Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de
Processamento de Linguagens

Ano Letivo de 2021/2022

Trabalho Prático 2
Linguagem de Templates

Grupo 05
a93234 - Diogo Matos
a83630 - Duarte Serrão
a93208 - Vasco Oliveira

15 de maio de 2022

Índice

1 Introdução

Foi escolhido o enunciado Pandoc, hipótese 3.

1.1 Contextualização

O objetivo é escrever um programa que dado um ficheiro template e um ficheiro YAML, produza um "texto final". Ora, a nossa tarefa será definir a linguagem desses templates, tendo em inspiração linguagens de templates já existentes, como os templates Pandoc, dados como exemplo no enunciado.

Esta nova linguagem criada pelo grupo será baseada a partir do conceito do projeto *Jupyter*[2]. Tal como este notebook, a nossa linguagem terá secções com apenas texto, com código, ou com comentários que serão ignorados. A gramática em si, focar-se-á na segunda secção, ou seja, no código a ser processado.

1.2 Problemas a Focar

O grupo terá de inventar uma linguagem markdown a ser processada pela gramática, tal como desenvolver essa própria gramática.

2 Criação de uma linguagem

A única parte a ser desenvolvida pelo grupo nesta fase, será o código que a gramática poderá reconhecer. Logo, foi preciso criar um conjunto de regras:

- Comentários;
- Declaração e manipulação de variáveis;
- Expressões Aritméticas;
- Impressão e formatação de strings;
- Iteração de listas com um loop `for`;
- `if then else`;
- Loop `while`;
- Expressões Boleanas;
- Aninhamento dentro dos loops e dos `if then else`.

Com as regras acima, alguns exemplos desta nova linguagem serão:

```
id = 0;
for ( student : class ) {
    /*
        Create a xml entry like:
        <student id=1> John Doe </student>
    */

    $ "<student id=" $ id $ " > " $ student $ "</student>\n" $;

    id = id + 1; //increment id
}
```

```
n = 0;
previous = 0;
current = 1;
next = 1;

while ( n < 10 ) {
    /*
        Fibonacci numbers:
        0: 0
        1: 1
        2: 1
        3: 2
        ...
    */

    if ( n == 0 )

        next = 0;

    else if ( n == 1 )

        next = 1;

    else
    {
        next = previous + current;
        previous = current;
        current = next;
    }

    $ n $ ": " $ next $ "\n" $;

    n = n + 1;
}
```

Apesar de não ser necessário, como um desafio, recorreu-se a uma AST (Abstract Syntax Tree) para representar o template, sendo processada prosteriormente ao parse do ficheiro template em si.

3 Sintaxe léxica

3.1 Tokens reservados

Por referência ao conjunto de regras acima, existirão 4 palavras que terão de ser reservadas, sendo estas:

```
reserved = {
        'for':   'FOR',
        'if':    'IF',
        'else':  'ELSE',
        'while': 'WHILE',
}
```

Reserva-se este conjunto de tokens, pois uma variável, por exemplo, nunca poderá ter o mesmo conjunto de caracteres que a palavra "for". Caso isto acontecesse, haveria o risco do parser assumir que se encontrava dentro de um loop, quando na verdade seria uma simples variável.

Posteriormente, no token 'ID', verifica-se se na verdade capturou uma destas palavras, alterando o token para o tipo correto nesse caso.

3.2 Estados

Tal como foi mencionado na secção 1.1, haverá três estados possíveis para este projeto.

- `INITIAL`: Secções de texto não processado;
- `section`: Secções de código que serão processados posteriormente;
- `commentblock`: Comentário multi linha que deve ser ignorado.

3.2.1 Estado INITIAL

Este estado contém apenas dois tokens: `TEXT`, para capturar tudo até o delimitador de secção ('/===='), e o `SECTIONBEGIN` para capturar esse mesmo delimitador e iniciar o contexto `section`. Também é de notar que este estado não ignorará nenhum caracter.

```
t_TEXT  = r'((?!/===)(.|\s))+'

def t_SECTIONBEGIN(t):
        r'/==='
        t.lexer.begin('section')
        return t

t_ignore = ''
```

3.2.2 Estado section

Este contexto é certamente o mais extenso. Como irá conter o texto a ser futuramente processado, será necessário separar cada componente diferente em diferentes tokens, de modo a que a gramática possa distingui-los e trabalhar com eles.

