Universidade do Minho

Licenciatura em Engenharia Informática

Processamento de Linguagens

Grupo 20

TP2: Compilador simPLY para PLY-Python

Joana Maia Teixeira Alves (A93290)

Maria Eugénia Bessa Cunha (A93264)

Vicente Gonçalves Moreira (A93296)

Maio 2022

Conteúdo

1 Introdução

Este trabalho prático foi realizado no âmbito da cadeira de Processamento de Linguagens, onde nos foi proposto uma série de enunciandos com desafios a resolver, sendo o nosso grupo responsável por decidir qual tema a ser abordado.

Para este trabalho prático, optamos por realizar o enunciado número 3, relativo à criação de um tradutor PLY-Simples para PLY, onde o objetivo é desenvolver uma sintaxe mais simples da bibloteca PLY que traduza ao máximo as funcionalidades presentes nesta, assim como criar um parser que tenha a capacidade de ler esta sintaxe simplificada e gerar o código *python* correspondente na sua sintaxe original.

1.1 Fases do processo de desenvolvimento

De forma a dividir e modularizar partes do projeto, assim como facilitar no processo de desenvolvimento do compilador final, decidimos criar um planeamento com 6 fases distintas de exploração, decisão e implementação. Estas são as seguintes:

- **Definição da linguagem "SPLY"**, analisando em grupo quais as vantagens e desvantagem da escrita de código PLY, procurando saber que problemas existem e que poderão ser resolvidos pela nossa linguagem, de forma a simplificar a escrita e melhorar a qualidade de interação ao utilizador. Também definimos quais os parâmetros e funcionalidades da biblioteca PLY que seriam adaptados pela linguagem.

- **Definição da Gramática**, definindo assim não só o modo de escrita da nossa linguagem, como também a ordem de escrita dentros dos vários parâmetros suportados.

- **Criação do Lexer -** Depois da gramática definida, começamos por explorar os vários símbolos terminais da nossa linguagem SPLY, decidindo quais seriam as suas palavras reservadas e o seu modo de escrita.

- **Criação do Parser -** Utilizando a gramática gerada, criamos o *parser* de forma a seguir ao rigor esta gramática. Este *parser* será também responsável pela recolha inicial de informação, assim como a deteção de erros encontrados na leitura individual dos parâmetros.

- **Verificador e Escritor -** Depois da informação recolhida, é necessário gerar um verificador que analise os dados por completo, encontrando possíveis problemas no código a ser gerado. Após verificado, esta informação passa pelo Escritor, que irá traduzir todos os dados lidos em código Python, conforme a sintaxe definida pela biblioteca PLY.

- **Aplicação Final (Main) -** Por último, unimos as várias componentes desenvolvidas e realizamos a aplicação final, responsável por todos os passos de compilação, assim como tratamento de erros e leitura de argumentos a serem definidos.

Iremos de seguida explorar cada fase em detalhe, expondo em cada as nossas decisões, justificações e estratégias de implementação.

2 Linguagem SPLY

Começamos a definição da nossa linguagem por analisar a sintaxe sugerida no enunciado do trabalho prático. Com isto, decidimos basear a nossa linguagem neste exemplo, utilizando os caracteres especiais '%%' para indicadores de headers/secções **(Lex ou Yacc)** e o caracter '%' para indicadores de parâmetros dentro dessa secção, ou para palavras reservadas da linguagem.

Decidimos permitir a troca de secções conforme o utilizador desejar, assim como permitir "interrupções"entre seccções, para a escrita de código do utilizador a não ser analisado. Decidimos, também, suportar a criação de comentários, utilizando a mesma sintaxe que a linguagem Python. Para além disto, decidimos que sintáticamente, um ficheiro vazio deveria ser aceite pela nossa linguagem, no entanto o compilador irá rejeitar a compilação deste visto que não existe qualquer conteúdo relevante para o mesmo.

