bdcalc – a calculator for large natural numbers

bdcalc is a command-line calculator and mini-programming language that works with very large unsigned integers, the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ used to carry out computations in cryptography. The numbers can theoretically be of unlimited size.

bdcalc has built-in functions to carry out number theoretical computations such as modular exponentiation and modular inversion, as used in the RSA and Diffie-Hellman algorithms. You can generate random prime numbers of a given bit length and test for primality.

bdcalc provides a mini-programming environment with assignments to stored variables, conditional statements (if-then-else) and control loops (for, while, repeat). See Control Flow Statments.

You can use bdcalc in interactive mode, typing commands on the console until you type `quit` to exit. Alternatively you can read input directly from a script file using the `-file` option; or you can enter your input in one line on the command line, perhaps as part of a batch file.

Contents

1 A quick introduction

bdcalc lets you assign numbers to variables, do arithmetic and call functions in the way you would expect.

```
> a=99
99
> b=a*11
1089
> ? a+b
1188
> println("a=",a," b=",b)
a=99 b=1089
> p=genprime(128)
187188849449410847765185120249130623559
> !p
0x8cd342d39544246b9743c7b68d5da247
```

```
> bitlen(p)
128
> ??isprime(p)
true
```

Each line of input is evaluated as soon as the 'Enter' key is pressed. To type multiple statements on one line, separate with semicolon characters (`;`).

```
> x=1;y=2;z=x+y;?z
3
```

2 A compact summary

We know no one actually reads the manual, so here is a very brief summary of just about everything. If something needs more explanation, see the later section.

Numbers in bdcalc are unsigned integers of arbitrary length represented by default in decimal format. Precede a number by "0x" to specify in hexadecimal and by "0xb" to specify in binary (a string of '1's and '0's).

```
17  6189700196426901374495621111  0x11  0b10001
```

Strings are used only for printing and are enclosed either in single or double quotes `'abc' "hello, world!"`. Escape sequences like `\n` can be used inside strings to represent control characters. More details in the Strings section.

A **variable** name can be any combination of letters, numbers and underscore characters, but the first character must be a letter. Variable names are case sensitive. Reserved keywords cannot be used as variable names. Use an assignment to store a value in a variable, e.g. `a = 42`. Variables are mutable and are zero unless assigned to. Only numbers can be stored.

A single **statement** is terminated by a newline character (i.e. pressing the 'Enter' key). To type multiple statements on one line, separate with semicolon characters (`;`). Use a backslash (\) to continue a single statement on more than one line.

All **expressions** evaluate to a number. Typing a valid expression will display its value. Expressions can contain the arithmetic operators: + to add, * to multiply, - to subtract, and / to divide, along with parentheses (...). Only natural (unsigned) integers are permitted, so no negative numbers, fractions or floating-point numbers.

Subtraction is *cut-off* subtraction where the result is always zero or more. **Division** is *integer* division in which the fractional part is discarded. The operator `div` is a synonym for `/`. The `mod` operator as in `a mod b` gives the remainder when a is divided by b. Use the `%` operator as a synonym for `mod`. Use `x++` and `x--` to increment and decrement the value of a variable (subject to cut-off at zero).

The **bitwise** expressions `a << n` and `a >> n` perform bitwise left and right shifts, respectively, of `a` by `n` bits. Use the bitwise operators `&`, ∧ and `|` to perform bitwise AND, XOR and OR of two numbers, respectively (converting them both to bitstrings beforehand,

padded with zero bits to the left if necessary). The operators `shl`, `shr`, `band`, `bxor` and `bor` are synonyms for these bitwise operators. To invert the bits in a number use the `compl()` function.

logical expressions are used in comparison operations. An expression `A` is `false` if `A` is equal to zero; otherwise it is `true`. A logical expression returns 1 if it is true and 0 if false. Use the keywords `true` and `false` to represent true and false, e.g. `B = true`. Construct logical expressions using the comparison operators `==` 'equal to', `<=` 'less than or equal to', `!=` 'not equal to', `>=` 'greater than or equal to', `<` 'less than' and `>` 'greater than'. Use the logical operators `and` (`&&`), `or` (`||`) and `xor` to specify multiple conditions for a logical expression, or `not` to negate a condition.