Em primeiro lugar distinguiu-se os tokens "literais", tais como o ponto-e-virgula (;) ou da soma (+), entre muitos outros:

```
t_section_ignore = ' \n\t\r'

t_section_FOR                      = r'for'
t_section_IF                       = r'if'
t_section_ELSE                     = r'else'
t_section_WHILE                    = r'while'
t_section_NUM                      = r'\d+(\.\d+)?'
t_section_FORMATTEDSTRINGDELIMITER = r'\$'
t_section_SEMICOLLON               = r';'
t_section_OCURLYBRACKETS           = r'{'
t_section_CCURLYBRACKETS           = r'}'
t_section_COLLON                   = r':'
t_section_PLUS                     = r'\+'
t_section_MINUS                    = r'-'
t_section_MULT                     = r'\*'
t_section_DIV                      = r'/'
t_section_OPARENTHESIS             = r'\('
t_section_CPARENTHESIS             = r'\)'
t_section_OR                       = r'\|\|'
t_section_AND                      = r'&&'
t_section_GT                       = r'>'
t_section_LT                       = r'<'
t_section_GE                       = r'>='
t_section_LE                       = r'<='
t_section_EQ                       = r'=='
t_section_NEQ                      = r'!='
t_section_NOT                      = r'!'
t_section_EQUAL                    = r'='
t_section_OSQUAREBRACKETS          = r'\['
t_section_CSQUAREBRACKETS          = r'\]'
```

Após a criação destes tokens, criou-se mais 5 que serão mais complexos, cada um com uma função bastante diferente.

Nome	Descrição	Código

SECTIONEND	Assinala o fim do contexto section	`def t\_section\_SECTIONEND(t):`
		`  r'===/'`
		`  t.lexer.begin('INITIAL')`
		`  return t`

STR	Captura o conteúdo entre aspas	`def t\_section\_STR(t):`
		`  r'\"(?P<content>.*?)`
		`  (?<!\\)\"'`
		`  t.value =`
		`  t.lexer.lexmatch.group`
		`  ('content')`
		`  return t`

ID	Verifica se capturou uma palavra reservada	`def t\_section\_ID(t):`
		`  r'[a-zA-Z_]\w*'`
		`  t.type =`
		`  reserved.get(t.value, 'ID')`
		`  return t`

COMMENTLINE	Ignora comentários single-line iniciados por //	`def t\_section\_COMMENTLINE(t):` `  r'//[^\n]+'`

COMMENTBEGIN	Inicia o comentário multi linha	`def t\_section\_COMMENTBEGIN(t):`
		`  r'/\*'`
		`  t.lexer.begin('commentblock')`

Tabela 1:

3.2.3 Estado commentblock

Quanto ao comentário multi linha, este é muito semelhante ao contexto 'INITIAL' no sentido em que vai capturar todo o texto até a um certo delimitador, neste caso '*/'. Tudo o que capta será ignorado.

```
def t_commentblock_COMMENTEND(t):
        r'\*/'
        t.lexer.begin('section')


def t_commentblock_COMMENTBLOCK(t):
        r'(\s|.)+?(?=\*/)'
```

4 Sintaxe Semântica

O template será constituido por secções de texto e de código intercaladas, sendo que esta está delimitada por '/===' e '===/', originando a seguinte porção da gramática:

```
Sections : Sections Section
         | Sections TEXT
         |

Section : /=== SectionContent ===/
```

O conteúdo da secção de código corresponderá simplesmente a uma lista de operações, sendo cada uma destas uma das possíveis opções:

- Bloco `for`
- Impressão de uma string formatada para o ficheiro de output
- Bloco `if-then-else`
- Manipulação de uma variável
- Bloco `While`

