

# Relatório do trabalho da disciplina de Processamento de Linguagens

## Trabalho Prático 02

Renato Silva - 23511

Mário Pinto - 23506

Ruben Costa - 23508

Engenharia Software Informática

Junho de 2023

r minha honra que não recebi qualquer apoio não autorizado na realização deste trabalho  
prático. Afirmando  
e que não copieei qualquer material de livro, artigo, documento web ou de qualquer outra

Renato Silva - 23511

Mário Pinto - 23506

Ruben Costa - 23508

## Índice:

|                                 |    |
|---------------------------------|----|
| Introdução.....                 | 4  |
| Problema a ser resolvido .....  | 5  |
| Objetivos Atingidos .....       | 5  |
| Objetivos não atingidos .....   | 6  |
| Problemas encontrados .....     | 6  |
| Arith_lexer.py.....             | 7  |
| Arith_grammar.py.....           | 11 |
| Arith_eval.py .....             | 16 |
| Arith_main.py .....             | 19 |
| Arith_lexer_test.py .....       | 21 |
| Arith_grammar_test.py .....     | 22 |
| Parsetab.py .....               | 24 |
| Parser.out.....                 | 25 |
| Fotos da Execução.....          | 26 |
| Teste a ler o ficheiro: .....   | 27 |
| Teste Escrever na consola:..... | 27 |
| Mais testes: .....              | 28 |
| Teste Do Ciclo: .....           | 28 |
| Conclusão .....                 | 30 |

## Introdução

Com este projeto pretendemos na UC de Processamento de Linguagens adquirir experiência na definição de analisadores léxicos e sintáticos, bem como na definição de ações semânticas que traduzem as linguagens implementadas.

O processo para a realização deste trabalho prático deverá passar pelas seguintes fases:

1. Especificar a gramática concreta da linguagem de entrada;
2. Construção de um reconhecedor léxico (a partir da biblioteca `lex`) para reconhecer os símbolos terminais identificados na gramática e testar esse reconhecedor com alguns exemplos de palavras da linguagem de entrada;
3. Construção de um reconhecedor sintático (a partir da biblioteca `yacc`) para reconhecer a gramática concreta e testar esse reconhecedor com alguns exemplos frases da linguagem de entrada;
4. Planeamento de uma árvore de sintaxe abstrata para representar a linguagem de entrada e associar ações semânticas de tradução, as produções da gramática de forma a construir a correspondente árvore de sintaxe abstrata;
5. Desenvolvimento do gerador de código para produzir a resposta solicitada, através da avaliação da árvore de sintaxe abstrata.

Neste sentido, foi proposto a implementação de uma calculadora, especificando expressões aritméticas.

A realização deste projeto permitirá alargar o nosso conhecimento para situações futuras, quer em termos profissionais quer em termos pessoais.

## Problema a ser resolvido

Neste trabalho prático pretendemos implementar uma ferramenta em Python, usando a biblioteca PLY, que interprete uma linguagem capaz de especificar algumas instruções que habitualmente encontramos em qualquer linguagem de programação.

A ferramenta a desenvolver começa por ler um ficheiro de texto (com extensão .ea, para expressões aritméticas) contendo uma sequência de comandos de especificação das expressões aritméticas, aplicando esses comandos de forma a calcular o resultado pretendido.

O resultado de processar o ficheiro de texto entrada.ea, apresentando ao utilizador no terminal.

## Objetivos Atingidos

Conseguimos com sucesso atingir quase todos os objetivos.

Objetivos atingidos:

1. Especificar a gramática concreta da linguagem de entrada;
2. Construção de um reconhecedor léxico (a partir da biblioteca lex) para reconhecer os símbolos terminais identificados na gramática e testar esse reconhecedor com alguns exemplos de palavras da linguagem de entrada;
3. Construção de um reconhecedor sintático (a partir da biblioteca yacc) para reconhecer a gramática concreta e testar esse reconhecedor com alguns exemplos frases da linguagem de entrada;
4. Planeamento de uma árvore de sintaxe abstrata para representar a linguagem de entrada e associar ações semânticas de tradução as produções da gramática de forma a construir a correspondente árvore de sintaxe abstrata;
5. Desenvolvimento do gerador de código para produzir a resposta solicitada, através da avaliação da árvore de sintaxe abstrata.

## Objetivos não atingidos

Não conseguimos criar o ficheiro em c com o programa equivalente à sequência de instruções.

Objetivo não cumprido:

- Criar um ficheiro em C, com um programa equivalente à sequência de instruções.

## Problemas encontrados

Durante a implementação do módulo de avaliação de expressões aritméticas, encontramos desafios relacionados às generalizações. Foi uma experiência desafiadora lidar com situações desconhecidas e exigiu criatividade para superar essas dificuldades.

Tivemos vários problemas ao tentar implementar para código c, então decidimos não executar essa parte no nosso trabalho.

## Arith\_lexer.py

O código em questão é a implementação do analisador léxico para a gramática da linguagem aritmética. O analisador léxico é responsável por identificar e retornar os tokens que compõem um programa escrito nessa linguagem.

A classe ArithLexer define os tokens que podem ser reconhecidos pelo analisador, bem como as suas regras de reconhecimento. Alguns dos tokens definidos são:

- `tokens`: uma tupla que lista os nomes dos tokens reconhecidos pelo lexer.
- `literals`: uma lista de caracteres literais, ou seja, caracteres que são reconhecidos como tokens individuais.
- `t_ignore`: uma string contendo os caracteres a serem ignorados pelo lexer, como espaços em branco.
- Os métodos `__init__`, `t_error`, `build`, `input` e `token` são utilizados para inicializar, configurar e utilizar o lexer.
- Os métodos `t_NUM`, `t_ESCREVER`, `t_VAR`, `t_FIM`, `t_PARA`, `t_EM`, `t_FAZER`, `t_ID`, `t_STRING` e `t_COMENTARIO` são responsáveis por reconhecer os padrões correspondentes aos tokens e criar os objetos de token correspondentes.
- O método `t_error` é chamado quando ocorre um erro léxico, ou seja, quando um caractere não corresponde a nenhum padrão de token conhecido.