The statement `?X` displays the single number or string `X` followed by a newline. To display a number `X` in hexadecimal use `!X` and in binary use `` `X `` (the backtick character). To display the result of a logical expression as `true` or `false` use `??A`. Synonymns for these quick-print statements are `puts X`, `puthex X`, `putbin X` and `putbool X`.

The function `println(S,T,...)` displays the numbers or strings in the list `S,T,...` followed by a newline. The `print()` functions does the same except without the newline character. Numbers are displayed by default in decimal. To format a number `X` inside a print list, use the special functions `hex(X)`, `bin(X)` or `bool(X)`. These special functions only work inside a print list.

For more advanced output, use the `printf("format",S,T,...)` function, which outputs a formatted string according to the 'format' string with number or string arguments `S,T,...`. The format string is of the form `%[flags][width][.precision]specifier` similar to that in C and Java, except without the `+` flag. Use `%d` and `%s` to display numbers and strings, and `%x`, `%b` and `%q` to display the value of a number in hexadecimal, binary and boolean, respectively. There are more details in the printf() section

The **conditional statements**

```
if B then <stmts> fi
if B then <stmts1> else <stmts2> fi
```

define code to be executed if a specified condition is true. The `<stmts>` blocks can contain multiple statements separated by semicolons. Don't forget the `fi` keyword!

The **while** statement

```
while B do <stmts> done
```

loops through `<stmts>` while the logical condition `B` is true. The **repeat** statement

```
repeat <stmts> until B
```

loops through `<stmts>` until `B` is true.

The **for** statement

```
for X in (M,N,...,W) do <stmts> done
```

loops through `<stmts>` for each value `X` in the comma-separated list `M,N,...,W`.

The "for-X-in-a-range" loop

```
for X in (M..N) do <stmts> done
```

loops through `<stmts>` with the variable `X` incrementing (or decrementing) by one each time in the range `M` to `N`. Use the range operator `..` literally as in `(2..7)`.

You can break a "for-X-in-a-range" loop with

```
for X in (M..N) do <stmts> breakif B done
```

which breaks if condition `B` is true. The `breakif B` statement can only be used as the last expression before the `done` keyword.

The built-in **functions** take arguments as specified in the Functions section. Most functions require an exact number of arguments, but some will take a list, e.g.

```
gcd(27,81,90,243)
```

The names of all built-in functions are in lower case letters.

Use the **define** statement to create user-defined functions. The statement

```
define cube(x) = x*x*x
```

defines a function `cube()` that takes one argument and returns its cube.

A **comment** starts with either a `#` character or the `//` pair. Everything following on the line is treated as a comment and ignored. Use comments in script files.

```
# This is a comment
a = 42 // this text is ignored
```

A **script file** is a plain ASCII text document containing valid statements, which are executed in order. Statements can be terminated by a newline character or semicolon. The statements in `if-then-else` and `for`, `while` and `repeat` loops can be broken and indented without using a backslash continuation character, but each statement in the `<stmts>` block must be terminated by a semicolon. The first part of the statement up to the `do` keyword must be on one line.

Use the `assert(B)` or `assert(B, 'msg')` statement to terminate a script file if a given condition is false. The `exit(N)` statement will terminate the script at the next available opportunity and return the value `N` to the operating system. Using `exit(N)` in interactive mode will terminate the session. Use `load 'filename'` to run a script in interactive mode.

The `help` statement will display a list of topics for which help is available. `help` ***keyword*** will display help on a given keyword (*Changed in v2.2:* quotes are no longer required around the keyword). Use `version` to display details about the current version of bdcalc.

The statement `system "string"` will pass the command `string` to the operating system, e.g. `system "dir"`. Use `getcwd` to find the current working directory and `chdir "path"` to change it.

Use `prompt "string"` to change the interactive prompt to `string`. Use `save` to save the values of all current variables to a file and `restore` to restore them.

The statement `verbose off` will turn off the display of values after typing a statement. `verbose on` turns it back on. The statement `showhex on` will display the "verbose" values in hexadecimal. `showhex off` reverts back to displaying in the default decimal.

