ESCOLA SUPERIOR DE TECNOLOGIA IPCA

# PROCESSAMENTO DE

## RELATÓRIO DE TRABALHO PRÁTICO

Emanuel de Jesus Correia Carvalho - 8820
Joel Filipe Lacerda Martins - 17439
João Nuno Gonçalves Veloso - 17729
INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE
DOCENTE: ALBERTO SIMÕES
24/01/2021

# Índice

# Introdução

O presente projeto é desenvolvido no âmbito da unidade curricular de Processamento de Linguagens, onde o principal objetivo passa pela aplicação dos conceitos de processamento de linguagens utilizando a linguagem de programação *Python*. O enunciado escolhido é o número dois, "Logo", onde se pretende implementar um interpretador capaz de simular o maior número possível de comandos da linguagem original. Nesta linguagem o utilizador controla uma tartaruga que vai desenhando linhas por onde passa, sendo assim capaz de desenhar as mais variadas figuras geométricas e não só.

# Conceitos Gerais

A realização do trabalho prático implica conhecimentos a nível de expressões regulares e analise léxica e sintática.

## Expressões Regulares

São uma forma concisa e flexível de identificar cadeias de carateres que sejam do nosso interesse, como caracteres específicos, palavras ou padrões de caracteres. São escritas numa linguagem formal que pode ser interpretada por um processador de expressões regulares. O termo deriva do trabalho do matemático *Stephen Cole Kleene* e serviu como base para os primeiros algoritmos computacionais de busca.
Um exemplo em que é usada este tipo de ferramenta é a função de procura num navegador de Internet como o Google Chrome em que escrevemos o padrão de caracteres ou símbolos desejados e ele localiza na página todos os padrões existentes.

## Análise Léxica

É o processo de converter uma sequência de caracteres numa sequência de *tokens*.

Um programa que realiza análise lexical pode ser denominado *lexer*, *tokenizer* ou scanner, e é geralmente combinado com um analisador, e juntos analisam a sintaxe de uma linguagem de programação. Em *Python* existe a necessidade implícita de analisar o código-fonte colocado no interpretador para que este funcione corretamente. São usados *tokens* para input e o *parser* para output.

## Análise Sintática

Avalia a gramática da linguagem utilizada, e o que é a gramática? É a especificação formal da estrutura das frases permitidas na linguagem. E para isso a gramática pode ter regras como:

- Símbolos terminais: que não podem ser derivados;
- Símbolos não terminais: que podem ser derivados, sendo substituídos por outro(s) símbolo(s);
- Símbolo inicial: representa uma frase completa da linguagem;
- Regras de derivação: definem a estrutura gramática, a utilização dos símbolos.

No entanto, a gramática mais utilizada é livre de contexto, onde apenas um símbolo não terminal no lado esquerdo de suas regras de derivação.

Existem dois métodos de analisar a gramática:

- *Top-down:* reconhecedor descendente, efetua o reconhecimento de uma frase construindo uma árvore de derivação, a partir do axioma, e terminando nos símbolos terminais. Este processo também pode ser visto como uma derivação pela esquerda da frase a reconhecer. Só pode ser aplicado em a gramáticas que satisfaçam a condição LL (1).
- *Bottom-up*: Os reconhecedores bottom-up (ou ascendentes) aplicam-se a praticamente qualquer gramática independente do contexto. Infelizmente a construção deste tipo de reconhecedores é bastante trabalhosa, e que torna quase impraticável a sua implementação manual. Na prática, utilizam-se ferramentas geradoras de analisadores sintáticos, que geram automaticamente estes reconhecedores.

