Say Goodbye to the For Loop with High Order Functions

Mike Harris

Pop Quiz

What do these do?

```
main ()
{
  int i, j, k, l;
  float x[8][2][8][2];


  for (i = 0; i < 8; i++)
    for (j = i; j < 8; j++)
      for (k = 0; k < 2; k++)
  for (l = 0; l < 2; l++)
    {
      if ((i == j) && (k == l))
        x[i][k][j][l] = 0.8;
      else
        x[i][k][j][l] = 0.8;
      if (x[i][k][j][l] < 0.0)
        abort ();
    }


  exit (0);
}
```

C

```c
int f(void)
{
  static _Complex double t;
  int i, j;
  for(i = 0;i<2;i++)
    for(j = 0;j<2;j++)
      t = .5 * 1.0;
  return t;
}
```

C

```c
int n;


void foo (int i)
{
  int a, b;


  if (!i)
    for (a = 1; a < 4; a++)
      if (a)
  for (b = 1; b < 3; b++)
    foo (b);


  n++;
}
```

C

I have no idea either.

Agenda

- Problem
- Map
- Filter
- Fold
- Theory

Problem Statment

Realistic Data

```
IList<(int Zip, double Price, int Quantity)> orders =
    new List<(int Zip, double Price, int Quantity)> {
        (53202, 1.89, 3),
        (60191, 1.99, 2),
        (60060, 0.99, 7),
        (53202, 1.29, 8),
        (60191, 1.89, 2),
        (53202, 0.99, 3)
    };
```

C#

Find the total for 53202?

For Loop

```
var total = 0.0;
for (int i = 0; i < orders.Count(); i++)
{
    if (orders[i].Zip == 53202)
        total += orders[i].Price * orders[i].Quantity;
}
```

C#

Foreach Loop

```
var total = 0.0;
foreach (var order in orders)
{
    if (order.Zip == 53202)
        total += order.Price * order.Quantity;
}
```

C#

Parts of the For Loop

Foreach Loop

```
var total = 0.0;
foreach (var order in orders)
{
    if (order.Zip == 53202)
        total += order.Price * order.Quantity;
}
```

C#

Foreach Loop

```
var total = 0.0;
foreach (var order in orders)
{
    if (Predicate)
        total += order.Price * order.Quantity;
}
```

C#

Foreach Loop

```
var total = 0.0;
foreach (var order in orders)
{
    if (Predicate)
        total += Mapping;
}
```

C#

Foreach Loop

```
var total = 0.0;
foreach (var order in orders)
{
    if (Predicate)
        Accumulate += Mapping;
}
```

C#

Foreach Loop

```
var Initial
foreach (var order in orders)
{
    if (Predicate)
        Accumulate += Mapping;
}
```

C#

LINQ

LINQ

```
var total = orders
    .Where(order => order.Zip == 53202)
    .Select(order => order.Price * order.Quantity)
    .Aggregate(0.0, (sub, amount) => sub + amount);
```

C#

LINQ

```
var total = orders
    .Where(Predicate)
    .Select(order => order.Price * order.Quantity)
    .Aggregate(0.0, (sub, amount) => sub + amount);
```

C#

LINQ

```
var total = orders
    .Where(Predicate)
    .Select(Mapping)
    .Aggregate(0.0, (sub, amount) => sub + amount);
```

C#

LINQ

```
var total = orders
     .Where(Predicate)
     .Select(Mapping)
     .Aggregate(0.0, Accumulate);
```

C#

LINQ

```
var total = orders
    .Where(Predicate)
    .Select(Mapping)
    .Aggregate(Initial, Accumulate);
```

C#

Compare

Compare

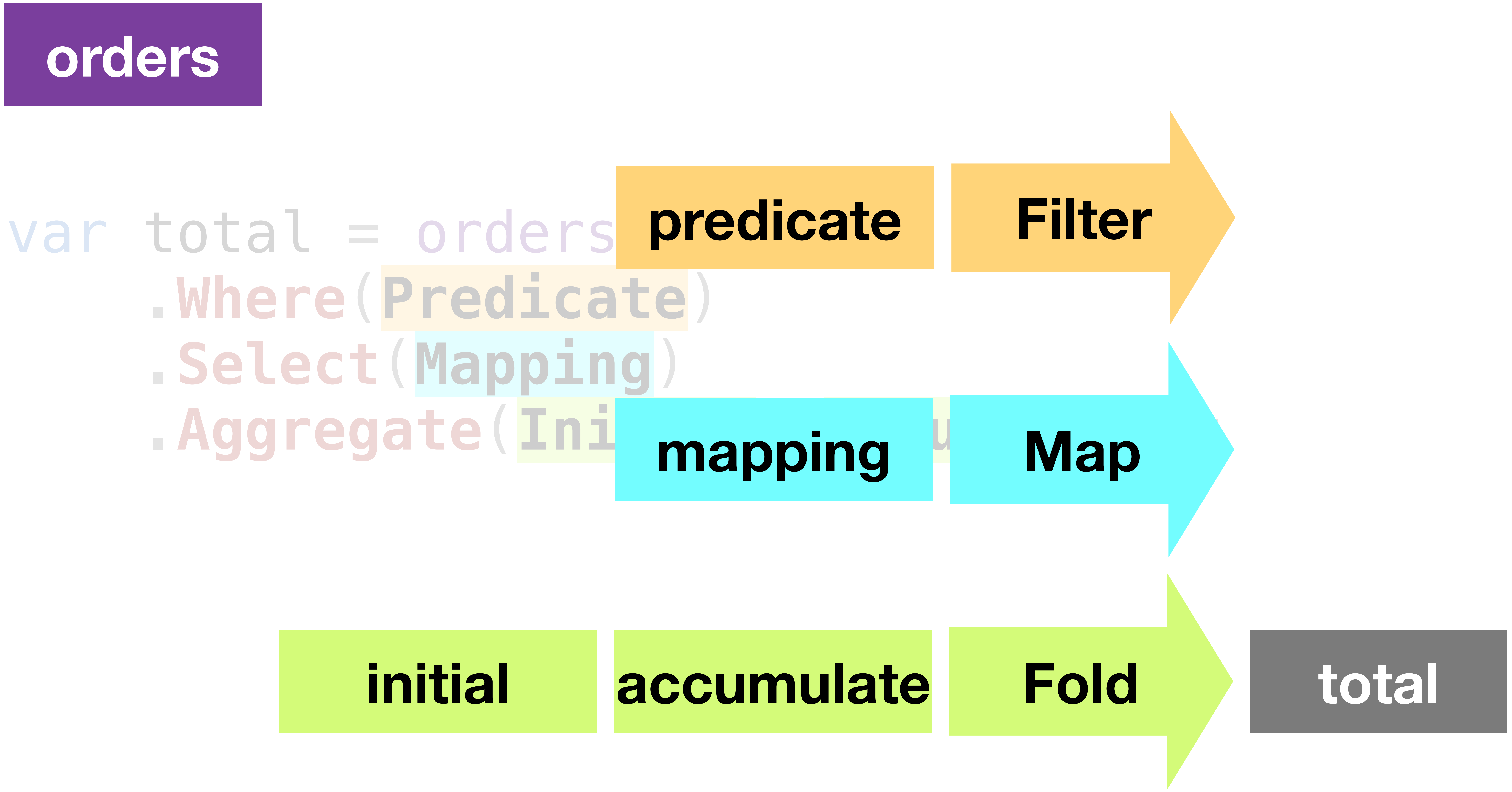
```
var total = orders
    .Where(Predicate)
    .Select(Mapping)
    .Aggregate(Initial, Accumulate);
```

```
var Initial
foreach (var order in orders)
{
    if (Predicate)
        Accumulate += Mapping;
}
```

