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

Ano lectivo 2013/2014

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 vários enunciados, dos quais deverá resolver um.

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.

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.

Conteúdo

1 Linguagem para definição de dados genealógicos

Este enunciado está a sofrer alterações pelo prof José João, assim que estiver fechado será disponibilizado.

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
```

2 Verificação automática de exercícios

Pretende-se desenvolver uma aplicação que recebe respostas a testes e as vai confrontar com a solução.

Mais detalhes em breve, a serem acrescentados pelo prof José João.

3 MSP: Mais Simples Possível

O tema escolhido para este projecto foi o da criação de uma máquina de stack virtual e do respectivo ambiente de edição e execução.

A máquina virtual será programada em Assembly numa linguagem muito simples definida para este contexto, o MSP. O MSP é uma linguagem simples, com um número de instruções reduzido e com uma sintaxe e uma semântica bastante acessíveis. O MSP destina-se a programar uma Máquina de Stack Virtual. Como se verá mais à frente, é recorrendo a uma stack que a máquina efectua os seus cálculos. A designação de virtual surge do facto de que tal máquina não tem existência real, sendo, neste caso, da vossa responsabilidade a criação da ilusão de que ela existe.

Este trabalho será dividido em duas fases. Na primeira fase serás responsável pela criação do ambiente de execução da máquina: edição de programas, carregamento, armazenamento e execução. Na segunda fase, terás de criar a máquina de stack e de executar nela os programas entretanto criados.

3.1 Ambiente de programação

O ambiente de programação que se pretende criar é muito semelhante ao que existia para as máquinas pessoais tipo ZX Spectrum nos anos 80.

Basicamente, um utilizador tem um monitor de caracteres à frente com uma área de trabalho útil de 25 linhas, em que cada linha tem 80 colunas. Por baixo dessas 25 linhas, existe uma linha especial que é a prompt de interacção. É nesta linha que o utilizador introduz os comandos do ambiente que levarão à criação e execução de programas.

Na figura 1 podemos ver o aspecto geral do ambiente.

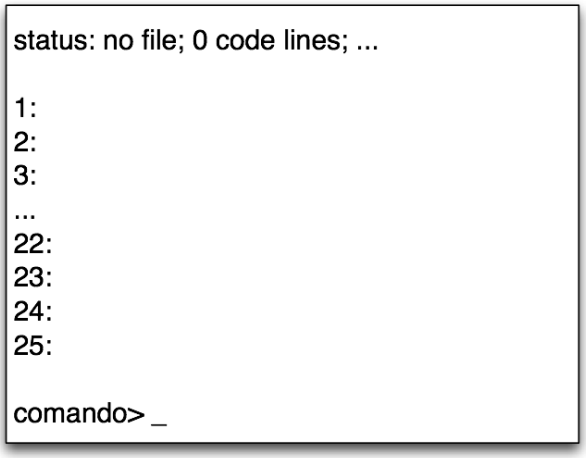

Figura 1: Estado geral do ambiente de edição

No início, se o sistema for invocado sem nenhum programa:

```
$msp
```

O sistema deverá aparecer com o aspecto da figura 1.

No entanto, se o sistema for invocado com um programa:

```
$msp prog1.msp
```

O ambiente deverá apresentar as primeiras 25 linhas do programa (se este tiver mais de 25) e reflectir na linha de status a informação relativa ao programa que foi carregado (como se pode ver na figura 2).

```
status: prog1.msp; 10 code lines; ...

 1: MEMORIA DE DADOS
 2: x 0 TAM 1 ; declaracao de x
 3: CODIGO
 4: PSHA x    ; endereço de x na stack
 5: INC       ; lê um carácter para a stack
 6: STORE     ; armazena em x o valor lido
 7: PSHA x    ; endereço de x na stack
 8: LOAD      ; coloca na stack o valor de x
 9: OUTC      ; escreve no écran o valor no topo da stack
10: HALT
...
22:
23:
24:
25:

