TRABALHO PRÁTICO II - LOGO

Licenciatura em Engenharia de Sistemas Informáticos

Processamento de Linguagem

Autores

João Azevedo Nº18845

João Rodrigues Nº19431

Carlos Santos Nº19432

Professor: Alberto Simões

Data: 19/01/2021

Resumo

O Logo é uma linguagem de programação educativa, com origem em 1967. Pretende-se implementar um interpretador um interpretador básico, capaz de simular o maior número possível de comandos da linguagem original. Nesta linguagem o programador controla uma tartaruga, que se vai mexendo através do espaço, e desenhando linhas por onde passa. O programa será um ficheiro de texto com uma ou mais linhas, e deve implementar, pelo menos, os seguintes comandos:

- O comando **fd** ou **foward** – move a tartaruga n pixies em frente: **fd** 10;
- O comando **bk** ou **back** – move a tartaruga, para trás: **back** 20;
- O comando **lt** ou **left** roda a tartaruga, para a esquerda, n graus: **left** 90;
- O comando **rt** ou **right** roda a tartaruga, para a direita, n graus: **right** 180;
- O comando **setpos**, **setxy**, **setx**, **sety** permite definir uma posição para a qual a tartaruga se deve movimentar: **setpos** [100 100], ou **setxy** 100 100 ou um eixo apenas, mantendo o outro eixo: **setx** -100;
- O comando gome move a tartaruga para o ponto inicial (0,0), e roda-a para a orientação original: **home**;
- Os comandos **pendown** e **pwnup**, respetivamente abreviados por **pd** e **pu**, permitem alternar entre o modo de desenho e o modo de movimentação livre;
- O comando **setpencolor** permite alterar a cor das linhas para os comandos que se seguem: **setpencolor** [99 0 0];
- O comando **make** permite definir o valor de uma variável: **make“varname**20;
- O comando **if** e **ifelse** permitem definir estruturas condicionais;
- O comando **repeat** repete um conjunto de comandos;
- O comando **while** permite definir ciclos;
- O comando **to** permite criar funções.

Main

O ficheiro main.py recebe da consola o nome de um ficheiro, verifica se ele é válido e usando as funções do ficheiro **svg** lê-lo, executa o **parser**, o **lexer** e por fim realiza a função **drawAll**.

Svg

O ficheiro **svg.py** é o *canvas* do programa. Neste ficheiro constam as funções que leem, abrem e escrevem em ficheiros e assim produzem o resultado final.

```
def drawLine(pos1, pos2, color):
    global text
    if pos2[0] <= 200 and pos2[1] <= 200 and pos2[0] >= 0 and pos2[1] >= 0 and pos1[0] <= 200 and pos1[1] <= 200 and pos1[0] >= 0 and pos1[1] >= 0:
        line = f'<line x1="{pos1[0]}" y1="{pos1[1]}" \
                       x2="{pos2[0]}" y2="{pos2[1]}" \
                       style="stroke: rgb({color[0]}, {color[1]}, {color[2]}); stroke-width: 1px"/>\n'
        text += line
    else:
        print("Drawing is out of limits", file=sys.stderr)
        exit(1)

def drawAll(file_name):
    global text
    open(file_name, 'w+').close()
    writeFile(file_name, '<svg viewBox="0 0 200.00 200.00" xmlns="http://www.w3.org/2000/svg">\n')
    writeFile(file_name, text)
    writeFile(file_name, '</svg>')
```

A função **drawLine** recebe um ponto origem, um ponto destino e uma cor e irá guardar a nova linha em HTML guardada em memória na variável **text**. Esta verifica também se a nova linha se encontra dentro dos limites de desenho, caso não se verifique emite uma mensagem de erro e para o programa.

A função **drawAll** irá escrever no ficheiro de resultado todo o texto HTML que se encontra guardado em memória e finalizar o programa.

