



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

1	Introdução	3
2	Análise e especificação do problema	4
2.1	Descrição e objetivos	4
2.2	Formato <i>PLY-Simple</i>	4
3	Proposta de solução	7
3.1	Estrutura da solução	7
3.2	Analizador lexical	7
3.3	Analizador sintático	7
3.4	Tradução do conteúdo	8
3.4.1	LEX	9
3.4.2	YACC	10
4	Testes de tradução	12
4.1	Ficheiros de teste	12
5	Conclusão	21
A	Definição dos tokens	22
B	Corpo do analisador sintático	24

Listas de Figuras

2.1	Sintax base do formato <i>PLY-Simple</i> .	5
4.1	Ficheiro de teste: <i>example1.txt</i> .	12
4.2	<i>Output</i> da tradução do ficheiro <i>example1.txt</i> .	13
4.3	Ficheiro de teste: <i>example2.txt</i> .	13
4.4	<i>Output</i> da tradução do ficheiro <i>example2.txt</i> .	14
4.5	Ficheiro de teste: <i>example3.txt</i> .	14
4.6	<i>Output</i> da tradução do ficheiro <i>example3.txt</i> .	14
4.7	Ficheiro de teste: <i>example4.txt</i> .	15
4.8	<i>Output</i> da tradução do ficheiro <i>example4.txt</i> .	15
4.9	Ficheiro de teste: <i>example5.txt</i> .	16
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
4.19	Ficheiro de teste: <i>example10.txt</i> .	20
4.20	Ficheiro de teste: <i>example11.txt</i> .	20
4.21	<i>Output</i> da tradução dos ficheiros <i>example10.txt</i> e <i>example11.txt</i> .	20

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. O processo de tradução ocorre com o auxílio de um analisador léxico e um analisador sintático, que visam validar a construção gramatical do ficheiro *PLY-Simple*, para que possam ser geradas as correspondentes componentes *lex* e *yacc*.

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, a fim de corretamente gerar as componentes necessárias para o seu posterior funcionamento.

Posto isto, o trabalho centra-se na criação de um programa 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 sintaxe final usada surgiu tomou este exemplo como base, tendo sido adotadas algumas alterações, com vista a acomodar as funcionalidades a implementar.

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.

```

%% 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: Sintax base do formato *PLY-Simple*.

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. Na versão atual de *PLY-Simple*, não são processados comentários, pelo que a sua utilização não é atualmente possível. Adicionalmente, e com vista a produzir um *parser* PLY válido, foi 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 %%*;
 - 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* deve iniciar-se com um *%*, sendo reservadas estas palavras 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 carácter opcional;
 - * A informação dos *literals* tem de ser feita sempre com recurso a uma string;
 - 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;
- A definição de um *token* é iniciada com o marcador %):
 - * Os *tokens* são definidos pela sua *regex* seguida de *simpleToken*('TOKEN', '') – onde TOKEN refere o nome do *token* a ser definido;
 - * *Tokens* podem ser definidos sobre a forma de funções, sendo necessário apresentar o *return* pretendido após a *regex*;
 - 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 plicas, 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 o marcador %), seguido de uma expressão regular e da palavra reservada *error*;
- Os caracteres /% precedem a definição de variáveis do *lexer*.
- Podem ser definidos estados para o *lexer*, com o marcador % states =
 - * A definição dos estados consiste numa lista de estados, começando com / e terminando com o marcador /;.

- YACC:

- A componente referente ao YACC é iniciada aquando do reconhecimento de %% YACC %%;
- A definição de uma gramática é de caráter obrigatório, sendo informado o utilizador com uma mensagem de erro na eventualidade de esta informação se encontrar omisssia;
- A informação da gramática inicia-se com a leitura de uma linha com o marcador /grammar
 - * Cada linha subsequente representa uma produção da gramática;
 - * As operações sintáticas a realizar encontram-se na mesma linha que a produção, sendo usados {} para a sua delimitação
- É necessário usar um % para definir a variável precedence, de caráter opcional, que descreve a prioridade dos *tokens* definidos no LEX;
 - * A definição das precedências consiste numa lista, começando com / e terminando com o marcador /;
- A definição de uma função inicia-se com a introdução dos caracteres ~) no início de uma linha
 - * Deve ser usada a palavra reservada def, seguida de um espaço para definir a função;
 - * O nome das funções pode conter caracteres alfanuméricos, assim como os argumentos da mesma;
 - * A enumeração da suas instruções ocorre nas linhas seguintes;
 - * A indentação das instruções, com um tab, deve ser inserida corretamente na sua escrita, de modo a seja possível determinar o seu aninhamento correto;
- Informação adicional:
 - * Todas as instruções adicionais associadas com a execução do parser ou possíveis testes são precedidas pelo marcador /%, que pode ser seguido (ou não) por um, e somente um, espaço;
 - * Em momento algum é efetuada verificação da correção das instruções adicionais fornecidas pelo utilizador através do ficheiro de input.

Embora seja conferida alguma liberdade acerca da presença de determinadas variáveis, as secções a inserir possuem uma ordem a ser respeitada aquando da sua definição – com exceção de LEX e YACC que podem ser definidos sob qualquer ordem. Assim sendo, foi definida a seguinte ordem de inserção das secções nas componentes:

- LEX: definição de variáveis, definição de funções, definição de estados, instruções do parser;
- YACC: definição das precedências, definição de dicionário, produções da gramática, inserção de funções, lista de instruções do parser.

Capítulo 3

Proposta de solução

3.1 Estrutura da 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. Os módulos *plySimple_lexer.py* e *plySimple_yacc.py* encontram-se executam a verificação sintática e lexical dos ficheiros *PLY-Simple*. O módulo *helper.py* dispõe de um conjunto de funções usadas nos módulos anteriores.

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) análise lexical e sintática do ficheiro de *input*; (ii) processamento e escrita do LEX; (iii) processamento e escrita do YACC. Para efetuar o processo de tradução, do ficheiro *PLY-Simple* para *PLY*, foram construídos analisadores (léxico e sintático), de modo a validar o ficheiro de *input* – o processo de tradução apenas ocorre caso o ficheiro recebido tenha sido validado pelos analisadores.

3.2 Analisador lexical

O analisador lexical construído permite definir o conjunto de palavras consideradas válidas para a linguagem definida pelo *PLY-Simple*, apresentado previamente na seção 2.2.

Assim sendo, a variável *tokens* define o conjunto de estruturas da linguagem válidas para o formato *PLY-Simple* – cujo código pode ser consultado no Anexo A. Os *tokens* foram todos definidos sob a forma de funções, com vista a poder efetuar as transições de estados sempre que necessário. Adicionalmente, foram ainda definidos os caracteres a ignorar e a ação a ser executada aquando da deteção de erros – sendo impressa uma mensagem com o erro.

Tal como foi mencionado, foram definidos alguns estados do *parser*, de modo a garantir que o reconhecimento dos *tokens* inseridos no ficheiro em formato *PLY-Simple* ocorra corretamente. A definição destes estados encontra-se discriminada no excerto de código abaixo apresentado, incorporado no ficheiro *plySimple_lexer.py*.

