LABORATORIUM 5

TREŚCI KSZTAŁCENIA:

- DEFINIOWANIE FUNKCJI REKURENCYJNYCH
- TWORZENIE FUNKCJI ITERACYJNYCH

Spis treści

W języku F#, funkcje iteracyjne i rekurencyjne służą do wykonywania operacji wielokrotnie, ale różnią się podejściem do przetwarzania. Oto informacje na temat definiowania obu typów funkcji, ich wad i zalet.

Funkcje Iteracyjne w F#

Funkcje iteracyjne to funkcje, które wykonują powtarzalne zadania za pomocą pętli (np. for, while). W F# można używać pętli, ale ze względu na funkcyjny charakter języka, pętle są rzadziej spotykane niż w językach imperatywnych.

Przykład funkcji iteracyjnej (sumowanie liczb od 1 do n):

```
let sumIter n =
    let mutable sum = 0
    for i in 1 .. n do
        sum <- sum + i
    sum
```

Zalety funkcji iteracyjnych

- Łatwość zrozumienia: Iteracje są często prostsze do zrozumienia dla osób przyzwyczajonych do programowania imperatywnego.
- Wydajność: Iteracyjne podejście może być bardziej wydajne pamięciowo w porównaniu do rekurencji (zwłaszcza gdy brakuje optymalizacji ogonowej).
- Kontrola nad pętlą: Dzięki możliwości modyfikacji zmiennych stanu, pętle iteracyjne dają pełną kontrolę nad każdym krokiem iteracji.

Wady funkcji iteracyjnych

- Brak czystości funkcyjnej: Iteracje wprowadzają zmienne stanu, co kłóci się z ideą programowania funkcyjnego, gdzie preferowane są funkcje czyste, bez efektów ubocznych.
- Mniejsza ekspresyjność: Wiele zadań, takich jak przetwarzanie kolekcji, jest trudniejszych do zapisania za pomocą iteracji niż z rekurencją.

Funkcje Rekurencyjne w F#

Funkcja rekurencyjna wywołuje sama siebie, aż osiągnie warunek bazowy. W F# można korzystać zarówno z rekurencji ogólnej, jak i zoptymalizowanej rekurencji ogonowej (ang. tail recursion), która jest szczególnie wydajna.

Przykład funkcji rekurencyjnej (sumowanie liczb od 1 do n):

```
let rec sumRec n =
    if n <= 0 then 0
    else n + sumRec (n - 1)
```

Rekurencja ogonowa (tail recursion)

Rekurencja ogonowa to rodzaj rekurencji, w której wywołanie rekurencyjne jest ostatnią operacją w funkcji. F# automatycznie optymalizuje rekurencję ogonową, co zapobiega przepełnieniu stosu i poprawia wydajność.

Przykład funkcji sumującej z rekurencją ogonową:

```
let sumTailRec n =
    let rec aux n acc =
        if n <= 0 then acc
        else aux (n - 1) (acc + n)
    aux n 0
```

Zalety funkcji rekurencyjnych

- Ekspresyjność i czytelność: Rekurencja jest często bardziej naturalnym sposobem wyrażania rozwiązań problemów podzielonych na mniejsze zadania, np. przetwarzanie struktury drzewa.
- Lepsza integracja z F#: Rekurencja (zwłaszcza ogonowa) jest lepiej dopasowana do funkcyjnego stylu F# i pozwala tworzyć funkcje bez zmiennych stanu.
- Optymalizacja ogonowa: F# optymalizuje funkcje rekurencyjne ogonowo, umożliwiając ich stosowanie na dużych danych bez ryzyka przepełnienia stosu.

Wady funkcji rekurencyjnych

- Potencjalne przepełnienie stosu: Rekurencja bez optymalizacji ogonowej może powodować przepełnienie stosu, co prowadzi do błędów.
- Wydajność: Rekurencja ogólna może być wolniejsza niż iteracja w niektórych przypadkach, gdyż każde wywołanie funkcji rekurencyjnej tworzy nową ramkę na stosie.
- Trudność w zrozumieniu: Dla programistów przyzwyczajonych do stylu imperatywnego, rekurencja może być mniej intuicyjna niż iteracja.

W F# rekurencja, zwłaszcza ogonowa, jest zwykle preferowana do rozwiązywania problemów, które można podzielić na mniejsze podzadania. Jednak iteracje nadal mogą być wygodne do prostszych zadań wymagających zmiennych stanu i bez konieczności podziału na mniejsze części. Wybór między iteracją a rekurencją zależy od charakteru problemu, przyzwyczajeń programisty i potrzeb wydajnościowych.

Przydatne funkcje dostępne w f#

F# udostępnia wiele przydatnych funkcji, które ułatwiają przetwarzanie danych, zarządzanie listami, sekwencjami i innymi strukturami danych. Poniżej przedstawiam kilka najczęściej używanych funkcji w F#:

Funkcje dla List

F# oferuje rozbudowany zestaw funkcji do przetwarzania list, które pozwalają na łatwe filtrowanie, mapowanie i manipulację danymi.

