

Relatório Projeto PL

Grupo 60 :

António Luís de Macedo Fernandes (a93312)

José Diogo Martins Vieira (a93251)

João Silva Torres (a93231)

Maio 15, 2022



Contents

1	Introdução	3
2	Definição da sintaxe	4
3	Esquema de tradução	5
4	Criação do tradutor	6
4.1	Definição das expressões regulares para o lex	6
4.2	Definição da gramática	7
4.3	Construção das Estruturas	10
4.4	Construção dos ficheiros	11
4.5	Error Handling	13
5	Options	15
5.1	Help	15
5.2	Template	15
6	Testes	17
6.1	Ficheiro Example	17
6.2	Ficheiro Projeto	20
7	Conclusão	22

1 Introdução

Para este trabalho foi-nos proposto a criação de um tradutor capaz de traduzir ficheiros ply-simple para ply. Para tal, foi necessário definir uma sintaxe para os ficheiros ply-simple, a partir dessa sintaxe, fazer o devido esquema de tradução, e finalmente, a implementação do tradutor. Será assim feita a devida explicação dos tópicos mencionados anteriormente, assim como a demonstração da conversão de alguns ficheiros de teste.

2 Definição da sintaxe

O primeiro passo passa pela definição da sintaxe.

Para tal, utilizando como base o exemplo dado no enunciado, fizemos algumas alterações de modo a facilitar o processo de parsing.

A primeira alteração feita é relativa às regras do Lex, no exemplo de base, as regras estavam definidas, como neste exemplo:

```
[a-zA-Z_][a-zA-Z0-9_]* return('VAR', t.value)
```

achamos que esta definição está um bocado confusa e tem algumas limitações, para o trabalho que precisa de desempenhar. A expressão regular não tem delimitadores, tornando assim difícil de conseguir reconhecer, a utilização do "return(token,código)" é demasiado complicada, podendo assim ser substituído por algo mais simples.

Assim, alteramos a sintaxe de modo a ser mais geral e para puderem ser usadas tanto nas regras do lex como nas regras do yacc, ficamos assim com uma definição de regra geral que está representada no exemplo abaixo:

```
VAR : "[a-zA-Z_][a-zA-Z0-9_]*" {}
```

Assim, o token agora encontra-se no início da linha, seguido de dois pontos, e a respetiva expressão regular entre aspas. Finalmente, encontra-se o código para esta regra que se vai encontrar dentro de chavetas.

Para delimitar, o que não é para ser feito parsing, temos a expressão "\$\$" que vai indicar que tudo o que se encontra depois disto, não vai sofrer alterações.

Temos assim, o novo ficheiro de exemplo que demonstra a sintaxe:

```

%%LEX

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

VAR : "[a-zA-Z_][a-zA-Z0-9_]*" { }
NUMBER : "\d+(\.\d+)?" {float(t.value)}
error : "." {print(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[p[1]] = p[3] }
stat : "exp" { print(p[1]) }
exp : "exp '+' exp" { p[0] = p[1] + p[3] }
exp : "exp '-' exp" { p[0] = p[1] - p[3] }
exp : "exp '*' exp" { p[0] = p[1] * p[3] }
exp : "exp '/' exp" { p[0] = p[1] / p[3] }
exp : "'-' exp %prec UMINUS" { p[0] = -p[2] }
exp : "'(' exp ')'" { p[0] = p[2] }
exp : "NUMBER" { p[0] = p[1] }
exp : "VAR" { p[0] = getval(p[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")
y.parse("4*7")

```

Figure 1: Exemplo da nova sintaxe

3 Esquema de tradução

Definida a sintaxe que iremos utilizar, tratamos de fazer o esquema de tradução.

O esquema de tradução é bastante simples.

Para as variáveis do PLY, aquelas precedem um '%'. Vão ficar exatamente com a mesma atribuição, tirando assim o '%', e fazendo a devida mudança para o token do ignore.

Para as regras do Lex, será traduzido da seguinte forma, para cada token será definida uma função, onde a primeira linha será a expressão regular, seguida do código, e por fim um return da variável passada como argumento.

Todos os comentários e definições de variáveis ficarão da mesma forma.

Para as regras do Yacc, para cada produção será definida uma função, que terá algo a diferenciar no nome para que estes não sejam iguais. A primeira linha será a regra, e de seguida o código.

Tudo o que se encontrar depois do "\$\$", ficará exatamente da mesma forma.

4 Criação do tradutor

Para o início da implementação do tradutor, foi necessário definir a gramática.

4.1 Definição das expressões regulares para o lex

Os tokens utilizados, foram acrescentados à medida que a gramática ia sendo criada. Serão apresentados agora, para não terem de ser explicados na explicação da gramática.