3 Functions

3.1 Built-in functions

The following functions are built in. All keywords are in lower-case. The parameters `X`, `Y`, `M`, `B`, `n` represent expressions which evaluate to large integers of theoretically unlimited size, but `n` is expected to be "small".

Function	Returns	Remarks
`bitlen(X)`	the bit length ℓ of the integer `X`	$2^{\ell-1} \le x < 2^{\ell}$
`bytelen(X)`	the length L of integer `X` in bytes	
`cbrt(X)`	the truncated integer cube root of `X`	$\lfloor \sqrt[3]{x} \rfloor$
`compl(X,n)`	the bitwise NOT of the rightmost `n` bits of `X`	
`fillbytes(n,b)`	an integer of `n` bytes each set to `b mod 256`	
`gcd(X,Y,Z,...)`	the greatest common divisor of `X,Y,Z,...`	
`genprime(n)`	a random prime p of length exactly `n` bits	$2^{n-1} \le p < 2^n$
`getbit(X,n)`	value 0 or 1 of bit `n` of `X` $= (b_{\ell-1} \dots b_n \dots b_1 b_0)$	
`getbyte(X,n)`	value of byte `n` of `X`	
`iif(B,X,Y)`	`X` if `B` is true, otherwise `Y`	[Note 1]
`isprime(X)`	`true` if `X` is prime, otherwise `false`	[Note 2]
`jacobi(X,M)`	the Jacobi symbol `(X | M)` in $\{0, 1, 2\}$	[Note 3]
`max(X,Y,Z,...)`	the maximum of `X,Y,Z,...`	
`min(X,Y,Z,...)`	the minimum of `X,Y,Z,...`	
`modexp(X,Y,M)`	`X` raised to the power `Y` modulo `M`	$x^y \bmod m$
`modinv(X,M)`	the inverse of `X` modulo `M`	$x^{-1} \bmod m$
`modmul(X,Y,M)`	`X` multiplied by `Y` modulo `M`	$xy \bmod m$
`modpowof2(X,n)`	`X` with all bits cleared at positions $\ge$ `n`	$x \bmod 2^n$
`pow(X,n)`	`X` raised to the power `n`	x^n
`prime(n)`	the `n`-th prime number	$1 \le n \le 10^4$
`randbits(n)`	a random integer r of length at most `n` bits	$0 \le r < 2^n$
`random(M)`	an integer r generated at random from [0, M-1]	$0 \le r < M$
`revbytes(X,[n])`	`X` as an `n`-byte integer with byte order reversed	
`setbit(X,n,b)`	`X` $= (b_{\ell-1} \dots b_n \dots b_1 b_0)$ with bit `n` set to value `b`	$b \in \{0, 1\}$
`setbyte(X,n,b)`	`X` with byte `n` set to value `b mod 256`	
`sha1(X,n)`	the SHA-1 hash of the rightmost `n` bits of `X`	[Note 4]
`sha256(X,n)`	the SHA-256 hash of the rightmost `n` bits of `X`	[Note 4]
`sqrt(X)`	the truncated integer square root of `X`	$\lfloor \sqrt{x} \rfloor$
`square(X)`	the square of `X`	x^2

3.2 Notes

1. The `iif(B,X,Y)` function evaluates both arguments `X` and `Y` before returning. If you don't want this behaviour, use the `if-then-else` statement.

2. The functions `isprime()` and `genprime()` test for primality by using 64 iterations of the Rabin-Miller algorithm, which gives a probability of error no greater than 2^{-128}.

3. The `jacobi()` function returns 2 instead of the usual -1, since we cannot represent negative numbers. See Jacobi and Legendre symbol.

4. Strictly speaking, `sha1(X,n)` and `sha256(X,n)` convert the integer `X` to a bit string of `n` bits, padding to the left with zeros if necessary, compute the hash of this bit string, convert the 160-bit digest value back into an integer, and return this integer.

3.3 Examples of functions in use

Really useful examples use large numbers typically one or two thousand bits long. Unfortunately these make for tedious reading in a manual, so the examples here use small numbers to illustrate the functions. Some of the examples are perhaps a bit silly, but hopefully illustrate the core principle of the function concerned.

