BogusForth v0.9.4
Language Reference Manual

Antonio Maschio
tbin@libero.it

January 2005

This is BogusForth, an obfuscated language based mainly upon forth (widely known), created by Charles Moore, and false, another obfuscated language by Wouter van Oortmerssen. Many ideas were also taken from TRUE, by Dewi. Therefore, thanks to Moore, Wouter and Dewi.

Using bf is easy, provided you have some knowledge of stack programming (forth) and/or RPN calculators, e.g. HP11C, HP12C or HP32SII (I got all of them).

If you have no stack programming experience, I plan to write a simple introduction about this subject. Check bf site from time to time.

This document was written with gvim *6.2, formatted (and corrected) with* LyX *1.3.1 +* latex *(Web2C 7.3.1) 3.14159 +* kpathsea *version 3.3.1 on a* Red-Hat 9A Linux box *patched with* Fedora Core 1.

Copyright Note: where Win32 is written, it must be interpreted as a Microsoft Windows® Operating System (any).

IMPORTANT: Language has slightly changed, from 0.8.6 to 0.9.4. See 5.1.

Contents

1 Introduction

Welcome to the BogusForth world. This is free software with ABSOLUTELY NO WARRANTY.

A few words to describe **bf** (rigorously NOT capitalized), an obfuscated language that aims to become something more than a simple play tool (as it is now, to tell the truth). The main points basing its philosophy are:

- the stack processing as the main programming tool (a **forth** philosophy that I love a lot)
- the key-char style tendency
- the minimal set of key-char target tendency: if something may be obtained by a combination of other commands, it is not a **bf** key-char or command.
- a correct math behavior[1]
- the *obfuscate-it* tendency

This makes **bf** a quite powerful language and interpreter, though simple and console-based. Of course, it's GPL, and its main distribution is by source files. If you are a professional programmer, probably you will find some naive and simple coding, but I got a real programming passion, and no school skills.

Anyway, as I use to say, it's GPL! Enjoy!

1.1 License

Read the GNU GENERAL PUBLIC LICENSE, Version 2, June 1991 or any following other, which is contained in the COPYING file into the package: **bf** comes under it.

1.2 First steps

1.2.1 Make

It's very easy to compile and install **bf**.

Compilation Execute the shell script *compile.sh*, from a shell (sh or bash); after its execution, you will find the **bf** program in the source directory:

```
$ ./compile.sh
```

If this shouldn't work, execute:

```
$ sh compile.sh
```

Installation This must be done in a two-parts process.

1) Installation of the Autoload File This must be done as user. Root won't install the autoload file into her/his home directory, since I presume root won't lose her/his time playing with **bf**. If she/he wants to, well, a hand-copy must be performed.

Simple execute one of the following:

```
$ ./install.sh
$ sh ./install.sh
```

The script will then inform you that the executable must be installed by root.

[1] For this matter, I due all my math rigorousness to [1.2], a real treasure for any aspect involved in getting correct mathematical results and programming. If you like the matter, it's a must.

2) Installation of the executable This must be done as root. First of all, assure that */usr/local/bin* is a correct location for you. If so, simply execute one of the following:

```
# ./install.sh
# sh install.sh
```

and **bf** will be available by anyone in your system. If your username is into the */etc/sudoers* file, you can execute

```
$ sudo ./install.sh
```

as user, give your password and install it, with no need to call or become root[2].

If you want another installation directory, you can do it so:

```
# ./install.sh your_dir
```

where *your_dir* is the directory where you want the executable to reside.

Of course, you can manually copy or move the executable in the location you want into your path (e.g. into *$HOME/bin*), if you're not root. **bf** doesn't depend on any particular file, neither library nor autoload file (a gentle advice will appear if no autoload file is found and quiet option is not set, but this won't affect any operation).

1.2.2 Usage

(from the `bf -h` execution:)

```
Usage: bf <options> <filename>
OPTIONS
-c          interpret bf filename leaving C file filename.c
                *** NOT IMPLEMENTED YET ***
-e          turn off error messages (ON by default)
-h, --help  give this help and exit
--license   show license type (GPL) and exit
-o          compile bf filename leaving an executable with the same name
                *** NOT IMPLEMENTED YET ***
-q          turn on quiet mode (OFF by default)
-s          turn on stack clearing on errors (OFF by default)
-V          show a quick language summary in continuous mode and exit
-v, --ref   show a quick language summary by screens and exit
--version   show version and exit
-w          turn off exit warning about stack not empty (ON by default)
--warranty  show warranty (none, of course) and exit
(Options are not cumulative - set them one by one)
With no filename, start in interactive mode.
If filename has no closing instructions (furnished through 0q)
fall back in interactive mode after interpreting file.
```

1.2.3 Starting options

The following options are accepted by **bf** (other unrecognized options are refused and will cause the interpreter to exit prematurely):

-c execute *filename* leaving **C** file *filename.c*. Feature not implemented yet, destined to be written starting from the 1.x versions (maybe). Probably the first version will be a simple skeleton-based version of *bf.c*, with `main()` containing a series of `parseString()` instructions. This is surely easy to implement, but not fast at all. Next step will be certainly obtaining assembler code to be compiled directly. It's a hard step, though, since I don't know assembler programming yet. If *filename* is null, this option is set off.

[2]Beware, though, since this practice is not much sure in a network-based system, as it may facilitate exploitations and attacks.

-e don't show error messages. This option turns on the suppressing of all error messages. It's destined to be used only during execution of well-tested software, though, since error is always behind the corner. By default this option is set off, and may be set during a program execution by the command `errorshow`.

-h –help show the above help menu and exit. Trivial.

–license show the declaration that GPL is the license of this software, and where to get it if not in the package, and exit.

-o compile *filename* leaving an executable (an exe in Win32). Not implemented. See -c.

-q quiet mode. This option turns off verbosity and doesn't print welcome nor any closing messages; it may be used for execution of software with its own messages. By default this option is set off, and may be set during a program execution by the command `quiet`.

-s turn on stack clearing after errors. By default, **bf** doesn't clear the stack after an error occurs. But if you want, in a forth style, you can have this option turned on. By default this option is set off, and may be set during a program execution by the command `clearstack`.

-V show vocabulary in continuous mode and exit. This option lets the user to obtain a quick reference guide. Simple digit

```
./bf -V > quick.txt
```

to get an updated quick reference guide, in pure ASCII text format.

-v –ref show a quick summary of available commands by screens and exit. This option is equivalent to the `ref` command into the **bf** interpreter.

–version show a little banner with the version number and exit.

-w turn off exit warning about a non-empty stack. I know that this may upset someone: I prefer to be warned, when I exit with a not cleared stack (I may have forgot something, since I tend to clear it by correct instructions, the same I do in forth). But if you get sick of it, turn it off. By default this option is set on, and may be set off during a program execution by the command `exitwarn`.

–warranty show a little banner with the warranty (none, of course), and exit .

You cannot cumulate options. Not yet. So you cannot type:

```
./bf -qew
```

to get the three options done, but:

```
./bf -q -e -w
```

If a filename is given, execution parses such file (one only at a time[3]), as if each line was digited on keyboard. This file may have a closing command (such as `3q` or `quit`), and then, after execution, the user will get back to the Unix/Win32 console. If no closing command is given, then **bf** will fall back into interactive mode, with the stack containing the results of the execution(s).

[3] If you need loading multiple files, write a `loader.bf` (or any other name) which contains a series of include, like:

```
"first.bf"(
"second.bf"(
...
"last.bf"(
```

and then load it with

```
./bf -q loader.bf
```

(or any other options you need).

1.3 Control Constants

The following section deals with arguments destined to those who will compile **bf**, willing to alter its default state.

The file *bf.h* sets up many variables that control the **bf** compilation process. If you change anything, of course, you must recompile then.

Let's review in full the ones you can change:

BUFFERNAME sets the name of the buffer file name. Set it to the name you prefer, possibly with the *.bf* extension.

PROMPT defines the prompt string to be printed. Default is `"> "` (greater than + blank char). Change it if you want another prompt.

DIRCOM defines the command to be executed as `ls` (or `dir` under Win32). By default it's `ls --color` under Unix and `dir /p /OGN` under Win32; change if you want to add some more options or use another command.

ALBF sets the name of the autoload file (which is *.albf* for Unix and *_albf* for Win32). See 1.2.1 for news about the way to install it, and read 1.5 for news about the autoload file. You can change its name to any one you want. The name, though, shouldn't contain relative paths such as *~/.mybfal*: they won't work.

RCNAME controls the name of the resources for **bf**. This option has not been implemented yet, so you can leave it as it is. The resources file name is *.bfrc* for Unix and *_bfrc* for Win32.

BFEDITOR sets the name of the preferred editor. A lot of editors are listed in *bf.h*: choose the one you prefer (commenting/uncommenting appropriately) or insert the one you want and recompile. By default `vi` is the editor set for Unix and `write` the one for Win32.

PAGER sets the name of the pager to see text files content. By default `less` is the one for Unix and for Win32 (provided you have less.exe). But again, more pagers are listed. Simply choose the one you prefer and recompile.

STACKMAX controls the depth of the stack. Its value is set to 256, which permits 255 stack positions[4]. Change it if you need more space.

RSTACKMAX controls the depth of the return stack[5]. Its value is set to 16. You can set it to any depth you want.

MAXWORDLENGTH controls the length of the defined words (see 3.12 and 2.3). Its value is set to 32 (for a length of 32 characters plus the terminator `'\0'`).

MAXWORDS sets the maximum number of words that `def` can define (or the depth of the added vocabulary). Its value is set to 512[6].

MAXLINE controls the dimension of the input string. It's value is 257 (256 for data, 1 for terminator character `'\0'`) by default. This leads to the following table:

input string	MAX LENGTH
command line	`MAXLINE`**-1 (default 256 chars)**
string `"s"`	`MAXLINE`**-2 (default 255 chars)**
function `[f]`	`MAXLINE`**-2 (default 255 chars)**

[4]Location 0 is unused. If `TOS` reaches 0, it means there is no value on the stack. I know it's a waste of memory.

[5]It's a great effort of imagination thinking it as a return stack, since it is used only to store temporarily objects. Its name should be 'second' or 'shareable' stack, or at least 'help' stack. But since I'm used to think in **forth** terms, that's it.

[6]Many **forth** vocabularies reach a number of 2000, 3000 words. If you think **MAXWORDS** is underestimated, change it.

INF defines a huge value to represent infinity. It is defined as 2^{1024} and, when invoked, the interpreter puts value *inf* on `TOS`. Of course it is not infinity, but any operation on it leaves it as *inf*, except that with *nan* (see also 2.1.2). Redefine it if you find a better way to represent it.

MINF defines a huge negative value to represent negative infinity. It is defined as -3^{1024} and the interpreter shows the symbol *-inf*. The same considerations expressed with **INF** apply here. Redefine it if you find a better way to represent it.

NAN defines what is an indefinite result in a math operation, called *nan* (Not A Number). It is defined as *inf/inf*. Redefine it if you find a better way to represent it[7].

PRECISION defines the precision value to represent floats in output. It is set to the maximum for doubles in C, which is 16. You can safely only reduce it; if you increase it, what you will see, after the 16th digit, is rubbish[8].