Começamos por enunciar a sintaxe básica para o início da linguagem:

- **Início de Seccção LEX:** Palavra reservada - **%%LEX**. A partir desta palavra reservada, serão lidos todos os parâmetros associados ao módulo LEX. Este pode não conter parâmetros.
- **Início de Seccção YACC:** Palavra reservada - **%%YACC**. A partir desta palavra reservada, serão lidos todos os parâmetros associados ao módulo YACC. Este pode não conter parâmetros.
- **Fim de Seccção:** Palavra reservada - **%%END**. A partir desta palavra reservada, é "fechada"a leitura da secção anterior, podendo ser escrito código que não será processado ou abrir novamente outra secção.
- **Parâmetro Error:** Visto que, em ambos os módulos este parâmetro pode ser utilizado, decidimos utilizar a mesma sintaxe desta em ambas as secções. De forma a simplificar a escrita deste parâmetro, decidimos que este deverá ser declarado com uma mensagem de erro (opcional), seguido do tratamento a ser efetuado (obrigatório), neste caso, ignorar o erro ou abandonar a execução do programa.

  Formato Ex: %error = "Mensagem de Erro" %skip

  %error = %noskip

2.1 Lex

- **Suporte de contextos:** Visto que os contextos/estados dentro do lexer são bastantes úteis na análise léxica, decidimos incluir o suporte para estes. Definimos assim que a sua declaração inicial seria opcional e semelhante à sintaxe original, tendo que o utilizador enumerar num tuplo de múltiplos elementos vários tuplos contendo os nomes dos contextos e o seu tipo.

  Formato Ex: %contexts = (("banana","inclusive"),("laranja","exclusive"))

- **Declaração de Literals:** Visto que a declaração deste parâmetro já é simples na sua sintaxe original, decidimos replicá-la.

  Formato Ex: %literals = ".,()"

- **Declaração de Ignore:** Esta também é uma declaração direta e simples, tendo, adicionalmente, suporte para associação de contextos, ou seja, regra *ignore* para um dado contexto.

  Formato Ex: %ignore = " \n\t\r"
  %ignore (contexto) = " \n\t\r"

- **Declaração de Tokens:** Visto que, na sintaxe da biblioteca Python é necessário não só criar as regras para os *tokens* como também criar uma listagem destes, obrigando a existência de um cuidado extra na adição/remoção de *tokens*, decidimos tornar a declaração destes ímplicita, ou seja, o utilizador apenas define as regras do *token* e este será adicionado à listagem de *tokens* automaticamente.

  Decidimos também incluir várias *features*, como a associação do contexto à qual a regra desse token é aplicado, a definição de uma função de conversão do tipo de dado, à escolha do utilizador e, por último, uma potencial mudança de contexto dentro dessa regra.

  Formato Ex: ID = r'[a-zA-Z]\w*'
  INT = r'\d+' %int
  INT (contexto) = r'\d+' %int %begin INITIAL

2.2 Yacc

- **Suporte de Precedence:** Visto que a declaração deste parâmetro já é relativamente simples, decidimos replicar parte da sintaxe original.

  Formato Ex: %precedence = (("left","+"),("left"," - "))

- **Declaração de variáveis:** Para suportar a criação e utilização de váriaveis dentro do *parser*, decidimos implementar esta funcionalidade, no entanto, apenas são permitidos certos valores de variáveis como: valores númericos; *strings*, listas vazias ou dicionários vazios.

  Formato Ex: i = 0

  hello = "Hello World"

  list = [ ]

- **Declaração de produções:** Sendo este o parâmetro principal deste módulo, tentamos simplificar a sua escrita, permitindo manter algumas das suas funcionalidades. Este é descrito pelo nome da sua produção, seguido do caracter ':' e seguido, por último, por uma *string* que irá conter a regra dessa produção. Para produções com várias regras, poderá ser repetido o nome da produção.

  Também decidimos implementar duas *features*, nomeadamente, a capacidade de introdução de código Python e a associação de um *alias*, ou seja, ser capaz de atribuir um nome extra a uma produção, de forma a que declarações da mesma produção possam ser distinguidas. Já o código Python tem de ser escrito dentro de chavetas e pode conter ou paragráfos ou o caracter ";"de forma a separar as várias linhas de código.

