Processamento de Linguagens
LEI (3ºano) + LCC (2ºano)
Trabalho Prático nº 2 (Yacc/Flex)

Ano lectivo 12/13

Objectivos e Organização

Este trabalho prático tem como principais **objectivos**:

- aumentar a experiência de uso do ambiente linux, da linguagem imperativa C (para codificação das estruturas de dados e respectivos algoritmos de manipulação), e de algumas ferramentas de apoio à programação;
- rever e aumentar a capacidade de escrever *gramáticas independentes de contexto* que satisfaçam a condição LR();
- desenvolver processadores de linguagens segundo o método da *tradução dirigida pela sintaxe*, suportado numa *gramática tradutora*;
- utilizar *geradores de compiladores* como o par lex/yacc

Para o efeito, esta folha contém 6 enunciados, dos quais deverá resolver pelo menos um[1].
O programa desenvolvido será apresentado a um dos membros da equipa docente, totalmente pronto e a funcionar (acompanhado do respectivo relatório de desenvolvimento) e será defendido por todos os elementos do grupo (3 alunos), em data a marcar.
O **relatório** a elaborar, deve ser claro e, além do respectivo enunciado, da descrição do problema, das decisões que lideraram o desenho da linguagem e a concepção da gramática, do esquema de tradução e respectivas acções semânticas (incluir as especificações `lex` e `yacc`), deverá conter exemplos de utilização (textos fontes diversos e respectivo resultado produzido). Como é de tradição, o relatório será escrito em LaTeX.
O pacote de software desenvolvido (um ficheiro compactado, `".tgz"`, contendo os ficheiros `".l"`, `".y"`, algum `".c"` ou `".h"` que precise, os ficheiros de teste `".txt"`, o relatório `".tex"` e a respectiva `"makefile"`) deve ser entregue através do sistema de submissão de TPs, até ao dia **23 de Junho**.

Enunciados

Para sistematizar o trabalho que se lhe pede em cada uma das propostas seguintes, considere que deve, em qualquer um dos casos, realizar a seguinte lista de tarefas:

1. Especificar a gramática concreta da linguagem de entrada.
2. Desenvolver um reconhecedor léxico e sintáctico para essa linguagem com recurso ao par de ferramentas geradoras flex/yacc.
3. Construir o gerador de código que produza a resposta solicitada. Esse gerador de código é construído associando acções semânticas de tradução às produções da gramática, recorrendo uma vez mais ao gerador yacc.

[1] Se resolver mais, cada um será avaliado separadamente e a nota final será a média das notas individuais obtidas.

Conteúdo

1 Formato Bilio::Thesaurus **2**

2 Linguagem para definição de dados genealógicos **3**

3 Report 2007: vamos escrever relatórios **4**

4 XML Workbench **7**

4.1 Reconhecedor de Documentos Estruturados 7

4.2 Interpretador de Comandos 8

4.3 Document Query Language 8

4.3.1 Interrogando os Documentos 9

4.3.2 A Linguagem para o Projecto 10

5 Yet Another Top-Down Parser Generator **14**

1 Formato Bilio::Thesaurus

O formato thesaurus ISO 2788 (T2788) é usado para representar ontologias / thesaurus. Dum modo simplificado contem metadados (indicação das linguas, relações externas, inversas das relações, titulo, etc) e um conjunto de conceitos.

Cada conceito inclui:

- um representante na linguagem de base
- traduções noutras línguas.
- relações conceptuais com outros conceitos

Pretende-se:

- escrever uma gramática de T2788
- Escrever uma gramática e analisador léxico capaz de reconhecer um documento em formato Thesaurus ISO 2788,
- Armazená-lo num estrutura de dados em memória (Hash table glib ?)
- Variante 1:
  - Fazer uma travessia da estrutura de dados e gerar uma página HTML para cada conceito, sendo construído hiperligações de acordo com as relações conceptuais.
- Variante 2:
  - gerar um dicionário (LaTeX ou similar).

