Practice 1

Introduction

The software to be used in the laboratory is Mathematica. The code to implement in the laboratory manipulates abstract objects such as finite automata, but both, the objects and the operations to implement, can be represented taking into account some simple operations. It is worth to be noted here that the target of the proposed exercises is not to obtain efficient implementations but to obtain some expertise on the abstract objects studied.

Mathematica is a very helpful package to develop general applications where mathematical, algebraic, numeric, symbolic or graphic operations play an important role. Mathematica is designed to deal with mathematic operations which usually have been implemented in FORTRAN. Most of the equations of the phisics-mathematics, as well as geometry, are already implemented in Mathematica. Furthermore, Mathematica has a wide set of graphical functions to plot the results obtained.

Mathematica applications involve almost all the research and development areas of knowledge, from engineering or economics to architecture. In all these areas Mathematica can be used as a working tool, as well as in the laboratory, whenever the subject to teach implies the use of applied mathematics.

Taking into account Mathematica as a programming language, it differentiates from FORTRAN, C or PASCAL in several aspects: first of all, as stated above, Mathematica deals easily with mathematic or symbolic expressions; and second, Mathematica is not compiled (although latest versions of Mathematica are able to generate C code) but interpreted. The main drawback of this is that, for a given algorithm, the running time of Mathematica is higher than in other languages. Nevertheless, the time needed to implement the same algorithm is much smaller in Mathematica than in any other (more efficient) language. All this allows to focus into conceptual details rather than on the implementation details.

Mathematica uses an interactive interface in which functions (modules) are not compiled but evaluated. Each one of the input expressions can be individually evaluated and used from that moment on. This allows to check each step of the computation, avoiding integration errors. Mathematica syntax is easy, and flexible and allows the connection with other software (although this will not be a target in this course). Mathematica is a Wolfram Research Inc. software and it is available in any platform. Mathematica code can be stored as an ASCII file or as Mathematica Notebooks (when using compatible versions of the software).

This first practice is devoted to the introduction of the programming language as well as to the implementation of some easy operations on strings or finite languages.

Before starting

The Mathematica front-end is the typical one of any application. Nevertheless, some hints to be considered are:

- Mathematica is case sensitive (a name identifier with different use of up or lower case lead to different identifiers)
- The introduction of an expression may imply several lines (separated by **intro**). The

evaluation of an expression is done when **shift+intro** are typed. All evaluated expressions are stored in the Mathematica kernel and can be used in successive evaluations

- Expressions are automatically numbered and organized into **cells**. Each evaluated expression has an input cell (In[..]) and an output cell (Out[..])
- While an expression is being evaluated it keeps a double bracket. A indefinite loop can be broken typing **Alt+.**. A Mathematica kernel can be erased using the **Quit Kernel** option in the **Evaluation** menu.
- Interactive help are provided by typing **?** followed by the command (the wild-card * can be used).
- Mathematica automatically indents the code while it is typed. Usually, bad indentation implies a syntactic error

Mathematica lists

Among the most important data structures in Mathematica (and the most widely used in this subject laboratory) are lists. For instance, we will use lists to represent strings of a given language, finite languages, or the transitions of finite automata that recognize such languages.

Lists are represented by a bracketed comma-separated sequence of elements, for instance: $list1 = \{a, b, \{c, d\}\}$ assigns to the identifier **list1** the list whose first element is a, the second one b, and the third one $\{c, d\}$. Note that lists can contain heterogeneous elements. Lists can be indexed using double square brackets. Thus, the i-the element of the list var can be accessed using $var[[i]]$. The index of the first element in a list is 1.

List operations

In order to save time, we will build automatically some lists. For this purpose we will use the instruction `l=Table[Random[Integer, {1, 9}], {i, 20}]` (**shift+intro**) that sets to the identifier l a random generated list of 20 numbers from 1 to 9 (note that both the range or the number of elements can be changed).

Let us build two lists **l1** and **l2** and test the following build-in functions:

Important: The following functions, unless otherwise stated, do not update any input parameter. To consider such update it is necessary to use an assignment to a variable. Note also that, in many cases, the following list show only one of the possible uses of the command. It is possible to find more information in the Mathematica help.

- `Length[l1]`: Length of the list
- `Join [l1, l2]`: Concatenation of two lists
- `Union[l1, l2]`: Returns a sorted list with the elements in $l1$ or $l2$
- `Intersection[l1, l2]`: Returns a sorted list with the elements which are into $l1$ and into $l2$

- `Complement[l1, l2]`: Returns a sorted list with the elements in $l1$ which are not in $l2$
- `Sort[l1]`: Returns the sorted version of $l1$
- `Reverse[l1]`: Returns the reverse of $l1$
- `First[l1]`: Returns the first element of the list
- `Rest[l1]`: Returns the list $l1$ without the first element
- `Take[l1, n]`: Returns a list with the first n elements in $l1$
- `Take[l1, {n,m}]`: Returns a list with the elements n through m of list elements in $l1$
- `Append[l1, x]`: Returns the list $l1$ with x added as the last element
- `Prepend[l1, x]`: Returns the list $l1$ with x added as the first element
- `AppendTo[l1, x], PrependTo[l1, x]`: Parameter-update versions of the previous functions fulfill the condition (indexes) of x in $l1$
- `MemberQ[l1,x]`: Returns $True$ if x is in $l1$ and $False$ otherwise

The Cases function

Due to its importance and wide use in this context, we highlight the following function:

`Cases[list, pattern]`: Returns a lists with the elements in $l1$ that fulfill the *pattern*. The wild-card _ can be used to set the pattern.

