Introdução à Lógica de Programação

Paulino Ng

2019-08-13

Ementa

Lógica de Programação e suas representações. Construção de Algoritmos. Constantes e Variáveis. Estrutura de Controle Linear e Condicional Simples e Composta. Estrutura de Controle de Repetição. Estruturas de Dados Homogêneas: Vetores.

Objetivo

Apresentar os fundamentos para a programação de computadores.

Competências

- Identificar os elementos de uma linguagem de programação
- Compreender os tipos básicos de dados e as estruturas de controle de fluxo de instruções
- Analisar os tipos de problemas simples que podem ser resolvidos com a programação de computadores

Habilidades

- Projetar pequenos programas
- Especificar soluções computacionais para problemas de pequena dimensão em computação
- Analisar problemas computacionais
- Programar pequenas aplicações

Bibliografia

Básica

1. MANZANO, José Augusto N.G.; OLIVEIRA, Jayr Figueiredo, Algoritmos: lógica para desenvolvimento de programação de computadores, 28. ed, Saraiva, 2016.
2. SOFFNER, Renato, Algoritmos e programação em linguagem em C, Saraiva, 2013.

3. IEPSEN, Edécio Fernando, Lógica de programação com JavaScript: uma introdução à programação de computadores com exemplos e exercícios para iniciantes, NOVATEC,2018.

Complementar

1. ALVER, William Pereira, Linguagem e lógica de programação, Érica, 2014.
2. SANTOS, Marcela Gonçalves dos, Algoritmos e programação, SAGAH, 2018.
3. SEBESTA, Robert W., Conceitos de Linguagens de Programação, 9ª Ed., Bookman, 2011.
4. MONK, Simon, Programação com Arduino: começando com o sketches, SAGAH, 2017.
5. MACHADO, Rodrigo Prestes; FRANCO, Márcia Islabão; BERTAGNOLLI, Silvia de Castro, Desenvolvimento de Software III: programação de sistemas web orientada a objetos em Java, SAGAH, 2016.

Compilação

Um programa é um texto com um conjunto de instruções em alguma linguagem de programação. Ele normalmente é armazenado num arquivo numa memória permanente de um computador. Este programa não é diretamente executável. O computador executa programas em linguagem de máquina - código binário. Para executar o programa escrito em linguagem de programação, este programa precisa ser traduzido para a linguagem do computador. O programa que faz esta tradução é o **compilador**.

A figura abaixo ilustra a compilação do programa `alo.c`.

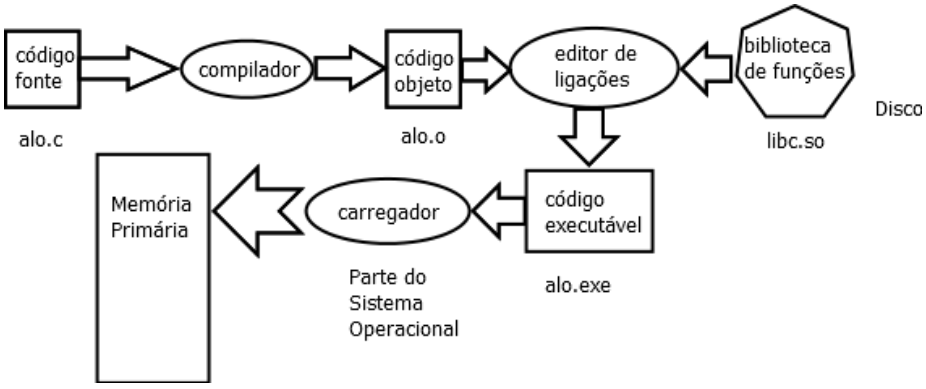

Figure 1: Figura 1 - Etapas da conversão de um programa em código fonte até ser executado.

O programa `alo.c` tem o seguinte texto:

```c
#include <stdio.h>

int main() {
  printf("Alo, Mamae!\n");
  return 0;
}
```

O código em linguagem C é um texto que o computador não entende. O compilador o converte para um *código objeto*. O arquivo `alo.o` é em código objeto. Este já é um código binário com as instruções em código de máquina, mas este código é relocável, isto é, ainda não tem sua posição na memória definida. O código faz referência à posição de suas instruções e dados e estas posições ainda não estão definidas.

