Compilers
Guide to Practical Classes

Department of Electronics. Telecommunications and Informatics
University of Aveiro

2022–2023, 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*

Block 2

Introduction to ANTLR

Summary:

- Installation and use of ANTLR.
- Construction of simple grammars.
- Introduction to programming in ANTLR.

Installation of ANTLR4

Download the file `antlr4-install.zip` available in elearning, extract its contents, open a terminal in the `antlr4-install` directory and finally run the command `./install.sh`.

Exercise 2.01

Define the following grammar in the file `Hello.g4`[1]:

```
grammar Hello;              // Define a grammar called Hello
greetings : 'hello' ID ;    // match keyword hello followed by an identifier
ID : [a-z]+ ;               // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ;   // skip spaces, tabs, newlines, \r (Windows)
```

a) Try to compile the `Hello.g4` grammar, and run the resulting translator (use the command `antlr4-test` to test the grammar).

b) Add a *visitor* (`Execute`) so that, at the end of the rule `greetings`, write in Portuguese: `Olá <ID>`.

[1] Do not copy the code from the pdf file, as some copied characters may not be ASCII (generating errors when compiling with antlr4).

You can generate a new *visitor* in two ways:

(a) Run the following commands:

```
antlr4-visitor Hello.g4
mv HelloBaseVisitor.java Execute.java
```

, and then change the contents of the `Execute.java` file (change the class declaration so that `Execute` extend the `HelloBaseVisitor` class, and replace the generic type `T`, with the desired data type, which, in this case, is `String`).

(b) Or, alternatively, run the command:

```
antlr4-visitor Hello Execute String
```

Note that `ANTLR` for each syntactic rule `r: a b;`, creates a method `visitR` which has as argument the context of that rule (`*Parser.RContext ctx`). This context is defined by the `*Parser.RContext` class which, if `a` and `b` are non-terminal, contains methods that return the respective `*Parser.AContext` and `*Parser.BContext` contexts. If you want to visit one of these contexts, simply call, in the `visitR` method, for example, `visit(ctx.a())`.

c) Append a farewell rule to the grammar (`bye ID`), causing the grammar to accept either rule (`greetings`, or `bye`)[2]. (To define a rule with more than one alternative in `ANTLR`, use the separator `|`, for example: `r: a | b ;`.)

d) Generalize identifiers to include capital letters and allow names with more than one identifier. (In `ANTLR` a rule with one or more repetitions and defined by `r+`, for example: `r: a+ ;`.)

e) Generalize the grammar in order to allow repetition until the end of the rules file of any of the rules described above (`greetings`, or `bye`).

Exercise 2.02

Consider the following grammar (`SuffixCalculator.g4`):

```
grammar SuffixCalculator;
program:
      stat* EOF       // Zero or more repetitions of stat
    ;
stat:
      expr? NEWLINE  // Optative expr followed by an end-of-line
    ;
expr:
      expr expr op=('*'|'/'|'+'|'-')  #ExprSuffix
    | Number                          #ExprNumber
    ;
Number: [0-9]+('.'[0-9]+)?; // fixed point real number
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
```

[2]As a response to *bye*, write in Portuguese: `Adeus <ID>`.

a) Try compiling this grammar and running the resulting translator.

b) Using this grammar and adding a *visitor* (`Interpreter`), try to implement a calculator in postfix (suffix, or *Reverse Polish Notation*) notation for the defined elementary arithmetic operations (that is, that it performs the calculations and presents the results for each processed row).

   (Note that when implementing a *visitor* in ANTLR it is not necessary to apply the algorithm described in exercise 1.03 (which resorts to a *stack*).)

Note that in `ANTLR` we can associate *visits/callbacks* with different alternatives in a syntactic rule:

```
r : a #altA
  | b #altB
  | c #altC
  ;
```

In this case, *visits/callbacks* will be created both in *visitors* and in *listeners*, to the three alternatives presented, the *visit/callback* not appearing for the rule `r`.

