Say Goodbye to the For Loop with High Order Functions

Mike Harris

Pop Quiz

What do these do?

```c
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

```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#

Find the total for 53202?

For Loop

```csharp
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

```csharp
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

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

C#

Foreach Loop

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

C#

Foreach Loop

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

C#

Foreach Loop

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

C#

Foreach Loop

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

C#

LINQ

LINQ

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

C#

LINQ

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

C#

LINQ

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

C#

LINQ

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

C#

LINQ

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

C#

Compare

Compare

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

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

C#

General

General

orders

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

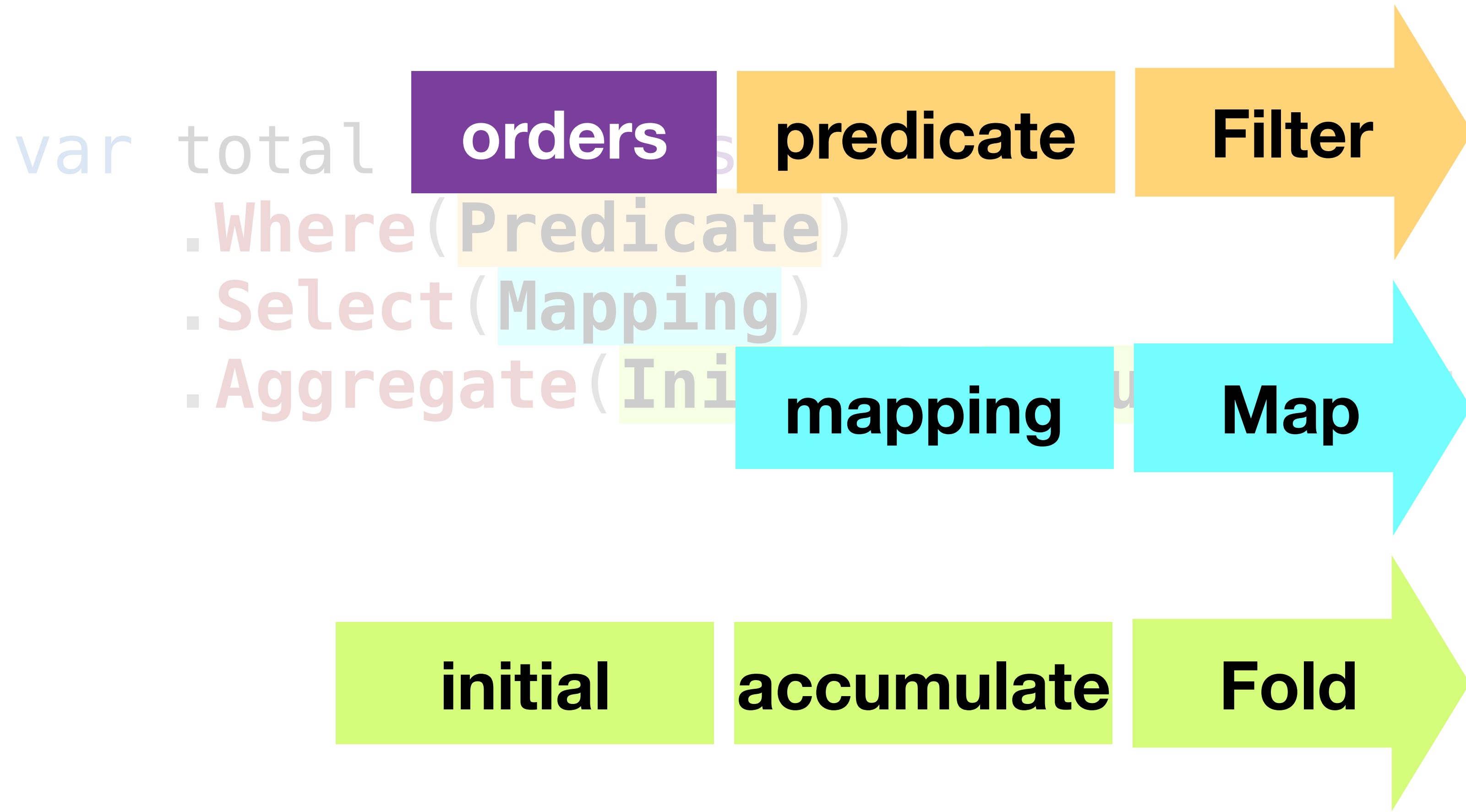| predicate | Filter |

| mapping | Map |

| initial | accumulate | Fold | total |

General

General

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

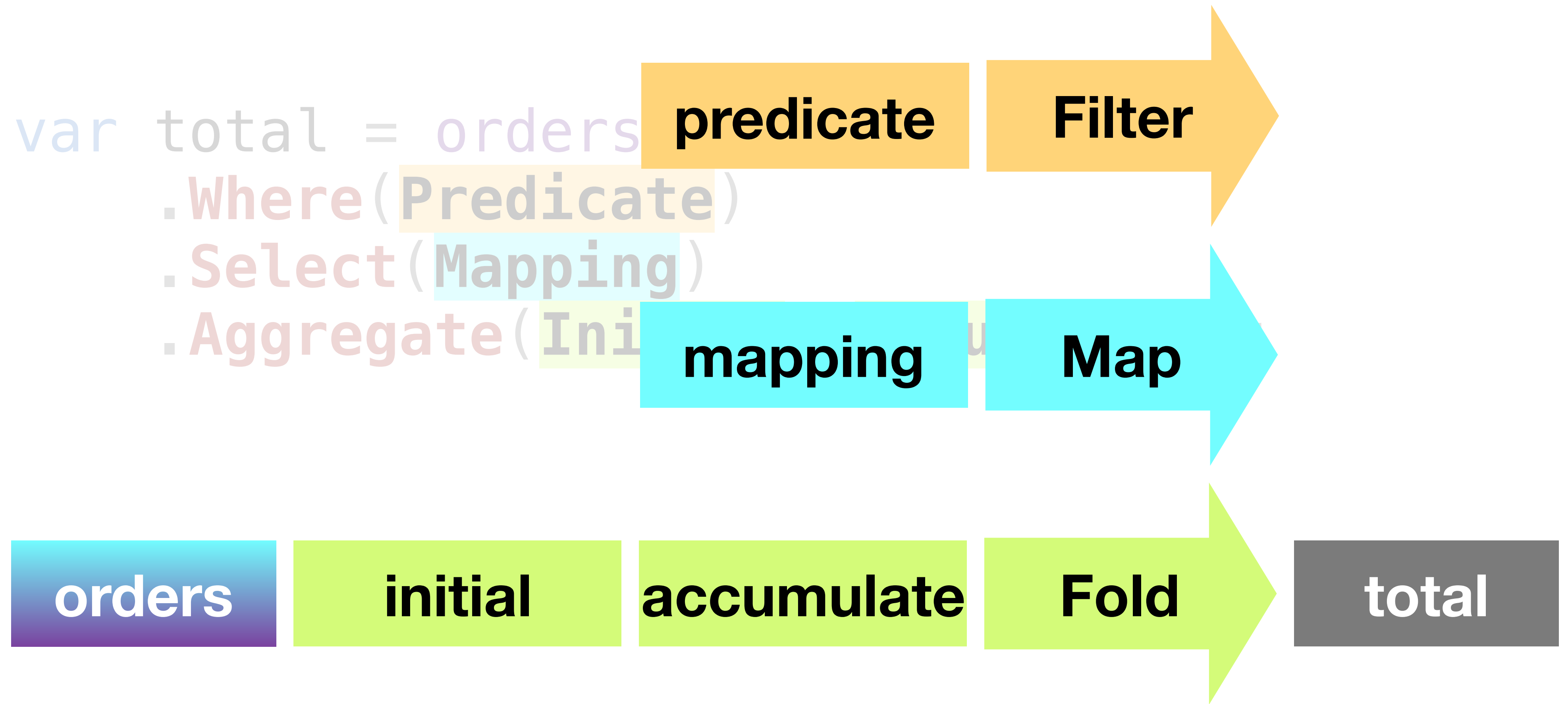| predicate	Filter

orders	mapping	Map

initial	accumulate	Fold

General

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

