Name

dc - arbitrary-precision decimal reverse-Polish notation calculator

SYNOPSIS

dc [**-cChiPRvVx**] [**--version**] [**--help**] [**--digit-clamp**] [**--no-digit-clamp**] [**--interactive**] [**--no-prompt**] [**--no-read-prompt**] [**--extended-register**] [**-e** *expr*] [**--expression**=*expr*...] [**-f** *file*...] [**--file**=*file*...] [*file*...] [**-I** *ibase*] [**--ibase**=*ibase*] [**-O** *obase*] [**--obase**=*obase*] [**-S** *scale*] [**--scale**=*scale*] [**-E** *seed*] [**--seed**=*seed*]

DESCRIPTION

dc(1) is an arbitrary-precision calculator. It uses a stack (reverse Polish notation) to store numbers and results of computations. Arithmetic operations pop arguments off of the stack and push the results.

If no files are given on the command-line, then dc(1) reads from **stdin** (see the **STDIN** section). Otherwise, those files are processed, and dc(1) will then exit.

If a user wants to set up a standard environment, they can use **DC_ENV_ARGS** (see the **ENVIRONMENT VARIABLES** section). For example, if a user wants the **scale** always set to **10**, they can set **DC_ENV_ARGS** to **-e 10k**, and this dc(1) will always start with a **scale** of **10**.

OPTIONS

The following are the options that dc(1) accepts.

-C, **--no-digit-clamp**

Disables clamping of digits greater than or equal to the current **ibase** when parsing numbers.

This means that the value added to a number from a digit is always that digit's value multiplied by the value of ibase raised to the power of the digit's position, which starts from 0 at the least significant digit.

If this and/or the **-c** or **--digit-clamp** options are given multiple times, the last one given is used.

This option overrides the **DC_DIGIT_CLAMP** environment variable (see the **ENVIRONMENT VARIABLES** section) and the default, which can be queried with the **-h** or **--help** options.

This is a **non-portable extension**.

-c, **--digit-clamp**

Enables clamping of digits greater than or equal to the current **ibase** when parsing numbers.

This means that digits that the value added to a number from a digit that is greater than or equal to the ibase is the value of ibase minus 1 all multiplied by the value of ibase raised to the power of the digit's position, which starts from 0 at the least significant digit.

If this and/or the **-C** or **--no-digit-clamp** options are given multiple times, the last one given is used.

This option overrides the **DC_DIGIT_CLAMP** environment variable (see the **ENVIRONMENT VARIABLES** section) and the default, which can be queried with the **-h** or **--help** options.

This is a **non-portable extension**.

-E *seed*, **--seed**=*seed*

Sets the builtin variable **seed** to the value *seed* assuming that *seed* is in base 10. It is a fatal error if *seed* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-e *expr*, **--expression**=*expr*

Evaluates *expr*. If multiple expressions are given, they are evaluated in order. If files are given as well (see below), the expressions and files are evaluated in the order given. This means that if a file is given before an expression, the file is read in and evaluated first.

If this option is given on the command-line (i.e., not in **DC_ENV_ARGS**, see the **ENVIRONMENT VARIABLES** section), then after processing all expressions and files, dc(1) will exit, unless **-** (**stdin**) was given as an argument at least once to **-f** or **--file**, whether on the command-line or in **DC_ENV_ARGS**. However, if any other **-e**, **--expression**, **-f**, or **--file** arguments are given after **-f-** or equivalent is given, dc(1) will give a fatal error and exit.

This is a **non-portable extension**.

-f *file*, **--file**=*file*

Reads in *file* and evaluates it, line by line, as though it were read through **stdin**. If expressions are also given (see above), the expressions are evaluated in the order given.

If this option is given on the command-line (i.e., not in **DC_ENV_ARGS**, see the **ENVIRONMENT VARIABLES** section), then after processing all expressions and files, dc(1) will exit, unless **-** (**stdin**) was given as an argument at least once to **-f** or **--file**. However, if any other **-e**, **--expression**, **-f**, or **--file** arguments are given after **-f-** or equivalent is given, dc(1) will give a fatal error and exit.

This is a **non-portable extension**.

-h, **--help**
Prints a usage message and exits.

-I *ibase*, **--ibase**=*ibase*
Sets the builtin variable **ibase** to the value *ibase* assuming that *ibase* is in base 10. It is a fatal error if *ibase* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-i, **--interactive**
Forces interactive mode. (See the **INTERACTIVE MODE** section.)

This is a **non-portable extension**.

-L, **--no-line-length**
Disables line length checking and prints numbers without backslashes and newlines. In other words, this option sets **BC_LINE_LENGTH** to **0** (see the **ENVIRONMENT VARIABLES** section).

This is a **non-portable extension**.

-O *obase*, **--obase**=*obase*
Sets the builtin variable **obase** to the value *obase* assuming that *obase* is in base 10. It is a fatal error if *obase* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-P, **--no-prompt**
Disables the prompt in TTY mode. (The prompt is only enabled in TTY mode. See the **TTY MODE** section.) This is mostly for those users that do not want a prompt or are not used to having them in dc(1). Most of those users would want to put this option in **DC_ENV_ARGS**.

These options override the **DC_PROMPT** and **DC_TTY_MODE** environment variables (see the **ENVIRONMENT VARIABLES** section).

This is a **non-portable extension**.