C#

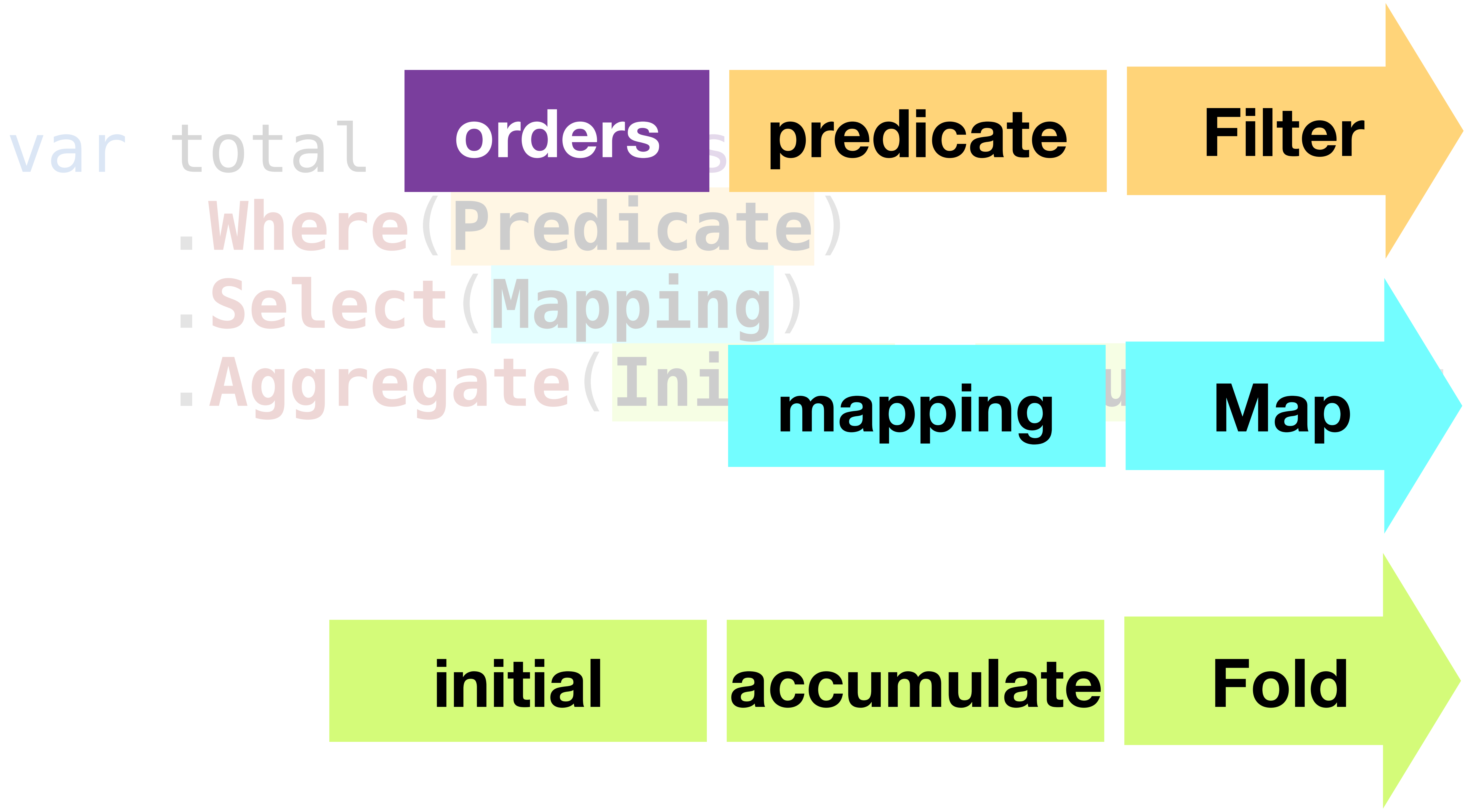
General

General



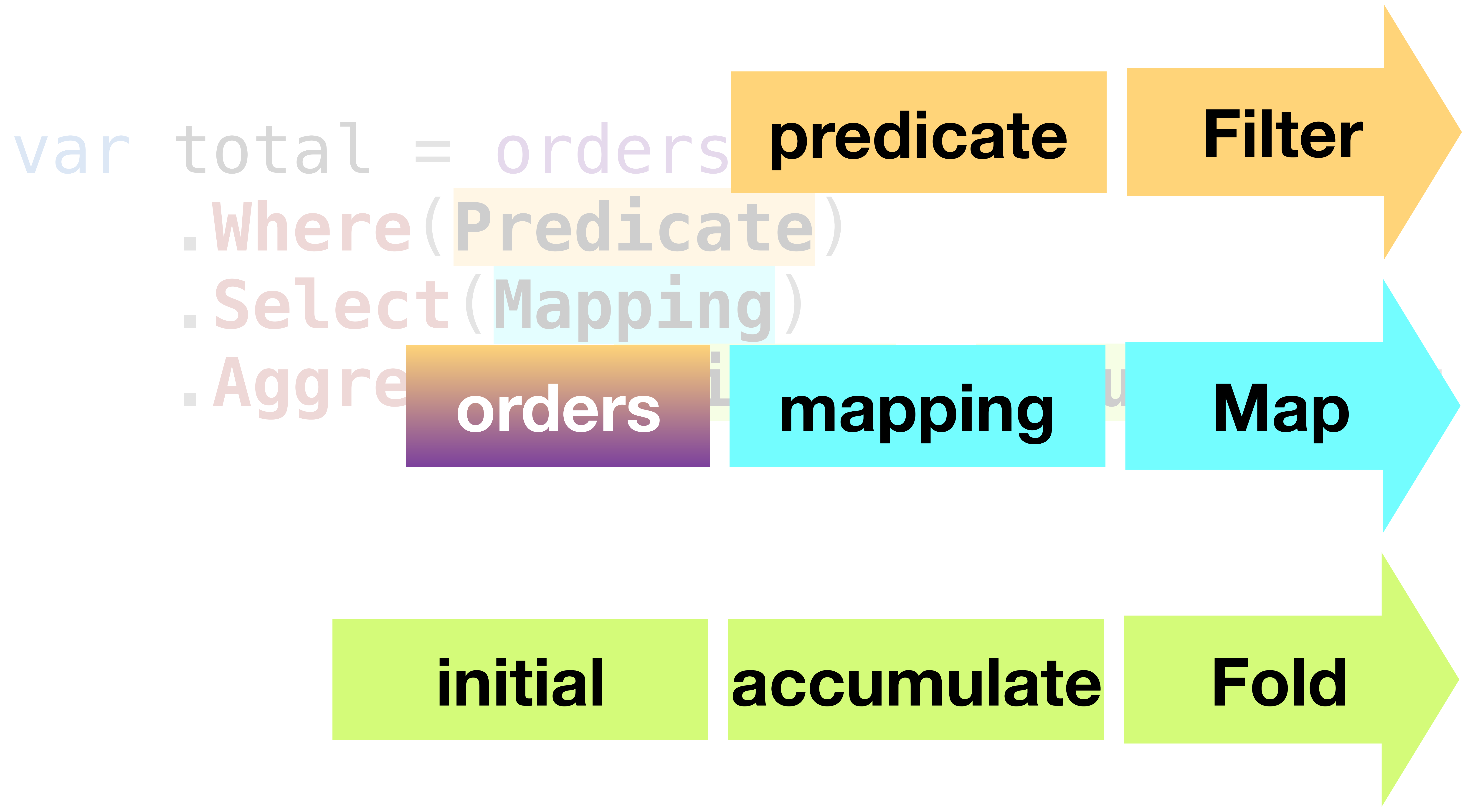
```
var total = orders
    .Where(Predicate)
    .Select(Mapping)
    .Aggregate(Ini
```

General