predicate | Filter

mapping | Map

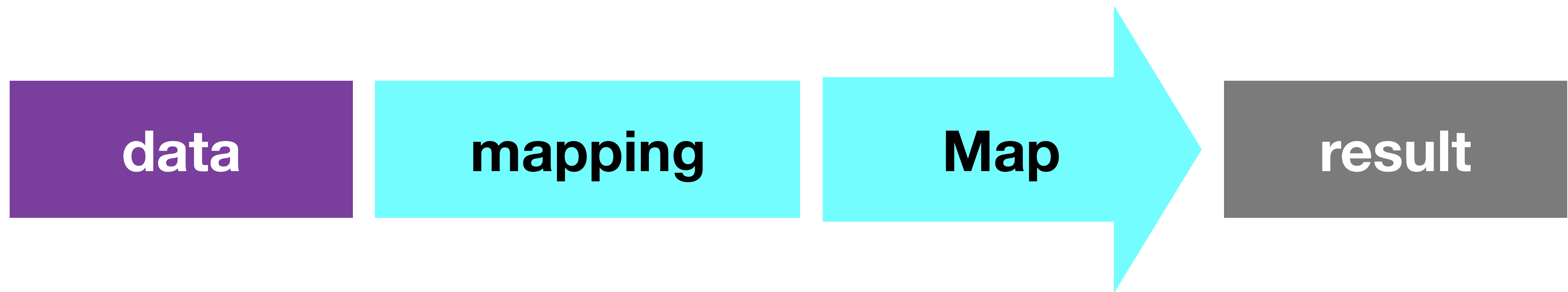orders | initial | accumulate | Fold | total

Higher Order Functions in General

Map

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

Map

data | mapping | Map → result

Map

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

Map

mapping | data | Map → result

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

Map

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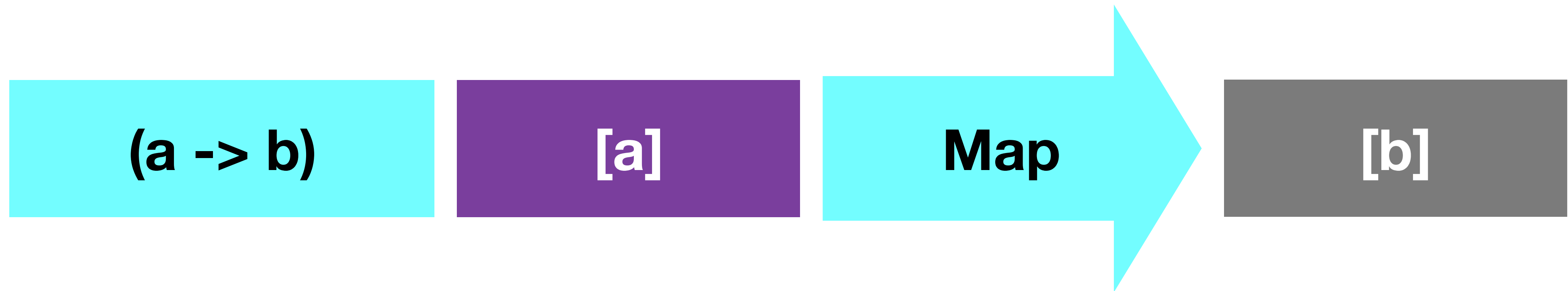
```csharp
IEnumerable<U> Map<T, U>(
    Func<T, U> mapping, IEnumerable<T> source)
{
    var result = new List<U>();
    foreach(var item in source)
    {
        result.Add(mapping(item));
    }

    return result;
}
```

C#

```csharp
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

(a -> bool) **[a]** **Filter** **[b]**

