

Num determinado mundo, os números de telefone são constituídos por 3 partes:

- código do país: 1 a 3 dígitos;
- código da região: 1 a 3 dígitos;
- nº de telefone: 4 a 10 dígitos.

Sabendo que o separador entre a 1ª e a 2ª parte e entre a 2ª e a 3ª parte pode ser o hífen ou o espaço mas terá de ser o mesmo entre as duas partes, ou seja, existindo um hífen entre a 1ª e a 2ª partes também terá de ser um hífen a estar entre a 2ª e 3ª partes, das expressões regulares que se apresentam a seguir assinale aquelas que fariam match apenas com os números de telefone **válidos**:

☐ a.  $(([0-9]\{1,3\}-)\{2\} | ([0-9]\{1,3\} \backslash )\{2\}) [0-9]\{4,10\}$

☐ b.  $(0|1|2|3|4|5|6|7|8|9)\{1,3\}(\backslash | -)(0|1|2|3|4|5|6|7|8|9)\{1,3\}(\backslash | -)(0|1|2|3|4|5|6|7|8|9)\{4,10\}$

☐ c.  $[0-9]\{1,3\}[\backslash -][0-9]\{1,3\}[\backslash -][0-9]\{4,10\}$

☐ d.  $[0-9][0-9]?[0-9]?[\backslash -][0-9]?[0-9]?[0-9][\backslash -][0-9][0-9][0-9][0-9]\{1,7\}$

## Pergunta 7

Considere o seguinte extrato de um filtro de texto em Python (em que o operador ':'=' calcula a expressão, atribui o valor à variável e verifique se esse valor é verdadeiro ou falso):

```
import sys
import re

for linha in sys.stdin:
    if s := re.search(r'<At>', linha):
        acc1(s.group())
    elif s := re.search(r'<([At])>', linha):
        acc2(s.group(1))
    elif s := re.search(r'\<\/?(A|t)\>', linha):
        acc3(s.group())
    elif s := re.match(r' [<At>]', linha):
        acc4(s.group())
    else:
        pass
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. Se o texto de entrada for "<</t>123" a ação "acc4 ('<')" é executada;
- ☐ b. Se o texto de entrada for "<At><At>" a ação "acc1 ('<At>')" é executada 2 vezes.
- ☐ c. se o texto de entrada for "egege </t>" a ação "acc3 ('</t>')" é executada;
- ☒ d. Se o texto de entrada for "egege <A>" a ação "acc2 ('A')" é executada;

O texto abaixo especifica em termos abstratos, de forma independente da linguagem, um Analisador Léxico para reconhecer os símbolos terminais de um dada linguagem de programação:

```
BEG = 1
END = 2
ID  = 3
NUMI = 4
NUMF = 5
%%
[=+\-*/()] { return text[0]; }
(?i:Inicio) { return BEG; }
[fF][iI][mM] { return END; }
[a-zA-Z][a-zA-Z0-9]* { return ID; }
[0-9]+ { return NUMI; }
[0-9]+\.[0-9]+ { return NUMF; }
.\n { ; } //ignore
%%
```

Selecione as alíneas abaixo que são afirmações verdadeiras.

- ☐ a. Se o texto de entrada for "a1.5/12-5." a sequência de símbolos retornados é: 3 5 '/' 4 '-' 5
- ☐ b. Se o texto de entrada for "4\*5.6 = FIM" a sequência de símbolos retornados é: 4 '\*' 4 '.' 4 '=' 2
- ☐ c. Se o texto de entrada for "Xico2= 1.2 +78." a sequência de símbolos retornados é: 3 '=' 5 '+' 4
- ☐ d. Se o texto de entrada for "(.x = ab12-Inic)" a sequência de símbolos retornados é: '(' 3 '=' 3 '-' 3 ')'

Considere o seguinte autômato **A1** descrito pelos seus ramos (triplos) em que **1** é o estado inicial e os números negativos denotam estados finais:

(1, m, 2)  
(2, v, 3)  
(3, space, -1)  
(1, m, 4)  
(4, a, 5)  
(5, k, 6)  
(6, e, 7)  
(7, space, -2)  
(1, l, 8)  
(8, s, 9)  
(9, space, -3)  
(1, l, 10)  
(10, a, 11)  
(11, space, -4)

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. O autômato **A1** é determinista porque não tem transições espontâneas;
- ☐ b. Como **A1** é não-determinista, não existe nenhuma **ER** equivalente a **A1**.
- ☒ c. O autômato **A1** é não-determinista porque tem estados com mais do que uma transição pelo mesmo símbolo;
- ☒ d. A **ER** 'm(v|ake) | l(a|s) ' define a mesma linguagem que **A1** ;

Quais das seguintes expressões regulares dão match a uma password com as seguintes condições:

- 6 a 12 caracteres de tamanho;
- Pelo menos uma maiúscula;
- Pelo menos uma minúscula;
- Pelo menos um dígito.

☐ a.  $r'^{[a-zA-Z0-9]\{,13\}}\$'$

☐ b.  $r'^{(.|\backslash n)^+\$}'$

☐ c.  $r'^{(?i:[a-zA-Z])|[0-9]\$}'$

☐ d.  $r'^{[a-z]\{6,12\}|[A-Z]\{6,12\}|[0-9]\{6,12\}}\$'$

O texto abaixo especifica em termos abstratos, de forma independente da linguagem, um Analisador Léxico para reconhecer os símbolos terminais de uma dada linguagem de programação:

```
BEG = 1
END = 2
ID  = 3
NUMI = 4
NUMF = 5
%%
[=+\-*/()]      { return text[0]; }
(?i:Inicio)     { return BEG; }
[fF][iI][mM]    { return END; }
[a-zA-Z][a-zA-Z0-9]* { return ID; }
[0-9]+          { return NUMI; }
[0-9]+\.[0-9]+  { return NUMF; }
.|\\n          { ; } //ignore
%%
```

Selecione as alíneas abaixo que são afirmações verdadeiras.

- ☐ a. Se o texto de entrada for "a1.5/12-5." a sequência de símbolos retornados é: 3 5 '/' 4 '-' 5
- ☐ b. Se o texto de entrada for "(.x = ab12-Inic)" a sequência de símbolos retornados é: '(' 3 '=' 3 '-' 3 ')'
- ☐ c. Se o texto de entrada for "4\*5.6 = FIM" a sequência de símbolos retornados é: 4 '\*' 4 '.' 4 '=' 2
- ☐ d. Se o texto de entrada for "Xico2= 1.2 +78." a sequência de símbolos retornados é: 3 '=' 5 '+' 4

## Pergunta 6

Quais das seguintes expressões regulares dão match a uma password com as seguintes condições:

- 6 a 12 caracteres de tamanho;
- Pelo menos uma maiúscula;
- Pelo menos uma minúscula;
- Pelo menos um dígito.

☐ a.  $r'^{^}[a-zA-Z0-9]\{,13\}\$'$

☐ b.  $r'^{^}[a-z]\{6,12\} | [A-Z]\{6,12\} | [0-9]\{6,12\}\$'$

☐ c.  $r'^{^}(.|\backslash n)^+\$'$

☐ d.  $r'^{^}(?i:[a-zA-Z]) | [0-9]\$'$



Num determinado mundo, os números de telefone são constituídos por 3 partes:

- código do país: 1 a 3 dígitos;
- código da região: 1 a 3 dígitos;
- nº de telefone: 4 a 10 dígitos.

Sabendo que o separador entre a 1ª e a 2ª parte e entre a 2ª e a 3ª parte pode ser o hífen ou o espaço mas terá de ser o mesmo entre as duas partes, ou seja, existindo um hífen entre a 1ª e a 2ª partes também terá de ser um hífen a estar entre a 2ª e 3ª partes, das expressões regulares que se apresentam a seguir assinale aquelas que fariam match apenas com os números de telefone válidos:

- ☐ a. `(0|1|2|3|4|5|6|7|8|9){1,3}(\ |-)(0|1|2|3|4|5|6|7|8|9){1,3}(\ |-)(0|1|2|3|4|5|6|7|8|9){4,10}`
- ☐ b. `[0-9]{1,3}[\ \-][0-9]{1,3}[\ \-][0-9]{4,10}`
- ☐ c. `[0-9][0-9]?[0-9]?[\ \-][0-9]?[0-9]?[0-9][\ \-][0-9][0-9][0-9][0-9]{1,7}`
- ☐ d. `(([0-9]{1,3}-){2}|([0-9]{1,3}\ ){2})[0-9]{4,10}`



Considere o seguinte extrato de um filtro de texto em Python (em que o operador ':' calcula a expressão, atribui o valor à variável e verifica se esse valor é verdadeiro ou falso):

```
import sys
import re

for linha in sys.stdin:
    if s := re.search(r'<At>', linha):
        acc1(s.group())
    elif s := re.search(r'<([At])>', linha):
        acc2(s.group(1))
    elif s := re.search(r'\<\/?(A|t)\>', linha):
        acc3(s.group())
    elif s := re.match(r'<At>', linha):
        acc4(s.group())
    else:
        pass
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. Se o texto de entrada for "<</t>123" a ação "acc4 ('<')" é executada;
- ☒ b. Se o texto de entrada for "egege <A>" a ação "acc2 ('A')" é executada;
- ☒ c. se o texto de entrada for "egege </t>" a ação "acc3 ('</t>')" é executada;
- ☒ d. Se o texto de entrada for "<At><At>" a ação "acc1 ('<At>')" é executada 2 vezes.

Considere os Terminais "str" (texto entre aspas), texto (sequência de caracteres) e "id" (sequência não nula de letras) e a seguinte Gramática Independente de Contexto (G4):

```
Anota -> Abre texto Fecha
Abre  -> '<' id '>'
      | '<' id LstA '>'
Fecha -> '<' '/' id '>'
LstA  -> Atr
LstA  -> LstA Atr
Atr   -> id '=' str
```

Selecione então as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. A gramática G4, tal como está escrita, não permite derivar uma frase como "<ttt>ola ole oli</z>".
- ☐ b. Se as 2ª e 3ª produções fossem trocadas pelas três produções a seguir, a linguagem L(G4) não se alterava:

```
Abre  -> '<' id Atts '>'
Atts  -> ε
Atts  -> id '=' str Atts
```

- ☐ c. A frase "<ttt=\"vv\" o=12>bla bla</ttt>" pertence à linguagem L(G4) gerada por esta gramática.
- ☐ d. A frase "<ttt a=\"1\">bla bla</ttt>" pertence à linguagem L(G4) gerada por esta gramática.

