Declarative Programming

Presentation

The material that we present here aims to guide the student in the study of Logic Programming and Functional Programming, both paradigms included within Declarative Programming.

This material has an eminently practical approach. We have reduced to a minimum the theoretical concepts, including only the elements that we consider essential to understand what a declarative program is. We have included the basic theoretical knowledge so that the student can start programming declaratively from the first session.

The material is divided into fourteen theoretical sessions and nine practical sessions with an approximate duration of one and a half hours per session. The theoretical sessions include exercises that serve to reinforce theoretical concepts. In the practical sessions, a series of exercises are proposed that the students must program using the computer and the Compiler or interpreter of Prolog or Haskell as the case may be.

How to use this material?

Understanding the philosophy of declarative programming is no easy task. It requires a change in the way a program thinks. We go from ordering a series of actions (imperative programming of C, C++, Java or Perl) to describing the problem using a series of rules (declarative programming Prolog and Haskell). To understand the paradigm of declarative programming we need to reinforce the idea that it is possible to program by changing the focus on program construction. This is achieved progressively, starting with simple exercises, increasing the difficulty to implement exercises with complex data structures such as trees or graphs.

At first, it can help the student to have the computer nearby and try the proposed exercises and see that they really work. It's not always obvious to see a declarative program work after looking at the code. More importantly, how the solution has been reached. This material uses an approach based on the principle of mathematical induction and relies on this in the process of building the programs. Not all the manuals found in the literature highlight this idea. Rather, they use a similar approach to describing other imperative languages, language syntax, and set of functions, methods, or predicates. We think that in declarative programming it is essential to understand that it is necessary to change the approach when programming and we place special emphasis on it. The idea that from the smallest case (n-1) we deduce the generic case (n) is repeated in most sessions.

After studying this material, if the student has understood how to build the programs, he will not need to mentally follow the execution of the program to know if it works. Based on the principle of induction, you should be able to tell whether the program can work or not after taking a look at the code. From this moment on, the exercises that at first seemed impossible to understand or program, are understandable from the declarative perspective.

We hope that this is the case and that after the study of this material the students understand that programs can be made in a different way.

Thanks

We want to thank the students who attended our classes with attention and interest by contributing many ideas that are now reflected in this teaching manual. To our colleague Manuel Maestre Hachero who kindly offered to review the material and correct it from the point of view of a mathematician. To Francisco Moreno Velo, Marcos del Toro Peral, Águeda López Moreno and Diego Muñoz Escalante who taught these subjects with us and contributed their ideas. To professors Jim Royer and Susan Older of Syracuse University in the United States who kindly allowed us to include the exercises from the second Haskell session of this manual.

Authors

Jose Carpio Cañada
Gonzalo Antonio Aranda Corral
Jose Marco de la Rosa

Contents

Theory

Introduction to declarative programming

Sección 0. Introduction

The objective of this subject is to introduce the student to a form of programming based on a paradigm very different from the more usual imperative paradigm.

This task will help the student to improve their capacity for abstraction and to practice with programming concepts such as recursion or lazy assessment, uncommon in other programming subjects.

The declarative paradigm also provides the student with a conceptual environment frequently used for the study of certain artificial intelligence problems.

Sección 1. Programming paradigms

Paradigm: (according to the Merrian-Webster dictionary)

"*A philosophical and theoretical framework of a scientific school or discipline within which theories, laws and generalizations, and the experiments performed in support of them are formulated; broadly : a philosophical or theoretical framework of any kind*."

"A philosophical and theoretical environment of a scientific school or discipline within which theories, laws, and generalizations of theses are formulated, as well as the experiments performed to justify them; More generally: a philosophical or theoretical environment of any kind."

Computer programming can be defined as the creation of coded descriptions that a computer can interpret to solve a certain problem. We will understand problem in its broadest form, including any interaction functionality we can imagine with the various components of the machine.

The programming task will depend entirely on the ability to interpret the code of the computer in question since we will have to adapt the descriptions generated to the language understood by the machine.

In the beginning, the 'programming language' of the machine was something as rudimentary as the electrical connection from one point to another within a circuit. Over time we have come to a very diverse range of languages that have been created incrementally on that primordial 'language'.

The various programming languages allow us to communicate with the computer with the intention of solving a problem. However, it is not trivial to establish the way in which it is most interesting to structure this communication and several different approaches have been developed from the beginnings of programming until today.

Each programming language approaches problem solving based on a set of concepts and a vision of how the computer should behave to solve those problems. Although there are many programming languages, the sets of concepts and visions of the problem domain are not as varied, so that each set is usually associated with several languages.

Each of these sets of vision and concepts condition the way in which problems and solutions are posed in order to express them so that the computer is able to solve them. Each of these sets is called **the programming paradigm**.

Examples of different programming paradigms are:

- Imperative programming
- Structured programming
- Declarative programming
- Logic programming
- Functional programming
- Event-driven programming
- Modular programming

- Side-oriented programming
- Object-oriented programming
- Restricted scheduling

Sección 2. Imperative programming

The first paradigm that is usually studied is the imperative paradigm. This paradigm understands that to solve a problem a series of steps must be carried out and the programmer is in charge of describing in an orderly and systematic way the steps that the computer must follow to obtain the solution.

Examples of imperative languages, although there are many more, are:

- BASIC
- C programming language
- Fortran
- Pascal
- Perl
- PHP

Program example: sorting with the bubble algorithm.

Opening statements:

```
void swap(int *x,int *y){
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
void bubble(int list[], int n){
   int i,j;
   for(i=0;i<(n-1);i++)
      for(j=0;j<(n-(i+1));j++)
             if(lista[j] > list[j+1])
                    exchange (&list[j],&list[j+1]);
}
```

Resolution of the problem:

```
void main(){
   // ...
   bubble(ready,ELEMENTOS_LISTA);
   // ...
}
```

Sección 3. Declarative programming

The declarative paradigm, on the other hand, proposes that the problems be described to the computer with a series of basic conceptual units that can be combined according to certain rules to generate new information. For the resolution of a problem, the descriptions that represent the domain in which the problem to be solved exists will be created and the problem will be posed as a question that must be answered either with one of the initial conceptual units or with a valid combination of them that is the computer itself who must search.

The conceptual units for the description of the problem will depend on the concrete programming subparadigm. In this subject we will study two subparadigms of the declarative programming paradigm: functional programming and logical programming.
In logic programming, the basic conceptual unit will be the logical predicate and in functional programming it will be the function.

In addition to the subparadigms mentioned, there are others such as algebraic languages (for example, SQL) or constraint-based programming.

Advantages:

- Compact and very expressive descriptions. It is possible to describe universes of problems with very few lines of language code that allow the solution of a large number of problems.
- Development of the program not so oriented to the solution of a single problem. With proper programming, it is enough to have described a problem domain correctly and know how to formulate our problem as a simple query in that domain. The variety of questions that can be answered with a single description of the specific problem domain is usually very high.
- There is no need to expend effort in designing an algorithm that solves the problem.

3.1 Functional programming

Functional programming is a subparadigm of declarative programming that uses function as a basic descriptive concept. This means that in our program we will describe functions and that these functions can be combined with each other to generate new functions. Among other actions that we can perform with functions, we can evaluate them. Our program will therefore consist of the definition of a series of functions that are, in turn, composition of basic functions and the problem we want to solve will normally be posed as the evaluation of a function based on those previously defined.

Among the different existing functional programming languages, in this subject we will study the **Haskell language**.

http://www.haskell.org/haskellwiki/Introduction

Program example: checking that a number is natural.

Opening statements:

```
natural::(Num a, Ord a)=> a -> Bool
natural 1 = True
natural n
| n > 1 = natural (n-1)
| otherwise = False
```

Resolution of the problem:

```
Main> natural (-1)
False :: Bool

Main> natural (10)
True :: Bool

Main> natural (12.5)
False :: Bool
```

3.2 Logic programming

Logic programming is another subparadigm of declarative programming that uses the logical predicate as a basic descriptive concept. Our program will consist of a series of predicates that will describe a world in which objects are related according to the rules of predicate logic. Our problems will raise statements for which the system will be able to obtain a logical explanation based on the programmed predicates if it existed.

Among the different existing functional programming languages, in this subject we will study the Prolog language specifically the SWI-Prolog free software version.

http://www.swi-prolog.org/

Example of program: descendant relations.

Opening statements:

```
% parent(Father,Son)
Father (John, Peter).
father (pedro, lluis).
father (Iker, Yeray).

% descendant(Descendant,Ascendant)
descendant(Descendant,Ascendant):-
       father (Ascendant, Descendant).
descendant(Descendant,Ascendant):-
       father(Ascendant,Intermediate),
       descendant (Descendant,Intermediate).
```

Troubleshooting:

```
?- father (john,X).
X = pedro.

?- descendant (Lluis, Juan).
true .

?- descendant (X, juan).
X = pedro ;
X = lluis ;
false.
```

Sección 4. Bibliography

General bibliography

Prolog Programming
Authors: W.F. Clocksin, C.S. Mellish
Editorial: Springer Verlag
Year: 1994

***Prolog* programming for artificial intelligence**
Authors: Ivan Bratko
Editorial: Addison Wesley
Year: 1990

Reasoning with *Haskell*
Authors: Blas C. Ruíz, F. Gutiérrez, y otros
Editorial: Thompson
Year: 2004

Specific bibliography

***Prolog*: the standard**
Authors: P. Deransart, A. EdDbali,
L. Cervoni
Editorial: Springer
Year: 1996

An introduction to computing in *Haskell*
Autores: Manuel M. T. Chakravarty, Gabriele C. Keller
Editorial: Pearson SprintPrint
Year: 2002

Programming languages. Principles and paradigms
Autores: A. Tucker, R. Noonan
Editorial: Mc GrawHill
Year: 2003

Functional programming with Haskell

Functional programming with Haskell

Sección 0. Introduction

(http://www.haskell.org/haskellwiki/Introduction)

What is functional programming?

Languages such as C, Pascal, Java are imperative languages. They are called imperatives because what we do when programming is indicate how a problem should be solved. We indicate the order in which the actions should be performed.

```
main() {
     primero_haz_esto();
     despues_esto_otro();
     por_ultimo_esto();
}
```

The goal in functional programming is to define WHAT to CALCULATE, but not how to calculate it.

An example of functional programming is the one we do when we program a spreadsheet:

- It does not specify in what order the cells should be calculated. We know that calculations will be performed in the proper order so that there are no conflicts with dependencies.

- We do not indicate how to organize memory. Apparently our spreadsheet is infinite, however, memory is only reserved for the cells we are using.

- We indicate the value of a cell using an expression, but we do not indicate the sequence of steps to perform to achieve this value.

In a spreadsheet we do not know when the assignments are made, therefore, we cannot make use of them. This is a fundamental difference from imperative languages such as C or Java, in which careful specification of assignments must be made and in which controlling the order of calls to functions or methods is crucial to making sense of the program.

Advantages of functional programming

- More concise programs
- Easier to understand
- Sin “core dumps”

Features of Haskell

Haskell is a pure functional programming language, with **static polymorphic types**, with lazy evaluation, very different from other programming languages. The name is taken from the mathematician Haskell Brooks Curry who specializes in mathematical logic. Haskell is based on the **lambda calculus**. The Greek letter lambda is the Haskell logo

- Type inference. Type declaration is optional
- Lazy evaluation: data is only calculated if required
- Compiled and interpreted versions
- Everything is an expression
- Functions can be defined anywhere, used as an argument, and returned as the result of an evaluation.

Haskell Deployments

There are different implementations of Haskell: GHC, Hugs, nhc98 and Yhc. For the realization of the practices we will use the implementation Hugs (http://haskell.org/hugs/).

Summary of existing Haskell implementations:

	Message s	Size	Tools	Notes
Hugs	+/-	++	-	Widely used to learn Haskell. Quick compilation. Rapid code development.
GHC	+	-	++	The generated code is very fast. Possibly the most used implementation.
NHC	?	+	++	Con posibilidad de profiling, debugging, tracing
Yhc	?	+	?	Still in development. .
Helium	++	++	-	Created for teaching.

For more information on the different implementations visit (http://www.haskell.org/haskellwiki/Implementations)

What will we learn from Haskell?

In this theoretical block, we will see a small introduction to Haskell, which will help us **to build small functions**, **have an idea of how to program in Haskell** and a **small vision about its possibilities**.

What will we have left to learn about Haskell?

- We will not see examples of Haskell connection with other languages.

When is C programming better?

C is generally faster and depending on the implementation, it may use much less memory. Haskell, needs to use much more memory and is slower than C.

In those applications where speed is important, C may be a better choice than Haskell, as C programming has much more control over the actual machine. The C language is closer to the machine than Haskell.

Is it possible to connect Haskell with other programming languages?

HaskellDirect is an IDL-based tool **(Interface Description Language)** that allows Haskell programs to work with software components. It is possible to use C/C++ functions from Haskell using **Green Card** or **C->Haskell**,

Sección 0. Defining Local Roles/Definitions

```
{- ------------------------------------------- -}
--DECLARATION
noNegativo::(Num a, Ord a)=>a->Bool
{- PURPOSE
   Returns True if x is >= 0, False otherwise
-}
--DEFINITION
noNegativo x = x >= 0
{-TESTS
pru1 = positive (-2.5) -- returns False
pru2 = positive 0-- returns True
pru3 = positive 5-- returns True
-}
{- ------------------------------------------- -}
```

Let's look at the elements needed to define a function.

The first thing we find is a **comment**.

-For **Comment on a block** `{-        -}`
- To **comment on a line** `--`

After the commented block, we find the header of the function.

```
<nombre_funcion>::<declaración_de_tipos>
```

The **function name** starts with a lowercase letter and can then continue with uppercase or lowercase.

To **define the types of the function** we can use variables of types belonging to some class (Eq, Ord, Enum, Num, Fractional, etc.) or with basic types (Int, Integer, Float, Double, Char, etc.).

Haskell can infer what types of data the function needs. The most generic header possible shall be inferred. To infer the type of data, Haskell uses the operators and functions used in the Definition of the function.

For the exercises proposed in this course, the definition of the headers of the functions will be mandatory.

Examples of data types used by operators:

a) If we use the operator "==" the type we use must be comparable (of class Eq).

```
Hugs> :info ==
infix 4 ==
(==) :: Eq a => a -> a -> Bool      -- class member
```

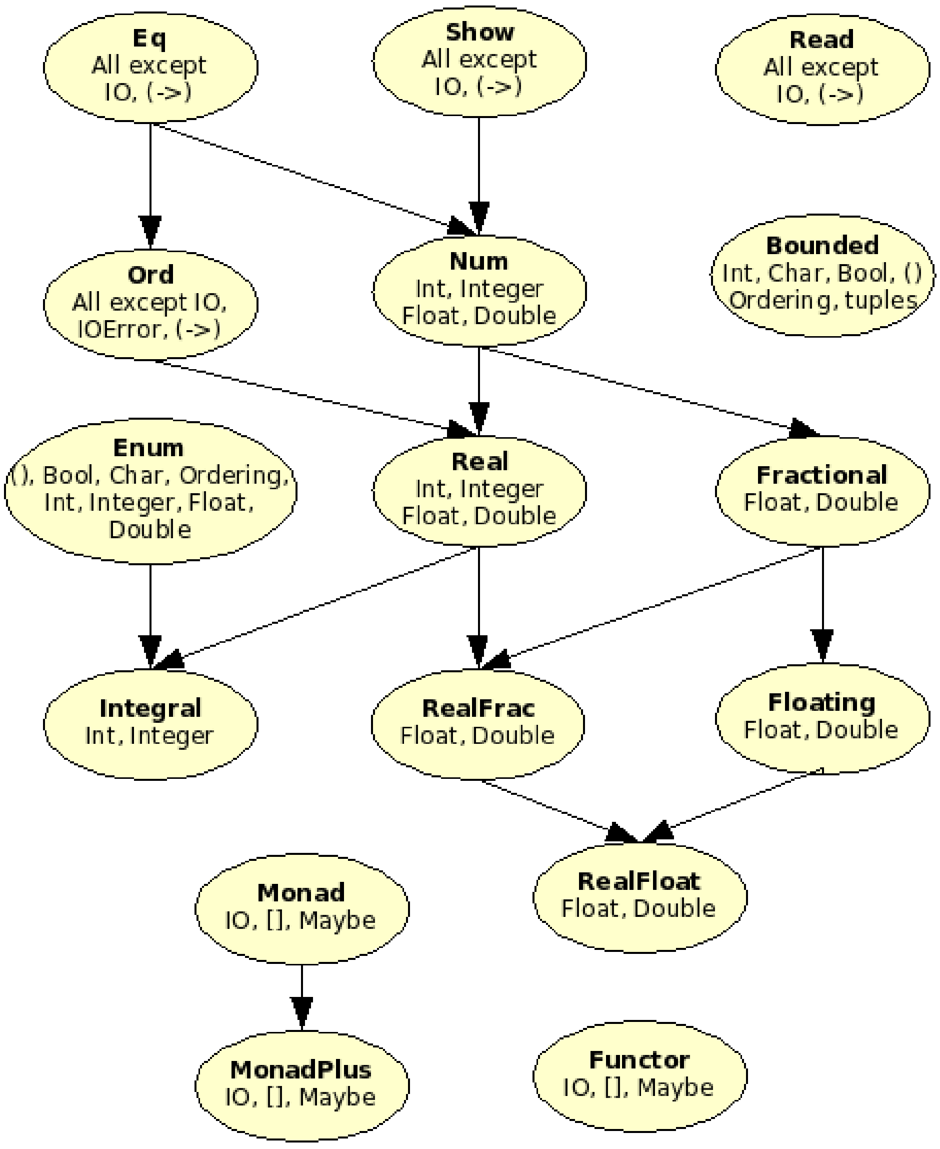
b) If our function contains the operator ">", the type must be orderable (of class Ord)

```
Hugs> :info <
infix 4 <
(<) :: Ord a => a -> a -> Bool      -- class member
```

c) If our function contains the "+" operator, the type must be numeric

```
Hugs> :info +
infixl 6 +
(+) :: Num a => a -> a -> a          -- class member
```

Type classification in Haskell:



Here are some **examples of defining function types**:

a) Identity function:

```
Identity::A->a
identity x = x
```

This definition indicates that the function receives a type a and returns a type a.

