

# 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);
}
```

```
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;
}
```

```
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++;  
}
```

**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)  
    };
```

**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;
}
```

# Foreach Loop

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

# 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;
}
```

# Foreach Loop

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



# Foreach Loop

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

# Foreach Loop

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

# Foreach Loop

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

**LINQ**

# LINQ

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

# LINQ

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

# LINQ

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

# LINQ

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



# LINQ

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

# Compare

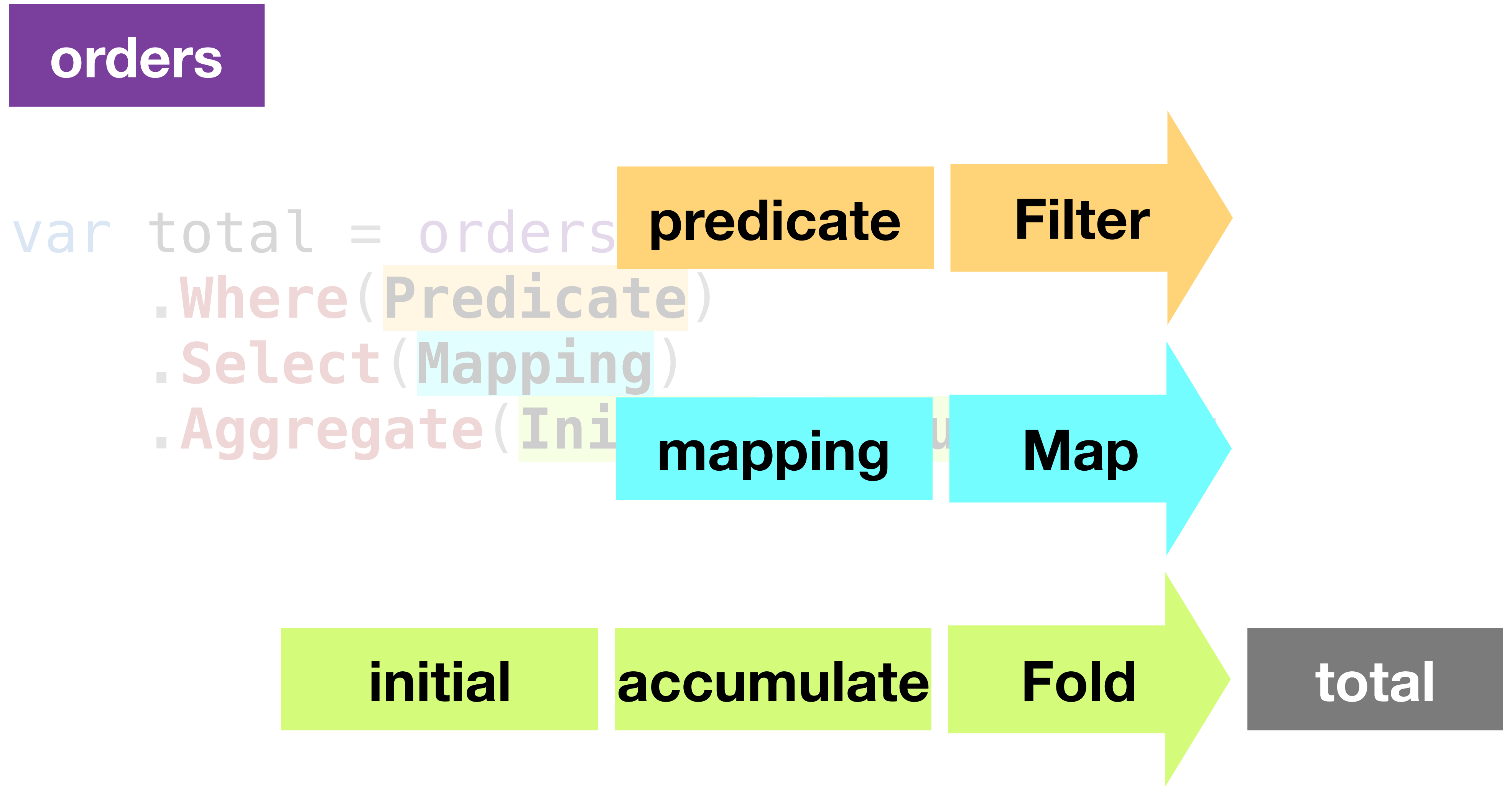
# Compare

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

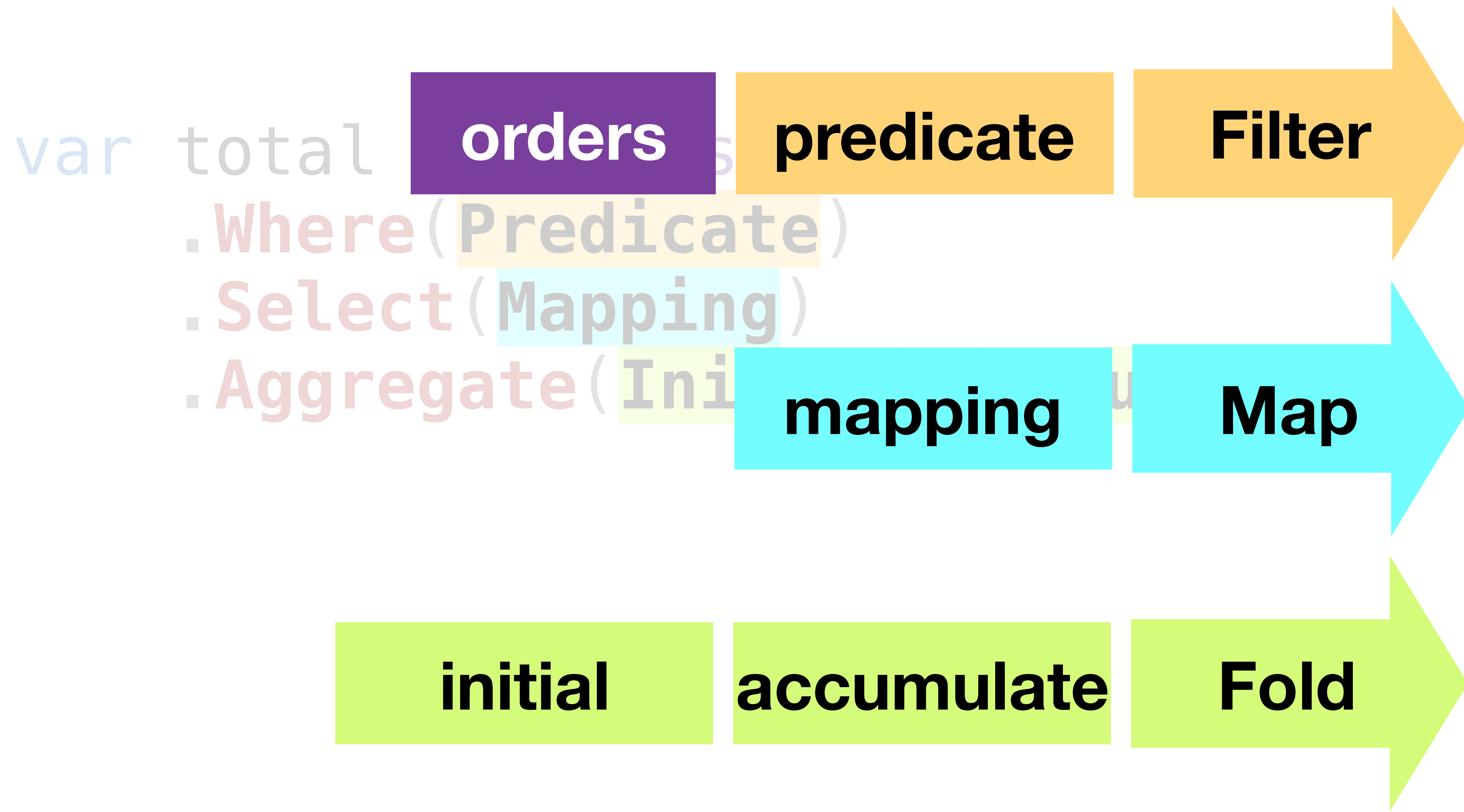
```
var Initial
foreach (var order in orders)
{
    if (Predicate)
        Accumulate += Mapping;
}
```