Além disso, são definidos os caracteres literais que são reconhecidos pelo analisador, como operadores matemáticos, parênteses, vírgula e ponto e vírgula.

A classe também possui métodos para construir o lexer, definir a entrada do lexer e obter o próximo token.

Por fim, o método `t_error` é chamado em caso de erro léxico, exibindo uma mensagem de erro.

```
arith_lexer.py 1 X
arith_lexer.py > ...
renators77, 33 minutes ago | 1 author (renators77)
1 import ply.lex as plex renators77, last week • início do lexer ...
2 +
renators77, 33 minutes ago | 1 author (renators77)
3 class ArithLexer:
4     tokens= (
5         "NUM", # numero inteiro
6         "STRING", # string delimitada por aspas duplas
7         "FIM", # atribuicao para o "FIM" do programa.
8         "ID", # identificador de variável ou função
9         "ESCREVER", # identificador
10        "VAR", # atribuicao de uma variavel
11        "PARA", # atribuicao "PARA" o ciclo
12        "EM", # atribuicao para "EM" ciclo
13        "FAZER", # atribuicao para "FAZER" ciclo
14        "COMENTARIO", #identificador de um comentario
15    )
16
17    literals = [
18        '+', # sinal de soma
19        '-', # sinal de subtração
20        '*', # sinal de multiplicação
21        '/', # sinal de divisão
22        '=', # sinal de igual
23        '(', # parêntese esquerdo
24        ')', # parêntese direito
25        '[', # parêntese reto esquerdo
26        ']', # parêntese reto direito
27        '>', # sinal de maior que
28        '<', # sinal de menor que
29        ',', # vírgula
30        ';', # ponto-e-vírgula
31        '.', # . para o ciclo
32    ]
33
34    #Ignorar espacos
35    t_ignore = " "
36
37    #Inicializa o lexer como None
38    def __init__(self):
39        self.lexer = None
40
41    #Reconhecer numeros inteiros e decimais
42    def t_NUM(self, t):
43        r'[0-9]+(\.[0-9]+)?'
44
45        t.value = int (t.value)
46        return t
47
48    #Reconhecer a Palavra ESC ou ESCREVER
49    def t_ESCREVER(self, t):
50        r'ESC(REVER)?'
51        return t
52
```

Figura 1 - Lexer (parte 1)



```

53     #Reconhecer a Palavra VAR
54     def t_VAR(self, t):
55         r'VAR'
56         return t
57
58     #Reconhecedor de FIM
59     def t_FIM(self, t):
60         r"FIM"
61         return t
62
63
64     #Reconhecedor de PARA
65     def t_PARA(self, t):
66         r"PARA"
67         return t
68
69
70     #Reconhecedor de EM
71     def t_EM(self, t):
72         r"EM"
73         return t
74
75     | renators77, 35 minutes ago • Done: Adicao do ciclo ...
76
77     #Reconhecedor de FAZER
78     def t_FAZER(self, t):
79         r"FAZER"
80         return t
81
82
83     #Reconhecer um identificador
84     def t_ID(self, t):
85         r'[a-zA-Z][a-zA-Z0-9]*'
86         return t
87
88     #Reconhecedor de uma string
89     def t_STRING(self, t):
90         r'".*?"'
91
92         t.value = t.value[1:-1] # Remove quotes
93         return t
94
95
96     #Reconhecedor de COMENTARIO
97     def t_COMENTARIO(self, t):
98         r'//[^\;]*'
99         return t

```

Figura 2 - Lexer (parte 2)

```

#cria o lexer
def build(self, **kwargs):
    self.lexer = plex.lex(module=self, **kwargs)

#define a entrada do lexer
def input(self, string):
    self.lexer.input(string)

#Obter proximo token do lexer
def token(self):
    token = self.lexer.token() #percorrer todos os tokens
    return token if token is None else token.type

#Ocorre quando ocorre erro lexico
def t_error(self, t):
    print(f"Unexpected token: [{t.value[:10]}]")
    exit(1)

```

Figura 3 - Lexer (parte 3)

## Arith\_grammar.py

O arquivo `arith_grammar.py` implementa a classe `ArithGrammar`, que é responsável por definir e construir a gramática utilizada para análise sintática de expressões matemáticas e comandos específicos.

A classe `ArithGrammar` possui os seguintes componentes e funcionalidades:

- `declared`: Um conjunto (set) que armazena as variáveis declaradas durante a análise sintática.
- `precedence`: Define a precedência e a associatividade dos operadores matemáticos utilizados nas expressões.
- `__init__()`: O construtor da classe, que inicializa as variáveis `yacc`, `lexer` e `tokens` utilizadas na análise sintática.
- `build()`: Constrói o analisador sintático, utilizando a classe `ArithLexer` do arquivo `arith_lexer.py` para a análise léxica. Ele configura o analisador léxico (`lexer`) e define os tokens utilizados pela gramática. Em seguida, cria o analisador sintático (`yacc`) utilizando a biblioteca `ply.yacc`.
- `parse()`: Realiza a análise sintática de uma string de entrada utilizando o analisador léxico e sintático configurados. Retorna o resultado da análise.
- `p_...()`: Métodos que definem as regras de produção da gramática. Cada método tem o prefixo `p_` seguido do nome da regra de produção correspondente. Esses métodos são chamados pelo analisador sintático durante a análise da string de entrada. Cada método recebe um parâmetro `p` que contém os tokens reconhecidos e outros valores relevantes para a regra de produção. Os métodos definem as ações a serem tomadas para cada regra de produção, construindo a árvore sintática abstrata (AST) conforme as regras da gramática.
- `p_error()`: Um método especial chamado quando ocorre um erro durante a análise sintática. Ele imprime uma mensagem de erro indicando o tipo de erro encontrado.

