PROGRAMAÇÃO WEB CLIENTE E SERVIDOR

UFCDs 5414-5417

GUIA DE LABORATÓRIO 4.1

FUNÇÕES

(ßeta)

OBJECTIVOS

- Aprender a maioria dos conceitos relacionados com funções, tais como, parâmetro, argumento, valor de retorno, função interna/externa, lambda, closure, etc.

INSTRUÇÕES

PARTE I – (RE)INTRODUÇÃO: FUNÇÕES, PARÂMETROS, INVOCAÇÃO E VARIÁVEIS LOCAIS

1. Aceda ao REPL do navegador ou do Node.js.

***NOTAS:** 1) Para vários dos exemplos que se seguem não é exibida a linha de comandos da consola. Pode introduzir esses exemplos linha-a-linha na consola, mas é mais conveniente escrever o código num ficheiro `.js` (um script) e depois executá-lo via Node ou através de uma página HTML com o elemento* **`script`** *apropriado.* **2)** *Todos os exemplos que utilizam as funções `prompt` e `alert` devem ser executados no navegador.*

2. Num dos exercícios anteriores desenvolvemos código para calcular a potência de dois números. Como essa é uma necessidade comum em programação (eg, computação gráfica), vamos definir uma função designada por `potencia` para podermos utilizar noutras ocasiões. Introduza o seguinte:

```
function potencia (base, expoente) {
    let resultado = 1;
    for (let i = 1; i <= expoente; resultado *= base, i += 1);
    return resultado;
}
```

As funções permitem dividir um programa em sub-operações e permitem que os programadores utilizem o que outros produziram em vez de começarem do zero. Uma correcta ***decomposição*** *do programa em funções permite dividi-lo em várias partes independentes que podem e devem ignorar os detalhes de implementação umas das outras. Uma função é, assim, uma espécie de caixa negra que recebe instruções para executar uma operação, executa-a e, no final, comunica o resultado.*
Tal como uma variável, que é uma zona de dados com um nome, o nome da variável, uma ***função*** *consiste de um bloco de código com um nome. Esse bloco de código pode ser executado desde que uma outra função, a* ***função invocadora****, invoque o nome da* ***função invocada*** *algures.*

Como vimos num laboratório anterior, em JavaScript iniciamos a definição de uma função com a palavra reservada ***function****, seguida do nome da função. A* ***função invocada*** *pode necessitar de "dados" para processar. Se for esse o caso, a função invocadora deve passá-los para os parâmetros da função. O que é um* ***parâmetro****? É uma variável que a função declarou na sua assinatura e que serve para a* ***função invocadora*** *"comunicar" com função invocada. Em JavaScript uma função pode possuir zero ou mais parâmetros e estes devem ser indicados logo a seguir ao nome da função entre parênteses. Os parênteses são obrigatórios, mesmo que a função não necessite de parâmetros. Utilizamos a vírgula para separar parâmetros.*

O conjunto de parâmetros de uma função é, por vezes, designado por ***lista de parâmetros****. Após a lista de parâmetros seguem-se as instruções da função entre chavetas.*

Por outro lado, uma função também pode comunicar informação de volta através da instrução ***return****. Assim que o fluxo de execução dentro da função "atinge" uma instrução* ***return****, a função termina e o resultado da expressão "à direita" do* ***return****, se alguma houver, é devolvido para a função invocadora.*

Uma função pode definir variáveis sendo que estas são designadas por ***variáveis locais*** *e deixam de existir assim que a função terminar. Uma variável local existe apenas dentro do bloco de instruções da função.*

Aplicando estas noções à função desenvolvida neste passo do laboratório:

*. **nome da função**: potencia*
*. **parâmetros**: base e expoente*
*. **retorno**: o valor da variável resultado*
*. **bloco de instruções da função**:*

```
{
    let resultado = 1;
    for (let i = 1; i <= expoente; resultado *= base, i += 1);
    return resultado;
}
```