Expressões Regulares

Esta foi a expressão regular usada de forma a ler apenas os comandos corretos:

```
def t_COMMAND(self, t):
    r"""(fd|forward|bk|back|lt|left|rt|right|setpos|setx(y)?|sety|home|pd|pendown|pu|penup|
    setpencolor|make|if(else)?|repeat|while|to|end)\b"""
    t.type = t.value.upper()
    return t
```

- Nesta expressão são lidos apenas os comandos pretendidos e suas abreviaturas, como o caso do **foward** ou **fd**.

Esta foi a expressão regular usada de forma a ler apenas os nomes das funções:

```
def t_NAMETO(self, t):
    r"""[a-zA-Z][A-Za-z0-9]*"""
    return t
```

- Nesta expressão são lidos apenas os nomes das funções.

Esta foi a expressão regular usada de forma a ler apenas os números:

```
def t_NUM(self, t):
    r"""[+-]?[0-9]+(\.[0-9]+)?"""
    t.value = float(t.value)
    return t
```

- Nesta expressão regular é lido qualquer número positivo ou negativo e com ou sem vírgulas, isto é -3 ou -3,14.

Esta foi a expressão regular usada de forma a ler apenas os nomes e os respetivos valores das varáveis:

```
def t_VARNAME(self, t):
    r"""[\"][A-Za-z][A-Za-z0-9]*"""
    return t
```

- Nesta expressão regular são lidas as variáveis que contenham o sinal **“**antes do seu nome e o respetivo valor associado, como por exemplo “numero 10.

Esta foi a expressão regular usada de forma a ler apenas os nomes das varáveis que já possuem valores associados:

```
def t_VARUSE(self, t):
    r"""[\:][A-Za-z][A-Za-z0-9]*"""
    return t
```

- Nesta expressão regular são lidas as variáveis que já possuam valores ou seja variáveis com “**:**” antes do nome da variável, um exemplo prático seria se já existir uma variável (“numero 10) e no código possuir “:numero” o valor associado a essa variável (10) irá ser usado.

Esta foi a expressão regular usada de forma a ler operadores lógicos:

```
def t_LOGIC(self, t):
    r"""<(=)?|>(=)?|==|!="""
    return t
```

- Nesta expressão regular são lidos operadores lógicos que possam ser usados como comparadores (==, >, <, <=, >=, !=).

Testes

À medida que o projeto estava a ser trabalhado era necessário fazer testes de forma a perceber se o **Lexer** estava a reconhecer as palavras no seu devido sítio assim sendo foi feito um teste a título de exemplo.

O Input foi o seguinte:

```
to square :lenght
    repeat 4 [ fd :lenght rt 90]
end
repeat 36 [ square 60 rt 10 ]
```

Com base no Input o Output foi:

```
C:\Users\jpint\AppData\Local\Programs\P
LexToken(TO,'to',1,6)
LexToken(NAMETO,'square',1,9)
LexToken(VARUSE,':lenght',1,16)
LexToken(REPEAT,'repeat',1,32)
LexToken(NUM,4.0,1,39)
LexToken(FD,'fd',1,43)
LexToken(VARUSE,':lenght',1,46)
LexToken(RT,'rt',1,54)
LexToken(NUM,90.0,1,57)
LexToken(END,'end',1,65)
LexToken(REPEAT,'repeat',1,73)
LexToken(NUM,36.0,1,80)
LexToken(NAMETO,'square',1,85)
LexToken(NUM,60.0,1,92)
LexToken(RT,'rt',1,95)
LexToken(NUM,10.0,1,98)

Process finished with exit code 0
```

Gramática

Na classe Parser são mantidas em memória as variáveis, funções e outras definições escolhidas pelo utilizador que são usadas para obter o resultado final.

```
class Parser:
    tokens = Lexer.tokens

    def __init__(self):
        self.parser = None
        self.lexer = None
        self.vars = {}
        self.functions = {}
        self.color = (0, 0, 0)
        self.pos = (100, 100)
        self.ang = 90
        self.draw_status = True
```

