



# Universidade do Minho

Licenciatura em Engenharia Informática

## Processamento de Linguagens

TP2: Tradutor *PLY-Simple* para PLY

### Grupo 48

Ana Murta (A93284)  
Ana Henriques (A93268)  
Rui Coelho (A58898)

maio, 2022

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Análise e especificação do problema</b>	<b>4</b>
2.1	Descrição e objetivos . . . . .	4
2.2	Formato <i>PLY-Simple</i> . . . . .	4
<b>3</b>	<b>Proposta de solução</b>	<b>7</b>
3.1	Ficheiro de input . . . . .	7
3.2	LEX . . . . .	8
3.3	YACC . . . . .	9
<b>4</b>	<b>Testes de tradução</b>	<b>11</b>
4.1	Ficheiros de teste . . . . .	11
<b>5</b>	<b>Conclusão</b>	<b>20</b>

# Listas de Figuras

2.1	Sintax base do formato <i>PLY-Simple</i> .	5
4.1	Ficheiro de teste: <i>example1.txt</i> .	11
4.2	<i>Output</i> da tradução do ficheiro <i>example1.txt</i> .	12
4.3	Ficheiro de teste: <i>example2.txt</i> .	12
4.4	<i>Output</i> da tradução do ficheiro <i>example2.txt</i> .	13
4.5	Ficheiro de teste: <i>example3.txt</i> .	13
4.6	<i>Output</i> da tradução do ficheiro <i>example3.txt</i> .	14
4.7	Ficheiro de teste: <i>example4.txt</i> .	14
4.8	<i>Output</i> da tradução do ficheiro <i>example4.txt</i> .	15
4.9	Ficheiro de teste: <i>example5.txt</i> .	15
4.10	<i>Output</i> da tradução do ficheiro <i>example5.txt</i> .	16
4.11	Ficheiro de teste: <i>example6.txt</i> .	16
4.12	<i>Output</i> da tradução do ficheiro <i>example6.txt</i> .	17
4.13	Ficheiro de teste: <i>example7.txt</i> .	17
4.14	<i>Output</i> da tradução do ficheiro <i>example7.txt</i> .	18
4.15	Ficheiro de teste: <i>example8.txt</i> .	18
4.16	<i>Output</i> da tradução do ficheiro <i>example8.txt</i> .	19
4.17	Ficheiro de teste: <i>example9.txt</i> .	19
4.18	Resultado da execução do ficheiro <i>example9_yacc.py</i> .	19

# Capítulo 1

## Introdução

O presente projeto centra-se no desenvolvimento de uma aplicação que permita efetuar a tradução de um ficheiro que escrito de acordo com o formato *PLY-Simple* para o formato PLY. Deste modo, o relatório aqui apresentado pretende dar conhecer o trabalho desenvolvido, assim como explicitar as decisões e estratégias adotadas pelo grupo aquando da criação da referida aplicação.

Em suma, a aplicação desenvolvida permite geração automática de *parsers*, através da leitura e análise de um ficheiro *PLY-Simple*, onde é integrada toda a informação relevante e necessária para este compilador efetuar a conversão de *PLY-Simple* para PLY.

## Capítulo 2

# Análise e especificação do problema

### 2.1 Descrição e objetivos

O compilador a desenvolver, tal como foi anteriormente mencionado, deve permitir a tradução de um ficheiro com sintaxe *PLY-Simple* em PLY. Um ficheiro escrito de acordo com uma estrutura *PLY-Simple* apresenta uma sintaxe mais simples e "limpa", comparativamente com um gerador de *parsers* escrito em PLY.

Embora o conteúdo necessário para desenvolver um ficheiro *PLY-Simple* seja, no ponto de vista sintático, menos complexo, é necessário que toda a informação nele presente seja processada e tratada de forma adequada.

Posto isto, o trabalho centra-se na criação de um compilador que permita a geração automática de *parsers* PLY, partindo de um ficheiro escrito em formato *PLY-Simple*.

### 2.2 Formato *PLY-Simple*

A Figura 2.1 apresenta a sintaxe *PLY-Simple* fornecida, que serviu como base para o desenvolvimento da aplicação e, posteriormente, para a criação de ficheiros de teste do programa desenvolvido. A syntax final usada surgiu com base na apresentada na figura, tendo sido efetuadas alterações com vista a acomodar as funcionalidades a implementar.

```

%% LEX
%literals = "+-/*=()"          ## a single char
%ignore   = " \t\n"
%tokens   = [ 'VAR', 'NUMBER' ]

[a-zA-Z_][a-zA-Z0-9_]*  return('VAR', t.value)
\d+(\.\d+)?           return('NUMBER', float(t.value))
.
error(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]",
      t.lexer.skip(1) )

%% YACC

%precedence = [
    ('left','+', '-'),
    ('left','*', '/'),
    ('right','UMINUS'),
]

# symboltable : dictionary of variables
ts = { }

stat : VAR '==' exp           { ts[t[1]] = t[3] }
stat : exp                   { print(t[1]) }
exp  : exp '+' exp          { t[0] = t[1] + t[3] }
exp  : exp '-' exp          { t[0] = t[1] - t[3] }
exp  : exp '*' exp          { t[0] = t[1] * t[3] }
exp  : exp '/' exp          { t[0] = t[1] / t[3] }
exp  : '-' exp %prec UMINUS { t[0] = -t[2] }
exp  : '(' exp ')'          { t[0] = t[2] }
exp  : NUMBER                { t[0] = t[1] }
exp  : VAR                   { t[0] = getval(t[1]) }

%%

def p_error(t):
    print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")

def getval(n):
    if n not in ts: print(f"Undefined name '{n}'")
    return ts.get(n,0)

y=yacc()
y.parse("3+4*7")

```

Figura 2.1: Sintaxe base do formato *PLY-Simple*.

Em suma, um ficheiro escrito em formato *PLY-Simple* possui duas seções distintas, uma dedicada ao LEX e outra ao YACC. Cada uma destas componentes apresenta as informações relevantes para a geração automática do *parser* – como, por exemplo, a definição dos *tokens*, no LEX, e o conjunto de produções da gramática, no YACC.