Exemplos:

| Gramática | |
|---|---|
| *frase* | -> *sujeito predicado* |
| *sujeito* | -> *artigo substantivo* |
| *predicado* | -> *verbo artigo substantivo* |
| *artigo* | -> o |
| *substantivo* | -> gato \| rato |
| *verbo* | -> perseguiu |

Figura 1 - Gramática a ser analisada

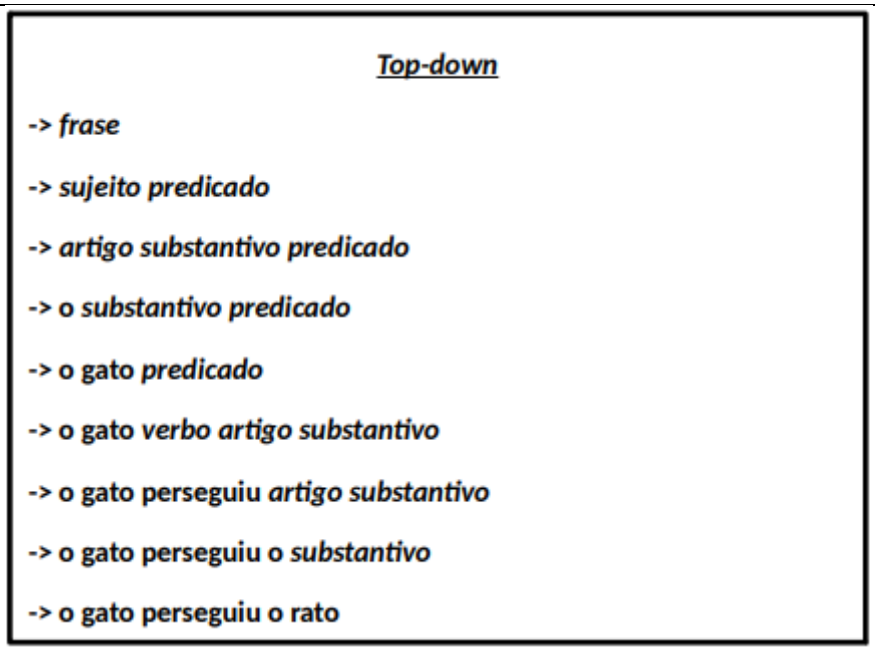


Figura 2 - Reconhecimento *Top-down*

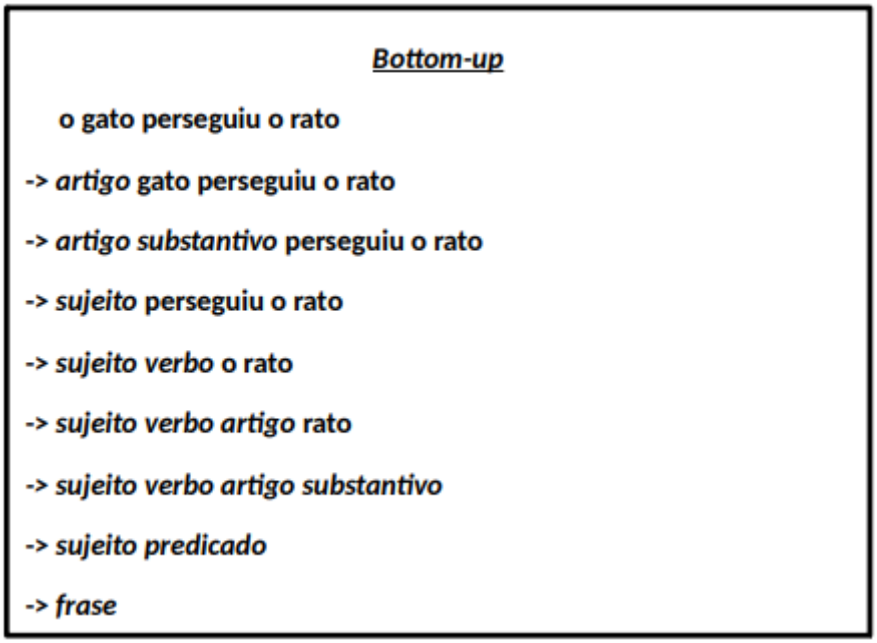


Figura 3 - Reconhecimento *Bottom-up*

## Conceitos aplicados

De seguida vamos demonstrar como aplicamos os vários conceitos lecionados em sala de aula e como os aplicamos no trabalho prático.