Example

`list={{a,a},{b,a},{b,b},{a,b}}`
`Cases[list,{a,_}]` returns $\{\{a,a\},\{a,b\}\}$
`Cases[list,{c,_}]` returns $\{\}$ (empty list)

Logical and relational operators

All these operations return $True$ or $False$. Its syntax is the following:

- Not `!`
- Conjunction `&&`
- Disjunction `||`
- Equal `==`
- Distinct `=!=`

- Relational operators (greater, lower, etc.) `>, <, >=, <=`.

Programming syntax

Mathematica has its own programming language that allows to implement new functions. Due to its interpreted nature, Mathematica allows the process of software development to be fully interactive. That is, each of the instructions of any function can be tested before the function is finished. For instance, it is possible to run a loop:

$$For[i = 1, i < 10, i++, Print[i]]$$

to print the first ten digits.

Modules

Mathematica modules give us the main tool to be used in the laboratory. Its general syntax is the following:

```
ModuleIdentifier[commalistofparameters]:=Module[{list of local variables},

      commands to run; (* separated by semicolons *)

      Return[expression]; (* whenever the module returns a value *)

] (* this is a comment that shows the end of the module *)
```

It is convenient to evaluate the module, the input variables, and the running of the module in separate cells.

It is also important to take into account that the initialization of the local variables cannot be carried out when they are defined. The use of undefined identifier in the code of a module implies the consideration of the identifier as a global variable.

Furthermore, it is convenient not to assign values to the input parameter identifiers. Whenever it is necessary to operate on these input values, it is possible to copy the parameter in a local variable using a simple assignment operation.

Example

Module that receives an integer and computes the sum of the integers lower or equals to the input value

```
Sum[n_] := Module [{i, sum},
      sum = 0;
      For[i = 0,i <= n,i++,
         sum = sum + i;
      ]; (* end for *)
      Return[sum];
] (* end Sum *)
```

A possible run of the module can be done by typing $sum[3]$ (which will return 6).

Conditional and loop commands

Control commands are considered as normal commands and therefore must end with a semicolon. The syntax is quite similar to other programming languages, but it is important to be aware of the differences.

- **Conditional**: `If[condition, commands-true, commands-false];`
- **Loop for**: `For[initialization, condition, increment, commands];`

  initialization is run. Then the loop begins by checking if the *condition* is true, in that case the commands are executed and the *condition* evaluated again. The loop ends when the *condition* is false.

- **Loop while:** `While[condition, commands];`

  The *condition* is evaluated and, if true, the commands executed. The loop ends when the *condition* is false.

In what follows an alphabet will be represented as a list of symbols, and a string will be represented as a list of symbols over some alphabet. Thus, the string *abbaca* will be represented as $\{a, b, b, a, c, a\}$ and the empty string will be represented as the string of length zero, that is $\{\}$.

A finite language is a set of strings. Therefore, it will be represented as a list of strings. For instance, the language $L = \{abba, bb\}$ will be represented as $\{\{a, b, b, a\}, \{b, b\}\}$. In the same way, the empty language will be represented as $\{\}$ (the language with no strings), and the language with only the empty string will be represented as $\{\{\}\}$.

Exercises

Exercise 1

Implement a Mathematica module, with input a string x and an integer n, that returns a list with the elements which appear at least n times in x

Hint: Consider the use of the **Count** built-in module whose use is described in the *Mathematica* help.

Exercise 2

Simulate the function `Length`

The expression $Length[l]$, where l is a list, returns the length of l. Implement (avoid using the function $Length$) an iterative module to simulate such function.

Hints:

- The empty list is represented by {}
- $l = Rest[l]$ updates the variable l to the list l without the first element
- Do recall the iterative command $While[condition, commands]$. Please be careful and avoid running an infinite loop. In that case **Alt + .** aborts the evaluation

Exercise 3

Implement a module that, taking into account an input list x, returns the list that contain first the elements of x in even positions and second, the elements that appear in odd positions of x.

Example: Given the string `{a,b,c,d,e}`, the module would return `{b,d,a,c,e}`.

Exercise 4

Implement a module that, given a string and two integers i and j, returns the string that considers the elements between the positions i and j in reverse order.

Example: Given the word `{a,b,c,d,e,f,g,h}` and the integers 3 and 6, the module would return `{a,b,f,e,d,c,g,h}`.

Exercise 5

Implement a Mathematica module, whose input is a string x and an integer n, to compute x^n

Exercise 6

Implement a Mathematica module that returns the segments of an input string x

Exercise 7

Implement a Mathematica module to return the product of two input finite languages

Exercise 8

Implement a Mathematica modue that, given a finite language L and a word u, returns the right quotient $u^{-1}L$.

Exercise 9

Implement a Mathematica module, whose input is a finite language L and an integer $n \geq 0$, that returns L^n

Exercise 10

Implement a Mathematica module that, given an input word x, return the symbol a such that there is a segment a^n in x where n is maximal.

Example: Given the word `{a,b,b,a,c,c,c,a,b,b}`, the module would return `c`.

Exercise 11

Implement a Mathematica module whose input are two strings x_n and x_m, where $|x_n| < |x_m|$. The module must return `False` if x_m is not a segment of x_m, and the position of the first symbol of x_n in x_m otherwise.

Exercise 12

Implement a Mathematica module whose input are a finite set of strings S and a string x_m, where $|x_i| < |x_m|$ for each $x_i \in S$. The module must return a list with the positions of x_m where a string in S appear.

i.e: Let `S={{a,b,b},{a,a}}`and $x_m = \{a, b, b, a, a, b, a, a, a, b, b\}$. The module should return $\{1, 4, 7, 8, 9\}$