(b) Equal function:

```
iguales::a->a->a->Bool
equal x and z = x==y && y==z

ERROR file:.\iguales.hs:2 - Cannot justify constraints
in explicitly typed binding
Expression: Equal
*** Type: a -> a -> a -> Bool
*** Given context : ()
*** Constraints: Eq a
```

We need to add a constraint to type "a"

```
iguales::Eq a=>a->a->a->Bool
equal x and z = x==y && y==z

Main> equal 1 1 1
True :: Bool
```

(c) Split function

```
divide::Fractional a => a -> a -> a
divide x y = x / y
```

We have different possibilities when defining functions:

- Using multiple equations (writing each equation on one line)
- Guards, "|"
- If then else
- Case
- In local definitions

1.1) Using multiple equations

```
factorial::Int->Int
factorial 0 = 1
factorial n = n * factorial (n-1)
```

1.2) Guards

```
factorial n
  |n==0= 1
  |n > 0= n * factorial (n-1)
  |otherwise = "negative value" error
```

1.3) If then else

```
factorial n = if (n==0) then 1 else n*factorial (n-1)
```

1.4) Case

```
traduce x = case x of
  1 ->  "A"
  2 ->  "B"
  3 ->  "C"
```

1.5) Local definitions

You can define a function at any point in another function:

```
divisible::Int->Int->Bool
divisible x y = remainder == 0
   where rest = mod x y
```

It is very important that the definition is some spaces to the right of the position in which the function begins to be defined. Otherwise, Haskell will display an error.

Sección 1. Operator priority

The following table defines the priorities of the operators defined in the Prelude module:

Notation	Priority	Operator
infixr	9	`.`
infixl	9	`!!`
infixr	8	`^, ^^, **`
infixl	7	`*, /, 'quot', 'brake', 'div', 'mod'`
infixl	6	`+, -`
infixr	5	`:`
infixr	5	`++`
infix	4	``==, /=, <, <=, >=, >,`elem`,`notElem` ``
infixr	3	`&&`
infixr	2	`||`
infixl	1	`>>, >>=`
infixr	1	`=<<`
infixr	0	``$, $!, `seq` ``

notation **The infix** indicates that the operator is infixed, that is, it is written between the operands. In the case of chaining operators, no priority is defined.

```
Hugs> 1 == 1 == 1
ERROR - Ambiguous use of operator "(==)" with "(==)"
```

We need to make the priority explicit in parentheses:

```
Hugs> (1 == 1) == 1
ERROR - Cannot infer instance
*** Instance: Num Bool
*** Expression : (1 == 1) == 1
```

The result of evaluating the first expression is True. The error is given by the definition of the operator "==":

```
Hugs> :info ==
infix 4 ==
(==) :: Eq a => a -> a -> Bool -- class member
```

The operator is defined for two elements of the same type, which belong
to class **Eq** (comparable).

```
Hugs> (1 == 1) == True
True :: Bool
```

Infixl indicates that, in case of equality of precedence, the left will be evaluated first:

```
Hugs> 1 - 2 - 1
-2 :: Integer
```

Infixr indicates that, in case of equality of precedence, the rightmost operator will be evaluated first:

```
Hugs> 2 ^ 1 ^ 2
2 :: Integer
```

When we use functions and operators in the same expression, the function will have higher priority:

```
Hugs> succ 5 + 1
7 :: Integer
```

The operator that has the highest precedence is the composition of functions ".".

```
Hugs> succ . pred 4
ERROR - Cannot infer instance
*** Instance: Enum (b -> a)
*** Expression : succ . pred 4
```

An error occurs because the `pred 4` operation is performed first. The result is number 3. Then you try to make the composition of the number 3 with the succ function. In order to perform a composition of functions, two functions are necessary. You cannot compose a function with a number. In this way, we show that between an operator (whatever it is) and a function, the function will first be evaluated:

```
Hugs> (succ . pred) 4
4 :: Integer
```

As an example, we will implement the following function, using the `fromIntegral` function, which converts integer values to a general numeric type, and avoiding unnecessary parentheses

$$\max(x, y) = \frac{(x+y)+|x-y|}{2}$$

```
max x y = fromIntegral (x+y + abs (x+y)) / 2.0
```

It would be left to define what is the order of precedence between two functions.

```
Hugs> succ pred 4
ERROR - Cannot infer instance
*** Instance: Enum (a -> a)
*** Expression : succ pred 4
```

In case of chaining of functions with the same priority, the evaluation is carried out from left to right. First you try to evaluate succ pred, and because `the succ function` is defined only for enumerable types, trying to evaluate the successor of `pred` fails to fail.

```
Hugs> succ (pred 4)
4 :: Integer
```

Sección 2. Lazy evaluation

Languages that use this technique only evaluate an expression when needed:

```
soloPrimero::a->b->a
soloPrimero x _ = x

Main> soloPrimero 4 (7/0)
4 :: Integer
```

The underline "_" denotes that a parameter is expected but does not need to be named because it is not to be used in the body of the function.

The `expression 7/0` in Haskell has the value `Infinity`. The evaluation of the previous expression would cause an error in most imperative languages, however in Haskell it can be evaluated because the second argument does not evaluate because it is not necessary.

Sección 3. Higher-order functions

Higher-order functions are those that receive one or more functions as input arguments and/or return a function as output.

An example of a higher-order function is the `map` function implemented in the `Prelude module`. This function receives a function and a list and applies the function to each item in the list:

```
map:: (a->b)->[a]->[b]
map _ [] = []
map f (cab:resto) = f cab : map f resto

Main> map succ [1,2,3]
[2,3,4] :: [Integer]
```

The `function f` that we pass as an argument to `map` must meet a constraint: the type of data it receives must be the same as the items in the list.

Sección 4. Currificación

The currification process takes its name from Haskell Brooks Curry whose work in mathematical logic served as the basis for functional languages.

It consists of making the call to a function using only some of the parameters that are further to the left.

```
suma::Int->Int->Int->Int
sum x and z = x + y + z
```

We can make the following calls:

a) `sum 1` -> returns a function that receives two integers and returns another.

b) `sum 1 2` -> returns a function that receives one integer and returns another.

c) `sum 1 2 3` -> returns integer 6

We can use the currified function as an argument to another function of
Higher Order:

```
Main> map (suma 1 1) [1,2,3]
[3,4,5] :: [Int]
```

Sección 5. Composition of functions

f.g (x) = f(g(x))

Haskell has an operator to compose functions. It's about
of operator ".":

```
Main> :info .
infixr 9 .
(.) :: (a -> b) -> (c -> a) -> c -> b
```

In order to compose two functions, they must meet a series of restrictions. If we look at the header of the operator ".", we can see that the second argument is a function (c->a), that the first argument `is a function (a->b), that the value to` which you want to apply the composition is of type c and that the output value is of type `c`.

The most important constraint is that if we want to make the composition `f.g, the output type of the function g` must be the input of the function `f`.

Let's look at an example:

```
Main> ((==True).( <0)) 5
False :: Bool
```

The function `(<0)` returns a `Bool type`, which is the one that receives the function `(==True)`. The result of the composition of the two functions is of type `Bool`.

Functional programming with Haskell

Sección 6. Lists

Haskell provides a mechanism to easily define lists:

```
a) [1..10]              d) ['q'..'with']b) [15..20]              (e)
[14..2]
c) [15..( 20-5)]
```

Extended list notation

The definition of a list with this definition consists of three parts: 1) generator, 2) constraints (there may be none), and 3) transformation. The generator produces items from a list, constraints filter some items from those generated, and the transformation uses the selected items to generate a result list.

```
[ (x,True) | x <- [1 .. 20], even x, x < 15]
 |__  |___   |______________|       Transformation Generator Constraints
```

In this example, `x <- [1..20]` is the generator, the constraints are `even x` and `x < 15`, y `(x, True)` is the transformation. This expression generates a list of pairs `(x,True)` where `x is` between 1 and 20, is even and less than 15. (`even` is a function defined in Haskell's standard Prelude module).

Most common functions on lists

Function header	Explanation
`(:) :: a -> [a] -> [a]`	Add an item to the top of the list
`(++) :: [a] -> [a] -> [a]`	Concatenar of lists
`(!!) :: [a] -> Int -> a`	Returns the nth element
`null :: [a] -> Bool`	Returns True if list == []
`length :: [a] -> Int`	Returns the length of a list
`head :: [a] -> a`	Returns the first item
`tail :: [a] -> [a]`	All but the first
`take :: Int -> [a] -> [a]`	Take the first n elements
`drop :: Int -> [a] -> [a]`	Delete the first n items
`reverse :: [a] -> [a]`	Reverses the order of elements
`zip :: [a] -> [b] -> [(a,b)]`	Create a list of zip pairs `"abc" "123" >> [('a','1'),('b','2'),('c','3')]`
`unzip :: [(a,b)] -> ([a],[b])`	`unzip [('a','1'),('b','2'),('c','3')] >> ("abc","123")`
`sum :: Num a => [a] -> a`	Add the items in a list
`product :: Num a => [a] -> a`	Multiply the items in a list

See information about functions defined for lists in the Prelude module.
http://www.cs.ut.ee/~varmo/MFP2004/PreludeTour.pdf

Sección 7. Patterns

Let's look at a possible implementation of the map function:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (cab:rest) = f cab : map f rest
```

In the example above:

`[]` is the pattern that matches an empty list.
`(cab:remainder)` pairs with a list (the ":" operator returns a list) whose first element is "cab" and the rest is "remainder".

```
Hugs> :info :
infixr 5 :
(:) :: a -> [a] -> [a]  -- data constructor
```

f pairs with any value of the expected type (by the header of the `map function`, f must be a function) and `f` is instantiated to that value (that function).

_ pairs with any value of the expected type, however, no assignment is made to the variable "_": the value is simply ignored.

Then pattern matching is a way to assign values to variables and, at the same time, a way to divide expressions into sub-expressions.

What can we use as a pattern?

There is a given set of pairing patterns, so it is not possible to make pairings with all the language constructs that seem possible to us but only with those that Haskell allows us. For example:

```
deleteThree ([x,y,z] ++ remainder) = rest
```

This definition of the deleteTres function causes an error:

```
ERROR file:.\deleteTres.hs:2 - Syntax error in input (unexpected symbol
"++")
```

The problem arises because the "`++`" operator is not allowed in the patterns. Only constructors and constants (`1`,`2`, , `True`, `False`etc.) are allowed in the patterns. A constructor has the form:

```
data Bool = True |  Falsedata [a] = [] | a:[a]
```

```
Main>  :info :
infixr 5 :
(:) :: a -> [a] -> [a] -- data constructor
```

When we request information about a Haskell operator, it indicates whether it is a constructor.

[] and "`:`" are constructors for lists. The `deleteThree function` can be implemented like this:

```
Emliminar::[a]->[a]
DeleteThree (_:_:_:remainder) = remainder
```

The type of the constructor is important, but not the number. The example above uses the ":" operator as the pattern three times.

Exception

There is an exception to the rule of using constructors in patterns. This is the pattern `(n+k)`.

```
predecessor::Int->Intpredecessor (n+1) = n
```

We cannot use any value to match it with `(n+k)`. We can only use integers (`Int` or `Integer`). Also, the value has to be greater than or equal to `k`.

```
Main> predecessor 0
{throw (PatternMatchFail (_nprint 0 (predecesor 0) []))} :: Integer
```

Pattern aliases

Patterns are sometimes used to split expressions. In these cases, it may be interesting to have access to the full expression. To do this, we will use pattern aliases. We will write `nombre_variable@<pattern>`. For example, a function that removes blank characters at the beginning of a string:

```
quitaBlancosPrinc::String->String
-- equivalent to quitaBlancosPrinc::[Char]->[Char]
removeWhitesPrinc cadena@(cab:rest)
  | cab == ' ' = removersBlancosPrinc rest
  | otherwise = cadena
```

or this one, a version of the factorial using a pattern alias:

```
factorial::Integral a => a->afactorial  0 = 1
factorial m@(n+1) = m * factorial n
```

We will use the patterns in equations, let and where clauses, case expressions and lists.

Sección 8. Tuples

We can group expressions of different types in a tuple. For example:

a) (1,2)
b) `('a',1,"Hello")`
c) `((1,2),[3,4],(0,'a'))`
d

) ((+), 7, (*))Let's see what kind the tuples are above

```
:Main> :t (1,2)
(1,2) :: (Num a, Num b) => (b,a)
Main> :t ('a',1,"Hola")
('a',1,"Hola") :: Num a => (Char,a,[Char])
Main> :t ((+),7,(*))
((+),7,(*)) :: (Num a, Num b, Num c) => (c -> c -> c,b,a -> a -> a)
Main> :t ((1,2), [3,4], (0,'a'))
((1,2),[3,4],(0,'a')) :: (Num a, Num b, Num c,  Num d) =>
((c,d),[b],(a,Char
```

)))Some examples of functions with tuples:

```
a) first (x,y) = xb) first2 (x,y,z) = x

Main> primero (1,2,3)
ERROR - Type error in application
*** Expression     : Main.primero (1,2,3)
*** Term           : (1,2,3)
Type: (c,d,e)
*** Does not match : (a,b)

Main> primero2 (1,2,3)
1 :: Integer
```

Defining types with tuples

Let's look at the following example:

```
type Input = (Person, Age, Phone)

type Persona = String
type Age = Int
type Telefono = String
type Listin = [Input]

find::Listin -> Person -> [Phone]

Find Person List = [Phone| (per, age, phone) <- list, person == per]

Main>  find [("Peter", 20, "636000000"), ("John", 21,"607222222"),
("Alberto", 24, "635111111")] "Peter"

["636000000"] :: [Main.Telefono]

Main> find [("Pedro", 20, "636000000"), ("Juan", 21,"607222222"),
("Alberto", 24, "635111111"), ("Pedro", 20, "635444444")] "Pedro"

["636000000","635444444"] :: [Main.Telefono]
```

Functional programming with Haskell

Sección 9. Recursion

In Haskell we have no possibility to define loops. The way to "iterate" is using recursion. A recursive function is one that in its definition contains a call to itself.

Recursion uses the same idea as the principle of induction. This principle is widely used in mathematics to show that a property holds for any value of the scope being treated.

Induction principle:

(1) The statement *P* is true for an initial value *n0* ("base case")

(2) P will be *true for a value n > n0*is true for , if it is true for the value before n, that is, if *P n-1.*

We can use the principle of induction to define the natural numbers:

The number 1 is natural.x is natural if n-1 is natural.

In Haskel:

```
natural 1 = Truenatural n = natural (n-1)
```

```
Main> natural 5
True :: BoolThe
```

principle of induction works!! - > Let's use it.

but...

```
Main> natural (-3)
```

```
It doesn't end!!!
```

The same happens with the number "`3.5`". We have not taken into account that, the value has to be greater than the first. A new version:

```
natural::(Num a, Ord a)=> a -> Bool
natural 1 = True
natural n
  | n > 1      = natural (n-1)
  | otherwise = False
```

```
Main> natural (-1)
False :: Bool
Main> natural 3.5
False :: Bool
```

Important recommendation: We will not mentally follow the sequence of recursive calls to solve the problems. **We will focus on defining well the base case (the first element that satisfies the property) and the relationship of a value `n` with the previous one**.

Example of recursion with lists

We will transfer the same idea to the resolution of a recursive problem with lists. We will recursively define a function that returns the number of items in a list.

For this problem, the first case is not 1. **What is the first known true case for lists?** In this case (and in most list problems) the first known case corresponds to the empty list. It is the smallest list, just as 0 is the smallest natural.

```
numElements [] = 0
```

This equation satisfies the property: **it is true that an empty list has zero elements**.

Let's see what happens with `n`. Now we do not have an `n number` and a previous number `n-1` as was the case with the natural ones. The example of the natural numbers says that, if the property for n-1 is satisfied, it will be fulfilled for n.

How do we construct the value n and `n-1` with lists?

We will use the pattern "`cab:rest`" that separates the first element from the rest.

Our n element `will` be the complete list, and the `n-1 element` will be the list with one less element (the content of the rest variable).

If numResto elements then numElements (cab:rest)

If `numItems (remainder)` is met, it returns the number of items contained in the rest of the list (all but the first).

By the principle of induction we know that:

If `) numElements` numElements rest is true (returns the correct value `(cab:rest)` will also be true. But...

How much is `numElementos (cab:rest) worth?`

To know how much it is worth, we will use the value returned by `numResto elements`

```
 numElements (cab:rest) = numRest elements
  |_____|        |___________________|
We want to know  this We will assume that we know how much value is worth
```

What change do I have to make in the partial exit, to obtain the total output?

This will always be the question we will always ask to solve a recursive problem.

In this case the transformation is simple. If we know how many items a list has that is missing an item, adding one will suffice it to know how many in the complete list.

```
 numElements (cab:rest) = 1+ numRest elements
  |_____|        |___________________|
We  want to know this We will assume that we know how much value is worth
```

Let's see another somewhat more complicated example:

We want to implement the foldl function defined in Haskell Prelude. This function is defined for non-empty lists. An example can be found in the document "A tour of the Haskell Prelude" http://www.cs.ut.ee/~varmo/MFP2004/PreludeTour.pdf

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Description: Joins the elements of a list using a binary operator and an initial element, using association on the left. When the input list is empty, returns the initial item.

Let's spend some time understanding what the function does. For this we can make some more examples:

```
Hugs> foldl (^) 1 []
1 :: Integer

Hugs> foldl (^) 1 [2]
1 :: Integer

(1^2)

Hugs> foldl (^) 1 [2,3]
1 :: Integer

(1^2)^3
```

Now that we have something clearer that makes the function. Let's start with the implementation:

1) **We will start with the base case**. It is usually the simplest case. It must be true on its own, regardless of what we write next. It is usually described in the definition of the function or is very obvious, as in the case of the `numElem function []` whose result is `0`.

   In this case:

   ```
   foldl f the [] = the
   ```

2) **Recursive case.** We will start by posing the recursive case as follows

   ```
   :foldl f el (cab:rest) = foldl f
   ```

   the rest For list problems, we will use in most cases the same technique, separating `the head` from `the rest` of the list using the operator "`:`". Now, we'll focus on figuring out what the function called with a list of one less item as a parameter returns. To do this it is best to write an example:foldl (^) 1 [2,3,4] -> (((1^2)^3)^4)

   with one element less:

   ```
   foldl (^) 1 [3,4] -> ((1^3)^4)Now
   ```

   we will ask:

   What transformation do I have to make to the result of the function that has one less item to become the result of the complete list?

   We look for the differences between partial and total output.

   ```
    (((1^2)^3)^4) -> Total output(
   ( 1 ^3)^4) -> Partial output
   ```

   Looking at the two outputs, we see that the difference is that the partial output contains a "1" and the total contains "1^2", with 2 being the first item in the list. The rest is the same. It seems that the difference is only in the second element of the function. The change that the partial function call needs is to change the element "`the`" to "`el^cab`". In a more generic way: "`f el cab`".

We will complete the recursive call with the change. Where we wrote "`the`", we will write "`f the cab`".