A partir da gramática apresentada abaixo, observa-se que `ListsOp` encontra-se em evidência. Também se observa que se utiliza recursividade à esquerda em `ListsOps`. Utiliza-se este tipo de recursividade, pois o YACC será mais eficiente.

```
SectionContent : ListOps

ListOps : ListOps Op
        |

Op : ForBlock
   | $ FormattedStr ;
   | IfBlock
   | VarManipulation ;
   | WhileBlock
```

De seguida será preciso desenvolver as porções da gramática dedicada às diferentes operações.

```
FormattedStr : FormattedStr Expression $
             |

ForBlock : for ( ID : ID ) Op
         | for ( ID : ID ) { ListOps }

IfBlock : if ( BoolExpression ) Op
        | if ( BoolExpression ) { ListOps }
        | if ( BoolExpression ) Op ElseBlock
        | if ( BoolExpression ) { ListOps } ElseBlock

ElseBlock : else Op
          | else { ListOps }

BoolExpression : ...

VarManipulation : ID = Expression

Expression : ...

WhileBlock : while ( BoolExpression ) Op
           | while ( BoolExpression ) { ListOps }
```

A estrutura das expressões, uma vez feita em aula, não vai ser analizada em detalhe, notando-se, apenas, as regras correspondentes aos fatores:

```
def p_factor_num(p):
    'Factor : NUM'

def p_factor_id(p):
    'Factor : ID'

# ID [ Expression ]
def p_factor_index(p):
    'Factor : ID OSQUAREBRACKETS Expression CSQUAREBRACKETS'

def p_factor_str(p):
    'Factor : STR'
```

Esta estrutura foi replicada nas expressões boleanas, em que expressões entre parêntesis tomam prioridade, seguido das comparações ($<$,$\leq$,$\geq$, $>$) e da negação (!), e só depois as operações AND (&&) e OR (||).

5 Implementação da Solução

5.1 YAML

Utilizou-se uma biblioteca externa, PyYAML[3], para fazer parse do ficheiro YAML:

```
dic = yaml.load(yaml_content, Loader=Loader)
```

5.2 Parse do Template e AST

5.2.1 Demonstração de uma AST

A partir do seguinte código exemplo, quererá-se gerar a AST demonstrada pela figura 1.

```
Example1
/===
a = 0;
while (a < 10) {
    $ "\n : " $ a $;
    a = a + 1;
}
===/
Example2
```

