Programação Funcional

Estrutura de Linguagens

https://github.com/fsantanna-uerj/EDL/

Francisco Sant'Anna

francisco@ime.uerj.br

Programação Funcional

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

```
a = b * 2
x = a + b
b = 10
```

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

```
a = b * 2
x = a + b
b = 10
b = 0
```

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

```
x = input()
```

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

```
x = input()
```

```
int g = 0;
int f (int x) {
  g++;
  return x + g;
}
f(0);
f(0);
```

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

In contrast, imperative programming changes state with commands in the source language, the most simple example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

In contrast, imperative programming changes state with commands in the source language, the most simple example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

In contrast, imperative programming changes state with commands in the source language, the most simple example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

In contrast, imperative programming changes state with commands in the source language, the most simple example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Programação Funcional

In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions[1] or declarations[2] instead of statements. In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function *f* twice with the same value for an argument *x* will produce the same result *f(x)* each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

In contrast, imperative programming changes state with commands in the source language, the most simple example being assignment. Imperative programming does have functions—not in the mathematical sense—but in the sense of subroutines. They can have side effects that may change the value of program state. Functions without return values therefore make sense. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program.[3]

Imperativo vs Funcional

Imperativo vs Funcional

- Comando vs Expressão

Imperativo vs Funcional

- Comando vs Expressão
- Sequência vs Dependência

Imperativo vs Funcional

- Comando vs Expressão
- Sequência vs Dependência
- Atribuição vs Definição

Programação Funcional

Programação Funcional

- Funções de Primeira Classe

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções

```
// funcional
t = { 10,1,5 }
table.sort(t,
  function (v1,v2)
    return v1 > v2
  end
) -- {10,5,1}
```

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções
- Funções Puras

```
// funcional
t = { 10,1,5 }
table.sort(t,
  function (v1,v2)
    return v1 > v2
  end
) -- {10,5,1}
```

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções
- Funções Puras
  - sem efeito colateral

```
// funcional
t = { 10,1,5 }
table.sort(t,
  function (v1,v2)
    return v1 > v2
  end
) -- {10,5,1}
```

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções
- Funções Puras
  - sem efeito colateral
  - qualquer ordem de avaliação (inclusive em paralelo)

```
// funcional
t = { 10,1,5 }
table.sort(t,
  function (v1,v2)
    return v1 > v2
  end
) -- {10,5,1}
```

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções
- Funções Puras
  - sem efeito colateral
  - qualquer ordem de avaliação (inclusive em paralelo)
- Funções Recursivas

```
// funcional
t = { 10,1,5 }
table.sort(t,
  function (v1,v2)
    return v1 > v2
  end
) -- {10,5,1}
```

Programação Funcional

- Funções de Primeira Classe
  - podem ser criadas, passadas, retornadas
- Funções de Alta Ordem
  - recebem ou retornam funções
- Funções Puras
  - sem efeito colateral
  - qualquer ordem de avaliação (inclusive em paralelo)
- Funções Recursivas
  - auto referência

```
// funcional
t = { 10,1,5 }
table.sort(t,
  function (v1,v2)
    return v1 > v2
  end
) -- {10,5,1}
```

Exercícios

Exercícios

1. Sobre o conceito de “transparência referencial”...
   - Explique esse conceito em uma frase com as suas próprias palavras.
   - Quais são as principais vantagens de transparência referencial? Por quê essas vantagens são possíveis?

Exercícios

1. Sobre o conceito de “transparência referencial”...
   - Explique esse conceito em uma frase com as suas próprias palavras.
   - Quais são as principais vantagens de transparência referencial? Por quê essas vantagens são possíveis?
2. Sobre funções recursivas…
   - Dê exemplos de funções recursivas que você já tenha escrito em trabalhos e projetos. (Faça uma busca rigorosa.) Por quê você optou por essa técnica?

Exercícios

1. Sobre o conceito de “transparência referencial”...
   - Explique esse conceito em uma frase com as suas próprias palavras.
   - Quais são as principais vantagens de transparência referencial? Por quê essas vantagens são possíveis?
2. Sobre funções recursivas…
   - Dê exemplos de funções recursivas que você já tenha escrito em trabalhos e projetos. (Faça uma busca rigorosa.) Por quê você optou por essa técnica?
3. Sobre funções de alta ordem...
   - Dê exemplos de funções de alta ordem que você já tenha escrito ou usado em trabalhos e projetos. (Faça uma busca rigorosa.) Por quê você optou por essa técnica?

Exercícios

1. Sobre o conceito de “transparência referencial”...
   - Explique esse conceito em uma frase com as suas próprias palavras.
   - Quais são as principais vantagens de transparência referencial? Por quê essas vantagens são possíveis?
2. Sobre funções recursivas…
   - Dê exemplos de funções recursivas que você já tenha escrito em trabalhos e projetos. (Faça uma busca rigorosa.) Por quê você optou por essa técnica?
3. Sobre funções de alta ordem...
   - Dê exemplos de funções de alta ordem que você já tenha escrito ou usado em trabalhos e projetos. (Faça uma busca rigorosa.) Por quê você optou por essa técnica?
4. Sobre declarações múltiplas…
   - O exemplo a seguir não adere ao modelo funcional pois possui duas declaração para a variável *b*. Mas o que deve acontecer quando o programador comete esse erro?

```
a = b * 2
x = a + b
b = 10
b = 0
```

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

O Básico

O Básico

- Listas encadeadas
  - principal estrutura de dados

O Básico

- Listas encadeadas
  - principal estrutura de dados

```
struct node {
    int val;
    struct node* nxt;
};
```

O Básico

- Listas encadeadas
  - principal estrutura de dados

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

O Básico

- Listas encadeadas
  - principal estrutura de dados

O Básico

- Listas encadeadas
  - principal estrutura de dados

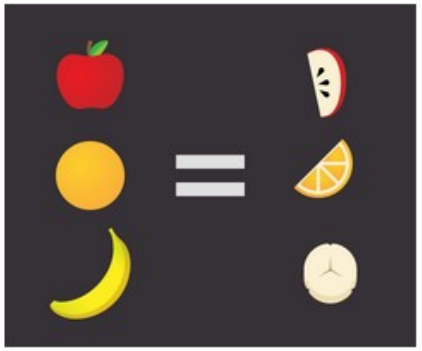
- Funções de Alta Ordem
  - `map`
  - `filter`
  - `fold (reduce)`

O Básico

- Listas encadeadas
  - principal estrutura de dados
- Funções de Alta Ordem
  - `map`
  - `filter`
  - `fold (reduce)`

Map

You have an array of items and want to transform each of them. The result is a new array of the exact same length containing the manipulated items.



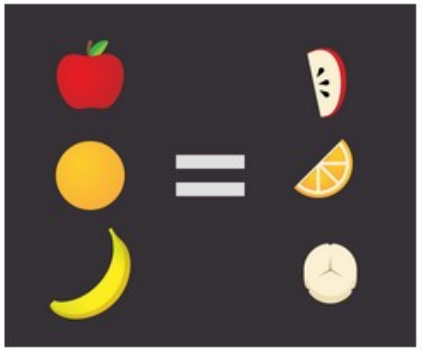
http://atendesigngroup.com/blog/

O Básico

- Listas encadeadas
  - principal estrutura de dados
- Funções de Alta Ordem
  - `map`
  - `filter`
  - `fold (reduce)`

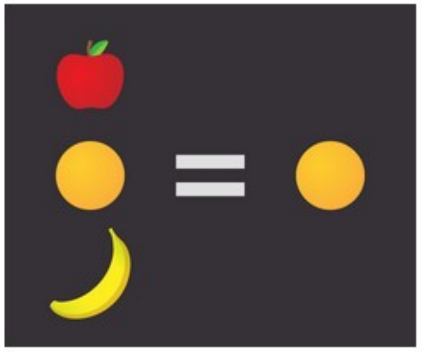
Map

You have an array of items and want to transform each of them. The result is a new array of the exact same length containing the manipulated items.



Filter

You have an array and want to filter out certain items. The result is a new array with the same items, but with some excluded. The length of the new array will be the same (if no values were omitted) or shorter than the original.



http://atendesigngroup.com/blog/

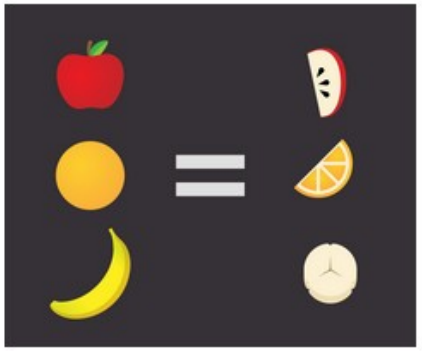
O Básico

- Listas encadeadas
  - principal estrutura de dados

- Funções de Alta Ordem
  - `map`
  - `filter`
  - `fold (reduce)`

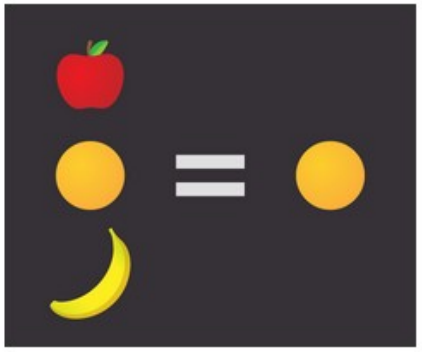
Map

You have an array of items and want to transform each of them. The result is a new array of the exact same length containing the manipulated items.



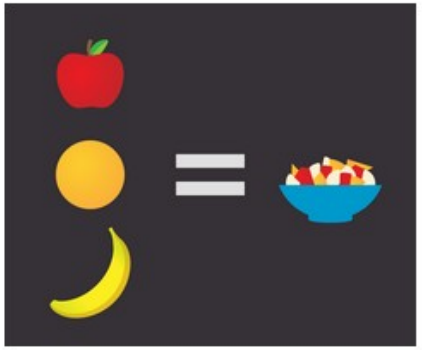
Filter

You have an array and want to filter out certain items. The result is a new array with the same items, but with some excluded. The length of the new array will be the same (if no values were omitted) or shorter than the original.



Reduce

You have an array of items and you want to compute some new value by iterating over each item. The result can be anything, another array, a new object, a boolean value etc.



http://atendesigngroup.com/blog/

Exercícios

1. Escreva um programa em C que represente a lista encadeada `[1,2,3,4]`.
   - Use `struct`, ponteiros e `malloc`.
2. Escreva a função `filter` em Python usando recursão.
   - A `filter` deve retornar uma nova lista.
   - Use um nome diferente, ex. `ffilter`, pois `filter` já existe.
3. Escreva a função `map` em Python usando um comando de loop.
   - A `map` deve alterar a lista existente.
   - Use um nome diferente, por ex. `fmap`, pois `map` já está existe.

```
def ffilter (f, l):
    ...
```

```
def fmap (f, l):
    ...
```

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

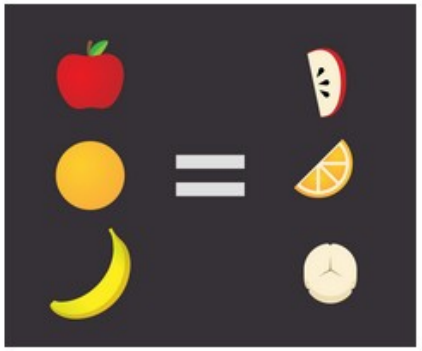
O Básico

- Listas encadeadas
  - principal estrutura de dados

- Funções de Alta Ordem
  - `map`
  - `filter`
  - `fold (reduce)`

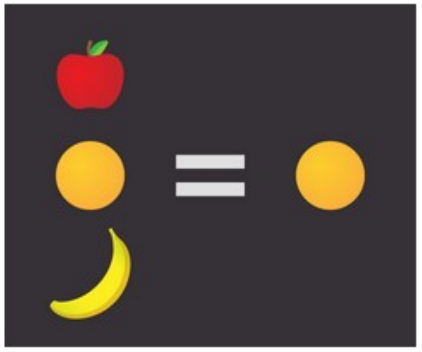
Map

You have an array of items and want to transform each of them. The result is a new array of the exact same length containing the manipulated items.



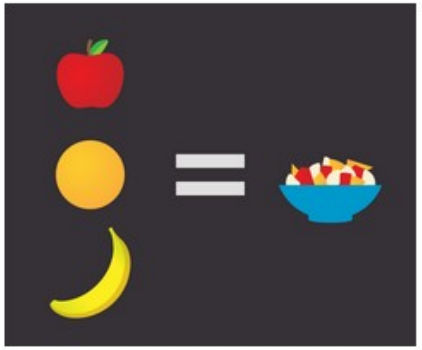
Filter

You have an array and want to filter out certain items. The result is a new array with the same items, but with some excluded. The length of the new array will be the same (if no values were omitted) or shorter than the original.



Reduce

You have an array of items and you want to compute some new value by iterating over each item. The result can be anything, another array, a new object, a boolean value etc.



