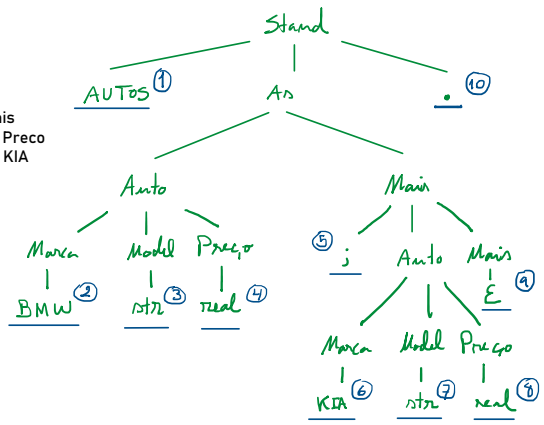
Resolução - Teste PL 2023

7 de junho de 2024 11:03

Questão 4
a)

1. Stant -> AUTOS As '.'
2. As -> Auto Mais
3. Mais -> & | ';' Auto Mais
4. Auto -> Marca Model Preco
5. Marca -> BMW | VW | KIA
6. Model -> str
7. Preco -> real



AUTOS BMW "i5 M60 xDrive Touring" 118.500,00 ; KIA "Ceed" 25.659,20 .

4 b)

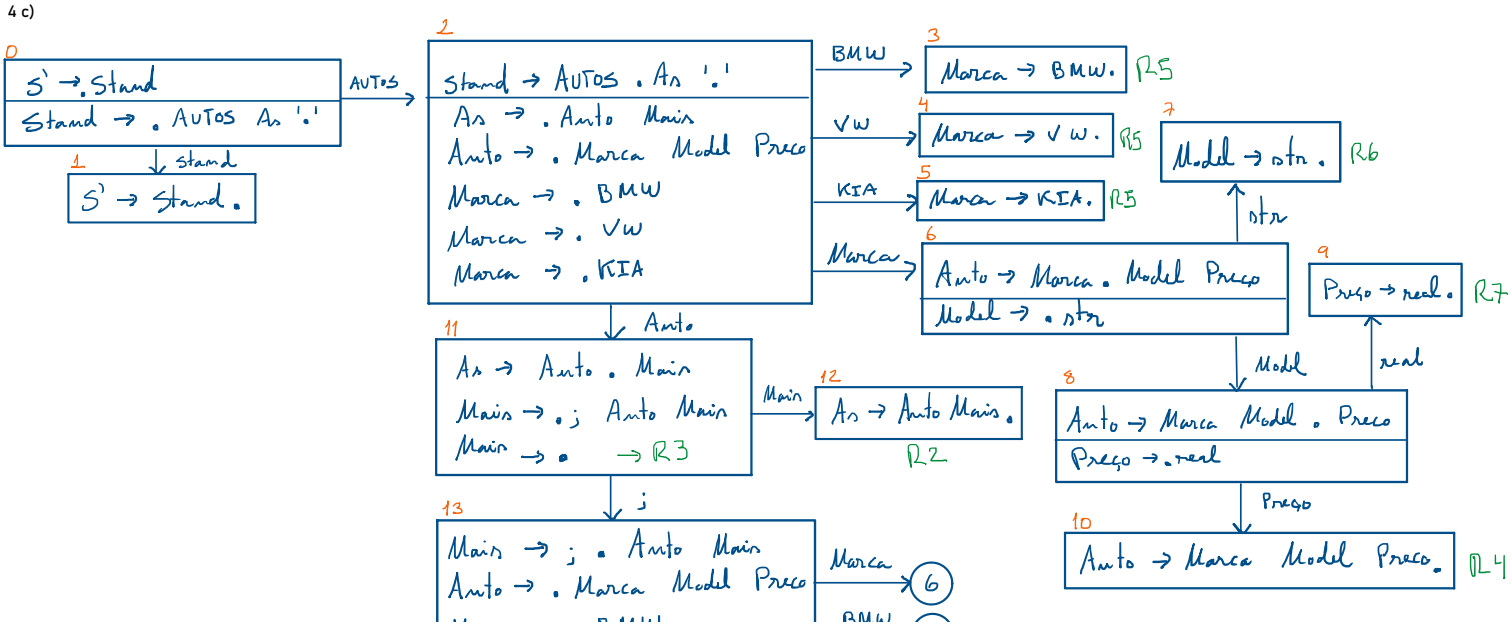
-	FIRST	FOLLOW
Stand -> AUTOS As '.'	AUTOS	$
As -> Auto Mais	BMW VW KIA	.
Mais -> & | ';' Auto Mais	; &	.
Auto -> Marca Model Preco	BMW VW KIA	. ;
Marca -> BMW | VW | KIA	BMW VW KIA	str
Model -> str	str	real
Preco -> real	real	. ;