# Flow

# Flow



# Flow



# Flow

```
var total = orders  
    .Where(Predicate)  
    .Select(Mapping)  
    .Aggregate(i, u)
```

**predicate**

**Filter**

**orders**

**mapping**

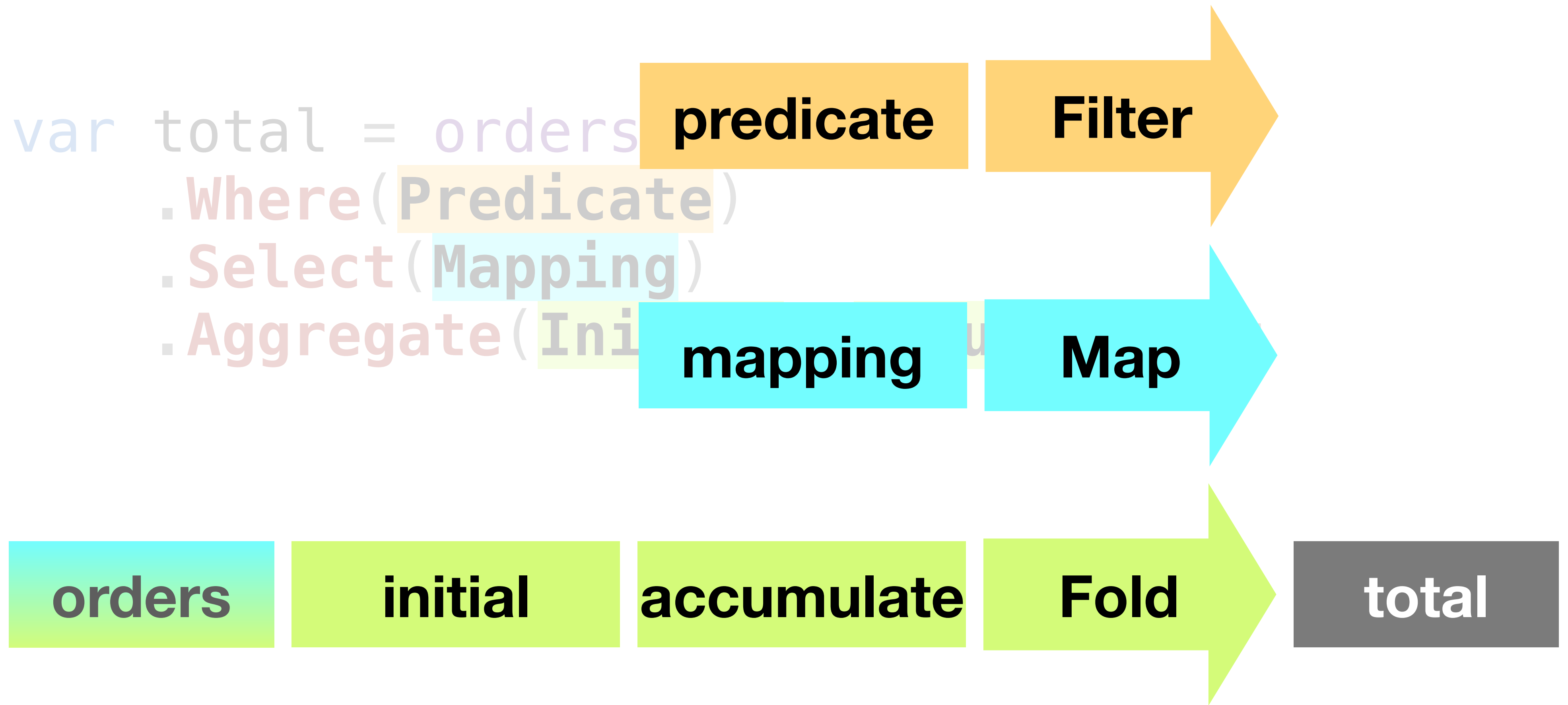
**Map**

**initial**

**accumulate**

**Fold**

# Flow



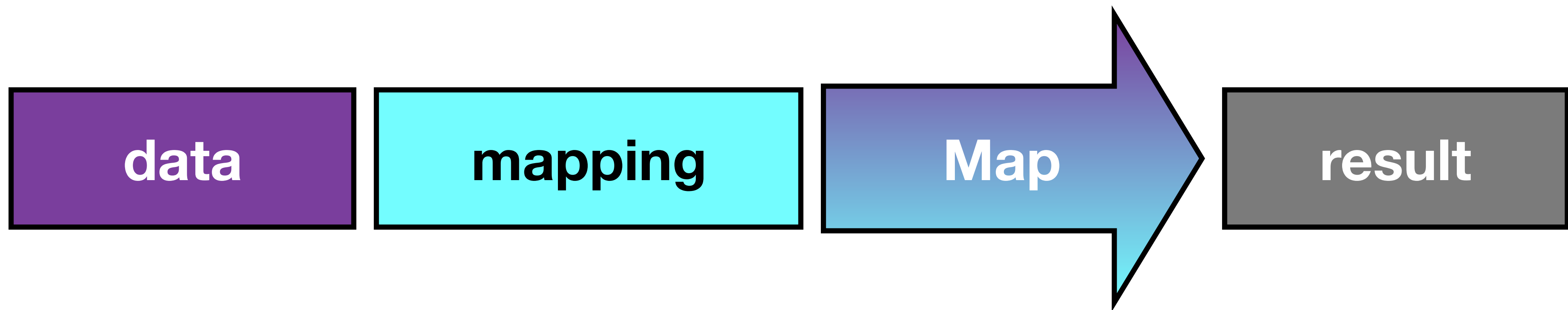