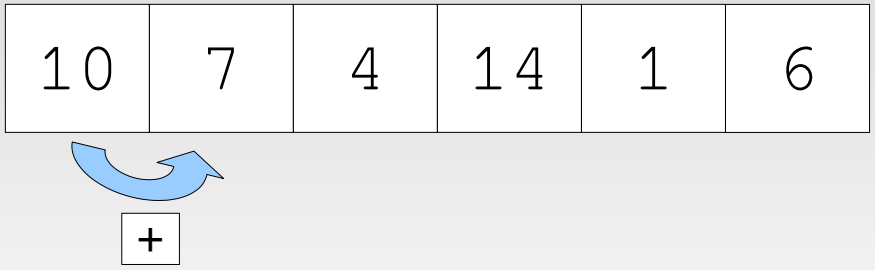
http://atendesigngroup.com/blog/

Fold - “Dobrar”

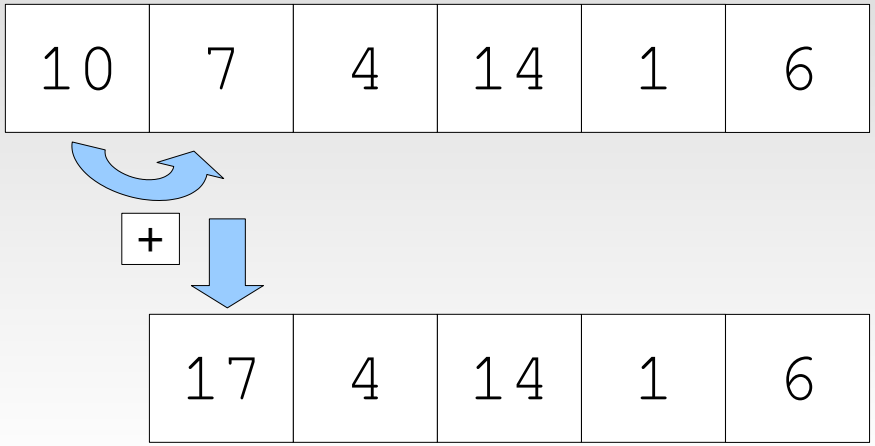
10	7	4	14	1	6

Fold - “Dobrar”

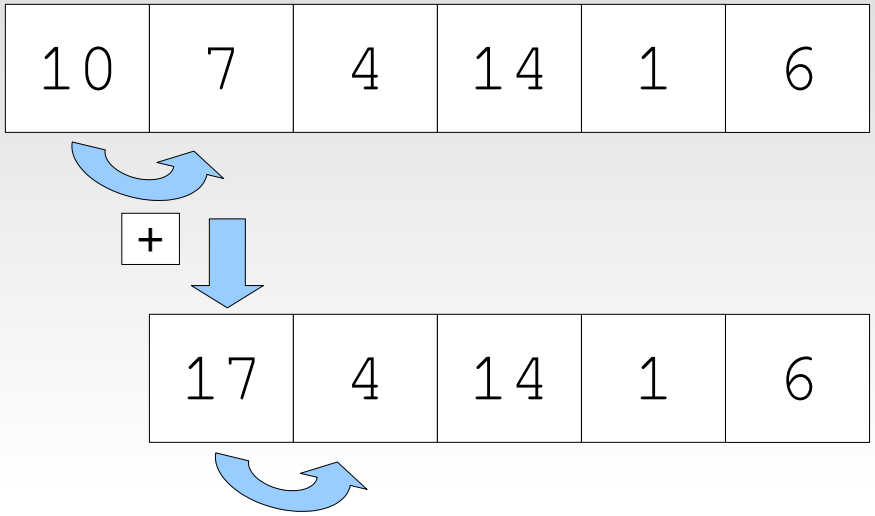
Fold - “Dobrar”



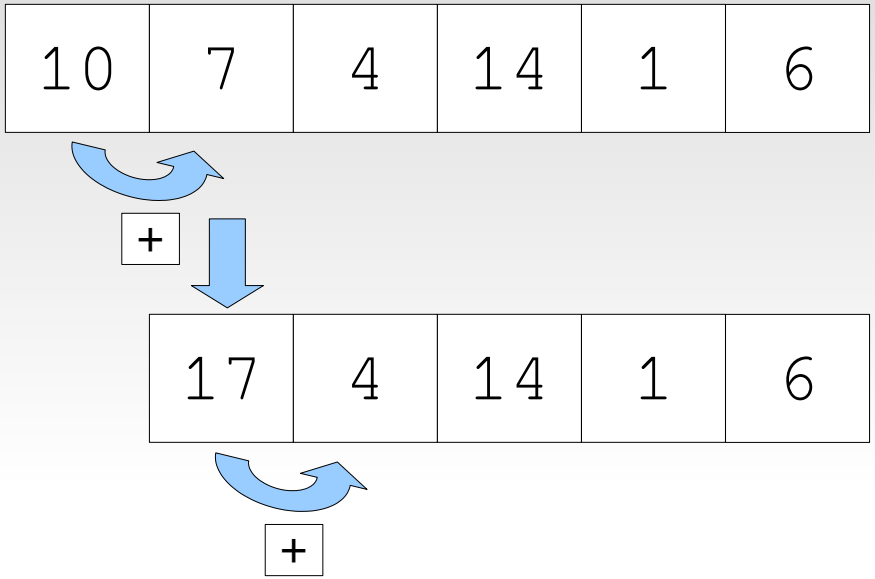
Fold - “Dobrar”



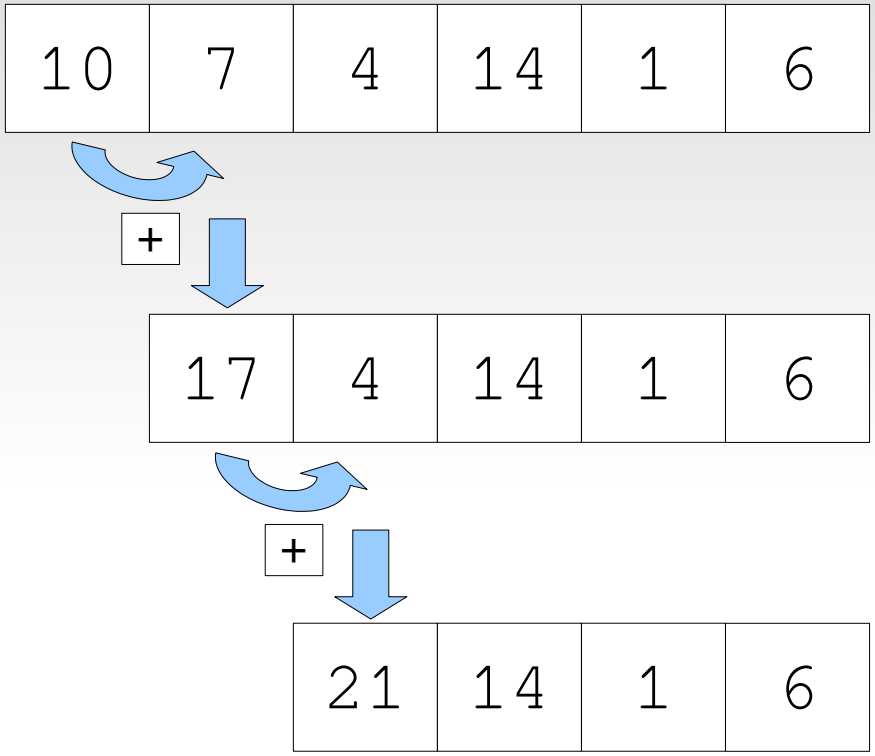
Fold - “Dobrar”



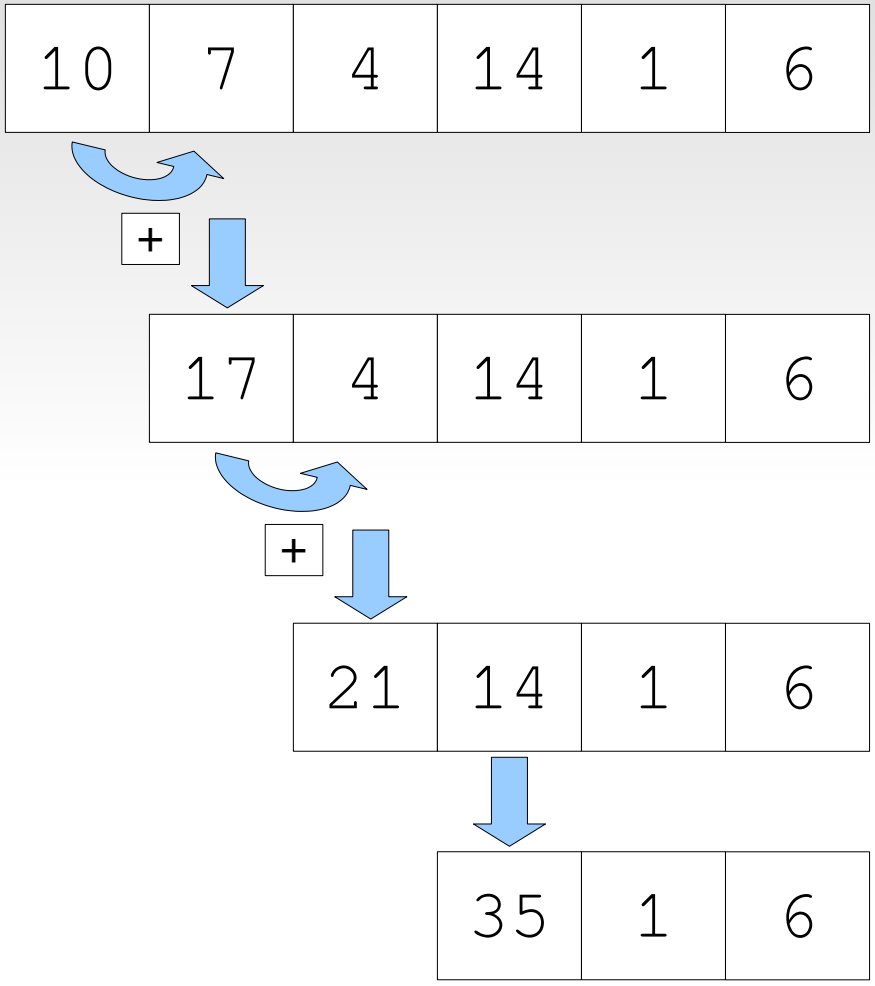
Fold - “Dobrar”



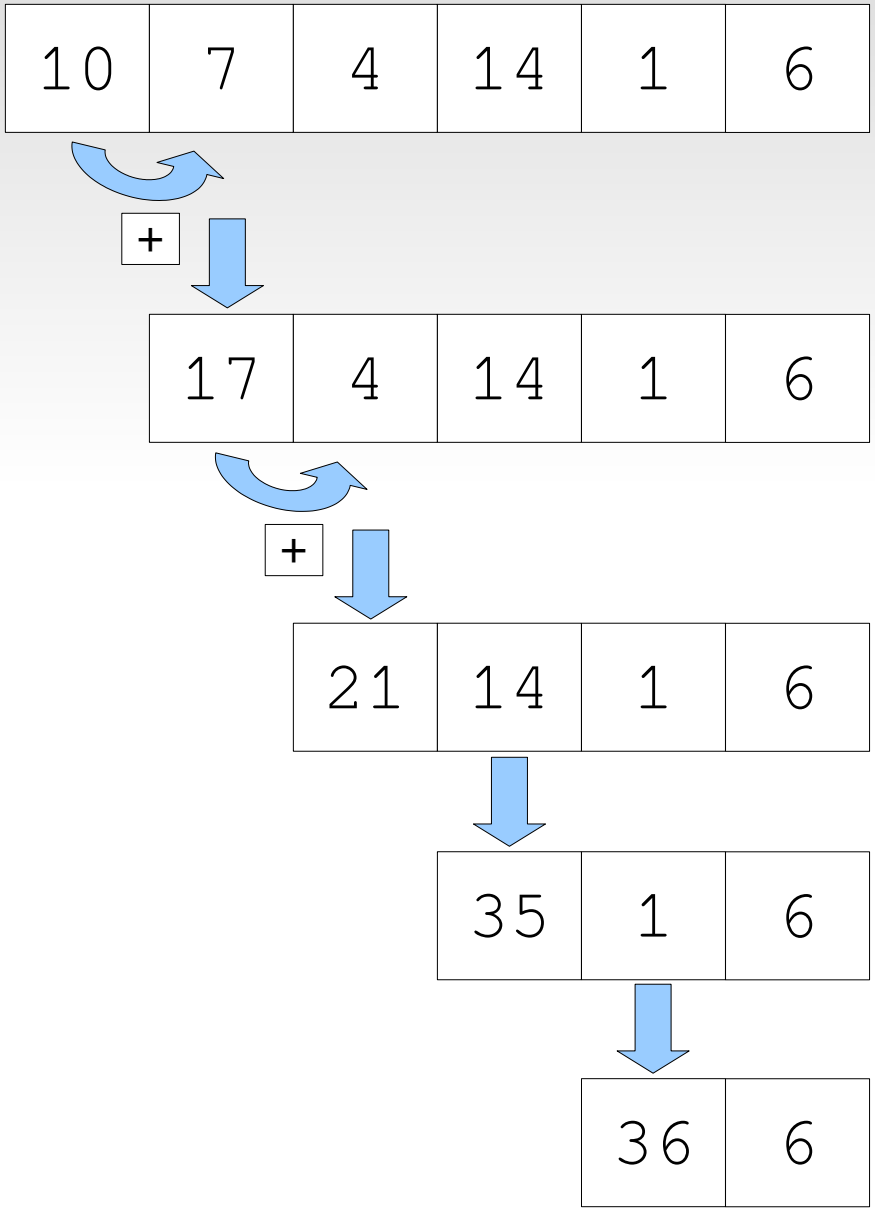
Fold - “Dobrar”