```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#

```csharp
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
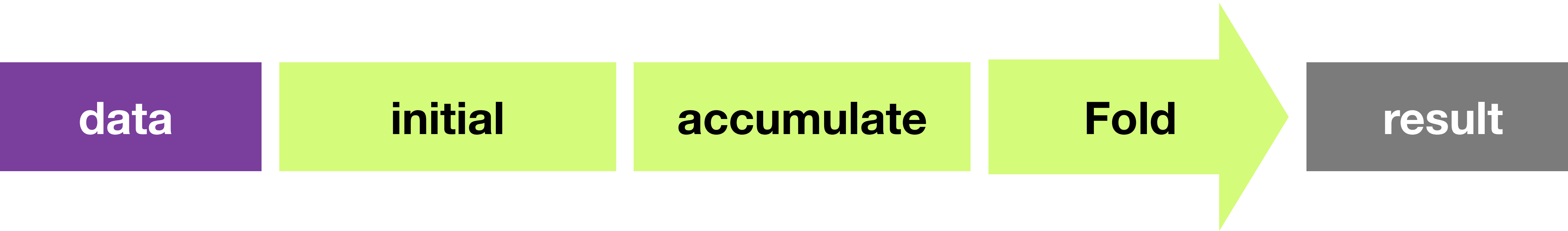
Fold

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

Fold

(state -> a -> state) state [a] Fold → state
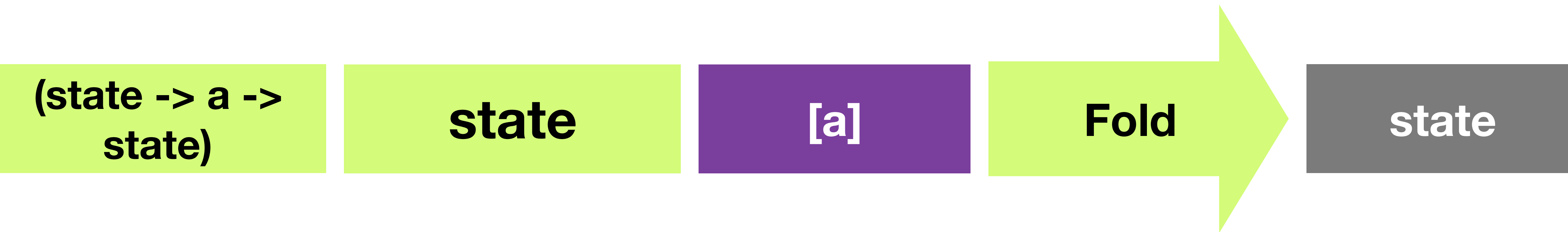
```csharp
U Fold<T, U>(
    Func<U, T, U> accumulate, U initial,
    IEnumerable<T> source)
{