comando> _
```

Figura 2: Ambiente invocado com um programa: prog1.msp

Pode encarar este ambiente como um editor de texto parecido com o *vi* que acompanha normalmente o sistema operativo Linux.

O ambiente tem no topo uma linha de status para dar alguma informação ao utilizador: qual o ficheiro que está a ser editado, quantas linhas contem esse ficheiro,...

Em baixo tem uma linha especial que será o ponto de entrada de comandos por parte do utilizador. Será nesta linha que o utilizador introduzirá comandos aos quais o ambiente deverá reagir.

Também à semelhança do vi o ambiente terá dois tipos de comandos:

- um primeiro tipo correspondente ao carregamento de programas, armazenamento, execução de um programa, listagem de um programa, ...
- um segundo tipo correspondente aos comandos de edição de um programa: acrescentar uma linha, apagar linhas, posicionar em determinada linha, ...

Quando tiver tempo, poderá sofisticar a interface do sistema fazendo com que esta aceite abreviaturas dos comandos.

Estes dois tipos de comandos enumeram-se e descrevem-se nas subsecções seguintes.

3.2 Comandos de Operação

Nesta secção apresentam-se os comandos operacionais que o seu sistema deverá suportar. Tome as descrições seguintes como guias e não se deixe limitar por elas. **Use a imaginação...**

O sistema deverá aceitar os seguintes comandos operacionais:

load <ficheiro> Carregamento de programas: se o ficheiro existir na directoria corrente o ambiente deverá carregá-lo exibindo depois um écran semelhante ao mostrado na figura 2. O ambiente deverá reagir com uma mensagem de erro à inexistência do ficheiro na directoria corrente. No caso deste comando ser dado com um programa já a ser editado, o utilizador deverá ser consultado quanto ao armazenamento do programa que estava a ser editado antes do novo ser carregado.

save <ficheiro> Armazenamento de programas: o ambiente deverá gravar o programa corrente no ficheiro designado. Se o nome do ficheiro fornecido diferir do ficheiro que estava a ser editado, o sistema deverá assumir este como o ficheiro actual para edição e deverá actualizar a linha de status.

history Historial de comandos: à semelhança de um sistema Unix este comando permite listar todos os comandos introduzidos pelo utilizador até ao momento. Como resultado deste comando o écran deverá exibir os últimos 25 comandos dados pelo utilizador numerados de 1 a 25. Depois da listagem, a introdução de um '
n' na linha de comando fará regressar o sistema ao estado anterior.

com <n> Execução de comandos anteriores: o n corresponde ao número de ordem de um comando anterior (dado pelo número de linha do editor); como resultado o sistema deverá voltar a executar esse comando.

hsave <ficheiro> Armazenamento do historial num ficheiro: o ambiente deverá o historial no ficheiro designado (mais tarde poderá ser interessante olhar para a descrição da interacção ocorrida numa sessão).

clear Reset do sistema: após este comando o sistema volta ao seu estado inical descartando tudo o que está a ser feito (programa que se está a editar e a memória de todos os comandos dados até ao momento; eventualmente, poderá ser pedido ao utilizador que confirme a execução deste comando.

run Execução do programa que está no editor.

help Lista os comandos disponíveis.

exit Saída e abondono do sistema: todos os recursos alocados em tempo de execução deverão ser libertados.

3.3 Comandos de Edição

Estes comandos estão relacionados com a introdução/criação/alteração dos programas no sistema e têm uma estrutura diferente: começam todos por número de linha, a seguir têm o nome do comando seguido de opções caso as haja:

<int> go Posicionamento da janela de visualização: após este comando a primeira linha da janela deverá corresponder à linha com número: `<int>`.

