Дополнительные возможности языка F# (единицы измерения, lazy, active patterns)

Выполнил студент 242 группы

Сокольвяк Сергей

Единицы измерения

- Синтаксис

```
[<Measure>] type unit-name [ = measure ]
```

- Примеры

```
[<Measure>] type cm
[<Measure>] type ml = cm^3
```

```
[<Measure>] type s  //секунды

[<Measure>] type hz = 1 / s
```

Правила записи

```
s s ~ s * s ~ s^2

m / s ~ m * s^-1
```

Как преобразует компилятор

`kg m s^-2` и `m /s s * kg` преобразуются в `kg m /s^2`.

Запись в литералах

```
[<Measure>] type cm

[<Measure>] type miles

[<Measure>] type hour


let speed = 55.0<miles/hour>

let length = 100.0<cm>

let count = 1.0<1>
```

Преобразование типов с единицей измерения

```
let convertcm2meters (x : float<cm>) = x / 100.0<cm/m>
```

Или так

```
let cmPerMeter : float<cm m^-1> = 100.0<cm/m>

let convertCmToMeters (x : float<cm>) = x / cmPerMeter
```

Замечание

Обращайте внимание на функции, которые требуют безразмерные величины, например, printf:

```
let printFunc (x : float<cm>) =
        printf "%f" (x / 1.0<cm>)
```

Преобразование к безразмерному типу

```
let convertHzToDimensionless (x : float<hz>) = x / 1.0<hz>
// let convertHzToDimensionless (x : float<hz>) = float x
```

Преобразование из безразмерного типа к типу с размерностью

```
let convertToMiles (x : float) = x * 1.0<miles>
```

Или так

```
open Microsoft.FSharp.Core
let x = 12.0
let height : float<cm> = LanguagePrimitives.FloatWithMeasure x
```

Generic Units

```
[<Measure>] type km
[<Measure>] type s

let genericSumUnits (x : float<'u>) (y: float<'u>) = x + y

let v1 = 7.9<km/s>
let v2 = 11.2<km/s>
let x1 = 1.2<km>
let t1 = 1.0<s>

// Все нормально, проверка единиц измерения прошла успешно
let result1 = genericSumUnits v1 v2

// Здесь будет ошибка, несоответствие единиц измерения
// let result2 = genericSumUnits v1 x1
```

Aggregate Types with Generic Units

```
type vector3D<[<Measure>] 'u> =
{ x : float<'u>; y : float<'u>; z : float<'u>}

let fstVec : vector3D<m> =
{ x = 1.0<m>; y = 0.0<m>; z = 0.0<m> }

let sndVec : vector3D<m> =
{ x = 0.0<m>; y = 1.0<m>; z = 0.0<m> }

let thdVec : vector3D<m> =
{ x = 1.0<m>; y = 0.0<m>; z = 1.0<m> }

let newVec : vector3D<m/s> =
{ x = 1.0<m/s>; y = -1.0<m/s>; z = 1.0<m/s> }
```

И напоследок

Единицы измерения используются для проверки единиц во время компиляции, но не сохраняются в среде выполнения. Следовательно, они не влияют на производительность. Кроме того, любые попытки реализовать функциональные возможности, основанные на проверке единиц в среде выполнения, окажутся неудачными. Например, невозможно реализовать функцию ToString для распечатки единиц.

```
let v1 = 7.9<km/s>

let printFunc2 (x : float<m/s>) =
       printf "%s" (x.ToString()) // 7.9
```

Lazy Computations

Определение

Отложенные вычисления — это вычисления, которые выполняются не немедленно, а когда фактически требуется результат. Это может помочь повысить производительность кода.

Синтаксис

```
let identifier = lazy ( expression )
```

Вычисление “ленивого” выражения

Чтобы принудительно выполнить вычисления, нужно вызвать метод **Force**.