O código também faz referência a funções pré-definidas (`printf()`) e o código destas funções estão em bibliotecas (arquivo `libc.so`) no disco do computador. O *editor de ligações* reune estes códigos objetos e calcula a posição em que eles devem ficar na memória resolvendo (calculando) os endereços que devem ser usados. O *editor de ligações* gera o arquivo executável, `alo.exe`.

O processador não acessa as instruções no disco, ele precisa que as instruções estejam na memória principal, o *Sistema Operacional*, *SO*, "carrega" o arquivo executável numa área pré-definida da memória principal e passa o fluxo de instruções para o programa carregado. Quando o programa termina, o *SO* volta a executar.

Linguagem de Máquina x *Assembly*

Seres humanos não programam em código binário, com exceções excêntricas, o mais próximo da linguagem de máuina que seres humanos programam é em linguagem *Assembly*. *Assembly* usa mnemônicos (palavras que imitam algo) para as instruções de máquina em vez de usar códigos binários. Além disso, na programação em *Assembly* são usados *rótulos* (*labels*) no lugar de endereços. Os *rótulos* são identificadores que facilitam entender que trecho de código corresponde a uma função ou um bloco de dados. Uma tabela liga os nomes com os endereços. Esta tabela é chamada de tabela de símbolos. Os compiladores trabalham com tabelas de símbolos mais complexas, mas uma das funções da tabela de símbolos é estabelecer este tipo de relacionamento entre identificadores e endereços. Programas em Assembly são convertidos para linguagens de máquina por um *assembler*, chamado em português de *montador*.

O *montador* usa uma tabela de símbolos fornecida no programa Assembly pelo programador para converter os *rótulos* em endereços. Se for necessários, o *montador* calcula o endereço a partir de outros endereços, somando os deslocamentos necessários de acordo com o número de bytes necessários para as instruções e os dados.

Compilação x Interpretação

Existem linguagens de programação cujos programas fontes não são compilados para códigos de máquina do computador onde eles são executados. Nestes casos, o programa fonte pode ser interpretado por um *interpretador*, isto é, o *interpretador* serve de intermediário e dinamicamente, converte o código fonte em instruções de máquina durante a execução do programa. Diferente do compilador que converte o programa antes da sua execução. Um esquema intermediário bastante usado atualmente é a compilação do código fonte para um código numa linguagem intermediária e o código da linguagem intermediária é que é interpretado. Isto acontece com o Java que é compilado em *Java Byte Code* e o código em Java Byte Code é interpretado por uma *JVM*, *Java Virtual Machine*. Algumas linguagens que eram interpretadas antes, agora também são compiladas em código intermediário dinamicamente. Entretanto, não existe a geração de arquivo compilado em algum tipo de código intermediário, o código compilado fica na memória principal e é interpretado a partir de lá. Isto ocorre, por exemplo com o Lisp e o Perl.

% Aula 2 de Lógica de Programação % Paulino Ng % 2019-08-20

Nesta aula veremos alguns conceitos básicos de computação para podermos entender melhor os conceitos de programação que serão vistos posteriormente. Vamos começar pela arquitetura de *von Neumann* que descreve conceitualmente a estrutura física dos computadores modernos. Veremos como isto afeta a programação dos computadores e como as linguagens de programação procuram fornecer uma abstração de alto nível para que não tenhamos de programar num nível conceitual próximo ao da máquina computacional.

Lembro que nesta disciplina só nos interessa o computador digital. No computador digital a unidade de informação é um *bit*. Um *bit* é algo que pode ter apenas 2 valores (estados), **0** ou **1**. Fisicamente o bit pode ser um sinal elétrico do tipo tensão (tensão baixa, quase zero, é **0**, alta, 2 a 5V, ou mais, é **1**, presença de corrente é **1**, corrente nula é **0**, frequência baixa é **1**, alta é **0**, enfim, escolha-se uma convenção para um tipo de implementação e defina-se o significado do bit e seus valores), pode ser diversas outras *convenções*.

Arquitetura de *von Neumann*

John von Neumann, um cientista, matemático e físico húngaro-americano, considerado como o último dos grandes matemáticos, é o autor de um artigo que descreve um dos primeiros computadores da história. Não vou reproduzir exatamente a arquitetura do artigo, mas resumir de maneira mais moderna o que foi proposto.