3. Invoque a função no REPL:

```
>>> potencia(2, 3)
8
>>> potencia(4, 1)
4
>>> potencia(2, 0)
1
>>> alert("5 ao quadrado: " + potencia(5,2));
```

```
>>> console.log(potencia(4, 7))
16384
>>> potencia(2, 3) * potencia(3, 4)
648
>>> potencia(3, 2)*3 + 10/potencia(2,2)
29.5
```

4. Uma função é ela própria um "valor" (mais concretamente, um objecto) com um tipo de dados. Introduza na linha de comandos:

```
>>> potencia
[Function: potencia]

>>> typeof potencia
'function'
```

A invocação de uma função com parâmetros e valor de retorno envolve os seguintes passos:

1. *Passar os valores dos* ***argumentos*** *da* ***função invocadora*** *para os correspondene parâmetros da* ***função invocada****, se for o caso disso.*
2. *Suspender a execução na função invocadora e executar a função invocada.*
3. *Após a execução da função invocada ter terminado, se houver necessidade, passar o valor devolvido para a função invocadora.*
4. *Retomar a execução da função invocadora.*

Cada parâmetro constitui um novo nome, ie, uma nova variavel local (ver em baixo). Os valores dos argumentos são passados para os parâmetros da função invocada por ***referência de objecto*** *(ou por* ***cópia de referência****), isto é, cada parâmetro recebe uma cópia da referência para o objecto referenciado pelo argumento. Se o valor do argumento for mutável (eg, uma lista) e a função invocada alterar esse valor através do parâmetro correspondente, essa alteração irá afectar a função invocadora.*
As variaveis definidas dentro de uma função, parâmetros incluidos, são ***locais*** *à função, isto é, elas "existem" apenas dentro da função e durante a invocação desta (são eliminadas quando a função terminar).*
O bloco de instruções da função pode ser qualquer sequência de instruções validas em JavaScript. No entanto, existem algumas instruções que só são validas dentro de funções. Por exemplo, a instrução ***return*** *só pode aparecer dentro uma função e serve para devolver um valor para a função invocadora. Assim que o fluxo de execução dentro da função invocada "atinge" uma instrução* ***return****, a função termina e o resultado da expressão "à direita" do* ***return****, se alguma existir, é devolvido para a função invocadora. Note-se que, em JavaScript, não é obrigatório que uma função defina parâmetros nem que devolva qualquer valor explícito. Se uma função não possuir um return, então o JavaScript garante que essa função devolve* ***undefined****. Por vezes designamos por* ***procedimento*** *ou* ***rotina*** *uma função que não devolve qualquer resultado.*

5. Podemos guardar uma função numa variável. Tente:

```
>>> let pot = potencia
>>> console.log("Dois ao cubo: " + pot(2, 3))
Dois ao cubo: 8
```

6. Vamos definir uma função feita para pedir um valor numérico ao utilizador. Esse valor deverá estar contido entre dois valores. Enquanto que o utilizador não inserir o valor correctamente, a função volta a solicitar a introdução do número. Introduza o seguinte

```
function pedeNumero(texto, limiteInf, limiteSup) {
    let numeroInvalido = true;
    let numero = 0;
    while (numeroInvalido) {
        numero = parseFloat(prompt(texto));
        numeroInvalido = !(numero >= limiteInf && numero <= limiteSup);
        if (numeroInvalido) {
            alert("Valor inválido!");
        }
    }
    return numero;
}
```

7. A título de curiosidade, esta função também poderia ser escrita da seguinte forma:

```
function pedeNumero(texto, limiteInf, limiteSup) {
    let numero = 0;
    while (true) {
```

```
        numero = parseFloat(prompt(texto));
        if (numero >= limiteInf && numero <= limiteSup) {
            break;
        }
        alert("Valor inválido!");
    }
    return numero;
}
```

8. Teste a função introduzindo as seguintes instruções no REPL:

```
>>> pedeNumero("Introduza um valor entre 1 e 10.", 1, 10)
>>> pedeNumero("Introduza um valor entre 3 e 4.", 3, 4)
>>> pedeNumero("Indique o montante da transferência: ", 0, 1000)
```