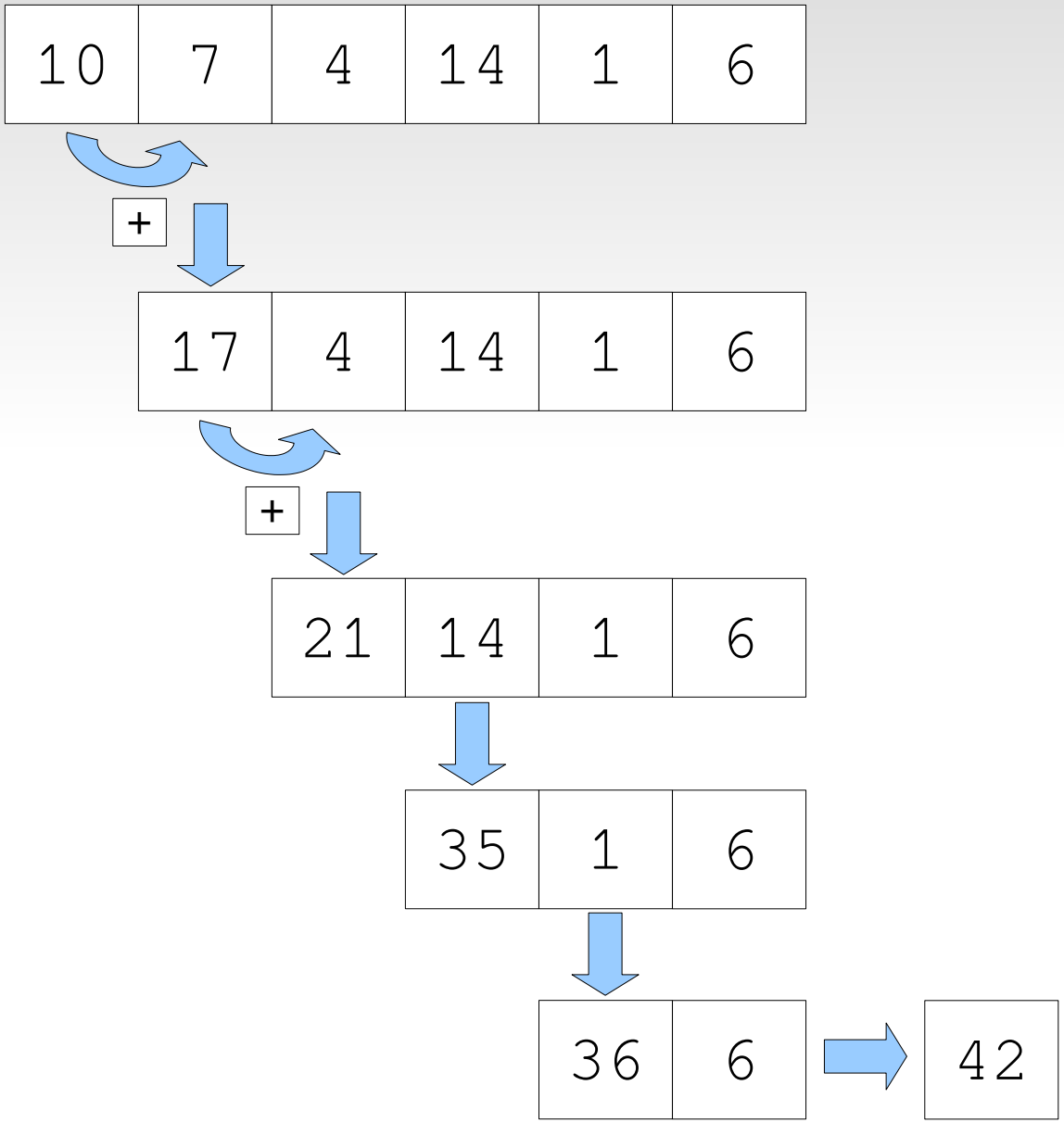
Fold - “Dobrar”



Fold - “Dobrar”



Fold - “Dobrar”



