



Introdução às Linguagens de Programação: conceitos elementares

Prof. Mauro Larrat

maurolarrat@ufpa.br

2016

Livro texto e referências

material do curso:

ROBERT W. SEBESTA, CONCEPTS OF PROGRAMMING LANGUAGES
TENTH EDITION, 2012.

outras referências:

Programming Language Design Concepts- David A. Watt, William
Findlay.

Por que estudar conceitos de Paradigmas de Programação?

Por que estudar Linguagens de Programação?

1 - Aumento da capacidade de expressar ideias:

- conhecer diferentes formas de expressar **ideias abstratas** e descrever **estruturas de dados e de controle** complexas;
- linguagens de programação possuem **capacidades** diferentes;

Exemplo:

A linguagem *C* não possui arrays associativos ou dicionários (índices são strings) , diferentemente de linguagens como PHP, Perl e C++. Contudo, podemos emular estas arrays através de estruturas da linguagem *C*.

Por que estudar Linguagens de Programação?

Exemplo de array associativo em C++:

```
#include <vector>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <map>
using namespace std;
int main(){
    map <string,int> mapa;
    mapa["brasil"] = 2;
    mapa["argetina"] = 3;
    mapa["inglaterra"] = 4;
    if(mapa["franca"])
        printf("%d\n",mapa["franca"]);
    if(mapa["brasil"])
        printf("%d\n",mapa["brasil"]);
    map<string,int>:: iterator mapit;
    for(mapit = mapa.begin(); mapit!=mapa.end(); mapit++)
        cout << mapit->first << " " <<mapit->second << endl;
    system("PAUSE");
}
```

Exemplo de array associativo em PHP:

```
<?php
$a['a'] = 10;
$a['b'] = 20;
$a['c'] = 30;
?>
```

Por que estudar Linguagens de Programação?

Exemplo de array associativo em C (retirado do site rosetacode):

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int size;
    void **keys;
    void **values;
} hash_t;

int main () {
    hash_t *h = hash_new(15);
    hash_insert(h, "hello", "world");
    hash_insert(h, "a", "b");
    printf("hello => %s\n", hash_lookup(h, "hello"));
    printf("herp => %s\n", hash_lookup(h, "herp"));
    printf("a => %s\n", hash_lookup(h, "a"));
    return 0;
}
```

```
hash_t *hash_new (int size) {
    hash_t *h = calloc(1, sizeof (hash_t));
    h->keys = calloc(size, sizeof (void *));
    h->values = calloc(size, sizeof (void *));
    h->size = size;
    return h;
}

void hash_insert (hash_t *h, void *key, void *value) {
    int i = hash_index(h, key);
    h->keys[i] = key;
    h->values[i] = value;
}