Пример

```
let x = 10
let result = lazy (x + 10)
let printResult (result : Lazy<int>) =
        printfn "%d" (result.Force())
```

Замечание

Метод **Force** вызывает выполнение вычисления только один раз. Последующие вызовы метода **Force** возвращают **тот же** результат, но при этом не выполняется никакой код.

```
let lazyMul =
    lazy (let value = 10 * 10
          printf "%s" "Value is "
          value)
printfn "%A" (lazyMul.Force()) // Value is 100
printfn "%A" (lazyMul.Force()) // 100
```

Модуль Control.LazyExtensions

На самом деле Lazy.Force<'T> - это один из методов расширения Control.LazyExtensions

Lazy.Create<'T>

Создает отложенное вычисление, которое при принудительном выполнении вычисляет результат заданной функции.

```
let lazyValue n = Lazy.Create (fun () ->
        let rec factorial n =
            match n with
            | 0 | 1 -> 1
            | n -> n * factorial (n - 1)
        factorial n)
let lazyVal = lazyValue 10
printfn "%d" (lazyVal.Force())
```

Больше информации

https://msdn.microsoft.com

Active Patterns

Определение

Активные шаблоны позволяют определять именованные разделы, на которые подразделяются входные данные, благодаря чему эти имена можно использовать в выражении шаблона так же, как при работе с размеченным объединением. Активные шаблоны можно использовать для разложения данных на составные части настраиваемым способом для каждого раздела.

Синтаксис

```
// Полное определение активного шаблона.
let (|identifer1|identifier2|...|) [ arguments ] = expression

// Определение частичного активного шаблона.
let (|identifier|_|) [ arguments ] = expression
```

Идентификаторы — это имена подмножеств набора всех значений аргументов. В определении активного шаблона может быть до семи разделов. *expression* описывает форму компонентов, на которые разбиваются данные. Можно использовать активное определение шаблона, чтобы назначить правила, позволяющие определить, к каким именованным разделам относятся значения, данные как аргументы. Символы (| и |) называются *полукруглыми двойными скобками*, а функция, созданная этим типом привязки let, называется *активным распознавателем*.

Пример

```
type Num =
    | Even
    | Odd

let isEven n =
    match (n % 2) with
    | 0 -> Even
    | _ -> Odd
```

С помощью активных шаблонов

```
let (|Even|Odd|) input =
       if input % 2 = 0 then Even else Odd
```

Пример

```
let (|Even|Odd|) input =
        if input % 2 = 0 then Even else Odd
let TestNumber (input : int) =
    match input with
    | Even -> printfn "%d is even" input
    | Odd -> printfn "%d is odd" input

TestNumber 7  // 7 is odd
TestNumber 11 // 11 is odd
TestNumber 32 // 32 is even
```

Обратите внимание на то, что будет
выведено в F# Interactive

```
val ( |Even|Odd| ) : input:int -> Choice<unit,unit>
```

Использование активных шаблонов для разложения типов данных различными способами

```
open System.Drawing

let (|RGB|) (col : System.Drawing.Color) =
     ( col.R, col.G, col.B )

let (|HSB|) (col : System.Drawing.Color) =
     ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let doSomethingWithRGB (col: System.Drawing.Color) =
    match col with
    | RGB(r, g, b) -> //do something...

let doSomethingWithHSB (col: System.Drawing.Color) =
    match col with
    | HSB(h, s, b) -> //do something else...
```

Зачем это надо?

Активные шаблоны позволяют разделять данные на разделы и компоненты в нужной форме и совершать соответствующие вычисления с нужными данными в форме, наиболее удобной для таких вычислений.

Частичные активные шаблоны

```
let (|Int|_|) str =
    match System.Int32.TryParse(str) with
    | (true, int) -> Some(int)
    | _ -> None

let (|Bool|_|) str =
    match System.Boolean.TryParse(str) with
    | (true, bool) -> Some(bool)
    | _ -> None

let testParse str =
    match str with
    | Int i -> printfn "'%i' is an int" i
    | Bool b -> printfn "'%b' is a bool" b
    | _ -> printfn "The value '%s' is something else" str

testParse "12"      // '12' is an int
testParse "true"    // 'true' is a bool
testParse "abc"     // The value 'abc' is something else
```

Параметризованные активные шаблоны

```
open System.Text.RegularExpressions

let (|FirstRegexGroup|_|) pattern input =
    let m = Regex.Match(input, pattern)
    if (m.Success) then Some m.Groups.[1].Value else None

let testRegex str =
    match str with
    | FirstRegexGroup "http://(.*?)/(.*)" host ->
        printfn "The value is a url and the host is %s" host
    | FirstRegexGroup ".*?@(.*)" host ->
        printfn "The value is an email and the host is %s" host
    | _ -> printfn "The value '%s' is something else" str

testRegex "http://google.com/test" // The value is a url and the host is google.com
testRegex "alice@hotmail.com" // The value is an email and the host is hotmail.com
```

Источники

https://fsharpforfunandprofit.com

https://msdn.microsoft.com