# Higher Order Functions

# Map

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

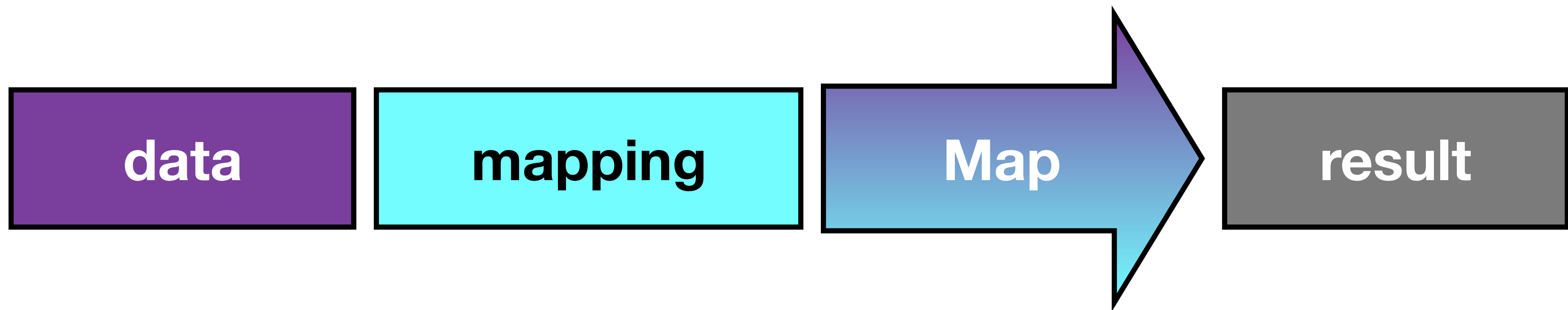
# Map



# Map

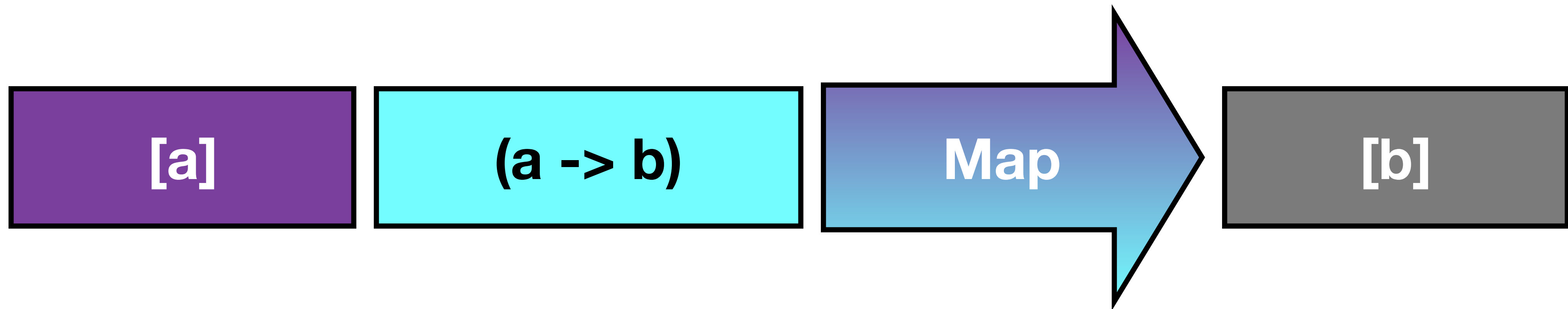
**$[a] \rightarrow (a \rightarrow b) \rightarrow [b]$**

