

Primeiro Exercício-Programa (EP1)

Prazo de entrega: até às 08:00 de 09/09/2024

(Atenção: o exercício deve ser realizado individualmente)




1. Introdução: Codificação de Caracteres

(A maior parte do conteúdo desta seção foi extraído do material do Prof. Paulo Feofiloff ¹.)

1.1 Caracteres

Um caractere é um símbolo tipográfico usado para escrever texto em alguma língua. Embora imperfeita, essa definição é suficiente para nossas necessidades. O número de caracteres usados pelas diferentes línguas do mundo é muito grande. A isso se somam os caracteres especiais usados por várias áreas da ciência.

A fim de organizar a representação dessa enorme variedade de caracteres, o consórcio Unicode de empresas de informática atribuiu nomes numéricos (conhecidos como **code points**) a mais de 1 milhão de caracteres. Uma minúscula amostra da lista de caracteres e seus *code points* é mostrada abaixo². Em geral, esses *code points* são escritos em notação hexadecimal. Além disso, é usual acrescentar o prefixo U+ a cada número:

Code Point		Code Point	
Unicode	Caractere	Unicode	Caractere
U+0021	!	U+00FF	ÿ
U+0022	"	U+03A3	Σ
U+0039	'	U+03B1	α
U+0041	A	U+2014	—
U+0042	B	U+770C	県
U+0061	a	U+1F33F	
U+007E	~	U+1F353	
U+00E7	ç	U+1F60E	

O conjunto de todos os caracteres da lista Unicode pode ser chamado alfabeto Unicode e cada caractere desse alfabeto pode ser chamado caractere Unicode.

Os primeiros 128 caracteres da lista Unicode são os mais usados. Esse conjunto de caracteres vai de U+0000 a U+007F e é conhecido como alfabeto ASCII. O alfabeto ASCII contém letras, dígitos decimais, sinais de pontuação, e alguns caracteres especiais. A lista dos 128 caracteres ASCII e seus *code points* Unicode está registrada na Tabela ASCII³.

¹ Unicode e UTF-8: <https://www.ime.usp.br/~pf/algoritmos/apend/unicode.html>

² A lista completa de caracteres e seus números Unicode pode ser vista em: https://en.wikipedia.org/wiki/List_of_Unicode_characters.

³ Tabela ASCII: <https://www.ime.usp.br/~pf/algoritmos/apend/ascii.html>

1.2 Esquemas de Codificação

Para armazenar os caracteres Unicode em arquivos digitais e na memória, poderíamos representar cada caractere pelo seu *code point* Unicode escrito em notação binária. Mas isso exigiria 3 bytes por caractere, o que é muito ineficiente dado que apenas 1 byte é suficiente para os caracteres mais comuns. É preciso recorrer, então, a representações mais complexas.

Um esquema de codificação (*character encoding*) é uma tabela que associa uma sequência de bytes com cada *code point* Unicode, e portanto com cada caractere Unicode. A sequência de bytes associada com um caractere é a codificação do caractere.

A próxima seção examina o esquema de codificação UTF-8.

1.3 Codificação UTF-8

Como já dito anteriormente, se usássemos um número fixo de bytes por caractere Unicode, precisaríamos de 3 bytes para cada caractere, o que seria ineficiente. A solução é recorrer a um código multibyte, que emprega um número variável de bytes por caractere: alguns caracteres usam 1 byte, outros usam 2 bytes, e assim por diante.

O código multibyte mais usado é conhecido como UTF-8. Ele associa uma sequência de 1 a 4 bytes (8 a 32 bits) com cada caractere Unicode. Os primeiros 128 caracteres usam o velho e bom código ASCII de 1 byte por caractere. Os demais caracteres têm um *code point* mais longo. Veja uma amostra⁴:

Code Point		Sequência de Bytes	Hexa da
Unicode		da Codificação UTF-8	Codificação
U+0021	!	00100001	0x21
U+0041	A	01000001	0x41
U+00E7	ç	11000011 10100111	0xC3A7
U+03A3	Σ	11001110 10100011	0xCEA3
U+2014	—	11100010 10000000 10010100	0xE28094
U+201C	“	11100010 10000000 10011100	0xE2809C
U+1F353	🍓	11110000 10011111 10001100 10111111	0xF09F8CBF
U+1F60E	😋	11110000 10011111 10011000 10001110	0xF09F988E

Em cada linha da tabela acima, os bits destacados em azul são os que formam o *code point* (em binário) do caractere. Por exemplo, para o caractere ‘ç’, a codificação UTF-8 é 11000011 10100111b (= 0xC3A7), mas os bits que formam o *code point* do caractere são apenas os destacados em azul (00011100111b = 0xE7).