Em resumo, o arquivo `arith_grammar.py` define a classe `ArithGrammar`, que implementa a gramática e as regras de produção para a análise sintática de expressões matemáticas e comandos específicos. Ele utiliza a biblioteca `ply.yacc` para construir o analisador sintático e trabalha em conjunto com a classe `ArithLexer` para realizar a

análise léxica. A análise sintática resulta na construção de uma árvore sintática abstrata (AST) que representa a estrutura da expressão ou comando analisado.



```
1 # arith_grammar.py
2 from arith_lexer import ArithLexer
3 import ply.yacc as pyacc
4
5 class ArithGrammar:
6
7     # #Adicionar as Variaveis declaradas
8     declared = set()
9
10    precedence = (
11        ('left', '+', '-'), # level=1, assoc=left
12        ('left', '*', '/'), # level=2, assoc=left
13        ('right', 'simetrico'),
14    )
15
16    # Construtor da classe Grammar onde inicializa:
17    def __init__(self):
18        self.yacc = None
19        self.lexer = None
20        self.tokens = None
21        self.declared = set()
22
23    # Construir o analisador sintatico, pega no lexer e configura-o
24    def build(self, **kwargs):
25        self.lexer = ArithLexer()
26        self.lexer.build(**kwargs)
27        self.tokens = self.lexer.tokens
28        self.yacc = pyacc.yacc(module=self, **kwargs) # cria analisador sintatico
29
30
31    # Realiza a analise sintatica do Input fornecido
32    def parse(self, string):
33        self.lexer.input(string)
34        return self.yacc.parse(lexer=self.lexer.lexer)
35
36
37    # Regras Producao da Gramatica:
38
39
40    # Lista de instruções da gramatica seguida de ;
41    def p_s(self, p):
42        """ S : LstV ';' """
43
44        p[0] = p[1] # Atribuicao das instruções à producao
45
46
47    #Definicao do FIM do programa
48    def p_fim(self, p):
49        """ S : FIM ';' """
50        exit(0)
```

Figura 4 - Grammar (parte 1)

```

# Construir uma lista de instruções separadas por ;
def p_expr_tail(self, p):
    """ LstV : LstV ';' Instrucao """

    lstArgs = p[1]['args']
    lstArgs.append(p[3])
    p[0] = dict(op='seq', args= lstArgs ) # retorna o valor da Producao LstV

# Lista de Instrucao contendo uma unica instrucao
def p_expr_head(self, p):
    """ LstV : Instrucao """

    p[0] = dict(op='seq', args=[p[1]]) # Permite construir a AST atraves da sequencia de instrucoes

# *Representa a Instrucao ESCREVER, seguida de um identificador
def p_expr_inst_operacao(self, p):
    """ Instrucao : ESCREVER Resultado
    | | | | VAR ListaSimbolos """

    p[0] = {'op': 'esc', 'args': [p[2]]} # Cria Dicionario *

#Representa a operacao de que Instrucao pode ser um Identificador
def p_expr_inst_operacao_identificador(self, p):
    """ Instrucao : Identificador """

    p[0] = p[1] # Cria Dicionario *

#Representa a operacao comentario
def p_expr_inst_operacao_comentario(self, p):
    """ Instrucao : COMENTARIO """

    p[0] = dict(op='comentario', args=[p[1]]) # Cria Dicionario *

#Representa a operacao Ciclo
def p_expr_inst_operacao_ciclo(self, p):
    """ Instrucao : PARA ID EM '[' MATH '.' '.' MATH ']' FAZER Resultado FIM PARA """

    p[0] = dict(op='ciclo', args=[p[2], p[5], p[8], p[11]]) # Cria Dicionario *

#Representa a operacao Escrever ou VAR com o uso do comentario
def p_expr_inst_operacao_ESCVAR_comentario(self, p):
    """ Instrucao : ESCREVER Resultado COMENTARIO
    | | | | VAR ListaSimbolos COMENTARIO """

    p[0] = {'op': 'esc', 'comentario': p[3], 'args': [p[2]]} # Cria Dicionario *

```

Figura 5 - Grammar (parte 2)

```

#Define que a ListaSimbolos pode ser uma lista de IDs
def p_expr_atribuicao_simbolo_id(self, p):
    """ ListaSimbolos : ListaIDs """

    p[0] = p[1]    #Cria no dicionario

#Representa que podemos ter separacao ',' entre a listaSimbolos
def p_expr_multiplo_simbolo(self, p):
    """ ListaSimbolos : ListaIDs ',' ListaSimbolos """

    p[0] = dict(op='seq', args=[p[1], p[3]])    #Cria no dicionario

#Define que a Lista de IDs, pode ser composta por ID
def p_expr_id(self, p):
    """ ListaIDs : ID """

    ArithGrammar.declared.add(p[1])    #Adiciona o ID ao set dos ID declarados

    p[0] = p[1]    #Cria no dicionario

#Define a atribuicao de um ID a um valor do math
def p_expr_atribuicao_id(self, p):
    """ ListaIDs : ID '=' MATH """

    p[0] = dict(op='atr', args=[p[1], p[3]])    #Cria no dicionario

#Representa a Atribuicao de um ID a um MATH
def p_expr_atribuicao_identificador(self, p):
    """ Identificador : ID '=' MATH """

    # #Verifica se o ID está declarado se Sim, executa esta produção senão dá exit
    if p[1] not in ArithGrammar.declared:
        print(f"Syntax error: undeclared ID '{p[1]}'")

        exit(1)

    p[0] = dict(op='atr', args= [ p[1] , p[3]] ) #Cria no dicionario

#Define que Resultado pode ser um MATH do ID definido pela producao VAR
def p_expr_listaResultados(self, p):
    """ Resultado : ListaResultados """

    p[0] = p[1]    #Cria no dicionario

