UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

INF01121 - MODELOS DE LINGUAGEM DE PROGRAMAÇÃO - 2015/1

TRABALHO FINAL

ALEXANDRE LEUCK - 220493
FELIPE LIMA - 192363
GIOVANI BOMBARDIERI - 205651

SUMÁRIO

1 APRESENTAÇÃO DA LINGUAGEM

JavaScript (ou ECMAScript) surgiu em 1995 dentro da empresa Netscape e acabou se solidificando na metade dos anos 2000 como uma das linguagens mais importantes dentro do desenvolvimento web. Além de seu predominante uso em páginas da web - interagindo com o usuário e controlando o browser - JavaScript também é bastante utilizada *server-side* através do ambiente de execução *Node.js*[5].

Algumas das características atribuídas à linguagem incluem:

- *Imperativa e estruturada*: Por ter muito da sua sintaxe derivada da linguagem C, adere a muitas características da mesma.
- *Funcional*: Em JavaScript, funções são elementos de primeira classe, e assim podem ter seus próprios métodos e propriedades. Isso vem do fato de grande parte da parte semântica da linguagem ter sido derivada de Scheme.
- *Tipagem dinâmica*: Seguindo o padrão da maioria das linguagens de script, os tipos em JavaScript estão associados a valores e não vinculados às variáveis.
- *Objetos dinâmicos*: Objetos em JavaScript são arrays associativos (dicionários, ou *hash maps*) que podem ser modificados no decorrer na execução.
- *Protótipos*: Para implementar o conceito de herança, ao invés de usar classes como a grande parte de linguagens orientadas a objetos faz, em JavaScript objetos herdam de outros objetos (herança prototipal)[7].

Ainda, a linguagem JavaScript traz, entre outras, as seguintes funcionalidades:

- Uma função *eval* para compilar e executar segmentos de código passados como strings em tempo de execução.
- Unificação de objeto e dicionário.
- Tratamento de funções como objetos de primeira classe.
- Expressões regulares[5].
- Ortogonalidade de operadores: É possivel combinar a maioria dos tipos com a maioria dos operadores, mesmo em situações onde não há sentido claro, como divisão de duas funções, que resulta em NaN.
- Coletor de lixo automático.

Como já mencionado, JavaScript é usada, na grande maioria das vezes, em scripts que serão executados pelo browser para trazer algum nível de interação com o usuário. O código é executado *client-side* junto ao HTML da página, no que é chamado *Dynamic HTML*. Exemplos de possíveis aplicações nesse contexto:

- Carregar elementos da página ou enviar informações para o servidor sem ter que recarregar a página.
- Animação de elementos da página.
- Conteúdo interativo, como áudio e vídeo.
- Validar formato campos de formulários antes de contatar o servidor.
- Processamento de imagens (Canvas 2D).
- Computação gráfica (WebGL)[5].

Existem alguns usos notáveis de JS fora do ambiente exclusivo de páginas web, como por exemplo:

- No outro lado das comunicações: *web servers*.
- Como uma linguagem de script incorporada, pode ser encontrada em extensões do Google Chrome, no visualizador de PDFs Acrobat Reader, no LibreOffice, entre muitos outros.
- Engine de script.
- Plataforma de aplicativos.
- Microcontroladores[5].

Por ser uma das linguagens mais populares em um dos campos de desenvolvimento mais populares, JavaScript está sempre sendo avaliada constantemente. Assim como toda outra linguagem de programação, ela possui qualidades e defeitos, os quais determinam em quais campos ela é mais recomendada para uso. Alguns dos benefícios a serem levados em conta nessa avaliação, principalmente comparando-a com outras linguagens "concorrentes" da mesma área - web - são:

- *Velocidade*: por funcionar *client-side*, a falta de necessidade de comunições extras com o servidor torna a execução significantemente mais rápida, o que é um dos fatores principais se tratando de ambientes web. Isso também acaba trazendo outro benefício, a diminuição de carga no servidor.
- *Simplicidade*: relativamente fácil tanto de se aprender quanto de se implementar.
- *Versatilidade*: podendo ser inserida em qualquer página web, JavaScript se integra muito bem com outras linguagens, o que torna seu domínio de aplicação muito amplo[2].

Por outro lado, alguns dos problemas apresentados por JS podem estar relacionados a:

- *Segurança*: apesar dos grandes benefícios, o fato da execução ocorrer client-side também pode trazer brechas que possivelmente serão exploradas pelos responsáveis pelo código JS malicioso[2].
- *Tipagem dinâmica*: agora lidando com outro tipo de segurança, a de tipos dentro do código. Nem todos os programadores acreditam que os benefícios de tipos dinâmicos superam suas desvantagens[5].
- Código JavaScript pode ser interpretado de diferentes maneiras por diferentes browsers, o que torna quase obrigatória a realização de testes em todos os browsers mais populares para garantir a ausência de erros[2].

