# PL2025 - semana 4: 2025-02-28

## Sinopsis

- O que é um analisador léxico?
- Criação de um tokenizer com expressões regulares;
- Introdução ao `ply.lex`: primeiros exemplos;
- `ply.lex` como tokenizer e a ser usado noutro programa.

---

## Analisador Léxico

Um analisador léxico é usado para converter uma string, uma sequência de carateres, numa sequência de tokens.

Por exemplo, se o input fosse:

```
x = 3 + 42 * (s - t)
```

Um analisador iria dividir aquela string nos seguintes tokens:

```
'x', '=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

Para podermos referenciá-los e trabalhar mais facilmente com os tokens podemos atribuir-lhes nomes. Por exemplo:

```
'ID','EQUALS','NUMBER','PLUS','NUMBER','TIMES',
'LPAREN','ID','MINUS','ID','RPAREN'
```

Podemos distinguir 3 tipos de tokens:

- Sinais (constituídos apenas por um caráter): '+', '*';
- Palavras reservados (constantes do tipo string): 'SELECT', 'FOR', 'IF';
- Símbolos terminais variáveis: 'ID', 'NUMBER'.

No caso dos símbolos terminais variáveis vamos ter de prever o armazenamento do seu valor. Assim, um token transforma-se num par: `(token_id, token_value)`. Por exemplo:

```
('ID','x'), ('EQUALS','='), ('NUMBER','3'),
('PLUS','+'), ('NUMBER','42), ('TIMES','*'),
('LPAREN','('), ('ID','s'), ('MINUS','-'),
('ID','t'), ('RPAREN',')'
```

# Criação de um tokenizer com expressões regulares

## Desafio: vamos criar um programa para uma linguagem simples

Exemplos de frases:

```
?a
?b
c = b * a / 2
!c
```

Quais os símbolos terminais desta linguagem?

```
   ??? Enumerar os símbolos
```

Temos de reconhecer estes símbolos em simultâneo com um único autómato. Como fazê-lo? Colocando de outra forma, como colocar o reconhecimento de todos estes símbolos numa única expressão regular?

```
tokens = r"..."
```

Organizando melhor e atribuindo identificadores aos símbolos:

```
token_specification = [
        (token_id,   token_expreg),
        ('NEWLINE', r'\n'),           # Line endings
        ('SKIP',    r'[ \t]+'),       # Skip over spaces and tabs
        ('ERRO',    r'.'),            # Any other character
    ]
```

Como juntar tudo numa única expressão regular?

```
tok_regex = '|'.join([f'(?P<{id}>{expreg})' for (id, expreg) in
token_specification])
```

Depois destas decisões a construção do tokenizer final torna-se fácil:

```
import sys
import re

def tokenize(code):
```

```
    token_specification = [
        ('NUM',   r'\d+'),            # Integer or decimal number
        ('ATRIB',   r'='),            # Assignment operator
        ('ID',       r'[_A-Za-z]\w*'), # Identifiers
        ('OP',       r'[+\-*/]'),      # Arithmetic operators
        ('NEWLINE',  r'\n'),           # Line endings
        ('SKIP',     r'[ \t]+'),       # Skip over spaces and tabs
        ('ERRO', r'.'),                # Any other character
    ]
    tok_regex = '|'.join([f'(?P<{id}>{expreg})' for (id, expreg) in
token_specification])
    reconhecidos = []
    linha = 1
    mo = re.finditer(tok_regex, code)
    for m in mo:
        dic = m.groupdict()
        if dic['NUM'] is not None:
            t = ("NUM", int(dic['NUM']), linha, m.span())
        elif dic['OP']:
            t = ("OP", dic['OP'], linha, m.span())
        elif dic['ATRIB'] is not None:
            t = ("ATRIB", "=", linha, m.span())
        elif dic['ID'] is not None:
            t = ("ID", dic['ID'], linha, m.span())
        elif dic['SKIP'] is not None:
            pass
        else:
            t = ("ERRO", m.group(), linha, m.span())
        if not dic['SKIP']: reconhecidos.append(t)

    return reconhecidos


for linha in sys.stdin:
    for tok in tokenize(linha):
        print(tok)
```

## Desafio: criar o tokenizer para a linguagem de listas

Considera os seguintes exemplos de listas de uma linguagem tipo Haskell ou Python:

```
[]
[1]
[1,2,3,4]
[[1,2],3,[4,5]]
```

Quais os tokens desta linguagem:

```
???
```

- Mostrar o gerador: definição dos tokens e processo de geração

## Introdução ao `ply.lex`

- O `ply` resulta de uma implementação em Python das conhecidas ferramentas do SO Unix `lex` e `yacc`.

Eis um primeiro exemplo:

```
# ------------------------------------------------------------
# calclex.py
#
# tokenizer for a simple expression evaluator for
# numbers and +,-,*,/
# ------------------------------------------------------------
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces and tabs)
t_ignore  = ' \t'

# Error handling rule
```

```
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Build the lexer
lexer = lex.lex()
```

1. Colocar o exemplo em funcionamento e mostrar as várias funcionalidades;
2. Para usarmos este tokenizer precisamos de passar o input ao seu método `input()`;

Exemplo de input:

```
data = '''
3 + 4 * 10
  + -20 *2
'''
```

Passando ao lexer:

```
lexer.input(data)
```

3. A seguir, as invocações do método `tok()` retornarão os sucessivos tokens até se atingir o fim do input onde será retornado o valor `None`:

Obtendo os tokens:

```
for tok in lexer:
    print(tok)
```

Em execução:

```
$ python example.py
LexToken(NUMBER,3,2,1)
LexToken(PLUS,'+',2,3)
LexToken(NUMBER,4,2,5)
LexToken(TIMES,'*',2,7)
LexToken(NUMBER,10,2,10)
LexToken(PLUS,'+',3,14)
LexToken(MINUS,'-',3,16)
LexToken(NUMBER,20,3,18)
LexToken(TIMES,'*',3,20)
LexToken(NUMBER,2,3,21)
```

4. Os tokens retornados por `lexer.token()` são instâncias de `LexToken`;

5. Este objeto tem os atributos: `type`, `value`, `lineno`, e `lexpos`;

```
for tok in lexer:
    print(tok.type, tok.value, tok.lineno, tok.lexpos)
```

6. `lexpos` é o índice do token na string de input;

### Lista de tokens

Todos os analisadores léxicos têm de definir a sua lista de tokens. Esta lista é usada para uma série de validações e também para comunicar com o analisador sintático no contexto de construção de um compilador.

No exemplo anterior, esta especificação está feita assim:

```
tokens = (
   'NUMBER',
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
)
```

### Especificação de cada token

1. Cada token deve ser especificado por uma expressão regular numa declaração;
2. Cada declaração deve ter um prefixo: `t_`;

```
t_PLUS = r'\+'
```

3. O nome dado a seguir ao prefixo `t_` deve constar da lista de tokens definida inicialmente;
4. Se precisarmos de realizar alguma ação no reconhecimento de um token devemos substituir a sua declaração pela definição de uma função;

Exemplo: no reconhecimento de um número, o lexer retorna o seu valor inteiro

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

5. Quando se usa uma função para especificar o token, a expressão regular é definida na string de documentação;
6. A função tem um único argumento que é uma instância de `LexToken`;
7. Por omissão, `type` fica com o nome da função reirando o prefixo `t_`;
8. A função pode alterar qualquer um dos atributos do `LexToken` passado como argumento mas, neste caso, deverá retornar o token;
9. Se a função não retornar nenhum valor, o token é descartado e a leitura prosseguirá com o próximo token;
10. Esta ferramenta usa o módulo `re`;
11. Os padrões são compildados com a flag `re.VERBOSE`, o que significa que o espaço normal é ignorado e os comentários são permitidos;
12. Se precisas de espaço num padrão usa `\s` ou `\ ` ;
13. Se precisas de `#` num padrão usa `\#` ou `[#]`;

### Prioridade/precedência das expressões regulares

Quando a ferramenta constrói a expressão principal usa o seguinte algoritmo:

- Tokens definidos por funções são colocados em primeiro lugar pela ordem em que aparecem definidos;
- Tokens definidos por strings são colocados a seguir por ordem decrescente de tamanho.

Por exemplo, dois tokens definidos como `=` e `==` têm de ser colocados pela ordem `==` e `=`.

### Palavras reservadas

Para as palavras reservadas pode adotar-se uma técnica que minimiza a confusão da prioridade e permite escrever menos código:

```
reserved = {
   'if' : 'IF',
   'then' : 'THEN',
   'else' : 'ELSE',
   'while' : 'WHILE',
   ...
}

tokens = ['LPAREN','RPAREN',...,'ID'] + list(reserved.values())

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'ID')    # Check for reserved words
    return t
```

Deve-se evitar escrever regras para palavras reservadas individuais:

```
t_FOR   = r'for'
t_PRINT = r'print'
```

Estas regras estariam ativas para identificadores como `"forget"` ou `"printed"`.

### Valores dos tokens

1. Os tokens retornados têm um valor guardado no atributo `value`;
2. Por omissão, o valor é o exto que fez match com a expressão regular;
3. No entanto, na definição de uma função para o token podemos colocar neste atributo o que quisermos, até uma estrutura de dados;

```
def t_ID(t):
    ...
    # Look up symbol table information and return a tuple
    t.value = (t.value, symbol_lookup(t.value))
    ...
    return t
```

4. Não é recomendável armazenar informação noutros atributos pois o lexer só irá passar o atributo `value` ao `yacc`;

### Ignorar certos tokens

Para ignorar tokens como comentários basta definir uma regra que não retorna nada:

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # No return value. Token discarded
```

Alternativamente, pode-se usar o prefixo `ignore_` na declaração do token para forçar este a ser ignorado:

```
t_ignore_COMMENT = r'\#.*'
```

**Tem sempre atenção à ordem quando usares declarações e não funções.**

### Números de linha e posições

Por omissão, o lexer não tem o conceito de linha ou número de linha. Para alterar isto, é preciso especificar uma regra especial:

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

Esta regra apenas altera a variável do estado interno do lexer `lineno`, como nada é retornado o token é ignorado.

### Tokens constituídos apenas por um caráter

```
literals = [ '+','-','*','/' ]
```

ou:

```
literals = "+-*/"
```

1. Em termos de prioridade, este tipo de tokens são colocados no fim;
2. Quando um token destes é retornado o seu tipo e o seu valor são o próprio caráter;
3. Podemos criar função de reconhecimento para estes tokens, mas temos de colocar lá o tipo explicitamente;

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = 'CA' # Chaveta a Abrir
    return t

def t_rbrace(t):
    r'\}'
    t.type = 'CF' # Chaveta a Fechar
    return t
```

### Tratamento de erros

1. A função `t_error()` é usada sempre que um caráter inesperado é detetado;
2. Quando isto acontece `t.value` contem o resto da string que ainda não foi analisada;

```
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

Neste caso, o caráter inesperado é colocado na saída e o processo de análise avança-o: `t.lexer.skip(1)`.

### EOF: Fim de ficheiro ou fim de input

1. A função `t_eof()` é chamada quando se atinge o fim do input;

2. Podemos usá-la para ir buscar mais input ou para produzir dados estatísticos da análise feita até ao momento.

```
def t_eof(t):
    # Get more input (Example)
    more = input('... ')
    if more:
        self.lexer.input(more)
        return self.lexer.token()
    return None
```

### Construção e utilização do analisador léxico

1. Usa-se a função `lex.lex()` para construir o analisador léxico:

```
lexer = lex.lex()
```

2. Dois métodos ficam disponíveis para controlar o analisador léxico:
    - `lexer.input(data)`: faz reset ao lexer e carrega uma nova string para ser analisada;
    - `lexer.token()`: Retorn o próximo token: uma instância de `LexToken` em caso de sucesso e `None` se o fim do input foi atingido.

### O decorador: `@TOKEN`

Nalgumas aplicações poderemos querer definir tokens mais complexos:

```
digit            = r'([0-9])'
nondigit         = r'([_A-Za-z])'
identifier       = r'(' + nondigit + r'(' + digit + r'|' + nondigit +
r')*)'

def t_ID(t):
    # want docstring to be identifier above. ?????
    ...
```

Nestes casos, podemos usar o decorador `@TOKEN`:

```
from ply.lex import TOKEN

@TOKEN(identifier)
def t_ID(t):
    ...
```

Esta metodologia poderá ser usada em todas as funções como alternativa às docstrings.

## Debug

```
lexer = lex.lex(debug=True)
```