```

Figura 6 - Grammar (parte 3)

```

# Define as operacoes matematicas entre simbolo e o seu grau de precedencia
def p_expr_operacao_math(self, p):
    """ MATH : MATH '+' MATH
        | MATH '-' MATH
        | MATH '*' MATH
        | MATH '/' MATH """

    p[0] = dict(op=p[2], args=[p[1], p[3]]) #Cria no dicionario

#Define MATH Negativos
def p_expr_sinalmenos(self, p):
    """ MATH : '-' MATH %prec simetrico """

    p[0] = dict(op='-', args=[p[2]]) #Cria no dicionario

#Trata de MATH entre parenteses
def p_expr_parenteses(self, p):
    """ MATH : '(' MATH ')' """

    # p[0] = {'entreParenteses': p[2]} #Cria no dicionario
    p[0] = p[2]

#Define identificadores que sejam numeros inteiros ou decimais
def p_expr_numero(self, p):
    """ MATH : NUM """

    # p[0] = {'numero': p[1]} #Cria no dicionario
    p[0] = p[1]

#Define Identificadores que sejam ID
def p_expr_var(self, p):
    """ MATH : ID """

    p[0] = {'var': p[1]} #Cria no dicionario

#Define Identificadores que sejam strings
def p_expr_string(self, p):
    """ MATH : STRING """

    # p[0] = {'String': p[1]} #Cria no dicionario
    p[0] = p[1]

#mensagem erro
def p_error(self, p):
    if p:
        print(f"Syntax error: unexpected '{p.type}'")
    else:
        print("Syntax error: unexpected end of file")
    exit(1)

```

Figura 7 - Grammar (parte 4)

## Arith\_eval.py

O nosso arquivo `arith_eval.py` é um código Python que contém a definição de uma classe chamada `ArithEval`. Essa classe é responsável por avaliar expressões aritméticas representadas em forma de árvore sintática abstrata (AST).

A classe `ArithEval` possui os seguintes componentes:

- `symbols`: um dicionário que armazena as atribuições de variáveis feitas durante a execução.
- `operators`: um dicionário que mapeia os operadores suportados pela linguagem para funções que realizam as operações correspondentes.
- `_attrib`: um método estático que realiza a atribuição de valor a uma variável, armazenando-a no dicionário `symbols`.
- `_comentario`: um método estático que não realiza nenhuma ação, provavelmente utilizado para representar um comentário na linguagem.
- `_ciclo`: um método estático que realiza um loop e imprime uma expressão um determinado número de vezes.
- `evaluate`: um método estático que avalia uma árvore de sintaxe abstrata (AST) da linguagem e retorna o resultado da expressão ou executa o comando correspondente.
- `_eval_operator`: um método estático que avalia um operador da AST e executa a função correspondente no dicionário `operators` com os argumentos fornecidos.

O código também possui tratamento de exceções para casos em que um tipo de AST desconhecido é encontrado ou uma variável não declarada é referenciada.

Em resumo, o arquivo `arith_eval.py` implementa uma classe `ArithEval` que pode ser usada para avaliar expressões aritméticas representadas em forma de AST, permitindo a atribuição de valores a variáveis e a execução de operações aritméticas.



```
arith_eval.py X
arith_eval.py > ...
renators77, 28 minutes ago | 1 author (renators77)
# arith_eval renators77, 3 days ago • Melhorias na gramatica e no eval. ...
+
renators77, 28 minutes ago | 1 author (renators77)
3 class ArithEval:
4
5     # guardar as atribuições de variáveis
6     symbols = {}
7
8     operators = {
9         "+": lambda args: args[0] + args[1],
10        "-": lambda args: args[0] - args[1],
11        "*": lambda args: args[0] * args[1],
12        "/": lambda args: args[0] / args[1],
13        "seq": lambda args: args[-1],
14        "atr": lambda args: ArithEval._attrib(args),
15        "esc": lambda args: print(args[0]),
16        "comentario": lambda args: ArithEval._comentario(),
17        "ciclo": lambda args: ArithEval._ciclo(args),
18    }
19
20    @staticmethod
21    def _attrib(args): # A=10 {'op':'atr' 'args': [ "A", 10 ]} => _attrib( [ 'A', 10 ] )
22        varid = args[0] # 'A'
23        value = args[1] # 10
24        ArithEval.symbols[varid] = value # symbols { 'A':10 }
25        # print(f'{varid} = {value}') # Imprimir o valor do símbolo (id)
26        return None
27
28    @staticmethod
29    def _comentario():
30        return None
31
32    @staticmethod
33    def _ciclo(args):
34        for x in range(args[1], int(args[2]) + 1):
35            print(args[3])
36        return
37
```

Figura 8 - Eval (Parte 1)

```

@staticmethod
def evaluate(ast):
    if type(ast) is int: # constant value, eg in (int, str)
        return ast
    if type(ast) is dict: # { 'op': ... , 'args': ... }
                        # { 'var': ... }
        return ArithEval._eval_operator(ast)
    if type(ast) is str:
        return ast
    raise Exception(f"Unknown AST type")

@staticmethod
def _eval_operator(ast):
    if 'op' in ast:
        op = ast["op"]
        args = [ArithEval.evaluate(a) for a in ast['args']]
        if op in ArithEval.operators:
            func = ArithEval.operators[op]
            return func(args)
        else:
            raise Exception(f"Unknown operator {op}")

    if 'var' in ast:
        varid = ast["var"] # ast={ 'var': "A" } => ast["var"] varid="A"
        if varid in ArithEval.symbols: # "A" in symbols { 'A':10 }
            return ArithEval.symbols[varid] # 10
        raise Exception(f"error: '{varid}' undeclared (first use in this function)")
    raise Exception('Undefined AST')

```

Figura 9 - Eval (parte 2)

## Arith\_main.py

O script importa os seguintes módulos:

- módulo `arith_grammar`, que contém as regras gramaticais para expressões aritméticas.
- módulo `arith_eval`, que fornece a funcionalidade de avaliação para expressões aritméticas.
- módulo `sys`, que fornece acesso a parâmetros e funções específicas do sistema.
- As classes `pprint` e `PrettyPrinter` do módulo `pprint`, usadas para imprimir de forma organizada a árvore sintática abstrata (AST, na sigla em inglês).