  Formato Ex: prog : "parametros"

  parametros (vazio) :

  parametros (rec) : "parametros parametro" {p[0] = p[1] + [p[2]]}

3 Gramática

O segundo passo do desenvolvimento deste trabalho prático passou pela definição da gramática. Esta foi definida tendo em conta as características de leitura *Bottom-Up* do *parser* do módulo YACC, como por exemplo, a recursividade à esquerda.

Assim, obtivemos uma gramática independente de contexto uma vez que do lado direito das produções possuímos mais do que um símbolo terminal e/ou não terminal.

Apresentamos, então, a gramática completa tendo subdividindo a apresentação da mesma nas suas secções principais: Inicialização, LEX e, por fim, YACC. (Este conteúdo está presente no ficheiro *Gramatica.txt*)

```
------------------------ INICIALIZAÇÃO ------------------------

prog : seccoes

seccoes :
        | seccoes seccao

seccao : LEXSTART lex termino
       | YACCSTART yacc termino

termino : END
        | seccao




-------------------- Parâmetros Partilhados --------------------

comError : COMERROR context '=' comErrorMessage skipOps

comErrorMessage :
                | STRING

skipOps : SKIP
        | NOSKIP
```

```
------------------------------ LEX ------------------------------

lex : lexParametros

lexParametros :
              | lexParametros lexParametro

lexParametro : lexToken
             | lexRegra

lexToken :  ID context '=' ER tokenFunc changeContext

context :
        | '(' ID ')'
        | ID

tokenFunc :
          | FSTR
          | FINT
          | FFLOAT

changeContext :
              | CHANGECONTEXT ID
              | CHANGECONTEXT '(' ID ')'

lexRegra : lexIgnore
         | lexLiterals
         | lexContexts
         | comError

lexIgnore : LEXIGNORE context '=' STRING

lexLiterals : LEXLITERALS '=' STRING

lexContexts : LEXCONTEXT '=' '(' lexContexTuplos ')'

lexContexTuplos : lexContexTuplo
                | lexContexTuplos ',' lexContexTuplo

lexContexTuplo : '(' STRING ',' STRING ')'
```

```
------------------------------ YACC ------------------------------

yacc : yaccParametros

yaccParametros :
               | yaccParametros yaccParametro

yaccParametro : yaccVar
              | yaccRegra
              | yaccProd

yaccRegra : yaccPrecedence
          | comError

yaccPrecedence : YACCPRECEDENCE '=' '(' yaccPreTuplos ')'

yaccPreTuplos : yaccPreTuplo
              | yaccPreTuplos ',' yaccPreTuplo

yaccPreTuplo : '(' STRING ',' STRING yaccPreTuploOP

yaccPreTuploOP : ')'
               | ',' STRING yaccPreTuploOP

yaccVar : ID '=' VarValue

VarValue : STRING
         | NUMVAL
         | EMPTYLIST
         | EMPTYDIC

yaccProd : ID yaccProdAlias ':' yaccProdValue yaccProdCod

yaccProdAlias :
              | '(' ID ')'
              | ID

yaccProdValue :
              | STRING

yaccProdCod :
            | CODIGO
```

4 Lexer

Para o analisador léxico (*lexer.py*), foi-nos necessário criar três contextos para além do contexto INITIAL, de modo a conseguirmos ler de forma correta os vários tokens de cada secção. Para além disto, decidimos que todas as palavras reservadas da linguagem serão *case insensitive*, de forma a facilitar a sua escrita.

Apresentamos, de seguida, os vários contextos desenvolvidos, assim como justificações para a sua criação/utilização e funcionalidades disponíveis nos mesmos.

4.1 Initial

Neste contexto, incluímos todos os *tokens* que serão partilhados entre as várias secções, como ID's (a ser utilizado nos nomes dos *tokens*, variáveis e produções), *strings*, comentários e *tokens* relacionados ao parâmetro de Erro.