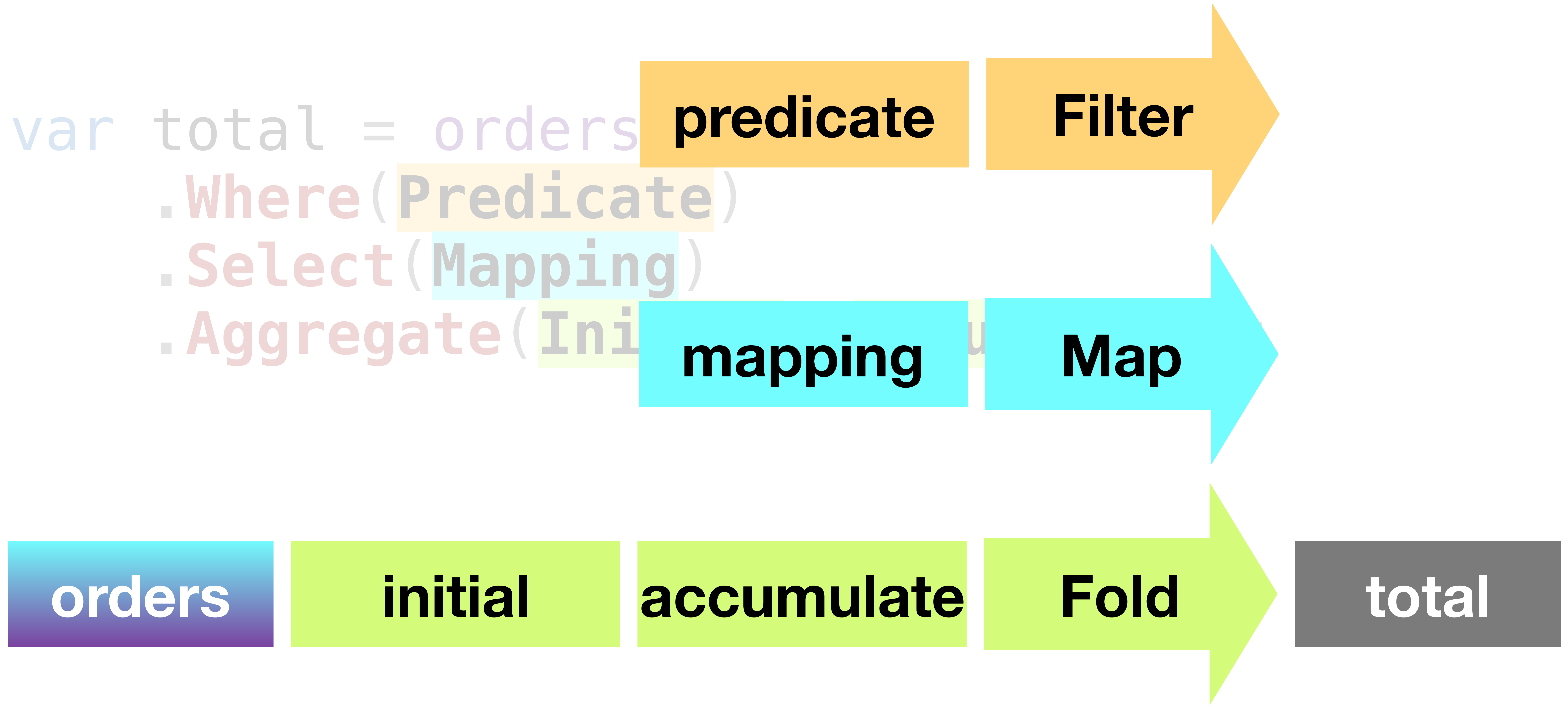
```
var total
    .Where(Predicate)
    .Select(Mapping)
    .Aggregate(Ini
```

General



```
var total = orders
    .Where(Predicate)
    .Select(Mapping)
    .Aggre
```