Neste trabalho prático utilizamos várias expressões regulares de forma a conseguirmos reconhecer todos os comandos pedidos pelo enunciado.

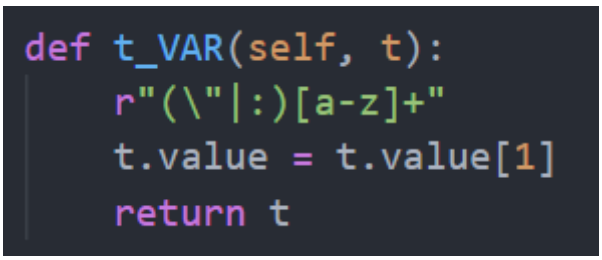


Figura 4 - Expressão Regular usada para reconhecimento de variáveis

Lex é um programa que gera analisadores léxicos. Ele é geralmente usado com o Yacc, um gerador de analisador sintático. Escrito originalmente por *Eric Schmidt* e *Mike Lesk*, ele é o gerador de analisador léxico padrão em diversos sistemas Unix. O lex lê um fluxo de entrada especificando um analisador que mapeia expressões regulares em blocos de código, e retorna um código fonte implementando o analisador.

Yacc é um gerador de analisador sintático desenvolvido por *Stephen C. Johnson* da AT&T para o sistema operacional *Unix*. Ele gera um analisador sintático, parte do compilador responsável por fornecer sentido sintático a um determinado código fonte, baseado numa gramática formal escrita numa forma similar ao formalismo de Backus-Naur.

O Yacc e o gerador de analisador léxico Lex são geralmente usados em conjunto. O Yacc usa uma gramática formal para analisar sintaticamente uma entrada, algo que o Lex não consegue fazer somente com expressões regulares (o Lex é limitado a simples máquinas de estado finito). Entretanto, o Yacc não consegue ler a partir duma simples entrada de dados, ele requer uma série de tokens, que são geralmente fornecidos pelo Lex. O Lex age como um pré-processador do Yacc. Segue abaixo duas figuras do relacionamento entre Lex e Yacc:

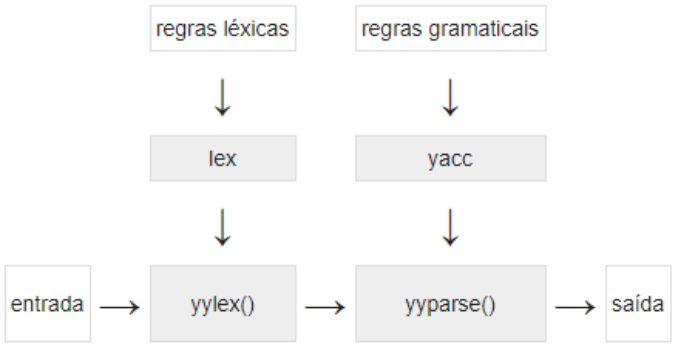


Figura 5 - Relação entre Lex e Yacc

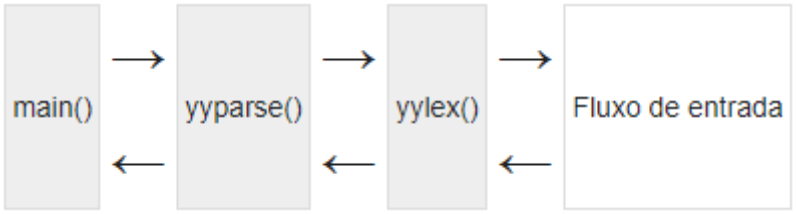


Figura 6 - Relação entre Yacc e Lex

*Python* parte cada linha lógica numa sequência de componentes léxicos elementares conhecidos como *tokens. Tokens* dividem-se em quatro tipos:

- Identificadores
- Palavras chave
- Operadores
- Delimitadores

```
tokens = ("forward", "fd", "right", "rt", "back", "bk", "left",
          "lt", "setpos", "setxy", "setx", "sety", "home", "pendown",
          "pd", "penup", "pu", "setpencolor", "make", "repeat", "while",
          "if", "ifelse", "NUMBER", "TO", "END", "STR", "VAR", "OPERATOR", "SIGN")
```

Figura 7 - Lista de  tokens usada neste projeto

```
# IF | IFELSE
def p_command14(self, p):
    """ command : if value SIGN value '[' program ']'
                | ifelse value SIGN value '[' program ']' '[' program ']' """

    args = {'value1': p[2], 'sign': p[3], 'value2': p[4], 'code1': p[6]}

    if len(p) == 11:
        args['code2'] = p[9]

    p[0] = Command("if", args)
```

Figura 8 - Comando *"if"* e *"ifelse"*

Utilizamos também o conceito de Árvore Sintática Abstrata lecionado em sala de aula.

*"In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.*

*The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches."*

(Retirado de Wikipedia: https://en.wikipedia.org/wiki/Abstract_syntax_tree)

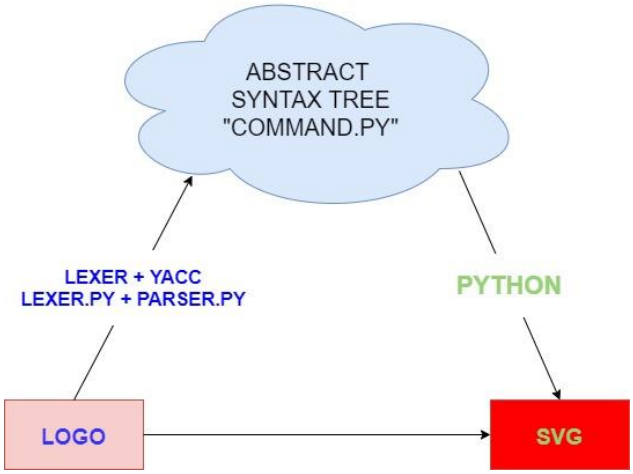


Figura 9 - Árvore Sintática Abstrata

De seguida observa-se um excerto do resultado da interpretação/tradução de um ficheiro *Logo* com cerca de 100 comandos.

```
x2="100.0" y1="1.0" y2="1.0" /><line x1="1.0" x2="1.0" y1="1.0" y2="1.0" /><line x1="1.0" x2="1.0" y1="10.0" y2="10.0" /><line
x1="10.0" x2="10.0" y1="1.0" y2="1.0" /><line x1="1.0" x2="1.0" y1="1.0" y2="1.0" /><line x1="1.0" x2="1.0" y1="1.0" y2="1.0" /
><line x1="1.0" x2="1.0" y1="1.0" y2="1.0" /><line x1="1.0" x2="1.0" y1="1.0" y2="1.0" /><line x1="1.0" x2="1.0" y1="1.0" y2="1.0" /
><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250" x2="250.0" y1="250" y2="280.0" /><line stroke="rgb(1%,1%,1%)"
stroke-width="2" x1="250.0" x2="220.0" y1="280.0" y2="280.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="220.0" x2="220.0"
y1="280.0" y2="250.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="220.0" x2="250.0" y1="250.0" y2="250.0" /><line stroke="rgb
(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="250.0" y2="260.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0"
x2="240.0" y1="260.0" y2="260.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="240.0" x2="240.0" y1="260.0" y2="250.0" /><line
stroke="rgb(1%,1%,1%)" stroke-width="2" x1="240.0" x2="250.0" y1="250.0" y2="250.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2"
x1="250.0" x2="250.0" y1="250.0" y2="251.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="251.0" y2="252.
0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="252.0" y2="253.0" /><line stroke="rgb(1%,1%,1%)"
stroke-width="2" x1="250.0" x2="250.0" y1="253.0" y2="254.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0"
y1="254.0" y2="255.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="255.0" y2="256.0" /><line stroke="rgb
