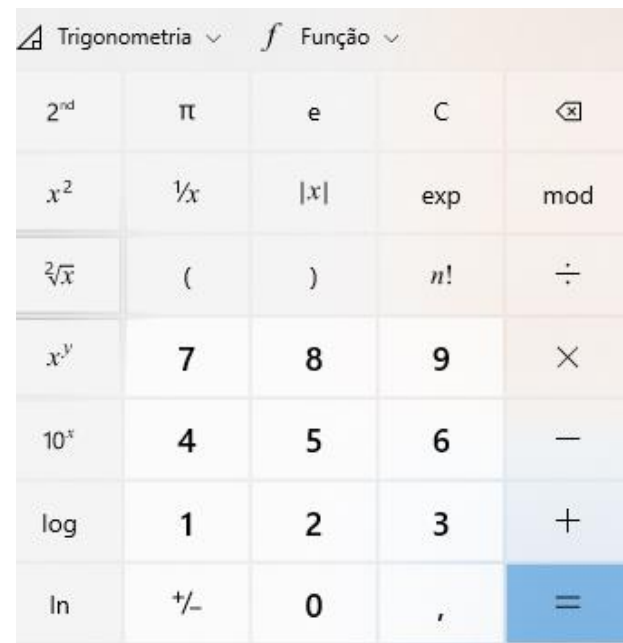
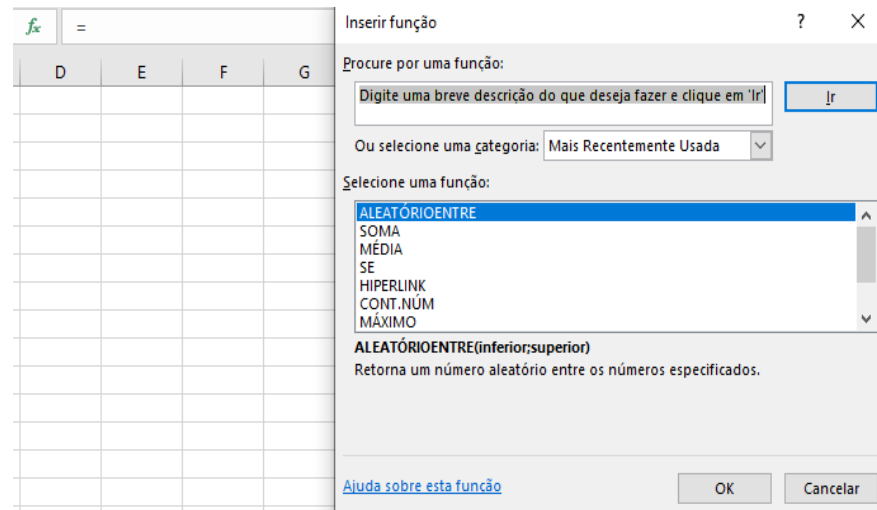


Programação de Computadores

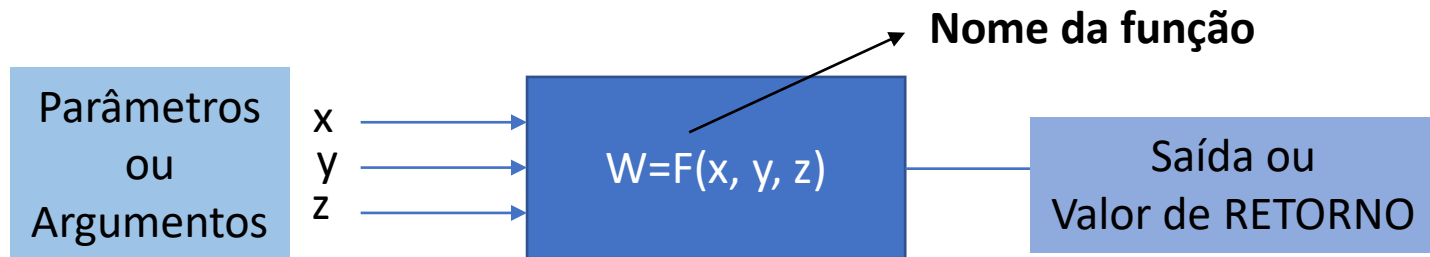
Prof. Dr. Josenalde Barbosa de Oliveira

josenalde.oliveira@eaj.ufrn.br



Modularização de Código - funções

- Ao programar, utilizamos funções de sistema ou de terceiros, naturalmente
- Reutilização de código (não reinventar a roda)
- Organização de código (mais limpo, inteligível)