```
states = [
    ("lex", 'exclusive'),
    ("fun", 'exclusive'),
    ("yacc", 'exclusive'),
    ("grammar", 'exclusive'),
    ("def", 'exclusive'),
    ("parser", 'exclusive')
]
```

3.3 Analisador sintático

O analisador sintático encontra-se descrito no ficheiro *plySimple_yacc.py*. O grande trabalho executado nesta componente centrou-se na definição da gramática de tradução, que define o conjunto de frases válidas para a linguagem, i.e., este conjunto de regras possibilita a validação de um ficheiro *PLY-Simple* – as produções da gramática de tradução encontram-se apresentadas no excerto de código abaixo apresentado.

```

"""
PlySimple : BLEX Lex BYACC Yacc EOF
| BLEX Lex EOF
| BYACC Yacc EOF
| BYACC Yacc BLEX Lex EOF
| EOF
Yacc : Precedence Dictionary Grammar Defs InstrList
Precedence : BPREC List
|
List : List psList
|
Dictionary : TS tsList
|
Grammar : BGRAM ProdList
|
ProdList : ProdList prod
|
Defs : Defs Def
|
Def : BDEF ListD
ListD : ListD definition
|
InstrList : InstrList Inst
|
Inst : BINST instruction
Lex : Vars Funs States InstrList
Vars : Vars Var
|
Var : LIT Liters
| IGN Liters
| TOK tokenList
Liters : literals
| literalsV2
States : BSTAT List
|
Funs : Funs Fun
|
Fun : BFUN function
"""

```

Através da análise das regras da gramática é possível verificar que, tal como foi anteriormente mencionado, existe alguma flexibilidade relativamente à presença de alguns componentes no ficheiro *PLY-Simple*. A título de exemplo, as primeiras produções permitem que um ficheiro com este formato possua as componentes LEX e YACC numa ordem arbitrária e, até mesmo, que uma destas componentes esteja ausente. A definição de uma gramática de tradução correta e, ao mesmo tempo, flexível foi crucial para a validação da correção do conteúdo do ficheiro *PLY-Simple*.

O corpo da execução do analisador sintático encontra-se apresentado no Anexo B, sob a forma de pseudocódigo a fim de demonstrar o raciocínio adotado. Caso o ficheiro seja não seja processado com sucesso, é enviada uma mensagem de aviso ao utilizador a indicar que o ficheiro não respeita a sintaxe definida para a linguagem. No caso contrário, o processo de tradução é iniciado.

3.4 Tradução do conteúdo

O processo de tradução começa com a divisão dos conteúdos do ficheiro entre as componentes de LEX e YACC, sendo efetuado com recurso à função *get_lex_yacc*. Esta função começa por iterar sobre o conteúdo do ficheiro, recebido como sendo uma lista de *strings*, identificando a posição em que se iniciam as componentes LEX e YACC – sendo, também, controlada a existência destas duas componentes. Tal como foi referido, 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. Assim sendo,

e na eventualidade de tal se verificar, é emitida uma mensagem de aviso ao utilizador, para informar que o ficheiro em processamento não incorpora uma das componentes, identificando qual a componente em falta.

3.4.1 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 recebido 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 interesse 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
        (...)

        if run_tokens and run_error and run_ignore:
            (...)

            for tok in tokens:
                tok_func, tok_no_func = process_tokens(tok, list_regex)
                res_toks_func = res_toks_func + tok_func
                res_toks_no_func = res_toks_no_func + tok_no_func

                res += res_literals + res_tokens_list + res_var + res_ignore
                + res_toks_no_func + "\n" + res_toks_func
            (...)
```