É criada uma instância de `ArithGrammar` e atribuída à variável `ag`.

O método `build()` é chamado em `ag` para construir a gramática aritmética.

O script verifica se o número de argumentos da linha de comando é 2. Se for, assume-se que um nome de arquivo foi fornecido como argumento. O script abre o arquivo, lê o seu conteúdo e armazena na variável `contents`.

- Se ocorrer uma exceção durante a leitura do arquivo ou a análise do conteúdo, a exceção é capturada e impressa na saída de erro padrão (`stderr`).

Se o número de argumentos da linha de comando não for 2, o script entra em um loop de entrada. Ele solicita repetidamente ao usuário uma expressão aritmética usando a função `input()` até que uma entrada vazia seja fornecida.

- Se ocorrer uma exceção durante a análise da expressão ou a avaliação da AST, a exceção é capturada e impressa.

Dentro do bloco `try`, o método `ag.parse()` é chamado para analisar o conteúdo do arquivo ou a entrada do usuário. A AST resultante é atribuída à variável `ast`.

A AST é impressa de forma organizada usando o método `pp.pprint()`.

O método `ArithEval.evaluate()` é chamado com a AST como argumento para avaliar a expressão aritmética. O resultado é descartado se for `None`.

Se ocorrer uma exceção durante a análise ou avaliação, a exceção será capturada e impressa.

```
arith_main.py > ...
1  # arith.py
2  from arith_grammar import ArithGrammar
3  from arith_eval import ArithEval
4  import sys
5  from pprint import PrettyPrinter
6  pp = PrettyPrinter(sort_dicts=False)
7
8  ag = ArithGrammar()
9  ag.build()
10
11
12  if len(sys.argv) == 2:
13      with open(sys.argv[1], "r") as file:
14          contents = file.read()
15          try:
16              ast = ag.parse(contents)
17              pp.pprint(ast)
18              ArithEval.evaluate(ast)
19          except Exception as e:
20              print(e, file=sys.stderr)
21  else:
22      for expr in iter(lambda: input(">> "), ""):
23          try:
24              ast = ag.parse(expr)
25              pp.pprint(ast)
26              res = ArithEval.evaluate(ast)
27              if res is not None:
28                  print(f"<< {res}")
29          except Exception as e:
30              print(e)
31
32
```

Figura 10 - main

## Arith\_lexer\_test.py

O código em questão realiza um teste do analisador léxico implementado na classe `ArithLexer`. Ele demonstra como usar o analisador léxico para obter os tokens de um programa escrito na linguagem aritmética.

No exemplo fornecido, o teste consiste em analisar a string `'ESCREVER x "y123" _variavel'`. Após chamar o método `build()` para construir o lexer, é feita a chamada ao método `input()` passando a string a ser analisada.

Em seguida, é utilizado um loop para obter os tokens do lexer, chamando repetidamente o método `token()`. O loop continua até que não haja mais tokens, ou seja, até que o método `token()` retorne `None`. Em cada iteração do loop, o token é impresso.

Os tokens resultantes serão exibidos na saída. No exemplo fornecido, os tokens esperados são `'ESCREVER'`, `'ID'`, `'STRING'` e `'ID'`.

```
arith_lexer_test.py > ...
1  # arith_lexer_test.py
2  from arith_lexer import ArithLexer
3
4  teste = ArithLexer()
5  teste.build()
6
7  # teste.input('ESCREVER "ESCREVER"; , = +') #ESCREVER STRING ; , = +
8  # teste.input('2 + 5') #NUM + NUM
9  # teste.input('VAR 2023;') #VAR NUM ;
10 teste.input('ESCREVER x "y123" _variavel') #ESCREVER ID STRING ID
11
12 while True:
13     token = teste.token()
14     if not token:
15         break
16     print(token, end=" ")
```

Figura 11 - Test Lexer

## Arith\_grammar\_test.py

O arquivo `arith_grammar_test.py` é um código Python que testa o funcionamento da classe `ArithGrammar` do arquivo `arith_grammar.py`, que é importado. Esta classe é responsável por analisar e interpretar frases escritas em uma gramática específica.

O código `arith_grammar_test.py` realiza o seguinte:

- Importa a classe `ArithGrammar` do arquivo `arith_grammar.py` e a classe `PrettyPrinter` do módulo `pprint`.
- Cria uma instância da classe `ArithGrammar` chamada `teste`.
- Chama o método `build()` na instância `teste` para construir a gramática.
- Define uma lista chamada `exemplos` que contém algumas frases para testar a gramática.
- Verifica sobre cada frase na lista `exemplos`.
  - Imprime a frase atual.
  - Chama o método `parse()` da instância `teste` passando a frase como argumento.
  - Imprime o resultado da análise e interpretação da frase usando a classe `PrettyPrinter`.