## Pergunta 5

Considere o seguinte extrato de um filtro de texto em Python, em que `MMM` é uma das funções disponíveis no módulo `'re'`:

```
s = re.MMM(r'\\w+{([^}]+)}', linha)
if (s):
    print(s.group(1))
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. Se `'linha'` for `"\tttt{({nhh} jdjnf \cmd{1234})}"` e o texto de saída `"({nhh1234}"` a função `MMM` corresponde a `'search'`;
- ☐ b. Se `'linha'` for `" \tttt[nhhhh} jdjnf \cmd{1234} huvb"` e o texto de saída `"1234"` a função `MMM` corresponde a `'search'`;
- ☐ c. Se o fragmento acima for acrescentado com

```
s = re.findall(r'\d+', linha)
if (s):
    print(s)
```

e `'linha'` for `"\tttt{nhhhh} jdjnf \cmd{1234}"` a saída terá mais de 1 linha;

- ☐ d. Se o fragmento acima for acrescentado com

```
s = re.findall(r'\d+', s)
if (s):
    print(s)
```

e `'linha'` for `"\tttt{nhhhh} jdjnf \cmd{1234}"` a saída terá mais de 1 linha.

Considere o seguinte autômato **A1** descrito pelos seus ramos (triplos) em que 1 é o estado inicial e os números negativos denotam estados finais:

(1, m, 2)  
(2, v, 3)  
(3, space, -1)  
(1, m, 4)  
(4, a, 5)  
(5, k, 6)  
(6, e, 7)  
(7, space, -2)  
(1, l, 8)  
(8, s, 9)  
(9, space, -3)  
(1, l, 10)  
(10, a, 11)  
(11, space, -4)

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. A ER `'m(v|ake) | l(a|s) '` define a mesma linguagem que **A1**;
- ☐ b. O autômato **A1** é determinista porque não tem transições espontâneas;
- ☐ c. O autômato **A1** é não-determinista porque tem estados com mais do que uma transição pelo mesmo símbolo;
- ☐ d. Como **A1** é não-determinista, não existe nenhuma ER equivalente a **A1**.

Considere o seguinte extrato de um filtro de texto em Python (em que o operador ':'=' calcula a expressão, atribui o valor à variável e verifica se esse valor é verdadeiro ou falso):

```
import sys
import re

for linha in sys.stdin:
    if s := re.search(r'<At>', linha):
        acc1(s.group())
    elif s := re.search(r'<([At])>', linha):
        acc2(s.group(1))
    elif s := re.search(r'\<\/?(A|t)\>', linha):
        acc3(s.group())
    elif s := re.match(r'<At>', linha):
        acc4(s.group())
    else:
        pass
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. Se o texto de entrada for "<</t>123" a ação "acc4 ('<')" é executada;
- ☐ b. Se o texto de entrada for "egege <A>" a ação "acc2 ('A')" é executada;
- ☐ c. se o texto de entrada for "egege </t>" a ação "acc3 ('</t>')" é executada;
- ☐ d. Se o texto de entrada for "<At><At>" a ação "acc1 ('<At>')" é executada 2 vezes.

Considere o seguinte extrato de um filtro de texto em Python, em que `MMM` é uma das funções disponíveis no módulo `'re'`:

```
s = re.MMM(r'\\w+{([^}]+)}', linha)
if (s):
    print(s.group(1))
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. Se o fragmento acima for acrescentado com

```
s = re.findall(r'\d+', s)
if (s):
    print(s)
```

e 'linha' for `"\tttt{nhhhh} jdjnfn \cmd{1234}"` a saída terá mais de 1 linha.

- ☐ b. Se 'linha' for `" \tttt[nhhhh} jdjnfn \cmd{1234} huvb"` e o texto de saída `"1234"` a função `MMM` corresponde a `'search'`;

- ☐ c. Se 'linha' for `"\tttt{({nhh} jdjnfn \cmd{1234})"` e o texto de saída `"({nhh1234"` a função `MMM` corresponde a `'search'`;

- ☐ d. Se o fragmento acima for acrescentado com

```
s = re.findall(r'\d+', linha)
if (s):
    print(s)
```

e 'linha' for `"\tttt{nhhhh} jdjnfn \cmd{1234}"` a saída terá mais de 1 linha;

Quais das seguintes expressões regulares dão match a uma password com as seguintes condições:

- 6 a 12 caracteres de tamanho;
- Pelo menos uma maiúscula;
- Pelo menos uma minúscula;
- Pelo menos um dígito.

☐ a.  $r'^{^}[a-zA-Z0-9]\{,13\}\$'$

☐ b.  $r'^{^}[a-z]\{6,12\} | [A-Z]\{6,12\} | [0-9]\{6,12\}\$'$

☐ c.  $r'^{^}(.|\backslash n)^+\$'$

☐ d.  $r'^{^}(?i:[a-zA-Z])|[0-9]\$'$



```

import sys
import re

for linha in sys.stdin:
    if re.search(r'==', linha):
        res = re.sub(r'==', '.EQ.', linha)
        print(res)
    elif re.search(r'~= ', linha):
        res = re.sub(r'~= ', '!=', linha)
        print(res)
    elif re.search(r'<=', linha):
        res = re.sub(r'<=', '<=', linha)
        print(res)
    else:
        print(linha)

```

- ☐ a. O programa transforma o texto `a ~= b+c;` em `a ~.EQ. b+c;`
- ☐ b. O programa transforma o texto `a == b+c;` em `a == b+c;`
- ☐ c. Se acrescentássemos à especificação uma nova 4ª regra:

```

elif re.search(r'[a-z][a-zA-Z0-9]*', linha):
    res = re.sub(r'[a-z][a-zA-Z0-9]*', '.ID.', linha)
    print(res)
else:
    print(linha)

```

o resultado do Filtro gerado pelo Flex a partir dessa nova especificação, aplicado ao texto: `a ~= b+c;` não se alterava, mas em geral o seu comportamento é diferente.

- ☐ d. Se a 2ª regra da especificação acima fosse alterada para:

```

elif re.search(r'~(?==)', linha):
    res = re.sub(r'~(?==)', '!=', linha)
    print(res)

```

o resultado do programa a partir dessa nova especificação, aplicado ao texto: `a ~= b+c;` não se alterava;

```

for linha in sys.stdin:
    segments = re.findall(r'(00+)|(11+)|(1)|(0)', linha)
    for (zs, us, u, z) in segments:
        if zs:
            print( str(len(zs)) + '0', end='')
        elif us:
            print( str(len(us)) + '1', end='')
        elif u:
            print('1', end='')
        elif z:
            print('0', end='')
        else:
            pass

    print('\n', end='')
print('\n', end='')

```

e indique quais das alíneas seguintes são verdadeiras:

- ☐ a. Se a expressão regular fosse alterada para `r'(00*)|(11*)|(1)|(0)'`, o resultado final para uma linha com conteúdo "010" não se alterava;
- ☐ b. Se o texto de input tivesse uma única linha com o conteúdo "0101010101" o resultado seria "0101010101";
- ☐ c. Se a expressão regular fosse alterada para `r'(00*)|(11*)|(1)|(0)'`, o resultado final para uma linha com conteúdo "0001110011" não se alterava;
- ☐ d. Se o texto de input tivesse uma única linha com o conteúdo "101110101111001" o resultado seria "103101041201";

Considere o seguinte extrato de um filtro de texto em Python, em que **MMM** é uma das funções disponíveis no módulo 're':

```
s = re.MMM(r'\\w+{([^}]+)}', linha)
if (s):
    print(s.group(1))
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a. Se o fragmento acima for acrescentado com

```
s = re.findall(r'\d+', linha)
if (s):
    print(s)
```

e 'linha' for "\tttt{nhhhh} jdjfn \cmd{1234}" a saída terá mais de 1 linha;

☐ b. Se 'linha' for " \tttt[nhhhh} jdjfn \cmd{1234} huvb" e o texto de saída "1234" a função **MMM** corresponde a 'search';

☐ c. Se 'linha' for "\tttt{({nhh} jdjfn \cmd{1234}" e o texto de saída "{nhh1234" a função **MMM** corresponde a 'search';

☐ d. Se o fragmento acima for acrescentado com

```
s = re.findall(r'\d+', s)
if (s):
    print(s)
```

e 'linha' for "\tttt{nhhhh} jdjfn \cmd{1234}" a saída terá mais de 1 linha.