Como palavras reservadas, temos os seguintes pares (palavra,token):

- 'literals' : 'LITERALS',
- 'ignore' : 'IGNORE',
- 'tokens' : 'TOKENS',
- 'precedence' : 'PRECEDENCE',
- 'LEX' : 'LEX',
- 'YACC' : 'YACC',

Como literals temos a seguinte lista:

```
['%', '#', '[', ']', '"', '\'', '=', ':', '(', ')', '$', '{', '}', '']
```

Como tokens, temos:

```
['str', 'int', 'float', 'allQuotes', 'allBraces', 'allNoConv', 'COMMENT']
```

Para reconhecer strings, temos o token **str**:

```
r"\w+"
```

Para reconhecer float's, temos o token **float**:

```
r"\d+\.\d+"
```

Para reconhecer int's, temos o token **int**:

```
r"\d+"
```

Para reconhecer comentários, temos o token **COMMENT**:

```
r'\#.*'
```

Para reconhecer tudo o que esteja entre chavetas ({}), temos o token **allBraces**:

```
r"{.*}"
```

Para reconhecer tudo o que esteja entre aspas (") ou ('), temos o token **allQuotes**:

```
r"\".*?\\"'|'.*?'
```

Este utiliza um quantificador non-greedy (ou lazy), de modo a que para aspa final irá parar logo que encontrar uma. Assim, para a frase: "hello\"world", esta expressão irá apanhar "hello\", quando se quiser usar aspas (") numa expressão regular deve-se usar como limitador as plicas(''), e vice-versa. Traz a limitação de não se poder usar as duas em simultâneo, mas tentaremos arranjar uma solução mais à frente.

Para reconhecer tudo o que esteja desde (\$\$) até ao fim do ficheiro, temos o token **allNoConv**:

```
r"\$\$(.*\s*)+\Z"
```

4.2 Definição da gramática

Como regra principal da gramática, teremos apenas:

```
Program : Fases
```

Fases

Esta indica que o nosso programa será constituído apenas por Fases, que será:

```
Fases : Fases Fase
```

```
Fases : Fase
```

Fases pode ser constuído por uma ou mais Fases, a recursividade é feita à esquerda pois o Ply utiliza Bottom-up.

Uma Fase poderá ser de duas formas:

```
Fase : '%' '%' FactorFase
```

```
Fase : allNoConv
```

A primeira será para usada para as fases do Lex e do Yacc. A segunda será usada para o final das fases, para apanhar tudo desde "\$\$" até ao final do ficheiro, com o token "allNoConv".

Para o FactorFase, temos:

```
FactorFase : FaseId Group
```

Onde FaseId vai ser apenas o identificador da Fase, podendo este ser uma das palavras reservadas:

- LEX
- YACC

No Group vai ser onde vai ser definido o que uma fase pode conter:

```
Group : Declarations Rules
```

Um Group é constituído por Declarations e Rules. Assim, todas as declarações têm de ser definidas antes das regras.

Declarations

```
Declarations : Declarations Declaration
Declarations :
```

Pode haver assim, zero ou mais Declaration. Esta pode ser de três formas:

```
Declaration : % PlyDeclaration
Declaration : COMMENT
Declaration : str '=' Var
```

A primeira será para reconhecer as variáveis do PLY, podendo ser uma das seguintes palavras reservadas:

- ignore
- literals
- tokens
- precedence

A segunda será para reconhecer comentários.

Por último, temos a declaração de variáveis por parte do utilizador. Para as variáveis do PLY temos as seguintes produções:

```
PlyDeclaration : IGNORE '=' allQuotes"
PlyDeclaration : LITERALS '=' LiteralsFactor
PlyDeclaration : TOKENS '=' Array"
PlyDeclaration : PRECEDENCE '=' PrecedenceFactor"
```

Estas produções vão dizer quais os tipos de variáveis que cada uma delas aceita.

Para ignore e para os tokens é apenas uma, string("allQuotes") e array, respetivamente.

Para os literals e precedence já aceitam dois tipos. Para os literals pode ser array ou string.

Para a precedence pode ser array ou tuple.

Já veremos mais à frente as produções para estes tipos.

Para as variáveis definidas pelo utilizador temos a Var, que poderá ser:

```
Var : Array
Var : Dict
Var : Tuple
Var : allQuotes ElemFactor
Var : int
Var : float
```

Para as variáveis temos 3 tipos compostos:

- Array
- Dict
- Tuple

E três tipos simples:

- allQuotes ElemFactor
- int
- float
- allQuotes

A primeira produção serve para apanhar tanto strings simples, como uma string seguida de ":" e uma Var, usada para os dicionários.

Para os três tipos compostos temos:

Para o Array:

Array : '[' Lista ']'

Para o Dicionário :

Dict : '{' Lista '}'
Dict : allBraces

É necessário ter segunda produção, pois caso o dicionário seja definido numa linha, vai ser apanhado por esta regra, e terá que ser feito o devido tratamento.

Para o Tuple:

Tuple : '(' Lista ')'

Todos usam a produção Lista para as suas definições, esta é definida da seguinte forma:

Lista :
Lista : Elems Final

Assim uma Lista poderá ser vazia ou conter Elems, seguido de um Final, este será apenas para detetar um "," ou vazio para o último elemento da lista, sendo assim aceites tanto [1,2] como [1,2,].

Para os Elems temos:

Elems : Elems ',' Var
Elems : Var

Podendo assim ter um ou mais Var, que serão separados por ','.

Rules

Para as Rules temos:

```
Rules : Rules Rule
Rules : Rule
```

Podendo assim, existir uma ou mais Rule.

Como já vimos na definição na sintaxe, a Rule vai ser:

```
Rule : str ':' allQuotes allBraces Comment
```

Onde a Rule vai ser apanhada pelo token "allQuotes", como já vimos anteriormente irá parar na segunda '"' que encontrar, isto previne que faça match com uma aspa que se encontra dentro da secção do código, mas também não permite que tenha aspas na expressão regular.

