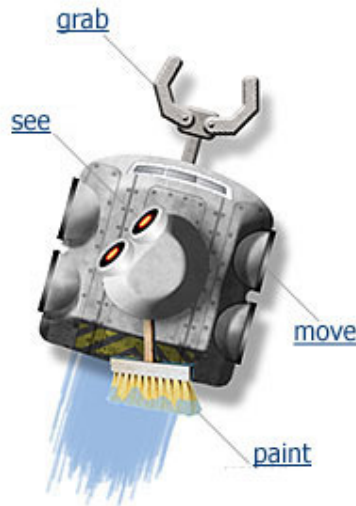






# Basic instructions




In each domain there are a fixed number of basic instructions you may use to write a program. You're only allowed to use this instructions directly when writing the script. For our robot the following basic instructions exist:





## Move

	adelante(n)	Move n steps forward
	atras(n)	Move n steps backward
	izquierda()	Turn left over 90 degrees
	derecha()	Turn right over 90 degrees
	norte(n)	Turn to head north and move n steps forward
	sur(n)	Turn to head south and move n steps forward
	este(n)	Turn to head east and move n steps forward
	oeste(n)	Turn to head west and move n steps forward

## Paint

	pintarBlanco()	Put the brush with white paint to the ground.
	pintarNegro()	Put the brush with black paint to the ground.
	detenerPintar()	Stop painting, hide the brush

## Grab

	tomar()	Get the beacon in front of the robot
	poner()	Put a beacon in front of the robot

## Flip coin

`lanzarMoneda()` Flip a coin to make a random choice.  
`lanzarMoneda()` will either be true or false.

[⬆ top](#)

## See

*Left*

*Front*

*Right*

`izquierdaEsObstaculo()` `frenteEsObstaculo()` `derechaEsObstaculo()`

`izquierdaEsClaro()` `frenteEsClaro()` `derechaEsClaro()`

`izquierdaEsBaliza()` `frenteEsBaliza()` `derechaEsBaliza()`

`izquierdaEsBlanco()` `frenteEsBlanco()` `derechaEsBlanco()`

`izquierdaEsNegro()` `frenteEsNegro()` `derechaEsNegro()`

## Comments

**# free text that will not be evaluated**

All text that appears after a hash, '#', will not be interpreted as instructions. The robot will proceed to read the next line in the script. Use this possibility to annotate parts of the script, just for yourself, as documentation about how these parts work

## Loops

**repetir(*n*){...instructions...}**

repeats the instructions between curly brackets exactly *n* times.

*Example:*

```
# a square of 2x2
repetir(4)
{
    adelante(2)
    derecha()
}
```

**repetir(){...instructions...}**

just keeps repeating the instructions between curly brackets for ever.

*Example:*

```
# just goes adelante
# (but eventually will stay hitting a wall)
repetir()
{
    adelante()
}
```

**repetirMientras(*condition*){...instructions...}**

repeats the instructions between curly brackets as long as the condition holds. This condition must be a perception/seeing instruction (for example frenteEsClaro)

*Example:*

```
# keeps going adelante,
# but stops when it can't go any further
repetirMientras(frenteEsObstaculo)
{
    adelante(1)
}
```

**truncar**

allows you to jump out of the loop (e.g. a repeat() section) so it stops

performing the instructions between curly brackets. The robot will resume performing the instructions left after the closing curly bracket of the loop.

*Example:*

```
# keep going adelante, until yu can't go any further
repetir()
{
    si(frenteEsObstaculo()) {
        truncar
    }
    otro
    {
        adelante(1)
    }
}
```

[^ top](#)

## If-structures

**si(condition){...instructions...}**

will perform the the instructions between curly brackets, only if the condition holds. Else the robot immediatly steps to the instructions written after the closing curly bracket. The condtion must be a perception/seeing instruction (for example: frenteEsClaro())

*Example:*

```
# si you see white paint on your izquierda, make it black
si(izquierdaEsBlanco())
{
    izquierda()
    adelante(1)
    pintarNegro()
    detenerPintar()
    atras(1)
    derecha()
}
```

**si(condition){...instructions...}otro{...instructions...}**

will perform the the instructions between the first pair of curly brackets, only if the condition holds. Then it will not perform the instructions of the else block (second pair of instructions). When the condition does not hold, the robot will only perform the instructions in between the second pair of curly brackets. After it performed one of the instruction blocks, it will read the instructions after the last curly bracket. The condtion must be a perception/seeing instruction (for example: frenteEsClaro())

*Example:*

```
# si you see white paint on your izquierda, make it black
# otro drive a few steps adelante
si(izquierdaEsBlanco())
{
```

```

        izquierda()
        adelante(1)
        pintarNegro()
        detenerPintar()
        atras(1)
        derecha()
    }
    otro
    {
        adelante(3)
    }

```

[^ top](#)

## Logical expressions

The *conditions* of *if*- and *repeatWhile*-structures is a so-called logical expression. Such an expression will result in the value verdadero or falso, which is then used to decide to step to the appropriate part of the code to resume execution.