## Python3: findall

```
import re
import sys

lista = re.compile(r'(\w+(, \w+)* e \w+)')
nlinha = 0

for linha in sys.stdin:
    nlinha = nlinha + 1
    res = lista.findall(linha)
    for (x, _) in res:
        print("linha", nlinha, ": ", x)
```

---

E analise com atenção o dataset seguinte (retirado do [lingua.pt](http://lingua.pt)).

```
1  Os alimentos preferidos de Eva
2  Eva é cozinheira e foi comprar alimentos para o almoço de domingo.
3  Ela prefere comprar as verduras e os legumes no supermercado que
4  fica ao lado de sua casa. As carnes ela compra sempre no açougue
5  e os peixes na peixaria.
6  Além disso, Eva costuma comer muitas frutas, como por exemplo,
7  banana, maçã, abacate, abacaxi, uva e melancia.
8  Ela costuma comprar frutas na quitanda que há no bairro onde mora.
9  Em sua casa, Eva possui uma horta onde planta alguns tipos de vegetais,
10 como cebola, hortelã e manjericão que são utilizados como temperos.
11 Aos domingos, Eva prepara bolos e tortas.
12 Ela usa farinha de trigo, ovos, leite e outros ingredientes
13 que também são comprados no supermercado.
14 Quando faz frio, ela prefere preparar sopas de legumes e carnes.
15 O caldo de lentilhas com pedaços de carne é uma de suas refeições
16 preferidas no inverno.
17 Nos dias mais quentes, Eva faz sorvetes e outros alimentos refrescantes,
18 como sucos de frutas gelados.
```

## Pergunta 2

Sabendo que:

- `\w` é uma abreviatura de `[a-zA-Z0-9_]`
- `for linha in sys.stdin:` lê linha a linha do canal de input até encontrar a marca de fim de ficheiro

Considere os 3 programas em Python seguintes, em que cada um usa uma funcionalidade diferente do módulo de expressões regulares:

---

### Python1: search

```
import re
import sys

lista = re.compile(r'(\w+(, \w+)* e \w+)')
nlinha = 0

for linha in sys.stdin:
    nlinha = nlinha + 1
    res = lista.search(linha)
    if(res):
        print("linha", nlinha, ": ", res.group(1))
```

### Python2: match

```
import re
import sys

lista = re.compile(r'(\w+(, \w+)* e \w+)')
nlinha = 0

for linha in sys.stdin:
    nlinha = nlinha + 1
    res = lista.match(linha)
    if(res):
        print("linha", nlinha, ": ", res.group(1))
```

Considere as expressões regulares (ER)

'e1 = a (a b)+ (c d | c f)\* j'

'e2 = (a a b)+ c (d\* | f\*) j'

'e3 = (a a b)+ (c d\* | c f\*) j'

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a.

as ER 'e2' e 'e3' geram exatamente as mesmas frases e por isso conclui-se que são ER equivalentes;

☐ b.

a frase "aabcfj" é válida nas linguagens L(e1) e L(e3);

☐ c.

as ER 'e1' e 'e2' são equivalentes porque geram exatamente as mesmas frases.

☐ d.

como a frase "aabcdj" é válida nas linguagens L(e1) e L(e2) então conclui-se que 'e1' é equivalente a 'e2';



E assinale das alíneas seguintes as que são verdadeiras.

- ☐ a. O programa `Pyton2:match` dá como resultado o seguinte output:

linha 7 : banana, maçã, abacate, abacaxi, uva e melancia

- ☐ b. O programa `Pyton1:search` dá como resultado o seguinte output:

linha 2 : cozinheira e foi  
linha 3 : verduras e os  
linha 7 : banana, maçã, abacate, abacaxi, uva e melancia  
linha 10 : cebola, hortelã e manjerição  
linha 11 : bolos e tortas  
linha 12 : trigo, ovos, leite e outros  
linha 14 : legumes e carnes  
linha 17 : sorvetes e outros  
linha 19 : alimentos e para  
linha 20 : assunto e lê

- ☐ c. O programa `Pyton3:findall` dá como resultado o seguinte output:

linha 7 : banana, maçã, abacate, abacaxi, uva e melancia  
linha 10 : cebola, hortelã e manjerição  
linha 11 : bolos e tortas  
linha 12 : trigo, ovos, leite e outros  
linha 14 : legumes e carnes  
linha 17 : sorvetes e outros

- ☐ d. O programa `Pyton1:search` dá como resultado o seguinte output:

linha 7 : banana, maçã, abacate, abacaxi, uva e melancia  
linha 10 : cebola, hortelã e manjerição  
linha 11 : bolos e tortas  
linha 12 : trigo, ovos, leite e outros  
linha 14 : legumes e carnes  
linha 17 : sorvetes e outros



## Pergunta 6

Considere as expressões regulares (ER)

'e1 = c (a b)+ ((j i)\* | (j k)\*)'

'e2 = (c a b)+ j (i\* | k\*)'

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a.

qualquer frase válida da linguagem L(e2) termina sempre por 'k';

☒ b. a frase 'cab' é a menor que pertence a L(e1);

☐ c. as ER 'e1' e 'e2' são equivalentes porque a frase 'cabji' pertence às linguagens L(e1) e L(e2);

☒ d.

a ER 'e1' pode escrever-se de forma menos compacta como

'(c (a b) (a b)\* | (c (a b) (a b)\* (j i)+) | (c (a b) (a b)\* (j k)+)'

sem alterar a linguagem gerada.

```
import re

aLer = True
while(aLer):
    linha = input()
    if(re.search(r'!!!', linha)):
        aLer = False
    elif(re.search(r'(dia|DIA)', linha)):
        print(1)
    elif(re.search(r'[Dd][Ii][Aa]', linha)):
        print(2)
    elif(re.search(r'[dia|DIA]', linha)):
        print(3)
    elif(re.search(r'dia{1,3}', linha)):
        print(4)
```

---

E considere o seguinte texto de input (baseado num poema de Fernando Pessoa), texto.txt:

---

```
Depois do dIa vem noite,
Depois da noite vem dia,
Depois do DIA vem noite,
Depois da noite vem dia,
e com o eco fica
diadia...
!!!
```

Se o programa fosse invocado da seguinte forma:

```
$ cat texto.txt | python dia.py
```

Irias obter uma sequência numérica no output.

Preenche a resposta com essa sequência sem deixar qualquer espaço entre os dígitos.

```
import re

aLer = True
while(aLer):
    linha = input()
    if(re.search(r'!!!', linha)):
        aLer = False
    elif(re.search(r'(dia|DIA)', linha)):
        print(1)
    elif(re.search(r'[Dd][Ii][Aa]', linha)):
        print(2)
    elif(re.search(r'[dia|DIA]', linha)):
        print(3)
    elif(re.search(r'dia{1,3}', linha)):
        print(4)
```

---

E considere o seguinte texto de input (baseado num poema de Fernando Pessoa), texto.txt:

---

```
Depois do dIa vem noite,
Depois da noite vem dia,
Depois do DIA vem noite,
Depois da noite vem dia,
e com o eco fica
diadia...
!!!
```

Considere a expressão regular (ER)  $'e1 = a b^+ c^* (d \mid a b)^* j'$

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a.

a ER  $'e1'$  é equivalente a  $'e4 = a b b^* (\varepsilon \mid c^+) ((a b) \mid d)^* j'$ ;

☐ b.

a ER  $'e1'$  é equivalente a  $'e2 = a b (bc)^* (d^* \mid (a b)^*) j'$ ;

☐ c.

a ER  $'e1'$  é equivalente a  $'e3 = a b^+ (c^* d \mid c^* (a b))^* j'$ ;

☐ d.

a menor frase que se pode derivar de  $'e1'$  tem 3 símbolos devendo sempre começar por  $'a'$ .

Considere as expressões regulares (ER)

'e1 = a (a b)+ (c d | c f)\* j'

'e2 = (a a b)+ c (d\* | f\*) j'

'e3 = (a a b)+ (c d\* | c f\*) j'

e selecione as alíneas abaixo que são afirmações verdadeiras:

☒ a.

a frase "aabc fj" é válida nas linguagens L(e1) e L(e3);

☐ b.

as ER 'e1' e 'e2' são equivalentes porque geram exatamente as mesmas frases.

☐ c.

como a frase "aabcdj" é válida nas linguagens L(e1) e L(e2) então conclui-se que 'e1' é equivalente a 'e2';

☒ d.

as ER 'e2' e 'e3' geram exatamente as mesmas frases e por isso conclui-se que são ER equivalentes;

Considere as expressões regulares (ER)

'e1 = a (a b)+ (c d | c f)\* j'

'e2 = (a a b)+ c (d\* | f\*) j'

'e3 = (a a b)+ (c d\* | c f\*) j'

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a.

a frase "aabcfj" é válida nas linguagens L(e1) e L(e3);

☐ b.

como a frase "aabcdj" é válida nas linguagens L(e1) e L(e2) então conclui-se que 'e1' é equivalente a 'e2';

☐ c.

as ER 'e2' e 'e3' geram exatamente as mesmas frases e por isso conclui-se que são ER equivalentes;

☐ d.

as ER 'e1' e 'e2' são equivalentes porque geram exatamente as mesmas frases.



Considere a expressão regular (ER) `'e1 = (b a b)* c (a d | f b )* j'`

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a.

a frase `"bcfbfbj"` pertence à linguagem gerada por `e1`;

☐ b.

a frase `"cj"` pertence à linguagem gerada por `e1`;

☐ c.

a frase `"babcdafbaj"` pertence à linguagem gerada por `e1`;

☐ d.

a frase `"babbabc"` é um prefixo válido de uma frase correta da linguagem gerada por `e1`.

Considere as expressões regulares (ER)

'e1 = c (a b)+ ((j i)\* | (j k)\*)'

'e2 = (c a b)+ j (i\* | k\*)

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a. a frase 'cab' é a menor que pertence a L(e1);

☐ b.

qualquer frase válida da linguagem L(e2) termina sempre por 'k';

☐ c. as ER 'e1' e 'e2' são equivalentes porque a frase 'cabji' pertence às linguagens L(e1) e L(e2);

☐ d.

a ER 'e1' pode escrever-se de forma menos compacta como

'(c (a b) (a b)\* | (c (a b) (a b)\* (j i)+) | (c (a b) (a b)\* (j k)+)'

sem alterar a linguagem gerada.

e selecione as alíneas abaixo que são afirmações verdadeiras:

☐ a.

o autómato finito determinista apresentado em cima (A1), reconhece as frases da linguagem  $L(e1)$ .

☐ b.

a ER 'e1' é equivalente à ER 'e2';

☐ c.

qualquer frase da linguagem  $L(e2)$  terá sempre um número par de 'a';

☐ d.