9. Uma vez que testar se um número está contido entre dois valores é, também, uma necessidade comum, vamos definir a função `entre` que devolve ***true*** se um número pertence ao intervalo definido por outros dois números:

```
function entre(num, limiteInf, limiteSup) {
  return num >= limiteInf && num <= limiteSup;
}
```

10. Teste a função `entre`:

```
>>> [entre(3, 1, 5), entre(5, 1, 3)]
[ true, false ]
>>> alert(entre(30, 1, 5))
```

11. Agora podemos utilizar a função `entre` na definição da função `pedeNumero`:

```
function pedeNumero(texto, limiteInf, limiteSup) {
    let numero = 0;
    while (true) {
        numero = parseFloat(prompt(texto));
        if (entre(numero, limiteInf, limiteSup)) {
            break;
        }
        alert("Valor inválido!");
    }
    return numero;
}
```

12. Pegando num exemplo anterior, a seguinte função calcula o preço com IVA de um montante. Para além do montante, a função necessita de saber o tipo de produto para saber a taxa de IVA a aplicar. Se o tipo de produto for "alimentação" o IVA a aplicar deverá ser 6%, se for "higiene" deverá ser 13%, caso contrário deverá ser 23%.

```
function comIVA (montante, tipoProduto) {
     let IVA;
     if (tipoProduto === "A") {       // "A"limentação
         IVA = 6;
     }
     else if ( tipoProduto === "H") { // "H"igiene
         IVA = 12;
     }
     else {                            // todos os outros
             IVA = 23;
     }
     return montante*(1+IVA/100);
}
```

13. Teste a função com valores apropriados. Por exemplo:

```
>>> comIVA(100, "A")
106
>>> comIVA(200, "H")
224.00...03
>>> comIVA(200, "X")
246
>>> comIVA(100)
123
```

*Em JavaScript todos os parâmetros são **opcionais**, isto é, não é obrigatório passar um valor para um determinado parâmetro durante a invocação. O parâmetro omitido durante a invocação fica com o valor **undefined**.*
*Se invocarmos a função `comIVA` sem dar uma valor ao parâmetro `tipoProduto` , este fica com **undefined** que é diferente de `"A"` e de `"H"`. Ou seja, o **else** é seleccionado e o `IVA` toma o valor de 23.*

PARTE II – NÚMERO VARIÁVEL DE ARGUMENTOS, "REST PARAMETER" E VALORES P/ OMISSÃO

14. Uma função pode aceitar um número variável de argumentos. Vamos definir a função `insere` que, dado um array ou uma string, e uma posição, insere os restantes argumentos nessa posição.

No caso de receber um array, esta função é destrutiva, ie, altera o array; no caso de receber uma string, devolve uma cópia com os valores inseridos no local indicado.

```
function insere(coleccao, start=0, ...novosItens) {
    let novaColeccao = coleccao;
    if (Array.isArray(coleccao)) {
        coleccao.splice(start, 0, ...novosItens);
```

```
    }
    else if (typeof novaColeccao === 'string') {
        novaColeccao = coleccao.slice(0, start)
                     + novosItens.join('')
                     + coleccao.slice(start);
    }
    return novaColeccao;
}
```

*Em JavaScript podemos definir **parâmetros com um argumento (ou valor) por omissão**. Quando um parâmetro tem um valor por omissão, então não é obrigatório passar um argumento para esse parâmetro. Deste modo, um parâmetro com valor por omissão é também designado de **parâmetro opcional**. Parâmetros com argumento por omissão devem ocorrer depois dos parâmetros obrigatórios na lista de parâmetros da função.*
*Ao ser precedido com ... o parâmetro `novosItens` vai "apanhar" todos os argumentos posicionais da função `insere`. Este parâmetro, designado de **rest parameter** (restantes), é um **array** onde `novosItens[0]` corresponde ao primeiro argumento após `coleccao`, `novosItens[1]` ao segundo, etc.*

