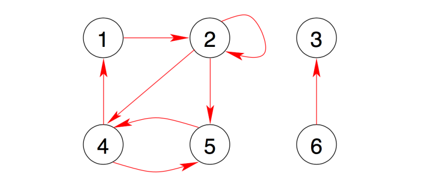
Exercícios Resolvidos

Programação com Grafos em Python

Tomando por base a representação de grafos orientados estudada, e a função básica de travessia em profundidade:



Grafo orientado g

```python
g = {}
g[1] = [2]
g[2] = [2,4,5]
g[3] = []
g[4] = [1,5]
g[5] = [4]
g[6] = [3]

def visit(g, s):
    color[s] = 'GRAY'
    for v in g[s]:
        if color[v] == 'WHITE':
            visit(g, v)
    color[s] = 'BLACK'
    print (s, 'BLACK')

```

1. Escreva uma função que conte o número de vértices alcançáveis a partir de um vértice. Por exemplo no grafo acima, esta função deve calcular como resultado 3 para o vértice 5, e 1 para o vértice 6.

RESOLUÇÃO:

```python
# função auxiliar
def count_reachable(g, s):
    c = 1
```

```
    color[s] = 'GRAY'
    for v in g[s]:
        if color[v] == 'WHITE':
            c += count_reachable(g, v)
    color[s] = 'BLACK'
    return c

def count_reachable_top(g, s):
    for v in g:
        color[v] = 'WHITE'
    return count_reachable(g, s)-1

```

2. O algoritmo de travessia da função `visit` parte de um vértice `s` dado e não abrange os vértices não alcançáveis a partir dele. Escreva uma função `visit_all` que recebe um grafo e inicia uma ou mais travessias, com origem em diferentes vértices, até que todo o grafo tenha sido visitado. Ao contrário de `visit`, a função deverá inicializar o dicionário `color`.

```
def visit_all(g):
    for v in g:
        color[v] = 'WHITE'

    for v in g:
        if color[v] == 'WHITE':
            visit(g, v)

```

Linguagens

1. Utilizando construções da teoria de conjuntos, escreva **duas expressões diferentes** para as seguintes linguagens.

   a. Linguagem sobre $\Sigma = \{a, b\}$ das palavras de comprimento múltiplo de 3 que têm, de 3 em 3 posições começando na segunda, obrigatoriamente o simbolo a.

b. Linguagem sobre $\Sigma = \{0,1\}$ das palavras que têm, em *pelo menos uma qualquer posição ímpar* (assumindo que a primeira posição é a posição 1) o símbolo 1. Por exemplo: 100, 001 pertencem à linguagem, mas 000, 010, ou 0101, não.

RESOLUÇÃO:

a. $L = \{aaa, aab, baa, bab\}^*$ ou $L = \{(xay)^i \mid x, y \in \{ab\} \text{ e } i \geq 0\}$

b. Definiremos a linguagem *complemento* da linguagem cujas palavras têm em *todas* as posições ímpares um 0. Numa primeira aproximação, escrevemos:
$L = \Sigma^* \setminus \{00, 01\}^*$ ou $L = \Sigma^* \setminus \{(0c)^i \mid c \in \{0,1\} \text{ e } i \geq 0\}$
Note-se que na linguagem $\{00, 01\}^*$ todas as palavras têm comprimento par, mas o seu complemento L inclui palavras de comprimento ímpar. Segundo a definição L deve de facto conter palavras de todos os comprimentos possíveis, no entanto esta solução não está correcta. De facto palavras de comprimento ímpar como 01010 não pertencem a $\{00, 01\}^*$, e por isso pertencerão a L, o que não é correcto.
A solução correcta será:
$L = \Sigma^* \setminus (\{00, 01\}^*\{\varepsilon, 0\})$ ou $L = \Sigma^* \setminus \{(0c)^i x \mid c \in \{0,1\} \text{ e } x \in \{\varepsilon, 0\} \text{ e } i \geq 0\}$

2. Considere o alfabeto $\Sigma = \{0,1,2\}$ e a ordem $0 < 1 < 2$. Liste as primeiras 20 palavras da linguagem Σ^*, considerando
   a. a ordem lexicográfica
   b. enumeração por comprimento crescente

RESOLUÇÃO:

a. 0, 00, 000, 0000, ... 00000000000000000000, ...

b. 0, 1, 2, 00, 01, 02, 10, 11, 12, 20, 21, 22, 000, 001, 002, 010, 011, 012, 020, 021, ...