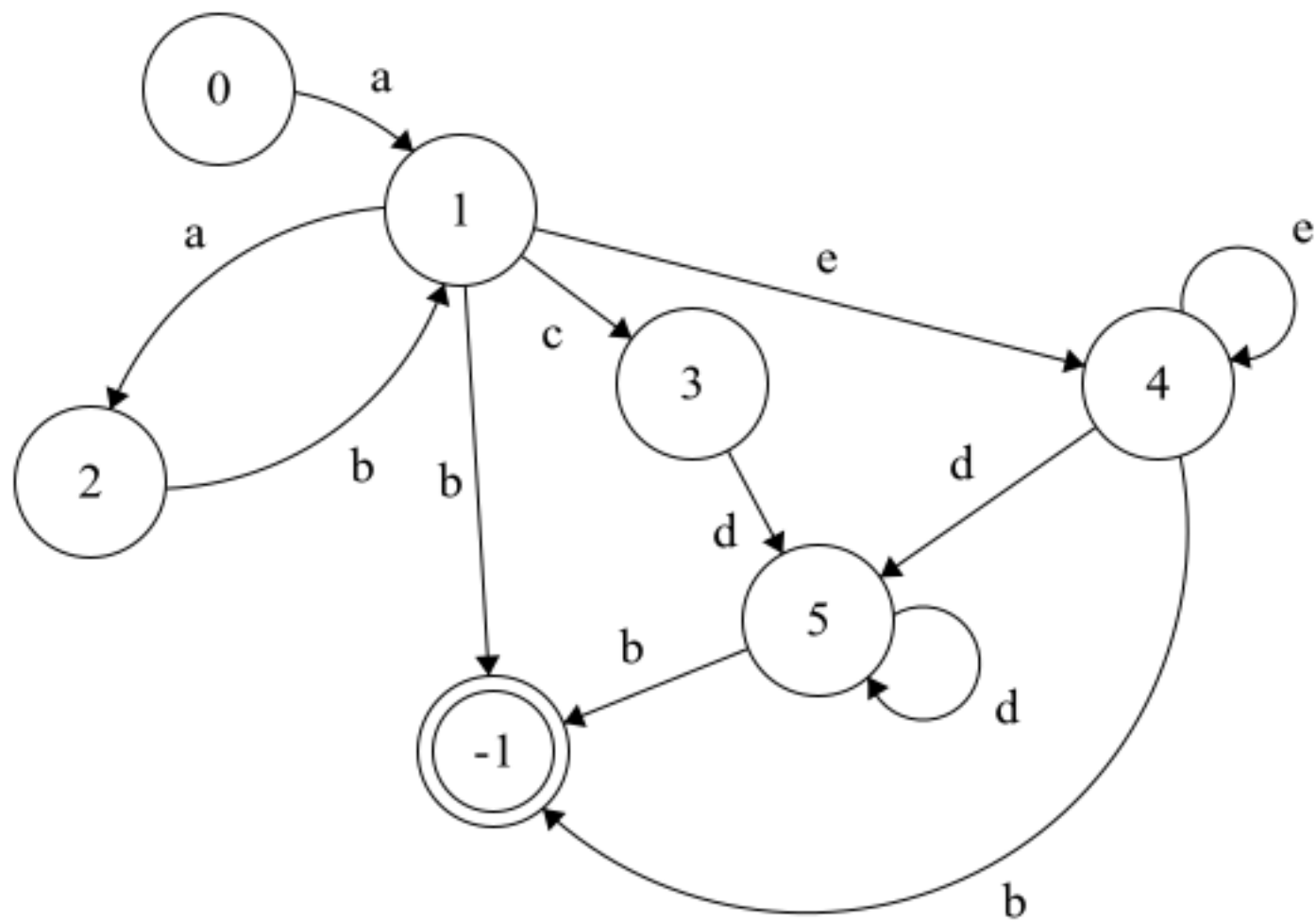
a frase "aababcbddb" é válida na linguagem  $L(e1)$ ;

Considere as expressões regulares (ER)

'e1 = a (a b)\* (c d+ | e\*) b'

'e2 = a (b a)\* (c d+ b | e\* b)'

e o autômato finito determinista (AFD) A1, com estado inicial 0 e estado final -1,



Considere os Terminais `NInt` (número inteiro), `NReal` (número decimal) e `Pal` (sequência de uma ou mais letras) e a seguinte Gramática Independente de Contexto (G3):

```
Frase -> '[' Elems ']'
Elems -> ε
Elems -> Elem Elems
Elem  -> NInt
      | NReal
      | Pal
      | Frase
```

Selecione então as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. A especificação `ply.lex` para gerar o Analisador Léxico compatível com o Parser gerado pelo `ply.yacc` para reconhecer `L(G3)` tem de conter pelo menos a seguinte regra:

```
t_NInt = r'[a-zA-Z]'
```

- ☐ b. A especificação `ply.lex` para gerar o Analisador Léxico compatível com o Parser gerado pelo `ply.yacc` para reconhecer `L(G3)` tem de conter pelo menos as seguintes declarações:

```
import ply.lex as lex
tokens = ( 'NInt', 'NReal', 'Pal' )
literals = ( '[', ']' )
```

- ☐ c. A especificação `ply.lex` para gerar o Analisador Léxico compatível com o Parser gerado pelo `ply.yacc` para reconhecer `L(G3)` não pode conter a seguinte declaração, sob pena de nunca derivar/seguir a 2ª produção:

```
t_ignore = ' \n\t'
```

- ☐ d. A especificação `ply.lex` para gerar o Analisador Léxico compatível com o Parser gerado pelo `ply.yacc` para reconhecer `L(G3)` tem de conter pelo menos a seguinte regra:

```
def t_Pal:
    r'[a-zA-Z]+'
```



the Karen we need



Miriam Miranda Pinto .

## Pergunta 6

Considere a GIC abaixo cujo axioma (ou símbolo inicial) é Trafego e o símbolo '&' representa o vazio.

```
T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }  
NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }  
P = {  
  p1: Trafego -> data Partidas Chegadas  
  p2: Partidas -> PARTIDAS Aviao Avioes  
  p3: Avioes -> '|' Aviao Avioes  
  p4:      | &  
  p5: Chegadas -> CHEGADAS Aviao Avioes  
  p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora  
  p7: NumVoo -> id  
  p8: Companh -> str  
  p9: OrigDest -> str  
}
```

e escolha as afirmações seguintes que são verdadeiras:

- ☐ a. A gramática não satisfaz a Condição LL(1) por ter recursividade à esquerda.
- ☐ b. A gramática não satisfaz a Condição LL(1) por ter mais do que 1 produção com o mesmo Lookahead.
- ☒ c. A gramática satisfaz a Condição LL(1) apesar de ter recursividade.
- ☒ d. Era possível desenvolver um Parser TD do tipo Recursivo-descendente para reconhecer as frases da respetiva linguagem.



## Pergunta 7

Considere o fragmento de GIC abaixo cujo axioma (ou símbolo inicial) é Vendas.

```
T = { '.', LOJA, STOCK, FATURAS, id, num, string }

NT = { Vendas, Nome, Stock, Prods, Prod, CodProd, Qt, Preço, Faturas, Fats }

p0:  Vendas      -->  LOJA Nome Stock Faturas '.'
p1:  Nome        -->  string
p2:  Stock       -->  STOCK Prods
p3:  Prods       -->  Prod
p4:              |    Prods Prod
p5:  Prod        -->  CodProd Qt Preço '.'
p6:  CodProd     -->  id
p8:  Qt          -->  num
p9:  Preço       -->  num
p10: Faturas     -->
p11:              |    FATURAS Fats
p12: Fats        -->  .....
```

e escolha as afirmações seguintes que são verdadeiras:

- ☐ a. `follow(Vendas) = { '.' }`
- ☐ b. Na GIC acima as produções p10 e p11 não verificam as Condição LL(1).
- ☒ c. A GIC acima não satisfaz a Condição LL(1).
- ☐ d. Na GIC acima não são aceites frases vazias

```

%{
#include "y.tab.h"

%}

%%

[ \t\n\r] ;

\(\ return yytext[0];

\) return yytext[0];

(\+|-)?[0-9]+ { yylval.valor = strdup(yytext); return num; }

. return ERRO;

```

## Analizador Sintático

```

%{
#include <stdio.h>
extern int yylex();
int yyerror();
%}

%union{
    char *valor ;
}

%token ERRO num
%type <valor> num ABin
%%

S
: ABin { printf("%s\n", $1 ); }
;

ABin
: '(' ')' { $$ = ""; }
| num { asprintf( &$$, "%s ", $1); }
| '(' num ABin ABin ')' { asprintf( &$$, "%s %s %s ", $3, $4, $2); }
;

%%

int main(){
    yyparse() ;
    return 0 ;
}

int yyerror(){
    printf("Erro sintático...") ;
    return 0 ;
}

```

Depois de analisá-los, assinale quais as afirmações verdadeiras.

- ☐ a. O analisador sintático apresentado não tem nenhum conflito no estado inicial do seu autômato LR(0).
- ☐ b. O resultado que se obtém para a frase
- (12 (8 () ()) (27 (16 () ()) (33 () ())))
- é a lista de números:
- 8 12 16 27 33
- ☐ c. A seguinte frase:
- (12 (8 () ()) (27 16 33))
- está mal construída e o parser irá dar erro.
- ☐ d. A gramática apresentada não pode ser reconhecida por um parser Recursivo Descendente.

↪ ⚠ Se passar para a pergunta seguinte, as alterações desta resposta não são guardadas.

Pergunta 6

Considere a GIC abaixo cujo axioma (ou símbolo inicial) é `Trafego` e o símbolo `'&'` representa o vazio.

```
T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }

NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }

P = {

p1: Trafego -> data Partidas Chegadas

p2: Partidas -> PARTIDAS Aviao Avioes

p3: Avioes -> '|' Aviao Avioes
p4:         | &

p5: Chegadas -> CHEGADAS Aviao Avioes

p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora

p7: NumVoo -> id

p8: Companh -> str

p9: OrigDest -> str

}
```

nas afirmações seguintes, considere que os dois índices da tabela correspondem ao símbolo NT que se está a derivar e ao valor do próximo símbolo.