Findo este processamento, o 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. 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.

3.4.2 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 carácter 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 gramática¹. 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]):
    (...)

    while pos < len(lines):
        if re.match(r'^/grammar$',lines[pos]):
            found_grammar = True
            for subline in lines[pos+1:]:
                if re.match(r'\w+ : .*',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*. 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 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]):
    (...)


```

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

```

for line in grammar:
    if line != "":
        group = (re.findall(r'([^\: ]+)(?: *: *(.*?){2,})(?:{ *(.*) *}',line))[0]
        prod = re.sub(r'(.*)', '', group[1])      # remover, por exemplo, %prec UMINUS
        op = re.findall(r'(.*)[^A-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]} -> ' in 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"""
(...)
```

Findo 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. Este ficheiro *PLY-Simple* obedece à sintaxe estipulada na Secção 2.2, sobressaindo, no entanto, as seguintes diferenças comparativamente à sintaxe base seguida:

- a gramática é introduzida a partir da leitura do marcador */grammar*;
- as instruções associadas à execução do *parser* devem ser iniciadas pelos caracteres */%*;
- a definição das funções é introduzida com os caracteres *~)*;
- a variável *precedence* pode ser escrita em várias linhas, tomando como terminação o marcador ; desde que o carácter / se apresente na segunda linha.

```
input> E example1.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 %% . error(f"Illegal character '{t.value}' [{t.lexer.lineno}]", t.lexer.skip(1) )
9
10 %% YACC %%
11
12 %precedence =
13   [(left,'+', '-'),
14   ('left','*', '/'),
15   ('right','UMINUS')];
16
17 %ts = {}
18
19 /grammar
20 stat : VAR '=' exp           { ts[t[1]] = t[3] }
21 stat : exp                  { print(t[1]) }
22 exp : exp '+' exp          { t[0] = t[1] + t[3] }
23 exp : exp '-' exp          { t[0] = t[1] - t[3] }
24 exp : exp '*' exp          { t[0] = t[1] * t[3] }
25 exp : exp '/' exp          { t[0] = t[1] / t[3] }
26 exp : '-' exp %prec UMINUS { t[0] = -t[2] }
27 exp : '(' exp ')'          { t[0] = t[2] }
28 exp : NUMBER               { t[0] = t[1] }
29 exp : VAR                  { t[0] = getval(t[1]) }
30
31 ~) def p_error(t):
32     print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
33
34 ~) def getval(n):
35     if n not in ts:
36         print(f"Undefined name '{n}'")
37     return ts.get(n, 0)
38
39 /* yacc.yacc()
40 /* y.parse("3+4*7")
41
42
43 $$
```

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.

```

example1_yacc.py U ...
output > # import ply.lex as lex
1 # import ply.yacc as yacc
2 tokens = ["VAR", "NUMBER"]
3 literals = "+-*/"
4 ignore = "\t\n"
5
6 def t_VAR(t):
7     r'[A-Za-z][A-Za-z0-9]*'
8     t.value = float(t.value)
9     return t
10
11 def t_NUMBER(t):
12     r'\d+\.\d*|\d+'
13     t.value = float(t.value)
14     return t
15
16 def t_error(t):
17     print(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
18     t.lexer.skip(1)
19
20 lexer = lex.lex()
21
22 def t_error(t):
23     print(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
24     t.lexer.skip(1)
25
26 tokens = ["NUMBER", "PLUS", "MINUS", "MULTIPLY", "DIVIDE", "ASSIGN"]
27 precedence = [("left", "+", "-"), ("left", "*", "/"), ("right", "UMINUS")]
28
29 class ts:
30     def __init__(self):
31         self.d = {}
32
33     def get(self, v):
34         if v not in self.d:
35             print(f"Undefined name '{v}'")
36             return None
37         return self.d[v]
38
39     def put(self, v, t):
40         self.d[v] = t
41
42 def p_start(p):
43     "stat : VAR '=' exp"
44     p[0] = p[3]
45
46 def p_exp(p):
47     "exp : NUM"
48     p[0] = p[1]
49
50 def p_exp(p):
51     "exp : exp '+' exp"
52     p[0] = p[1] + p[3]
53
54 def p_exp(p):
55     "exp : exp '-' exp"
56     p[0] = p[1] / p[3]
57
58 def p_exp(p):
59     "exp : NUMBER"
60     p[0] = p[1]
61
62 def p_exp(p):
63     "exp : VAR"
64     p[0] = getval(p[1])
65
66 def p_error(t):
67     print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
68
69 yacc.yacc()
70 y.parse('3+4*5')
71

```

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.

