Processamento de Linguagens (3º ano - Licenciatura em Engenharia Informática)

Trabalho Prático

Relatório de Desenvolvimento - Conversor de GEDCOM

Eduardo Fernando Cruz Henriques
A93186

José dos Santos Mendes
A92974

28 de maio de 2023

Resumo

Neste relatório iremos documentar o nosso processo de raciocínio para a criação de um conversor de ficheiros de dados genealógicos, no formato *GEneological Data COMmunication*(ou *GEDCOM*) para um formato XML. Abaixo iremos demonstrar um resumo do que foi feito, assim como as justificações para as nossas decisões e demonstração dos resultados.

Conteúdo

Capítulo 1

Introdução

No contexto do trabalho prático para a disciplina de Processamento de Linguagens, fomos introduzidos a oito problemas/enunciados diferentes dos quais tivemos de selecionar o que nos despertou mais interesse. Escolhemos o enunciado número 7 - o conversor de ficheiros GEDCOM para XML/HTML - porque o formato de ficheiros GEDCOM é bastante único e nunca tínhamos trabalhado com este tipo de dados/ficheiros(dados genealógicos).

Para a conceção do nosso *parser* e do nosso *lexer* usamos as ferramentas *ply.yacc* e *ply.lex* da livraria *PLY*, na versão 3.10 do *Python*.

O nosso programa irá converter os ficheiros através de inputs no terminal, e este irá apresentar algumas das mudanças que aplicará ás linhas durante a conversão para XML. Finalmente, irá gerar o resultado num ficheiro á parte.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Ficheiros GEDCOM são uma excelente formato para armazenar dados genealógicos de maneira a obter informação específica a uma pessoa de um determinado período de tempo mais facilmente. Neste projeto, pretende-se que o grupo de trabalho criemos um conversor de *GEDCOM* 5.5 para *XML* ou *HTML*. Escolhemos converter os ficheiros para o formato *XML* porque já tínhamos bastante experiência com este formato usando outras ferramentas.

Para isto tivemos de criar uma Gramática Independente do Contexto(GIC) necessária para gerar um parser e de criar um lexer adequado á gramática escrita. Para além disto o programa deve informar o utilizador de algumas das conversões que está a fazer para obter uma linha no formato XML.

Existem várias fontes de informação sobre o formato GEDCOM na Internet, sendo uma destas fornecida no enunciado. Também nos foram fornecidos alguns ficheiros de teste, assim como os seus possíveis outputs.

Um exemplo de output possível demonstrado pelos docentes é:

```
<genoa>
    <pessoa>
        <id>I1</id>
        <nome>Victoria Hanover</nome>
        <título>Queen of England</título>
        <sexo>F</sexo>
        <dataNasc>1819-05-24</dataNasc>
        <localNasc>Kensington,Palace,London,England</localNasc>
        <dataÓbito>1901-01-22</dataÓbito>
        <pai>I133</pai>
    <mae>I138</mae>
    </pessoa>
    ...
</genoa>
```

2.2 Estrutura do Projeto e Desenho da Solução

Agora iremos demonstrar a estrutura geral e processo de pensamento para a criação das estruturas do nosso programa.
O nosso programa apresenta a seguinte estrutura, dentro da diretoria **src**. Apenas iremos apresentar aqui os ficheiros e diretorias relevantes para o seu funcionamento. Os restantes ficheiros e diretorias podem ser removidos, porque ou serão gerados após correr o programa ou apenas serviram para testar versões iniciais deste:

- output/
  - example.xml
  - (...)
- test/
  - example.ged.txt
  - (...)
- lexer.py
- parser.py
- main.py
- pessoa.py
- familia.py
- tag_handler.py

Quando executamos a aplicação pelo ficheiro main.py, podemos converter um dos quatro ficheiros fornecidos pelos docentes [2] ou selecionar manualmente um ficheiro '.ged.txt' que esteja na diretoria 'test'.
O programa irá gerar o ficheiro xml correspondente e irá guardá-lo na diretoria 'output' independentemente da opção escolhida(caso ele exista no evento de ser um ficheiro fornecido manualmente).

Durante o processo de conversão, irá apresentar as tags que encontra e o nome do elemento gerado correspondente. O nome da tag irá ser alterado se esta pertencer um dos dicionários criados por nós com tags já conhecidas. Caso não seja esse o caso a tag irá manter a sua nomenclatura inicial, especialmente pela possibilidade do utilizador criar tags. Existe a opção de converter vários ficheiros numa única execução do programa.

Também é de valor mencionar que o nosso conversor ignora quaisquer valores no cabeçalho do ficheiro, mas converte tanto as pessoas como as famílias para o formato xml, ao contrário do exemplo possível de conversão dado pelos docentes, que apenas converte as pessoas e recolhe informações sobre os seus pais.

Iremos agora fazer uma descrição superficial de alguns ficheiros apresentados acima, antes de descrever em detalhe o lexer(2.2.1) e o parser(2.2.3), e como interligamos estes componentes a ambos:

1. Pessoa: Ficheiro python que contém a classe pessoa que irá conter o id da pessoa, a familia em que é um dos filhos *(FAMC)*, a familia em que é um dos pais *(FAMS)* e as linhas do ficheiro que lhe pertencem.Irá também conter a função **lookup** que permite no fim da leitura do ficheiro adicionar linhas à pessoa indicando o seu pai a sua mãe e os seus irmãos.