Considere que a gramática apresentada foi estendida com a produção:

```
Z -> Trafego '$'
```

Assinale as que são verdadeiras:

- ☐ a. TabelaLL1[ Avioes, '|' ] = p3 + p4 (conflito)
- ☐ b. TabelaLL1[ Avioes, \$ ] = p4
- ☒ c. TabelaLL1[ Avioes, CHEGADAS ] = p4
- ☒ d. TabelaLL1[ Companh, str ] = p8 + p9 (conflito)

↪ ⚠ Se passar para a pergunta seguinte, as alterações desta resposta não são guardadas.

```

T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }

NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }

P = {

p1: Trafego -> data Partidas Chegadas

p2: Partidas -> PARTIDAS Aviao Avioes

p3: Avioes -> '|' Aviao Avioes
p4:         | &

p5: Chegadas -> CHEGADAS Aviao Avioes

p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora

p7: NumVoo -> id

p8: Companh -> str

p9: OrigDest -> str

}

```

e escolha as afirmações seguintes que são verdadeiras:

- ☐ a. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Paser RD para a GIC acima

```

rec_Chegadas( proxSimb )
{
  recTerm( CHEGADAS, proxSimb );
  rec_Aviao( proxSimb );
  rec_Avioes( proxSimb );
}

```

- ☐ b. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Paser RD para a GIC acima

```

rec_Partidas( proxSimb )
{
  se (proxSimb == PARTIDAS)
    entao { recTerm( PARTIDAS, proxSimb );
            rec_Aviao( proxSimb );
            rec_Avioes( proxSimb ); }
  senão { erroSintaxe() }
}

```

- ☐ c. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Paser RD para a GIC acima

```

rec_Trafego( proxSimb )
{
  se (proxSimb == data)
    entao { rec_Partidas( proxSimb );
            rec_Chegadas( proxSimb ); }
  senão { erroSintaxe() }
}

```

- ☐ d. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Paser RD para a GIC acima

```

rec_Aviao( proxSimb )
{
  se (proxSimb == NumVoo)
    entao { rec_NumVoo( proxSimb );
            rec_Companh( proxSimb );
            rec_OrigDest( proxSimb );
            rec_hora( proxSimb ); }
  senão { erroSintaxe() }
}

```



Este Teste não permite ação retroativa. São proibidas alterações nas respostas após o envio.



**Tempo restante: 15 minutos, 02 segundos.**

⌵ **Estado de Conclusão da Pergunta:**



⚠ Se passar para a pergunta seguinte, as alterações desta resposta não são guardadas.

## Pergunta 4

Considere o fragmento de GIC abaixo cujo axioma (ou símbolo inicial) é Vendas.

```
T = { '.', LOJA, STOCK, FATURAS, id, num, string }

NT = { Vendas, Nome, Stock, Prods, Prod, CodProd, Qt, Preço, Faturas, Fats }

p0:  Vendas      -->  LOJA Nome Stock Faturas '.'
p1:  Nome        -->  string
p2:  Stock       -->  STOCK Prods
p3:  Prods       -->  Prod
p4:              |   Prods Prod
P5:  Prod        -->  CodProd Qt Preço '.'
p6:  CodProd     -->  id
p8:  Qt          -->  num
p9:  Preço       -->  num
p10: Faturas     -->
p11:              |   FATURAS Fats
p12: Fats        -->  .....
```

e escolha as afirmações seguintes que são verdadeiras:

- ☒ a. A GIC acima não satisfaz a Condição LL(1).
- ☐ b. Na GIC acima as produções p10 e p11 não verificam as Condição LL(1).
- ☐ c. `follow(Vendas) = { '.' }`
- ☐ d. Na GIC acima não são aceites frases vazias

Considere a GIC abaixo cujo axioma (ou símbolo inicial) é Trafego e o símbolo '&' representa o vazio.

```
T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }  
NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }  
P = {  
  p1: Trafego -> data Partidas Chegadas  
  p2: Partidas -> PARTIDAS Aviao Avioes  
  p3: Avioes -> '|' Aviao Avioes  
  p4:      | &  
  p5: Chegadas -> CHEGADAS Aviao Avioes  
  p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora  
  p7: NumVoo -> id  
  p8: Companh -> str  
  p9: OrigDest -> str  
}
```

e escolha as afirmações seguintes que são verdadeiras:

- ☐ a. Era possível desenvolver um Parser TD do tipo Recursivo-descendente para reconhecer as frases da respetiva linguagem.
- ☐ b. A gramática não satisfaz a Condição LL(1) por ter mais do que 1 produção com o mesmo Lookahead.
- ☒ c. A gramática satisfaz a Condição LL(1) apesar de ter recursividade.
- ☐ d. A gramática não satisfaz a Condição LL(1) por ter recursividade à esquerda.

## Analizador Léxico

```
%{
#include "y.tab.h"

%}

%%

[ \t\n\r] ;

\(\( return yytext[0];

\) return yytext[0];

(\+|-)?[0-9]+ { yylval.valor = strdup(yytext); return num; }

. return ERRO;
```

## Analizador Sintático

```
%{
#include <stdio.h>
extern int yylex();
int yyerror();
%}

%union{
    char *valor ;
}
%token ERRO num
%type <valor> num ABin
%%

S
: ABin { printf("%s\n", $1 ); }
;

ABin
: '(' ')' { $$ = ""; }
| num { asprintf( &$$, "%s ", $1); }
| '(' num ABin ABin ')' { asprintf( &$$, "%s %s %s ", $3, $4, $2); }
;

%%

int main(){
    yyparse() ;
    return 0 ;
}

int yyerror(){
    printf("Erro sintático...") ;
    return 0 ;
}
```

Depois de analisá-los, assinale quais as afirmações verdadeiras.

- ☐ a. O resultado que se obtém para a frase

(12 (8 () ()) (27 (16 () ()) (33 () ())))

é a lista de números:

8 12 16 27 33

- ☐ b. O analisador sintático apresentado não tem nenhum conflito no estado inicial do seu autômato LR(0).

- ☐ c. A seguinte frase:

(12 (8 () ()) (27 16 33))

está mal construída e o parser irá dar erro.

- ☐ d. A gramática apresentada não pode ser reconhecida por um parser Recursivo Descendente.

Analizador Sintático

```
%{
#include <stdio.h>
extern int yylex();
int yyerror();
%}
%union{
    char *valor ;
}
%token ERRO num
%type <valor>  num ABin
%%
S
    : ABin { printf("%s\n", $1 ); }
    ;

ABin
    : '(' ')' { $$ = ""; }
    | num { asprintf( &$$, "%s ", $1); }
    | '(' num ABin ABin ')' { asprintf( &$$, "%s %s %s ", $3, $4, $2); }
    ;
%%
int main(){
    yyparse() ;
    return 0 ;
}
int yyerror(){
    printf("Erro sintático...") ;
    return 0 ;
}
```

Depois de analisá-los, assinale quais as afirmações verdadeiras.

- ☐ a. A gramática apresentada não pode ser reconhecida por um parser Recursivo Descendente.
- ☐ b. A seguinte frase:  

(12 (8 () ())) (27 16 33))

  
está mal construída e o parser irá dar erro.
- ☐ c. O analisador sintático apresentado não tem nenhum conflito no estado inicial do seu autômato LR(0).
- ☐ d. O resultado que se obtem para a frase  

(12 (8 () ())) (27 (16 () ())) (33 () ()))

  
é a lista de números:  

8 12 16 27 33



Considere o seguinte excerto dum módulo em Python que faz o reconhecimento de uma árvore genealógica.

Preencha as ações semânticas necessárias de modo a que o parser escreva o número total de nomes contidos na árvore genealógica.

Com a entrada: `Ana( -, Rui (Maria (-,-), Joao(-, -)) )` deverá escrever **4**

**Não insira espaços em branco desnecessários** nas expressões que vai escrever.

```
def p_Axioma(p):  
    "Axioma : GenT"  
    print()
```

```
def p_GenT_empty(p):  
    "GenT : '-'"  
    
```

```
def p_GenT(p):  
    "GenT : name '(' GenT ',' GenT ')'"  
    
```

Considere a GIC abaixo cujo axioma (ou símbolo inicial) é Trafego e o símbolo '\$' representa o vazio.

```
T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }  
NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }  
P = {  
  p1: Trafego -> data Partidas Chegadas  
  p2: Partidas -> PARTIDAS Aviao Avioes  
  p3: Avioes -> '|' Aviao Avioes  
  p4:      | &  
  p5: Chegadas -> CHEGADAS Aviao Avioes  
  p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora  
  p7: NumVoo -> id  
  p8: Companh -> str  
  p9: OrigDest -> str  
}
```

nas afirmações seguintes, considere que os dois índices da tabela correspondem ao símbolo NT que se está a derivar e ao valor do próximo símbolo.

Considere que a gramática apresentada foi estendida com a produção:

Z -> Trafego '\$'

Assinale as que são verdadeiras:

- ☐ a. TabelaLL1[ Avioes, '|' ] = p3 + p4 (conflito)
- ☐ b. TabelaLL1[ Companh, str ] = p8 + p9 (conflito)
- ☐ c. TabelaLL1[ Avioes, CHEGADAS ] = p4
- ☐ d. TabelaLL1[ Avioes, \$ ] = p4

```

        então { rec_Partidas( proxSimb );
                rec_Chegadas( proxSimb ) }
        senão { erroSintaxe() }
    }

```

- ☐ c. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Paser RD para a GIC acima

```

rec_Partidas( proxSimb )
{ se (proxSimb == PARTIDAS)
    então { recTerm( PARTIDAS, proxSimb );
            rec_Aviao( proxSimb );
            rec_Aviones( proxSimb ) }
    senão { erroSintaxe() }
}

```

- ☐ d. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Paser RD para a GIC acima

```

rec_Chegadas( proxSimb )
{ recTerm( CHEGADAS, proxSimb );
  rec_Aviao( proxSimb );
  rec_Aviones( proxSimb );
}

```

Considere a GIC abaixo cujo axioma (ou símbolo inicial) é Trafego e o símbolo 'ε' representa o vazio.

```
T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }

NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }

P = {

p1: Trafego -> data Partidas Chegadas

p2: Partidas -> PARTIDAS Aviao Avioes

p3: Avioes -> '|' Aviao Avioes
p4:         | ε

p5: Chegadas -> CHEGADAS Aviao Avioes

p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora

p7: NumVoo -> id

p8: Companh -> str

p9: OrigDest -> str

}
```

e escolha as afirmações seguintes que são verdadeiras:

- ☒ a. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Parser RD para a GIC acima

```
rec_Aviao( proxSimb )
{ se (proxSimb == NumVoo)
    entao { rec_NumVoo( proxSimb );
            rec_Companh( proxSimb );
            rec_OrigDest( proxSimb );
            rec_hora( proxSimb ) }
    senão { erroSintaxe() }
}
```

- ☐ b. O código seguinte representa corretamente uma função de reconhecimento de um símbolo não terminal dum Parser RD para a GIC acima

```
rec_Trafego( proxSimb )
{ se (proxSimb == data)
    entao { rec_Partidas( proxSimb );
            rec_Chegadas( proxSimb ) }
    senão { erroSintaxe() }
```

Considere a GIC abaixo cujo axioma (ou símbolo inicial) é `Trafego` e o símbolo `'&'` representa o vazio.

