IPCA

Escola Superior Tecnologia

Processamento de Linguagens

Trabalho Prático 2

Joel Phillippe Melo Figueiras, nº20809

Nuno Miguel Carvalho Araújo, nº 20078

Professor Alberto Simões

2021/2022

Indíce

1 Introdução **1**

1.1 Contextualização 1

1.2 Motivação e Objetivos 1

2 Portugol **2**

3 Os tokens **3**

4 A gramática **5**

5 A avaliação da gramática **7**

5.1 logic_eval.py 7

5.2 logic_eval_c.py 8

6 A execução do código **9**

7 Conclusão **10**

1 Introdução

1.1 Contextualização

No âmbito da unidade curricular de Processamento de Linguagens, foi pedido a elaboração de um dos dois temas propostos.

O tema escolhido foi o Tema A (Portugol) e pretende que coloquemos em prática os conceitos adquiridos ao longo da unidade curricular.

Para o desenvolvimento do projeto, é solicitada a utilização da ferramenta PLY.

1.2 Motivação e Objetivos

O presente trabalho prático tem como objetivo a implementação de um reconhecimento léxico e sintático da linguagem "Portugol".

Através desse reconhecimento terá de ser possível interpretar e executar o algoritmo processado bem como gerar o código em Linguagem C.

A opção a ser executada será escolhida pelo utilizador, por isso, o programa terá de estar preparado para executar ambas as funcionalidades de forma distinta.

2 Portugol

Como referenciado anteriormente, o trabalho tem como principal objetivo a leitura de um algoritmo em Portugol e a sua respetiva execução ou conversão para código em Linguagem C, conforme a escolha feita pelo utilizador.

Para dar início ao projeto tivemos de aprofundar os nossos conhecimentos em Portugol, pois era um tema ainda desconhecido para nós.

Após algum estudo e aprofundamento de conceitos, selecionamos os comandos em Portugol que pretendemos implementar no nosso trabalho, sendo eles os seguintes:

- Identificação do tipo e assinatura de variáveis – consiste em identificar o tipo de variável (inteiro, real, carater ou lógico) e a sua assinatura. Exemplo: "inteiro: num;".
- Atribuição de valores à variável – consiste na atribuição de valores às variáveis previamente assinadas. Exemplo: "num $\leftarrow$ 3;".
- Escrever – Representa a impressão de conteúdo como textos, valores das variáveis, etc. Exemplo: "escreva(num);".
- Ler – Comando responsável pela leitura de valores no input e a respetiva atribuição às variáveis. Exemplo: "leia(num);".
- Se – Comando para identificação e validação de condições. Exemplo: "se (num $>$ 5) então ... fim_se".
- Se ... senão – Comando para validação de condições e execução em conformidade com a validação (verdadeira ou falsa). Exemplo: "se (num $>$ 5) então ... senão ... fim_se".
- Enquanto – Estrutura de repetição cujo objetivo é a verificação da condição de forma recursiva até que esta seja verdadeira. Exemplo: "enquanto (num $>$ 5) faça ... fim_enquanto".
- Para – Estrutura de repetição para verificação de condição de forma recursiva com incremento automático. Exemplo: "para num de 5 até 10 passo 1 faça ... fim_para".
- Função – Sub-rotina com o objetivo de realizar uma tarefa especifica quando chamada. Exemplo: "função inteiro fun(inteiro num): ... fim_função".
- Chamada – Comando para chamada da função previamente mencionada.
  Exemplo: "num $\leftarrow$ fun(7)".

3 Os tokens

Após a seleção dos comandos de Portugol a ser implementados, criamos o ficheiro logic_lexer.py para identificação das keywords e dos tokens necessários ao desenvolvimento do programa.

