Functional Programming in practice ... and more!

RICCARDO TERRELL

"We cannot solve our problems with the same thinking we used when we created them"

- Albert Einstein

Agenda

- The benefits of functional programming
- Domain Specific Language – the power of ADT and functional programming
- DSL in practice
- Real World (Production) example

Objectives

- Multi-paradigm programming is more powerful than polyglot programming
- Bend technonlogies to your needs

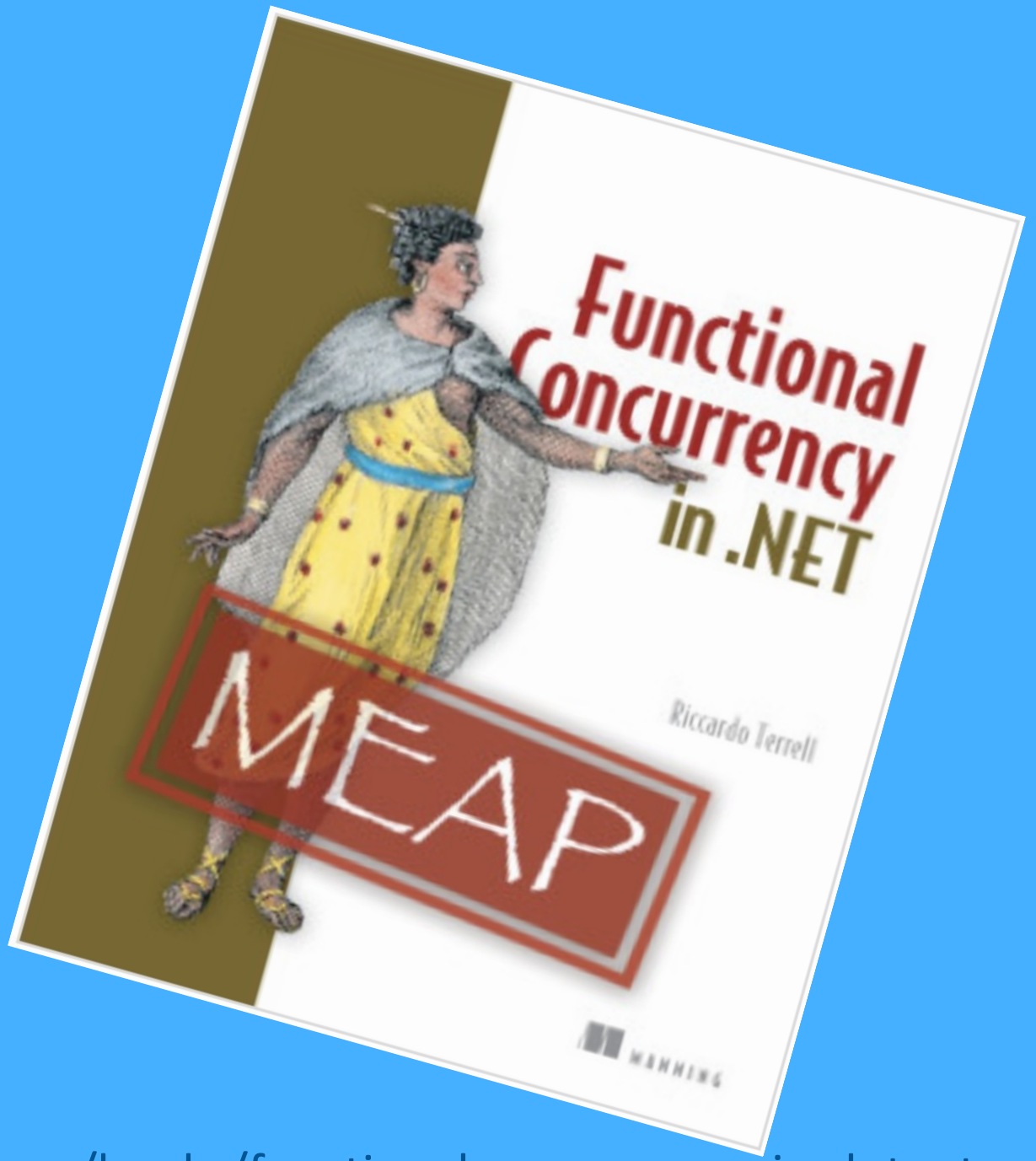
Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC
- +/- 19 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Author of the book “Functional Concurrency in .NET” - Manning
- *Believes in the art of finding the right tool for the job*
- *Organizer of the DC F# User Group*

excella CONSULTING

MVP Microsoft® Most Valuable Professional

@trikace *www.rickyterrell.com* *tericcardo@gmail.com*



https://www.manning.com/books/functional-concurrency-in-dotnet


What about you?

Functional Programming

- declarative
- functions as values
- side-effects free
- referential transparency
- immutable
- composition

Functional programming?

- **Functional programming is a style of programming** that enables you:

  - Re-use code (via **function composition**)

  - Eliminate bugs (via **immutability**)

  - Control (no avoid) **side effects**

… but why? … *why FP?*

There are four primary reasons for FP's newly established popularity:

1. FP offers concurrency/parallelism without tears
2. FP has succinct, concise, understandable syntax
3. FP offers a different programming perspective
4. FP is becoming more accessible

It's really clear that the imperative style of programming has run its course. ... We're sort of done with that. … However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains.

-Anders Hejlsberg
C# Architect

(from his MIX07 keynote)

“I invented the term Object-Oriented,
and I can tell you
I did not have
C++ in mind.”

-Alan Kay

Functional Programming Preachings!

- Avoid Side-Effects!
  - Do not modify variables passed to them
  - Do not modify any global variable
- Avoid Mutation

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??
```

Reasoning about your code

```
int Sum(List<int> values) {

      int sum = 0;
      for(int i = 0;i < arr.Length; i++){
            sum += values[i];
            values[i] = 0;

      }
}
```

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??

int Sum(IEnumerable<int> values) { … }
```

Immutability

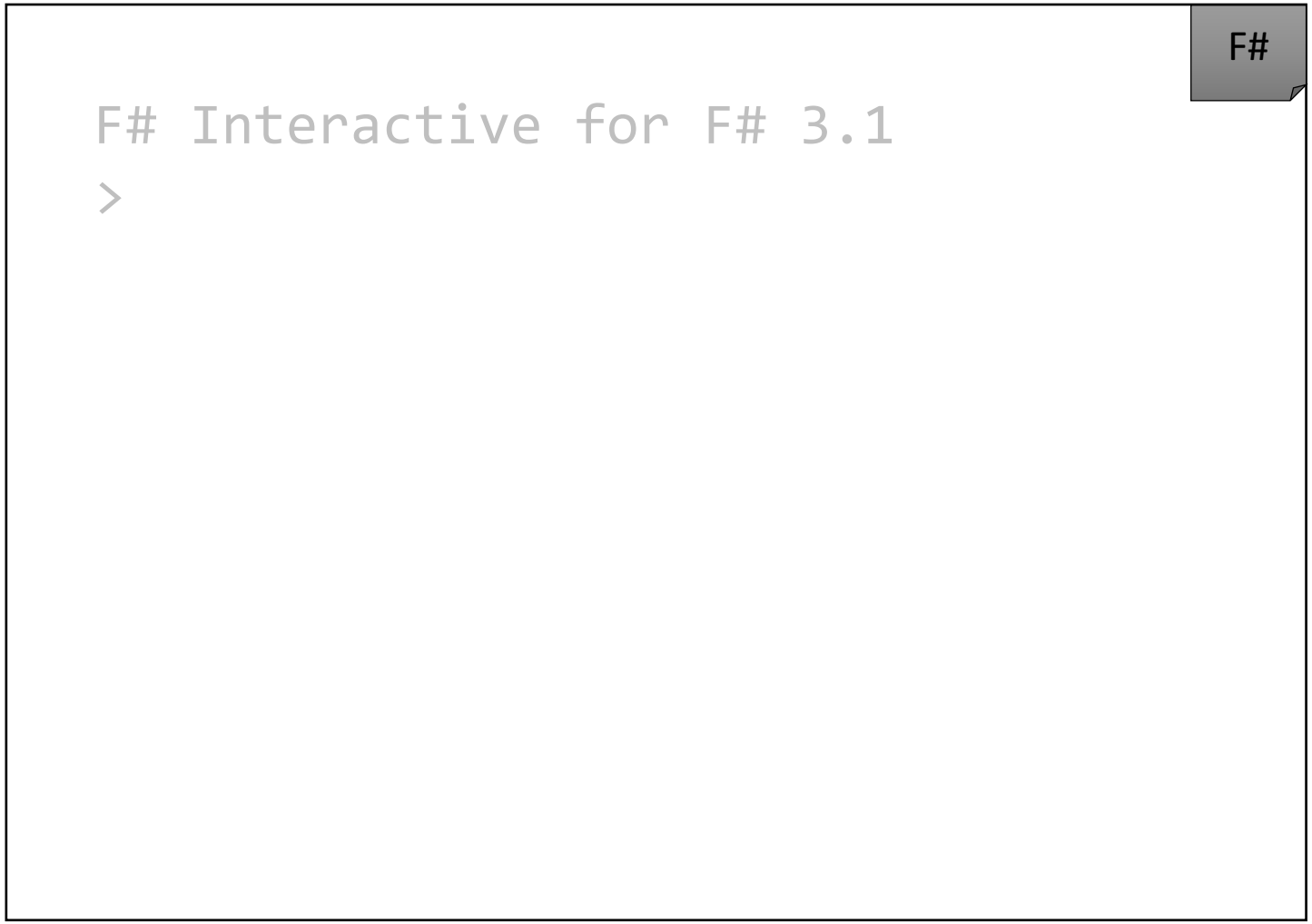
CompSci 101
X = X + 1

Increment!

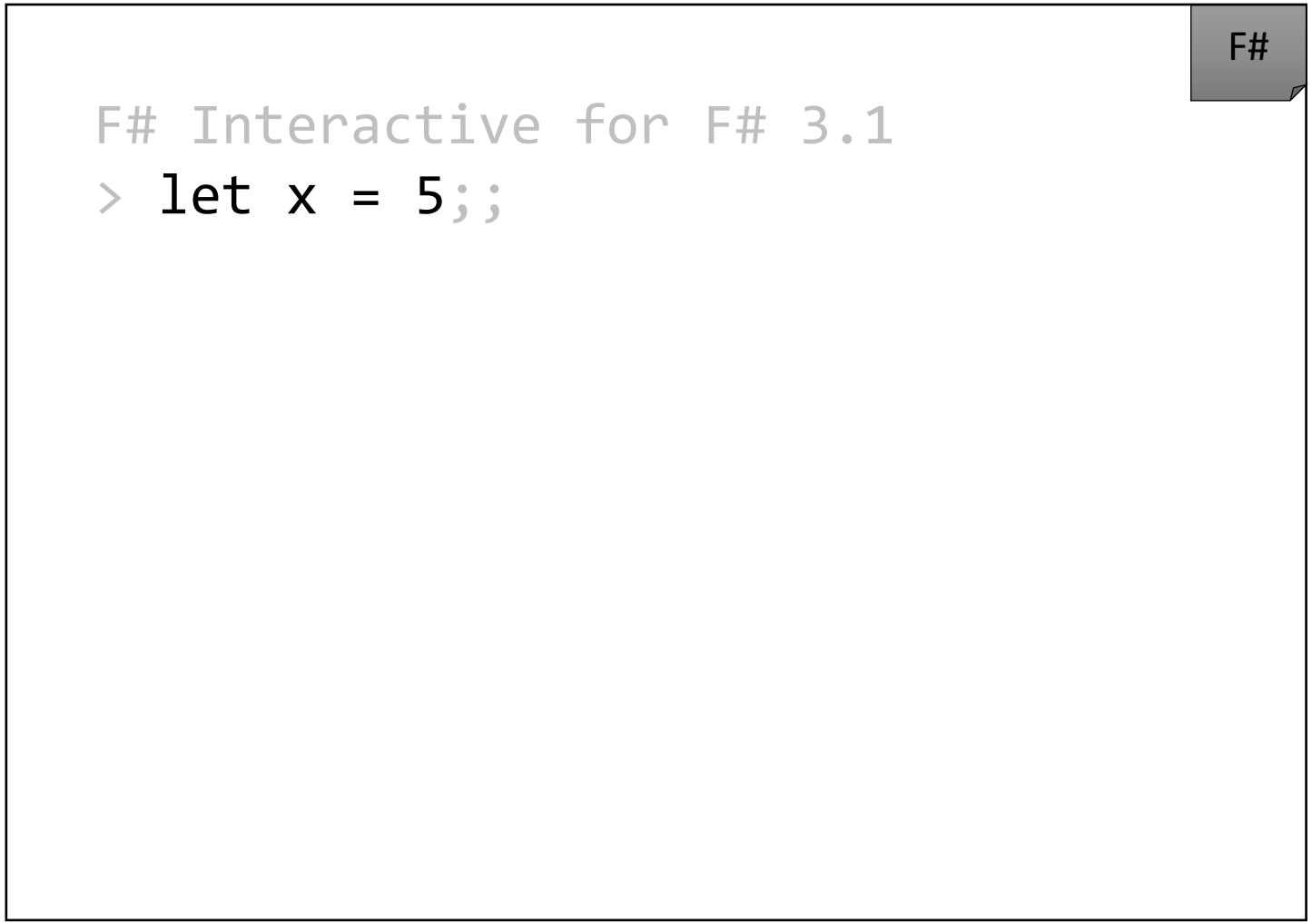
High school math
X = X + 1

You fail!

