GNU dc

an arbitrary precision calculator
version 1.4.1

by Ken Pizzini
original manual by Richard Stallman

This manual documents version 1.4.1 of GNU `dc`, an arbitrary precision calculator.

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 instead of reading from standard input. All normal output is to standard output; all error messages are written to standard error.

To exit, use '`q`'. *C-c* (or whatever other keystroke your system uses to generate a `SIGINT`) does not exit; it is used to abort macros that are looping, etc.

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 (using upper case letters `A` through `F` as `"digits"` when working with input bases greater than ten), 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 Invocation

`dc` may be invoked with the following command-line options:

'`-e` *expr*'

'`--expression=`*expr*'
: Evaluate *expr* as `dc` commands.

'`-f` *file*'

'`--file=`*file*'
: Read and evaluate `dc` commands from *file*.

'`-h`'

'`--help`' Print a usage message summarizing the command-line options, then exit.

'`-V`'

'`--version`'
: Print the version information for this program, then exit.

If any command-line parameters remain after processing the options, these parameters are interpreted as additional *file*s whose contents are read and evaluated. A file name of `-` refers to the standard input stream. If no `-e` option was specified, and no files were specified, then the standard input will be read for commands to evaluate.

3 Printing Commands

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

‘`n`’
: Prints the value on the top of the stack, popping it off, and does not print a newline after. (This command is a GNU extension.)

‘`P`’
: Pops off the value on top of the stack. If it it a string, it is simply printed without a trailing newline. Otherwise it is a number, and the integer portion of its absolute value is printed out as a "base (UCHAR_MAX+1)" byte stream. Assuming that (UCHAR_MAX+1) is 256 (as it is on most machines with 8-bit bytes), the sequence `KSK0k1/ _1Ss[ls*]Sxd0>x [256~Ssd0<x]dsxx sx[q]Sq[Lsd0>qaPlxx]dsxx sx0sqLqsxLxLK+k` could also accomplish this function. (Much of the complexity of the above native-dc code is due to the ~ computing the characters backwards, and the desire to ensure that all registers wind up back in their original states.) (Details of the behavior with a number are a GNU extension. Traditional `dc` happened to "support" similar functionality for a limited range of inputs as an accidental side-effect of its internal representation of numbers.)

‘`f`’
: Prints the entire contents of the stack 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.

All numeric output is split to fit within 70 columns, by default. When a number is broken up in this way, the split is indicated by a "\" at the end of the to-be-continued output lines. The column width at which output is split can be overridden by setting the *DC_LINE_LENGTH* environment variable to the desired width. A *DC_LINE_LENGTH* of 0 (zero) disables the line-split feature altogether. Invalid values of *DC_LINE_LENGTH* are silently ignored. (The *DC_LINE_LENGTH* variable is a GNU extension.)

4 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 the largest of the precision value, the number of fraction digits in the multiplier, or the number of fraction digits in the multiplicand; but in no event exceeding the number of digits required for an exact result.

‘`/`’
: 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 value.

'%' Pops two values, computes the remainder of the division that the '/' command would do, and pushes that. The value computed is the same as that computed by the sequence `Sd dld/ Ld*-` .

'~' Pops two values, divides the second one popped from the first one popped. The quotient is pushed first, and the remainder is pushed next. The number of fraction digits used in the division is specified by the precision value. (The sequence `SdSn lnld/ LnLd%` could also accomplish this function, with slightly different error checking.) (This command is a GNU extension.)

'^' 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 value specifies the number of fraction digits in the result.

'|' Pops three values and computes a modular exponentiation. The first value popped is used as the reduction modulus; this value must be a non-zero number, and the result may not be accurate if the modulus is not an integer. The second popped is used as the exponent; this value must be a non-negative number, and any fractional part of this exponent will be ignored. The third value popped is the base which gets exponentiated, which should be an integer. For small integers this is like the sequence `Sm^Lm%`, but, unlike `^`, this command will work with arbitrarily large exponents. (This command is a GNU extension.)

'v' Pops one value, computes its square root, and pushes that. The maximum of the precision value and the precision of the argument is used to determine the number of fraction digits in the result.