Já fora do escopo de vantagens/desvantagens da linguagem, é necessário notar que há algumas limitações que, por si só, JavaScript ainda não consegue transponir, necessitando de alguma ferramenta ou script externo para poder implementar. Abaixo, algumas dessas limitações:

- Não consegue acessar bancos de dados.
- Não consegue escrever em arquivos localizados no servidor.
- Não consegue ler ou escrever em arquivos localizados no cliente - exceto *cookies*.
- Não consegue acessar páginas web localizadas em domínios diferentes[10].

2 TUTORIAL

Este tutorial é voltado para pessoas que já tem familiaridade com linguagens estilo C, contendo apenas uma breve descrição dos comandos e aspectos da linguagem.

2.1 Tipos em JavaScript

JavaScript é uma linguagem de tipagem dinâmica, na qual não é necessário declarar o tipo de uma variável. Os tipos básicos da linguagem são:

- Boolean - true ou false
- Number - valores ponto-flutuante de 64 bits
- String - sequência de caracteres unicode
- Object - conjunto de chaves e valores (dicionário)
- Function - tipo especial de objeto, que pode ser executado
- undefined - tipo que contém apenas o valor undefined

2.2 Variáveis

Variáveis são declaradas com a palavra-chave var, e têm seu escopo determinado pela função à qual pertence, ou tendo escopo global se declarada fora de uma função.

```
var indefinida; // Declaração, valor inicializado para undefined
var nome = “Carlos”; // Declaração e inicialização
```

2.3 Operadores

JavaScript conta com os operadores aritméticos +, -, *, / e % (resto da divisão). Estes operadores possuem grande ortogonalidade, sendo capazes de operar com a maioria dos tipos (mesmo que o sentido não seja claro). É possível somar inteiros e combinar strings com o mesmo operador +. A atribuição é feita com =, e pode ser combinada com operadores binários, como +=, -=, etc.

```
var a = 2;
var b = 3;
a = a + b; // a recebe 2 + 3
b += a; // b recebe b + a
b = “2” + “3” // b recebe o string “23”
```

Para comparações, temos os operadores >, <, >=, <=, ==, !=. As comparações realizam coerção de tipos, ou seja, converte tipos diferentes para o mesmo tipo para realizar a comparação. Assim, temos “0 == false” e “emptyString == false” como verdades. Para evitar a coerção de tipos, utilizam-se os operadores === e !==, que retornariam falso em ambos os casos.

```
var a = 5, resultado;

// resultado recebe true, pois há coerção de string “5” para number 5
resultado = (a == “5”);

// resultado recebe false, pois não há coerção
resultado = (a === “5”);
```

Os operadores && e || são utilizados para combinar afirmações lógicas, e contam com comportamento de curto-circuito, executando o segundo operador somente quando necessário para determinar o resultado lógico. Isso é útil para verificar se um objeto existe antes de acessar algum membro:

```
var valor = o && o.getMembro();
```

Ou para definir valores padrões:

```
var valor = variavel || "valor padrao";
```

2.4 Estruturas de Controle

Estrutura if/else para execução condicional de código, com encadeamento de estruturas:

```
if (valor > 0) {
      sinal = “+”;
} else if (valor < 0) {
      sinal = “-”;
} else {
      sinal = “”;
}
```

Estrutura while, que executa um loop básico, avaliando a expressão a cada iteração para determinar se continuará executando o loop.

```
while (valor > 0) {
      valor -= 1;
}
```

Estrutura do/while, que executa o loop pelo menos uma vez antes de avaliar a expressão.

```
do {
	valor /= 2;
} while (valor > 2);
```

Estrutura for, que executa loops com inicialização, condição e incremento:

```
for (i = 0; i < max; i++) {
	valor++; // Operador de incremento
}
```

2.5 Funções como objetos de primeira classe

Um *comando* `function` é expandido para uma atribuição de uma *expressão* `function` a uma variável:

```
function minhaFuncao(a, b) {
	return a + b;
}
```

é expandido para

```
var minhaFuncao = function (a, b) {
	return a + b;
}
```

2.6 Funções anônimas (lambda):

Exemplo de função anônima sendo declarada e executada imediatamente, devido ao uso dos parênteses após sua definição, com valor recebendo seu resultado.

```
var valor = (function () {
	var i, resultado;
	// código...
	return resultado;
}());
```

