**Entrega Trabalho Prático Portugol**

**Processamento de Linguagens**

**Aluno/os:**

21140 - Pedro Vieira Simões

21149 - Duarte Ribeiro de Melo

**Professor/es: Alberto Simões**

**Licenciatura em Engenharia de Sistemas Informáticos**

Barcelos, Janeiro de 2022

Índice

[Introdução 3](#_Toc93186273)

[Abordagem geral do problema 4](#_Toc93186274)

[Lexer 5](#_Toc93186275)

[Tokens 5](#_Toc93186276)

[Expressões Regulares 5](#_Toc93186277)

[Gramática 7](#_Toc93186278)

[Abordagem gramatical do problema / solução 7](#_Toc93186279)

[Produções / gramática 7](#_Toc93186280)

[Explicação das produções da gramática 8](#_Toc93186281)

[Abordagem/programação da gramática em Python 8](#_Toc93186282)

[Eval 9](#_Toc93186283)

[Interpretador 9](#_Toc93186284)

[Writer 9](#_Toc93186285)

[Outputs 9](#_Toc93186286)

# Introdução

Este relatório visa explicar a abordagem ao problema A colocado no enunciado do Trabalho Prático 2 da disciplina de Processamento de Linguagens do curso de Engenharia de Sistemas Informáticos lecionada pelo Professor Alberto Simões no Instituto Politécnico do Cávado e do Áve no ano letivo 2021/2022.

O tema / problema A do enunciado é relativo à criação de um interpretador em *yacc* para a linguagem Portugol e, para além disso, a geração do código C referente ao algoritmo processado.

Citando o enunciado – “O Portugol é uma linguagem algorítmica escrita em português. O seu objetivo é, ser por um lado, uma forma de os estudantes de programação começarem a esquematizar os seus programas, mas também garantir que os algoritmos definidos não se limitam ao papel, e podem ser executados.”

Após leitura e análise da sintaxe desta linguagem, revisão de conceitos teóricos dados nas aulas e leitura e análise de código escrito pelo professor, optou-se por este tema dada alguma familiaridade com Portugol / pseudo-código e C, alguma possibilidade de aproveitamento do código escrito em aula e dificuldade do tema.

# Abordagem geral do problema

Após leitura e análise da sintaxe, tipagem e regras do Portugol, construiu-se uma gramática que corresponde às regras e ordem de instruções desta linguagem. A gramática foi construída de forma que o utilizador que insira o código tenha de cumprir determinada ordem de execuções, porém, mantendo sempre a liberdade para que o utilizador consiga dar asas à sua imaginação e assim executar diferentes algoritmos.

Foram também escritas expressões regulares, que visam parametrizar lexicalmente o *input* recebido.

Após as expressões regulares, construiu-se um *eval* que visa interpretar e avaliar o código que, previamente passou pela confirmação da gramática e do *lexer*, e, neste ficheiro Python, ganhará a devida funcionalidade esperada pelo utilizador.

# Lexer

## Tokens

Como forma de abordagem, foi pensado em usar um conjunto de palavras-chaves como *tokens* sendo estas as funcionalidades/elementos do Portugol, tais como: início, fim, símbolos, tipos de variáveis, *strings*, números, booleanos, comparadores, palavras inicializadoras de estruturas de ciclo e decisão/condição, funções de leitura e escrita e a declaração/atribuição de variáveis.

Consoante estes *tokens*/funcionalidades necessários, foi necessário criar expressões regulares para algumas destas, cujas quais surgem inframencionadas.

## Expressões Regulares

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê o comentário pretende ignorá-lo e, por isso, não é feito nenhum *handling* daquilo que é lido por esta expressão regular.

Presume-se que um comentário seja uma linha ou parte desta que comece com um “#”.

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê uma *string* pretende devolver o valor que é lido, de forma a ser passado ou posteriormente avaliado pelo eval.

Presume-se que uma *string* é qualquer cadeia de caracteres.

São removidos o primeiro e último caracter, as aspas.

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê um número pretende devolver o valor que é lido, de forma a ser passado ou posteriormente avaliado pelo eval.

Presume-se que um número é um ou vários dígitos seguidos ou não de um ponto e caso esse ponto exista, existe pelo menos um dígito depois dele.

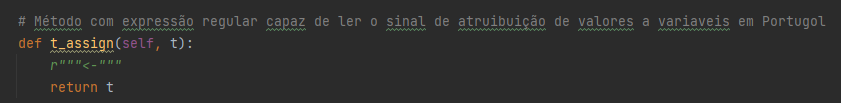
O valor lido é passado para *float*, o tipo de dados com que trabalhamos neste projeto sempre que se trata de números.

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê um tipo de variável pretende devolver aquilo que é lido, de forma a ser passado ou posteriormente avaliado pelo eval.

Presume-se que um tipo de variável seja uma destas *strings* (não confundir, obviamente, esta *string* com a da outra expressão regular) – “inteiro”, “caracter”, “logico” ou “real”.



A expressão regular que lê o *token* *assign* pretende ler a cadeia de caracteres que se utiliza quando queremos atribuir um valor a uma variável. Exemplo: a <- 5

Este *token* é retornado, pois é importante para a interpretação futura do código.

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê o *token* leq, pretende ler a cadeia de caracteres que compara se um valor é menor ou igual que outro. Exemplo: 4 <= 5

Este *token* é retornado, pois é importante para a interpretação do código.