M_PHI and the physical constants are taken from [2.2] and [2.3]. Change them only if you have more precise values[9].

The other constants and `#define`s shouldn't absolutely be touched, unless you know what you are doing.

1.4 The Win32 definition

Win32 OSes are all executed with the WIN32 variable set. This means that the compilation under Win32 or Unix is different, in terms of interpreters and environment. In particular, `snprintf()` and `getcwd()` are known under my win32 gcc interpreter as `_snprintf()` and `_getcwd()`. In second place, tools like `less` are not common under Windows. So Win32 users are advised to download *less.exe* from the **bf** download page and install it. Some minor differences involve the `dir` and `edit` commands.

If you compile **bf** under Unix, no surprises.

If you compile **bf** under Win32, instead of using the precompiled version, you may make sure if you have `djgpp` or `minsys` installed. So, *ls.exe* and *less.exe* could already be present. Even *vi.exe*, I wish. In this case you could safely uncomment the `#undef WIN32` line in *bf.h*, recompile and gain the power of a Unix environment (though under Windows).

1.5 The autoload file

At the start, **bf** looks for a file whose name is contained into the **ALBF** definition (see 1.3), and executes it. Such a file is a collection of **bf** commands. You can:

- store values or functions in particular variables;
- define commands to easy your session;
- print strings that will be shown at start;
- put specific values on the stack;
- see a file (command `show`);
- set active debug option (command `debug`);

[7]Your interpreter may have NAN already defined. In this case, the *bf.h* files retains original definition.

[8]Try this, setting PRECISION to, say, 64:

```
> 2.0 3/64k.
```

[9]Of course, if you find a mistake, write me immediately.

- set active write-to-buffer-file option (command `]`);
- do any other thing **bf** can do. See it as a *.bashrc* or an *autoexec.bat* file.

Autoload file may not be present. In this case, a gentle advice will be printed, if not in quiet mode.

1.6 The interpreter

The interpreter accepts strings as input, strings which define a particular series of instructions. Input strings are evaluated from left to right, just like in **forth**.

Any line is 256 characters long; this is not a limitation since any command, using the stack, may reside on a single line, and the result would be the same as multiple commands in a line.

1.7 The stacks

Two stacks form the basis of the **bf** environment. The **data stack** , which is the normal stack for storing any value for current use, and a second helpful stack that I call **return stack**. The latter, in particular, is not really a return stack. It's not used by the system to store temporarily indexes and calling addresses. It simply helps in performing operations with `~>`, `~<` and `~:`[10] (see 3.11).

1.8 The buffered input

If you input a command line longer than 256 chars, the first 256 will be set as the input line. And the rest? Keep quiet: since the input is buffered, they are simply set as the second input line (and if it exceeds 256, the rest will be seen as the third and so on).

Beware, though! Commands are single chars, and may be divided, but numbers, strings, comments, system commands and functions are not, so they may get split if input strings are beyond the limit. For practical and secure data, limit input lines to be not longer than the screen width (typically 80 chars).

1.9 Notation

Every following key-chars and command is described with classic **forth** notation, that is the stack content before the command execution, a dash (which may represent the command itself, or its position) and the stack content after the command execution, all enclosed into parenthesis:

```
( xxx - yyy )
```

If nothing appears before the dash, nothing is required for the command execution, but the command itself produces some data:

```
( - yyy )
```

If nothing appears after the dash, the command consumes the needed values and (apparently) doesn't produce any more value:

```
( xxx - )
```

If nothing appears before and after the dash, the command is an information command, that is, its execution is (apparently) self sufficient:

```
( - )
```

The following notation stack values are used:

[10] Which are the parallel of the **forth**'s `R>`, `>R` and `R@`.

`n`	a number (a flag, a char, an integer or a double)
`f`	a flag (a True/False value and in general any integer)
`c`	a char (in general, any integer in the ASCII range 0-255)
`i`	an integer (a zero may be signaled directly)
`d`	a double
`"s", "f"`	a string (the second stands for a file name)
`[f]`	a function
`a`	(all) a number, a string or a function
`v`	a variable
`x*a, y*a`	an undefined number of data
`<stream>`	the characters following the command (the input stream line).

Possibly, an index may be postponed to the value, as in `n1` or `i2`, to distinguish different values.

In special cases, more representative letters are used, widely understood. E.g., see `~d,` in which the year is reported as `yyyy`.

Finally, you may meet the words `TOS` (Top Of Stack) and `EOL` (End Of [input] Line).

2 Entities

Here are the *objects* of the **bf** language.

2.1 Numbers

Available number types are **integers** (with values from $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$, a range with limits around $\pm 2.147 \times 10^9$) and **floats** (with values from -2^{1023} to 2^{1023}, a range with limits around $\pm 89.88 \times 10^{306}$).

Integers may be input simply by their digits:

```
> 345
```

Floats must be input with a dot, which must not be a leading character; so

```
> 2.
```

will work as 2.0, but

```
> .2
```

won't (will act as CR + 2). To introduce .2 digit

```
> 0.2
```

Negative numbers must be input with \ after them (not necessarily immediately after), as in

```
> 345\ 2.3 \
```

This will store -345 and -2.3.

2.1.1 Internal representation of numbers

bf has an improved number treating. Now integers are real integer and floats are real floating point values. So now, the `v` command really turns an integer into a float, possibly without losing data; the opposite loses all the decimal digits, and the number may exceed integer 32 bit capacity[11].

Floating point numbers are built with the double **C** type. This means that[12] a number will be represented with 64 bits (eight bytes); for i86 and SPARC platforms, this means that you can reach the value of $\pm 2^{1023}$ for a signed double, with 16 decimals[13]. This affects the float representation (see 1.3) to display not more than 16 digits for a float.

[11] Here is a session illustrating this:

```
> 2.0 534^
> :
----------------
  Stack output
      Top
562364224317899547851317313460747732358712139787739579137594446576479697583
935983789880085762986357143740113829190111890401033256936787461126064397608
14548189184.000000
     Bottom
----------------
> v
> :
----------------
  Stack output
      Top
-2147483648
      Bottom
----------------
```

[12] See [2.2].

[13] I often ask myself if long doubles (and long integers) could be more useful. My answer, until now, is no. On

2.1.2 Special numbers

Some peculiar floating point numbers may be introduced in a special way[14]:

- the π constant as `3..14`
- the Euler constant e as `2..71`
- the golden ratio constant as `1..61`
- the light speed constant as `2..99` (in m/s)
- the gravity acceleration constant as `9..80` (in m/s^2)
- the gravitational constant as `6..67` (in $m^3/kg\ s^2$)
- the parsec unity as `3..08` (in m)
- the light year unity as `9..46` (in m)
- the astronomical unity as `1..49` (in m)[15]
- the $+\infty$ (*inf*) constant as `0..00`
- the $-\infty$(-inf) constant as `00..0`
- the *nan* constant as `0...0`

Notice:
a) the two (or three) dots;
b) that if you input, say, `3.14`, the interpreted number will be `3.14`, not π;
c) that `3..1415` (another similar approximation) won't work (the interpreted value will be `3.0`). Only what indicated (a digit, two dots and two digits) will be interpreted as a command to push the corresponding special value onto the stack.

The *inf/nan* numbers: With *nan*, *inf* and *-inf* you can do some pretty things. It may be interesting to notice that

```
> 0..00 00..0 *  { inf * (-inf) }
```

returns *-inf*, as it should be (it's a peculiar multiplication, in which the sign is retained). On the other hand:

```
> 0..00 0...0 *  { inf * nan }
```

returns *nan* (it means that the property Not A Number conditions ANY number, including *inf*). Suppose now you have a math result you want to check (you may think it's too big); then, supposing the result is into R:

```
> R@ 0..00 =["Infinite"i.]?
```

will print the string (or do whatever you want) if R is infinite.

Notice that the same is not applicable to *nan*. The test

i86 platforms, while I could reach a signed value of 2^{16383}(!), it could be represented with 20 decimals, which isn't a great gain, against the 16 of doubles. It would be different for SPARC machines, on which the decimals would be 34. But how many of you work on SPARC? If one (1 only) of you works on SPARC and WANTS absolutely long doubles, I'll introduce them. Just ask.

[14] See [2.3] and [2.4].

[15] Values are respectively 1.61803398874989484820 (golden ratio), $2.99792458x10^8$ m/s (light speed), 9.80665 m/s^2 (gravity acceleration), $6.67390x10^{-11}$ $m^3/(kg * s^2)$ (gravitational constant), $3.086x10^{16}$ m (1 parsec), $9.463x10^{15}$ m (1 light year), $1.496x10^{11}$ m (astronomical unit).

```
> 0...0 0...0 =
```

gives 0 (False) as result, so you cannot check if a number is *nan*. But of course you can print it. If R contains a *nan*:

```
> R@i.
```

will print *nan*.

On Win32, **bf** returns `1.#QNAN0`, `1.#INF00` and `-1.#INF00` respectively for *nan*, *inf* and *-inf*.

Finally, notice the design behind the *nan* and *-inf*/*inf* pattern representation. `00..0` may be seen as ∞`..0` (the left branch of the x-axis); at the far 'bottom' of the left x-axis there is $-\infty$. Analogously, `0..00` may be seen as `0..`∞ (the right branch of the x-axis), with $+\infty$ at the far 'bottom'[16]. `0...0` is quite a mixture between the two, and symmetric.

2.2 Variables

bf has variables, though of a simple kind. They can host numbers, strings or functions, and they are completely untyped.

Variables are identified by single letters ($a \div z$ and $A \div Z$), but notice that a and A are distinct variables, so the programmer may use 26 x 2 = 52 variables. Not much, but a wise use of the stack may help (see 3.1).

At start, variables are initialized to integer 0. Anyway, initialize them by yourself (it may happen that a variable has to contain a string, e.g. for file treatment; in this case a number won't fit).

2.2.1 Allocated space

The space allocated by variables corresponds to an array with index from 0 to the ASCII value 'z', that is 122. But only positions corresponding to alphabetic chars (from 65 to 90 and from 97 to 122) are used. The wasting is due to the following facts:

- it's easier to manage one single index;
- it will be possible in the future (in a way not yet clear to me) to use more variables, so the structure, at that point, will be ready.

In the meantime, consider that this approach makes my duty easier.

2.3 Definitions

Definitions are the last component brought to the **bf** language. They are created with the `def` instruction:

```
> def <name> <instructions>
```

Here `<name>` is any valid string and `<instructions>` are a series of **bf** commands. For example (a trivial one):

```
> def DUP % "Duped!"i.
```

will define a `DUP` command that when encountered will `%` and print a message. `def` may be also use to create constants:

```
> def gravity 9..80
```

[16] I'm particularly proud of these patterns.

defines a constant gravity, that when executed will put the exact gravity value (in the International System unities) on TOS.

Of course, a definition may be introduced into another definition:

```
> def DUPDUP DUP DUP
```

See the command def at 3.12 for more about definitions.