```

example2_yacc.py U ...
output > #! /usr/bin/python
1 # LEX %
2
3 %tokens = ['PA', 'PF', 'NUM', 'VIRG']
4 %ignore = "\t\n"
5
6 %% \d+
7 %% ,
8 %% \
9 %% \
10 %% .
11 %% YACC %
12
13 /grammar
14
15 Lista : PA PF
16     {}
17 Lista : PA Elems PF
18     {}
19 Elems : Elemt Resto
20     {}
21 Resto : VIRG Elems
22     {}
23 Elemt : NUM
24     {}
25 Elemt : Lista
26     {}
27 ~) def p_error(p):
28     print("Erro sintático",p)
29
30 parser = yacc.yacc()
31
32 $$

```

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_lexer.py
1 import ply.lex as lex
2
3 tokens = ['PA', 'PF', 'NMN', 'VING']
4 t_ignore = " \t\n\r"
5 t_PA = r'PA'
6 t_PFN = r'PF'
7 t_NMN = r'NMN'
8 t_VING = r'VING'
9
10 def t_error(t):
11     print(f"illegal character '{t.value[0]}', [{t.lexer.lineno}]")
12     t.lexer.skip(1)
13
14 lexer = lex.lex()
15

example2_yacc.py
1 import ply.yacc as yacc
2 from example2_lexer import *
3
4 # GRAMMAR:
5 # p1: Lista -> PA PF
6 # p2: PA Elems
7 # p3: Elems -> Eles Resto
8 # p4: Resto ->
9 # p5: | VING Elems
10
11
12
13 def p_Lista_p1(p):
14     "Lista : PA PF"
15
16
17 def p_Lista_p2(p):
18     "Lista : PA Elems PF"
19
20
21 def p_Elems_p3(p):
22     "Elems : Eles Resto"
23
24
25 def p_Elems_p4(p):
26     "Resto : "
27
28
29 def p_Resto_p5(p):
30     "Resto : VING Elems"
31
32
33 def p_error(p):
34     print("erro sintático", p)
35
36 parser = yacc.yacc()
37

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```

annaphens@Anas-Air ~ % cd src
annaphens@Anas-Air src % python3.10 plySimple_yacc.py example2.txt
[example2] opened successfully.
[example2] lexical analysis successful.
[example2] translated LEX successfully.
[example2] translated YACC successfully.
[example2] generated parser.
annaphens@Anas-Air src %
annaphens@Anas-Air src %
annaphens@Anas-Air src %

```

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

Tal como mencionado na Secção 2.2, É esperado que o ficheiro siga uma determinada estrutura e, como tal, após a inserção dos *tokens* deve, caso exista, seguir-se a definição dos literais – com recurso ao marcador `%literals=`. Todavia, no ficheiro *example3.txt*, ilustrado na Figura 4.5, como é inserido um `%signal` a seguir aos *tokens*, o parser não é capaz de determinar um *match* com os *tokens* esperados, pelo que o ficheiro não respeita a gramática definida, originando, assim, uma mensagem de erro devido à sua errada composição sintática.

```

input > 例 example3.txt
1 %% LEX %%
2
3 %tokens = ['INI','FIM','nome','real','int','sinal']
4 %signal = ['=+\\-*(/)']
5 %ignore = "\n\t"
6
7 %(?:begin)\n|                                return('INI','')
8 %\|{|\{eE|\{N|\{dD|                                return('FIM','')
9 %[a-zA-Z] |[a-zA-Z0-9]*|                                return('nome','')
10 %[0-9]+.|[0-9]+|                                return('real','')
11 %[0-9]+|                                return('int','')
12 %.|                                error(f"illegal character '{t.value[0]}', [{t.lexer.lineno}]")
13
14 /* lexer = lex.lex()
15 */
16 $$