3 EXEMPLOS

3.1 Definir classes, atributos e métodos

JavaScript não tem classes, mas existem formas de simular o comportamento de classes utilizando objetos como protótipos ou funções como construtores:

```
// Definição da classe com objeto (protótipo)
var pessoaPrototype = {
    // Atributo nome com valor padrão
    nome: 'indefinido',

    // Método seApresenta
    seApresenta: function () {
        console.log('Oi, meu nome é ' + this.nome + '! :)');
    }
};

// Definição da classe com função (construtor)
function Pessoa() {

    // Atributo nome com valor padrão
    this.nome = 'indefinido';

    // Método seApresenta
    this.seApresenta = function () {
        console.log('Oi, meu nome é ' + this.nome + '! :)');
    };
}

// Definição da classe de forma híbrida
function Pessoa() {}
Pessoa.prototype = {

    // Atributo nome com valor padrão
    nome: 'indefinido',

    // Método seApresenta
    seApresenta: function () {
        console.log('Oi, meu nome é ' + this.nome + '! :)');
    }
};
```

3.2 Realizar encapsulamento e proteção dos atributos

O suporte à atributos e métodos privados é facilmente simulado com uso de closures (variáveis locais no escopo de execução do construtor). Métodos públicos privilegiados (com acesso às propriedades privadas) devem ser definidos dentro do construtor.

```
function Pessoa() {

    // Atributo privado definido como variável local do construtor,
    // sendo portanto inacessível fora do escopo desta definição
    var nome = 'indefinido';

    // Método privado, definido localmente no escopo do construtor
    // Retorna true se contém apenas letras e espaços
    function validaNome(nome) {
        return nome.trim().match(/^[a-zA-ZÀ-ü ]+$/);
    }

    // Método público, sendo acessível por pessoa.getNome()
    this.getNome = function () {
        return nome; // Acesso a variável privada, sem uso do this
    }

    // Método público, sendo acessível por pessoa.setNome()
    this.setNome = function (novoNome) {
          if (validaNome(nome)) // Utilização de método privado, sem uso do
this
            nome = novoNome;
    }
}
```

3.3 Construtores padrão e alternativos

Não é possível ter duas versões do mesmo construtor em JavaScript, mas a linguagem não cria nenhuma restrição quanto ao número de parâmetros que podem ser passados à uma função qualquer, e todos eles são acessíveis por meio um objeto similar a um *array* chamado *arguments* que é automaticamente inicializado no escopo das funções.

```
function Pessoa() {
    var nome;
    var num_args = arguments.length;

    // Construtor padrão: Pessoa()
    if (num_args === 0) {
        nome = 'indefinido';
    }

    // Construtor alternativo: Pessoa(string)
    else if (num_args === 1 && typeof arguments[0] === 'string') {
        nome = arguments[0];
    }

    // Proteção para argumentos inválidos
    else {
        throw new Error('Argumento(s) inválido(s) para Pessoa().');
    }
}
```

3.4 Destrutores

Não são necessários destrutores, pois a linguagem possui coletor de lixo automático. A linguagem não oferece construções equivalentes a destrutores.

3.5 Instanciar e usar objetos

A forma de instanciar um novo objeto varia de acordo com o padrão seguido para definir as "classes". Entretanto, a forma de acessar os atributos e métodos é universal.

```
// instanciação a partir de objeto (protótipo)
var pessoa = Object.create(pessoaPrototype);

// instanciação a partir de função (construtor)
var pessoa2 = new Pessoa();

// acesso a atributos (públicos)
console.log(pessoa.nome) // Mostra na tela o nome da pessoa

// acesso a métodos
pessoa.seApresentar() // Pessoa se apresenta
pessoa.setNome("Fulano"); // Altera atributo por método setter.
```

3.6 Definir escopo

Estratégia 1: Objeto (dicinário) como namespace:

```
var UFRGS = {}
UFRGS.login = function (usuario, senha) {
    // ...
}
```

Estratégia 2: Escopo local (encapsulamento de módulos). Utiliza-se escopo de função (única construção que define escopo na linguagem). Função anônima será executada com a delimitação de escopo:

```
(function () {
    var minhaVariavel;
    function minhaFuncao() {};
    // ...[meu módulo]...
}());
```

Estratégia 3: Híbrida. Função anônima retorna namespace de acesso publico (dicionário de métodos).

```
var MY_MODULE = (function () {
    // Implementação privada
    // [...]

    // retorna métodos públicos
    return {
        myAction: function (args) {
            // do something
        }
    };
})();
// ^ Função é executada imediatamente,
// MY_MODULE recebe o dicionário de funções públicas do módulo.
```