```
 foldl f el (cab:rest) = foldl f

 the remainder  |__
_|
       we will change the by f the cab foldl f the (cab:rest) =



 foldl  f (f the cab) rest
```

Finally the function will be as follows

```
:foldl :: (a -> b -> a) -> a -> [b] -> afoldl f el [] = el foldl f el
(cab:rest) = foldl f (f el cab) resto

>  Main> foldl2 (^) 1 [2,3,4]
1 ::  Integer

Mainfoldl (++) "com
" ["po", "ner"]"compose" :: [Char]
```

Logic programming with Prolog

Logic programming with Prolog

Sección 0. Introduction

Propositional Logic:

Use simple statements (propositions). We will write **rules** to indicate that if an event occurs (**antecedent**) then a consequence (**consequential**) will occur.

If it rains -> the plants will be watered

|____________| |_________________| ***Consequential Background***

|__

RULE

With the previous rule, it is not indicated that it *is raining*. To say that it rains we need a **fact**.

Rain

|__

FACT

A **clause** is a rule or a fact.

With clauses we will create **a knowledge base** (a representation of what we know). Once the *knowledge base is defined*, we will carry out **consultations**.

A *"logic program"* consists of two elements:

"Logical program" = Knowledge Base + Queries

Prolog syntax: **We will write the rules starting with the consequent**. All **clauses** end with a period "." We write the rule *If it rains -> the plants will be watered* in this way: **`se_riegan_las_plantas :- it rains.`**

First Prolog program using only propositions:

```
% My first program Prologse_riegan_las_plantas :- rain.rain.
```

To affirm that it rains, we will write the following fact: `it rains`.

We must make consultations, to obtain some result:

```
1 ?- consult('d:/lluvia.pl').
% d:/compiled 0.00 sec, 672 bytes

Yes2 ?- llueve.

Yes3 ?- se_riegan_las_plantas.

Yes
```

In line 1 we load the program "`lluvia.pl`" using the consult procedure. Prolog indicates compile time and compiled bytes. In case of any query, except for error, Prolog responds "`Yes`" or "`No`".

On line 3 we ask if it rains. The answer is "`Yes`." The next question is se_riegan_las_plantas? The answer is "`Yes`."

Let us imagine that we wish to express the following: *If it rains then, the tomatoes and lettuce and* *strawberries are watered.*

Using propositional logic we would write the following:

If it rains -> se_riegan_los_tomates ^ se_riegan_las_lechugas ^ se_riegan las_fresas.

To facilitate the representation of knowledge and the association of concepts, **predicates are used**.

First-Order Logic or First-Order Predicate Calculus:

It extends propositional logic using variables, predicates, and quantifiers of variables.

We will rewrite the previous rule using the predicate `se_riegan(Plants)`**.** This predicate represents that a certain plant is watered.

```
If it rains -> se_riegan(tomatoes) ^ se_riegan(lettuce) ^
se_riegan(strawberries)
```

This new definition using predicates would ask what is watered? and Prolog would answer, tomatoes, lettuce and strawberries.

However, this rule written in this way cannot be implemented directly in Prolog since it is not a *Horn Clause.*

Horn clauses. A *Horn clause* is one that is formed by a conjunction of zero or more **terms in** the antecedent and a single term in the consequent one.

```
      If it rains -> se_riegan(tomatoes)
 |_____|   |________________|          Term Term
```

To resolve whether a certain query (**formula**) is true or false, a resolution algorithm is used. This resolution algorithm is ***decidable*** (always ends with a valid solution) if the clauses are all *Horn Clauses*. Otherwise, the resolution algorithm could not be guaranteed to finish.

In some cases we can find an equivalence for a clause to be written as Horn's clause.

1. $a \wedge b$ □ c **C. de Horn**

2 a □ $b \wedge c$ **. Not Horn C. => Equivalence** a □ b a □ c ¿ {¿¿¿ ¿

3. $a \vee b$ □ c **It is not Horn's C. => Equivalence** a □ c b □ c ¿ {¿¿¿ ¿

4. a □ $b \vee c$ **It is not C. De Horn => There is no equivalence**

Our program on rain and irrigation using Horn clauses would look like this:

```
% Rain, irrigation and vegetables using predicates

se_riegan(tomatoes):- It rains.
se_riegan(lettuce):- It rains.
se_riegan(strawberries):- It rains.
Rain.
```

We can ask then: What is watered?

Prolog syntax: We will write the variables starting with a capital letter.

```
2 ?- se_riegan(Vegetables).

Vegetables = tomatoes ;
Vegetables = lettuce ;
Vegetables = strawberries
3 ?
```

To get all the answers, we type ";" after each answer.

Closed world hypothesis:

The undefined is false. In order to simplify the definition of the knowledge base, an agreement is reached to respond as false to everything not defined. If we ask any questions about a value that does not exist, Prolog will answer "No".

```
3 ?- se_riegan(apples).

No
```

Logic programming with Prolog

Sección 0. Introduction (continued)

Variables

We will write the variables **starting with a capital letter**. You do not need to declare a variable before you can use it. A **variable in Prolog can be instantiated with any type of data** (tree, list, graph, etc.). **A variable, once instantiated, does not change its value.**

Examples of variables:

1) Person
2) List
3) X
4) Tree
5) _

Examples of use:

```
?- X = 1, X = 2.No

?- Person = manuel. Person = manuelYes

?- List = [1,2,3]List = [1,2,3]

? List = manuelList = manuelYes
```

In the last of the previous examples, we note that Prolog does not do any kind of check when assigning values to variables. We can assign to a variable called "`List`" (which should contain a list) any value. If the List variable `is` free (it was not previously assigned a value), it will be possible to perform the following unification `List=manuel` and Prolog will answer that it is true if `List` is instantiated to the value "`manuel`".

Free and Instantiated Variables

We will say that **a variable is free if a value has not yet been assigned** through a unification.

We will say that a **variable is instantiated if it has already been assigned a value** through a unification. That variable cannot change its value.

```
?- X=1, X=2.No
```

For readability, we **must assign to the variables names representative of what they will contain:** if the variables contain numbers we can call them `X, Y, ,Z`etc.; if the variables contain lists, we will call them `List`, `ListR`, and so on. This will help us to understand the programs more easily.

Scope of a variable

It is common in other programming languages that if a variable is defined within a function, the variable can be referenced from any point in the function. We say that the scope of the variable is the function in which it is defined.

In Prolog, the scope of a variable is restricted to one clause.

Beyond that clause, the variable cannot be referenced. This is one of Prolog's most important differences from most programming languages.

See the following example:

```
sum(X,Y,Z):- X \= Y, Z is X + Y.sum(X,X,Z):- Z is X +
X.
```
Code 1: First version of the sum predicate

In the previous example the variable X and the variable Z appear in the two clauses, however there is no connection between them. We can change the name of the variables and the program will work in the same way.

This fact allows that observing only one clause we can know if it is correct or not, regardless of what is written next.

```
sum(X,Y,Z):- X \= Y, Z is Y + Z.suma(N,N,R):- R is N +
N.
```
Code 2: Second version of the sum predicate

This second version of the addition predicate implementation works exactly the same as the previous one.

Anonymous variable ("_")

When we want to indicate that an argument has a value, but we do not care what that value is, we will use the *anonymous variable*, which is represented by the character "_".

The anonymous variable has a sense of "*all*" or "*any value*".

For example: If I want to indicate that when it rains *all* the plants are watered I can write:

```
se_riegan(Plant):- Rain.
Rain.
```

When you run the above program, Prolog displays the following message:

```
Warning (d:/rain4.pl:1):
Singleton variables: [Plant]
```

This message indicates that a variable appears only once in a clause, which means that there is no restriction on the value that the variable can take and, therefore, it is not necessary to give it a name. It is only a warning with which the program can be executed, however, we must correct them. If we get used to seeing them, when we really make a mistake when writing the name of a variable, it is very possible that we will not pay any attention to the warning.

This can happen for two reasons:

1) Let it be correct. I really want to indicate that there is a variable but I will not use its value, in which case we will write `se_riegan(_):-rains.`
2) That I misspelled the name of a variable, in which case I must correct the error:

```
grow(Plamta):- se_riega(Plant).

se_riega(_):- Rain.
Rain.
```

It will cause the following warning.

```
 Warning (d:/rain4.pl:1):
        Singleton variables: [Plamta, Plant]
```

Sección 0. Unification

We will say that two terms unify:

1) If they have no variables, they unify if they are identical.

2) If they have variables, they unify if it is possible to find a substitution of the variables so that they become identical.

Examples:

```
?- X=1.X=1Yes

?- X = woman (maria). X = woman(maria)Yes

?- X = 2 + 3.X=2+3Yes

?- X=1, X=2, X=Y.No

?- X=2+X.X=+**Yes

?- (1+2) = 1+2.Yes

? 1+1 = +(1,1). Yes

? (1+1)+1 = 1+(1+1). No
```

Operators

The most commonly used operators in Prolog are=", `"="`, `"=="`, `"is"`, `"==="`, "<", "=< ">".

"=" unifying:

The unification operator does not evaluate arithmetic operations. Before unifying a variable to a value we will say that the variable is "free" or "uninstantiated". After unifying a value to a variable, the variable is no longer free and we will say that the variable is "instantiated".

```
?- X = 1+1.

X = 1+1.
Yes
```

"==" Identity comparator:

True ee if the values compared are exactly the *same value*.

```
?- 2 == X.
No.

?- 2 == 2.
Yes.

?- 2 == 1+1.
No.
```

"`is`" evaluator of arithmetic expressions:

Evaluate on the right and unify with the left. In order to perform an arithmetic operation, all variables to the right of the operator must be instantiated.

```
?- X is 1+1.

X = 2
```

"`=:=`" evaluator and comparator of arithmetic expressions:
It evaluates right and left and it is true if the result of the evaluations is identical.

```
?- 2 =:= 1+1. Yes
```

Examples:

```
?-  X is 2 + 3, X = 5.
X=5Yes

?- X is 2 + 3, X = 2 + 3.No

?- 2 + 3 is X.ERROR: is/2: Arguments are not sufficiently instantiate

?- X is 5, 6 is X + 1.X=5Yes

?- X = 5, 6 = X + 1.No

?- 6 is X + 1, X = 5.
ERROR: is/2: Arguments are not sufficiently instantiate
```

Table of operators

Precedence	Notation	Operator
1200	xfx	`:-`
1000	xfx	`,`
900	fy	`\+`
700	xfx	`<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @<, @>=, \=, \==, is`
500	yfx	`+, -, xor`
500	fx	`+, -, ?, \`
400	yfx	`*, /, mod, rem`
200	xfx	`*`
200	xfy	`^`

- Operations whose operator has a lower precedence number will be held earlier.
- YFX: associative on the left. (Example: X is 2/2/2, X=0.5).
- xfy: associative on the right. (Example: X is 2^1^2, X=2).

For a complete list of operators run
`apropos(operators).`

Logic programming with Prolog

Sección 1. Data types

Although Prolog is not a typed language, there are certain values with peculiarities of notation and associated predicates that differentiate them from other values by turning them into a kind of data type. In any case, we will need to delimit groups of values that act as differentiated data types in our programs. In this section, we'll look at how to represent and use some important data types with Prolog.

2.1 Lists

We will notice the lists in square brackets, separating the elements by
Eat. Here are some features of lists in Prolog:

- Lists **contain terms (variables, constants, or structures) in general,** so they are not restricted to containing values of one of those types in particular or with uniform notation (numbers only, only variables, etc.).
- **It is possible to nest lists.**
- We will notice **the empty list ("[]")** with an opening bracket and a closing bracket.

Examples of lists:

```
 2)    [1,2,3,4]
 3)    []
 4)    [a,b,c,d]
 5)    [1,’a’,b,X,[1,2,3]]
 6)    [[1,2],[3,4],[5,6]]
 7)    [se_riegan(Plants),[3,4],peter]
```

Note: If we want a variable to be installed to a list, we will do it as follows:

```
?- List = [1,2,3,4]
List=[1,2,3,4]
```

Remember that a free variable can be instantiated with any type of data. A common error is as follows:

```
?- [List] = [1,2,3,4]
No
```

By `[List]` we mean a list that contains a single term.

Operator "|"

We will use the operator "|" (vertical bar), to separate the first item in a list from the rest. This operator is very useful for constructing recursive predicates and we will use it frequently.

Examples:

```
1 ?- [Head| Remainder] = [1,2,3]
Head=1
Rest=[2,3]

2 ?- [Head| Remainder] = [1]
```

```
Head=1
Rest=[]

3 ?- [C1,C2| Remainder] = [1,2,3]
C1=1
C2=2
Rest=[3]

4 ?- [_|_]= []
No

5 ? [Head| Remainder] = [[1,2,3]]
Head=[1,2,3]
Remainder = []
```

Note: We must bear in mind that **the *rest* is always a list**. In the second of the previous examples [C|R]=[1], unification is possible, the rest being an empty list "[]", R=[].

Let's now look at some examples of predicates that work with lists:

We will implement the num_elem`/2 predicate` that counts the number of items in a list.

We will implement the predicate in several steps:

Step 1) Base case.

It is usually the easiest. When dealing with lists, the base case usually refers to the empty list. **The base case should not depend on other rules that refer to the defined predicate**.

```
% num_elem(+List, -Int)
num_elem([],0).
```

Step 2) We raise the recursive case.

We construct the recursive case **with a somewhat smaller problem size**. Usually with the rest. Instead of a `List variable`, we type [Head| Rest], which will separate the first from the list from the rest.

```
num_elem( [Head| Rest],       ):-
      num_elem(Rest, Num_resto) ,
```

If the recursive call works, it will return the number of items of the rest.

Remember the principle of induction:
`If is true for n0 and is true for n-1` with `n>n0` n>n0, then it is true for any

Example in Prolog, `true(N) :- Nmenos1 is N-1, true(Nmenos1).`

Step 3) From the recursive case (partial result), we get the total result. In this case, simply add 1 to `NumResto`, to get the total.

```
num_elem([Head| Rest], NumLista ):-
      num_elem(Rest, NumResto),
      numLista is NumResto + 1.
```

Step 4) Finally, we check that all cases are covered only once. In this example, we have a case for empty lists and a generic case for a list of 1 or more items. We have considered all possible list sizes.

We will take into account that when writing `[Head| Rest]` we force the list to have at least one element since the rest may be empty but the head must necessarily unify with a term.

The program would look like this:

```
% num_elem(+List, -Int)
num_elem([],0).
num_elem([Head| Rest], NumLista ):-

      num_elem(Rest, NumResto),
      numLista is NumResto + 1.
```

Exercises:

Implement the following predicates using recursion:

`reverse(+List, -ListR)` which is true when `ListR` unifies with a list containing the same items as `List` but in reverse order.

`aniadir_final(+Elem, +List, -ListR)` which is true when `ListaR` unifies with a list that contains the same items as the `List` list plus the `Elem element` added at the end.

`elemento_enesimo(+Pos, +List, -Elem)` which is true when `Elem` unifies with the item occupying the `Pos` position within `List`.

Predefined predicates

There are predefined predicates in Prolog for lists that we can use in our programs. If we make the following help`(length) query`. We will obtain the following result:

```
length(? List, ? Int)

      True if Int represents the number of elements of
      list List. Can be used to create a list holding
      only variables.
```

The behavior is similar to our previously `implemented num_elem/2` predicate .

Note: In section 4.1 of the SWI-Prolog manual (?- help(4-1).
We find information about the notation used in the description of predicates:
? Indicates that the argument can be input or output.
+ indicates that the argument is usually input.
- indicates that the argument is usually output.

Other predefined predicates for lists are:

```
append/3
member/2
reverse/2
nth0/3
nth1/3
```

Reversible predicates

One of the features that differentiates Prolog from other programming languages is the ability of predicates (unlike a function defined in C, for example), to be used reversibly.

We can use them in a similar way to a function, that is, by entering values in the inputs, waiting for the output variables to unify with the result. However, we can also use them the other way around, that is, **by entering the output waiting for Prolog to find the input(s)** needed to make the predicate true.

Let's look at an example:

```
6 ?- length([1,2,3], Length).
Length=3

7 ? length(Lista, 3).
List = [_G331, _G334, _G337]
```

In example 7 we have used the `length/2` predicate in a contrary way to the usual one. We indicate what the output is, in this case the length of the list, and we expect Prolog to find a possible list of length 3. Prolog's response in `List=[_G331,_G334,_G337]`. It is Prolog's way of referring to a list of any three elements (`_G331._G334._G337` identify variables).

This way of using predicates is not found in imperative languages. **We must make use of the reversibility of predicates**.

Exercises:

1) Check the operation of predefined predicates in Prolog for lists.

2) Check if the implementation of the num_elem`/2 predicate` seen in this topic is reversible and reason why.

3) Review the exams of previous years and perform the problems of lists that are found.

Bibliography

SWI-Prolog Help

Logic programming with Prolog

Sección 2. Data types (continued)

2.1 Lists (continued)

Lists are the most commonly used data type in Prolog because they are used by other types of data, such as generic trees and graphs.

It is very important that we have a good understanding **of how lists work** and, above all, **how recursive predicates should be constructed**. If we understand these concepts well, we may become good declarative programmers. Otherwise, it is very possible that we need too much time to solve the problems and end up thinking that Declarative Programming is too complex and not worth using.

Professor David Enrique Losada Carril of the University Santiago de Compostela on the website of the subject Declarative Programming gives some keys to solve problems declaratively: "**think about specifying the problem and its constraints and not about sequencing instructions;** "

Important recommendations to solve the exercises:

- Do not try to follow the step-by-step execution of the predicates to find the solution.
- Think of predicates as static.

Follow these steps:

1) **Raise the recursive case with the size of the recursive call a little smaller** (n-1, the rest of a list, etc.);

2) Then ask: **If the recursive call works, what do I have in the "partial" output?** (Use a simple example if necessary to understand what the output variables of the recursive call contain.)

3) Once this is clarified, ask **what changes must be made to the output of the recursive flame ("partial") to obtain the "total" output?** (that of the general case n, of the complete list, etc.).

4) Finally, **check that the basic case is correct and that all possible cases are contemplated.**

If after solving the problem in this way, **there is no assurance that it works... Try it in SWI-Prolog!**

Let's look at the following examples:

1) Invert a list
2) Concatenar of lists
3) Bubble sorting
4) Sorting by insertion
5) Sorting by Quicksort
6) Find the primes between X and Y
7) Swap items in a list
8) Find the item that appears most often in a list

1) Invert a list