O token "allBraces" irá captar tudo o código correspondente a esta regra. Temos assim definida a gramática para o nosso tradutor.

Temos por fim a produção Comment, que serve para identificar um comentário, sendo este opcional, no fim de uma regra. Temos assim a produção para o Comment:

```
Comment : COMMENT
Comment :
```

4.3 Construção das Estruturas

Temos como objetivo final a geração de dois ficheiros, um para o Lex e outro para o Yacc. Para tal, não vamos fazer a construção de apenas uma string durante as produções, mas vamos ter uma estrutura que irá guardar informação e no final do parsing, com essa estrutura vamos construir os dois ficheiros, caso uma das fases não se encontre, irá ser gerado o ficheiro para a fase que exista.

Para tal, vão ser preciso variáveis para o parser. Essas variáveis vão ser as seguintes:

- yacc
- lex
- current
- noConv
- sucess

As variáveis **yacc** e **lex**, serão um tuplo de duas posições compostas por um dicionário, que será para as regras e as variáveis do Ply, a segunda será um array para as variáveis e comentários.

O **current** será uma variável de auxílio para fazer referência ao tuplo onde serão adicionadas as regras e variáveis consoante a fase que nos encontramos.

O **noConv** será uma string onde será posto tudo o que não vai sofrer alteração.

A **sucess** será um bool inicializado a True, que para aquando da identificação de erros, será posto a False, e já não vai ser feita a construção dos ficheiros.

Durante o parsing, estas variáveis serão preenchidas da seguinte forma:

Na produção da *FaseId*, será atualizado a variável **current** para o tuplo a que esta fase corresponde, para a fase do Lex, a variável **lex**. Para a fase do Yacc, a variável **yacc**.

Na produção da *Declaration*, caso seja um comentário ou uma variável definida pelo utilizador, irá ser adicionado ao dicionário das variáveis. Para o comentário, a key irá ser o próprio comentário e o value será uma string vazia. Para a variável, a key será o nome da variável e o value será o seu resultado, já em formato de string.

Por fim, na produção *Rule* é onde serão guardadas as regras. Para tal, verificamos se o nome da regra se encontra no dicionário das regras, caso não se encontre, é adicionado um par onde a key corresponde ao nome e o value vai ser uma lista vazia. Tem de ser uma lista, pois nas produções do Yacc, um nome pode ter várias produções associadas a este.

De seguida, iremos buscar o valor, a que corresponde ao nome, que vai ser um array, e iremos adicionar um dicionário composto por dois campos, o campo "rule" e o campo "code", que irão corresponder ao que o próprio nome indica.

Assim, no final do parsing, teremos a variável **yacc** e **lex**, com as variáveis e as regras, correspondentes, assim como a variável **noConv**.

4.4 Construção dos ficheiros

Com as estruturas devidamente preenchidas, resta apenas construir a string para escrever no ficheiro.

Ficheiro Lex

Para tal temos a função *buildLex*, que recebe como argumentos o dicionários das regras e o array das variáveis, e vai retornar a string pronta para escrever no ficheiro.

Primeiramente, vamos adicionar o devido import.

De seguida, vamos descobrir qual a letra usada para a variável de argumento nas funções, para isso temos a função *findVarLex* que irá às regras ver qual foi a letra usada quando se fazia referência a atributos do lex token, como por exemplo: "value", "type", "lexer", etc. Caso não tenha sido usada nenhuma vez, iremos usar "t" como default, este resultado será guardado na variável global **lexVar**. Após isso, usamos a função *buildVar* para obter as variáveis, as variáveis já estão no formato string prontas para serem escritas, resta apenas adicionar a mudança de linha.

Com as variáveis feitas, resta nos apenas as regras.

Iremos percorrer assim o dicionário das regras, onde estas vão ser transformadas em função, ou não, caso sejam as variáveis do PLY.

Par as variáveis do PLY, iremos fazer a devida conversão para string caso seja uma lista. Caso não seja, iremos escrever os valores do par separados por um "=". Iremos ter em atenção dois casos específicos:

- caso a key seja "ignore" mudar para "t_ignore"
- caso a key seja "tokens" e existir uma variável "reserved" acrescentar o devido no final da string

Para as regras definidas pelo utilizador, primeiramente iremos verificar se o nome da regra se encontra na lista de Tokens, caso não se encontre um erro é levantado e a execução do programa para.

Caso se encontre, iremos construir a devida função. Par tal, temos a função *lexFunction* que trata disso, esta função vai receber o nome da função assim como o dicionário associado a esta.

Para superar a limitação da expressão regular não poder ter aspas e plicas em simultâneo, iremos aceitar que utilize aspas como delimitador e dentro da expressão regular em vez de aspas se possa meter "

"" , assim antes de adicionar a regra à string, irá se fazer a devida substituição. Adicionamos assim a regra, para o code vamos verificar se existem cast a precisar de ser extendidos assim como a devida separação das linhas.

Para tal, iremos usar a função *splitStatements* que irá criar uma lista com os statements correspondentes a uma linha, para tal, vai dar split segundo o ",". E depois irá verificar se esta vírgula, é usada dentro de parênteses '()', de chavetas '{}' ou de aspas '"', caso seja irá juntar essa parte com a parte seguinte de modo a no final ter uma lista com as devidas linhas.