Tokens do estado '**INITIAL**'		
Token	Descrição	Expressão Regular
END	Fim de Secção	%%(?i:END)
ID	Identificador genérico	[a-zA-Z_]\w*
COMERROR	Definição de Error	%(?i:error)
SKIP	Opção de SKIP	%(?i:skip)
NOSKIP	Opção de NO SKIP	%(?i:noskip)
STRING	String com " ou '	(f?"[^"]*")|('[^']*')
COMMENT	Comentários	#.*
NEWLINE	Parágrafo Simples	\n
'ignore'	Caracteres a serem ignorados	\t \r

4.2 Outside

Este contexto é do tipo exclusivo e é utilizado para a leitura léxica "fora"das secções SPLY. Este apenas deteta os inícios de secção e ignora tudo o que for lido.

Tokens do estado '**OUTSIDE**'		
Token	Descrição	Expressão Regular
LEXSTART	Início da Secção LEX	%%(?i:LEX)
YACCSTART	Início da Secção YACC	%%(?i:YACC)
NEWLINE	Parágrafo Simples	\n
'ignore'	Caracteres a serem ignorados	=,():[]

4.3 Lex

Este contexto é do tipo inclusivo e contém todos os *tokens* do módulo LEX, incluindo as suas palavras reservadas e o *token*s de captura de Expressões Regulares.

Tokens do estado '**LEX**'		
Token	Descrição	Expressão Regular
YACCSTART	Início da Secção YACC	%%(?i:YACC)
ER	Expressão Regular	'.+'
FSTR	Função de Conversão para STR	%(?i:str)
FINT	Função de Conversão para INT	%(?i:int)
FFLOAT	Função de Conversão para FLOAT	%(?i:float)
LEXIGNORE	Função de Ignore	%(?i:ignore)
LEXLITERALS	Definição de Literals	%(?i:literals)
LEXCONTEXT	Definição de Contextos	%(?i:contexts)
CHANGECONTEXT	Alteração de Contexto	%(?i:begin)

4.4 Yacc

Este contexto é do tipo inclusivo e contém todos os *tokens* do módulo YACC. Este inclui principalmente os valores permitidos para as suas varáveis.

Tokens do estado '**YACC**'		
Token	Descrição	Expressão Regular
LEXSTART	Início Secção LEX	%%(?i:LEX)
NUMVAL	Valor Númerico	-?\d+(.\d+)?
EMPTYLIST	Lista Vazia	[*]
EMPTYDIC	Dicionário Vazio	%((?i:newdict)|(?i:dict)|(?i:dic)|(?i:newdic))
CODIGO	Bloco de código	{(.|\n)*?}
YACCPRECEDENCE	Precedências	%(?i:precedence)

5 Parser

Para o *parser* (*parser.py*), começamos por definir todas as regras de produção já desenvolvidas no capítulo "Grámatica". Depois, para cada produção, acrescentamos código para a leitura dos vários parâmetros, gravando estes em variáveis internas do *parser*, assim como uma deteção de erros inicial.

5.1 Váriáveis de Armazenamento

Para as variáveis internas do *parser*, decidimos utilizar 3 dicionários distintos, um para cada módulo (LEX e YACC) e um dicionário de uso mais geral, para variáveis do compilador.

- **mylex:** Este dicionário é responsável por armazenar, para cada contexto, os vários parâmetros lidos, ou seja as suas chaves correspondem a nomes de contextos e o seu valor será, também, um dicionário.

  Em cada um destes dicionários internos, estão contidos os vários parâmetros desse contexto, assim como *flags* para evitar repetição de parâmetros, e a sua lista de *tokens*, que será constituida por uma lista de dicionários que irão conter os valores de cada *token*.

- **myyacc:** Este dicionário contém todos os dados e valores correspondentes ao módulo YACC, ou seja, o seu valor de precedência, a lista de variáveis declaradas, sendo que estas são definidas pelo tuplo (Nome,Valor) e por último, a lista de produções que, à semelhança da lista de *tokens*, é composta por dicionários os quais contêm os vários valores de uma produção.