- Em JS não especificamos o tipo de cada parâmetro nem o tipo de retorno da função, ou seja, o tipo de W no exemplo acima. É inferido pelo interpretador JS

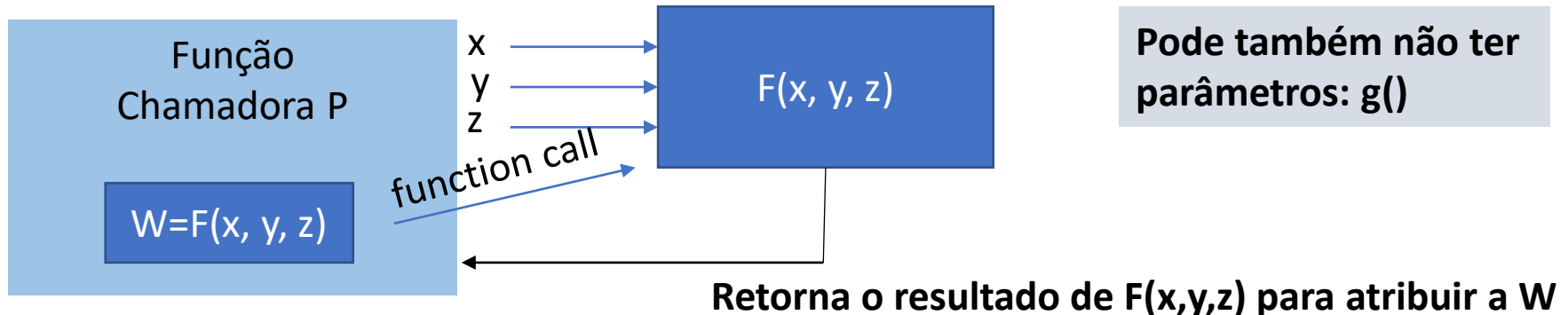
```
function F(x,y,z) {  
  //corpo da função  
  return <valor>;  
}
```

```
let W = F(1,5,10);  
  
// x <- 1, y <- 5, z <- 10
```

Com funções nomeadas,
pode usar antes de definir

Modularização de Código - funções

- Uma função é definida (protótipo, assinatura), implementada (corpo), e é CHAMADA (function call) por outra função (ou código principal)
- Seu valor de retorno pode ser atribuído a uma variável ou utilizado onde se aguarda um valor explícito
- Contudo, uma função pode ser do tipo **void** (também chamado procedimento), **que executa algo (faz algo)**, mas não retorna valores à função que chamou-a para serem atribuídos



Modularização de Código - funções

```
function soma(a,b) {  
    console.log(soma.arguments)  
    return a+b;  
}  
let x = soma(5,10);  
console.log(x)
```

JS admite expressões de funções, as quais não precisam ter nome e podem ser atribuídas à variáveis. Exemplo:

```
let digaOi = function(a) {  
    alert('Oi ' + a + '!');  
}  
  
digaOi('pedro');  
  
digaOi = function() { //redefine  
    alert('Oi!');  
}  
  
digaOi(); //sem parâmetros
```

```
function soma(a,b) {  
    return a+b;  
}  
let x = soma(5); //faltando b  
console.log(x); //undefined
```

```
function soma(a,b=2) { //valor default  
    return a+b;  
}  
let x = soma(5); //irá assumir b=2  
console.log(x); //7
```

```
function soma(a,b,c) { //ignora c  
    return a+b;  
}  
let x = soma(5,2);  
console.log(x); //7
```

Modularização de Código – funções - escopo

```
let x = 10; //escopo (visibilidade) global
function f(a) {
    let y = 2; //escopo local à f()
    return x + a + y; //acessa x em f() pois é global
}
console.log(f(2)); //14
alert(x); //10
alert(y); //ERRO, não é acessível aqui ReferenceError: y is not defined
```

```
function f(v) { //vetor como parâmetro
    for (let i=0; i<v.length;i++) {
        v[i] *= 2;
    }
    return v;
}
```

```
let a = [1, 4, 6];
f(a).toString(); //v recebe cópia de a
```