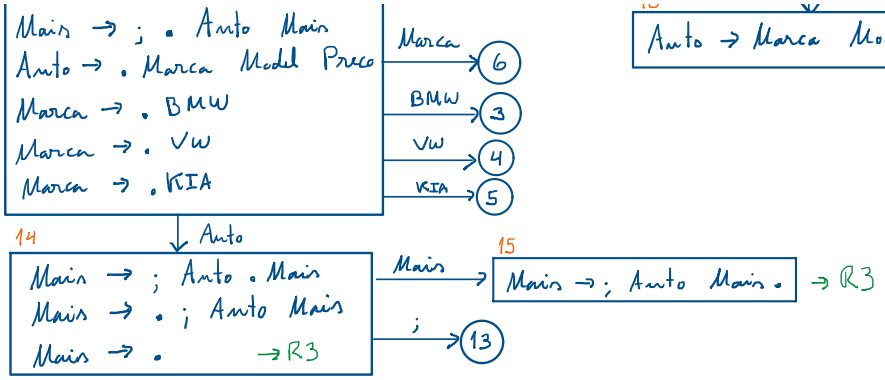
Follow(Mais) = Follow(Main) ∪ Follow(As)
Follow(Auto) = First(Mais)
= {;, ε}
= {;} ∪ Follow(Mais)
= {;, .}
Follow(Marca) = First(Model)
Follow(Preço) = Follow(Auto)

-	AUTOS	.	;	BMW	VW	KIA	str	real	$
Stand	Stand -> AUTOS As '.'								
As				As -> Auto Mais	As -> Auto Mais	As -> Auto Mais			
Mais		Mais -> &	Mais -> ';' Auto Mais						
Auto				Auto -> Marca Model Preco	Auto -> Marca Model Preco	Auto -> Marca Model Preco			
Marca				Marca -> BMW	Marca -> VW	Marca -> KIA			
Model							Model -> str		
Preco								Preco -> real	

É uma gramática LL(1) pois nenhuma célula da tabela tem mais do que um atributo, assim, não ocorrem situações em se tem de escolhe entre 2 ou mais produções diferentes.

4 c)





SLR(1) Table

-	AUTOS	.	;	BMW	VW	KIA	str	real	$	Stand	As	Mais	Auto	Marca	Model	Preco
0	S2									1						
1									ACC							
2				S3	S4	S5								6		
3							R5									
4							R5									
5							R5									
6							S7								8	
7								R6								
8								S9								10
9		R7	R7													
10		R4	R4													
11		R3	S13									12				
12		R2														
13				S3	S4	S5						6	14			
14		R3	S13									15				
15		R3														

Não ocorrem conflitos shift-reduce.

