Tec. Esp. Tec. Prog. SI. / Téc. Esp. em Gestão de Redes e Sistemas Informáticos / Téc. Esp. em Cibersegurança

Introdução à Programação

UFCD(s): 5117, 5091, 5089, 804, 805, etc.

Computadores e Linguagens

Computadores e Programas…

- Recapitulando: O que faz um computador?
  - Cálculos
  - Guarda ou transfere os resultados (dos cálculos)
- "Computação": Conjunto de operações para obter, representar e armazenar um resultado
- Então, um computador leva a cabo … computações. Muitas, imensas, num curtíssimo intervalo de tempo

- Instruções: operações ou comandos reconhecidos pelo computador
- Que instruções são válidas?
  - Aquelas que estão pré-definidas para um determinado computador
  - As que pertencem ao "catálogo" de instruções que o computador reconhece, ou seja, ao Código Máquina do CPU
  - **Código Máquina**:
    - Conjunto de instruções reconhecidas directamente por um determinado tipo de CPU
    - Tipicamente, varia de CPU para CPU
    - Cada instrução tem uma representação em binário (como seria de esperar…)
- **Programa**: sequência de instruções em Código Máquina

Código Máquina e *Assembly*…

- Como tudo num computador digital, instruções são cadeias de bits
- Um programa é uma sequência de cadeias de bits de 0 e 1
- Cadeias de bits que não são arbitrárias; constam do catálogo, ie, do Código Máquina
- Problema: É difícil memorizar e ler cadeias de bits

```
11000010
00001010
10000000
00000100
10000000
01010111
00001001
10000000
11001010
10000000
00010110
11111111
```

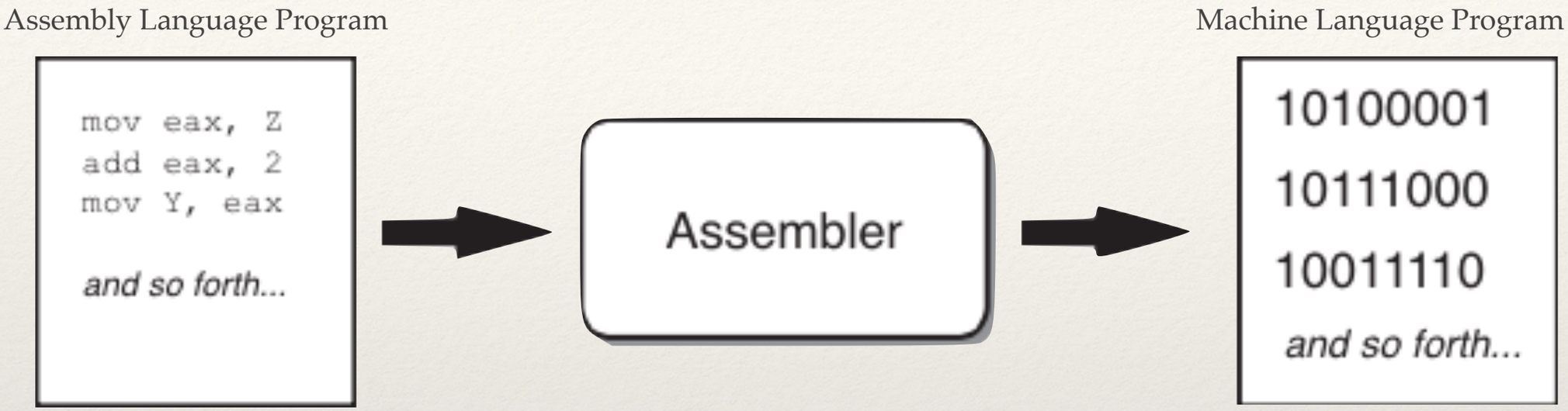
O que faz este programa? …

Ideia: porque não representar as instruções binárias em texto e desenvolver um programa em Código Máquina para traduzir cada instrução textual para as respectivas cadeias de bits?