General



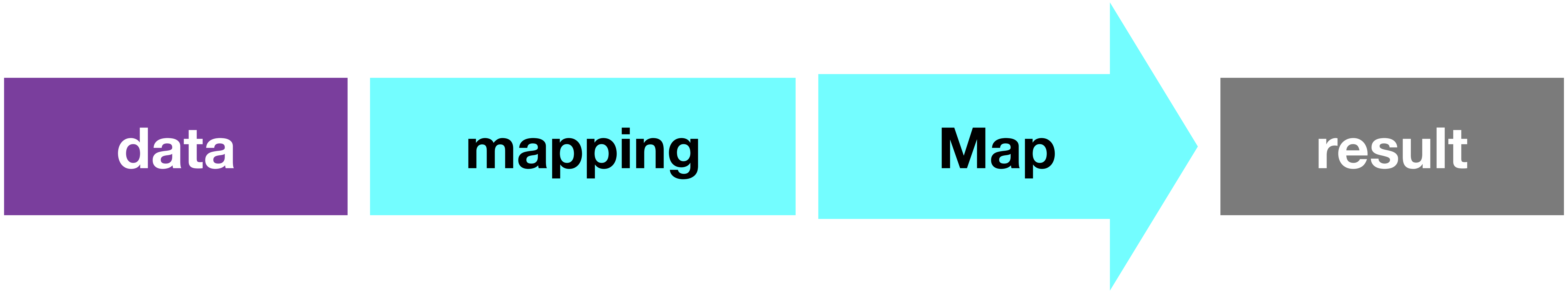
```
var total = orders
    .Where(Predicate)
    .Select(Mapping)
    .Aggregate(Ini
```

Higher Order Functions in General

Map

(a -> b) -> [a] -> [b]

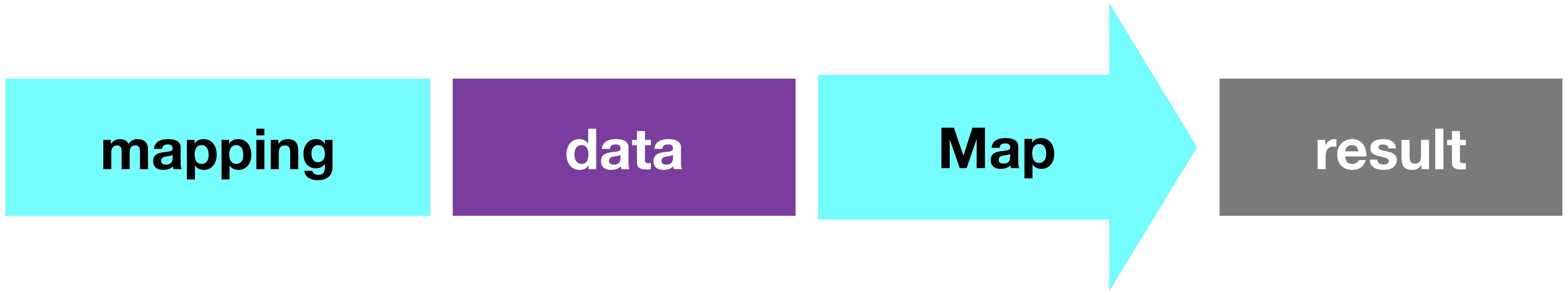
Map



Map

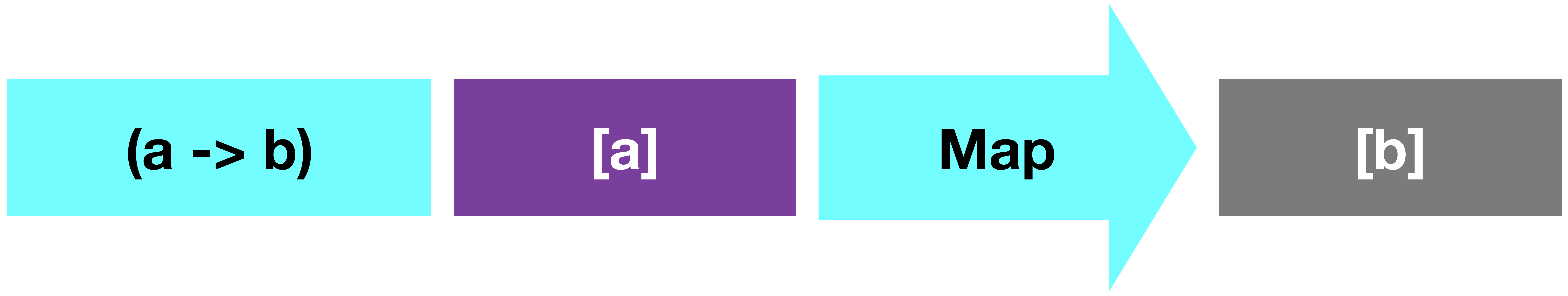
(a -> b) -> [a] -> [b]

Map



(a -> b) -> [a] -> [b]

Map



```csharp
IEnumerable<U> Map<T, U>(
    Func<T, U> mapping, IEnumerable<T> source)
{
    var result = new List<U>();
    foreach(var item in source)
    {
        result.Add(mapping(item));
    }

    return result;
}
```

C#

```
var result = new List<U>();
foreach(var item in source)
{
    result.Add(Mapping);
}
```

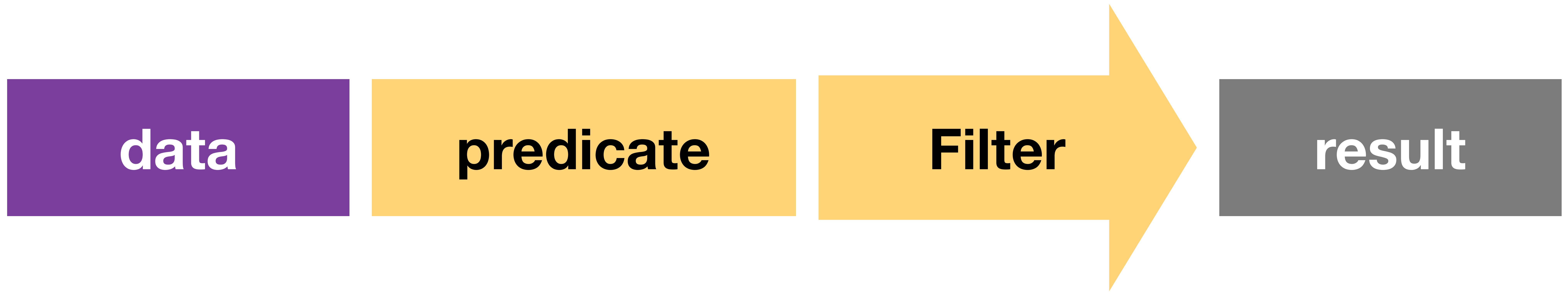
C#

Filter

(a -> bool) -> [a] -> [a]