Em relação às keywords, por uma questão de organização, optamos por fazer a seguinte divisão:

- varTypes - Keywords usadas nos tipos de variável (”inteiro”, ”real”, carater” e ”logico”)
- logic - keywords usadas nos estados lógicos das funções (”verdadeiro”, ”falso”, ”e”, ”ou” e ”xou”)
- conditions - keywords usadas na implementação de condições (”se”, ”fim_se”)
- cycles - keywords usadas na implementação de ciclos (”enquanto”, ”fim_enquanto”, ”para”, ”de”, ”passo” e ”fim_para”)
- other - outras keywords usadas no programa (”Inicio”, ”Fim”, ”escreva”, ”leia”, ”retorna”, ”void”)

Devidamente identificadas as keywords, procedemos ao reconhecimento dos sinais literais necessários, que foram os seguintes: ”<>()+-/*;[],:=”. Decidimos também criar uma regra denominada de ”t_ignore” para ignorar os espaços e os ”$\backslash t \backslash n$”.

Por fim, chegou a altura da identificação e da implementação das expressões regulares para os tokens necessários.

Em relação aos tokens, fizemos a sua divisão em três segmentos:

1º segmento - os tokens usados apenas para identificação da palavra na língua portuguesa (aceitação de sinais de acentuação, etc.), sendo eles, ”funcao”, ”fim_funcao”, ”nao”, ”senao”, ”entao”, ”ate” e ”faca”. A regra utilizada na expressão regular destes tokens é apenas reescrever a palavra corretamente na nossa língua.

2º segmento - os tokens usados para reconhecimento léxico, sendo eles:

- **dif:** reconhece o sinal ”!=” utilizado em portugol para verificar se um valor é diferente do outro.
- **eq:** reconhece o sinal ”==” utilizado em portugol para verificar se um valor é igual a outro.

- **maieq:** reconhece o sinal ">=" utilizado em portugol para verificar se um valor é maior que o outro
- **meneq:** reconhece o sinal "<=" utilizado em portugol para verificar se um valor é menor que o outro.
- **assing:** reconhece o sinal "←" utilizado em portugol para atribuição de valor a uma variável.
- **float(e.r.: "[0-9]+\.[0-9]+"):** reconhece valores de virgula flutuante.
- **int (e.r.: "[0-9]+"):** reconhece valores inteiros.
- **string (e.r: "[^"]*"):** reconhece textos que estão dentro de aspas.

3º segmento - token para identificação de palavras. Se a palavra for reservada é identificada como keyword, caso contrário é definida como nome de variável (*varName*).

4 A gramática

À medida que criamos o reconhecimento dos tokens e das keywords no ficheiro *logic_lexer.py*, começamos também a planear a estrutura da gramática a ser implementada no ficheiro *logic_grammar.py*.

No entanto, a gramática foi sofrendo alterações, conforme as necessidades, ao longo da implementação do projeto.

No final, obtivemos uma gramática com a seguintes estrutura:

- port : Inicio code Fim | fun Inicio code Fim
- code : code | code com
- fun : funcao varType varName '(' vars ')' ':' code fim_funcao
  | funcao void varName '(' vars ')' ':' code fim_funcao
- vars : Ø | varType varName | vars ',' varType varName
- com : lines | cond | cycle
- lines : varName assing value ';' | varType ':' varName_list ';'
  | escreva value_list ';' | leia value_list ';' | retorna value ';'
- cond : se value entao code fim_se | se value entao code senao code fim_se
- cycle : para varName de value ate value passo value faca code fim_para
  | enquanto value_list faca code fim_enquanto
- value : varName | bool | calc | string | varName '(' value_list ')' | varName '(' ')'
- varType : inteiro | real | carater | logico
- varName_list : varName | varName_list ',' varName
- value_list : value | value_list ',' value | '(' value_list ')'
- bool : opt | value e value | value ou value | value xou value
- calc : int | float | '-' value %prec uminus | value '+' value | value '-' value | value '*' value
  | value '/' value | value '<' value | value '>' value | value dif value | value eq value
  | value maieq value | value meneq value
- opt : verdadeiro | falso | nao opt