Depois desta separação, iremos percorrer cada linha para ver se é necessário extender o cast, caso seja, é feito o devido tratamento. Também é verificado, se o utilizador tem uma linha com "#reserved", caso tenha significa que é nesta função que quer que seja verificado as palavras reservadas, e é feita a devida verificação.

Por fim, com todas as regras feitas verificamos apenas se houve algum token que não teve uma regra associada, caso haja, é levantado um erro e encerra o programa. Caso contrário, é retornada a string já pronta para escrever no ficheiro.

Ficheiro Yacc

A construção do ficheiro para o Yacc é muito semelhante à do Lex.

A função responsável por construir o Yacc é a *buildYacc* que recebe como argumentos o dicionário das regras e o array das variáveis.

Primeiramente, começamos por fazer o import, para tal vamos verificar se foi usado algum nome para o import ou não.

De seguida, iremos obter as variáveis com a função *buildVar* já utilizada anteriormente.

Vamos também encontrar a variável utilizada como argumento nas funções, para tal temos a função *findVarYacc*, que irá encontrar a variável utilizada para aceder aos elementos das produções.

Resta apenas as regras, para tal iremos percorrer o dicionário das regras.

Iremos verificar se a key é "precedence", pois é a única regra do PLY que se pode encontrar no Yacc, caso seja, iremos fazer a devida conversão para string.

Caso contrário, serão produções feitas pelo utilizador e para tal, iremos usar a função *buildGrammarRules* para construir as produções.

Nesta função iremos percorrer as produções, utilizando um contador, para adicionar esse número ao nome, caso o utilizador queira dar nome à função pode fazê-lo, para tal na parte do comentário tem de ter uma parte que diga $N=\{\text{nome}\}$, caso haja, vai se ter em conta esse nome para a função. Adicionamos depois a regra, antecedendo com o nome da produção e ":". De seguida, falta nos apenas o código, para tal, vamos utilizar a função *buildCodeStatements*, para fazer a devida separação das linhas, para tal utilizaremos outra vez a função *splitStatements*. Feita a divisão das linhas, iremos adicionar cada linha à string resultante.

Assim, retornamos a string resultante e adicionamos a conteúdo na variável "noConv", ficando assim pronto para escrever no ficheiro.

Ficamos assim com as duas strings prontas para escrever no ficheiro. É feita a devida escrita em cada ficheiro e termina assim o programa.

4.5 Error Handling

De modo a prevenir erros do utilizador na escrita do ficheiro, utilizamos várias produções na gramática feitas exclusivamente para os detetar e dar informação relativa a esse erro, de modo a facilitar a correção deste.

Veremos então as situações em que existe a verificação de erros:

1. Declaration : error '=' Var

- Para identificar variáveis que não sejam do tipo String
- **Output:** "Info : variable name has to be string"

2. PlyDeclaration : LITERALS '=' error

- Para identificar quando não se atribui uma variável do tipo array ou string à variável do Ply, Literals
- **Output:** "Info : literals value must be a string or array"

3. PlyDeclaration : IGNORE '=' error

- Para identificar quando não se atribui uma variável do tipo string à variável do Ply, Ignore
- **Output:** "Info : ignore value must be a string"

4. `PlyDeclaration : TOKENS '=' error`

- Para identificar quando não se atribui uma variável do tipo array à variável do Ply, Tokens
- **Output:** "Info : tokens value must be an array"

5. `PlyDeclaration : PRECEDENCE '=' error`

- Para identificar quando não se atribui uma variável do tipo array ou tuplo à variável do Ply, Precedence
- **Output:** "Info : precedence value must be a tuple or an array"

6. `PlyDeclaration : error '=' Var`

- Para identificar quando é utilizado um nome inválido como variável do Ply
- **Output:** "Info : There's no ply variable with name:'{value}'" (value -> the variable used by user)

7. `Rule : error ':' allQuotes allBraces Comment`

- Para identificar quando o nome de um rule não é do tipo string.
- **Output:** "Info : Rule name '{value}' is invalid. Has to be string." (value -> the rule name used by user)

8. `Rule : str ':' error allBraces Comment`

- Para identificar quando a rule não se encontra entre aspas(" ") ou plicas(' ').
- **Output:** "Info : Rule '{value}' is invalid. The rule has to be inside quotes()." (value -> the first token identified that was not inside the delimiters)

