

DBN Vocabulary

by Zach Lieberman and Rusty Soo-Tho
edited by Megan Galbraith and Martin Gomez

paper

variable

smaller?

pen

set

notsmaller?

line

naming

command

calculations

copy

nesting

comments

repeat

pause

parameters

nested repeat

mouse

coordinates

same?

key

notsame?

time

summary

DBN is actually multi-lingual. Here's a sampling.

english (en)	español (es)	français (fr)	japanese (jp)	filipino (ph)
paper	papel	papier	かみ	papel
pen	stilo	plume	ペン	panulat
line	línea	ligne	せん	guhit
repeat	repita	répeter	くりかえし	ulitin
forever	siempre	toujours	ずっと	magpakailanman
set	ponga	mettre	おく	ilagay
command	instrucción	fonction	コマンド	atas
number	número	numéro	ばんごう	bilang
field	área	région	りょういき	linang
refresh	refrese	àneuf	リフレッシュ	sariwain
mouse	ratón	souris	マウス	daga
keyboard	teclado	clef	キーボード	tipaan
net	internet	réseau	ネット	lambat
time	hora	heure	じかん	panahon
same?	¿igual?	pareil?	おなじ?	tulad?
notsame?	¿noigual?	paspareil?	おなじでない?	hinditulad?
smaller?	¿menos?	pluspetit?	すくない?	masmaliit?
notsmaller?	¿nomenos?	plusgrand?	すくなくない?	masmalaki?
fast	rápido	rapide	はやい	dali
load	carga	charge	ふか	ikarga

Send questions, comments or request instructions on how to do multi-lingual DBN to martin (at) decode.ateneo.edu

DBN with Filipino will be released very soon.

DBN Vocabulary

paper

the background color for the page. Default is white. Fills the page with color. Should come first, unless you want to clear the screen after you've drawn. Takes one parameter - a number between 0 and 100. 0 is white (think 0% black) and 100 is black (100% black). Some examples :

```
paper 0
```

```
paper 50
```

```
paper 100
```

DBN Vocabulary**pen**

used to set the pen color. Default is black. Takes one parameter - a number between 0 and 100. 0 is white (think 0% black) and 100 is black (100% black). When you use the line or dot (set) command, it renders in that color. The color can be changed, as well. For example:

```
pen 30
line 0 0 100 100

pen 60
line 0 100 100 0

pen 80
line 50 0 50 50
```

DBN Vocabulary**line**

takes 4 parameters x1, y1, x2, y2. These are the x and y coordinates of both ends of the line. The order that the ends are typed in doesn't matter. for example:

```
line 0 0 100 100
```

draws the same line as

```
line 100 100 0 0
```

and so on. There are some good examples of reasons why you wont see a line -- if the line is the same color as the paper, if the line is outside of the 0, 0 to 100, 100 boundaries, or if the paper command is used after the line command.

DBN Vocabulary

calculations

are a way of asking the program to do math for us. For example, if we said:

```
paper (50 - 10)
```

the computer would take the “(“ as a signal to “do something”. In this case, we are saying do subtraction. It’s important that parantheses are used, because the computer gets easily tripped up with:

```
paper 50 - 10
```

It sees only paper 50 and then is confused about “- 10”

it also is used to organize the order of operations. For example, specifying $((10 - 5) * 13)$ forces the interior “(“ parenthesis to be evaluated first. Order of operations is a concept we’ll get into a little later.

DBN Vocabulary**comments**

add text to programs in way where they are skipped over as code is evaluated. They are often used to provide descriptions of the code. As well, they are used to break up blocks of code and increase legability. Most languages have the ability to contain comments. Often, the way you specify comments are different for each language. Java, JavaScript and ActionScript use `//`. Java uses `/*` to open multi-line comments and `*/` to close. Perl uses `#`.

DBN Vocabulary**parameters**

These are numerical arguments passed to commands. For example, paper takes one parameter (the color). The order in which you pass the parameters is very important. For example, 10 paper wont work. Also paper10 will fail. DBN wants parameters to come after the command. A variable can be used as a parameter. We will cover this later.

