Linguagens Formais e Autómatos, Compiladores
Guião das Aulas Práticas

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

2019–2020, 2^o semestre

Bloco 1

Programação em Java

Resumo:

- Programação em Java.
- Estruturas de Dados.
- Recursividade.

Exercício 1.01
Crie uma calculadora simples que leia (do dispositivo de entrada) operações matemáticas como

```
12.3 + 7.2
```

e escreva o resultado respectivo (`19.5` neste exemplo).

As operações serão sempre do género `<número> <operador> <número>`, com as três partes separadas por espaços ou em linhas diferentes. Implemente as quatro operações básicas usando os operadores `+`, `-`, `*` e `/`. Note que o operador é uma palavra (string) que contém apenas um símbolo. Se for introduzido um operador inválido, deve escrever uma mensagem apropriada para o dispositivo de saída de erros (`System.err`).

Exercício 1.02
Construa uma calculadora com as quatro operações aritméticas básicas que funcione com a notação pós-fixa (*Reverse Polish Notation*[1]). Esta notação dispensa a utilização de parêntesis e tem um implementação muito simples assente na utilização de uma pilha de números reais. Sempre que aparece um operando (número) ele é carregado para a pilha. Sempre que aparece um operador (binário), são retirados os dois últimos números da pilha (se não existirem temos um erro sintáctico na expressão) e o resultado da operação é colocada na PILHA.

[1]Nesta notação os operandos são colocados antes do operador. Assim `2 + 3` passa a ser expresso por `2 3 +`.

Implemente este programa por forma a que os operandos e os operadores sejam palavras (strings separadas por espaços) lidas do *standard input*.

Exemplo de utilização:

```
$ echo "1 2 3 * +" | java -ea b1_2
Stack: [1.0]
Stack: [1.0, 2.0]
Stack: [1.0, 2.0, 3.0]
Stack: [1.0, 6.0]
Stack: [7.0]
```

A biblioteca nativa `Java` implementa pilhas com a classe `java.util.Stack`.

Exercício 1.03

O ficheiro `numbers.txt` contém uma lista de números com as suas representações numéricas e as suas descrições por extenso.

Fazendo uso de um *array* associativo, escreva um programa que traduza, palavra a palavra, todas as ocorrências por extenso de números pelo respectivo valor numérico (mantendo todas as restantes palavras). Exemplo de utilização:

```
$ echo "A list of numbers: eight million two hundred thousand five hundred twenty-four" | java -ea b1_3
A list of numbers: 8 1000000 2 100 1000 5 100 20 4
```

Exercício 1.04

Utilizando o *array* associativo do exercício anterior, construa um programa que converta um texto representando um número, para o respectivo valor numérico. Por exemplo:

```
$ echo "eight million two hundred thousand five hundred twenty-four" | java -ea b1_4
eight million two hundred thousand five hundred twenty-four -> 8200524

$ echo "two thousand and thirty three" | java -ea b1_4
two thousand and thirty three -> 2033
```

Tenha em consideração as seguintes regras na construção do algoritmo[2]:

- Os números são sempre descritos partindo das maiores ordens de grandeza para as mais pequenas (*million, thousand, ...*);
- Sempre que descrições consecutivas de números se fazem por ordem crescente (*eight million*, ou *two hundred thousand*), o valor respectivo vai sendo acumulado por multiplicações sucessivas ($8 * 1000000$, e $2 * 100 * 1000$);
- Caso contrário, o valor acumulado é somado ao total.

Não tenha em consideração o problema de validar a sintaxe correcta na formação de números (por exemplo: *one one million*, *eleven and one*, ...).

[2]Utilize este algoritmo simplificado, mesmo sabendo que não funciona para todos os casos.

Exercício 1.05

Modifique o exercício 1.01 por forma a poder definir e utilizar variáveis numéricas. Por exemplo:

```
n = 10
4 * n
n = n + 1
n + 5
```

Utilize um *array* associativo para armazenar e aceder aos valores das variáveis (a biblioteca `Java` tem a interface `java.util.Map` para esse fim, fornecendo implementações em `java.util.Hashtable` ou `java.util.HashMap`).