9. `Rule : str ':' allQuotes error Comment`

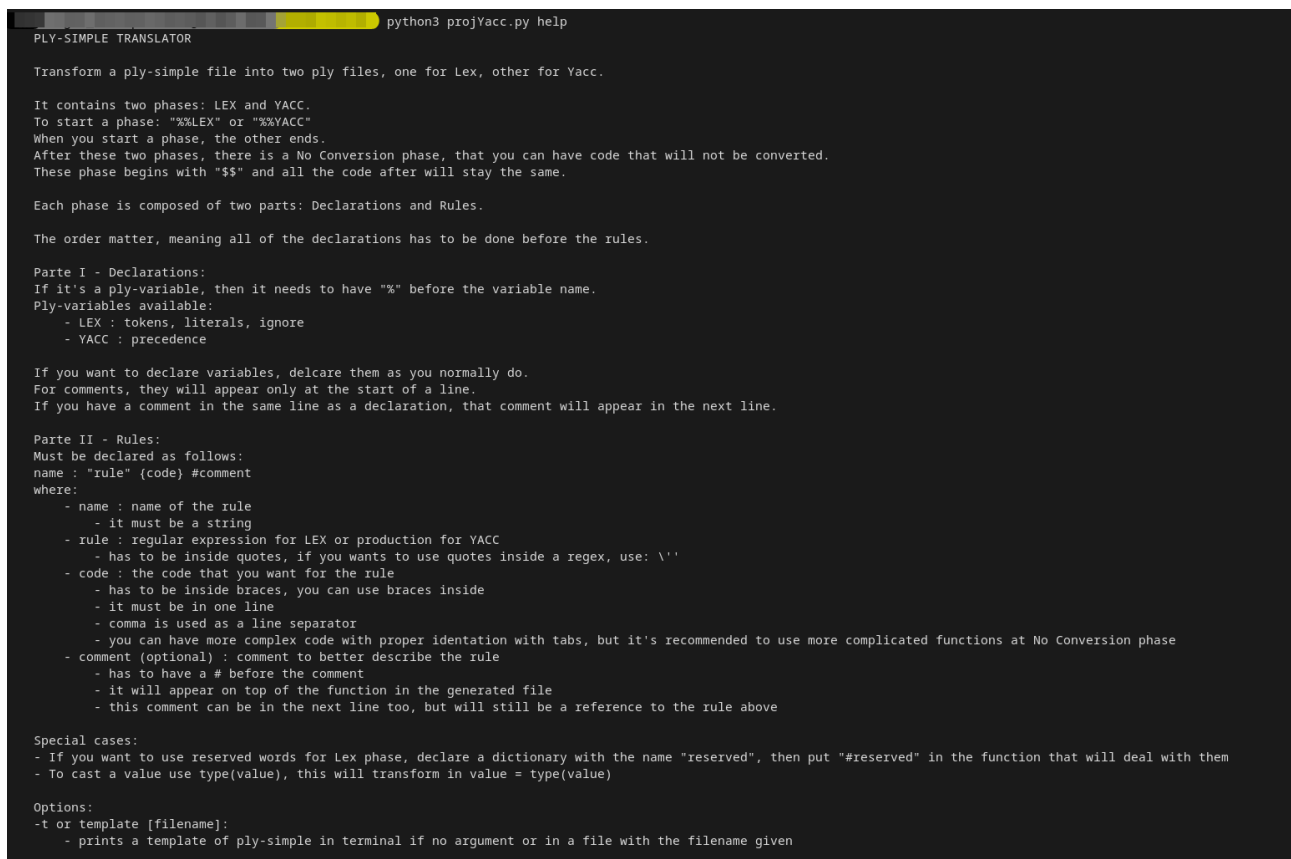
- Para identificar quando o código para a rule não se encontra dentro de chavetas ({}).
- **Output:** "Info : The code '{value}' for function '{name}' is invalid. The code has to be inside braces " (value -> the first token identified that was not inside the delimiters, name -> rule name)

5 Options

De modo a facilitar a utilização da aplicação, foram adicionadas duas opções:

5.1 Help

O utilizador ao passar como argumento "help" ou "-h", irá ser presente com um pequeno texto que serve de resumo para explicar o tradutor. Vejamos abaixo essa funcionalidade.



```
python3 projYacc.py help

PLY-SIMPLE TRANSLATOR

Transform a ply-simple file into two ply files, one for Lex, other for Yacc.

It contains two phases: LEX and YACC.
To start a phase: "%LEX" or "%YACC"
When you start a phase, the other ends.
After these two phases, there is a No Conversion phase, that you can have code that will not be converted.
These phase begins with "$$" and all the code after will stay the same.

Each phase is composed of two parts: Declarations and Rules.

The order matter, meaning all of the declarations has to be done before the rules.

Parte I - Declarations:
If it's a ply-variable, then it needs to have "%" before the variable name.
Ply-variables available:
- LEX : tokens, literals, ignore
- YACC : precedence

If you want to declare variables, declare them as you normally do.
For comments, they will appear only at the start of a line.
If you have a comment in the same line as a declaration, that comment will appear in the next line.

Parte II - Rules:
Must be declared as follows:
name : "rule" (code) #comment
where:
- name : name of the rule
  - it must be a string
- rule : regular expression for LEX or production for YACC
  - has to be inside quotes, if you wants to use quotes inside a regex, use: \'
- code : the code that you want for the rule
  - has to be inside braces, you can use braces inside
  - it must be in one line
  - comma is used as a line separator
  - you can have more complex code with proper idention with tabs, but it's recommended to use more complicated functions at No Conversion phase
- comment (optional) : comment to better describe the rule
  - has to have a # before the comment
  - it will appear on top of the function in the generated file
  - this comment can be in the next line too, but will still be a reference to the rule above

Special cases:
- If you want to use reserved words for Lex phase, declare a dictionary with the name "reserved", then put "#reserved" in the function that will deal with them
- To cast a value use type(value), this will transform in value = type(value)

Options:
-t or template [filename]:
  - prints a template of ply-simple in terminal if no argument or in a file with the filename given
```

Figure 2: Option help

5.2 Template

O utilizador ao passar como argumento "template" ou "-t", irá ser presente um pequeno template de modo a facilitar o utilizador a começar um ficheiro ply-simple. Podendo também passar como argumento um nome de um ficheiro para ser criado com esse conteúdo.

```
python3 projYacc.py -t
%%LEX

%literals = ""
%ignore = ""
%tokens = []

#Regular Expressions
RULE : "regex" {code} # comment

%% YACC

%precedence = []

# Grammar
RULE : "production" {code}

$$

# No Conversion Phase

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

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

Figure 3: Option template