# Map



**$[a] \rightarrow (a \rightarrow b) \rightarrow [b]$**

# Map



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

    return result;
}
```

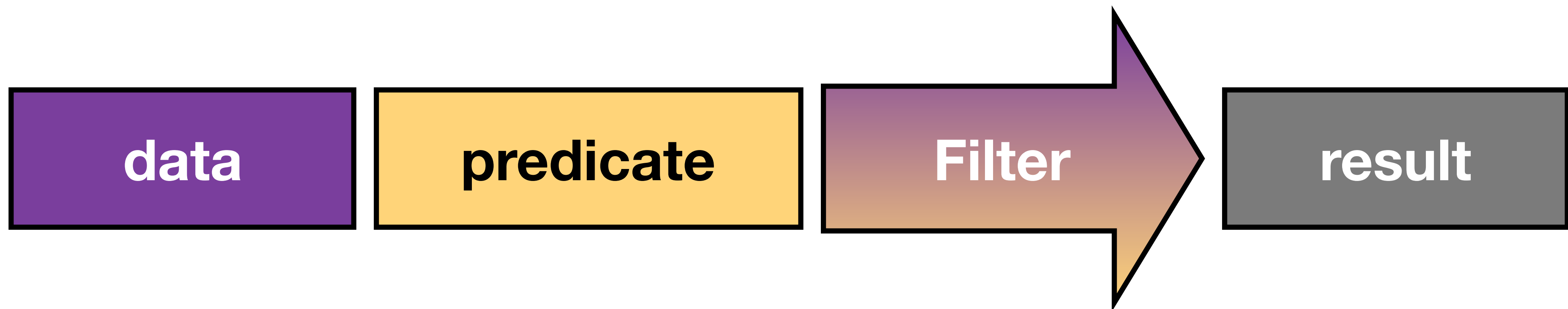
```
var result = new List<U>();  
foreach(var item in source)  
{  
    result.Add(Mapping);  
}
```



# Filter

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

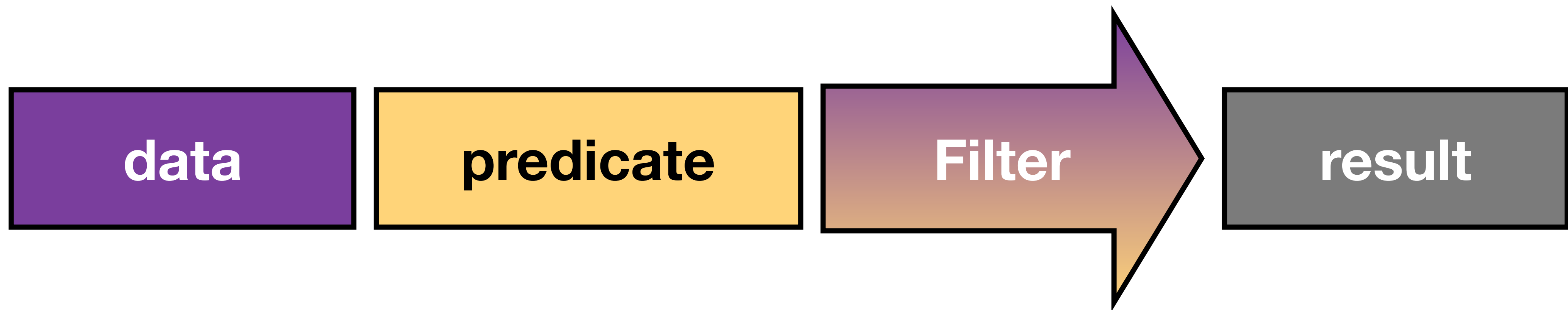
# Filter



# Filter

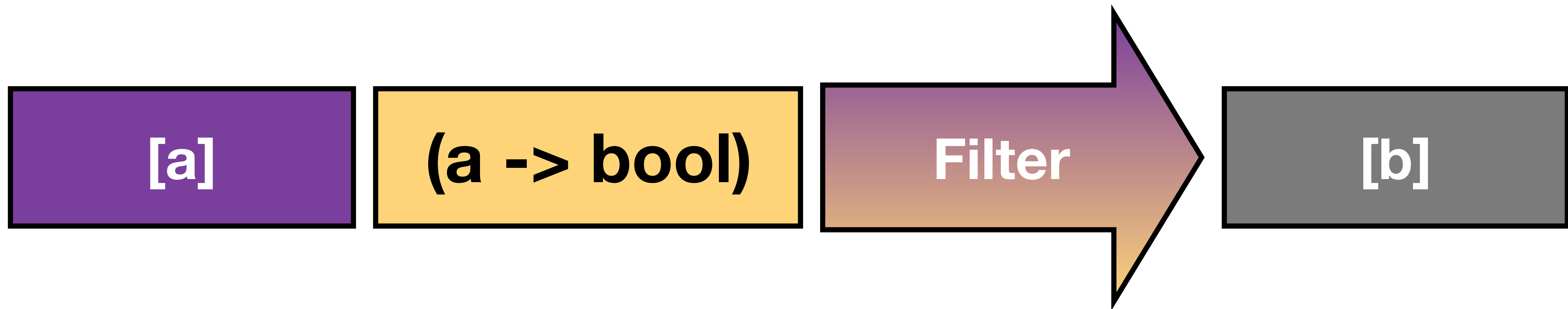
**$[a] \rightarrow (a \rightarrow \text{bool}) \rightarrow [a]$**

# Filter



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

# Filter



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

    return result;
}
```

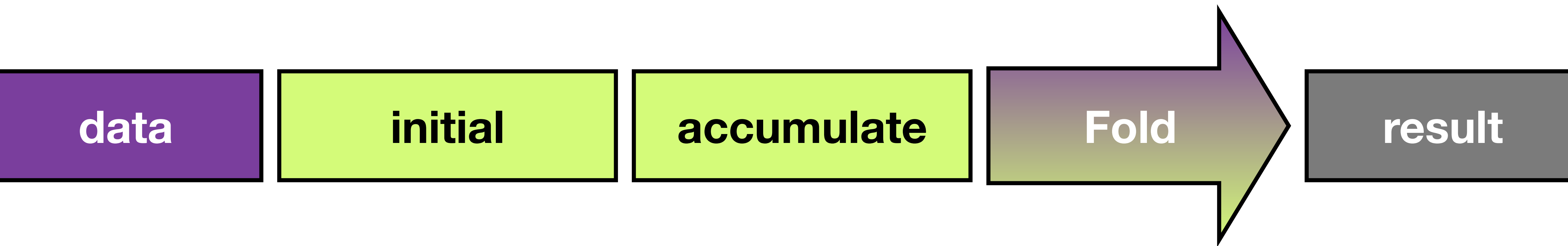
```
var result = new List<T>();  
foreach(var item in source)  
{  
    if (Predicate)  
        result.Add(item);  
}
```

# Fold

**$[a] \rightarrow \text{state} \rightarrow (\text{state} \rightarrow a \rightarrow \text{state}) \rightarrow \text{state}$**