2.4 Strings

Strings are input enclosing them into double quotes "..." (<stream> - "s"). If you omit the closing ", everything from the first " up to EOL will be a string. If you need to input a double quote char into your string as itself, escape it with a \ before it, like in

```
"Call me \"Hugo\", pliiz"
```

Strings are 255 chars long at max (that is, if you start them with the " and go on typing until the input line is longer than the limit - see 1.8). This is because of the first double quote char (see also 1.3). Besides, when you input a string, the enclosing double quote chars won't be counted in the string length.

2.4.1 A strong string advice

If you unify two strings (through u, see 3.10), only the first 255 of the union will be given. The rest will be lost. So be careful in joining long strings.

Anyway, don't despair! I have never used such long strings... have you?

2.5 Functions

Functions can host a complete command line, or a simple amount of data and are input enclosing them into square brackets [...] (<stream> - [f]). If you omit the closing], everything from [up to EOL will be the function.

Functions may be 255 char long at max (that is, if you start them with the [and go on typing until the input line is longer than the limit). This is because of the enclosing first [char (see 1.3).

In a certain and a proper sense, functions are strings that can be executed; you may find useful storing general data into functions. You're free. But take care not to execute that functions, of course!

2.5.1 A functional function advice

Functions may be nested, but they must reside on a single line. If you omit the last], everything from the first [to the end of line will be contained into that function. But beware, this is valid only for the first (outermost) function. Don't let an inner function be without its closing][17].

2.6 Comments

Comments are input enclosing them into curly brackets {...}. If you omit the last }, everything from the first { up to the end of line will be a string (this feature is similar to the // comment of **C++**), but beware: any opening { after the first will be ignored, since only the first } found will close the comment. So

```
{ ... { ... { ... }
```

is a closed comment (in a certain sense, it's a series of nested comments, isn't it?), but

[17] Though sometimes not closing even the innermost functions may work the same, I don't consider a good practice to do it, unless you are sure of the result, and want a *great* obfuscation!

```
{ ... { ... } xxx }
```

is not closed with the last `}`, since `xxx` and `}` (which is a standalone command, see stacks commands) will be executed;

```
{ ... { ... { ... xxx
```

is instead a perfectly valid open comment.

2.6.1 Capitalized comments

Capitalized text is parsed and discarded (all basic key-chars and command are lower case); this let the programmer use capitalized text as comment (assured that no other char is used). Don't get confused with variables, since they must be used with `!`, `:` or `@` just after them.

Be careful, though, since writing a comment like

```
THAT'S RIGHT!
```

causes `'` to be executed like a command (and to report an error) and `T!` to store `TOS` in T (and report an error if stack is empty). This comment should be written :

```
THAT IS RIGHT
```

or

```
{ THAT'S RIGHT! }
```

No problem instead if you want to write a number, like this:

```
LIMIT FOR A 32 BITS MACHINE;
```

This is a correct comment, since 32 is put on `TOS`, and then discarded it with `;`.

In all, I consider this feature a nice one.

2.7 Truth values

Tests treat only integer numbers as truth values, along the following, common table:

0 zero - False value.

1 one - True value.

Anyway, any non-zero value is True, like in **C**. Convert double to integer (see `v`) if you need to pass a double as a test value.

In the following sections, a *false* value will be reported as False, a *true* value as True (capitalized initials). On the other hand, false will mean the 'false' language interpreter and TRUE will mean the 'TRUE' language interpreter. Any other form is a mistake of mine.

2.8 Conclusion

Any other char than those compounding the language is discarded, so you may over-obfuscate your code by inserting any char here and there, confounding the reader (but not the interpreter, I hope!), as in the following:

```
ç23FR■45I■%**Di.
```

which performs 45*45*23 printing 46575, but in a very obscure way...

NOTE: throughout the whole book, I haven't put any command char into some quoting, like `')'` or `"$"`. First, because `'` and `"` are command by themselves, and second because in **bf** it needs to think about chars as entities. So you could find:

...assure to `e` before ...

... please, `%` and then `(` ...

This could be confusing, in the beginning, but for just a while.

3 The language

The **bf** language is composed (apart from a bunch of system commands) by key-chars (occasionally they may be linked to another char to complete their function). Only those contained into the base ASCII chart (33-127) are used, so that any one around the world may use it. But of course keyboards are not all equal, are they? So it may happen that some key is not available straight away. I reported here and there news on how to digit some special character based on my Italian experience (I have an Italian keyboard and a US keyboard, but I don't know about French, German, Greek and other keyboards). So, please, write me if you've found some difficulty to manage any key.

3.1 Variable treatment key-chars

The following key-chars must be put ***JUST*** after the variable name. If a blank or any other char is interposed, they will be treated as different key-char (see 3.4 and 3.5).

`!` exclamation mark `( a v - )`

store a value into variable v, as in

```
> 23f!
```

(will store 23 into f). `TOS` will be removed after that. Notice that no chars must be put between the variable and the `!` sign. Please, notice also that

```
> 2.3f ! { a space before ! }
```

will put 2.3 and the name of the buffer file on `TOS`. Then, `!` will try to get the square root out of `TOS`, failing.

`:` colon `( v - a )`

fetch variable v content, as in

```
> f:
```

(with reference to the previous example, will leave 23 on `TOS`). Again, notice that no chars must be put between the variable and the `:` sign.

`@` at `( v - x*a )`

execute function contained in variable v, as in:

```
> f@
```

of course, an error will be raised if f doesn't contain a function. Again, notice that no chars must be put between the variable and the `@` sign.

3.2 Functions execution key-char

The following key-char operates on functions. An error will be raised if something which is not a function is passed to it.

`@` at `( [f] - x*a )`

execute $[f]$, which may be left on `TOS` by a direct command, as in

```
> ["executed"i.]@
```

or, more useful, after a variable fetching, as in

```
> f:@.
```

Notice the different sequence with respect to the preceding definition of `@`. The two are equivalent, and in particular the first

```
> f@
```

is equivalent (a shortcut) to

```
> f:@
```

They produce the same result. This second form is maintained because more logically coherent. Notice that the restriction of no chars between the variable name and the `@` command doesn't apply here. While `:` pushes the variable content on `TOS`, `@` takes this values from the stack, and in the meantime you may have changed it, or performed other commands.

3.3 Control flow key-chars

The following key-chars rule the program flow, letting taking decisions or iterating operations[18].

`?` query `( f [f] | f [f1] [f2] - x*a )`

if-then/if-then-else control flow; in the first form:

```
> f[f]?
```

execute $[f]$ if f is True, in the second:

```
> f[f1][f2]?
```

execute $[f1]$ if f is True, otherwise $[f2]$; f must be a truth value (an integer value, interpreted as 0 or *not* 0 for False and True respectively).

`#` sharp `( [f1] [f2] - x*a )`

while cycle; while $[f1]$ returns True, execute $[f2]$. Of course, $[f1]$ must return any integer which may be interpreted as a truth value (0 for False, *not* 0 for True). For example:

```
> 100[%0>][%i.1-]#
```

prints all number from 100 to 1, one per line. The value 100 has been furnished to give $[f1]$ a first check, and to leave a value that $[f2]$ will use. This kind of parameters are often used with `#`.

`d` letter d `( n2 n1 [f] | [f1] [f2]- x*a )`

do-loop/do +loop cycle; in the first form:

```
> n2 n1[f]d
```

execute $[f2]$ $n2 - n1$ times (that is from $n1$ to $n2 - 1$), in the second:

[18] Versions up to 0.9.0 retained a "stop parsing" command (`t` or `z`, depending on versions, see 5.1). After realizing that this command could easily be replaced by `{` (that is *begin a comment just in that point*, as I do in C, generally), I decided to remove it from the language.

```
> n2 n1[f1][f2]d
```

execute $[f2]$ $n2-n1$ times, with the increment returned by $[f1]$, which is a very special function: it is executed only once, before the loop, and may set variables or print anything (it's a plain function, above all), but its main purpose is to leave a signed integer that will be taken as the loop increment. So

```
> 10 1[2][...]d
```

will step 2 (index will be 1, 3, 5, 7 and 9), and

```
> 10 1["Setting up loop"i.e:%w!t:+2/][:i.]d
```

will step $(e+t)/2$ times, after writing a short message and having stored e value into w.

```
> 1 10 [2\][...]d
```

will step -2 with a downward loop (index will be 10, 8, 6, 4, 2). Notice that $[f1]$ must return a not null signed integer (0 would produce an endless loop), or an error message will be displayed.

If the increment value is negative of backward loops or negative on positive ones, the loop won't even begin. No message is shown, of course, since this is not a loop mistake, but a programmer's choice (for example, indexes may be the result of some operation, leading to "strange" values).

`:` colon `( - i )`

if used inside a do-loop, leave index on `TOS`. Try this:

```
> 1 10[2\][:i.]d
```

will print all the loop indexes from 10 down to 1, step -2.

`h` letter h `( - )`

leave cycle; must be used into loop function ($[f2]$) of a do-loop, as in:

```
> 10 1[:%5=[h]?i.]d
```

this loop will step from 1 to 10, but if the index will be found equal to 5 (and it will be, since step is 1), the program will halt. Notice also that execution is performed the same (that is, 5 is printed), because a stopping flag is set while the string is being parsed. If you wish not to print the current index, you should stop a step before (that is detect 4 rather than 5).

Of course, `h` will work under a while cycle too, as in:

```
> 1[%11>~][%5=[h]?%i.1+]#
```

which produces the same effect of the previous example[19].

[19] See the ex.bf example provided in the bf package. It contains some interesting loop types, for nesting and interruptions schemes.

3.4 Stack managing key-chars

The following key-chars (quite all derived from **forth**, of course) alter the stack content in some way[20]. Some of them are duplication of more general key-chars, like `r` and `p`. The reason I maintain such duplications, apart from historical reasons, is that these shortcuts are *faster*.

`%` percent `( a - a a )`

dup; duplicate `TOS`. It is a shortcut to `1p` (1 pick).

`;` semi-colon `( a - )`

drop; remove `TOS`.

`$` dollar `( a1 a2 - a2 a1 )`

swap; exchange the two values on `TOS` (numbers, strings or functions). It is a shortcut to `2r` (2 roll).

`_` underscore `( a3 a2 a1 - a2 a1 a3 )`

rot; rotate the three values on `TOS` (numbers, strings of functions), bringing third topmost value on `TOS`. It a shortcut to `3r` (3 roll).

`r` letter r `( x*a i - y*a )`

roll, with control over limits. Rotate i^{th} element of stack.

`0r` does nothing; `1r` does nothing; `2r` = `$`; `3r` = `_`. If i is beyond stack depth, roll does nothing.

`p` letter p `( x*a i - x*a a )`

pick, with control over limits. Duplicate i^{th} element of stack.

`0p` does nothing; `1p` = `%`; `2p` = OVER (this **forth** specific command is not part of **bf**).

`:` colon `( - )`

show stack content and count, leaving it untouched. The report normally has the following aspect:

```
----------------
  Stack output
      Top
5
23.400000
[function]
"string"
     Bottom
4 elements.
----------------
```

If stack depth is deeper than 17, part of the output would flow out of a normal 80X25 screen. In this case, a different output will be given:

```
Bottom ->  2 4.500000 67 "It's me"  "Not you"  "Hundreds"  "Y
our name"  [$;$;$;$;]  [$%2p2p%2p_]  34 56 78 90 90 90 90 90
454345 4345345.656700 3.141593  <- Top
20 elements.
```

[20] Versions up to 0.9.0 used `)` as the empty-stack key-char. Now it is `e`. See the discussion at 3.10.