C#

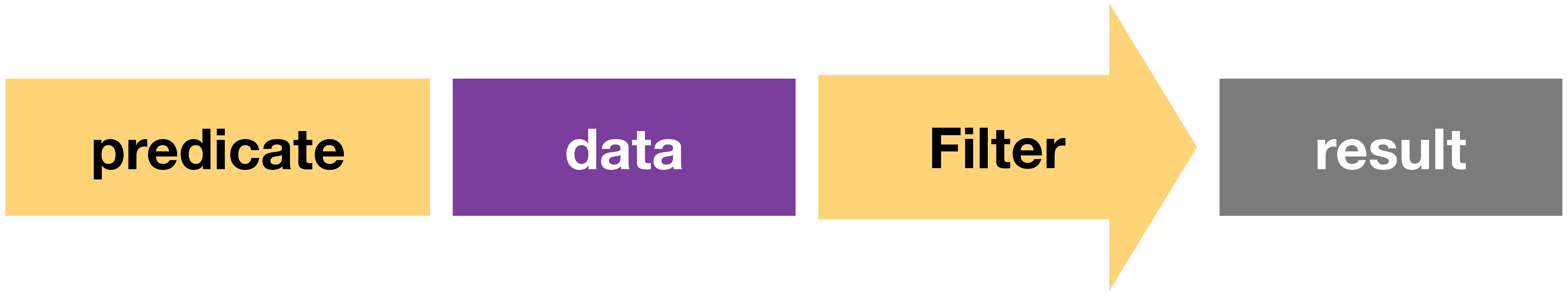
Filter



Filter

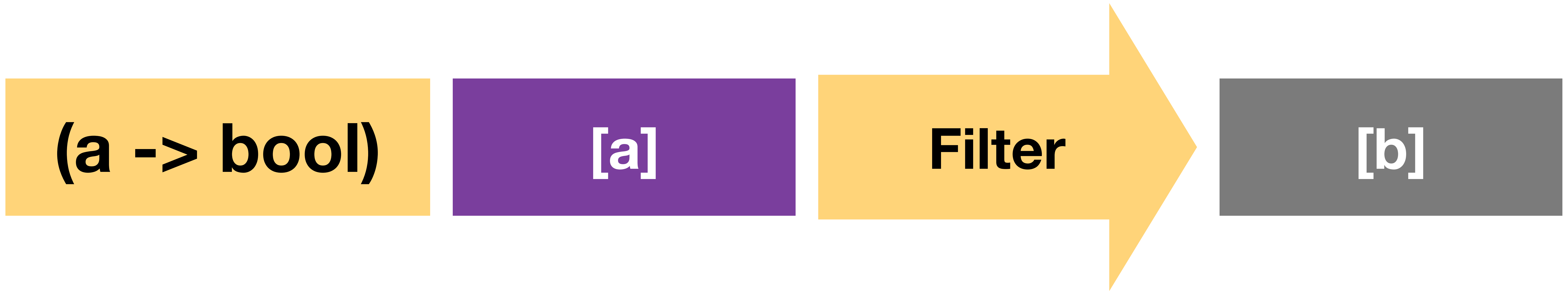
(a -> bool) -> [a] -> [a]

Filter



(a -> bool) -> [a] -> [a]

Filter



```csharp
IEnumerable<T> Filter<T>(
    Func<T, bool> predicate, IEnumerable<T> source)
{
    var result = new List<T>();
    foreach(var item in source)
    {
        if (predicate(item))
            result.Add(item);
    }

    return result;
}
```

C#

```
var result = new List<T>();
foreach(var item in source)
{
    if (Predicate)
        result.Add(item);
}
```

C#

Fold

(state -> a -> state) -> state -> [a] -> state

C#

Fold

data
initial
accumulate
Fold
result

Fold

```
(state -> a -> state) -> state -> [a] -> state
```

Fold

accumulate
initial
data
Fold
result

(state -> a -> state) -> state -> [a] -> state

Fold

(state -> a -> state)
state
[a]
Fold
state

```csharp
U Fold<T, U>(
    Func<U, T, U> accumulate, U initial,
    IEnumerable<T> source)
{
    var result = initial;
    foreach(var item in source)
    {
        result = accumulate(result, item);
    }

    return result;
}
```

C#

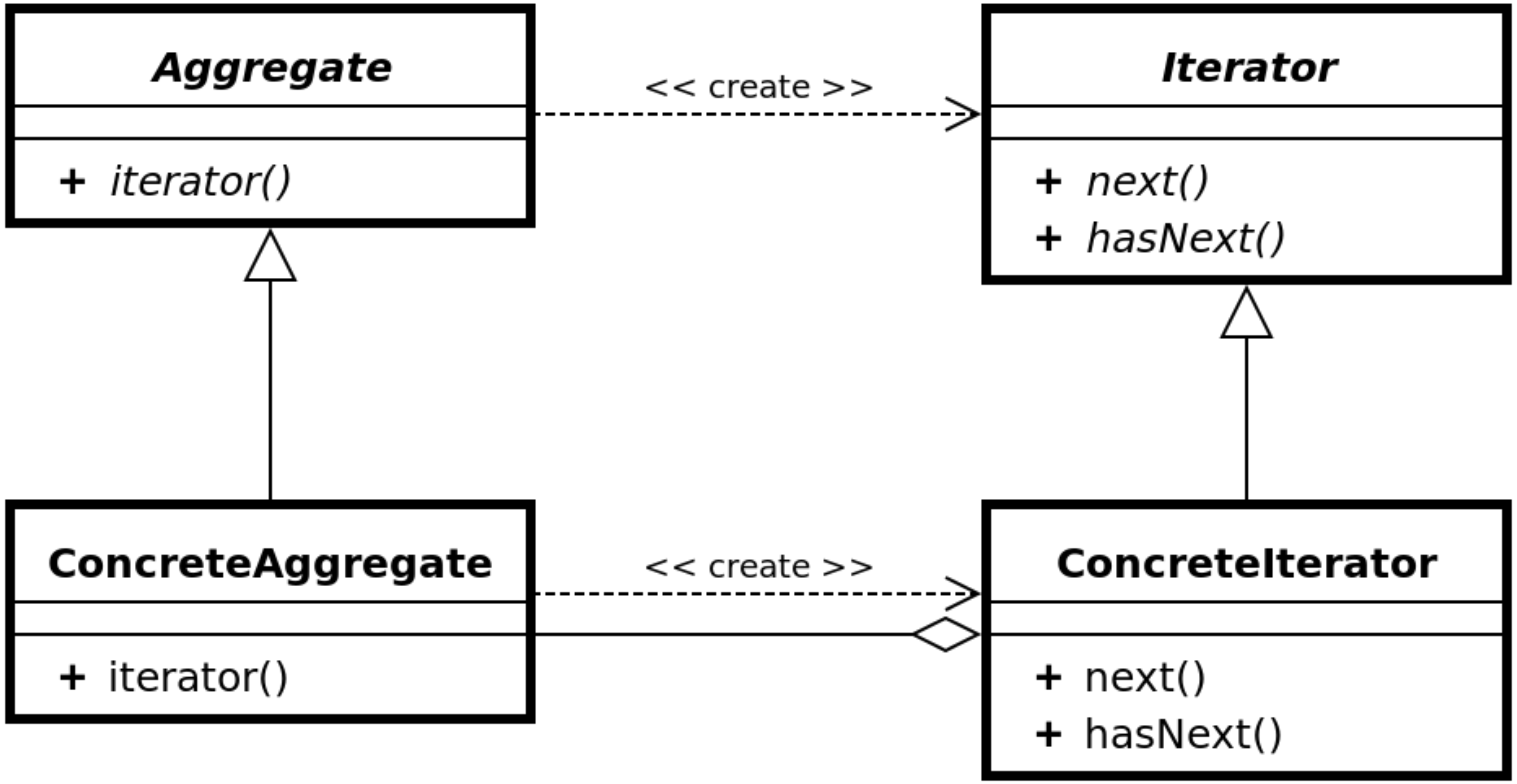
```
var result = Initial;
foreach(var item in source)
{
    result = Accumulate;
}
```

