Cut to the chase with Functional Programming

"I have no special talents. I am only passionately curious."

- Albert Einstein

Riccardo Terrell

Agenda

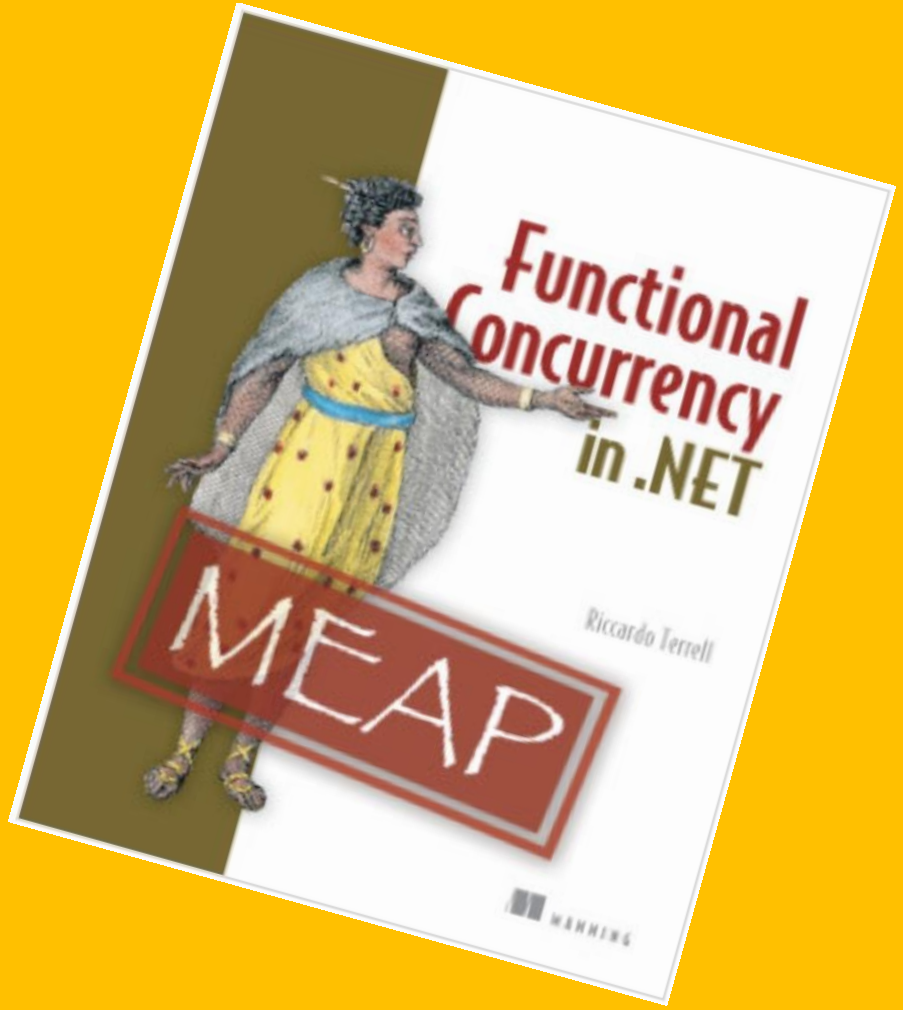
- What & Why Functional Programming
- Algebraic Data Type
- Real ! World example!!

Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC ~10 years
- +/- 19 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- DC F# User Group - Organizer
- *Polyglot programmer - believes in the art of finding the right tool for the job*

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

MVP Microsoft® Most Valuable Professional



Use code coupon

fcnetmu

for 40% off

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

Functional Programming

- declarative
- functions as values
- side-effects free

- referential transparency
- immutable
- composition

… 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)

FP 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) { … }
```

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”

Functional programming?

- **Functional programming is a style of programming**

  that enables you:

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

  - Eliminate bugs (via **immutability**)

IEnumerable is Functional

```
static IEnumerable<R> Select<T, R>(this IEnumerable<T> source, IEnumerable<T, T> selector)

static IEnumerable<R> SelectMany<T, R>(this IEnumerable<T> source,
                                       Func<T, IEnumerable<R>> selector)
```

```
var result =        from a in Enumerable.Range(1,10)
                    from b in Enumerable.Range(20, 30)
                    select a + b;
```

Task is Functional

```
static Task<R> Select<T, R>(this Task<T> source, Func<T, T> selector)

static Task<R> SelectMany<T, R>(this Task<T> source, Func<T, Task<R>> selector)
```

```
int result =        from a in Task.Run(() => 40)
                    from b in Task.Run(() => 42)
                    select a + b;
```

Algebraic data type

Composite type

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” }
```