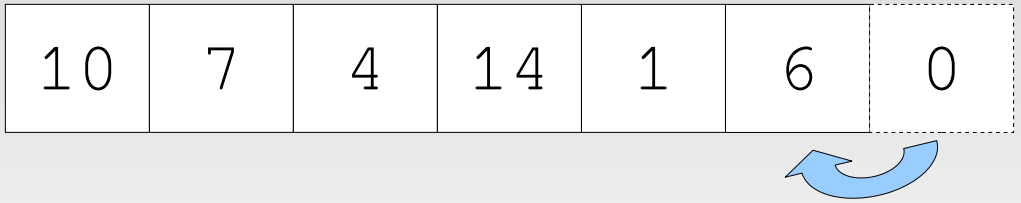
`foldr` - Dobra da dir para esq

10	7	4	14	1	6

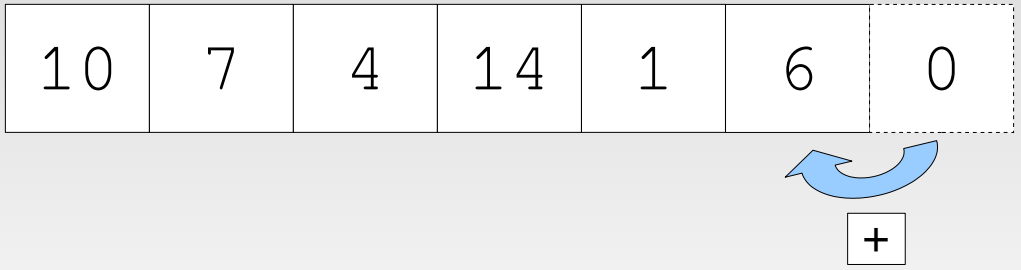
`foldr` - Dobra da dir para esq

10	7	4	14	1	6	0

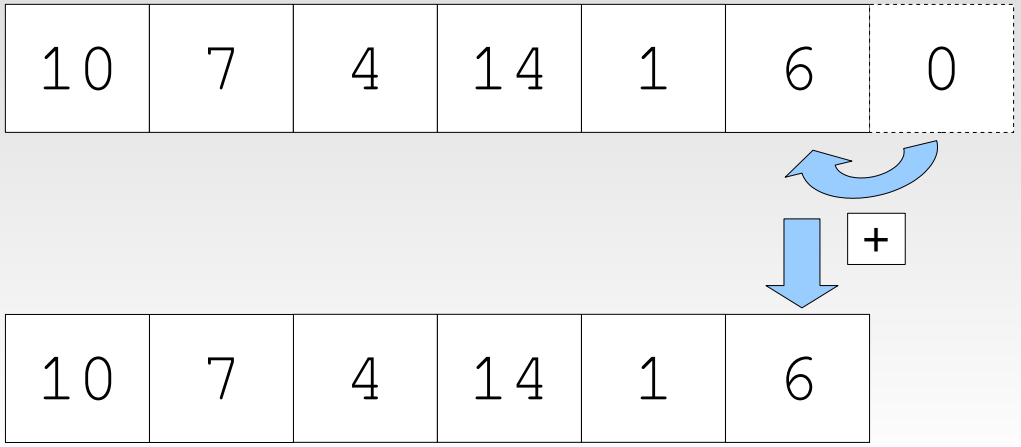
`foldr` - Dobra da dir para esq



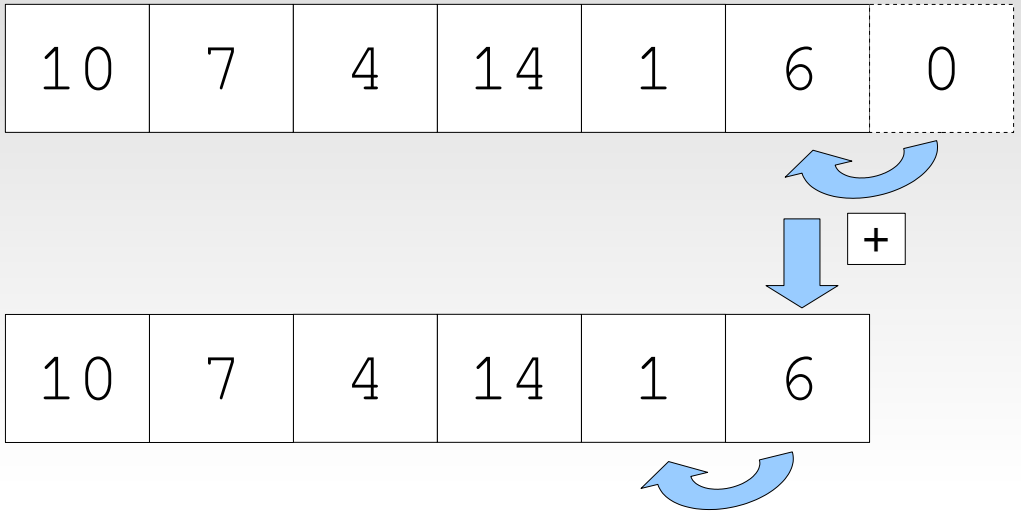
`foldr` - Dobra da dir para esq



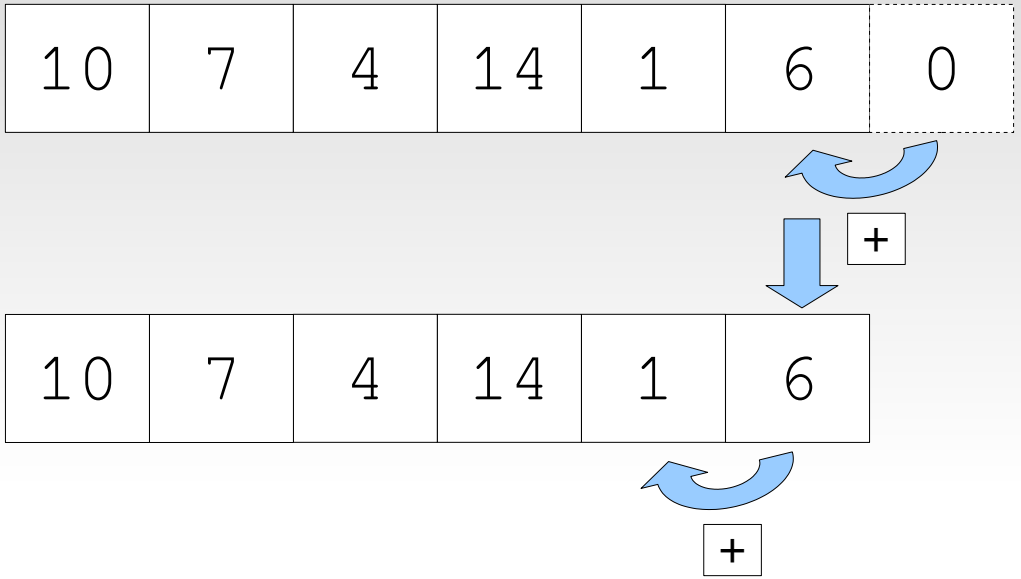
`foldr` - Dobra da dir para esq



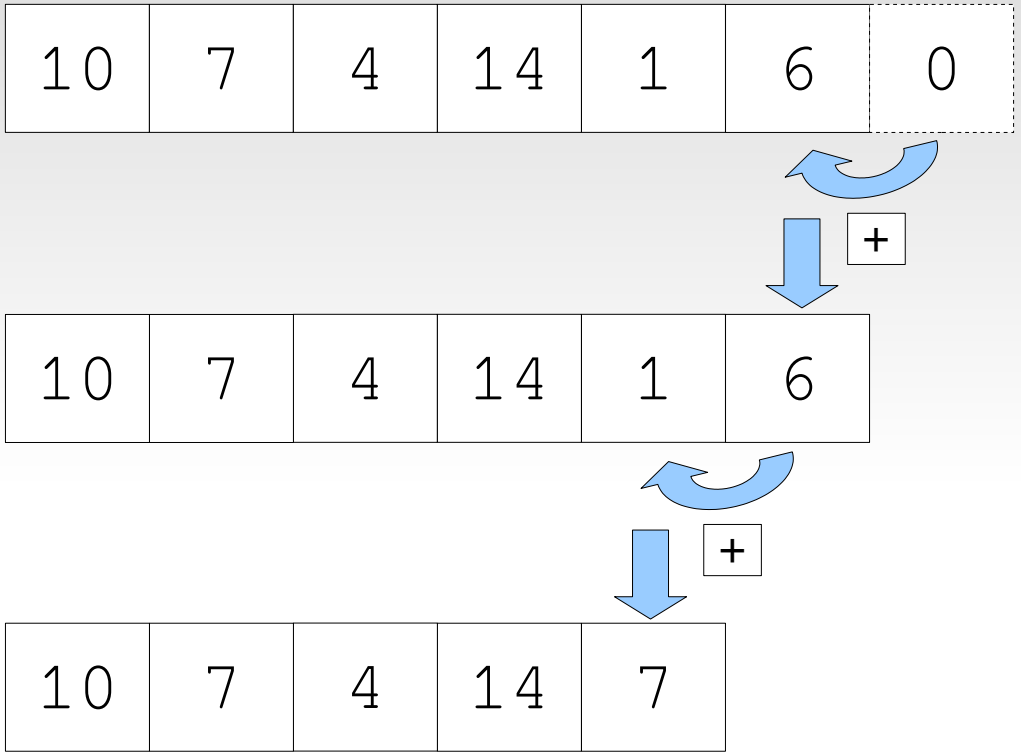
`foldr` - Dobra da dir para esq



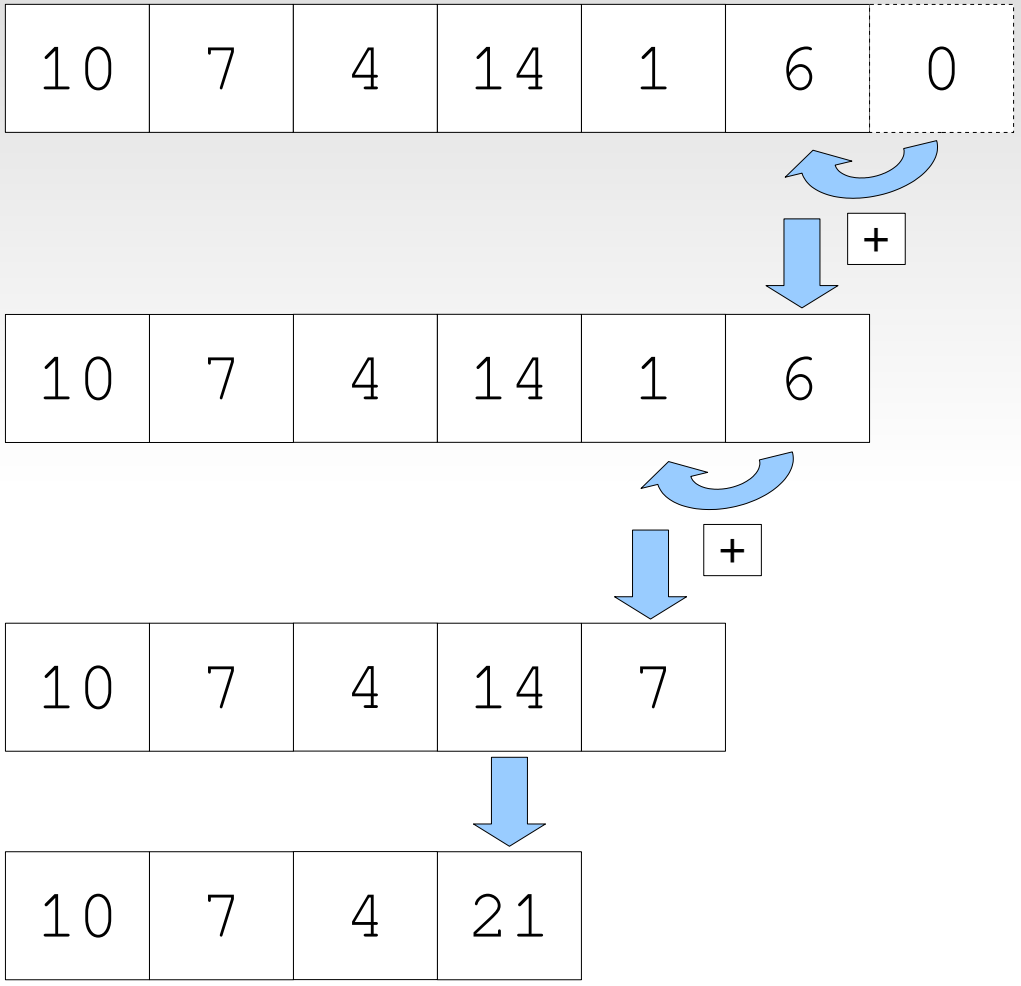