-R, **--no-read-prompt**

Disables the read prompt in TTY mode. (The read prompt is only enabled in TTY mode. See the **TTY MODE** section.) This is mostly for those users that do not want a read prompt or are not used to having them in dc(1). Most of those users would want to put this option in **BC_ENV_ARGS** (see the **ENVIRONMENT VARIABLES** section). This option is also useful in hash bang lines of dc(1) scripts that prompt for user input.

This option does not disable the regular prompt because the read prompt is only used when the **?** command is used.

These options *do* override the **DC_PROMPT** and **DC_TTY_MODE** environment variables (see the **ENVIRONMENT VARIABLES** section), but only for the read prompt.

This is a **non-portable extension**.

-S *scale*, **--scale**=*scale*

Sets the builtin variable **scale** to the value *scale* assuming that *scale* is in base 10. It is a fatal error if *scale* is not a valid number.

If multiple instances of this option are given, the last is used.

This is a **non-portable extension**.

-v, **-V**, **--version**

Print the version information (copyright header) and exits.

-x --extended-register

Enables extended register mode. See the *Extended Register Mode* subsection of the **REGISTERS** section for more information.

This is a **non-portable extension**.

-z, **--leading-zeroes**

Makes dc(1) print all numbers greater than **-1** and less than **1**, and not equal to **0**, with a leading zero.

This is a **non-portable extension**.

All long options are **non-portable extensions**.

STDIN

If no files are given on the command-line and no files or expressions are given by the **-f**, **--file**, **-e**, or **--expression** options, then dc(1) reads from **stdin**.

However, there is a caveat to this.

First, **stdin** is evaluated a line at a time. The only exception to this is if a string has been finished, but not ended. This means that, except for escaped brackets, all brackets must be balanced before dc(1) parses and executes.

STDOUT

Any non-error output is written to **stdout**. In addition, if history (see the **HISTORY** section) and the prompt (see the **TTY MODE** section) are enabled, both are output to **stdout**.

Note: Unlike other dc(1) implementations, this dc(1) will issue a fatal error (see the **EXIT STATUS** section) if it cannot write to **stdout**, so if **stdout** is closed, as in **dc >&-**, it will quit with an error. This is done so that dc(1) can report problems when **stdout** is redirected to a file.

If there are scripts that depend on the behavior of other dc(1) implementations, it is recommended that those scripts be changed to redirect **stdout** to **/dev/null**.

STDERR

Any error output is written to **stderr**.

Note: Unlike other dc(1) implementations, this dc(1) will issue a fatal error (see the **EXIT STATUS** section) if it cannot write to **stderr**, so if **stderr** is closed, as in **dc 2>&-**, it will quit with an error. This is done so that dc(1) can exit with an error code when **stderr** is redirected to a file.

If there are scripts that depend on the behavior of other dc(1) implementations, it is recommended that those scripts be changed to redirect **stderr** to **/dev/null**.

SYNTAX

Each item in the input source code, either a number (see the **NUMBERS** section) or a command (see the **COMMANDS** section), is processed and executed, in order. Input is processed immediately when entered.

ibase is a register (see the **REGISTERS** section) that determines how to interpret constant numbers. It is the "input" base, or the number base used for interpreting input numbers. **ibase** is initially **10**. The max allowable value for **ibase** is **16**. The min allowable value for **ibase** is **2**. The max allowable value for **ibase** can be queried in dc(1) programs with the **T** command.

obase is a register (see the **REGISTERS** section) that determines how to output results. It is the "output" base, or the number base used for outputting numbers. **obase** is initially **10**. The max allowable value for **obase** is **DC_BASE_MAX** and can be queried with the **U** command. The min allowable value for **obase** is **0**. If **obase** is **0**, values are output in scientific notation, and if **obase** is **1**, values are output in engineering notation. Otherwise, values are output in the specified base.

Outputting in scientific and engineering notations are **non-portable extensions**.

The *scale* of an expression is the number of digits in the result of the expression right of the decimal point, and **scale** is a register (see the **REGISTERS** section) that sets the precision of any operations (with exceptions). **scale** is initially **0**. **scale** cannot be negative. The max allowable value for **scale** can be queried in dc(1) programs with the **V** command.

seed is a register containing the current seed for the pseudo-random number generator. If the current value of **seed** is queried and stored, then if it is assigned to **seed** later, the pseudo-random number generator is guaranteed to produce the same sequence of pseudo-random numbers that were generated after the value of **seed** was first queried.

Multiple values assigned to **seed** can produce the same sequence of pseudo-random numbers. Likewise, when a value is assigned to **seed**, it is not guaranteed that querying **seed** immediately after will return the same value. In addition, the value of **seed** will change after any call to the **'** command or the **"** command that does not get receive a value of **0** or **1**. The maximum integer returned by the **'** command can be queried with the **W** command.

Note: The values returned by the pseudo-random number generator with the **'** and **"** commands are guaranteed to **NOT** be cryptographically secure. This is a consequence of using a seeded pseudo-random number generator. However, they *are* guaranteed to be reproducible with identical **seed** values. This means that the pseudo-random values from dc(1) should only be used where a reproducible stream of pseudo-random numbers is *ESSENTIAL*. In any other case, use a non-seeded pseudo-random number generator.

The pseudo-random number generator, **seed**, and all associated operations are **non-portable extensions**.

Comments

Comments go from **#** until, and not including, the next newline. This is a **non-portable extension**.