Exemplo:

```
1 |                                   # directivas / metadados
2 | %language PT EN                   # línguas: PT EN
3 | %baselang EN                      # língua de base: EN
4 | %inv NT BT                        # NT, BT são relações inversas  a NT B => b BT a
5 |                                   # conceitos:
6 | animal                            # termo na baselang
7 | PT animal                         # LINGUA termo
8 | NT cat, dog, cow,                 # NT = narrow term = termo específico
```

```
  fish, ant
NT camel
BT Life being                          # BT = broader term = termo genérico
                                       # linha em brando : separador de conceitos
cat
PT gato
BT animal
SN domestic animal to be kicked when   # scope note = nota explicativa
  anything bad occurs.

#comentário                           # desde o símbolo '#' até ao fim da linha
```

2 Linguagem para definição de dados genealógicos

Ao falar de histórias de vida, histórias de família, etc é muitas vezes necessário incluir dados referentes às relações de parentesco e outros dados de cariz genealógico.

Pretende-se definir uma notação compacta e formal para definir este tipo de dados.

Idealmente, esta notação genealógica (ngen) deve cobrir elementos como:

- Nomes. Exemplo:
  - `João Manuel Rodrigues da Silva`
  - `João Manuel/Rodrigues da Silva` (separação nome aplelido)
  - `João Manuel Rodrigues da Silva%2` (distingue entre 2 elementos com o mesmo nome)
- Eventos e Datas. Exemplo:
  - `*1996` (evento=Nasceu ; data 1996)
  - `+1996` (evento=Faleceu; data 1996)
  - `+c1996` (evento=Faleceu; data cerca de 1996)
  - `cc(1996) P` (evento=casamento; data 1996; com a pessoa P)
  - `ev(ID:1996)` (evento=ID; data 1996)
- Parentescos. Exemplo dada uma pessoa P1:
  - `P P2` (relação P1 tem como pai P2)
  - `PM P2` (relação P1 tem como Avó paterna P2)
  - `P P2` (relação P2 tem como pai P1)
  - `PP P2` (relação P2 tem como Avô paterno P1)
  - `F P2` (relação de P2 com casamento atrás descrito)

  Estas relações têm inverso
- Ficheiros / documentos auxiliares. Exemplo dada uma pessoa P1:
  - `FOTO file.jpg`
  - `HIST file.tex` (em file.tex há uma história em que P1 participa)

```
Manuel da Silva *1977 +2011 [3]  // nome, nascimento, morte, #I=3
M  Maria da Silva +2009          // mãe nome, morte da mãe
P  Joaquim Oliveira da Silva     // Pai
MM Joaquina *1930                // mãe da mãe (nome, data nasc)
MP [45]                          // pai da mãe é o #I45, descrito anteriormente
FOTO f.jpg                       // refere-se a #I3
HIST h1.tex
CC 2000 [2]                      // #I3 casou-se em 2000 com #I8, #F=2
Maria Felisbina *1980 [8]        // Conjugue, nome, nascimento, #I=8
```

```
F  Serafim da Silva *2004        // Filho (ref. ao CC anterior #I3 #I8)
F  Ana da Silva *2006 [7]{       //
     FOTO f1.jpg                 // dados extra referentes à Ana #I7
     HIST h1.tex
   }
```

Pretende-se portanto:

- definir a linguagem
- escrever uma gramática
- gerar factos elementares (ou o que acharem interessante gerar)

Exemplo de alguns factos elementares (pode alterar os detalhes)

```
#I3 nome Manuel da Silva
#I3 data-nacimento  1977
#I3 data-falecimento 2011
#I3 tem-como-M #aut1
#aut1 nome Maria da Silva
#aut1 data-nascimento 2009
#I3 tem-como-P #aut2
#aut2 nome Joaquim Oliveira da Silva
#I3 temo-como-MM #aut3
#aut3 nome Joaquina
#aut3 data-nascimento 1930
#I3 temo-como-MP #I45
#I3 FOTO f.jpg
#I3 HIST h1.tex
#F2 = #I3 #I8
#F2 data-casamento 2000
#I8 nome Maria Felisbina
#I8 data-nascimento 1980
#aut4 nome Serafim da Silva
#aut4 data-nascimento 2004
#F2 tem-como-F #aut4
#I7 nome Ana da Silva
#I7 data-nascimento 2006
#F2 tem-como-filho #I7
#I7 FOTO f1.jpg
#I7 HIST h1.tex
```

3 Report 2007: vamos escrever relatórios

A escrita de relatórios técnicos é muito importante no contexto em que te estás a inserir. Neste projecto, irás desenvolver um compilador que aceitará relatórios escritos numa determinada linguagem e gerará a respectiva versão HTML e, como extra, poderá gerar também uma versão em LaTeX.