15. Pode agora testar esta função com:

```
>>> let nums = [7, 12, 2, 5, 8, 4, 10]
>>> let codigo = "XYABC";
>>> insere(nums, 3, 78, 44)
[ 7, 12, 2, 78, 44, 5, 8, 4, 10 ]
>>> nums
[ 7, 12, 2, 78, 44, 5, 8, 4, 10 ]
>>> insere(codigo, 2, "--", 44, "M")
"XY--44MABC"
```

PARTE III – FUNÇÕES INTERNAS E VARIÁVEIS LOCAIS

16. Em JavaScript podemos definir funções dentro de funções. A título de exemplo, vamos definir uma função para detectar se uma string é um palíndromo (texto que se lê de igual forma da esquerda para a direita ou da direita para a esquerda). Opcionalmente, esta função ignora todos os caracteres que não sejam letras e números (eg, símbolos de pontuação) e a capitalização das letras.

```
function ePalindromo(txt, apenasAlfaNum=false, ignoraCap=false) {
    function filtraAlfaNum(txt) {
        return Array.from(txt).filter((car) => /^[0-9a-zA-Z]$/.test(car));
    }
    function ePal(seqCars) {
        if (seqCars.length <= 1) {
            return true;
        }
        return seqCars[0] === seqCars[seqCars.length-1]
            && ePal(seqCars.slice(1, -1));
    }

    txt = ignoraCap && txt.toLowerCase() || txt;
    return ePal(apenasAlfaNum ? filtraAlfaNum(txt) : txt);
}
```

A função `ePalindromo` utiliza duas funções auxiliares `filtraAlfanum`, que faz o que nome indica (ou seja, filtra os caracteres alfanuméricos), e `ePal` que na verdade contém o algoritmo de reconhecimento do palíndromo. Ambas as funções poderiam ter sido definidas fora de `ePalindromo`, só que dado que não têm utilidade fora dela (bem, talvez `filtraAlfanum` tenha utilidade noutros contextos), ***foi decidido "arrumá-las" dentro da própria função ePalindromo****.*
A função `filtraAlfanum` devolve uma sequência de caracteres apenas com letras e números. Como uma ***função interna tem acesso às variáveis definidas na função externa****, na verdade, `filtraAlfanum` não necessita de parâmetros. Decidimos definir o parâmetro txt porque consideramos ser uma boa prática de programação uma função não depender de parâmetros externos (excepção feita a closures, como veremos a seguir; aí não há hipótese).*
De notar que a função `ePal` invoca-se a si própria na sua própria definição. É o que se designa por ***função recursiva****. De facto, um palíndromo é um texto onde o primeiro e último caracteres são iguais e onde o restante texto é ele próprio um palíndromo. Ou seja, a noção de palíndromo aparece na sua própria definição (o definido participa na sua definição).*
*A função `ePal` traduz directamente esta noção. Esta possibilidade, de uma função invocar-se a si própria, permite exprimir determinados algoritmos de forma mais simples e declarativa. Todavia, na maioria das linguagens, e em JavaScript em particular, funções recursivas tendem a necessitar mais memória temporária (*stack memory*). Devem ser utilizadas com cautela.*

17. Teste agora a função com:

```
>>> [ePalindromo("ABCBA"), ePalindromo("ABCBAA"), ePalindromo("ABC BA")]
[ true, false, false ]
>>> [ePalindromo("ABC78CBA"), ePalindromo("###ABCC BA"), ePalindromo("ABa!")]
[ false, false, false ]
>>> [ePalindromo("ABC78CBA", true), ePalindromo("###ABCC BA", true)]
[ false, true ]
>>> ePalindromo("ABa!", true, true)
true
```

18. Como é complicado memorizar a ordem dos parâmetros, especialmente quando se lê o código algum tempo depois deste ter sido desenvolvido, vamos utilizar um objecto de especificação com os parâmetros:

```
function ePalindromo(txt, spec={apenasAlfaNum: false, ignoraCap: false}) {
    let {apenasAlfaNum, ignoraCap} = spec
    txt = ignoraCap && txt.toLowerCase() || txt;
    return ePal(apenasAlfaNum ? filtraAlfaNum(txt) : txt);

    function filtraAlfaNum(txt) {
        return Array.from(txt).filter((car) => /^[0-9a-zA-Z]$/.test(car));
    }
    function ePal(seqCars) {
        if (seqCars.length <= 1) {
            return true;
        }
```

```
            return seqCars[0] === seqCars[seqCars.length-1]
                && ePal(seqCars.slice(1, -1));
        }
    }
```