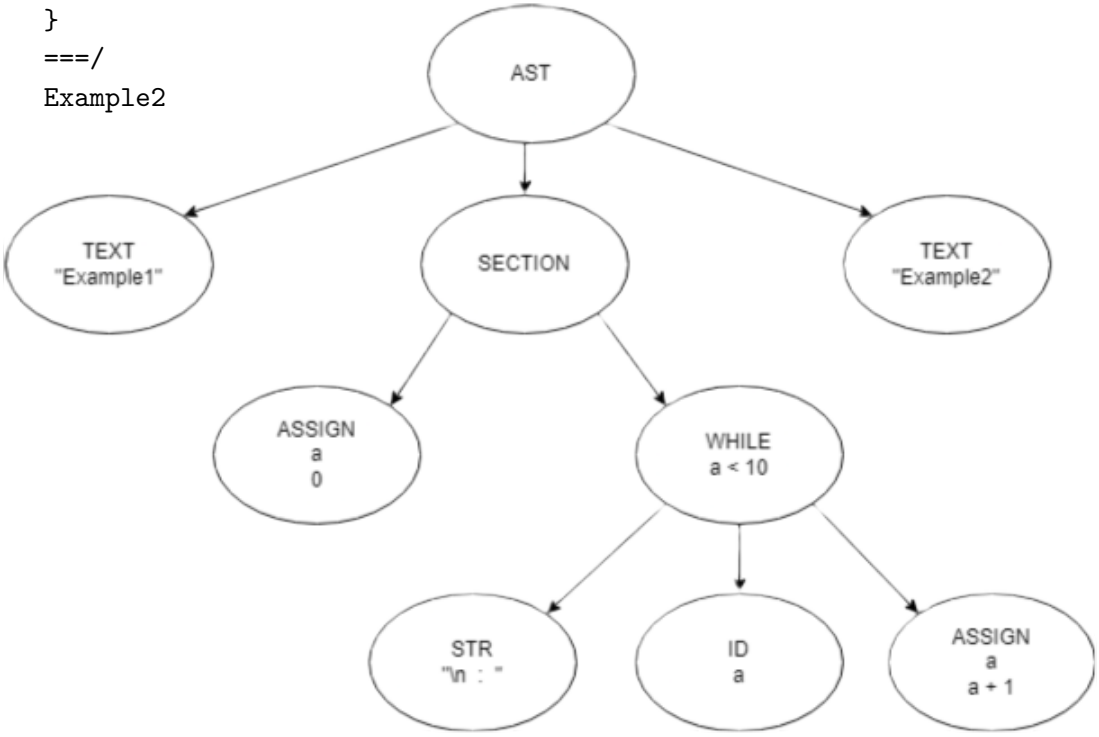

Figura 1: AST correspondente

5.2.2 Parser

A AST será representada por uma lista de nodos, por exemplo:

```
[ nodo1, ( nodo2, ( ramo1 , ramo2 ) ), nodo3, ... ]
```

Tipos dos nodos principais:

- ('text', *texto a imprimir no output*)
- ('section', *lista de operações*)
- ('str', *string impressa no output, depois de formatada*)
- ('id', *nome da variável a imprimir*)
- ('exp', *expressão aritmética*)
- ('for', (*nome da variável que vai conter o elemento*, *lista a iterar*, *lista de operações*))
- ('if', (*condição*, *operações a realizar se a condição for verdadeira*, *operações a realizar caso contrário*))
- ('assign', (*nome da variável manipulada*, *expressão aritmética*))
- ('while', (*condição*, *operações a realizar enquanto a condição for verdadeira*))

A AST será gerada aquando do parse do conteúdo, utilizando a funcionalidade do YACC já observada nas aulas. A título de exemplo:

```
def p_sections_section(p):
    'Sections : Sections Section'
    p[0] = p[1] + [('section', p[2])]


def p_sections_text(p):
    'Sections : Sections TEXT'
    p[0] = p[1] + [('text', p[2])]


def p_sections_empty(p):
    'Sections : '
    p[0] = []
```

Posteriormente, é apenas necessário invocar a função de parse, e guardar o valor retornado, que corresponderá à AST construída.

```
ast = parser.parse(content)
```

5.3 Execução da AST

Agora que a AST foi gerada, é necessário processá-la e executá-la. Essa é a tarefa da função 'solve'.

5.3.1 Função Solve

A função solve recebe a raiz da AST, e lida com os nodos principais, supramencionados. O excerto de código que lida com os tipos mais simples será o seguinte.

```
for op in ast:
    op_type = op[0]

    if op_type == 'text':
        output.write(op[1])
        # escrever diretamente no output

    elif op_type == 'section':
        solve(op[1], dic, output)
        # chamada recursiva

    elif op_type == 'str':
        output.write(format(op[1]))
        # escrever no output depois de formatação

    elif op_type == 'id':
        output.write(str(dic[op[1]]))
        # escrever o valor guardado no dicionário

    elif op_type == 'exp':
        output.write(format(str(evaluate_expression(op[1], dic))))
        # escrever no output a solução da expressão

    elif op_type == 'assign':
        var, exp = op[1]
        dic[var] = evaluate_expression(exp, dic)
        # alterar o valor da variável no dicionário
```

Os 'for's, 'if's e 'while's são operações mais complexas e, portanto, necessitam de processamento também mais complexo, que será descrito de seguida.

FOR:

```
var, lst, ops = op[1]

for element in dic[lst]:
        dic[var] = element
        solve(ops, dic, output)

dic.pop(var)
```

Cria uma variável temporária com nome var no dicionário, que irá conter o valor de cada elemento da lista a iterar. Em cada iteração, é chamada recursivamente a função solve nas operações que o ciclo encapsula.