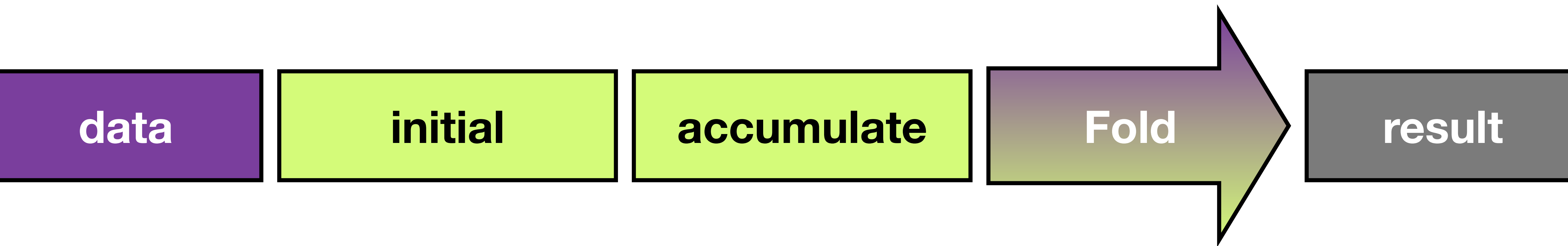
# Fold



# Fold

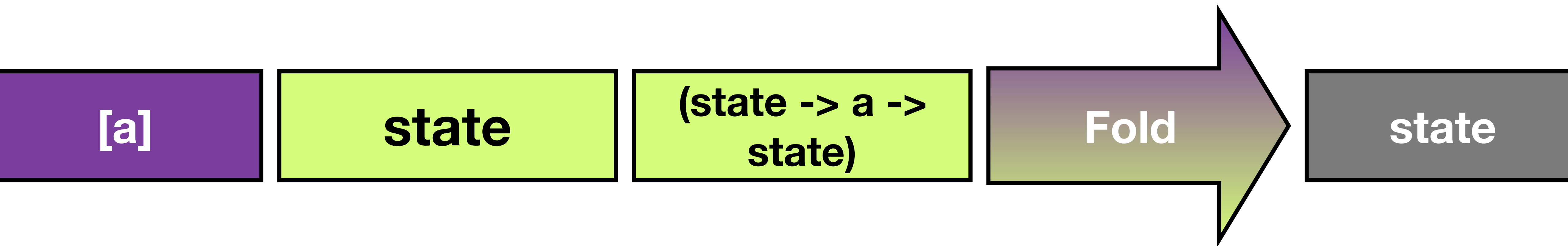
**[a] -> state -> (state -> a -> state) -> state**