No ficheiro apresentado, a informação referente ao LEX precede os conteúdos do YACC. Contudo, a ordem de apresentação dos conteúdos no ficheiro *PLY-Simple* é irrelevante para o seu processamento. A aplicação desenvolvida é capaz de efetuar traduções da componente LEX ou YACC quando apenas uma delas existe no ficheiro *PLY-Simple*, pelo que foi decidido que a ordem de apresentação dos conteúdos não deve influenciar o processamento da informação presente no ficheiro. Adicionalmente, e com vista a produzir um *parser* PLY válido, tal como foi referido anteriormente, a sintaxe apresentada na Figura 2.1 foi alterada, tendo sendo estipulado o seguinte conjunto de regras sintáticas para a redação correta de ficheiros em formato *PLY-Simple*:

- LEX:

- O início da componente referente ao LEX é marcado através de `%% LEX` ou `%%LEX`;
- A informação referente ao LEX termina quando for reconhecido o marcador de início do YACC ou, então, até ao final do ficheiro se não estiver definido uma componente para o YACC;
- A definição de *tokens*, *ignore*, *literals* ou outra variável deve iniciar-se com um `%` e sem espaço no meio, sendo reservadas as palavras *tokens* e *ignore*, respetivamente, para o seu reconhecimento;
  - \* Caso uma variável não se inicie com um `%` é lançado um erro para informar o utilizador;
- A informação referente aos *literals* pode encontrar-se omissa devido ao seu caráter opcional;

- A informação referente aos *tokens*, *ignore* e *error* é de carácter obrigatório, sendo lançado um erro caso algum destes componentes esteja em falta;
- A ordem de definição dos *tokens* é irrelevante;
- *Tokens* definidos apenas pela sua expressão regular:
  - \* Os *tokens* são definidos pela sua *regex* seguida de *simpleToken('TOKEN', '')* – onde TOKEN refere o nome do *token* a ser definido;
- *Tokens* definidos sobre a forma de funções:
  - \* A definição da função requer *regex* seguida do *return* pretendido;
  - \* O *return* apresenta sempre dois campos: o primeiro refere o respetivo *token* e o segundo o conteúdo da função para o *token*;
  - \* Cada campo do *return* deve ser contido entre placas, i.e., ‘’;
  - \* Na eventualidade da função não efetuar operações (apenas reconhecimento do *token*), o segundo argumento é ‘’, i.e., vazio;
- A definição da função de erro é determinada pela linha que contém uma expressão regular seguida da palavra reservada *error*;
- Os caractéres */%* precedem a definição de variáveis do *lexer*.

- YACC:

- A componente referente ao YACC é iniciada aquando do reconhecimento de *%% YACC* ou *%%YACC*;
- A definição de uma gramática é de carácter obrigatório, sendo informado o utilizador com uma mensagem de erro na eventualidade de esta informação se encontrar omisa;
- A informação da gramática inicia-se com a leitura de uma linha com o marcador */%* e termina quando for reconhecida uma linha com *%%*;
- É necessário usar um *%* para definir a variável *precedence*, de carácter opcional, que descreve a prioridade dos *tokens* definidos no LEX;
- A definição de uma função segue o formato *defFUNCAO(ARGS):*, onde FUNCAO refere o nome da função a definir e ARGS os seus argumentos;
  - \* A enumeração das suas instruções de uma função ocorrem nas linhas posteriores, sendo indicadas pela indentação, i.e., é inserido um caractere *tab*;
- Informação adicional:
  - \* Todas as instruções adicionais associadas com a execução do *parser* ou possíveis testes são precedidas pelo marcador */%*.

# Capítulo 3

## Proposta de solução

O presente capítulo pretende dar a conhecer a proposta de solução desenvolvida pelo grupo de trabalho. Com vista a seguir um modo de desenvolvimento de *software* modular, a solução do problema em mãos passou por dividir o trabalho a ser executado em módulos. Assim sendo, o ficheiro *main.py* controla o fluxo de execução do programa, recorrendo aos módulos *helper.py*, *processLEX.py* e *processYACC.py* para executar as diversas operações necessárias. A execução do programa, através da linha de comandos, requer como argumento a lista de ficheiros a serem traduzidos: podendo ser uma lista singular, com apenas um ficheiro, ou com vários ficheiros, separados por um espaço.

Em suma, o processo de tradução de *PLY-Simple* para PLY decorre segundo três etapas distintas: (i) leitura do ficheiro de *input*; (ii) processamento e escrita do LEX; (iii) processamento e escrita do YACC.

### 3.1 Ficheiro de input

A determinação do ficheiro de *input* a ser traduzido é efetuada aquando da inicialização do programa. O utilizador recebe uma mensagem na qual deve indicar o nome, extensão incluída, do ficheiro a ser traduzido. Uma vez aberto, com a função *open\_file*, o ficheiro é lido e o seu conteúdo é retornado sob a forma de uma lista de *strings*, onde cada *string* corresponde a uma linha do ficheiro. Adicionalmente, é retornado o nome do ficheiro inserido no terminal, para que posteriormente sejam guardadas as componentes LEX e YACC com o nome do ficheiro original. Caso o processo de abertura e leitura ocorra com sucesso, é, então, iniciado o processamento do seu conteúdo.

O processamento inicial do ficheiro fornecido centra-se na divisão dos conteúdos do ficheiro entre as componentes de LEX e YACC, sendo efetuado com recurso à função *get\_lex\_yacc*, que retorna duas listas com os conteúdos lidos e pré-processados – uma lista referente ao LEX e outra ao YACC.

Assim sendo, esta função começa por iterar sobre as strings da lista gerada pela função *open\_file*, identificando a posição em que se iniciam as componentes LEX e YACC. Adicionalmente, para efeitos de controlo da validade do ficheiro fornecido como *input*, é controlada a existência destas duas componentes. Foi estipulado que um ficheiro pode conter apenas uma das componentes de forma isolada, i.e., um ficheiro é, inicialmente, válido mesmo que contenha apenas a componente referente ao LEX ou ao YACC. Na eventualidade de tal se verificar, é emitida uma mensagem de aviso ao utilizador, de modo a informar que o ficheiro em processamento não incorpora uma das componentes, identificando qual a componente em falta.