Autómatos Determinísticos

1. Desenhe DFAs, devidamente *totalizados*, para o reconhecimento das linguagens definidas acima:

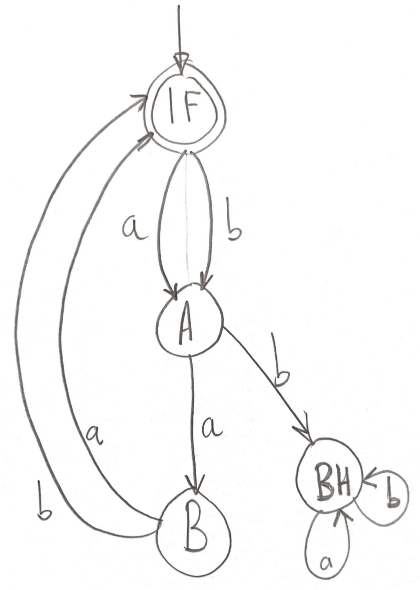
   a. Linguagem sobre $\Sigma = \{a, b\}$ das palavras de comprimento múltiplo de 3 que têm, de

3 em 3 posições começando na segunda, obrigatoriamente o simbolo a.

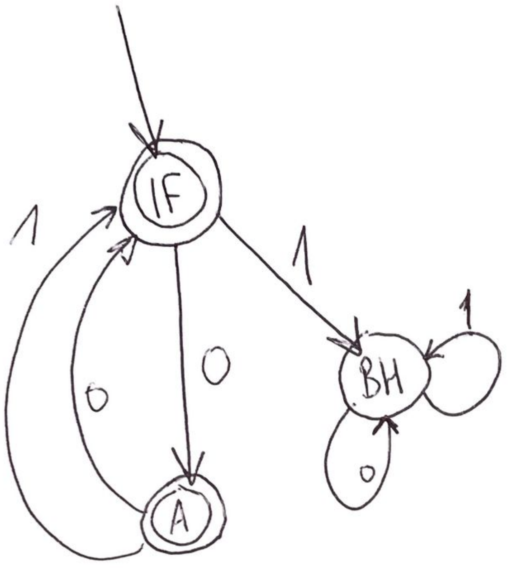
b. Linguagem sobre $\Sigma = \{0, 1\}$ das palavras que têm, em *pelo menos uma qualquer posição ímpar* (assumindo que a primeira posição é a posição 1) o símbolo 1.

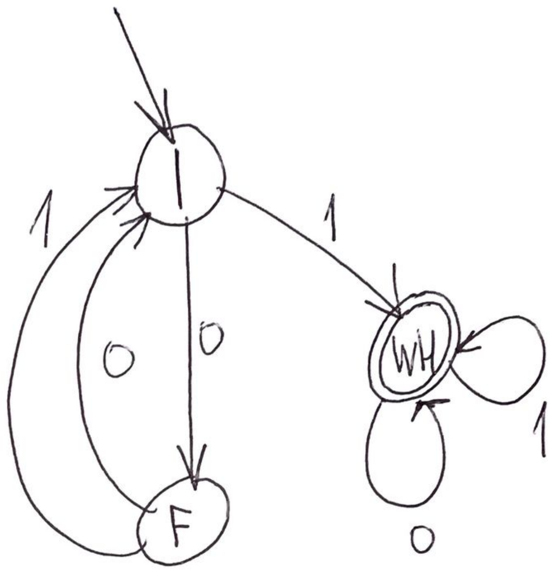
RESOLUÇÃO:

a.