2. Familia: Ficheiro python que contém a classe familia onde irá ser guardado o identificador da mesma, as crianças dessa familia, o marido, a esposa e as linhas do ficheiro que lhe pertencem.

3. Tag handler: Ficheiro python que contém dois dicionários das tags utilizadas nos ficheiros fornecidos pelos professores. Um deles é para guardar tags singulares, seguidas apenas pelo seu elemento correspondente(exemplo: a tag *TITL* passará a Título), e a outra que contém as tags múltiplas, normalmente se encontram dentro de um sub-nível de outras(DEAT, BIRT, etc...). Onde as tags múltiplas irão ser processadas de forma a alterar o seu output em função do tipo.

2.2.1 Lexer

O nosso lexer foi adaptado duas vezes até obtermos a versão final que estamos a usar. Primeiramente, tínhamos um token para cada *tipo* de *tag*, mas percebemos que não era possível considerando o enorme número de *tags* que a sintaxe de GEDCOM tem após analisá-la usando o link [1] fornecido pelos docentes e os ficheiros de input [2].
A versão final do lexer tem 17 tokens, 4 dos quais são definidos por funções. Os tokens principais são:

```
def t_CONTENT(t):
    r"""(_?[A-Z]{3,15}|@[^IF][^@\n]+?@)[\ \t]([^\n]+)"""
    return t

def t_MULTITAG(t):
    r'[A-Z]{3,15}(?=\n)'
    return t

 def t_INDI(t):
    r"INDI"
    return t

def t_FAM(t):
    r"FAM\b"
    return t

t_POINTER = r"\@[IF][^@]+?\@"

    t_BEGIN = r"\n(?=0\ @[IF])"
```

O token ***POINTER*** vai detetar valores que começem e acabem com @ e a primeira letra seja ou I ou F, de maneira a ser apenas um apontador de família ou de pessoa.Após esta primeira letra irá ler qualquer valor que não seja p @ final.
O token ***CONTENT*** foi criado para ler o conteudo de uma linha que contêm uma tag singular, vai ler a tag primeiro, ou um apontador que não seja de família ou de individuo seguido de todos os valores até ao fim da linha.

O token ***BEGIN*** é uma espécie de token de controlo utilizado no fim de cada pessoa/família. Ele verifica se a próxima linha vai pertencer a uma nova pessoa/família através do positive lookahed *(?=0 @[IF])* que verifica se a linha vai ser 0 assim como se a seguir ao nível irá estar o pointer correto(um @ seguido por I ou F).

Os restantes tokens que apresentamos aqui são bastante simples mas estão inseridos em funções que retornam o token porque a biblioteca PLY dá prioridade a tokens que têm uma função associada.
Achamos que definir as prioridades desta maneira seria mais simples do que usar algo como estados, apesar de ser mais complicado de compreender á primeira vista. Mas é necessário para que o nosso lexer funcione corretamente.

Não comentamos sobre os restantes tokens pela sua expressão regular ser apenas a palavra que tentamos ler assim como seu tratamento ser elementar.

2.2.2 Gramática/Sintaxe

Inicialmente construímos uma versão simples da gramática sem usar as funções para adaptar as produções. O nosso objetivo era apenas verificar que o parser está a ser bem construído e esta gramática inicial não incluía o *header* ou a secção dedicada ás famílias.
Após alterar algumas produções e adaptando a gramática para implementar recursividade á esquerda, o parser estava a funcionar como pretendido.
A versão final da GIC é a seguinte:

```
gedcom  : START_FILE header BEGIN people families --> formato geral do ficheiro

header : header LEVEL restHeader --> ler o header de maneira recursiva
        | LEVEL restHeader --> cada linha do header contem o nivel e o resto da linha

restHeader : CONTENT --> caso seja uma tag singular(após a tag vai ter o conteudo)
            | MULTITAG -->caso seja uma multitag(tag que define o tipo de conteudo,ex: BURI)

people : people person --> ler as pessoas de maneira recursiva
       | person

person :  LEVEL POINTER INDI conteudo BEGIN
    |
    V
cada pessoa contem um nível o seu apontador único e a tag INDI seguida do conteudo da pessoa
(o token BEGIN indica o fim da pessoa)

conteudo : conteudo LEVEL restPerson --> ler o conteudo da pessoa de maneira recursiva
         | LEVEL restPerson --> cada linha contem o nível seguida do conteúdo

restPerson : CONTENT --> igual ao header
            | MULTITAG --> igual ao header

families : families family --> ler as famílias de maneira recursiva
         | family
```

```
family :  LEVEL POINTER FAM conteudoF BEGIN --> caso normal
            | LEVEL POINTER FAM conteudoF END_FILE --> caso seja ultima familia
            | LEVEL POINTER FAM BEGIN --> caso excepcional da familia não ter conteudo
            | LEVEL POINTER FAM END_FILE --> caso excepcional da ultimna familia não ter cont
                |
                V
cada familia contem um nível o seu apontador único e a tag FAM seguida do conteudo da pessoa
(o token BEGIN indica o fim da pessoa)

conteudoF : conteudoF LEVEL restFams --> ler o conteudo da pessoa de maneira recursiva
          | LEVEL restFams --> cada linha contem o nível seguida do conteúdo

restFams :  CONTENT --> igual ao header
            |  MULTITAG --> igual ao header
```

Ou seja, um ficheiro tem de ter obrigatoriamente uma ou mais pessoas e uma ou mais famílias, e tem de ter uma secção para o cabeçalho, que necessita de conter pelo menos mais uma linha para além da inicial. A secção das famílias já tem incluída a linha final que deve ser lida(*0 TRLR*).