```
%-----------------------------------------------------
% invest(+List, -ListR) is true when ListaR
% unifies with a list containing the same
% items listed in reverse order.
%-----------------------------------------------------
invest([], []).
invest([Cab| Rest], RT) :-
   invest(Rest, R),
   append(R, [Cab], RT).
```

2) Concatenar of lists

```
%-----------------------------------------------------
% concatenate(+List1, +List2, -ListR).
%is true when ListR unifies with a list
%containing the items in the List1 list
%in the same order and followed by items
%of the List2 list in the same order.
%-----------------------------------------------------
concatenate([], L, L).
concatenate([Cab| Rest], List2, [Cab| R]):-
   concatenate(Rest,List2,R).
```

3) Bubble sorting

```
%-----------------------------------------------------
% ordena_burbuja (+List, -List).
%is true when ListR unifies with a list that
%contains the same items as Sorted List
%from lowest to highest.
%-----------------------------------------------------
ordena_burbuja(List, List):- sorted(List).
ordena_burbuja(List, RT):-
  append(Ini,[E1,E2| Fin], List),
  E1>E2,
  append(Ini, [E2,E1| Fin], R),
  ordena_burbuja(R, RT).
%-----------------------------------------------------
% sorted(+List)
%is true when List unifies with a list
%containing its items sorted from least to
%major.
%-----------------------------------------------------
ordered([]).
ordered([_]).
ordered([Cab1, Cab2| Rest]):-
   Cab1 =< Cab2,
   ordered([Cab2| Rest]).
```

4) Sorting by insertion

```
%------------------------------------------------------
% inserta_en_list_ord (+Elem, +List, -List).
%is true when ListR unifies with a list
%containing the items in the sorted list
%List, with Elem element inserted in shape
%sorted.
%------------------------------------------------------
inserta_en_list_ord(Element, [], [Element]).
inserta_en_list_ord(Elem, [Cab| Rest], [Elem,
Cab| Rest]):-
   Elem =< Cab.
inserta_en_list_ord(Elem, [Cab| Rest], [Cab| R]):-
   Elem > Cab,
 inserta_en_list_ord (Elem, Resto, R).
%------------------------------------------------------
% ordena_insercion (+List, -List).
%is true when ListR unifies with a list that
%contains the same items as Sorted List
%from lowest to highest.
%------------------------------------------------------
ordena_insercion([],[]).
ordena_insercion([Cab| Resto], RT):-
   ordena_insercion (Rest, R),
   inserta_en_list_ord(Cab,R, RT).
```

5) Sorting by Quicksort

```
%------------------------------------------------------
% divides(+Elem, +List, -Minors, -Majors)
%is true when Minors unifies with a list that
%contains List items that are smaller
%o equal to Elem and Seniors unifies with a list
%containing List items that are
%greater than Elem.
%------------------------------------------------------
-
divide(_,[],[],[]).
divide(Elem, [Cab| Rest], Minors, [Cab| Seniors]):-
  Cab > Elem,
  divide(Elem, Rest, Minors, Majors).
divide(Elem, [Cab| Rest], [Cab| Minors], Major):-
  Cab =< Elem,
  divide(Elem, Rest, Minors, Majors).
%----------------------------------------------------
% ordena_quick (+List, -List).
%is true when ListR unifies with a list that
%contains the same items as Sorted List
%from lowest to highest.
%---------------------------------------------------
ordena_quick([],[]).
ordena_quick([Cab| Rest], R):-
  divide(Cab, Resto, Men, May),
  ordena_quick(Men, RMen),
  ordena_quick(May, RMay),
  append(RMen, [Cab| RMay], R).
```

6) Find the primes between X and Y

```
%-------------------------------------------------
% lista_divisores(+X, +Y, -ListaR).
%is true when ListR unifies with a list
%containing the numbers whose remainder
%of the integer division of X by Z equals 0
%for Z values between 1 and Y.
lista_divisores(_, 1, [1]).
lista_divisores(X, Y, [Y| R]):-
  AND > 1,
  Y2 is Y-1,
  lista_divisores(X,Y2, R),
  0 is X mod Y.
lista_divisores(X, Y, R):-
  AND > 1,
  Y2 is Y-1,
  lista_divisores(X,Y2, R),
  Z is X mod Y, Z \== 0.
%----------------------------------------------------
% prime(+X)
%is true if X unifies with a prime number.
%----------------------------------------------------
first(X):- lista_divisores(X,X,[X,1]).
%----------------------------------------------------
% primesBetweenxy(+X, +Y, -ListR)
%is true when ListR unifies with a list
%containing primes ranging from X to
%And both included in ascending order.
%----------------------------------------------------
primosEntrexy(X,X,[]).
primesEntrexy(X,Y, [X|R]):- X<Y,
  X2 is X+1,
  primosEntrexy(X2,Y, R),
  first(X).
primesEntrexy(X,Y, R):- X<Y,
  X2 is X+1,
  primosEntrexy(X2,Y, R),
  \+ first(X).
```

7) Swap items in a list

```
%---------------------------------------------------
% selecciona_uno(+List, -Elem, -Rest)
%is true when Elem unifies with any
%list item List and Unifies Rest
%with a list containing the items of
%List, in the same order minus the item
%Element.
%---------------------------------------------------
selecciona_uno([Ca| A], Ca, R).
selecciona_uno([Ca| Co], E, [Ca| R]):-
  selecciona_uno (Co, E, R).
%---------------------------------------------------
% swap (List, List).
%is true when ListR unifies with a list
%containing List items in order
%different. This predicate generates all the
%Possible lists by backtraking.
%---------------------------------------------------
trade-in([], []).
trade-in(L, [E| RP]):-
  selecciona_uno(L, E, R),
  exchange (R, RP).
```

8) Find the item that appears most often in a list

```
%-------------------------------------------------------
% mas_veces (+List, -Elem, -Num)
%is true when Elem unifies with the element
%repeated most times in the List list
%y Num unifies with the number of times it is
%Repeat that item.
%-------------------------------------------------------
mas_veces([],_,0).
mas_veces([Ca| Co], Ca, N2):-
  mas_veces(Co,El,N),
  Ca=He,
  N2 is N+1.
mas_veces([Ca| Co], He, N):-
  mas_veces(Co,El,N),
  Ca\=He.
```

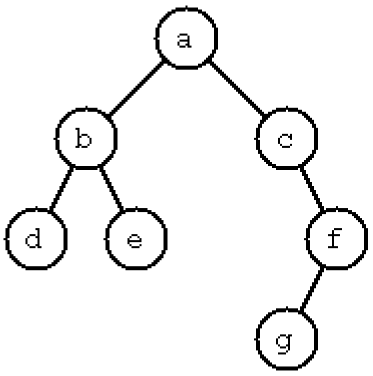
Logic programming with Prolog

Sección 2. Data types (continued)

2.2 Trees

Binary trees

A binary tree is
- Null tree, or,
- A structure composed of one element and two successors that are binary trees.



In Prolog we will represent a null tree with the atom 'nil' and the non-empty tree with the term a(Et, HI, HD), where "Et" represents the root label and HI and HD are the left and right subtrees respectively.

```
A1=a(a,a(b,a(d,nil,nil),a(e,nil,nil)),a(c,nil,a(f,a(g,nil,nil),nil)))
```

Other examples are a tree containing only one node
`A2 = a(a,nil,nil)` or the empty tree `A3 = nil`

Let's look at an example with the cuenta_nodos predicate for binary trees.

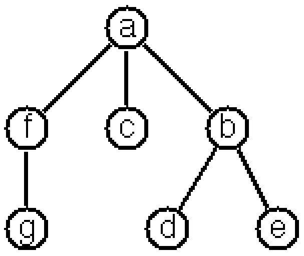
```
/* cuenta_nodos(+Arbol_binario, ? Num_nodos)

It is true when Num_nodos unifies with the
Number of nodes in the "Arbol_binario" tree */


cuenta_nodos(nil, 0).

cuenta_nodos(a(_, HI, HD), R):-
  cuenta_nodos(HI, RI),
  cuenta_nodos(HD, RD),
  R is RI + RD + 1.

dato(a(a,a(b,a(d,nil,nil),a(e,nil,nil)),a(c,nil,a(f,a(
g,nil,nil),nil)))).

/* 1 ?- data(A), cuenta_nodos(A, N).
A = a(a, a(b, a(d, nil, nil), a(e, nil, nil)), a(c,
nil, a(f, a(g, nil, nil), nil)))
```

Another example is the lista_hojas predicate for binary trees.

```
/*  lista_hojas(+Arbol_binario, ? Lista_hojas)

It is true when Lista_hojas unified with a list
containing the labels of the leaves of the
"Arbol_binario" tree.
*/

lista_hojas(nil, []).

lista_hojas(a(_, nil, HD),LD):-
  HD \=nil,
  lista_hojas(HD, LD).   lista_hojas(a(_,HI,nil),
LI):-
  HI \= nil,
  lista_hojas(HI, LI).
   lista_hojas(a(_, HI, HD), LR):-
  HI \= nil, HD\=nil,
  lista_hojas(HI, LI),
  lista_hojas(HD, LD),
  append(LI, LD, LR).  lista_hojas(a(Et, nil,nil),
[Et]).



dato(a(a,a(b,a(d,nil,nil),a(e,nil,nil)),a(c,nil,a(f,a(
g,nil,nil),nil)))).

/* 1 ?- data(A), lista_hojas(A, R).

A = a(a, a(b, a(d, nil, nil), a(e, nil, nil)), a(c,
nil, a(f, a(g, nil, nil), nil)))
R = [d, e, g] */
```

Generic trees

A generic tree is composed of a root and a list of successors that are themselves generic trees. Although we could establish a representation in our approximation, for simplicity we will consider that a generic tree will never be empty, that is, we will not have an equivalent to the null tree of binary trees.



In Prolog we will represent a generic tree by the term a(Et, ListChildren), where `Et` is the root label and `ListChildren` is a list with the descendant trees. The example tree will be represented by the following term Prolog.

```
A = a(a,[a(f,[a(g,[]))),a(c,[]),a(b,[a(d,[])),a(e,[])]))
```

Methodology for the implementation of generic trees

With the idea of simplifying the implementation of generic tree predicates, we will write clauses that unify with a tree structure and another clause(s) that will unify with tree lists.

In the case of clauses for tree lists, we will follow the same principle used in lists.

- A base case, which will usually be empty list. - A recursive case, which separates the first from the rest, makes the recursive call on the rest of the list (in this case trees) and, from the result, builds the solution.

We will implement the cuenta_nodos predicate for generic trees as an example:

```
/*  cuenta_nodos(+Arbol_generico, ? Num_nodos)

   It is true when Num_nodos unifies with the
   Number of nodes in the "Arbol_generico" tree */

%clause for a generic tree
cuenta_nodos(a(_, Lista_hijos), R):-
   cuenta_nodos(Lista_hijos, N),
   R is N + 1.

% clauses for tree list
cuenta_nodos([], 0).
cuenta_nodos([Cab| Rest], ):-
  cuenta_nodos(Cab, NCab),
  cuenta_nodos(Rest, NResto),
  R is Ncab + Nresto.

data(a(a,[a(f,[a(g,[])]),a(c,[]),a(b,[a(d,[]),a(e,[]])))).

/* given(A), cuenta_nodos(A,N).
A = a(a, [a(f, [a(g, [])]), a(c, []), a(b, [a(d, []), a(e,
[]))])
N = 7 */
```

Another example is found in the depth predicate for generic trees:

```
/*
profundidad_ag(+Arbol_generico, ? P)
   it is true when P unifies with the depth of
   Generic tree "Arbol_genérico"

*/

profundidad_ag(a(_, Lista_hijos), R):-
 profundidad_ag(Lista_hijos, PH),
 R is PH+1.

profundidad_ag([], 0).

profundidad_ag([Cab| Rest], PCab):-
   profundidad_ag(Cab, PCab),
   profundidad_ag(Rest, PResto),
   PCab >= PResto.

profundidad_ag([Cab| Rest], PResto):-
   profundidad_ag(Cab, PCab),
   profundidad_ag(Rest, PResto),
   PCab < PResto.

data(a(a,[a(f,[a(g,[])]),a(c,[]),a(b,[a(d,[]),a(e,[]])
)).

/*
given (A), profundidad_ag(A,N).

A = a(a, [a(f, [a(g, [])]), a(c, []), a(b, [a(d, []),
a(e, [])])])
N = 3
```

Exercise:

Consider how the proposed generic tree representation could be extended to represent null generic trees.

Logic programming with Prolog

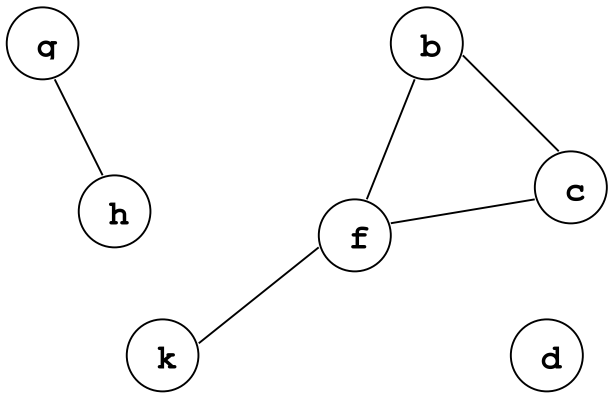
Sección 2. Data types (continued)

2.3 Graphs

Referencia: Ninety-Nine Prolog Problems:
https://prof.ti.bfh.ch/hew1/informatik3/prolog/p-99

A **graph** is defined as a set of nodes and a set of edges, where each edge joins two nodes.

There are different ways to represent a graph in Prolog. One of these methods is to represent each arc with an independent (done) clause. To represent the following graph with this method we will write:



```
arista(h,g).
arista(k,f).
arista(f,b).
...
```

We can call this representation "*Clau-edge representation*". With this method we cannot represent the isolated nodes (in the previous example the node "d"). This representation can make the implementation of the path somewhat easier, however it complicates tasks such as going through all the edges or visiting all the nodes.

Another method is to represent the entire graph in a single object. Following the definition given above of a graph as a pair of two sets (edges and nodes), we can use the following term in Prolog to represent our example graph:

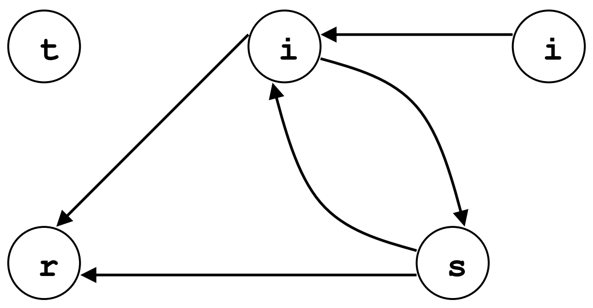
```
Graro([B,C,D,F,G,H,K],[ARISTA(B,C), ARISTA(B,F), ARISTA(C,F), ARISTA(F,K),
ARISTA(G,H)])
```

or in a more compact way

```
Graro([B,C,D,F,G,H,K],[A(B,C), A(B,F), A(C,F), A(F,K), A(G,H)])
```

We will call this representation "*Term-Grapher Representation*". Note that the set is represented by a list with no repeating items. Each edge appears only once in the list of edges; For example, an edge from node X to node Y is represented as A(x,Y) and the term `A(Y,x)` does not appear in the list. The *Term-Graph Representation is the representation that we will use by default*.

In SWI Prolog there are predefined predicates for working with sets. Run `help(11-1-1).` in SWI Prolog for information about the "Set Manipulation" section.



Other representations can be found at "Ninety-Nine Prolog Problems"
https://prof.ti.bfh.ch/hew1/informatik3/prolog/p-99/

The edges can be directed, when they only bind the nodes involved in the origin direction -> destination and not in the opposite; or undirected when binding nodes both ways.

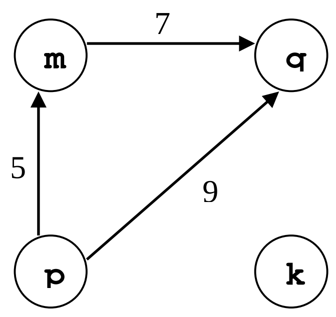
When the edges are directed we will call the graph Directed Graph.

Depending on the problem to be solved we will need to use one type of graph or another.

Nodes and edges of directed and undirected graphs can incorporate additional information. It is possible to replace the name of the node with a structure containing, for example, the name of the city and the postal code city `('Huelva',27002)`. On the other hand, it is also possible to add information to the edges, such as the cost of traversing them.

We will call graphs that add additional information to vertices or edges "labeled graphs".

Example:



Edge-clause representation

```
        arista(m,q,7).
        arista(p,q,9).
        arista(p,m,5).
```

Term-Graph Representation

```
        grafo([k,m,p,q],[arista(m,p,7),arista(p,m,5),arista
        (p,q,9)])
```

Most graph

problems are solved by constructing a path that joins two nodes of the graph.

Predicates that find a path in a graph are recursive.

The predicate path will have the following header

```
way(+Start, +End, +Graph, +Visited, ? Road
<? Peso_total>, <? Path2>)
```

Start: Represents the initial node of the path
End: Represents the end node of the road
Graph: Graph in which we look for the way
Visited: List of nodes or edges (depending on the case), which is used to prevent the predicate from iterating in a cycle.
Path: List of nodes or edges (depending on the case) that represents the path.

The above fields are required in most graph problems.

<Peso_total> : Sometimes it is necessary to indicate what the total weight of the road is.

<Camino2> : Sometimes it is necessary to indicate both the list of visited nodes and the list of visited edges. Example roads visited, cities visited and kilometers traveled.

Base case:
First possibility and most common.