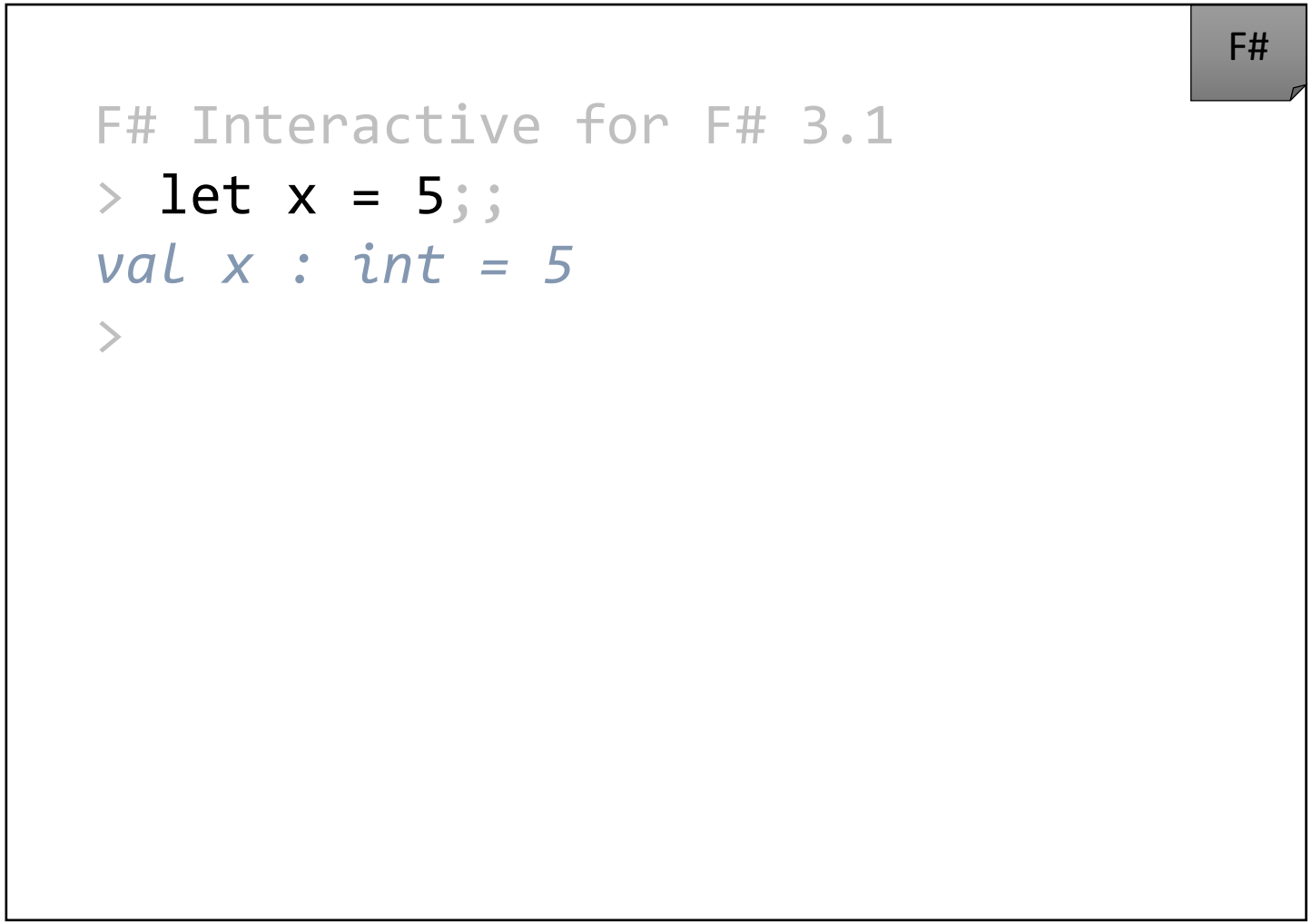
Immutability



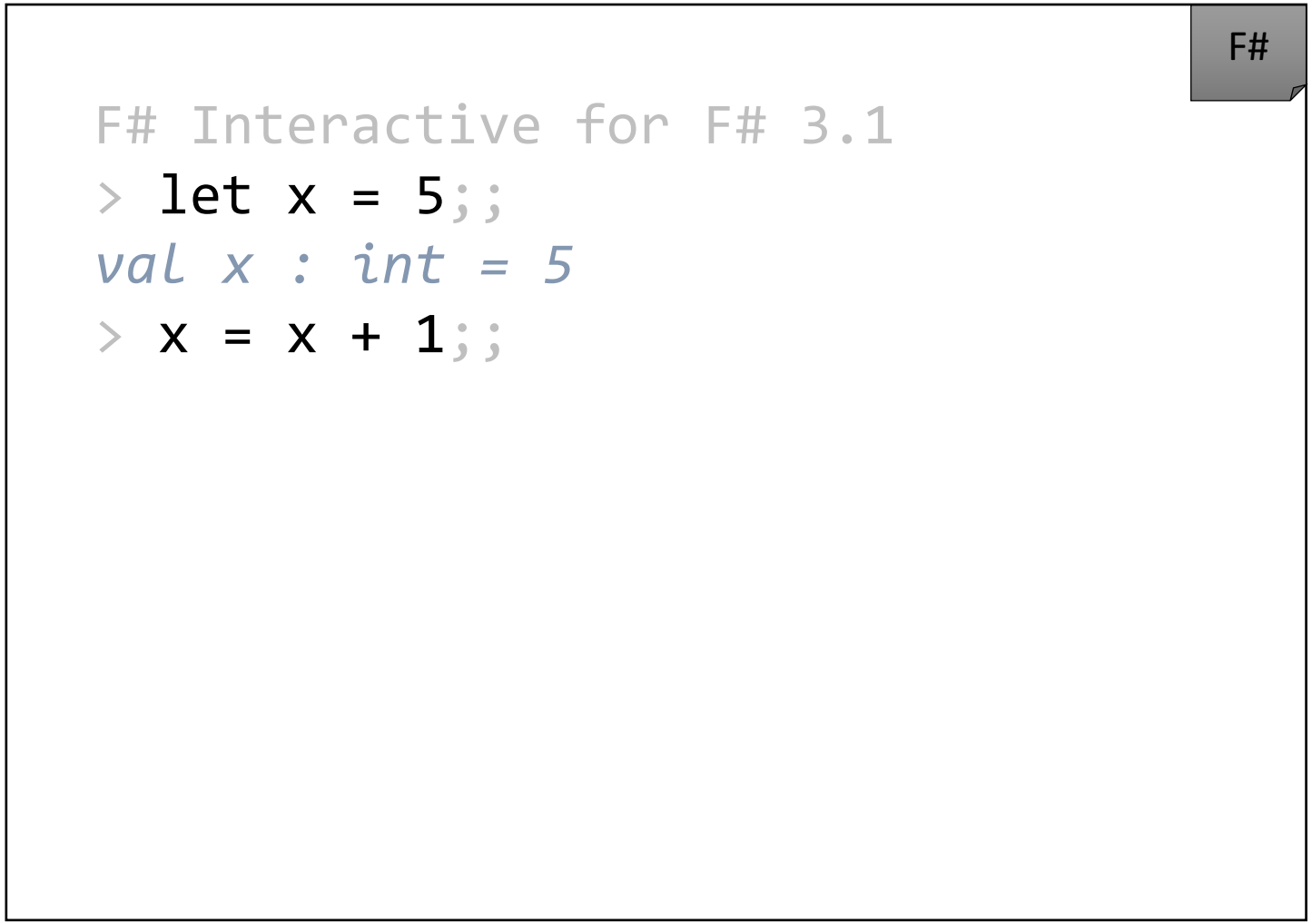
Immutability



Immutability



Immutability



Immutability

F#

```
F# Interactive for F# 3.1
> let x = 5;;
val x : int = 5
> x = x + 1;;
val it : bool = false
>
```

Immutability

F#

```
F# Interactive for F# 3.1
> let x = 5;;
val x : int = 5
> x = x + 1;;
val it : bool = false
> x <- 7;;
```

Immutability

F#

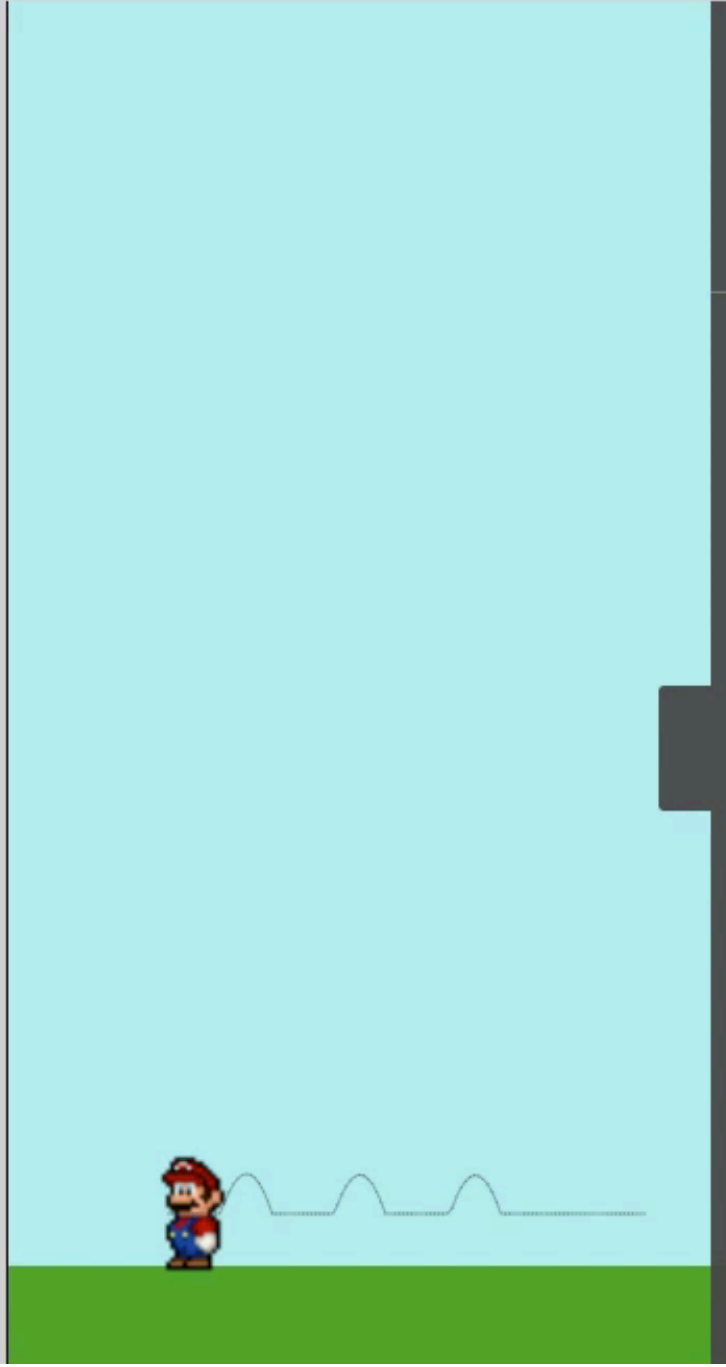
```
F# Interactive for F# 3.1
> let x = 5;;
val x : int = 5
> x = x + 1;;
val it : bool = false
> x <- 7;;
  x <- 7;;
  ^^^^^^
//stdin(19,1): error FS0027: This
value is not mutable
>
```