- *Assembly:*
  - Linguagem de programação de baixo nível, ie, lida com aspectos particulares de uma determinada arquitectura: registo, modos de endereçamento, etc
  - Conjunto de instruções textuais designadas de mnemónicas, a maioria das quais com tradução directa para código máquina
  - Maioria das instruções Assembly tem tradução directa para código máquina
  - Comparativamente com outras linguagens, é fácil fazer um tradutor de Assembly para código máquina

```
Capitalize:
    MOV A, C      ; C = number of characters left
    CPI A, 00h    ; compare with 0
    JZ AllDone    ; if C is 0, we're finished

    MOV A, [HL]   ; get the next character
    CPI A, 61h    ; check if it's less than 'a'
    JC SkipIt     ; if so, ignore it

    CPI A, 78h    ; check if it's greater than 'z'
    JC SkipIt     ; if so, ignore it

    SBI A, 20h    ; it's lowercase, so subtract 20h
    MOV [HL], A   ; store the character

SkipIt:
    INX HL            ; increment the text address
    DCR C             ; decrement the counter
    JMP Capitalize    ; go back to the top

AllDone:
    RET
```

Programa em Assembly do CPU 8080 para transformar letras minúsculas em maiúsculas



Programas em Assembly não podem ser executados directamente pelo CPU
Um Assembler traduz de Assembly para Código Máquina

- Problemas ao escrever programas em Assembly:
  - Continua a ser uma linguagem específica de um tipo de CPU => programas não são portáveis para outras arquitecturas!
  - Implica conhecer muito bem cada arquitectura
  - Instruções demasiado primitivas:
    - Lidam com as "necessidades" do computador: mexer bits de um lado para o outro, somar bits, enviar dados (ie,…bits) para periférico
    - Não permitem abstração
  - Fazem com que sejam necessárias muitas instruções para escrever programas básicos
  - Código difícil de manter
  - Optimizada para ser executada por um computador e "pessimizada" para ser lida por um humano

Linguagens de Programação

Ideia: Porque não criar uma linguagem menos dependente do CPU e desenvolver um programa para, de alguma forma, transformar essa linguagem em Assembly?

- Linguagens de Alto-Nível:
  - Independentes do CPU
  - Instruções menos primitivas, mais complexas (neste caso o que é bom)
  - Permitem maior abstração
  - Podem ser desenvolvidas para determinadas propósitos: gráficos, bases de dados, finanças, inteligência artificial, etc.
  - Ou podem ser de propósito geral
  - Optimizadas para serem lidas entendidas por um humano, mas são difíceis de serem executadas directamente por um computador (o hardware seria tremendamente complexo)

Exemplos

```python
def primos_entre_si(a, b):
    for n in range(2, min(a, b) + 1):
        if a % n == b % n == 0:
            return False
    return True

print(primos_entre_si(9, 3))
print(primos_entre_si(9, 5))
print(primos_entre_si(10, 4))
```

```c
#include <stdio.h>

int primos_entre_si(int a, int b) {
    int min = a < b ? a : b;
    int n;
    for (n = 2; n <= min; n++)
        if (a % n == 0 && b % n == 0)
            return 0;
    return 1;
}

int main() {
     printf("%d\n", primos_entre_si(9, 3));
     printf("%d\n", primos_entre_si(9, 5));
     printf("%d\n", primos_entre_si(10, 4));
     return 0;
}
```

Programas escritos em Python e C para calcular se dois números são primos(*) entre si