# Fold



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

# Fold



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

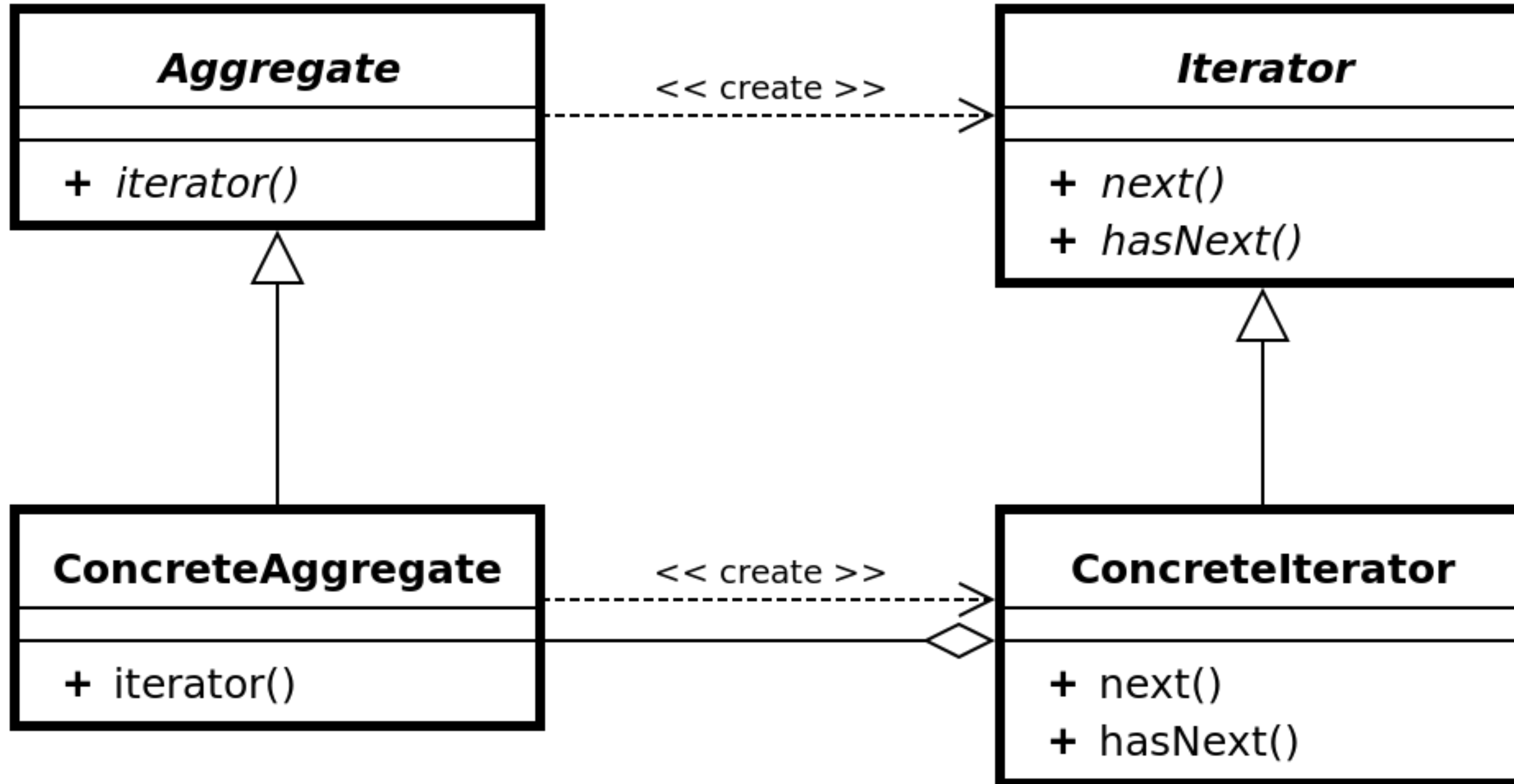
    return result;
}
```

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

# Higher Order Functions in C#

# Iterator Pattern



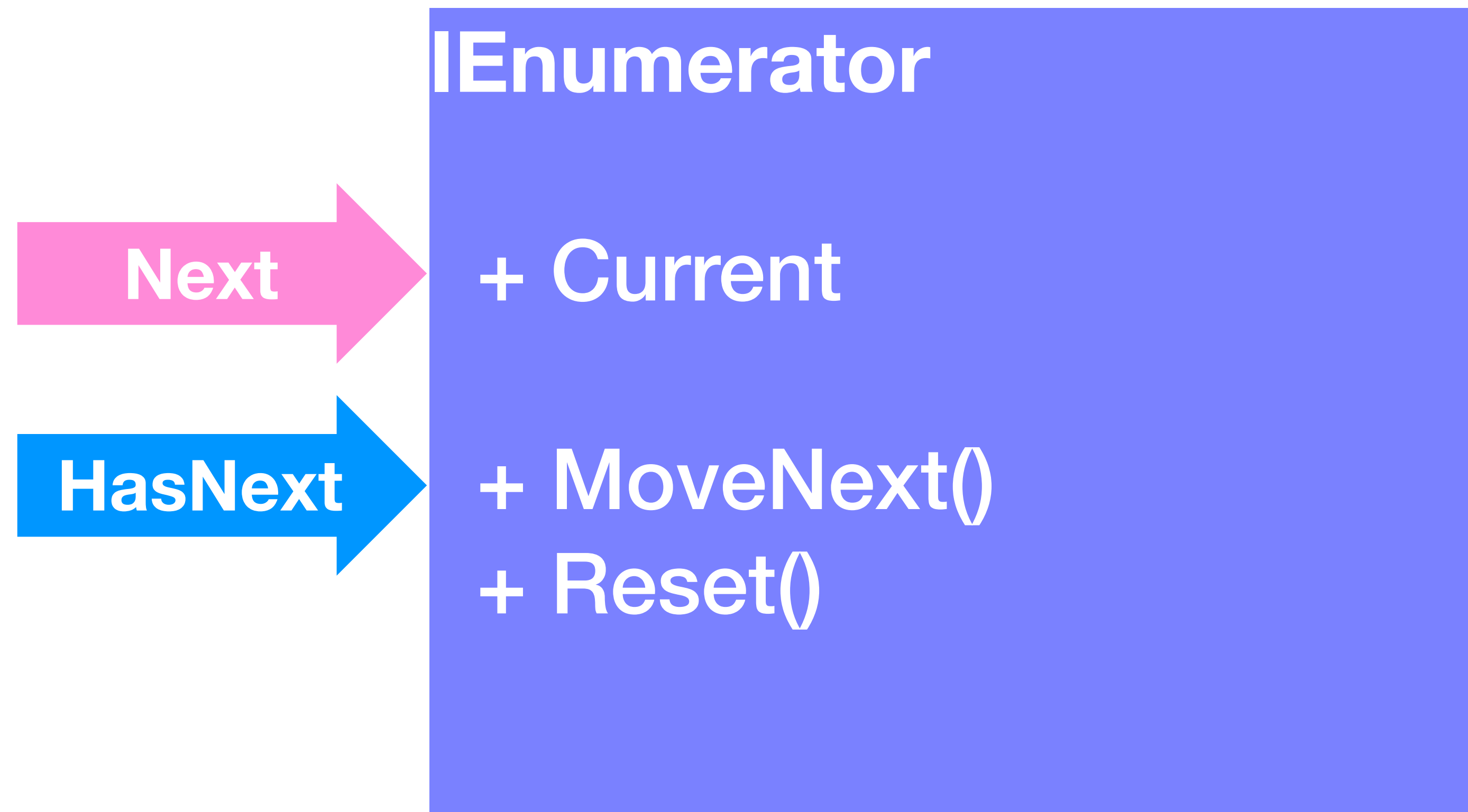


# IEnumerator

+ Current

+ MoveNext()

+ Reset()



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

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

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

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

# Map

$[a] \rightarrow (a \rightarrow b) \rightarrow [b]$