`T = { '|', '-', ':', PARTIDAS, CHEGADAS, data, id, str, hora }`

`NT = { Trafego, Partidas, Chegadas, Avioes, Aviao, NumVoo, Companh, Origem }`

`P = {`

`p1: Trafego -> data Partidas Chegadas`

`p2: Partidas -> PARTIDAS Aviao Avioes`

`p3: Avioes -> '|' Aviao Avioes`

`p4:           | &`

`p5: Chegadas -> CHEGADAS Aviao Avioes`

`p6: Aviao -> NumVoo '-' Companh '-' OrigDest ':' hora`

`p7: NumVoo -> id`

`p8: Companh -> str`

`p9: OrigDest -> str`

`}`

e escolha as afirmações seguintes que são verdadeiras:

☐ a. `first( Aviao ) = {id}`

☐ b. `follow( Partidas ) = {CHEGADAS}`

☐ c. `follow( Avioes ) = {}`

☐ d. `lookahead( p4 ) = {}`

# Pergunta 1

Considere o fragmento de GIC abaixo cujo axioma (ou símbolo inicial) é Vendas.

`T = {'.', LOJA, STOCK, FATURAS, id, num, string }`

`NT = { Vendas, Nome, Stock, Prods, Prod, CodProd, Qt, Preco, Faturas, Fats }`

`p0: Vendas --> LOJA Nome Stock Faturas .'`

`p1: Nome --> string`

`p2: Stock --> STOCK Prods`

`p3: Prods --> Prod`

`p4:           | Prods Prod`

`p5: Prod --> CodProd Qt Preco .'`

`p6: CodProd --> id`

`p8: Qt --> num`

`p9: Preco --> num`

`p10: Faturas -->`

`p11:           | FATURAS Fats`

`p12: Fats --> .....`

e escolha as afirmações seguintes que são verdadeiras:

- ☐ a. A GIC acima não satisfaz a Condição LL(1).
- ☐ b. Na GIC acima as produções p10 e p11 não verificam as Condição LL(1).
- ☐ c. `follow(Vendas) = {'.', }`
- ☐ d. Na GIC acima não são aceites frases vazias

```
%{
char acumul[1000]; int i=0;

void proc(char* texto);
%}
%x SA
%%
"\caption{"      { BEGIN SA; }
"}"              { acumul[i++]='\0'; proc(acumul); BEGIN INITIAL; }
.|\\n            { acumul[i++]=yytext[0]); }
.|\\n            { ; }
%%
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☒ a. O filtro gerado a partir duma especificação semelhante à original mas em que a 3ª regra seja trocada por:
 

```
[^}]+      { acumul = strdup( yytext); }
```

 comporta-se da mesma maneira que o original.
- ☒ b. O filtro gerado a partir duma especificação semelhante à original mas em que se retira (elimina) a 4ª regra comporta-se da mesma maneira que o original.
- ☐ c. O filtro gerado a partir desta especificação apenas processa as legendas (caption) de figuras em LaTeX copiando para a saída todo o restante texto.
- ☐ d. O filtro gerado a partir desta especificação apenas processa as legendas (caption) de figuras em LaTeX ignorando todo o restante texto.



Considere os Terminais "str" (texto entre aspas), "texto" (sequência de caracteres) e "id" (sequência não nula de letras) e a seguinte Gramática Independente de Contexto (G4):

Anota -> Abre texto Fecha

Abre -> '<' id '>'

      | '<' id LstA '>'

Fecha -> '<' '/' id '>'

LstA -> Atr

LstA -> LstA Atr

Atr -> id '=' str

e o seguinte texto de entrada:

<data evento="nasce" norma="20001012">dez de dezembro</data>

Averigue então a veracidade das seguintes afirmações:

- ☐ a. Para reconhecer frases desta linguagem, o Analisador Léxico teria de recorrer ao uso de 'states' (estados internos, ou start-conditions) para conseguir reconhecer o símbolo terminal "texto".
- ☐ b. Ao reconhecer a frase, a árvore de derivação teria 14 folhas (símbolos terminais retornados pelo analisador léxico) mas ao fim de construir 2 nodos intermédios não se conseguia chegar à raiz por existirem erros sintáticos.
- ☐ c. Ao reconhecer a frase, e após construir o nodo intermédio 'Atr' não se conseguia chegar à raiz por existirem ambiguidades derivadas do símbolo 'LstA' ter 2 alternativas.
- ☐ d. Ao reconhecer a frase, a árvore de derivação teria 14 folhas (símbolos terminais retornados pelo analisador léxico) e teria 7 nodos intermédios incluindo a raiz que seria etiquetada pelo símbolo 'Anota'.



Considere a seguinte especificação Flex:

```
%%  
==      { printf(".EQ."); }  
~=      { printf("!="); }  
=\\<    { printf("<="); }  
%%
```

e escolha as afirmações verdadeiras:

- ☐ a. O Filtro gerado pelo Flex a partir especificação acima, transforma o texto

`a ~= b+c;`

em

`a != b+c;`

- ☐ b. Se acrescentássemos à especificação uma nova 4ª regra:

```
[a-z][a-zA-Z0-9]+ { printf(".ID."); }
```

o resultado do Filtro gerado pelo Flex a partir dessa nova especificação, aplicado ao texto:

`a ~= b+c;`

não se alterava, mas em geral o seu comportamento é diferente.

- ☐ c. Se a 2ª regra da especificação acima fosse alterada para:

```
~/= { printf("!="); }
```

o resultado do Filtro gerado pelo Flex a partir dessa nova especificação, aplicado ao texto:

`a ~= b+c;`

não se alterava.

- ☐ d. O Filtro gerado pelo Flex a partir especificação acima, transforma o texto

`a ~= b+c;`

em

`a ~.EQ. b+c;`

Considere a expressão regular (ER):

$$e = (a a b)^+ c d$$

e selecione as alíneas abaixo que são afirmações verdadeiras:

- ☐ a. O Autômato Determinista 'AD' equivalente a 'e' tem apenas 1 saída, por 'a', a partir do seu estado inicial.
- ☐ b. O Autômato Determinista 'AD' equivalente a 'e' tem no mínimo 6 estados sendo 1 final.
- ☐ c. O Autômato Determinista 'AD' equivalente a 'e' pode ter 1 saída por 'a' e outra por 'c' a partir do seu estado inicial.
- ☐ d. O Autômato Determinista 'AD' equivalente a 'e' tem exatamente 5 estados sendo 1 final.

Num determinado mundo, os números de telefone são constituídos por 3 partes:

- código do país: 1 a 3 dígitos;
- código da região: 1 a 3 dígitos;
- nº de telefone: 4 a 10 dígitos.

Sabendo que o separador entre a 1ª e a 2ª parte e entre a 2ª e a 3ª parte é o hífen ('-') ou o espaço, das expressões regulares que se apresentam a seguir assinale aquelas que fariam match com os números de telefone:

☐ a. `[0-9]{1,3}[\ -][0-9]{1,3}[\ -][0-9]{4,10}`

☐ b. `[0-9][0-9]?[0-9]?[\ -][0-9]?[0-9]?[0-9][\ -][0-9][0-9][0-9][0-9]{1,7}`

☐ c. `([0-9]{1,3}(\ |-)){2}[0-9]{4,10}`

☐ d. `(0|1|2|3|4|5|6|7|8|9){1,3}(\ |-)(0|1|2|3|4|5|6|7|8|9){1,3}(\ |-)(0|1|2|3|4|5|6|7|8|9){4,10}`

### Pergunta 3

Considere o seguinte analisador léxico especificado em flex:

```
%{
#define num 1001
#define ERRO -1
%}
%%

[ \t\n\r] ;
\(      return yytext[0];
\)      return *yytext;
\,      return *yytext;

(\+|-)?([0-9]|[1-9][0-9]|1[0-9][0-9])(\.[0-9]+)?/[ \t\n\r,)] return num;
.      return ERRO;
```

Faça corresponder cada um dos seguintes textos de input à respectiva sequência de tokens gerada pelo analisador.

#### Pergunta

#### Correspondência Correta

(77.11112223331, 149.99999999) ☒ b. '(' 1001 ',' 1001 ')'

(  
345  
,  
-8.99  
) ☒ f. '(' -1 1001 ',' 1001 ')'

(0.0),  
(0.0) ☒ d. '(' 1001 ')', '(' 1001 ')'

-9.9  
120.5  
4450  
(34) ☒ a. 1001 1001 -1 -1 1001 '(' 1001 ')'

## Pergunta 1

Considere as expressões regulares e1 e e2 (ignorando os espaços que foram lá colocados para legibilidade):

$$e1 = (a a b)^+ c (d \mid a b f)^* j$$
$$e2 = (a a b)^+ c (d^* \mid a b f^*) j$$

e **selecione as alíneas** abaixo **que são afirmações verdadeiras**:

Respostas:

a. Como a frase "aabaabcj" é válida nas linguagens  $L(e1)$  e  $L(e2)$  então conclui-se que 'e1' é equivalente a 'e2'.

As ER 'e2' e

$$e5 = (a a b)^+ (c d^* j \mid c a b f^* j)$$

☒ b. são equivalentes porque definem exatamente a mesma linguagem.

A ER 'e1' pode escrever-se mais abreviadamente na forma

$$e1 = (a b)^+ c (d \mid a b f)^* j$$

sem alterar a linguagem gerada.

c.

As ER 'e2' e

$$e5 = (a a b)^+ (c d^* j \mid c a b f^* j)$$

☒ d. geram exatamente as mesmas frases e por isso conclui-se que são ER equivalentes.

Considere os Terminais "str" (texto entre aspas), "texto" (sequência de caracteres) e "id" (sequência não nula de letras) e a seguinte Gramática Independente de Contexto (G4):