```

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

Essa mensagem de erro pode ser observada na Figura 4.6, sendo a seguinte: [ERROR] file *example3.txt* does not respect lexical/syntactic structure for PLY-Simple. Consequentemente, nem o ficheiro *example3_lexer.py*, nem o ficheiro *example3_yacc.py* são gerados pelo tradutor.

```

annaphens@Anas-Air src %
annaphens@Anas-Air src %
annaphens@Anas-Air src % python3.10 plySimple_yacc.py example3.txt
[example3] opened successfully.
Illegal character '%', [1]
Illegal character 's', [1]
Illegal character 'i', [1]
Illegal character 'g', [1]
Illegal character 'n', [1]
Illegal character 'a', [1]
Illegal character 'r', [1]
Illegal character 't', [1]
[ERROR] file example3.txt does not respect lexical/syntactic structure for PLY-Simple.
[ERROR: lexToken(literals2, "'[=+\\\\-*(/)']", 1, 73)]
annaphens@Anas-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 > E example4.txt
1  %% LEX %%
2
3  %literals = "()+*"
4  %tokens = ['num','FIM']
5  %ignore = " \t\n"
6
7  %%d+           return('num','t.value=int(t.value)')
8  %%(.           simpleToken('FIM','')
9  %%.           error(f"Caracter ilegal: {t.value[0]}"),t lexer.skip(1))
10
11 /* lexer = lex.lex()
12
13 %% YACC %%
14
15 /grammar
16
17 Z : Exp FIM          {print("Resultado: ", p[1])}
18 Exp : '(' '+' Lista Exp ')' {p[0] = somatorio(p[3]) + p[4]}
19 Exp : '(' '*' Lista Exp ')' {p[0] = produtorio(p[3]) * p[4]}
20 Exp : num             {p[0] = p[1]}
21 Lista : Lista Exp    {p[0] = p[1] + [p[2]]}
22 Lista : Exp           {p[0] = [p[1]]}
23
24 ~) def p_error(p):
25     print("Erro sintático: ", p)
26     parser.success = False
27
28 ~def produtorio(lista):
29     res = 1
30     for elem in lista:
31         res *= elem
32     return res
33
34 ~def somatorio(lista):
35     res = 0
36     for elem in lista:
37         res += elem
38     return res
39
40 /* parser = yacc.yacc()
41
42 %% import sys
43 %% for linha in sys.stdin:
44     # parser.success = True
45     # parser.parse(linha)
46     if parser.success:
47         print('Frase válida: ', linha)
48     else:
49         print('Frase inválida.')
50
51 $$
```

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_lexer.py* e *example4_yacc.py*, sem qualquer mensagem de erro ou warning.

Figura 4.8: *Output* da tradução do ficheiro *example4.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 *example5.txt*, ilustrado na Figura 4.9, no qual apenas não está presente a componente LEX.

```
input > %% example5.txt
1 %% YACC %%
2
3 /grammar
4 S : A a S          {}
5 S : A              {}
6 A : A a F          {}
7 A : F              {}
8 F : F S            {}
9
10
11 ~) def p_error(p):
12     print("Erro sintático: ", p)
13     parser.success = False
14
15
16
17 % parser = yacc.yacc()
18
19 $$
```

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*.

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

No seguinte ficheiro de teste, *example6.txt*, apresentado na Figura 4.11, a gramática não é introduzida pelo marcador */grammar*, levando a que o tradutor não consiga reconhecê-la. Isto causará problemas visto que, como mencionado na Secção 3.4.2, a gramática é uma componente obrigatória do YACC.

```
input > %% example6.txt
1  %% LEX %%
2
3  %tokens = ['Z', 'U']
4  %ignore = " \t\n"
5
6  %% 0           simpleToken('Z','')
7  %% 1           simpleToken('U','')
8  %% .           error("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      Ø
15 S : U S U      Ø
16 S : Z          Ø
17 S : U          Ø
18
19 ~) def p_error(p):
20     print("Erro sintático: ", p)
21     parser.success = False
22
23 /* parser = yacc.yacc()
24
25 $$
```

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] file example6.txt does not respect lexical/syntactic structure for PLY-Simple.** Isto impossibilitará, então, o tradutor de gerar tanto o ficheiro *example6_lexer.py*, como o *example6_yacc.py*.

```
annaphens@Anas-Air src %
annaphens@Anas-Air src % python3.10 plySimple_yacc.py example6.txt
[example6] opened successfully.
Illegal character ':', [1]
Illegal character 'Z', [1]
Illegal character 'S', [1]
Illegal character 'Z', [1]
Illegal character 'S', [1]
Illegal character ':', [1]
Illegal character 'U', [1]
Illegal character 'S', [1]
Illegal character 'U', [1]
Illegal character 'S', [1]
Illegal character ':', [1]
Illegal character 'Z', [1]
Illegal character 'S', [1]
Illegal character ':', [1]
Illegal character 'U', [1]
[ERROR] file example6.txt does not respect lexical/syntactic structure for PLY-Simple,
["ERROR : LexToken(tslist,'{}',1,340)"]
annaphens@Anas-Air src %
```