Purity

a –> b

```elm
-- MODEL

type Model =
    { x   : Float
    , y   : Float
    , vx  : Float
    , vy  : Float
    , dir : Direction
    }

data Direction = Left | Right

type Keys = { x:Int, y:Int }

mario : Model
mario =
    { x = 0
    , y = 0
    , vx = 0
    , vy = 0
    , dir = Right
    }


-- UPDATE

step : (Float, Keys) -> Model -> Model
step (dt, keys) mario =
    mario
        |> gravity dt
        |> jump keys
        |> walk keys
        |> physics dt
        |> Debug.watch "mario"

jump : Keys -> Model -> Model
jump keys mario =
    if keys.y > 0 && mario.vy == 0 then { mario | vy <- 2.8 } else mario

gravity : Float -> Model -> Model
gravity dt mario =
    { mario |
        vy <- if mario.y > 0 then mario.vy - dt/4 else 0
    }

physics : Float -> Model -> Model
physics dt mario =
```

Hints: ☑ Options: ☐ Auto-update: ☑ Hot Swap | Compile



166

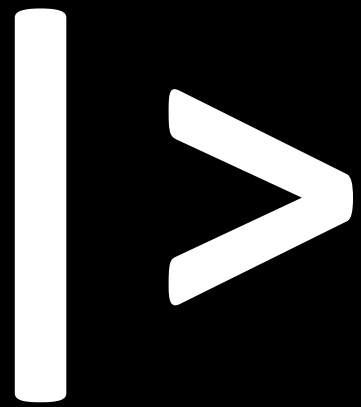
0 214

arrows

```
{
  x = -1,
  y = 0
}
```

mario

```
{
  dir = Left,
  vx = -1,
  vy = 0,
  x = -217.20,
  y = 0
}
```



|>

(Elixir)

expr |> expr

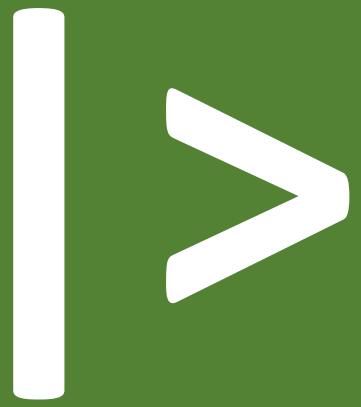
the expression on left
pipes forward
as the implicit ***1st*** argument
to the expression on the right

|>

(F#, Elm, …)

expr |> expr

the expression on left
pipes forward
as the implicit ***last*** argument
to the expression on the right



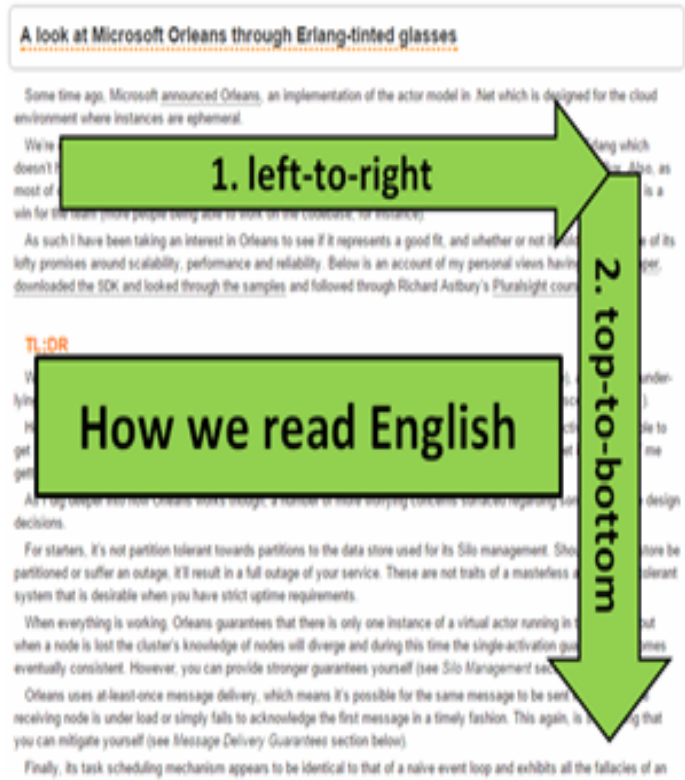
Transform

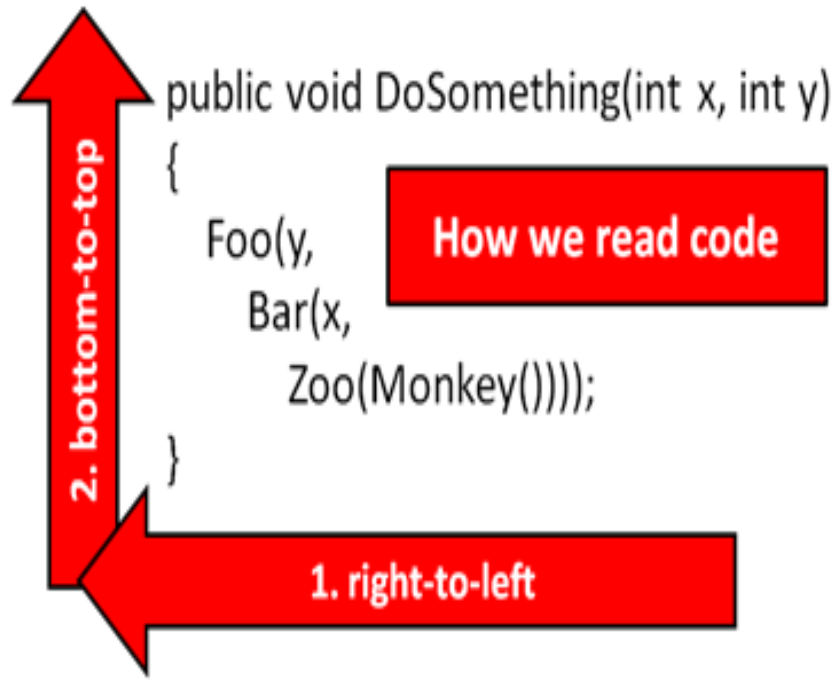
```
"Drink your Ovaltine"
```

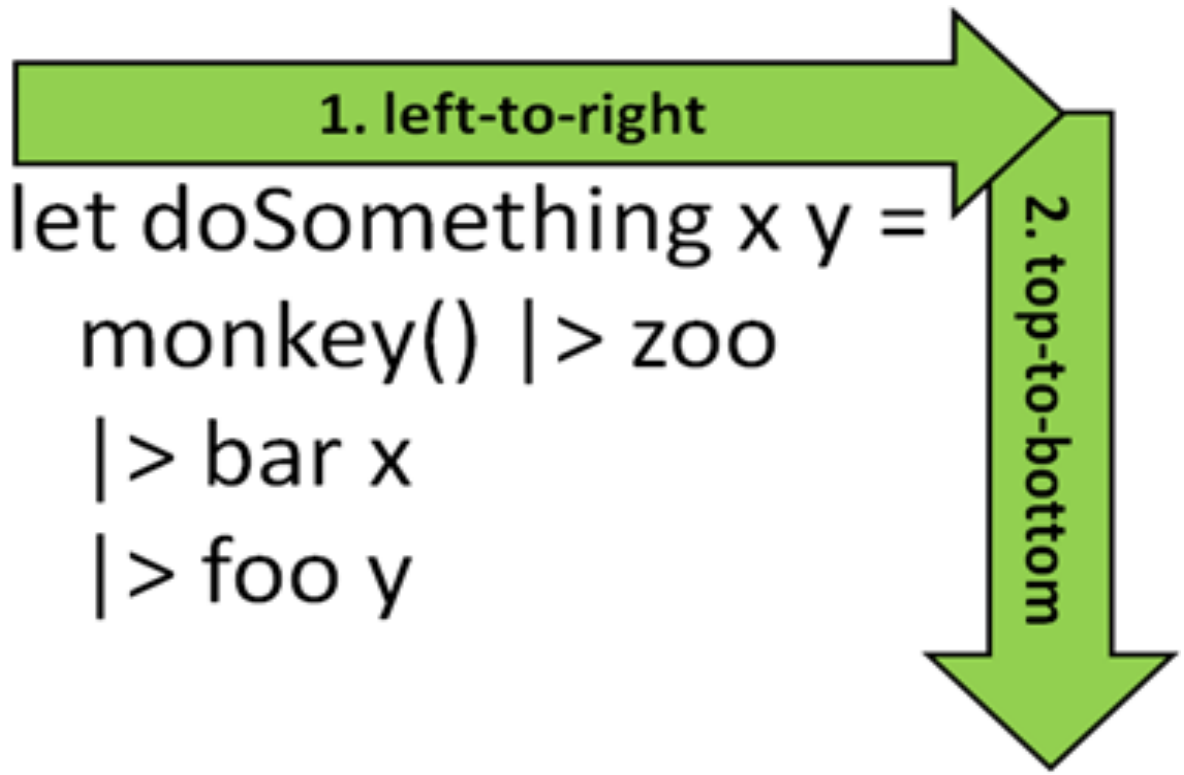
into…

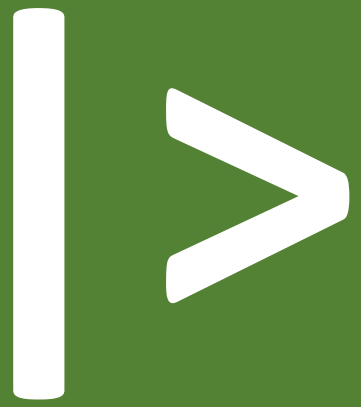
```
"🌮DRINK_YOUR_OVALTINE🌮"
```

```
yell(
  wrap(
    snake_space(
      "Drink your Ovaltine"),
    "🌮"))
```


A look at Microsoft Orleans through Erlang-tinted glasses
Some time ago, Microsoft announced Orleans, an implementation of the actor model in .Net which is designed for the cloud environment where instances are ephemeral.
As such I have been taking an interest in Orleans to see if it represents a good fit, and whether or not
lofty promises around scalability, performance and reliability. Below is an account of my personal views
downloaded the SDK and looked through the samples and followed through Richard Astbury's Pluralsight
TL;DR
decisions.
For starters, it's not partition tolerant towards partitions to the data store used for its Silo management.
partitioned or suffer an outage. It'll result in a full outage of your service. These are not traits of a masterless
system that is desirable when you have strict uptime requirements.
When everything is working, Orleans guarantees that there is only one instance of a virtual actor running in
when a node is lost the cluster's knowledge of nodes will diverge and during this time the single-activation
eventually consistent. However, you can provide stronger guarantees yourself (see Silo Management
Orleans uses at-least-once message delivery, which means it's possible for the same message to be
receiving node is under load or simply fails to acknowledge the first message in a timely fashion. This again,
you can mitigate yourself (see Message Delivery Guarantees section below).
Finally, its task scheduling mechanism appears to be identical to that of a naive event loop and exhibits all the fallacies of an
1. left-to-right
2. top-to-bottom
How we read English


public void DoSomething(int x, int y)
{
Foo(y,
Bar(x,
Zoo(Monkey()));
}
How we read code
2. bottom-to-top
1. right-to-left


