Introduction

Many programming languages provide syntax that allows writing computations for generating sequences, asynchronous programming or for working with monads. They all use different syntax and work with different abstract computation types.

F# computation expressions are a flexible syntactic sugar for writing abstract computations. The library author controls what constructs to use by providing different operations. As a result, they can choose natural syntax for every computation type.

We identify what abstract computations can be encoded using this mechanism and give examples of the most suitable syntax.

Sequence expressions

```
let rec listFiles dir = seq {
    yield! Dir.GetFiles(dir)
    for subdir in Dir.GetDirectories(dir) do
        yield! listFiles(subdir) }
```

Combines monadic and monoidal computation type

- **combine**: Seq a → Seq a → Seq a
- **yield**: a → Seq a
- **for**: Seq a → (a → Seq b) → Seq b

Asynchronous workflows

```
let trafficLight() = async {
    while true do
        for color in [green; orange; red] do
            do! Async.Sleep(1000)
            displayLight(color) }
```

Monad with imperative control flow constructs

- **bind**: Async a → (a → Async b) → Async b
- **for**: [a] → (a → Async 1) → Async 1
- **while**: (1 → bool) → Async 1 → Async 1

Asynchronous sequences

```
let htmlStrings = asyncSeq {
    for url in addressStream do
        let! html = wc.AsyncDownloadString(url)
        do! Async.Sleep(1000)
        displayLight(html)
        yield! html!
}
```

Monad with imperative control flow constructs

- **bind**: Async a → (a → AsyncSeq b) → AsyncSeq b
- **for**: [a] → (a → AsyncSeq b) → AsyncSeq b
- **for**: AsyncSeq a → (a → AsyncSeq b) → AsyncSeq b
- **yield**: a → AsyncSeq a

Non-standard computations in C# and Python

**Async in C# 5 (left)**: Binding using await does not block the running thread and uses continuation passing style.

**Generators in Python (right)**. The `yield` keyword is used to return a sequence of results from a function.

**Haskell do notation**. Syntax for working with monads.

```
def duplicate(inputs):
    for number in inputs:
        yield number
        yield number * 10
    ```

**Unify single-purpose syntactic sugar**

- Unify extensions
- Standard syntax
- Flexible interpretation

**Customize binding and control flow**

- Reuse the standard F# syntax
- Make operation types flexible

**Unify extensions**

- Standard syntax
- Flexible interpretation

**Reinterpret standard F# constructs**

- Reuse the standard F# syntax
- Make operation types flexible

**Make operation types flexible**

- Reuse the standard F# syntax
- Make operation types flexible