2.2.3 Parser

Através das produções criadas no parser foi nos possível converter os ficheiros .ged.txt para .xml mais facilmente. Para tal utilizamos as seguintes produções de modo a atingir este resultado:

```
def p_person_pointer_indi(p):
    """person :  LEVEL POINTER INDI conteudo BEGIN"""
        global lista_pessoas
        global pessoa_atual

        id_int = p[2].replace("@", '')  # obter ID

        p[0] = "<ID>" + p[2] + "</ID>"
        print(p[0] + "\n")

        pessoa_atual.add_id(p[2])  # associar ID á pessoa
        lista_pessoas[id_int] = pessoa_atual  # guardar pessoa no dicionario (ID->PESSOA)
        pessoa_atual = Pessoa()  # dar reset a pessoal atual
```

Onde tal como está escrito nos comentários estamos a obter o ID através de um processamento do token ***POINTER***.
Após isto associamos esse ID à pessoa correspondente(a classe criada por nós em pessoa.py), guarda-mo-la no dicionário de pessoas para processamento futuro, e damos "*reset*"à classe pessoa_atual de maneira a estarmos prontos para receber o conteúdo da pessoa seguinte na lista.

```
def p_restPerson_single(p):
     """restPerson : CONTENT"""
        global pessoa_atual
```

```
        global tag_atual
        tag = p[1].split(" ", 1)[0]

        # atraves do tipo(nascimento, morte, etc.) e da tag que analisou,
        # substitui a tag. Exemplo: muda_tag("DATE", "Nasc") = "DataNasc"
        #                           muda_tag("NAME", tipo_irrelevante) = "Nome"
        print(f"{tag} -> {muda_tag(tag, tipo, pessoa_atual.currentLevel)}")

        tag = muda_tag(tag, tipo, pessoa_atual.currentLevel)
        cont = p[1].split(" ", 1)[1]

        if pessoa_atual is not None:
            p[0] = '\t' + '<' + tag + '>' + cont.replace("&", " and ") + '</' + tag + '>'
            print(p[0] + "\n")
            if tag == "FAMS":
                pessoa_atual.add_fams(cont.strip())
            elif tag == "FAMC":
                pessoa_atual.add_famc(cont.strip())
            elif tag == "CONT":
                    # nao escrevemos notas
                if tag_atual != "NOTE":
                    pessoa_atual.add_cont(cont, tag_atual)
            else:
                tag_atual = tag
                    # nao escrevemos notas
                if tag != "NOTE":
                    pessoa_atual.add_line(p[0])
                else:
                    print("null person")
```

Mais uma vez iremos utilizar os comentários para facilitar a perceção daquilo que esta produção faz. Neste caso estamos a fazer o processamento de cada linha dentro de uma pessoa.

Para tal utilizamos funções como *muda_tag* no ficheiro *tag_handler.py* e *cont.replace* para alterar a tag para o valor que iremos escrever e para reescrever caracteres não permitidos em .xml respetivamente.

O tratamento é feito dentro da produção quando são recebidas linhas com as seguintes tags:

- ***FAMC***: adicionar o apontador ,no objeto, para a familia onde esta pessoa é um dos filhos.
- ***FAMS***: adicionar o apontador ,no objeto, para a familia onde esta pessoa é um dos pais.
- ***CONT***: adicionar a linha lida anteriormente a sua continuação
- ***NOTE*** : ignorar a linha em questão por ser uma nota

```
def p_family(p):
    """family : LEVEL POINTER FAM conteudoF BEGIN
              | LEVEL POINTER FAM conteudoF END_FILE
              | LEVEL POINTER FAM BEGIN
              | LEVEL POINTER FAM END_FILE"""
```

```
    global familiy_tree
    global familia_atual

    id_int = p[2].replace("@", '')
    familia_atual.add_id(id_int)
    familiy_tree[p[2]] = familia_atual
    familia_atual = Familia()
```

Aqui estamos a fazer algo semelhante aquilo feito em *p_person_pointer_indi* mas para as famílias.

```
    def p_restFams_single(p):
"""restFams : CONTENT"""
global familia_atual
global familiy_tree

tag = p[1].split(" ", 1)[0]
cont = p[1].split(" ", 1)[1]

print(f"{tag} -> {muda_tag(tag, tipo, pessoa_atual.currentLevel)}")
tag = muda_tag(tag, tipo, pessoa_atual.currentLevel)

# escape do '&'
mod = re.sub(r'&', r'&amp;', cont)

p[0] = '\t' + '<' + tag + '>' + mod + '</' + tag + '>'
print(p[0] + "\n")

if familia_atual is not None:
if tag == "Mulher":
familia_atual.add_wife(cont.strip())
elif tag == "Marido":
familia_atual.add_husband(cont.strip())
elif tag == "Descendente":
familia_atual.add_child(cont.strip())

familia_atual.add_line(p[0])
else:
print("familia nula")
```

Mais uma vez semelhante a produção feita para as pessoas em *p_restPerson_single* mas o tratamento é para tags diferentes, sendo neste caso:

- ***Mulher***: adicionar a *familia_atual* a esposa.
- ***Marido***: adicionar a *familia_atual* o marido.
- ***FAMC***: adicionar a *familia_atual* uma criança.

```
def p_restFams_mult(p):
    """restFams    : MULTITAG"""
    tag = muda_tag(p[1], None, pessoa_atual.currentLevel)
    global tipo
    tipo = tag
```

Neste caso estamos a alterar o tipo que irá ser relevante na alteração das tags na função *muda_tag*.