1. left-to-right
2. top-to-bottom
let doSomething x y =
monkey() |> zoo
|> bar x
|> foo y



Transform

```
"Drink your Ovaltine"
```

into…

```
"🌮DRINK_YOUR_OVALTINE🌮"
```

```
"Drink your Ovaltine"
|> snake_space
|> wrap("🌮")
|> yell
```

Union Types

F#

```
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float
```

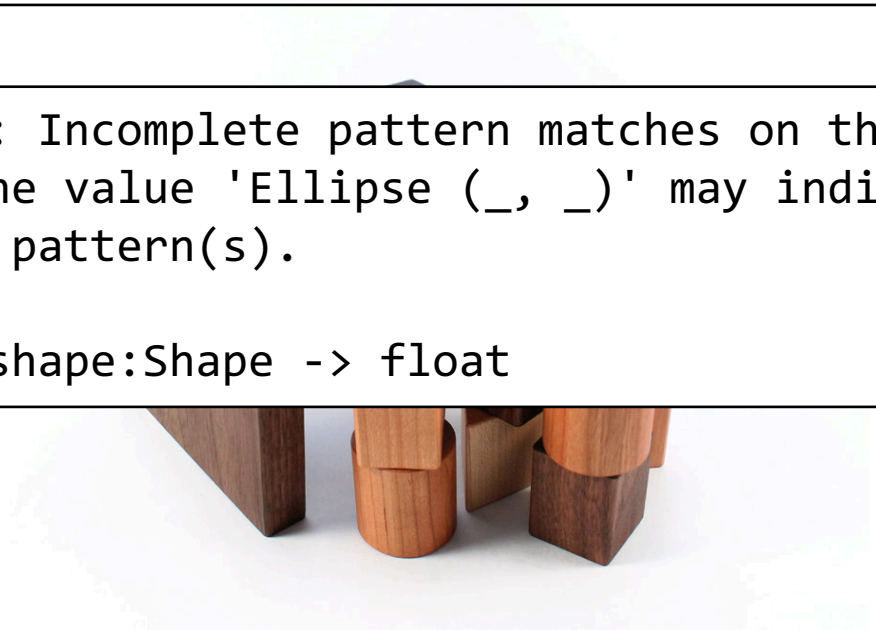
F#

```
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float

let GetArea shape =
    match shape with
    | Circle(radius) -> 3.14 * radius * radius
    | Square(width) -> width * width
    | Triangle(``base``, height) -> ``base`` * height/ 2.0
    | Rectangle(width, height) -> width * height
```

F#

```
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float
    | Ellipse of float * float

let GetArea shape =
    match shape with
    | Circle(radius) -> 3.14 * radius * radius
    | Square(width) -> width * width
    | Triangle(``base``, height) -> ``base`` * height/ 2.0
    | Rectangle(width, height) -> width * height
```

```
warning FS0025: Incomplete pattern matches on this expression.
For example, the value 'Ellipse (_, _)' may indicate a case not
covered by the pattern(s).

val GetArea : shape:Shape -> float
```



Oh, and Null

F#

```
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float

let GetArea shape =
    match shape with
    | Circle(radius) -> 3.14 * radius * radius
    | Square(width) -> width * width
    | Triangle(``base``, height) -> ``base`` * height/ 2.0
    | Rectangle(width, height) -> width * height
```

F#

```fsharp
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float

let GetArea shape =
    match shape with
    | Circle(radius) -> 3.14 * radius * radius
    | Square(width) -> width * width
    | Triangle(``base``, height) -> ``base`` * height/ 2.0
    | Rectangle(width, height) -> width * height
```

```
F# Interactive
>
```

F#

```
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float

let GetArea shape =
    match shape with
    | Circle(radius) -> 3.14 * radius * radius
    | Square(width) -> width * width
    | Triangle(``base``, height) -> ``base`` * height/ 2.0
    | Rectangle(width, height) -> width * height
```

```
F# Interactive
> GetArea null;;
```

F#

```
type Shape =
    | Circle of float
    | Square of float
    | Triangle of float * float
    | Rectangle of float * float

let GetArea shape =
    ma
    |
    |
    |
    |
```

MAKE INVALID STATES UNREPRESENTABLE

```
F# Int
> GetArea null;;
error FS0043: The type 'Shape' does not have 'null' as a proper
value
>
```

Why bother?

Why bother?

- Pure functions can be executed in parallel without interfering with one another
- Pure functions can be “perfectly” cached
- Pure functions can be “partially” applied
- Functions can receive and return functions, for which all of the above hold true
- Allows for greater “modularity” and “composability”

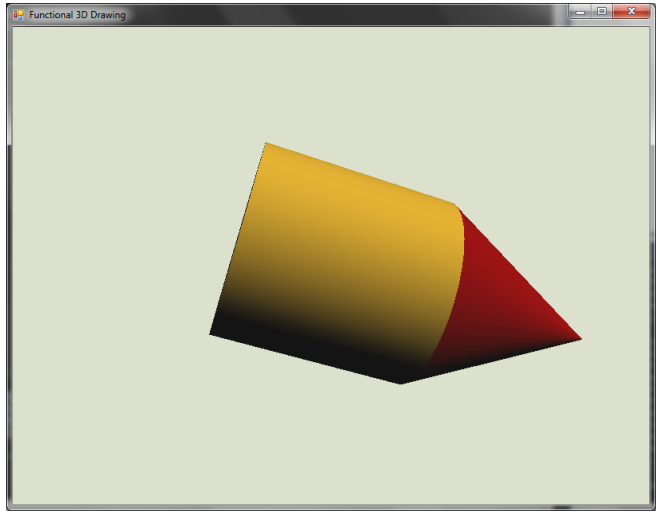
What are domain-specific languages and why?

Domain-specific languages

Language for solving **specific problems**

```
Fun.cube
```

```
Fun.cylinder
|> Fun.translate (0.0, 0.0, 1.0)
|> Fun.color Color.Goldenrod $
Fun.cone
|> Fun.color Color.DarkRed
```

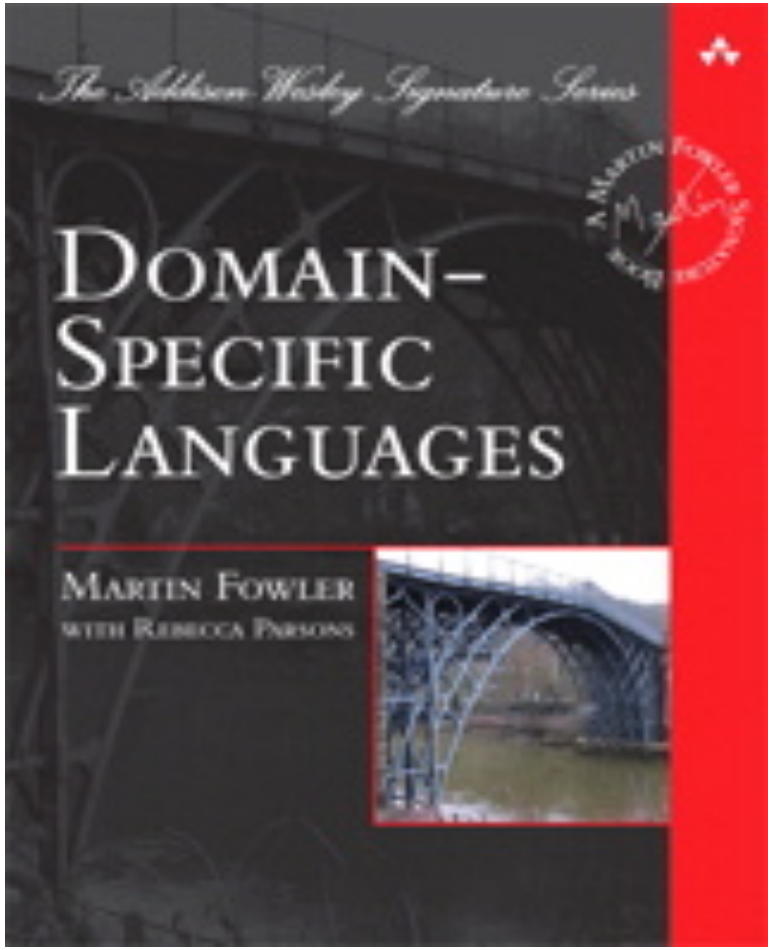


Contrast with **general purpose languages**

Why DSL

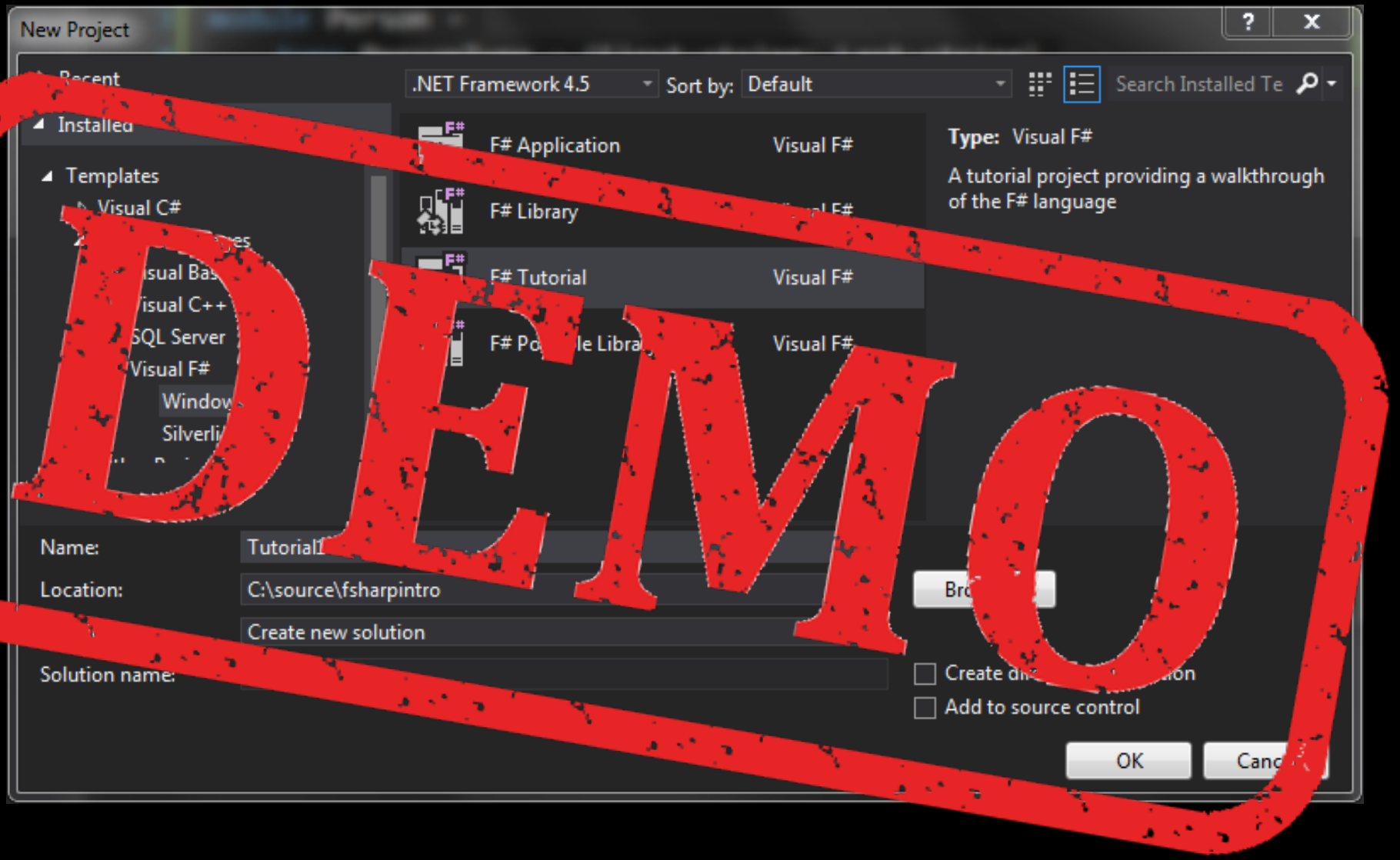
- Productivity
- Reliability
- Correctness
- Maintainability
  - Easier to reason