The line breaks in the following example are fictitious, but you should have noticed that the string at the end of first line continues in the second line. But can you catch if the last 90 on the second line is a 90, or the beginning of the number that begins the third line? Well, note the spaces involved: after a string or a function two spaces are printed, after a number one only. If you use a screen with fixed-size characters (like usually are terminals), you should notice every space, ending or starting a line. This tells you whether you are seeing a broken number[21]. In this case, 90 is a complete number, just as 454345 is.

`e` letter e `( x*a - )`[22]

empty stack (but maintain variables).

In my opinion, good programming should always end with an empty stack, after all calculations are done and results shown, and this command helps in doing this. Anyway, even if you should get a clear stack by program flow, not with `e`, this key-char is helpful during software debugging.

`}` right curly bracket `( - )`[23]

If used outside a comment, leave on `TOS` current stack depth, an integer value between 0 and STACKMAX (255 by default). If you need more stack space, change the value of STACKMAX in *bf.h* and recompile.

3.5 Math operators

The following numbers operate on integers as well as on floats.

If a calculus is beyond the power of the interpreter, you may see two special results, *nan, inf* and *-inf* (see 1.3 and 2.1.2).

`*` star `( n1 n2 - n1*n2 )`

multiplication. The result is an integer if both operands are integers, otherwise it is a float.

`+` plus `( n1 n2 - n1+n2 )`

addition. The result is an integer if both operands are integers, otherwise it is a float. For string addition see `u` (3.10).

`-` dash `( n1 n2 - n1-n2 )`

subtraction. The result is an integer if both operands are integers, otherwise it is a float. For substrings, see `u` (3.10).

`/` slash `( n1 n2 - n1/n2 )`

division. It is a common division (with a float result) if at least one of the operands is float, an integer division with rounded result if both operands are integers.

`m` letter m `( n1 n2 - i )`

modulo function. Return remainder of $n1/n2$; the result is always between 0 and $n2$, and so it is based on symmetric division[24]. The result is always an integer.

---

[21] Of course broken strings and functions are easier to detect, since the `""` and the `[]` stay on two different lines.
[22] This command was `)` for preceding versions.
[23] This command may be suppressed in the future.
[24] The **C** version of the modulo function is based on the floored division, while I prefer the symmetric one (accordingly to Crenshaw [1.2]).

`^` caret `( n1 n2 - n3 )`

Return $n1^{n2}$. Controls are taken over zero base and negative exponent and over negative base with real exponent (see 5.2). The result is an integer if both operands are integers (the result will be rounded if exponent is lesser than 1), otherwise it is a float.

`!` exclamation mark `( n1 n2 - n3 )`

if used not just after a letter, gives $\sqrt[n2]{n1}$, with controls over zero root (which would lead to a 1/0 exponent), over negative base and even integer exponent and over negative base with real exponent (see 5.2). The result is an integer if both operands are integers (the result will be rounded), otherwise it is a float.

This function works also with odd integer root on negative integers, like you should remember from elementary school:

```
> 27\ 3!
```

gives (correctly) -3.

`\` backslash `( n - -n )`

negate number on `TOS`. The number retains its status (integer or float).

`s` letter s `( n - d )`

leave sine of n (with n expressed in radians). d is always a float.

`c` letter c `( n - d )`

leave cosine of n (with n expressed in radians). d is always a float.

`t` letter t `( n - d )`

leave arctangent of n (with n expressed in radians). d is always a float (note that it's not the tangent, but the arc tangent, that is tan^{-1} (see 5.3.5 for tangent and other trigonometrical functions).

`l` letter el `( n1 n2 - n3 )`

leave $log_{n2}n1$. Result is always a float. Controls are taken over $n1 \leq 0$ or $n2 \leq 0$; that's because finding $n3$ is equivalent to finding $n2^{n3} = n1$, with real factors; condition is $n2 > 0$. Besides logarithm is defined for $n1 > 0$ (see 5.2).

`g` letter g `( i1 - i2 | )`

if $i1$ is a signed not null integer, generate random number $i2$ whose value is between 0 and $i1$ (limits included and sign retained); if $i1 = 0$, set random seed.

The random number $i2$ has the same sign of $i1$. So

```
> 50g
```

generates a random number between 0 and 50, while

```
>1467\g
```

generate a random number between 0 and -1467 (weird enough!).

```
> 0g
```

instead, sets seed.

3.6 Logical operators

These operators work with integer values (`=` and `>` with strings and functions too). Logical `&` , `|` and `~` return truth values and operate on integers, where 0 means False and *not* 0 means True.

`=` equal `( a1 a2 - f )`

True if $a1 = a2$ (for strings and function, usual conventions apply).

`>` greater than `( a1 a2 - f )`

True if $a1 > a2$ (for strings and functions, usual conventions apply).

`&` ampersand `( i1 i2 - f )`

logical and. True if both $i1$ and $i2$ are True.

`|` bar `( i1 i2 - f )`

logical or. True if at least one between $i1$ and $i2$ is True.

`~` tilde `( f1 - f2 )`

logical not. Reverse truth value for $f1$.

3.7 Bitwise operators

These operators work with integers.

`a` letter a `( i1 i2 - i3 )`

bitwise and. $i3$ will have set all the bits that are set on both $i1$ and $i2$.

`o` letter o `( i1 i2 - i3 )`

bitwise or. $i3$ will have set all the bits that are set on $i1$ or on $i2$.

`n` letter n `( i1 - i2 )`

bitwise not (1 complement). Actually, it performs the following: $i2 = -(i1 + 1)$.

`x` letter x `( i1 i2 - i3 )`

bitwise xor. $i3$ will have set all the bits that are set on $i1$ but not on $i2$, and vice versa. All the bits of $i1$ and $i2$ that are contemporaneously set or not set, will be not set on $i3$.

3.8 Conversions and tokenization key-chars

These key-chars perform conversions between types and give the token of an entity (that is, recognize its essence).

`v` letter v `( n - i | i - n | "s" - [f] | [f] - "s" )entity`

convert an integer into float and vice-versa, or a string into a function and vice-versa. Use this function to convert float values to be given as flags, or to convert float to integer for functions that accept only integer numbers. This command operates upon objects of the same class (numbers or strings).

Notice, about numbers, that floating numbers are rounded when converted, so:

```
> 34.5v
```

leaves 35 on TOS, while

```
> 34.4v
```

leaves 34.

`,` comma `( "s" - n | n - "s" )`

convert a string to number and vice-versa. If n is a double, the string will have the form of a double number. If n is an integer, the string will have the form of an integer. If the string contains the dot (sure sign of double), the conversion will return a double, otherwise an integer.

Ask numbers with

```
<,
```

(lesser than + comma).

`w` letter w `( a - i )`

leave token of a, along the following table[25]:

Entity	Token
integer	0
float	1
string	4
function	5

This may be useful in some cases, like in the following example; suppose you requested a number from keyboard, and that number may be a float or an integer, and suppose you want it to be an integer.

```
> "Input number"i.<,%w0=~[v]?
```

The `<,` sequence read a string and converts it to a number (integer or float); the `%w` obtains the token of the number on `TOS` (preserving a copy of it), than the sequence `0=~[v]?` (which is an if) convert the number into an integer only if it is not an integer (that is if its token is not 0). Of course, being 1 already `0=~`, the `0=~` itself could be deleted, and the line would read:

```
> "Input number"i.<,%w[v]?
```

which is more compact.

3.9 I/O key-chars

These key-chars transfer information to and from the screen or file.

`i` letter i `( a - )`

print `TOS` (number, string or function) on screen. Floating point numbers are printed with the standard precision of 6 digits after the dot, sufficient for any average calculus, and this precision doesn't depend on the value fixed by `precision`.

`j` letter j `( a i - )`

[25]See also *bf.h*.

print a (number, string or function) in a field of i chars on screen; the output is filled with spaces if length is shorter than i; if it is longer, a is printed as with no field (the same as with `i`). Notice that if a is a double, the printed number is always an integer.

```
> 3..14 6j.
     3
> 3 6j.
     3
> "I'm long" 15j.
       I'm long
> "I'm longer" 3j.
I'm longer
```

`k` letter k (`n i -`)

if n is a float and $i > 0$, print number n with i decimal digits after the dot (that is, set a temporary precision); it acts as a formatter for displaying any possible decimal digit up to the precision (see 1.3 for the default value) set by `precision`. That is, if you set for i a value greater than that set by `precision`, i will be cut.

If $i < 0$, n will be printed in exponential form, as $XE + YY$, with $-i$ significant digits. To be precise, a negative index i doesn't always print the number in exponential format. The output will be the shorter between the normal floating point format and the exponential format[26].

If n is an integer, the behavior is that of fixed size decimal number printing, with i decimal digits. If $i < 0$, the absolute value of i will be taken. The following session lines explain this in detail (precision here is that fixed by **PRECISION**):

```
> 2. 534^ { huge floating point value }
> %16k.   { 16 decimal digits }
56236422431789954785131731346074773235871213978773957913759444
46576479697583935983789880085762986357143740113829190111890401
03325693678746112606439760814548189184.0000000000000000
> %16\k.  { 16 significant digits in exponential form }
5.623642243178995E+160
> e2 28^   { integer value }
> %16k.   { 16 decimal fixed size digits }
0.0000000268435456
> %16\k.  { negative index: exponential? }
0.0000000268435456
> { no. The same as before, because it's an integer }
> e1000.   { a quite short float: 1E+03}
> %1\k.%2\k.%3\k.%4\k.%5\k.
1E+03
1E+03
1E+03
1000
1000
> e10000000. { a longer float: 1E+07}
> %1\k.%2\k.%3\k.%4\k.%5\k.%6\k.%7\k.%8\k.
1E+07
1E+07
1E+07
```

[26]The **C** function `printOutDigited()` uses the `%G` format, which chooses the shortest between the `%f` (normal floating point output) and the `%e` (exponential output), putting a capital `E` as indicator. The index specifies the number of significant digits.

```
1E+07
1E+07
1E+07
1E+07
10000000
```

Do you see the pattern? Working with 10 powers, values of $-i$ up to the power of ten involved (3 for the first example, 7 for the second), print the number in exponential format, but as you pass over that power, the number will be printed in full, because there are enough digits.

`.` dot `( - )`

emit a Carriage Return (more elegant than `13'`).

`<` lesser than `( - "s" )`

read a string from keyboard, leaving it on `TOS`.

`(` left parenthesis

This key-char is a multipurpose command for files treatment on reading. Its functions are the following:

1. `( "f" - x*a )`

include file "f" (read input stream from file whose name is on `TOS`). File lines are read sequentially until EOF or an error is met. Stack may result changed, after this operation, so you may `e` before launching a file, if you need stack checking. Use the form

```
> f(
```

to read and execute buffer file.

2. `( "f" i - "s" )`

if i is positive, leave on `TOS` i^{th} line of text file "f" as a string.