Antes, vamos entender o conceito de arquitetura de computador. É bem similar ao conceito de arquitetura de edificações. É óbvio que a arquitetura tem a ver com a estrutura física, mas não necessariamente com a realização material. Como na construção civil, o engenheiro civil está preocupado com a realização

física do prédio, o arquiteto está preocupado com aspectos funcionais, estéticos, ambientais, ... Na computação, o arquiteto de um computador se preocupa com funcionalidades dos componentes do computador, conjunto de instruções do processador, protocolos de comunicação entre os componentes do computador, ... Os engenheiros de computação, projetistas de hardware, HW, são as pessoas preocupadas em realizar fisicamente, dizemos implementar o computador. Veremos a arquitetura, não a implementação.

Componentes da arquitetura de *von Neumann*

Os componentes da arquitetura de *von Neumann* são:

1. **Processador**: responsável por executar as instruções
2. **Memória**: responsável por armazenar os dados e as instruções
3. **Barramento**: responsável pela comunicação entre os componentes
4. **Entradas**: responsável por capturar/receber os dados externos ao computador
5. **Saídas**: responsável por enviar os dados para o meios externos ao computador

Memória principal do computador

Ela é formada pelos módulos de memória DDR nos PCs que os alunos usam. A memória pode ser entendida como uma tabela onde são guardados os dados e as instruções do computador. Na tecnologia atual, a tabela tem uma única coluna e em cada linha temos um *byte* (8 bits).

Processador

- ciclo de busca e execução
- cjto de instruções
- cálculo da próxima instrução
- sequencialidade das instruções na arquitetura de von Neumann
- interação com os barramentos

Barramento

Sistema de comunicação que interliga os outros componentes permitindo a transferência de dados/instruções.

Entradas e Saídas

Componentes que permitem adquirir dados de e enviar dados para fora do computador. Geralmente, são compostos por duas partes: um controlador de dispositivo e um periférico.