Uma vez determinadas as posições e a existência das componentes, é efetuada uma limpeza ao conteúdo das listas, sendo removidas linhas vazias. O excerto de código abaixo apresentado consiste numa porção da função descrita, com foco na seleção das linhas relevantes para LEX e YACC.

```
def get_lex_yacc(lines: List[str]):  
    (...)  
    for line in lines:  
        if re.search(r'%% *LEX', line):  
            pos_lex = pos  
            lex_exists = True  
        if re.search(r'%% *YACC', line):  
            pos_yacc = pos  
            yacc_exists = True  
    pos = pos + 1
```

```

if pos_lex > pos_yacc and yacc_exists and lex_exists:
    # for yacc
    for line in lines[pos_yacc+1:pos_lex]:
        if not re.search(r'^$',line):
            lines_for_YACC.append(line)
    # for lex
    for line in lines[pos_lex+1:]:
        if not re.search(r'^$',line):
            lines_for_LEX.append(line)

elif lex_exists and yacc_exists:
    # for lex
    for line in lines[pos_lex+1:pos_yacc]:
        if not re.search(r'^$',line):
            lines_for_LEX.append(line)
    # for yacc
    for line in lines[pos_yacc+1:]:
        if not re.search(r'^$',line):
            lines_for_YACC.append(line)

elif lex_exists and not yacc_exists:
    # for lex
    for line in lines[pos_lex+1:]:
        if not re.search(r'^$',line):
            lines_for_LEX.append(line)

elif not lex_exists and yacc_exists:
    # for yacc
    for line in lines[pos_yacc+1:]:
        if not re.search(r'^$',line):
            lines_for_YACC.append(line)
(...)
```

## 3.2 LEX

Uma vez determinado o conteúdo referente à componente LEX, é usada a função *translate\_lex* para gerar a *string* correspondente à tradução para LEX do *input* que é recebida como parâmetro. O processamento da informação inicia-se com a determinação da lista de expressões regulares e na determinação da lista de *tokens*. A procura da linha de interesse, para os *tokens*, é efetuada com recurso à regex `r'%tokens'`, sendo usada a regex `r'(?:(\s?)(.*)(?:\s?))'` para capturar os *tokens* presentes nessa linha.

A determinação das restantes componentes é efetuada de forma similar: é determinada uma expressão regular para encontrar a linha de interessse e, uma vez encontrada, são efetuadas as operações necessárias para o seu processamento inicial. O excerto de código abaixo apresentado mostra o processo iterativo que permite determinar o conteúdo presente no ficheiro, com particular foco nas componentes referentes ao *ignore* e *error*.

```

def translate_lex(lines_for_LEX: List[str]):
    (...)

    for line in lines_for_LEX:
        if re.match(r'%ignore',line):
            ignore_match = line
            res_ignore = "t_ignore" + ignore_match[ignore_match.index("ignore")
                + len("ignore"):] + "\n"
            run_ignore = True

        elif re.search(r'.*?error',line):
            error_match = line
            error_message = (re.findall(r'(:f\*)(.*)(?:\\,)',error_match))[0]
            run_error = True

    (...)
```

Uma vez processado inicialmente, o processo de tradução inicia-se caso as componentes obrigatórias para a definição do LEX se encontrem presentes (*tokens*, *ignore* e *error* – sendo a sua presença controlada por variáveis booleanas que vão sendo atualizadas à medida em que estas componentes são detetadas). Tal como foi mencionado anteriormente, a presença destas componentes é de caráter obrigatório, sendo lançada uma mensagem de erro caso alguma destas se encontre em falta.

O processamento e tradução dos *tokens* é efetuado com recurso à função auxiliar *process.tokens*, cujo código se encontra abaixo apresentado. Esta função, que recebe como argumento um *token* e a lista de expressões regulares anteriormente calculada, processa os diversos *tokens* definidos no ficheiro *PLY-Simple*, preparando-os com o formato desejado, quer estes se encontrem definidos de forma simples (marcados pela palavra reservada *simpleToken*) quer estejam definidos sob a forma de uma função (onde é usada a palavra reservada *return* para a sua identificação).

```
def process_tokens(tok: str, list_regex: List[str]):
    tok_no_func = ""
    tok_func = ""
    tok = re.sub(r' *|\\"', '\"', tok)

    for element in list_regex:
        if re.search(f'\w*(?i:{tok})\\"', element):
            regex = re.findall(r'^(.*)(:simpleToken|return)', element)[0]
            regex = re.sub(r' +$', '', regex)
            regex = re.sub(r'\\{2}', r'\\', regex)

        if re.search('return', element):
            group = (re.findall(r'^(?:(\s*\'.*?\')\s*,\s*\'.*?\')\s*\')', element))[0]
            tok_func += lex_function(group[0], regex, group[1])
        elif re.search('simpleToken', element):
            tok_no_func += f't_{tok}' + " = r'" + regex + "'\n"

    return tok_func, tok_no_func
```

Por fim, são agrupados todos os dados já traduzidos, numa única *string*, para que, mais tarde, possam ser escritos no ficheiro. São adicionados, primeiramente, os conteúdos referentes aos *literals*, seguindo-se da lista de *tokens* e de variáveis. Posteriormente, insere-se a informação relativa ao *ignore*, os *tokens* (sem e com funções associadas). Por fim, adiciona-se a essa *string* o conteúdo da função de erro, assim como os comandos *lexer* – `lexer = lex.lex()` e restantes informações/instruções associadas.

Tendo terminado o processo de tradução dos conteúdos referentes ao LEX, é usada a função *write\_file\_lex* para produzir o ficheiro de *output* referente. Nesta função, são adicionados os *imports* necessários para que o *lexer* funcione devidamente.

### 3.3 YACC

O processo de tradução e escrita da componente YACC é efetuado com recurso às funções *translate\_yacc* e *write\_file\_yacc*, respetivamente.