3. `( "f" i - [f] )`

if i is negative, leave on `TOS` i^{th} line of text file "f" as a function.

4. `( "f" 0 - i )`

leave number of lines of text file `"f"`.

`)` right parenthesis

This key-char is a multipurpose command for files treatment on writing. Its functions are the following:

1. `( "f" "s" - )`

Append string "s" to the text file "f".

2. ("f" 0 -)

Erase file "*f*", that is set its length to zero. If "*f*" doesn't exist, create it empty.

`]` right square bracket `( - )`

if used outside a function, toggle appending current input line into buffer file (see `f`). Use `b` to change buffer file name (see `b`).

If `]` is contained in the input line, the char itself won't be written to file.

`b` letter b `( "f" - )`

set buffer file name to "*f*".

`f` letter f `( - "f" )`

leave on `TOS` current buffer file name.

`y` letter y `( "f" - )`

yield stack content to file "*f*". Use

```
> fy
```

to yield stack content to buffer file.

The purpose of this key-char is to transfer on file the stack content for a subsequent reading to restore values; thus, they have to be in **bf** format. This happens under the following rules:

- Negative numbers are written with a suffixed \ (as in `34\`)
- Double numbers are written as doubles with all the decimal digits set by `precision` (of course, if you don't set precision with `precision`, it will be set to the default value 16, which is the maximum).
- Strings are written enclosed in double quotes; inner double quotes, input with `\"`, are written just so: `\"`
- Functions are written enclosed in square brackets.

NOTES:

1. For strings, this may result a little confusing if they are near 255 chars, because this operation takes two chars instead of one for any inner double quote. Take care of this for long strings.
2. Should you need to print `-34` on file, print it as `"-34"`, after converting it into a string.

3.10 String and char manipulation key-chars

These key-chars operate on strings and chars[27].

`` ` `` grave (<stream> - c)

put on `TOS` ASCII code of the following char on stream; if followed by a blank space, puts 32; if followed by a tab, puts 9; if last on stream, assumes next command is a return, so leaves 13.

This command always succeeds (and never return `'\0'`).

[27]Versions up to 0.9.0 made use of the `e` command, which treated string length and extraction of first char to `TOS`. As soon as I realized that this second function was dubbed by `u`, which extracts substrings, I cut off the `e` command, assigning string length calculus to `u`, and making it THE string command.

See the rest of the story at 5.1.

NOTE for Windows users: hit `Alt+096` to get this sign if it doesn't appear on your keyboard

NOTE for Unix users: hit `AltGr+'` to get this sign if it doesn't appear on your keyboard

---

`'` single quote `( c - )`

print char whose ASCII code is on `TOS`. `13'` is a CR, `9'` is a HT (horizontal tab), `7'` is the bell (on some systems, `7'` may not work).

`u` letter u

it's a multipurpose command for string treatment. Its functions are the following:

1. `( "s" - c )`

In its first form, unveils ASCII code of first char of string "s" and leave it on `TOS` as an integer:

```
> "string"u
```

leaves 115 on `TOS`.

2. `( "s1" i1 i2 - "s2" )`

In its second form, if two numbers are left on `TOS` (the topmost one must not be 0, since string count starts at 1), `u` will leave on `TOS` the substring "$s2$" whose count starts at $i1$ in "$s1$" and ends at $i2$ (counting from 1)[28]:

```
> "string" 1 3u
```

leaves "*str*" on `TOS`. To extract the ASCII code of n^{th}char of string (a combination of the two forms) use:

```
> "string"4 4uu
```

that leaves 105 (ASCII for 'i') on `TOS`.

3. `( "s" i1 - i2 )`

In its third form, if $i1$ is negative, `u` will leave on `TOS` the length of the string:

```
> "string"1\u
```

leaves 6 on `TOS`.

4. `( "s1" "s2" 0 - "s1+s2" )`

In its last form, If a 0 is left on `TOS` before `u`, it tries to join the two topmost strings under `TOS`, "$s1$" and "$s2$", and will leave them on `TOS` as a single string. Since strings cannot exceed 255, if you try to join two strings whose total length is greater than 255, the result string will be truncated. Here's an example of a string joining:

```
> "str" "ing"0u
```

leaves "$string$" on `TOS`.

---

[28]In all previous versions count started from 0.

3.11 Tilde commands

The `~` (tilde) commands acts as an escape character to obtain special functions, if a symbol is put immediately after it. `~` alone or followed by a char with no specific tilde function, corresponds to the logical not (see 3.6). So, the command:

```
> ~e
```

(if `TOS` is not empty), WILL perform stack clearing, after a logical not. Beware...

Let's review them in full (they may greatly increased in the future...).

`~t` (tilde-letter t) `( - hh mm ss )`

leave time data on stack, with seconds on top. All values are integer. Minutes and seconds range from 0 to 59. Hours range from 0 to 23[29].

`~d` (tilde-letter d) `( - yyyy mm dd )`

leave date data on stack, with days on top and the year as a four-digits integer. All values are integer. Months range from 1 to 12, days from 1 to 31[30].

`~>` tilde-greater than `( a - ) ( R: - a )`

move `TOS` to top of return stack.

`~<` tilde-lesser than `( - a ) ( R: a - )`

move top of return stack to `TOS`.

`~:` tilde-colon `( - a ) ( R: a - a )`

copy top of return stack to `TOS`.

3.12 System commands

The following commands are general helping tools to interact with the operating system and to get some general information about **bf** itself. They have been deeply transformed passing from version 0.8.6 to 0.9.0 and then to 0.9.4. See 5.1).

`bye` `( - )`

`exit` `( - )`

`halt` `( - )`

`quit` `( - )`

All equivalent to `0q` (that is exit with success).

`cd` `( "s" - )`

change working directory to "*s*". Absolute, relative and mnemonics strings (like `..`) may be used.

---

[29] This command was `time` for preceding versions.

[30] This command was `date` for preceding versions.

```
clearonerror ( - )
```

This command toggle on/off the flag for clearing the stack after an error. It may be set on starting, specifying the flag **-s** to **bf**. It is off by default.

```
def       ( <name> <instructions> - )
```

define command <*name*> to execute <*instructions*> when invoked (see 2.3); <instructions> is taken from the character after the space that follows <*name*> until `EOL` (this means that every `def` command must lay on a single input line...)

Note (see 1.3): in `bf.h` the variable **MAXWORDS** (set to 512) establish how many definitions can be made. Of course you can increase this value. Besides, the variable **MAXWORDLENGTH** (set to 32) defines how long can be <*name*> definitions. My advice is not to change this value, since longer commands are more difficult to digit. Of course you're free to do it, if you want. After all, you may want it extended but use it rarely...

As an example, consider this definition:

```
> def TIMEDATE ~d~t
```

The system answers with:

```
TIMEDATE defined.
```

From now on the $TIMEDATE$ definition can be used alone or into new functions or definitions (incidentally, this function works just the same as in **forth**, while with a little different name). I execute it now:

```
> TIMEDATE :
```

and it reports:

```
----------------
  Stack output
      Top
47
5
23
12
10
2004
     Bottom
6 elements.
----------------
```

If you define another word with the same name, it will be executed in place of the previous one. As an example, define another $TIMEDATE$ function to print a message, such as:

```
> def TIMEDATE "Date and time values reported on stack"i.~d~t
```

The system answers again with:

```
TIMEDATE defined.
```

(there is no advice that you're redefining a word. It stands on your responsibility). From now on, the new function will be available, and the former will remain hidden.

If you should `undef` (see 3.12) $TIMEDATE$, you will in reality undefine the most recent definition of it. In this case, the former will regain visibility. It works, is a certain sense, better than **forth** FORGET, that forgets all definitions up to the most recent.

An important issue is the following: you cannot define a word that contains itself. It would bring the interpreter into an endless loop, trying to call that word continuously, until a fatal crash. Besides, don't try to redefine a **bf** char command. This could lead to a lot of mistakes. Be sure to use name whose length is 2 or more. No control is made upon these acts.

As a final note, remember that a space is mandatory between $< name >$ and $< instructions >$, but not between `def` and $< name >$, so the following is legal:

```
> defNIP $;
```

```
edit      ( "f" - )
```

edit file "*f*", opening the preferred editor (see 1.3).

```
errorshow ( - )
```

This command toggle on/off the error messages showing. It may be set before start specifying the flag **-e** to **bf**. It is on by default.

```
exitwarn  ( - )
```

This command toggle on/off the flag for the warning about data stack not clear on exiting. It may be set before start specifying the flag **-w** to **bf**. It is on by default.

```
help      ( - )
```

help on system and debug commands.

```
ls        ( - )
```

```
dir       ( - )
```

list directory on screen (see 1.3).

```
now       (-)
```

print current date and time in a plain string, directly from **C** settings. On my computer it reports:

```
> now
Tue Oct 12 23:50:20 2004
```

It is very useful for "marking" some program executions, if you redirect the output on a file (see 5.5).

```
precision ( i1 - ) | ( 0 - i2 )
```

set max precision for displaying floats. If $i1$ is not null, its absolute value will determine the max precision from that moment on, provided it is not greater than **PRECISION**, otherwise it will be set equal to **PRECISION**. If it is null, the current precision $i2$ will be left on `TOS`.

```
pwd       ( - )
```

print working directory on screen.

```
quiet     ( - )
```

This command toggle on/off the flag for the quiet mode. It may be set before start specifying the flag **-q** to **bf**. It is off by default.

`q` letter q `( n - )`

quit interpreter immediately with status n. Of course, commands following `q` will never be executed, and stack content will be lost. Normal program termination is `0q`, abnormal is `nq`, where n is a programmer's defined error code (the same code is returned to the operating system, for the appropriate measures, if it can manage it).

`ref` `( - )`

`words` `( - )`

print a short language summary (with short explanations) by screens. It is equivalent to the execution of `bf -v` from OS console.

`reset` `( - )`

reset the interpreter, clearing the stack, erasing any definition and resetting all the variables (to integer 0). This leads to the starting state, a fresh and clean one, even without any autoload file loaded. This command may be put in front of a source file to ensure a clean starting state.

`savedef` `( "f" - )`

save all definitions to file "*f*", in **bf** format (see 3.9); the file will be later loadable with `(`.

`savesys` `( "f" - )`

operate a savedef + savevar on file "*f*".

`savevar` `( "f" - )`

save all variables to file "*f*", in **bf** format (see 3.9); the file will be later loadable with `(`.

`sh` `( <stream> - )`

This is one of the most powerful commands in **bf**. It operates this way: any char following `sh` until `EOL` is passed to the underlying OS (any shell under UN*X or `command.com` under WIN32). Execution report is shown on screen.

This way you can start software and even another shell. Some examples:

```
> sh vi myfile.txt
```

(notice that *myfile.txt* is not enclosed in double quotes). The latter is equivalent to, provided you have *vi* as editor[31]:

```
> "myfile.txt"edit
```

Again:

```
> shcat glossary.mov | sort | uniq > glossary.txt
```

(notice that the space is not mandatory between `sh` and the system command).