*Um **objecto de especificação** é um objecto literal cujas propriedades são parte ou a totalidade do parâmetros de uma função. As vantagens de um objecto de especificação são: 1) Podemos dar nome aos argumentos, o que torna a invocação da função mais legível; 2) A ordem é irrelevante, ou seja, ao contrário do que sucede com uma lista de parâmetros "normal", não temos que passar os argumentos por nenhuma ordem; e 3) São opcionais, isto é, podemos ignorar as propriedades que entendermos se a dada altura, durante a vida da função, considerarmos que determinados parâmetros já não são necessários.*
*Uma outra alteração nesta versão foi o facto de termos movido o corpo da função externa para cima, ao passo que as definições das funções internas estão em baixo. Em JavaScript, à semelhança do que sucede com variáveis definidas com **`var`** em qualquer parte da função, e de variáveis/constantes definidas com **`let`**/**`const`** em qualquer local do bloco de instruções de uma função, todas as funções são automáticamente colocadas no topo da função. Este mecanismo é designado de **hoisting** ("içar" de acordo com o Google Translate).*

19. Uma função externa pode devolver uma função interna podendo esta "aprisionar" o estado de uma variável definida na função externa. Por exemplo, vamos definir uma função que recebe um valor e devolve um somador de N unidades (ie, devolve uma outra função que recebe um argumento, soma-lhe N e devolve o resultado).

```
function somador(quantidade) {
  function soma (numero) {
    return numero + quantidade;
  }
  return soma;
}
```

*O parâmetro `quantidade` é também uma **variável local** da função `somador`. No início dissemos que uma variável local "desaparece" assim que a função terminar. Na verdade, não é bem assim...*
Uma variável local só desaparece quando todas as referências para essa variável desaparecerem. Quando uma função devolve uma função interna (neste caso, a função `somador` devolve a função `soma`) e esta função referencia uma variável local (neste caso, a função `soma` referencia a variável `quantidade`), então a variável local `quantidade` não é eliminada após execução de `somador`. Ou seja, todo o âmbito que existia "à data" de devolução da função interna mantém-se. À frente, a propósito de closures, voltamos a este tópico.

20. Em JavaScript seria mais comum devolver uma função anónima utilizando para tal uma expressão function:

```
function somador (quantidade) {
   return function(numero) {
      return numero + quantidade;
   }
}
```

*A palavra-reservada **`function`** permite também definir uma expressão "função". Uma expressão "função" produz .... uhmm ... uma função. O nome desta função é opcional. Ou seja, através de uma expressão "função" podemos definir **funções anónimas,** ou **lambdas**, termo utilizado em programação funcional para designar este tipo de funções. Por norma, uma lambda aparece à direita de um = ou de um **`return`**, ou como argumento de uma outra função.*
*A função `somador` também se designa por **higher order function** (ver explicação em baixo).*

21. Agora podemos utilizar o `somador` da seguinte forma:

```
>>> let somaDez = somador(10)
>>> let somaCinco = somador(5)
>>> [somaDez(100), somaCinco(100)]
[ 110, 105 ]
```

*Quer a função `soma`, definida e devolvida na primeira versão de `somador`, quer a `lambda`, devolvida na segunda versão, são designadas de **fecho** (**closure**). Uma closure é uma função que envolve e retém o âmbito (scope) da função envolvente. De facto, o âmbito de `soma` abrange todas as variáveis definidas dentro desta função assim como todas as variáveis definidas em `somador`.*
A função `somador`, para todos os efeitos, é uma função geradora de código. Ela gera uma outra função que soma n ao argumento.