NUMBERS

Numbers are strings made up of digits, uppercase letters up to **F**, and at most **1** period for a radix. Numbers can have up to **DC_NUM_MAX** digits. Uppercase letters are equal to **9** plus their position in the alphabet (i.e., **A** equals **10**, or **9+1**).

If a digit or letter makes no sense with the current value of **ibase** (i.e., they are greater than or equal to the current value of **ibase**), then the behavior depends on the existence of the **-c**/**--digit-clamp** or **-C**/**--no-digit-clamp** options (see the **OPTIONS** section), the existence and setting of the **DC_DIGIT_CLAMP** environment variable (see the **ENVIRONMENT VARIABLES** section), or the default, which can be queried with the **-h**/**--help** option.

If clamping is off, then digits or letters that are greater than or equal to the current value of **ibase** are not changed. Instead, their given value is multiplied by the appropriate power of **ibase** and added into the number. This means that, with an **ibase** of **3**, the number **AB** is equal to **3^1*A+3^0*B**, which is **3** times **10** plus **11**, or **41**.

If clamping is on, then digits or letters that are greater than or equal to the current value of **ibase** are set to the value of the highest valid digit in **ibase** before being multiplied by the appropriate power of **ibase** and added into the number. This means that, with an **ibase** of **3**, the number **AB** is equal to **3^1*2+3^0*2**, which is **3** times **2** plus **2**, or **8**.

There is one exception to clamping: single-character numbers (i.e., **A** alone). Such numbers are never clamped and always take the value they would have in the highest possible **ibase**. This means that **A** alone always equals decimal **10** and **Z** alone always equals decimal **35**. This behavior is mandated by the standard for bc(1) (see the STANDARDS section) and is meant to provide an easy way to set the current **ibase** (with the **i** command) regardless of the current value of **ibase**.

If clamping is on, and the clamped value of a character is needed, use a leading zero, i.e., for **A**, use **0A**.

In addition, dc(1) accepts numbers in scientific notation. These have the form **<number>e<integer>**. The exponent (the portion after the **e**) must be an integer. An example is **1.89237e9**, which is equal to **1892370000**. Negative exponents are also allowed, so **4.2890e_3** is equal to **0.0042890**.

WARNING: Both the number and the exponent in scientific notation are interpreted according to the current **ibase**, but the number is still multiplied by **10^exponent** regardless of the current **ibase**. For example, if **ibase** is **16** and dc(1) is given the number string **FFeA**, the resulting decimal number will be **2550000000000**, and if dc(1) is given the number string **10e_4**, the resulting decimal number will be **0.0016**.

Accepting input as scientific notation is a **non-portable extension**.

COMMANDS

The valid commands are listed below.

Printing

These commands are used for printing.

Note that both scientific notation and engineering notation are available for printing numbers. Scientific notation is activated by assigning **0** to **obase** using **0o**, and engineering notation is activated by assigning **1** to **obase** using **1o**. To deactivate them, just assign a different value to **obase**.

Printing numbers in scientific notation and/or engineering notation is a **non-portable extension**.

p Prints the value on top of the stack, whether number or string, and prints a newline after.

This does not alter the stack.

n Prints the value on top of the stack, whether number or string, and pops it off of the stack.

P Pops a value off the stack.

If the value is a number, it is truncated and the absolute value of the result is printed as though **obase** is **256** and each digit is interpreted as an 8-bit ASCII character, making it a byte stream.

If the value is a string, it is printed without a trailing newline.

This is a **non-portable extension**.

f Prints the entire contents of the stack, in order from newest to oldest, without altering anything.

Users should use this command when they get lost.

Arithmetic

These are the commands used for arithmetic.

+ The top two values are popped off the stack, added, and the result is pushed onto the stack. The *scale* of the result is equal to the max *scale* of both operands.

- The top two values are popped off the stack, subtracted, and the result is pushed onto the stack. The *scale* of the result is equal to the max *scale* of both operands.

*****
: The top two values are popped off the stack, multiplied, and the result is pushed onto the stack. If **a** is the *scale* of the first expression and **b** is the *scale* of the second expression, the *scale* of the result is equal to **min(a+b,max(scale,a,b))** where **min()** and **max()** return the obvious values.

/
: The top two values are popped off the stack, divided, and the result is pushed onto the stack. The *scale* of the result is equal to **scale**.

The first value popped off of the stack must be non-zero.

%
: The top two values are popped off the stack, remaindered, and the result is pushed onto the stack.

Remaindering is equivalent to 1) Computing **a/b** to current **scale**, and 2) Using the result of step 1 to calculate **a-(a/b)*b** to *scale* **max(scale+scale(b),scale(a))**.

The first value popped off of the stack must be non-zero.

~
: The top two values are popped off the stack, divided and remaindered, and the results (divided first, remainder second) are pushed onto the stack. This is equivalent to **x y / x y %** except that **x** and **y** are only evaluated once.

The first value popped off of the stack must be non-zero.

This is a **non-portable extension**.

^
: The top two values are popped off the stack, the second is raised to the power of the first, and the result is pushed onto the stack. The *scale* of the result is equal to **scale**.

The first value popped off of the stack must be an integer, and if that value is negative, the second value popped off of the stack must be non-zero.

v
: The top value is popped off the stack, its square root is computed, and the result is pushed onto the stack. The *scale* of the result is equal to **scale**.

The value popped off of the stack must be non-negative.

_
: If this command *immediately* precedes a number (i.e., no spaces or other commands), then that number is input as a negative number.