A tradução das linhas do ficheiro representa um processo iterativo que, ao percorrer a lista de *strings* recebidas como *input*, para efetuar o processamento das várias componentes do YACC, como é o caso da gramática (de caráter obrigatório), da precedência e das instruções referentes ao *parser*. A título de exemplo, e conferindo algum foco à questão da gramática, a localização das linhas de interesse para a gramática é efetuada do seguinte modo: é usada uma *regex* para determinar a linha onde a gramática a ser usada se inicia, sendo armazenadas numa variável todas as linhas até ser detetado o marcador referente ao fim da gramática<sup>1</sup>. O excerto de código apresentado configura o processo descrito. Importa, ainda, referir que não é efetuado, durante todo o processo de tradução, qualquer tipo de validação face ao conteúdo da gramática, i.e., a definição de uma gramática correta é da responsabilidade do utilizador que fornece o ficheiro de *input*, uma vez que a ferramenta desenvolvida centra-se no processo de tradução dos ficheiros *PLY-Simple*.

```
def translate_yacc(lines: List[str]):
    (...)
```

---

<sup>1</sup>Um raciocínio similar é usado, por exemplo, para a determinação das precedências existentes no YACC, caso a variável exista.

```

while pos < len(lines):
    # process grammar
    if re.match(r'^/%$', lines[pos]):
        found_grammar = True
        for subline in lines[pos+1:]:
            if not re.match(r'^%%$', subline):
                grammar.append(subline)
                pos = pos + 1
            else: break
        pos = pos + 1
    (...)
```

O processamento das funções descritas no YACC é realizado com recurso à função auxiliar *process-function*, que permite localizar e tratar devidamente as funções apresentadas no ficheiro *PLY-Simple* – com recurso à regex `r'def '`. Uma vez determinadas as funções, é extraída, caso esteja presente, a função de erro.

A tradução da gramática ocorre, naturalmente, na eventualidade de esta estar presente. Tal como foi anteriormente referido, a gramática é uma componente obrigatória do YACC, pelo que a sua ausência desencadeia uma mensagem de erro. Assim sendo, após a validação da presença de gramática no ficheiro, é utilizada a função *process\_grammar* para executar o seu processo de tradução. Com recurso a esta função é possível trabalhar as produções recolhidas aquando do início do processamento do YACC, recorrendo a uma *regex* para capturar especificamente as linhas de interesse com as regras de produção e respetivas operações (caso existam). O trecho de código abaixo apresentado demonstra as expressões regulares utilizadas no processo, assim como a lógica operacional seguida em cada iteração do processamento das linhas da gramática. No final do ciclo, é retornada uma *string* que contém as diversas regras de produção, já devidamente convertidas.

```

def process_grammar(grammar: List[str]):
    (...)

    for line in grammar:
        if line != "":
            group = (re.findall(r'([^\n]+)(?: *: *(.*?){2,})(?:{ *(.*) *})', line))[0]
            prod = re.sub(r'^(.*)', '', group[1])
            op = re.findall(r'^(.*?[^\nA-Za-z])([A-Za-z])(\[\d+\]+\?)', group[2])
            calc = ""
            for s in op:
                s = (s[0], 'p', s[2])
                calc += "".join(s)
            if any(f'{group[0]} -> {s}' for s in defGrammar):
                defGrammar.append(f"# p{i}:      | {prod}")
            else: defGrammar.append(f"# p{i}: {group[0]} -> {prod}")
            res_grammar += f"""def p_{group[0]}_p{i}(p):
    {group[0]} : {prod}
    {calc}\n\n"""
            i = i+1
    (...)
```

Fundo o processo de tradução, e tal como ocorreu para o processo do LEX anteriormente descrito, foi gerada uma *string* com toda informação traduzida, a fim de poder ser escrita posteriormente em ficheiro. Assim sendo, esta *string* resultante contém, pela seguinte ordem, a gramática (para que seja inserida como comentário no ficheiro de *output*), a informação de precedências (caso exista), os dicionários de variáveis, as definições de funções e regras de produção – seguindo-se a função de erro – e a informação adicional processada acerca do *parser*.

Por fim, e tal como para a secção do LEX, foi criada uma função responsável pela escrita do conteúdo traduzido a partir do *PLY-Simple*. Esta função, *write\_file\_yacc*, adiciona ao conteúdo já processado os devidos *imports* necessários para o funcionamento correto do ficheiro, aquando da sua escrita.

# Capítulo 4

## Testes de tradução

A presente secção procura apresentar o conjunto de testes efetuados para a validação da execução correta do programa desenvolvido. Seguidamente, são apresentados os ficheiros de teste criados, assim como o resultado obtido, e esperado, aquando da sua conversão.

### 4.1 Ficheiros de teste

Apresentado na Figura 4.1, temos o ficheiro *example1.txt*, que foi adaptado da sintaxe base fornecida do formato *PLY-simple*, ilustrado na Figura 2.1. Como se pode verificar, este ficheiro *PLY-simple* obedece à sintaxe estipulada na Secção 2.2, sobressaindo a diferença de que, comparativamente com sintaxe base seguida, a gramática deve estar contida entre os caracteres `/%` e `%%` e as instruções associadas à execução do *parser* devem ser introduzidas pelos caracteres `/%`.

```
input > # example2.txt > ...
1  %% LEX
2  $literals = "+-/==()"
3  $ignore = "\t\n"
4  $tokens = ['VAR', 'NUMBER']
5
6  [A-Za-z]_0-9A-Za-z]* return('VAR', '')
7  \d+(\.\d+)?           return('NUMBER', 't.value = float(t.value)')
8  .
9  |                   | error(f'Illegal character '{t.value[0]}', [{t.lexer.lineno}]),
10 |                   | t.lexer.skip(1)
11 %%
12 %% YACC
13 %precedence =
14   ('left','+', '-'),
15   ('left','*', '/'),
16   ('right','UMINUS'),
17 ]
18
19 # symboltable : dictionary of variables
20 ts = {}
21
22 %%
23
24 stat : VAR `=' exp      { ts[t[1]] = t[3] }
25 stat : exp              { print(t[1]) }
26 exp : exp `+' exp     { t[0] = t[1] + t[3] }
27 exp : exp `-' exp     { t[0] = t[1] - t[3] }
28 exp : exp `*' exp     { t[0] = t[1] * t[3] }
29 exp : exp `/' exp     { t[0] = t[1] / t[3] }
30 exp : `-' exp %prec UMINUS
31 exp : `(` exp `)`
32 exp : NUMBER
33 exp : VAR               { t[0] = getval(t[1]) }
34
35 %%
36
37 def p_error(t):
38     print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
39
40 def getval(n):
41     if n not in ts: print(f"Undefined name '{n}'")
42     return ts.get(n, 0)
43
44 /*yacc()
45 /*y_parse("3+4*7")
```

Figura 4.1: Ficheiro de teste: *example1.txt*.