A especificação da gramática da linguagem para a escrita de relatórios é dada abaixo (com alguns pormenores em branco). Deverás analisá-la, completá-la e implementá-la.

Na análise da gramática tem em conta as seguintes considerações:

- Símbolos capitalizados pertencem à família dos não-terminais: Report, Abstract, TRowList, ...;
- Símbolos em maiúsculas pertencem à família dos terminais constantes (palavras reservadas ou símbolos carácter): BTITLE, EGRAPH, ...;

- Símbolos em minúsculas pertencem à família dos terminais variáveis: texto, path, ...;
- A definição de cada símbolo terminal ficará a seu cargo, a gramática apenas indica onde eles deverão aparecer (seja imaginativo, proponha alterações, ...);
- Os não terminais marcados com "?"são opcionais e deverão ser tratados à semelhança de "SubTitle";
- Como é suposto utilizarem o `yacc` para implementarem o compilador a gramática foi escrita com recursividade à esquerda, alterem-na se optarem por uma metodologia de parsing Top-Down.

Estruturalmente, um Relatório é composto por 3 partes: uma parte inicial, um corpo e uma parte final.

```
27      Report --> BREPORT FrontMatter Body BackMatter EREPORT
```

A parte inicial é constituída por um título, um subtítulo (opcional), uma lista de autores, uma data, a indicação de uma instituição (opcional), uma lista de palavras-chave (opcional), um resumo, uma secção (opcional) de agradecimentos, um índice (opcional), um índice de figuras (opcional) e um índice de tabelas (opcional).

```
28      FrontMatter --> BFM Title SubTitle? Authors Date Institution? Keywords?
29                      Abstract Aknowledgements? Toc? Lof? Lot? EFM

30      Title --> BTITLE texto ETITLE
31      SubTitle --> BSUBTITLE texto ESUBTITLE
32                 | &
33      Authors --> Authors Author
34                | Author
35      Author --> BAUTHOR Name Nident? Email? Url? Affilliation? EAUTHOR
36      Name --> BNAME texto ENAME
```

O resumo e a secção de agradecimentos são constituídos por uma lista de parágrafos. E os vários índices, por marcas de posição (apenas aparece uma palavra reservada indicando que naquele ponto deve ser colocado um índice que tem de ser gerado).

```
37      Abstract --> BABS ParaList EABS
38      Aknowledgements --> BAKN ParaList EAKN

39      Toc --> TOC | &
40      Lof --> LOF | &
41      Lot --> LOT | &
```

O Corpo do Relatório é constituído por uma lista de capítulos e um capítulo, por sua vez, é constituído por um título e uma lista de elementos.

```
42      Body --> BBODY ChapterList EBODY
43      ChapterList --> ChapterList Chapter
44                    | Chapter Chapter --> BCHAP Title ElemList ECHAP
```

Uma secção tem um modelo semelhante ao do capítulo só que em vez do subelemento `Section` tem o subelemento `SubSection` (o mesmo acontecerá com a `SubSection` e a `SubSubSection`).

```
45      Section --> BSEC Title ElemListSec ESEC

46      ElemList --> ElemList Elem
47                 | Elem
```

Um elemento pode ser um parágrafo, um elemento flutuante (tabela ou figura), uma lista (descritiva, de itens ou numerada), um bloco de código, uma secção, um sumário e poderá acrescentar todos os que achar necessários.

```
48      Elem --> Paragraph
49             | Float
50             | List
51             | CodeBlock
52             | Section
53             | Summary
```