b. Seguiremos o mesmo princípio de complementaridade que foi utilizado para escrever uma expressão de conjuntos para esta linguagem.
Começamos pois por definir um autómato para a linguagem cujas palavras têm em *todas* as posições ímpares um 0 (à esquerda). Note-se que o autómato aceita todas as palavras de comprimento par ou ímpar com 0 nas posições ímpares, uma vez que o estado A é final. O autómato é totalizado pela introdução do estado BH ("buraco negro"): a partir do momento em que um 1 é lido numa posição ímpar, a palavra não mais poderá ser aceite.
Para **obter o complemento deste autómato** (que por definição reconhece o complemento da sua linguagem) **basta tornar finais os estados que não o eram e vice-versa**, obtendo-se o autómato da direita. Note-se que neste autómato, uma vez lido um 1 numa posição ímpar, a palavra será aceite, quaisquer que sejam os símbolos lidos em seguida. O estado final WH é por isso normalmente designado por "buraco branco", sendo num certo sentido o oposto de um buraco negro).





Autómatos de Pilha

1. Considere a gramática livre de contexto com as seguintes regras para o alfabeto $\{a, b\}$, com símbolos não-terminais S e X, sendo S o inicial:

   - $S \rightarrow XS$
   - $S \rightarrow \varepsilon$
   - $X \rightarrow aXb$
   - $X \rightarrow Xb$
   - $X \rightarrow ab$

   a. Escreva uma derivação para a palavra “aabbaabbb”
   b. Defina um autómato de pilha que reconheça a linguagem definida por esta gramática, escolhendo uma das variantes apresentadas em cima
   c. Simule uma execução do autómato que definiu, que aceite a mesma palavra “aabbaabbb”

   Pode também repetir este exercício para a gramática da linguagem $\{a^i b^j c^{i+j} \mid i, j \geq 0\}$ apresentada acima neste módulo.

   RESOLUÇÃO:

   a.

   $$S \rightarrow XS \rightarrow XXS \rightarrow XX\varepsilon \rightarrow aXbX \rightarrow aabbX \rightarrow aabbXb \rightarrow aabbaXbb \rightarrow aabbaabbb$$

b. Construímos o autómato dado pelo método “top down”. Terá os alfabetos $\Sigma = \{a, b\}$ e $\Gamma = \{a, b, S\}$. A função δ será definida pelas transições descritas na seguinte tabela, em que $(q', S') \in \delta(q, x, Y)$.

q	x	Y	Nome	q'	S'
q_1	ε	S	expand-1	q_1	$[X, S]$
q_1	ε	S	expand-2	q_1	$[\ ]$
q_1	ε	X	expand-3	q_1	$[a, X, b]$
q_1	ε	X	expand-4	q_1	$[X, b]$
q_1	ε	X	expand-5	q_1	$[a, b]$
q_1	a	a	match-a	q_1	$[\ ]$
q_1	b	b	match-b	q_1	$[\ ]$

c.

(q_1 , aabbaabbb , [S])

expand-1	(q_1 , aabbaabbb , [X, S])
expand-3	(q_1 , aabbaabbb , [a, X, b, S])
match-a	(q_1 , abbaabbb , [X, b, S])
expand-5	(q_1 , aabbaabbb , [a, b, b, S])
match-a	(q_1 , bbaabbb , [b, b, S])
match-b	(q_1 , baabbb , [b, S])
match-b	(q_1 , aabbb , [S])
expand-1	(q_1 , aabbb , [X, S])
expand-4	(q_1 , aabbb , [X, b, S])
expand-3	(q_1 , aabbb , [a, X, b, b, S])
match-a	(q_1 , abbb , [X, b, b, S])
expand-5	(q_1 , abbb , [a, b, b, b, S])
match-a	(q_1 , bbb , [b, b, b, S])
match-b	(q_1 , bb , [b, b, S])
match-b	(q_1 , b , [b, S])
match-b	(q_1 , ε , [S])
expand-2	(q_1 , ε , [ ])