Exercise 2.03

Consider the following grammar for an integer calculator (`Calculator.g4`):

```
grammar Calculator;

program:
      stat* EOF
    ;

stat:
      expr? NEWLINE
    ;

expr:
      expr op=('*'|'/'|'%') expr   #ExprMultDivMod
    | expr op=('+'|'-') expr       #ExprAddSub
    | Integer                      #ExprInteger
    | '(' expr ')'                 #ExprParent
    ;

Integer: [0-9]+;  // implement with long integers
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
COMMENT: '#' .*? '\n' -> skip;
```

a) Try compiling this grammar and running the resulting translator.

b) Using this grammar and adding a *visitor* (`Interpreter`), try to implement a calculator for the defined elementary arithmetic operations (that is, to carry out the calculations and present the results to each line processed).

c) Add the unary operators + and −.

Note that the ambiguity between binary and unary operators that use the same symbol (+ and −) can only be resolved syntactically, and that these operators must take precedence over all others (for example, in the expression `-3+4` the unary operator − applies to `3` and not to the sum result).

Exercise 2.04
Taking into account the grammar of the 2.02 exercise, develop a prefix calculator. That is, where the binary operator appears **before** the two operands (as with the entry in the 1.07 exercise). Implement the interpreter with a *visitor*.

Exercise 2.05
Implement a grammar to parse the files used in the 1.04 exercise. Using a *listener*, change the resolution of this problem to include this grammar in the solution.

Exercise 2.06
Change problem 2.03 by adding the possibility to define and use variables. To that end consider a new statement (i.e. a new alternative in `stat`):

```
stat: ···  // make necessary changes
assignment: ID '=' expr;
···
expr:
    ···
    | ID                          #ExprId
    ···
ID: [a-zA-Z_]+ ;
···
```

To support recording values associated with variables, use an associative array[3].

Exercise 2.07
Download the `Java` language grammar (`https://github.com/antlr/grammars-v4`), and try parsing simple `Java` programs (version 8).

Use support for *listeners* from `ANTLR` to write the name of class and methods subject to parsing.

[3] java.util.HashMap.

Exercise 2.08
Using the grammar defined in the 2.06 problem and using *visitors* from ANTLR, convert an infix arithmetic expression (operator in the middle of the operands), into an equivalent suffix expression (operator at the end). To resolve the ambiguity problem with the unary operators + and −, make it so that in the converted suffix expression, these operators appear, respectively, as `!+` and `!-` . For example:

- $2 + 3 \quad \rightarrow \quad 2\ 3\ +$
- $a = 2 + 3 * 4 \quad \rightarrow \quad a\ =\ 2\ 3\ 4\ *\ +$
- $3 * (2 + 1) + (2 - 1) \quad \rightarrow \quad 3\ 2\ 1\ +\ *\ 2\ 1\ -\ +$
- $3 * +(4/2) + (-2 - 1) \quad \rightarrow \quad 3\ 4\ 2\ /\ \ !+\ *\ 2\ !-\ 1\ -\ +$

Exercise 2.09
We intend to implement a calculator for rational numbers (numerator and denominator represented by integers).

Define a grammar for this language taking into account the following example program (arithmetic operators must have the usual precedence):

```
// basic:
print 1/4;   // escreve na consola a fracção 1/4
print 3;     // escreve na consola a fracção 3
3/4 -> x;    // guarda a fracção 3/4 na variável x
print x;     // escreve na consola a fracção armazenada na variável x
// more advanced:
print 1/4-(1/4);
4/2 * (-2/3 + 4) - 2 -> x;
print x;
print x*x:x+(x)-x;   // a divisão de fracções é feita pelo operador :
print (1/2)^3;       // operador potência para expoentes inteiros (base entre parêntesis)
print reduce 2/4;    // redução de fracções
```

Note 1: Try to make the best possible use of the exemplified instructions in order to make the language as generic as possible. However, you can consider that literal fractions (e.g. 1/4, 2, −3/2) are always either an integer (i.e. unit denominator), or a ratio between two literal integers (where only the numerator can be negative).

Note 2: There are files `*.txt` that exemplify several programs.

Implement this interpreter with *visitors*.