (float 3)
```

Funkcje te pozwalają również na konwersję łańcuchów znaków na określony typ np:

```
let x = 12.0 + (float "3")
```

Musimy przy tym pamiętać, że jeżeli łańcuch znaków nie będzie zawierał poprawnej liczby, to rzucony zostanie wyjątek.

1.5 Wyrażenia

Każda instrukcja w języku F# jest wyrażeniem, które zwraca wartość. Oczywiście jak w każdym języku, możemy deklarować wyrażenia arytmetyczne, bitowe, relacyjne oraz inne. Mogą się one składać z zmiennych, wartości stałych, wywołań funkcji połączonych za pomocą odpowiednich operatorów. Ponieważ są one w większości takie same jak w innych językach (np. C++ czy C#) nie będę ich tutaj szczegółowo rozpisywał. Należy zwrócić jednak uwagę na operatory równości i nierówności oraz negacji logicznej, które są zdefiniowane odpowiednio za pomocą symboli "=", «>óraz ńot".

Do bardziej rozbudowanych wyrażeń należy wyrażenie warunkowe, które w F# przyjmuje postać:

```
if warunek then
  wyrazenie_prawda
else
  wyrazenie_falsz
```

Wyrażenie `warunek` musi zwracać wartość logiczną (typu `bool`). Do jego zdefiniowania można wykorzystać operatory logiczne: true (prawda), false (fałsz), || (logiczne lub), && (logiczne i), not (negacja) oraz operatory relacyjne: >,<,<=,>=,<>,=.

Zwróćcie uwagę, że choć składania tego wyrażenia jest podobna do instrukcji warunkowej istniejącej w innych językach programowania, mówimy tutaj o wyrażeniu. Oznacza to, że zwraca ono wartość, która może być wykorzystana do dalszych obliczeń, a samo wyrażenie może być częścią większej konstrukcji np:

```
let x = 13
let wynik = 20 + if x > 12 then 10 else 0
```

Ponieważ instrukcja warunkowa jest wyrażeniem, z obu swoich części musi ona zwracać wartość, a ponadto wartości te muszą być tego samego typu.

R Jak wspomniano, wszystkie instrukcje, jako wyrażenia, muszą mieć określony typ danych i zwracać wartość. Są jednak instrukcje, w których nie można jawnie określić wartości zwracanej. Przykładem takiej instrukcji jest chociażby polecenie `printfn`, które jest odpowiedzialne za wyświetlenie komunikatu w konsoli. W takim przypadku wyrażenie to zwraca wartość specjalnego typu o nazwie `Unit`. Typ ten ma tylko jedną wartość oznaczaną jako `()`.

Możliwe jest również wykorzystanie wyrażenia warunkowego tylko z określonym warażeniem_prawda, ale w tym przypadku wyrażenie to musi zwracać wartość typu Unit.

1.6 Deklarowanie funkcji

W F# funkcje możemy deklarować za pomocą wyrażenia funkcyjnego, które przyjmuje postać:

```
fun <lista parametrow> -> <cialo funkcji>
```

Przykładowo wydanie następującego polecenia:

```
fun x -> x+1
```

zakończy się uzyskaniem następującego wyniku:

```
val it : x:int -> int
```

Zwróćmy tutaj uwagę na kilka rzeczy. Po pierwsze nie podaliśmy nazwy funkcji (utworzyliśmy funkcję anonimową, dlatego interpreter określił ją jako wartość o nieokreślonej nazwie `it`). Po drugie choć nie podaliśmy typów zmiennych ani rezultatu funkcji interpreter poprawnie rozpoznał je jako `int` - dlaczego?. Po trzecie i najważniejsze interpreter określił rezultat działania instrukcji jako `val`, czyli wartość.

W językach funkcyjnych, funkcje traktowane są jako óbywatele"pierwszej klasy, co oznacza, że można je traktować jak każdą inną wartość. Powoduje to, że funkcję podobnie jak np. liczbę całkowitą, możemy powiązać z etykietą za pomocą instrukcji `let`:

```
let inkrementuj = fun x -> x+1
```

w tym przypadku w interpreterze zobaczymy:

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

Zwróćmy uwagę na rezultat powyższych fragmentów kodu. Zauważmy, że to co występuje po słowie val jest niczym innym jak wspomnianą wcześniej sygnaturą funkcji (jej zapis jest zgodny z notacją matematyczną, przy czym notacja F# dołącza jeszcze nazwy parametrów funkcji).

Oczywiście możemy deklarować również funkcje wielu parametrów. Zapisujemy je w następujący sposób:

```
let dodaj = fun a b -> a+b
```

Powyższa instrukcja do nazwy dodaj przypisze funkcję anonimową o dwóch parametrach a i b. Zwróćmy uwagę, że listy parametrów **nie musimy** ujmować w nawiasy okrągłe i **nie możemy** rozdzielać ich przecinkami. Do rozdzielania parametrów wykorzystuje się białe znaki (spacja lub tabulacja).

W powyższej instrukcji zobaczymy

```
val dodaj : a:int -> b:int -> int
```

Ta dziwna sygnatura wyjaśni się w dalszej części tego opracowania, gdy będziemy szczegółowo omawiali jak F# traktuje funkcje wielu zmiennych.

Ponieważ funkcje są najczęściej deklarowaną jednostką języka F# można skorzystać z trochę uproszczonej składni:

```
let dodaj a b = a+b
```

w której po słowie kluczowym `let` podajemy nazwę funkcji, a następnie jej parametry (oddzielając je oczywiście za pomocą białych znaków). Następnie podajemy znak równości i wyrażenie, które stanowi ciało funkcji.

Bez względu na sposób deklaracji, utworzoną funkcję możemy wykorzystać do wyznaczenia jej wartości:

```
let wynik = dodaj 12 34
```

Jak widzimy, wywołanie funkcji (lub jak określa się to w językach funkcyjnych zaaplikowanie funkcji do prametrów) dokonuje się poprzez podanie jej nazwy oraz listy wymaganych atrybutów. Wyrażenia wyznaczające wartości poszczególnych argumentów oddzielamy od siebie białymi znakami. Jeżeli wyrażenie określające wartość atrybutu nie jest odwołaniem do etykiety lub nie jest to wartość stała, to takie wyrażenie należy wziąć w nawiasy np:

```
let wynik1 = dodaj (12+34) 56
let wynik2 = dodaj (dodaj 12 34) 56
```

R Jeżeli chcielibyśmy zadeklarować funkcję nie przyjmującą parametrów lub nie zwracającą wyniku należy wykorzystać typ Unit. Przykładowo poniższa funkcja nieprzyjmuje żadnego parametru i zawsze zwróci wartość 1:

```
let zawsze1 () = 1
```

Podobnie jak przy deklarowaniu innych wartości, przy tworzeniu funkcji nie musimy jawnie określać typu jej parametrów oraz typu wartości zwracanej. Czasami jednak jest to konieczne. Możemy to wtedy zrobić w następujący sposób:

```
let dodaj (a:float) (b:float) : float = a+b
```

W tym przypadku typ podany pomiędzy znakami ':' oraz '=' oznacza typ wyniku funkcji.

Jako ciało funkcji możemy wykorzystać dowolne wyrażenie, również takie które będzie zawierało wiele linii kodu np.