22. Vejamos um outro exemplo com lambdas. Podemos definir uma função que permite verificar se todos os caracteres de uma string verificam um determinado critério:

```
function todos(str, criterio) {
    for (let ch of str) {
        if (!criterio(ch)) {
            return false;
        }
    }
    return true;
}
```

De modo a tornar este algoritmo genérico, a função `todos` define o parâmetro funcional `criterio`. Este parâmetro deve receber uma função como o critério de verificação.
*Uma função com **parâmetros funcionais**, ie, com parâmetros cujos argumentos devem ser outras funções, ou que devolvem funções através de **`return`**, é designada de função de **ordem mais alta** (tradução livre de **higher order function**).*

23. A função todos pode ser invocada das seguintes formas:

```
todos("ABCDEF", function(ch) {
    return ch >= 'A' && ch <= 'Z';
});                                     // devolve true
todos("ABCDEF12", function(ch) {
    return ch >= 'A' && ch <= 'Z';
});                                    // devolve false
todos("ABCDEF12", function(ch) {
    return ch >= 'A' && ch <= 'Z' || ch >= '0' && ch <= '9';
});                                     // devolve true
```

24. Com a versão ES6 da linguagem JavaScript, apareceu uma notação mais conveniente para definir lambdas que apenas devolvem o resultado de uma expressão. Esta notação é representada pelo símbolo => ("*fat arrow*" ou "seta gorda"), dispensa as chavetas e a palavra-reservada ***`return`***.

```
todos("ABCDEF", ch => ch >= 'A' && ch <= 'Z');
todos("ABCDEF01", ch => ch >= 'A' && ch <= 'Z');
```

```
todos("ABCDEF01", ch => ch >= 'A' && ch <= 'Z' || ch >= '0' && ch <= '9');
```

25. Outro exemplo: dada uma lista de valores pretendemos filtrar todos os valores pares para uma outra lista. Primeiro começamos por definir a função `ePar` que devolve `true` se o seu argumento for um número par:

```
function ePar(num) {
    return num % 2 === 0;
}
```

26. Para efeitos de teste, defina a lista de números `nums`:

```
>>> let nums = [1, 7, 2, 8, 5, 171, 90, 17]
```

27. Agora vamos recorrer à função built-in `filter` passando-lhe a função `ePar` como primeiro argumento:

```
>>> nums.filter(ePar)
```

28. As funções anónimas são especialmente interessantes para serem utilizadas como argumentos de outras funções. De seguida vemos o exemplo anterior mas com uma lambda a substituir a função `ePar`:

```
>>> nums.filter(x => x % 2 === 0)
```

29. Os métodos `filter`, `map`, `forEach`, `reduce`, `some` e `every` são autênticos baluartes da programação funcional. Consulte a sua documentação na MDN e teste (resultados omitidos):

```
>>> nums.filter(x => x * 2)
>>> nums.forEach(x => console.log(x))
>>> nums.forEach((x, i) => console.log(`${i}: ${x}`))
>>> nums.reduce((acumulador, valorActual) => acumulador + valorActual, 0)
```

30. E agora um outro exemplo mais complexo. Na sequência de um exemplo anterior, podemos definir uma despesa através do tipo de produto a que a despesa se refere, do montante (sem IVA) da despesa e de uma pequena descrição da despesa (algumas chaveta foram omitidas por brevidade).

```
function despesa (descricao, tipoProduto, montante) {
  var montanteComIVA = comIVA(montante, tipoProduto);

  function apresenta () {
    print("         Descricao: " + descricao);
    print("   Tipo de produto: " + tipoProduto);
```

```
    print("   Montante S/ IVA: " + montante);
    print("   Montante C/ IVA: " + montanteComIVA);
  }

  function precoFinal () {
    return montanteComIVA;
  }

  function operacao(codigoOperacao) {
    if (codigoOperacao == "apresenta") apresenta();
    else if(codigoOperacao == "precoFinal") return precoFinal();
    else alert("Operação inválida");
  }

  return operacao;
}
```

31. Uma vez criada a despesa, podemos "consultar" determinados "aspectos" da despesa.

```
let despesa1 = despesa("Café", "A", 0.60);
let despesa2 = despesa("Sabonente", "H", 2.50);

despesa1("apresenta");
despesa2("apresenta");
print("Preço final da despesa 1" + despesa1("precoFinal"));
```

EXERCÍCIOS DE REVISÃO

1. Defina *função*, *parâmetro*, *argumento*, *argumento com nome* e *parâmetro opcional*.

2. Suponha que pretende utilizar a função `Math.pow` mas pretende invocá-la com o nome elevado_a. Como é que poderia proceder para atingir esse objectivo?