```
python3 projYacc.py -t filename.plysimple
filename.plysimple was sucessfully generated
cat filename.plysimple
%%LEX

%literals = ""
%ignore = ""
%tokens = []

#Regular Expressions
RULE : "regex" {code} # comment

%% YACC

%precedence = []

# Grammar
RULE : "production" {code}

$$

# No Conversion Phase

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

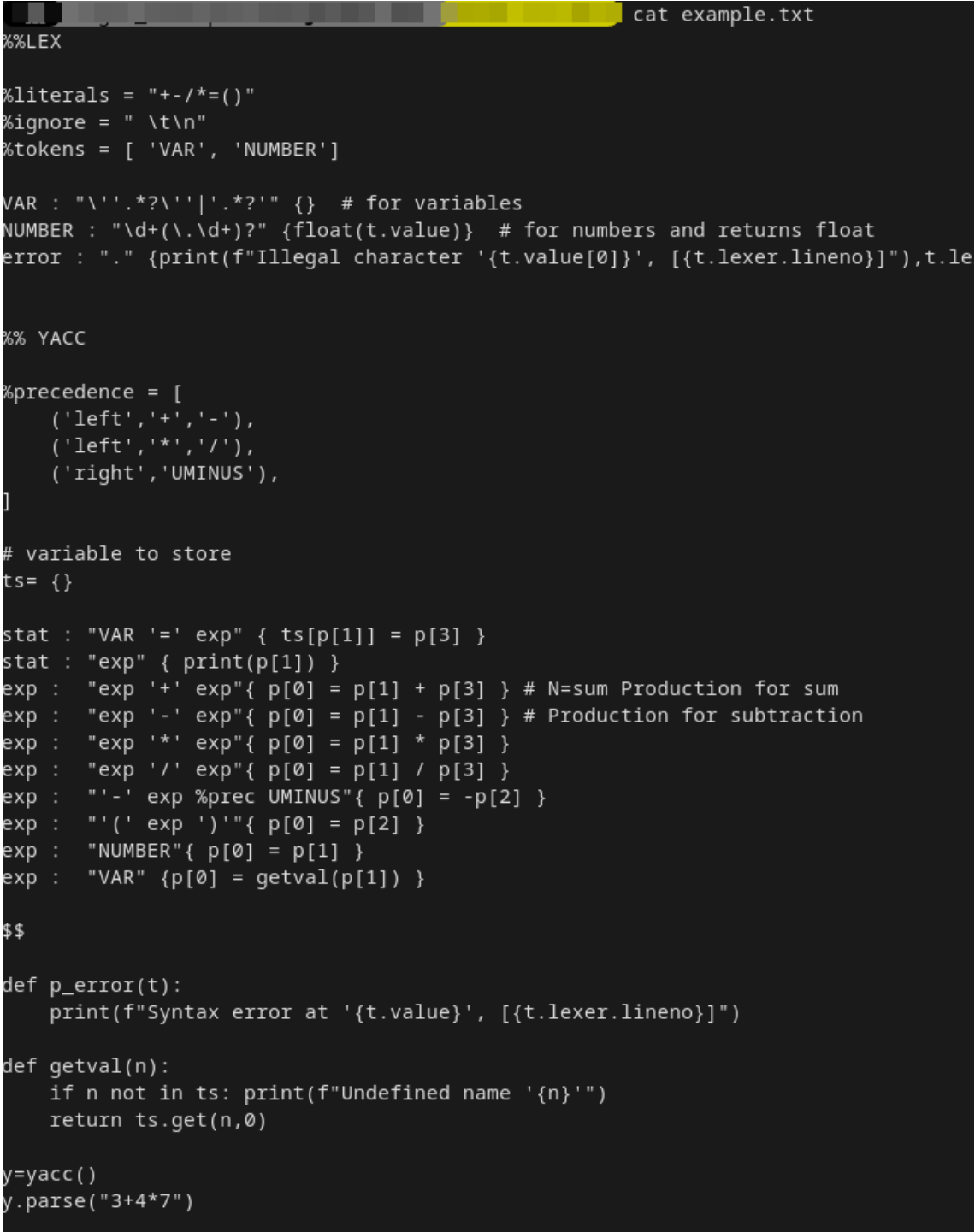
y=yacc()
y.parse("3+4*7")
```

Figure 4: Option template com um nome

6 Testes

Vejamos agora um ficheiro ply-simple a ser traduzido para dois ficheiros PLY, Lex e Yacc.

6.1 Ficheiro Example

A terminal window with a dark background and a yellow title bar. The title bar contains the text "cat example.txt". The terminal displays the content of a file named "example.txt", which is a PLY (Python Lex and Yacc) program. The program is divided into two sections: "%%LEX" and "%% YACC". The LEX section defines literals, ignores whitespace, and defines tokens for variables and numbers. The YACC section defines precedence for operators, a stack variable 'ts', and several grammar rules for expressions. At the bottom, there are Python functions for error handling and getting variable values, followed by the initialization of the yacc parser and the parsing of the expression "3+4*7".