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

Para comprovar que o tradutor não enfrenta nenhuns 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 a componente YACC é, então, definido antes da componente LEX.

```
input > E example7.txt
1 %% YACC %%
2
3 /grammar
4
5 Call : Comandos FIM          {}
6 Comandos : Comandos Comando  {}
7 Comandos : Comando          {}
8 Comando : id `*' Exp        {p.parser.registros[p[1]] = p[3]}
9 Comando : `!` Exp           {p.print(p[2])}
10 Comando : `?` id            {p.parser.registros[p[2]] = input("Introduzir um valor inteiro: ") }
11 Comando : DUMP             {p.print(p.parser.registros)}
12 Exp : Exp `+` Termo         {p[0] = p[1] + p[3]}
13 Exp : Exp `-*` Termo        {p[0] = p[1] - p[3]}
14 Exp : Termo                {p[0] = p[1]}
15 Termo : Termo `*` Fator    {p[0] = p[1] * p[3]}
16 Termo : Fator              {p[0] = p[1]}
17 Fator : num                 {p[0] = p[1]}
18 Fator : `(` Exp `)`
19 Fator : id                  {p[0] = p.parser.registros[p[1]]}
20
21
22
23 /* parser = yacc.yacc()
24 /* parser.registros = {}
25
26 %% LEX %%
27
28 %literals = "+-*()=?"
29 %tokens = ['num', 'id', 'DUMP', 'FIM']
30 %ignore = " \t\n"
31
32 %|[_A-Za-z]\w*      simpleToken('id','')
33 %|!      simpleToken('DUMP','')
34 %|.
35 %|.
36 %|d+
37 %|.
38
39 /* lexer = lex.lex()
40
41
42 $$
```

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*.

```
example2.yacc.u
output > example2_yacc.py ...
  import p_lexer as lex
  ...
  literals = '\"-+/*)?!>\"'
  tokens = [ 'NUM', 'ID', 'DUMP', 'TERM' ]
  t_LNUM = r'(\.\d+|\d+)\d*'
  t_ID = r'[A-Z][a-zA-Z]*'
  t_DUMP = r'\".*\"'
  t_TERM = r'\n'
  ...
  def t_NUM(t):
    r'(\.\d+|\d+)\d*'
    t.value = int(t.value)
    return t
  ...
  def t_error(t):
    print("Illegal character '%s'" % t.value)
    t.lexer.skip(1)
  ...
  lexer = lex.lex()
  ...

example2.yacc.u
output > example2.yacc.py ...
  import p_yacc_lexer as yacc
  ...
  #GRAMMAR#
  # 1: Cal -> Comandos FIN
  # 2: ID -> Comando
  # 3: DUMP -> DUMP
  # 4: TERM -> TERM
  # 5: Calculo -> Calculo id '+' Exp
  # 6: Calculo -> Calculo id '-' Exp
  # 7: Calculo -> Calculo id '*' Exp
  # 8: Calculo -> Calculo id '/' Exp
  # 9: Calculo -> Calculo id '^' Exp
  # 10: Exp -> '+' id
  # 11: Exp -> '-' id
  # 12: Exp -> '*' id
  # 13: Exp -> '/' id
  # 14: Exp -> '^' id
  # 15: Exp -> Termo
  # 16: Exp -> Fator
  # 17: Fator -> num
  # 18: Fator -> '(' Exp ')'
  # 19: Fator -> id
  ...
  def p_Calc(p):
    "Calc : Comandos FIN"
    p[0] = p[1]
  def p_Commando(p):
    "Comando : ID"
    p[0] = p[1]
  def p_Commando_p1(p):
    "Comandos : Comandos Comando"
    p[0] = p[1] + p[2]
  def p_Commando_p2(p):
    "Comandos : Comando"
    p[0] = p[1]
  def p_Calculo(p):
    "Calculo : id '+' Exp"
    p[0] = p[1] + p[2]
  def p_Calculo_p1(p):
    "Calculo : id '-' Exp"
    p[0] = p[1] - p[2]
  def p_Calculo_p2(p):
    "Calculo : id '*' Exp"
    p[0] = p[1] * p[2]
  def p_Calculo_p3(p):
    "Calculo : id '/' Exp"
    p[0] = p[1] / p[2]
  def p_Calculo_p4(p):
    "Calculo : id '^' Exp"
    p[0] = p[1] ** p[2]
  def p_Termo(p):
    "Termo : Fator"
    p[0] = p[1]
  def p_Fator(p):
    "Fator : num"
    p[0] = p[1]
  def p_Fator_p1(p):
    "Fator : '(' Exp ')'"
    p[0] = p[1]
  def p_Fator_p2(p):
    "Fator : id"
    p[0] = p[1]
  ...
  parser = yacc.yacc()
  parser.registros = {}

example2.yacc.py -P example2.yacc.u
  ...
  # 1: Cal -> Comandos FIN
  # 2: ID -> Comando
  # 3: DUMP -> DUMP
  # 4: TERM -> TERM
  # 5: Calculo -> Calculo id '+' Exp
  # 6: Calculo -> Calculo id '-' Exp
  # 7: Calculo -> Calculo id '*' Exp
  # 8: Calculo -> Calculo id '/' Exp
  # 9: Calculo -> Calculo id '^' Exp
  # 10: Exp -> '+' id
  # 11: Exp -> '-' id
  # 12: Exp -> '*' id
  # 13: Exp -> '/' id
  # 14: Exp -> '^' id
  # 15: Exp -> Termo
  # 16: Exp -> Fator
  # 17: Fator -> num
  # 18: Fator -> '(' Exp ')'
  # 19: Fator -> id
  ...
  def p_Calc(p):
    "Calc : Comandos FIN"
    p[0] = p[1]
  def p_Commando(p):
    "Comando : ID"
    p[0] = p[1]
  def p_Commando_p1(p):
    "Comandos : Comandos Comando"
    p[0] = p[1] + p[2]
  def p_Commando_p2(p):
    "Comandos : Comando"
    p[0] = p[1]
  def p_Calculo(p):
    "Calculo : id '+' Exp"
    p[0] = p[1] + p[2]
  def p_Calculo_p1(p):
    "Calculo : id '-' Exp"
    p[0] = p[1] - p[2]
  def p_Calculo_p2(p):
    "Calculo : id '*' Exp"
    p[0] = p[1] * p[2]
  def p_Calculo_p3(p):
    "Calculo : id '/' Exp"
    p[0] = p[1] / p[2]
  def p_Calculo_p4(p):
    "Calculo : id '^' Exp"
    p[0] = p[1] ** p[2]
  def p_Termo(p):
    "Termo : Fator"
    p[0] = p[1]
  def p_Fator(p):
    "Fator : num"
    p[0] = p[1]
  def p_Fator_p1(p):
    "Fator : '(' Exp ')'"
    p[0] = p[1]
  def p_Fator_p2(p):
    "Fator : id"
    p[0] = p[1]
  ...
  parser = yacc.yacc()
  parser.registros = {}
```