A conversão deste ficheiro de teste não produziu nenhuma mensagem de erro, nem nenhum *warning*, podendo o resultado obtido ser analisado na Figura 4.2.

```

anaphen@Mas-Air src %
anaphen@Mas-Air src %
anaphen@Mas-Air sri % python3.9 main.py example1.txt
[example1] opened successfully
[example1] parsed successfully
[example1] translated YACC successfully.
anaphen@Mas-Air src %

```

Figura 4.2: *Output* da tradução do ficheiro *example1.txt*.

O próximo ficheiro de teste usado, ilustrado na Figura 4.3, possui uma gramática que descreve a criação de listas de inteiros. Em contraste com o anterior, este ficheiro apresenta a particularidade dos *tokens* serem definidos apenas pela sua expressão regular, seguida de *simpleToken('Token', '')*. Outro pormenor que se destaca é a não existência de operações para nenhuma das produções da gramática, sendo isto simbolizado pelos caracteres {}, que não possuem qualquer conteúdo.

```

input > 例 example1.txt
1  %% LEX
2
3  %tokens = ['PA', 'PF', 'NUM', 'VIRG']
4  %ignore = " \t\n"
5
6  \d+                               simpleToken('NUM','')
7  \,                               simpleToken('VIRG','')
8  \)
9  \(
10 .                                error(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]", t.lexer.skip(1) )
11
12 %%YACC
13
14 /%
15
16 Lista : PA PF                      {}
17 Lista : PA Elems PF                  {}
18 Elems : Elem Resto                 {}
19 Resto :
20 Resto : VIRG Elems                 {}
21 Elemt : NUM                         {}
22 Elemt : Lista                       {}
23
24 %%
25
26 def p_error(p):
27     print("Erro sintático",p)
28
29 %parser = yacc.yacc()

```

Figura 4.3: Ficheiro de teste: *example2.txt*.

A conversão deste ficheiro de teste também não produziu nenhuma mensagem de erro, nem nenhum *warning*, estando o resultado obtido representado na Figura 4.4.

```

example2.lex.py
1  tokens = ['PA', 'PF', 'NUM', 'VIRG']
2
3  t_PA = r'PA'
4  t_PF = r'PF'
5  t_NUM = r'\d+'
6  t_VIRG = r','
7
8  def t_error(t):
9      print(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
10     t.lexer.skip(1)
11
12 lexer = lex.lex()
13
14
15

example2.yacc.py
1  import ply.yacc as yacc
2  from example2_lex import *
3
4  # GRAMMAR
5  # p1: Lista --> PA PF
6  # p2: PA Elems PF
7  # p3: Elems --> Elems Resto
8  # p4: Resto --> VIRG Elems
9  # p5: Elems --> NUM
10 # p6: Elem --> NUM
11 # p7: | Lista
12
13 def p_Lista_p1(p):
14     | "Lista : PA PF"
15
16     | "Lista : PA Elems PF"
17
18     | "Listas : PA Elems PF"
19
20     | "Elems : Elem Resto"
21
22     | "Resto : "
23
24     | "Resto : VIRG Elems"
25
26     | "Resto : "
27
28 def p_Elems_p2(p):
29     | "Elems : NUM"
30
31     | "Elems : Lista"
32
33     | "Elems : "
34
35 def p_Error(p):
36     | "print('Erro sintático', p)"
37
38 parser = yacc.yacc()

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
anaphen@Mas-Air:~/src % python3.9 main.py example2.txt
[example2] opened successfully.
[example2] translated YACC successfully.
[example2] translated YACC successfully.
anaphen@Mas-Air:~/src %

```

Figura 4.4: *Output* da tradução do ficheiro *example2.txt*.

Com o objetivo de demonstrar que o tradutor implementado funciona de modo singular para cada uma das componentes LEX e YACC, i.e., na situação de uma não estar definida, o tradutor continua funcional e converte a componente existente, escrevemos o ficheiro de teste *example3.txt*, no qual apenas está presente o LEX.

```

input > 例 example3.txt > ...
1  %% LEX
2  %tokens = ['INI','FIM','nome','real','int','sinal']
3  %sinal = "[=+-*/()]"'
4  %ignore = " \n\t"
5
6  (?begin)\|{
7  \|}[eE] [nN] [dD]
8  [a-zA-Z] [a-zA-Z0-9]* 
9  [0-9]+. [0-9]+
10 [0-9]+
11 .
12
13 lexer = lex.lex()


```

Figura 4.5: Ficheiro de teste: *example3.txt*.