O parágrafo tem um conteúdo composto por texto onde podem aparecer livremente alguns elementos: referências, pedaços de texto com diferentes características de formatação (bold, itálico, ...), acrónimos, ...

```
54      Paragraph --> BPARA ParaContent EPARA
55      ParaContent --> ParaContent texto
56                    | ParaContent FreeElement
57                    | &
58      FreeElement --> Footnote
59                    | Ref
60                    | Xref
61                    | Citref
62                    | Iterm
63                    | Bold
64                    | Italic
65                    | Underline
66                    | InlineCode
67                    | Acronym

68      Ref --> BREF target EREF
69      Xref --> BXREF target EXREF
70      Citref --> BCIT target ECIT
71      Iterm --> BITERM texto EITERM

72      Bold --> BBOLD BContent EBOLD
73      BContent --> BContent texto
74                 | BContent Italic
75                 | BContent Underline
76                 | &

77      Italic --> BITALIC IContent EITALIC
78      IContent --> IContent texto
79                 | IContent Bold
80                 | IContent Underline
```

```
81              | &

82      Underline --> BUNDERLINE UContent EUNDERLINE
83      UContent --> UContent texto
84                 | UContent Bold
85                 | UContent Italic
86                 | &
```

Por sua vez, os elementos flutuantes têm a seguinte estrutura:

```
87      Float --> Figure
88              | Table

89      Figure --> BFIG Graphic Caption EFIG
90      Graphic --> BGRAPH path format? EGRAPH
91      Caption --> BCAPTION texto ECAPTION

92      Table --> BTABLE Caption TRowList
93      TRowList --> TRowList TRow
94                 | TRow
95      TRow --> ...
```

Este esboço gramatical é apenas um ponto de partida. Deverá ser completado e na sua implementação muitas decisões terão de ser tomadas: por exemplo, quais serão os símbolos terminais?

4 XML Workbench

Neste projecto, pretende-se desenvolver uma plataforma para manipulação de documentos XML.

Esta plataforma terá dois níveis: num primeiro nível é preciso reconhecer um documento XML e construir uma sua representação em memória; num segundo nível pretende-se generalizar permitindo o carregamento de vários documentos para memória sobre os quais se poderão fazer várias operações: selecção de partes, geração de novos documentos a partir dos que estão carregados, estatísticas, ...

Podemos dividir este enunciado em 3 partes que se descrevem nas secções seguintes.

4.1 Reconhecedor de Documentos Estruturados

Como já foi referido, nesta fase o alunos deverá desenvolver um parser que valide um documento XML e crie em memória uma representação do mesmo.

A título apenas de exemplo apresenta-se uma possível gramática para um documento XML:

```
1       Documento --> ElemList '$'

2       ElemList --> ElemList Elem
3                  | Elem

4       Elem --> char
```

```
5           | '&' id ';'
6           | '<' id AttrList '>' ElemList '<' '/' id '>'
7           | '<' id AttrList '/' '>'

8       AttrList --> Attr AttrList
9                  | &

10      Attr --> id '=' valor
```

No reconhecimento do documento, o parser desenvolvido deverá verificar os seguintes invariantes:

- todas as anotações correspondentes a elementos com conteúdo são abertas e fechadas correctamente (não há marcas cruzadas e nada fica por fechar ou nada é fechado sem ter sido aberto antes);
- o documento tem que obrigatoriamente começar com a abertura dum elemento (que irá englobar todo o documento).

4.2 Interpretador de Comandos

O parser desenvolvido no ponto anterior será uma peça de algo bem maior: o tal *"XML Workbench"*.

Pretende-se agora criar um ambiente de trabalho que aceite os seguintes comandos:

LOAD <path para o documento> id — Este comando irá usar o parser desenvolvido no ponto anterior para reconhecer e carregar um documento XML. No fim, deverá ainda criar uma entrada numa estrutura de dados interna em que o identificador `id` fica associado ao documento reconhecido;

LIST — Mostra no écran a lista de documentos carregados e respectivos ids;

SHOW id — Mostra no écran o documento associado ao identificador id em formato ESIS (ou noutro formato semelhante definido por si);

EXIT — Sai do programa;

HELP — Imprime no écran um texto parecido com esta lista de comandos.

Pode usar a imaginação à vontade para acrescentar comandos a esta lista.

Considere ainda a seguinte gramática proposta para este interpretador (pode alterá-la à vontade):

```
11      Interp --> ComList
12      ComList --> Comando
13                | ComList Comando

14      Comando --> LOAD fich-id id
15                | SHOW id
16                | LIST
17                | EXIT
18                | HELP
```

4.3 Document Query Language

Neste ponto, todos grupos de trabalho deverão estar munidos dum interpretador de comandos que permite carregar documentos, visualizá-los, fornecendo assim um primeiro conjunto de

facilidades básicas num sistema documental.