Modularização de Código – funções anônimas

Funções sem nome – são úteis em várias situações

Exemplo 1: chamadas temporizadas

```
setTimeout(function() {  
    console.log('função chamada após 2s com timeout');  
}, 2000); //tempo em milissegundos (atraso na execução)
```

```
setInterval(function() {  
    console.log('função chamada a cada 1s com interval');  
}, 1000);
```

```
setInterval(function(p1,p2) {  
    console.log('função chamada a cada 1s com interval');  
    console.log(p1+p2);  
}, 1000,10,15);
```

Modularização de Código - funções

Funções anônimas executadas imediatamente após definição

//expressão de função anônima:

```
(function() {  
    console.log('executado imediatamente')  
})();
```


```
let aluno = {  
    nome : 'josenalde oliveira',  
    turma : 'if21'  
};
```

```
(function() {  
    console.log(`${aluno.nome}` + ' da turma ' + `${aluno.turma}`);  
})(aluno) //passando parâmetro
```

As funções anônimas também podem ser representadas em JS com o símbolo => (seta)

Modularização de Código - funções

```
let show = function() {  
  alert('anonima 1');  
};  
  
show();  
  
let show2 = () => alert('anomina 2');  
  
show2();
```



```
let soma = function(a,b) {  
  return a+b;  
}  
  
alert(soma(2,4));  
  
let soma2 = (a,b) => a+b; //sem return  
  
alert(soma2(5,3));
```


Programação dinâmica

```
function fatorial(n) {  
  if (n == 1) return 1;  
  let f = 1;  
  for (let i = n; i > 1; i--) f *= i;  
  return f;  
}
```

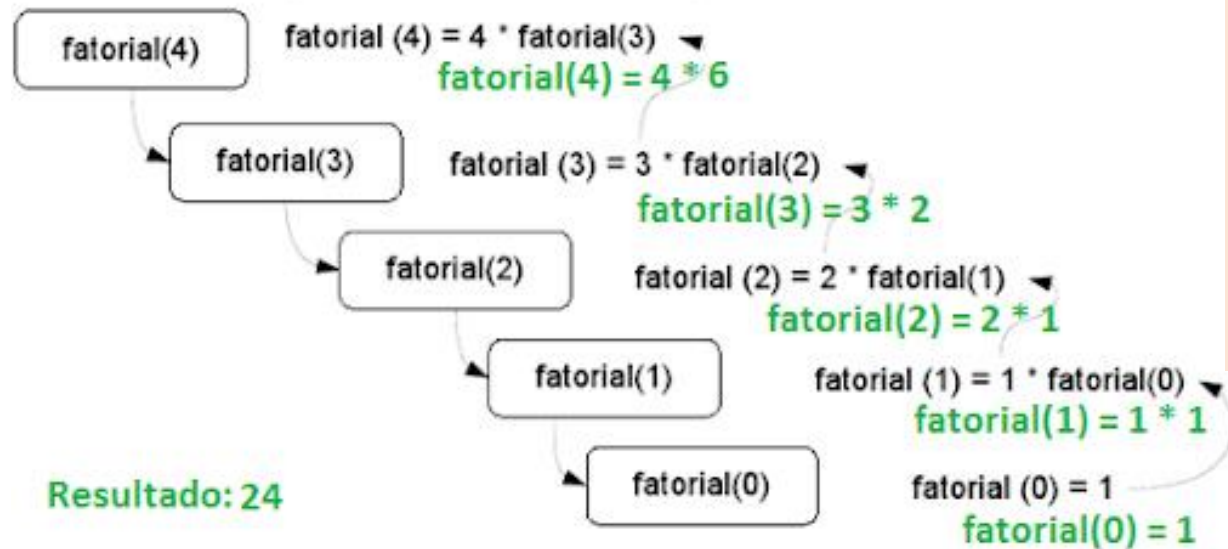
```
let x = 5, y = 10;  
alert(fatorial(x));  
fibo(x);
```

```
function fibo(t) { // imprime até o t. termo da série de Fibonacci  
  let v = [];  
  v[0] = 0;  
  v[1] = 1;  
  if (t > 2) { // assume t1=0, t2=1 FIXOS  
    for (let k = 2; k < t; k++) {  
      v[k] = v[k-1] + v[k-2]; // atual depende dos anteriores  
    }  
  }  
  alert(v.toString());  
}
```

Recursividade

- Chamadas sucessivas a mesma função (auto chamamento)
- Importante definir critério de parada, senão loop infinito
- Emula um laço de repetição onde a variável de controle avança ou recua até não atender determinado critério
- Elegante, porém eficiência deve ser avaliada

*Lembrem-se que a leitura é de baixo para cima!