3.7 Especificar classes de base e classes especializadas (herança)

Como JavaScript não tem classes, a herança acontece de objeto para objeto (herança prototipal). Diferente da herança clássica em que os objetos carregam estado e herdam apenas estrutura e comportamento e carregam estado, na herança prototipal, objetos herdam estrutura, comportamento e estado. A cadeia de herança pode ser tão longa quanto necessária e sempre acaba em *`Object.prototype`*.

```
// Herança de protótipo
alunoPrototype = Object.create(pessoaPrototype);
```

Mesmo quando o objeto é criado utilizando uma função construtora com o operador *new*, o novo objeto estará herdando de um objeto. Neste caso, o objeto gerado sempre herdará do objeto que estiver atribuído à propriedade *prototype* da função construtora.

```
// Herança do construtor
function Aluno() {
      // Chamar o construtor da classe pai:
      Pessoa.call(this);
}
// Herdando o protótipo.
Aluno.prototype = Object.create(Pessoa.prototype);
```

No exemplo abaixo é implementada uma função para simular uma herança mais semelhante àquela observada em linguagens com suporte a classes. A função extend recebe o construtor de uma classe pai e um construtor para uma nova classe filho, e realiza a herança adotando a política de execução automática e implícita do construtor pai antes do construtor filho.

```
// Implementando herança com reuso de construtores
function extend(ClassePai, ClasseFilha, proprieties) {

    function Constructor() {

        // Chama o construtor pai, se existir
        if (typeof ClassePai === 'function')
            ClassePai.apply(this, arguments);

        // Chama o construtor filho, se existir
        if (typeof ClasseFilha === 'function')
            ClasseFilha.apply(this, arguments);
    }

    // Herda o protótipo da classe pai
    switch (typeof ClassePai) {
        case 'function':
            Constructor.prototype = Object.create(ClassePai.prototype);
            break;
        case 'object':
            Constructor.prototype = Object.create(ClassePai);
            break;
    }

    // Copia as propriedades novas para o novo protótipo
    Object.keys(proprieties).forEach(function (key) {
        Constructor.prototype[key] = proprieties[key];
    });

    // Retorna a função construtora
    return Constructor;
}

// Uso:
var Aluno = extend(
    Pessoa,           // Classe pai
    function () {},   // Construtor da classe filha
    {                 // Propriedades e métodos
        matricula: '00000000';
    });
```

3.8 Suportar Herança múltipla e políticas

Não é possível utilizar o mecanismo de herança prototipal disponibilizado pela linguagem para realizar herança múltilpla, mas é possível simular ela com simples cópia de atributos:

```
// Herda múltiplos protótipos, copiando seus métodos e atributos
function multi_extend() {
    var prototypesList = arguments;
    var i, extended = {};

    // Para cada protótipo recebido (em ordem inversa)
    for (i = prototypesList.length - 1; i >= 0; i -= 1) {
        // Adiciona a chave ao protótipo combinado
        Object.keys(prototypesList[i]).forEach(function (key) {
            extended[key] = prototypesList[i][key];
        });
    }
    return extended;
}

// Protótipo retangulo
var retangulo = {
    x: 0,
    y: 0,
    largura: 0,
    altura: 0
}

// Protótipo clicavel
var clicavel = {
    onclick: function () { console.log("Botão foi clicado! :("); }
}

// Protótipo botao
var botao = multi_extend(clicavel, retangulo);

// Nesta política de herança, são priorizados os métodos e atributos dos protótipos
// que foram passados primeiro à função multi_extend.

// Como a função multi_extend copia os protótipos em ordem inversa, os métodos e atributos
// que foram passados por último podem ser sobrescritos por aqueles que foram passados primeiro

// Como os atributos são copiados, não funciona como a herança prototipal com link ativo
// (mudanças nos protótipos pais não vão repercutir nos filhos).
```

3.9 Definir classes abstratas e concretas

O protótipo, em JavaScript, pode ser interpretado como uma classe abstrata, pois não tem construtor, apenas métodos e atributos. Já uma função construtora pode ser interpretada como classe concreta.

```
// Classe abstrata como protótipo, sem construtor
var PessoaAbstrata = {
    nome : null,
    getNome : function () {
        return nome;
    },
    setNome : function (nome) {
        this.nome = nome;
    }
}

// Classe concreta como função construtora
function PessoaConcreta() {
    var nome = null;

    this.getNome = function () {
        return nome;
    };

    this.setNome = function (nome) {
        this.nome = nome;
    };
}
```