Otherwise, the top value on the stack is popped and copied, and the copy is negated and pushed onto the stack. This behavior without a number is a **non-portable extension**.

b The top value is popped off the stack, and if it is zero, it is pushed back onto the stack. Otherwise, its absolute value is pushed onto the stack.

This is a **non-portable extension**.

| The top three values are popped off the stack, a modular exponentiation is computed, and the result is pushed onto the stack.

The first value popped is used as the reduction modulus and must be an integer and non-zero. The second value popped is used as the exponent and must be an integer and non-negative. The third value popped is the base and must be an integer.

This is a **non-portable extension**.

$ The top value is popped off the stack and copied, and the copy is truncated and pushed onto the stack.

This is a **non-portable extension**.

@ The top two values are popped off the stack, and the precision of the second is set to the value of the first, whether by truncation or extension.

The first value popped off of the stack must be an integer and non-negative.

This is a **non-portable extension**.

H The top two values are popped off the stack, and the second is shifted left (radix shifted right) to the value of the first.

The first value popped off of the stack must be an integer and non-negative.

This is a **non-portable extension**.

h The top two values are popped off the stack, and the second is shifted right (radix shifted left) to the value of the first.

The first value popped off of the stack must be an integer and non-negative.

This is a **non-portable extension**.

G The top two values are popped off of the stack, they are compared, and a **1** is pushed if they are equal, or **0** otherwise.

This is a **non-portable extension**.

N The top value is popped off of the stack, and if it a **0**, a **1** is pushed; otherwise, a **0** is pushed.

This is a **non-portable extension**.

(The top two values are popped off of the stack, they are compared, and a **1** is pushed if the first is less than the second, or **0** otherwise.

This is a **non-portable extension**.

{ The top two values are popped off of the stack, they are compared, and a **1** is pushed if the first is less than or equal to the second, or **0** otherwise.

This is a **non-portable extension**.

) The top two values are popped off of the stack, they are compared, and a **1** is pushed if the first is greater than the second, or **0** otherwise.

This is a **non-portable extension**.

} The top two values are popped off of the stack, they are compared, and a **1** is pushed if the first is greater than or equal to the second, or **0** otherwise.

This is a **non-portable extension**.

M The top two values are popped off of the stack. If they are both non-zero, a **1** is pushed onto the stack. If either of them is zero, or both of them are, then a **0** is pushed onto the stack.

This is like the **&&** operator in bc(1), and it is *not* a short-circuit operator.

This is a **non-portable extension**.

m The top two values are popped off of the stack. If at least one of them is non-zero, a **1** is pushed onto the stack. If both of them are zero, then a **0** is pushed onto the stack.

This is like the **||** operator in bc(1), and it is *not* a short-circuit operator.

This is a **non-portable extension**.

Pseudo-Random Number Generator

dc(1) has a built-in pseudo-random number generator. These commands query the pseudo-random number generator. (See Parameters for more information about the **seed** value that controls the pseudo-random number generator.)

The pseudo-random number generator is guaranteed to **NOT** be cryptographically secure.

' Generates an integer between 0 and **DC_RAND_MAX**, inclusive (see the **LIMITS** section).

The generated integer is made as unbiased as possible, subject to the limitations of the pseudo-random number generator.

This is a **non-portable extension**.

" Pops a value off of the stack, which is used as an **exclusive** upper bound on the integer that will be generated. If the bound is negative or is a non-integer, an error is raised, and dc(1) resets (see the **RESET** section) while **seed** remains unchanged. If the bound is larger than **DC_RAND_MAX**, the higher bound is honored by generating several pseudo-random integers, multiplying them by appropriate powers of **DC_RAND_MAX+1**, and adding them together. Thus, the size of integer that can be generated with this command is unbounded. Using this command will change the value of **seed**, unless the operand is **0** or **1**. In that case, **0** is pushed onto the stack, and **seed** is *not* changed.

The generated integer is made as unbiased as possible, subject to the limitations of the pseudo-random number generator.

This is a **non-portable extension**.

Stack Control

These commands control the stack.

c Removes all items from ("clears") the stack.

d Copies the item on top of the stack ("duplicates") and pushes the copy onto the stack.

r Swaps ("reverses") the two top items on the stack.

R Pops ("removes") the top value from the stack.

Register Control

These commands control registers (see the **REGISTERS** section).

s*r* Pops the value off the top of the stack and stores it into register *r*.

l*r* Copies the value in register *r* and pushes it onto the stack. This does not alter the contents of *r*.

S*r* Pops the value off the top of the (main) stack and pushes it onto the stack of register *r*. The previous value of the register becomes inaccessible.

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

Parameters

These commands control the values of **ibase**, **obase**, **scale**, and **seed**. Also see the **SYNTAX** section.

i Pops the value off of the top of the stack and uses it to set **ibase**, which must be between **2** and **16**, inclusive.

If the value on top of the stack has any *scale*, the *scale* is ignored.

o Pops the value off of the top of the stack and uses it to set **obase**, which must be between **0** and **DC_BASE_MAX**, inclusive (see the **LIMITS** section and the **NUMBERS** section).

If the value on top of the stack has any *scale*, the *scale* is ignored.

k Pops the value off of the top of the stack and uses it to set **scale**, which must be non-negative.

If the value on top of the stack has any *scale*, the *scale* is ignored.

j Pops the value off of the top of the stack and uses it to set **seed**. The meaning of **seed** is dependent on the current pseudo-random number generator but is guaranteed to not change except for new major versions.

The *scale* and sign of the value may be significant.