A logical expression can be a [perception instruction](#), e.g.: verdadero(). Basic instructions may also be composed with the boolean operators ~, &, l.

*Example:*

```

si(izquierdaEsBlanco())
{
    izquierda()
    adelante(1)
}

```

The condition can however also be refined to more indicate more precisely when the corresponding instructions should be executed by using (a combination of) the following operators.

Operation	Alternative notation	Number of arguments	Explanation
no	~	1	<p>Negates the value of the argument :</p> <p><i>Truth table :</i></p> <p>~ verdadero = falso ~ falso = verdadero</p> <p><i>Example:</i></p> <p>~ frenteEsClaro()</p>
y	&	2	<p>Only true when both arguments are true.</p> <p><i>Truth table:</i></p> <p>verdadero &amp; verdadero =</p>

			verdadero verdadero & falso = falso falso & verdadero = falso falso & falso = falso  <i>Example:</i> frenteEsClaro() & derechaEsBlanco()
o		2	True when at least one of the arguments is true.  <i>Truth table:</i> verdadero   verdadero = verdadero verdadero   falso = falso falso   verdadero = falso falso   falso = falso  <i>Example:</i> frenteEsClaro()   derechaEsBlanco()

The values verdadero and falso can also be applied directly as if it was a perception instruction.

The order of the operators is of importance (just like multiplying and adding numbers). The operation ~ binds strongest, followed by &, followed by |. Brackets can be used to influence the order of evaluation.

*Examples:*

```

repetirMientras(no frenteEsClaro() y (izquierdaEsBlanco()
o derechaEsBlanco())){
    adelante(1)
}

si(lanzarMoneda() y no derechaEsBlanco())
{
    derecha()
    atras(1)
}

si(verdadero y falso){
    # this instruction is never executed
    adelante(1)
}

```

[^ top](#)

**procedimiento *name(par1, par2, ... , parN){...instructions...}***  
defines a new procedure with the name you want. The procedure can

**Procedures** have zero or more parameters, which you may also give useful names. Here they are called `par1`, `par2`, . . . , `parN`. These are the variables you can use in the instruction between curly brackets. The code in a procedure will not be performed automatically, you have to write a 'procedure call' every time you want to perform the instructions in the definition (See next instruction).

Tip: create a new procedure when you use a sequence of instructions more than once.

*Example:*

```
# define how to draw a rectangle
procedimiento rectangle(width, height)
{
    pintarBlanco()
    repetir(2)
    {
        adelante(height)
        derecha()
        adelante(width)
        derecha()
    }
    detenerPintar()
}
```

*name(arg1, arg2, . . . , argN)*

is the call to the procedure with the corresponding name and the same amount parameters as you have arguments. The argument, here called `arg1`, `arg2`, . . . , `argN`, are the particular values that will be used in the procedure definition.

*Example:*

```
# these instructions will be performed
adelante(1)
rectangle(3,2) # a call to the 'rectangle' procedimiento
adelante(3)
rectangle(1,4) # another call with other arguments

# this is the definition of 'rectangle'
procedimiento rectangle(width, height)
{
    pintarBlanco()
    repetir(2)
    {
        adelante(height)
        derecha()
        adelante(width)
        derecha()
    }
    detenerPintar()
}
```

[^ top](#)

## End

### fin

will cause the entire program to stop when this instruction is performed.

*Example:*

```
# stop after 5 steps, o earlier when you encounter a
beacon on the derecha
repetir(5)
{
    adelante(1)
    si(derechaEsBaliza())
    {
        fin # stops execution of the program
    }
}
# normal fin of the program
```

# Programming structures

Here you'll find the grammatical structures that are allowed to define the behaviour of the robot.

## Example scripts

In the next example programs you'll see how basic instructions and programming structures can lead to interesting behaviour of the robot. Try to understand the script and why it works. Even better is to try them yourself by copying the code and pasting it into the RoboMind script panel. Make sure the right map is used before running them.

- Example 1: [Write your name](#)
- Example 2: [Find the spot](#)
- Example 3: [Line follower](#)
- Example 4: [Maze runner](#)

### Example 1 : Write your name

Because the robot is able to paint, you're able to create simple drawing programs. Using `pintarBlanco()` and `detenerPintar()` you can command the robot to put its brush on the ground or not. When you let the robot move, it will leave a line on the ground. In this manner you can write characters like 'A'.