Nesta fase, vamos adicionar um novo comando à lista dos já existentes:

```
19      QLE: [selector de documentos] [query-exp]

20      [selector de documentos] --> * "todos os docs carregados"
21                                 | id "apenas o doc com ident=id"
22                                 | id1,id2,...,idn

23      [query-exp] --> "definida mais à frente"
```

O resto do enunciado irá descrever através da apresentação de exemplos as várias facetas das expressões de query que se pretendem suportar.

4.3.1 Interrogando os Documentos

A operação de seleccionar os elementos com os quais se quer fazer alguma coisa, ou aos quais se quer aplicar algum processamento, tem sido, desde há algum tempo, uma preocupação das pessoas que trabalham nesta área. Começou por surgir na transformação e na formatação: era preciso seleccionar os elementos que se queriam transformar, ou que se queriam mapear num ou mais objectos com características gráficas (formatação). Este esforço é visível no DSSSL ; o primeiro elemento das suas regras é uma expressão de "query"que selecciona os elementos aos quais será aplicado o processamento especificado. Por último, esta necessidade surgiu ligada às linguagens de "query"para documentos estruturados, como as que foram propostas na conferência dedicada a esse tópico.

Assim se chegou, rapidamente, à conclusão de que a operação de selecção necessária para a transformação ou formatação era muito semelhante à necessária nos sistemas de bases de dados documentais para a realização de "queries".

Depois de algum tempo de discussão (moderada pelo W3C - World Wide Web Consortium), começa a emergir algum consenso na utilização do XSLT , uma sublinguagem de padrões presente no XSL - a proposta de normalização para a especificação de estilos a associar a documentos XML. O XSLT tornou-se um standard e foi já alvo de um estudo formal por parte de Wadler , apresentado na conferência mundial da área ("Markup Technologies 99"), e onde ele define a linguagem usando semântica denotacional (formalismo de cariz funcional utilizado para especificar a sintaxe e a semântica de linguagens).

Depois dum estudo de algumas destas linguagens (em particular todas as que já foram referidas), foi fácil constatar que o XSLT é um denominador comum de uma grande parte delas, aquelas que foram desenvolvidas a pensar em documentos estruturados, tratando-se portanto de uma linguagem específica. Houve, no entanto, uma linguagem que cativou a atenção do autor, pela sua simplicidade e recurso à teoria de conjuntos, a linguagem proposta por Tim Bray na QL'98 - The Query Languages Workshop designada por Element Sets. Um estudo mais atento da linguagem e do seu historial, revelou ser esta a especificação por detrás do conhecido motor de procura Pat comercializado pela OpenText e utilizado na maior parte dos primeiros portais da Internet.

Enquanto as linguagens do tipo XSLT assentam numa sintaxe concreta e específica, a Element Sets define uma notação abstracta baseada em cinco operadores da teoria de conjuntos: contido (within), contém (including), união (+), intersecção ($\hat{()}$) e diferença ($-$). Bray argumenta ser capaz de especificar uma grande percentagem de queries que possam ser necessárias num sistema de arquivo documental à custa da combinação daqueles cinco operadores. Numa primeira

análise e a título comparativo, apresentam-se a seguir dois exemplos, uma query simples e uma mais complicada que irão ser especificadas respectivamente recorrendo a XSLT e a Element Sets.

Query Simples Pretende-se seleccionar todos os parágrafos (PARA) pertencentes à introdução (INTROD) que contenham uma ou mais notas de rodapé (FOOTNOTE) ou uma ou mais referências (REF) a outros elementos no documento.

Em Element Sets a query seria:

```
1    set1 = Set('PARA') within Set('INTROD')
2    set2 = set1 including Set('FOOTNOTE')
3    set3 = set1 including Set('REF')
4    set4 = (set2 + set3) - (set2 ^ set3)
```

Apesar de complexa, foi fácil especificar esta query. Bastou excluir (diferença de conjuntos) os elementos resultantes da query anterior que continham ambos os elementos (intersecção de conjuntos), REF e FOOTNOTE.