If a previously used **seed** value is used again, the pseudo-random number generator is guaranteed to produce the same sequence of pseudo-random numbers as it did when the **seed** value was previously used.

The exact value assigned to **seed** is not guaranteed to be returned if the **J** command is used. However, if **seed** *does* return a different value, both values, when assigned to **seed**, are guaranteed to produce the same sequence of pseudo-random numbers. This means that certain values assigned to **seed** will not produce unique sequences of pseudo-random numbers.

There is no limit to the length (number of significant decimal digits) or *scale* of the value that can be assigned to **seed**.

This is a **non-portable extension**.

I Pushes the current value of **ibase** onto the main stack.

O Pushes the current value of **obase** onto the main stack.

K Pushes the current value of **scale** onto the main stack.

J Pushes the current value of **seed** onto the main stack.

This is a **non-portable extension**.

T Pushes the maximum allowable value of **ibase** onto the main stack.

This is a **non-portable extension**.

U Pushes the maximum allowable value of **obase** onto the main stack.

This is a **non-portable extension**.

V Pushes the maximum allowable value of **scale** onto the main stack.

This is a **non-portable extension**.

W Pushes the maximum (inclusive) integer that can be generated with the **'** pseudo-random number generator command.

This is a **non-portable extension**.

Strings

The following commands control strings.

dc(1) can work with both numbers and strings, and registers (see the **REGISTERS** section) can hold both strings and numbers. dc(1) always knows whether the contents of a register are a string or a number.

While arithmetic operations have to have numbers, and will print an error if given a string, other commands accept strings.

Strings can also be executed as macros. For example, if the string **[1pR]** is executed as a macro, then the code **1pR** is executed, meaning that the **1** will be printed with a newline after and then popped from the stack.

[*characters***]**

Makes a string containing *characters* and pushes it onto the stack.

If there are brackets (**[** and **]**) in the string, then they must be balanced. Unbalanced brackets can be escaped using a backslash (\) character.

If there is a backslash character in the string, the character after it (even another backslash) is put into the string verbatim, but the (first) backslash is not.

a The value on top of the stack is popped.

If it is a number, it is truncated and its absolute value is taken. The result mod **256** is calculated. If that result is **0**, push an empty string; otherwise, push a one-character string where the character is the result of the mod interpreted as an ASCII character.

If it is a string, then a new string is made. If the original string is empty, the new string is empty. If it is not, then the first character of the original string is used to create the new string as a one-character string. The new string is then pushed onto the stack.

This is a **non-portable extension**.

x Pops a value off of the top of the stack.

If it is a number, it is pushed back onto the stack.

If it is a string, it is executed as a macro.

This behavior is the norm whenever a macro is executed, whether by this command or by the conditional execution commands below.

>*r* Pops two values off of the stack that must be numbers and compares them. If the first value is greater than the second, then the contents of register *r* are executed.

For example, **0 1>a** will execute the contents of register **a**, and **1 0>a** will not.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

>*r***e***s*
Like the above, but will execute register *s* if the comparison fails.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

This is a **non-portable extension**.

!>*r* Pops two values off of the stack that must be numbers and compares them. If the first value is not greater than the second (less than or equal to), then the contents of register *r* are executed.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

!>*r***e***s*
Like the above, but will execute register *s* if the comparison fails.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

This is a **non-portable extension**.

<*r* Pops two values off of the stack that must be numbers and compares them. If the first value is less than the second, then the contents of register *r* are executed.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

<*r***e***s*
Like the above, but will execute register *s* if the comparison fails.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

This is a **non-portable extension**.

!<*r* Pops two values off of the stack that must be numbers and compares them. If the first value is not less than the second (greater than or equal to), then the contents of register *r* are executed.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

!<*r***e***s*
Like the above, but will execute register *s* if the comparison fails.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

This is a **non-portable extension**.

=*r* Pops two values off of the stack that must be numbers and compares them. If the first value is equal to the second, then the contents of register *r* are executed.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

=*r***e***s*
Like the above, but will execute register *s* if the comparison fails.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

This is a **non-portable extension**.

!=*r* Pops two values off of the stack that must be numbers and compares them. If the first value is not equal to the second, then the contents of register *r* are executed.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

!=*r***e***s*
Like the above, but will execute register *s* if the comparison fails.

If either or both of the values are not numbers, dc(1) will raise an error and reset (see the **RESET** section).

This is a **non-portable extension**.

? Reads a line from the **stdin** and executes it. This is to allow macros to request input from users.

q During execution of a macro, this exits the execution of that macro and the execution of the macro that executed it. If there are no macros, or only one macro executing, dc(1) exits.

Q Pops a value from the stack which must be non-negative and is used the number of macro executions to pop off of the execution stack. If the number of levels to pop is greater than the number of executing macros, dc(1) exits.

, Pushes the depth of the execution stack onto the stack. The execution stack is the stack of string executions. The number that is pushed onto the stack is exactly as many as is needed to make dc(1) exit with the **Q** command, so the sequence **,Q** will make dc(1) exit.

This is a **non-portable extension**.

Status

These commands query status of the stack or its top value.

Z Pops a value off of the stack.

If it is a number, calculates the number of significant decimal digits it has and pushes the result. It will push **1** if the argument is **0** with no decimal places.

If it is a string, pushes the number of characters the string has.

X Pops a value off of the stack.

If it is a number, pushes the *scale* of the value onto the stack.