`foldr` - Dobra da dir para esq



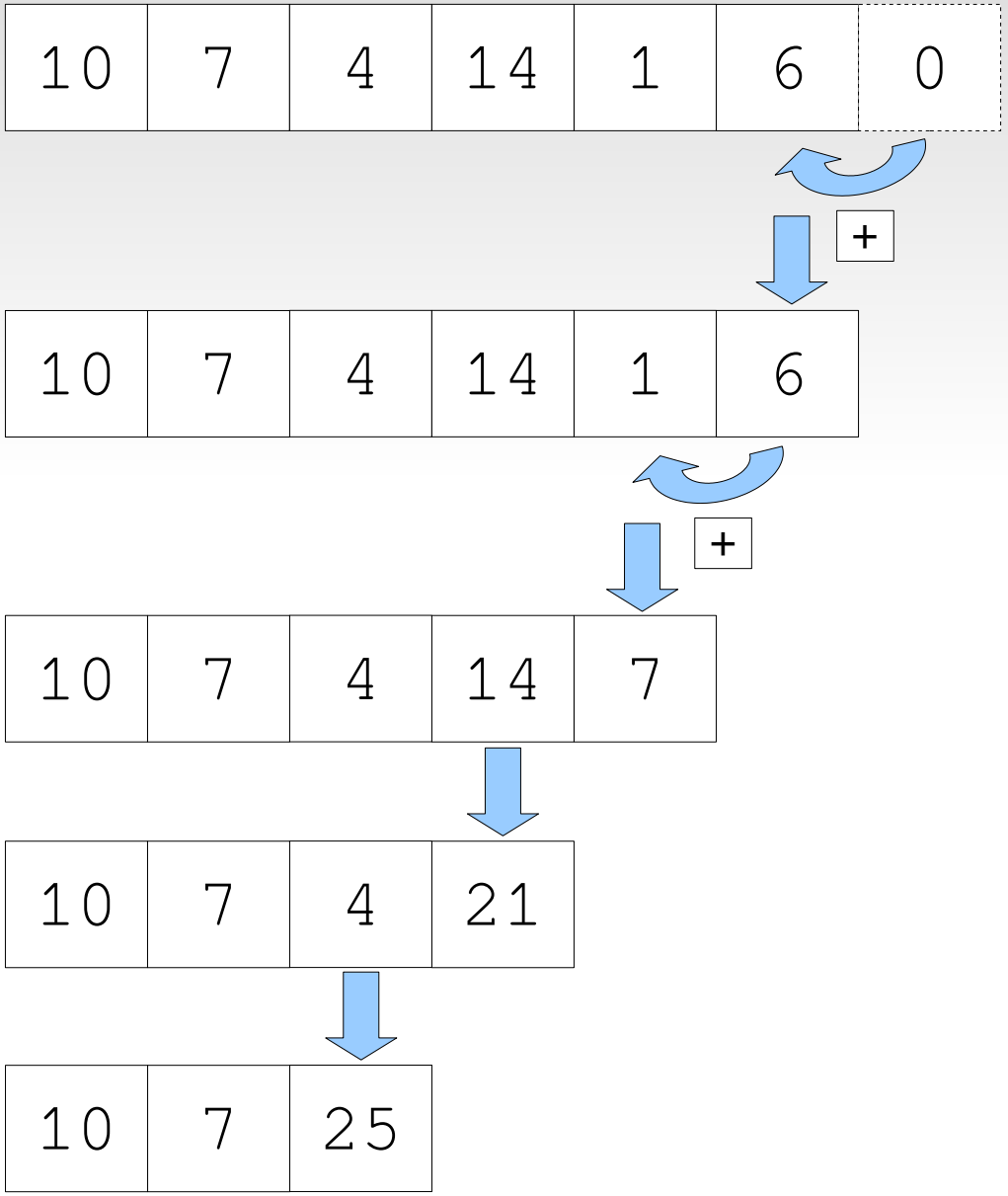
`foldr` - Dobra da dir para esq



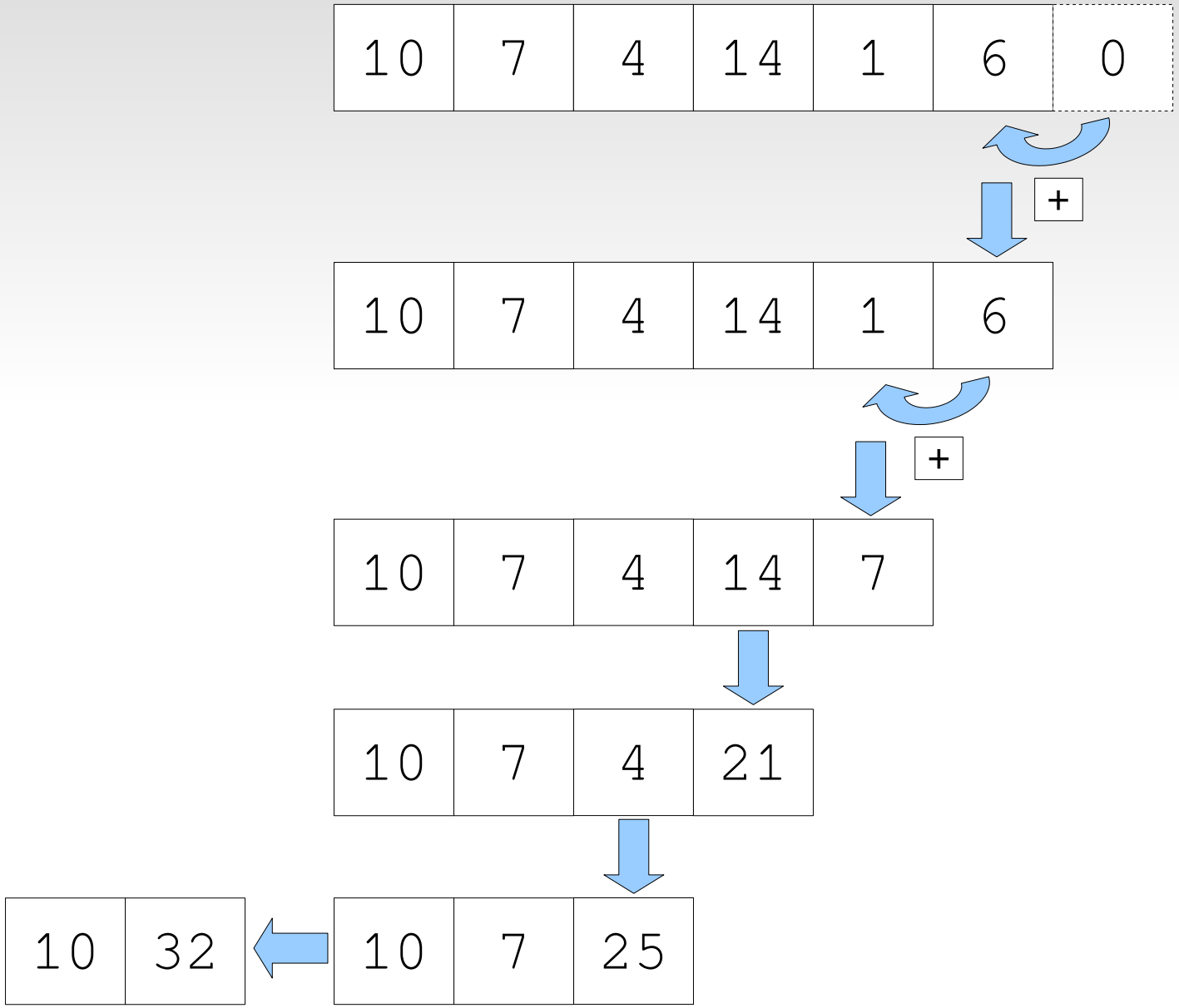
`foldr` - Dobra da dir para esq



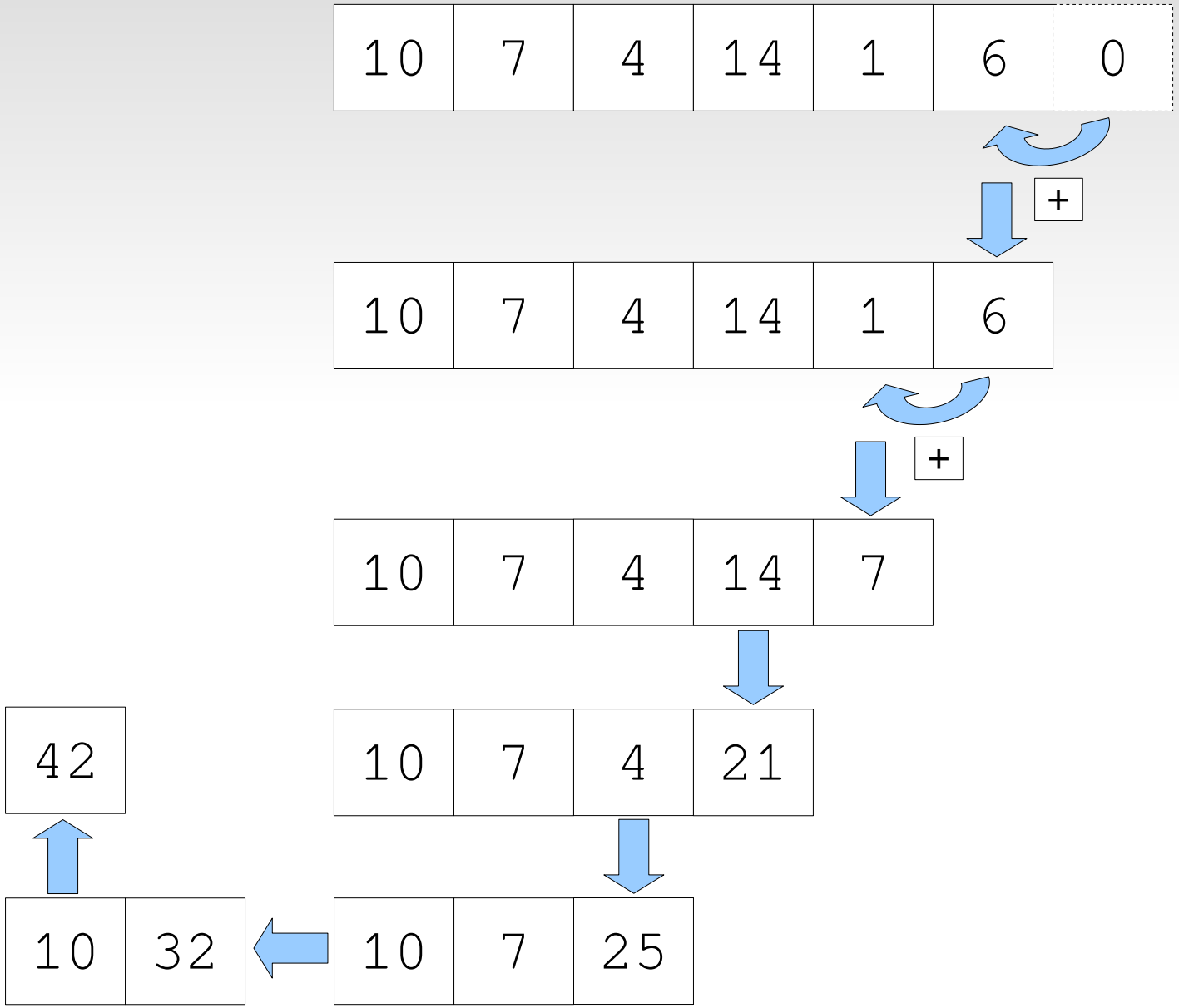
`foldr` - Dobra da dir para esq



`foldr` - Dobra da dir para esq



`foldr` - Dobra da dir para esq



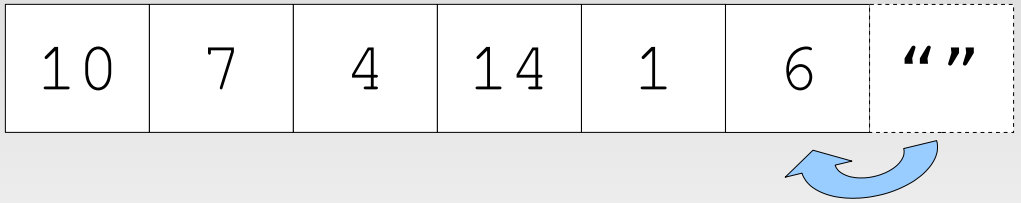
`foldr` - Dobra da dir para esq

10	7	4	14	1	6

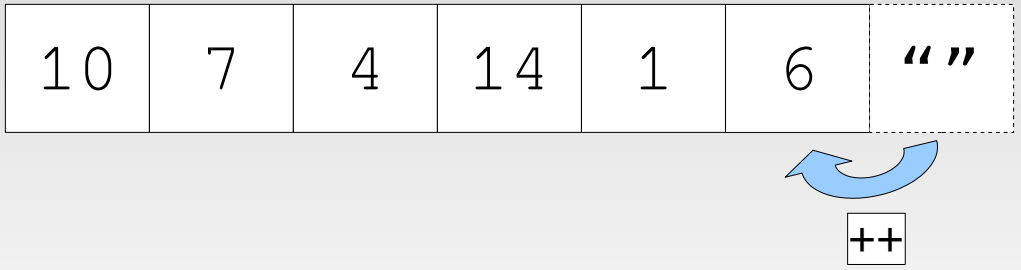
`foldr` - Dobra da dir para esq

10	7	4	14	1	6	“ ”