Durante a execução do código, cada frase da lista `exemplos` é processada pela gramática implementada na classe `ArithGrammar`, e o resultado da análise é impresso na tela. As frases podem conter instruções como escrever mensagens ou atribuir valores a variáveis, de acordo com a gramática definida.

```
arith_grammar_test.py X
tp02-d04 > arith_grammar_test.py > ...
renators77, 58 minutes ago | 1 author (renators77)
1 # arith.py renators77, 3 days ago • Adicao producao na gramatica e re
2 from arith_grammar import ArithGrammar
3 from pprint import PrettyPrinter
4
5 pp = PrettyPrinter(sort_dicts=False)
6
7 teste = ArithGrammar()
8 teste.build()
9
10 + exemplos = [ # exemplos a avaliar de forma independente...
11                 # 'ESCREVER "ola mundo!";',
12                 # 'ESCREVER "PL ", 2, a2 = 1 , "o ano de", "ESI";',
13                 # 'ESCREVER "soma de ", 9, "com ", 3*4 , "=", 9+2*3;',
14                 # 'VAR ano=2023, mes="maio";'
15                 'VAR B, c;',
16             ]
17 for frase in exemplos:
18     print(f"-----")
19     print(f"--- frase '{frase}'")
20     resposta = teste.parse ( frase )
21     print("resultado: ")
22     pp.pprint(resposta)
```

Figura 12 - Test Grammar

## Parsetab.py

O arquivo `parsetab.py` é gerado automaticamente e contém informações sobre a tabela de análise utilizada pelo analisador sintático. Esta tabela é criada a partir da definição da gramática do arquivo `arith_grammar.py`.

A tabela `lr_action` contém informações sobre as ações a serem tomadas em cada estado do analisador sintático. Cada estado é representado por um número e as ações possíveis são deslocamento (shift) e redução (reduce). As chaves representam os números dos estados e os valores são dicionários com os símbolos de entrada e as ações correspondentes.

A tabela `lr_goto` contém informações sobre os desvios (goto) a serem realizados em cada estado do analisador sintático. As chaves representam os números dos estados e os valores são dicionários com os símbolos não terminais de destino e os números dos estados correspondentes.

A lista `lr_productions` contém as produções da gramática, representadas por tuplas. Cada tupla contém o símbolo não terminal à esquerda da produção e uma lista dos símbolos (terminais e não terminais) à direita da produção. A lista também inclui informações adicionais, como o nome da função de ação semântica associada a cada produção e o arquivo e linha em que a produção foi definida.

Essas estruturas de dados são usadas pelo analisador sintático gerado pelo analisador LALR para analisar a entrada de acordo com a gramática definida no arquivo `arith_grammar.py`.



```

1  # -*- coding: utf-8 -*-
2  # parstab.py
3  # This file is automatically generated. Do not edit.
4  # pylint: disable=C,R
5  _tabversion = '3.10'
6
7  _lr_method = 'LALR'
8
9  _lr_signature = 'left+left/rightintrinsicCOMENTARIO EM ESCRIVER FAZER FIM ID NUM PARA STRING VAR S : leftV ']' S : FIM ']' leftV : Instrucao leftV : Instrucao Instrucao : ESCRIVER Resultado Va
10
11 _lr_action_items = { FIM : [[0,16,15,18,19,20,33,40,41,42,43,44,45,55],[3,-17,-19,-26,-27,-28,-24,-18,-20,-21,-22,-23,-25,56]], ESCRIVER : [[0,11],[5,5]], VAR : [[0,11],[6,6]], COMENTARIO : [[0,11,13,14,15,18,19,20,21,22,23,33,40,41,42,43,44,45,46,47],[8,8,27,-17,-19,-26,-27,-
12
13 _lr_action = {}
14 for k, _y in _lr_action_items.items():
15     for _x, _y in zip(_x0, _y1):
16         if not _x in _lr_action: _lr_action[_x] = {}
17         _lr_action[_x][_y] = _y
18 del _lr_action_items
19
20 _lr_goto_items = {'S':[[0],[1,1]],'leftV':[[0],[2,1]],'Instrucao':[[0,11],[4,26]],'Identificador':[[0,11],[7,7]],'Resultado':[[5,28,54],[13,40,55]],'ListaResultados':[[5,28,54],[14,14,14]],'MATH':[[5,16,17,25,28,29,30,31,32,37,48,51,54],[15,33,34,39,15,41,42,43,44,47,49],
21
22 _lr_goto = {}
23 for k, _y in _lr_goto_items.items():
24     for _x, _y in zip(_x0, _y1):
25         if not _x in _lr_goto: _lr_goto[_x] = {}
26         _lr_goto[_x][_y] = _y
27 del _lr_goto_items
28
29 _lr_productions = [
30 ('S' -> 'S', 'S', 1, None, None, None),
31 ('S' -> leftV ']', 'S', 2, p_2, 'arith_grammar.py', 42),
32 ('S' -> FIM ']', 'S', 2, p_2, 'arith_grammar.py', 40),
33 ('leftV' -> leftV ']', 'Instrucao', 'leftV', 3, p_expr_tail, 'arith_grammar.py', 55),
34 ('leftV' -> Instrucao 'leftV', 3, p_expr_head, 'arith_grammar.py', 64),
35 ('Instrucao' -> ESCRIVER Resultado, 'Instrucao', 2, p_expr_inst_operacao, 'arith_grammar.py', 71),
36 ('Instrucao' -> VAR ListaSimbolos, 'Instrucao', 2, p_expr_inst_operacao, 'arith_grammar.py', 72),
37 ('Instrucao' -> Identificador, 'Instrucao', 1, p_expr_inst_operacao_identificador, 'arith_grammar.py', 79),
38 ('Instrucao' -> COMENTARIO, 'Instrucao', 1, p_expr_inst_operacao_comentario, 'arith_grammar.py', 86),
39 ('Instrucao' -> PARA ID EM [ MATH ] FAZER Resultado FIM PARA, 'Instrucao', 13, p_expr_inst_operacao_ciclo, 'arith_grammar.py', 93),
40 ('Instrucao' -> ESCRIVER Resultado COMENTARIO, 'Instrucao', 3, p_expr_inst_operacao_escova_comentario, 'arith_grammar.py', 100),
41 ('Instrucao' -> VAR ListaSimbolos COMENTARIO, 'Instrucao', 1, p_expr_inst_operacao_escova_comentario, 'arith_grammar.py', 101),
42 ('ListaSimbolos' -> ListaIDs, 'ListaSimbolos', 1, p_expr_atribuicao_simbolo_id, 'arith_grammar.py', 108),
43 ('ListaSimbolos' -> ListaIDs, 'ListaSimbolos', 1, p_expr_atribuicao_simbolo_id, 'arith_grammar.py', 115),
44 ('ListaIDs' -> ID, 'ListaIDs', 1, p_expr_id, 'arith_grammar.py', 121),
45 ('ListaIDs' -> ID = MATH, 'ListaIDs', 3, p_expr_atribuicao_id, 'arith_grammar.py', 131),
46 ('Identificador' -> ID = MATH, 'Identificador', 3, p_expr_atribuicao_identificador, 'arith_grammar.py', 138),
47 ('Resultado' -> ListaResultados, 'Resultado', 1, p_expr_listaresultados, 'arith_grammar.py', 150),
48 ('Resultado' -> ListaResultados, 'Resultado', 3, p_args_multiple_resultado, 'arith_grammar.py', 150),
49 ('ListaResultados' -> MATH, 'ListaResultados', 1, p_args_resultado, 'arith_grammar.py', 163),
50 ('MATH' -> MATH = MATH, 'MATH', 3, p_expr_operacao_math, 'arith_grammar.py', 170),
51 ('MATH' -> MATH = MATH, 'MATH', 3, p_expr_operacao_math, 'arith_grammar.py', 171),
52 ('MATH' -> MATH = MATH, 'MATH', 3, p_expr_operacao_math, 'arith_grammar.py', 172),
53 ('MATH' -> MATH = MATH, 'MATH', 3, p_expr_operacao_math, 'arith_grammar.py', 173),
54 ('MATH' -> MATH, 'MATH', 2, p_expr_virguletoes, 'arith_grammar.py', 180),
55 ('MATH' -> ( MATH ), 'MATH', 1, p_expr_parentheses, 'arith_grammar.py', 186),
56 ('MATH' -> MATH, 'MATH', 1, p_expr_var, 'arith_grammar.py', 193),
57 ('MATH' -> ID, 'MATH', 1, p_expr_var, 'arith_grammar.py', 200),
58 ('MATH' -> STRING, 'MATH', 1, p_expr_string, 'arith_grammar.py', 207),
59 ]