```
cat example.txt
%%LEX

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

VAR : "\'\'.*?\'\'|\'.*?\'" {} # for variables
NUMBER : "\d+(\.\d+)?" {float(t.value)} # for numbers and returns float
error : "." {print(f"Illegal character '{t.value[0]}'", [{t.lexer.lineno}]),t.le

%% YACC

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

# variable to store
ts= {}

stat : "VAR '=' exp" { ts[p[1]] = p[3] }
stat : "exp" { print(p[1]) }
exp : "exp '+' exp" { p[0] = p[1] + p[3] } # N=sum Production for sum
exp : "exp '-' exp" { p[0] = p[1] - p[3] } # Production for subtraction
exp : "exp '*' exp" { p[0] = p[1] * p[3] }
exp : "exp '/' exp" { p[0] = p[1] / p[3] }
exp : "'-' exp %prec UMINUS" { p[0] = -p[2] }
exp : "'(' exp ')'" { p[0] = p[2] }
exp : "NUMBER" { p[0] = p[1] }
exp : "VAR" { p[0] = getval(p[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")
```

Figure 5: Ficheiro ply-simple "example"

```
python3 projYacc.py example.txt  
example_lex.py was successfully generated  
example_yacc.py was successfully generated
```

Figure 6: Execução do tradutor com o ficheiro "example"

```
cat example_lex.py  
import ply.lex as lex  
  
literals = "+-/*=()"  
  
t_ignore = " \t\n"  
  
tokens = ['VAR', 'NUMBER']  
  
# for variables  
def t_VAR(t):  
    r"\".*?\\"'|'.*?'"  
    return t  
  
# for numbers and returns float  
def t_NUMBER(t):  
    r"\d+(\.\d+)?"  
    t.value = float(t.value)  
    return t  
  
def t_error(t):  
    print(f"Illegal character '{t.value[0]}'", [{t.lexer.lineno}])  
    t.lexer.skip(1)  
lex.lex()
```

Figure 7: Ficheiro gerado para o Lex

```
cat example_yacc.py
from ply.yacc import *
from example_lex import tokens, literals

# variable to store
ts = {}
precedence = (('left', '+', '-'), ('left', '*', '/'), ('right', 'UMINUS'))

def p_stat_0(p):
    "stat : VAR '=' exp"
    ts[p[1]] = p[3]

def p_stat_1(p):
    "stat : exp"
    print(p[1])

# Production for sum
def p_exp_sum(p):
    "exp : exp '+' exp"
    p[0] = p[1] + p[3]

# Production for subtraction
def p_exp_0(p):
    "exp : exp '-' exp"
    p[0] = p[1] - p[3]

def p_exp_1(p):
    "exp : exp '*' exp"
    p[0] = p[1] * p[3]

def p_exp_2(p):
    "exp : exp '/' exp"
    p[0] = p[1] / p[3]

def p_exp_3(p):
    "exp : '-' exp %prec UMINUS"
    p[0] = -p[2]

def p_exp_4(p):
    "exp : '(' exp ')'"
    p[0] = p[2]

def p_exp_5(p):
```

```
def p_exp_5(p):
    "exp : NUMBER"
    p[0] = p[1]

def p_exp_6(p):
    "exp : VAR"
    p[0] = getval(p[1])

def p_error(t):
    print(f"Syntax error at '{t.value}', [{t.lexeme.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")
```

Figure 8: Ficheiro gerado para o Yacc