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê o *token* dif, pretende ler a cadeia de caracteres que compara se um valor é diferente de outro. Exemplo: 4 != 5

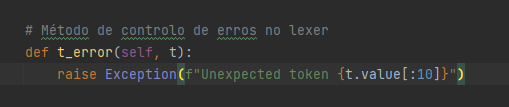
Este *token* é retornado, pois é importante para a interpretação futura.

Uma imagem com texto

Descrição gerada automaticamente

A expressão regular que lê o *token* *keywords*, capaz de ler todos os tipos de *keywords* presentes no lexer, no caso da palavra não se encontrar na lista de *keywords*, então assume-se que esta *string* é um nome de uma variável.

Basicamente, presume-se que qualquer “*keyword”* que não seja efetivamente *keyword* refere-se a uma variável.



Este procedimento é apenas para controlo de erros no *lexer*, basicamente, caso algo não seja esperado por ele, é escrito este erro no ecrã.

# Gramática

## Abordagem gramatical do problema / solução

A forma como foi abordada e construída a gramática deste problema visa corresponder à tipagem / formato da linguagem Portugol. Por exemplo, em Portugol, as funções surgem antes do “início”, as declarações de variáveis são necessárias e os ciclos “if” são escritos de uma determinada forma, e dentro destes podemos ter um determinado conjunto de instruções – com todas estas limitações / obrigações do Portugol, surge então a gramática que deve conseguir cumprir esta “ordem” de operações e obrigatoriedade de determinadas regras.

## Produções / gramática

portugol -> func\_list ';' inicio code

func\_list -> func | func\_list ';' func

func -> FUNCAO var '(' args ')' com\_list ';' FIMFUNCAO

code -> s | code ';' s

s -> comando | fim

ciclo -> PARA var DE e ATE e FACA com\_list ';' FIMPARA

ciclo2 -> ENQUANTO n FACA com\_list ';' FIMENQUANTO

comando -> e | ciclo | VAR assign e | ESCREVA '(' e\_list ')'

| LEIA '(' var\_list ')' | VARTYPE ':' var\_list

| SE e ENTAO com\_list ';' SENAO com\_list ';' FIMSE

| SE e ENTAO com\_list ';' fimse

com\_list -> comando | com\_list ';' comando

e\_list -> e | e\_list ',' e

e -> var | '(' e ')' | b | n | string | var '(' e\_list ')' | VAR '(' ')'

n -> NR | '-' e | e '+' e | e '-' e | e '\*' e | e '/' e

| e '<' e | e leq e | e '>' e | e geq e | e '=' e | e dif e

b -> f | e OR e | e AND e | e XOR e

f -> true | false | not f

var\_list -> var | var\_list ',' var

args: € | var\_list

## Explicação das produções da gramática

A seguinte explicação tenta, de forma sucinta e numa tentativa de explicar de forma “temporal” ou por ordem de instruções a gramática implementada.

No Portugol, podemos ou não ter funções (uma só ou várias) declaradas, e essas mesmas funções surgem antes da palavra “início”.

Depois do “início” surge então o “código”, código este que pode ser repetido as vezes que o utilizador quiser até que chegue o “fim”.

Este código (s) pode então ser um comando ou a palavra “fim”.

Um comando pode ser uma variável, a chamada de uma função, um ciclo, uma atribuição de valor a uma variável, um “escreva” (*output* de texto no ecrã), um “leia” (*input* do utilizador para uma variável), uma declaração de variável, condições (se, então, etc).

Dentro de um ciclo ou dentro de uma condição podemos escrever mais comandos.

Um “número” (produção) pode ser apenas efetivamente um número, uma conta ou uma comparação entre números.

Um “booleano” (produção) pode ser apenas efetivamente um booleano ou comparações entre booleanos.

## Abordagem/programação da gramática em Python

Precedências

Uma imagem com texto

Descrição gerada automaticamente

A lista de precedências organiza as operações por ordem de prioridade, isto é, as operações que surgem em cima têm menos prioridade que as que surgem mais em baixo.

A nossa solução necessita de uma lista de precedências pois há operações que devem ter prioridade sobre outras. Ex: no caso de o utilizador inserir “3+4\*5” a multiplicação deverá ser feita primeiro. Analisando o código, percebemos que a multiplicação na lista de precedências tem prioridade sobre a soma.

A lógica por trás da gramática implementada através de produções em *yacc* está explicada na gramática acima.

Apenas são guardados os dados que sejam relevantes para futuramente interpretar o código em Portugol no *eval*.

Uma imagem com texto

Descrição gerada automaticamente

A função *parse* visa conferir e “passar pela gramática” o código lido, depois do código cumprir com as produções, é passado para o *eval*, que o interpretará / avaliará.

# Eval

Como supramencionado, “após as expressões regulares, construiu-se um *eval* que visa interpretar e avaliar o código que, previamente passou pela confirmação da gramática e do *lexer*, e, neste ficheiro Python, ganhará a devida funcionalidade esperada pelo utilizador.”

Como o código introduzido pelo utilizador podia seguir dois “caminhos”, o de interpretar ou gerar o código C, criamos dois *evals* de forma a que o código fosse avaliado de formas diferentes.

## Interpretador

## Writer

# Outputs