Most arithmetic operations are affected by the *precision value*, 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.

5 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.

'r' Reverses the order of (swaps) the top two values on the stack. (This can also be accomplished with the sequence `SaSbLaLb`.) (This command is a GNU extension.)

'R' Pops the top-of-stack as an integer *n*. Cyclically rotates the top *n* items on the updated stack. If *n* is positive, then the rotation direction will make the topmost element the second-from top; if *n* is negative, then the rotation will make the topmost element the *n*-th element from the top. If the stack depth is less than *n* then the entire stack is rotated (in the appropriate direction), without any error being reported. (This command is a GNU extension.)

6 Registers

`dc` provides at least 256 memory registers[1], each named by a single character. You can store a number in a register and retrieve it later.

'`sr`' Pop the value off the top of the stack and store it into register *r*.

'`lr`' Copy the value in register *r*, and push it onto the stack. The value `0` is retrieved if the register is uninitialized or its stack has become empty. 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.

'`Sr`' 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.

'`Lr`' 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 '`lr`' command.

7 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. The input radix must be between 2 and 16 inclusive. The output radix must be at least 2. 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`' Pops the value off the top of the stack and uses it to set the output radix.

'`k`' Pops the value off the top of the stack and uses it to set the precision.

'`I`' Pushes the current input radix on the stack.

'`O`' Pushes the current output radix on the stack.

'`K`' Pushes the current precision on the stack.

---

[1] The exact number of registers provided by `dc` depends on the range of an `unsigned char` in the C compiler used to create the `dc` executable.

8 Strings

`dc` has a limited ability to 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). Note that all square brackets ('`[`'s and '`]`'s) must be balanced; there is no mechanism provided for handling unbalanced square brackets.

'`a`'
: The mnemonic for this is somewhat erroneous: asciify. The top-of-stack is popped. If it was a number, then the low-order byte of this number is converted into a 1-character string and pushed onto the stack. Otherwise the top-of-stack was a string, and the first character of that string is pushed back. (This command is a GNU extension.)

'`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 not greater (is less than or equal to) what was the second-to-top.

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

'`!<`*r*'
: Similar but invokes the macro if the original top-of-stack is not less (is greater than or equal to) what was the second-to-top.

'`=`*r*'
: Similar but invokes the macro if the two numbers popped are equal.

'`!=`*r*'
: Similar but invokes the macro if the two numbers popped are not equal.

'`?`'
: 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 command exits from the macro and also from the macro which invoked it. If called from the top level, or from a macro which was called directly from the top level, the '`q`' command will cause `dc` to exit.

'`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.

9 Status Inquiry

'Z' Pops a value off the stack, calculates the number of decimal digits it has (or number of characters, if it is a string) and pushes that number.

Note that the digit count for a number does *not* include any leading zeros, even if those appear to the right of the radix point. This may seem a bit strange at first, but it is compatible with historical implementations of `dc`, and can be argued to be useful for computing the magnitude of a value: `dSaXLaZ-` will compute the power-of-ten multiplier which would be needed to shift the decimal point to be immediately before the leftmost non-zero digit.

'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 0.

'z' Pushes the current stack depth: the number of objects on the stack before the execution of the 'z' command.

10 Miscellaneous

'!' Will run the rest of the line as a system command. Note that parsing of the !<, !=, and !> commands take precedence, so if you want to run a command starting with <, =, or > you will need to add a space after the !.

'#' Will interpret the rest of the line as a comment. (This command is a GNU extension.)

':*r*' Will pop the top two values off of the stack. The old second-to-top value will be stored in the array *r*, indexed by the old top-of-stack value.

';*r*' Pops the top-of-stack and uses it as an index into the array *r*. The selected value is then pushed onto the stack.

Note that each stacked instance of a register has its own array associated with it. Thus '1 *0:*a 0Sa 2 *0:*a La *0;*ap' will print 1, because the 2 was stored in an instance of *0:a* that was later popped.

11 Reporting bugs

Email bug reports to `bug-dc@gnu.org`.

Table of Contents