```
way(End, End, _, []).
```

We have indicated before that the list of visited has the sole function of preventing the predicate from falling into an infinite loop. In the base case, having no recursive call, the list of `Visited` is irrelevant. We can make the following reasoning: **If I am already at the end of the road I have finished, regardless of the points I have passed.**

The output variable Camino, in this case will be an empty list or a list with the End node if what I want to return is the list of nodes I have visited.

Alternative base case:

```
way(Start, End, _, [edge(Start, End)]):-
  arista(Start, End).
```

We will use it only when it is essential. In general, the above base case works well and is simpler. One type of problem in which we will use this alternative base case is that of obtaining the cycles of a graph.

Recursive case:

To build the recursive case we will seek to advance to some `vertex TMP` and from that point we will look for a path to the end.

```
Home ----→   TMP ----------------------→ End

path(Start, End, graph(Vertices, Edges),
Visited, [edge(Home,TMP)| Way]):-
  connected(Home, TMP, Edges),
  \+ visited(Start, End, Visited),
  path(TMP, End, [edge(Start, TMP)| Visited],
 Way).
```

Depending on the type of graph and the objective pursued, the connected and visited predicates can be implemented differently.

Connected predicate on undirected graphs:

```
% connected(+Start, +End, +Edges)
connected(Start, End, Edges):- member(edge(Start, End), Edges).

connected(Start, End, Edges):- member(edge(End, Start), Edges).
```

In this predicate the cut "!" (which we will discuss in detail in section 3) is intended to prevent Prolog from attempting to seek a solution along a path that we know will not return any valid solution.

Connected predicate on directed graphs:

```
% connected(+Start, +End, +Edges)
connected(Start, End, Edges):- member(edge(Start, End), Edges).
```

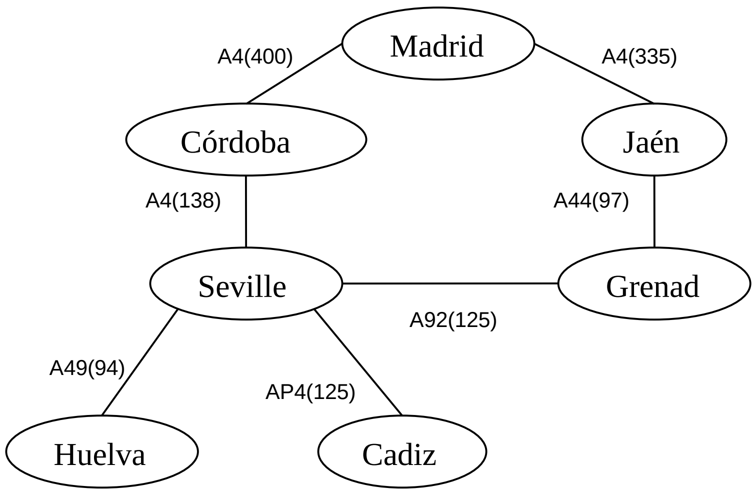
Predicate `visited` in undirected graphs:

```
% visited(+Start, +End, +Visited)
visited(Start, End, Edges):- member(edge(Start, End), Visited).

visited(Start, End, Edges):- member(edge(End, Start), Visited).
```

Predicate `visited` in directed graphs:

```
% visited(+Start, +End, +Visited)
visited(Start, End, Edges):- member(edge(Start, End), Visited).
```

Examples:



In this case, it is an undirected graph.

```
Data(graph([Madrid, Cordoba, Seville, Jaen, Granada,
Huelva, Cadiz],
         [arista(huelva, sevilla, a49, 94),
          Arista(Seville, Cadiz,AP4, 125),
          Arista(Seville, Granada, A92, 256),
          Arista(Granada, Jaen,A44, 97),
          Arista(Seville, Cordoba, A4, 138),
          Arista(Jaen,Madrid, A4, 335),
          Arista(Cordoba, Madrid, A4, 400)]
)).
```

```
% connected(+Home, +End, +Edges, -Road, -Kilometers)
connected(Start, End, Edges, C, K):-
  member(edge(Start, End,C,K), Edges).


connected(Start, End, Edges, C, K):-member(edge(End, Home,C,K), Edges).



% visited(+Start, +End, +Visited)
visited(Start, End, Visited):-member(edge(Home, End,_,_), Visited).
visited(Start, End, Visited):- member(edge(End,Start,_,_), Visited).


%road(Home, End, Grafo, Visited, Cities,Roads, Kilometers)
way(End, End, _, _, [End], [], 0).


road(Start, End, G, Visited, [Home| Cities],[Road| Roads], K2):-
  G = graph(_, edges),
  connected(Home, TMP, Edges, Road, K),
  \+ visited (Home, TMP, Visited),
  path(TMP, End, G, [edge(Home,TMP,_,_)| visited], cities, roads, kilometers),
  K2 is Kilometros + K.


% data(G), camino(huelva, madrid, G,[],C,Ca,K).
```

We will execute the objective `path` to find the solutions. In this
If we want to find alternatives to get from Huelva to
Córdoba. The first option passes through Granada and Madrid traveling 1182
kilometers and the second has a length of 232 kilometers.

```
1 ?- data (G), road (huelva, cordoba, G, [], Ca,C,K).
G = graph([madrid, cordoba, sevilla, jaen, granada,
Huelva, Cadiz], [Arista(Huelva, Seville, A49, 94),
Arista(Sevilla, Cadiz, AP4, 125), Arista(Sevilla,
Granada, A92, 256), Arista(Granada, Jaen, A44, 97),
Arista(Seville, Cordoba, A4, 138), Arista(Jaen, Madrid,
A4, 335), Arista (Cordoba, Madrid, A4, 400)]),
Ca = [huelva, sevilla, granada, jaen, madrid, cordoba],
C = [a49, a92, a44, a4, a4],
K = 1182 ;
G = graph([madrid, cordoba, sevilla, jaen, granada,
Huelva, Cadiz], [Arista(Huelva, Seville, A49, 94),
Arista(Sevilla, Cadiz, AP4, 125), Arista(Sevilla,
Granada, A92, 256), Arista(Granada, Jaen, A44, 97),
Arista(Seville, Cordoba, A4, 138), Arista(Jaen, Madrid,
A4, 335), Arista (Cordoba, Madrid, A4, 400)]),
Ca = [huelva, sevilla, cordoba],
C = [a49, a4],
K = 232 ;
false.
```

Logic programming with Prolog

Sección 2. Control of implementation

The *cut*

We will notice the cut with an admiring closure "!". Given the following clause containing the cut:

`H:- B1, B2,..., Bm, !, Bm+1, ..., Bn`.

Consider that it was invoked by a goal **`G`** that unifies with **`H`**.

First effect of the cut:
By the time the cut-off is reached, the system has already found solutions for the terms `B1,B2,...,Bm`. **The solutions found for `B1,B2,..` `Bm` remain but any other alternative for these objectives is ruled out**. For the rest of the `Bm+1,...,Bn objectives, the usual` techniques will be applied *backtraking* .

Second effect of the cut:
Any attempt to satisfy objective `G` with another clause is ruled out.

There are two modes of use of cutting. What some authors call green cuts and red cuts.

Green ***cuts*** are those that **do not modify the solutions of the program**. That is, with cutting and without cutting the program produces the same solutions. It is used to improve execution efficiency.

The ***red cuts*** are those that **modify the solutions of the program** (when removing the cut the solutions are different). These types of cuts make programs less declarative and should be used with reservations. In declarative programming no matter the order in which the clauses are written, the solutions must be the same. However, the use of red cuts prevents the clauses from changing order.

Example:

```
cuenta_hojas(nil,0).
cuenta_hojas(a(_,nil,nil), 1):- !.
cuenta_hojas(a(_,HI,HD), R):-
  cuenta_hojas(HI, RI),
  cuenta_hojas(HD, RD),
  R is RI+RD.
```

In this example, if we eliminate the cut, for the query `cuenta_hojas(a(1,nil,nil), R)`, the predicate would give two solutions, one correct (R=1) and one incorrect `(R=0)`.

Another version of the same problem with "green" cuts:

```
cuenta_hojas(nil,0).
cuenta_hojas(a(_,nil,nil), 1):- !.

cuenta_hojas(a(_,HI,HD),  ):-
  HI \= nil, HD \= nil, !,
  cuenta_hojas(HI, RI).
  cuenta_hojas(HD, RD),
  R is RI + RD.

cuenta_hojas(a(_, nil, HD), RD):-
  HD \= nil, !,
  cuenta_hojas(HD, RD).

 cuenta_hojas(a(_, HI, nil), RI):-
  HD \= nil,
  cuenta_hojas(HI, RI).
```

In this other example, by eliminating the cuts the program gets the same solutions. In this case, the cut prevents some checks from being performed. For example, if we reach the cut of the usual clause 2, it does not make sense to look for a solution in the clauses that appear below.

Resolution tree

Also known as a deduction tree, it represents how Prolog resolves a particular query.

Resolution algorithm.

Repeat while there are terms:

1) **We search** the top-down knowledge base for **unifications of the objective term in the heads of rules and in facts.**

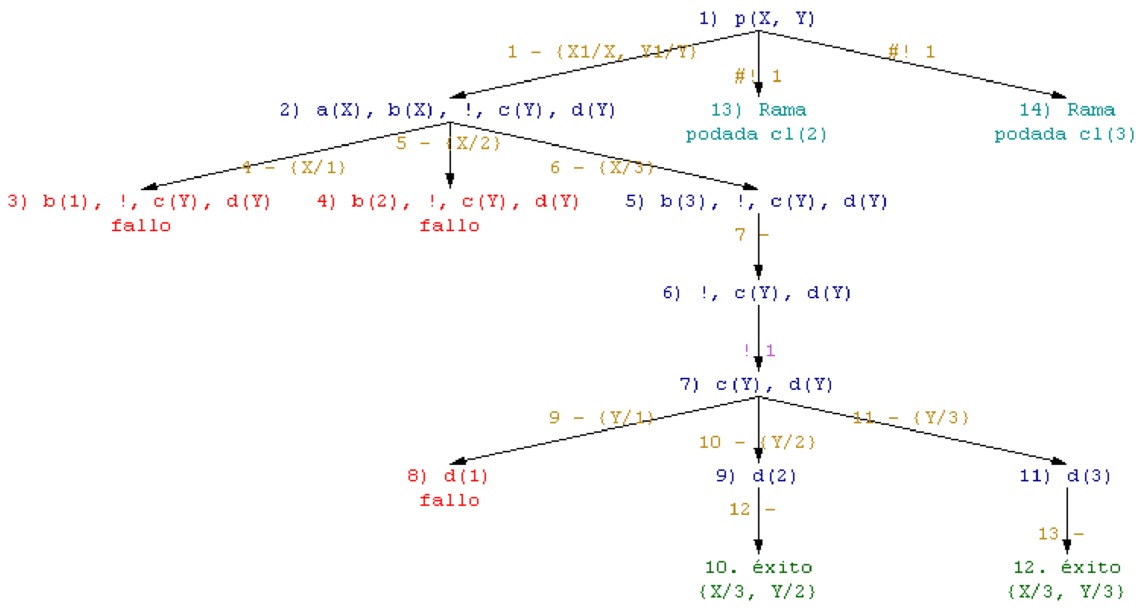
   a. We open **as many execution branches as unifications** and begin to solve for the one that is further to the left.

   b. If there **is no unification, a failure occurs and the next branch is scanned immediately to the right** at the same level. If there were no branches at the same level, we would move to the upper level starting with the branch that is most to the left.

2) In the current branch, **we substitute the head for the body of the rule intanciando the variables.** We set as the current objective term the one that is most to the left and we turn to step 1.

Example:

```
1.  p(X,Y):- a(X), b(X), !, c(Y), d(Y).
 2. p(1,Y):- c(Y).
3. p(X,2):- c(X).
4. a(1).
5. a(2).
6. a(3).
7. b(3).
8. b(4).
9. c(1).
10. c(2).
11. c(3).
12. d(2).
13. d(3).
```



Steps for creating resolution trees

1) **We will number the** clauses of the program from top to bottom in ascending order.

2) We will write the proposed objective and **look for unifications of the objective in the heads of the rules and in the facts.**

3) **We will draw as many branches as there are unifications** and **label each branch with the number of the clause**. If there is no unification we will note a failure for the objective.

4) **We will start by solving by the branch that is most to the left**.

5) We will note the substitution of variables in the branch and **replace the head with the body of the rule,** without forgetting the previous terms if any.

6) **We will work out the terms from left to right**. Remember that for the consequent to be true, all the terms of the antecedent must be true. If one of the terms is false, we will not continue to check the rest of the terms.

7) **When a term becomes true, we remove it and continue to solve the rest from left to right**.

8) **When we have solved all the terms, we will mark that branch with "success" and write down the values of the solutions** for the variables of the objective.

Treatment of cutting in resolution trees

When we reach a cut in a resolution tree, we will follow these steps:

1) **We will identify the clause that contains the cut and the** elements `H`, `G`, the terms that are before the cut **`B1, B2,..., Bm`** and those that are after **`Bm+1, ..., Bn.`** As indicated by the definition of the cut **`H:- B1, B2,...,Bm,!, Bm+1, ..., Bn`**.

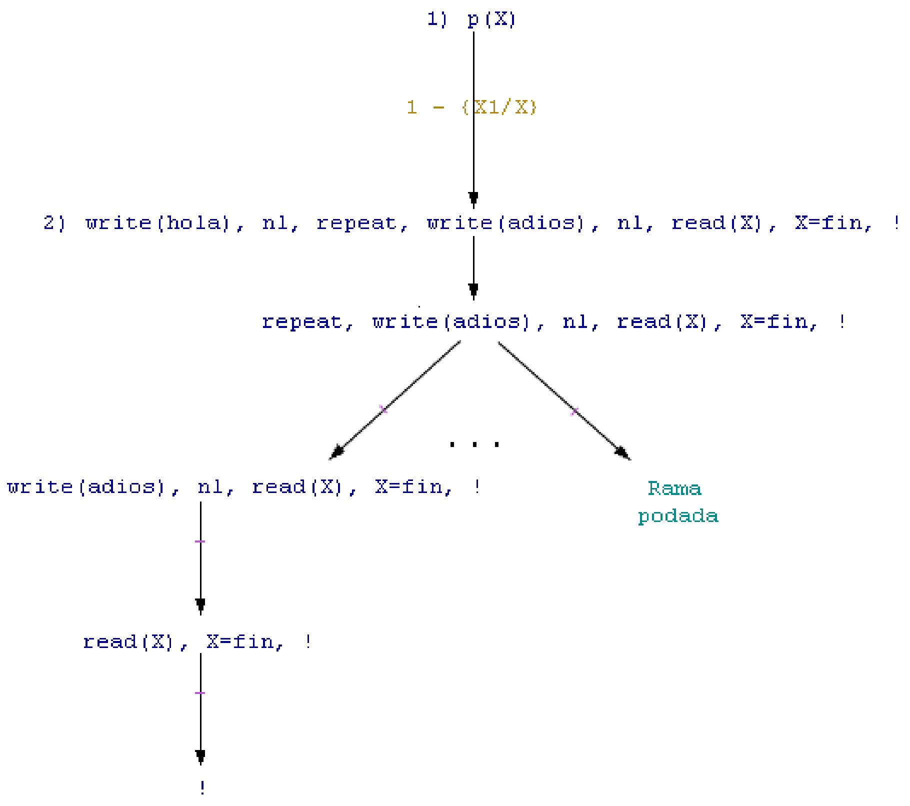
2) **We will check if there are terms before the cut**. If they exist we will have to apply the first effect of the cut, marking with '**`Pruned branch`**' the branches corresponding to the alternatives for the terms that are before the cut.

3) We will identify the objective we were trying to satisfy (G) that unified with the head (**`H`**) of the ruler that contains the cut and, applying the second effect, we will mark as **`'Pruned branch`**' the branches corresponding to the attempt to satisfy objective G with other clauses.

Term "repeat"

The term repeat has an effect similar to a term that had infinite unifications. Let's look at an example:

```
1. p(X):- write(‘hola’), repeat, write(‘adios’),
read(X), X=’fin’, !.
```


1) p(X)
1 - {X1/X}
2) write(hola), nl, repeat, write(adios), nl, read(X), X=fin, !
repeat, write(adios), nl, read(X), X=fin, !
. . .
write(adios), nl, read(X), X=fin, !
Rama podada
read(X), X=fin, !
!

This example once shows `hello`, type `goodbye` and read a term by keyboard. If the term equals `end`, the cut is reached and all branches of the repeat are eliminated and the program ends. If no end is introduced, the program does not end.

'fail' element

When the term fail is reached in a clause, it becomes false.

Example to test:

```
q([C|R],0):-s(C),q(R,1).
q([C,next(a)],1):-!,r([C]).
q([C,[N]],1):-r([C]),!,N is 2+3.
r(1).
r([[]]).
r([next(a)]).
s(R):-!,t(R).
s(1):-q([],1).
t(1):-fail.
```

SLD-Draw

You can use the SLD-Draw tool (http://www.lcc.uma.es/~pacog/sldDraw/) to draw resolution trees that do not contain '`repeat`'. If you are in doubt about whether a resolution tree is correct, the best option is to implement it, enter it in Prolog, and request all solutions using ";". Prolog will give us in all cases all valid solutions. SLD-Draw sometimes does not produce all valid solutions.

Logic programming with Prolog

Sección 3. State problems

A method to solve problems using logic programming is to trace a path that goes through different situations (states) until reaching a solution.

A transition from one state to another occurs by making a move. In this way, the key to solving a problem with this methodology will be to define a state and movements; And based on these two elements a path will be built that starts from an initial state and after a sequence of movements reaches a final state.

We will build the solution in four steps:

1) **Definition of a generic state and the initial and final states.**
2) **Implementation of movements.**
3) **Implementation of the road.**
4) **Implementation of the solution predicate.**

We will explain the resolution of state problems using the example of cannibals and missionaries.

Cannibals and Missionaries

Statement of the problem:

- There are three missionaries and three cannibals on the left bank of a river
- Both missionaries and cannibals wish to cross to the right bank of the river.
- To cross, they have a boat that can transport only two people at a time.
- The number of cannibals on each shore should never be greater than the number of missionaries, otherwise the missionaries would become the dinner of the cannibals.
- Plan a sequence of movements to cross the missionaries and cannibals so that in the end they are all safe and sound on the right bank.

We will represent the problem as a set of:

- States: which represent a snapshot of the problem. - Movements: which transform one state into another.

We find a relationship between state problems and graphs by associating nodes with states and edges with motions. If we remember the way to construct a path in graphs, the sequence of motions of a state problem will have basically the same structure.

1) Definition of a generic state and the initial and final states.

- A state represents an instant of the problem.
- Using the information of the term `state( ... )` We must be able to represent the problem without losing relevant information avoiding duplicating information.
- The status does not include action information in the status.

Turning now to the problem of cannibals and missionaries, the information we need to represent the problem is:

- The number of missionaries on the left bank
- The number of cannibals on the left bank
- The place where the boat is located.

All the necessary information to solve the problem, we can obtain it from these three data. In Prolog, we can represent this state with a term of 3 arguments.