Domain-Specific Languages, Martin Fowler



DSL = model + syntax

- Primitives (data elements)
- Combinators
- Semantic & Syntax


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Domain-specific language approach

We have a class of problems

- Create a language for the class
- Use language to solve them

Domain model

- Understand the problem domain!
- Using ADT - discriminated unions

DSLs and functional languages

- Internal DSL – is just library
  - Flexible syntax and type checking
- External DSL – is a stand-alone language *(easier to build)*

Domain-specific language approach

Class of problems

Constructing 3D objects
Makefiles, stock price modelling, testing, …

Domain-specific language

Primitives – basic building blocks
Composition – how to put them together

Domain Specific Languages

The functional Way

- Rich type systems
- Great support for symbolic programming

Domain-specific languages
the functional way

- Using **discriminated unions**
  - Defines the **language**
  - Limits the **expressivity**
  - Simplify **common** uses
  - Multiple **processing** functions

Domain-specific languages

the functional way

- Using **discriminated unions**
  - Defines the **language**
  - Limits the **expressivity**
  - Simplify **common** uses
  - Multiple **processing** functions

- Using **function values**
  - Defines the **representation**
  - **Unlimited** expressivity
  - **Run** the function to evaluate

```fsharp
// set up the vocabulary
type DateScale = Hour | Hours | Day | Days | Week | Weeks
type DateDirection = Ago | Hence

// define a function that matches on the vocabulary
let getDate interval scale direction =
       let absHours = match scale with
              | Hour | Hours -> 1 * interval
              | Day | Days -> 24 * interval
              | Week | Weeks -> 24 * 7 * interval
       let signedHours = match direction with
              | Ago -> -1 * absHours
              | Hence -> absHours
       System.DateTime.Now.AddHours(float signedHours)

// test some examples
let example1 = getDate 5 Days Ago
let example2 = getDate 1 Hour Hence

// the C# equivalent would probably be more like this:
// getDate().Interval(5).Days().Ago()
// getDate().Interval(1).Hour().Hence()
```

Functional data structures (ADT)

- A way of thinking about problems
- Model data using composition of primitives

Tuple Record

- Combine more values of different types
- Multiple fields in a class

Discriminated Union

- Represent one of several alternatives
- Class hierarchy (with subclasses)

Algebraic data type

```
// Tuple
let me = { “Ricky”;  40 }
let (name, age) = me

// Record Type
type RecordType = { a : TypeA; b : TypeB }

type Person = { Name : string;  Age : int}
let me = { Name = “Ricky”;  Age = 40 }
```

Records in F#

- Named type declared using **type**
  - Sealed type, fields immutable by default

```
type Product =
  { Name : string
    Price : int }
```

- Easy to work with in a functional way
  - Create modified copy using **with** construct

```
let tea = { Name = "Tea"; Price = 42 }

let discaunt product =
  { product with Price = product.Price / 2
}
```

Discriminated Unions in F#

- Named type declared using **type**
  - Compiles to class hierarchy

```
type FamilyState =
| Married of string
| Single
```

- Processed using pattern matching

```
let famst = Merried("Ada")
match famst with
| Single -> printf "single"
| Married(who) -> printfn "married to %s" who
```

  - Compiler checks that no case is missing
  - Cannot access property of another case

Discriminated Unions & Pattern Matching

```
// An instance of 'Suit' can only have one of four possible values
type Suit = Diamonds | Hearts | Spades | Clubs

let printSuitName suit =
       match status with
       | Diamonds -> printfn "Suit is Diamands"
       | Hearts -> printfn "Suit is Hearts"
       | Spades -> printfn "Suit is Spades"
       | Clubs -> printfn "Suit is Clubs"

// Discriminated Unions can hold data too!
type Card =
| ValueCard of int * Suit
| Jack of Suit
| Queen of Suit
| King of Suit
| Ace of Suit


let myPokerHand = [ ValueCard(2, Hearts) ValueCard(4, Clubs) Ace(Clubs) ]
```

Algebraic data type

```
// Discriminated Union
type tree =
| Leaf of int
| Node of tree * tree
```

```
type tree =
| Leaf of int
| Node of tree * tree

let simpleTree = Node (Leaf 1, Node (Leaf 2, Node
                            (Node (Leaf 4, Leaf 5), Leaf 3)))
let countLeaves tree =
  let rec loop sum tree =
      match tree with
      | Leaf(_) -> sum + 1
      | Node(tree1, tree2) -> sum + (loop 0 tree1) + (loop 0 tree2)
   loop 0 tree

printfn "countLeaves simpleTree: %i" (countLeaves simpleTree)
```

Pipeline operator

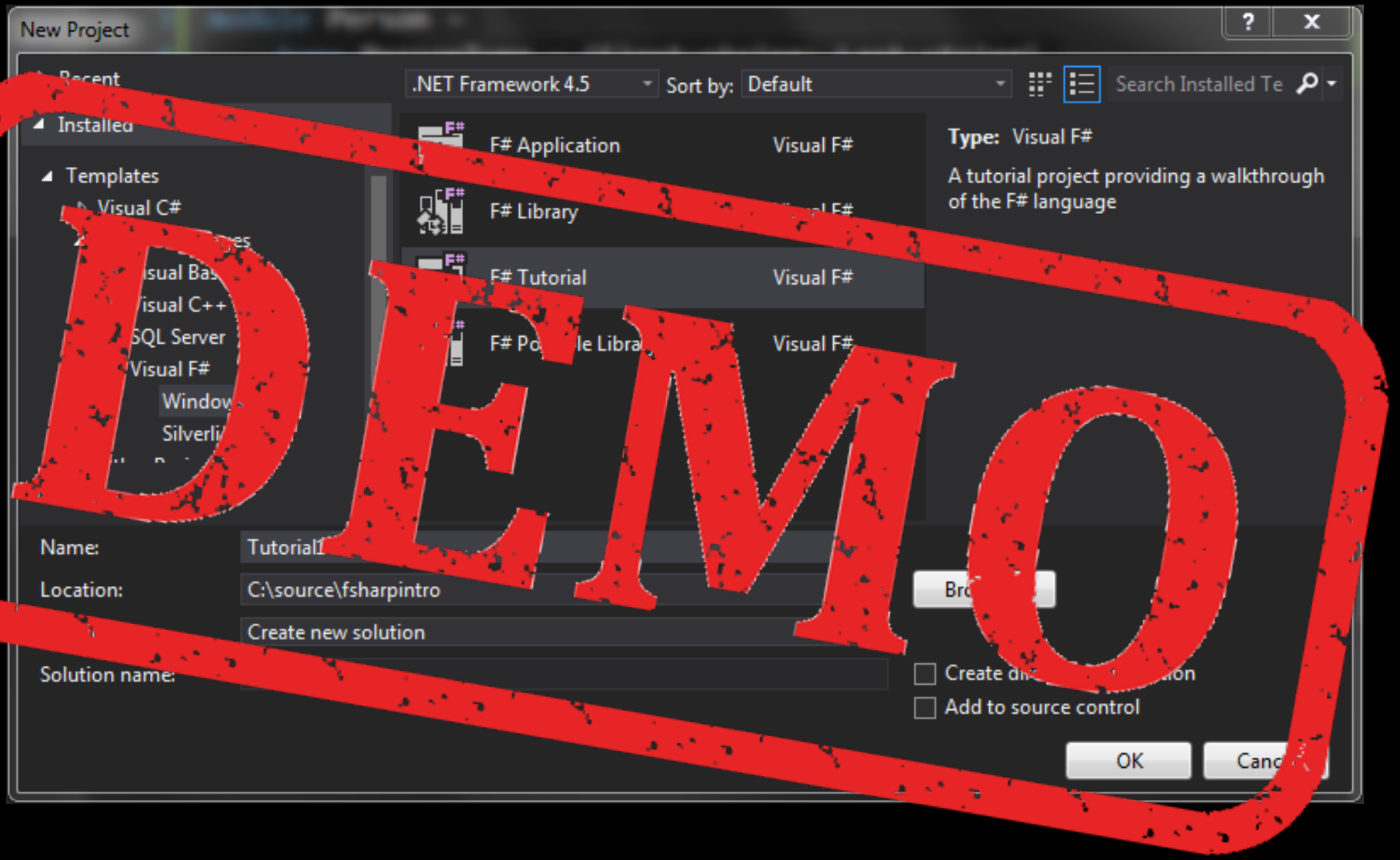
```
accounts
|> Seq.filter (belongsTo "John S.")
|> Seq.map calculateInterest
|> Seq.filter (flip (>) threshold)
|> Seq.fold (+) 0.0
```

Infix operator

```
type Account with
        static member (<<-)(x: Account, name) = x.addName(name)

let acc1 = Account("acc-1", "David P.")
acc1 <<- "Mary R." <<- "Shawn P." <<- "John S."
```