3.3.1 Greatest common divisor, `gcd`

The greatest common divisor (gcd) of $527 = 17 \times 31$ and $1003 = 17 \times 59$ is 17.

```
> gcd(527,1003)
17
```

If p, q, r are distinct primes, then the gcd of $u = pq$ and $v = pr$ is p.

```
> p=genprime(128)
> q=genprime(128)
> r=genprime(128)
> p;q;r
177567251864897131063391792740453364899
225909525713785216246109438136117714037
258078597510856248545436006399384610151
> u=p*q
> v=p*r
> g=gcd(u,v)
> g
177567251864897131063391792740453364899
> println("g==p is ",bool(g==p))
g==p is true
```

3.3.2 Modular inverse, `modinv`

One property of a modular inverse is as follows: if p is a prime, and $u = pm - 1$ for some integer m, then w, the inverse of u modulo p, equals $p - 1$ and $wu \bmod p$ equals one.

```
> p = 177567251864897131063391792740453364899
177567251864897131063391792740453364899
> m=random(1<<32)
> m
2205954264
> u=p*m - 1
> u
391705236398131778190655979498407345592096979335
> w = modinv(u,p)
> w
```

```
177567251864897131063391792740453364898
> w*u mod p
1
```

3.3.3 Integer logarithm to base 2

To compute the "truncated integer logarithm" g to base 2 of the integer N, so that $g = \lfloor \log_2 N \rfloor$, use the identity

$$\lfloor \log_2 N \rfloor = \texttt{bitlen}(N) - 1$$

For example, the integer logarithm to base 2 of all numbers in the range 32 to 63 is 5, but that of 64 is 6.

```
> for N in (32,33,62,63,64) do
   g=bitlen(N)-1;println("N=",N," lg(N)=",g) done
N=32 lg(N)=5
N=33 lg(N)=5
N=62 lg(N)=5
N=63 lg(N)=5
N=64 lg(N)=6
```

3.3.4 Jacobi and Legendre symbol

The function `jacobi(a,n)` computes the Jacobi symbol $(a \mid n)$, where the result 2 denotes the usual value of -1.

```
> jacobi(2813,9907)
1
> jacobi(1001,9907)
2
> jacobi(10000*9907,9907)
0
```

To work with products of Jacobi symbols, use the fact that $2 \equiv -1 \pmod 3$ and reduce the final product modulo 3 to get the equivalent result, e.g.

$$-1 \times -1 = 1 \Leftrightarrow 2 \times 2 = 4 \equiv 1 \pmod 3, \text{and}$$

$$1 \times -1 = -1 \Leftrightarrow 1 \times 2 = 2 \equiv -1 \pmod 3.$$

If n is prime then the Jacobi symbol becomes the Legendre symbol. In that case $(a \mid n) = 1$ implies that a is a quadratic residue of n; $(a \mid n) = 2$ (usually -1) implies that a is a non-quadratic residue; and $(a \mid n) = 0$ means that a is a multiple of n.

In the example above, since 9907 is prime the result 1 means that 2813 is a quadratic residue of 9907; that is, there exists an integer x such that $x^2 \pmod{9907} = 2813$. The script file `qr-find.bdscr` described below shows how to find x (*Spoiler:* the answer is $x = 3511$).

3.3.5 Powers of 2 and shift-left

2^{32} is equal to `1<<32` (or, should you prefer, `1 shl 32`).

```
> !a=pow(2,32)
0x100000000
> ??a==1<<32
true
```

But using `<<` is much faster.

3.3.6 `modpowof2`

Use `modpowof2` to truncate a bitstring. The function `modpowof2(X,n)` computes X mod 2^n, which is equivalent to doing a bitwise AND of X with $2^n - 1$, which gives the rightmost n bits of X.

```
> !x = 0xdeadbeef01facade
0xdeadbeef01facade
> !modpowof2(x,24)
0xfacade
> !x mod (1<<24)
0xfacade
> !x band (1<<24)-1
0xfacade
```

3.3.7 `setbit` and `getbit`

Use `setbit` and `getbit` to set or find specific bits in a bitstring.

```
> a = setbit(0,15,1)|setbit(0,0,1)
> putbin a
0b1000000000000001
> bitlen(a)
16
> getbit(a,15)
1
> getbit(a,14)
0
> getbit(a,0)
1
```