2. Considere a seguinte gramática livre de contexto para expressões aritméticas construídas com os operadores + e *, parêntesis (,), e variáveis id $\in \{x, y, z, \ldots\}$

- S → S+X
- S → X
- X → X*Y
- X → Y
- Y → (S)
- Y → id   (uma regra para cada variável)

a. Desenhe a árvore de sintaxe da expressão $x + y * z$ segundo esta gramática
b. Defina os autómatos para o reconhecimento da linguagem definida por esta gramática:
   i. pelo método *top-down*
   ii. pelo método *bottom-up*
c. Simule a execução do autómato e a construção da árvore de sintaxe, em ambos os casos.

RESOLUÇÃO:

a.

```
           S
     /     |     \
    S      +      X
    |         /   |   \
    X        X    *    Y
    |        |         |
    Y        Y         z
    |        |
    x        y
```

b. Resolve-se aqui apenas pelo método *bottom-up.*

q	x	Y	Nome	q'	S'
q_1	ε	[X, +, S]	reduce-1	q_1	[S]
q_1	ε	[X]	reduce-2	q_1	[S]
q_1	ε	[Y, * , X]	reduce-3	q_1	[X]
q_1	ε	[Y]	reduce-4	q_1	[X]
q_1	ε	[ ), S, ( ]	reduce-5	q_1	[Y]
q_1	ε	[id]	reduce-6	q_1	[Y]
q_1	+	[]	shift-+	q_1	[+]

q_1	*	[ ]	shift-*	q_1	[*]
q_1	(	[ ]	shift-(	q_1	[( ]
q_1	)	[ ]	shift-)	q_1	[ )]
q_1	id	[ ]	shift-id	q_1	[id]

c.

(q_1 , x+y*z , [ ])
shift-id   (q_1 , +y*z , [x])
reduce-6   (q_1 , +y*z , [Y])

```
    Y
    |
    x
```

reduce-4   (q_1 , +y*z , [X])

```
    X
    |
    Y
    |
    x
```

reduce-2   (q_1 , +y*z , [S])

```
    S
    |
    X
    |
    Y
    |
    x
```

shift-+   (q_1 , y*z , [+, S])
shift-id   (q_1 , *z , [y, +, S])
reduce-6   (q_1 , *z , [Y, +, S])

```
    S
    |
    X
    |
    Y          Y
    |          |
```

```
    x        y
```

reduce-4 $(q_1$, *z , [X, +, S])

```
    S
    |
    X        X
    |        |
    Y        Y
    |        |
    x        y
```

shift-* $(q_1$, z , [*, X, +, S])

shift-id $(q_1$, ε , [z, *, X, +, S])

reduce-6 $(q_1$, ε , [Y, *, X, +, S])

```
    S
    |
    X        X         Y
    |        |         |
    Y        Y         z
    |        |
    x        y
```

reduce-3 $(q_1$, ε , [X, +, S])

```
    S             X
    |         /   |   \
    X        X    *    Y
    |        |         |
    Y        Y         z
    |        |
    x        y
```

reduce-1 $(q_1$, ε , [S])