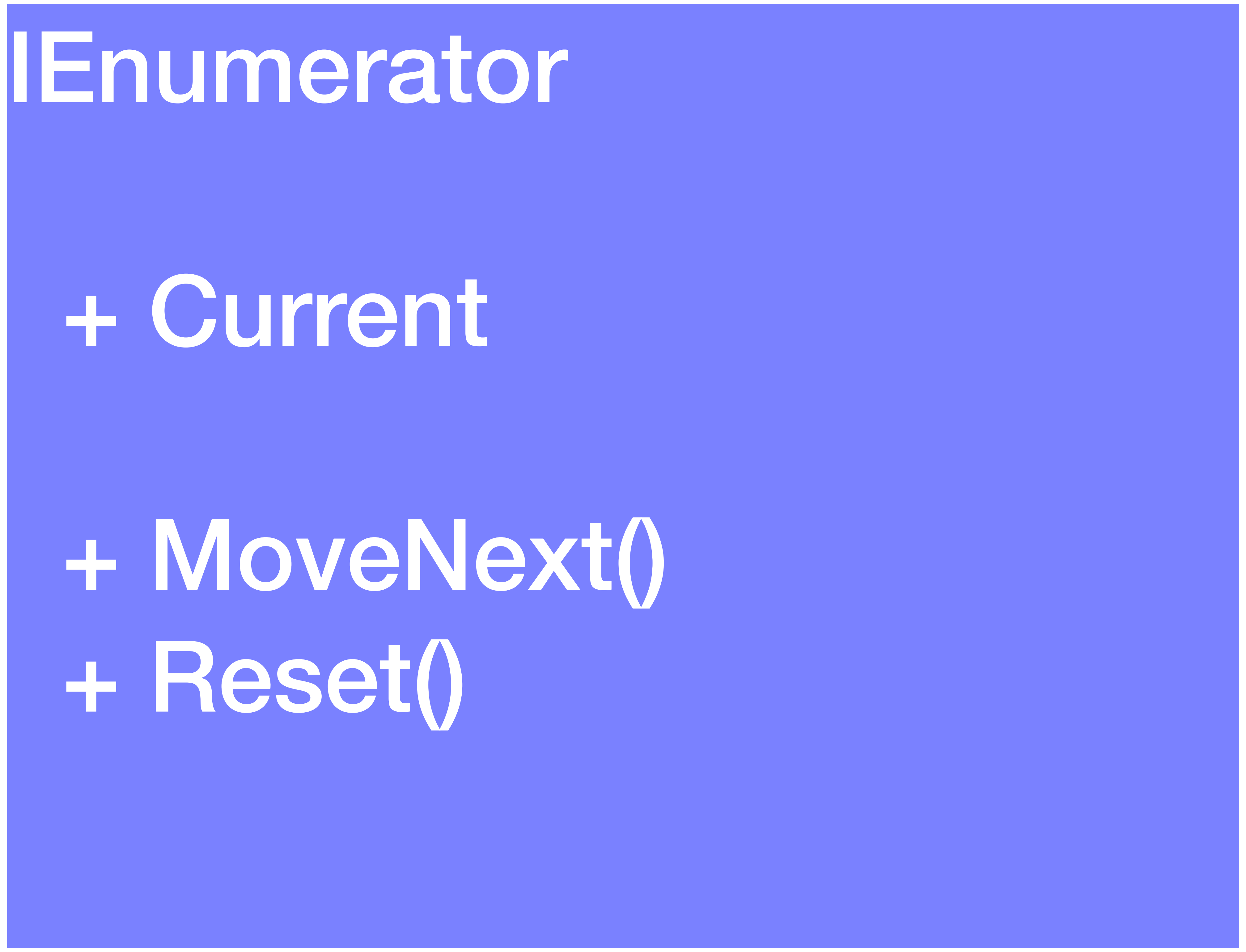
C#

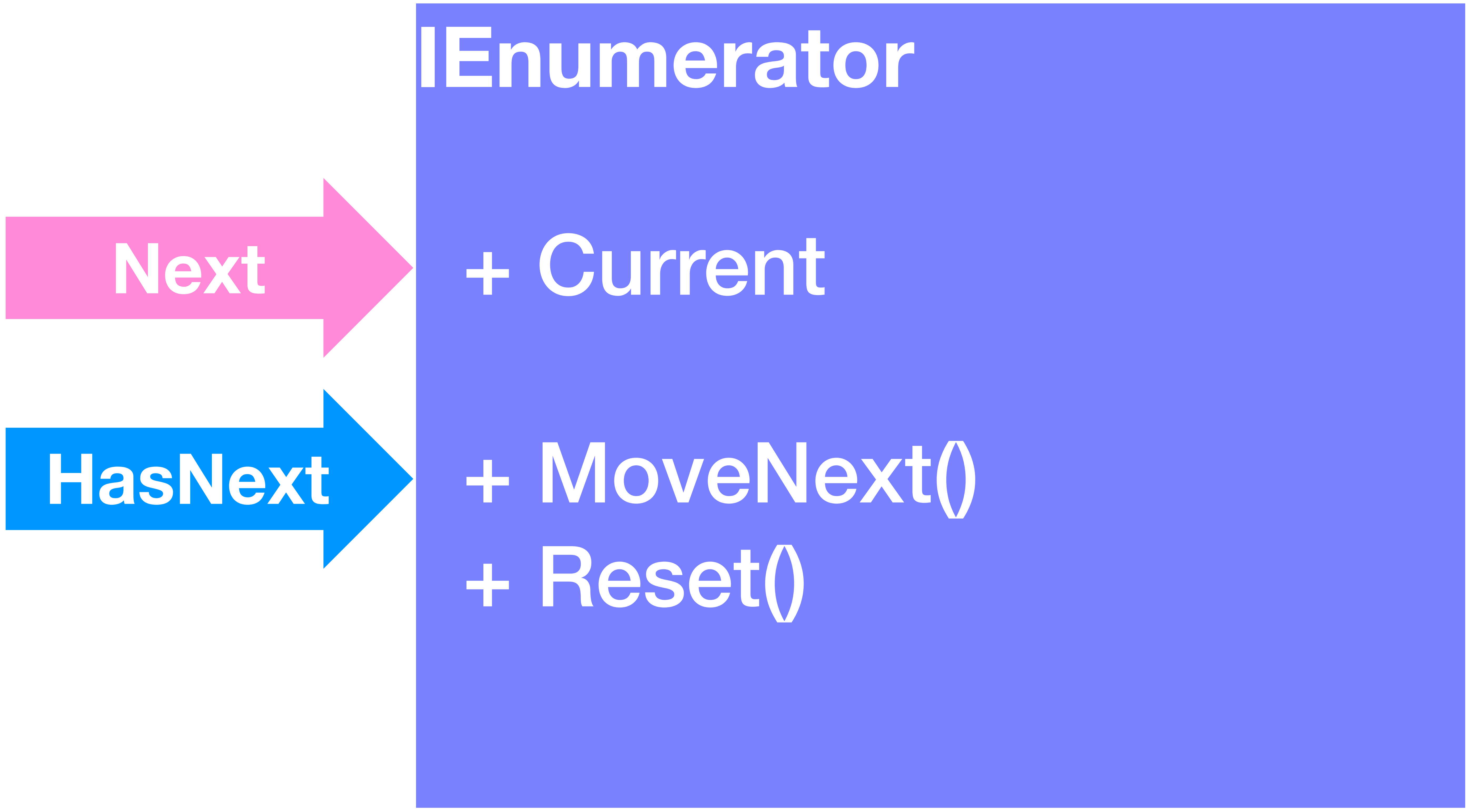
Higher Order Functions in C#

C#

Iterator Pattern


Aggregate
+ iterator()
<< create >>
Iterator
+ next()
+ hasNext()
ConcreteAggregate
+ iterator()
<< create >>
ConcreteIterator
+ next()
+ hasNext()


IEnumerator
+ Current
+ MoveNext()
+ Reset()


IEnumerator
+ Current
+ MoveNext()
+ Reset()
Next
HasNext

```
void Iterate<T>(Action<T> f, IEnumerator<T> source)
{
    while(source.MoveNext())
    {
        f(source.Current);
    }
}
```

```
void Iterate<T>(Action<T> f, IEnumerator<T> source)
{
    while(HasNext)
    {
        f(source.Current);
    }
}
```

```
void Iterate<T>(Action<T> f, IEnumerator<T> source)
{
    while(HasNext)
    {
        f(Next);
    }
}
```

```
void Iterate<T>(Function, IEnumerator<T> source)
{
    while(HasNext)
    {
        Function(Next);
    }
}
```

Map

(a -> b) -> [a] -> [b]

C#

```
private sealed class SelectArrayIterator<TSource, TResult> : Iterator<TResult>, IPartition<TResult>
{
  private readonly TSource[] _source;
  private readonly Func<TSource, TResult> _selector;

  public override bool MoveNext()
  {
      if (_state < 1 | _state == _source.Length + 1)
      {
          Dispose();
          return false;
      }

      int index = _state++ - 1;
      _current = _selector(_source[index]);
      return true;
  }
```

C#

```csharp
private sealed class SelectArrayIterator<TSource, TResult> : Iterator<TResult>, IPartition<TResult>
{
  private readonly TSource[] _source;
  private readonly Func<TSource, TResult> _selector;

  public override bool HasNext()
  {
      if (_state < 1 | _state == _source.Length + 1)
      {
          Dispose();
          return false;
      }

      int index = _state++ - 1;
      _current = _selector(_source[index]);
      return true;
  }
```

C#

```
private sealed class SelectArrayIterator<TSource, TResult> : Iterator<TResult>, IPartition<TResult>
{
  private readonly TSource[] _source;
  private readonly Func<TSource, TResult> _selector;

  public override bool HasNext()
  {
      if (_state < 1 | _state == _source.Length + 1)
      {
          Dispose();
          return false;
      }

      int index = _state++ - 1;
      Next = _selector(_source[index]);
      return true;
  }
```

C#

```
private sealed class SelectArrayIterator<TSource, TResult> : Iterator<TResult>, IPartition<TResult>
{
  private readonly TSource[] _source;
  Function

  public override bool HasNext()
  {
      if (_state < 1 | _state == _source.Length + 1)
      {
          Dispose();
          return false;
      }

      int index = _state++ - 1;
      Next = Function(_source[index]);
      return true;
  }
```

C#

```