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

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*.

A seguir, 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\`]+```+``` error(f"Illegal character '{t.value[0]}", [(t.lexer.lineno)], t.lexer.skip(1) )
9 %%
10
11 /* lexer = lex.lex()
12
13 %% YACC %%
14
15 /grammar
16 Lista : Lista Ficheiro          {}
17 Lista :                      {}
18 Ficheiro : `(` id id `)`
19
20 ~ def p_error(p):
21     print("Erro sintático: ", p)
22     parser.success = False
23
24 /* parser = yacc.yacc()
25 /* parser.items = []
26
27 $$
```

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 3.4.2: 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*.

Finalmente, temos os ficheiros de teste *example10.txt*, no qual não está definida uma gramática, e *example11.txt*, no qual não está definida o YACC, que estão apresentados nas Figuras 4.19 e 4.20, respetivamente.

```
input > %% example10.txt
1 %% LEX %%
2 %literals = "+-/*=()\""
3 %ignore = " \t\n\r"
4 %tokens = ['VAR', 'NUMBER']
5
6 %% LEX %%
7 %d!(\.\w+)? return('NUMBER', t.value = float(t.value))
8 %% YACC %%
9 . error(f"Illegal character '{t.value[0]}'", [t.lexer.lineno]), t.lexer.skip(1)
10
11 %% YACC %%
12 %precedence = [('left','+', '-'), ('left','*', '/'), ('right','UMINUS')]
13
14 %ts = {}
15
16
17 ~) def p_error(t):
18     print(f"Syntax error at '{t.value}', [{t.lexer.lineno}]")
19
20 ~) def getval(n):
21     if n not in ts:
22         print(f"Undefined name '{n}'")
23     return ts.get(n, 0)
24
25 %% yacc %%
26 %% y.parse("3+4*7")
27
28
29 $$
```

Figura 4.19: Ficheiro de teste: *example10.txt*.

Figura 4.20: Ficheiro de teste: *example11.txt*.

Como podemos observar pela Figura 4.21, a tradução do ficheiro *example10.txt* gerou a mensagem de erro [ERROR] *grammar not found on YACC*, enquanto que a do *example11.txt* gerou a mensagem de alerta [WARNING] *nothing defined for YACC in PLY-Simple*. Consequentemente, apenas obtivemos a escrita dos ficheiros *example10_lexer.py* e *example11_lexer.py*.