Capítulo 3

Problemas Encontrados e Soluções

3.1 Conflitos na *gramática* e *erros* no formato final

3.1.1 Conflitos gramaticais

Durante os testes para a gramática que nós criamos, observamos alguns conflitos *shift-reduce*, sendo aquele que nos deu mais problemas foi relacionado com o facto do parser não saber se deve reduzir usando produção que acaba de reconhecer uma lista de pessoas/famílias ou executar um *shift* para continuar a ler o próximo elemento dessa lista.

3.1.2 Erros no output(Formato XML)

Para além destes conflitos, estávamos a escrever o caractere '&' nos ficheiros XML, o que gerava um erro de parse devido ao caractere ser usado para **Character Reference**, por exemplo &#num, enquanto que nós o usamos para escrever frases, por exemplo "Fortress of P&P".

3.2 Soluções finais

3.2.1 Conflitos gramaticais

A principal maneira que nós utilizamos para resolver os nossos conflitos *shift-reduce* foi com a criação de novos tokens, por exemplo o token *BEGIN*, que indicava o fim de cada elemento e o início do próximo.

3.2.2 Formato XML

O problema do caractere '&' foi o mais simples de resolver: quando adicionamos uma linha `<tag>{conteúdo}</tag>` á classe Pessoa atual, dizemos que qualquer símbolo '&' no conteúdo será substituído pela string " and ", que tem significado semelhante. Isto faz com que o conflito nos ficheiros XML desapareça.

Sabemos que a solução mais usada para este problema seria substituir & por &amp mas na nossa opinião esta torna o ficheiro final menos legível.

Capítulo 4

Testes

Para realizar os testes, existem vários tipos de inputs que esperamos que a nossa aplicação consiga processar. Alguns exemplos serão apresentados abaixo, juntamente com os resultados.

4.1 Cenários apresentados

4.1.1 Cenário básico

O nosso programa consegue analisar o seguinte ficheiro:

```
0 HEAD
1 FILE ROYALS.GED
0 @I1@ INDI
1 NAME Victoria  /Hanover/
1 TITL Queen of England
1 SEX F
1 BIRT
2 DATE 24 MAY 1819
2 PLAC Kensington,Palace,London,England
1 DEAT
2 DATE 22 JAN 1901
2 PLAC Osborne House,Isle of Wight,England
1 BURI
2 PLAC Royal Mausoleum,Frogmore,Berkshire,England
1 REFN 1
1 FAMS @F1@
1 FAMC @F42@
0 @F1@ FAM
1 HUSB @I2@
1 WIFE @I1@
1 CHIL @I3@
1 CHIL @I4@
1 CHIL @I5@
1 CHIL @I6@
1 CHIL @I7@
```

```
1 CHIL @I8@
1 CHIL @I9@
1 CHIL @I10@
1 CHIL @I11@
1 DIV N
1 MARR
2 DATE 10 FEB 1840
2 PLAC Chapel Royal,St. James Palace,England
0 @F42@ FAM
1 HUSB @I133@
1 WIFE @I138@
1 CHIL @I1@
1 MARR
2 DATE 11 JUL 1818
2 PLAC Kew Palace
0 TRLR
```

E convertê-lo no seguinte ficheiro XML:

```
<genoa>
        <pessoas>
        <pessoa>
                <ID>@I1@</ID>
                <Nome>Victoria  /Hanover/</Nome>
                <Titulo>Queen of England</Titulo>
                <Sexo>F</Sexo>
                <DataBIRT>24 MAY 1819</DataBIRT>
                <LocalBIRT>Kensington,Palace,London,England</LocalBIRT>
                <DataDEAT>22 JAN 1901</DataDEAT>
                <LocalDEAT>Osborne House,Isle of Wight,England</LocalDEAT>
                <LocalBURI>Royal Mausoleum,Frogmore,Berkshire,England</
                   LocalBURI>
                <Ref>1</Ref>
                <mae>@I138@</mae>
                <pai>@I133@</pai>
        </pessoa>
        </pessoas>
        <familias>
        <familia>
                <ID>F1</ID>
                <Marido>@I2@</Marido>
                <Mulher>@I1@</Mulher>
                <Descendente>@I3@</Descendente>
                <Descendente>@I4@</Descendente>
                <Descendente>@I5@</Descendente>
                <Descendente>@I6@</Descendente>
                <Descendente>@I7@</Descendente>
                <Descendente>@I8@</Descendente>
                <Descendente>@I9@</Descendente>
```

```
                <Descendente>@I10@</Descendente>
                <Descendente>@I11@</Descendente>
                <Divorcio>N</Divorcio>
                <DataMARR>10 FEB 1840</DataMARR>
                <LocalMARR>Chapel Royal,St. James Palace,England</
                   LocalMARR>
        </familia>
        <familia>
                <ID>F42</ID>
                <Marido>@I133@</Marido>
                <Mulher>@I138@</Mulher>
                <Descendente>@I1@</Descendente>
                <DataMARR>11 JUL 1818</DataMARR>
                <LocalMARR>Kew Palace</LocalMARR>
        </familia>
        </familias>
</genoa>
```