- **mycontents:** Por último, este dicionário contém informação geral e auxiliar ao compilador como quais secções foram lidas, a lista de todos os *tokens*, produções e *literals* (utilizado para verificações) assim como uma lista de linhas onde se encontra o código SPLY lido.

5.2 Deteção de Erros

Nesta fase de recolha de dados, é possível detetar antecipadamente potenciais erros na escrita semântica do código SPLY, como a repetição de parâmetro, tokens e variáveis. Expomos de seguida em mais detalhe todos os controlos de erro presentes nesta fase, assim como o seu funcioanamento e justificação:

- **Erros Sintáticos**
- **Repetição do Parâmetro Error:** Parâmetros error repetidos para a mesma secção.
- **Repetição de Tokens:** Tokens com o mesmo nome e associados ao mesmo contexto.
- **Repetição do Parâmetro Ignore:** Parâmetros Ignore repetidos para o mesmo contexto.
- **Repetição do Parâmetro Literals:** Parâmetros Literals repetidos.
- **Repetição do Parâmetro Context:** Parâmetros Context repetidos.
- **Tipo de Contexto:** Verificação da tipagem de contexto.
- **Repetição do Parâmetro Precedence:** Parâmetros Precedence repetidos.
- **Tipo de Precedence:** Verificação da tipagem do parâmetro precedence.
- **Repetição de Variáveis:** Repetição de variáveis com o mesmo nome.
- **Repetição de Produções:** Repetição de produções com o mesmo nome e alias.

6 Verificador e Escritor

6.1 Verificador

Depois de todo o ficheiro SPLY ser *parsed* com sucesso, ou seja, não existirem problemas de escrita ou semântica inicial, o conteúdo do *parser* passa por uma fase extra de verificação e ajuste. Esta fase ocorre dentro da função *verifyData* presente no ficheiro *auxiliary.py* e nesta função são verificados os dados "ao todo", procurando por possíveis erros. Estes podem ser os seguintes:

- **Contextos declarados apenas implicitamente:** Como permitimos a associação direta de contextos em vários parâmetros assim como todas as "ordens"de declaração, o *parser* inicialmente aceita todas as declarações implícitas efetuadas, no entanto, como é necessário saber a tipagem destes contextos (e não achamos boa solucão utilizar um valor desta por defeito), é necessário que todos os contextos implícitos estejam na declaração **%contexts**.

- **Conteúdos das produções com elementos desconhecidos:** Utilizando a lista de *tokens*, *literals* e nome de produções, é possível verificar, para cada regra de produção, se os vários elementos nesta correspondem a algum dos símbolos terminais ou não terminais já declarados. (**Nota:** Caso não seja encontrada qualquer secção LEX no ficheiro lido, esta verificação é ignorada).

  Nesta fase ocorre, também, um ajuste dos caracteres literais contidos nas regras das produções, pois, como estes podem ser escritos com ou sem plicas, é necessário garantir que estas apareçam nos dados finais.

Este verificador também declara possíveis *Warnings* (a ser mais explorado na secção *Aplicação Final*), ou seja, escritas que, apesar de não estarem erradas semanticamente, podem levar a um resultado não desejado pelo utilizador. Estes normalmente ocorrem devido à falta de declarações de parâmetros dentro de contextos exclusivos, sendo que os valores por defeito são escolhidos, avisando o utilizador. Alguns exemplos como:

- Regra Ignore em falta num contexto exclusivo, sendo utilizado o *default*: " "
- Regra Error em falta numa secção, sendo utilizado o *default*: Mensagem de erro: "Erro Léxico"ou "Erro Sintáxico"com o comando "skip".
- Secção LEX em falta

6.2 Escritor

Esta componente está presente no ficheiro *strFormatter.py* e contém todas as funções de tradução dos dados do *parser* nas várias funções necessárias ao funcionamento do modulo PLY, escrevendo apenas os parâmetros os quais existem informação. Esta apenas devolve em formato String o código necessário, tendo este que ser escrito num ficheiro pela entidade invocadora.