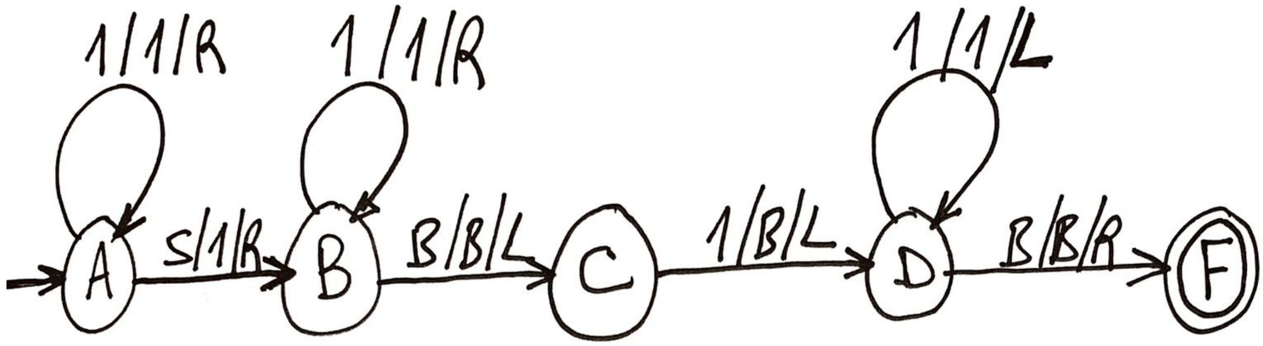
```
           S
     /     |     \
    S      +      X
    |         /   |   \
    X        X    *    Y
    |        |         |
    Y        Y         z
    |        |
    x        y
```

Máquinas de turing

1. É conveniente para a implementação de operações aritméticas com máquinas de Turing utilizar *notação unária* para os números. Enquanto na notação decimal são utilizados 10 símbolos diferentes e na notação binária são utilizados 2, na notação unária é utilizado apenas um símbolo. Utilizaremos aqui o símbolo 1 mas qualquer outro serviria. Nesta notação um número natural pode ser representado por uma sequência, por exemplo 5 será representado por 11111.
   Neste exercício pretende-se implementar a operação de adição através de uma MT. Considere que a fita contém inicialmente os dois números que se pretende somar, representados por sequências de "1" e separados por um símbolo "S". A cabeça da máquina está inicialmente posicionada sobre o primeiro símbolo do primeiro número. Por exemplo para somar 3+4, a fita deverá conter inicialmente
   ... | 1 | 1 | 1 | S | 1 | 1 | 1 | 1 | ...
   e no final deverá conter apenas
   ... | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ...

   RESOLUÇÃO:
   Informalmente basta "apagar" o símbolo "S". Mas note-se que isto requer mover todos os "1" que se encontram à sua direita para a esquerda...

   Para isso:
   - Fazemos (estado inicial A) avançar a cabeça, mantendo os "1", até encontrar o símbolo "S"
   - Substituímos o "S" por um "1"
   - Fazemos (estado B) avançar a cabeça, mantendo os "1", até encontrar o símbolo "B" (*Blank,* vazio)
   - neste momento foi seguramente escrito um "1" em excesso, que terá de ser apagado. Mesmo que o segundo número seja 0, foi escrito um "1" sobre o "S"
   - Move-se então a cabeça para a esquerda, ficando sobre o último "1" (estado C), e em seguida novamente para a esquerda, substituindo esse "1" por "B"
   - Move-se a cabeça completamente para a esquerda (estado D), até ao primeiro símbolo "B", e finalmente avança-se novamente a cabeça para a direita, para ficar sobre o primeiro "1", ficando a máquina agora no estado final "F"



Por exemplo:

... | 1 | 1 | 1 | S | 1 | 1 | ... [A]

... | 1 | 1 | 1 | S | 1 | 1 | ... [A]

... | 1 | 1 | 1 | S | 1 | 1 | ... [A]

... | 1 | 1 | 1 | S | 1 | 1 | ... [A]

... | 1 | 1 | 1 | 1 | 1 | 1 | ... [B]

... | 1 | 1 | 1 | 1 | 1 | 1 | ... [B]

... | 1 | 1 | 1 | 1 | 1 | 1 | | ... [B]

... | 1 | 1 | 1 | 1 | 1 | 1 | ... [C]

... | 1 | 1 | 1 | 1 | 1 | | ... [D]

....

... | 1 | 1 | 1 | 1 | 1 | | ... [D]

... | | 1 | 1 | 1 | 1 | 1 | | ... [D]

... | 1 | 1 | 1 | 1 | 1 | | ... [F]