<int> insert <comando msp> Inserção de uma linha nova: a nova linha deverá ser inserida imediatamente antes da linha com número `<int>` e o seu conteúdo será `<comando msp>`.

<int> append <comando msp> Acrescento de uma linha: a nova linha deverá ser acrescentada imediatamente a seguir à linha com número `<int>` e o seu conteúdo será `<comando msp>`.

<int> delete Apagar uma linha: a linha com número `<int>` deverá ser apagada.

<int> replace <comando msp> Substituir uma linha: a linha com número `<int>` é substituída pelo novo `<comando msp>` introduzido pelo utilizador.

3.4 Máquina Virtual de Stack

Numa Máquina de Stack, podem-se identificar os seguintes blocos:

Memória – Área lógica para armazenamento de instruções e valores. Divide-se em 3 blocos independentes com funcionalidades distintas:

- Memória de Programa (ou abreviadamente MProg);
- Memória de Dados (ou MDados);
- Memória de Trabalho (Stack).

Descodificador – Unidade encarregue de interpretar e mandar executar cada instrução;

Unidade Lógica e Aritmética – Unidade que tem a responsabilidade de efectuar as operações lógicas e aritméticas (na nossa máquina será materializada num conjunto de funções que efectuarão as operações referidas);

Input e Output – São duas posições de memória especiais, com endereços fixos e bem conhecidos: input e output; Têm a característica especial de estarem ligadas ao exterior, permitindo a entrada de valores para a máquina (input) e a saída de valores da máquina (output); em concreto, estas células de memória estão ligadas ao teclado (input) e ao monitor (output).

Bus Interno – É o "corredor"lógico que liga as várias unidades permitindo a circulação e comunicação de instruções e valores (na nossa máquina implementada em C não terá uma materialização concreta).

Instruction Pointer (IP) e Stack Pointer (SP) – São os únicos registos da nossa máquina. O IP contem sempre, num dado instante, o endereço da próxima instrução que deve ser executada (endereço de MProg). O SP contem o endereço da última posição ocupada na Stack(ou seja, a posição do valor no topo da Stack).

A figura 3 apresenta a arquitectura da Máquina de Stack de acordo com os blocos funcionais descritos anteriormente.

Cada célula de memória (MProg, MDados ou Stack) tem capacidade para armazenar 2 bytes (sequência de 16 bits) e está associada a um endereço que a identifica univocamente.

Um endereço é um número inteiro que tem o valor 0 para a primeira posição de memória e sofre incrementos de 1 para as posições seguintes até à última célula. Nesta máquina virtual

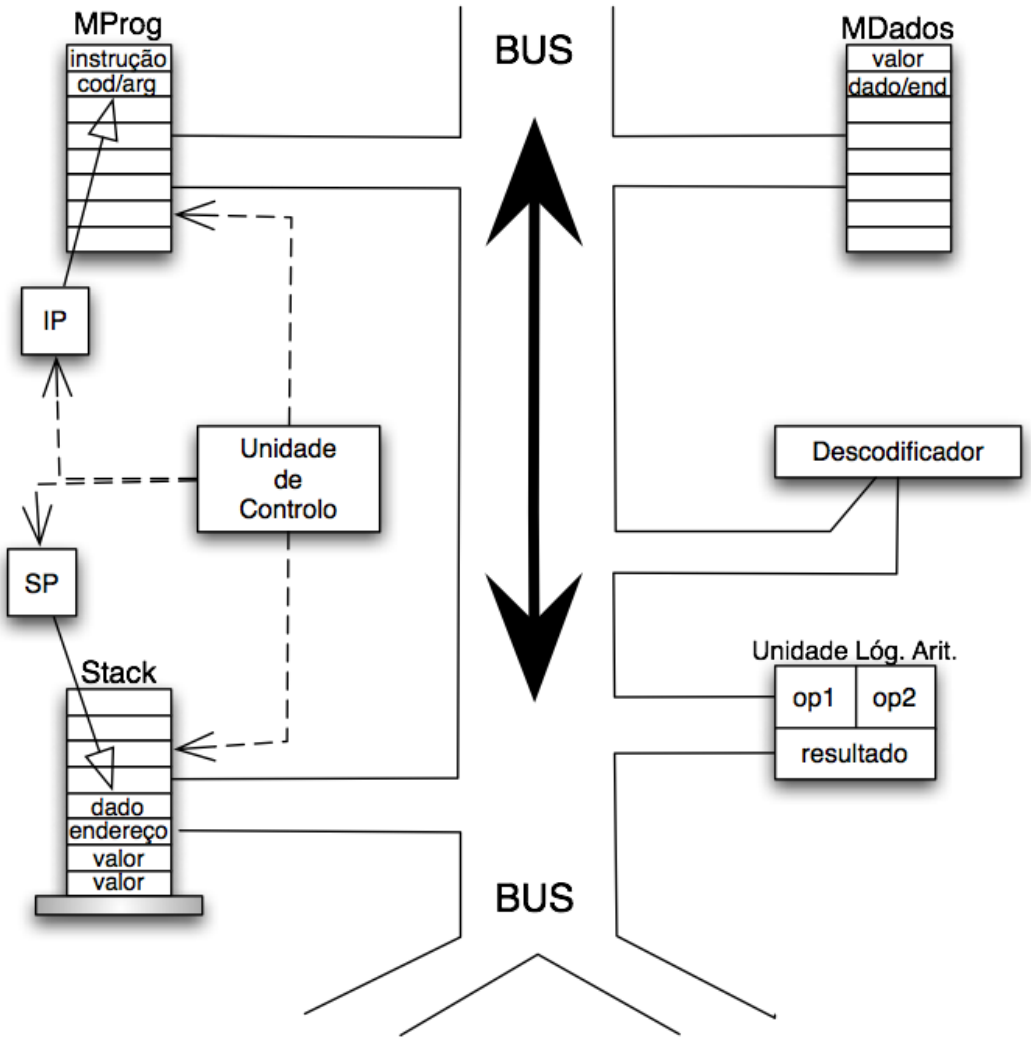

Figura 3: Arquitectura da Máquina de Stack

definiu-se o valor 65536 como o número total de células de memória. Portanto, a gama de endereços possíveis na nossa máquina virtual é dada pelo seguinte intervalo: [0, 65535].

O valor que é armazenado em determinada célula de memória pode representar uma instrução, um argumento de uma instrução, um valor inteiro (como veremos mais à frente num dos intervalos: [0, 65536] ou [-32767, +32767]), um carácter ou um componente de um endereço.

A interpretação do significado de cada valor é da responsabilidade da máquina e baseia-se no tipo de memória à qual se está a aceder em determinado instante:

Memória de Programa (MProg) – Armazena as instruções que vão ser executadas pela máquina. Um programa é uma sequência de instruções armazenadas na MProg. Uma célula da MProg pode conter o código de uma instrução ou o argumento de uma instrução. A maioria das instruções da linguagem MSP não contem argumentos. Os argumentos, se existirem, localizam-se nas posições seguintes à do código da instrução;

Memória de Dados (MDados) – Armazena valores. Esses valores correspondem aos dados do problema ou aos resultados calculados pelo programa. Cada valor ocupa dois bytes, podendo representar grandezas numéricas no intervalo [-32768,32767], ou caracteres (pelo seu código ASCII respectivo), ou ainda endereços de células da MDados;