```
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;
    }
}
```

```
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;
    }
}
```

```
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;
    }
}
```

```
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;
    }
}
```

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

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

# Filter

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

```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IListProvider<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;
    }
}
```

```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IListProvider<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;
    }
}
```



```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IListProvider<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;
    }
}
```

```
internal sealed class WhereArrayIterator<TSource> : Iterator<TSource>, IListProvider<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;
    }
}
```

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

    Dispose();
    return false;
}
```

# Fold

**$[a] \rightarrow \text{state} \rightarrow (\text{state} \rightarrow a \rightarrow \text{state}) \rightarrow \text{state}$**

```
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;
}
```

```
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;
}
```

```
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;
}
```

```
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;
}
```



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

# LINQ Execution

# Code

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

# 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)  
    };
```

# 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);
```

# 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; });
```

# Answer

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

# 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);
```



# Theory

# Fusion Property of Iterators

# Fusion Property of Iterators

iterator  $f$   $\circ$  iterator  $g$  = iterator  $(f \circ g)$

# Foreach Loop

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

# Foreach Loop

```
var total = 0.0;  
Iterate (var order in orders)  
{  
    Filter(order.Zip == 53202)  
    Fold(+, Map(order.Price * order.Quantity));  
}
```

# Foreach Loop

iterator *filter* |> iterator *map* |> iterator *fold*

# Foreach Loop

iterator *filter* |> iterator *map* |> iterator *fold* =  
( iterator *fold*  
  ( iterator *map*  
    ( iterator *filter* ) ) )

# Foreach Loop

$\text{iterator } \textit{filter} \mid > \text{iterator } \textit{map} \mid > \text{iterator } \textit{fold} =$   
 $( \text{iterator } \textit{fold} ( \text{iterator } \textit{map} ( \text{iterator } \textit{filter} ) ) ) =$   
 $\text{iterator } \textit{fold} \circ \text{iterator } \textit{map} \circ \text{iterator } \textit{filter}$



# Foreach Loop

$\text{iterator } filter \mid > \text{iterator } map \mid > \text{iterator } fold =$   
 $( \text{iterator } fold ( \text{iterator } map ( \text{iterator } filter ) ) ) =$   
 $\text{iterator } fold \circ \text{iterator } map \circ \text{iterator } filter =$   
 $\text{iterator } (fold \circ map \circ filter)$

# Foreach Loop

iterator *fold* ◦ iterator *map* ◦ iterator *filter* =  
iterator (*fold* ◦ *map* ◦ *filter*)

# LINQ

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

# LINQ

```
var total = orders
    .Filter(order => order.Zip == 53202)
    .Map(order => order.Price * order.Quantity)
    .Fold(0.0, (sub, amount) => sub + amount);
```

# LINQ

iterator *filter* ◦ iterator *map* ◦ iterator *fold* =  
iterator (*filter* ◦ *map* ◦ *fold*)

# Universal Principal

# C# List Comprehension

```
(from order in orders
 where order.Zip == 53202
 select new {Amount = order.Price * order.Quantity})
.Sum(order => order.Amount);
```

# JavaScript



# PowerShell

```
($orders |  
  Where-Object { $_.Zip -eq 53202 } |  
  Select-Object @{  
    Name = "Amount";  
    Expression = {$_.Price * $_.Quantity} } |  
  Measure-Object Amount -Sum).Sum
```



# F#

orders

```
|> List.filter (fun x -> x.Zip = 53202)
|> List.map (fun x -> x.Price * (double x.Quantity))
|> List.sum
```

# T-SQL

```
select distinct
  sum(price * quantity) over (partition by zip)
from (
  values
    (53202, 1.89, 3),
    (60191, 1.99, 2),
    (60060, 0.99, 7),
    (53202, 1.29, 8),
    (60191, 1.89, 2),
    (53202, 0.99, 3)
) as orders(zip, price, quantity)
where zip = 53202
```

**Thank you**

# Next Steps

# Example Code

- C#, <https://github.com/MikeMKH/talks/tree/master/say-goodbye-to-the-for-loop-with-higher-order-functions/csharp>
- PowerShell, <https://github.com/MikeMKH/talks/tree/master/say-goodbye-to-the-for-loop-with-higher-order-functions/powershell>
- F#, <https://github.com/MikeMKH/talks/tree/master/say-goodbye-to-the-for-loop-with-higher-order-functions/fsharp>
- T-SQL, <https://github.com/MikeMKH/talks/tree/master/say-goodbye-to-the-for-loop-with-higher-order-functions/sql>

# Images

- UML Iterator Pattern, By Trashtoy - My own work written with text editor., Public Domain, <https://commons.wikimedia.org/w/index.php?curid=1698830>
- PowerShell Logo, [https://upload.wikimedia.org/wikipedia/commons/2/2f/PowerShell\\_5.0\\_icon.png](https://upload.wikimedia.org/wikipedia/commons/2/2f/PowerShell_5.0_icon.png)

# 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>