O resultado obtido pode ser observado na Figura 4.6, onde, no terminal, é lido o seguinte *warning*: [WARNING] nothing defined for YACC in PLY-Simple. De qualquer modo, apesar do YACC não ter sido traduzido, o tradutor continuou funcional e, como tal, foi gerado um ficheiro *example3\_lexer.py*, tal como previsto.

```

example3_lex.py U
output > ℹ example3_lex.py > ...
1 import ply.lex as lex
2
3 tokens = ['INIT','FIM','nome','real','int','sinai']
4 t_ignore = "\n\t"
5
6 def t_INIT(t):
7     r'(?:begin)\n|'
8     return t
9
10 def t_FIM(t):
11     r'\)(?:end)\n|([N] )([D])'
12     return t
13
14 def t_nome(t):
15     r'([a-zA-Z] ([a-zA-Z0-9])*'
16     return t
17
18 def t_real(t):
19     r'([0-9]+.[0-9]+)'
20     return t
21
22 def t_int(t):
23     r'([0-9]+)'
24     return t
25
26 def t_error():
27     print("Illegal character '{t.value[0]}', [{t.lex.token}]")
28     t.lexer.skip(1)
29
30 lexer = lex.lex()
31

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

annaphens@nas-Air src %  
[example3] lexica translated successfully.  
[example3] translated LEX successfully.  
[WARNING] nothing defined for YACC in PLY-Simple.  
annaphens@nas-Air src %

Figura 4.6: *Output* da tradução do ficheiro *example3.txt*.

Na Figura 4.7, temos o ficheiro de teste *example4.txt*, que apresenta uma diferença em relação aos outros ficheiros: as instruções associadas à execução do *parser* são dadas pelo utilizador, i.e. são lidas como *input*, e cada linha de interesse referente a estas instruções são introduzidas com os caracteres */%*.

```

input > ℹ example4.txt
1 %% LEX
2
3 $literals = ['(',')','+','-','*']
4 $tokens = ['num','FIM']
5 $ignore = "\r\n"
6
7 \d+           return('num','t.value=int(t.value)')
8 \+
9 .             simpleToken('FIM','')
10 .             error("Caracter ilegal: , {t.value[0]}",t.lexer.skip(1))
11 lexer = lex.lex()
12
13 %% YACC
14
15 /**
16
17 Z : Exp FIM          {print("Resultado: ", p[1])}
18 Exp : '(' '*' Lista Exp ')'
19     | '(' '*' Lista Exp ')'
20     | num
21 Lista : Lista Exp
22 Lista : Exp
23
24 /**
25
26 def p_error(p):
27     print("Erro sintático: ", p)
28     parser.success = False
29
30 def produtorio(lista):
31     res = 1
32     for elem in lista:
33         res *= elem
34     return res
35
36 def somatorio(lista):
37     res = 0
38     for elem in lista:
39         res += elem
40     return res
41
42 #parser = yacc.yacc()
43
44 #import sys
45 #for linha in sys.stdin:
46 #    parser.success = True
47 #    parser.parse(linha)
48 #    if parser.success:
49 #        print("Frase válida: ", linha)
50 #    else:
51 #        print("Frase inválida.")


```

Figura 4.7: Ficheiro de teste: *example4.txt*.

O resultado obtido com a conversão do ficheiro *example4.txt* está ilustrado na Figura 4.8, o que resultou na criação dos ficheiros *example4.lex.py* e *example4-yacc.py*, sem qualquer mensagem de erro ou *warning*.

```

example4_yacc.py U ...
output > ⌂ example4_yacc.py > ⌂ produtorio
1 import ply.yacc as yacc
2
3 tokens = ['+', '*', '(', ')', 'num', 'EOF']
4 t_ignore = " \t\n"
5 t_FIN = r'\.'
6
7 def t_num(t):
8     r"\d+"
9     t.value=int(t.value)
10    return t
11
12 def t_error(t):
13     print("Carácter ilegal: ", t.value[0])
14     t.lexer.skip(1)
15
16 lexer = lex.lex()
17
18 def p_Produtorio(p):
19     "Produtorio : ('+' '*' Lista_Exp ')'"
20     p[0] = produtorio(p[3]) * p[4]
21
22 def p_Lista_Exp(p):
23     "Lista_Exp : num"
24     p[0] = p[1]
25
26 def p_Error(p):
27     "Z : Exp FIN"
28     print("Resultado: ", p[1])
29
30 def p_Exp_P4(p):
31     "Exp : ('+' '*' Lista_Exp ')'"
32     p[0] = somatorio(p[3]) + p[4]
33
34 def p_Exp_P3(p):
35     "Exp : ('-' '*' Lista_Exp ')'"
36
37 def p_Exp_P2(p):
38     "Exp : num"
39     p[0] = p[1]
40
41 def p_Lista_S0():
42     "Lista : Lista_Exp"
43     p[0] = p[1] + [p[2]]
44
45 def p_Lista_N0(p):
46     "Lista : Exp"
47     p[0] = p[1] + [p[2]]
48
49 def p_Error(p):
50     print("Erro sintático: ", p)
51     parser.success = False
52
53 parser = yacc.yacc()
54 import sys
55 for linha in sys.stdin:
56     parser.parse(linha)
57     if parser.success:
58         print('Frase válida: ', linha)
59     else:
60         print('Frase inválida.')
61
62
63

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

annaphens@nas-Air src % annaphens@nas-Air src % python3.9 main.py example4.txt [example4] Translated LEX successfully. [example4] Translated YACC successfully. annaphens@nas-Air src % [ ]

Figura 4.8: *Output* da tradução do ficheiro *example4.txt*.

O próximo ficheiro de teste – *example5.txt*, ilustrado na Figura 4.9 – não tem a componente LEX definida. Tal como para o ficheiro *example3.txt*, este teste servirá para demonstrar que o tradutor funciona de forma independente para ambas as componentes LEX e YACC.

```

input > ⌂ example5.txt
1 %% YACC
2
3 /%
4
5 S : A a S      {}
6 S : A          {}
7 A : A a F      {}
8 A : F          {}
9 F : F S        {}
10
11 %%
12
13 def p_error(p):
14     print("Erro sintático: ", p)
15     parser.success = False
16
17
18 %parser = yacc.yacc()

```

Figura 4.9: Ficheiro de teste: *example5.txt*.