Exercício 1.06

Considere uma tabela de correspondências de palavras em duas línguas, como a que se segue:

`dic1.txt`

```
armas guns
barões barons
as the
os the
e and
```

Pretende-se fazer um programa (`b1_6.java`) que num texto dado, substitua cada ocorrência de uma palavra conhecida pela sua "tradução". Por exemplo, usando a tabela acima, o texto: "as armas e os barões assinalados" seria traduzido em "the guns and the barons assinalados".

a) Comece por escolher uma estrutura de dados adequada para armazenar a tabela de correspondências e crie uma função que preencha essa estrutura com as correspondências lidas de um ficheiro. Detalhes:

- A tabela de correspondências é um ficheiro com uma correspondência por linha.
- A primeira palavra da linha é a versão original e o resto é a tradução respectiva, composta por uma palavra ou mais.
- Considera-se "palavra" qualquer sequência de caracteres delimitada por espaços.

b) Complete o programa para fazer o pretendido, tendo em consideração que:

- Cada palavra deve ser substituída pela sua correspondência, quando exista tradução; ou mantida igual à original, no caso contrário.
- O programa recebe o ficheiro de correspondências como primeiro argumento, sendo os restantes argumentos os ficheiros de entrada. A tradução de todos esses ficheiros deve ser escrita no dispositivo de saída.

- Cada linha dos textos de entrada deve produzir uma linha à saída.

c) Considere uma tabela de correspondências como a seguinte:

dic2.txt	
armas	dispositivos de combate
barões	nobres que se distinguiram em combate
combate	batalha ou guerra
guerra	conflito armado

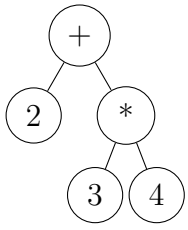
Altere o programa anterior de forma que cada palavra é substituída pela sua definição, mas com as palavras da definição também substituídas sucessivamente até que o resultado contenha apenas palavras não definidas.

Exemplo: "as armas e os barões assinalados" → "as dispositivos de batalha ou conflito armado e os nobres que se distinguiram em batalha ou conflito armado assinalados"

Exercício 1.07
Podemos escrever expressões aritméticas três formas distintas consoante a posição do operador[3]:

- *Notação infixa*: `2 + 3` ou `2 + 3 * 4` ou `3 * (2 + 1) + (2 - 1)`
- *Notação prefixa*: `+ 2 3` ou `+ 2 * 3 4` ou `+ * 3 + 2 1 - 2 1`
- *Notação sufixa*: `2 3 +` ou `2 3 4 * +` ou `3 2 1 + * 2 1 - +`

Independentemente da notação utilizada, uma expressão aritmética pode ser representada por uma árvore binária em que os nós representam as (sub)expressões existentes (os números serão sempre folhas da árvore). Por exemplo a expressão (prefixa) `+ 2 * 3 4` é expressa pela árvore binária:



A geração desta árvore binária tendo como entrada uma expressão em notação *prefixa* é bastante simples (se feita recursivamente):

[3]No exercício 1.02 já utilizámos uma destas notações (sufixa) como forma de simplificar o cálculo de expressões numéricas.

```
createPrefix()
{
   if (in.hasNextDouble()) // next word is a number
   {
      // leaf tree with the number
   }
   else // next word is the operator
   {
      // tree with the form: operator leftExpression rightExpression
      // leftExpression and rightExpression can also be created with createPrefix
   }
}
```

a. Implemente um módulo – `ExpressionTree` – que cria uma árvore binária a partir de uma expressão em notação *prefixa* para os 4 operadores aritméticos elementares (+, −, ⋆ e /)[4];

b. Implemente um novo serviço no módulo – `printInfix` – que escreve a expressão (já lida) na notação *infixa*;

c. Implemente o módulo de uma forma robusta por forma a detectar expressões inválidas (note que a responsabilidade por uma expressão inválida é exterior ao programa pelo que deve fazer uso de uma aproximação defensiva);

d. Implemente um novo serviço no módulo – `eval` – que calcula o valor da expressão.

[4]Considere que cada número e operador é uma palavra a ser lida no *standard input*