Decodificação de UTF-8

Como o número de bytes por caractere não é fixo, a decodificação de uma sequência de bytes não é fácil. Como saber onde termina o *code point* de um caractere e começa o *code point* do caractere seguinte? O esquema de codificação UTF-8 foi construído de modo que os primeiros bits da codificação do caractere dizem quantos bytes o *code point* ocupa.

Observe as sequências de bytes que codificam os exemplos de *code points* na tabela mostrada acima. Quando o primeiro bit do primeiro byte da sequência é 0, então esse é o

⁴ A lista das codificações UTF-8 de todos os caracteres Unicode pode ser vista em: <https://www.utf8-chartable.de/>

único byte da codificação do caractere. Quando a codificação do caractere tem 2 bytes, os três bits mais significativos dela são 110; quando tem 3 bytes, os quatro bits mais significativos são 1110; e quando tem 4 bytes, os cinco bits mais significativos são 11110. Com exceção do primeiro byte de uma codificação, os demais bytes têm 10 nos dois bits mais significativos, para indicar que o byte é uma continuação da codificação do caractere.

A tabela abaixo mostra o esquema geral da codificação UTF-8. As posições indicadas com **b** nos bytes de uma codificação indicam os bits que compõem o valor do *code point*.

Faixa de Code Points	Byte 1	Byte 2	Byte 3	Byte 4	Número de Bits do Code Point
U+0000 - U+007F	0bbbbbbb				7 bits
U+0080 - U+07FF	110bbbb	10bbbbbb			5+6 = 11 bits
U+0800 - U+FFFF	1110bbbb	10bbbbbb	10bbbbbb		4+6+6 = 16 bits
U+010000 - U+10FFFF	11110bbb	10bbbbbb	10bbbbbb	10bbbbbb	3+6+6+6 = 21 bits

2. Descrição do Exercício-Programa

2.1 O Que o Programa Deve Fazer

Implemente um programa em linguagem de montagem (NASM para Linux 64 bits) que leia **byte a byte** um arquivo texto cujos caracteres são codificados em UTF-8 e obtenha os **code points** armazenados nele. Cada *code point* formado a partir de bits extraídos de um byte ou de uma sequência de bytes lidos deve ser convertido em uma *string* com a representação em hexadecimal do *code point*.

Por exemplo, o caractere '😎' é armazenado como a sequência de bytes 11110000 10011111 10011000 10001110, portanto o seu *code point* é 11111011000001110b. Logo, sua *string* da representação hexadecimal deve ser '0x1F60E'. Cada *string* de *code point* gerada deve ser gravada como uma linha no arquivo texto de saída do programa.

Considere, como exemplo, o arquivo **texto1.txt** (disponibilizado junto com o enunciado do EP). Esse arquivo contém apenas uma linha com emoticons e caracteres de diferentes línguas finalizada com o caractere de quebra de linha ‘\n’ (código 0xA), como mostrado a seguir:

@県丁にÇ暑گۆلتهşж□

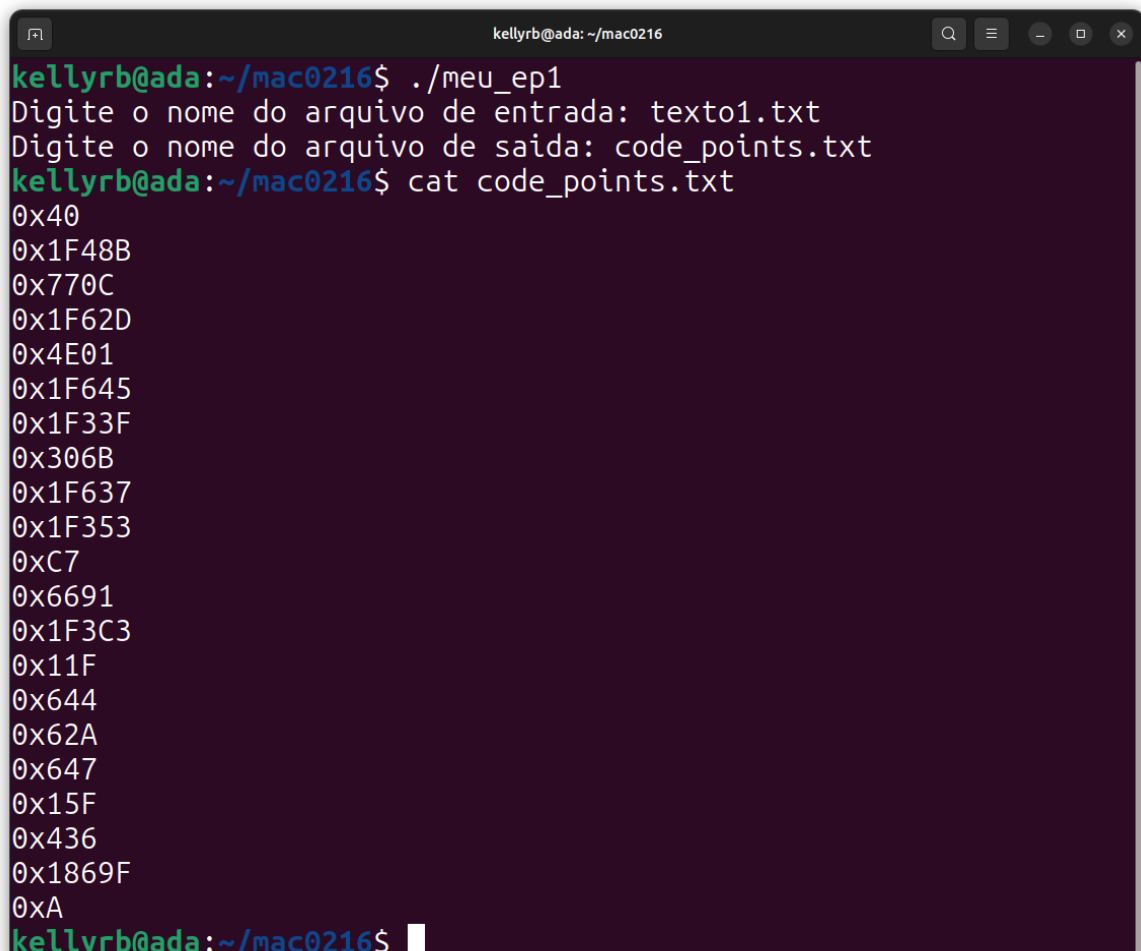
Veja abaixo como fica a impressão do arquivo na saída padrão usando o comando **cat** do Linux:

```
kellyrb@ada: ~/mac0216
kellyrb@ada:~/mac0216$ cat texto1.txt
@🍷県😭丁👧🌿に👤🍓Ç暑👉ğđł ş*甌
kellyrb@ada:~/mac0216$
```

O conteúdo desse arquivo em bits é o mostrado a seguir. Na exibição abaixo, foram incluídos espaços para separar bytes e '|' para separar as sequências de bytes que codificam *code points* diferentes, apenas para facilitar a visualização da codificação UTF-8 de cada *code point* (essas separações não existem no arquivo lido). Observe que o último byte do arquivo é justamente o caractere de quebra de linha '\n' (00001010b = 0xA).

```
01000000 | 11110000 10011111 10010010 10001011 | 11100111 10011100 10001100 |  
11110000 10011111 10011000 10101101 | 11100100 10111000 10000001 | 11110000  
10011111 10011001 10000101 | 11110000 10011111 10001100 10111111 | 11100011  
10000001 10101011 | 11110000 10011111 10011000 10110111 | 11110000 10011111  
10001101 10010011 | 11000011 10000111 | 11100110 10011010 10010001 | 11110000  
10011111 10001111 10000011 | 11000100 10011111 | 11011001 10000100 | 11011000  
10101010 | 11011001 10000111 | 11000101 10011111 | 11010000 10110110 |  
11110000 10011000 10011010 10011111 | 00001010
```

O seu programa deve ler da entrada padrão o nome do arquivo de entrada e também o nome do arquivo de saída. A figura abaixo mostra uma execução do programa, exemplificando como deve ser a interação dele com o usuário. Ela também mostra como deve ser o conteúdo do arquivo de saída gerado pelo programa com os *code points* em hexadecimal correspondente ao arquivo de entrada **texto1.txt**.