Procedendo agora à explicação da gramática:

- **port:** é o nosso estado inicial. Dependendo da estrutura do ficheiro interpretado o estado seguinte pode ser o *fun* ou o *code*.
- **code:** o estado *code* pode ser apenas uma linha de código ou um conjunto de comandos.
- **fun:** este estado define as sub-rotinas que podem ser implementadas, sendo elas funções ou procedimentos (*void*).
- **vars:** estado que identifica os dados que passam por argumento para as sub-rotinas, sendo estes dados o tipo de variável e a variável a ser usada.
- **com:** neste estado temos a identificação dos comandos executáveis no programa. Optamos por efetuar a divisão em três comandos distintos, sendo eles as *lines*, as *cond* e os *cycles*.
- **lines:** como descrito anteriormente, as *lines* fazem parte dos comandos executáveis. Neste estado estão delineados os comandos de código mais simples como a assinatura de variáveis, a atribuição de valores, a escrita, a leitura, etc.
- **cond:** aqui temos reconhecidas as condições "se" e "se ... senão".
- **cycles:** o último elemento dos comandos executáveis são os *cycles* onde são identificados os ciclos "para" e "enquanto".
- **value:** estado onde são especificados os valores que as variáveis podem receber.
- **varType:** estado que identifica quais os tipos que as variáveis podem tomar, sendo eles "inteiro", "real", "carater" e "logico".
- **varName_list:** estado útil para a necessidade de armazenar mais do que uma variável.
- **value_list:** tem a mesma funcionalidade que *varName_list*, mas para o armazenamento de mais que um valor.
- **bool:** identifica os estados e cálculos booleanos.
- **calc:** neste estado estão discriminadas todas as operações matemáticas que podem ser executadas.
- **opt:** apura os estados booleanos existentes, "verdadeiro", "falso" e "não opt".

5 A avaliação da gramática

Finalizada e delineada a estrutura da gramática a ser implementada, procedemos à estruturação dos ficheiros responsáveis pela sua validação.

Com o receio de que a validação da gramática para execução do algoritmo pudesse variar muito da validação para escrita em Linguagem C, optamos por efetuar as validações de forma distinta.

Para essas validações foram criados dois ficheiros, o *logic_eval.py* e o *logic_eval_c.pt*.

5.1 logic_eval.py

Começamos então por criar o ficheiro que validasse a gramática e executasse o algoritmo processado.

Criamos também listas para armazenar as variáveis de cada tipo, ou seja, criamos uma lista para guardar as variáveis do tipo inteiro, outra para as do tipo real, etc.

A primeira função a ser implementada foi a função *"eval"* que tem como objetivo receber e identificar a árvore de sintaxe abstrata.

Sendo a árvore identificada, ela é enviada para a segunda função implementada, *"_eval_dict"*. Esta função tem a finalidade de percorrer a árvore recebida e validar os seus dados um por um. Para a sua validação é necessário identificar o operador e efetuar a respetiva operação.

Dada a necessidade da identificação dos operadores mencionada anteriormente, criamos a tabela dos operadores. Esta tabela é constituída por todos os operadores utilizados na gramática, sendo eles operadores matemáticos, lógicos e de controlo manual (CM).

Matemáticos	Lógicos	CM
+	ou	*assign*
-	e	*escreva*
*	xou	*leia*
/	nao	*if*
	<	*if_else*
	>	*while*
	!=	*for*
	==	*fun*
	>=	*call*
	<=	

Definidos os operadores, foi necessário criar regras para a sua implementação. No caso dos operadores matemáticos e lógicos a operação é feita de forma autónoma com recurso à linguagem de programação utilizada, que, no nosso caso, foi o *python*.

Para os restantes operadores tivemos de criar as regras para a sua implementação.