*(*) Da Wikipedia: "Chamamos números **primos** entre si (ou **coprimos**) ao conjunto de números onde o único divisor comum a todos eles é o número 1."*

```
        .section __TEXT,__text,regular,pure_instructions
        .globl   _primos_entre_si
        .align   4, 0x90
_primos_entre_si:
        .cfi_startproc
## BB#0:
        pushq    %rbp
Ltmp2:
        .cfi_def_cfa_offset 16
Ltmp3:
        .cfi_offset %rbp, -16
        movq     %rsp, %rbp
Ltmp4:
        .cfi_def_cfa_register %rbp
        movl     %edi, -8(%rbp)
        movl     %esi, -12(%rbp)
        movl     -8(%rbp), %esi
        cmpl     -12(%rbp), %esi
        jge      LBB0_2
## BB#1:
        movl     -8(%rbp), %eax
        movl     %eax, -24(%rbp)
        jmp      LBB0_3
LBB0_2:
        movl     -12(%rbp), %eax
        movl     %eax, -24(%rbp)
LBB0_3:
        movl     -24(%rbp), %eax
        movl     %eax, -16(%rbp)
        movl     $2, -20(%rbp)
LBB0_4:                                 =1
        movl     -20(%rbp), %eax
        cmpl     -16(%rbp), %eax
        jg       LBB0_10
## BB#5:
        movl     -8(%rbp), %eax
        cltd
        idivl    -20(%rbp)
        cmpl     $0, %edx
        jne      LBB0_8
## BB#6:
        movl     -12(%rbp), %eax
        cltd
        idivl    -20(%rbp)
        cmpl     $0, %edx
        jne      LBB0_8
## BB#7:
        movl     $0, -4(%rbp)
        jmp      LBB0_11
LBB0_8:
        jmp      LBB0_9
LBB0_9:
        movl     -20(%rbp), %eax
        addl     $1, %eax
        movl     %eax, -20(%rbp)
        jmp      LBB0_4
```

```
LBB0_10:
        movl     $1, -4(%rbp)
LBB0_11:
        movl     -4(%rbp), %eax
        popq     %rbp
        retq
        .cfi_endproc

        .globl   _main
        .align   4, 0x90
_main:
        .cfi_startproc
## BB#0:
        pushq    %rbp
Ltmp7:
        .cfi_def_cfa_offset 16
Ltmp8:
        .cfi_offset %rbp, -16
        movq     %rsp, %rbp
Ltmp9:
        .cfi_def_cfa_register %rbp
        subq     $16, %rsp
        movl     $9, %edi
        movl     $3, %esi
        movl     $0, -4(%rbp)
        callq    _primos_entre_si
        leaq     L_.str(%rip), %rdi
        movl     %eax, %esi
        movb     $0, %al
        callq    _printf
        movl     $9, %edi
        movl     $5, %esi
        movl     %eax, -8(%rbp)
        callq    _primos_entre_si
        leaq     L_.str(%rip), %rdi
        movl     %eax, %esi
        movb     $0, %al
        callq    _printf
        movl     $10, %edi
        movl     $4, %esi
        movl     %eax, -12(%rbp)
        callq    _primos_entre_si
        leaq     L_.str(%rip), %rdi
        movl     %eax, %esi
        movb     $0, %al
        callq    _printf
        movl     $0, %esi
        movl     %eax, -16(%rbp)
        movl     %esi, %eax
        addq     $16, %rsp
        popq     %rbp
        retq
        .cfi_endproc

        .section __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz   "%d\n"
```

Programa escrito em Assembly x86-64 OS X para calcular se dois números são primos entre si

- Linguagens de Programação:
  - São linguagens formais:
    - Desenhadas para aplicações específicas, tal como a linguagem matemática, sinalização automóvel, código morse, etc.
    - Semântica bem definida; não são ambíguas
      - Cada elemento tem um significado preciso, independentemente do contexto
    - Regras sintáticas rígidas e propositadamente limitadas
  - Apropriadas para expressar computações
  - NOTA: estas propriedades são comuns a linguagens de baixo-nível e de alto-nível. Código Máquina é uma linguagem formal.