3. O que é exibido pelas seguinte instruções?

`let y = 10;`
`function f(x) {`
`  console.log(x + y);`
`}`
`f(3);`
`function g() {`
`  let y = 20;`
`  f(3);`
`}`
`g();`	

```
let y = 5;
function func1() {
    function f(x) {
        console.log(x + y);
    }
    f(10);
}
function func2() {
    function f(x) {
        console.log(x + y);
    }
    let y = 50;   // A
    f(10);        // B
}
func1();
func2();
```

NOTA: *O que acontece se:*

1 - Em A, definirmos `y` *com* **`var`***?*

2 - Trocarmos a ordem das instruções assinaladas com A e B?

3 - O mesmo que 2, mas utilizarmos **`var`** *em vez de* **`let`** *para definir* `y`*?*

```
let y = 10;
function func3() {
    function f1() {
        let y = 1;
    }
    function f2() {
        y = 2;
    }
    function f3() {
        this.y = 3;
    }
    let y = 0;
    f1();
    console.log(y);
    f2();
    console.log(y);
    f3();
    console.log(y);
}
func3();
console.log(y);
```

NOTA: *Assuma que <u>não</u> está a ser utilizado* `'strict mode'` *.*

O que acontece se não utilizarmos **`let`** *para definir a variável* `y` *global?*

4. Dada o array `vals = [2, 0, 1, 3, 2, 0, 1, 5]` e a string `txt = 'Dinamarca'`, com que valores ficam as variáveis nas atribuições seguintes:

4.1
```
a = vals.filter(x => x > 2)
```

4.2
```
b = vals.map(x => x > 2)
```

4.3
```
c1 = [...txt].filter(function(ch) { return 'aeiou'.includes(ch);})
```

4.4
```
c2 = new Set([...txt].filter(function(ch) { return 'aeiou'.includes(ch);}))
```

4.5
```
d = [...txt].map(ch => String.fromCharCode(ch.charCodeAt()+1))
             .filter(ch => 'aeiouAEIOU'.includes(ch))
             .join('/')
```

4.6
```
e = vals.every(x => (x >= 0 && x <= 2))
```

4.7
```
f = vals.some(x => (x >= 0 && x <= 2))
```

4.8
```
h = vals.reduce((x, y) => x > y ? x : y, -Infinity)
```

EXERCÍCIOS DE PROGRAMAÇÃO

***Instruções**: Onde apropriado, para alguns dos problemas seguintes pode ser conveniente desenvolver uma pequena página HTML com os elementos necessários para que o utilizador introduza a informação necessária e visualize os resultados pretendidos. Para esses problemas, ignore preocupações estilísticas e, em particular, não se preocupe com cores, layouts e tipos de letra. Concentre-se na implementação correcta do código JavaScript.*

5. Desenvolva a função `confirma` que solicita uma confirmação ao utilizador. Enquanto o utilizador não introduzir `'sim'`, `'s'`, `'não'`, `'nao'` ou `'n'`, a função repete a mensagem de confirmação. Ao fim de um número de tentativas (quatro, por omissão) a função desiste e assume que o utilizador não confirmou. A função devolve **`true`**, se o utilizador confirmar, e **`false`**, caso contrário. Além da mensagem de confirmação e do número de tentativas, a função recebe uma mensagem de erro que deverá ser exibida quando o utilizador não introduzir o pedido. Este parâmetro deve possuir o valor por omissão: `"Introduza (S)im ou (N)ao"`.

6. Desenvolva a função `inverte` que recebe uma sequência de elementos e devolve uma lista com os elementos por ordem inversa. Não utilize `reverse` ou qualquer outra função já desenvolvida para o efeito.