```
Anota -> Abre texto Fecha
Abre  -> '<' id '>'
      | '<' id LstA '>'
Fecha -> '<' '/' id '>'
LstA  -> Atr
LstA  -> LstA Atr
Atr   -> id '=' str
```

e o seguinte texto de entrada:

```
<data evento="nasce" norma="20001012">dez de dezembro</data>
```

Averigue então a veracidade das seguintes afirmações:

- ☐ a. Ao reconhecer a frase, e após construir o nodo intermédio 'Atr' não se conseguia chegar à raiz por existirem ambiguidades derivadas do símbolo 'LstA' ter 2 alternativas.
- ☐ b. Ao reconhecer a frase, a árvore de derivação teria 14 folhas (símbolos terminais retornados pelo analisador léxico) mas ao fim de construir 2 nodos intermédios não se conseguia chegar à raiz por existirem erros sintáticos.
- ☐ c. Ao reconhecer a frase, a árvore de derivação teria 14 folhas (símbolos terminais retornados pelo analisador léxico) e teria 7 nodos intermédios incluindo a raiz que seria etiquetada pelo símbolo 'Anota'.
- ☐ d. Para reconhecer frases desta linguagem, o Analisador Léxico teria de recorrer ao uso de 'states' (estados internos, ou start-conditions) para conseguir reconhecer o símbolo terminal "texto".

Considere o seguinte excerto de um ficheiro que implementa um parser que reconhece uma sequência de intervalos fechados de números inteiros. Por exemplo: [1, 2] [4, 1] [2, 5] [5, 6]

```
def p_sequencia(p):
    "sequencia : intervalos"
    print(p[1])

def p_intervalos_intervalo(p):
    "intervalos : intervalo"
    p[0] = p[1]

def p_intervalos_intervalos(p):
    "intervalos : intervalos intervalo"
    p[0] = p[2] if p[1] < p[2] else p[1]

def p_intervalo(p):
    "intervalo : '[' NUM ',' NUM ']'"
    p[0] = p[4] - p[2] if p[2] <= p[4] else -1
```

Selecione as alíneas seguintes que são verdadeiras.

- ☐ a. O parser imprime a largura do menor intervalo da sequência, caso haja pelo menos um intervalo válido.
- ☐ b. Se o texto de entrada for '[3, 1] [1, 4] [2, 3]' o parser deverá escrever '3'.
- ☐ c. Se o texto de entrada for '[3, 3] [4, 1] [5, 2]' o parser deverá escrever '3'.
- ☐ d. No caso do texto de entrada apenas conter intervalos inválidos, i.e. o limite inferior é maior que o limite superior, o parser escreverá '-1'.



Considere os Terminais "str" (texto entre aspas), "texto" (sequência de caracteres) e "id" (sequência não nula de letras) e a seguinte Gramática Independente de Contexto (G4):

```
Anota -> Abre texto Fecha
Abre  -> '<' id '>'
      | '<' id LstA '>'
Fecha -> '<' '/' id '>'
LstA  -> Atr
LstA  -> LstA Atr
Atr   -> id '=' str
```

e o seguinte texto de entrada:

```
<data evento="nasce" norma="20001012">dez de dezembro</data>
```

Averigue então a veracidade das seguintes afirmações:

- ☐ a. Ao reconhecer a frase, a árvore de derivação teria 14 folhas (símbolos terminais retornados pelo analisador léxico) mas ao fim de construir 2 nodos intermédios não se conseguia chegar à raiz por existirem erros sintáticos.
- ☐ b. Ao reconhecer a frase, e após construir o nodo intermédio 'Atr' não se conseguia chegar à raiz por existirem ambiguidades derivadas do símbolo 'LstA' ter 2 alternativas.
- ☐ c. Ao reconhecer a frase, a árvore de derivação teria 14 folhas (símbolos terminais retornados pelo analisador léxico) e teria 7 nodos intermédios incluindo a raiz que seria etiquetada pelo símbolo 'Anota'.
- ☐ d. Para reconhecer frases desta linguagem, o Analisador Léxico teria de recorrer ao uso de 'states' (estados internos, ou start-conditions) para conseguir reconhecer o símbolo terminal "texto".



Considere o seguinte excerto dum módulo em Python que faz o reconhecimento de uma árvore genealógica.

Preencha as ações semânticas necessárias de modo a que o parser escreva o número total de nomes contidos na árvore genealógica.




Com a entrada: `Ana( -, Rui (Maria (-,-), Joao(-, -)))` deverá escrever **4**

Não insira espaços em branco desnecessários nas expressões que vai escrever.

```
def p_Axioma(p):
    "Axioma : GenT"
    print([Exp1])

def p_GenT_empty(p):
    "GenT : '-'"
    [Exp2]

def p_GenT(p):
    "GenT : name '(' GenT ',' GenT ')'"
    [Exp3]
```

- Resposta Especificada para Exp1  p[1]
- Resposta Especificada para Exp2  pass
- Resposta Especificada para Exp3  p[0]=(p[3],p[5])

Respostas Corretas para Exp1		
Método de avaliação	Resposta Correta	Diferenciação de maiúsculas e minúsculas
 Correspondência Exata	p[1]	
Respostas Corretas para Exp2		
Método de avaliação	Resposta Correta	Diferenciação de maiúsculas e minúsculas
 Correspondência Exata	p[0]=0	
Respostas Corretas para Exp3		
Método de avaliação	Resposta Correta	Diferenciação de maiúsculas e minúsculas
 Correspondência Exata	p[0]=1+p[3]+p[5]	

Considere a seguinte gramática independente de contexto que reconhece determinadas frases compostas por sequências de 0's e 1's:

$$\begin{array}{l} S \rightarrow 0 S 1 S \\ \quad | 1 S 0 S \\ \quad | \epsilon \end{array}$$

Das frases seguintes assinale aquelas que seriam consideradas válidas de acordo com esta gramática.

- ☐ a. 00001111
- ☐ b. 00101101
- ☐ c. 01
- ☐ d. 00110001

## Pergunta 8

6,6 em 10 pontos

Considere os Terminais `NInt` (número inteiro), `NReal` (número decimal) e `PAL` (sequência de uma ou mais letras) e a seguinte Gramática Independente de Contexto (G1) :

```
Frase -> '[' Elms ']'
Elms  -> ε
Elms  -> Elem Elms
Elem  -> NInt
      | NReal
      | Pal
      | Frase
```

Selecione então as alíneas abaixo que são afirmações verdadeiras:

- Respostas Selecionadas: ☒ a. A frase "[9.1 [2 [a] 3 [43.1 88] ]]" pertence à linguagem L(G3) gerada por esta gramática.
- ☒ c. Para a frase "9 0.2 abs" pertencer à linguagem L(G3) gerada por esta gramática era obrigatório estar envolvida em parêntesis retos .
- Respostas Corretas: ☒ a. A frase "[9.1 [2 [a] 3 [43.1 88] ]]" pertence à linguagem L(G3) gerada por esta gramática.
- ☒ b. A frase "[[ [ ABC ] ]]" pertence à linguagem L(G3) gerada por esta gramática.
- ☒ c. Para a frase "9 0.2 abs" pertencer à linguagem L(G3) gerada por esta gramática era obrigatório estar envolvida em parêntesis retos .

Os seguintes textos são frases válidas de uma linguagem DTDL baseada no alfabeto  $T=\{'<!', '>', '(', ')', ',', '|', '+', '*', \text{ELEMENT}, \text{PC}, \text{id}\}$

```
<!ELEMENT id (PC)>
<!ELEMENT id (id, id)>
<!ELEMENT id ((id, id) | PC)>
<!ELEMENT id ( id+, id*, id )>
<!ELEMENT id ( id, (PC|id)+)>
```

Escolha então as afirmações verdadeiras:

☐ a. A DTDL pode ser definida pela seguinte GIC:

```
Dtd  -> '<!' Elem '>'
Elem -> ELEMENT NomeE '(' Regra ')'
Regra -> T
      | Regra Oper T
Oper  -> ','
      | '|'
T     -> F
      | F '*'
      | F '+'
F     -> PC
      | id
      | '(' Regra ')'
NomeE -> id
```

☐ b. Ignorar esta alínea.

☐ c. A DTDL pode ser definida pela seguinte GIC:

```
Dtd  -> Cabec DefE Fecho
Cabec -> '<!' ELEMENT
Fecho -> '>'
DefE  -> NomeE Regra
Regra -> '(' Exp ')'
Exp   -> T RE
RE    -> Oper T RE
      | ε
Oper  -> ','
      | '|'
T     -> PC
      | id
      | Regra
      | T '*'
      | T '+'
NomeE -> id
```

☐ d. Ignorar esta alínea.

Considere o seguinte programa em Python:

```
import sys
import re

for linha in sys.stdin:
    segments = re.findall(r'(00+)|(11+)|(1)|(0)', linha)
    for (zs, us, u, z) in segments:
        if zs:
            print( str(len(zs)) + '0', end='')
        elif us:
            print( str(len(us)) + '1', end='')
        elif u:
            print('1', end='')
        elif z:
            print('0', end='')
        else:
            pass

    print('\n', end='')
print('\n', end='')
```

e indique quais das alíneas seguintes são verdadeiras:

Respostas Seleccionadas: ☒ a. Se o texto de input tivesse uma única linha com o conteúdo "0101010101" o resultado seria "0101010101";

Respostas: ☒ a. Se o texto de input tivesse uma única linha com o conteúdo "0101010101" o resultado seria "0101010101";

☒ b. Se a expressão regular fosse alterada para `r' (00*) | (11*) | (1) | (0) '`, o resultado final para uma linha com conteúdo "0001110011" não se alterava;

c. Se a expressão regular fosse alterada para `r' (00*) | (11*) | (1) | (0) '`, o resultado final para uma linha com conteúdo "010" não se alterava;

☒ d. Se o texto de input tivesse uma única linha com o conteúdo "101110101111001" o resultado seria "103101041201";

Considere o seguinte excerto de um ficheiro que implementa um parser que reconhece uma sequência de intervalos fechados de números inteiros. Por exemplo: [1,2] [4,1] [2,5] [5,6]

```
def p_sequencia(p):  
    "sequencia : intervalos"  
    print(p[1])  
  
def p_intervalos_intervalo(p):  
    "intervalos : intervalo"  
    p[0] = p[1]  
  
def p_intervalos_intervalos(p):  
    "intervalos : intervalos intervalo"  
    p[0] = p[2] if p[1] < p[2] else p[1]  
  
def p_intervalo(p):  
    "intervalo : '[' NUM ',' NUM ']'"  
    p[0] = p[4] - p[2] if p[2] <= p[4] else -1
```

Selecione as alíneas seguintes que são verdadeiras.


- ☐ a. Se o texto de entrada for '[3,1] [1,4] [2,3]' o parser deverá escrever '3'.
- ☐ b. Se o texto de entrada for '[3,3] [4,1] [5,2]' o parser deverá escrever '3'.
- ☐ c. No caso do texto de entrada apenas conter intervalos inválidos, i.e. o limite inferior é maior que o limite superior, o parser escreverá '-1'.
- ☐ d. O parser imprime a largura do menor intervalo da sequência, caso haja pelo menos um intervalo válido.




## Pergunta 4

Quais das seguintes expressões regulares dão match a uma password com as seguintes condições:


- 6 a 12 caracteres de tamanho;
- Pelo menos uma maiúscula;
- Pelo menos uma minúscula;
- Pelo menos um dígito.

Respostas Seleccionadas:  b.  $r'^{^}[a-z]\{6,12\}|[A-Z]\{6,12\}|[0-9]\{6,12\}\$'$

Respostas:

 a.  $r'^{^}(.|\backslash n)^+\$'$

b.  $r'^{^}[a-z]\{6,12\}|[A-Z]\{6,12\}|[0-9]\{6,12\}\$'$

 c.  $r'^{^}[a-zA-Z0-9]\{,13\}\$'$

d.  $r'^{^}(?i:[a-zA-Z])|[0-9]\$'$



Considere o seguinte extrato de um filtro de texto em Python, em que **MMM** é uma das funções disponíveis no módulo 're':

```
s = re.MMM(r'\\w+{([^}]+)}', linha)
if (s):
    print(s.group(1))
```

e selecione as alíneas abaixo que são afirmações verdadeiras:

Respostas Seleccionadas:

```
Se o fragmento acima for acrescentado com
s = re.findall(r'\d+', s)
if (s):
    print(s)
```

☒ a. e 'linha' for "\tttt{nhhhh} jdjfn \cmd{1234}" a saída terá mais de 1 linha.

```
Se o fragmento acima for acrescentado com
s = re.findall(r'\d+', linha)
if (s):
    print(s)
```

☒ b. e 'linha' for "\tttt{nhhhh} jdjfn \cmd{1234}" a saída terá mais de 1 linha;

Respostas:

```
Se o fragmento acima for acrescentado com
s = re.findall(r'\d+', s)
if (s):
    print(s)
```

a. e 'linha' for "\tttt{nhhhh} jdjfn \cmd{1234}" a saída terá mais de 1 linha.

```
Se o fragmento acima for acrescentado com
s = re.findall(r'\d+', linha)
if (s):
    print(s)
```

☒ b. e 'linha' for "\tttt{nhhhh} jdjfn \cmd{1234}" a saída terá mais de 1 linha;

c. Se 'linha' for "\tttt{({nhh} jdjfn \cmd{1234})}" e o texto de saída "({nhh1234}" a função **MMM** corresponde a 'search';

☒ d. Se 'linha' for " \tttt[nhhhh} jdjfn \cmd{1234} huvb" e o texto de saída "1234" a função **MMM** corresponde a 'search';



Considere o programa em Python seguinte e escolha as afirmações verdadeiras:

```
import sys
import re

for linha in sys.stdin:
    if re.search(r'==', linha):
        res = re.sub(r'==', '.EQ.', linha)
        print(res)
    elif re.search(r'~= ', linha):
        res = re.sub(r'~= ', '!=', linha)
        print(res)
    elif re.search(r'<=', linha):
        res = re.sub(r'<=', '<=', linha)
        print(res)
    else:
        print(linha)
```

☐ a. O programa transforma o texto `a == b+c;` em `a == b+c;`

☐ b. Se acrescentássemos à especificação uma nova 4ª regra:

```
elif re.search(r'[a-z][a-zA-Z0-9]*', linha):
    res = re.sub(r'[a-z][a-zA-Z0-9]*', '.ID.', linha)
    print(res)
else:
    print(linha)
```

o resultado do Filtro gerado pelo Flex a partir dessa nova especificação, aplicado ao texto: `a ~= b+c;` não se alterava, mas em geral o seu comportamento é diferente.

☐ c. Se a 2ª regra da especificação acima fosse alterada para:

```
elif re.search(r'~(==)', linha):
    res = re.sub(r'~(==)', '!=', linha)
    print(res)
```

o resultado do programa a partir dessa nova especificação, aplicado ao texto: `a ~= b+c;` não se alterava;

☐ d. O programa transforma o texto `a ~= b+c;` em `a ~.EQ. b+c;`

Num determinado mundo, os números de telefone são constituídos por 3 partes:

- código do país: 1 a 3 dígitos;
- código da região: 1 a 3 dígitos;
- nº de telefone: 4 a 10 dígitos.

Sabendo que o separador entre a 1ª e a 2ª parte e entre a 2ª e a 3ª parte pode ser o hífen ou o espaço mas terá de ser o mesmo entre as duas partes, ou seja, existindo um hífen entre a 1ª e a 2ª partes também terá de ser um hífen a estar entre a 2ª e 3ª partes, das expressões regulares que se apresentam a seguir assinale aquelas que fariam match apenas com os números de telefone **válidos**:

☐ a.  $(([0-9]\{1,3\}-)\{2\} | ([0-9]\{1,3\} \backslash )\{2\}) [0-9]\{4,10\}$

☐ b.  $(0|1|2|3|4|5|6|7|8|9)\{1,3\} (\backslash | -) (0|1|2|3|4|5|6|7|8|9)\{1,3\} (\backslash | -) (0|1|2|3|4|5|6|7|8|9)\{4,10\}$

☐ c.  $[0-9]\{1,3\} [ \backslash -] [0-9]\{1,3\} [ \backslash -] [0-9]\{4,10\}$

☐ d.  $[0-9] [0-9]? [0-9]? [ \backslash -] [0-9]? [0-9]? [0-9] [ \backslash -] [0-9] [0-9] [0-9] [0-9]\{1,7\}$

## Pergunta 8

Considere o programa em Python seguinte e escolha as afirmações verdadeiras:

```
import sys
import re

for linha in sys.stdin:
    if re.search(r'==', linha):
        res = re.sub(r'==', '.EQ.', linha)
        print(res)
    elif re.search(r'~= ', linha):
        res = re.sub(r'~= ', '!=', linha)
        print(res)
    elif re.search(r'<=', linha):
        res = re.sub(r'<=', '<=', linha)
        print(res)
    else:
        print(linha)
```

- ☐ a. Se a 2ª regra da especificação acima fosse alterada para:

```
elif re.search(r'~(?==) ', linha):
    res = re.sub(r'~(?==) ', '!=', linha)
    print(res)
```

o resultado do programa a partir dessa nova especificação, aplicado ao texto: `a ~= b+c`; não se alterava;

- ☐ b. O programa transforma o texto `a == b+c`; em `a == b+c`;

- ☐ c. Se acrescentássemos à especificação uma nova 4ª regra:

```
elif re.search(r'[a-z][a-zA-Z0-9]*', linha):
    res = re.sub(r'[a-z][a-zA-Z0-9]*', '.ID.', linha)
    print(res)
else:
    print(linha)
```

o resultado do Filtro gerado pelo Flex a partir dessa nova especificação, aplicado ao texto: `a ~= b+c`; não se alterava, mas em geral o seu comportamento é diferente.

- ☐ d. O programa transforma o texto `a ~= b+c`; em `a ~.EQ. b+c`;



Considere o seguinte programa em Python:

```
import sys
import re

for linha in sys.stdin:
    segments = re.findall(r'(00+)|(11+)|(1)|(0)', linha)
    for (zs, us, u, z) in segments:
        if zs:
            print( str(len(zs)) + '0', end='')
        elif us:
            print( str(len(us)) + '1', end='')
        elif u:
            print('1', end='')
        elif z:
            print('0', end='')
        else:
            pass

    print('\n', end='')
print('\n', end='')
```

e indique quais das alíneas seguintes são verdadeiras:

- ☐ a. Se o texto de input tivesse uma única linha com o conteúdo "0101010101" o resultado seria "0101010101";
- ☐ b. Se a expressão regular fosse alterada para `r'(00*)|(11*)|(1)|(0)'`, o resultado final para uma linha com conteúdo "0001110011" não se alterava;
- ☐ c. Se o texto de input tivesse uma única linha com o conteúdo "101110101111001" o resultado seria "103101041201";
- ☐ d. Se a expressão regular fosse alterada para `r'(00*)|(11*)|(1)|(0)'`, o resultado final para uma linha com conteúdo "010" não se alterava;