    var result = initial;
    foreach(var item in source)
    {

        result = accumulate(result, item);
    }


    return result;
}
```

C#

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

C#

Higher Order Functions in C#

C#

Iterator Pattern
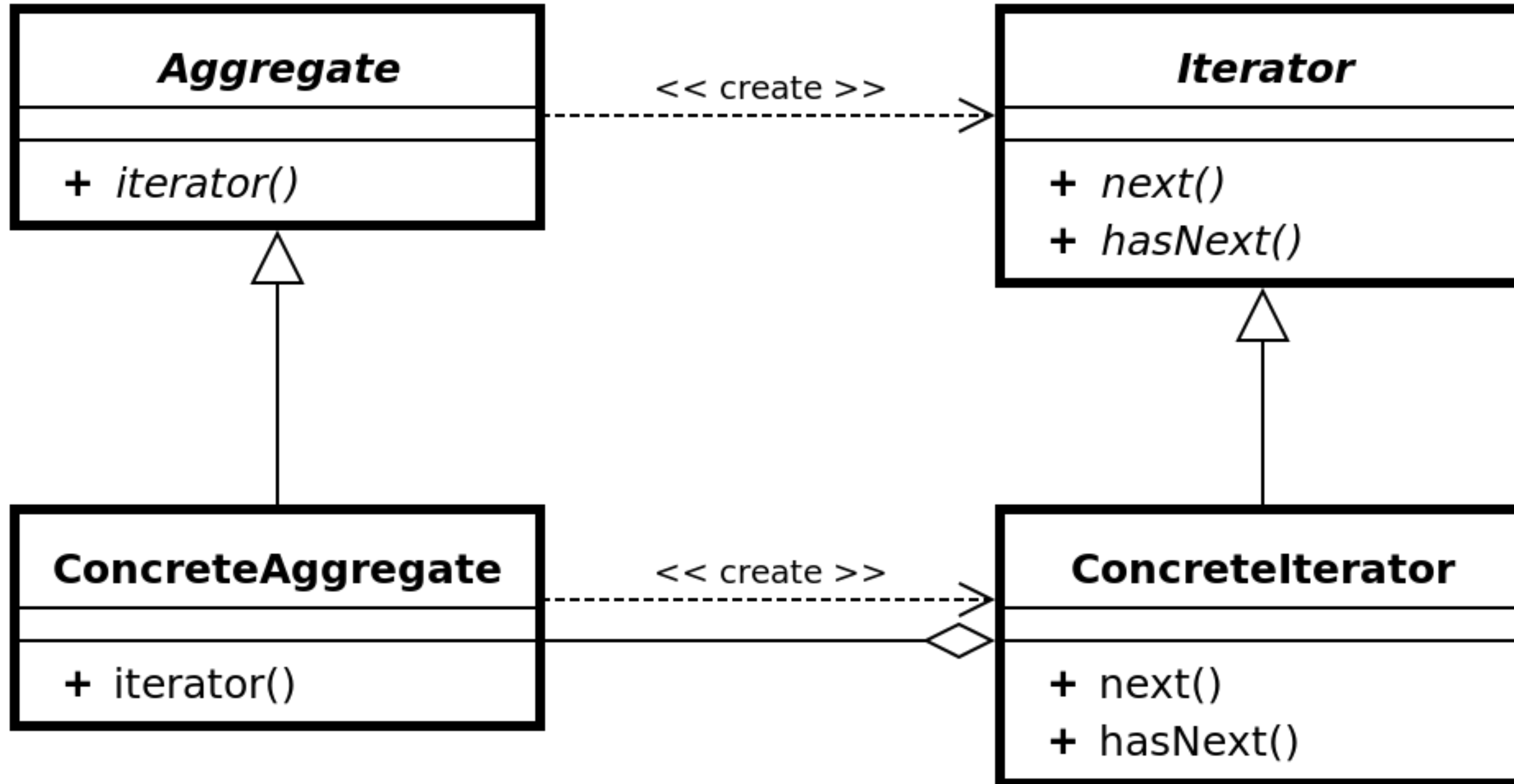
IEnumerator

+ Current

+ MoveNext()
+ Reset()

IEnumerator

Next → + Current

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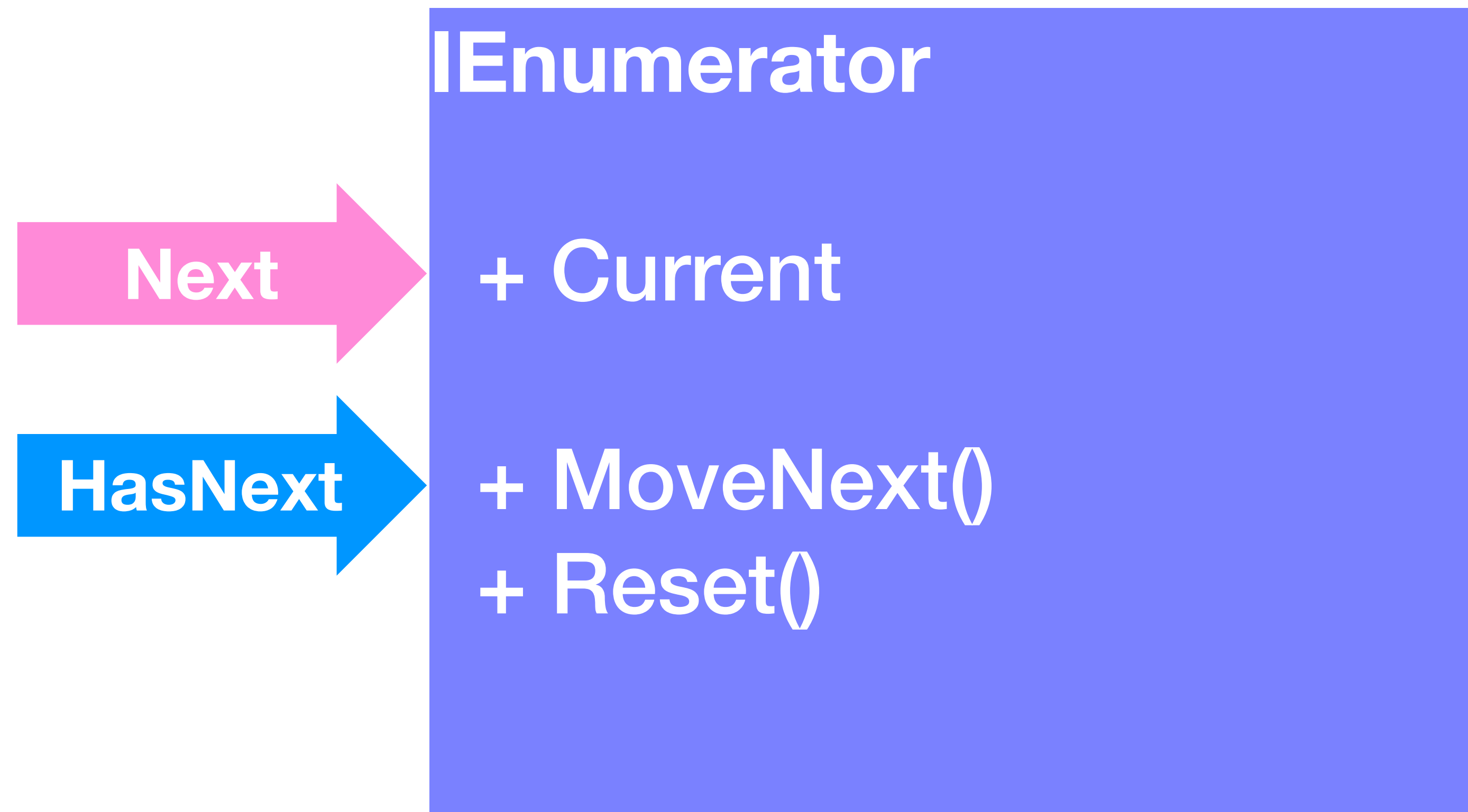
```csharp
void Iterate<T>(Action<T> f, IEnumerator<T> source)
{
    while(source.MoveNext())
    {
        f(source.Current);
    }
}
```

```csharp
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);
    }
}
```

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

Map

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

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 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#

```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;
        Next = _selector(_source[index]);
        return true;
    }
}
```

C#

```csharp
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#

```csharp
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#

```csharp
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#

```csharp
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#

```csharp
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#

```csharp
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#

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


    return result;
}
```

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


    return result;
}
```

C#

```csharp
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#

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


return result;
```

C#

LINQ Execution

Code

```csharp
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?

```csharp
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?

```csharp
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

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

C#

Does exactly what we want

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

C#

Theory

Fusion Property of Iterators

Fusion Property of Iterators

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

Foreach Loop

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

C#

Foreach Loop

```csharp
var total = 0.0;
Iterate (var order in orders)
{

    Filter(order.Zip == 53202)
        Fold(+, Map(order.Price * order.Quantity));
}
```

C#

Foreach Loop

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

LINQ

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

C#

LINQ

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

C#

LINQ

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

Universal Principal

C# List Comprehension

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

C#

JavaScript

PowerShell

F#

T-SQL

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