- **operador *assing*:** recebe um valor e uma variável. A primeira coisa que faz é verificar qual o tipo de variável recebida. Em seguida, confirma se o tipo de valor inserido está de acordo com o tipo de variável. Em caso afirmativo, a variável é armazenada com o respetivo valor. Caso contrário, é encontrada uma *"Exception"* no programa.
- **operador *escreva*:** tem como finalidade imprimir os argumentos recebidos.
- **operador *leia*:** faz a leitura de valores inseridos no *"input"* do programa.
- **operadores *if* e *if_else*:** são dois operadores que recebem uma condição e executam o código dependendo da condição. No caso do *"if"* só executa o código se a condição se verificar. Em relação ao *"if_else"*, ele recebe dois blocos de código, um é executado se a condição se verificar e o outro é executado caso não se verifique.
- **operadores while e for:** são operadores que recebem uma condição e o código. A verificação da condição é feita de forma recursiva e, enquanto falsa, o código é executado. A diferença entre os dois operadores é que no *"while"* o incremento tem de ser efetuado de forma manual, enquanto no *"for"* é executado de forma automática.
- **operadores *fun* e *call*:** são operadores responsáveis pela implementação e execução de sub-rotinas. O *"fun"* tem como objetivo fazer a assinatura da função, enquanto o *"call"* recebe os argumentos e procede à execução da sub-rotina.

5.2 logic_eval_c.py

Quando concluído o ficheiro para avaliação da gramática e execução do algoritmo criamos, então, outro ficheiro para avaliação e conversão do algoritmo para Linguagem C.

Em relação à estrutura do ficheiro, é similar ao *"logic_eval.py.* Contém também a tabela dos operadores, as listas para armazenamento das variáveis por tipo e as funções de identificação e validação da árvore de sintaxe abstrata.

A única diferença é que nos operadores de controlo manual, as regras de implementação são apenas regras de impressão de texto. Os dados são recebidos por parâmetros nas funções e formatados de forma a que sejam escritos em Linguagem C.

6 A execução do código

Para executar os ficheiros mencionados nos tópicos anteriores e cumprir com os objetivos do enúnciado, foi necessário acrescentar mais funcionalidades ao código. Para isso criamos dois ficheiros, o *"logic.py"* e o *"my_lib.py"*.

O ficheiro *"logic.py"* foi definido como ficheiro principal na execução do programa como um todo. Neste ficheiro criamos um pequeno algoritmo que questiona se o utilizador pretende executar o algoritmo ou gerar o código C do ficheiro portugol selecionado.

A opção selecionada pelo utilizador é enviada para a função *"run_batch"* inserida no ficheiro *"my_lib.py"* e, conforme a opção pretendida, é executada a validação da gramática.

No ficheiro *"my_lib.py"* temos também outras funções que são úteis ao longo da execução do programa. Essas funções são maioritariamente para manipulação de estruturas de dados.

Com a execução do código são também gerados dois ficheiros, o *"parser.out"* e o *"parsetab.py"*. Estes ficheiros são gerados automaticamente a partir da utilização da biblioteca *"ply.yacc"* e são documentos úteis para a verificação da estrutura e da implementação da gramatica.

7 Conclusão

De uma forma geral, ficamos satisfeitos com o trabalho desenvolvido.

Falando concretamente das duas partes distintas do trabalho, achamos que fizemos um melhor trabalho na execução do algoritmo. Nesta parte, conseguimos executar todos os comandos de portugol selecionados.

No entanto, na parte da escrita em código C, tivemos algumas dificuldades e conseguimos apenas implementar alguns comandos de forma correta.

Temos consciência que podíamos ter implementado melhor o enunciado pretendido , mas, infelizmente, não tivemos tempo para tal.

Contudo, concluímos que a estrutura do trabalho implementada foi útil para a compreensão e interiorização dos conteúdos lecionados durante a unidade curricular de Processamento de Linguagens.