3.3.8 `compl`

The `compl(X,n)` function returns the bitwise complement (i.e. the bitwise NOT) of the rightmost `n` bits of `X`. The value of `X` is padded out to the left with zero bits or truncated, if necessary, before complementing. Note that the bitstring displayed by `putbin` does not show any leading zero bits.

```
> x=17
> `x
0b10001
> `compl(x,5)
0b1110
> `compl(x,6)
0b101110
> `compl(x,15)
0b111111111101110
```

3.3.9 `randbits` and `genprime`

Note that `genprime(n)` generates a random prime number of *exactly* `n` bits, so bit `n-1` is always set and the bit length of the result is always `n`. But `randbits(n)` generates a random integer of length *at most* `n` bits, so the bit length of the result may or may not be equal to `n`.

To ensure a random number of exactly `n` bits, either bitwise OR with `1<<(n-1)` or use `setbit` to set bit `n-1`. For example,

```
> n=128
> r = randbits(n)
> !r
0x54462a26347716a8cf6ee8c3c4fc20e
> bitlen(r)
123
> r = setbit(r,n-1,1)
> !r
0x854462a26347716a8cf6ee8c3c4fc20e
> bitlen(r)
128
> r = randbits(n) | 1<<(n-1)
> !r
0xaf456f1e4cc97de60abde8a2b667c8c4
> bitlen(r)
128
```

To ensure an odd random number, set the right-most bit using a bitwise OR with 1.

```
> r=randbits(128) | 0x1
> !r
0xae6f39f2e6225701338f8c77094ddf65
> r mod 2
1
```

3.3.10 `sha1` and `sha256`

Use `sha1(X,n)` to compute the 160-bit SHA-1 hash (message digest) of a bit string and `sha256(X,n)` to compute the 256-bit SHA-256 hash. The `n` rightmost bits of `X` are hashed, suitably padded to the left with zero bits if needed. You must specify the number of bits. In practice, `n` is usually a multiple of 8, representing whole bytes, but the function will cope with an odd number of bits if that is what you want.

```
> !sha1(0x616263,24)   # the 24-bit string representing ASCII "abc"
0xa9993e364706816aba3e25717850c26c9cd0d89d
> !sha1(0xa3,9)        # the 9 bits '010100011'
0xf582fa68b71ecdf1dcfc4946019cf5a18225bd2
> !sha1(0b010100011, 9)
0xf582fa68b71ecdf1dcfc4946019cf5a18225bd2
> !sha1(0,0)           # the empty string
0xda39a3ee5e6b4b0d3255bfef95601890afd80709
```

This example computes the SHA-256 digest of one million (1,000,000) repetitions of the character 'a' (0x61). Note the use of `verbose off` to avoid printing out the value of `b` (a huge number with 2.4 million digits).

```
> verbose off
> b = fillbytes(1000000, 0x61)
> ? bytelen(b)
1000000
> !sha256(b, bytelen(b)*8)
0xcdc76e5c9914fb9281a1c7e284d73e67f1809a48a497200e046d39ccc7112cd0
```

3.3.11 Byte manipulation functions

The byte manipulation functions treat the integer `X` as an array of bytes in network ("big-endian") order.

The function `bytelen(X)` returns the length `L` of the integer `X` in bytes, where `L` is the index of the left-most significant byte, numbered from 0 at the right. Equivalently, the *byte length* of an integer X is defined as the smallest integer L satisfying $X < 2^{8L}$.

Use `getbyte` and `setbyte` to query and manipulate the individual bytes in an integer, represented as a byte array, indexed from the right.

Use `revbytes(X,n)` to reverse the order of bytes, treating the integer `X` as an array of `n` bytes. You can use this to represent a number in "little-endian" order.

```
> !a = 0xdeadbeef42
0xdeadbeef42
> bytelen(a)
5
> !getbyte(a,1)
```

```
0xef
> !b = setbyte(a,1,0x00)
0xdeadbe0042
> !revbytes(a,5)
0x42efbeadde
> !revbytes(a,2)
0x42ef
> !revbytes(a,7)
0x42efbeadde0000
```

Changed in v2.2: The `n` argument in `revbytes` is optional. If omitted it defaults to `bytelen(X)`.

```
> !a = 0xdeadbeef42
0xdeadbeef42
> !revbytes(a)
0x42efbeadde
```

Be careful, the default behaviour of `revbytes(X)` may not be what you want if the leading byte in `X` might be zero.