3.10 Suportar polimorfismo por inclusão

Por ser uma linguagem com tipagem fraca, o polimorfismo funciona automaticamente sem ser necessária nenhuma construção adicional. A única exigência é que os objetos possuam métodos com os mesmos nomes (não é necessária a herança). E sempre é possível verificar a existência do método antes de tentar executá-lo.

```
var PessoaAbstrata = {
    seApresenta : function () {
        console.log("Olá, eu sou uma pessoa.");
    }
}

var Pessoa = extend(

    // Classe pai
    PessoaAbstrata,

    // Construtor da classe filha
    function () {},

    // Propriedades e métodos
    {});

var Aluno = extend(

    // Classe pai
    PessoaAbstrata,

    // Construtor da classe filha
    function () {},

    // Propriedades e métodos
    {
        seApresenta : function () {
            console.log("Olá, eu sou um aluno.");
        }
    });
}

// Método de PessoaAbstrata não sobreescrito
var pessoa1 = Pessoa();
pessoa1.seApresenta(); // Mostra "Olá, eu sou uma pessoa."

// Método de PessoaAbstrata sobreescrito
var pessoa2 = Aluno();
pessoa2.seApresenta(); // Mostra "Olá, eu sou um aluno."
```

3.11 Suportar polimorfismo paramétrico

Pela natureza de tipagem dinâmica, qualquer tipo e qualquer número de argumentos pode ser passados para qualquer função, permitindo que o próprio programador controle a consistência das entradas se achar necessário.

```
// JavaScript tem tipagem dinamica, os tipos não estão vinculados
// às variáveis, mas somente aos valores.

// Algoritmo genérico que lida com elementos de qualquer tipo
function seleciona(elemento1, elemento2, seletor) {
    if (seletor == 0)
        return elemento1;
    else
        return elemento2;
}

// Estrutura genérica que trabalha com elementos de qualquer tipo
function Fila() {
    var listaEmbutida = [];

    this.enqueue = function (valor) {
        listaEmbutida.push(valor); // Coloca ao fim da lista
    }

    this.dequeue = function () {
        return listaEmbutida.shift(); // Retira do início da lista
    }
}

var fila = new Fila();
fila.enqueue(1);
fila.enqueue('string');
fila.enqueue(pessoa);
fila.dequeue(); // Retorna o elemento 1, primeiro elemento da fila
```

3.12 Especificar/suportar polimorfismo por sobrecarga

Como é sempre possível passar qualquer tipo e número de parâmetros para qualquer função, não é necessário o suporte a polimorfismo por sobrecarga e, por este motivo, o mesmo não é suportado pela linguagem.

```
// Não existe polimorfismo por sobrecarga na linguagem,
// É necessário fazer checagem dos tipos e número de parâmetros manualmente.

// Pera, isso é o construtor? sim, melhor ser getPessoa
function getPessoa(idOuNome) {
    switch (typeof idOuNome) {
        case 'number':
            // [busca no banco de dados pessoa com id passado no argumento]
            return pessoa;

        case 'string':
            // [busca no banco de dados pessoa com nome passado no argumento]
            return pessoa;

        default:
            throw new error('Argumento inválido para getPessoa');
    }
}

// Por ser uma linguagem de tipagem dinâmica, polimorfismo é implementado
apenas verificando os tipos dos argumentos
// não existe polimorfismo real.
```

3.13 Especificar e usar funções como elementos de primeira ordem

Em JavaScript, funções são objetos, e portanto podem ser atribuídas, passadas como parâmetro e retornadas de funções.

```
function add(a, b) {
    return a + b;
}

// Armazena a função de subtração em uma variável
var sub = function (a, b) {return a - b;};

// Função que recebe duas funções e retorna a indicada por index
function selectFunction(fun1, fun2, index) {
    if (index == 0)
        return fun1;
    else
        return fun2;
}

selectFunction(add, sub, 0); // Retorna função add
selectFunction(add, sub, 1); // Retorna função sub
selectFunction(add, sub, 0)(1, 2); // Calcula add(1, 2), retorna 3
```

3.14 Especificar e usar funções de ordem maior

Como todas as funções da linguagem são naturalmente de primeira classe, é natural a utilização de funções que lidam com outras funções. A linguagem ainda conta com construções nativas funcionais como operações de foldr e foldl (que são chamadas reduceRight e reduce, respectivamente), que combinam listas utilizando uma função.

```
function sum(a, b) {
    return a + b;
}

[1,2,3,4,5,6].reduce(sum); // Equivalente a foldl, em JavaScript
[1,2,3,4,5,6].reduceRight(sum); // Equivalente a foldr, em JavaScript

// Função que aplica a função "fun" a todos os elementos do arranjo "array"
function map(array, fun) {
    var i, newArray = [];
    for (i = 0; i < array.length; i += 1) {
        newArray[i] = fun(array[i]);
    }
    return newArray;
}

// Exemplo de função que será passada como parâmetro
function square(value) {
    return value*value;
}

map([2, 3, 4], square) // Retorna [4, 9, 16]
```