Statically Resolved Type Parameters

```
let inline (+@) x y = x + y * y

// Call that uses int
printfn "%d" (1 +@ 1)

// Call that uses float
printfn "%d" (1.0 +@ 0.5)
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

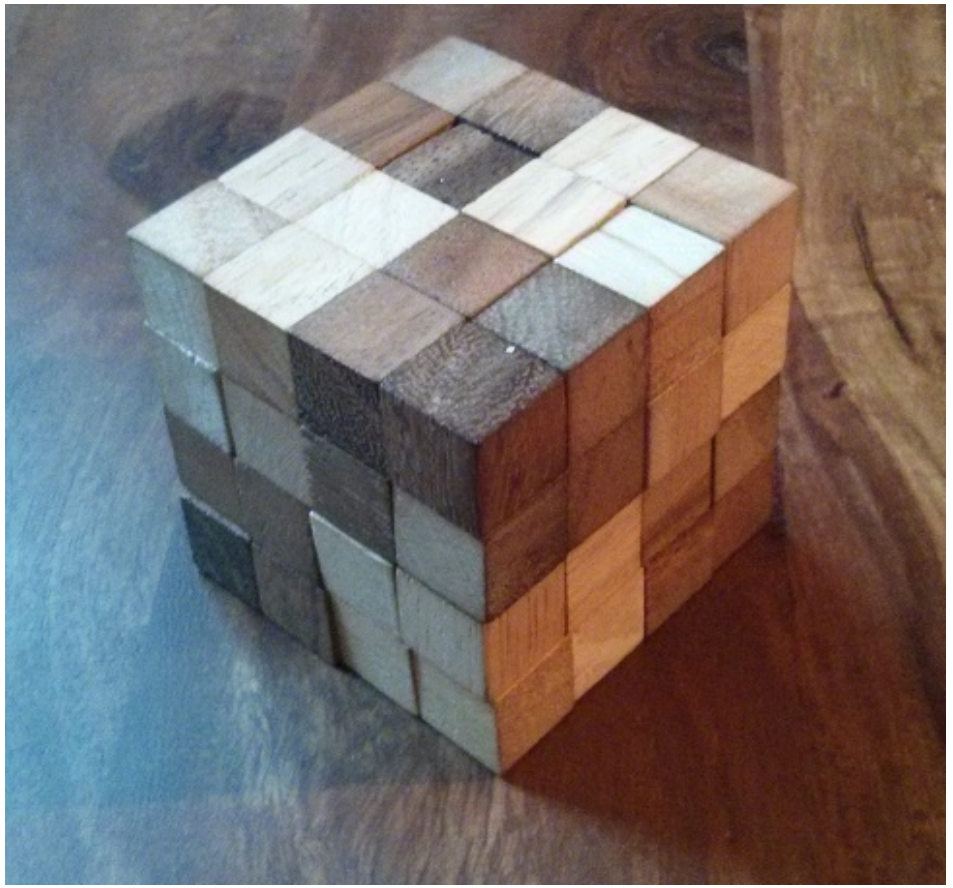
Demo: Building a castle

Domain Specific Language

- Defines a few **simple primitives**
- Extensible by **composition**

Single-purpose or general-purpose?

- Most code is **single-purpose**
- Can use **general-purpose** if needed



DSL Summary

How To: Building your own DSL

1. Understand problem domain
   Primitives & **Combinators**

2. Model the language using
   Discriminated Unions

3. Add convenient Syntax
   Internal or **External**

Internal DSL: Building Blocks

Vanilla .NET

Method chaining

Enumerations

Classes

Operator Overloading

Attributes

Iterators & LINQ

Extension methods

F# Specific

Pipelining

Discriminated Unions

Records

Custom Operators

Quotations

Computation Expressions

Functions

Advanced Embedded DSLs

Computation expressions

- Reinterpret expression composition
- Add constructs with F# 3.0 queries

Meta-programming with quotations

- Reinterpret F# expressions

Active patterns

- More expressive pattern language
- Implementing external DSLs

Domain-specific languages

Advantages

Readability

- Greater for External DSL

Maintainability

- Hides the implementation
- Internals can be changed

Domain Focus

- Non-experts can read it

Disadvantages

Additional abstraction

- Smaller for Internal DSL

Time to implement

- Easier for Internal DSL

Time to learn

- Avoid crazy operators
- Make it familiar

Let’s get real

Real production code

Take away

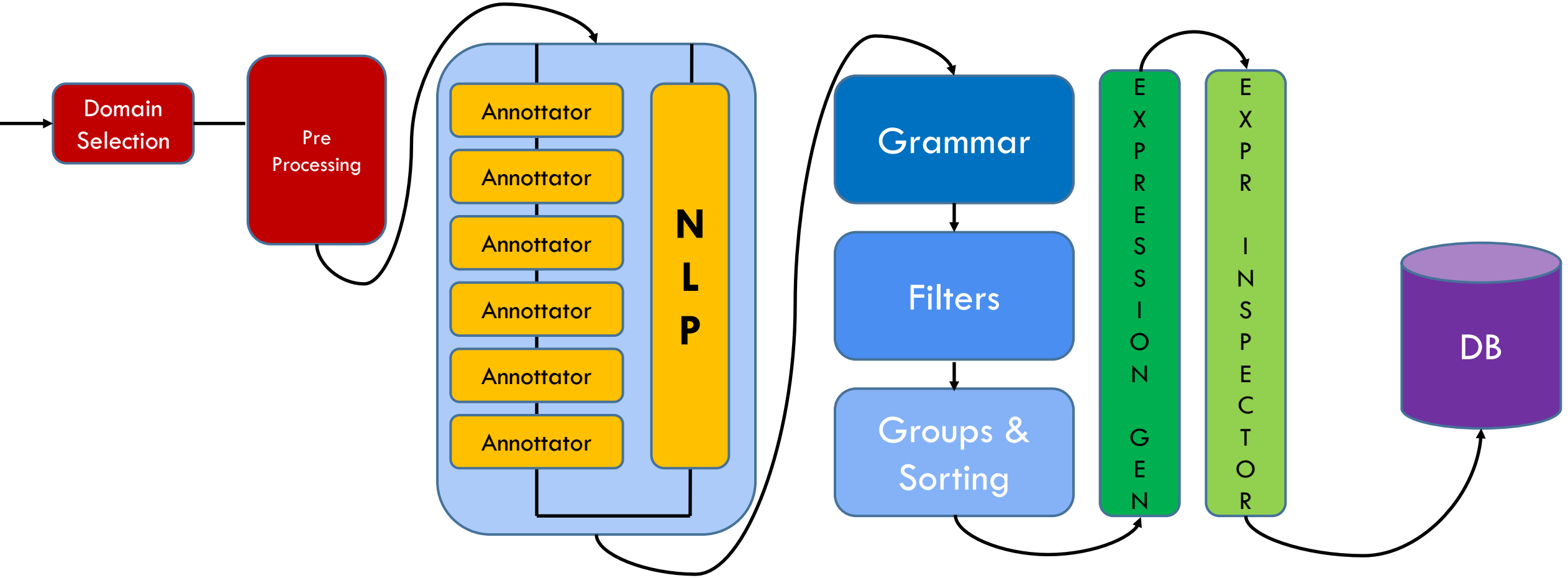
"Problems cannot be solved with the same mind set that created them"

— Albert Einstein

The case – Expand to UI-Conversation

- Dynamic query generation
- Linq.Expression builder in C#
- EntityFramework *dependencies*
  - *(looking for better performance)*
- Sql Server *dependencies*
  - *(looking for better performance, better with 2014 column store but could be better)*

What's going on



Draft Filter (simple one)

```
public abstract class DraftFilter<T> : IFilterRule<T>   where T : class, Data.IDraft
{
     public Task<Func<IQueryable<T>, Expression<Func<T, bool>>>> ApplyAsync(
        IQueryContext context, CancellationToken cancellationToken)
    {
        var tokens = context.Tokens;
        var draftYears = tokens.ExceptIgnored().OfType<DraftToken>().ToArray();

        var ignore = new[] { SportLexicon.Noun.Player.Draft.Key };

        var result = ContextAwareExpression.Where<T, DraftToken>((query, subject, ex) =>
        {
            if (subject.Year.HasValue)
                return game => game.draft_year == subject.Year;
            if (subject.TopNPicks.HasValue)
                return game => game.draft_overall_pick <= subject.TopNPicks;
             return null;
        }, tokens, draftYears, defaultAsOr: true, ignoreOperators: true, ignore: ignore);
        return Task.FromResult(result);
    }
}
```

GameIndex Filter (comlpex)

```
public abstract class GameIndexFilter<T> : IFilterRule<T>   where T : class, Data.ITeamGame
{
    public int Applies(IToken[] tokens)
    {
        if (tokens.Of(SportLexicon.Noun.Game.Key).Any())
            return 1100;

        return -1;
    }

    public Task<Func<IQueryable<T>, Expression<Func<T, bool>>>> ApplyAsync(
        IQueryContext context, CancellationToken cancellationToken)
    {
        var tokens = context.Tokens;
        var nouns = tokens.Of(SportLexicon.Noun.Game.Key).Where(s => !tokens.AnyBefore(s, new[]
                {
                    SportLexicon.Noun.Player.Key
                },
                Token.NumberKey,
                EnglishLexicon.Adjective.Superlative.Key,
                EnglishLexicon.Possessive.Key,
                EnglishLexicon.Punctuation.Key)).ToArray();
```

GameIndex Filter (comlpex)

```
public abstract class GameIndexFilter<T> : IFilterRule<T>   where T : class, Data.ITeamGame
{
    public int Applies(IToken[] tokens)
    {
        if (tokens.Of(SportLexicon.Noun.Game.Key).Any())
            return 1100;

        return -1;
    }

    public Task<Func<IQueryable<T>, Expression<Func<T, bool>>>> ApplyAsync(
        IQueryContext context, CancellationToken cancellationToken)
    {
        var tokens = context.Tokens;
        var nouns = tokens.Of(SportLexicon.Noun.Game.Key).Where(s => !tokens.AnyBefore(s, new[]
                {
                    SportLexicon.Noun.Player.Key
                },
                Token.NumberKey,
                EnglishLexicon.Adjective.Superlative.Key,
                EnglishLexicon.Possessive.Key,
                EnglishLexicon.Punctuation.Key)).ToArray();

        var ignore = new[]
        {
            SportLexicon.Noun.Game.Key,
            EnglishLexicon.Adjective.Superlative.First.Key,
            EnglishLexicon.Adjective.Superlative.Last.Key,
            SportLexicon.Noun.GameLocation.Key,
            SportLexicon.Noun.GameType.Key,
            SportLexicon.Noun.Season.Key,
```

GameIndex Filter (comlpex)

```
public abstract class GameIndexFilter<T> : IFilterRule<T>   where T : class, Data.ITeamGame
{
    public int Applies(IToken[] tokens)
    {
        if (tokens.Of(SportLexicon.Noun.Game.Key).Any())
            return 1100;

        return -1;
    }

    public Task<Func<IQueryable<T>, Expression<Func<T, bool>>>> ApplyAsync(
        IQueryContext context, CancellationToken cancellationToken)
    {
        var tokens = context.Tokens;
        var nouns = tokens.Of(SportLexicon.Noun.Game.Key).Where(s => !tokens.AnyBefore(s, new[]
                {
                    SportLexicon.Noun.Player.Key
                },
                Token.NumberKey,
                EnglishLexicon.Adjective.Superlative.Key,
                EnglishLexicon.Possessive.Key,
                EnglishLexicon.Punctuation.Key)).ToArray();

        var ignore = new[]
        {
            SportLexicon.Noun.Game.Key,
            EnglishLexicon.Adjective.Superlative.First.Key,
            EnglishLexicon.Adjective.Superlative.Last.Key,
            SportLexicon.Noun.GameLocation.Key,
            SportLexicon.Noun.GameType.Key,
            SportLexicon.Noun.Season.Key,
        };

        var result = ContextAwareExpression.Where<T>((query, subject, ex) =>
        {
            var superlative = tokens.Before(subject, new[]
            {
                EnglishLexicon.Adjective.Superlative.First.Key,
                EnglishLexicon.Adjective.Superlative.Last.Key
            }, MathLexicon.Math.Operator.Logical.Key,
                Token.NumberKey,
                SportLexicon.Noun.Season.Key,
                SportLexicon.Noun.GameLocation.Key,
                SportLexicon.Noun.GameType.Key);

            if (superlative == null)
                return null;

            superlative.Ignore();

            var relationalOperator = subject.Aspects.OfType<RelationalAspect>().Select(i => i.Operator).FirstOrDefault()
                                     ?? Token.From(MathLexicon.Math.Operator.Relational.Leading.Equal.Key);

            var isEqualTo = OperatorExpressions.IsEqualTo(relationalOperator);
            var isGreaterThan = OperatorExpressions.IsGreaterThan(relationalOperator);
            var isLessThan = OperatorExpressions.IsLessThan(relationalOperator);
            var isInclusive = OperatorExpressions.IsInclusive(relationalOperator);

            // Find the coefficient (will always be in front of the subject: "last 5 games")
            var coefficientToken = tokens.Before(subject, Token.NumberKey,
                SportLexicon.Noun.Season.Key, // last 10 playoff games
                SportLexicon.Noun.GameLocation.Key,
```