4.1.2 Cenário Avançado

O nosso programa consegue analisar o seguinte ficheiro:

```
0 HEAD
1 SOUR PAF
2 NAME Personal Ancestral File
2 VERS 5.2.18.0
2 CORP The Church of Jesus Christ of Latter-day Saints
3 ADDR 50 East North Temple Street
4 CONT Salt Lake City, UT 84150
4 CONT USA
1 DEST PAF
0 @I1@ INDI
1 NAME Mizraim //
2 GIVN Mizraim
1 SEX M
1 _UID A0249835B2ABD6118B8E0004760DB7A0CC1E
1 FAMS @F1@
1 FAMC @F2@
0 @I2@ INDI
1 NAME Phut //
2 GIVN Phut
1 SEX M
1 _UID A2249835B2ABD6118B8E0004760DB7A0CE3E
1 FAMC @F2@
0 @I3@ INDI
1 NAME Cush //
2 GIVN Cush
1 SEX M
```

```
1 _UID A4249835B2ABD6118B8E0004760DB7A0D05E
1 FAMS @F3@
1 FAMC @F2@
0 @I4@ INDI
1 NAME Sabtah //
2 GIVN Sabtah
1 SEX M
1 _UID A6249835B2ABD6118B8E0004760DB7A0D27E
1 FAMC @F3@
0 @F1@ FAM
1 _UID A1249835B2ABD6118B8E0004760DB7A0CD2E
1 HUSB @I1@
1 CHIL @I96@
1 CHIL @I97@
1 CHIL @I98@
1 CHIL @I99@
1 CHIL @I100@
1 CHIL @I101@
1 CHIL @I102@
0 @F2@ FAM
1 _UID A3249835B2ABD6118B8E0004760DB7A0CF4E
1 HUSB @I34@
1 WIFE @I7@
1 CHIL @I3@
1 CHIL @I2@
1 CHIL @I6@
1 CHIL @I1@
0 @F3@ FAM
1 _UID A5249835B2ABD6118B8E0004760DB7A0D16E
1 HUSB @I3@
1 CHIL @I5@
1 CHIL @I4@
1 CHIL @I79@
1 CHIL @I80@
1 CHIL @I81@
1 CHIL @I82@
0 TRLR
```

E convertê-lo no seguinte ficheiro XML:

```
1 <genoa>
2         <pessoas>
3         <pessoa>
4                 <ID>@I1@</ID>
5                 <Nome>Mizraim //</Nome>
6                 <NomeDado>Mizraim</NomeDado>
7                 <Sexo>M</Sexo>
8                 <_UID>A0249835B2ABD6118B8E0004760DB7A0CC1E</_UID>
9                 <mae>@I7@</mae>
```

```
                <pai>@I34@</pai>
                <irmao>@I3@</irmao>
                <irmao>@I2@</irmao>
                <irmao>@I6@</irmao>
        </pessoa>
        <pessoa>
                <ID>@I2@</ID>
                <Nome>Phut //</Nome>
                <NomeDado>Phut</NomeDado>
                <Sexo>M</Sexo>
                <_UID>A2249835B2ABD6118B8E0004760DB7A0CE3E</_UID>
                <mae>@I7@</mae>
                <pai>@I34@</pai>
                <irmao>@I3@</irmao>
                <irmao>@I6@</irmao>
                <irmao>@I1@</irmao>
        </pessoa>
        <pessoa>
                <ID>@I3@</ID>
                <Nome>Cush //</Nome>
                <NomeDado>Cush</NomeDado>
                <Sexo>M</Sexo>
                <_UID>A4249835B2ABD6118B8E0004760DB7A0D05E</_UID>
                <mae>@I7@</mae>
                <pai>@I34@</pai>
                <irmao>@I2@</irmao>
                <irmao>@I6@</irmao>
                <irmao>@I1@</irmao>
        </pessoa>
        <pessoa>
                <ID>@I4@</ID>
                <Nome>Sabtah //</Nome>
                <NomeDado>Sabtah</NomeDado>
                <Sexo>M</Sexo>
                <_UID>A6249835B2ABD6118B8E0004760DB7A0D27E</_UID>
                <pai>@I3@</pai>
                <irmao>@I5@</irmao>
                <irmao>@I79@</irmao>
                <irmao>@I80@</irmao>
                <irmao>@I81@</irmao>
                <irmao>@I82@</irmao>
        </pessoa>
        </pessoas>
        <familias>
        <familia>
                <ID>F1</ID>
                <_UID>A1249835B2ABD6118B8E0004760DB7A0CD2E</_UID>
                <Marido>@I1@</Marido>
```

```
                <Descendente>@I96@</Descendente>
                <Descendente>@I97@</Descendente>
                <Descendente>@I98@</Descendente>
                <Descendente>@I99@</Descendente>
                <Descendente>@I100@</Descendente>
                <Descendente>@I101@</Descendente>
                <Descendente>@I102@</Descendente>
        </familia>
        <familia>
                <ID>F2</ID>
                <_UID>A3249835B2ABD6118B8E0004760DB7A0CF4E</_UID>
                <Marido>@I34@</Marido>
                <Mulher>@I7@</Mulher>
                <Descendente>@I3@</Descendente>
                <Descendente>@I2@</Descendente>
                <Descendente>@I6@</Descendente>
                <Descendente>@I1@</Descendente>
        </familia>
        <familia>
                <ID>F3</ID>
                <_UID>A5249835B2ABD6118B8E0004760DB7A0D16E</_UID>
                <Marido>@I3@</Marido>
                <Descendente>@I5@</Descendente>
                <Descendente>@I4@</Descendente>
                <Descendente>@I79@</Descendente>
                <Descendente>@I80@</Descendente>
                <Descendente>@I81@</Descendente>
                <Descendente>@I82@</Descendente>
        </familia>
        </familias>
</genoa>
```

Capítulo 5

Conclusão

Com a realização deste projeto, conseguimos aprofundar os nossos conhecimentos sobre Processamento de Linguagens, tendo a base destes nas aulas teóricas e práticas da disciplina.