3.15 Especificar e usar listas para a manipulação de estruturas, entidades e elementos

Para este exemplo, foi criada uma função de ordem maior que recebe uma estrutura qualquer de objetos e arranjos e aplica a função recebido a todos os elementos e subelementos primitivos de forma recursiva (referencias circulares causam loops infinitos).

```
// Aplica fun a todos os valores de tipo simples

function mapProfundidade(value, fun) {
    switch (typeof value) {
        case 'number':
            return fun(value);
        case 'string':
            return fun(value);
        case 'function':
            return value;
        case 'object':
            if (Array.isArray(value)) {
                return mapProfundidadeArray(value, fun);
            }
            if (value instanceof Date) {
                return new Date(value);
            }
            if (value instanceof RegExp) {
                return new RegExp(value);
            }
            return mapProfundidadeObjeto(value, fun);
    }
}

function mapProfundidadeArray(lista, fun) {
    var novaLista = [];
    lista.forEach(function(value, idx, lista) {
        novaLista[idx] = mapProfundidade(value, fun);
    });
    return novaLista;
}

function mapProfundidadeObjeto(objeto, fun) {
    var novoObj = {};
    var membro;
    for (membro in objeto) {
        novoObj[membro] = mapProfundidade(objeto[membro], fun);
    }
    return novoObj;
}
```

3.16 Especificar e usar funções não nomeadas (ou lambda)

```
// Função lâmbda para sucessor
function (value) { return value+1; }

// Passa a função lambda para map, retornando o sucessor de cada elemento
map([1, 2, 3], function (value) { return value+1;} ) // Retorna [2, 3, 4]
```

3.17 Especificar e usar funções que utilizem currying

```
function soma(a) {
    return function (b) {
        return a + b;
    };
}

// Avaliação completa
soma(3)(4) // Retorna 7

// Avaliação parcial, retorna uma nova função que incrementa 2 ao valor recebido
var incrementaDois = soma(2);
incrementaDois(3); // Retorna 5
```

3.18 Especificar e usar funções que utilizem pattern matching na sua definição

Não há construção equivalente em JavaScript, sendo necessário utilizar uma cadeia de if/else.

```
function fatorial(n) {
    if (n === 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

3.19 Especificar e usar recursão como mecanismo de iteração

Exemplo de recursão com função de ordem maior, que itera recursivamente sobre uma lista aplicando a função fornecida:

```
function foldr(list, fun) {
    var head = list[0];
    var tail = list.slice(1);

    if (list.length >= 2)
        return fun(head, foldr(tail, fun));
    else
        return 0;
}

var produto = foldr([1,2,3,4], function (a,b) { return a * b; });
```

3.20 Especificar e usar delegates.

Delegate de métodos é facilmente atingido na linguagem porque todas as funções possuem os métodos apply e call que recebem como primeiro argumento um objeto, que executam a mesma função como um método deste objeto e se diferenciam na forma de passar parametros para o método: a primeira recebe um arranjo representando os argumentos, a segunda recebe vários argumentos que são repassados.

```
var pessoa = {
    nome: "pessoa",
    falar: function () {
        console.log(this.nome + ": Oi, eu sou uma pessoa.");
    }
}

var cachorro {
    nome: "cachorro",
    falar: function () {
        console.log(this.nome + ": Au! Au! Au!");
    }
}

// Fazer pessoa falar como um cachorro:
cachorro.falar.call(pessoa); // Imprime "pessoa: Au! Au! Au!"

// Nesse exemplo, usamos o método call para delegar o método falar de
cachorro
// para ser usado pela pessoa como se fosse um método próprio.
```

3.21 Suporte para concorrência e paralelismo

Para demonstrar o suporte a concorrência, foi implementado o algoritmo Sleep Sort, um algoritmo de ordenação baseado em timers. Cada valor da lista não-ordenada ativa um processo filho com um timer correspondente ao seu valor. Por exemplo, o elemento com valor 5 ativa um processo que espera por 5 segundos e retorna a mensagem “5”. Assim, os números são recebidos de volta pelo processo pai com valores em ordem crescente.

Arquivo sleep_main.js:

```
var child_process = require('child_process'); // Importa biblioteca

// Ordena uma lista e envia o resultado à função callback
function sleepsort(lista, callback) {
    var listaOrdenada = [], childProcesses = [], i;

    // Função que coloca o elemento recebido por um filho na lista ordenada
    var onNumberReceived = function (number) {
            listaOrdenada.push(number);

            // Se todos os números foram recebidos
            if (listaOrdenada.length >= lista.length) {
                // Envia a lista ordenada à função de callback
                callback(listaOrdenada);
            }
        };