```
example10_lexer.py -- TP2
...
example10_lexer.py X
output> <example10_lexer.py> ...
1 import ply.lex as lex
2
3 literals = ['+', '-', '*', '/']
4 tokens = ['NAME', 'NUMBER']
5 t_ignore = ' \t\n'
6
7 def t_NAME(t):
8     r'[a-zA-Z_][a-zA-Z_0-9a-zA-Z]*'
9     return t
10
11 def t_NUMBER(t):
12     r'\d+(\.\d+)?'
13     t.value = float(t.value)
14     return t
15
16 def t_error(t):
17     print(f"Illegal character '{t.value[0]}', [{t.lexpos}, {t.lineno}]")
18     t.lexer.skip(1)
19
20 lexer = lex.lex()
21

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
annphegns-Mac-Air src % python3.10 ply/simple yacc.py example9.txt example8.txt example10.txt
example10 approved by lexical and syntactic analysis.
example9 approved by lexical and syntactic analysis.
example8 translated LEX successfully.
example7 opened successfully.
example6 opened successfully.
example5 opened successfully.
example4 approved by lexical and syntactic analysis.
example3 translated LEX successfully.
example2 opened successfully.
example11 opened successfully.
example10 approved by lexical and syntactic analysis.
example9 approved by lexical and syntactic analysis.
[WARNING] nothing defined for YACC in PLY-Simple.
example10 approved by lexical and syntactic analysis.
```

Figura 4.21: *Output* da tradução dos ficheiros *example10.txt* e *example11.txt*.

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, na utilização da linguagem de programação *Python* como ferramenta de desenvolvimento de *software* e, particularmente, na criação de gramáticas de tradução.

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. Algumas funcionalidades futuras poderiam ser inseridas, de modo a fortalecer o programa desenvolvido e a aumentar o conjunto de funcionalidades que este fornece. Nomeadamente a inclusão de comentários no formato *PLY-Simple* e a utilização de operações sintáticas (criando árvore de sintaxe abstrata representativa do conteúdo do ficheiro).

Apêndice A

Definição dos tokens

```
def t_ANY_EOF(t):
    r'\$\$'
    return t

def t_ANY_BINST(t):
    r'/%'
    t.lexer.begin("parser")
    return t

def t_parser_instruction(t):
    r'.*(\.| |=|\w+).*'
    t.lexer.begin("INITIAL")
    return t

def t_ANY_BYACC(t):
    r'%%\s*YACC\s*%%'
    t.lexer.begin("yacc")
    return t

def t_yacc_PREC(t):
    r'%\s*precedence\s*=\s*'
    return t

def t_yacc_preceList(t):
    r'\[(.*\)\\];'
    return t

def t_yacc_TS(t):
    r'%\s*ts\s*=\s*'
    return t

def t_yacc_tsList(t):
    r'\{\}'
    return t

def t_ANY_BGRAM(t):
    r'/grammar'
    t.lexer.begin("grammar")
    return t

def t_grammar_prod(t):
    r'.*\.:.*\{.*\}'
    return t

def t_ANY_BLEX(t):
    r'%%\s*LEX\s*%%'
```

```

t.lexer.begin("lex")
return t

def t_lex_TOK(t):
    r'%\s*tokens\s*=\s*'
    return t

def t_lex_tokenList(t):
    r'\[\*\.\*\]\'
    return t

def t_lex_LIT(t):
    r'%\s*literals\s*=\s*'
    return t

def t_lex_literals(t):
    r'\".*"'
    return t

def t_lex_literalsV2(t):
    r'\'.*\'''
    return t

def t_lex_IGN(t):
    r'%\s*ignore\s*=\s*'
    return t

def t_ANY_BFUN(t):
    r'\s*%\)\s*'
    t.lexer.begin("fun")
    return t

def t_fun_function(t):
    r'.*(return|error|simpleToken).*'
    t.lexer.begin("INITIAL")
    return t

def t_ANY_BDEF(t):
    r'\~\)'
    t.lexer.begin("def")
    return t

def t_def_definition(t):
    r'def\s*\w+\(\w+\):\s*\{\|\s.*\|\}'

def t_ANY_error(t):
    print(f"Illegal character '{t.value[0]}', [{t.lexer.lineno}]")
    t.lexer.skip(1)

```

Apêndice B

Corpo do analisador sintático

```
files = sys.argv[1:]
for file_name in files:
    try:
        input = open_file(file_name, 'YACC')
    (...)

if input != "":
    parser.success = True
    parser.parse(input)
    if parser.success:
        lines = input.splitlines()
        (...)

lex_exists, yacc_exists, lines_for_LEX, lines_for_YACC = get_lex_yacc(lines)
if lex_exists:
    try:
        res = translate_lex(lines_for_LEX)
        write_file_lex(file_name, res)
    (...)

if yacc_exists:
    try:
        res = translate_yacc(lines_for_YACC)
        write_file_yacc(file_name, res)
    (...)

(...)
```