- Linguagens Naturais:
  - O que designamos por *Línguas* - Português, Inglês, etc.
  - Não foram desenhadas; pelo contrário, resultam de um processo de evolução, baseado em tradições, aspectos culturais, necessidades, etc.
  - Mais ricas que as linguagens formais, mas deixam espaço a ambiguidades e subtilezas
  - Ambiguidades são muitas vezes resolvidas através do contexto e através de redundâncias

Interpretador

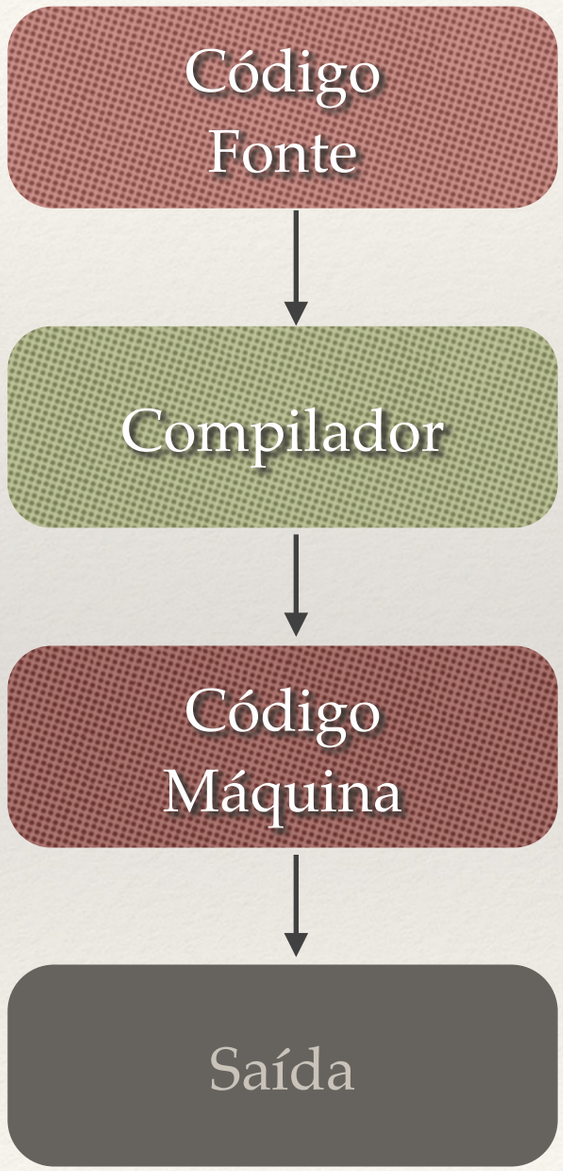
- Tipicamente, linguagens de programação de alto nível são processadas por um de dois tipos de programas: interpretadores e compiladores
- Interpretador: lê, interpreta e executa uma instrução de cada vez
- Dois programas a correr em simultâneo: o interpretador e o programa interpretado

- Vantagens da interpretação de programas (vs. Compilação):
  - Dependendo da linguagem, informação sobre erros tende a ser mais precisa
  - Carregamento é mais rápido na maioria das vezes porque programa não precisa de ser compilado (ver à frente)
  - Desenvolvimento mais interactivo e introspectivo: muita informação sobre o código fonte do programa está disponível para averiguar o estado do programa e para alterá-lo se necessário
  - Permitem tornar o programa mais flexível porque muitas decisões podem ser tomadas durante a execução do mesmo (quando existe mais informação)

- Desvantagens da interpretação de programas (vs. Compilação):
  - Programas tendem a ser mais lentos (pelos menos dois programas estão a correr: o programa em questão e o interpretador)
  - Instalar o programa implica instalar o interpretador
  - Em geral, há menos oportunidades de optimizar o programa
  - Isto pode levar a que programas sejam menos eficientes na utilização da memória

Compilador

- Traduz as linhas de código da linguagem de alto nível em questão para Assembly/Código Máquina
- O código máquina obtido após compilação é por vezes designado de código objecto. O ficheiro com o código objecto é designado de ficheiro objecto.
- É também designado de executável se estiver preparado para o sistema operativo colocar o programa "a correr"



- Vantagens da compilação de programas (vs. Interpretação):
  - Programas ***tendem*** a ser mais rápidos: não há interpretador a correr e o compilador pode introduzir várias optimizações
  - Programas ***tendem*** a ocupar menos memória
  - A instalação do programa ***tende*** a ser mais simples