Temos agora, a especificação em XSLT:

```
1    INTROD/PARA[(FOOTNOTE and (not REF)) or (REF and (not FOOTNOTE))]
```

Do estudo comparativo realizado entre os dois tipos de linguagem, e do qual os dois exemplos acima fazem parte, podemos concluir que, em termos da operação de selecção, são mais ou menos equivalentes, não se tendo encontrado nenhuma situação que uma solucionasse e a outra não. Vão diferir é no método como fazem a selecção: o XSLT usa a árvore documental e toda a operação de selecção é feita em função dessa estrutura; a Element Sets, por outro lado, não usa a árvore documental, manipula o documento como um conjunto de elementos usando uma sintaxe mais universal. Mas esta diferença existe apenas perante o utilizador que usa a linguagem porque em termos de implementação não se pode fugir às travessias da árvore documental.

Ao contrário do que o leitor poderia supor nesta altura, a escolha não recaiu sobre a Element Sets mas sim sobre uma linguagem do tipo XSLT, a XQL - XML Query Language . Os motivos por detrás desta escolha são muito simples. Apesar dos paradigmas, em termos de selecção, serem equivalentes, as linguagens do tipo XSLT vão além da selecção, permitem ter um segundo nível de selecção baseado em restrições sobre o conteúdo.

4.3.2 A Linguagem para o Projecto

A linguagem XSLT fornece um método bastante simples para descrever a classe de nodos que se quer seleccionar. É declarativa em lugar de procedimental. Apenas é preciso especificar o tipo dos nodos a procurar usando um tipo de padrões simples baseado na notação de directorias dum sistema de ficheiros (a sua estrutura é equivalente à de uma árvore documental). Por exemplo, livro/autor, significa: seleccionar todos os elementos do tipo autor contidos em elementos livro.

A XQL é uma extensão do XSLT. Adiciona operadores para a especificação de filtros, operações lógicas sobre conteúdo, indexação em conjuntos de elementos, e restrições sobre o conteúdo dos elementos. Basicamente, é uma notação para a especificação de operações de extracção de informação de documentos estruturados.

Como já foi dito, vamos começar por descrever operadores relacionados com a selecção mas a

linha divisória entre selecção e restrição irá sendo diluída ao longo do texto, confundindo-se até, para os casos em que a integração das duas é muito forte.

Padrões e Contexto Uma expressão de selecção é sempre avaliada em função dum contexto de procura. Um contexto de procura é um conjunto de nodos a que uma expressão se pode aplicar de modo a calcular o resultado. Todos os nodos no contexto de procura são filhos do mesmo nodo pai; o contexto de procura é constituído por todos os nodos que são filhos deste nodo pai e respectivos atributos mais os atributos do nodo pai.

As expressões de selecção poderão ser absolutas (o contexto é seleccionado em função do nodo raíz - "/"), ou relativas (o contexto é seleccionado em função do contexto actual - "."). Na especificação do contexto pode ainda ser usado o operador "//"com o significado de descendência recursiva.

Exemplos:

Seleccionar todos os elementos autor no contexto actual :

```
./autor
    ou
autor
```

Seleccionar o elemento raíz (report) deste documento :

```
/report
```

Seleccionar todos os elementos autor em qualquer ponto do documento actual :

```
//autor
```

Seleccionar todos os elementos capítulo cujo atributo tema é igual ao atributo especialidade :

```
capítulo[/report/@especialidade = @tema]
```

Seleccionar todos os elementos título que estejam um ou mais níveis abaixo do contexto act :

```
.//título
```

Quantificador: todos O operador "*"quando usado numa expressão de selecção selecciona todos os elementos nesse contexto.

Exemplos:

Seleccionar todos os elementos filhos de autor :

```
autor/*
```

Seleccionar todos os elementos nome que sejam netos de report :

```
report/*/nome
```

Seleccionar todos os elementos netos do contexto actual :

```
*/*
  ou
./*/*
```

Seleccionar todos os elementos que tenham o atributo identificador :

```
*[@identificador]
```

Atributos Como já se pôde observar nalguns exemplos, o nome de atributos é precedido por "@". Os atributos são tratados como subelementos, imparcialmente, sempre que possível. De notar que os atributos não podem ter subelementos pelo que não poderão ter operadores de contexto aplicados ao seu conteúdo (tal resultaria numa situação de erro sintáctico). Os atributos também não têm conceito de ordem, são por natureza anárquicos pelo que nenhum operador de indexação deverá ser-lhes aplicado.