public override bool HasNext()
{
    if (_state < 1 | _state == _source.Length + 1)
    {
        Dispose();
        return false;
    }

    int index = _state++ - 1;
    Next = Function(_source[index]);
    return true;
}
```

C#

Filter

(a -> bool) -> [a] -> [a]

C#

```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IIListProvider<TSource>
{
    private readonly TSource[] _source;
    private readonly Func<TSource, bool> _predicate;

    public override bool MoveNext()
    {
        int index = _state - 1;
        TSource[] source = _source;

        while (unchecked((uint)index < (uint)source.Length))
        {
            TSource item = source[index];
            index = _state++;
            if (_predicate(item))
            {
                _current = item;
                return true;
            }
        }

        Dispose();
        return false;
    }
```

C#

```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IIListProvider<TSource>
{
    private readonly TSource[] _source;
    private readonly Func<TSource, bool> _predicate;

    public override bool HasNext()
    {
        int index = _state - 1;
        TSource[] source = _source;

        while (unchecked((uint)index < (uint)source.Length))
        {
            TSource item = source[index];
            index = _state++;
            if (_predicate(item))
            {
                _current = item;
                return true;
            }
        }

        Dispose();
        return false;
    }
```

C#

```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IIListProvider<TSource>
{
    private readonly TSource[] _source;
    private readonly Func<TSource, bool> _predicate;

    public override bool HasNext()
    {
        int index = _state - 1;
        TSource[] source = _source;

        while (unchecked((uint)index < (uint)source.Length))
        {
            TSource item = source[index];
            index = _state++;
            if (_predicate(item))
            {
                Next = item;
                return true;
            }
        }

        Dispose();
        return false;
    }
```

C#

```csharp
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IIListProvider<TSource>
{
    private readonly TSource[] _source;
    Function

    public override bool HasNext()
    {
        int index = _state - 1;
        TSource[] source = _source;

        while (unchecked((uint)index < (uint)source.Length))
        {
            TSource item = source[index];
            index = _state++;
            if (Function(item))
            {
                Next = item;
                return true;
            }
        }

        Dispose();
        return false;
    }
```