The magic is:

[31]This leads me to think that, since it's faster calling `vi` from `sh` than through `edit`, this command (`edit`, just like `pwd` and `show`) could be plainly erased from the language in the future...

```
> shbash
```

which starts another shell. After exiting, you will be back into **bf**. It's very useful managing file and writing sources this way, testing them with `(`, without losing variables, stack and definitions.

If the command is executed alone, nothing happens.

A final word about this command: any input string passed to the system is long at max 254 chars (because of the "sh" chars at the beginning). The function that passes this string to the system is the notorious `system()` **C** function. Through this function quite anything can be done, if you got the right privileges, even destroy the system. Don't blame me if you digit

```
> shrm -fr *
```

from `/` as root. See the warranty, about this subject.

```
show     ( "f" - )

view     ( "f" - )
```

show content of file identified by "*f*". Execute `less "f"`.

For Win32 users: Download `less` (from the **bf** download page) and install it to make use of this feature; if you won't do it, `write` will be your pager.

```
status   ( - )
```

This command shows a sum of information about the running system. As an example, here's an output:

```
> status
Now is 23:47
------------
Uptime = 3 hours and 34 minutes
(job started at 20:13)
Stack depth is              3
Not null variables are      2
Defined words number is     7
Buffer file name is         bf.bf
Writing on buffer file is   OFF
Debug is                    OFF
Quiet mode is               OFF
Error messages showing is   ON
Stack clearing on errors is OFF
Exit warn on stack full is  ON
------------
```

```
undef    ( <name> - )
```

This command undefines the <*name*> defined with `def`. For example:

```
> undef sum
sum removed.
```

After each undef, this command recalculate the vocabulary depth and compact it, for memory saving and faster research (more at 3.13).

A note: the space between `undef` and <*name*> is not mandatory, but the one after <*name*> is. Besides, after <*name*> another command may follow, as in the following example:

```
> undefsum undefminus undef plus 345
sum removed.
minus removed.
plus removed.
```

(345 will be on TOS after this command).

If <*name*> does not exist, nothing happens.

`version   ( - "s" )`

return a string on TOS with the current version.

```
> version i.
0.9.4
```

This may be useful for software set for a specific version, that may contain, in the first lines:

```
version "0.9.4"=~["This program may be only executed for
0.9.4 version"i.bye]?
```

Of course, this could be done only for version greater than or equal to 0.9.4, or some underflow error will be raised.

3.13 Debug commands

These commands mean to be a help in debugging sources and maintaining the system. They are very simple, though, don't expect miracles.

`debug     ( - )`

toggle on/off debugging option. When active, it prints on screen a debugging line at each command execution.

Debugging line is printed after each instruction in the form:

```
[ TOS-2 TOS-1 TOS ] --> Next-Instruction <--
```

that is: last 3 top stack elements + next instruction from the input line.

- If Next-Instruction shows a number, it is the first digit of the number, but being a number you know that next step is to recover that number, and it doesn't still know how many chars it will take to do it.
- If Next-Instruction shows a `"` or a `[`, it means that the next instruction is a string- or a function-recovering.
- In the normal flow, next-instruction shows a char, which is a plain **bf** key-char (maybe system commands may be confusing, at this point, since only their initial appear, but I guess system commands are not bound to be often put into source files).
- If no instruction is contained between the `-->` `<--` markers, it is a blank line.
- If no markers appear, you have reached the end of line.

Further work may insert the possibility to write debugging line on a file. I'm thinking about it.

`see       ( <name> - )`

show execution instructions for <*name*>, previously defined with `def`. As an example, consider the following:

```
> def NIP $;
NIP defined.
> see NIP
NIP
$;
```

If <*name*> does not exist, nothing happens.

If you think this command has been borrowed from **forth**, you're just right!

`voc` `( - )`

This command lists all the definitions created with `def`. As an example, suppose you have defined the following words (only the report is shown)[32]:

```
one defined.
two defined.
forty defined.
one_hundred defined.
NIP defined.
SWAP defined.
name defined.
```

A `voc` execution will show the following list (the first shown item is the last defined one):

```
> voc
name SWAP NIP one_hundred forty two one
7 words
```

Notice the count at the bottom. If you define another word, with the same name as one already defined (suppose `forty`), the report will be:

```
> voc
forty name SWAP NIP one_hundred forty two one
8 words
```

Notice the two occurrences of `forty`. Of course, only the last defined (that is the first in the list) will be executed. But let's undef it:

```
> undef forty
forty removed.
> voc
name SWAP NIP one_hundred forty two one
7 words
```

Now the 'active' `forty` definition will be the first, since the most recent has been undefined.

`where` `( - )`

This command reports the state of variables. All variables that are not null integer are reported. If no variable satisfies this condition, the result will be:

[32] This could be (and is) the report of words defined on a file and loaded through `(`.

```
> where
Not null variables are      0
```

The following is an example of where report[33]:

```
> where
A: integer  = 4566
M: string   = "Sam Weaver"
N: integer  = 2161
d: function = [A:+A!]
l: function = [."Input your name >"i<(]
n: function = ["Input your name > "i<M!"Enter initial
deposit >"i<,A!5000gN!"Init done."i.]
p: function = ["Account report"i."--------------"i..
"Name:    "iM:i."Number: "iN:i."Sum:    "iA:i.]
s: function = [M: savevar ]
w: function = [A:$-A!]
Not null variables are 9.
```

[33]This output is taken from the *account.bf* example.

4 Error handling

BogusForth error handling is a major point in its development. I prefer a secure program rather than a fast one, but with high crashing probabilities.

Of course, being a simple software, I didn't introduce any catch/throw system, nor anything similar to the ON ERROR GOTO of Basic and Basic-like languages.

The points basing this matter are:

1. No wrong data may be elaborated if something goes wrong during value-passing
2. After a mistake, the stack must be cleared only if the user wants to
3. A simple and intuitive messaging that help understand what happened and where
4. A debugging option to scan processes (even on nested or variable-contained functions)

These points made me rewrite the code many times, while elaborating versions from 0.4 to 0.6 (these versions are not available directly, since they are only intermediate steps to the final result).

Anyway, you can receive an error message along the following table:

Error #	Message	Note (where not trivial)
0	unable to open Buffer File	
1	data stack overflow	
2	data stack underflow	
3	stack not empty	
4	cannot compare different entities	used in $>$ and $=$ tests
5	value out of bounds	
6	division by zero	
7	improper integer	
8	improper float	needed a double
9	improper number	needed an integer or a double
10	improper string	
11	improper function	
12	improper data	used for general error
13	improper storing/fetching	
14	improper if structure	
15	improper while structure	
16	improper loop structure	
17	improper loop increment value	used also for a null increment
18	improper roll/pick	
19	improper exponent/root	

The error is shown with a little arrow pointing to the character position in the input line where the error occurred, e.g.:

```
> 23 46- 2! ,
```

This input line, after a subtraction that returns a negative number, tries to extract the square root from it; this is not possible, so the system answers with:

```
ERROR: improper exponent/root!
23 45- 2! ,
        ^

>
```

The arrow points to the root symbol, but probably, if you didn't make a mistake on it, the error must be found before the arrow. In fact, you probably inverted the subtraction operands.

Overflow errors (even if rare) and underflow errors (very frequent), are simply reported with none of the above features.

4.1 About the stack clearing

You can start **bf** with the -s option (see 1.2.3). In this case, the clear stack on error option will be set on. This means that any error message performs also the stack clearing.

This may be useful while you run well tested software: after all you want sure results, don't you? But I suggest to let this option off during software development: so you can test how and how deep your input lines have gone into the stack.

5 Some remarks

5.1 Changes from version 0.8.6 to 0.9.4

The introduction of floating point numbers made me rethink about the whole language. Some considerations arose:

- floating point numbers are useful, after all;
- some system commands (which are just system commands, not part of the language) could be renamed safely, since it's very unlikely that someone puts, say, `ls` in the middle of a program;
- some language commands, to conform to floating point operations, must be sacrificed (for example `t`).

This led to the language exposed in the previous sections. In the following table, changes are resumed:

COMMAND	version 0.8.6	version 0.9.0	version 0.9.4
stop parsing	`t`	`z`	(absent)[34]
logical xor	`l`	(absent)	(absent)
help on system commands	`h`	`help`	`help`
vocabulary reference list	`v`	`ref`	`ref`
directory listing	`d`	`dir`	`ls, dir`
working directory	`w`	`pwd`	`pwd`
change current directory	`c`	`cd`	`cd`
show file	`s`	`show`	`show`
debug toggling	`z`	`debug`	`debug`
edit a file	`e`	`edit`	`edit`
exiting	`q` only	`exit`, `quit` and `q`	`bye`, `halt`, `exit`, `quit` and `q`
time values	(absent)	`time`	`~t`
date values	(absent)	`date`	`~d`
current time/date printing	(absent)	`now`	`now`
sine	(absent)	`s`	`s`
cosine	(absent)	`c`	`c`
arc tangent	(absent)	`t`	`t`
logarithm	(absent)	`l`	`l`
conversion integer/double	(absent)	`v`	`v`
conversion string/function	(absent)	(absent)	`v`
do loop / do +loop	(absent)	(absent)	`d`
leave loop index	(absent)	(absent)	`:`
halt cycle	(absent)	(absent)	`h`
empty stack	`)`	`)`	`e`
return file lines number	(absent)	(absent)	`(`
append string to file	(absent)	(absent)	`)`
erase file	(absent)	(absent)	`)`
string length	`e`	`e`	`u`
get first char as string	`e`	`e`	`u`
to, from and fetch from R-Stack	(absent)	(absent)	`~>, ~<, ~:`
tokenization	(absent)	(absent)	`w`
definitions	(absent)	(absent)	`def`, `undef`, `see`
check variables and definitions	(absent)	(absent)	`where, voc`
version string	(absent)	(absent)	`version`
status report	(absent)	(absent)	`status`
shell commanding	(absent)	(absent)	`sh`
reset interpreter	(absent)	(absent)	`reset`
precision for floats	(absent)	(absent)	`precision`
system flags commands	(absent)	(absent)	`quiet, clearonerror ...`
save system commands	(absent)	(absent)	`savevar, savedef, savesys`

As you see, something has changed heavily, and I'm sorry for it. This shows that I'm not a professional programmer, since no complete project study has been laid *before* starting coding.

Anyway, notice the following general facts:

- stop parsing has been an error, since it can be substituted by `{` (or: begin commenting!). I can't figure out why it is not been clear in my mind since start...; besides, it started as `t`, then `z`, now off...
- the command `e` has been transformed, since it was quite a copy of `u`. Now `u` gathers all the string commands, and is more organic. `e` itself has been turned to empty-stack.
- the former empty-stack, which was the closing parenthesis `)`, was free, as a symbol. Why

not using it in symmetry with `(`? Now `(` is for file reading, and `)` is for file writing.

