DC, An Arbitrary Precision Calculator

by Richard Stallman

Published by the Free Software Foundation,
675 Massachusetts Avenue,
Cambridge, MA 02139 USA

1 Introduction

DC is a reverse-polish desk calculator which supports unlimited precision arithmetic. It also allows you to define and call macros. Normally DC reads from the standard input; if any command arguments are given to it, they are filenames, and DC reads and executes the contents of the files before reading from standard input. All output is to standard output.

To exit, use '`q`'. *C-c* does not exit; it is used to abort macros that are looping, etc. (Currently this is not true; *C-c* does exit.)

A reverse-polish calculator stores numbers on a stack. Entering a number pushes it on the stack. Arithmetic operations pop arguments off the stack and push the results.

To enter a number in DC, type the digits, with an optional decimal point. Exponential notation is not supported. To enter a negative number, begin the number with '`_`'. '`-`' cannot be used for this, as it is a binary operator for subtraction instead. To enter two numbers in succession, separate them with spaces or newlines. These have no meaning as commands.

2 Printing Commands

‘`p`’ Prints the value on the top of the stack, without altering the stack. A newline is printed after the value.

‘`P`’ Prints the value on the top of the stack, popping it off, and does not print a newline after.

‘`f`’ Prints the entire contents of the stack and the contents of all of the registers, without altering anything. This is a good command to use if you are lost or want to figure out what the effect of some command has been.

3 Arithmetic

‘`+`’ Pops two values off the stack, adds them, and pushes the result. The precision of the result is determined only by the values of the arguments, and is enough to be exact.

‘`-`’ Pops two values, subtracts the first one popped from the second one popped, and pushes the result.

‘`*`’ Pops two values, multiplies them, and pushes the result. The number of fraction digits in the result is controlled by the current precision flag (see below) and does not depend on the values being multiplied.

‘`/`’ Pops two values, divides the second one popped from the first one popped, and pushes the result. The number of fraction digits is specified by the precision flag.

‘`%`’ Pops two values, computes the remainder of the division that the ‘`/`’ command would do, and pushes that. The division is done with as many fraction digits as the precision flag specifies, and the remainder is also computed with that many fraction digits.

‘`^`’ Pops two values and exponentiates, using the first value popped as the exponent and the second popped as the base. The fraction part of the exponent is ignored. The precision flag specifies the number of fraction digits in the result.

‘`v`’ Pops one value, computes its square root, and pushes that. The precision flag specifies the number of fraction digits in the result.

Most arithmetic operations are affected by the "precision flag", which you can set with the ‘`k`’ command. The default precision value is zero, which means that all arithmetic except for addition and subtraction produces integer results.

The remainder operation (‘`%`’) requires some explanation: applied to arguments ‘`a`’ and ‘`b`’ it produces ‘`a - (b * (a / b))`’, where ‘`a / b`’ is computed in the current precision.

4 Stack Control

`c`
: Clears the stack, rendering it empty.

`d`
: Duplicates the value on the top of the stack, pushing another copy of it. Thus, ‘4d*p’ computes 4 squared and prints it.

5 Registers

DC provides 128 memory registers, each named by a single ASCII character. You can store a number in a register and retrieve it later.

‘`s`*r*’ Pop the value off the top of the stack and store it into register *r*.

‘`l`*r*’ Copy the value in register *r*, and push it onto the stack. This does not alter the contents of *r*.

Each register also contains its own stack. The current register value is the top of the register’s stack.

‘`S`*r*’ Pop the value off the top of the (main) stack and push it onto the stack of register *r*. The previous value of the register becomes inaccessible.

‘`L`*r*’ Pop the value off the top of register *r*’s stack and push it onto the main stack. The previous value in register *r*’s stack, if any, is now accessible via the ‘l*r*’ command.

The ‘`f`’ command prints a list of all registers that have contents stored in them, together with their contents. Only the current contents of each register (the top of its stack) is printed.

6 Parameters

DC has three parameters that control its operation: the precision, the input radix, and the output radix. The precision specifies the number of fraction digits to keep in the result of most arithmetic operations. The input radix controls the interpretation of numbers typed in; *all* numbers typed in use this radix. The output radix is used for printing numbers.

The input and output radices are separate parameters; you can make them unequal, which can be useful or confusing. Each radix must be between 2 and 36 inclusive. The precision must be zero or greater. The precision is always measured in decimal digits, regardless of the current input or output radix.

‘`i`’ Pops the value off the top of the stack and uses it to set the input radix.

‘`o`’
‘`k`’ Similarly set the output radix and the precision.

‘`I`’ Pushes the current input radix on the stack.

‘`O`’
‘`K`’ Similarly push the current output radix and the current precision.

7 Strings

DC can operate on strings as well as on numbers. The only things you can do with strings are print them and execute them as macros (which means that the contents of the string are processed as DC commands). Both registers and the stack can hold strings, and DC always knows whether any given object is a string or a number. Some commands such as arithmetic operations demand numbers as arguments and print errors if given strings. Other commands can accept either a number or a string; for example, the '`p`' command can accept either and prints the object according to its type.

'`[`*`characters`*`]`'
: Makes a string containing *characters* and pushes it on the stack. For example, '`[foo]P`' prints the characters '`foo`' (with no newline).

'`x`'
: Pops a value off the stack and executes it as a macro. Normally it should be a string; if it is a number, it is simply pushed back onto the stack. For example, '`[1p]x`' executes the macro '`1p`', which pushes 1 on the stack and prints '`1`' on a separate line.

  Macros are most often stored in registers; '`[1p]sa`' stores a macro to print '`1`' into register '`a`', and '`lax`' invokes the macro.

'`>`*r*'
: Pops two values off the stack and compares them assuming they are numbers, executing the contents of register *r* as a macro if the original top-of-stack is greater. Thus, '`1 2>a`' will invoke register '`a`''s contents and '`2 1>a`' will not.

'`<`*r*'
: Similar but invokes the macro if the original top-of-stack is less.

'`=`*r*'
: Similar but invokes the macro if the two numbers popped are equal. This can also be validly used to compare two strings for equality.

'`?`'
: Reads a line from the terminal and executes it. This command allows a macro to request input from the user.

'`q`'
: During the execution of a macro, this comand does not exit DC. Instead, it exits from that macro and also from the macro which invoked it (if any).

'`Q`'
: Pops a value off the stack and uses it as a count of levels of macro execution to be exited. Thus, '`3Q`' exits three levels.

8 Status Inquiry

‘`Z`’	Pops a value off the stack, calculates the number of digits it has (or number of characters, if it is a string) and pushes that number.
‘`X`’	Pops a value off the stack, calculates the number of fraction digits it has, and pushes that number. For a string, the value pushed is -1.
‘`z`’	Pushes the current stack depth; the number of objects on the stack before the execution of the ‘`z`’ command.
‘`I`’	Pushes the current value of the input radix.
‘`O`’	Pushes the current value of the output radix.
‘`K`’	Pushes the current value of the precision.

9 Notes

The ‘:’ and ‘;’ commands of the Unix DC program are not supported, as the documentation does not say what they do. The ‘!’ command is not supported, but will be supported as soon as a library for executing a line as a command exists.

Table of Contents