- List.map – przekształca każdy element listy według określonej funkcji.
```
let numbers = [1; 2; 3; 4]
let squared = List.map (fun x -> x * x) numbers  // Wynik: [1; 4; 9; 16]
```
- List.filter – filtruje listę, zachowując tylko elementy, które spełniają dany warunek.
```
let evenNumbers = List.filter (fun x -> x % 2 = 0) numbers  // Wynik: [2; 4]
```
- List.fold – stosuje akumulator i przechodzi przez każdy element listy, umożliwiając agregację wartości.
```
let sum = List.fold (+) 0 numbers  // Wynik: 10
```
- List.reduce – podobne do fold, ale bez początkowego akumulatora; stosuje funkcję do par elementów.
```
let product = List.reduce (*) numbers  // Wynik: 24
```
- List.iter – wykonuje daną funkcję dla każdego elementu listy, np. wyświetlanie elementów.
```
List.iter (printfn "%d") numbers
```

Funkcje dla Tablic (Array)

Tablice w F# również mają własny zestaw funkcji, które działają analogicznie do tych w List.

- Array.map – przekształca każdy element tablicy.
- Array.filter – filtruje tablicę.
- Array.fold – przechodzi przez każdy element tablicy, stosując akumulator.
- Array.sort – sortuje elementy tablicy.

Funkcje dla Sekwencji (Seq)

Sekwencje są używane do pracy z dużymi lub nieskończonymi zestawami danych, ponieważ umożliwiają leniwe (lazy) obliczenia.

- Seq.map – przekształca każdy element sekwencji.
- Seq.filter – filtruje sekwencję.
- Seq.take – pobiera pierwsze n elementów sekwencji.
```
let firstThree = Seq.take 3 (Seq.initInfinite id)  // Wynik: [0; 1; 2]
```
- Seq.skip – pomija pierwsze n elementów.
```
let afterThree = Seq.skip 3 (Seq.initInfinite id)  // Wynik: [3; 4; 5; ...]
```
- Seq.concat – łączy wiele sekwencji w jedną.
```
let merged = Seq.concat [[1; 2]; [3; 4]]  // Wynik: [1; 2; 3; 4]
```

Funkcje dla Operacji Funkcyjnych

W F# funkcje pierwszego rzędu (first-class functions) pozwalają na użycie funkcji jako wartości.

- (>>), (<<) – kompozycja funkcji, łącząca funkcje w jedną, np. (f >> g).
- id – funkcja identycznościowa, która zwraca swój argument.
- apply – umożliwia przypisanie argumentu do funkcji. Jest używana głównie w składni operacyjnej.

Funkcje Modułu Option

Option to typ danych w F# stosowany do reprezentowania wartości opcjonalnych (mogących być None lub Some).

- ption.map – stosuje funkcję do wartości w Option tylko, gdy jest Some.

```
let squareSome = Option.map (fun x -> x * x) (Some 3)  // Wynik: Some 9
```

- Option.bind – podobna do map, ale pozwala zwrócić nowy Option.
- Option.defaultValue – zwraca domyślną wartość, gdy Option jest None.

Funkcje Modułu Result

Moduł Result pozwala pracować z wynikami, które mogą być sukcesem lub błędem (Ok lub Error).

- Result.map – stosuje funkcję do wyniku, jeśli jest Ok.
- Result.bind – podobnie jak map, ale pozwala na zwrócenie nowego Result.
- Result.defaultError – pozwala ustawić domyślny wynik błędu.

Funkcje dla Operacji Asynchronicznych

F# obsługuje programowanie asynchroniczne, co pozwala na wykonywanie operacji bez blokowania głównego wątku.

- async { ... } – definiuje obliczenia asynchroniczne.
- Async.Start – uruchamia operację asynchroniczną.
- Async.RunSynchronously – uruchamia operację asynchroniczną w sposób synchroniczny i blokujący.

Zadania do samodzielnego rozwiązania

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

Zadanie 1. **Rekurencyjne generowanie ciągu Fibonacciego**

Napisz funkcję rekurencyjną, która oblicza n-ty wyraz ciągu Fibonacciego. Następnie zoptymalizuj ją, stosując funkcję z ogonową rekurencją.

Zadanie 2. Wyszukiwanie elementu w drzewie binarnym

Zaimplementuj funkcję rekurencyjną do wyszukiwania elementu w drzewie binarnym. Napisz też iteracyjną wersję tej funkcji z użyciem stosu symulowanego za pomocą listy.

Zadanie 3. Generowanie permutacji listy:

Utwórz funkcję rekurencyjną generującą wszystkie możliwe permutacje listy liczb całkowitych.

Zadanie 4. Rekurencyjne rozwiązywanie problemu wież Hanoi

Zaimplementuj funkcję rekurencyjną do rozwiązania problemu wież Hanoi i napisz funkcję iteracyjną, która symuluje ten proces bez użycia stosu.

Zadanie 5. Implementacja algorytmu QuickSort

Napisz funkcję rekurencyjną realizującą algorytm QuickSort i porównaj ją z iteracyjną wersją, w której stos jest symulowany ręcznie.