Some parameter examples :

```
paper 10 // 1 parameter
pen 40 // 1 parameter
line 20 20 50 10 // 4 parameters
```

The separator in DBN is a space. Other languages have other ways of passing and seperating commands. JavaScript uses parentheses and commas, i.e. makeMove(3, 4).

DBN Vocabulary

coordinates

Different programming languages have different ways of referring to their coordinate system. The most important is the (0,0) point, or the origin. in DBN, the origin is located at the bottom left corner. The top right corner is 100,100. The box is 101 pixels by 101 pixels. In contrast, Java refers to the top left corner as the origin. The DBN interface has a little ruler attached for your reference.

DBN Vocabulary**variable**

A variable is basically a verbal place holder for numeric information. You chose a name for your variable (ex: myHeight) and assign to it a value that you want it to keep track of. In DBN, this would look like:

```
set myHeight 60
```

You can change the value of the variable later, for example as your height grows. Think of a variable as a container that you are asking to hold something for you, like your jar of blue gumballs (mmm... gumballs). You can change the value of that variable (hence the name “variable”) as easily as if you took another gumball out of the jar.

Why are variables significant?

- a. They allow us to create relationships between different parts of our program. For example, let’s say you wanted the color of a rectangle to change depending on it’s position on the screen. Those two values — position and color — have a relationship, and by setting a variable, your program can monitor that relationship for you.
- b. They allow us to abstract as we code. Instead of specifying a triangle at the specific points, we can program a general triangle which takes different values for its variables. Programmers are by nature lazy (see Programming Perl for an explanation of why that is) and by using variables, we can write more powerful,

abstracted code.

DBN Vocabulary**set**

This is the command that DBN requires before you assign a value to a variable. The syntax looks like this:

```
set myHeight 60
```

myHeight is the name of the variable. 60 is the value that you are putting in that variable.

The set command is also used for dots (individual pixels). The syntax in that situation looks like this:

```
set [50 50] 100
```

This puts a dot (which has the Pen color) in the middle of the page. [50 50] is the x and y position of the dot and 100 is the color.

DBN Vocabulary**naming**

Having a consistent naming scheme is very important, especially when you programs get larger and larger. There's really no reason to give them super technical names like `dt145` or `x4_id9` when `my_width` or `big_red_dot` will work just as well and will be easier to remember. One letter names, although used often in DBN, can be quite confusing at times. Your best bet is to use logical, concise names that will make your code legible for others and yourself.

There are some limitations to what you can name your variables. You can't use any words that are already used as commands (`set`, `paper`, `pen...`) and you can't start a variable name with a number. Other than that, you can use any style you want. For mutiple word variables, it's common to either separate the words with an `_underscore` (`big_red_dot`) or by capitalizing each word, starting with the second (`bigRedDot`).

I would recommend using camelCase as the using underscores to separate words is already deprecated.

DBN Vocabulary**copy**

makes one variable equal to another. Basically you are copying the contents of one into the other. Example:

```
Paper 0
Set G 25
Pen G
Set H G // we create a variable H which has the value of G...
Line H 0 H 100
```

Saying “Set H G” gives H the value of G (in this case 25). In this program, by linking H and G, the position of the vertical line is determined by its color. Try this in DBN and change G in the Set G line.

DBN Vocabulary**repeat**

A repeat loop is a way of telling the computer to do something more than one time.

The format is like this :

```
Repeat variableName start end
{
    code to be executed
}
```

Basically, a repeat loop takes a starting value as a parameter, an end value, then counts its way up (or down), each time setting the value of the variable to whatever it's counted to. It's one of those things that makes a lot more sense when you see it in action. A practical example :

```
Repeat count 25 50
{
    line 0 count 100 count
}
```

DBN Vocabulary**nested repeat**

Whenever DBN sees a repeat loop, it will do it until it's done. Repeat loops can be nested in repeat loops. Here's an example :

```
Repeat a 0 10
{
  Repeat b 0 10
  {
    Set [a b] (a*b)
  }
}
```

The important concept here is that for every one of the outer loops ($a = 0$, $a = 1$, $a = 2$) we do the eleven inner loops ($b = 0$, $b = 1$, $b = 2...$). Give it a try and it'll make sense.