```
State(Caníbales_izq, Misioneros_izq, Pos_barca)
```

Using this representation, we will define the initial and final states:

```
initial(state(3,3,left)).
(state(0,0,dch)).
```

2) Implementation of movements.

The next step is to define the possible movements. In the case of the problem of cannibals and missionaries we would have the following movements:

- Pass a missionary left or right.
- Pass a cannibal left or right.
- Pass two cannibals left or right.
- Pass two missionaries left or right.
- Pass a missionary and a cannibal to the left or right.

The move predicate, written '`mov`', will usually have the following header:

```
/* mov(? Movement? Estado_anterior, ? Estado_posterior) is true when
Movement unifies with a valid Movement, Estado_anterior unified with a
valid state, and Estado_posterior unified with the state resulting from
applying the "Movement" movement to the "Estado_anterior" state */
```

It is possible that for some kind of problem it is necessary to include some more argument. For example, if we wanted to calculate how long it takes to reach the solution, we can associate a cost in time to each movement.

In order to implement the movements, it is necessary to think about the necessary conditions before performing the movement, that is, how the state before the movement should be. For example, in order to pass a missionary on the left it is necessary that there is at least one missionary on the left and that the boat is on the right.

```
mov(pasar_un_mis_izq, state(MI, CI, dch), state(MI2, CI)):-
  MI < 3, MI2 is MI + 1.
```

We are not checking if the new state is a valid state. When we have many movements (in this case there are 10 movements) it is interesting not to include in the movements the checks of whether the new state is valid since we would repeat in each movement the same check. It is better to include a valid predicate `(State)` that will be true, when `State` is a valid state.

Another way to write the movements is to make them more generic to avoid writing several very similar movements. However, this simplification is not always easy to find. A simplification for movements is then proposed that reduces them to only two clauses plus a set of 10 facts. We will use a pass predicate that we will represent by 10 facts.

`pass(? Num_misioneros, ? Num_canibales, ? Place)` will be true when `Num_misioneros and Num_canibales` unifies with a combination of missionaries and cannibals valid according to the specification of the problem and when place unifies with `'left'` or `'right'`.

```
pass(1,0,left).
market(1,0,dch).
pass(0,1,left).
market(0,1,dch).
pass(2,0,left).
market(2,0,dch).
pass(0,2,left).
```

```
market(0,2,dch).
pass(1,1,left).
market(1,1,dch).
```

The movements would be as follows

```
mov(pass(M, C, left), state(MI,CI, dch), state(MD, CD, left)):-
  pass(M,C,left),
  NT is M + C, NT =< 2, NT >= 1,
  M = < MI, C = < CI
  MD is MI + M, CD is CI + C.

mov(pass(M, C, dch), state(MI, CI, left), state(MD, CD, dch)):-
  market(M,C,dch),
  NT is M + C, NT =< 2, NT >= 1,
  M = < MI, C = < CI
  MD is MI -M, CD is CI - C.
```

3) Implementation of the road.

The solution to this type of problem is given by a sequence of movements that, starting from an initial state, makes the problem evolve to the desired final state.

The implementation of this path is basically the same as we use in the path of graphs, using states instead of nodes and movements instead of edges.

The path predicate will have at least the following arguments:

```
/* path(+Estado_inicial, +Estado_final, +Visited, -Way)is true when
Estado_inicial and Estado_final unified with valid states, Visited unifies
with a list of visited states. */

path(Start, Start, _, []).

road(Start, End, Visited, [mov| Way]):-
  length(Visitados, L), L < 10,
  mov(Mov, Home, Int),
  \+ member(Int, Visited),
  road(Int, Fin, [Int| Visited], Way).
```

Depending on the problem, the path predicate may have some more argument if we want, for example, to save the total cost of the path.

This path has the particularity of limiting the length of the road to at most 11 (10 visited + 1 final step). This limitation makes sense when the possibilities of exploration are many and we know that in a maximum number of steps is the solution.

We can also include checking if the new state is correct after making the move. This check will be done with a validation predicate that we will call `valid (State).` In this case the check is performed inside the mov predicate.

Finally, indicate that the list of visited will contain states, since it is possible to repeat the same movement as long as you do not go through the same state twice (to avoid falling into an infinite loop).

4) Implementation of the solution predicate
This predicate makes use of `path` and the initial and final states and its implementation is very simple. Its goal is to make the correct call to the predicate `path`. It is important to note that the list of initially visited states must contain the initial state.

```
solution(Path):- initial(Ei), final(Ef), way(Ei, Ef,[Ei], Way).
```

The following is the complete implementation of the solution:

```
/* state(Canibales_izq, Misioneros_izq, Pos_barca) */
initial(state(3,3,left)).
(state(0,0,dch)).
/* pass(? Num_misioneros, ? Num_canibales, ? Place)
   It is true when Num_misioneros and Num_canibales unifies
with a valid combination of missionaries and missionaries
valid according to the specification of the problem and when
place unifies with 'left' or 'dch'. */
pass(1,0,left).
market(1,0,dch).
pass(1,1,left).
market(1,1,dch).
pass(0,1,left).
market(0,1,dch).
pass(2,0,left).
market(2,0,dch).
pass(0,2,left).
market(0,2,dch).

/* mov(? Movement? Estado_anterior, ? Estado_posterior) is
true when Movement unifies with a valid Movement,
Estado_anterior unified with a valid state, and
Estado_posterior unified with the state resulting from
applying the "Movement" movement to the "Estado_anterior"
state */

mov(pass(M, C, left), state(MI,CI, dch), state(MD, CD,
left)):-
  pass(M,C,left),
  NT is M + C, NT =< 2, NT >= 1,
  M = < MI, C = < CI
  MD is MI + M, CD is CI + C.

mov(pass(M, C, dch), state(MI, CI, left), state(MD, CD,
dch)):-
  market(M,C,dch),
  NT is M + C, NT =< 2, NT >= 1,
  M = < MI, C = < CI
  MD is MI -M, CD is CI - C.
/* path(+Estado_inicial, +Estado_final, +Visited, -Way)is
true when Estado_inicial and Estado_final unified with valid
states, Visited unifies with a list
*/
path(Start, Start, _, []).

road(Start, End, Visited, [mov| Way]):-
  length(Visitados, L), L < 10,
  mov(Mov, Home, Int),
  \+ member(Int, Visited),
  road(Int, Fin, [Int| Visited], Way).

solution(Path):- initial(Ei), final(Ef), way(Ei, Ef,[Ei],
Way).
/*
2 ?- solution(S). S=[pass(1, 1, right), pass(1, 0, left),
pass(1, 1, right), pass(1, 0, left), pass(1, 1, dch), pass(1,
0, left), pass(2, 0, right), pass(1, 0, left), pass(2, 0,
dch)]
*/
```

Referencias: Problem Solving in Prolog: **Bratko capítulo 12**
http://www.cse.unsw.edu.au/~billw/cs9414/notes/mandc/mandc.html

Internship

Internship with Haskell

Introduction to the Haskell Hugs programming environment

In this first practice we will familiarize ourselves with the Hugs programming environment and look at the basic data types.

Sección 1. User account

Each student will have a user account. The account has associated disk space. It is recommended not to leave any important material on this account. We will use this space, only as temporary storage space. After each session we will save the files on a USB stick or send them to our email account.

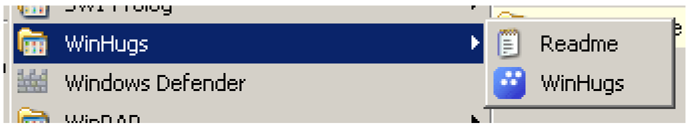
Sección 0. Starting

- Select the boot of the Windows XP operating system.

- Enter the login provided by the teacher. The first time you start the system you must change the password. Remember your password, as it will not be possible to consult your password later

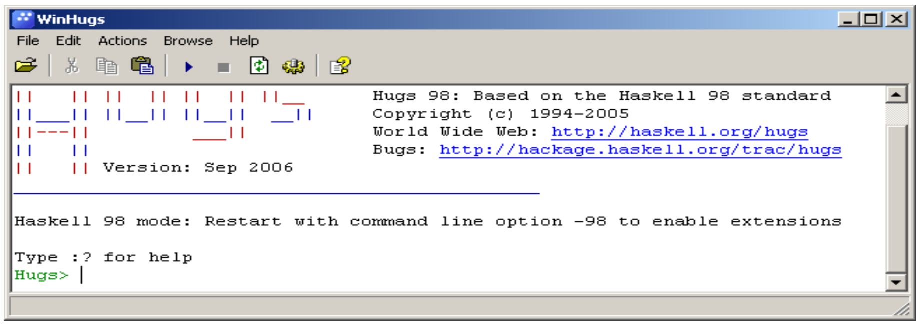
Sección 1. Run Hugs

- In the Windows "Start" menu select the WinHugs program



- A window like this will appear:

- From this window we will load the programs and make the queries. The mode of operation of this window is very similar to that of a calculator. When we write an expression in this window and press ↵, we ask the interpreter to show us the result of the expression.



```
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006
______________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> |
```

- Run in Hugs:

  ```
  Hugs> 1+1
  ```

  verify that the interpreter displays "2".

  Run

  ```
  Hugs> :type $$Hugs
  ```

  displays "`1+1 :: Num a =>`

  a" The initial expression has not been reduced, a "show" function has been used that shows the result of the expression, leaving this expression invariant.

Sección 2. Write a program

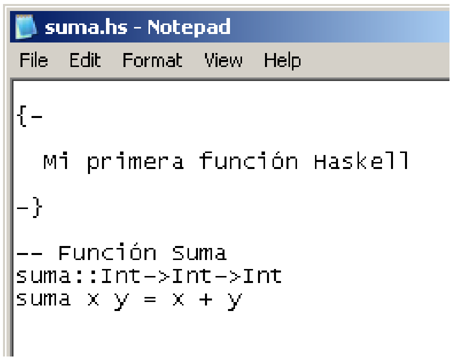
To write a program we will use a text editor that does not enter special characters. The Windows notepad will serve us. Hugs has this set as the default editor. We can change this editor using the "`:set`" command.

Before starting with the editing of the files, we will create the directory "`H:\declarative`".

The easiest way to write a new program is to enter :edit `<nombre_fichero>` in the Hugs:Hugs prompt

```
>   .
```

:edit "H:\\declarative\\suma.hs"This command will edit the `suma.hs file` We will write the following in the `suma.hs file`:



```
{-

  Mi primera función Haskell

-}

-- Función Suma
suma::Int->Int->Int
suma x y = x + y
```

We will save the contents of the file and close the window. While editing the file we will not be able to access the Hugs window.

Then we will load the file. To do this, we will write `:load <nombre_fichero>:`

```
Hugs>  :load  "H:\\declarative\\suma.hs"
```

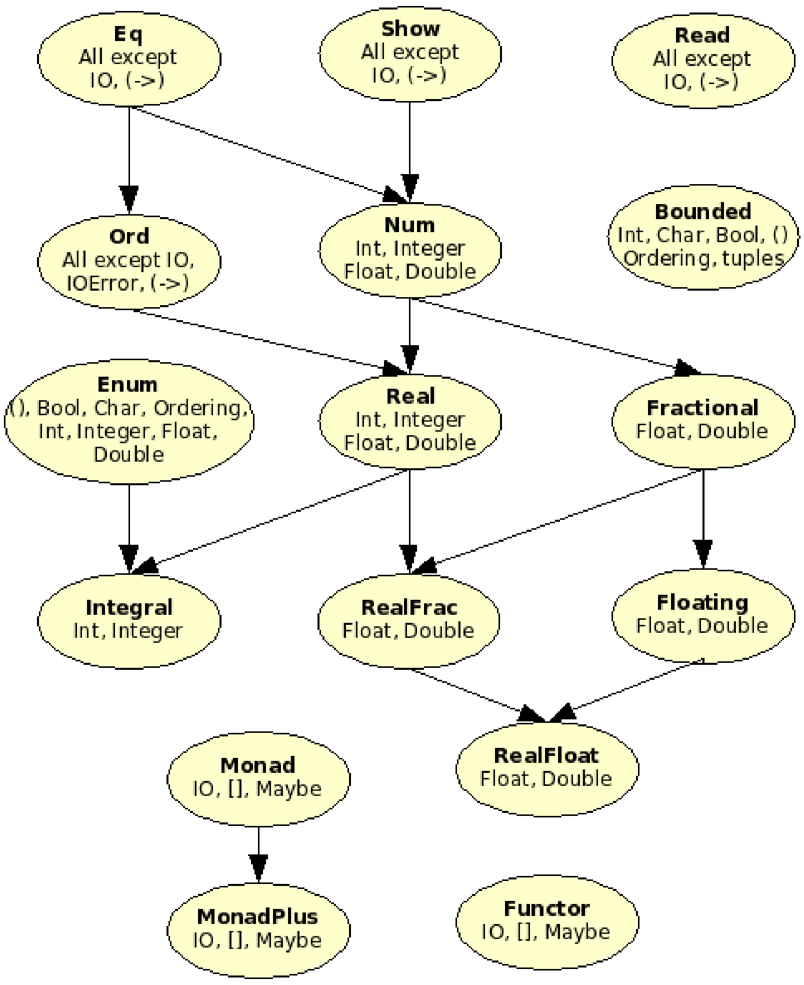
We can check if the functions have been loaded into memory by executing `:info <nombre_función>:`

```
Hugs> :info suma
suma :: Int -> Int -> Int
```

The result is the header of the `addition` function.

Sección 4. Data types

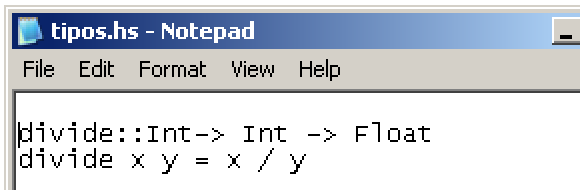
The following graph shows the relationship between the classes that group the different types of data:



The `Num` class, for example, includes the Int, `Integer`, `Float`, and `Double` types. The `Fractional class includes the` type `Float` and the `Double` type. Below we will see some examples that show particularities of Haskell with the data types:

```
Hugs>  :edit  "H:\\declarative\\types.hs"
```

We will write the following:



```
divide::Int-> Int -> Float
divide x y = x / y
```

Then we will save the file and load it into memory:

```
Hugs>  :load "H:\\declarative\\types.hs"
```

When loading the file the following error message appears:

```
ERROR file:h:\declarative\types.hs:3 - Type error in explicitly typed
binding*** Term : divide*** Type : Int -> Int -> Int*** Does not match :
Int -> Int -> Float
```

This error indicates that the definition we have given to the header does not match the header that Hugs has deduced.

To solve the problem, let's see what kind of data the "/" function expected. We will execute before `:load` without arguments so that it deletes from memory the file with the error and we can continue.

```
  Hugs> :load

Hugs> :info /infixl 7 /(/) :: Fractional a => a -> a -> a -- class member
```

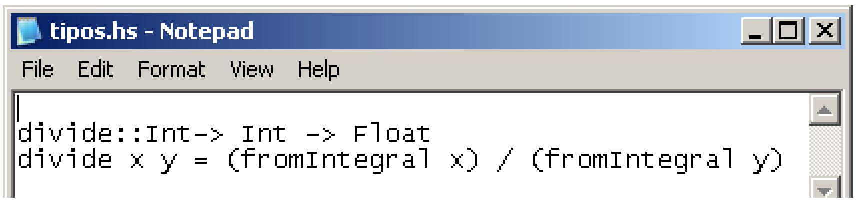
The header indicates that the data type that accepts the function "/" is of type `Fractional,  that` is, `Float` or `Double` and that all arguments are of the same type. For this reason we cannot define the split function of type `Int->Int->Float`

We can solve this problem in two ways.

(a) The first is to leave the header the same and using one of the functions that convert types.

In this case we would use the `fromIntegral function`:

```
  Hugs> :edit
```



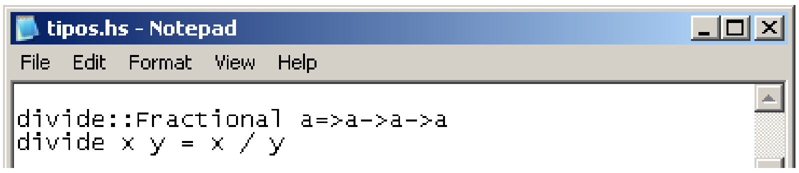
```
divide::Int-> Int -> Float
divide x y = (fromIntegral x) / (fromIntegral y)
```

```
 "H:\\declarative\\types.hs"Hugs>  :load "H:\\declarative\\types.hs"
Main> divide 4 2
2.0
```

Note that the `Hugs`> prompt indicates that there is no module loaded and that the Main prompt> indicates that there is at least one file loaded in memory.

(b) The second solution is to change the header to accept input values of type `Fractional`
"H:\\declarative\\types.hs"Main> :load "H:

```
Main> :Main> :edit
```



```
divide::Fractional a=>a->a->a
divide x y = x / y
```

```
 \\declarative\\types.hs"
 divide 4  2
2.0
```

To finish with the data types, remember that although Haskell is able to infer a header for the function from the implementation, **We will always write the header.**

Sección 3. Summary

:t	Displays the data type of the expression
:type	Displays the data type of the expression
:?	Displays the list of basic commands
$$	Last expression entered
:load	Upload a file
:edit	Open the default editor
:r	Update loaded modules
:reload	Update loaded modules
:set	Displays the list of options to enable/disable
:set +/-option	Turn an option on or off.
:set +t	Turn on the option that displays the data type
:q	Exit Hugs

Supplementary material

- Use the manual: "**A tour of the Haskell Prelude"**
http://www.cs.ut.ee/~varmo/MFP2004/PreludeTour.pdf
to consult the most common functions defined by default in Hugs.

Exercises

1. Implement the `double function` that receives one number and returns double.

2. Implement the square function that receives a number and returns its square.

3. Implement the function `isPositive` to be true when its only argument is greater than 0

4. Given the following definition of the maximum of two numbers: $\max(x,y)=\frac{(x+y)+|x-y|}{2}$
implement a function that returns the greater of two numbers of type `Float`.

5. Using the previous function, define a new function that calculates the maximum of three numbers.

6. Type the function `between 0y9` that takes an integer and returns `True` if it is between 0 and 9 and `False` otherwise.

7. Implement the function `is MultiploDe3` that takes an integer and returns True if it is a multiple of 3 or False otherwise.

8. Set the function the `fourequals::Int->Int->Int->Int->Bool` function that returns `True` if all four arguments are equal.

9. Use the above function to create a new so-called `lostresiguales` that receives three integers and is true if all three integers are equal.

10. Set the function `how manyequals::Int->Int->Int->Int` that receives three integers and returns an integer indicating how many of the three input arguments are equal.

11. Implement the function of exercise 4 using only integer values.

Lists in Haskell

In this session we will look at some Haskell mechanisms for defining lists.

Sección 1. Lists

In Haskell, a list is a sequence of values of the same type, for example `[1,2,3,4,5] is a list of 5 integers (type Int)`, on the other hand `[True, False, False, True]` is a list of 4 Boolean elements (Bool type).

Haskell allows you to define infinite lists, for example `[1..]` It is an infinite list of integers.

A text string (type `String`), is a list of characters (type `Char`). If we ask Haskell to evaluate the list `['a', 'b', 'c']` he will answer that the result is "`abc`".

Haskell provides a mechanism for easily defining lists.

Evaluate the following lists at the Hugs prompt:

a) `[1..10]`
b) `[15..20]`
c) `[15..(20-5)]`
d) `['q' .. 'with']`
e) `[14 .. 2]`

Are the results as expected? For numbers, characters, and other enumerated types (any type in which it makes sense to speak of the successor), the expression `[m.. n]` generates the list `[m,m+1,m+2,..,n]`. If `n` is less than `m` (as in example e) above), Haskell returns an empty list.

More generally, the expression `[m,p.. n]` generates the list:

```
[m,m+(p-m), m+2(p-m),m+3(p-m).. n']
```

where `n'` is the highest value of the form m+k*(p-m) ≤ n. In order to clarify this notation, run the following examples in Hugs:

a) `[1,3..10]`
b) `[15,15.5..20]`
c) `[1,1.2..2.0]`
d) `['a','d'..' z']`
e) `[10,8 .. -3]`
f) `[1,2 .. 0]`

What will happen when evaluating this expression: `[1,1..5]`?

Use the Actions -> Stop menu, if necessary.

Sección 2. Extended list notation

The definition of a list with this definition consists of three parts: 1) Generator, 2) Constraints (there may be none) and 3) Transformation. The generator produces items from a list, constraints filter some items from those generated, and the transformation uses the selected items to generate a result list.

Example:

```
[(x,True) | x <- [1 .. 20], even x, x < 15]
```

In the above expression, `x <- [1..20]` is the generator, the constraints are `even x` and `x < 15`; y `(x, True)` is the transformation. This expression generates a list of pairs `(x,True)` where x is between 1 and 20, is even and less than 15 (even is a function defined in Haskell's standard Prelude module). Check how the items appear in the generated order.

Check the previous result, with the evaluation of:

```
[(x,True)|x<-[20,19 .. 1],even x, x < 15]
```

Again, the elements appear in the order they were generated.

Evaluate:

```
[[m+n]| (m,n)<-[(3,6),(7,3),(8,4),(1,3),(4,8)],n==2*m]
```

This expression generates a list of lists of integers, with those pairs whose second component is twice as large as the first. This expression demonstrates how to use templates `(m,n)` to isolate a certain component from the tuple.

Now try the following expression:

```
[(x,y) | x <- [1..5], y == 1]
```

In this case, Haskell displays the following error `Undefined variable "and".` The variables are defined in the generator. If a variable does not appear in this section, you cannot use it in another section.

We can use extended list notation to define more general functions. For example, the following function is a generalization of the above expression:

```
aniadirPares::[(Int,Int)] -> [[Int]]
aniadirPares listaPares = [[m+n] | (m,n) <-
  listaPares, n == 2*m]
```

The function's type declaration specifies that it accepts a list of integer pairs and returns a list of integer lists.

Test the function with the following arguments:

```
aniadirPares [(3,6),(7,3),(8,4),(1,3),(4,8)]
aniadirPares [(3,7),(8,9),(1,3)]
```

Finally, the following example shows how to use extended list notation as part of a slightly more complex function. The following functions indicate when a list contains only even values (or only odd values).

```
todosPares, todosImpares::[Int]->Bool
todosPares lista=(list==[x|x<-list,x 'mod' 2==0])
todosImpares lista=([]==[x|x<-lista,x 'mod' 2==0])
```

For example, evaluate the following expressions:

```
todosPares [1 .. 20]
todosImpares [1 .. 20]
todosPares [1,3 .. 20]
todosImpares [1,3 .. 20]
todosPares []
```

Sección 3. Using templates with lists

The following function uses extended list notation. Receives a list as an argument and returns another list with all its elements multiplied by 2.

```
doblarTodos::[Int]->[Int]
foldAll list = [2*x |x <- list]
```

This list definition includes the `x <- list` generator and a `2*x` transformation. The output of the list always contains the same number of items as the original list because the definition does not include any restrictions. For example, `foldingAll [1,2,3,4,5]` returns a list with five items.

Next, a version of `Fold All` using recursion:

```
doblarTodos::[Int]->[Int]
doblarTodos [] = []
doblarTodos (cab:resto)=2*cab : doblarTodos
  remainder
```

The following is a recursive version of the `aniadirPares function`, implemented in the previous section:

```
aniadirPares [] = []
aniadirPares ((m,n):restaurant) = [m+n]: aniadirPares
  remainder
```

Again we can see how the recursive definition uses exactly the same transformation as the implementation using extended list notation.

Sección 4. Selecting items

We have seen that using extended list notation we can filter some items from the original list by adding restrictions. For example, the following function accepts a list of integers and returns twice as many items less than 10, removing the rest:

```
doblarAlgunos::[Int]->[Int]
doblarAlgunos list = [ 2*x | x <- list, x < 10]
```

Here is a version of the `bend function` using recursion:

```
doblarAlgunos [] = []
foldSome (cab:rest)
    | cab < 10 = 2*cab : doblarAlgunos restaurant
    | otherwise = cab : doblarAlgunos resto
```

This function is similar to the `DoubleAll function except that` elements are added`not all 2*x`. In this version we only add to the list those elements that satisfy that `x < 10`.

Another similar example is this implementation of the `aniadirPairs` function using recursion:

```
aniadirPares [] = []
aniadirPares ((m,n):restaurant)
   | n == 2*m = [m+n] : aniadirPares resto
   | otherwise = aniadirPares resto
```

Sección 5. Combining results

Extended list notation is useful for converting lists to other new lists, however, it does not provide a way to convert lists to something other than a list. For example, extended list notation cannot be used to find the maximum of one list.

We can use for this purpose the same type of templates that we have seen in recursive functions. The idea is that we can combine an item in the list with the result obtained by the recursive call.

For example, consider the following function that adds the items in an integer list:

```
sumaLista::[Int]->Int
sumaLista [] = 0
sumSum (cab:remainder) = cab + sumRemainder list
```

We can see how this definition is similar to the one we have seen above, except for the use of the "+" operator to combine the results, instead of the ":" operator; and because it makes no sense to return an empty list in the first equation, since `sumList` returns an integer.

Next we will see two other examples of recursive functions that combine the first element in the list with the result of evaluating the rest of it. In the first function, the "+" operator is used to add the square `(head*head)` of the first element:

```
sumaCuadrados::[Int]->Int
sumaCuadrados [] = 0
sumaCuadrados (cab:resto) = cab*cab + sumaCuadrados
   remainder
```

The second example uses the "++" operator to concatenate the first of the lists with the result of evaluating the rest of a list of lists:

```
concatena::[[a]]->[a]
concatenate [] = []
concatenate (cab:rest) = cab++ concatenate rest
```

Exercises

Generate each of the following lists using extended list notation, with the list `[1..10]` as the generator. That is, each solution must have the following form, where the whites and only the whites must be completed:

```
[ ____________________ | x <- [1 .. 10] ____________
```

More explicitly, the response must use the x<- [1.. 10] generator, .and must not add any function calls to this definition: for example, do not use `reverse [x | x <- [1 .. 10]]` to create the list `[10,9,8,7,6,5,4,3,2,1]` Similarly, modifications of the type `[x|x <- [10,9..1]] and [x|x <- reverse[1 .. 10]]` are also prohibited.

1. `[11,12,13,14,15,16,17,18,19,20]`
2. `[[2],[4],[6],[8],[10]]`
3. `[[10],[9],[8],[7],[6],[5],[4],[3],[2],[1]]`
4. `[True,False,True,False,True, False,True,False,True,False]`
5. `[(3,True),(6,True),(9,True),(12,False),(15,False), (18,False)]`
6. `[(5,False),(10,True),(15,False),(40,False)]`
7. `[(11,12),(13,14),(15,16),(17,18),(19,20)]`

8. `[[5,6,7],[5,6,7,8,9],[5,6,7,8,9,10, 11],[5,6,7,8,9,10,11,12,13]]`
9. `[21,16,11,6,1]`
10. `[[4],[6,4],[8,6,4],[10,8,6,4],[12,10,8,6,4]]`

Patterns, tuples, recursion, and extended list notation in Haskell

In this session we will perform exercises using **patterns, tuples, recursion and extended notation of lists**.

1. Implement the function `dividers` that receives an entire argument and returns the list of its divisors.

```
Main> dividers 9
[1,3,9]
```

2. Using the above function, program the prime function to return true in case its only integer argument is a prime number. We will not consider the number 1 as prime.

3. Create an expression with which the primes between 1 and 100 are obtained. Use extended list notation for this exercise.

4. Find out how `map` and functions work and `filter` implement them using extended list notation. Calling the new functions `maps` and `filters`.

5. Schedule an evaluations function with the following header evaluations

```
::[a]->[(a->b)]->[[b]]
```

The resulting list list contains lists with the results of applying the functions of the second list to each of the values in the first list. For example:

```
Main> Reviews [1,2,3] [double, triple]
[[2,3],[4,6],[6,9]]6
```

. Use the above function to evaluate whether the following values `[0,(3.14/2),((-3.14)/2), 3.14,(-3.14)]` satisfy that the sine is greater than 0, the cosine is 0, and the tangent is 0. Compose the list of functions using the composition operator ".". The result for this example will be:

```
[[False,False,True],[True,False,False],[False,False,False],[True,False,Fals
e],[False,False,False]]
```

7. Implement a function that returns the prime factor decomposition of an integer. The function will return a list of tuples such that the first component will be the prime factor and the second will be the number of times that prime factor divides the original argument.

Example:

```
Main> descomposicion 60
[(2,2),(3,1),(5,1)]
```

8. Find out what the `takeWhile` and functions return and `dropWhile` implement your own version. Call the new functions `takeWhile` and `deleteWhile`.

```
Main> tomarMientras (<5) [1..10]
[1,2,3,4]

Main> eliminarMientras (<5) [1..10]
[5,6,7,8,9,10]
```

9. Program the function `quita_blancos` that removes the initial blanks from a string of characters.

```
Main> removeWhites "bcd fgh"
"bcd fgh"
```

10. Review the exercises performed and try to locate the points where some of Haskell's most important features were used: *currification and higher-order functions.*

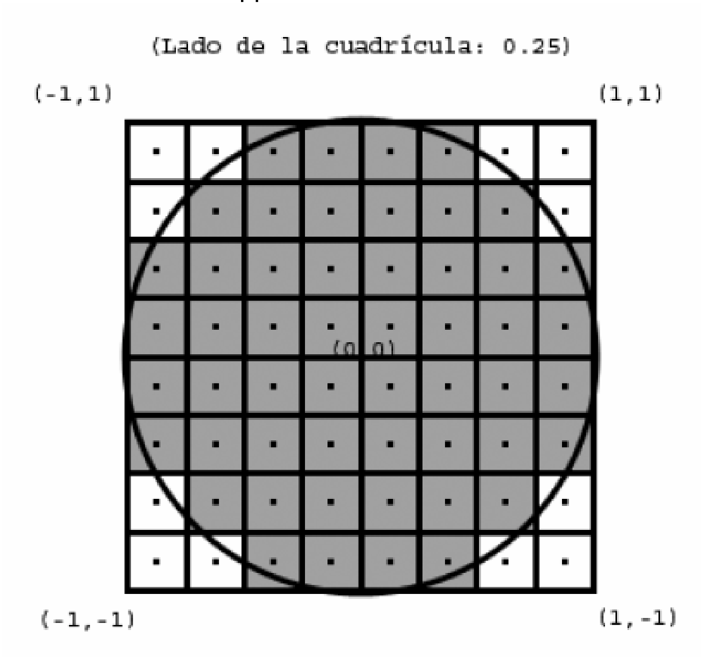
Haskell exercises from previous exams

In this last session of Haskell we will perform some exercises that appeared in previous exams.

1. Implement a function that approximates the computation of the integral of a function on the interval `[a, b]` and given a precision factor *t*.

```
integral function a b t
```

2. (February 2006) Implement a function that approximates the value of *pi*. To obtain the approximate value of *pi* we will use a square of side 2. In the center of the square we will fix in center of reference (0,0). We will make a grid inside the square of side *t*. Finally, we will count how many centers of the squares of the grid are inside the circle. This will give us an approximate value of *pi. The lower t*, the more accurate the approximation is.



Circle area:

$$A = \pi \cdot r^2$$

Distance between points (x1,y1) and (x2,y2):

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3. (November 2006) Implement in Haskell a function that calculates the hash key of a string. The hash function used is:

f ([a1,a2,…,an]) = a1*pn+a2*pn-1+…+p1*an.

Where the *ai* are the characters included in the list and *pi* are the i-esimos prime numbers (p1=2, p2=3, p3=5, p4=7 ...).

4. (September 2007) Assuming sets are implemented in Haskell as LISTS WITHOUT REPETITION, implement the functions:

`belongs to the set` item: Returns True if an item belongs to that set, otherwise False.

`subset set1 set2`: Returns True if the first argument is a subset of the second.

`equal set1 set2`—Returns True if two sets are equal.

`union set1 set2`—Returns a list that is the union of the two sets (NO REPETITIONS).

Internship with Prolog

Introduction to the SWI-Prolog Environment

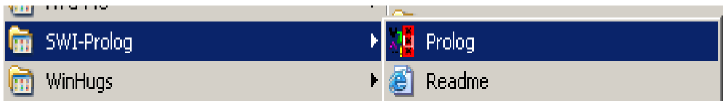
In this first practice we will familiarize ourselves with the SWI-Prolog programming environment.

Sección 1. Starting

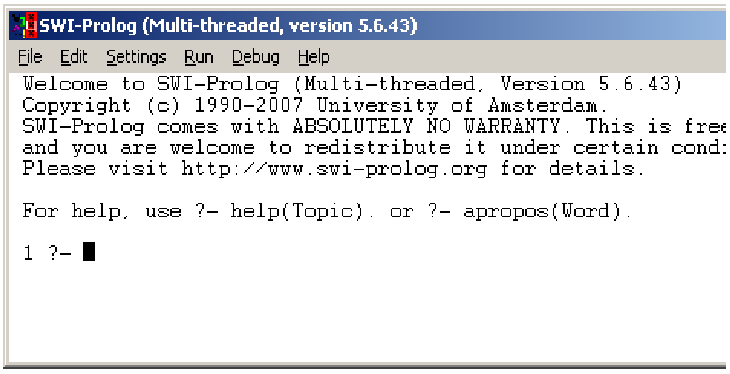
- Select the boot of the Windows XP operating system.

Sección 4. Run SWI-Prolog

- In the Windows "Start" menu select the SWI-Prolog program



- A window like this will appear:


SWI-Prolog (Multi-threaded, version 5.6.43)
File Edit Settings Run Debug Help

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.6.43)
Copyright (c) 1990-2007 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
and you are welcome to redistribute it under certain cond:
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- 
```

- From this window we will load the programs and make the queries.

Sección 5. Write

a program To write a program we will use a text editor that does not enter special characters. The Windows notepad will serve us. SWI-Prolog has this editor configured by default.

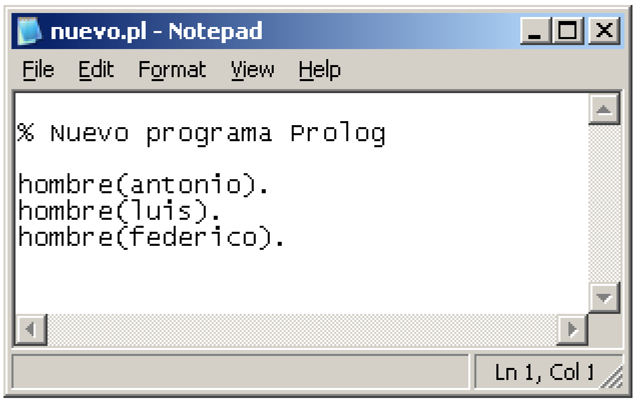
The easiest way to write edit a new program is to type edit(

```
file('<nombre_fichero.pl>')).
```

at the SWI-Prolog:1? prompt

```
- edit(file('new.pl')).
```

This command will edit the file '`nuevo.pl`'. We will use the ".pl" extension for Prolog programs. The file will be saved by default in the "My Documents\Prolog" directory. We will write the following considering that predicates and constants begin with lowercase:



```
% Nuevo programa Prolog

hombre(antonio).
hombre(luis).
hombre(federico).
```

We will save the contents of the file and close the window. While editing the file we cannot access the SWI-Prolog window.