```
python example_yacc.py
Generating LALR tables
31.0
```

Figure 9: Execução do ficheiro Yacc

6.2 Ficheiro Projeto

Para testar o nosso tradutor num ficheiro de maior dimensão, decidimos converter manualmente o nosso projeto para Ply-Simple para efeitos de teste, voltar a converter para Ply.

Para tal, temos o seguinte ficheiro ply-simple:

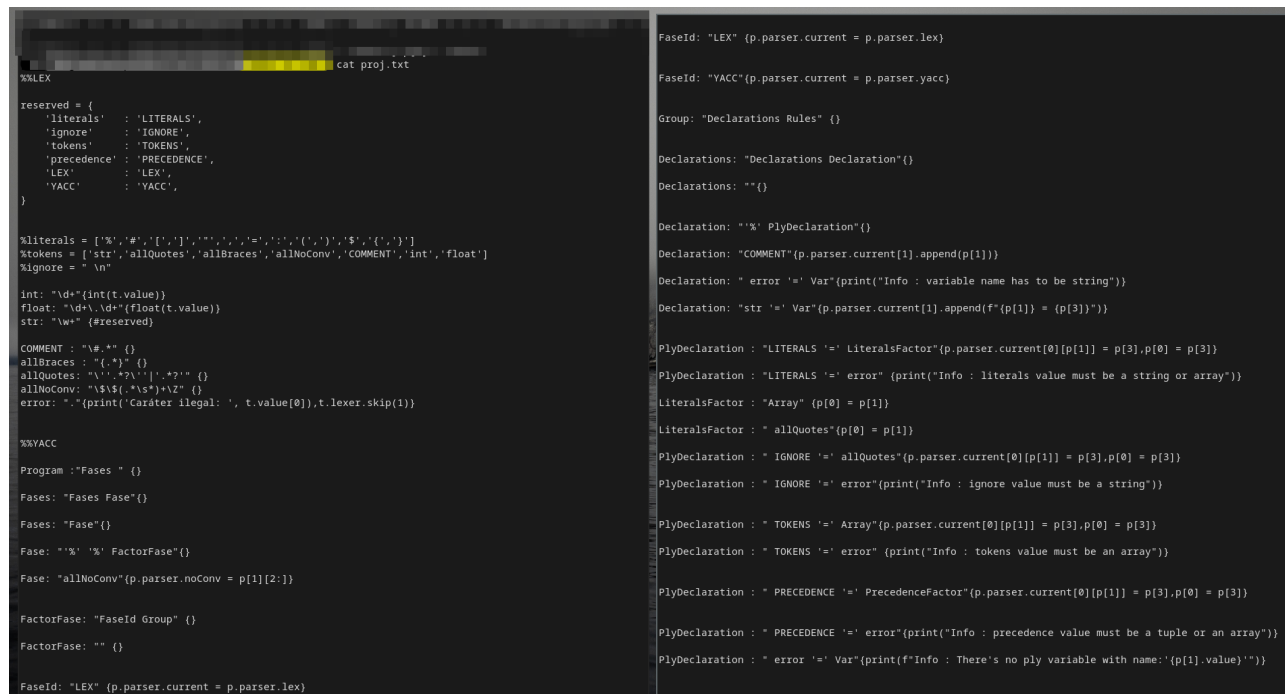


Figure 10: Primeira parte do ficheiro ply-simple do projeto

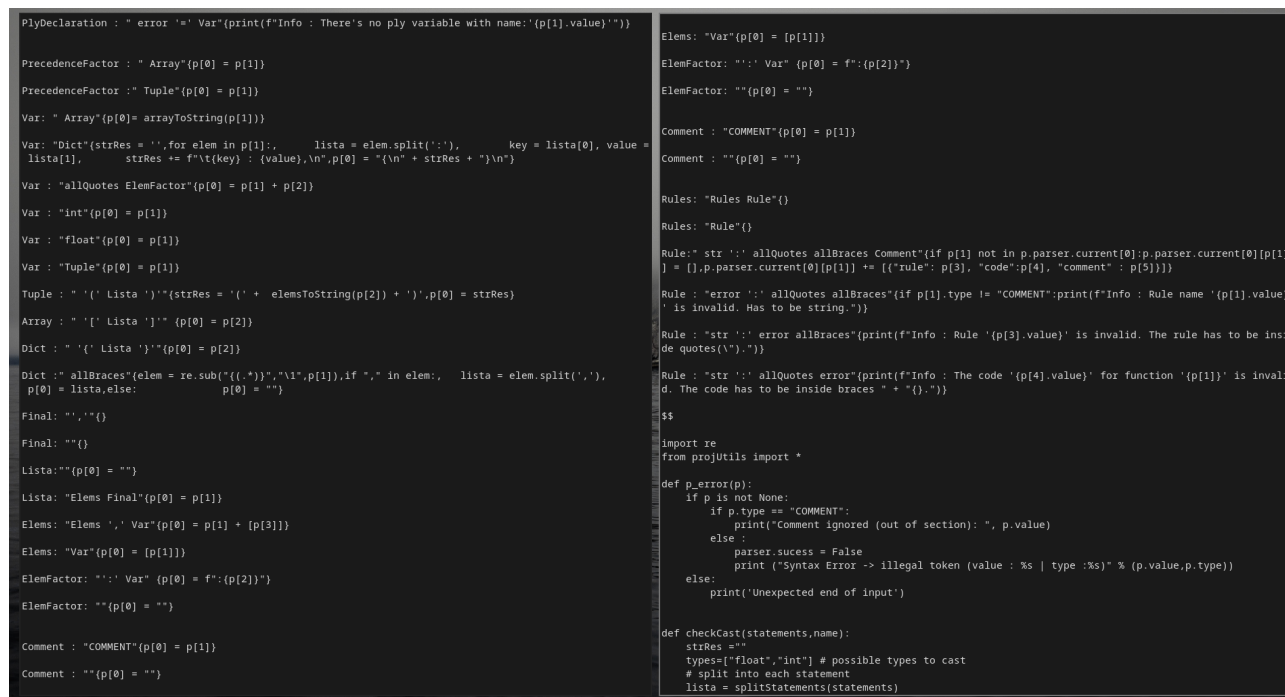


Figure 11: Segunda parte do ficheiro ply-simple do projeto

O restante ficheiro mantém-se inalterado, pois é a parte de No Conversion.

Os ficheiros foram corretamente gerados, vejamos por isso, a seguinte sequência de execuções.

```
python3 projYacc.py proj.txt
Generating LALR tables
proj_lex.py was successfully generated
proj_yacc.py was successfully generated
python3 proj_yacc.py example.txt
Generating LALR tables
example_lex.py was successfully generated
example_yacc.py was successfully generated
python3 example_yacc.py
Generating LALR tables
31.0
```

Figure 12: Sequência de execuções para o ficheiro Projeto

7 Conclusão

Em conclusão, acreditamos que tivemos um bom proveito do trabalho. Conseguimos alcançar todos os objetivos propostos, como ainda adicionamos algumas funcionalidades extra como a ajuda e o template. Este projeto ajudou a desenvolver a nossa capacidade de desenvolvimento de gramáticas, expressões regulares e tradutores.

Como possível trabalho futuro, achamos que seria uma mais-valia a implementação de suporte para código multi-linha para as regras.