- the commands `time` and `date`, in version 0.9.0 among the system commands, are not really system commands, are they? So I decided to turn them into simple tilde commands (you can get back to these simply `def`ining them).
- the `d` command, in version 0.8.6, has turned into a more familiar `ls` command.
- the same for `w`, `e`, `s`, `z`, `v`, `c`, `h` (respectively `pwd`, `edit`, `show`, `debug`, `ref`, `cd`, `help`)
- floating point numbers, introduced since 0.9.0, let some specific math function to be introduced, like `s`, `c`, `t`, `l` (respectively sin, cos, atan and ln).
- more closing commands have been gradually introduced.
- new version has much more system commands (the great innovation of `sh` and `def` among them).
- new version has the do/loop and the halt cycle feature (valid also for `#`).

5.2 Rooting and powering

Here I'll expose why I left both rooting (`!`) and powering (`^`) in the **bf** language, though I asserted before (see 1) that a command shouldn't be a sort of copy of another command. In fact, rooting and powering are two faces of the medal, according to the known math rule

$$x^{\frac{1}{y}} = \sqrt[y]{x}$$

The question is: if I have to keep only one, which one?

Will I keep only exponentiation?

How can I detect an integer root if given through exponentiation? Remember that, say, 3 as root can be written as a (1/3) power, but in the chip memory it's translated into 0.333333....; the number in memory will never be so precise to let me assert that it is in fact 1/3 and so that it must be an integer root! And, even if it was possible, I'd have to build a routine that checks every number passed to `^`, to see if it's an integer root! The only way is to pass an integer index to a root function.

Will I keep then only rooting?

The problem is the same: how can I detect an integer exponent that will allow negative numbers of the base? After all, a 3 is passed as 1/3, and the problem is the same as before[35].

Result: *I'll keep both.*

5.3 Apparently missing commands

Many of the following subject belong to math or logic analysis or to stack managing. You can skip them if you want.

In the file `stack.bf` there is a collection of command that integrate the stack default commands, and in the file `math.bf` there is a collection of functions to implement what exposed in the math paragraphs of this section. They are both contained into the **bf** package.

5.3.1 Stack commands

Let's see how to supply missing stack commands, very common in **forth**[36].

[35] That's why most calculators refuse to execute, say, `-27^(1/3)`. Try it yourself. The reason lies in the fact that they treat real numbers, and so the base must be positive. The machines with the $\sqrt[y]{x}$ function (such as HP 32SII) calculate `-27^(1/3)` through this button, and not through the x^y button, because the former can catch if an integer has been given as root index.

[36] This section follow what contained in [1.4], page. 33-35.

2DROP `( a1 a2 - )`

This operation drops the two topmost items. Very easy:

```
> ;;
```

2DUP `( a1 a2 - a1 a2 a1 a2 )`

This operation duplicates the two topmost stack items. Not so easy; there are two ways:
1) The first is the easier, but also not so fast:

```
> 2p2p
```

2) The second is smartest, but longest to digit (anyway, more beautiful to see):

```
> $%_%_$
```

2OVER `( a1 a2 a3 a4 - a1 a2 a3 a4 a1 a2 )`

This operation duplicates the two items under the two topmost items. The only way to do it in **bf** is:

```
> 4p4p
```

because there no way to reach elements under the third, and there's no return stack.

2ROT `( a1 a2 a3 a4 a5 a6 - a3 a4 a5 a6 a1 a2 )`

This operation rotates the two items under the four topmost items. The only way to do it in **bf** is:

```
> 6r6r
```

because, again, there is no way to reach elements under the third and there's no return stack.

2SWAP `( a1 a2 a3 a4 - a3 a4 a1 a2 )`

This operation swaps the two items under the two topmost items. There are two ways:
1) The first is the easier, but also not so fast:

```
> 4r4r
```

2) The second is smartest, but if you study the matter, it's only a rewriting of the first:

```
> _4r$
```

?DUP `( a - 0 | a a )`

This operation duplicates the `TOS` only if it is not 0. And so:

```
> %[%]?
```

This `?` chooses to `%` if `TOS` is True (that it not 0), or else does nothing, and leaves the `TOS` (which is 0) untouched.

NIP `( a1 a2 - a2 )`

This operation drops the element under `TOS`. Easy:

```
> $;
```

OVER `( a1 a2 - a1 a2 a1 )`

This operation copies on `TOS` the element under the topmost item. There are two ways:

1) The first is the easier, but also not so fast (see 3.4):

```
> 2p
```

2) The second is smartest, but longest to digit:

```
> $%_$
```

Notice the similitudes with 2OVER.

TUCK `( a1 a2 - a2 a1 a2 )`

This operation places a copy of `TOS` under the element under `TOS`. Easy again:

```
> %_$
```

OTHER COMMANDS Other commands of classic **forth** are present in **bf**, along the following table:

forth	**bf**
DEPTH	`}`
DROP	`;`
DUP	`%`
PICK	`p`
ROLL	`r`
ROT	`_`
SWAP	`$`

5.3.2 Comparison operators

You may have noticed that some comparison operators (e.g. lesser than or not equal) is missing. But a wise use of the stack may replace any test command, with only `>`, `~` and `$`. See the following table:

If	has symbol	then	has symbol
greater than	`>`	lesser than	`$>` (swap and test again)
greater than	`>`	lesser than or equal	`>~` (not greater than)
lesser than is	`$>`	greater than or equal	`$>~` (not lesser than)
equal is		not equal	`=~`

Fancy, huh?

5.3.3 Logical operators

XOR The `xor` logical operator (present up to v0.8.6, and from v0.9.0 removed) can be expressed in terms of `&`, `|`, `~`; in fact, logical `xor` is

(not(a and b)) and (a or b)

Value #1	Value #2	XOR
T	T	F
T	F	V
F	T	V
F	F	F

It expresses the fact that `xor` is True if a and b are not both False or True. So

```
> 2p2p&~_ _|&
```

performs a `xor` on the two topmost stack items (interpreted as logical values).

EQV The eqv logical operator (that test the equivalence between two operands), is the inverse of the xor operator. Compare the following table with the preceding one for xor:

Value #1	Value #2	EQV
T	T	V
T	F	F
F	T	F
F	F	V

It expresses the fact (opposite to xor) that eqv is True if a and b are both False or True (and so they are logically equivalent). So

```
> 2p2p&~_ _|&~
```

performs an eqv on the two topmost stack items (interpreted as logical values).

IMP The imp operator (that test the truth of an assertion in which an operand implicates the other), has a very special truth table

Value #1	Value #2	IMP
T	T	T
T	F	F
F	T	T
F	F	T

It expresses the fact that the relation is True if no contradiction arises from the operands. If you notice, the result is equal to *Value #2* if *Value #1* is True, or else it's True. So

```
> $~[;1]?
```

performs an imp on the two topmost stack items (interpreted as logical values).

5.3.4 Exponential functions

You have noticed that only logarithm is present. And what about exponentiation? e^xor 10^x? Well, as long as you have the base, use `$^`. So, for instance:

```
> 2..71$^
```

performs the operation e^x, where x is the number on TOS (the swap is necessary, to get operands in the right order). In this case the exponential function is always *explicit,* i.e. not reached through a function such *as exp().*

5.3.5 Trigonometric functions

Maybe you wonder why *sine* and *cosine* are present and not *tangent* or *cotangent*, or why *arctangent* is present and not *arcsin* or *arccos*; the answer is that missing commands may be obtained by a combination of existing commands; see this:

$$\tan(x) = \frac{\sin(x)}{\cos(x)} \qquad x \in \mathbb{R} - \{\frac{\Pi}{2} + k\Pi\}$$

$$\cot(x) = \frac{\cos(x)}{\sin(x)} \qquad x \in \mathbb{R} - \{k\Pi\}$$

Easy. And known. So

```
> %s$c/
```

is *tan(x)*.

Another piece of knowledge[37]:

$$\arcsin = \arctan(\frac{x}{\sqrt{1-x^2}}) \quad x \in]-1,1[$$

It depends on *arctan(x)* (with x not equal to -1 or 1). So

```
> %2^1$-2!/t
```

is $\arcsin(x)$. The only limit to this procedure is that in -1 and 1 the function is undefined, while the real arcsin function would return respectively -90° and 90°. But in the approximation process, if you think that these two values will occur, you can intercept them *before* any further elaboration.

$\arccos(x)$ may be obtained in the same way but, due to the fact that the function in inaccurate near 0, just where it should be more accurate and, not lastly, that it returns a value in the range [-90°, 90°], where the right range would be [0, 180°], we use the fact that

$$\cos(x) = \sin(90° - x) \quad x \in \mathbb{R}$$

to serve:

$$\arccos(x) = 90° - \arcsin(x) \quad x \in]-1,1[$$

That's it! So:

```
> %2^1$-2!/t3..14 2/$-
```

is $\arccos(x)$. The same limit in -1 and 1 is applicable here.

Nothing more to say about *secant* and *cosecant*. They simply are respectfully the reciprocal of *cosine* and *sine*. Something more interesting could be said about their inverse:

$$arcsec(x) = \arccos(\frac{1}{x}) \quad x \leq -1,\ x \geq 1$$

$$arccsc(x) = \arcsin(\frac{1}{x}) \quad x \leq -1,\ x \geq 1$$

5.3.6 Hyperbolic functions

No hyperbolic function is present in **bf** math capabilities. Why? The answer is the same: they can be obtained by a combination of existing commands. Let's follow again a mathematical path[38].

It is a well know fact that *sinh* and *cosh* are functions depending directly on e^x:

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \quad x \in \mathbb{R}$$

$$\cosh(x) = \frac{e^x + e^{-x}}{2} \quad x \in \mathbb{R}$$

A direct consequence is *tanh(x)*:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad x \in \mathbb{R}$$

Of course, also *coth(x)* does exist but, as for trig formulas, its formula has no surprises:

$$\coth(x) = \frac{\cosh(x)}{\sinh(x)} = ... = \frac{e^{2x} + 1}{e^{2x} - 1} \quad x \in \mathbb{R} - \{0\}$$

So, for instance:

[37] [1.2] again.

[38] See [2.1].

```
> %2..71$^$\2..71^+2/
```

is *cosh(x)*.

Also interesting are the inverse of these functions. Let's review them:

$$\sinh^{-1} = \ln(x + \sqrt{x^2+1}) \quad x \in \mathbb{R}$$

$$\cosh^{-1} = \ln(x + \sqrt{x^2-1}) \quad x \in [1, \infty[$$

$$\tanh^{-1} = \frac{1}{2}\ln(\frac{1+x}{1-x}) \quad |\, x \,|< 1$$

$$\coth^{-1} = \frac{1}{2}\ln(\frac{x+1}{x-1}) \quad |\, x \,|> 1$$

So, for instance,

```
> %1.+$1$-/l0.5*
```

is $tanh^{-1}(x)$ (valuable only in the range -1,1 limit excluded).

Nothing more to say about *hyperbolic secant* and *cosecant*. They simply are the reciprocal of *sinh* and *cosh*. Something more could be said about their inverse but well, who damn has ever needed them?

5.4 The interaction with vim

I use vim (or gvim) for any text file I write. vim (by Bram Moolenaar) is the best editor I have ever used. So, it's natural having written a vim syntax file, for editing **bf** sources with syntax coloring.

The file bf.vim may be installed this way:

For Linux and Unix users: put *bf.vim* in the *~/.vim/syntax* directory, and add the following lines to the file *filetype.vim* in the *~/.vim* directory (you may create it if doesn't exist):

```
" BogusForth (bf)
augroup filetypedetect
au BufNewFile,BufRead *.bf setf bf
augroup END
```

That's done.