void *hash_lookup (hash_t *h, void *key) {
    int i = hash_index(h, key);
    return h->values[i];
}
```

Por que estudar Linguagens de Programação?

2 - Aperfeiçoar a escolha da linguagem correta:

- Familiaridade com uma ampla gama de linguagens **facilita a escolha da linguagem mais adequada** para a resolução de determinado problema.
- deixar utilizar uma linguagem com uma **característica nativa** para usar uma **simulação desta característica** em uma outra linguagem mais conhecida pelo desenvolvedor **pode tornar o código menos elegante**, mais pesado e menos seguro.

Por que estudar Linguagens de Programação?

3 - Aumenta a habilidade em aprender novas linguagens:

- Entender os conceitos fundamentais de diferentes tipos de linguagens de programação **tornam mais fácil o aprendizado de linguagens** com conceitos similares.
- Por exemplo, quem conhece os fundamentos conceituais de orientação a objetos pode ter mais facilidade em aprender a linguagem Java e C++.
- A similaridade entre diferentes linguagens **facilita o aprendizado** das mesmas. Ex.: Java, C++ e C são similares em sintaxe e semântica (http://www.tiobe.com/tiobe_index/index.htm).

Por que estudar Linguagens de Programação?

4 - Melhor entendimento do significado do paradigma:

- Entender restrições dos paradigmas através da prática conduz ao entendimento do **porquê** da linguagem ser desenvolvida de **determinada** maneira, permitindo o uso mais inteligente dos recursos da linguagem.
- permite-nos visualizar **como** um computador **executa** várias construções da linguagem.
- Conhecer questões de implementação de estruturas da linguagem fornece dicas sobre a **eficiência relativa** de diversas construções alternativas que podem ser escolhidas para desenvolver um programa.

Por que estudar Linguagens de Programação?

5 - Aperfeiçoamento de linguagens já conhecidas:

- É comum que desenvolvedores não conheçam todos os recursos oferecidos pela linguagem que se está mais familiarizada.
- Conhecer mais sobre outros paradigmas **facilita o aprendizado de recursos nunca utilizados** ou menos conhecidos de linguagens mais familiares.

Por que estudar Linguagens de Programação?

Estudo de caso: ALGOL60 x FORTRAN (ver livro referência)

Alguns programadores acreditam que seria melhor se a linguagem de programação ALGOL60 tivesse sido mais disseminada na década de 60 do que a linguagem FORTRAN.

Isto não aconteceu (em parte) devido aos programadores e projetistas de software não possuírem (na sua maioria) na época conhecimentos aprofundados nos recursos da ALGOL60, tais como recursão, construções em blocos e estruturas de controle bem definidas. FORTRAN era mais intuitiva do que ALGOL60, na sua essência.

Áreas de Aplicação de Linguagens de Programação

Aplicações Científicas

- Os primeiros computadores digitais (entre 1940 e 1950) foram desenvolvidos para **aplicações científicas**.
- A funcionalidade principal consistia na **computação de números largos** com representação em ponto flutuante.
- As estruturas de dados mais comuns eram **arrays e matrizes**; as estruturas de controle mais comuns eram **laços de repetição e atribuições**.
- O foco principal era o **desempenho** oferecidos pelas linguagens mais próximas da linguagem de máquina. A linguagem **FORTRAN** predominou sobre as outras linguagens nessa época.

Aplicações de Negócios

- A partir de **1950** as aplicações começaram serem desenvolvidas com **foco em negócio**, favorecendo o surgimento de linguagens de programação adequadas a este propósito.
- A primeira linguagem com foco em negócio foi a **COBOL**.
- Estas linguagens são caracterizados pela **facilidade na produção de relatórios** bem elaborados, formas precisas de **descrever e armazenar números decimais e dados de caracteres**, e a capacidade de **especificar operações aritméticas decimais**.

Aplicações de Inteligência Artificial

- Introdução da ideia de **manipulações computacionais simbólicas**, além de **manipulações computacionais numéricas**.
- O uso de **listas ligadas** foi mais empregado para a manipulação de informações, pela flexibilidade de manipulações sobre arrays e matrizes simples.
- Foco em linguagens baseadas em **blocos de funções e lógica matemática**.
- Linguagens como **LISP** (1965, variante: Scheme, 1990), **PROLOG** (1970) e **C** são linguagens muito utilizadas em aplicações inteligentes.

Aplicações em Programação de Sistemas

- **Sistemas operacionais** e ferramentas de apoio são exemplos de aplicações de sistemas.
- Programação de sistemas **estendem ou aprimoram o funcionamento** de um sistema operacional. Inclui o desenvolvimento de drivers, compiladores, utilitários do sistema atualizações para sistemas operacionais ou mesmo novos sistemas operacionais.
- Em resumo, são programas responsáveis por controlar adequadamente o hardware (acesso direto) para que estes possam executar os aplicativos, ou fornecem um ambiente para o desenvolvimento e uso de aplicativos.

Aplicações em Programação de Sistemas

- Alguns exemplos são **PL/S** (e derivadas) para mainframes da IBM e **ALGOL** da UNISYS - ambas linguagens muito próximas da linguagem de máquina assembly.
- Hoje em dia linguagens como **C** e **C++** são mais utilizadas para o desenvolvimento de sistemas. A exemplo, o sistema operacional **UNIX**, o qual é quase todo construído em **C**.

Aplicações em Programação para WEB

- As aplicações WEB geralmente são compostas por linguagens de programação (javascript, php) e linguagens de marcação (HTML) de texto.
- Com algumas exceções (C++ CGI - Common Gateway Interface e Apache, por exemplo), as linguagens de programação para WEB atuam na forma de **scripts**, que fornecem funcionalidades embutidas no código de marcação de texto.

Critérios de Avaliação de Linguagens de Programação

Objetivos

Avaliar recursos das linguagens de programação, concentrando-se nos impactos sobre o processo de desenvolvimento de software, incluindo a manutenção deste.

Veremos a seguir três critérios e nove características destes critérios as quais vamos nos basear para avaliar linguagens de programação.

Critérios e características para avaliação de LP

CARACTERÍSTICAS	PRINCIPAIS CRITÉRIOS		
	LEGIBILIDADE	ESCRITA	CONFIABILIDADE
SIMPLICIDADE	x	x	x
ORTOGONALIDADE	x	x	x
TIPOS DE DADOS	x	x	x
SINTAXE	x	x	x
NÍVEL DE ABSTRAÇÃO		x	x
EXPRESSIVIDADE		x	x
CHECAGEM DE TIPOS DE DADOS			x
MANIPULAÇÃO DE EXCEÇÃO			x
ALIASING RESTRITO			x

Critérios de Avaliação de Linguagens de Programação: Legibilidade

Critério: legibilidade

A linguagem possui facilidade de leitura e entendimento do código?

Antes da década de 70 o **objetivo era eficiência** (facilidade de interpretação por parte do computador e não do programador).

A partir da década de 70 passou a existir a **ideia de ciclo de vida** de software (custo de manutenção do software passa a ser tão importante quanto a eficiência).

A **facilidade de manutenção** é determinada em grande parte pela **legibilidade** dos programas, tornando-se uma importante **medida de qualidade** dos programas e linguagens de programação.

Critério: legibilidade

A importância na manutenção de software com foco em legibilidade foi um **marco** importante onde se começou a **diferenciar** entre a **programação orientada à máquina e orientada ao usuário**.

Um programa deve ser desenvolvido em uma linguagem de programação que se adéqua ao **contexto do problema** computacional, do contrário, a leitura do código pode ser complicada e difícil.

Critério: legibilidade e simplicidade

Problemas quanto à legibilidade e simplicidade da linguagem:

- 1 - grande quantidade de instruções;
- 2 - multiplicidade de operações similares;
- 3 - sobrecarga de operadores;
- 4 - simplicidade em excesso: falta de recursos da linguagem;

Critério: legibilidade e simplicidade

Uma linguagem com um grande número de construções básicas é mais difícil de aprender do que uma com **número reduzido de construções**.

Os programadores que devem usar uma linguagem grande, muitas vezes aprendem um **subconjunto da linguagem** e ignoram suas outras características.

problemas de legibilidade ocorrem sempre que o autor do programa aprendeu um subconjunto diferente do subconjunto com o qual o leitor do programa é familiar.

Critério: legibilidade e simplicidade

Uma segunda característica é a **multiplicidade**, isto é, ter mais do que uma maneira de realizar uma operação em particular.

Por exemplo, em Java, C ou C++, um utilizador pode incrementar uma variável inteira de quatro modos diferentes:

```
count = count + 1;
```

```
count += 1 ;
```

```
count++ ;
```

```
++count ;
```

Critério: legibilidade e simplicidade

Um terceiro problema potencial é a **sobrecarga de operador**, em que um único símbolo operador tem mais do que um sentido.

Sobrecarregar um operador significa redefinir um símbolo, de modo que ele se aplique também a **tipos de dados definidos pelo usuário** como classes e estruturas.

Vamos à um exemplo em C++, onde criamos uma classe **Ponto**, que representa um par ordenado em um plano.

Tentaremos somar dois objetos Pontos com o operador padrão "+":

Critério: legibilidade e simplicidade

```
#include <iostream>
using namespace std;
class Ponto{
    private:
        int x, y;
    public:
        Ponto();
        Ponto(int x1, int y1);
        void print();
};
```

```
#include "Ponto.h"

Ponto::Ponto(){
    x = 0;
    y = 0;
}

Ponto::Ponto(int x1, int y1){
    x = x1;
    y = y1;
}

void Ponto::print(){
    cout << "(" << x << ", " << y << ")" << endl;
}
```

Critério: legibilidade e simplicidade

Se instanciarmos três objetos da classe Ponto e tentarmos somar dois destes pontos e armazenar o resultado no terceiro ponto,

```
Ponto p1, p2, p3;  
p1 = p2 + p3;  //erro!
```

ERRO: Tipos definidos pelos usuários não podem ser manipulados com os mesmos operadores que funcionam com os tipos primitivos!!!

Critério: legibilidade e simplicidade

Contudo, no C++, é possível sobrecarregar operadores através de funções operadoras.

```
tipo_retorno Nome_classe::operator simbolo_operador(parametros){  
    //definição da (nova) função do operador  
}
```

Critério: legibilidade e simplicidade

```
#include <iostream>
using namespace std;
class Ponto{
    private:
        int x, y;
    public:
        Ponto();
        Ponto(int x1, int y1);
        void print();
        Ponto operator +(Ponto p); //função operadora
};
```


Critério: legibilidade e simplicidade

Definição da **funções operadora** no C++:

```
Ponto Ponto::operator +(Ponto p){  
    Ponto aux;  
    aux.x = x + p.x;  
    aux.y = y + p.y;  
    return aux;  
}
```

Mesmo que a sobrecarga de operadores **diminua a necessidade de novos operadores**, pode ser confuso para o leitor entender o uso de um mesmo operador com mais de uma função (função personalizada).

Critério: legibilidade e simplicidade

Um ponto importante é que a **simplicidade em excesso** (como no caso da linguagem de máquina) pode tornar a escrita (e leitura) do **código mais complexa**, devido a falta de estruturas de códigos mais robustas, como laços de repetição e declarações condicionais.

Exemplo de um simples laço de repetição com decisão:

```
xor cx,cx ;cx-register is the counter, set to 0.  
loop1:  
  nop    ;Whatever you wanna do goes here(should not change cx).  
  inc cx    ;Increment.  
  cmp cx,3    ;Compare cx to the limit 3.  
  jle loop1   ;Loop while less or equal.
```