Na Figura 4.10, temos o *output* conseguido com a conversão do ficheiro *example5.txt*, onde, no terminal, se lê o seguinte *warning*: **[WARNING] nothing defined for LEX in PLY-Simple**. Como a componente LEX não estava especificada no ficheiro *PLY-simple*, o tradutor apenas gerou o ficheiro *example5\_yacc.py*.

```

output > ℹ example5_yacc.py 1, U
1 import ply.yacc as yacc
2 from example5_lex import *
3
4 # GRAMMAR:
5 # p1: S -> A a S
6 # p2:   | A
7 # p3:   A -> A a F
8 # p4:   | F
9 # p5:   F -> F S
10
11 def p_S_p1(p):
12     "S : A a S"
13
14 def p_S_p2(p):
15     "S : A"
16
17 def p_A_p3(p):
18     "A : A a F"
19
20 def p_A_p4(p):
21     "A : F"
22
23 def p_F_p5(p):
24     "F : F S"
25
26 def p_error(p):
27     print("Erro sintático: ", p)
28     parser.success = False
29
30 parser = yacc.yacc()
31

```

PROBLEMS OUTPUT TERMINAL DEBÚGUE CONSOLE

```

annaphens@anas-Air src %
annaphens@anas-Air src %
annaphens@anas-Air src % python3.9 main.py example5.txt
annaphens@anas-Air src % opened successfully.
[WARNIN] grammar not defined for LEX in PLY-Simple.
[examples] Translated YACC successfully.
annaphens@anas-Air src %
annaphens@anas-Air src %

```

Figura 4.10: *Output* da tradução do ficheiro *example5.txt*.

O seguinte ficheiro de teste, *example6.txt*, apresentado na Figura 4.11, não contém a sua gramática escrita entre os caracteres `/%` e `%%`, levando a que o tradutor não consiga reconhecê-la. Isto causará problemas visto que, como mencionado na Secção 3.3, a gramática é uma componente obrigatória do YACC.

```

input > ℹ example6.txt
1 %% LEX
2
3 %tokens = ['Z', 'U']
4 %ignore = "\t\n"
5
6 0 simpleToken('Z','')
7 1 simpleToken('U','')
8 . error(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]", t.lexer.skip(1))
9
10 lexer = lex.lex()
11
12 %%YACC
13
14 S : Z S Z          {}
15 S : U S U          {}
16 S : Z              {}
17 S : U              {}
18
19 def p_error(p):
20     print("Erro sintático: ", p)
21     parser.success = False
22
23 /%parser = yacc.yacc()

```

Figura 4.11: Ficheiro de teste: *example6.txt*.

Como consequência da gramática não ser reconhecida, no terminal, é lida a seguinte mensagem de erro: `[ERROR] grammar not found on YACC`. Isto impossibilitará, então, o tradutor de gerar o ficheiro *example6\_yacc.py*, tal como se pode analisar pela Figura 4.12.

The screenshot shows the VS Code interface with the following details:

- Editor:** The file `example6_lexer.py` is open in the main editor area. The code defines a lexer with tokens for digits, whitespace, and error handling.
- Terminal:** The terminal at the bottom shows the command `python3.9 main.py example6.txt` being run, followed by the output: "[example6] translated LEX successfully. [ERROR] grammar not found on YACC.".
- Bottom Status Bar:** Shows the current file is `main*`, and the status bar indicates Ln 13, Col 1, Spaces: 4, UTF-8, LF, Python 3.10.2 64-bit.

Figura 4.12: *Output* da tradução do ficheiro *example6.txt*.

Para comprovar que o tradutor não enfrenta problemas em processar a componente YACC antes da componente LEX, construiu-se o ficheiro de teste *example7.txt*, presente na Figura 4.13, no qual o YACC é, então, definido antes do LEX.

```
input > E example7.txt > ...
1 %%YACC
2
3 /%
4
5 Calc : Comandos FIM
6 Comandos : Comandos Comando
7 Comandos : Comando
8 Comando : id '=' Exp
9 Comando : '!'? Exp
10 Comando : '?'
11 Comando : DUMP
12 Exp : Exp '+' Termo
13 Exp : Exp '-' Termo
14 Exp : Termo
15 Termo : Termo '*' Fator
16 Termo : Fator
17 Fator : num
18 Fator : '(' Exp ')'
19 Fator : id
20
21 %%
22
23 /*parser = yacc.yacc()
24 /*parser.registos = {}*/
25
26 %% LEX
27
28 %literals = ['+', '-', '*', '/', '(', ')', '=', '?', '!']
29 %tokens = ['num', 'id', 'DUMP', 'FIM']
30 %ignore = " \t\n"
```

Figura 4.13: Ficheiro de teste: *example7.txt*.

Assim como se pode observar pela Figura 4.14, a conversão do ficheiro *example7.txt* gera os dois ficheiros esperados sem lançar nenhuma mensagem de erro ou de *warning*.

```

example7.lex.py U
output > example7.lex.py >_
    import ply.lex as lex
    literals = ['+', '-', '*', '/', '=', '<', '>', '!', '(', ')', '{', '}']
    t_ignore = " \t\n"
    t_id = r'[A-Za-z]+\w*'
    t_NUM = r'\d+'
    t_FIM = r'\n'

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

def t_error():
    print("Illegal character '{t.value[0]}', [{t.lex.lineno}]")
    tlexer.skip(1)

lexer = lex.lex()