3.3.12 `fillbytes()`

Use `fillbytes(n,b)` to return an integer of `n` bytes each set to `b mod 256`. (*New in v2.2.*)

```
> !d = fillbytes(11,0x61)
0x6161616161616161616161
> bytelen(d)
11
```

3.4 Integers, bytes, bit strings and bit length

We blur a few distinctions between an integer and its corresponding bit string here, and implicitly assume a conversion between them when it suits us to do so.

If $(b_{\ell-1}, \ldots, b_1, b_0)$ represents a bit string with $b_i \in \{0, 1\}$, where $b_{\ell-1}$ is the most significant bit and b_0 is the least significant bit, then the equivalent integer X is defined by

$$X = \sum_{i=0}^{\ell-1} b_i 2^i = b_{\ell-1} \times 2^{\ell-1} + \ldots + b_i \times 2^i + \ldots + b_1 \times 2 + b_0$$

We adopt the convention of indexing the bits from right to left with the right-most bit numbered zero.

The *bit length* of an integer X is defined as the smallest integer ℓ satisfying $X < 2^\ell$. Equivalently, the bit length of a bit string is the integer ℓ where $b_{\ell-1}$ is the left-most bit with value '1'.

Similarly, the byte manipulation functions treat the integer `X` as an array of bytes in network ("big-endian") order. See Byte manipulation functions.

4 Displaying values

Type `?X` to display a single number or string X followed by a newline.

```
> ? 2+3
5
> ? "hello world"
hello world
```

The function `println(S,T,...)` outputs the numbers or strings in the list `S,T,...` followed by a newline. The `print()` function does the same except without the newline. By default a space separator is inserted between each item in the list (*Changed in v2.2:* previously there was nothing).

```
> println("x=",42,"y=",43)
x= 42 y= 43
```

Use the `setsep` statement to change the default separator (*New in v2.2*).

```
> println(10,20,30)
10 20 30
> setsep ''
> println(10,20,30)
102030
> setsep ', '
> println(10,20,30)
10, 20, 30
```

The function `printf("format",S,T,...)` outputs a formatted string according to 'format' with a list of arguments `S,T,...`. See The printf() function.

To print a single number `X` in hexadecimal form, type `!X`. To print in binary form, use `` `X `` (that's the backtick character (`` ` ``)) and to print the boolean value `true` or `false`, use `??X`. These only work as "stand-alone" statements.

```
> a=17; ?a; !a; `a; ??a
17
0x11
0b10001
true
```

To format numbers inside a `println()` or `print()` function, use `hex()`, `bin()` or `bool()`. These only work as part of a print list.

```
> a=33; println(a," in hex=",hex(a)," in binary=",bin(a))
33 in hex=0x21 in binary=0b100001
```

4.1 The `printf()` function

The `printf()` function provides a subset of the standard C print conversion specification only for unsigned integers and strings.

```
%[flags][width][.precision]specifier
Format flags:
  '-' = left justify
  '#' = prefix (0x, 0X, 0b) on an (x, X, b) conversion
  '0' = pad with leading zeros
  (space and '+' not applicable)
[width][.precision]:
   as per standard, including '*'
Conversion specifiers:
  'd', 'i', 'u' = integer in decimal
  'x' = integer in hexadecimal (lower-case)
  'X' = integer in hexadecimal (upper-case)
  'b', 'B' = integer as bit string
  's' = string
  'q' = boolean (i.e. a Question?) "true" or "false"
  '%' = print a '%' character
(the length qualifiers 'h' 'l' 'L' etc are not applicable)

> a=33; printf("%d in hex=%x in binary=%b\n", a,a,a)
33 in hex=21 in binary=100001
```

4.2 Verbose mode and `showhex`

The default behaviour after typing a statement is to display the result in decimal.

```
> a = 123
123
```

Use `verbose off` to turn off this behaviour. You must then explicitly use a print function to display a value.

```
> verbose off
> b = 456789
> ?b
456789
```

Use `verbose on` to go back to the default behaviour. To display the "verbose" values in hexadecimal, use `showhex on`. To revert to displaying in decimal, use `showhex off`

```
> verbose on
> showhex on
> a
0x7b
> b
0x6f855
> showhex off
> b
456789
```

5 Control flow statements

5.1 Conditional statements: `if-then-fi` and `if-then-else-fi`

`if B then <stmts> fi`

executes the statements in `<stmts>` only if the boolean expression `B` is true.