IF:

```
condition, ifops, elseops = op[1]

if(evaluate_bool_expression(condition, dic)):
        solve(ifops, dic, output)
else:
        solve(elseops, dic, output)
```

É resolvida a expressão condition e, caso o seu valor seja 'true', é chamada a função solve nas operações 'ifops'. Caso o valor seja 'false', é chamada nas operações 'elseops', que corresponde a operações encapsuladas por um bloco 'else', sendo que, na situação em que não existe este bloco, 'elseops' será uma lista vazia, e a chamada recursiva não fará nada.

WHILE:

```
condition, ops = op[1]

while(evaluate_bool_expression(condition, dic)):
        solve(ops, dic, output)
```

Enquanto a expressão condition tiver valor 'true', as operações encapsuladas serão executadas.

5.3.2 Format

Antes de imprimir a string para o output, é necessário formatá-la, substituindo os '\n' pelo new-line correspondente, por exemplo.

```
symbols = {
        '\\n': '\n',
        '\\t': '\t',
        '\\"': '\"'
}

for symbol in symbols:
        string = string.replace(symbol, symbols[symbol])

return string
```

5.3.3 Evaluate Expression

Esta função é utilizada para calcular o valor de uma expressão, que é, por si, um ramo da AST, cujos nodos correspondem às diferentes operações.

A solução para os fatores mencionados previamente corresponde a:

```
op_type = exp[0]

if(op_type == 'id'):
        return dic.get(exp[1])

if(op_type == 'num'):
        try:
                result = int(exp[1])
        except:
                result = float(exp[1])
        return result

if(op_type == 'str'):
        return exp[1]

if(op_type == 'index'):
        return dic[exp[1][0]].get(evaluate(exp[1][1], dic), None)
```

Para operações resolveu-se da seguinte forma:

```
left = evaluate_expression(exp[1][0], dic)
right = evaluate_expression(exp[1][1], dic)

if(op_type == 'add'):
        return left + right

if(op_type == 'sub'):
        return left - right

if(op_type == 'mult'):
        return left * right

if(op_type == 'div'):
        return left / right
```

5.3.4 Evaluate Bool Expression

Muito semelhante à função anterior, esta é utilizada para calcular o valor de expressões booleanas.

Fatores:

```
if(cond_type == 'id'):
        return (dic.get(condition[1], None))
```

Operações Unárias:

```
if(cond_type == 'not'):
        return not evaluate(condition[1], dic)
```

Operações Binárias:

```
left, right = condition[1]

if(cond_type == 'and'):
        return evaluate(left, dic) and evaluate(right, dic)

if(cond_type == 'or'):
        return evaluate(left, dic) or evaluate(right, dic)

if(cond_type == '>'):
        return evaluate(left, dic) > evaluate(right, dic)

if(cond_type == '<'):
        return evaluate(left, dic) < evaluate(right, dic)

if(cond_type == '>='):
        return evaluate(left, dic) >= evaluate(right, dic)

if(cond_type == '<='):
        return evaluate(left, dic) <= evaluate(right, dic)

if(cond_type == '=='):
        return evaluate(left, dic) == evaluate(right, dic)

if(cond_type == '!='):
        return evaluate(left, dic) != evaluate(right, dic)
```

6 Conclusão e Trabalho Futuro

Após a conclusão deste trabalho, o grupo pode adquirir conhecimentos mais aprofundados relativamente a definição de linguagens e as respetivas gramáticas.

Numa futura versão hipotética deste projeto, propôs-se trabalhar na 'intuitividade' da linguagem, isso é, apesar de apresentar mais funcionalidade que os templates Pandoc, é bastante explícita, o que, de certa maneira, não é apelativo para uma linguagem de templates.

Referências

[1] PLY Documentation, `https://ply.readthedocs.io/en/latest/ply.html`

[2] Project Jupyter Homepage, `https://jupyter.org/`

[3] PyYAML Homepage, `https://pyyaml.org/`