```

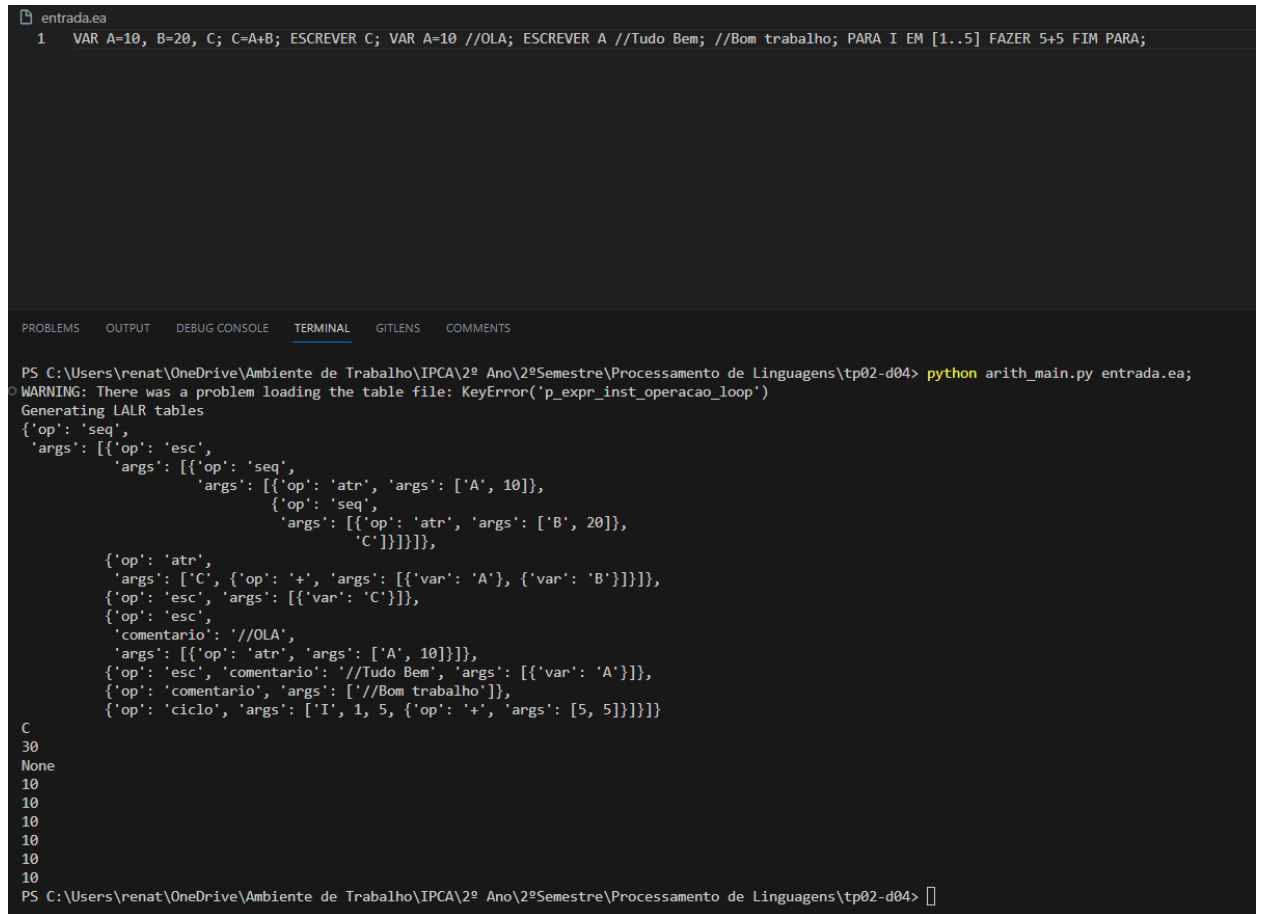
Figura 13 - Parstab

Parser.out

Este arquivo é criado automaticamente pelo PLY ao especificar a gramática e as configurações adequadas. Ele contém o código necessário para criar um analisador léxico e sintático que pode ser usado para processar a sintaxe de uma determinada linguagem.