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 let

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

Let’s get real

Real production code

http://www.statmuse.com

Take away

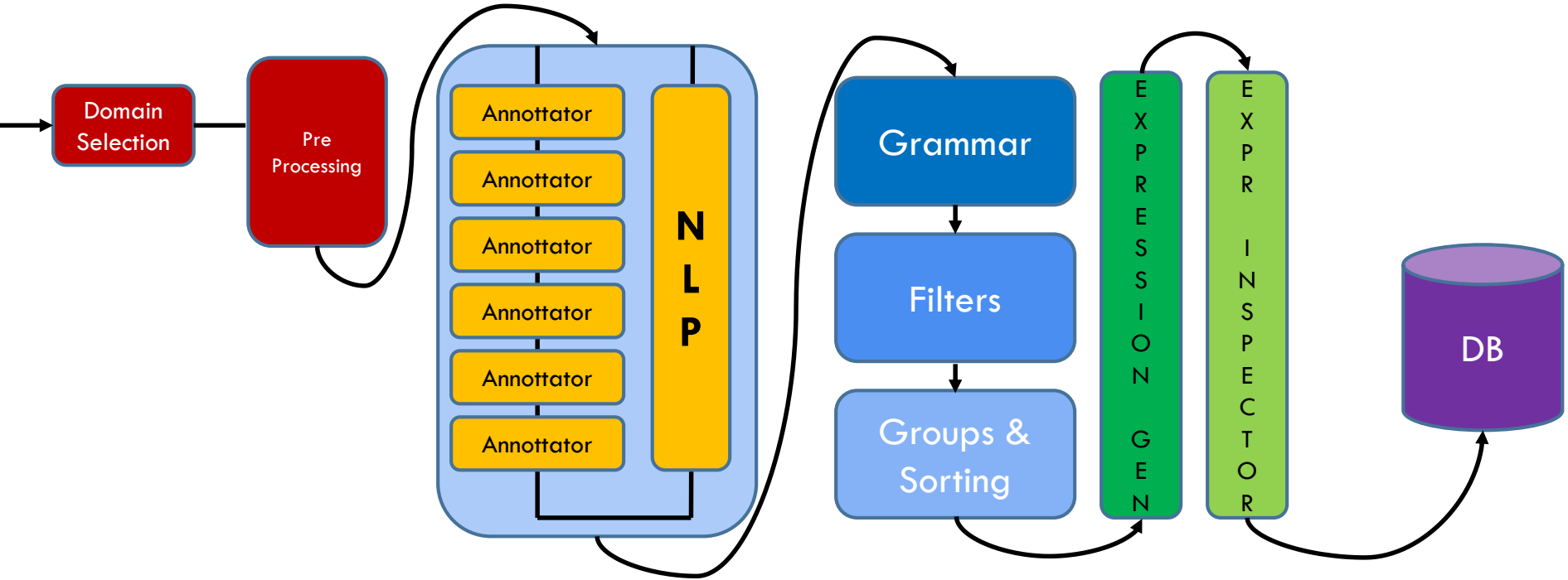
"Problems cannot be solved with the same mind set that created them"

— 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
```

```
// 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).ToAr

            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.te
        var gameIds = games.Select(i => new { team_id = i.Key, Games = (isLast ? i.OrderByDescending(x => x.game_date) : i.OrderBy(x => x.game_date)).Tak