```
let przyklad x =
  if x<0 then
    printfn "wartosc parametru jest ujemna"
  else
    printfn "wartosc parametru jest nieujemna"
```

W powyższym przykładzie należy zwrócić szczególną uwagę na zastosowane wcięcia kodu. W przeciwieństwie do innych języków, których składania oparta jest na składni C++ lub Pascal gdzie blok kodu oznacza się odpowiednio symbolami { } lub begin end, w F# blok kodu określa się za pomocą wcięć. Każda linijka kodu, która ma należeć do tego samego bloku musi rozpoczynać się w tej samej kolumnie. Jeżeli chcemy zdefiniować blok zagnieżdżony w innym bloku musi on być bardziej wcięty niż blok nadrzędny (przy czym liczba spacji nie ma znaczenia). Jeżeli powyższy program zapisalibyśmy, tak aby wszystkie jego linie rozpoczynały się w tej samej kolumnie, to w najlepszym przypadku uzyskamy ostrzeżenie, a w najgorszym oczywiście nasz program nie będzie działał poprawnie.

Wewnątrz bloku kodu możemy deklarować zmienne lokalne, które będą widoczne tylko wewnątrz niego. Robimy to również za pomocą instrukcji `let`:

```
let przyklad1 x =
  let y = 12
  if x > 0 then
    let z = 12
    x+y+z
  else
    let z = 34
    x+y+z
```

W przypadku funkcji instrukcja `let` jest wyrażeniem, czyli musi zawierać wyrażenie określające jej wartość. Dlatego poniższa funkcja jest błędna (wyrażenie `let` nie ma ciała):

```
let przyklad1 x =
    let y = 12
```

Musimy pamiętać, że blok kodu określa zakres widoczności zmiennej. Z tego powodu poniższa funkcja również będzie błędna (w linii 10 zmienna z jest zapisana poza zakresem swojej widoczności)

```
let przyklad2 x =
  let y = 12
  let w =
```

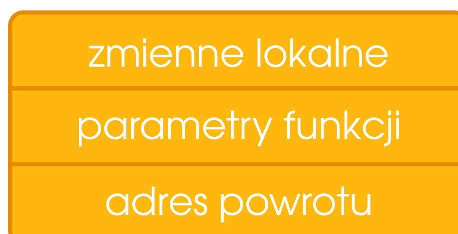
```
    if x > 0 then
      let z = 12
      x+y+z
    else
      let z = 34
      x+y+z
  z
```

1.7 Rekurencja

Ponieważ w programowaniu funkcyjnym staramy się unikać zmiany stanu (nawet lokalnego), dlatego nie powinniśmy korzystać z instrukcji typowo imperatywnych. Jedną z nich jest pętla (w dowolnym rodzaju) [3]. Wobec tego jedynym sposobem, aby w językach funkcyjnych [4], wielokrotnie wywołać ten sam kod jest wykorzystanie funkcji rekurencyjnych, czyli takich, które wywołują same siebie. W przypadku języka F# (jak również w przypadku innych języków funkcyjnych), deklaracje rekurencyjne tworzymy za pomocą specjalnej instrukcji `let rec`. Przykładowo, poniżej mamy zadeklarowaną typową funkcję rekurencyjną pozwalającą wyznaczyć nam wartość funkcji silnia dla zadanego argumentu:

```
let rec silnia x =
  if x<0 then
    failwith "nieprawidlowe dane wejsciowe"
  elif x = 0 then
    1
  else
    x*silnia (x-1)
```

Jest to typowy zapis funkcji silnia wynikający wprost z jej definicji matematycznej. Jak mam nadzieję pamiętacie, w językach imperatywnych podczas wywołania funkcji na stosie rezerwowana jest pamięć, dla tzw. ramki aktywacji. Zawiera ona wszystkie elementy konieczne do uruchomienia funkcji, m.in. wartości zmiennych lokalny oraz parametry funkcji i tzw. adres powrotu (miejsce w pamięci zawierającą instrukcję, która należy wykonać jako następną po zakończeniu funkcji). Schematycznie ramkę taką (w uproszczonej formie) można przedstawić jako:

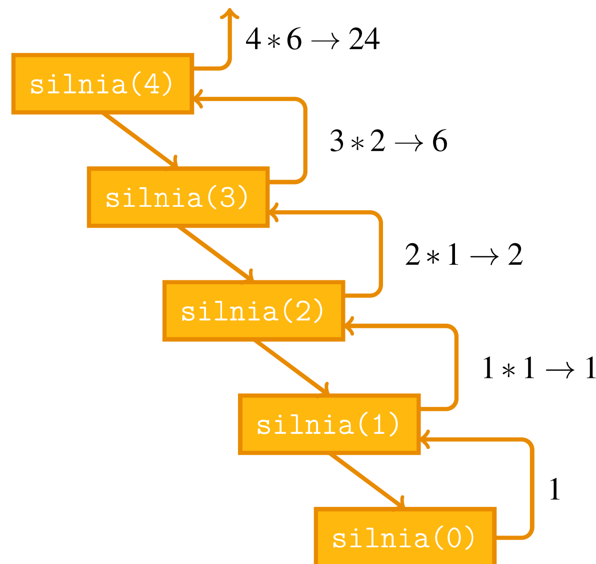


Rysunek 1.1: Uproszczona struktura ramki aktywacji

[3] Zastanówcie się dlaczego?

[4] Oczywiście korzystając z języków czysto funkcyjnych lub nie korzystając z cech imperatywnych

Oczywiście, (prawie) każde wywołanie funkcji tworzy swoją ramkę aktywacji, która jest automatycznie usuwana z pamięci, gdy następuje powrót z funkcji. Z tego powodu wywołania funkcji określa się jako instrukcje ciężkie. Jest to szczególnie widoczne podczas uruchamiania funkcji rekurencyjnych. Poniżej schematycznie przedstawione jest zaaplikowanie przedstawionej wcześniej funkcji silnia dla argumentu 4: Oczywiście



pamięć przeznaczona na stos nie jest nieograniczona, dlatego dla im większego argumentu będziemy chcieli policzyć wartość funkcji silnia, tym więcej będzie wywołań funkcji i tym więcej pamięci na stosie będziemy potrzebować. Obliczenia będziemy mogli prowadzić do momentu, gdy pamięć nam się skończy, a wtedy uzyskamy wyjątek Process is terminated due to StackOverflowException.

Czy to oznacza, że programy rekurencyjne, zwłaszcza te tworzone w językach funkcyjnych, muszą być nieefektywne? Niekoniecznie. Przyjżyjmy się jaki kod języka pośredniego jest generowany dla powyższej funkcji:

```
method public static int32
silnia(
int32 x
) cil managed
{
.maxstack 5
.locals init (
  [0] string V_0
)

IL_0000: ldarg.0       // x
IL_0001: ldc.i4.0
IL_0002: bge.s         IL_0006
IL_0004: br.s          IL_0008
IL_0006: br.s          IL_0015
IL_0008: ldstr         "x nie moze byc mniejszy od zera"
IL_000d: stloc.0       // V_0
IL_000e: ldloc.0       // V_0
```

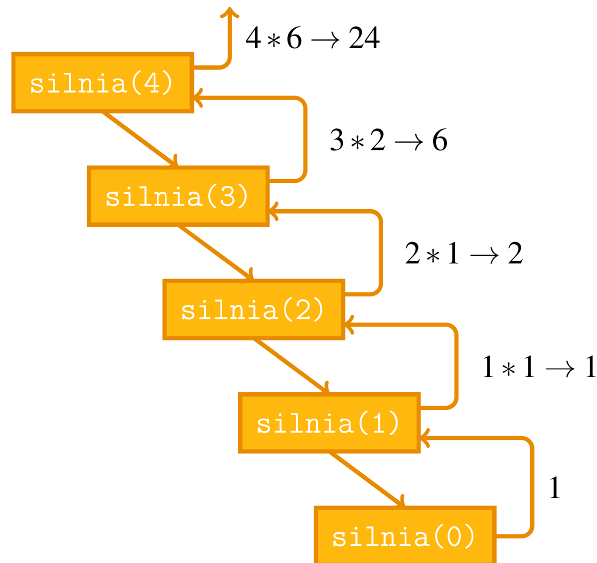
```
IL_000f: call         class [mscorlib]System.Exception [FSharp.↪
   Core]Microsoft.FSharp.Core.Operators::Failure(string)
IL_0014: throw
IL_0015: ldarg.0      // x
IL_0016: brtrue.s     IL_001a
IL_0018: br.s         IL_001c
IL_001a: br.s         IL_001e
IL_001c: ldc.i4.1
IL_001d: ret
IL_001e: ldarg.0      // x
IL_001f: ldarg.0      // x
IL_0020: ldc.i4.1
IL_0021: sub