(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="256.0" y2="257.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0"
x2="250.0" y1="257.0" y2="258.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="258.0" y2="259.0" /><line
stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="259.0" y2="260.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2"
x1="250.0" x2="250.0" y1="260.0" y2="261.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="261.0" y2="262.
0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="262.0" y2="263.0" /><line stroke="rgb(1%,1%,1%)"
stroke-width="2" x1="250.0" x2="250.0" y1="263.0" y2="264.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0"
y1="264.0" y2="265.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="265.0" y2="266.0" /><line stroke="rgb
(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="266.0" y2="267.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0"
x2="250.0" y1="267.0" y2="268.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="268.0" y2="269.0" /><line
stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="269.0" y2="270.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2"
x1="250.0" x2="250.0" y1="270.0" y2="271.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="271.0" y2="272.
0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="272.0" y2="273.0" /><line stroke="rgb(1%,1%,1%)"
stroke-width="2" x1="250.0" x2="250.0" y1="273.0" y2="274.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0"
y1="274.0" y2="275.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="275.0" y2="276.0" /><line stroke="rgb
(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="276.0" y2="277.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0"
x2="250.0" y1="277.0" y2="278.0" /><line stroke="rgb(1%,1%,1%)" stroke-width="2" x1="250.0" x2="250.0" y1="278.0" y2="279.0" /><line
```

Figura 10 - "Tradução" de comandos para SVG

Trata-se de um ficheiro do tipo *SVG* que sendo aberto com um visualizador de imagens obtemos a seguinte figura:

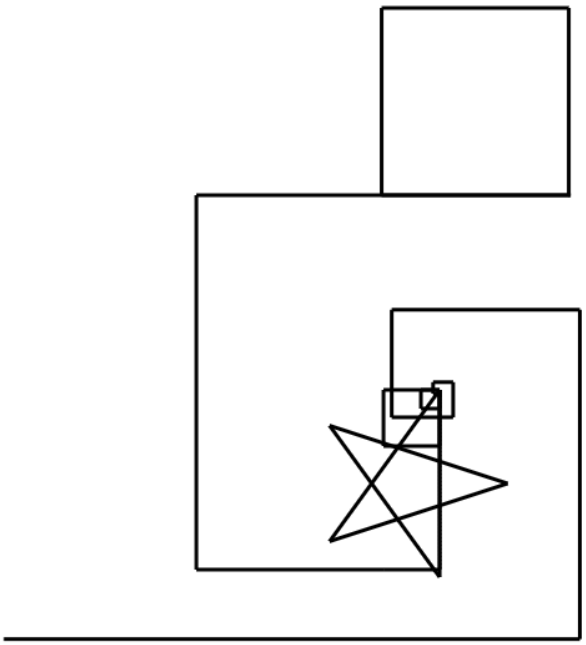

Figura 11 - Resultado da "tradução"

# Conclusão

Na realização deste projeto deparamo-nos com algumas dificuldades iniciais, devido ao tema, pois estávamos a interpretar erradamente o enunciado o que provocou alguma perda de tempo. No entanto assim que conseguimos entender melhor com a ajuda do Professor conseguimos executar todos os pontos pedidos com alguma dificuldade, mas sempre progredindo em direção ao objetivo.

Finalizado este desafio, conseguimos perceber de uma forma mais aprofundada de como podemos fazer o processamento de uma linguagem, o que impulsionou o nosso pensamento lógico e crítico, especialmente como programadores.

Através deste trabalho prático, foi possível analisar alguns conceitos e perceber a sua aplicação nos vários campos da computação.