Stack – Memória de trabalho semelhante no conteúdo à MDados. Contudo, é acedida e usada segundo uma filosofia LIFO ("Last In First Out"): os valores que se vão introduzindo na Stack ficam empilhados uns por cima dos outros, de tal forma que apenas se pode aceder (ler, consultar) o valor que está no cimo (topo) da Stack, e que corresponde ao último valor que lá foi armazenado. Assim, apenas é necessário conhecer um único endereço: o da última posição, ou topo da Stack. Considera-se que o topo é o endereço da Stack relativo ao último valor armazenado. O próximo valor a guardar na Stack será armazenado no endereço a seguir ao topo, e o primeiro valor a sair será sempre o do topo. A Máquina de Stack foi desenhada de tal forma que todas as operações esperam ter os seus operandos armazenados em posições consecutivas a partir do topo, retiram-nos de lá e, em sua substituição, coloca-se no topo o resultado da sua computação. Na Máquina de Stack, o topo "cresce do fim para o princípio", ou seja, do último endereço disponível (65535) em direcção ao primeiro (0).

3.5 Funcionamento

Para um programa ser executado o IP é carregado com o endereço de MProg onde está a primeira instrução a ser executada (como no nosso caso estamos a interpretar linha a linha o programa, basta a indicação do número de linha onde está a primeira instrução).

A partir daqui o funcionamento é sistemático: descodifica-se a instrução, dão-se as ordens necessárias para a executar, e coloca-se no IP a indicação da próxima instrução a executar (atenção que na maior parte dos casos esta será a instrução seguinte mas no caso de haver saltos poderá ser outra qualquer).

Este processo continua até que seja executada a instrução que manda parar a execução do programa.

Como exemplo de acções a realizar para executar uma dada instrução podemos apresentar entre outras as seguintes:

- transferência de dados entre o topo da Stack e uma das células especiais input ou output;
- realização de uma operação aritmética (adição, subtracção, multiplicação ou divisão) ou lógica (conjunção, disjunção ou negação);
- alteração da sequência normal de execução do programa forçando o IP a tomar um valor que não é o da linha seguinte.

3.6 Linguagem MSP

Relativamente à linguagem MSP, e quando se estiver a analisar a linguagem, convem estar atento aos seguintes pormenores:

- em cada linha da Zona de Dados (limitada por MEMORIA DE DADOS e por CODIGO) e da Zona de Código (limitada por CODIGO e estendendo-se até à última linha do texto do programa) não é permitida mais do que uma declaração de variável e de label respectivamente;
- também não é permitida mais de uma instrução em cada linha da Zona de Código;
- são permitidas linhas vazias ou de comentário em qualquer parte da Zona de Dados e de Código;
- os identificadoes de variáveis ou labels devem começar por letra podendo seguir-se-lhe mais letras, digitos ou o carácter underscore;
- a presença dos delimitadores das Zonas de Dados e Código (MEMORIA DE DADOS e CODIGO) é obrigatória mesmo que estas estejam vazias;
- um comentário começa pelo carácter ';' ao qual pode seguir-se qualquer sequência de caracteres que não inclua o fim de linha, que termina o comentário;
- em números positivos o sinal '+' é opcional; nos números negativos o sinal '-' é obrigatório.

3.6.1 Sintaxe e Semântica

Zona de Dados: declaração de variáveis

A Zona de Dados é iniciada pelas palavras reservadas MEMORIA de DADOS e prolonga-se até ao início da Zona de Código, iniciada pela palavra reservada CODIGO.

É na Zona de Dados que se declaram e podem inicializar as variáveis que irão ser utilizadas pelo programa.

A declaração de uma variável tem a seguinte forma:

```
1    identificador-variável endereço TAM tamanho VAL valor1 valor2 ... valorn
```

Por exemplo:

```
1    x 100 TAM 20 VAL 4
```

Declara uma variável com identificador "x", no endereço 100, com tamanho 20 e com o primeiro valor inicializado com 4 (os valores nos endereços 101 ao 119 são inicializados por omissão com o valor 0).

A semântica das declarações compreende as seguintes características:

- cada identificador de variável é único, não podendo haver mais que uma variável com o mesmo identificador;
- não é obrigatório declarar variáveis contiguamente, a partir do endereço 0, podendo-se deixar buracos na Memória de Dados;
- o espaço reservado para uma variável não pode colidir com o reservado para outra, i.e., os intervalos [endereço, endereço+tamanho-1] têm de ser mutuamente exclusivos para todas as variáveis declaradas;
- a inicialização do espaço alocado a uma variável é facultativa, sendo garantida, por omissão, a inicialização desse espaço com o valor 0;
- a iniciliazção das variáveis pode ser feita com qualquer valor inteiro no intervalo [-128, 255];
- é possível não declarar qualquer variável na Memória de Dados; neste caso, qualquer acesso que se faça a esta Memória deverá ter em conta uma inicialização a 0 desse espaço.

Zona de Código: instruções

A Zona de Código inicia-se pela palavra reservada CODIGO e prolonga-se até ao fim do texto.

O conjunto das instruções pode-se dividir em três grupos de acordo com as respectivas funcionalidades: instruções para manipulação de valores e endereços, instruções aritméticas e lógicas e instruções de controlo da sequência de execução do programa.

Manipulação de Valores e Endereços

PUSH <valor> Coloca valor no topo da Stack;

PSHA <endereço ou identificador de variável> Coloca o endereço fornecido ou o endereço da variável correspondente ao identificador fornecido no topo da Stack;

LOAD Retira da Stack um endereço e vai buscar a essa posição de memória o valor lá armazenado que coloca no topo da Stack;

LDA Retira da Stack um endereço e vai buscar a essa posição de memória um endereço que coloca no topo da Stack;

STORE Retira da Stack primeiro um valor, depois um endereço e coloca na posição da Memória de Dados referente ao endereço o valor retirado inicialmente da Stack;

STRA Retira da Stack primeiro um endereço, depois outro endereço e coloca na posição da Memória de Dados referente ao segundo endereço o primeiro endereço retirado da Stack;

IN Coloca no topo da Stack o conteúdo da posição especial input resultante da leitura de um valor via teclado;

OUT Escreve no monitor o valor que é retirado do topo da Stack;

INC Coloca na Stack o conteúdo (código ASCII) da posição especial input, resultante da leitura de um carácter via teclado;

OUTC Escreve no monitor o carácter cujo código ASCII é retirado da Stack;

Observações: as instruções PUSH, LOADe STORE operam sobre valores inteiros enquanto que PSHA, LDA e STRA operam sobre endereços.

Instruções Aritméticas e Lógicas

ADD Retira dois inteiros da Stack, soma-os e coloca na Stack o resultado;

SUB Retira dois inteiros da Stack, subtrai o segundo ao primeiro e coloca na Stack o resultado;

MUL Retira dois inteiros da Stack, multiplica-os-os e coloca na Stack o resultado;

DIV Retira dois inteiros da Stack, divide o primeiro pelo segundo e coloca na Stack o resultado;

ADDA Retira um inteiro e um endereço da Stack; soma o inteiro ao endereço calculando um novo endereço que é por fim colocado na Stack;

AND Retira dois inteiros da Stack, calcula a sua conjunção lógica e coloca na Stack o resultado;

OR Retira dois inteiros da Stack, calcula a sua disjunção lógica e coloca na Stack o resultado;

NOT Retira um inteiro da Stack, calcula a sua negação lógica e coloca na Stack o resultado;

EQ Retira dois inteiros da Stack, verifica se são iguais e coloca na Stack o resultado da comparação;

NE Retira dois inteiros da Stack, verifica se são diferentes e coloca na Stack o resultado da comparação;

LT Retira dois inteiros da Stack, verifica se o primeiro é menor que o segundo e coloca na Stack o resultado dessa comparação;

LE Retira dois inteiros da Stack, verifica se o primeiro é menor ou igual que o segundo e coloca na Stack o resultado dessa comparação;

GT Retira dois inteiros da Stack, verifica se o primeiro é maior que o segundo e coloca na Stack o resultado dessa comparação;

GE Retira dois inteiros da Stack, verifica se o primeiro é maior ou igual que o segundo e coloca na Stack o resultado dessa comparação;

ANDB Retira dois inteiros da Stack, calcula a sua conjunção bit-a-bit e coloca na Stack o resultado;

ORB Retira dois inteiros da Stack, calcula a sua disjunção bit-a-bit e coloca na Stack o resultado;

NOTB Retira um inteiro da Stack, calcula a sua negação bit-a-bit e coloca na Stack o resultado.

Observações:

- Em todas as operações binárias o segundo operando é retirado da Stack antes do primeiro;
- À excepção de ADDA, cujo resultado é um endereço, todas as outras operações resultam num inteiro;
- Nas operações lógicas, o resultado é 0 (FALSO) ou 1 (VERDADEIRO).

Instruções de Controlo

JMP <id-label ou endereço ou endereço-relativo> Salta incondicionalmente para o endereço calculado a partir do argumento (i.e., coloca em IP esse endereço); a execução do programa continua com o código armazenado no endereço da MProg especificado por id-label, ou no endereço (em valor absoluto relativo ao início da MProg) ou somando ao IP o endereço-relativo passado como argumento; neste último caso, o valor do argumento deverá obrigatoriamente conter o sinal + ou - antes do valor numérico;

JMPV <id-label ou endereço ou endereço-relativo> Retira o valor no topo da stack e testa-o (assumindo que lá está um valor Booleano): se for Verdadeiro salta para o endereço calculado a partir do argumento; se for Falso a execução continua na instrução apontada pelo valor do IP;

CALL <id-label ou endereço ou endereço-relativo> Guarda na Stack o endereço da instrução que se lhe segue e prossegue executando a instrução que se encontra no endereço passado como argumento (Chamada incondicional de uma subrotina);

RET Efectua o retorno de uma subrotina: a execução do programa continua na instrução que está no endereço que é retirado do topo da Stack;

HALT Pára a execução do programa;

NOOP Não faz nada, serve apenas para gastar tempo...

Observações:

- Um endereço absoluto é um endereço relativo ao iníco da MProg (endereço 0);
- Quando às instruções JMP, JMPF e CALL é passado um endereço relativo, o respectivo endereço calcula-se somando ao endereço da instrução seguinte o off-set que foi passado;
- A instrução HALT deve ser usada para terminar os programas.

4 Gestão de Campeonatos de Orientação

Um campeonato de Orientação é constituído por um conjunto de provas.

Cada atleta é livre de participar nas provas que entender. Para a classificação no campeonato irão contar os melhores **N** resultados que o atleta conseguir, em que **N** está definido de início pelo regulamento do campeonato.

Neste projeto, terás de desenvolver uma aplicação para cálculo do ranking final de um campeonato atendendo aos requisitos que a seguir se vão descrever.

4.1 Formato dos ficheiros de resultados

Um ficheiro com resultados é produzido em cada prova. Estes ficheiros têm a seguinte estrutura:

```
Stno;Chip;Database Id;Surname;First name;YB;S;Block;nc;Start;Finish;Time;...;
5063;9992323;5063;Castro;Sandro;72;M;5;0;52:16:00;01:07:50;15:34;0;52;...
90051;424534;467;Silva;Rui;98;M;5;0;44:21:00;01:00:28;16:07;0;6604;DIF;...
...
```

Ou seja, são ficheiros CSV (formato do MS Excel) em que a primeira linha identifica os campos de cada registo (cada identificador está separado do seguinte por ';'), e as linhas seguintes correspondem cada uma a um registo (o ';' continua aqui a ser o separador de campo).

4.2 Configuração da aplicação

Para além destes ficheiros e antes de os processar a aplicação deverá ler um ficheiro de configuração que irá definir o processamento a realizar: campos a extrair para o resultado final, o **N**úmero de provas que pontuam, etc.

Eis um exemplo de um ficheiro de configuração:

```
=titulo= VII Torneio COMmapa 2014
=nprovas= 8
=N= 6
=campos= First name; Surname; Time; Short; Long
...
```

titulo – indica a designação do torneio;

nprovas – indica o número de provas que se irão realizar durante o torneio;

N – indica o número de melhores resultados que serão contabilizados na classificação final do atleta;

campos – indica quais, e a ordem em que deverão aparecer na página resultado a ser produzida, os campos a serem retirados do ficheiro CSV;

outros – outros parâmetros de configuração que achar por bem incluir.

Outra alternativa seria:

```
=titulo= VII Torneio COMmapa 2014
=nprovas= 8
=N= 6
=campos= $5; $4; $12; $19; $20
...
```

Aqui os campos são designados pela sua posição.

4.3 Funcionamento

A aplicação a desenvolver deverá ter um ambiente de consola com alguns comandos básicos:

Carregar configuração – o sistema deverá ser sempre inicializado com esta operação; posteriormente, esta operação deverá estar sempre disponível para ser possível alterar a configuração;

Carregar base de dados – o sistema irá carregar para memória o estado de um torneio previamente gravado;

Carregar resultado de prova – o sistema irá carregar e processar um ficheiro CSV; como resultado desta operação deverá produzir um ficheiro HTML com a pontuação de cada atleta, ordenado por ordem decrescente de pontos;

Calcular ranking – em qualquer momento poderá ser invocada esta operação à qual o sistema deverá responder com uma listagem em HTML do ranking;

Gravar base de dados – o sistema deverá gravar em memória secundária (num ficheiro ou em vários ficheiros, defina como vai guardar a informação) o estado atual da aplicação;

Sair – o sistema terminará a aplicação, se o estado atual não tiver sido previamente gravado deverá lançar um alerta e pedir confirmação para prosseguir.

4.4 Cálculo da pontuação

Para cálculo da pontuação de um atleta numa prova deve-se usar o seguinte algoritmo:

- O atleta com melhor tempo recebe 100 pontos;
- Os restantes atletas recebem: $tempo_{melhor}/tempo_{atleta} * 100$.

Por exemplo, se o atleta A numa prova fez 50 minutos e foi o melhor, recebe 100 pontos. Um atleta B que faça 100 minutos irá receber 50 pontos.

Para cálculo do ranking dos atletas, somam-se as **N** melhores pontuações obtidas por cada atleta.

4.5 Requisitos

O trabalho a desenvolver deverá compreender as seguintes tarefas:

- Especificação de uma gramática para a linguagem de comandos da consola;
- Especificação de uma gramática para a linguagem de especificação de configurações;
- Especificação de uma gramática para os ficheiros de resultados;
- Desenvolver os respetivos parsers;
- Adicionar as ações semânticas necessárias.

5 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.