- <u>Desvantagens</u> da compilação de programas (vs. Interpretação):
  - Informação sobre erros tende a ser mais difícil de seguir porque já se "perdeu o rasto" ao código fonte.
  - É necessário um passo extra: compilar o programa
  - Menos introspectivo
  - Mais rígido

Ciclo de Vida de um Programa

- Interessa-nos classificar as várias fases na vida de um programa (*lifecycle*):
  - Porque interessa-nos saber que funcionalidades e mecanismos estão disponíveis em cada uma destas fases
- Fase de Compilação / *Compile Time*:
  - Quando o programa está a ser compilado
- Fase de Execução / *Run Time*:
  - Quando o programa está a correr
  - Outras designações: Momento de execução, Tempo de execução, *Runtime, Execution Time*

- Fase de Carregamento / *Load Time*:
  - Quando programa compilado é carregado para memória
  - Após compilação e antes da execução
- Fase de Leitura / *Read Time*:
  - Quando o código-fonte é lido para ser compilado
- Fase de Ligação ("Linkagem") / *Link Time*:
  - Quando os vários blocos/unidades compilados do programa são ligados entre si
  - Quando todo o programa compilado é ligado a bibliotecas (ver à frente)

Linguagens "Interpretadas"

- Linguagens processadas por um interpretador
- Determinadas linguagens ***tendem*** a ser interpretadas ainda que possam ser compiladas:
  - Exemplos: Python, JavaScript, Ruby, PHP, Perl, etc.
- Algumas destas linguagens podem ser compiladas para o código máquina de uma máquina virtual
  - É este código máquina que depois é interpretado
  - Exemplos: Python, Perl, PHP

- Características habituais:
  - Mais dinâmicas e flexíveis
  - Mas, mais lentas e ineficientes
  - Tendem a ser menos robustas (mas isto é discutível…)
  - Mas tendem a ser mais produtivas para os programadores (outra vez, isto é discutível…)

Linguagens "Compiladas"

- Linguagens processadas por um compilador
- Determinadas linguagens ***tendem*** a ser compiladas ainda que possam ser interpretadas:
  - Exemplos: C, C++, C#, Objective-C, Java, Fortran
- Características habituais:
  - Mais rápidas e eficientes
  - Tendem a ser mais robustas

- Algumas destas linguagens podem ser compiladas para o código máquina de uma máquina virtual:
  - Este código máquina é depois compilado pela máquina virtual para o código máquina da arquitectura onde o programa vai correr
  - Porque esta 2a compilação ocorre quando o programa é executado, estes compiladores tendem a ser designados de **JIT** (*Just In Time compilers)*
    - Exs: Java → Bytecodes/JVM → ASM, C# → CIL/CLI+CLR → ASM
- Algumas destas linguagens são compiladas para outras linguagens de alto-nível.
  - Exemplos: os primeiros compiladores de C++ compilavam para C, CoffeeScript é uma linguagem que é compilada para JavaScript
- Outras são compiladas na primeira execução do programa, de forma automática, o que faz com que pareçam ser "interpretadas" (alguns Lisps)

Linguagens de *Scripting*

- *Script:*
  - Programa que corre num contexto ou ambiente especial:
    - No *browser*: JavaScript, C# Script
    - No contexto de uma linha de comandos do sistema operativo: Bash, Perl, PowerShell
    - Como extensão de um jogo de computador: Scheme com Gimp, Python com Blender
    - Como extensão de um jogo de computador: Python e Lua com uma série de jogos
    - Como extensão de um programa produtividade: VBA com MS Office, VBA/JavaScript/Python com OpenOffice/LibreOffice

- *Script* (continuação):
  - Tende a ser um programa curto, com propósito muito específico
  - Tende a ser um programa fácil de alterar, modificar e "arrancar"
  - Tende a automatizar tarefas que seriam desempenhadas uma-por-uma por um humano
  - Não tem preocupações de eficiência computacional; é mais importante a eficiência dos programadores e dos utilizadores
  - Mas antes de mais um *script* é um programa (mais uma vez, é um conceito algo vago e talvez artificial)