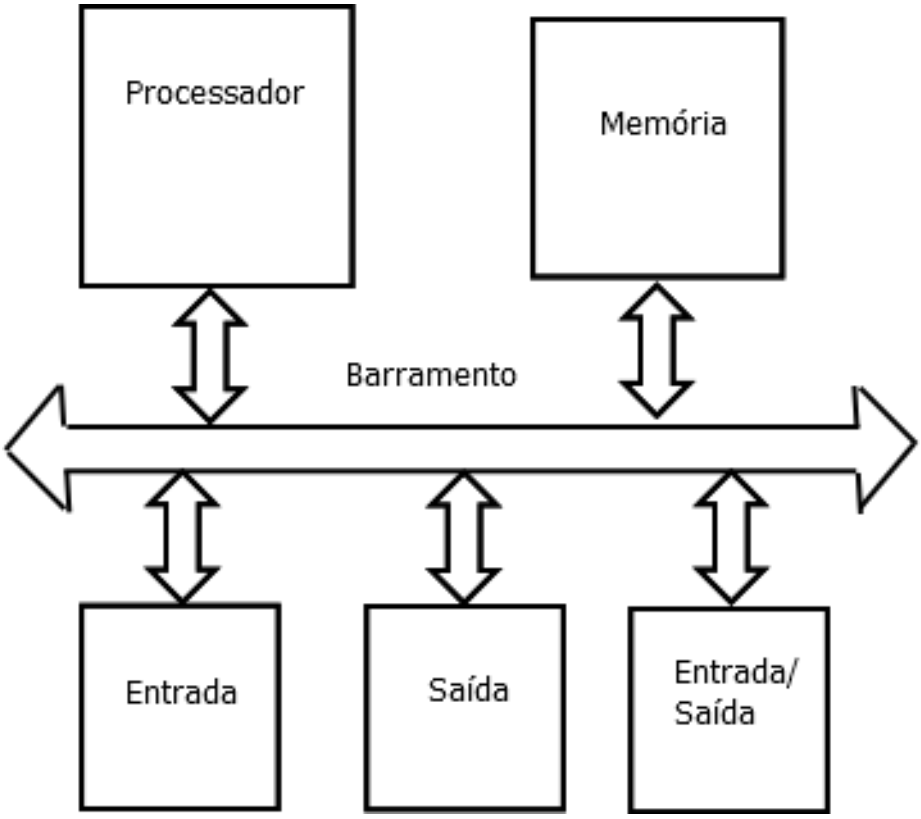

Figure 2: Arquitetura de um computador monoprocessador

Linguagens de Programação

Programar diretamente na linguagem do computador é muito complicado para seres humanos. A linguagem mais próxima da linguagem de máquina é a linguagem *Assembly* que usa mnemônicos para as instruções. Os mnemônicos são palavras, ou siglas/abreviações, que em inglês lembram o que a instrução faz. Exemplos de mnemônicos são: MOV, CPY, CMP, JMP, JZ, JC, ADD, SUB, MUL, DIV, ... Algumas instruções de máquina precisam dos endereços de memórias de dados ou de instruções, para não ficar usando números sem significados, o *Assembly* permite o uso de *rótulos* para os endereços e o programador pode definir o valor destes *rótulos* ou deixar o *Assembler* calcular o endereço a ser usado. O *Assembler* é o programa que converte o texto *Assembly* (programa *Assembly*) em linguagem de máquina. Em português, chamamos o *Assembler* de *montador*.

Alguns pequenos programas (subprogramas) são escritos em *Assembly* por precisarem acessar detalhes de *hardware* que não são acessíveis com linguagens de alto nível, como partes de um *driver* de *SO* ou por necessitarem de um desempenho melhor do que o compilador pode oferecer. Observe que compiladores modernos são capazes de otimizar o código muito melhor do que a maioria dos programadores. A programação de microcontroladores muitas vezes é realizada em *Assembly*.

Conceitos de mais alto nível

As linguagens de programação procuram proporcionar conceitos de mais alto nível e esconder conceitos de baixo nível. Assim, linguagens estruturadas fornecem estruturas de fluxo de instruções que eliminam os desvios incondicionais que Dijkstra denunciou como culpados por muitos erros de programação. No lugar de desvios, usam-se estruturas sequenciais, condicionais, de repetição e chamada de sub-programas (funções ou procedimentos).

Tipos de dados O hardware do computador praticamente só conhece valores binários (digitais). Estes não são facilmente interpretados por seres humanos. É mais compreensível entender os dados do mundo real com números decimais, reais e textos. As linguagens de programação fornecem representações binárias para estes tipos de dados livrando o programador das representações binárias na maior parte do tempo. Assim, as linguagens de programação oferecem tipos de dados inteiros, ponto flutuante para números reais, caracteres em algum tipo de codificação. Atualmente, usa-se muito a codificação ASCII e a UTF-8. Os textos são formados com sequências de caracteres.

Estruturas de controle de fluxo de instrução As instruções na maioria das linguagens de programação são supostas serem executadas **sequencialmente**. A sequência das instruções pode precisar seguir caminhos diferentes dependendo do valor de alguns dados. Isto é obtido com instruções condicionais. Em que o valor de uma condição determina o fluxo de instruções a ser seguido.

Frequentemente, uma sequência de instruções precisa ser repetida, para tanto, usa-se as **instruções de repetição**.

Muitas vezes, percebe-se que trechos de código realizam um tipo de processamento bastante específico e precisamos usar este tipo de processamento em diferentes programas e partes de programas. Este tipo de ideia sugere que existem subprogramas que são repetidamente usados nos programas. O conceito de subprograma foi expandido e a instrução que usa um subprograma é denominada de **chamada de subprograma**. Apesar de rodar o mesmo código em cada chamada de subprograma, podemos querer mudar alguns dados que são processados pelo subprograma. Estes dados que mudam de chamada para chamada de subprograma são chamados de **parâmetros** dos subprogramas. Os valores dos dados que são passados para os subprogramas são chamados de **argumentos**. Muitas vezes os termos parâmetros e argumentos são usados como sinônimos, apesar de não serem. Nesta disciplina vamos usá-los como sinônimos.

Os subprogramas podem ser divididos em 2 tipos diferentes: **funções** e **procedimentos**. As funções procuram calcular algo baseados nos valores dos argumentos na chamada. Elas sempre retornam um valor. Os procedimentos podem realizar tarefas diferentes, em geral, interações com os meios externos ou alterações dos estados de dados globais. Os procedimentos não retornam valores.

A chamada de um subprograma desvia o fluxo de instruções para o subprograma. O subprograma retorna ao programa (ou subprograma) que o chamou quando termina.