```

  

```

example7.yacc.py U
output > example7.yacc.py >_
1   import ply.yacc as yacc
2   from example7.lex import *
3
4   # GRAMMAR
5   # p1: Calculos -> Comandos FIM
6   # p2: Comandos -> Comandos Comando
7   # p3: Comando -> id '=' Exp
8   # p4: Exp -> '+' Termo
9   # p5: Termo -> '-' Factor
10  # p6: Factor -> num
11  # p7: Factor -> '(' Exp ')'
12  # p8: Exp -> '*' Termo
13  # p9: Termo -> '-' Factor
14  # p10: Factor -> num
15  # p11: Termo -> Termo '*' Factor
16  # p12: Factor -> num
17  # p13: Factor -> '(' Exp ')'
18  # p14: Factor -> id
19  # p15: Factor -> ''
20
21  def p_Calc_p1(p):
22      "Calculos : Comandos FIM"
23
24  def p_Comandos_p2(p):
25      "Comandos : Comandos Comando"
26
27  def p_Comandos_p3(p):
28      "Comandos : Comando"
29
30  def p_Commando_p4(p):
31      "Comando : id '=' Exp"
32      p.parser.registros[p[1]] = p[3]
33
34  def p_Commando_p5(p):
35      "Comando : '' Exp"
36      print(p[1])
37
38  def p_Commando_p6(p):
39      "Comando : '?' id"
40      p.parser.registros[p[1]] = p[2]
41
42  def p_Commando_p7(p):
43
44  def p_Calc_p1(p):
45      "Calculos : "
46
47  def p_Ex_p2(p):
48      "Exp : Exp '+' Termo"
49      p[0] = p[1] + p[3]
50
51  def p_Ex_p3(p):
52      "Exp : Exp '*' Termo"
53      p[0] = p[1] * p[3]
54
55  def p_Ex_p4(p):
56      "Termo : Termo '*' Factor"
57      p[0] = p[1] * p[3]
58
59  def p_Termo_p1(p):
60      "Termo : Termo '-' Factor"
61      p[0] = p[1] - p[3]
62
63  def p_Termo_p2(p):
64      "Termo : Factor"
65
66  def p_Fator_p3(p):
67      "Factor : num"
68
69  def p_Fator_p4(p):
70      "Factor : '(' Exp ')'"
71      p[0] = p[2]
72
73  def p_Fator_p5(p):
74      "Factor : id"
75      p[0] = p.parser.registros[p[1]]
76
77  parser = yacc.yacc()
78  parser.registros = {}
79

```

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
anaphens-MacBook-Air:src anaphens
anaphens-MacBook-Air:src % python3.9 main.py example6.txt example7.txt
[example6] opened successfully.
[example7] opened successfully.
[example6] translated YACC successfully.
[example7] opened successfully.
[example7] translated YACC successfully.
[example7] translated YACC successfully.
anaphens@MacBook-Air:src %

```

Figura 4.14: *Output* da tradução do ficheiro *example7.txt*.

O próximo ficheiro de teste que importa introduzir é o *example8.txt*. Como sabemos, o LEX necessita da especificação da linguagem em formato *regex* e de um conjunto de funções adicionais para a manipulação dos *tokens* gerados. No entanto, isso implica que a lista de *tokens* esteja primeiramente definida. Na situação de não estar, uma situação de erro deve acontecer. Posto isto, com o ficheiro *example8.txt*, ilustrado na Figura 4.15, pretende-se averiguar as consequências que esta ausência implicará.

```

input > E example8.txt
1  %% LEX
2
3  #não possui tokens
4  %ignore = " \t\n"
5
6  \(
7      simpleToken('PA','')
8  \)
9      simpleToken('PF','')
10 \d+
11 simpleToken('NUM','')
12 .
13 error(f"Illegal character '{t.value[0]}', [{t.lex.lineno}]",t lexer.skip(1) )
14
15 %%YACC
16
17 /%
18
19
20 def p_error(p):
21     print("Erro sintático",p)
22     parser.success = False
23
24 %parser = yacc.yacc()
25 %parser.abins = {}

```

Figura 4.15: Ficheiro de teste: *example8.txt*.

Assim sendo, na Figura 4.16, verifica-se precisamente o esperado: no terminal, é apresentada a mensagem de erro [ERROR] variables missing or not introduced with caracter '%' on LEX, o que leva a que o ficheiro *example8.lex.py* não seja escrito.

Figura 4.16: *Output* da tradução do ficheiro *example8.txt*.

Finalmente, é apresentado o ficheiro *PLY-simple example9.txt* na Figura 4.17, que não evidencia nenhuma particularidade que não tenha sido previamente mencionada. A sua conversão é efetuada com sucesso, gerando os dois ficheiros esperados, sem lançar qualquer mensagem de erro ou *warning*.

```
input > %% example9.txt
1 %% LEX
2
3 %literals = ['[', ']']
4 %tokens = ['id', 'num']
5 %ignore = " \t\n"
6
7 \d+           return('num','t.value = int(t.value)')
8 \"[\\n\\r]+\"    simpleToken('id','')
9 .
10 error(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]", t.lexer.skip(1) )
11
12 lexer = lex.lex()
13
14 %% YACC
15
16 /*
17 Lista : Lista Ficheiro      {}
18 Lista :
19 Ficheiro : '(' id id ')'
20
21 %%
22
23 def p_error(p):
24     print("Erro sintático: ", p)
25     parser.success = False
26
27 #parser = yacc.yacc()
28 #parser.items = []
```

Figura 4.17: Ficheiro de teste: *example9.txt*.

Todavia, ao testar a funcionalidade do seu *parser*, obteve-se os erros ilustrados na Figura 4.18. Isto serve para reforçar o que havia sido anteriormente mencionado na Secção 3.3, a definição de uma gramática correta é da responsabilidade do utilizador que fornece o ficheiro de *input*. Assim sendo, não se tomou nenhuma ação em relação a estes resultados já que o nosso foco apenas se destinou ao processo de tradução.

```
ply.yacc.YACCError: UNKNOWN conflict in state 1
ruipingcoelho@MBP-de-Rui % python3.9 example9_yacc.py
WARNING: Token 'num' defined, but not used
WARNING: There is 1 unused token
Generating LALR tables
ruipingcoelho@MBP-de-Rui %
```

Figura 4.18: Resultado da execução do ficheiro *example9\_yacc.py*.

# Capítulo 5

## Conclusão

O presente trabalho centrou-se no desenvolvimento de uma aplicação capaz de traduzir ficheiro em formato *PLY-simple* para PLY, gerando de forma automática *parsers* funcionais para o ficheiro de *input*. No decurso da conceitualização e criação do programa previamente apresentado, a equipa conseguiu fortalecer os conteúdos teóricos abordados no decurso do semestre, tendo aprimorado competências no domínio das expressões regulares, no processamento de linguagens, na geração de compiladores e na utilização da linguagem de programação *Python* como ferramenta de desenvolvimento de *software*. Face aos objetivos estabelecidos e ao trabalho apresentado, a equipa considera ter alcançado com sucesso as metas esperadas, criando um programa funcional e simples que preenche os requisitos estipulados.