    // Cria um processo para cada elemento da lista
    for (i = 0; i < lista.length; i += 1) {
        childProcesses[i] = child_process.fork('sleep_child');
        childProcesses[i].on('message', onNumberReceived);
        childProcesses[i].send(lista[i]);
    }
}

// Imprime a lista resultante
function imprimeLista(lista) {
    console.log(lista);
    process.exit();
}

// Inicia o sleepsort, chamando imprimeLista ao final
sleepsort([5, 2, 4, 1, 3], imprimeLista); // Imprime [1, 2, 3, 4, 5]
```

Arquivo sleep_child.js:

```
function sleep(number) {

    setTimeout(function () {
        process.send(number);
    }, 1000 * number)
}
process.on('message', sleep);
```

4 ANÁLISE CRÍTICA

4.1 Simplicidade

Nota: 7/10

A definição da linguagem é bastante compacta, sem diversos dos elementos encontrados comumente em outras linguagens, como classes, extensões, tipos de atributos. Há um tratamento uniforme de valores como objetos, incluindo funções.

4.2 Ortogonalidade

Nota: 9/10

É possível combinar a maioria do tipo com a maioria dos operadores, mesmo em construções que não fazem sentido.

“abc” + “def” + 123; // Retorna string “abcdef123”

function1() {}
function2() {}
function1 / function2; // Construção permitida, retorna NaN

1 = 2; // Erro de atribuição

4.3 Expressividade

Nota: 8/10

Construções como array.forEach (itera sobre todos os elementos), array.every (retorna true se para todos os elementos do arranjo a função passada retorna true) e outros métodos de ordem maior dão grande expressividade à linguagem, além da capacidade de definir seus próprios métodos para tipos básicos.

4.4 Adequabilidade e variedade de estruturas de controle

Nota: 8/10

Todas as estruturas de controle usuais em linguagem procedurais estão disponíveis, contando ainda com a aceitação de strings em switches e construções como “for in”, que itera sobre membros de um objeto. JavaScript conta também com estruturas de controle funcionais que recebem funções de ordem maior, porém faltam algumas características funcionais como pattern matching na definição de funções, que devem ser substituídos por uma cadeia de if/elses ou switches.

4.5 Mecanismos de definição de tipos

Nota: 5/10

O programador consegue definir seus construtores para classes de objetos e detectar se os objetos foram construídos por esses construtores (se pertencem à classe), mas não definem novos tipos, todos são tratados como tipo *Object* pela linguagem.

4.6 Suporte a abstração de dados e de processos

Nota: 9/10

JavaScript tem grande suporte à abstração de dados, utilizando objetos como estruturas, hashmaps etc. Processos são facilmente abstraídos com uso de funções para definir métodos, construtores, eventos etc.

4.7 Modelo de tipos

Nota: 5/10

O modelo de tipos não protege o programador de erros por não fazer verificação alguma e permitir uma grande ortogonalidade até em operações que não fazem sentido, como já mencionado, sem fornecer ao programador nenhuma dica de onde o erro foi introduzido.

4.8 Portabilidade

Nota: 9/10

Por ser uma linguagem que foi introduzida a vários ambientes (navegadores e servidores), existem poucos dispositivos que não são capazes de suportar a linguagem. Praticamente todos os computadores e dispositivos móveis estão equipados com algum interpretador da linguagem. Não há necessidade de adaptar código, porém, deve-se tomar o cuidado de usar uma versão menos recente da especificação, de mais ampla adoção.

4.9 Reusabilidade

Nota: 6/10

A linguagem não restringe o programador para encorajá-lo a criar estruturas de baixo acoplamento, o que pode acabar gerando grande dependência de componentes. Mesmo assim, há bastante disponibilidade de código devido à ampla adoção da linguagem.

4.10 Suporte e documentação