7. Desenvolva a função `invertePalavras` que recebe uma string e devolve uma nova string com todas as palavras por ordem inversa. Assuma que as palavras estão separadas apenas por um espaço.

8. O método `Array.prototype.filter(p)` filtra utiliza a função `p` para filtrar elementos. Se `p` devolver ***true***, então esse elemento fará parte do array com todos os elementos seleccionados. Desenvolva os seguintes filtros prontos a utilizar com `filter`:

8.1 `inBetween(a, b)` – cria filtro para seleccionar valores entre `a` e `b` (inclusive).

8.2 `inCollection(...values)` - cria filtro para seleccionar valores que pertençam ao conjunto de valores em `values`.

8.3 `like(pattern)` - cria filtro que selecciona todas as strings que verifiquem o padrão dado pela expressão regular `pattern`.

9. Desenvolva uma variação de `slice` que aceita:

- Uma sequência de elementos que possa ser iterada por um ciclo ***let-of*** e que possua uma propriedade `length` e que possa ser indexada com `[ ]`

- `start` e `end` com a semântica de `Array.prototype.slice`

- `step` que indica um incremento para levar `start` até `end`; se `step` for positivo e `start >= end` , ou see `step` for negativo e `start <= end`, é devolvida uma sequência vazia; se `step === 0` é devolvido ***undefined*** ou lançada uma excepção.

A função devolve os elementos numa string, se a sequência original for uma string, num array, para qualquer outro tipo de dados da sequência original. Por omissão o valor de `step` é `1`. `start` e `end` têm os valores por omissão que têm em `Array.prototype.slice`.

10. Desenvolva `cslice` que é idêntica à anterior mas que devolve uma *closure* que permite aceder ao próximo elemento da fatia.

11. Desenvolva a função `reversed` que devolve uma closure para obter os elementos por ordem inversa da sequência passada como argumento.

12. Desenvolva `dateRange`, uma variação da função `range` definida em lodash mas para datas. Os parâmetros `start`, `end` e são datas; `step` deve ser um inteiro com dias. A função devolve uma *closure* que permite aceder aos dias entre `start` e `end`.

13. Desenvolva a função `substitui` que recebe três strings e devolve uma nova string onde todas as ocorrências da segunda string na primeira são substituídas pela terceira string.

```
>>> substitui('aXYZbXYZc', 'XYZ', '1')  -> 'a1b1c'
>>> substitui('aXYZbXYZc', 'XYZ', '')   -> 'abc'
>>> substitui('abab', 'b', 'XYZ')       -> 'aXYZaXYZ'
```

Desenvolva esta versão sem utilizar funções/métodos para trabalhar com strings (como `String.prototype.replace/split`, etc.). Pode utilizar arrays, `Array.prototype.join` e `String.prototype.indexOf/search`.

14. Desenvolva a função `randLetter` que selecciona aleatoriamente uma letra e devolve-a. Esta função deverá possuir um parâmetro opcional `type` que indica a `randLetter` que tipo de letra deve seleccionar: se for `'U'` escolhe uma letra maiúscula; se for `'L'`, escolhe uma minúscula; para qualquer outro valor de `type`, `randLetter` escolhe uma letra tendo em conta as duas categorias de valores. Por omissão `type` tem o valor `''` (string vazia) o que indica que pode devolver uma letra maiúscula ou minúscula.

15. Desenvolva a função `randLetters` que devolve uma string com letras seleccionadas aleatoriamente. Possui dois parâmetros: `n`, que indica quantas letras terá a string, e `type` (ver `randLetter`). Por omissão, `n` tem o valor `2` e `type` tem o valor `''` (string vazia).

REFERÊNCIAS:

[1]: *Marijn Haverbeke, "Eloquent JavaScript, 3rd Ed.", 2018, No Starch Press*, https://eloquentjavascript.net/index.html
[2]: JavaScript: MDN (Mozilla Developer Network): https://developer.mozilla.org/en-US/docs/Web/JavaScript
[3]: Gramar and Types @ MDN, https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Grammar_and_Types