Fotos da Execução

Teste a ler o ficheiro:



The image shows a VS Code editor window with a file named 'entrada.ea' open. The file contains a single line of code: `1 VAR A=10, B=20, C; C=A+B; ESCRIVER C; VAR A=10 //OLA; ESCRIVER A //Tudo Bem; //Bom trabalho; PARA I EM [1..5] FAZER 5+5 FIM PARA;`. Below the editor, the 'TERMINAL' tab is active, showing the command `python arith_main.py entrada.ea;` and its output. The output includes a warning about a problem loading the table file, followed by a detailed JSON representation of the generated LALR tables. The output ends with the values `C`, `30`, and `None`.

```
entrada.ea
1  VAR A=10, B=20, C; C=A+B; ESCRIVER C; VAR A=10 //OLA; ESCRIVER A //Tudo Bem; //Bom trabalho; PARA I EM [1..5] FAZER 5+5 FIM PARA;

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS COMMENTS

PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> python arith_main.py entrada.ea;
WARNING: There was a problem loading the table file: KeyError('p_expr_inst_operacao_loop')
Generating LALR tables
{'op': 'seq',
 'args': [{'op': 'esc',
            'args': [{'op': 'seq',
                        'args': [{'op': 'atr', 'args': ['A', 10]},
                                {'op': 'seq',
                                    'args': [{'op': 'atr', 'args': ['B', 20]},
                                            'C']}]}}}],
          {'op': 'atr',
            'args': ['C', {'op': '+', 'args': [{'var': 'A'}, {'var': 'B'}]}]},
          {'op': 'esc', 'args': [{'var': 'C'}]},
          {'op': 'esc',
            'comentario': '//OLA',
            'args': [{'op': 'atr', 'args': ['A', 10]}]},
          {'op': 'esc', 'comentario': '//Tudo Bem', 'args': [{'var': 'A'}]},
          {'op': 'comentario', 'args': ['//Bom trabalho']},
          {'op': 'ciclo', 'args': ['I', 1, 5, {'op': '+', 'args': [5, 5]}]}]
C
30
None
10
10
10
10
10
10
PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04>
```

Figura 14 - Teste Ler o Ficheiro

Teste Escrever na consola:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL OUTPUT COMMENTS
PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> & C:/Python311/Python.exe C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04\arith_main.py
Generating LALR tables
>> VAR A=10;
{'op': 'seq',
 'args': [{'op': 'esc', 'args': [{'op': 'atr', 'args': ['A', 10]}]}]}
None
>> VAR B=20, D=30, E;
{'op': 'seq',
 'args': [{'op': 'esc',
            'args': [{'op': 'seq',
                        'args': [{'op': 'atr', 'args': ['B', 20]},
                                {'op': 'seq',
                                    'args': [{'op': 'atr', 'args': ['D', 30]},
                                            'E']}]}]}]}]}
E
>> E=B+D;
{'op': 'seq',
 'args': [{'op': 'atr',
            'args': ['E', {'op': '+', 'args': [{'var': 'B'}, {'var': 'D'}]}]}]}
>> ESCREVER E;
{'op': 'seq', 'args': [{'op': 'esc', 'args': [{'var': 'E'}]}]}
50
>> ESCREVER E //TESTE123;
{'op': 'seq',
 'args': [{'op': 'esc', 'comentario': '//TESTE123', 'args': [{'var': 'E'}]}]}
50
o >> FIM;
PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> 

```

Figura 15 - Teste Escrever na Consola

Mais testes:

```

PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> & C:/Python311/Python.exe C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04\arith_main.py
>> A=10;
Syntax error: undeclared ID 'A'
PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> & C:/Python311/Python.exe C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04\arith_main.py
>> VAR A;
{'op': 'seq', 'args': [{'op': 'esc', 'args': ['A']}]}
A
>> A=10;
{'op': 'seq', 'args': [{'op': 'atr', 'args': ['A', 10]}]}
>> //FUNCIONA;
{'op': 'seq', 'args': [{'op': 'comentario', 'args': ['//FUNCIONA']}]}
>> ESCREVER "OLA TUDO BEM", 2+5;
{'op': 'seq',
 'args': [{'op': 'esc', 'args': ["OLA TUDO BEM{'op': '+', 'args': [2, 5]}"]}]}
OLA TUDO BEM{'op': '+', 'args': [2, 5]}
>> 

```

Figura 16 – Testes

Teste Do Ciclo:

```

● PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> & C:/Python/2º Ano/2ºSemestre/Processamento de Linguagens/tp02-d04/arith_main.py
>> PARA I EM [1..5] FAZER 2*5 FIM PARA;
{'op': 'seq',
 'args': [{'op': 'ciclo', 'args': ['I', 1, 5, {'op': '*', 'args': [2, 5]}]}]}
10
10
10
10
10
10
● >> FIM;
○ PS C:\Users\renat\OneDrive\Ambiente de Trabalho\IPCA\2º Ano\2ºSemestre\Processamento de Linguagens\tp02-d04> 

```

Figura 17 - Teste Do Ciclo

## Conclusão

Com este trabalho conseguimos pôr à prova as capacidades de cada um na disciplina de Processamento de Linguagens, aplicando bastante o conhecimento que fomos adquirindo ao longo das aulas.

Conclui-se que os objetivos deste projeto foram quase todos alcançados, podendo assim ser possível aprender a aplicar os conhecimentos lecionados durante as aulas, tendo desta forma conseguido perceber como é feito (passo a passo) na definição de analisadores léxicos e sintáticos e também na definição de ações semânticas que traduzem as linguagens implementadas ao longo do projeto.