Aprendemos a manipular e ajustar uma Gramática Independente do Contexto com o objetivo de criar um parser bottom-up, a tokenizar um texto de input complexo através da construção de um lexer, e a usar regras nas produções que criamos em conjunto com estruturas de dados existentes em Python para resolver problemas que não eram imediatamente óbvios ou rápidos de resolver.

Infelizmente, devido a constrições temporais não conseguimos implementar tratamento de erros complexo e algumas das indentações são feitas por nós manualmente em vez de serem calculadas no parser. Isto é algo que teremos de resolver no futuro.

Apêndice A

Main

Código da função main.py

```
1
2 from parser import gedcomToXML
3 #                                                    #
4 #    PROJETO PROCESSAMENTO LINGUAGENS 22-23          #
5 #                                                    #
6
7 if __name__ == "__main__":
8     gedcomToXML()
```

Apêndice B

Lexer

```
import ply.lex as lex
import re
import json

level = 0  # nivel
comm_level = 0  #
indi_level = 0  #

name = ""  # nome da pessoa
pointer = ""  # pointer para uma pessoa
dic = dict()  # dicionario que associa o pointer ao nome da pessoa


def t_error(t):
        #print("erro" + t.value)
        t.lexer.skip(1)


#    states = [
#        ("pessoa", "exclusive"),
#        ("familia", "exclusive")
#    ]
#
tokens = [

        "LEVEL",

        "BIRTH",
        "DEATH",
        "MAR",  # casamento
        "BURIAL",

        "POINTER",

        "CONTENT",

        "CONT",
        "CHAN",
        "FAM",
```

```
40          "INDI",
41          "CHR",
42          "GEDC",
43          "BEGIN",
44          "START_FILE",
45          "END_FILE"
46 ]
47
48 # expressoes tokens e estados
49
50 t_ANY_ignore = r"\t\s"
51
52 t_BEGIN = r"\n(?=0\ @[IF])"
53
54 t_ANY_CHR = r"CHR"
55
56 t_ANY_BIRTH = "BIRT"
57 t_ANY_DEATH = "DEAT"
58 t_ANY_BURIAL = r"BURI"
59 t_ANY_MAR = r"MARR"
60 t_ANY_GEDC = r"GEDC"
61 t_CHAN = r"CHAN"
62 t_FAM = r"FAM\b"
63
64 t_CONT = r"CONT"
65
66 t_ANY_INDI = r"INDI"
67
68
69 t_START_FILE = r"0\ HEAD"
70 t_END_FILE = r"0\ TRLR"
71
72
73 t_LEVEL = r"\d"
74
75
76 def t_CONTENT(t):
77         r"""(_?[A-Z]{3,15}|@[^IF][^@\n]+?@)[\ \t]([^\n]+)"""
78         return t
79
80
81 t_ANY_POINTER = r"\@[IF][^@]+?\@"
82
83
84 lexer = lex.lex()
85
86 if __name__ == "__main__":
87
88         with open("test/sintaxe.txt", 'r') as f:
89                 lines = f.readlines()
90                 for line in lines:
91                         lexer.input(line)
92                         for token in lexer:
93                                 print(f"{token.type:<10} | {token.value:<50}")
```