IL_0022: call         int32 Program::silnia(int32)
IL_0027: mul
IL_0028: ret

} // end of method Program::silnia
```

Oczywiście nie interesuje nas całość tego kodu, a jedynie trzy ostanie linijki. Są one odpowiedzialne za rekurencyjne wywołanie funkcji silnia (IL_0022 call), wykonanie mnożenia wyniku tego wywołania przez wartość parametru x (IL_0027 mul) oraz zakończenie funkcji i zwrócenie do funkcji wywołującej wartości znajdującej się na samej górze stosu (w tym przypadku jest to wartość mnożenie z poprzedniej instrukcji) - IL_0028 ret. Czyli w przypadku tego programu adresem powrotu ustawianym podczas rekurencyjnego wywołania funkcji silnia będzie adres linii IL_0027, ponieważ to jest następna instrukcja do wykonanie po zakończeniu wywoływania funkcji. Właśnie ze względu na występowanie tej instrukcji konieczne jest tworzenie nowej ramki stosu podczas wywołania funkcji w linii (IL_0022).

Napiszmy ten sam program, ale w trochę inny sposób:

```
let rec silnia (x:bigint) (a:bigint) =
  if x<0I then
    failwith "nieprawidlowe dane wejsciowe"
  elif x = 0I then
    a
  else
    silnia (x-1I) (x*a)
```

Zwróćmy uwagę, że w tym przypadku funkcja zyskała dodatkowy parametr a. Zawiera on częściowy wynik mnożenia wartości wartości funkcji, przez jej argument. Jest to pokazane na schemacie 1.2. Ponieważ mnożenie wynik mnożenia jest teraz przekazywany jako parametr do funkcji silnia samo mnożenie musi być wykonane przed wywołaniem funkcji (a nie po jak w przypadku wcześniejszej wersji). Co nam to daje? Przyjrzyjmy się znowu wygenerowanym przez kompilator instrukcjom języka pośredniego.



Rysunek 1.2: Wywołanie funkcji silnia z akumulatorem dla argumentu 4

```
.method private hidebysig static int32
SilniaAcc(
int32 x,
int32 n
) cil managed
{
.maxstack 3
.locals init (
[0] bool V_0,
[1] bool V_1,
[2] int32 V_2
)
IL_0000: nop
IL_0001: ldarg.0       // x
IL_0002: ldc.i4.0
IL_0003: clt
IL_0005: stloc.0       // V_0
IL_0006: ldloc.0       // V_0
IL_0007: brfalse.s     IL_0014
IL_0009: ldstr         "x nie moze byc mniejszy od zera"
IL_000e: newobj        instance void [System.Runtime]System.↪
    Exception::.ctor(string)
IL_0013: throw
IL_0014: ldarg.0       // x
IL_0015: ldc.i4.0
IL_0016: ceq
IL_0018: stloc.1       // V_1
IL_0019: ldloc.1       // V_1
IL_001a: brfalse.s     IL_0020
IL_001c: ldarg.1       // n
IL_001d: stloc.2       // V_2
IL_001e: br.s          IL_002e
IL_0020: ldarg.0       // x
IL_0021: ldc.i4.1
```

```
IL_0022: sub
IL_0023: ldarg.0      // x
IL_0024: ldarg.1      // n


IL_0025: mul
IL_0026: call         int32 Test.Program::SilniaAcc(int32,int32)
IL_002b: stloc.2      // V_2
IL_002c: br.s         IL_002e
IL_002e: ldloc.2      // V_2
IL_002f: ret

} // end of method Program::SilniaAcc
```

Znowu interesujące są dla nas tylko wybrane instrukcje zaznaczone pogrubieniem. Zwróćmy uwagę, że w tym momencie instrukcja IL_0025 mul jest uruchamiana przed rekurencyjnym wywołaniem funkcji silnia (linia IL_0026). Po tym wywołaniu funkcji następuje ciąg instrukcji:

1. IL_002b stloc.2 - zapisanie wyniku funkcji silnia (który znajduje się na szczycie stosu) do drugiej zmiennej lokalnej i usunięcie go ze stosu.
2. IL_002c br.s IL_002e - wykonanie skoku bezwarunkowego do następnej linii.
3. IL_002e ldloc.2 - odczytanie wartości drugiej zmiennej lokalnej i umieszczenie go na szczycie stosu
4. IL_002f ret - zakończenie funkcji i zwrócenie do funkcji wywołującej wartości znajdującej się na samej górze stosu

Zwróćmy uwagę, że w tym przypadku wygenerowane przez kompilator języka C# instrukcje, są nadmiarowe i nie mają większego sensu. Równie dobrze, bez wpływu na sposób działania funkcji trzy pierwsze z powyższych czterech można wyrzucić, czyli końcówka powyższego kodu wyglądałaby następująco:

```
IL_0025: mul
IL_0026: call         int32 Test.Program::SilniaAcc(int32,int32)
IL_002b: ret

} // end of method Program::SilniaAcc
```

Jak widzimy, po rekurencyjnym wywołaniu funkcji silnia następną instrukcją jest już tylko zakończenie tej funkcji. Dlatego adres powrotu, wskazujący na wnętrze funkcji silnia jak również wartości zmiennych lokalnych i parametrów nie są nam już dalej potrzebne. Możemy je nadpisać nowymi wartościami i dzięki czemu ponownie wykorzystać utworzoną już ramkę aktywacji. Z tego powodu interpretery i kompilatory języków funkcyjnych pozwalają generowanie efektywnego kodu języka maszynowego, które wszystkie wywołania odpowiednio napisanych funkcji rekurencyjnych obsłużyć za pomocą jednej funkcji aktywacji. Kompilator języka F# idzie nawet krok dalej:

```
.method public static int32
silnia(
int32 x,
int32 n
) cil managed
{
        .custom instance void [FSharp.Core]Microsoft.FSharp.Core↪
          .CompilationArgumentCountsAttribute::.ctor(int32[])
```

```
        = (01 00 02 00 00 00 01 00 00 00 01 00 00 00 00 00 ) // ↪
           ................
        .maxstack 5
        .locals init (
        [0] string V_0
        )
        IL_0000: ldarg.0       // x
        IL_0001: ldc.i4.0
        IL_0002: bge.s         IL_0006
        IL_0004: br.s          IL_0008
        IL_0006: br.s          IL_0015
        IL_0008: ldstr         "x nie moze byc mniejszy od zera"
        IL_000d: stloc.0       // V_0
        IL_000e: ldloc.0       // V_0
        IL_000f: call          class [System.Runtime]System.↪
           Exception [FSharp.Core]Microsoft.FSharp.Core.↪
           Operators::Failure(string)
        IL_0014: throw
        IL_0015: ldarg.0       // x
        IL_0016: brtrue.s      IL_001a
        IL_0018: br.s          IL_001c
        IL_001a: br.s          IL_001e
        IL_001c: ldarg.1       // n
        IL_001d: ret
        IL_001e: ldarg.0       // x
        IL_001f: ldc.i4.1
        IL_0020: sub
        IL_0021: ldarg.0       // x
        IL_0022: ldarg.1       // n
        IL_0023: mul
        IL_0024: starg.s       n
        IL_0026: starg.s       x
        IL_0028: br.s          IL_0000
} // end of method Program::silnia
```

Zwróćmy uwagę, że w tym przypadku nigdzie nie znajdziemy, instrukcji pozwalającej na rekurencyjne wywołanie funkcji Silnia. Jak w takim przypadku działa nasza funkcja rekurencyjna, skoro nie ma wywołania rekurencyjnego? Wszystko wyjaśni się jeżeli na podstawie kodu języka pośredniego odtworzymy kod C#[5]:

```
public static int silniaAcc(int x, int n)
{
  int num;
  for (; x >= 0; x = num)
  {
    if (x == 0)
      return n;
    num = x - 1;
    n = x * n;
  }
  throw Operators.Failure("nieprawidlowe dane wejsciowe");
}
```

Czyli nasza funkcja rekurencyjna została przez kompilator zamieniona na funkcję wykorzystującą bardzo efektywną pętlę. A to wszystko dzięki wykorzystaniu rekurencji

[5]tak, oprogramowanie DotPeek również to potrafi

ogonowej.

1.8 Jednostki miary

Język F# został utworzony jako język do analizy i przetwarzania danych. Jednak domyślny zestaw typów numerycznych nie jest zbyt odpowiedni do tego typu zastosowań. Stwórzmy prostą funkcję:

```
let predkosc (dystans:float) (czas:float) = dystans/czas
```

Funkcja ta ma obliczyć prędkość na podstawie przebytego dystansu oraz czasu w jakim to nastąpiło, zarówno dystans jak i czas zdefiniowane są jako wartości typu `float`.

Zdefiniujmy następnie kilka zmiennych:

```
let objetosc = 100.0
let odleglosc = 100.0
let czas = 1.0
```

i wywołajmy naszą funkcję w celu wyznaczenia prędkości:

```
let uzyskanaPredkosc = predkosc odleglosc czas
```

Oczywiście na pierwszy rzut oka, wszystko wydaje się w porządku, ale np. fizyk zapytałby w tym momencie jakich jednostkach wyrażony jest dystans, czas i prędkość. Możemy powiedzieć, że odległość wyrażona jest w kilometrach, czas w godzinach, no a prędkość w kilometrach na godzinę, ale z naszego programu to w żaden sposób nie wynika.

Co gorsza, ponieważ zarówno parametry i zmienne są zdefiniowane jak wartości typu `float` nic nie może nas powstrzymać przez następującymi, nie mającymi większego sensu, wywołaniami funkcji:

```
let uzyskanaPredkosc = predkosc czas odleglosc
let uzyskanaPredkosc = predkosc objetosc czas
```

Oczywiście możemy powiedzieć niech się programista pilnuje, ale mimo wszystko nie tego możemy oczekiwać od języka do przetwarzania danych.

Na szczęście F# ma bardzo ciekawy mechanizm, pozwalający rozwiązać powyższe problemy. Są to **jednostki miary**. Pozwalają one na zdefiniowanie nowego typu danych, który dodatkowo musimy opatrzyć atrybutem `Measure`. Przykładowo poniższy kod definiuje trzy jednostki miary: godziny, metry i kilometry:

```
[<Measure>] type h
[<Measure>] type m
[<Measure>] type km
```

Dzięki nim możemy jawnie określić, że parametr `dystans` powinien być liczbą rzeczywistą wyrażającą odległość w kilometrach, a parametr `czas` liczbą rzeczywistą wyrażającą czas w godzinach:

```
let predkosc (dystans:float<km>) (czas:float<h>) = dystans/czas
```

Dzięki temu wiemy w jakich jednostkach powinny być podane poszczególne parametry. Ponadto F# jest w stanie określić automatycznie w jakich jednostkach będzie wyrażony wynik działania funkcji:

```
val predkosc : dystans:float<km> -> czas:float<h>-> float<km/h>
```

To powinno usatysfakcjonować naszego fizyka, ale jak to wpłynie na poprawność programu? Znacząco ją poprawi, ponieważ teraz nie możemy pod parametry funkcji podstawić zwykłych wartości typu `float`. Wykonanie kodu:

```
predkosc odleglosc czas
```

zwróci błąd error FS0001: This expression was expected to have type 'float<km>' but here has type 'float', ponieważ wartości przekazane do funkcji muszą być wyrażone w tych samych jednostkach co jej parametry. Wobec tego zmienne odległość i czas powinny być teraz zdefiniowane jako:

```
let odleglosc = 100.0<km>
let czas = 1.0<h>
```

A co jeżeli odległość będziemy mieli wyrażoną dajmy na to w metrach? Powinniśmy napisać funkcję konwertującą. Aby to zrobić musimy określić ile metrów możemy umieścić w jednym kilometrze:

```
let metryNaKilometr : float<m*km^-1> = 1000.0<m/km>
```

a następnie wartość tę wykorzystać do stworzenia funkcji zamieniającej odległość w metrach na odległość w kilometrach:

```
let konwertujMetryNaKilometry (odleglosc:float<m>) = odleglosc ↩
    / metryNaKilometr
```

W powyższym przykładzie należy zwrócić uwagę na sposób definiowania jednostek wyrażonych za pomocą ilorazu. Możemy to zrobić albo bezpośrednio `m/km`, albo za pomocą potęgowania `m*km^-1`. W tym drugim przypadku zamiast znaku '*' można wykorzystać spację.

1.9 Kilka przydatnych funkcji dostępnych w F#

Poniżej przedstawię kilka przydanych instrukcji, które mogą być wam potrzebne przy realizacji zadań przedstawionych w ostatniej części tej instrukcji:

1. `open <nazwa modulu lub przestrzeni nazw>` - otwiera moduł lub przestrzeń nazw dzięki czemu nie będziemy musieli odwoływać się do elementów wewnątrz niego za pomocą pełnej nazwy. Przykładowo, jeżeli będziemy chcieli obliczyć obwód koła, musimy skorzystać ze stałej PI zdefiniowanej w klasie `Math` znajdującej się w przestrzeni nazw `System`:

```
let promien = 12.0
let obwod = 2.0*System.Math.PI*promien
```

Jeżeli niewiele razy w aplikacji będziemy się odnosili do elementów zdefiniowanych w przestrzeni System, to taki zapis jest akceptowalny. Jednak, jeżeli takich miejsc będzie dużo wygodniej będzie nam zapisać to w następujący sposób:

```
open System

let promien = 12.0
let obwod = 2.0*Math.PI*promien
```

2. Console.ReadLine() - metoda ta pozwoli nam wczytać jedną linię tekstu z konsoli. Zwraca oczywiście łańcuch znaków. Klasa Console znajduje się w przestrzeni nazw System, wobec tego konieczne będzie otwarcie tej przestrzeni lub podawanie pełnej nazwy.
3. Console.Write(<lancuch znakow>), Console.WriteLine(<lancuch znakow>) - metody te pozwalają na wyświetlenie wiadomości w standardowym wyjściu (konsola lub interpreter). Metoda WriteLine dodatkowo przenosi kursor do nowej linii.
4. printf,printfn - podobnie jak metody Console.Write oraz Console.WriteLine metody te służą do wyświetlenia komunikatu na standardowym wyjściu. Są one jednak wygodniejsze w użyciu. Przykładowo program:

```
let x = 12.0
let wynik = sin x
System.Console.WriteLine($"śąWartoci funkcji sin dla argumentu {x} jest {wynik}")
```

z wykorzystaniem funkcji printfn zapiszemy następująco:

```
let x = 12.0
let wynik = sin x
printfn "Wartoscia funkcji sin dla argumentu %f jest %f" x wynik
```

Wartości zmiennych x oraz wynik będą podstawione w miejsce "%f". Zapis "%f" określa typ i sposób wyświetlenia wartości i może m.in. przyjmować jedną z wartości:

Specyfikacja formatu	typ
%b	bool
%s	string
%c	char
%d,%i	int (jako wartość ze znakiem)
%u	int (jako wartość bez znaku)
%x,%X	int (jako wartość szesnastkowa)

Co ciekawe funkcje te pozwalają sprawdzanie typów danych np. jeżeli zapiszemy:

```
printfn "Wartosc %d" (sin 12.0)
```

uzyskamy następujący komunikat o błędzie: error FS0001: The type 'float' is not compatible with any of the types byte,int16,int32,int64,sbyte,uint16,uint32,uint64, nativeint,unativeint, arising from the use of a printf-style format string
Funkcja `printfn` dodatkowo przenosi kursor do nowej linii.

5. `sprintf` - czasami zamiast wyświetlać komunikat w konsoli będziemy chcieli go zapisać do zmiennej. Funkcja `sprintf` nam to umożliwia zwracając odpowiednio sformatowany tekst w formie łańcucha znaków. W inny względzie działa dokładnie tak samo jak `printf`.
6. `failwith` - funkcja służąca do generowania wyjątku z komunikatem podanym jako parametr.
7. Generowanie liczb losowych - aby skorzystać w F# z generatora liczb losowych należy najpierw stworzyć obiekt klasy `Random`, a następnie wywołać jedną z jego metod `Next()`, `Next(min, max)` lub `NextDouble()`. Ta ostatnia pozwala na generowanie wartości z przedziału 0,1. Przykładowo, jeżeli chcemy wygenerować wartości z przedziału do 10 do 20 możemy to zrobić następująco:

```
let rand = new Random()
rand.Next(10, 21)
```

1.10 Tworzenie aplikacji w F#

W tej części przedstawię wam podstawo narzędzia, w których możecie tworzyć aplikacje w F# .

1.10.1 Visual Studio

Ponieważ F# jest dziełem firmy Microsoft podstawowym narzędziem, w którym możemy tworzyć aplikacje w tym języku jest środowisko Visual Studio, aktulanie dostępne w wersji 2019. Do naszych zajęć w zupełności wystarczy darmowa edycja Community.

W celu tworzenia nowego projektu, uruchamiamy Visual Studio, po czym powinniśmy zobaczyć następujące okno:

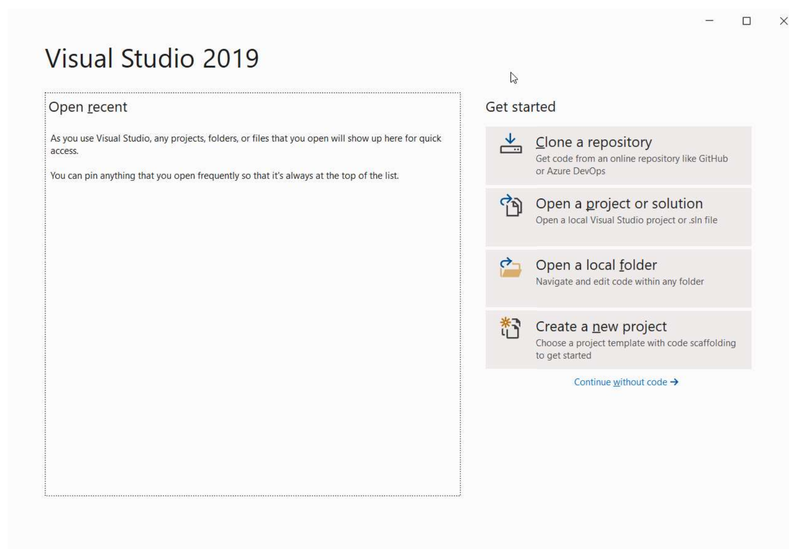
Możemy w nim stworzyć nowy projekt lub otworzyć już istniejący (wybierając go z dysku, lub listy ostatnio otwartych projektów). Okno to możemy również otworzyć wybierając opcję Project z menu File>New.

Ponieważ nie mamy jeszcze żadnego projektu – musimy utworzyć nowy, dlatego wybieramy opcję "Create a new project". Spowoduje to wyświetlenie następującego okna tworzenia nowego projektu:

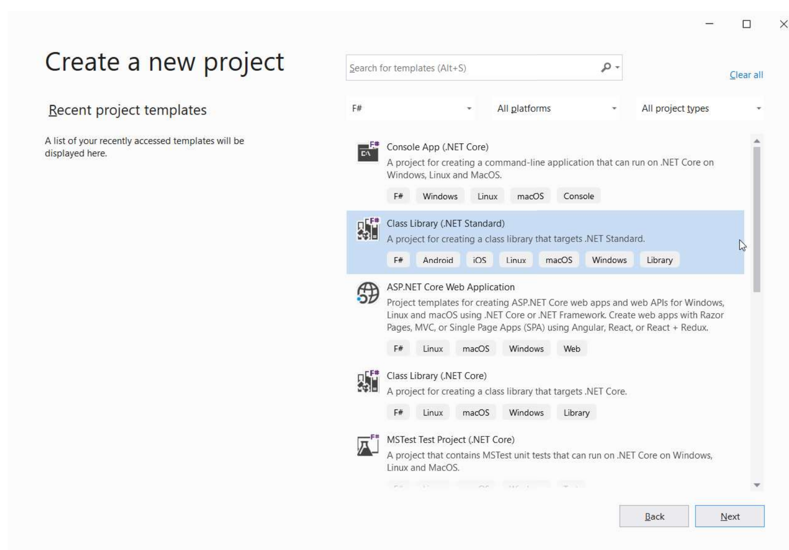
Możemy w nim wybrać język, w którym chcemy tworzyć aplikację oraz jej typ. W przypadku aplikacji dla języka F# dostępne jest tworzenie m.in. aplikacji konsolowych (Console Application), aplikacji internetowych (ASP.NET Core Web Application), biblioteki DLL (Class Library) oraz Tutorial. Ostatni z wymienionych jest prostym samouczkiem pokazującym najważniejsze cechy jezyka F# . Zwróćmy uwagę, że niektóre z tych szablonów sa dostępne dla .NET Framework jak również dla platformy .NET Core.

My wybierzemy szablon Console App (.NET Core) i przejdziemy do następującego okna konfiguracji nowego projektu:

W oknie tym podajemy nazwę naszego projektu, katalog w którym ma on być umieszczony oraz nazwę tworzonego rozwiązania. W Visual Studio Rozwiązaniem nazywamy



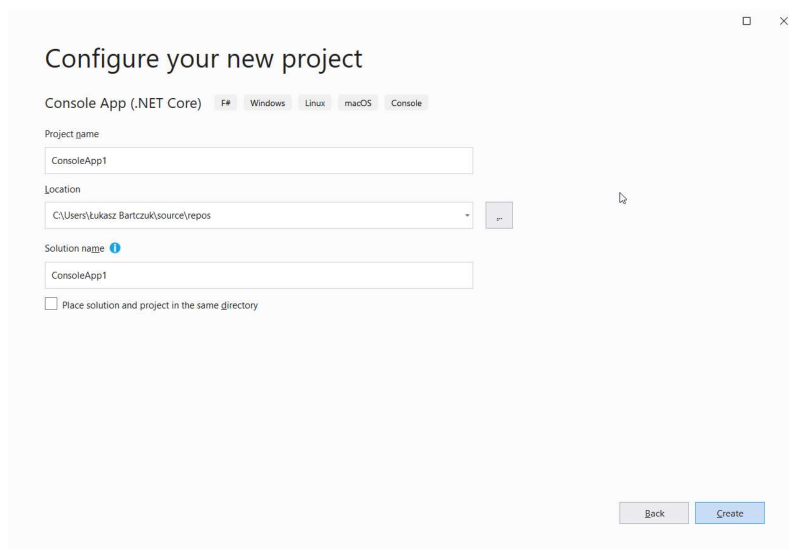
Rysunek 1.3: Okno tworzenia nowego lub otwarcia istniejącego projektu Visual Studio



cały zestaw projektów (to może dowolny rodzaj projektu, przygotowany w dowolnym języku wspierwanym przez VS), które łącznie służą do rozwiązania danego problemu programistycznego. Po podaniu wszystkich wymaganych informacji, przyciskamy przycisk ”Create” i czekamy na utworzenie projektu.

Utworzony projekt jest bardzo prostą, aczkolwiek w pełni działającą aplikacją, wyświetlającą w konsoli komunikat "Hello World from F# !":

```
// Learn more about F# at http://fsharp.org

open System

[<EntryPoint>]
let main argv =
    printfn "Hello World from F#!"
```




```
    0 // return an integer exit code
```

Cały program składa się z jednej funkcji main, która stanowi punkt wejścia do aplikacji (jest to pierwsza i w tym konkretnym przypadku jedyna funkcja uruchamiana w aplikacji) [6].

Aby ją uruchomić, należy wcisnąć przycisk F5 (uruchamianie w trybie debugowania) lub F6 (uruchomienie bez debugowania).

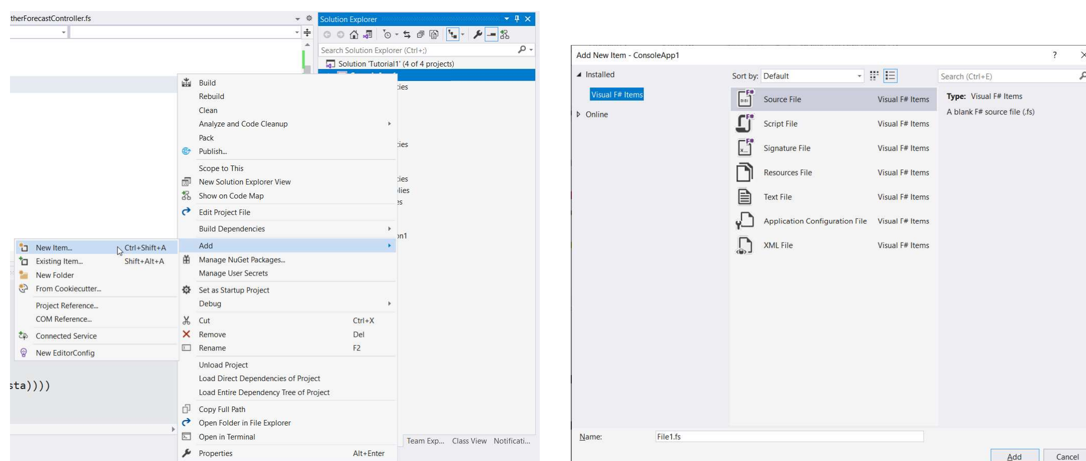
Utworzenie aplikacji składającej się z jednego pliku nie stanowi w F# żadnego problemu funkcje i inne elementy aplikacji możemy wpisywać bezpośrednio do tego pliku. Musimy jednak pamiętać, że dowolny symbol musi być zdefniowany przed jego wykorzystaniem. Przykładowo:

```
    // Learn more about F# at http://fsharp.org

open System

let wyswietlKomunikat () = printfn "Ala ma kota"

[<EntryPoint>]
let main argv =
    printfn "Hello World from F#!"
    wyswietlKomunikat ()

    0 // return an integer exit code
```

Sprawa się jednak trochę komplikuje w przypadku, gdy mamy aplikację składającą się z kilku plików. Dodatkowe pliki możemy dołączać do aplikacji np. poprzez kliknięcie prawym przyciskiem myszy na nazwie projektu w oknie Solution Explorer i wybranie z menu

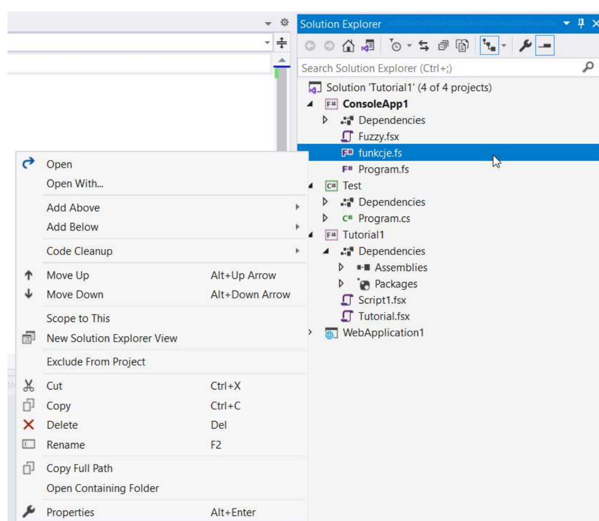
[6]Nazwa main jest zwyczajowa i pozostała jeszcze po języku C++. W F# punkt wejścia do aplikacji może mieć dowolną nazwę. Istotne jest aby funkcja ta była oznaczona atrybutem `[<EntryPoint>]`

kontekstowego opcji Add/New Item. W wyświetlonym oknie dialogowym wybieramy opcję ”Source File”, a w okienku ”Name” podajemy nazwę nowego pliku:



Rysunek 1.4: Dodawanie nowego pliku źródłowego do aplikacji F#

Musimy jednak pamiętać, że pliki aplikacji F# w Visual Studio są kompilowane w kolejności ich występowania w oknie Solution Explorer. Dlatego jeżeli mamy dwa pliki np. "program.fsóraz "funkcje.fs"i w pierwszym pliku chcemy się odwołać do funkcji zdefiniowanych w drugim muszą one występować w kolejności: najpierw "funkcje.fs", potem "program.fs". Plik zawierający punkt wejścia do aplikacji musi być ostatnim plikiem w projekcie. Aby ułatwić zarządzanie projektem w menu kontekstowym okna Solution Explorer możemy przesuwać pliki w górę lub w dół (oczywiście aby wyświetlić to menu musimy kliknąć prawym przyciskiem myszy, na nazwie pliku, którego pozycję chcemy zmienić). Zwróćmy uwagę, że w tym menu występują również opcje umożliwiające doda-



Rysunek 1.5: Menu kontekstowe umożliwiające zarządzenie projektem aplikacji F#

wanie nowego elementu aplikacji przed (”Add Above”) lub po (”Add Below”) wybranym pliku.

W przypadku tworzenia aplikacji składającej się z wielu plików należy pamiętać o jeszcze jednym bardzo ważnym fakcie. Każdy plik musi rozpoczynać się od deklaracji modułu lub przestrzeni nazw [7]. Jeżeli aplikację tworzymy w Visual Studio i nowy plik do aplikacji dodaliśmy w sposób opisany powyżej, to odpowiednia deklaracja jest automatycznie dodawana przez środowisko. Przykładowo:

```
module Funkcje

let wyswietlKomunikat () = printfn "Ala ma kota"
```

Listing 1.1: Zawartość pliku funkcje.fs

Jeżeli następnie do takiej funkcji będziemy chcieli odnieść się w innym pliku np. ”Program.fs”, to taki moduł musimy najpierw otworzyć za pomocą instrukcji `open`:

```
// Learn more about F# at http://fsharp.org
open System
open Funkcje

[<EntryPoint>]
let wejscie argv =

    printfn "Hello World from F#!"
    wyswietlKomunikat ()

    0 // return an integer exit code
```

Listing 1.2: Zawartość pliku Program.fs

Głównym zadaniem modułów jest porządkowanie naszego kodu oraz dbanie o to, aby w aplikacji nie występowały konflikty nazw. Może się jednak zdarzyć, że różne moduły, z których musimy skorzystać, będą zawierały np. funkcje o takich samych nazwach. W takim przypadku wywołana zostanie funkcja z ostatniego otwartego modułu. Jeżeli jednak, będziemy potrzebowali odwołać się do funkcji w innym module, możemy to zrobić za pomocą jej pełnej nazwy składającej się z nazwy modułu oraz funkcji np.

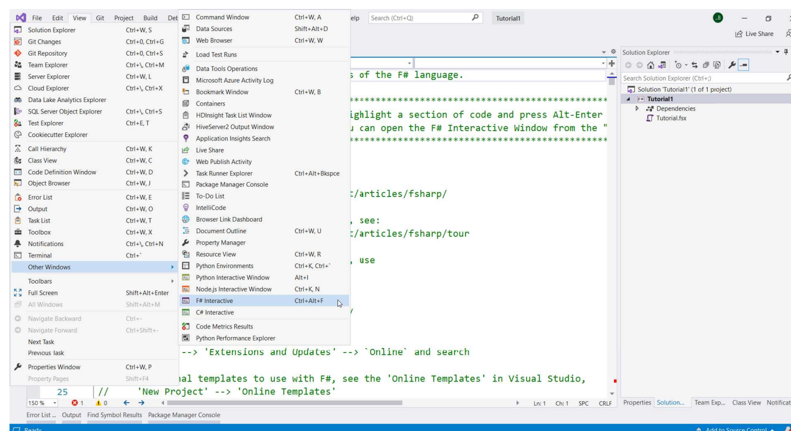
```
// Learn more about F# at http://fsharp.org
open System

[<EntryPoint>]
let wejscie argv =

    printfn "Hello World from F#!"
    Funkcje.wyswietlKomunikat ()