Exemplos:

Seleccionar o atributo valor no contexto actual :

```
@valor
```

Seleccionar o atributo dólar de todos os elementos preço no contexto actual :

```
preço/@dólar
```

Seleccionar todos os elementos capítulo que tenham o atributo língua :

```
capítulo[@língua]
```

Seleccionar o atributo língua de todos os elementos capítulo :

```
capítulo/@língua
```

Exemplo inválido :

```
preço/@dólar/total
```

Filtro subquery O resultado duma query pode ser refinado através de uma subquery (restrição aplicada ao resultado da query principal), indicada entre "["e "]"(nos exemplos anteriores já apareceram várias sem nunca se ter explicado a sua sintaxe e semântica).

A subquery é equivalente à cláusula SQL WHERE.

O valor resultante da aplicação de uma subquery é booleano e os elementos para os quais o valor final seja verdadeiro farão parte do resultado final.

Há operadores nas subqueries que permitem testar o conteúdo de elementos e atributos.

Exemplos:

Seleccionar todos os elementos capítulo que contenham pelo menos um elemento excerto :

```
1  capítulo[excerto]
```

Seleccionar todos os elementos título pertencentes a elementos capítulo que tenham pelo m :

```
1  capítulo[excerto]/titulo
```

Seleccionar todos os elementos autor pertencentes a elementos artigo que tenham pelo men :

```
1  artigo[excerto]/autor[email]
```

Seleccionar todos os elementos artigo que contenham elementos autor com email :

```
1  artigo[autor/email]
```

Seleccionar todos os elementos artigo que tenham um autor e um título :

```
1  artigo[autor][titulo]
```

Como se pode observar nalguns destes exemplos, algumas das restrições que pretendemos colocar sobre os documentos podem ser especificadas com os construtores e operadores já apresentados. A linha divisória entre a selecção e a restrição parece já um pouco diluída.

Expressões booleanas As expressões booleanas podem ser usadas nas subqueries e estas, já nos permitem especificar condições contextuais como a restrição de valores a um domínio. Uma expressão booleana tem a seguinte forma:

```
1      val-esquerda operador val-direita
```

Os operadores são normalmente binários, tomam como argumentos um valor à esquerda e um valor à direita: `or`, `and` e `not` (este é unário tomando o valor à direita).

Com estes operadores e o agrupamento por parentesis podem especificar-se queries bastante complexas.

Exemplos:

Seleccionar todos os elementos autor que tenham um email e um url :

```
autor[email and url]
```

Seleccionar todos os elementos autor que tenham um email e um url :

```
autor[email and url]
```

No universo das queries, o resultado seria o conjunto de autores que tivessem email e url

Seleccionar todos os elementos autor que tenham um email ou um url e pelo menos uma pu
:

```
autor[(email or url) and publicação]
```

Seleccionar todos os elementos autor que tenham um email e nenhuma publicação
:

```
autor[email and not publicação]
```

Seleccionar todos os elementos autor que tenham pelo menos uma publicação e não tenham
:

```
autor[publicação and not (email or url)]
```

Equivalência A igualdade é notada por = e a desigualdade por ! =.

Podemos usar strings nas expressões desde que limitadas por aspas simples ou duplas.

Exemplos:

Seleccionar todos os autores que têm o subelemento organização preenchido com o valor 'U
:

```
autor[organização = 'U.Minho']
```

Seleccionar todos os elementos que têm o atributo língua preenchido com o valor 'pt'
:

```
*[@língua = 'pt']
```

A linguagem possui todos os operadores relacionais habituais, cuja utilização não foi aqui exemplificada, porém, a sua semântica é bem conhecida e este enunciado já tem um grau de complexidade elevado. Fica ao critério dos grupos de trabalho a sua implementação.

5 Yet Another Top-Down Parser Generator

Neste projecto, pretende-se que o aluno construa um gerador de parsers segundo a filosofia Top-Down. Nesse sentido, será necessário definir uma linguagem para a especificação de gramáticas e todos os algoritmos que verificam se uma gramática especificada pode ser processada pela ferramenta (se é LL(1)), e neste caso gerem o código necessário para implementar o parser (um recursivo descendente ou um dirigido por tabela).

Alternativamente poderão usar uma abordagem Bottom-Up com as devidas validações: LR0, SLR1, LALR.