Apêndice C

Parser

```
1  import ply.yacc as yacc
2  from pessoa import Pessoa
3  from familia import Familia
4  from lexer import tokens
5  from tag_handler import muda_tag
6  import os
7  import re
8
9  # contem todas as pessoas
10 lista_pessoas = dict()
11 familiy_tree = dict()
12
13 tag_atual = None
14 pessoa_atual = Pessoa()  # pessoa vazia
15 familia_atual = Familia()  # Familia vazia
16 tipo = None
17
18
19 def p_gedcom(p):
20         """gedcom        : START_FILE header BEGIN people families"""
21         print("li_um_ficheiro_gedcom")
22
23
24 # ------------------------------------------------------------ PESSOA
       ------------------------------------------------------------
25
26
27 def p_header(p):
28         """header : header LEVEL restHeader
29                   | LEVEL restHeader"""
30
31
32 def p_header_rest(p):
33         """restHeader : CONTENT
34                       | multTag"""
35
36
37
38
```

```
39 def p_people(p):
40         """people : people person
41                           | person"""
42
43
44 def p_person_pointer_indi(p):
45         """person :  LEVEL POINTER INDI conteudo BEGIN"""
46         global lista_pessoas
47         global pessoa_atual
48
49         id_int = p[2].replace("@", '')  # obter ID
50
51         p[0] = "<ID>" + p[2] + "</ID>"
52         print(p[0] + "\n")
53
54         pessoa_atual.add_id(p[2])              # associar ID á pessoa
55         lista_pessoas[id_int] = pessoa_atual  # guardar pessoa no dicionario (ID->
               PESSOA)
56         pessoa_atual = Pessoa()                # dar reset a pessoal atual
57
58
59 def p_conteudo_list(p):
60         """conteudo              : conteudo LEVEL restPerson
61                                  | LEVEL restPerson  """
62
63
64 def p_restPerson_single(p):
65         """restPerson    : CONTENT"""
66         global pessoa_atual
67         global tag_atual
68         tag = p[1].split(" ", 1)[0]
69
70         # atraves do tipo(nascimento, morte, etc.) e da tag que analisou,
71         # substitui a tag. Exemplo: muda_tag("DATE", "Nasc") = "DataNasc"
72         #                           muda_tag("NAME", tipo_irrelevante) = "Nome"
73
74         print(f"{tag} -> {muda_tag(tag, tipo, pessoa_atual.currentLevel)}")
75
76         tag = muda_tag(tag, tipo, pessoa_atual.currentLevel)
77         cont = p[1].split(" ", 1)[1]
78
79         if tag == "Nome":         # tirar as barras ('/') do nome
80                 tag.replace('/', '')
81
82         # nao escrevemos notas
83         if pessoa_atual is not None:
84                 p[0] = '\t' + '<' + tag + '>' + cont + '</' + tag + '>'
85                 print(p[0] + "\n")
86                 if tag == "FAMS":
87                         pessoa_atual.add_fams(cont.strip())
88                 elif tag == "FAMC":
89                         pessoa_atual.add_famc(cont.strip())
90                 elif tag == "CONT":
91                         if tag_atual != "NOTE":
```

```
92                                      pessoa_atual.add_cont(cont, tag_atual)
93                  else:
94                          tag_atual = tag
95                          if tag != "NOTE":
96                                  pessoa_atual.add_line(p[0])
97          else:
98                  print("null person")
99
100
101 def p_restPerson_mult(p):
102         """restPerson    : multTag"""
103         p[0] = p[1]
104
105
106 # ------------------------------------------------------ FAMILIA
    ------------------------------------------------------------------------

107
108
109 def p_families(p):
110         """families : families family
111                                 | family """
112
113
114 def p_family(p):
115         """family : LEVEL POINTER FAM conteudoF BEGIN
116                           | LEVEL POINTER FAM conteudoF END_FILE
117                           | LEVEL POINTER FAM BEGIN
118                           | LEVEL POINTER FAM END_FILE"""
119         global familiy_tree
120         global familia_atual
121
122         id_int = p[2].replace("@", '')
123         familia_atual.add_id(id_int)
124         familiy_tree[p[2]] = familia_atual
125         familia_atual = Familia()
126
127
128 def p_conteudo_fam(p):
129         """conteudoF             : conteudoF LEVEL restFams
130                                                 | LEVEL restFams"""
131         global pessoa_atual
132         #if len(p) == 4:
133         #       pessoa_atual.currentLevel = int(p[2])
134         #else:
135         #       pessoa_atual.currentLevel = int(p[1])
136
137
138 def p_restFams_single(p):
139         """restFams      : CONTENT"""
140         global familia_atual
141         global familiy_tree
142
143         tag = p[1].split(" ", 1)[0]
144         cont = p[1].split(" ", 1)[1]
```

```
145
146         print(f"{tag} -> {muda_tag(tag, tipo, pessoa_atual.currentLevel)}")
147         tag = muda_tag(tag, tipo, pessoa_atual.currentLevel)
148
149         # escape do '&'
150         mod = re.sub(r'&', r'&amp;', cont)
151
152         p[0] = '\t' + '<' + tag + '>' + mod + '</' + tag + '>'
153         print(p[0] + "\n")
154
155         if familia_atual is not None:
156                 if tag == "Mulher":
157                         familia_atual.add_wife(cont.strip())
158                 elif tag == "Marido":
159                         familia_atual.add_husband(cont.strip())
160                 elif tag == "Descendente":
161                         familia_atual.add_child(cont.strip())
162
163                 familia_atual.add_line(p[0])
164         else:
165                 print("familia nula")
166
167
168 def p_restFams_mult(p):
169         """restFams      : multTag"""
170         p[0] = p[1]
171
172
173 # ------------------------------------------------------------ TAGS
    ------------------------------------------------------------

174
175
176 def p_multTag_birth(p):
177         """multTag               : BIRTH"""
178         p[0] = "Nasc"
179         global tipo
180         tipo = p[0]
181
182
183 def p_multTag_change(p):
184         """multTag               : CHAN"""
185         p[0] = "Mudanca"
186         global tipo
187         tipo = p[0]
188
189
190 def p_multTag_death(p):
191         """multTag               : DEATH"""
192         p[0] = "Obito"
193         global tipo
194         tipo = p[0]
195
196
197 def p_multTag_chr(p):
```

```
        """multTag                    : CHR"""
        p[0] = "Batismo"
        global tipo
        tipo = p[0]


def p_multTag_burial(p):
        """multTag                    : BURIAL"""
        p[0] = "Enterro"
        global tipo
        tipo = p[0]


def p_multTag_marriage(p):
        """multTag                    : MAR"""
        p[0] = "Casamento"
        global tipo
        tipo = p[0]


def p_multTag_gedc(p):
        """multTag                    : GEDC"""


def p_multTag_cont(p):
        """multTag   : CONT"""


def p_error(p):
        print(p)
        p.success = False
        print("Syntax Error!", p)
        exit()


parser = yacc.yacc()
parser.success = True


def gedcomToXML():

        menuStr = '''
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PROCESSAMENTO DE LINGUAGENS 22-23 %
%                                      %
%       CONVERSOR GEDCOM -> XML        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Selecione a opção:

1 => FAMÍLIAS DA BIBLIA
2 => FAMÍLIAS ROMANAS
3 => FAMÍLIAS GREGAS
4 => FAMÍLIAS REAIS
```

```
5 => SAIR
'''

        errorMsg = '''
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%            OPCAO INVALIDA!!!      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''

        opt = None

        while opt != 5:
                opt = input(menuStr)
                match opt:
                        case "1":
                                parse("test/familias_biblia.ged.txt", "output/output_Biblia.xml", opt)
                        case "2":
                                parse("test/familias_romanas.ged.txt", "output/output_Romanas.xml", opt)
                        case "3":
                                parse("test/familias_gregas.ged.txt", "output/output_Gregas.xml", opt)
                        case "4":
                                parse("test/familias_reais.ged.txt", "output/output_Reais.xml", opt)
                        case "5":
                                print("TERM...")
                                exit()
                        case _:
                                print(errorMsg)
                                opt = None


def parse(gedcom_path, output_path, opt):

        success_msg = f'''
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   OPCAO {opt} EFETUADA COM SUCESSO     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
'''

        # dar parse ao ficheiro
        f_gedcom = open(gedcom_path, encoding="utf-8")
        lines = f_gedcom.read()
        parser.parse(lines)
        f_gedcom.close()
        print('%' * 50)

        # colocar as tags <genoa></genoa> dentro do ficheiro completo, <pessoa></pessoa> em torno de cada pessoa
        # e as tags <familia></familia> em torno de cada familia enquanto adiciona
        # as tags <pai></pai> e <mae></mae> ás pessoas que deu parse
```

```
301
302         f_xml = open(output_path, "w")
303         f_xml.write("<genoa>\n")
304         for elem in lista_pessoas.keys():
305                 p = lista_pessoas[elem]
306                 p.lookup(familiy_tree, p.famc)
307                 f_xml.write("\t<pessoa>" + p.__str__() + "\t</pessoa>\n")
308         for familia in familiy_tree.keys():
309                 fam = familiy_tree[familia]
310                 f_xml.write("\t<familia>" + fam.__str__() + "\t</familia>\n")
311         f_xml.write("</genoa>\n")
312
313         print(success_msg)
```