Subprogramas

A ideia de subprograma é bastante simples: Existem trechos de programas que precisamos escrever repetidamente. Para evitar estas reescritas que deixam o texto do programa muito grande. Sempre que estes trechos repetitivos executam uma tarefa muito bem definida, transformamos estes trechos em subprogramas. No lugar onde escreveríamos estes trechos de código, chamamos o subprograma que realiza a tarefa. Em linguagens antigas, existia uma instrução `CALL` para chamar os subprogramas. Em linguagens mais modernas, os subprogramas são chamados de *funções* quando retornam valores e *procedimentos* quando não retornam nenhum valor. A chamada dos subprogramas não precisam mais de um `CALL` explicito, basta escrever o nome da função, ou do procedimento e a lista de argumentos, quando houver, entre parênteses, `()`.

A definição do código da função ou do procedimento é chamado de *implementação* da função. O código do que a função deve executar é chamado de *corpo da função.*

% Aula 5 de Lógica de Programação % Paulino Ng % 2019-09-10

Utilização do `jupyter-notebook`

Estudo do texto disponível em https://github.com/pangpucsp/Curso_Python3.

Realização dos exercícios e atividades propostas em https://github.com/pangpucsp/Curso_Python3/blob/master/licao1.ipynb. Para baixar este texto sem baixar todo o repositório, é necessário usar um procedimento "complexo" de vários passos.

1. Acesse a página no *link* dado acima.
2. Clique no botão `Raw`. Você verá o texto do arquivo licao1.ipynb. Selecione todo o texto com `ctrl-a` e copie com `ctrl-c`.
3. Coloque o conteúdo no `notepad++` com:

a. Lance o `notepad++` com win-r (botão de Windows, segundo botão da esquerda para direita em baixo no teclado, acionado junto com a tecla r) e preencha a janela de diálogo com `notepad++` e dê OK.
b. Numa aba nova do `notepad++`, faça ctrl-v para colar o texto copiado.
c. Salve no diretório: `C:\Users\laba` com o nome `licao1.ipynb`.

Agora, o aluno deve ser capaz de rodar o *notebook* no Jupyter-notebook. Execute o programa *jupyter-notebook* a partir da janela de aplicações do Windows.

Observe que se a instalação está correta (igual a de todos os outros computadores do laboratório), o *jupyter-notebook* deve lançar uma janela de *Prompt do DOS*, com os logs da execução do programa servidor e criar uma *página* com a URL: `localhost:8888/tree`. Se for a primeira vez em que o *jupyter-notebook* é lançado, o MS Windows pode perguntar qual software deve ser associado com a extensão `.html`, escolha o *Google Chrome*. A página do servidor lista o conteúdo do diretório `C:\Users\laba`. O arquivo `licao1.ipynb` deve aparecer na listagem. Clique nele e vai ser criado um *kernel* para executar as instruções Python das células do notebook.

% Aula 6 de Lógica de Programação % Paulino Ng % 2019-09-17

Laboratório

Resolução dos exercícios com Python no Jupyter-notebook

Vamos primeiro executar o jupyter-notebook e usá-lo para executar algumas instruções de Python.

Para executar instruções Python no notebook, acione o botão New, no topo a direita na barra de ferramentas. Selecione a opção `Python 3` do menu. Você foi redirecinado para uma nova página com uma área de texto esperando pelo seu código Python. Esta área é chamada de *célula* do notebook.

Execute nas células do notebook

Coloque na célula, a expressão abaixo:

```
'Alo, Mamae!'
```

Execute a célula pelo teclado com Shift+Enter ou com o *mouse* clicando no botão Run na barra de ferramentas.

Ao executar uma célula, o notebook escreve o resultado do cálculo da última instrução se ela resulta num valor e abre uma nova célula para o usuário colocar/calcular novas instruções. A instrução que foi "executada", na verdade "calculou" o valor textual `Alo, Mamae!`. Muitas linguagens de programação não consideram uma instrução válida um *valor literal* solto como o dado. Mas o Python considera um valor como uma instrução válida (inútil, mas válida).

Experimente "calcular" outros valores como (uma linha em cada célula):

```
42
```

```
14 * 3
```

```
'Alo, ' + 'Mamae!'
```

Atribuição de valores a variáveis

Vamos colocar (atribuir) valores em variáveis usando o operador de atribuição `=`.