```
// Use 1 as the default coefficient ("last game" implies "last 1 games")
        int coefficient = hasCoefficient ?
            (int)coefficientToken.NumericValue.Value
            : 1;

        var isLast = superlative.Is(EnglishLexicon.Adjective.Superlative.Last.Key);

        var teamTokens = tokens.OfType<TeamToken>().Where(t => !t.IsOpponent).ToArray();

        if (teamTokens.Any())
        {
            var team_ids = teamTokens.SelectMany(t => t.Teams.Keys).ToArray();

            var teamGameQuery = query.Where(i => team_ids.Contains(i.team_id)).Select(i => new { i.game_id, i.game_date }).Distinct();

            var teamGames = (isLast ? teamGameQuery.OrderByDescending(x => x.game_date) : teamGameQuery.OrderBy(x => x.game_date)).Take(coefficient).ToArray();

            var teamgame_ids = teamGames.Select(g => g.game_id);

            var teamgame_dates = teamGames.Select(g => g.game_date);

            if (!isEqualTo)
            {
                var referenceDate = (isGreaterThan && isInclusive) || (isLessThan && !isInclusive)
                    ? teamgame_dates.Min()
                    : teamgame_dates.Max();

                var game_date = Expression.Property(_parameterProvider.Parameter, "game_date");
                var refgame_date = Expression.Constant(referenceDate);

                return TimeExpression.Evaluate<T>(relationalOperator, game_date, refgame_date, _parameterProvider.Parameter);
            }
            else
            {
                return i => teamgame_ids.Contains(i.game_id);
            }
        }

        // Cases:
        // "the last N games"
        // "the first N games"
        // "the first game"
        var activeTeams = GetActiveTeams();
        var games = query.Where(i => activeTeams.Contains(i.team_id)).Select(i => new { i.team_id, i.game_id, i.game_date }).Distinct().GroupBy(i => i.team_id).T
        var gameIds = games.Select(i => new { team_id = i.Key, Games = (isLast ? i.OrderByDescending(x => x.game_date) : i.OrderBy(x => x.game_date)).Take(coeff

        Expression<Func<T, bool>> expression = (i => false);
        foreach (var gameId in gameIds)
        {
            if (!isEqualTo)
            {
                var referenceDate = (isGreaterThan && isInclusive) || (isLessThan && !isInclusive)
                    ? gameId.Games.Min(g => g.game_date)
                    : gameId.Games.Max(g => g.game_date);

                Expression<Func<T, bool>> teamConstraint = (i) => i.team_id == gameId.team_id;

                var gameDate = Expression.Property(_parameterProvider.Parameter, "game_date");
                var refGameDate = Expression.Constant(referenceDate);

                var gameDateConstraint = TimeExpression.Evaluate<T>(relationalOperator,
```

EfInspector combines Expressions

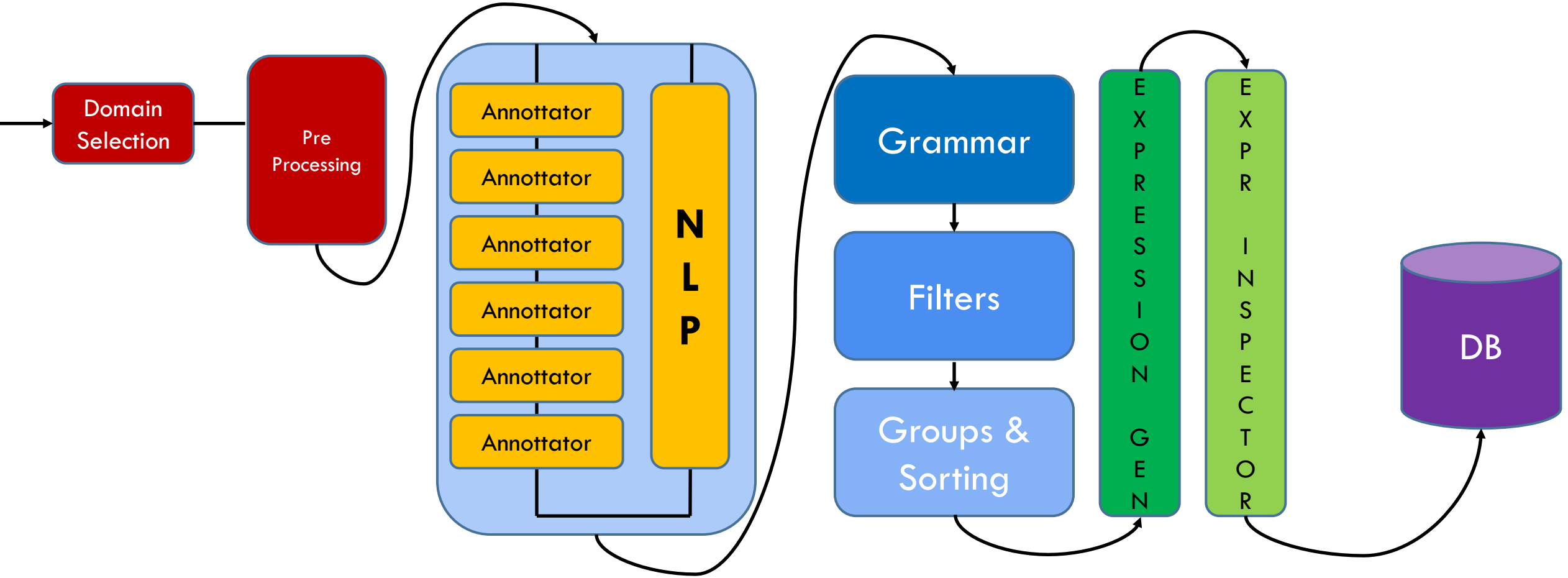
```
public class EfSchemaInspector : ISchemaInspector
    {
        private readonly IParameterCollection _parameterCollection;
        private readonly IDictionary<Type, Dictionary<string, Func<ParameterExpression,
MemberExpression>>> _memberMaps = new Dictionary<Type, Dictionary<string,
Func<ParameterExpression, MemberExpression>>>();

        public EfSchemaInspector(IParameterCollection parameterCollection,
                                 IAssemblyToLoad assemblyToLoad)
        {
            _parameterCollection = parameterCollection;

            var assemblies = AppDomain.CurrentDomain
                .GetAssemblies()
                .Where(a =>  a.GetCustomAttributes(assemblyToLoad.AssemblyToLoad, false))
                .ToArray();
```

```
public void CatalogTypes(IEnumerable<Type> types)
{
    _memberMaps.AddRange(types
        .ToDictionary(t => t, t =>
        {
            var paths = IterateProps(t);
            return paths.ToDictionary<string, string, Func<ParameterExpression, MemberExpression>>(p => p,
            {
                return (input) =>
                {
                    // Ignore the type name (Skip 1)
                    var propertyNames = p.Split('.').Skip(1);

                    Expression property = input;
                    foreach (var prop in propertyNames)
                    {
                        property = Expression.PropertyOrField(property, prop);
                    }
                    return property as MemberExpression;
                };
            });
        }));
}
private static IEnumerable<string> IterateProps(Type baseType)
{
    return IteratePropsInner(baseType, baseType.Name, 0);
}
```

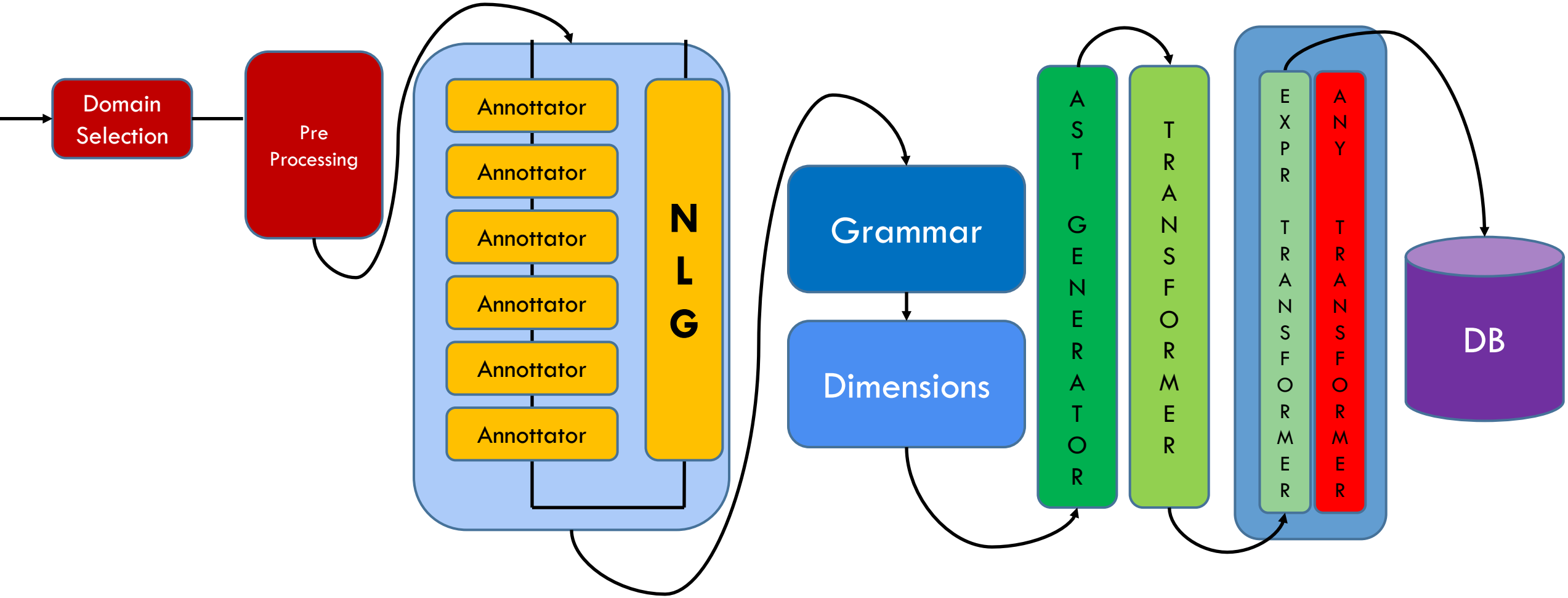
```
public class EfSchemaInspector : ISchemaInspector
    {
        private readonly IParameterCollection _parameterCollection;
        private readonly IDictionary<Type, Dictionary<string, Func<ParameterExpression,
MemberExpression>>> _memberMaps = new Dictionary<Type, Dictionary<string,
Func<ParameterExpression, MemberExpression>>>();

        public EfSchemaInspector(IParameterCollection parameterCollection,

            _parameterCollection = parameterCollection;

            var assemblies = AppDomain.CurrentDomain
               .GetAssemblies()
               .Where(a =>  a.GetCustomAttributes(assemblyToLoad.AssemblyToLoad, false))
               .ToArray();



            var entities = assemblies
                .SelectMany(x => x.GetTypes())
                .Where(x => !x.IsAbstract && typeof(Entity).IsAssignableFrom(x));

            CatalogTypes(entities);
        }

        public void CatalogTypes(IEnumerable<Type> types)
        {
            _memberMaps.AddRange(types
                .ToDictionary(t => t, t =>
                {
                    var paths = IterateProps(t);
                    return paths.ToDictionary<string, string, Func<ParameterExpression, MemberExpression>>(p => p, p =>
                    {
                        return (input) =>
                        {
                            // Ignore the type name (Skip 1)
                            var propertyNames = p.Split('.').Skip(1);

                            Expression property = input;
                            foreach (var prop in propertyNames)
                            {
                                property = Expression.PropertyOrField(property, prop);
                            }
                            return property as MemberExpression;
                        };
                    });
                }));
        }

        private static IEnumerable<string> IterateProps(Type baseType)
        {
            return IteratePropsInner(baseType, baseType.Name, 0);
        }

        private static IEnumerable<string> IteratePropsInner(Type baseType, string baseName, int depth)
        {
            var props = baseType.GetProperties();

            foreach (var property in props)
            {
                var name = property.Name;
                var type = property.PropertyType;

                // Limit to 3 levels deep
                if (depth > 3 ||
                    (type.IsGenericType && type.GetGenericTypeDefinition() == typeof(ICollection<>))
                    || type.IsPrimitive || type == typeof(string) || type == typeof(DateTime)
                    || type == typeof(TimeSpan) || IsNullableType(type))
                    yield return string.Format("{0}.{1}", baseName, property.Name);
                else
                    foreach (var info in IteratePropsInner(type, name, depth + 1))
                        yield return string.Format("{0}.{1}", baseName, info);
            }
        }

        public IEnumerable<Type> GetTypeMaps(params string[] maps)
        {
            if (maps == null) return null;

            return _memberMaps.Keys.Where(k => maps.Any(k.Name.EqualsIgnoreCase));
        }
```