4 d)

2 funções de um parser RD (recursivo-descendente)
- uma para reconhecer qualquer símbolo terminal, (tokenizer?)
- e outra para reconheer o símbolo não terminal Mais

```
from enum import Enum
import re

class Tokens(Enum):
    BMW = 0
    VW = 1
    KIA = 2
    STR = 3
    REAL = 4
    SEMICOLON = 5
    PERIOD = 6
    AUTOS = 7
    EOF = -1

def tokenizer(s: str):
    tokens = []
    while len(s) > 0:
        if m := re.match(r"\bAUTOS\b", s):
            tokens.append((Tokens.AUTOS, "AUTOS"))
        elif m := re.match(r"\bBMW\b", s):
            tokens.append((Tokens.BMW, "BMW"))
        elif m := re.match(r"\bVW\b", s):
            tokens.append((Tokens.VW, "Vw"))
        elif m := re.match(r"\bKIA\b", s):
            tokens.append((Tokens.KIA, "KIA"))
        elif m := re.match(r";", s):
            tokens.append((Tokens.SEMICOLON, ";"))
        elif m := re.match(r"\.", s):
            tokens.append((Tokens.PERIOD, "."))
        elif m := re.match(r"(\-?(?:0|[1-9]\d*)(?:(\.|\,)\d+){0,2}(?:[eE]\d+)?)", s):
            tokens.append((Tokens.REAL, m.group()))
        elif m := re.match(r"\"([^\"]+)\"", s):
            tokens.append((Tokens.STR, m.group(1)))
        elif s[0] in " \n\t":
            s = s[1:]
        else:
            raise ValueError(f"Invalid character - {s[0]}")

        s = s[m.end():] if m else s

    return tokens + [(Tokens.EOF, None)]

tokens = None

def current_token():
    global tokens
    return tokens[0]

def next_token():
    global tokens
    tokens.pop(0)

def match_token(expected_token):
    cur = current_token()
    if cur[0] == expected_token:
        next_token()
        return cur[1]
    else:
        raise ValueError(f"Expected {expected_token}, but got {current_token()}")


def parse_stand():
    if current_token()[0] == Tokens.AUTOS:
        next_token()
        As = parse_as()
        if current_token()[0] == Tokens.PERIOD:
            next_token()
            return [As, '.']
        else:
            raise ValueError(f"Expected {Tokens.PERIOD}, but got {current_token()}")
    else:
        raise ValueError(f"Expected {Tokens.AUTOS}, but got {current_token()}")
```

```
def parse_as():
    auto = parse_auto()
    mais = parse_mais()
    return [auto, mais]

def parse_mais():
    match current_token():
        case (Tokens.SEMICOLON, _):
            next_token()
            auto = parse_auto()
            mais = parse_mais()
            return [';', auto, mais]
        case _:
            return "&"


def parse_auto():
    return [parse_marca(), match_token(Tokens.STR), match_token(Tokens.REAL)]

def parse_marca():
    match current_token():
        case (Tokens.BMW, _):
            next_token()
            return 'BMW'
        case (Tokens.VW, _):
            next_token()
            return 'VW'
        case (Tokens.KIA, _):
            next_token()
            return 'KIA'
        case _:
            raise ValueError(f"Expected a car brand, but got {current_token()}")

def parse(tk):
    global tokens
    tokens = tk
    return parse_stand()

def print_nested_array(array, level=0):
    indent = '    ' * level
    for item in array:
        if isinstance(item, list):
            print(f"{indent}[")
            print_nested_array(item, level + 1)
            print(f"{indent}]")
        else:
            print(f"{indent}{item}")

if __name__ == "__main__":
    text = 'AUTOS BMW "i5 M60 xDrive Touring" 118.500,00 ; KIA "Ceed" 25.659,20 ; KIA "Ceed" 25.659,20 .'
    tk = tokenizer(text)

    for a in tk:
        print(a)
    print()

    try:
        result = parse(tk)
        print_nested_array(result)
    except ValueError as e:
        print(e)
```

