Calculating with dc

Introduction

dc(1) is a non standard, but commonly found, reverse-polish Desk Calculator. According to Ken Thompson, "dc is the oldest language on Unix; it was written on the PDP-7 and ported to the PDP-11 before Unix [itself] was ported".

Historically the standard bc(1) has been implemented as a *front-end to dc*.

Simple calculation

In brief, the *reverse polish notation* means the numbers are put on the stack first, then an operation is applied to them. Instead of writing `1+1`, you write `1 1+`.

By default `dc`, unlike `bc`, doesn't print anything, the result is pushed on the stack. You have to use the "p" command to print the element at the top of the stack. Thus a simple operation looks like:

```
$ dc <<< '1 1+pq'
2
```

I used a "here string" present in bash 3.x, ksh93 and zsh. if your shell doesn't support this, you can use `echo '1 1+p' | dc` or if you have GNU `dc`, you can use `dc -e '1 1 +p'`.

Of course, you can also just run `dc` and enter the commands.

The classic operations are:

- addition: `+`
- subtraction: `-`
- division: `/`
- multiplication: `*`
- remainder (modulo): `%`
- exponentiation: `^`
- square root: `v`

GNU `dc` adds a couple more.

To input a negative number you need to use the `_` (underscore) character:

```
$ dc <<< '1_1-p'
2
```

You can use the *digits* `0` to `9` and the *letters* `A` to `F` as numbers, and a dot (`.`) as a decimal point. The `A` to `F` **must** be capital letters in order not to be confused with the commands specified with lower case characters. A number with a letter is considered hexadecimal:

```
dc <<< 'Ap'
10
```

The **output** is converted to **base 10** by default

Scale And Base

`dc` is a calulator with abitrary precision, by default this precision is 0. thus `dc <<< "5 4/p"` prints "1".

We can increase the precision using the `k` command. It pops the value at the top of the stack and uses it as the precision argument:

```
dc <<< '2k5 4/p' # prints 1.25
dc <<< '4k5 4/p' # prints 1.2500
dc <<< '100k 2vp'
1.4142135623730950488016887242096980785696718753769480731766797379907
\
324784621070388503875343276415727
```

dc supports *large* precision arguments.

You can change the base used to output (*print*) the numbers with `o` and the base used to input (*type*) the numbers with `i` :

```
dc << EOF
20 p# prints 20, output is in base 10
16o # the output is now in base 2 16
20p # prints 14,  in hex
16i # the output is now in hex
p   # prints 14 this doesn't modify the number in the stack
10p # prints 10 the output is done in base 16
EOF
```

Note: when the input value is modified, the base is modified for all commands, including `i` :

```
dc << EOF
16i 16o # base is 16 for input and output
10p # prints 10
10i # ! set the base to 10 i.e. to 16 decimal
17p # prints 17
EOF
```

This code prints 17 while we might think that `10i` reverts the base back to 10 and thus the number should be converted to hex and printed as 11. The problem is 10 was typed while the input base 16, thus the base was set to 10 hexadecimal, i.e. 16 decimal.

```
dc << EOF
16o16o10p #prints 10
Ai # set the base to A in hex i.e. 10
17p # prints 11 in base 16
EOF
```

Stack

There are two basic commands to manipulate the stack:

- `d` duplicates the top of the stack
- `c` clears the stack

```
$ dc << EOF
2   # put 2 on the stack
d   # duplicate i.e. put another 2 on the stack
*p  # multiply and print
c p # clear and print
EOF
4
dc: stack empty
```

`c p` results in an error, as we would expect, as c removes everything on the stack. *Note: we can use `#` to put comments in the script.*

If you are lost, you can inspect (i.e. print) the stack using the command `f`. The stack remains unchanged:

```
dc <<< '1 2 d 4+f'
6
2
1
```

Note how the first element that will be popped from the stack is printed first, if you are used to an HP calculator, it's the reverse.

Don't hesitate to put `f` in the examples of this tutorial, it doesn't change the result, and it's a good way to see what's going on.

Registers

The GNU `dc` manual says that dc has at least **256 registers** depending on the range of unsigned char. I'm not sure how you are supposed to use the NUL byte. Using a register is easy:

```
dc <<EOF
12 # put 12 on the stack
sa # remove it from the stack (s), and put it in register 'a'
10 # put 10 on the stack
la # read (l) the value of register 'a' and push it on the stack
+p # add the 2 values and print
EOF
```

The above snippet uses newlines to embed comments, but it doesn't really matter, you can use `echo '12sa10la+p'| dc` , with the same results.