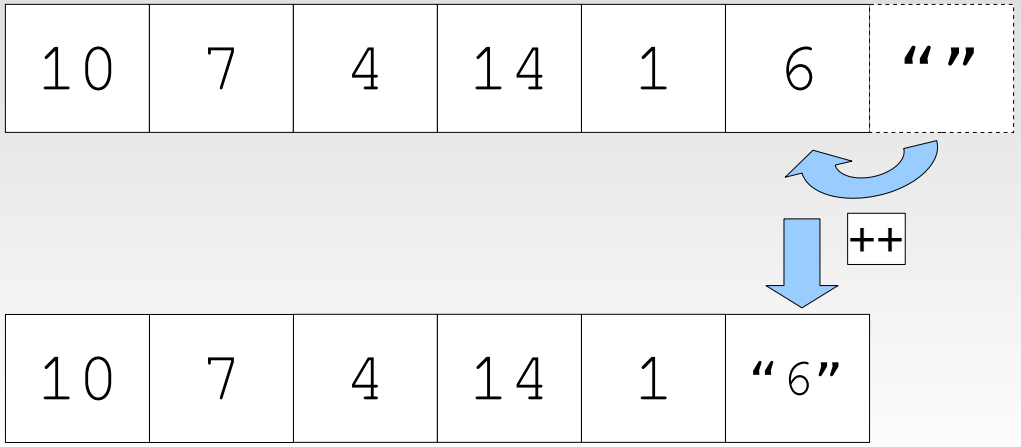
`foldr` - Dobra da dir para esq



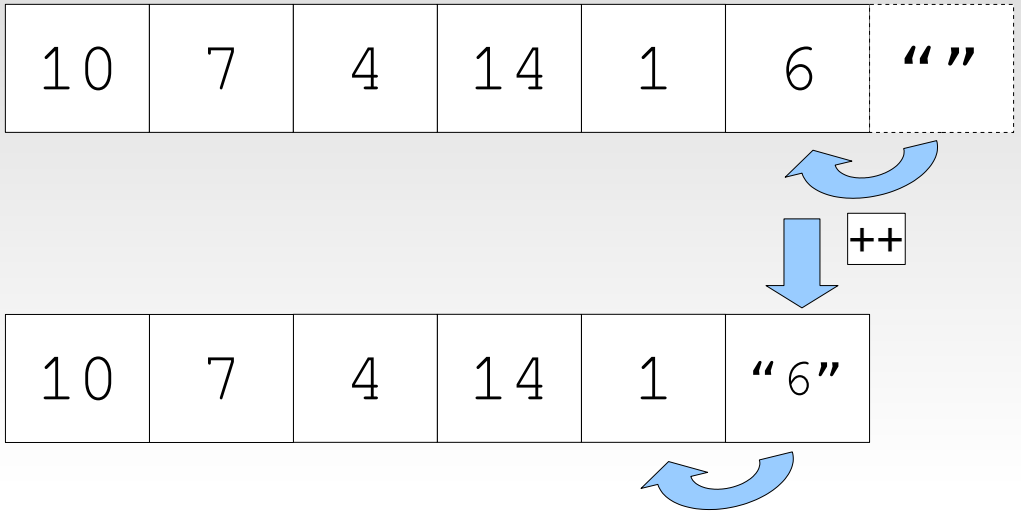
`foldr` - Dobra da dir para esq



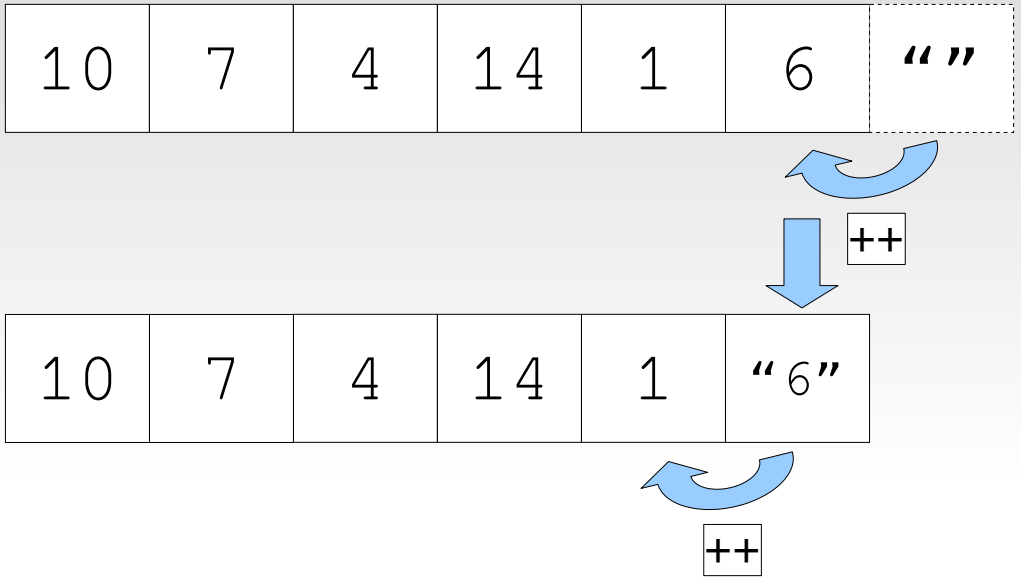
`foldr` - Dobra da dir para esq



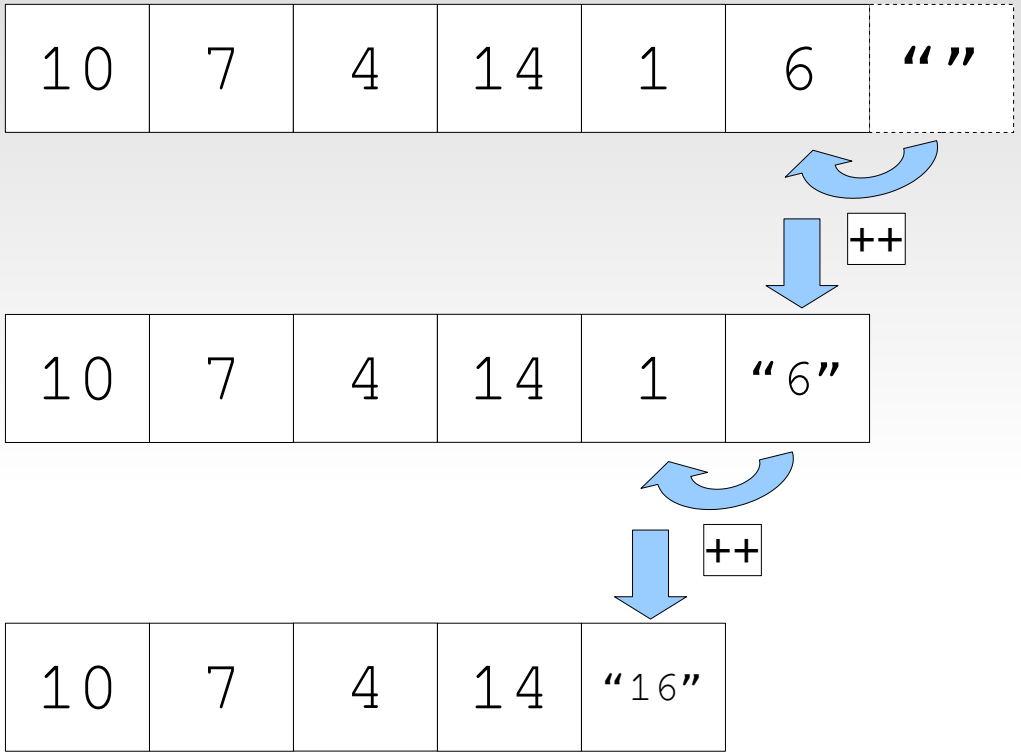
`foldr` - Dobra da dir para esq



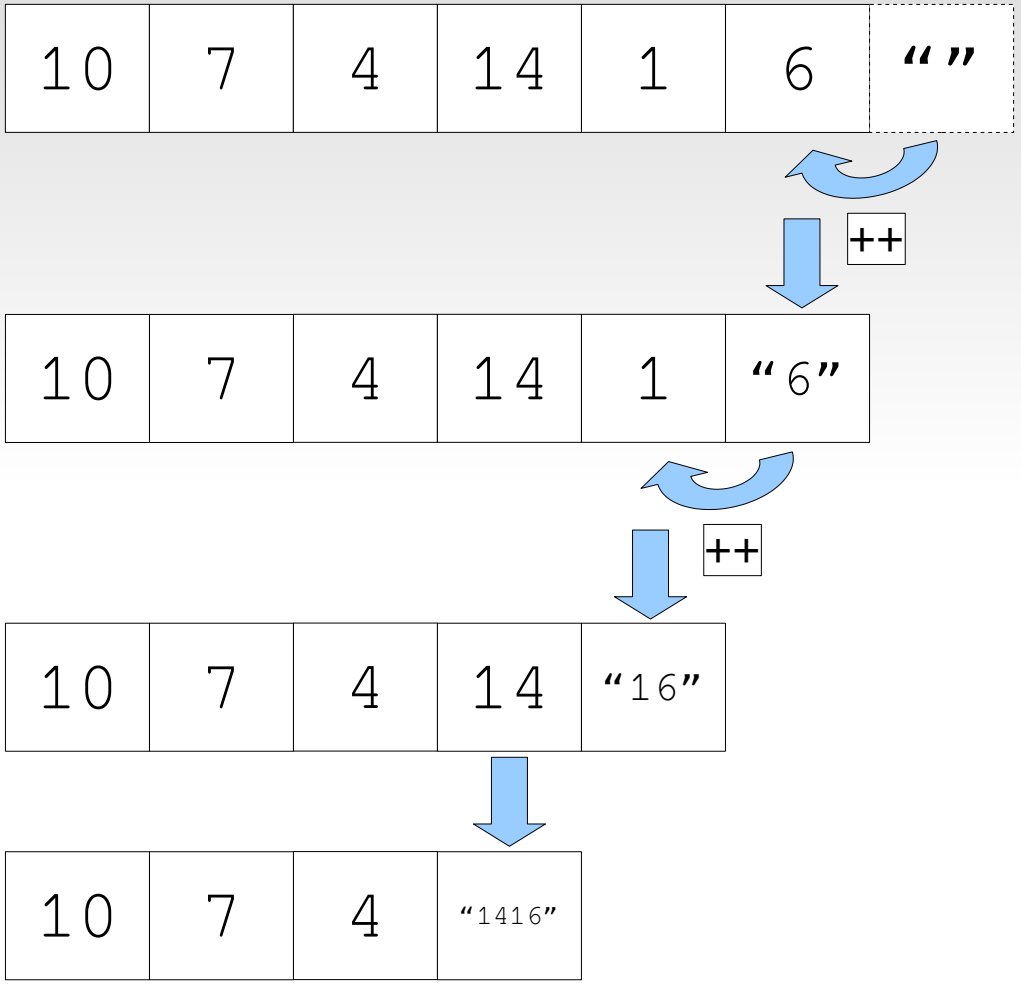
`foldr` - Dobra da dir para esq



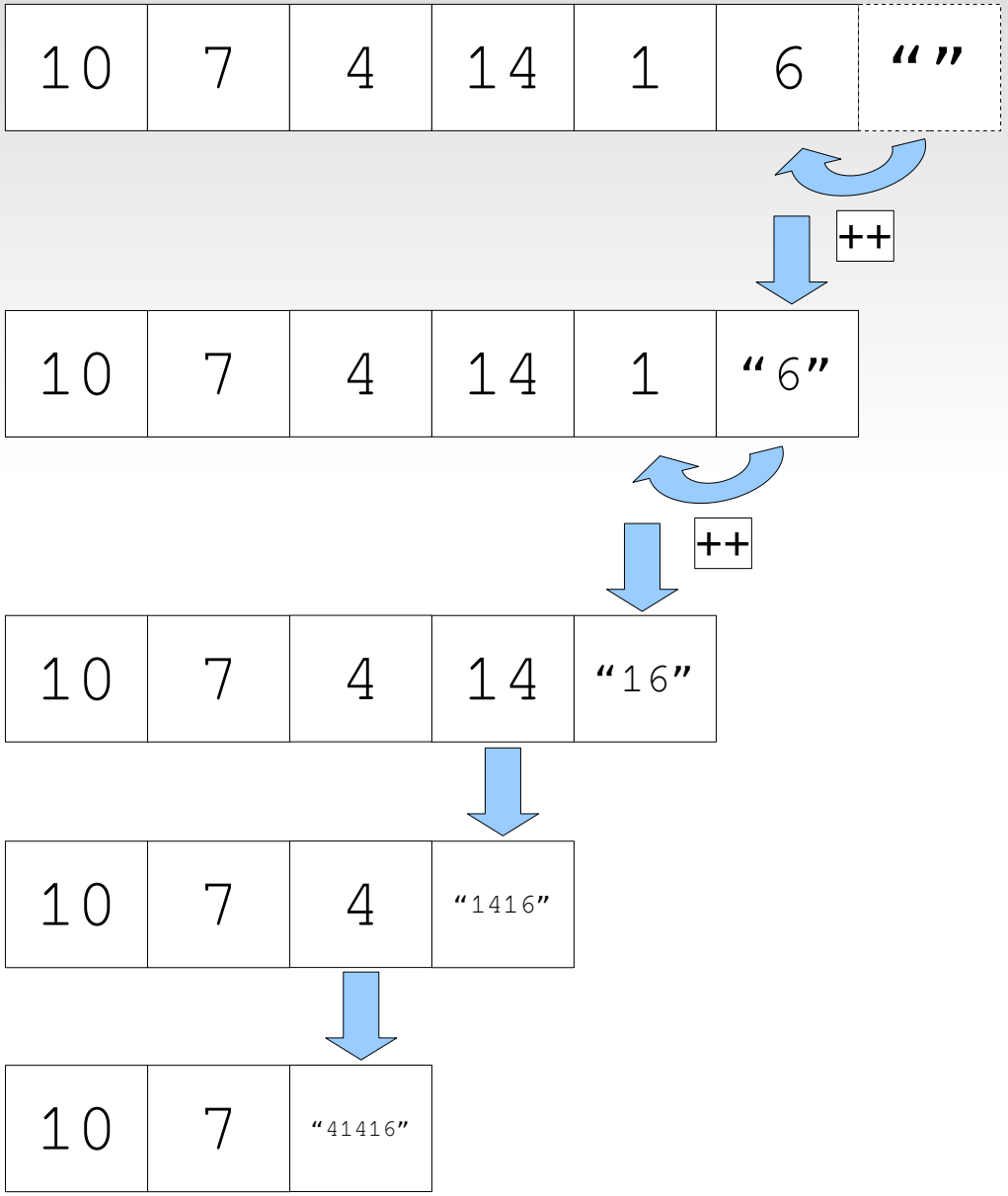
`foldr` - Dobra da dir para esq



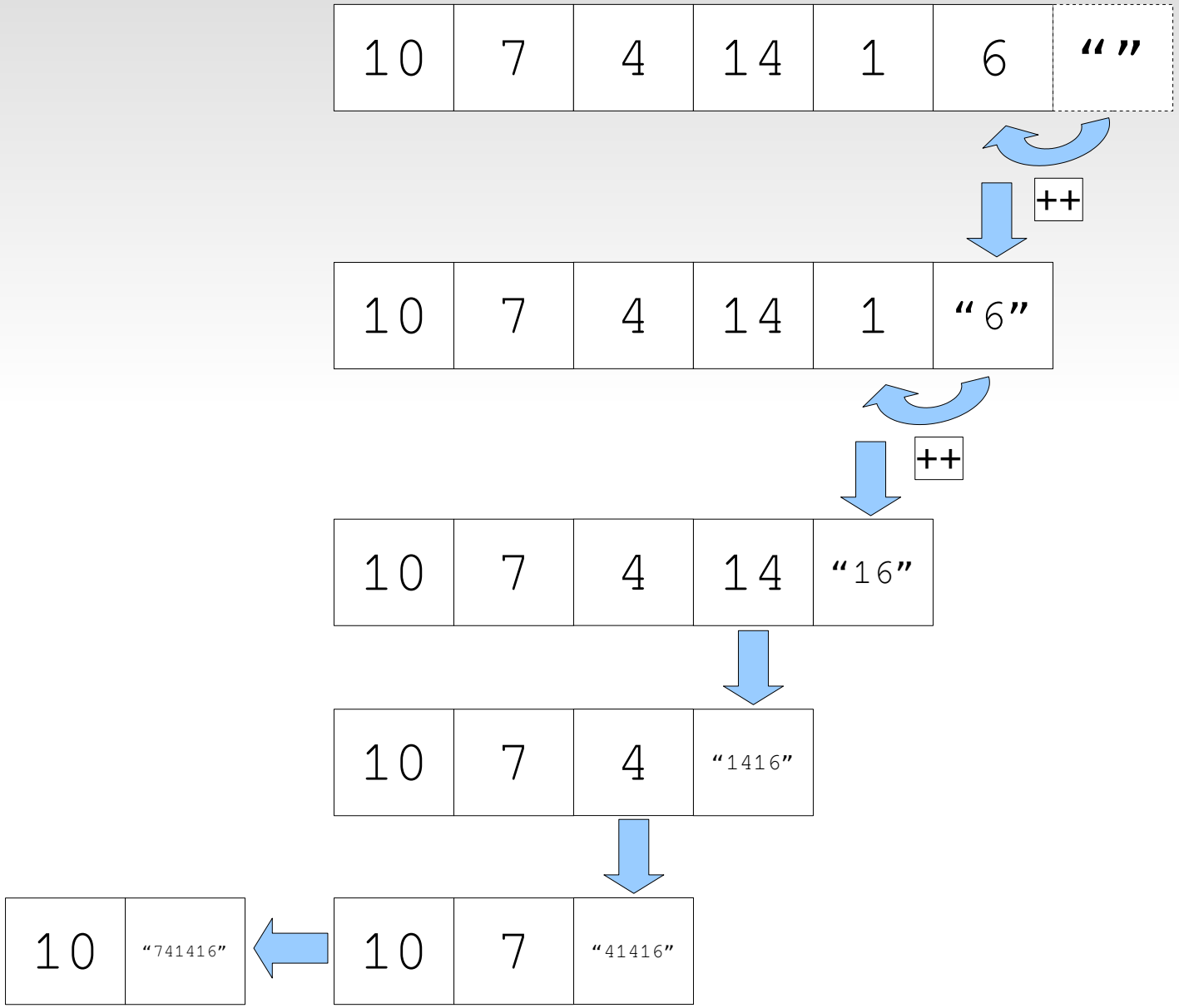
`foldr` - Dobra da dir para esq



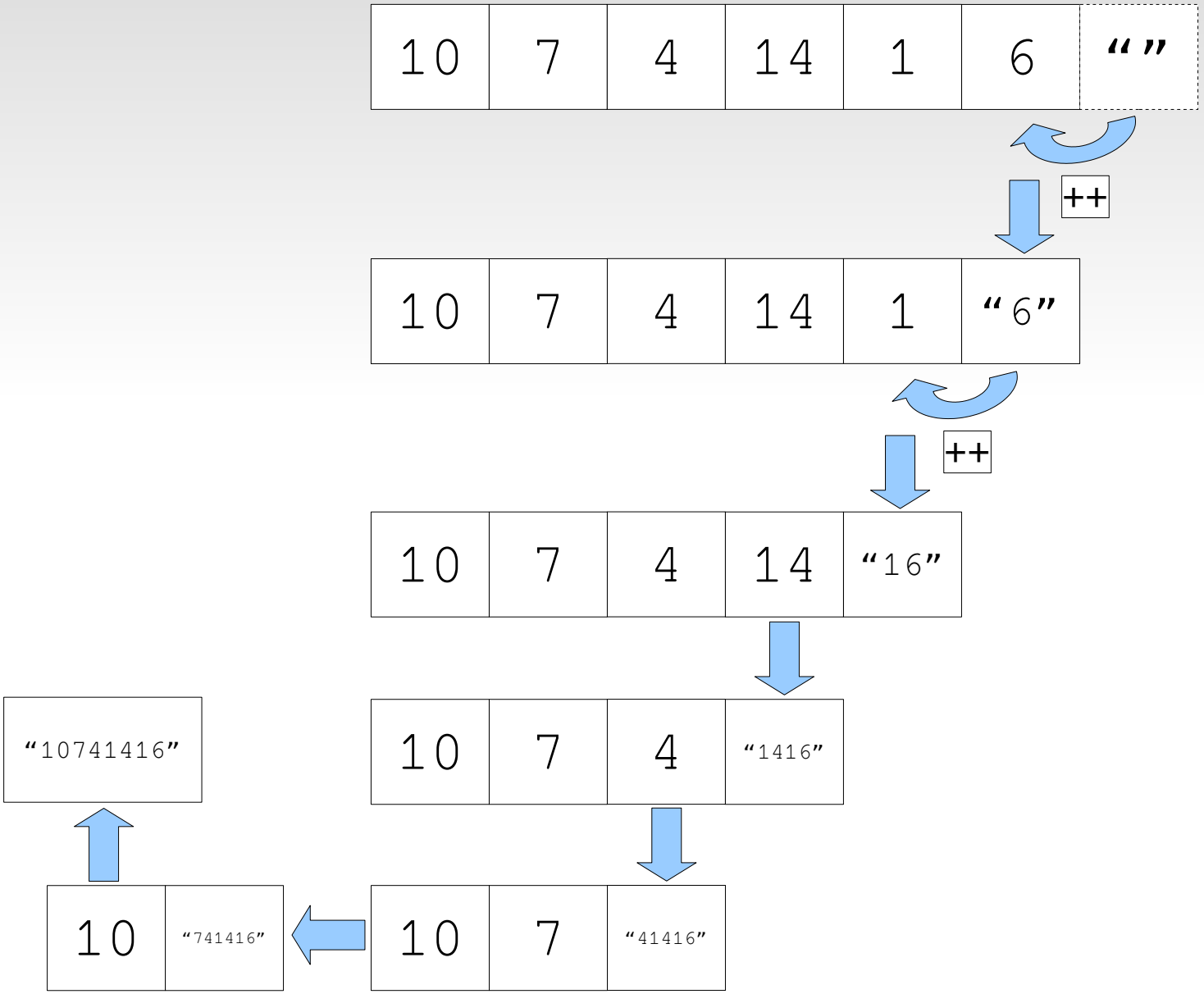
`foldr` - Dobra da dir para esq



`foldr` - Dobra da dir para esq



`foldr` - Dobra da dir para esq



Exercícios

Exercícios

1. Sobre as funções `foldl` e `foldr`...
   - Execute o passo-a-passo de ambas para as seguintes chamadas:
     - `foldr (-) 0 [1,2,3,4]`
     - `foldl (-) 0 [1,2,3,4]`
   - Essas funções são equivalentes? Por quê?

Exercícios

1. Sobre as funções `foldl` e `foldr`...
   - Execute o passo-a-passo de ambas para as seguintes chamadas:
     - `foldr (-) 0 [1,2,3,4]`
     - `foldl (-) 0 [1,2,3,4]`
   - Essas funções são equivalentes? Por quê?
2. Dê 2 exemplos de usos reais para cada uma das funções `map`, `filter` e `fold`.

