Compilers
Guide to Practical Classes

Department of Electronics. Telecommunications and Informatics
University of Aveiro

2023–2024, 2^{nd} semester

Block 1

Java Programming

Summary:

- Programming in Java.
- Data Structures.
- Recursion.

Exercise 1.01
Create a simple calculator that reads (from standard input) math operations like

```
12.3 + 7.2
```

and write the respective result (`19.5` in this example).

The operations will always be like `<number> <operator> <number>`, with the three parts separated by spaces or on different lines. Implement the four basic operations using the operators `+`, `-`, `*` and `/`. Note that the operator is a word (string) that contains only one symbol. If an invalid operator is entered, you must write an appropriate message to the error output device (`System.err`).

Exercise 1.02
Modify the previous exercise so that you can define and use numerical variables. For example:

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

Use an associative array to store and access variable values (The `Java` library has the `java.util.Map` interface for this purpose, providing a possible implementation in: `java.util.HashMap`).

Exercise 1.03
Build a calculator with the four basic arithmetic operations that works with postfix notation (*Reverse Polish Notation*[1]). This notation dispenses with the use of parentheses and has a very simple implementation. based on the use of a stack of real numbers. Whenever an operand appears (number) it is pushed onto the stack. Whenever an operator (binary) appears, the last two numbers from the stack are removed (if they do not exist we have an error syntax in the expression) and the result of the operation is pushed onto the STACK.

Implement this program so that the operands and operators are words (space separated strings) read from *standard input*.

Usage example:

```
$ 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]
```

The native `Java` library implements stacks with the `java.util.Stack` class.

Exercise 1.04
The `numbers.txt` file contains a list of numbers with their numerical representations and their descriptions in full.

Using an associative array, write a program that translates, word by word, all occurrences of numbers in words by their numerical value (keeping all other words). Usage example:

```
$ 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
```

Exercise 1.05
Using the associative array from the previous exercise, build a program that converts a text representing a number, to the respective numerical value. For example:

```
$ 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
```

Keep the following rules in mind when building the algorithm[2]:

- Numbers are always described starting from the largest orders of magnitude to the smallest (*million, thousand, ...*);

[1]In this notation operands are placed before the operator. So `2 + 3` becomes be expressed by `2 3 +`.
[2]Use this simplified algorithm, even though it doesn't work for all cases.

- Whenever consecutive descriptions of numbers are made in ascending order (*eight million*, or *two hundred thousand*), the respective value is accumulated by successive multiplications ($8 * 1000000$, and $2 * 100 * 1000$);
- Otherwise, the accumulated value is added to the total.

Do not take into account the problem of validating the correct syntax in the formation of numbers (for example: *one one million*, *eleven and one*, ...).

Exercise 1.06
Consider a table of word correspondences in two languages, like the one below:

```
dic1.txt
armas guns
barões barons
as the
os the
e and
```

The intention is to write a program (`b1_6.java`) that, in a given text, replaces each occurrence of a known word by its "translation". For example, using the table above, the text: "as armas e os barões assinalados" would be translated into "the guns and the barons assigned".

a) Start by choosing a suitable data structure to store the table of correspondences and create a function that fills this structure with the correspondences read from a file. Details:
   - The match table is a file with one match per line.
   - The first word of the line is the original version and the rest is the respective translation, consisting of one word or more.
   - "word" is any string of characters delimited by spaces.

b) Complete the program to do what you want, taking into account that:
   - Each word must be replaced by its correspondence, when there is a translation; or kept the same as the original, otherwise.
   - The program receives the correspondence file as the first argument, the remaining arguments being the input files. The translation of all these files must be written to the output device.
   - Each line of input strings must produce one line of output.

c) Consider a table of correspondences like the following:

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

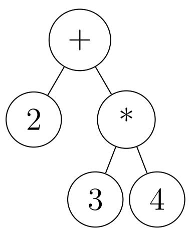
Change the previous program so that each word is replaced by its definition, but with the definition words also replaced successively until the result contains only undefined words.

Example: “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”

Exercise 1.07
We can write arithmetic expressions in three different ways depending on the operator position[3]:

- *Infix notation*: `2 + 3` or `2 + 3 * 4` or `3 * (2 + 1) + (2 - 1)`
- *Prefix notation*: `+ 2 3` or `+ 2 * 3 4` or `+ * 3 + 2 1 - 2 1`
- *Suffix notation*: `2 3 +` or `2 3 4 * +` or `3 2 1 + * 2 1 - +`

Regardless of the notation used, an arithmetic expression can be represented by a tree binary in which the nodes represent the existing (sub)expressions (the numbers will always be leaves of the tree). For example the expression (prefix) `+ 2 * 3 4` is expressed by the binary tree:



The generation of this binary tree having as input an expression in *prefix* notation is quite simple (if done recursively):

[3]In the exercise 1.03 we already used one of these notations (suffix) as way to simplify the calculation of numeric expressions.

```
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. Implement a module – `ExpressionTree` – that creates a binary tree from an expression in *prefix* notation for the 4 arithmetic operators elementary (`+`, `-`, `*` and `/`)[4];

b. Implement a new service in the module – `printInfix` – that writes the expression (already read) in *infix* notation;

c. Implement the module in a robust way to detect invalid expressions (note that the responsibility for an invalid expression is outside the program for what it must do use of a defensive approach);

d. Implement a new service in the module – `eval` – that calculates the value of the expression.

[4]Consider that each number and operator is a word to be read into *standard input*