The register can contain more than just a value, **each register is a stack on its own**.

```
dc <<EOF
12sa #store 12 in 'a'
6Sa # with a capital S the 6 is removed
    # from the  main stack and pushed on the 'a' stack
lap # prints 6, the value at the top of the 'a' stack
lap # still prints 6
Lap # prints 6 also but with a capital L, it pushes the value in 'a'
    # to the main stack and pulls it from the 'a' stack
lap # prints 12, which is now at the top of the stack
EOF
```

Macros

`dc` lets you push arbitrary strings on the stack when the strings are enclosed in `[]` . You can print it with `p` : `dc <<< '[Hello World!]p'` and you can evalute it with x: `dc <<< '[1 2+]xp'` .

This is not that interesting until combined with registers. First, let's say we want to calculate the square of a number (don't forget to include `f` if you get lost!):

```
dc << EOF
3 # push our number on the stack
d # duplicate it i.e. push 3 on the stack again
d**p # duplicate again and calculate the product and print
EOF
```

Now we have several cubes to calculate, we could use `dd**` several times, or use a macro.

```
dc << EOF
[dd**] # push a string
sa # save it in register a
3 # push 3 on the stack
lax # push the string "dd**" on the stack and execute it
p # print the result
4laxp # same operation for 4, in one line
EOF
```

Conditionals and Loops

`dc` can execute a macro stored in a register using the `lR x` combo, but it can also execute macros conditionally. `>a` will execute the macro stored in the register `a`, if the top of the stack is *greater than* the second element of the stack. Note: the top of the stack contains the last entry. When written, it appears as the reverse of what we are used to reading:

```
dc << EOF
[[Hello World]p] sR  # store in 'R' a macro that prints Hello World
2 1 >R               # do nothing 1 is at the top 2 is the second ele
ment
1 2 >R               # prints Hello World
EOF
```

Some `dc` have `>R <R =R`, GNU `dc` had some more, check your manual. Note that the test "consumes" its operands: the 2 first elements are popped off the stack (you can verify that `dc <<< "[f]sR 2 1 >R 1 2 >R f"` doesn't print anything)

Have you noticed how we can *include* a macro (string) in a macro? and as `dc` relies on a stack we can, in fact, use the macro recursively (have your favorite control-c key combo ready ;)) :

```
dc << EOF
[ [Hello World] p   # our macro starts by printing Hello World
  lRx           ]   # and then executes the macro in R
sR                  # we store it in the register R
lRx                 # and finally executes it.
EOF
```

We have recursivity, we have test, we have loops:

```
dc << EOF
[ li       # put our index i on the stack
  p        # print it, to see what's going on
  1 -      # we decrement the index by one
  si       # store decremented index (i=i-1)
 0 li >L   # if i > 0 then execute L
] sL       # store our macro with the name L

10 si      # let's give to our index the value 10
lLx        # and start our loop
EOF
```

Of course code written this way is far too easy to read! Make sure to remove all those extra spaces newlines and comments:

```

dc <<< '[lip1-si0li>L]sL10silLx'
dc <<< '[p1-d0<L]sL10lLx' # use the stack instead of a register
```

I'll let you figure out the second example, it's not hard, it uses the stack instead of a register for the index.

Next

Check your dc manual, i haven't decribed everything, like arrays (only documented with "; : are used by bc(1) for array operations" on solaris, probably because *echo '1 0:a 0Sa 2 0:a La 0;ap'* | *dc* results in *Segmentation Fault (core dump)* , the latest solaris uses GNU dc)

You can find more info and dc programs here:

- http://en.wikipedia.org/wiki/Dc_(Unix) (http://en.wikipedia.org/wiki/Dc_(Unix))

And more example, as well as a dc implementation in python here:

- http://en.literateprograms.org/Category:Programming_language:dc (http://en.literateprograms.org/Category:Programming_language:dc)
- http://en.literateprograms.org/Desk_calculator_%28Python%29 (http://en.literateprograms.org/Desk_calculator_%28Python%29)

The manual for the 1971 dc from Bell Labs:

- http://cm.bell-labs.com/cm/cs/who/dmr/man12.ps (http://cm.bell-labs.com/cm/cs/who/dmr/man12.ps) (dead link)

Discussion

richard hartshorn, 2012/02/26 22:20 (), 2012/03/04 17:08 ()

I am using gnu dc dated 2006, playing with it really,, and the | operator gives different results…

```
dc -e "0 k 3 4 5 | p q"
```

gives 1, correct! (3^4) mod 5 is 81 mod 5 is 1.

The wobbly appears here…

```
dc -e "8 k 3 4 5 | p q"
```

gives zero, wrong! Even though the scale setting is irrelevant to the calculation. Bug do you think? Or a historic 'feature' everyone else is used to.

howto/calculate-dc.txt Last modified: 2018/06/21 23:36 by izxle

Bash Hackers Wiki