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Haskell

Haskell

- GHC
  - https://www.haskell.org/ghc/

Haskell

- GHC
  - `https://www.haskell.org/ghc/`
- Repl.it (GHC)
  - `https://repl.it/languages/haskell`

Listas

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`
  - Construção: `:` `-- m = 0 : [1,2,3]`

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`
  - Construção: `:` `-- m = 0 : [1,2,3]`
  - Cabeça: `head` `-- head m → 0`

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`
  - Construção: `:` `-- m = 0 : [1,2,3]`
  - Cabeça: `head` `-- head m → 0`
  - Cauda: `tail` `-- tail m → [1,2,3]`

```
struct node {
    int val;
    struct node* nxt;
};

struct node* n = malloc(...)
```

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`
  - Construção: `:` `-- m = 0 : [1,2,3]`
  - Cabeça: `head` `-- head m → 0`
  - Cauda: `tail` `-- tail m → [1,2,3]`

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`
  - Construção: `:` `-- m = 0 : [1,2,3]`
  - Cabeça: `head` `-- head m → 0`
  - Cauda: `tail` `-- tail m → [1,2,3]`
  - **`++` `last` `null` `length` `reverse` `take` `drop` `elem` `sort`** `...`

Listas

- Conjunto infinito de valores homogêneos
- Principal mecanismo estrutural em linguagens funcionais
- Imutáveis
- Operações (em Haskell)
  - Literais: `[1,2,3]`
  - Construção: `:` `-- m = 0 : [1,2,3]`
  - Cabeça: `head` `-- head m → 0`
  - Cauda: `tail` `-- tail m → [1,2,3]`
  - **`++ last null length reverse take drop elem sort ...`**
  - `https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html`
  - `https://wiki.haskell.org/How_to_work_on_lists`

Exercícios

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Use a função `sort` de Haskell em algum exemplo.

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Use a função `sort` de Haskell em algum exemplo.
2. Crie um exemplo com listas.
   - Use `map`, `filter` e `fold`.

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Use a função `sort` de Haskell em algum exemplo.
2. Crie um exemplo com listas.
   - Use `map`, `filter` e `fold`.
3. Crie um exemplo com listas dentro de listas.
   - Use `map`, `filter` e `fold`.

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Haskell

Haskell

- Estática com inferência de tipos

Haskell

- Estática com inferência de tipos
- Tipos paramétricos

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Sobre Haskell vs Python...
   - Considere os critérios externos e internos do vídeo 1.5.
   - Considere os tempos de amarração do vídeo 2.2.
   1. Compare a inferência de tipos de Haskell com a tipagem dinâmica de Python.
   2. Compare a tipagem paramétrica de Haskell com a tipagem "pato" de Python (*duck typing*).

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Sobre Haskell vs Python...
   - Considere os critérios externos e internos do vídeo 1.5.
   - Considere os tempos de amarração do vídeo 2.2.
   1. Compare a inferência de tipos de Haskell com a tipagem dinâmica de Python.
   2. Compare a tipagem paramétrica de Haskell com a tipagem "pato" de Python (*duck typing*).
2. Quais são os problemas do tipo `String` de Haskell?

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Map

Map

- Transformador de listas

Map

- Transformador de listas
- Função de alta ordem

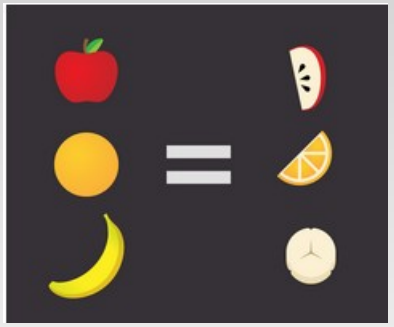
Map

- Transformador de listas
- Função de alta ordem
- Recebe uma função de transformação `f` e uma lista `l`

Map

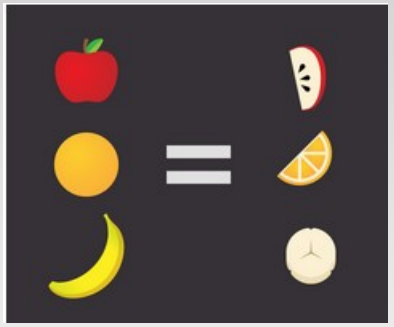
- Transformador de listas
- Função de alta ordem
- Recebe uma função de transformação `f` e uma lista `l`
- `map f l`

Map



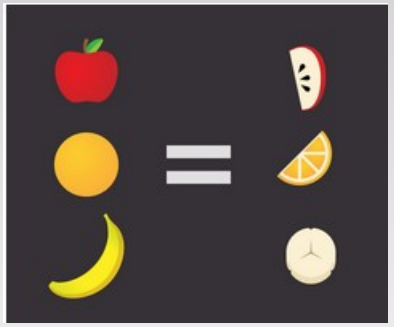
- Transformador de listas
- Função de alta ordem
- Recebe uma função de transformação `f` e uma lista `l`
- `map f l`

Map



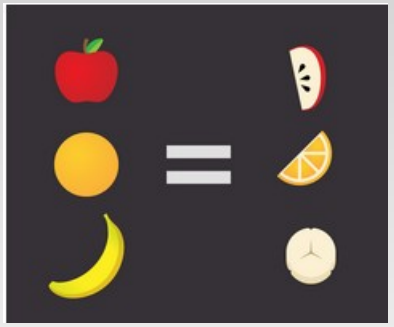
- Transformador de listas
- Função de alta ordem
- Recebe uma função de transformação `f` e uma lista `l`
- `map f l`
- Retorna uma nova lista `m` de mesmo tamanho com valores de tipos possivelmente diferentes

Map



- Transformador de listas
- Função de alta ordem
- Recebe uma função de transformação `f` e uma lista `l`
- `map f l`
- Retorna uma nova lista `m` de mesmo tamanho com valores de tipos possivelmente diferentes
- Aplica `f` a cada elemento de `l`, *mapeando-os* para novos elementos em `m`

Map



- Transformador de listas
- Função de alta ordem
- Recebe uma função de transformação `f` e uma lista `l`
- `map f l`
- Retorna uma nova lista `m` de mesmo tamanho com valores de tipos possivelmente diferentes
- Aplica `f` a cada elemento de `l`, *mapeando-os* para novos elementos em `m`
-
```
map :: (a → b) → [a] → [b]
       f         l      m
```

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Implemente a função `map` em Haskell usando recursão.

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Implemente a função `map` em Haskell usando recursão.
2. Considere uma lista em que cada elemento é uma lista com três inteiros representando os coeficientes *a,b,c* de uma equação de segundo grau:
   - ```
     l1 :: [[Int]]
     l1 = [ [1,12,-13], [1,2,1], [2,-16,-18], ... ]
     ```
     ```

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Implemente a função `map` em Haskell usando recursão.
2. Considere uma lista em que cada elemento é uma lista com três inteiros representando os coeficientes *a,b,c* de uma equação de segundo grau:
   - `l1 :: [[Int]]`

     `l1 = [ [1,12,-13], [1,2,1], [2,-16,-18], ... ]`
   - Calcule uma nova lista com todas as raízes de todas as equações.
   - A resposta de cada equação deve seguir o seguinte formato:
     - `[]:` nenhuma raíz
     - `[x]:` 1 raíz x
     - `[x,y]:` 2 raízes x e y

   `m1 = [ [1,-13], [], [9,-1], … ]`

Exercícios

- Em todos os exercícios, sempre cite as suas fontes.

1. Implemente a função `map` em Haskell usando recursão.
2. Considere uma lista em que cada elemento é uma lista com três inteiros representando os coeficientes *a,b,c* de uma equação de segundo grau:
   - `l1 :: [[Int]]`

     `l1 = [ [1,12,-13], [1,2,1], [2,-16,-18], ... ]`
   - Calcule uma nova lista com todas as raízes de todas as equações.
   - A resposta de cada equação deve seguir o seguinte formato:
     - `[]:` nenhuma raíz
     - `[x]:` 1 raíz x
     - `[x,y]:` 2 raízes x e y

   `m1 = [ [1,-13], [], [9,-1], … ]`

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Haskell

- Tuplas
- Pattern Matching (casamento de padrão)

Listas vs Tuplas

Listas vs Tuplas

- Lista

Listas vs Tuplas

- Lista
  - Conjunto **infinito** de valores **homogêneos**

Listas vs Tuplas

- Lista
  - Conjunto **infinito** de valores **homogêneos**
  - `[1,2,3,…] :: [Int]`
- Tuplas
  - Conjunto **finito** de valores **heterogêneos**
  - `(“Joao”,25,True) :: (String,Int,Bool)`

Listas vs Tuplas

- Lista
  - Conjunto **infinito** de valores **homogêneos**
  - `[1,2,3,…] :: [Int]`
- Tuplas
  - Conjunto **finito** de valores **heterogêneos**
  - `(“Joao”,25,True) :: (String,Int,Bool)`

```
struct node {
    int val;
    struct node* nxt;
};
```

Listas vs Tuplas

- Lista
  - Conjunto **infinito** de valores **homogêneos**
  - `[1,2,3,…] :: [Int]`
- Tuplas
  - Conjunto **finito** de valores **heterogêneos**
  - `(“Joao”,25,True) :: (String,Int,Bool)`

```
struct node {
    int val;
    struct node* nxt;
};
```

```
struct pessoa {
   char nome[255];
   int  idade;
   bool masc;
}
```

Pattern Matching

Pattern Matching

- `if` estrutural

Pattern Matching

- **`if`** estrutural

```
case lista of
  1 : 2 : [] -> ...
  5 : 1 : 0 : [] -> ...
  [] -> ...
```

Pattern Matching

- **`if`** estrutural

```
case lista of
  1 : 2 : [] -> ...
  5 : 1 : 0 : [] -> ...
  [] -> ...
```

```
case tup of
  (“Joao”, 25, True) -> ...
  (“Maria”, 10, False) -> ...
```

Pattern Matching

- **`if`** estrutural

```
case lista of
  1 : 2 : [] -> ...
  5 : 1 : 0 : [] -> ...
  [] -> ...
```

```
case tup of
  (“Joao”, 25, True) -> ...
  (“Maria”, 10, False) -> ...
```

- Permite uso de variáveis que “casam” com qualquer valor
  - **A variável pode ser usada na expressão de retorno**

Pattern Matching

- **`if`** estrutural

```
case lista of
  1 : 2 : [] -> ...
  5 : 1 : 0 : [] -> ...
  [] -> ...
```

```
case tup of
  (“Joao”, 25, True) -> ...
  (“Maria”, 10, False) -> ...
```

- Permite uso de variáveis que “casam” com qualquer valor
  - **A variável pode ser usada na expressão de retorno**

```
case lista of
  1 : x -> {usa x}
  y -> {usa y}
```

Pattern Matching

- **`if`** estrutural

```
case lista of
  1 : 2 : [] -> ...
  5 : 1 : 0 : [] -> ...
  [] -> ...
```

```
case tup of
  (“Joao”, 25, True) -> ...
  (“Maria”, 10, False) -> ...
```

- Permite uso de variáveis que “casam” com qualquer valor
  - **A variável pode ser usada na expressão de retorno**

```
case lista of
  1 : x -> {usa x}
  y -> {usa y}
```

```
case tup of
  (“Joao”, x, y) -> {usa x e y}
  (i, 10, _) -> {usa i}
```

Tuplas + Pattern Matching

Tuplas + Pattern Matching

```
struct pessoa {
    char nome[255];
    int  idade;
    bool masc;
}
```

Tuplas + Pattern Matching