        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

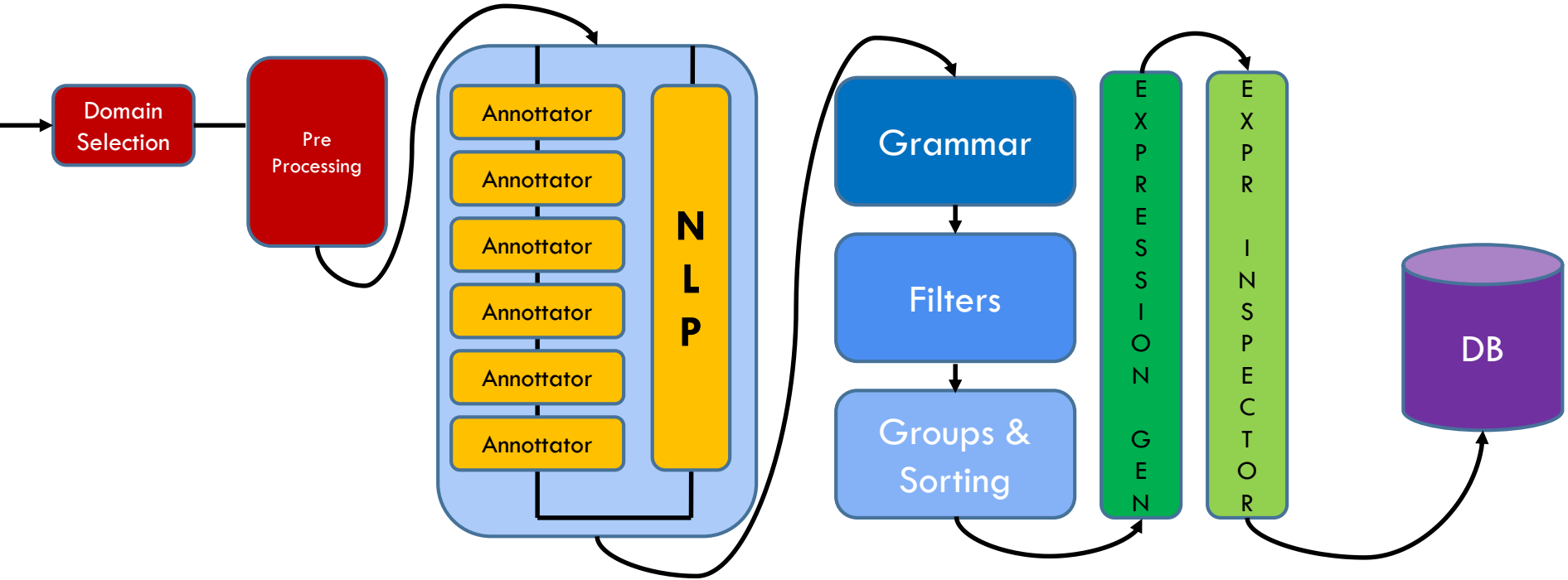
```
public class EfSchemaInspector : ISchemaInspector
    {
        private readonly IParameterCollection _parameterCollection;
        private readonly IDictionary<Type, Dictionary<string, Func<ParameterExpression,
MemberExpression>>> _memberMaps = new Dictionary<Type, Dictionary<string,
Func<ParameterExpression,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

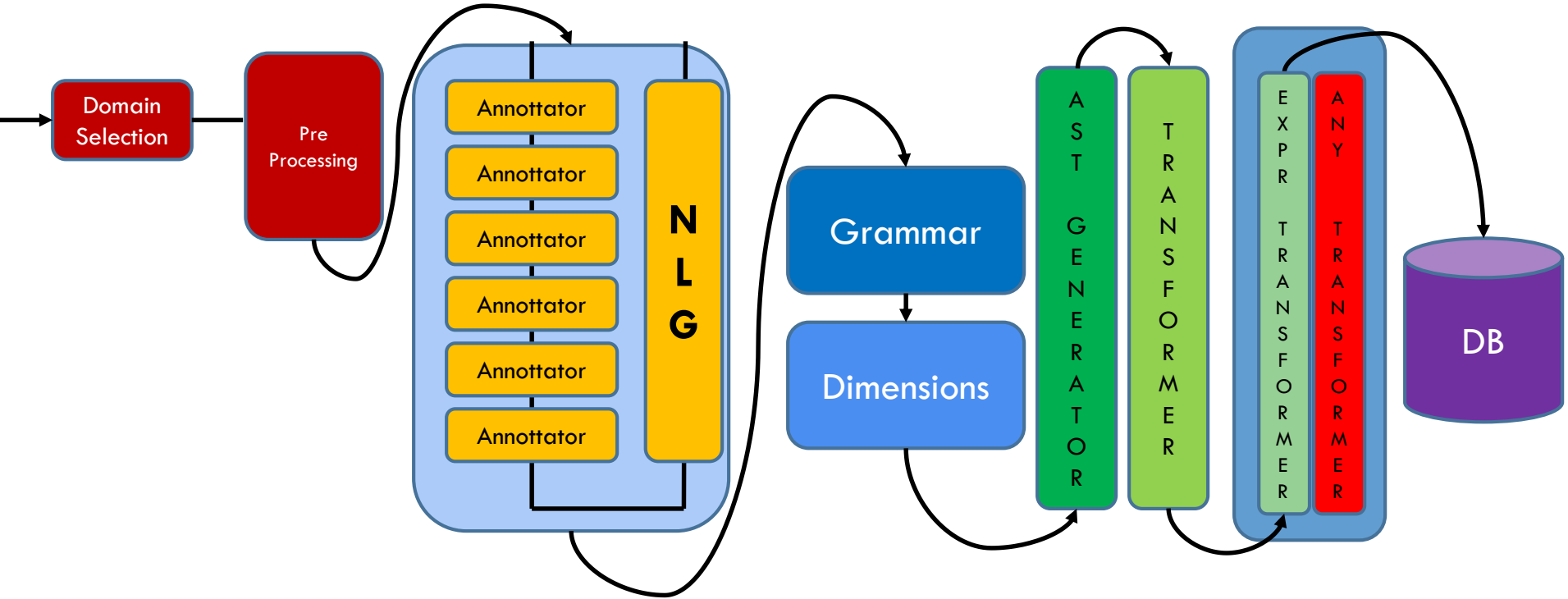
```
public class EfSchemaInspector : ISchemaInspector
    {
        private readonly IParameterCollection _parameterCollection;
        private readonly IDictionary<Type, Dictionary<string, Func<ParameterExpression,
MemberExpression>>> _memberMaps = new Dictionary<Type, Dictionary<string,
Func<ParameterExpression,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