4 e)

```
def p_stand(p):
    """stand : AUTOS As '.'"""
    p[0] = p[1]
    return p

def p_As(p):
    """As : Auto Mais"""
    a, b = p[1]
    c, d = p[2]
    p[0] = (a + b, c + d)
    return p

def p_Mais(p):
    """Mais : ';' Auto Mais
            |
    """
    if len(p) == 1:
        p[0] = (0, 0)
    else:
        a, b = p[2]
        c, d = p[3]
        p[0] = (a + b, c + d)
    return p

def p_Auto(p):
    """Auto : Marca Model Preco"""
    p[0] = (p[2], p[3])
    return p

def p_Marca_BMW(p):
    """Marca : BMW"""
    p[0] = p[1]
    return p

def p_Marca_VW(p):
    """Marca : VW"""
    p[0] = p[1]
    return p

def p_Marca_KIA(p):
    """Marca : KIA"""
    p[0] = p[1]
    return p

def p_Model(p):
    """Model: str"""
    p[0] = 1
    return p

def p_Preco(p):
    """Preco : real"""
    p[0] = float(p[1])
    return p
```

Questão 3
a)

Recursividade à <u>direita</u>:

Valor -> Const | VAR | Lista | Atomo
Const -> NUM | ANUM
Lista -> '[' Valor Lista' ']'
Lista' -> ',' Valor Lista' | &

Atomo -> FUNC Args

Args -> '(' Valor Args' ')' | &
Args' -> ',' Valor Args' | &

Regra -> Atomo IMP Atomos '.'
Atomos -> Atomo Atomos' | &
Atomos' -> ',' Atomo Atomos' | &

Programa -> Regra Regras
Regras -> ',' Regra Regras | &

Recursividade à <u>esquerda</u>:

Valor -> NUM | VAR | Lista | Atomo
Lista -> '[' ListaAux Valor ']'
ListaAux -> ListaAux Valor COMMA | &

Atomo -> FUNC Args | ANUM
Args -> '(' ArgsAux Valor ')' | &
ArgsAux -> ArgsAux Valor COMMA | &

Regra -> Atomo IMP Atomos '.'

Atomos -> AtomosAux Atomo | &
AtomosAux -> AtomosAux Atomo COMMA | &

Programa -> Regras Regra
Regras -> Regras Regra COMMA

--- Lexer ---

```
import ply.lex as lex
""" Exemplo:
pred1 :- at2, at3.
pred(o1,o2) :- pred2(o1), pred3(o2), pred4(o1,o2).
pred2(X) :- pred5([X, Y, Z]).
"""
tokens = (
    'VAR',
    'NUM',
    'ANUM',
    'FUNC',
    'IMP',
    'LSBRACKET',
    'RSBRACKET',
    'COMMA',
    'LBRACKET',
    'RBRACKET',
    'PERIOD'
)
states = (
    ('rule', 'exclusive'),
    ('head', 'exclusive'),
    ('body', 'exclusive'),
    ('args', 'exclusive')
)
t_ignore = ' \t\n'
t_rule_head_body_args_ignore = ' \t\n'

"""Regra"""
def t_INITIAL_FUNC(t):
    r'\w+(?:(?=\s*\:\-)|(?=\()'
    t.lexer.push_state('rule')
    return t

def t_rule_LBRACKET(t):
    r'\('
    t.lexer.push_state('args')
    return t

def t_rule_IMP(t):
    r':\-'
    t.lexer.push_state('body')
    return t

def t_rule_body_PERIOD(t):
    r'\.'
    t.lexer.pop_state()
    t.lexer.pop_state()
    return t

"""Args"""
def t_args_NUM(t):
    r"(?:\+|\-)?\d+(?:\,\d+)?"
    return t


def t_args_ANUM(t):
    r"(\'[^\']+\'|[a-z][a-z0-9]*)"
    return t

def t_args_VAR(t):
    r"[A-Z]+"
    return t

def t_args_LSBRACKET(t):
    r"\["
    return t

def t_args_RSBRACKET(t):
    r"\]"
    return t

def t_args_COMMA(t):
    r"\,"
    return t

def t_args_RBRACKET(t):
    r'\)'
    t.lexer.pop_state()
    return t

"""Body"""
def t_body_FUNC(t):
    r'\w+(?=\()'
    return t

def t_body_ANUM(t):
    r'(\"[^\"]+\"|[a-z][a-z0-9]*)'
    return t

def t_body_VAR(t):
    r"[A-Z]+"
    return t

def t_body_LBRACKET(t):
```

```
    r'\('
    t.lexer.push_state('args')
    return t

def t_body_COMMA(t):
    r'\,'
    return t

"""Any"""
def t_ANY_error(t):
    print(f"Illegal character: {t.value[0]}")
    raise lex.LexError("Illegal character")

def t_ANY_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

lexer = lex.lex()

def run_tests():
    tests = [
        {
            "name": "asd",
            "input": """
                pred1 :- at2, at3.
                pred(o1,o2) :- pred2(o1), pred3(o2), pred4(o1,o2).
                pred2(X) :- pred5([X, Y, Z]).
            """
        }
    ]

    for test in tests:
        print(f"Test: {test['name']}\n")
        lexer.input(test['input'])
        for tok in lexer:
            print(tok)
        print('\n----------------\n')

if __name__ == '__main__':
    run_tests()
```