Coloque numa célula:

```
x = 42
```

Execute esta célula. Observe que diferente de quando "executamos" valores (cálculo de expressões), a execução da célula não resultou em nenhum valor. Isto porque em Python, a atribuição não gera um valor. Em muitas linguagens (C, C++, Java), a atribuição resulta no valor atribuído. Isto permite atribuições em cascata como: `a = b = c = 42`. Este tipo de atribuição não é possível em Python.

Para verificar que a variável `x` tem o valor `42`, vamos pedir para imprimir o valor da variável `x` com a função `print()`. Na célula nova, coloque:

```
print(x)
```

Execute e veja que o valor `42` é impresso. Isto porque o `print()` imprime o valor de seus argumentos.

Vamos trocar o valor de `x` por `'Alo, Mamae!'`.

```
x = 'Alo, Mamae!'
print(x)
```

Observe que o valor `42` da variável `x` foi substituído pelo texto `Alo, Mamae!`. Diferente de linguagens fortemente tipadas como o C, C++, Java, o Python não exige (na verdade, não permite) a declaração de variáveis com tipos e permite que o tipo dos valores guardados numa variável mude.

O Python também permite que *múltiplas atribuições* sejam realizadas numa mesma instrução:

```
x,y,z = 7,11,13
print('x =', x)
print('y =', y)
print('z =', z)
```

Diferença entre atribuições simultâneas e sequenciais

O Python é uma das raras linguagens que permitem atribuições simultâneas, estas atribuições simultâneas são diferentes das atribuições sequenciais. Por exemplo, sejam as atribuições simultâneas abaixo:

```
x,y = 1,2
x,y = y,x+y
print('x =', x, '\ty =', y)
```

Este código imprime: `x = 2  y = 3`

Se a segunda atribuição simultânea for realizada sequencialmente:

```
x,y = 1,2
x = y
y = x+y
print('x =', x, '\ty =', y)
```

Este código imprime: `x = 2  y = 4`

Se invertermos a ordem das atribuições de `x` e `y`, temos:

```
x,y = 1,2
y = x+y
x = y
print('x =', x, '\ty =', y)
```

Este código imprime: `x = 3  y = 3`

A diferença está em que quando a atribuição simultânea foi feita, o interpretador Python calculou os valores que estavam à direita da atribuição antes de realizar a atribuição. Isto é, calculou `y,x+y` e encontrou `2,3`. Aí, atribuiu `2` ao `x` e `3` ao `y`. Nos outros dois casos, os valores de `x` e `y` foram modificados antes das atribuição desejada. Este fenômeno é conhecido como *efeito colateral* da concorrência das atribuições. Ele é particularmente complexo quando existem acessos simultâneos a bancos de dados.

Cálculo do fatorial de n

Vamos resolver com Python o cálculo do fatorial de um número. Lembrando que:

$$n\,! = 1\,.\,2\,.\,\ldots\,.\,n$$

Numa primeira tentativa para o cálculo do fatorial, vamos fixar o valor de `n` para um valor cujo fatorial sabemos calcular. Por exemplo, sabemos que $5 ! = 120$.

O código abaixo usa uma técnica de acumulador para as multiplicações, o `1` é usado para inicializar o acumulador pois qualquer coisa multiplicada por 1 é a própria coisa. Isto é, o 1 é neutro na multiplicação. Vamos usar uma estrutura de repetição para fazer as sucessivas multiplicações. Multiplicamos sucessivamente o acumulador pela variável de controle que vai variar de 2 até n (5, neste protótipo/exemplo).

```
n = 5
acc = 1
i = 2
while i <= n:
    acc *= i
    i += 1
print('fatorial de ', n, 'eh', acc)
```

A instrução `acc *= i` é uma atribuição com operação, ela é o mesmo que `acc = acc * i`. Isto é, coloque no acumulador `acc`, o valor da multiplicação do conteúdo de `acc` com o conteúdo de `i`. Isto é, na primeira iteração, `1 * 2`, pois `acc` e `i` valem, respectivamente, `1` e `2`. Agora o `acc` tem o valor `2`. Na instrução seguinte, a variável de controle `i` recebe `2 + 1`. Como `3 <= 5` é verdadeiro, temos uma segunda iteração, no fim da qual, `acc` e `i` adquirem os valores `6` e `4`. Após a terceira iteração temos: `24` e `5`. Após a quarta iteração, `120` e `6`. Como `6 <= 5` é falso, o `while` termina. Aí, o `print()` é executado.