Exemplo: FATORIAL

$$5! = 5 * 4!$$

$$= 5 * 4 * 3!$$

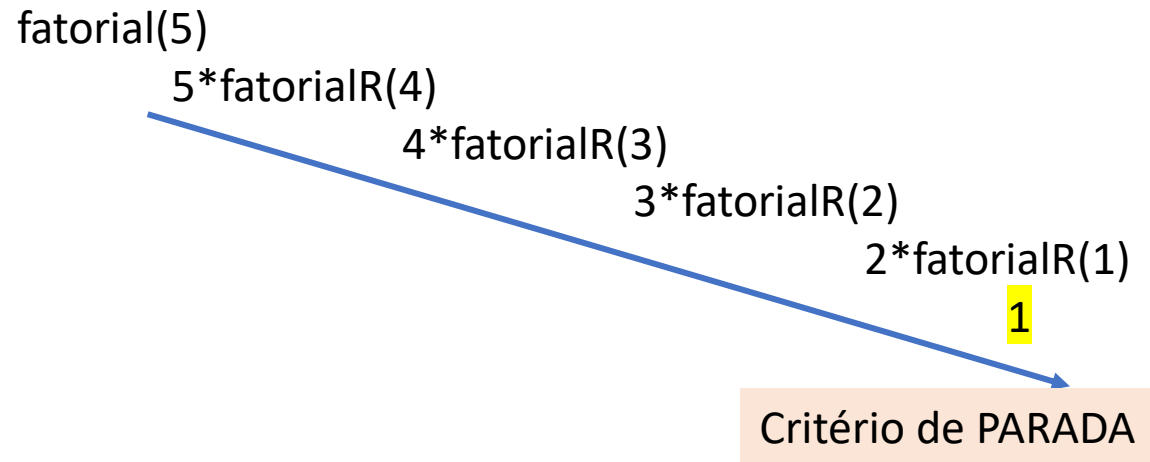
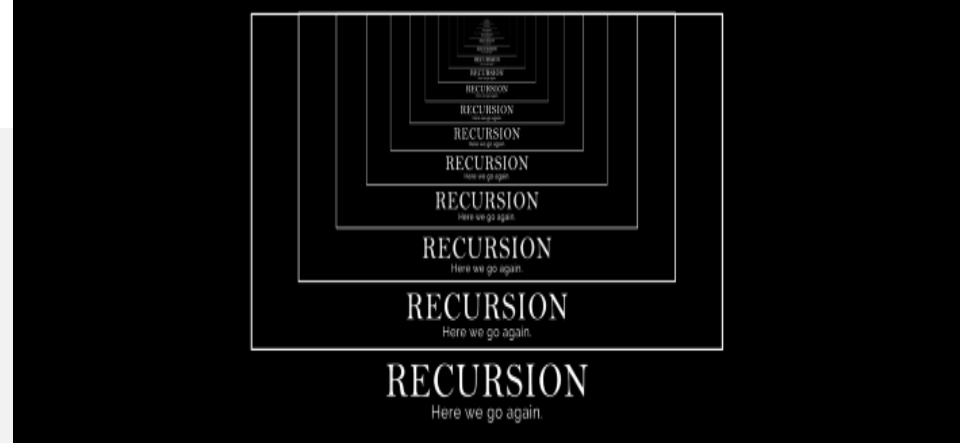
$$= 5 * 4 * 3 * 2!$$

$$= 5 * 4 * 3 * 2 * 1!$$

Recursividade

```
function fatR(n) {  
  if (n == 1) return 1;  
  return (n*fatR(n-1));  
}
```

```
console.log(fatR(5));
```

