**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](#_Toc93251502)

[Abordagem geral do problema 4](#_Toc93251503)

[Lexer 5](#_Toc93251504)

[Tokens 5](#_Toc93251505)

[Expressões Regulares 5](#_Toc93251506)

[Gramática 8](#_Toc93251507)

[Abordagem gramatical do problema / solução 8](#_Toc93251508)

[Produções / gramática 8](#_Toc93251509)

[Explicação das produções da gramática 8](#_Toc93251510)

[Abordagem/programação da gramática em Python 9](#_Toc93251511)

[Eval 10](#_Toc93251512)

[Interpretador 10](#_Toc93251513)

[Abordagem do *eval\_interpretador* 10](#_Toc93251514)

[Operators 10](#_Toc93251515)

[Symbols / Symbol Table 11](#_Toc93251516)

[Métodos do *Eval\_Interpreter* 11](#_Toc93251517)

[Writer 17](#_Toc93251518)

[Abordagem do *eval\_writer* 17](#_Toc93251519)

[Operators 17](#_Toc93251520)

[Métodos 18](#_Toc93251521)

# 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 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.

Em suma, reuniu-se alguma informação inerente ao Portugol e à UC e, consoante a informação recebida deu-se então a criação da solução, que visa receber um ficheiro de texto em Portugol, passá-lo por uma gramática que confirma se o ficheiro inserido cumpre com as exigências das produções e das expressões regulares e, por fim, interpretar este código.

No decorrer das próximas páginas, a abordagem é explicada mais detalhadamente.

# 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.

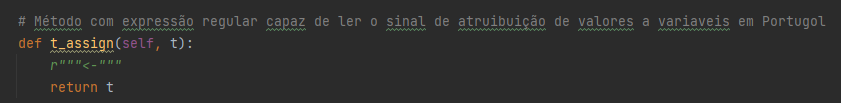
O valor lido é passado para *float*, o tipo de dados com que trabalhamos neste projeto sempre que se trata de números.

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

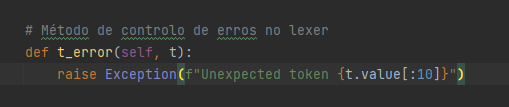
Este *token* é retornado, pois é importante para a interpretação futura.

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 que o código fosse avaliado de formas diferentes.

## Interpretador

### Abordagem do *eval\_interpretador*

O *eval\_interpreter* foi abordado com o objetivo de interpretar o pseudocódigo escrito pelo utilizador em um ficheiro. De forma que este consiga entender o que está a ser lido, foi necessário criar uma classe que consiga aceder às informações que foram registadas a partir da gramática implementada.



Esta classe contém 2 atributos:

* *Operators*
* *Symbols*

### Operators

*Operators* é uma tabela com os diferentes operadores, dos quais cada um obtém os argumentos guardados em listas ou dicionários, a forma como cada operação guarda os argumentos e onde cada uma é chamada está definida na gramática em cada produção.

A cada operação é indicado o procedimento de interpretação respetivo usando os argumentos guardados.

Uma imagem com texto

Descrição gerada automaticamente

### Symbols / Symbol Table

Como no pseudocódigo existe declaração de variáveis e atribuição de valores a variáveis e posteriormente é necessário chamar essas mesmas variáveis, elas têm de ser guardadas em algum sítio (assim como as funções).

Inicialmente o dicionário de símbolos encontrava-se no *eval* porém, com a necessidade de criar uma *stack* de variáveis – para, por exemplo, poder utilizar variáveis locais e apagá-las no fim da sua utilização – criou-se uma Classe Symbol Table que, como atributo, tem o dicionário de símbolos. Desta forma, conseguimos alterar as funções base de um dicionário, conseguindo assim converter o mesmo numa *stack* de variáveis.

Como o pseudocódigo (e o C) guardam os *vartypes* de cada variável, a forma como guardamos uma variável na nossa *stack* é esta:

Nome\_variável -> [vartype, valor]

### Métodos do *Eval\_Interpreter*

Uma imagem com texto

Descrição gerada automaticamente

Esta função visa retornar o valor armazenado numa variável.

Como mencionado acima, uma variável é guardada numa lista, cujo primeiro elemento é o *vartype* e o segundo o valor da variável.

O nosso objetivo com a função é, enquanto que o que se encontrar dentro de *value* for uma lista, *value* torna-se o último elemento de si próprio.

Uma imagem com texto

Descrição gerada automaticamente

Este procedimento visa declarar uma variável.

O valor, como se trata de uma declaração, é nulo.

Uma imagem com texto

Descrição gerada automaticamente

Função que visa chamar uma função.

Procura a função na tabela de símbolos e, caso exista, guarda em variáveis o seu código e parâmetros e, posteriormente, define os parâmetros recebidos e interpreta o código.

No fim, apaga as variáveis recebidas como parâmetros.

Caso a função não seja encontrada (não esteja definida), é retornado um erro.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento que visa declarar uma função, guardando a mesma na tabela de símbolos, para posteriormente poder ser chamada.