In the map `openArea.map` you'll have enough space to create a nice piece of art. Try using the remote control (Run > Remote control) to let the robot paint.

See also: basic instructions: [paint](#), [move](#)

```
#character 'A'
```





```
pintarBlanco()
```

```
adelante(2)
derecha()
adelante(1)
derecha()
adelante(2)
atras(1)
derecha()
adelante(1)
```

```
detenerPintar()
```

[^ top](#)

## Example 2 : Find the white spot

In the map *findSpot1.map* which you can find in Rhomboid's default map folder, you'll find somewhere on the left a corner containing a white spot. Assume you don't know in advance at what distance this corner is located. How is the robot able to find it if it starts somewhere along the wall? Of course you can count the number of steps the robot has to move yourself, but there are better ways.

Let the robot make one step forward at the time, and let him check if it already sees the spot on its left. If it sees the spot, it has to stand on it and then it is finished. Else it should make another step forward and check again if it already is near the spot. This can be repeated in a loop.



See also: basic instructions: [move](#), [see](#), programming structures: [loops](#), [if-structures](#), [end](#)

```
repetir(){
    si(izquierdaEsBlanco()){
        # There's a white spot on your izquierda
        izquierda()
        adelante(1)
        fin
    }
    otro{
        # There's no white spot yet
        adelante(1)
    }
}
```

Another approach that shows the same results is the following script. Here the robot repeats moving forward until it doesn't see the wall anymore (so it must be the corner we are looking for). Then the robot turns and moves forward to land on the spot.

```
repetirMientras(izquierdaEsObstaculo()){
    adelante(1)
}
```

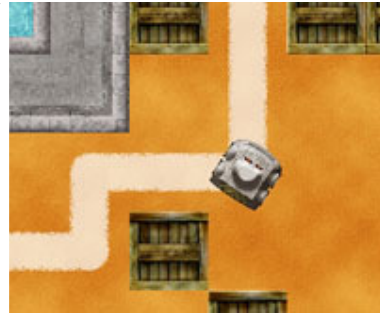
```
izquierda()  
adelante(1)
```

[↩ top](#)

### Example 3 : Line follower

In default.map you find a white line in the east. This line forms a track through the environment. How can you make the robot follow this track?

The solution is somewhat similar to the previous example. After the robot is brought to the beginning of the track, it determines what to do step by step.



See also: basic instructions: [move](#), [see](#),  
programmeerstructuren: [loops](#), [procedures](#), [if-structures](#),  
[end](#)

```
# Go to the start of the line  
derecha()  
adelante(8)  
  
# Keep on track step by step  
repetir()  
{  
    si(frenteEsBlanco()){  
        adelante(1)  
    }  
    otro si(derechaEsBlanco()){  
        derecha()  
    }  
    otro si(izquierdaEsBlanco()){  
        izquierda()  
    }  
    otro si(frenteEsObstaculo()){  
        fin  
    }  
}
```

Another approach is using so-called recursive procedures. In the next script you see that a procedure follow() is defined. In this procedure, after the correct step is performed, you see that follow() is used in its own definition. What you get is a sequence of follow(...follow(...follow(...)...)) that perform the correct action each time step to fulfil the task. This cyclic definition is called recursion, and it is likely that you'll find it somewhat strange the first time you see it. No worries, play with it and becomes more clear.

```
# Go to the start of the line  
derecha()  
adelante(8)
```

```

# Start tracking
follow()

# Recursive definition for tracking
procedimiento follow(){
    si(frenteEsBlanco()){
        adelante(1)
        follow()
    }
    otro si(derechaEsBlanco()){
        derecha()
        follow()
    }
    otro si(izquierdaEsBlanco()){
        izquierda()
        follow()
    }
    otro si(frenteEsObstaculo()){
        # Finish the current procedimiento call
        # (because there is nothing to after
        # this regresar, all calls will be finished
        # y the program stops)
        regresar
    }
}

```

[^ top](#)

## Example 4 : Maze runner

How to escape a maze in general? It appears that this hard question has a simple solution. By always following the wall on the right side (or always following the wall on the left side) you will find the exit for sure.

The next script makes the robot find the beacon in for example *maze1.map*.

See also: basic instructions: [move](#), [see](#), programming structures: [loops](#), [if-structures](#), [end](#)

```

repetir(){
    si(derechaEsObstaculo()){
        si(frenteEsClaro()){
            adelante(1)
        }
        otro{
            izquierda()
        }
    }
    otro{
        derecha()
        adelante(1)
    }

    si(frenteEsBaliza()){
        tomar()
        fin
    }
}

```