```
6       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).

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

9       Title --> BTITLE texto ETITLE
10      SubTitle --> BSUBTITLE texto ESUBTITLE
11                 | &
12      Authors --> Authors Author
13                | Author
14      Author --> BAUTHOR Name Nident? Email? Url? Affilliation? EAUTHOR
15      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).

```
16      Abstract --> BABS ParaList EABS
17      Aknowledgements --> BAKN ParaList EAKN

18      Toc --> TOC | &
19      Lof --> LOF | &
20      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.

```
21      Body --> BBODY ChapterList EBODY
22      ChapterList --> ChapterList Chapter
23                    | 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`).

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

25      ElemList --> ElemList Elem
26                 | 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.

```
27      Elem --> Paragraph
28             | Float
29             | List
30             | CodeBlock
31             | Section
32             | 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, ...

```
33      Paragraph --> BPARA ParaContent EPARA
34      ParaContent --> ParaContent texto
35                    | ParaContent FreeElement
36                    | &
37      FreeElement --> Footnote
38                    | Ref
39                    | Xref
40                    | Citref
41                    | Iterm
42                    | Bold
43                    | Italic
44                    | Underline
45                    | InlineCode
46                    | Acronym

47      Ref --> BREF target EREF
48      Xref --> BXREF target EXREF
49      Citref --> BCIT target ECIT
50      Iterm --> BITERM texto EITERM

51      Bold --> BBOLD BContent EBOLD
52      BContent --> BContent texto
53                 | BContent Italic
54                 | BContent Underline
55                 | &

56      Italic --> BITALIC IContent EITALIC
57      IContent --> IContent texto
58                 | IContent Bold
```

```
59              | IContent Underline
60              | &

61      Underline --> BUNDERLINE UContent EUNDERLINE
62      UContent --> UContent texto
63              | UContent Bold
64              | UContent Italic
65              | &
```

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

```
66      Float --> Figure
67              | Table

68      Figure --> BFIG Graphic Caption EFIG
69      Graphic --> BGRAPH path format? EGRAPH
70      Caption --> BCAPTION texto ECAPTION

71      Table --> BTABLE Caption TRowList
72      TRowList --> TRowList TRow
73              | TRow
74      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?

6 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.

6.1 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).

6.2 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
```

6.3 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.

6.3.1 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.

6.3.2 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 :

```
1  ./autor
2      ou
3  autor
```

Seleccionar o elemento raíz (report) deste documento :

```
1  /report
```

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

```
1  //autor
```

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

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

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

```
1  .//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 :

```
1  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 :

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

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

```
1  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
:

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

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

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

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

```
1  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
:

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

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

```
1  *[@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.

7 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.