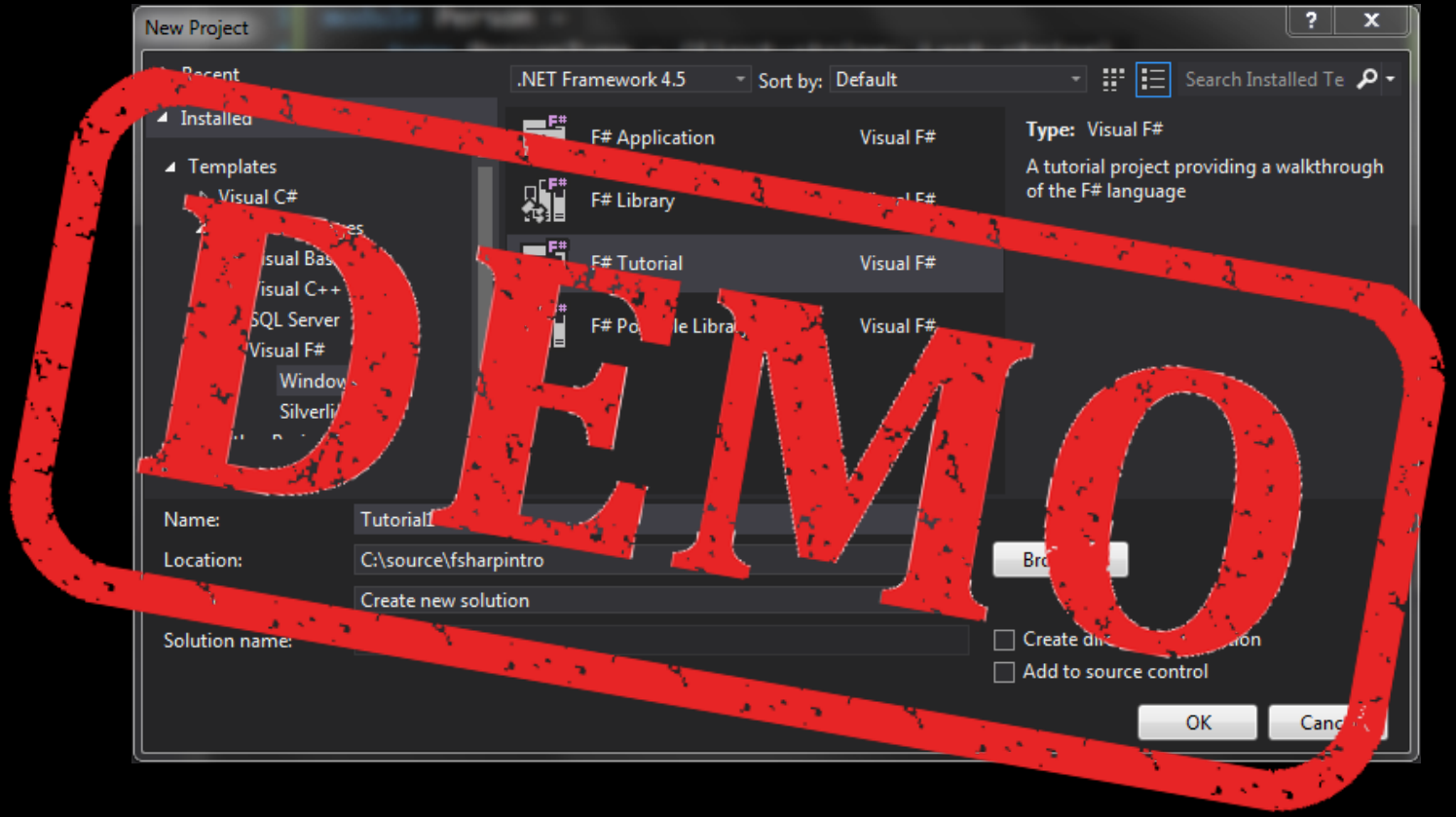
```
let rec convertExpression ctx e =
    match e with
    | GetColumn(s) -> ctx.ColumnGetter s
    | Constant(String v) -> upcast Expression.Constant(v)
    | Constant(Number v) -> upcast Expression.Constant(v)
    | Constant(Boolean v) -> upcast Expression.Constant(v)
    | Binary(e1, e2, op) ->
        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#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
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