A terminal window with a dark purple background and white text. The window title is 'kellyrb@ada: ~/mac0216'. The user enters './meu_ep1'. The program prompts 'Digite o nome do arquivo de entrada: texto1.txt' and 'Digite o nome do arquivo de saída: code_points.txt'. The user then enters 'cat code_points.txt'. The program outputs a list of hexadecimal values: 0x40, 0x1F48B, 0x770C, 0x1F62D, 0x4E01, 0x1F645, 0x1F33F, 0x306B, 0x1F637, 0x1F353, 0xC7, 0x6691, 0x1F3C3, 0x11F, 0x644, 0x62A, 0x647, 0x15F, 0x436, 0x1869F, and 0xA. The prompt 'kellyrb@ada: ~/mac0216\$' is visible at the bottom with a cursor.

```
kellyrb@ada:~/mac0216$ ./meu_ep1
Digite o nome do arquivo de entrada: texto1.txt
Digite o nome do arquivo de saída: code_points.txt
kellyrb@ada:~/mac0216$ cat code_points.txt
0x40
0x1F48B
0x770C
0x1F62D
0x4E01
0x1F645
0x1F33F
0x306B
0x1F637
0x1F353
0xC7
0x6691
0x1F3C3
0x11F
0x644
0x62A
0x647
0x15F
0x436
0x1869F
0xA
kellyrb@ada:~/mac0216$
```

Além do arquivo texto1.txt, foram disponibilizados com o enunciado do EP mais dois arquivos UTF-8 para você testar seu programa: os arquivos texto2.txt e texto3.txt. Ambos contêm mensagens em várias línguas postadas em uma rede social (cada linha do arquivo é uma mensagem diferente). Os arquivos com os respectivos code points em hexadecimal para eles também foram disponibilizados, para que você possa conferir se a saída do seu programa está como a esperada.

Vale mencionar que o arquivo texto3.txt contém ~700KB. Não estranhe se a execução do programa usando esse arquivo como entrada demorar alguns (poucos) segundos.

2.2 Como o Programa Deve ser Feito

No seu programa, você deve (obrigatoriamente) implementar e usar todas as funções descritas a seguir.

1) `le_string(char* buffer, int tam_max)`

Lê da entrada padrão (STDIN) uma sequência de caracteres finalizada por ENTER (caracter 0xA) e armazena-a na memória, finalizando com '\0' (caractere 0x0). Usa a `sys_read`.

ENTRADAS:

- `char* buffer`: endereço inicial do espaço de memória onde a função armazenará a string lida.
- `int tam_max`: a quantidade máxima de caracteres a serem lidos. Usado para evitar 'estouro' do buffer caso o usuário digite mais caracteres do que o espaço disponível para armazenamento.

SAÍDA:

- Devolve no registrador RAX a quantidade de caracteres lidos.

2) `escreve_string(char* buffer)`

Escreva uma string na saída padrão (STDOUT). A função supõe que a string é finalizada com '\0' (código 0x0). Usa a `sys_write`.

ENTRADA:

- `char* buffer`: ponteiro para a string (ou seja, o endereço da sua posição de memória inicial).

3) `abre_arquivo(char* nome_arquivo, int modo_abertura)`

Abre um arquivo. Usa a `sys_open` (`const char *pathname, int flags, mode_t mode`).

Obs.: No parâmetro *mode* da `sys_open`, passa o valor 438 (constante `PERMISSION_MODE`) como modo de permissão de acesso (que corresponde à permissão de leitura e escrita).

ENTRADAS:

- `char* nome_arquivo`: endereço inicial da string do nome (ou do caminho+nome)
- `int modo_abertura`: valor 0 (constante `RDONLY`) para indicar abertura para leitura ou valor 577 (constante `WRONLY_CREAT_TRUNC`) para indicar abertura para escrita e criando o arquivo caso ele não exista ainda ou sobrescrevendo o conteúdo dele caso ele já exista.

SAÍDA:

- Devolve no registrador RAX o descritor do arquivo aberto.

4) fecha_arquivo(int descritor_arquivo)

Fecha um arquivo aberto previamente. Usa a sys_close.