- Linguagem de *Scripting* em geral é:
  - "Interpretada": não queremos perder tempo em compilações
  - Dinâmica: toda a informação sobre o próprio programa está disponível quando este está a correr
  - Flexível: sem regras muito rígidas, o que permite que rapidamente se faça uma versão funcional do *script*

Linguagens Dinâmicas

- Permitem que muitas operações sobre o código estejam disponíveis na fase de execução (em *runtime*)
- São introspectivas (ou, melhor dizendo, permitem introspecção):
  - O programa pode "olhar" para si próprio e tomar decisões em função do "estado em que se encontra"
  - Quase todos os elementos da linguagem (variáveis, funções, classes, módulos, etc.) possuem uma representação "utilizável" em tempo de execução (*first class*).
- Permitem que o programa se modifique a si próprio
- Em linguagem dinâmicas compiladas, o próprio compilador está disponível na fase de execução

- Tendem a suportar tipificação dinâmica:
  - Não é obrigatório classificar a informação com tipos de dados
    - Tipo de Dados / *Data Type* => universo de valores + conjunto de operações disponíveis para esses valores + representação em memória
    - Exemplos: números inteiros, texto, valores lógicos/booleanos, números reais, números complexos, contas bancárias, matrículas de automóveis
  - O tipo de dados reside nos valores (e permanece em *runtime*) e não nos identificadores através dos quais acedemos a esses valores
  - A informação sobre o tipo de dados dos objectos é validada em tempo de execução

- Linguagens de *scripting* tendem a ser dinâmicas
- Algumas linguagens dinâmicas oferecem alguns dos mecanismos que encontramos nas linguagens estáticas:
  - Exemplo: tipificação opcional dos identificadores
- Permitem tomar decisões em tempo de execução em função do tipo de dados dos identificadores
- Tendem a ser mais flexíveis do que linguagens estáticas porque são feitos menos compromissos
- Mas também por isto, tendem a ser mais ineficientes e a ocupar mais memória
- Exemplos: Lisp, Python, Perl, Ruby, PHP, Smalltalk

Linguagens Estáticas

- O programador deve informar o compilador/interpretador sobre os tipos de dados de cada um dos identificadores da linguagem
- Selecção das operações a aplicar é feita em tempo de compilação, em função dos tipos de dados dos identificadores/objectos da linguagem
- Tendem a ser linguagens compiladas.
- Programas tendem a ser mais robustos porque:
  - O compilador verifica essa compatibilidade durante a fase de compilação
  - Logo, ***tendem*** a ocorrer menos erros na fase de execução
- Mas isto torna os programas menos flexíveis perante alterações porque obrigam os programadores a ter que tomar decisões mais cedo do que desejariam (veremos exemplos disto mais à frente)

- Determinadas linguagens estáticas suportam algum dinamismo
  - C++ suporta alguma introspecção sobre os tipos de dados e tomadas de decisão em tempo de execução com RTTI
  - Java disponibiliza (entre outras coisas) a palavra-reservada **`instanceof`** que permite averiguar o tipo de dados de um determinado objecto
  - C++ e Java permitem selecção das operações em tempo de execução baseadas nos tipos de determinado objectos (mas apenas para objectos dentro de uma hierarquia)
  - Tipificação dinâmica: em C#, a palavra-reservada **`dynamic`** permite indicar que um determinado identificador não vai estar sujeito às regras da tipificação estática
- Exemplos: C++, Java, Eiffel, C#, ML, F#

Referências

[1] Guttag, John. *" Introduction to Computation and Programming Using Python - Revised and Expanded Edition"*, MIT Press, 2013, Cap. 1

[2] Bjarne Stroustrup, *"Programming: Principles and Practice Using CPP 2nd Edition"*, Addison-Wesley, 2014

[3] Carlos Rafael, Apontamentos sobre Algoritmos