```
struct pessoa {
    char nome[255];
    int  idade;
    bool masc;
}
```

```
type Pessoa = (String,Int,Bool)

nome :: Pessoa -> String
nome (x,_,_) = x

idade :: Pessoa -> Int
idade (_,y,_) = y

masc :: Pessoa -> Bool
masc (_,_,z) = z
```

Exercícios

Exercícios

1. Crie um tipo novo para representar um Aluno com nome, nota da P1 e nota da P2.

Exercícios

1. Crie um tipo novo para representar um Aluno com nome, nota da P1 e nota da P2.
2. Crie uma lista com pelo menos 10 alunos. Usando a função `map`...
    1. Exiba uma lista com o nome de cada aluno.
    2. Exiba uma lista com a média de notas de cada aluno.

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Filter

Filter

- Filtro de listas

Filter

- Filtro de listas
- Função de alta ordem

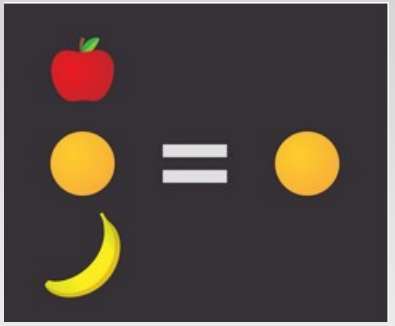
Filter

- Filtro de listas
- Função de alta ordem
- Recebe uma função de filtro `f` e uma lista `l`

Filter

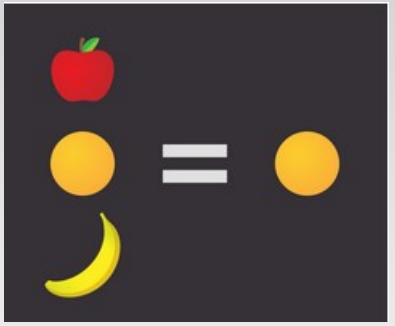
- Filtro de listas
- Função de alta ordem
- Recebe uma função de filtro `f` e uma lista `l`
- `filter f l`

Filter



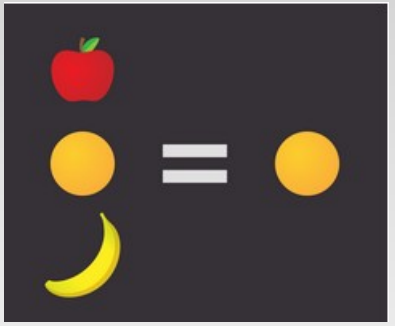
- Filtro de listas
- Função de alta ordem
- Recebe uma função de filtro `f` e uma lista `l`
- `filter f l`

Filter



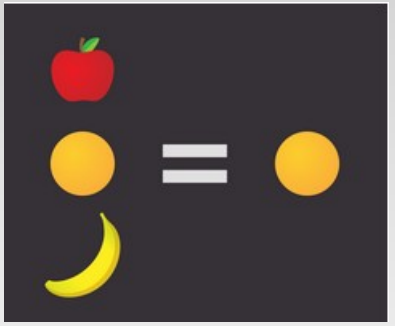
- Filtro de listas
- Função de alta ordem
- Recebe uma função de filtro `f` e uma lista `l`
- `filter f l`
- Retorna uma nova lista `m` de no máximo mesmo tamanho com valores do mesmo tipo

Filter



- Filtro de listas
- Função de alta ordem
- Recebe uma função de filtro `f` e uma lista `l`
- `filter f l`
- Retorna uma nova lista `m` de no máximo mesmo tamanho com valores do mesmo tipo
- Aplica `f` a cada elemento de `l`, *filtrando* os que satisfizerem `f` para novos elementos em `m`

Filter



- Filtro de listas
- Função de alta ordem
- Recebe uma função de filtro `f` e uma lista `l`
- `filter f l`
- Retorna uma nova lista `m` de no máximo mesmo tamanho com valores do mesmo tipo
- Aplica `f` a cada elemento de `l`, *filtrando* os que satisfizerem `f` para novos elementos em `m`
- `filter :: (a → Bool) → [a] → [a]`

  `f` `l` `m`

Exercícios

Exercícios

- Considere o tipo Aluno com nome, nota da P1 e nota da P2. Considere uma lista com pelo menos 10 alunos.

1. Exiba uma lista com somente o nome dos alunos aprovados com média >= 7. Use `filter` e `map`.

Programação Funcional

(continuação...)

Estrutura de Linguagens

`https://github.com/fsantanna-uerj/EDL/`

Francisco Sant'Anna

`francisco@ime.uerj.br`

Foldr

Foldr

- Agregador de listas

Foldr

- Agregador de listas
- Função de alta ordem

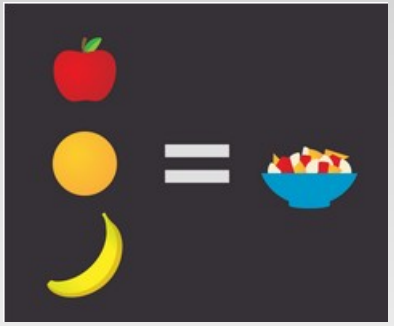
Foldr

- Agregador de listas
- Função de alta ordem
- Recebe uma função de acumulação `f`, um valor inicial v_0 e uma lista `l`

Foldr

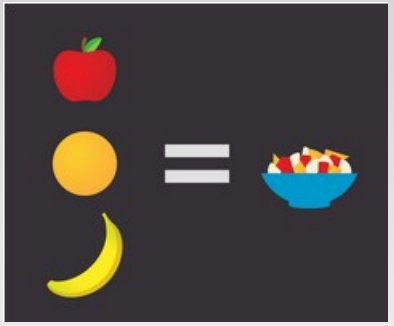
- Agregador de listas
- Função de alta ordem
- Recebe uma função de acumulação `f`, um valor inicial v_0 e uma lista `l`
- `foldr f` v_0 `l`

Foldr



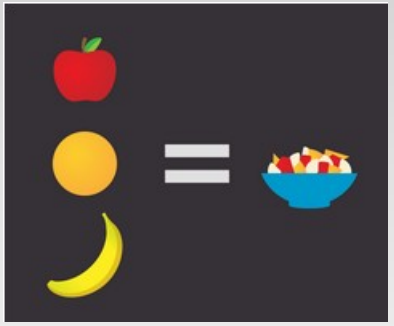
- Agregador de listas
- Função de alta ordem
- Recebe uma função de acumulação `f`, um valor inicial v_0 e uma lista `l`
- `foldr f` v_0 `l`

Foldr



- Agregador de listas
- Função de alta ordem
- Recebe uma função de acumulação `f`, um valor inicial v_0 e uma lista `l`
- `foldr f` v_0 `l`
- Retorna um valor qualquer v_n do mesmo tipo de v_0

Foldr



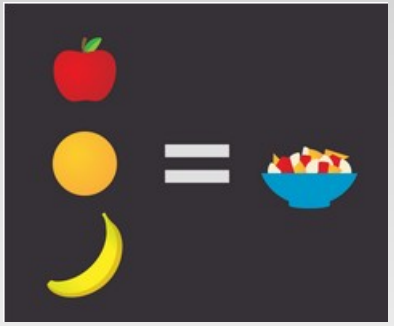
- Agregador de listas
- Função de alta ordem
- Recebe uma função de acumulação `f`, um valor inicial v_0 e uma lista `l`
- `foldr f` v_0 `l`
- Retorna um valor qualquer v_n do mesmo tipo de v_0
- `aplica f (`l_n `,` v_0`)` → v_1

  `f (`l_{n-1} `,` v_1`)` → v_2

  `...`

  `f (`l_1 `,` v_{n-1}`)` → v_n (que é o valor final)

Foldr



- Agregador de listas
- Função de alta ordem
- Recebe uma função de acumulação `f`, um valor inicial v_0 e uma lista `l`
- `foldr f` v_0 `l`
- Retorna um valor qualquer v_n do mesmo tipo de v_0
- `aplica f (`l_n `,` v_0`)` → v_1

  `f (`l_{n-1} `,` v_1`)` → v_2

  `...`

  `f (`l_1 `,` v_{n-1}`)` → v_n (que é o valor final)
- `foldr :: (a → b → b) → b → [a] → b`

  `f` v_0 `l` v_n

Exercícios

- Considere o tipo Aluno com nome, nota da P1 e nota da P2. Considere uma lista com pelo menos 10 alunos.

1. Exiba a média da turma. Use `map`, `foldr` e `length`.
2. Exiba a quantidade de alunos aprovados.
3. Exiba a média dos alunos reprovados. Use `map`, `filter`, `foldr` e `length`.

Programação Funcional

Estrutura de Linguagens

https://github.com/fsantanna-uerj/EDL/

Francisco Sant'Anna

francisco@ime.uerj.br

Programação Funcional

- Comando → Expressão
- Sequência → Dependência
- Atribuição → Definição
- Funções…
  - de primeira classe
  - de alta ordem
  - puras
  - recursivas

Programação Funcional

Estrutura de Linguagens

https://github.com/fsantanna-uerj/EDL/

Francisco Sant'Anna

francisco@ime.uerj.br

Exemplo

Exemplo

```
bandas :: [[String]]
bandas = [ ["Gilberto Gil"],
           ["Victor","Leo"],
           ["Gonzagao"],
           ["Claudinho","Bochecha"] ]

type Musica = (String, Int, Int)
musicas :: [Musica]
musicas = [ ("Aquele Abraco",        0, 100),
            ("Esperando na Janela", 0, 150),
            ("Borboletas",           1, 120),
            ("Asa Branca",           2, 120),
            ("Assum Preto",          2, 140),
            ("Vem Morena",           2, 200),
            ("Nosso Sonho",          3, 150),
            ("Quero te Encontrar",   3, 100) ]
```

Exemplo

1. Somente o nome das músicas:
   - `["Aquele Abraco", "Esperando na Janela", ...]`

```
bandas :: [[String]]
bandas = [ ["Gilberto Gil"],
           ["Victor","Leo"],
           ["Gonzagao"],
           ["Claudinho","Bochecha"] ]

type Musica = (String, Int, Int)
musicas :: [Musica]
musicas = [ ("Aquele Abraco",        0, 100),
            ("Esperando na Janela", 0, 150),
            ("Borboletas",           1, 120),
            ("Asa Branca",           2, 120),
            ("Assum Preto",          2, 140),
            ("Vem Morena",           2, 200),
            ("Nosso Sonho",          3, 150),
            ("Quero te Encontrar",   3, 100) ]
```

Exemplo

1. Somente o nome das músicas:
   - `["Aquele Abraco", "Esperando na Janela", ...]`
2. Somente músicas com >= 2min:
   - `[("Esperando...",1,150), ("Borboletas",2,120), ... ]`

```
bandas :: [[String]]
bandas = [ ["Gilberto Gil"],
           ["Victor","Leo"],
           ["Gonzagao"],
           ["Claudinho","Bochecha"] ]

type Musica = (String, Int, Int)
musicas :: [Musica]
musicas = [ ("Aquele Abraco",        0, 100),
            ("Esperando na Janela", 0, 150),
            ("Borboletas",           1, 120),
            ("Asa Branca",           2, 120),
            ("Assum Preto",          2, 140),
            ("Vem Morena",           2, 200),
            ("Nosso Sonho",          3, 150),
            ("Quero te Encontrar",   3, 100) ]
```

Exemplo

1. Somente o nome das músicas:
   - `["Aquele Abraco", "Esperando na Janela", ...]`
2. Somente músicas com >= 2min:
   - `[("Esperando...",1,150), ("Borboletas",2,120), ... ]`
3. Maior duração:
   - `200`

```
bandas :: [[String]]
bandas = [ ["Gilberto Gil"],
           ["Victor","Leo"],
           ["Gonzagao"],
           ["Claudinho","Bochecha"] ]

type Musica = (String, Int, Int)
musicas :: [Musica]
musicas = [ ("Aquele Abraco",        0, 100),
            ("Esperando na Janela", 0, 150),
            ("Borboletas",           1, 120),
            ("Asa Branca",           2, 120),
            ("Assum Preto",          2, 140),
            ("Vem Morena",           2, 200),
            ("Nosso Sonho",          3, 150),
            ("Quero te Encontrar",   3, 100) ]
```

Exemplo

1. Somente o nome das músicas:
   - `["Aquele Abraco", "Esperando na Janela", ...]`
2. Somente músicas com >= 2min:
   - `[("Esperando...",1,150), ("Borboletas",2,120), ... ]`
3. Maior duração:
   - `200`
4. Nomes com >= 2min:
   - `["Esperando...", "Borbolet`

```
bandas :: [[String]]
bandas = [ ["Gilberto Gil"],
           ["Victor","Leo"],
           ["Gonzagao"],
           ["Claudinho","Bochecha"] ]

type Musica = (String, Int, Int)
musicas :: [Musica]
musicas = [ ("Aquele Abraco",        0, 100),
            ("Esperando na Janela", 0, 150),
            ("Borboletas",           1, 120),
            ("Asa Branca",           2, 120),
            ("Assum Preto",          2, 140),
            ("Vem Morena",           2, 200),
            ("Nosso Sonho",          3, 150),
            ("Quero te Encontrar",   3, 100) ]
```

Exemplo

1. Somente o nome das músicas:
   - `["Aquele Abraco", "Esperando na Janela", ...]`
2. Somente músicas com >= 2min:
   - `[("Esperando...",1,150), ("Borboletas",2,120), ... ]`
3. Maior duração:
   - `200`
4. Nomes com >= 2min:
   - `["Esperando...", "Borbolet`
5. Bandas:
   - `[["Gilberto Gil"],["Gilber`
     `["Victor","Leo"],`
     `["Gonzagao"],["Gonzagao"],...]`

```
bandas :: [[String]]
bandas = [ ["Gilberto Gil"],
           ["Victor","Leo"],
           ["Gonzagao"],
           ["Claudinho","Bochecha"] ]

type Musica = (String, Int, Int)
musicas :: [Musica]
musicas = [ ("Aquele Abraco",        0, 100),
            ("Esperando na Janela", 0, 150),
            ("Borboletas",           1, 120),
            ("Asa Branca",           2, 120),
            ("Assum Preto",          2, 140),
            ("Vem Morena",           2, 200),
            ("Nosso Sonho",          3, 150),
            ("Quero te Encontrar",   3, 100) ]
```

Exercícios

Exercícios

- Considere a base de músicas e autores do slide anterior.

Exercícios

- Considere a base de músicas e autores do slide anterior.

1. Exiba os nomes dos componentes de uma banda em uma única string com os nomes separados por vírgula. Pode usar a função `intercalate`.

Exercícios

- Considere a base de músicas e autores do slide anterior.

1. Exiba os nomes dos componentes de uma banda em uma única string com os nomes separados por vírgula. Pode usar a função `intercalate`.
2. *Pretty-print* das músicas:
   - ```
     Nome: Borboletas

     Autores: Victor, Leo

     Duracao: 120

     ...
     ```
     ```

Programação Funcional

Estrutura de Linguagens

https://github.com/fsantanna-uerj/EDL/

Francisco Sant'Anna

francisco@ime.uerj.br