--- YACC ---

```
import ply.yacc as yacc
from lexer_teste2023 import tokens
from typing import Tuple

def p_Programa(p):
    """Programa : Regras Regra"""
    p[0] = p[1] + [p[2]]

def p_Regra(p):
    """Regra : Atomo IMP Atomos PERIOD"""
    p[0] = (p[1], p[3])

def p_Regras(p):
    """Regras : Regras Regra
              |
    """
    if len(p) == 1:
        p[0] = []
    else:
        p[0] = p[1] + [p[2]]

def p_Atomos(p):
    """Atomos : AtomosAux Atomo
              |
    """
    if len(p) == 1:
        p[0] = []
    else:
        p[0] = p[1] + [p[2]]

def p_AtomosAux(p):
    """AtomosAux : AtomosAux Atomo COMMA
                 |
    """
    if len(p) == 1:
        p[0] = []
    else:
        p[0] = p[1] + [p[2]]

def p_Atomo(p):
    """Atomo : FUNC Args
             | ANUM
    """
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = (p[1], p[2])

def p_Args(p):
    """Args : LBRACKET ArgsAux Valor RBRACKET
            |
    """
    if len(p) == 1:
        p[0] = []
    else:
        p[0] = p[2] + [p[3]]


def p_ArgsAux(p):
    """ArgsAux : ArgsAux Valor COMMA
               |
    """
    if len(p) == 1:
        p[0] = []
    else:
        p[0] = p[1] + [p[2]]

def p_Valor(p):
    """Valor : NUM
             | VAR
             | Atomo
             | Lista
    """
    p[0] = p[1]

def p_Lista(p):
    """Lista : LSBRACKET ListaAux Valor RSBRACKET"""
    p[0] = p[2] + [p[3]]

def p_ListaAux(p):
    """ListaAux : ListaAux Valor COMMA
                |
    """
    if len(p) == 1:
        p[0] = []
    else:
        p[0] = p[1] + [p[2]]


def p_error(p):
    print("Syntax error in input!", p)
    parser.exito = False

def print_nested_array(array, level=0, tuple=False):
```

```
    indent = '  ' * level
    for item in array:
        if isinstance(item, list):
            print(f"{indent}[")
            print_nested_array(item, level + 1)
            print(f"{indent}]")
        elif tuple and isinstance(item, Tuple):
            print(f"{indent}(")
            print_nested_array(item, level + 1)
            print(f"{indent})")
        else:
            print(f"{indent}{item}")

parser = yacc.yacc()
parser.exito = True
test = """
    pred1 :- at2, at3.
    pred(o1,o2) :- pred2(o1), pred3(o2), pred4(o1,o2).
    pred2(X) :- pred5([X, Y, Z]).
"""
result = parser.parse(test)
print_nested_array(result, tuple=True)
"""
(
  ('pred1', [])
  [
    at2
    at3
  ]
)
(
  ('pred', ['o1', 'o2'])
  [
    ('pred2', ['o1'])
    ('pred3', ['o2'])
    ('pred4', ['o1', 'o2'])
  ]
)
(
  ('pred2', ['X'])
  [
    ('pred5', [['X', 'Y', 'Z']])
  ]
)
"""
```