We will load the file with the command `consult('<nombre_fichero>').`

```
? - consult('new.pl').
```

We can check if the file was indeed loaded using the `listing command`, which will show the loaded program.

```
? - listing.

hombre(antonio).
Man (Luis).
Man (Federico).

Yes?-
```

From this moment we can start making inquiries

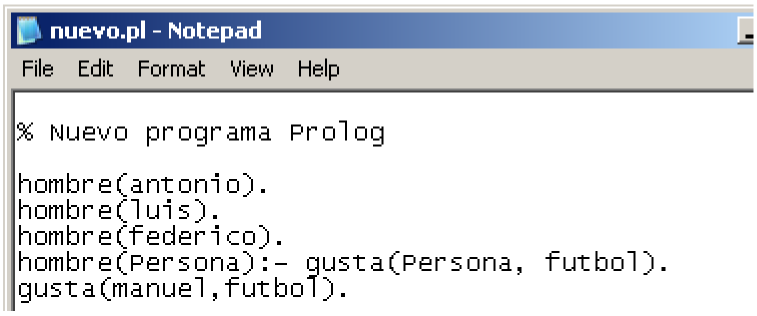
```
? - Man (Antonio).

Yes

? - Man (Federico).

Yes

? - Man (Manuel).

By
```

the *hypothesis of the closed world*, the undefined is false.

We will modify the previous program to include the following lines:

```
man(person):- like(Person,football).gusta(manuel,football).
```

We will take into account that `Persona` is a variable and we will write it in capital letters.



```
% Nuevo programa Prolog

hombre(antonio).
hombre(luis).
hombre(federico).
hombre(Persona):- gusta(Persona, futbol).
gusta(manuel,futbol).
```

After modifying a file, we update the changes by executing the "make" command.

```
?- make.
```

We will now carry out the previous query: Manuel man?

```
? man (Manuel).

Yes
```

Sección 5. Operators

The most commonly used operators in Prolog are `"="`, `"=="`, `"is"` and `"=:="`.

"=" we will read unification.

Two terms unify:

1) If they have no variables, they unify if they are identical (equal character to character).
2) If they have variables, they will unify if it is possible to find a substitution of the variables so that they become identical.

The unification operator does not evaluate arithmetic operations. Before unifying a variable to a value we will say that the variable is "free" or "uninstantiated". After unifying a value to a variable, the variable is no longer free and we will say that the variable is instantiated. A variable, once instantiated, does not change value.

```
 ?- X = 1+1.

 X = 1+1.
Yes
```

`"=="` identity

```
?- 2 == 1+1.

No.
```

`"is"` evaluates arithmetic expressions. Evaluate on the right and unify with the left.

```
?- X is 1+1.

X = 2?-
```

`"=:="` evaluates arithmetic expressions and compares. It evaluates arithmetic expressions on the right and left and **it is true if the result of the evaluations is the same value.**

```
?- 2 =:= 1+1.

Yes
```

Sección 6. Scope of a variable

In most programming languages, if the variable is defined within a function, the variable can be referenced from any point in the function. We say that the scope of the variable is the function in which it is defined.

In Prolog, the scope of a variable is restricted to one clause. Outside of a clause the variable cannot be referenced. This is one of Prolog's most important differences from most programming languages. Thus, in the Prolog program:

```
nombre_numero(X, 0):- X = zero.
nombre_numero(X, 1):- X = one.
```

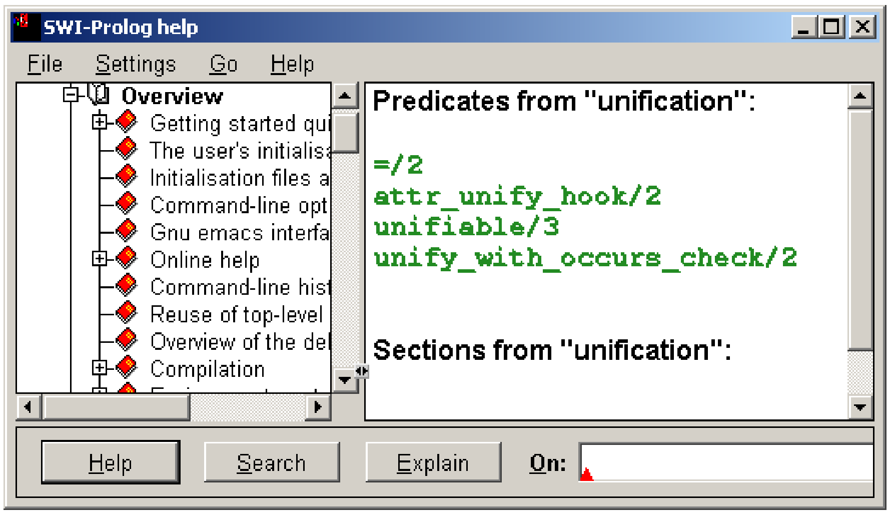
the variable X that appears in the consequent of the first clause is the same as the one that appears in the antecedent of that same clause but is different from the variable X that appears in the consequent and antecedent of the second clause.

Sección 7. Predicates of help

We can consult information about any predicate with the command "apropos".

```
apropos(unification).
```

The following window will appear:



Predicates from "unification":

=/2
attr_unify_hook/2
unifiable/3
unify_with_occurs_check/2

Sections from "unification":

We will invoke this same window with the help predicate. Predicates in Prolog are named as follows `nombre_predicado/arity`. Arity is the number of arguments the predicate has. For example, the predicate unification =/2, has arity 2.

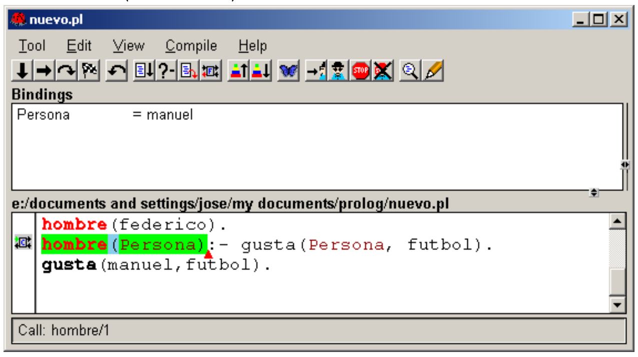
Sección 8. Debugger

We will enable the graphical debugger with the predicate "`guitracer`".

```
?- guitracer.
% The graphical front-end will be used for subsequent tracing

Yes
```

To debug a Prolog program we will use the "trace" command. The next query we make after the "trace" predicate will be executed step by step. If the graphical debugger is enabled, the following window will be displayed. To advance the execution we will press the space bar or use the forward and backward icons.

```
?-Trace. Yes

? - man(manuel).
```



```
Persona        = manuel
```

e:/documents and settings/jose/my documents/prolog/nuevo.pl

```
hombre(federico).
hombre(Persona):- gusta(Persona, futbol).
gusta(manuel,futbol).
```

Call: hombre/1

Exercises

1. Unification.

Check if they unify the following terms:

```
(a) X=1.
(b) X = woman(maria).
(c) X = 2 + 3.
(d) X=1, X=2, X=Y.
(e) X=2+X.
(f) X=1, Y=1, X=Y.
(g) X=2, Y=1+1, X = Y.
(h) X=3+4, Y= 7, X = Y.
(i) X = 2*6, X=2*Y.
(j) 5 = 2 + 3.
(k) 2 + 3 = 2 + Y.
(l) f(X, b) = f(a, Y).
(m) 2*X = Y*(3+Y).
(n) (1+2) = 1+2.
(o) 1+1 = +(1.1).
(p) (1+1)+1 = 1+(1+1).
```

2. Arithmetic and Unification exercises.

Check if they unify the following terms:

```
(a) X is 2 + 3, X = 5.
(b) X is 2 + 3, X = 2 + 3.
(c) 2 + 3 is X.
(d) X is 5, 6 is X + 1.
(e) X = 5, 6 = X + 1.
(f) 6 is X + 1, X = 5.
(g) Y = 2, 2*X is Y*(Y+3).
(h) Y = 2, Z is Y*(Y+3).
```

Simple predicates in Prolog

In this session we will implement some simple predicates using recursion in most cases.

Exercises

1. Implement `natural(X)` which will be true if `X` is a natural number.

```
?- nat(6).
yes

?- nat(-13).
No
```

2. Implement the Prolog factorial predicate `(Number, Result)` which will be true if `Result` unifies with the factorial of "`Number`".

3. Implement the `predicate fib(N, F)`. Which will be true when "F unifies with the Nth Fibonacci number". These numbers are:
fib(1)=1,
fib(2)=1,
fib(3)=2,
fib(4)=3,
fib(5)=5,
.....,
fib(n)=fib(n-1) + fib(n-2),

4. Given a series of dishes, classified into first course, second course and dessert with their respective associated price, prepare a predicate that finds a complete menu for a cost less than N. Example:

```
menu(Primer_plato, Segundo_plato, Dessert, Price),
Price < 50.

Primer_plato=soup
Segundo_plato=ham
Dessert=ice cream
```

5. Implement the predicate `sum (X,Y,Z)` that represents the sum of two natural numbers using the representation of the Italian mathematician **Giuseppe Peano** (1858–1932) that is based on the use of the symbol 0 and the predicate `s(X)` that represents the next of `X`.

0=0
1 = s(0)
2 =s(s(0))
3=s(s(s(0)))
……

```
amount(s(0),s(0),Z).
Z=s(s(0))
Yes
```

6. Using the representation of the previous exercise, implement the predicates subtract (X, Y, Z) and `product (X, Y, Z).`

7. Implement exercise 3 so that the `predicate fib(N,F)` is reversible, that is, it is possible to make the call leaving the first argument free and instantiating the second. This exercise is somewhat more complicated than the previous ones and to solve it it will be necessary to use the arithmetic of G. Peano.

Example:

```
fib(N,s(s(0)))
N=s(s(s(0)))
Yes
```

Predicates on lists in Prolog

In this session we will implement some predicates on lists.

Exercises

1. Try to predict the unification that results in the following examples and then confirm by running each one:

```
?- [X| Y] = [a, b, c, d].
?- [X, Y| Z] = [a, b, c].
?- [X, Y| Z] = [a, b, c, d].
?- [X, Y, Z| A] = [a, b, c].
?- [X, Y, Z| A] = [a, b].
?- [X, Y, a] = [Z, b, Z].
?- [X, Y| Z] = [a, W].
```

2. Define in Prolog the following recursive predicates to handle Lists:

(a) Given two items, create a list of those two items.

(b) Insert an item in a list:
  1) At the beginning of the list.
  2) At the end of the list.
  3) At position N of a list.

c) Concatenate two lists and use that predicate to implement the Predicates:
  1) `Prefix`
  2) `suffix`
  3) `Sublist`

d) Invert a list.

and) Delete an item in a list:
  1) Deleting only one occurrence of the item.
  2) Deleting all occurrences of that element.

f) Change a certain element for another:

  1) Change only one occurrence of the element.
  2) Change all occurrences of that element.

g) Given a list of length n, generate another list of length 2n that is a palindrome.

h) Move a position to the right all the elements of a list. For example, move from the [x1,x2,...,xn] list to the `[xn,x1,...,xn-1]` list.

i) Move a position to the left all the elements of a list. For example, move from the [x,x2,...,xn] list to the `[x2,...,xn,x1]` list.

3. Define a `predicate divide(+N, +LOrig, -May, -Men)` that holds if the `May` list contains the elements greater than an `N number in the LOrig list` number in the `LOrig` and `Men` contains the minor elements of an `N` list.

```
?-divide(4,[3,1,2,5,0], Major, Minor).
Seniors = [5]
Minors = [1, 2, 3, 0]
```

4. Define a predicate , mezclar_ord(+L1, +L2, -Resul) so that, being `L2` L1 and L2 two ordered lists, it is true that `the Resul` list contains the mixture of the elements of the two lists `L1 and` and is also an ordered list.

```
?-mezclar_ord([1,2,7],[0, 3,5], Rasul).
Rasul = [0, 1, 2, 3, 5, 7]
```

5. Defining a sort predicate `(List, R)` that is fulfilled if the R list contains the ordered List items.

```
?-orders([3,1,2],V).
V = [1, 2, 3]
```

a) As permutations of a list, until it finds the ordered.

(b) As separation of lists, sorting and mixing.

6. Write a predicate `prod(+L, ? P)` which means "`P` is the product of the items in the list of integers `L`". You must be able to generate the `P.` P and also check a given

7. Write a `pScalar predicate(+L1, +L2, -P)` which means "`P` is the scalar product of the two vectors `L1` and `L2`". The two vectors are given by the two lists of integers `L1` and `L2`. The predicate must fail if the two vectors have a different length.

8. `dividers(+N, ? L)`. Given a natural `N`, `L` is the list of divisors of `N` in increasing order. For example, if `N is 24, L` will be `[1,2,3,4,6,8,24]`. It must answer `YES` if a given L is the list of the divisors of `N` given `a given N and NOT` otherwise; and it must be able to generate `L` for a . With `divisors(6,[2,3,6,1])` you have to answer `NO` (the list is not ordered!).

9. `swap(+L, -P)`. "List `P` contains a permutation of the items in list `L`." The L list will initially be instantiated, and P will not. Example: , L = [1,2,3] would give `P = [1,2,3], P = [1,3,2], P = [2,1,3]`etc. In the event that the list to be exchanged has repeated elements, repeated permutations should be allowed.

10. A set can be modeled by a list of elements without repetitions. Adopting this representation, implement the following operations on sets in Prolog language.

a) Determine if an element belongs to a set.
(b) Incorporate an element into an assembly.
c) Joining of the assemblies.
d) Find the intersection of two sets.
e) Calculate the difference between two sets.
f) Given a list of elements with repetitions, construct a
An assembly that contains all the items in that list.

11. A MULTI-set can be modeled by a list of elements (element, multiplicity). Adopting this representation, implement the following operations on sets in Prolog language.

a) Determine if an element belongs to a set.
b) Calculate the multiplicity of an element.

Trees in Prolog

Binary trees

predicate `1. Implement the cuenta_nodos/2` that counts the number of nodes that a binary tree has.

predicate `2. Implement the cuenta_internos/2` that counts the number of INTERNAL nodes (having at least one child) that have a binary tree.

predicate `3. Implement the cuenta_hojas/2` that counts the number of LEAF nodes (which have no children) that a tree has.

predicate `4. Implement the suma_nodos/2` that summarizes the contents of all nodes in a tree.

5. Type the predicate `member/2` that indicates whether an element `X` belongs to a tree.

6. Implement the `equal/2`, `symmetric/2 and isomorph/2 predicates` that will be true when two trees meet these properties.

predicate `7. Implement the depth/2` which will be true when the second argument unifies with the depth of the tree.

8. Implement the balanced predicate `/1` which will be true when the tree is balanced.

9. Implement the following predicates that transform the paths of a tree into a list.
`inorden/2`
`preorden/2`
`postorden/2`
`width/2` (left to right)

Generic trees

Repeat the previous exercises, except the inorder, preorder and postorder of exercise 9, considering generic trees instead of binaries.

Binary search trees,

Implement the following predicates in Prolog:

`creatingEmptyTree(X)`—returns `to X.` an empty tree

`insert(`E, A, NA): inserts element `E` into tree `A`, returning the new tree into `NA` (assume that if the element to be inserted already belongs to the tree, then the same tree is returned).

`height(A, X)`—returns `X` the height of the A tree `in` .

`balanced(A)`—Determines whether the A-tree is balanced or not.

`routes (A, Pre, Post, In)`: from tree `A` returns in `Pre, Post and In` their routes in preorder, postorder and inorder respectively.

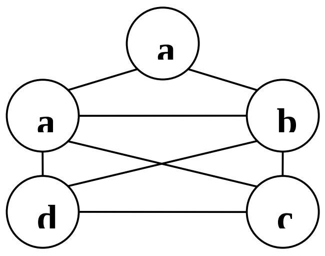
Implement a predicate that constructs a binary search tree from a list of integers.

Example:

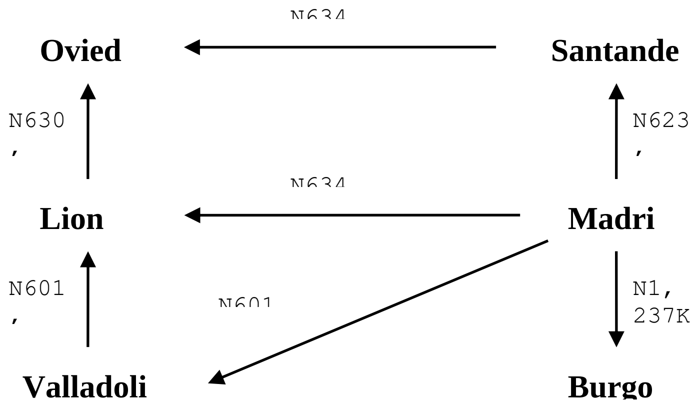
```
?- build([3,2,5,7,1],T).
T = t(3,t(2,t(1,nil,nil),t(5,nil,t(7,nil,nil)))
```

Graphs in Prolog

1. Write a Prolog program that paints an envelope like the one in the figure without lifting the pencil from the paper, that is, it goes through the graph passing exactly once through each arc.



2. Represent the graph of the figure in Prolog:



Write a Prolog program that finds all possible paths between Madrid and Oviedo, building a list of the cities through which it passes.

Add the necessary elements for you to calculate:

- The list of roads through which each road passes.
- The distance traveled by each of them.

What would happen if the proposed graph were not directed and acyclic like the one proposed? To analyze this possible circumstance, introduce two more arcs in the graph:

- León-Palencia (N 610,130 Km.)
- Palencia-Valladolid (N 620, 47 Km.)

and check the behavior of the program.

Propose a solution to avoid loops and implement it.

3. A graph can be expressed as `G = (V, A)`, where `V` is the set (list without repetition) of vertices and `A` the set of arcs `(ai, aj)`, labeled by the distance separating vertex `ai` from vertex `aj`.

Create a program in Prolog with the predicates:

- `path(G,N1,N2,Cam)`, which finds the path between two nodes and returns a list of the nodes through which it passes. Through *backtraking* you must find all possible paths.
- `distance(G,N1,N2,D)`, which calculates the distance between any pair of vertices. Deploy two versions: one that supports cycles and one that does not.
- `cycles (G,Cycle)` , which find the cycles within a graph. This consists of finding a path that goes from a node to itself.

- `related(G)`, which verifies that a graph is fully connected. A graph is said to be fully connected when there are paths between all its nodes.
- `degree (G,N,Gr)` that tells us the degree of a node within a graph. The degree of a node is the number of nodes it connects to.
- `listdegrees(G,L)` that generates a list with all nodes and their degrees.
- `pintargrafo(G,LC,LNod) that` paints the G graph `with a list of` list with `LC colors and` returns an `LNod` each of the nodes and their color, so that two adjacent nodes cannot have the same color.