A função Parse que é executada no main recebe todo o conteúdo inserido pelo utilizador e também todos os **tokens** reconhecidos pelo **Lexer**, depois verifica os comandos e por fim executa-os com a função **exec**.

```
def Parse(self, content, **kwargs):
    self.lexer = Lexer()
    self.lexer.Build(content, **kwargs)
    self.parser = yacc.yacc(module=self, **kwargs)
    program = self.parser.parse(lexer=self.lexer.lexer)
    self.exec(program)

def exec(self, program):
    for command in program:
        command.run(self)
```

No Parser.py tem um conjunto de analisadores sintáticos de cada comando.

```
def p_command0(self, p):
    """ command  :  FORWARD value
                 |  FD value """
    args = {'distance': p[2]}
    p[0] = Command("forward", args)
```

- Na função p_command0, o comando **forward** ou **fd** é precedido de um valor, esse valor corresponde à distância que irá andar para a frente.

```
def p_command1(self, p):
    """ command  :  BACK value
                 |  BK value """
    args = {'distance': p[2]}
    p[0] = Command("back", args)
```

- Na função p_command1, comando **back** ou **bk** é precedido de um valor, esse valor corresponde à distância que irá andar para trás.

```
def p_command2(self, p):
    """ command  :  LT value
                 |  LEFT value """
    args = {'degrees': p[2]}
    p[0] = Command("left", args)
```

- Na função p_command2, comando **left** ou **lt** é precedido de um valor, esse valor corresponde aos graus que pretende virar no sentido anti-horário.

```
def p_command3(self, p):
    """ command  :  RT value
                 |  RIGHT value """
    args = {'degrees': p[2]}
    p[0] = Command("right", args)
```

- Na função p_command3, o comando **right** ou **rt** é precedido de um valor, esse valor corresponde aos graus que pretende virar no sentido horário.

```
def p_command4(self, p):
    """ command  :  SETPOS '[' value value ']' """

    args = {'new_pos': (p[3], p[4])}
    p[0] = Command("set_position", args)
```

```
def p_command5(self, p):
    """ command  :  SETXY  value value """

    args = {'new_pos': (p[2], p[3])}
    p[0] = Command("set_position", args)
```

- Na função p_command4 e p_command5, o comando **setpos** e **setxy** são precedidos de dois valores entre parênteses ([10 10]) e sem eles (10 10) respetivamente. Estes valores têm como função colocar o objeto nos pontos que se pretende.

```
def p_command6(self, p):
    """ command  :  SETX  value """

    args = {'new_x': p[2]}
    p[0] = Command("set_position", args)
```

```
def p_command7(self, p):
    """ command  :  SETY  value """

    args = {'new_y': p[2]}
    p[0] = Command("set_position", args)
```

- Na função p_command6 e p_command7, os comandos **setx** ou **sety** são precedidos de um valor, esse valor corresponde a posição x que se pretende alterar no caso do comando **setx** e a posição y no caso do comando **sety**.

```
def p_command8(self, p):
    """ command  :  HOME """
    args = {}
    p[0] = Command("set_position", args)
```

- Na função p_command8, o comando **home** tem como função voltar a posição inicial do desenho.

```
def p_command9(self, p):
    """ command  :  PD
                 |  PENDOWN """
    args = {'new_status': True}
    p[0] = Command("change_status", args)
```

```
def p_command10(self, p):
    """ command  :  PU
                 |  PENUP """
    args = {'new_status': False}
    p[0] = Command("change_status", args)
```

- Na função p_command9, o comando **pendown** ou **pd** tem a função de desenhar as instruções seguintes de forma a que se consiga ver o desenho feito.
- No caso da p_command10, o comando **penup** ou **pu** tem a função de desenhar as instruções seguintes de forma a que não se consiga ver o desenho feito.

```
def p_command11(self, p):
    """ command  :  SETPENCOLOR '[' value value value ']' """
    args = {'new_color': (p[3], p[4], p[5])}
    p[0] = Command("pen_color", args)
```