DBN Vocabulary**same?**

checks whether two parameters passed to it are the same. If so, it executes whatever is in the block directly below it.

```
Same? a b
{
    // code to execute
}
```

Parameters a and b can be variables or numbers (remember that variables in the end become numbers when DBN looks them up in memory)

```
Same? 2 b
{
    // code to execute
}
```

Checks if 2 and the value of “b” are equal.

```
Same? 10 10
{
    // ..
}
```

Checks if $10 = 10$, in case you were unsure of that ...

DBN Vocabulary**notsame?**

similar to same, except this checks for not equal (in other circles commonly referred to as != with the ! meaning not).

```
NotSame? a b
{
    // code to execute
}
```

Parameters a and b can be variables or numbers (remember that variable in the end become numbers when DBN looks them up in memory)

```
NotSame? 59 yo
{
    // code to execute
}
```

Checks if 59 and the value of “yo” are not equal.

```
NotSame? 20 10
{
    // code to execute
}
```

Checks if $20 \neq 10$. Can't you sleep better knowing DBN can evaluate this?

DBN Vocabulary**smaller?**

checks whether the first of two parameters passed to it is smaller than the second. If so, it executes whatever is in the block directly below it. (in other languages, the “less than” symbol `<` is used).

```
Set a 25
Smaller? a 26
{
    // code to execute
}
```

Parameters `a` and `b` can be variables or numbers (remember that variable in the end become numbers when DBN looks them up in memory)

```
Repeat a 0 10
{
    Smaller? a 5
    {
        //draw a square
    }
}
```

This code will draw 5 squares because the square code will only execute when “a”

is 0, 1, 2, 3, and 4.

DBN Vocabulary**notsmaller?**

This is the opposite of smaller. It evaluates if the first of the two parameters is larger than the second. If it is, all the code in the block below it will be executed.

```
Set a 25
NotSmaller? a 24
{
    // code to execute
}
```

Parameters a and b can be variables or numbers (remember that variables in the end become numbers when DBN looks them up in memory)

```
Repeat a 0 10
{
    NotSmaller? a 5
    {
        //draw a square
    }
}
```

This code will draw 5 squares because the square code will only execute when “a” is 6, 7, 8, 9 and 10.

DBN Vocabulary**command**

so far we have used the standard commands that come with DBN, like line, paper, pen etc. We can also create our own custom commands. A command is a block of code that we assign a name and specify how many parameters it needs to be passed.

```
Command drawRect x1 y1 x2 y2 color
{
    Pen color
    Line x1 y1 x1 y2
    Line x1 y1 x2 y1
    Line x2 y1 x2 y2
    Line x2 y2 x1 y2
}
```

First, we type the command “command” (confusing, eh?) to tell dbn that we are giving the block of code below a name (aka: a place in memory). After that we’ve typed five variables (x1 y1 x2 y2 color) to hold the values of the five parameters we require for our command to work (in this case it’s the lower right corner coordinates, the upper left corner coordinates, and the color of the lines making the square). The block of code that follows is what is going to be executed whenever we use that command.

To call the command, we simply type into our program:

```
drawRect 10 10 90 90 100
```

DBN sees the command and automatically plugs the parameters into the variables in the order they were declared (x1=10, y1=10, x2=90.....) and then executes the code with those values.

**** It is important to define you commands BEFORE you ask DBN to execute them. Remember, the code is read line by line, starting from the top. If your code looks like this:**

```
drawRect 10 10 90 90 100

Command drawRect x1 y1 x2 y2 color
{
    Pen color
    Line x1 y1 x1 y2
    Line x1 y1 x2 y1
    Line x2 y1 x2 y2
    Line x2 y2 x1 y2
}
```

DBN will hit the “drawRect...” line first and go “huh?” because you haven’t yet taught it what drawRect means.

DBN Vocabulary**nesting**

We can also use the commands we have created to create other commands. This is called **nesting commands**.

```
Command pinwheel x y size color
{
    drawRect (x-size) y x (y+(size/2)) color
    drawRect x y (x+(size/2)) (y+size) color
    drawRect x (y-(size/2)) (x+size) y color
    drawRect (x-(size/2)) (y-size) x y color
}

pinwheel 50 50 12 100
```

This creates a command called “pinwheel” that uses the previous drawRect command to place four squares in the shape of a pinwheel. All we have to do is give it the x and y of where we want it to go, the size of the wing, and the color we want.