If it is a string, pushes **0**.

u Pops one value off of the stack. If the value is a number, this pushes **1** onto the stack. Otherwise (if it is a string), it pushes **0**.

This is a **non-portable extension**.

t Pops one value off of the stack. If the value is a string, this pushes **1** onto the stack. Otherwise (if it is a number), it pushes **0**.

This is a **non-portable extension**.

z Pushes the current depth of the stack (before execution of this command) onto the stack.

y*r* Pushes the current stack depth of the register *r* onto the main stack.

Because each register has a depth of **1** (with the value **0** in the top item) when dc(1) starts, dc(1) requires that each register's stack must always have at least one item; dc(1) will give an error and reset otherwise (see the **RESET** section). This means that this command will never push **0**.

This is a **non-portable extension**.

Arrays

These commands manipulate arrays.

:*r* Pops the top two values off of the stack. The second value will be stored in the array *r* (see the **REGISTERS** section), indexed by the first value.

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

Y*r* Pushes the length of the array *r* onto the stack.

This is a **non-portable extension**.

Global Settings

These commands retrieve global settings. These are the only commands that require multiple specific characters, and all of them begin with the letter **g**. Only the characters below are allowed after the character **g**; any other character produces a parse error (see the **ERRORS** section).

gl Pushes the line length set by **DC_LINE_LENGTH** (see the **ENVIRONMENT VARIABLES** section) onto the stack.

gx Pushes **1** onto the stack if extended register mode is on, **0** otherwise. See the *Extended Register Mode* subsection of the **REGISTERS** section for more information.

gz Pushes **0** onto the stack if the leading zero setting has not been enabled with the **-z** or **--leading-**

zeroes options (see the **OPTIONS** section), non-zero otherwise.

REGISTERS

Registers are names that can store strings, numbers, and arrays. (Number/string registers do not interfere with array registers.)

Each register is also its own stack, so the current register value is the top of the stack for the register. All registers, when first referenced, have one value (**0**) in their stack, and it is a runtime error to attempt to pop that item off of the register stack.

In non-extended register mode, a register name is just the single character that follows any command that needs a register name. The only exceptions are: a newline (**'\n'**) and a left bracket (**'['**); it is a parse error for a newline or a left bracket to be used as a register name.

Extended Register Mode

Unlike most other dc(1) implentations, this dc(1) provides nearly unlimited amounts of registers, if extended register mode is enabled.

If extended register mode is enabled (**-x** or **--extended-register** command-line arguments are given), then normal single character registers are used *unless* the character immediately following a command that needs a register name is a space (according to **isspace()**) and not a newline (**'\n'**).

In that case, the register name is found according to the regex **[a-z][a-z0-9_]*** (like bc(1) identifiers), and it is a parse error if the next non-space characters do not match that regex.

RESET

When dc(1) encounters an error or a signal that it has a non-default handler for, it resets. This means that several things happen.

First, any macros that are executing are stopped and popped off the stack. The behavior is not unlike that of exceptions in programming languages. Then the execution point is set so that any code waiting to execute (after all macros returned) is skipped.

Thus, when dc(1) resets, it skips any remaining code waiting to be executed. Then, if it is interactive mode, and the error was not a fatal error (see the **EXIT STATUS** section), it asks for more input; otherwise, it exits with the appropriate return code.

PERFORMANCE

Most dc(1) implementations use **char** types to calculate the value of **1** decimal digit at a time, but that can be slow. This dc(1) does something different.

It uses large integers to calculate more than **1** decimal digit at a time. If built in a environment where **DC_LONG_BIT** (see the **LIMITS** section) is **64**, then each integer has **9** decimal digits. If built in an environment where **DC_LONG_BIT** is **32** then each integer has **4** decimal digits. This value (the number of decimal digits per large integer) is called **DC_BASE_DIGS**.

In addition, this dc(1) uses an even larger integer for overflow checking. This integer type depends on the value of **DC_LONG_BIT**, but is always at least twice as large as the integer type used to store digits.

LIMITS

The following are the limits on dc(1):

DC_LONG_BIT
The number of bits in the **long** type in the environment where dc(1) was built. This determines how many decimal digits can be stored in a single large integer (see the **PERFORMANCE** section).

DC_BASE_DIGS
The number of decimal digits per large integer (see the **PERFORMANCE** section). Depends on **DC_LONG_BIT**.

DC_BASE_POW
The max decimal number that each large integer can store (see **DC_BASE_DIGS**) plus **1**. Depends on **DC_BASE_DIGS**.

DC_OVERFLOW_MAX
The max number that the overflow type (see the **PERFORMANCE** section) can hold. Depends on **DC_LONG_BIT**.

DC_BASE_MAX
The maximum output base. Set at **DC_BASE_POW**.

DC_DIM_MAX
The maximum size of arrays. Set at **SIZE_MAX-1**.

DC_SCALE_MAX
The maximum **scale**. Set at **DC_OVERFLOW_MAX-1**.

DC_STRING_MAX
The maximum length of strings. Set at **DC_OVERFLOW_MAX-1**.

DC_NAME_MAX
The maximum length of identifiers. Set at **DC_OVERFLOW_MAX-1**.

DC_NUM_MAX
The maximum length of a number (in decimal digits), which includes digits after the decimal point. Set at **DC_OVERFLOW_MAX-1**.

DC_RAND_MAX
The maximum integer (inclusive) returned by the **'** command, if dc(1). Set at **2^DC_LONG_BIT-1**.