- Na função p_command11, o comando **setpencolor** é precedido de três valores entre parênteses [100 0 100], esses valores correspondem a uma cor que se pretenda ver no desenho, assim sendo até caso contrário as instruções seguintes serão desenhadas com essa mesma cor escolhida.

```
def p_command12(self, p):
    """ command  :  MAKE VARNAME value """
    args = {'var_name': p[2][1:], 'value': p[3]}
    p[0] = Command("make", args)
```

- Na função p_command12, o comando **make** é precedido de uma variável (varname) e de um valor, esse valor corresponde ao valor da variável que poderá ser usada em outras estruturas com por exemplo as estruturas condicionais, entre outras.

```
def p_command13(self, p):
    """ command  :  IF condition '[' program ']'"""
    args = {'condition': p[2], 'code': p[4]}
    p[0] = Command("if", args)
```

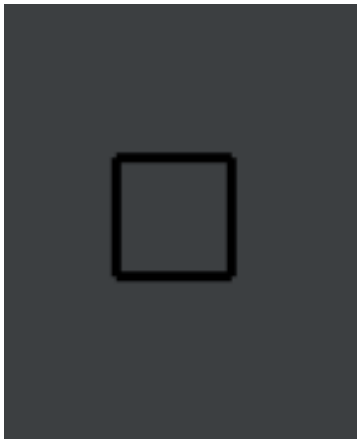
- Na função p_command13, o comando **if** é precedido de uma condição, e entre parênteses o código que se pretenda ser executado.

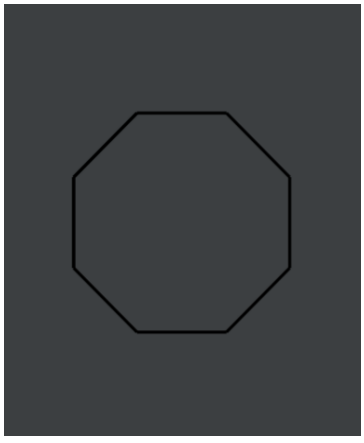
```
make "idade 10
if :idade > 1 [fd 50]
```

```
def p_command14(self, p):
    """ command  :  IFELSE condition '[' program ']' '[' program ']' """
    args = {'condition': p[2], 'code1': p[4], 'code2': p[7]}
    p[0] = Command("ifelse", args)
```

- Na função p_command14, o comando **ifelse** é precedido de uma condição, e entre os primeiros parênteses o código que se pretenda ser executado no caso de essa condição seja verdadeira e os segundos parênteses o código a executar caso a condição seja falsa.

```
make "altura 30
ifelse :altura < 10
[ repeat 10 [fd 30 right 45]]
[ repeat 20 [back 15 left 90]]
```

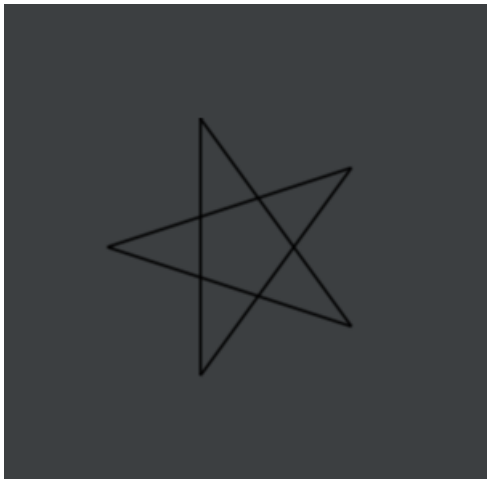




```
def p_command15(self, p):
    """ command  :  REPEAT value '[' program ']' """
    args = {'var': p[2], 'code': p[4]}
    p[0] = Command("repeat", args)
```