ENTRADA:

- int descritor_arquivo: descritor do arquivo a ser fechado.

5) le_byte_arquivo(int descritor_arquivo, char* byte_arq)

Lê um byte de um arquivo aberto previamente para leitura. Usa a sys_read.

ENTRADAS:

- int descritor_arquivo: descritor do arquivo aberto para leitura.

- char* byte_arq: endereço da posição de memória onde será armazenado o byte lido do arquivo.

SAÍDA:

- Devolve em RAX o número de bytes lidos (ou seja, o valor devolvido pela chamada à sys_read).

6) grava_string_arquivo(int descritor_arquivo, char* buffer)

Grava string em um arquivo previamente aberto para escrita. A função supõe que a string é finalizada com '\n' (código 0xA) e com '\0' (código 0x0). Usa a sys_write.

ENTRADAS:

- int descritor_arquivo: descritor do arquivo aberto para escrita.

- char* buffer: ponteiro para a string (ou seja, o endereço da sua posição de memória inicial).

7) gera_string_hexadecimal(int valor, char* buffer)

Converte um número em uma string com a representação em hexadecimal dele. Por exemplo, para o inteiro 128526 (11111011000001110b), a string em hexadecimal é '0x1F60E'.

A função finaliza a string gerada com um caractere de quebra de linha '\n' (código 0xA) e com o '\0' (código 0x0).

ENTRADAS:

- int valor: o número inteiro a ser convertido.

- char* buffer: endereço da posição inicial da região de memória previamente alocada que receberá a string gerada na conversão.

No corpo do programa principal, você deve chamar todas as funções obrigatórias. Para facilitar a implementação, foi fornecido juntamente com o enunciado do EP1 um esqueleto do programa, contendo a documentação das funções obrigatórias e algumas constantes úteis.

2.3 Orientações Importantes

- **Organize e documente muito bem o seu código! Isso será avaliado na correção do EP.**

- Você não pode mudar os parâmetros de entrada, o tipo de valor de retorno e nem o comportamento esperado das funções obrigatórias.
- Os parâmetros das funções obrigatórias devem ser sempre passados pela pilha, seguindo as convenções vistas em aula. As variáveis locais das funções também devem ficar na pilha. Caso a função devolva um valor, o valor é devolvido no RAX.
- Para facilitar a implementação e a chamada das funções obrigatórias, considere que os parâmetros do tipo `int` são armazenados em 8 bytes, assim como os endereços de memória (ou seja, os ponteiros).
- Você pode criar funções adicionais se julgar necessário.
- A medida que você termina a implementação de uma função, teste a chamada dela de forma isolada, para verificar se ela gera o resultado esperado. Será mais fácil identificar e corrigir possíveis problemas desse modo. Inspecionar o código do programa inteiro para encontrar erros pode ser bem mais complicado.
- Todas as chamadas de sistema (*syscalls*) que podem ser usadas no EP foram indicadas na descrição das funções obrigatórias.
- Para a implementação do EP, você não precisará usar nenhum comando/instrução de linguagem de montagem que não tenha sido visto no curso.
- Você não precisa fazer tratamento de erros nos dados de entrada do programa. Por exemplo, você pode supor que o usuário sempre fornecerá um nome de arquivo de entrada válido (ou seja, o nome de um arquivo que existe no caminho indicado e que contém somente caracteres codificados em UTF-8).
- Para verificar se a saída do seu programa está correta, além de inspecionar visualmente os code points gravados e compará-los aos resultados disponibilizados junto com o enunciado, você pode usar o programinha em Python mostrado abaixo. Esse programa lê de um arquivo as *strings* de *code points* geradas pelo programa do EP, converte-as em números inteiros e depois os usa para gerar caracteres e gravá-los num arquivo texto. Se os *code points* gerados pelo seu programa do EP estiverem corretos, então o arquivo gerado pelo programinha de Python deve ser idêntico ao usado como entrada para a geração dos *code points*.

```

arq_entrada = open("code_points.txt", "r")
arq_saida = open("arquivo_utf8_reconvertido.txt", "w")
for linha in arq_entrada:
    # converte linha de code point lida do arquivo em int
    code_point = int(linha.strip(),16)
    # grava no arq. de saída o caractere correspondente em utf-8
    arq_saida.write(chr(code_point))
arq_entrada.close()
arq_saida.close()

```