Exponent
The maximum allowable exponent (positive or negative). Set at **DC_OVERFLOW_MAX**.

Number of vars
The maximum number of vars/arrays. Set at **SIZE_MAX-1**.

These limits are meant to be effectively non-existent; the limits are so large (at least on 64-bit machines) that there should not be any point at which they become a problem. In fact, memory should be exhausted before these limits should be hit.

ENVIRONMENT VARIABLES

As **non-portable extensions**, dc(1) recognizes the following environment variables:

DC_ENV_ARGS
This is another way to give command-line arguments to dc(1). They should be in the same format as all other command-line arguments. These are always processed first, so any files given in **DC_ENV_ARGS** will be processed before arguments and files given on the command-line. This gives the user the ability to set up "standard" options and files to be used at every invocation. The most useful thing for such files to contain would be useful functions that the user might want every time dc(1) runs. Another use would be to use the **-e** option to set **scale** to a value other than **0**.

The code that parses **DC_ENV_ARGS** will correctly handle quoted arguments, but it does not understand escape sequences. For example, the string **"/home/gavin/some dc file.dc"** will be correctly parsed, but the string **"/home/gavin/some "dc" file.dc"** will include the backslashes.

The quote parsing will handle either kind of quotes, **'** or **"**. Thus, if you have a file with any number of single quotes in the name, you can use double quotes as the outside quotes, as in **"some 'dc' file.dc"**, and vice versa if you have a file with double quotes. However, handling a file with both kinds of quotes in **DC_ENV_ARGS** is not supported due to the complexity of the parsing, though such files are still supported on the command-line where the parsing is done by the shell.

DC_LINE_LENGTH

If this environment variable exists and contains an integer that is greater than **1** and is less than **UINT16_MAX** (**2^16-1**), dc(1) will output lines to that length, including the backslash newline combo. The default line length is **70**.

The special value of **0** will disable line length checking and print numbers without regard to line length and without backslashes and newlines.

DC_SIGINT_RESET

If dc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), then this environment variable has no effect because dc(1) exits on **SIGINT** when not in interactive mode.

However, when dc(1) is in interactive mode, then if this environment variable exists and contains an integer, a non-zero value makes dc(1) reset on **SIGINT**, rather than exit, and zero makes dc(1) exit. If this environment variable exists and is *not* an integer, then dc(1) will exit on **SIGINT**.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

DC_TTY_MODE

If TTY mode is *not* available (see the **TTY MODE** section), then this environment variable has no effect.

However, when TTY mode is available, then if this environment variable exists and contains an integer, then a non-zero value makes dc(1) use TTY mode, and zero makes dc(1) not use TTY mode.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

DC_PROMPT

If TTY mode is *not* available (see the **TTY MODE** section), then this environment variable has no effect.

However, when TTY mode is available, then if this environment variable exists and contains an integer, a non-zero value makes dc(1) use a prompt, and zero or a non-integer makes dc(1) not use a prompt. If this environment variable does not exist and **DC_TTY_MODE** does, then the value of the **DC_TTY_MODE** environment variable is used.

This environment variable and the **DC_TTY_MODE** environment variable override the default, which can be queried with the **-h** or **--help** options.

DC_EXPR_EXIT

If any expressions or expression files are given on the command-line with **-e**, **--expression**, **-f**, or **--file**, then if this environment variable exists and contains an integer, a non-zero value makes dc(1) exit after executing the expressions and expression files, and a zero value makes dc(1) not exit.

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

DC_DIGIT_CLAMP

When parsing numbers and if this environment variable exists and contains an integer, a non-zero value makes dc(1) clamp digits that are greater than or equal to the current **ibase** so that all such digits are considered equal to the **ibase** minus 1, and a zero value disables such clamping so that those digits are always equal to their value, which is multiplied by the power of the **ibase**.

This never applies to single-digit numbers, as per the bc(1) standard (see the **STANDARDS** section).

This environment variable overrides the default, which can be queried with the **-h** or **--help** options.

EXIT STATUS

dc(1) returns the following exit statuses:

0 No error.

1 A math error occurred. This follows standard practice of using **1** for expected errors, since math errors will happen in the process of normal execution.

Math errors include divide by **0**, taking the square root of a negative number, using a negative number as a bound for the pseudo-random number generator, attempting to convert a negative number to a hardware integer, overflow when converting a number to a hardware integer, overflow when calculating the size of a number, and attempting to use a non-integer where an integer is required.

Converting to a hardware integer happens for the second operand of the power (**^**), places (**@**), left shift (**H**), and right shift (**h**) operators.

2 A parse error occurred.

Parse errors include unexpected **EOF**, using an invalid character, failing to find the end of a string or comment, and using a token where it is invalid.

3 A runtime error occurred.

Runtime errors include assigning an invalid number to any global (**ibase**, **obase**, or **scale**), giving a bad expression to a **read()** call, calling **read()** inside of a **read()** call, type errors (including attempting to execute a number), and attempting an operation when the stack has too few elements.

4 A fatal error occurred.

Fatal errors include memory allocation errors, I/O errors, failing to open files, attempting to use files that do not have only ASCII characters (dc(1) only accepts ASCII characters), attempting to open a directory as a file, and giving invalid command-line options.

The exit status **4** is special; when a fatal error occurs, dc(1) always exits and returns **4**, no matter what mode dc(1) is in.

The other statuses will only be returned when dc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), since dc(1) resets its state (see the **RESET** section) and accepts more input when one of those errors occurs in interactive mode. This is also the case when interactive mode is forced by the **-i** flag or **--interactive** option.