- Na função p_command15, o comando **repeat** é precedido de um valor, que representa o número de repetições que devem ser feitas ao código que esta entre parênteses.

```
repeat 5 [ fd 100 rt 144 ]
```



```
def p_command16(self, p):
    """ command  :  WHILE '[' condition ']' '[' program ']'
                 | WHILE condition '[' program ']' """
    if len(p) == 8:
        args = {'condition': p[3], 'code': p[6]}
    else:
        args = {'condition': p[2], 'code': p[4]}
    p[0] = Command("while", args)
```

- Na função p_command16, o comando **while** tem duas variantes, no caso da primeira a condição deve estar entre parênteses assim como o código a executar. Na segunda variante a condição não necessita de estar entre parênteses, no entanto o código a executar sim. Em ambas as situações o código só executa caso a condição seja verdadeira.

```
make "altura 10
while [:altura < 20][
repeat 4 [fd 15 rt 35]
make "altura :altura+4
]
```

```
def p_command17(self, p):
    """ command  :  TO NAMETO program END
                 |  TO NAMETO vars program END"""
    if len(p) == 6:
        args = {'nameto': p[2], 'vars': p[3], 'code': p[4]}
    else:
        args = {'nameto': p[2], 'code': p[3]}
    p[0] = Command('function', args)
```

- Na função p_command17, o comando **to** tem duas variantes, no caso da primeira possui o nome da função (nameto), o conjunto de código a ser executado (program) e o terminador de uma função (end). Na segunda possui o nome da função (nameto), os argumentos (vars) que a função necessita para ser executada, o conjunto de código a ser executado (program) e o terminador de uma função (end).

```
to tree :size
if :size < 5 [forward :size back :size]
if :size > 5 [
    forward :size/3
    left 30 tree :size*2/3 right 30
    forward :size/6
    right 25 tree :size/2 left 25
    forward :size/3
    right 25 tree :size/2 left 25
    forward :size/6
    back :size]
end

tree 65
```

Comandos

Foi criada uma classe Command.py, a função desta classe é armazenar os comandos e os seus respetivos argumentos (ex: **forward** 10; command ("**forward**", 10)).

Foi criado um dicionário que relaciona o comando a uma função, como é o caso do " **forward**: do_forward ", isto é o sempre que for chamado o comando **forward** ele irá fazer um conjunto de instruções da função "do_forward" e de forma igual com os restantes comandos.

```
class Command:
    dispatch_table = {
        "forward": do_forward,
        "back": do_back,
        "left": do_left,
        "right": do_right,
        "set_position": do_set_position,
        "change_status": do_change_status,
        "pen_color": do_pen_color,
        "make": do_make,
        "if": do_if,
        "ifelse": do_ifelse,
        "repeat": do_repeat,
        "while": do_while,
        "function": do_function,
        "call_function": do_call_function,
    }

    def __init__(self, name, args):
        self.name = name
        self.args = args
```

Na class Command consta uma tabela de nome dispatch_table que procura pela função que deve ser executada consoante o nome recebido da classe Parser.

```
def do_repeat(command, parser):
    count = 0
    var = parser.value(command.args['var'])
    code = command.args['code']

    while var > count:
        parser.exec(code)
        count += 1
```

```
def do_forward(command, parser):
    dist = parser.value(command.args['distance'])
    new_pos = (parser.pos[0]+dist*math.cos(math.radians(parser.ang)), parser.pos[1]-dist*math.sin(math.radians(parser.ang)))
    if parser.draw_status:
        svg.drawLine(parser.pos, new_pos, parser.color)
    parser.pos = new_pos
    return 0
```

Nos exemplos acima estão duas funções, na primeira função o comando **repeat** é executado. Na segunda calcula a nova posição do desenho, caso esteja em modo desenho irá guardar o desenho na classe **svg** e no final atualiza a posição atual na classe **Parser**.

Conclusão

O projeto foi bastante interessante e sobretudo exigente, o seu desenvolvimento potenciou conhecimento que nos será útil em projetos futuros. Houve algumas dificuldades na sua estruturação, no entanto foram anuladas com o decorrer do tempo.

Em suma, abordamos todos os tópicos pretendidos e conseguimos cumprir todos os objetivos propostos.