Apêndice D

Ficheiros Auxiliares

D.1 Familia

```
1     class Familia:
2
3     def __init__(self):
4         self.id = None
5         self.child_list = []
6         self.wife = None
7         self.husband = None
8         self.linhas = []
9
10    def add_id(self, ref):
11        self.id = ref
12        ID_line = "\t\t<ID>" + ref + "</ID>\t\n"
13        self.linhas.insert(1, ID_line)
14
15    def add_line(self, l):
16        self.linhas += '\n\t' + l
17
18    def __str__(self):
19        res = ""
20        for l in self.linhas:
21            res += l
22        return res + "\n"
23
24    def add_wife(self, wife):
25        self.wife = wife
26
27    def add_husband(self, husb):
28        self.husband = husb
29
30    def add_child(self, child):
31        self.child_list.append(child)
```

D.2 Pessoa

```
1
2     from familia import Familia
3 # ["<pessoa>\n", "<\\pessoa>"]
4
5
6 class Pessoa:
7         def __init__(self):
8                 self.id = None  # sem ID atribuido
9                 self.linhas = []
10                self.currentLevel = 1
11                self.fams = None
12                self.famc = None
13
14        def add_line(self, linha):
15                self.linhas += '\n\t' + linha
16
17        def add_cont(self, linha, tag_atual):
18                back = len(tag_atual) + 3
19                self.linhas[:-back] += linha.replace('>>', '')
20
21        def add_id(self, ref):
22                self.id = ref
23                ID_line = "\t\t<ID>" + ref + "</ID>\t\n"
24                self.linhas.insert(1, ID_line)
25
26        def add_fams(self, ref):
27                self.fams = ref
28
29        def add_famc(self, ref):
30                self.famc = ref
31
32        def __str__(self):
33                res = ""
34                for l in self.linhas:
35                        res += l
36                return res + "\n"
37
38        def lookup(self, family_tree, famID):
39                if famID is not None:
40                        family = family_tree[famID]
41                        if family.wife is not None:
42                                self.linhas += "\n\t\t<mae>" + family.wife + "</mae>"
43                        if family.husband is not None:
44                                self.linhas += "\n\t\t<pai>" + family.husband + "</
                                    pai>"
45                        if len(family.child_list) != 0:
46                                for irmao in family.child_list:
47                                        if self.id != irmao:
48                                                self.linhas += "\n\t\t<irmao>" +
                                                    irmao + "</irmao>"
```

D.3 Tag Handler

```
    double_tags = {
                        "DATE": "Data",
                        "PLAC": "Local",
                        "TIME": "Hora"
}

single_tags = {
                        "NAME": "Nome",
                        "TITL": "Título",
                        "SEX": "Sexo",
                        "ALIAS": "Alcunha",
                        "DIV": "Divorcio",
                        "REFN": "Ref",
                        "HUSB": "Marido",
                        "WIFE": "Mulher",
                        "CHIL": "Descendente",
                        "GIVN": "NomeDado"
}


# tipo -> Obito, Nasc, Batismo, Enterro, Casamento,...


def muda_tag(tag, tipo, level):
        if tag in double_tags.keys():
                return double_tags[tag] + tipo
        elif tag in single_tags.keys():
                return single_tags[tag]
        else:
                return tag
```

Bibliografia

[1] Uma descrição do formato juntamente com um rascunho da gramática: `http://homepages.rootsweb.com/~pmcbride/gedcom/55gcch1.htm`

[2] Os ficheiros que usamos como input, fornecidos pelos docentes: `https://www4.di.uminho.pt/~jcr/AULAS/didac/ontologias/`