7 Aplicação Final

O objetivo desta fase será criar o código da aplicação final. Esta componente será responsável por unir e encadear as várias fases de leitura, análise léxica e sintática, processamento, verificação e escrita do ficheiro final, assim como a leitura de argumentos invocados pelo utilizador e o tratamento de erros. Esta componente está presente no ficheiro *simPLY.py* e recorre as funcionalidades de todos os outros ficheiros presentes no projeto.

7.1 Argumentos disponíveis

Para este projeto, decidimos disponibilizar uma lista de argumentos o qual o utilizador poderá invocar de forma a aceder a funcionalidades extra do compilador. Estes são os seguintes:

- **-input:** Inserção do nome de ficheiro de entrada a ser lido e processado. Este não é de invocação obrigatória, sendo que caso algum argumento (sem hífen) que não contenha nenhuma flag associada seja encontrado, este é interpretado como o ficheiro de input.
- **-output:** Inserção do nome de ficheiro de saída. Este não pode ser igual ao ficheiro de entrada e, caso não seja explícito, será utilizado o nome **[INPUT]-simPLY.py** por defeito.
- **-help:** Menu de ajuda ao utilizador, indicando o modo de utilização da ferramenta, assim como os vários argumentos permitidos.
- **-debug:** Após a leitura e verificação do ficheiro de entrada, será gerado um ficheiro *debug.JSON* que contém todos os conteúdos lidos pelo *parser* e, caso ocorra algum erro de execução, será imprimido na consola o *traceback* do erro.
- **-wall:** Caso esta flag seja emitida, o programa irá avisar o utilizador de potenciais problemas no ficheiro final a ser escrito, como a omissão de parâmetros requeridos onde o valor por defeito foi utilizado ou a impossibilidade de verificações das regras de produções do analisador sintático.
- **-verbose:** Escrita na consola dos vários processos que estão a ocorrer ao longo da execução do compilador de forma a informar o utilizador.
- **-plyonly:** Por defeito, o ficheiro de saída final contém todo o conteúdo externo do ficheiro de entrada, ou seja, todo o conteúdo não SPLY escrito no ficheiro de entrada é preservado, sendo o resultado da compilação escrito na sua totalidade no local onde a primeira secção SPLY é iniciada. Caso o utilizador só pretenda obter o resultado da compilação SPLY (secções LEX e/ou YACC), poderá utilizar esta *flag*.

8 Exemplos de Utilização

Nesta secção vamos apresentar vários exemplos de ficheiros *input* do programa, assim como os respetivos ficheiros *output* e eventuais mensagens na linha de comandos. Estes exemplos vão se dividir em duas categorias: ficheiros cuja compilação tenha como resultado erro no programa e ficheiros que exemplificam o bom funcionamento do programa.

8.1 Módulo LEX

Neste primeiro exemplo, decidimos intencionalmente referenciar um contexto chamado 'inverno', sem incluir este na declaração de contextos, de forma a testar parte do verificador, pois apesar de o ficheiro apresentado estar sintáticamente correto, a compilação deste não é permitida.

```
%%LEX

INT = r'\d+' %int
VERAO = r'(?i:verao)' %begin verao
WORD = r'[a-zA-Z]+'

YAY verao = r'(?i:yay)' %begin INITIAL

%ignore = " \n\t\r"
%error = "Erro léxico" %skip

%contexts = (("verao","exclusive"))
%ignore inverno = " "

%%END

teste = "20 ola boas 50 verao yay 20"

lexer.input(teste)
for tok in lexer:
    print(tok)
```

Resultados Consola:

```
ERROR: Contexto 'inverno' declarado implicitamente mas sem declaração explícita.
```

Depois de corrigido o erro, voltamos a compilar este utilizando as *flags* -verbose"e -wall", obtendo uma compilação bem-sucedida.