Uma imagem com texto, exterior, prateado, captura de ecrã

Descrição gerada automaticamente

Procedimento que visa escrever no ecrã um número, uma string, uma variável, etc.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento que interpreta o ciclo “for” / “para”.

Recebe o valor inicial – *lower* – e o máximo – *higher* – e coloca o valor de *lower* em *value*.

Depois, enquanto que o *value* for menor ou igual que *higher*, interpreta o código que está dentro do ciclo e incrementa o *value*.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento que visa interpretar o ciclo “while” / “enquanto”.

Enquanto a expressão recebida no primeiro *index* dos argumentos for verdadeira, executa o código que se encontra na segunda posição dos argumentos.

Uma imagem com texto

Descrição gerada automaticamente

Função que visa interpretar a condição “if” / “se”.

Caso não exista um “else” / “senão” e caso a expressão seja verdadeira, retorna a interpretação do código que se encontra dentro do “se”.

Caso exista “senão” e caso a expressão seja verdadeira, retorna a interpretação do código que se encontra dentro do “se”, caso a expressão seja falsa, retorna a interpretação do código que se encontra dentro do “else” / “senão”.

Uma imagem com texto

Descrição gerada automaticamente

O procedimento *assign* visa criar ou alterar valores da tabela de símbolos.

Uma imagem com texto

Descrição gerada automaticamente

Esta função visa alterar o valor de uma variável, ambos recebidos por parâmetros.

Confirma se a variável existe na tabela de símbolos.

Confirma se o *var\_type* na tabela de símbolos coincide com o tipo de variável que está a tentar atribuir à variável e, caso isso se verifique, altera o valor da variável na tabela de símbolos.

Caso estas confirmações não se verifiquem, são retornados erros.

Uma imagem com texto

Descrição gerada automaticamente

Esta função visa ler dados do *input* do utilizador para uma variável ou várias.

Para cada variável recebida nos parâmetros, confirma se esta existe na tabela de símbolos.

Recebe o *input* do utilizador e, caso possível, passa este mesmo *input* para *float*.

Depois, o valor da variável na tabela de símbolos é alterado.

Caso a função não exista na tabela de símbolos, é retornado um erro.

Uma imagem com texto

Descrição gerada automaticamente

Função que visa avaliar de diferentes formas a AST consoante o seu tipo de dados.

Uma imagem com texto

Descrição gerada automaticamente

Função que visa avaliar a AST caso seja um dicionário.

## Writer

### Abordagem do *eval\_writer*

O writer foi abordado tendo em conta a gramática implementada, utilizando assim a informação obtida nas diferentes operações implementadas no mesmo para escrever os comandos inseridos pelo utilizador em código C.



Este *eval* é composto por 4 atributos:

* *Operators*
* *Symbols*
* *f*, utilizado para abrir, escrever e fechar um file com o nome *“main.c”*
* *c\_code*, que é uma string que irá conter todo o código C final a ser escrito no ficheiro alvo

### Operators

Neste *eval cada um d*o*s* operadores, cuja função foi mencionada no *Interpreter*, sofreu algumas alterações, tendo em conta que o objetivo desta vez não é interpretar código mas sim escreve-lo num documento.

A alteração que mais se destaca é a o armazenamento de operações em formato *string*, para a maior facilidade de escrita das mesmas no *“main.c”*

Uma imagem com texto

Descrição gerada automaticamente

### Métodos

Uma imagem com texto

Descrição gerada automaticamente

Procedimento que visa escrever uma operação para o ficheiro C.

Com a função *return\_value\_of\_var* é obtido o real valor das variáveis e, posteriormente, é chamada a função *write­\_to\_file* para efetivamente escrever no ficheiro.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento que visa escrever os argumentos recebidos, sem os avaliar, para o ficheiro C.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento com funcionalidade de escrever a declaração de uma variável em C, sem atribuir um valor à mesma.

Tendo em conta o tipo de variável implementado em Portugol, é escolhido esse mesmo tipo com a sintaxe de C.

Uma imagem com texto

Descrição gerada automaticamente

Função que visa traduzir de Portugol para C a chamada de uma função.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento que escreve uma função em C, não executando a mesma.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento responsável por escrever a função de escrever um valor no terminal do utilizador (*printf()* em C).

Uma imagem com texto

Descrição gerada automaticamente

Procedimento com a funcionalidade de escrever a função enquanto/*while*() em C.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento responsável pela escrita de uma estrutura de decisão if em C, consegue também suportar else, para realizar um else if() é necessário incluir outra estrutura if dentro do else.

Uma imagem com texto

Descrição gerada automaticamente

Procedimento responsável pela escrita da alteração/atribuição de um valor a uma variável.

Uma imagem com texto

Descrição gerada automaticamente

Função que visa avaliar de diferentes formas a AST consoante o seu tipo de dados.

Caso a ast2 seja “inicio”, é escrito o *int main()* em C.

Caso seja “fim”, é fechada a função *int main*, o ficheiro C é escrito com tudo aquilo que foi incrementado à variável *c\_code* e é fechado.

# Outputs