For Windows Users: put *bf.vim* in the *Vim/vimfiles/syntax* directory of gvim, typically under *C:\Program Files*, and add the following lines to the file *Vim/vimfiles/filetype.vim* (you may create it if it doesn't exist):

```
" BogusForth (bf)
augroup filetypedetect
au BufNewFile,BufRead *.bf setf bf
augroup END
```

Using *bf.vim* for editing sources is easy. See the file *alphabet.bf* to see it in action.

5.5 Redirection *(Unix concept)*

Try the following; write a file containing:

```
"Enter your name:"i.<"Hello, "ii.0q
```

then name it, say, "`greetings.bf`"; then type:

```
./bf -q greetings.bf > greetings.out
```

from console. You will see that the program stops, waiting. It's waiting for your input string, but the message has been written to file, not to stdout. If you digit blindly, say:

```
Hugo
```

the program will end (correctly, though with no evidence on your side). Now, look at `greetings.out`; you will see:

```
Enter your name:
Hello, Hugo
```

In Unix, when you redirect the output, you may redirect the input too. To submit an input to a program, create a source, e.g. named `Hugo`, containing the string Hugo (that is the input). Then digit

```
./bf -q greetings.bf < Hugo > greetings.out
```

and that's all. The program behaves as usual, getting the input from the file `Hugo` (which should contain the exact sequence of needed data) and redirecting the output to `greetings.out`, in a pure and standard way, from the Unix point of view.

See the program ask.bf (and the input/output files inputask and outputask) for a more complete example; outputask has been obtained with

```
./bf -q ask.bf <inputask >outputask
```

Of course this subject is not completely satisfactory to me. What if I don't have the data to be written on the input file? What is I want to ask the user for data, to be input directly from terminal, but I want the same data written on file? In this case, I have to write data directly on file with **bf** commands, not through bash.

I'm working on it.

5.6 Hex, octal and binary numbers

Starting from v0.8.4 it's no more possible to input Hex, Octal or Binary numbers with `H`, `O` or `B` postponed to the number. Apart from the fact that that routine was full of bugs, (try to input `7FFFH` in 0.8.2!), it would have retained letters `A`÷`F`, `H` and `O` for this purpose. These letters could have been used neither as variables nor as comments.

So I removed this feature for now, though I'm really thinking to reintroduce in future the binary inputing. Why not, if I may need it?

Never mind! Only **forth** and **C** have a strong need to use other-than-decimal numbers, and only if they use memory location or are connected to some pin (often identified by a hex). **bf** doesn't.

6 bf quick reference guide

(from the `bf -V` execution:)

```
bf v0.9.4 QUICK REFERENCE VOCABULARY
Language key-chars
-----------------
!  exclamation mark ( a v - )  - store a into v (no chars between v and !)
:  colon ( v - a ) - leave v content on TOS (no chars between v and :)
@  at ( [f] | v - x*a ) - execute function [f] on TOS or in variable v
?  query ( f [f] - x*a ) - if-then; execute [f] if f
         ( f [f1] [f2] - x*a ) - if-then-else; execute [f1] if f, or else [f2]
#  sharp ( [f1] [f2] - x*a ) - while cycle; while [f1] execute [f2]
d  ( n2 n1 [f] - x*a ) - do loop cycle; execute [f] n2 - n1 times
   ( n2 n1 [f1] [f2] - x*a ) - do +loop; [f1] must leave a signed integer
:  colon ( - i ) - if used inside a loop, leave current loop index on TOS
h  halt cycle; must be used into [f2] of # and d, (see while and do-loop)
%  percent ( a - a a ) - dup
;  semi-colon ( a - ) - drop
$  dollar ( a1 a2 - a2 a1 ) - swap
_  underscore ( a3 a2 a1 - a2 a1 a3 ) - rot
r  ( x*a i - y*a ) - roll, rotate ith element
p  ( x*a i - x*a a ) - pick, copy ith element
:  colon - show stack (it must be preceeded by any non-alphabetic char)
e  ( x*a - ) - empty stack
}  right curly bracket ( - i ) - leave current stack depth
g  ( i1 - i2 | ) - leave random number between 0 and i1, or set seed if i1 = 0
*  star  ( n1 n2 - n1*n2 ) - (integer) multiplication
+  plus  ( n1 n2 - n1+n2 ) - (integer) addition
-  dash  ( n1 n2 - n1-n2 ) - (integer) subtraction
/  slash ( n1 n2 - n1/n2 ) - (integer) division
m  ( n1 n2 - i ) - modulo (integer remainder of n1/n2)
^  caret ( n1 n2 - n3 ) - leave n1 to the power of n2
!  exclamation mark ( n1 n2 - n3 ) - leave n1 to the root of n2
\  backslash ( n - -n ) - negate TOS
=  equal ( a1 a2 - f ) - True if a1 = a2
>  greater than ( a1 a2 - f ) - True if a1 > a2
&  ampersand ( i1 i2 - f ) - logical and
|  bar ( i1 i2 - f ) - logical or
~  tilde ( f1 - f2 ) - logical not
s  ( n - d ) - sin (n in radians)
c  ( n - d ) - cos (n in radians)
t  ( n - d ) - arctan (n in radians)
l  letter l ( n1 n2 - n3 ) - leave logarithm in base n2 of n1
a  ( i1 i2 - i3 ) - bitwise and
o  ( i1 i2 - i3 ) - bitwise or
n  ( i1 - i2 ) - bitwise not (1s complement)
x  ( i1 i2 - i3 ) - bitwise xor
v  ( i - d | d - i ) - convert an integer to a float and vice versa
   ( "s" - [f] | [f] - "s" ) - convert a string to a function and vice versa
,  comma ( "s" - n ) - convert string on TOS to number (integer or float)
         ( n - "s" ) - convert number on TOS (integer or float) to string
i  ( a - ) - print TOS on standard output
j  ( a i - ) - print a in a field of i chars on standard output (space padded)
k  ( n i - ) - if i >= 0, print n with i decimal digits (zero padded)
   ( d i - ) - if i < 0, possibily, print d with |i| digits (exponential form)
.  dot - emit a Carriage Return
<  lesser than ( - "s" ) - read from standard input leaving string "s"
(  left paren  ( "f" - x*a ) - include file "f" and execute it
               ( "f" i - "s" ) - if i > 0 leave ith line of file "f" as "s"
               ( "f" i - [f] ) - if i < 0 leave ith line of file "f" as [f]
               ( "f" 0 - i ) - leave number of lines of text file "f"
)  right paren ( "f" "s" - ) - append string "s" to file "f"
               ( "f" 0 - ) - empty file "f" (set it to zero length)
f  ( - "f" ) - leave current buffer file name
b  ( "f" - ) - set buffer file name to "f"
y  ( "f" - ) - yield stack content, in bf format, to file "f"
]  right square bracket - toggle appending input lines to buffer file
```

```
`  grave ( <stream> - c ) - leave on TOS ASCII code of next char
'  tick ( c - ) - print char whose ASCII code is on TOS
u  ( "s" - c ) - unveil first ASCII code of string "s", leaving it on TOS
   ( "s1" i1 i2 - "s2" ) - leave substring "s2" of "s1", from i1 > 0 to i2
   ( "s" i1 - i2 ) - if i1 < 0, leave length of string "s"
   ( "s1" "s2" 0 - "s1+s2" ) - leave union of strings "s1" and "s2"
~  tilde ( - yyyy mm dd ) - followed by d leave date values on stack
         ( - hh mm ss ) - followed by t leave time values on stack
         ( a - ) ( R: - a ) - followed by > move TOS to return stack
         ( - a ) ( R: a - ) - followed by < move top of return stack to TOS
         ( - a ) ( R: a - a ) - followed by : copy top of return stack to TOS
"  quote ( <stream> - "s" ) start a string until first " or EOL; \ escapes "
[  left square bracket ( <stream> - [f] ) start a function until last ] or EOL
{  left curly bracket - start skipping text (comments) until first } or EOL
w  ( a - i ) leave token of a (0 = int; 1 = float; 4 = string; 5 = function)
Debugging commands
------------------
debug    toggle on/off debugging
see      ( <name> - ) show instructions linked to most recent <name>
voc      print a list of all defined words
where    print value of all non null variables
System flag commands
------------------
clearonerror ( - ) toggle on/off stack clearing on error
errorshow    ( - ) toggle on/off error showing
exitwarn     ( - ) toggle on/off exit warninf on stack not empty
quiet        ( - ) toggle on/off quiet mode
System commands
---------------
bye       the same as 0q (also exit, halt, quit)
cd        ( "s" - ) change to directory "s"
def       ( <name> <instructions> - )  - link to <name> the <instructions>
          following it until EOL; <name> is then stored into vocabulary
edit      ( "f" - ) edit file "f" with an editor (set in bf.h)
help      help on system commands
ls        list current directory (also dir)
now       print current date & time in C format
precision ( i1 - ) | ( 0 - i2 ) set/get max precision for floats
pwd       print working directory
q         ( n - ) quit with error code n (0 means success)
ref       show a quick language summary by screens (also words)
reset     reset all variables, erase all defined words and clear stack
savedef   ( "f" - ) save an image of defined words on file "f"
savesys   ( "f" - ) execute savedef + savevar on file "f"
savevar   ( "f" - ) save an image of variables on file "f"
sh        ( <stream> - ) execute shell command starting from next char to EOL
show      ( "f" - ) show content of file "f" using a pager (set in bf.h)
status    show some information about the running bf session
undef     ( <name> - ) - removes <name> from the vocabulary
version   ( - "s" ) - leave a string with the version number
```

7 Bibliography and Internet resources

7.1 Bibliography

[**1.1**] Antonio Chiffi, *Analisi matematica*, vol. I, Alceo 1982

[**1.2**] Jack W. Crenshaw, *Math toolkit for real-time programming*, CMP books 2001

[**1.3**] Brian Kernighan & Dennis Ritchie, *Linguaggio C* - II ed., Jackson Libri 1989 (Italian version of *The C Programming language* - second edition)

[**1.4**] Edward Conklin & Elizabeth Rather, *Forth Programmer's Handbook*, Forth, Inc 1997

7.2 Internet resources

[**2.1**] `www.geocities.com/phengkimving/index.htm`, a very interesting site in which many math subjects are exposed, with graphics and exercises.

[**2.2**] `http://web.mit.edu/sunsoft_v6.1/SUNWspro/DOC5.0/lib/locale/C /html/common/ug/ncg_math.doc.html`, a place in which to look for double arithmetic modeling. As stated in the introduction: "*This chapter discusses the arithmetic model specified by the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic ('the IEEE standard' or 'IEEE 754' for short)*".

[**2.3**] `http://en.wikipedia.org/wiki/Mathematical_constant`. Many math constants are reported, much more than you even need.

[**2.4**] `http://www.chemie.fu-berlin.de/chemistry/general/constants_en .html` and `http://www.astro.wisc.edu/~dolan/constants.html`. Many many physical constants are reported. Two good sites, if you like the matter.