Resultados Consola:

```
Iniciando a leitura do ficheiro Exemplos/v2.sply
Ficheiro lido com sucesso
WARNING: Parâmetro Error em falta (Lex) no contexto exclusivo 'verao'. Valor por
defeito utilizado
Verificado!
Escrevendo no ficheiro output: Exemplos/v2-simPLY.py
Compilação terminada com Sucesso! (Output: Exemplos/v2-simPLY.py)
```

Python Gerado:

```
#------------------- Inicio da Compilacao --------------------
#--------------------LEX------------------

import ply.lex as lex

tokens = ['INT', 'VERAO', 'WORD', 'YAY']
states = (('verao', 'exclusive'),)

#----- CONTEXT: INITIAL -----

def t_INT(t):
  r'\d+'
  t.value = int(t.value)
  return t

def t_VERAO(t):
  r'(?i:verao)'
  t.lexer.begin("verao")
  return t

def t_WORD(t):
  r'[a-zA-Z]+'
  return t

t_ignore = " \n\t\r"

def t_error(t):
  print("Erro lexico")
  t.lexer.skip(1)


#----- CONTEXT: verao -----

def t_verao_YAY(t):
  r'(?i:yay)'
  t.lexer.begin("INITIAL")
  return t
```

```
t_verao_ignore = " "

def t_verao_error(t):
  print("Erro lexico")
  t.lexer.skip(1)


lexer = lex.lex()

#------------------LEX END------------------
#----------------- Compilacao Terminada :) ------------------

teste = "20 ola boas 50 verao yay 20"
lexer.input(teste)
for tok in lexer:
    print(tok)
```

8.2 Módulo YACC

Para o próximo exemplo, expandimos o ficheiro anterior de modo a incluir um *parser* do módulo YACC. Também decidimos gerar um erro, através da modificação de um termo dentro da segunda produção do símbolo não terminal "termos", utilizando um termo desconhecido ao compilador, ou seja, um termo não referenciado quer nos *tokens*, *literals* ou outras produções.

```
%%LEX

#------------Tokens--------------
INT = r'\d+' %int
VERAO = r'(?i:verao)' %begin verao
WORD = r'[a-zA-Z]+'

YAY verao = r'(?i:yay)' %begin INITIAL

%ignore = " \n\t\r"
%error = "Erro léxico" %skip

%literals = "->"
%contexts = (("verao",'exclusive'))
%ignore verao = " "
```

```
#---------------------YACC----------------------
%%YACC

#-------------VARS------------
somatorio = 0
wordlist = []

#-----------PROD-------------
prog : "termos"

termos (vazio) :
termos (rec) : "termos teeermozzz"

termo (num) : "INT"
{p.parser.somatorio += p[1]}
termo word : "WORD" {p.parser.wordlist.append(p[1])}
termo (verao) : "VERAO '-' > YAY" {
    print("YAY")
}

%error = %noskip

%%END



teste = "20 ola boas 50 verao yay 20"

lexer.input(teste)
for tok in lexer:
    print(tok)

print("\n\n\nPARSE\n")

parser.parse(teste)
print("somatorio: "+str(parser.somatorio))
print("lista: "+str(parser.wordlist))
```

Resultados Consola:

```
ERROR: Termo desconhecido 'teeermozzz' encontrado na
produção -> termos (rec) : termos teeermozzz
```

Depois do erro emendado, voltamos a correr o compilador com a utilização das flags -verbose", -wall"e -debug", obtendo uma compilação bem-sucedida.