```
        public IEnumerable<Expression<Func<T, TMember>>> GetConstants<T, TMember>(IToken token, params string[] constants)
        {
            if (constants.None()) return new Expression<Func<T, TMember>>[0];

            var parameter = _parameterCollection.Find(typeof(T));

            return constants
                        .Where(e => MatchesPropertyType(e, typeof(TMember)))
                        .Select(c =>
                        {
                            var constant = Expression.Constant(c, c.GetType());
                            var converter = Expression.Convert(constant, typeof(TMember)).Expand();
                            return Expression.Lambda<Func<T, TMember>>(converter, parameter);
                        });
        }

        public Expression<Func<T, TMember>> GetMap<T, TMember>(string property)
        {
            var type = typeof(T);

            if (!_memberMaps.ContainsKey(type)) return null;

            var parameter = _parameterCollection.Find(typeof(T));
            var memberExpressions = _memberMaps[type]
                                        .Where(k => k.Key.EndsWith(property))
                                        .OrderBy(k => k.Key.Length)
                                        .Select<KeyValuePair<string, Func<ParameterExpression, MemberExpression>>, Expression>(i =>
                                        {
                                            var body = i.Value(parameter);
                                            return body;
                                        }).Distinct();
            return memberExpressions
                        .Where(e => MatchesPropertyType(e, typeof(TMember)))
                        .Select(m =>
                        {
                            var converter = Expression.Convert(m, typeof(TMember)).Expand();
                            return Expression.Lambda<Func<T, TMember>>(converter, parameter);
                        })
                        .FirstOrDefault();
        }

        public Expression<Func<T, TMember>> GetExpression<T, TMember>(string expression)
        {
            var parameter = _parameterCollection.Find(typeof(T));

            try
            {
                return DynamicExpression.ParseLambda<T, TMember>(expression, parameter).Expand();
            }
            catch (ParseException)
            {
                return null;
            }
        }

        public IEnumerable<Expression<Func<T, TMember>>> GetMaps<T, TMember>(IToken token, params string[] maps)
        {
            if (maps == null || !maps.Any()) return new Expression<Func<T, TMember>>[0];

            // Ensure that we prefix any naked maps with "."
            maps = maps.Select(m => m.Contains(".") ? m : "." + m).ToArray();

            var type = typeof(T);
            if (type.IsGenericType && type.GetGenericTypeDefinition() == typeof(IEnumerable<>))
            {
                var aggregateFunction = token.Aspects.OfType<AggregateAspect>().Any(a => a.IsAverage) ? "Average" : "Sum";
                return GetExpressions<T, TMember>(token, maps.Select(m =>
                {
                    var property = m.Trim('.');
                    return string.Format("{0}(Double?({1}))", aggregateFunction, property);
                }).ToArray());
            }

            if (!_memberMaps.ContainsKey(type)) return new Expression<Func<T, TMember>>[0];

            var parameter = _parameterCollection.Find(typeof (T));
            var memberExpressions = _memberMaps[type]
                                        .Where(k => maps.Any(k.Key.EndsWith))
                                        .OrderBy(k => k.Key.Length)
                                        .Select<KeyValuePair<string, Func<ParameterExpression, MemberExpression>>, Expression>(i =>
                                        {
```

What's going on



AST – QueryExpression solution



Query Expression

```
// Represents expression, which can be evaluated to value
// of various types
type QueryExpression =
    | GetColumn of ColumnName
    | Binary of QueryExpression * QueryExpression * BinaryOperator
    | Unary of QueryExpression * UnaryOperator
    | Call of KnownMethod * QueryExpression list
    | Constant of QueryValue
    | ContainsElements of CollectionReference * QueryExpression

and ColumnName = string
```

Binary Operator

```
type BinaryOperator =
    | Plus
    | Minus
    | Divide
    | Multiply
    | Equals
    | IfThen
    | RelationOperator of RelationOperator
    | And
    | Or
```

```
type RelationOperator =
    | LessThan
    | GreaterThan
    | LessThanOrEqual
    | GreaterThanOrEqual
```

Filter Dimension & Predicate

```
[<Interface>]
type IPreFilterDimension =
    abstract member toFilter : unit -> QueryPredicate

type FilterQuery = QueryExpression

type QueryPredicate =
    | Predicate of FilterQuery
    | Wrong of string
```

Player Filter

```
type PlayerDimension =
    | Player of InputString * int list
    | PlayerPosition of InputString * int

    interface IFilterDimension with
        member this.toFilter () =
            match this with
            | Player(input,personIds) ->
                personIds
                |> Seq.map(fun id ->
                  Condition(Equals(personIdColumn, (number id))))
                |> reduceWithOr
                |> Predicate
            | PlayerPosition(input,positionId) ->
                positionIdColumn === NullVal(positionId) |> Predicate
```

Player Filter

```
type PlayerDimension =
    | Player of InputString * int list
    | PlayerPosition of InputString * int

    interface IFilterDimension with
        member this.toFilter () =
            match this with
            | Player(input,personIds) ->
                personIds
                |> Seq.map(fun id ->
                  Condition(Equals(personIdColumn, (number id))))
                |> reduceWithOr
                |> Predicate
            | PlayerPosition(input,positionId) ->
                positionIdColumn === NullVal(positionId) |> Predicate
```

QueryExpression transformer

```
let transform (input : IQueryable<'T>) (tfs:QueryExpression) =
    match tfs with
    | Predicate e ->
        // transform filter converting the expression to `Func<'T, bool>`
        // and call the LINQ `Where` operation
        input.Where(makeFunction<'T, bool> convertExpression e)
    | SortBy [] -> input
    | SortBy((e, ord) :: es) ->
        let input =
            match ord with
            | Ascending -> input.OrderBy( ...
            | Descending -> input.OrderByDescending( ...
        let output = es |> Seq.fold thenSortBy input
        upcast output
    | GroupBy(e, aggs) -> ...
```

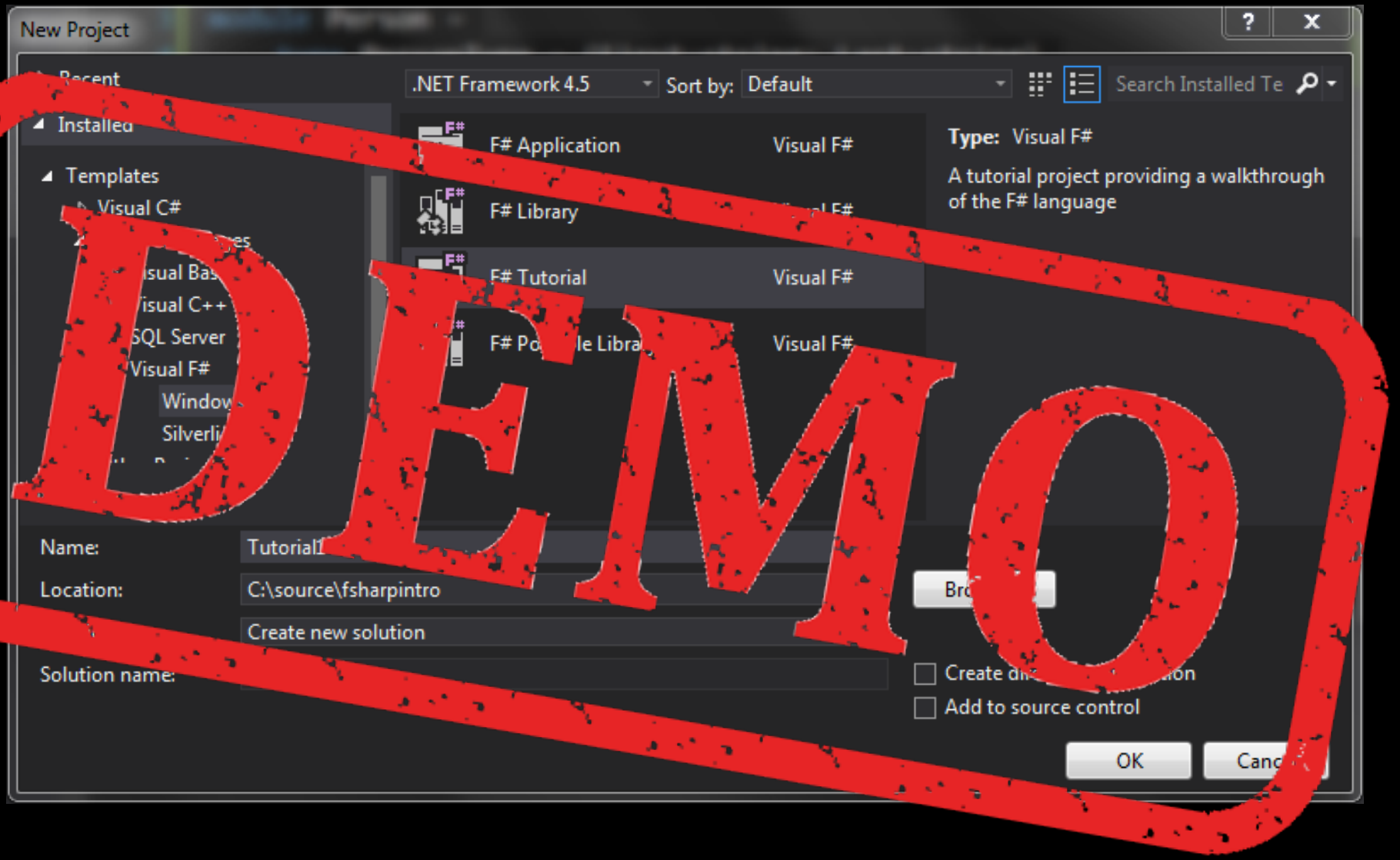
```
let rec convertExpression ctx e =
    match e with
    | GetColumn(s) -> ctx.ColumnGetter s
    | Constant(String v) -> upcast Expression.Constant(v)
    | Constant(Number v) -> upcast Expression.Constant(v)
    | Constant(Boolean v) -> upcast Expression.Constant(v)
    | Binary(e1, e2, op) ->
        let e1 = convertExpression ctx e1
        let e2 = convertExpression ctx e2
        match op with
        | And -> upcast Expression.AndAlso(e1, e2)
        | Or -> upcast Expression.OrElse(e1, e2)
        | RelationOperator(rel) ->
            match rel with
            | LessThan -> upcast Expression.LessThan(e1, e2)
            | LessThanOrEqual -> upcast Expression.LessThanOrEqual(e1, e2)
            | GreaterThan -> upcast Expression.GreaterThan(e1, e2)
            | GreaterThanOrEqual -> upcast Expression.GreaterThanOrEqual(e1, e2
        | Equals -> upcast Expression.Equal(e1, e2)
    | ...
```

Little DSL

```
let shouldEqual (value : QueryValue) (colName : ColumnName) =
            binaryOp colName value Equals

let shouldNotEqual (value : QueryValue) (colName : ColumnName) =
            Unary(colName |> shouldEqual value, Not)

let shouldBeSmallerThan (value : QueryValue) (colName : ColumnName) =
            RelationOperator(LessThan) |> binaryOp colName value

let shouldBeSmallerEqualThan (value : QueryValue) (colName : ColumnName) =
            RelationOperator(LessThanOrEqual) |> binaryOp colName value

let (===) a b = a |> shouldEqual b
let (=/=) a b = a |> shouldNotEqual b
```

Result

- Code more maintainable
- Code more expressive
- Code more concise (less bug)
- Removed dependencies
- Easy to expand
  - First transformer Linq.Expression ☹
  - Cassandra


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO


That's all Folks!

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

contacts

Twitter	*@trikace*
Blog	*www.rickyterrell.com*
Email	*tericcardo@gmail.com*
GitHub	*www.github.com/rikace/presentation*