Rule 0 S' -> Programa
Rule 1 Programa -> Regras Regra
Rule 2 Regra -> Atomo IMP Atomos PERIOD
Rule 3 Regras -> Regras Regra
Rule 4 Regras -> <empty>
Rule 5 Atomos -> AtomosAux Atomo
Rule 6 Atomos -> <empty>
Rule 7 AtomosAux -> AtomosAux Atomo COMMA
Rule 8 AtomosAux -> <empty>
Rule 9 Atomo -> FUNC Args
Rule 10 Atomo -> ANUM
Rule 11 Args -> LBRACKET ArgsAux Valor RBRACKET
Rule 12 Args -> <empty>
Rule 13 ArgsAux -> ArgsAux Valor COMMA
Rule 14 ArgsAux -> <empty>
Rule 15 Valor -> NUM
Rule 16 Valor -> VAR
Rule 17 Valor -> Atomo
Rule 18 Valor -> Lista
Rule 19 Lista -> LSBRACKET ListaAux Valor RSBRACKET
Rule 20 ListaAux -> ListaAux Valor COMMA
Rule 21 ListaAux -> <empty>

Questão 2: módulo re (4v = 1 + 1 + 1 + 1)

```
import re
html = '<a href="http://www.pl-uminho.pt"><h1><b>Teste de PL</b></h1></a>'
html2 = '<a href="http://www.pl-uminho.pt">Teste de PL</a>'
match = re.search(r'<a\s+href="([^"]*)">(.*?)</a>', html)
match2 = re.search(r'<a\s+href="([^"]*)">(.*?)</a>', html2)
if match:
    url = match.group(1)
    content = match.group(2)
    print(url, content)
if match2:
    url = match2.group(1)
    content = match2.group(2)
    print(url, content)
```

a) Falso
O segundo grupo de captura fica com o conteúdo dentro de <a>
b) Falso
O segundo do grupo não fica com o resto do conteúdo de <a ...>,
mas sim com o conteudo entre <a> e
c) Falso
Deveria imprimir duas linhas
Ambas são validas para o regex utilizado
d) Verdadeiro

Questão 1: Expressões Regulares (4.5v = 1.5 + 1.5 + 1.5)

a) Considere as expressões regulares seguintes (ER):

```
e1 = a (a b)+ (c d | c f)* j
e2 = (a a b)+ c (d* | f*) j
```

e mostre que e1 e e2 não são equivalente, apresentando 2 frases que derivam de ambas e depois uma frase derivada de e1 e uma frase derivada de e2 que não é válida na outra ER.

b) Especifica uma expressão regular que faça match com todas as strings binárias, compostas apenas por zeros e uns, que contenham pelo menos dois uns consecutivos;

c) Do que é que as pessoas gostam?

Escrever uma função python que dado um texto devolva a lista das palavras que se seguem a `gostar de`, (mais precisamente "gost... de/do/da/dos/das X").

Exemplo:

```
In: O Manel gosta de passear e sempre gostou da praia.
Out: [passear, praia]
```

a)
Válida para e1 e para e2 : a a b c d j
Válida para e1 e para e2: a a b c f j
Válida apenas para e1: a a b a b c d c d j
Válida apenas para e2: a a b a a b c d d f d f d j

b)

[01]*11[01]*

c)

```
import re
from typing import Tuple

def get_preferences(text: str) -> list[str]:
    pattern = r"gost(?:ariamos|aremos|arias|aste|aram|aria|arem|amos|arei|arás|emos|am|em|ais|as|es|ou|a|o|e)\b\s+d(?:as|os|o|a|e)\s+(\w+)"
    preferences = re.findall(pattern, text)
    return [g[0] if isinstance(g, Tuple) else g for g in preferences]

if __name__ == "__main__":
    text = "O Manel gosta de passear e sempre gostou da praia."
    for el in get_preferences(text):
        print(el)
```