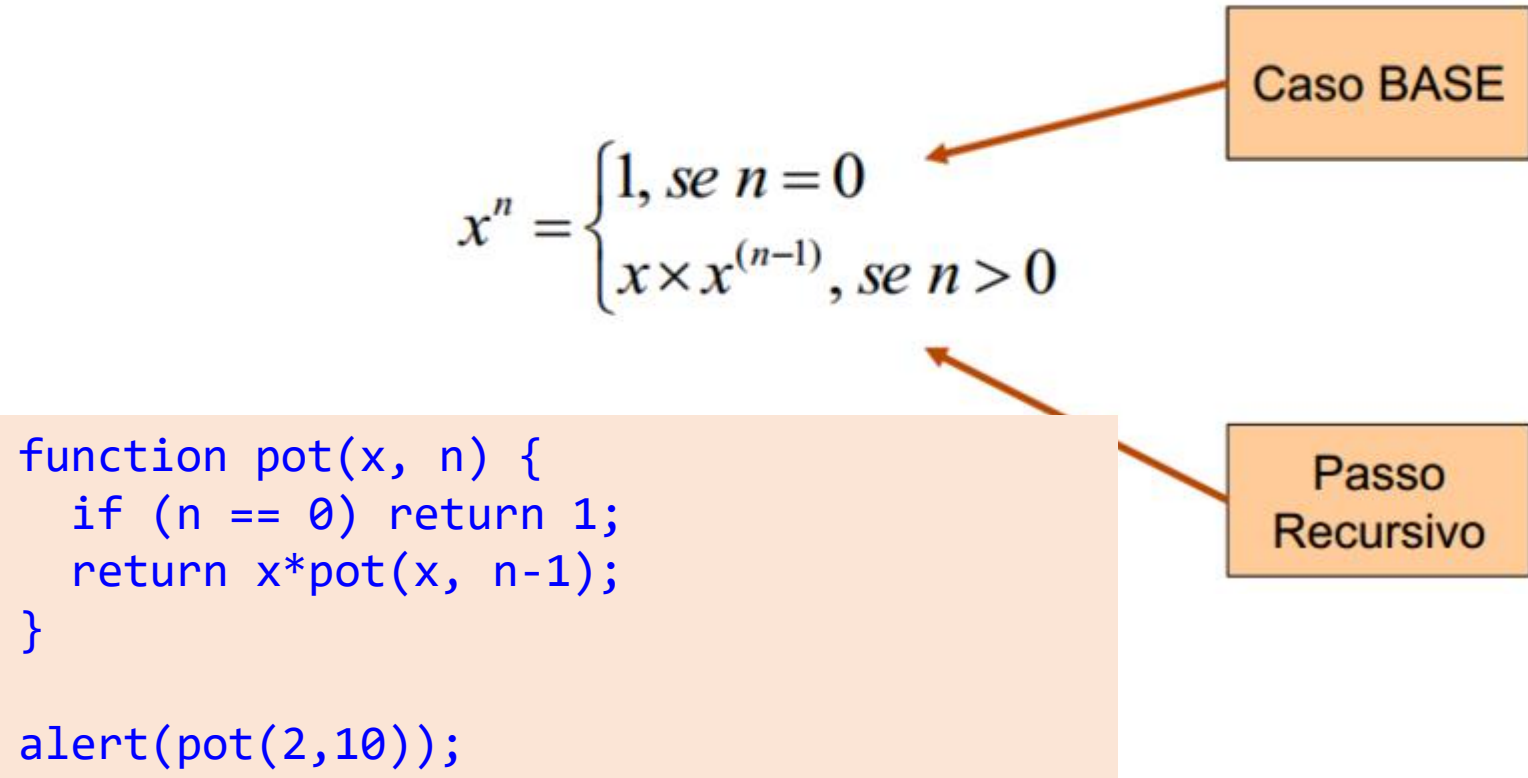


Recursividade

Exercício: forneça a definição recursiva para a operação de potenciação

$$x^n = \begin{cases} 1, & \text{se } n = 0 \\ x \times x^{(n-1)}, & \text{se } n > 0 \end{cases}$$

Caso BASE



```
function pot(x, n) {  
  if (n == 0) return 1;  
  return x*pot(x, n-1);  
}
```

```
alert(pot(2,10));
```

Passo
Recursivo