DBN Vocabulary

pause

is used to stall the execution of your code for a specific amount of time. Values given to pause are executed in hundreths of a second. When DBN sees a pause, it stops and waits for the specified amount of time before continuing with the rest of the code.

```
Pause 5
```

This would cause DBN to wait for 5 hundreths of a second.

```
Pause 200
```

This would cause DBN to wait for 2 seconds.

DBN Vocabulary**mouse**

DBN has one input from the computer, the mouse. It lets you access the coordinate positions of the mouse as well as check whether the mouse is being pressed or not. Therefore, if you move or click the mouse in the DBN window, values are stored in <mouse 1>, <mouse 2>, and <mouse 3>. You can use these values to pass to commands, set variables, etc.

```
Forever
{
  paper <mouse 1>
}
```

This changes the screen color to be the same value as the x mouse position.

```
Forever
{
  Line 0 0 <mouse 1> <mouse 2>
}
```

This draws a line that starts in the bottom left corner and ends wherever the mouse is.

```
Same? <mouse 3> 1
{
```

```
// code to be executed  
}
```

This executes the code in the block only when the mouse is pressed. <mouse 3> equals 0 when the mouse is not being presses, and it equals 1 when it is being pressed.

DBN Vocabulary**key**

In the DBN system, a total of 26 keys from the standard keyboard are available to sense the presence of pressure from a finger, pencil, or whatever can come into physical contact with the keyboard. The 26 keys correspond to the letters A through Z. The state of each key can be accessed through the external data interface < > with the keyword Key followed by a numeric descriptor ranging from 1 to 26. For example, when the A key is pressed, <Key 1> is 100; when it is released, <Key 1> is 0. Recall that the mouse button, identified by <Mouse 3>, behaves in an identical manner.

For now, try the ff:

```
Fast
Paper 100
Pen 0
// spread out flare positions
// and shade based upon height
Line 0 0 100 0
Forever
{
  Repeat A 1 26
  {
    Same? <Key A> 100
```

```
    {  
      Set [A 0] ([A 0]+1)  
      Set V (100-[A 0])  
      Set [(A*3) [A 0]] V  
    }  
  }  
}
```

Run then start pressing the letters of the alphabet.

Parts of this section is from Design by Numbers by John Maeda, MIT Press, 1999.

DBN Vocabulary**time**

I began designing clocks when I realized that time is the most relevant subject to depict by means of a dynamically changing form. The clock of the computer can be accessed with the external data tag Time with one descriptor. <Time 1> is the hour 0 to 23, <Time 2> is the minutes 0 to 59, and <Time 3> is the number of seconds 0 to 59. Given the ability to computationally observe the progress of time, a form that can reflect the time is easily constructed as three lines, where each line shall represent the respective hour, minute, and second reading.

```
// display time as lines

Forever
{
  Paper 0
  Pen 100
  Line 0 75 <Time 1> 75
  Line 0 50 <Time 2> 50
  Line 0 25 <Time 3> 25
}
```

Parts of this section is from Design by Numbers by John Maeda, MIT Press, 1999.

DBN Vocabulary**summary**

paper <p> // paper color

pen <p> // pen color

line <x1> <y1> <x2> <y2> // lines

field <x1> <y1> <x2> <y2> <fill> // filled area

[<x> <y>] // get dot

set <variable> <value> // assignment

set [<x> <y>] <color> // set dot

repeat <variable> <start> <end> {} // iteration

same? <value1> <value2> // if this then..

notsame? <value1> <value2> // if not then..

smaller? <value1> <value2> // less than

notsmaller? <value1> <value2> // greater than

command <name> <variables1> ... <variable n> // get creative

number <name> <param1> .. <param n> { value <result> } // custom math

forever {} // never stop

pause <hundreths> // wait..

<array n> where n is between 1 and 1000 // storage

<mouse 1> <mouse 2> <mouse 3> // inputs

<key number> // keys

<time 1> <time 2> <time 3> <time 4> // time

// // comments

+ - * / % // math

() // embed

fast // so fast!

refresh // no flicker