Função que calcula o fatorial Agora que sabemos como calcular o fatorial de n, para podermos reutilizar o código para uso futuro, vamos transformar o código numa função e usá-lo:

```
def fatorial(n):
  acc = 1
  i = 2
  while i <= n:
    acc *= i
    i += 1
  return acc

print('O fatorial de 20 eh', fatorial(20))
```

Cálculo do n-ésimo elemento da sequência de Fibonacci

A sequência de Fibonacci é dada por: `0, 1, 1, 2, 3, 5, 8, 13, ...` Os números da sequência são calculados de forma que um elemento da sequência é sempre a soma dos 2 anteriores.

Estes números aparecem em muitos fenômenos naturais e em muitas ocasiões na

computação. O primeiro número da sequência é `0`, o segundo é `1` e a partir daí, sempre calculamos o próximo como a soma dos 2 anteriores. O algoritmo para calcular o n-ésimo elemento da sequência consiste em calcular sequencialmente os anteriores até termos os 2 anteriores ao n-ésimo e, aí, somamos estes 2 e obtemos o n-ésimo. Assim, informalmente, o algoritmo pode ser descrito por:

1. Se o `n` é `1` ou `2`, o n-ésimo elemento da sequência é, `0` ou `1`, respectivamente.
2. Usaremos uma variável para memorizar o valor da sequência duas posições atrás, vamos chamá-la de `fMenos2` e outra variável para uma posição atrás, `fMenos1`. Inicializamos `fMenos2` com 0 e `fMenos1` com 1.
3. Usaremos uma variável para indicar que já calculamos o i-ésimo elemento da sequência, o nome desta variável é `i`. Ela é inicializada com 2.
4. Enquanto `i < n`, calculamos a soma de `fMenos2` com `fMenos1` e colocamos em `soma`, este é o novo valor da sequência, quer dizer que podemos aumentar o `i` de 1 e agora `fMenos2` recebe o valor de `fMenos1` e `fMenos1` recebe o valor de soma. Quando este `enquanto` acabar de repetir estas atualizações, teremos o `i` igual ao `n` e o valor em `soma` ou `fMenos1` é o valor que procuramos.

Como fizemos com o fatorial, vamos primeiro escrever o programa para um caso especial, n = 8, que sabemos ser 13.

```
n = 8
if n == 1:
  print("O primeiro elemento da sequencia eh 0")
elif n == 2:
  print("O segundo elemento da sequencia eh 1")
else:
  fMenos2 = 0
  fMenos1 = 1
  i = 2
  while i < n:
    soma = fMenos1 + fMenos2
    i += 1
    fMenos2 = fMenos1
    fMenos1 = soma
  print('O ' + str(i) + '-esimo elemento da sequência de Fibonacci eh', soma)
```

Observe que a condição de continuidade é `i < n` e não `i <= n`. Por que?

> Quando a malha de repetição vai terminar? Quando o `i` não for mais menor do que `n`. A primeira vez em que isto ocorre é quando `i == n`. Logo a condição de continuidade é `i < n`, se inclui-se a igualdade, a malha de repetição continuaria até `i == n + 1`.

Exercício: Converta o programa para uma função que calcula o n-ésimo elemento da sequência de Fibonacci e chame a função para calcular o centésimo elemento.

Solução: Programa que imprime o valor do centésimo elemento da sequência de Fibonacci

Cálculo da string reversa

Problema: dado um texto, calcular o texto invertido, o primeiro carácter é o último, o segundo o penúltimo, ...

No Python, os caracteres de uma *string* podem ser acessados por índice. O primeiro carácter tem índice `0`. Além disso, um carácter em Python é uma *string* com um único carácter. Um algoritmo simples para reverter uma *string* é:

1. Usar uma variável para acumular os caracteres na ordem contrária à que esta sendo lido o texto. A variável começa com a *string* vazia.
2. Para cada carácter no texto, concatená-la como primeiro carácter ao acumulador. Isto é, o último carácter lido é o primeiro carácter no acumulador.

Em Python, este algoritmo pode ser escrito como:

```
texto = 'Texto a ser invertido'
otxet = ""
for ch in texto:
  otxet = ch + otxet
print('"' + texto + '" invertido fica:', otxet)
```

Exercício: Converta o programa em uma função. Que recebe o texto como argumento e retorna o texto invertido.

Solução: Programa com a função pedida