Resultados Consola:

```
Iniciando a leitura do ficheiro Exemplos/v3.sply
Ficheiro lido com sucesso
WARNING: Parâmetro Error em falta (Lex) no contexto exclusivo 'verao'. Valor por
defeito utilizado
Verificado!
DEBUG: A descarregar os conteúdos do parser em 'debug.JSON'
Escrevendo no ficheiro output: Exemplos/v3-simPLY.py
Compilação terminada com Sucesso! (Output: Exemplos/v3-simPLY.py)
```

Python Gerado:

```python
#------------------- Inicio da Compilacao --------------------
#--------------------LEX-----------------
import ply.lex as lex

tokens = ['INT', 'VERAO', 'WORD', 'YAY']
literals = "->"
states = (('verao', 'exclusive'),)

#----- CONTEXT: INITIAL -----
def t_INT(t):
  r'\d+'
  t.value = int(t.value)
  return t

def t_VERAO(t):
  r'(?i:verao)'
  t.lexer.begin("verao")
  return t

def t_WORD(t):
  r'[a-zA-Z]+'
  return t

t_ignore = " \n\t\r"

def t_error(t):
  print("Erro lexico")
  t.lexer.skip(1)


#----- CONTEXT: verao -----
def t_verao_YAY(t):
  r'(?i:yay)'
  t.lexer.begin("INITIAL")
  return t
```

```
t_verao_ignore = " "

def t_verao_error(t):
  print("Erro lexico")
  t.lexer.skip(1)

lexer = lex.lex()
#------------------LEX END------------------
#--------------------YACC------------------
import ply.yacc as yacc

def p_prog_p1(p):
  "prog : termos"

def p_termos_vazio(p):
  "termos : "

def p_termos_rec(p):
  "termos : termos termo"

def p_termo_num(p):
  "termo : INT"
  p.parser.somatorio += p[1]

def p_termo_word(p):
  "termo : WORD"
  p.parser.wordlist.append(p[1])

def p_termo_verao(p):
  "termo : VERAO '-' '>' YAY"
  print("YAY")

def p_error(p):
  print("Erro Sintatico")
  exit()

parser = yacc.yacc()
parser.somatorio = 0
parser.wordlist = []
#------------------YACC END------------------
#----------------- Compilacao Terminada :) ------------------

teste = "20 ola boas 50 verao yay 20"
lexer.input(teste)
for tok in lexer:
    print(tok)

print("\n\n\nPARSE\n")
parser.parse(teste)
print("somatorio: "+str(parser.somatorio))
print("lista: "+str(parser.wordlist))
```

9 Conclusão e Apreciação Crítica

9.1 Apreciação Crítica

O grupo de trabalho encontra-se bastante satisfeito com a aplicação final desenvolvida, tendo cumprido com sucesso os requerimentos do enunciado proposto, assim como expandir as capacidades do compilador para além do necessário, através de deteção de erros mais granulares e inclusão de *features* de facilidade de uso na escrita de código PLY.

No entanto, reconhecemos que há possíveis melhorias a serem feitas em algumas partes do projeto, como por exemplo: um melhor tratamento dos *tokens* recolhidos pelo *lexer*, pois apesar destes serem tratatos ao nível do *parser*, isto trouxe muita repetição de código em várias produções; a capacidade de analisar e detetar erros nos blocos de código Python escritos pelo utilizador, podendo ser resolvido através da integração de um segundo analisador léxico e sintático e, por último, uma definição mais simples da declaração de um dicionário vazio aquando da criação de uma variável dentro do módulo YACC.

9.2 Conclusão Final

Com a conclusão deste trabalho, o grupo sente-se mais confiante na resolução de futuros desafios e problemas, na área de Processamento de Linguagens. Também pudemos aplicar neste projeto de forma mais profunda, dando continuação ao trabalho prático anterior, os vários conhecimentos teóricos obtidos ao longo do semestre, desde a utilização de expressões regulares, a analisadores léxicos, à formulação e criação de gramáticas independentes de contexto e à aplicação de conhecimentos sobre tratamento e organização de dados.

Para além disto, com a modulação e desenvolvimento deste compilador, o grupo desenvolveu também uma apreciação pelo esforço necessário envolvido no planeamento e criação dos vários compiladores que usamos no dia a dia, assim como as várias ferramentas que necessitam de análises léxicas, sintáticas e semânticas de ficheiros introduzidos pelos utilizadores.