These exit statuses allow dc(1) to be used in shell scripting with error checking, and its normal behavior can be forced by using the **-i** flag or **--interactive** option.

INTERACTIVE MODE

Like bc(1), dc(1) has an interactive mode and a non-interactive mode. Interactive mode is turned on automatically when both **stdin** and **stdout** are hooked to a terminal, but the **-i** flag and **--interactive** option can turn it on in other situations.

In interactive mode, dc(1) attempts to recover from errors (see the **RESET** section), and in normal execution, flushes **stdout** as soon as execution is done for the current input. dc(1) may also reset on **SIGINT** instead of exit, depending on the contents of, or default for, the **DC_SIGINT_RESET** environment variable (see the **ENVIRONMENT VARIABLES** section).

TTY MODE

If **stdin**, **stdout**, and **stderr** are all connected to a TTY, then "TTY mode" is considered to be available, and thus, dc(1) can turn on TTY mode, subject to some settings.

If there is the environment variable **DC_TTY_MODE** in the environment (see the **ENVIRONMENT VARIABLES** section), then if that environment variable contains a non-zero integer, dc(1) will turn on TTY mode when **stdin**, **stdout**, and **stderr** are all connected to a TTY. If the **DC_TTY_MODE** environment variable exists but is *not* a non-zero integer, then dc(1) will not turn TTY mode on.

If the environment variable **DC_TTY_MODE** does *not* exist, the default setting is used. The default setting can be queried with the **-h** or **--help** options.

TTY mode is different from interactive mode because interactive mode is required in the bc(1) specification (see the **STANDARDS** section), and interactive mode requires only **stdin** and **stdout** to be connected to a terminal.

Command-Line History

Command-line history is only enabled if TTY mode is, i.e., that **stdin**, **stdout**, and **stderr** are connected to a TTY and the **DC_TTY_MODE** environment variable (see the **ENVIRONMENT VARIABLES** section) and its default do not disable TTY mode. See the **COMMAND LINE HISTORY** section for more information.

Prompt

If TTY mode is available, then a prompt can be enabled. Like TTY mode itself, it can be turned on or off with an environment variable: **DC_PROMPT** (see the **ENVIRONMENT VARIABLES** section).

If the environment variable **DC_PROMPT** exists and is a non-zero integer, then the prompt is turned on when **stdin**, **stdout**, and **stderr** are connected to a TTY and the **-P** and **--no-prompt** options were not used. The read prompt will be turned on under the same conditions, except that the **-R** and **--no-read-prompt** options must also not be used.

However, if **DC_PROMPT** does not exist, the prompt can be enabled or disabled with the **DC_TTY_MODE** environment variable, the **-P** and **--no-prompt** options, and the **-R** and **--no-read-prompt** options. See the **ENVIRONMENT VARIABLES** and **OPTIONS** sections for more details.

SIGNAL HANDLING

Sending a **SIGINT** will cause dc(1) to do one of two things.

If dc(1) is not in interactive mode (see the **INTERACTIVE MODE** section), or the **DC_SIGINT_RESET** environment variable (see the **ENVIRONMENT VARIABLES** section), or its default, is either not an integer or it is zero, dc(1) will exit.

However, if dc(1) is in interactive mode, and the **DC_SIGINT_RESET** or its default is an integer and non-zero, then dc(1) will stop executing the current input and reset (see the **RESET** section) upon receiving a **SIGINT**.

Note that "current input" can mean one of two things. If dc(1) is processing input from **stdin** in interactive mode, it will ask for more input. If dc(1) is processing input from a file in interactive mode, it will stop processing the file and start processing the next file, if one exists, or ask for input from **stdin** if no other file exists.

This means that if a **SIGINT** is sent to dc(1) as it is executing a file, it can seem as though dc(1) did not respond to the signal since it will immediately start executing the next file. This is by design; most files that users execute when interacting with dc(1) have function definitions, which are quick to parse. If a file takes a long time to execute, there may be a bug in that file. The rest of the files could still be executed without problem, allowing the user to continue.

SIGTERM and **SIGQUIT** cause dc(1) to clean up and exit, and it uses the default handler for all other signals. The one exception is **SIGHUP**; in that case, and only when dc(1) is in TTY mode (see the **TTY MODE** section), a **SIGHUP** will cause dc(1) to clean up and exit.

COMMAND LINE HISTORY

dc(1) supports interactive command-line editing.

If dc(1) can be in TTY mode (see the **TTY MODE** section), history can be enabled. This means that command-line history can only be enabled when **stdin**, **stdout**, and **stderr** are all connected to a TTY.

Like TTY mode itself, it can be turned on or off with the environment variable **DC_TTY_MODE** (see the **ENVIRONMENT VARIABLES** section).

Note: tabs are converted to 8 spaces.

LOCALES

This dc(1) ships with support for adding error messages for different locales and thus, supports **LC_MESSAGES**.

SEE ALSO

bc(1)

STANDARDS

The dc(1) utility operators and some behavior are compliant with the operators in the IEEE Std 1003.1-2017 ("POSIX.1-2017") bc(1) specification at https://pubs.opengroup.org/onlinepubs/9699919799/utilities/bc.html .

BUGS

None are known. Report bugs at https://git.gavinhoward.com/gavin/bc .

AUTHOR

Gavin D. Howard <gavin@gavinhoward.com> and contributors.