    0 // return an integer exit code
```

Listing 1.3: Zawartość pliku Program.fs

[7] Nazwy modułów oraz przestrzeni nazw zwyczajowo podajemy w notacji Pascal, tzn. Pierwsza litera każdego wyrazu w nazwie jest duża

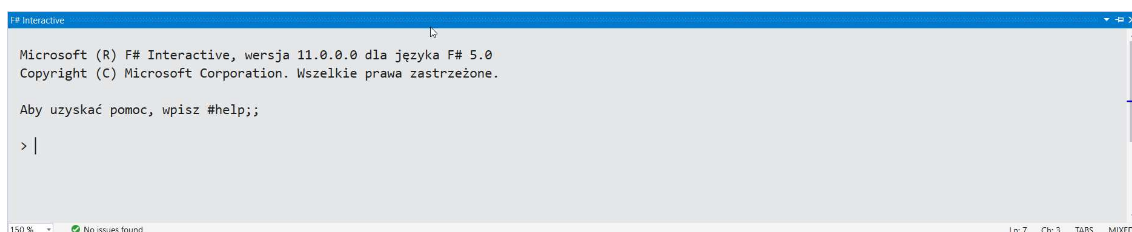
1.10.2 F# Interactive

Visual Studio jest wyposażone w bardzo ciekawe narzędzie o nazwie F# Interactive. Jest to interpreter języka F# działający w formie aplikacji REPL (Read Eval Print Loop). Czyli możemy w nim wydawać komendy języka F# , które natychmiast zostaną wykonane. Jest to bardzo wygodne narzędzie, które można wykorzystać do sprawdzenia sposobu działania danego fragmentu kodu, bez potrzeby kompilowania i uruchamiania całej aplikacji. Można je otworzyć z menu View>Other Windows>F# Interactive:



Rysunek 1.6: Wyświetlenie okna F# Interactive

Spowoduje to wyświetlenie w dolnej części ekranu okna:



Rysunek 1.7: Okna F# Interactive

w którym możemy podawać instrukcje języka F# . Większość instrukcji F# będą składały się z wielu linii, dlatego klawisz ”Enter” spowoduje w tym przypadku przejście do następnej linii kodu. Aby wpisany kod został przetworzony przez interpreter należy zakończyć go podwójnym średnikiem ’;;’.

Jeżeli kod funkcji mamy wpisany do pliku aplikacji, nie musimy go przepisywać do interpretera. Wystarczy go zaznaczyć (albo za pomocą myszy, albo strzałkami z wciśniętym klawiszem SHIFT) i nacisnąć kombinację klawiszy LEWY ALT+ENTER.

Każda, nawet wieloliniowa instrukcja wprowadzona do interpretera jest traktowana podczas wykonania jako jedna całość. Niestety od strony edycyjnej każda linijka kodu jest osobną jednostką. Co powoduje, że jeżeli chcemy coś w tym kodzie zmienić konieczne jest jego ponowne wprowadzanie. Jednocześnie najczęściej nie chcemy, aby kod wpisywany do interpretera stanowił część aplikacji. Ponadto, w oknie tym nie możemy korzystać z podpowiedzi IntelliSense.

Aby te dwa wymagania pogodzić w Visual Studio wprowadzono nowy rodzaj pliku dla aplikacji F# – Script File. Ma on rozszerzenie .fsx i jego zawartość może być wykonana tylko w interpreterze (pliki te nie są uwzględniane podczas kompilowania aplikacji). Możemy w nim wprowadzać dowolną zawartość F# i dowolnie ją edytować oraz korzystać w pełni z podpowiedzi IntelliSense. Jednocześnie zawarty w skrypcie kod możemy uruchomić tylko poprzez jego zaznaczenie i wysłanie do interpretera, poprzez wciśnięcie kombinacji klawiszy LEWY ALT+ENTER.

1.10.3 Visual Studio Core

Jeżeli z jakichś względów ktoś nie chce lub nie ma możliwości korzystać z Visual Studio, aplikacje F# może stworzyć w dowolnym edytorze. W tym celu należy zainstalować platformę .NET Core. Wraz z platformą zainstalowane zostaną wymagane narzędzia, przede wszystkim polecenie dotnet, które służy m.in. do utworzenia, kompilowania oraz uruchomienia aplikacji.

W celu utworzenia aplikacji należy:

1. utworzyć dla niej katalog
2. uruchomić konsolę
3. przejść do utworzonego katalogu
4. wydać polecenie `dotnew new console -lang F#`

Opcja console określa typ aplikacji, którą chcemy utworzyć. Oprócz niej mamy jeszcze do dyspozycji następujące rodzaje aplikacji:

Szablon	Nazwa
Console Application	console
Class library	classlib
Worker Service	worker
Unit Test Project	mstest
NUnit 3 Test Project	nunit
NUnit 3 Test Item	nunit-test
xUnit Test Project	xunit
ASP.NET Core Empty	web
ASP.NET Core Web App (Model-View-Controller)	mvc
ASP.NET Core Web API	webapi

Wykorzystanie opcji -lang jest w tym przypadku obowiązkowe, ponieważ domyślnie wszystkie projekty są tworzone w języku C#.

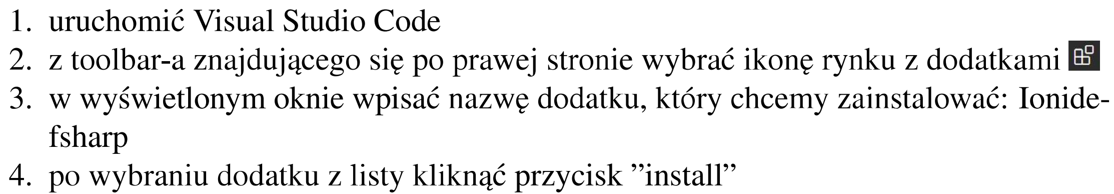
Budowanie projektu odbywa się za pomocą polecenia `dotnet build`, a jego uruchomienie za pomocą `dotnet run`. Oczywiście oba te polecenia można wydać będąc w katalogu aplikacji (tym w którym znajduje się plik projektu z rozszerzeniem .fsproj).

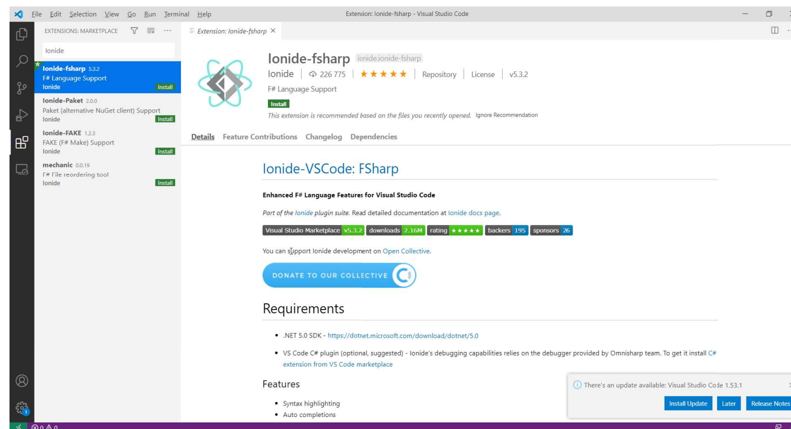
Dodatkowo z poziomu konsoli, za pomocą polecenia `dotnet fsi` możemy również uruchomić interpreter F# .

Mamy teraz możliwość tworzenia aplikacji F# nawet w Notatniku. Oczywiście nie jest to zbyt wygodne narzędzie. Zdecydowanie lepszym wyborem jest np. Visual Studio Code. Jest to w pełni darmowy edytor kodu. Można go uruchomić zarówno w systemie Windows, jak również w Linux-ie oraz na macOS. Dzięki mechanizmowi rozszerzeń możemy w tym narzędziu tworzyć aplikacje ale praktycznie w dowolnym języku programowania, nie tylko wspieranych przez Microsoft.

W celu wygodnego korzystania z Visual Studio Code podczas tworzenia aplikacji F# , konieczne jest zainstalowanie dodatku Ionide-fsharp. Aby to zrobić należy:

1. uruchomić Visual Studio Code
2. z toolbar-a znajdującego się po prawej stronie wybrać ikonę rynku z dodatkami
3. w wyświetlonym oknie wpisać nazwę dodatku, który chcemy zainstalować: Ionide-fsharp
4. po wybraniu dodatku z listy kliknąć przycisk ”install”

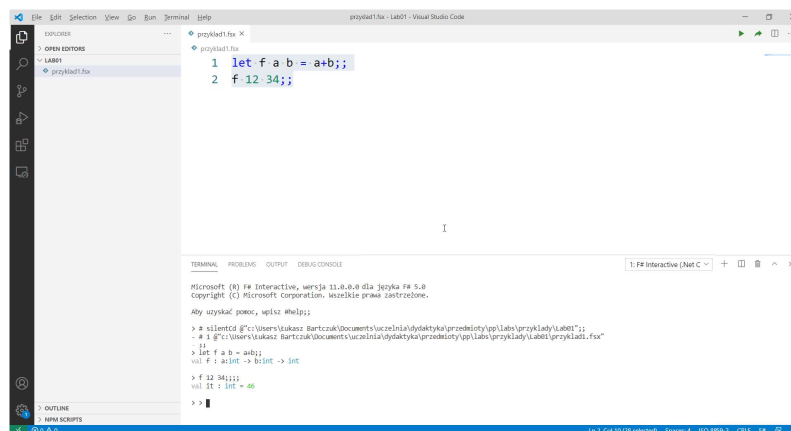




Rysunek 1.8: Instalacja dodatku Ionide-fsharp w Visual Studio Code

Możemy teraz tworzyć aplikacje F# , jednak nadal musimy ręcznie wpisywać komendy polecenia `dotnet`. Możemy to zrobić poprzez wbudowany w Visual Studio Code terminal, który uruchamiamy z menu View>Terminal lub Terminal>New Terminal.

Dodatek Ionide-fsharp daje nam dostęp do podświetlania składni F# , podpowiedzi IntelliSense oraz interpretera F# (wystarczy zaznaczyć fragment, który chcemy wysłać do itnerpretera i wcisnąć kombinację klawiszy LEWY ALT+ENTER)



Rysunek 1.9: Interpreter F# w Visual Studio Code

1.10.4 Wykorzystnie narzędzi online

Jeżeli ktoś nie chce instalować platformy .NET, do tworzenia aplikacji w F# może skorzystać z jednego z wielu narzędzi dostępnych poprzez przeglądarkę internetową. Takie