Nota: 8/10

A linguagem é extensivamente documentada e padronizada, com padrões mantidos pela Ecma International (sob o nome de ECMAScript), e documentação não-oficial, porém mais amigável, suportada pela Mozilla Foundation[4].

4.11 Tamanho de código

Nota: 7/10

A linguagem é bastante flexível, não sendo em geral necessário código “boilerplate”, como “int main(int argc, char **argv) { return 0; }” em C ou classes obrigatórias como em Java. Assim, o código tende a ser bastante compacto.

4.12 Generalidade

Nota: 9/10

Javascript é uma linguagem muito flexivel onde muitos design patterns podem ser implementados.

4.13 Eficiência

Nota: 7/10

JavaScript é uma das linguagens interpretadas mais fortemente otimizadas, com compilação just-in-time, ganhando muito em eficiência quando comparado a outras linguagens script como Python. Porém, ainda perde para linguagens compiladas como Java e C.

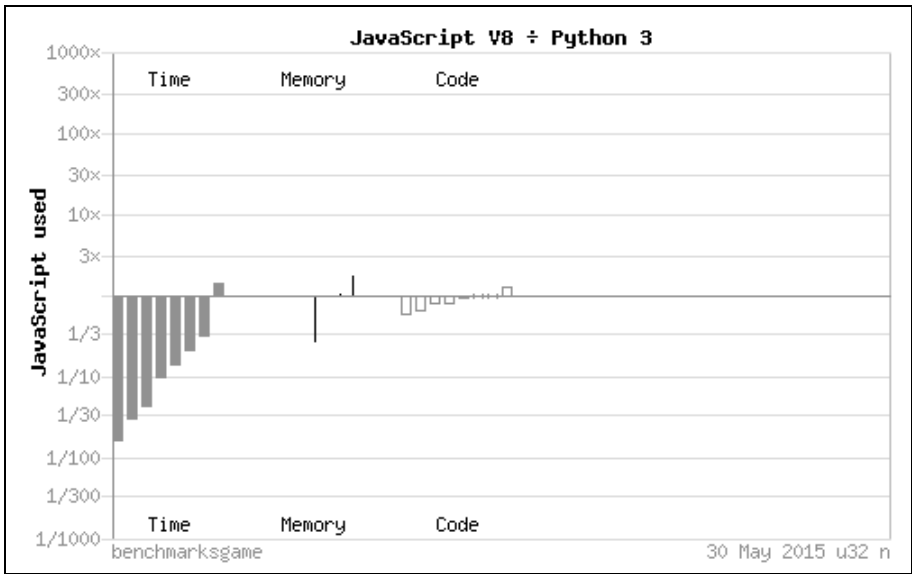

Figura 1: Comparação entre JavaScript e Python 3. As barras indicam cada um dos programas de exemplo, disponíveis no link. Percebe-se uma quantidade parelha de uso de memória e densidade de código, mas grande vantagem em eficiência.
(Fonte: http://benchmarksgame.alioth.debian.org/u32/compare.php?lang=v8&lang2=python3)

4.14 Custo

Nota: 5/10

Embora a linguagem seja de fácil aprendizado inicialmente, a ortogonalidade elevada e a permissividade dificultam a detecção de erros, que podiam facilmente ser manifestados em tempo de compilação do script. Além de não haver verificação de tipos, não há checagem no número de parâmetros, o que pode implicar custos de desenvolvimento grandes para encontrar erros no programa. Ainda, embora vantajosos, eventos concorrentes introduzem um novo grau de complexidade incomum a outras linguagens.

5 CONCLUSÕES

A partir dos exemplos desenvolvidos e dos aspectos analisados, percebemos o quanto JavaScript é uma linguagem versátil, podendo implementar diversas construções mesmo que não exista suporte direto de linguagem, como o suporte a classes e herança clássica. Alguns aspectos funcionais da linguagem, muitas vezes ignorados, também puderam ser trabalhados mais a fundo, como a utilização de funções de ordem maior.

Ao mesmo tempo, percebemos que embora JavaScript seja altamente dinâmica e tenha grande aplicação prática, não é uma linguagem que auxilia o programador na confiabilidade do código, permitindo construções bastante ortogonais que implicam grandes riscos e pouca utilidade prática.

Mesmo assim, são problemas mitigáveis com uso mais restrito dos recursos da linguagem aliado a boas práticas de programação específicas para esta linguagem.

6 BIBLIOGRAFIA

[1] A Survey of the JavaScript Programming Language
http://javascript.crockford.com/survey.html

[2] Advantages & Disadvantages of JavaScript
http://www.mediacollege.com/internet/javascript/pros-cons.html

[3] Design patterns in Javascript
http://addyosmani.com/resources/essentialjsdesignpatterns/book/

[4] Documentação não oficial do JavaScript (Mozilla):
https://developer.mozilla.org/en-US/docs/Web/JavaScript

[5] JavaScript
http://en.wikipedia.org/w/index.php?title=JavaScript&oldid=664648403

[6] Node.js (JavaScript server-side):
http://nodejs.org/

[7] Prototype-based programming
http://en.wikipedia.org/w/index.php?title=Prototype-based_programming&oldid=664367278

[8] Re-introduction to JavaScript (Mozilla):
https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

[9] The World's Most Misunderstood Programming Language - D. Crockford:
http://javascript.crockford.com/javascript.html

[10] What are some limitations of JavaScript?
http://www.quora.com/What-are-some-limitations-of-JavaScript

[11] What Javascript Can Not Do
http://javascript.about.com/od/reference/a/cannot.htm