C#

```
public override bool HasNext()
{
    while (index < source.Length))
    {
        TSource item = source[index];
        index = _state++;
        if (Function(item))
        {
            Next = item;
            return true;
        }
    }

    Dispose();
    return false;
}
```

C#

Fold

(state -> a -> state) -> state -> [a] -> state

C#

```csharp
public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,
TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func)
{
    if (source == null)
    {
        throw Error.ArgumentNull(nameof(source));
    }

    if (func == null)
    {
        throw Error.ArgumentNull(nameof(func));
    }

    TAccumulate result = seed;
    foreach (TSource element in source)
    {
        result = func(result, element);
    }


    return result;
}
```

C#

```
public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,
TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func)
{
    if (source == null)
    {
        throw Error.ArgumentNull(nameof(source));
    }

    if (func == null)
    {
        throw Error.ArgumentNull(nameof(func));
    }

    TAccumulate result = seed;
    HasNext (TSource element in source)
    {
        result = func(result, element);
    }


    return result;
}
```

C#

```
public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,
TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func)
{
    if (source == null)
    {
        throw Error.ArgumentNull(nameof(source));
    }

    if (func == null)
    {
        throw Error.ArgumentNull(nameof(func));
    }

    TAccumulate result = seed;
    HasNext (TSource Next in source)
    {
        result = func(result, Next);
    }


    return result;
}
```

C#

```
public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,
TAccumulate seed, Function)
{
    if (source == null)
    {
        throw Error.ArgumentNull(nameof(source));
    }

    if (func == null)
    {
        throw Error.ArgumentNull(nameof(func));
    }

    TAccumulate result = seed;
    HasNext (TSource Next in source)
    {
        result = Function(result, Next);
    }


    return result;
}
```

C#

```
TAccumulate result = seed;
HasNext (TSource Next in source)
{
    result = Function(result, Next);
}


return result;
```

C#

LINQ Execution

Code

```
var total = orders
    .Where(order => order.Zip == 53202)
    .Select(order => order.Price * order.Quantity)
    .Aggregate(0.0, (sub, amount) => sub + amount);
```

C#

Data

```csharp
IList<(int Zip, double Price, int Quantity)> orders =
    new List<(int Zip, double Price, int Quantity)> {
        (53202, 1.89, 3),
        (60191, 1.99, 2),
        (60060, 0.99, 7),
        (53202, 1.29, 8),
        (60191, 1.89, 2),
        (53202, 0.99, 3)
    };
```

C#

In what order does this execute?

```
var total = orders
    .Where(order => order.Zip == 53202)
    .Select(order => order.Price * order.Quantity)
    .Aggregate(0.0, (sub, amount) => sub + amount);
```

C#

In what order does this execute?

```
var spy = new List<string>();

orders
    .Where(order =>
       { spy.Add("filter"); return order.Zip == 53202; })
    .Select(order =>
       { spy.Add(“map");
         return order.Price * order.Quantity; })
    .Aggregate(0.0, (sub, amount) =>
       { spy.Add("fold"); return sub + amount; });
```

C#

Answer

```
new List<string> {
    "filter", "map", "fold",
    "filter",
    "filter",
    "filter", "map", "fold",
    "filter",
    "filter", "map", "fold"
},
```

C#

Does exactly what we want

```
var total = orders
    .Where(order => order.Zip == 53202)
    .Select(order => order.Price * order.Quantity)
    .Aggregate(0.0, (sub, amount) => sub + amount);
```

C#

Higher Order Functions in F#

F#

Map

(a -> b) -> [a] -> [b]

F#

```
let map mapping x =
    match x with
    | [] -> []
    | [h] -> [mapping h]
    | h::t ->
        let cons = freshConsNoTail (mapping h)
        mapToFreshConsTail cons mapping t
        cons
```

F#

Filter

(a -> bool) -> [a] -> [a]

```
let rec filter predicate l =
    match l with
    | [] -> l
    | h :: ([] as nil) -> if predicate h then l else nil
    | h::t ->
        if predicate h then
            let cons = freshConsNoTail h
            filterToFreshConsTail cons predicate t
            cons
        else
            filter predicate t
```

F#

Fold

(state -> a -> state) -> state -> [a] -> state

F#

```fsharp
let fold<'T,'State> folder (state:'State) (list: 'T list) =
    match list with
    | [] -> state
    | _ ->
        let f = OptimizedClosures.FSharpFunc<_,_,_>.Adapt(folder)
        let mutable acc = state
        for x in list do
            acc <- f.Invoke(acc, x)
        acc
```

Theory

Thank you

Next Steps

Images

- UML Iterator Pattern, By Trashtoy - My own work written with text editor., Public Domain, https://commons.wikimedia.org/w/index.php?curid=1698830

gcc Source Code

- example 1, https://github.com/gcc-mirror/gcc/blob/e11be3ea01eaf8acd8cd86d3f9c427621b64e6b4/gcc/testsuite/gcc.c-torture/execute/930614-2.c#L1-L20
- example 2, https://github.com/gcc-mirror/gcc/blob/e11be3ea01eaf8acd8cd86d3f9c427621b64e6b4/gcc/testsuite/gcc.c-torture/compile/pr25513.c#L1-L9
- example 3, https://github.com/gcc-mirror/gcc/blob/e11be3ea01eaf8acd8cd86d3f9c427621b64e6b4/gcc/testsuite/gcc.c-torture/compile/pr43186.c#L1-L15

LINQ Source Code

- Select, https://github.com/dotnet/corefx/blob/a673a117846205fc1a5c648c29451ff3da83554d/src/System.Linq/src/System/Linq/Select.cs#L199-L226

- Where, https://github.com/dotnet/corefx/blob/a673a117846205fc1a5c648c29451ff3da83554d/src/System.Linq/src/System/Linq/Where.cs#L198-L255

- Aggregate, https://github.com/dotnet/corefx/blob/a673a117846205fc1a5c648c29451ff3da83554d/src/System.Linq/src/System/Linq/Aggregate.cs#L40-L59

F# Source Code

- map, https://github.com/Microsoft/visualfsharp/blob/f62158bae5a300be60abf3d97ae7cb4f83e7267d/src/fsharp/FSharp.Core/local.fs#L247-L254
- filter, https://github.com/Microsoft/visualfsharp/blob/f62158bae5a300be60abf3d97ae7cb4f83e7267d/src/fsharp/FSharp.Core/local.fs#L490-L500
- fold, https://github.com/fsharp/fsharp/blob/e19ddca7d6049ae04cc6a827e803555285d19b26/src/fsharp/FSharp.Core/list.fs#L214-L222