interpretery i kompilatory online są dostępne pod adresami np. https://try.fsharp.org/, https://fable.io/repl/, https://repl.it/languages/fsharp.

1.11 Zadania

1.11.1 Tworzenie prostych funkcji

Zadanie 1.1 Napisz funkcję obliczającą pole koła
Zadanie 1.2 Napisz funkcję wyznaczającą wartość pierwiastków równania kwadratowego
Zadanie 1.3 Napisz funkcję, która sprawdzi czy z trzech podanych wartości rzeczywistych, da się zbudować trójkąt.
Zadanie 1.4 Napisz funkcję, która obliczy pole trójkąta, na podstawie długości jego boków. Jeżeli z podanych wartości nie da się zbudować trójkąta rzuć wyjątek z odpowiednią wiadomością.

1.11.2 Rekurencja

Zadanie 1.5 Napisz funkcję, która rekurencyjnie oblicza sumę n pierwszych liczb naturalnych
Zadanie 1.6 Napisz funkcję, która rekurencyjnie wyznaczy wartość x^n, gdzie x i n są dowolnymi liczbami naturalnymi.
Zadanie 1.7 Napisz funkcję, wyznaczającą wartość n-tego elementu ciągu fibbonaciego
Zadanie 1.8 Napisz funkcję, która dla dowolnych liczb całkowitych n i k obliczy wartość dwumianu Newtona: $\binom{n}{k=\frac{n!}{k!(n-k)!}}$ zdefiniowanego w sposób rekurencyjny:

$$\binom{n}{k} = \begin{cases} 1 & \text{dla } k = 0 \\ 1 & \text{dla } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{dla } k \neq 0 in \neq k \end{cases}$$

Zadanie 1.9 Napisz funkcję, która rekurencyjnie sprawdza, czy dana liczba jest liczbą pierwszą.
Zadanie 1.10 Napisz funkcję, która na podstawie 1000 rzutów kostką do gry określi prawdopodobieństwo wyrzucenia szóstki
Zadanie 1.11 Napisz funkcję, która na podstawie 1000 rzutów dwiema kostkami do gry określi prawdopodobieństwo wyrzucenia dwóch szóstek
Zadanie 1.12 Napisz funkcję określającą największy wspólny dzielnik
Zadanie 1.13 Oblicz przybliżoną wartość szeregu nieskończonego. Zatrzymaj obliczenia kiedy wartość bezwzględna kolejnego elementu w szeregu będzie mniejsza niż ustalona dokładność e (np. $e = 10^{-7}$):

1. $\sum_{i=1}^{\infty} \frac{1}{i^2}$
2. $\sum_{i=1}^{\infty} \frac{(-1)^i}{i!}$
3. $\sum_{i=1}^{\infty} \frac{1}{i(i+1)}$
4. $\sum_{i=1}^{\infty} \frac{(-2)^i}{i!}$

Zadanie 1.14 Napisz powyższe funkcje z wykorzystaniem rekurencji ogonowej

1.11.3 Jednosteki miary

Zadanie 1.15 Napisz funkcję zamieniającą termperaturę wyrażoną w stopniach Celsjusza na temperaturę wyrażoną w stopniach Fahrenheita zgodnie z poniższym wzorem:

$$T_{Fahrenheit} = 32 + \frac{9}{5} \cdot T_{Celsius}$$

Zdefiniuj wymagane jednostki miary oraz odpowiedni konwerter.
Zadanie 1.16 Napisz funkcję dokonującą odwrotnej transformacji.

1.11.4 Łańcuchy znaków

Zadanie 1.17 Napisz funkcję, która sprawdzi czy dane słowo jest palindromem
Zadanie 1.18 Napisz funkcję, która policzy ile razy w podanym tekście wystąpił określony znak
Zadanie 1.19 Napisz funkcję, która określi liczbę wyrazów w podanym tekście
Zadanie 1.20 Napisz funkcję, która określi liczbę cyfr występujących pod rząd w tekście.

1.11.5 Tworzenie aplikacji

Zadanie 1.21 Napisz aplikację konsolową, który pozwoli użytkownikowi na wprowadzenie imienia i nazwiska i wyświetli komunikat "Witaj <imie i nazwisko>"
Zadanie 1.22 Napisz aplikację konsolową, który pozwoli użytkownikowi na wprowadzenie roku. W odpowiedzi program powinien wyświetlić informacje czy jest to rok przestępny czy nie.
Zadanie 1.23 Napisz aplikację konsolową, która na podstawie podanych użytkownika 3 liczb rzeczywistych wyświetli na ekranie komunikat, czy da się z nich utworzyć trójkąt równoboczny, równoramienny, prostokątny, dowolny inny lub że z podanych wartości nie da się utworzyć trójkąta.
Zadanie 1.24 Napisz aplikację konsolową, która przyjmie od użytkownika ciąg 11 cyfr. Sprawdź, czy cyfry te tworzą poprawny numer PESEL, oraz na jego podstawie wyświetl informacje o dacie urodzenia danej osoby oraz czy jest to kobieta czy mężczyzna.
Zadanie 1.25 Napisz aplikację konsolową, która przyjmie od użytkownika dowolny tekst, i wyświetli go zakodowanego zgodnie z szyfrem Cezara
Zadanie 1.26 Napisz aplikację konsolową, która przyjmie od użytkownika dowolny tekst zakodowany zgodnie z szyfrem Cezara, i odkoduje go.
Zadanie 1.27 Napisz aplikację konsolową, która wczytuje liczbę całkowitą określającą liczbę minut od północy. Program powinien wyświetlać w odpowiedzi konkretną godzinę, która będzie odpowiadać tej liczbie. Jeżeli podana liczba godzin będzie przekraczała liczbę godzin wciągu doby na ekranie powinien pojawić się stosowny komunikat
Zadanie 1.28 Napisz aplikację konsolową, która wczytuje liczbę całkowitą określającą liczbę minut do startu. Program w odpowiedzi powinien wyświetlać komunikaty: "do startu pozostało ... minut". Jeżeli odliczanie dojdzie do zera powinien wyświetlić się odpowiedni komunikat. Odpowiednią funkcję proszę umieścić w osobnym pliku.
Zadanie 1.29 Napisz aplikację konsolową, która rozwiązuje problem Collatza. Program powinien wczytywać od użytkownika liczbę całkowitą dodatnią. Na jej podstawie obliczamy kolejne wartości:

$$C_{n+1} = \begin{cases} 0.5 * C_n & \text{jeżeli } C_n \text{ jest parzysta} \\ 3 * C_n + 1 & \text{w przeciwnym przypadku} \end{cases}$$

gdzie C_0 jest wartością wprowadzoną przez użytkownika. Przypuszcza się, że od nie ważne od jakiej liczby zaczniemy, to osiągniemy wartość 0.

Napisz program, który wyświetli kolejne wyrazy tego ciągu wraz z ich numerem. Np. jeżeli zaczniemy od wartości 5 na ekranie powinniśmy zobaczyć:

1. 5
2. 16
3. 8
4. 4
5. 3
6. 1

Zadanie 1.30 Napisz aplikację konsolową, która pozwoli na wczytywanie z klawiatury liczb całkowitych. Oblicz średnią tych liczb. Użytkownik powinien wprowadzać wartości dopóki nie poda wartości ujemnej. Odpowiednią funkcję proszę umieścić w osobnym pliku.