`if B then <stmts1> else <stmts2> fi`

executes the statements in `<stmts1>` if the boolean expression `B` is true otherwise the statements in `<stmts2>` are executed.

The `<stmts>` block can contain multiple statements separated by semicolons.

Do not forget the `fi`!

```
> x=7; if x<8 then ? "is true" fi
is true
> x=8; if x<8 then ? "is true" else ? "is false" fi
is false
> x=8; if x<8 then ? "huh!" fi; ? "All done"
All done
```

5.2 Loops: while and repeat

`while B do <stmts> done`

loops through `<stmts>` while the boolean statement `B` is true.

`repeat <stmts> until B`

loops through `<stmts>` until the boolean statement `B` is true.

```
> x=0; while x<=5 do ?x; x++ done
0
1
2
3
4
5
> x=4; repeat ?x; x-- until x==0
4
3
2
1
```

5.3 Loops: for-do-done

```
for X in (M,N,...,W) do <stmts> done
```

loops through `<stmts>` for each X in the list `M,N,...,W`.

```
> for p in (2,3,6,7) do ??isprime(p) done
true
true
false
true
```

```
for X in (M..N) do <stmts> done
```

loops through `<stmts>` for X incrementing (or decrementing) the value of `X` by one each time in the range `M` to `N`

```
> for x in (2..5) do ?x done
2
3
4
5
```

```
for X in (M..N) do <stmts> breakif B done
```

loops through `<stmts>` for X incrementing by one in the range `M` to `N` stopping if the boolean statement `B` is true. The `breakif B` statement can only be used as the last statement before the `done` keyword.

```
> for x in (1..5) do ?x breakif x==3 done; println("stopped with x=", x)
1
2
3
stopped with x=3
```

Loops can be nested.

```
> for i in (2,5) do for j in (11..7) do
  println(i,' ',j,' ',pow(i,j)) done done
2 11 2048
2 10 1024
2 9 512
2 8 256
2 7 128
5 11 48828125
5 10 9765625
5 9 1953125
5 8 390625
5 7 78125
```

6 Strings

Strings are used in print statements and can be enclosed in either single or double quotes.

```
'abc'
"Hello, world!"
```

Use escape sequences like `\n` for newline, `\t` for tab and `\\` for backslash. To include a double quote `"` in a string enclose it in single quotes `'`, and vice versa.

```
> ? "abc\np\\qr\txyz"
abc
p\qr    xyz
> ? "abc'xyz"
abc'xyz
```

6.1 Escape sequences

Escape sequence	Meaning
newline	Ignored
\\	Backslash (\)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage return (CR)
\t	ASCII Horizontal tab (TAB)
\v	ASCII Vertical tab (VT)
\f	ASCII Formfeed (FF)
\b	ASCII Backspace (BS)
\a	ASCII Bell (BEL)

All unrecognized escape sequences are left in the string unchanged; that is, the backslash is left in the string.

To create a raw string, in which backslashes are not interpreted as escape characters, specify **r** before the opening quote character.

```
> ? r"A \raw stri\ng"
A \raw stri\ng
```

7 Expressions and operators

7.1 Arithmetic and bitwise expressions

These are listed in decreasing order of precedence.

Operator	Synonym	Description
`a * b`		multiply `a` and `b`
`a / b`	`a div b`	integer division: the quotient of `a` divided by `b`
`a % b`	`a mod b`	integer modulus: the remainder of `a` divided by `b`
`a + b`		add `a` and `b`
`a - b`		subtract `b` from `a` using cut-off subtraction: $a - b = 0$ if $b \geq a$
`a << n`	`a shl n`	bitwise left shift of `a` by n bits
`a >> n`	`a shr n`	bitwise right shift of `a` by n bits
`a & b`	`a band b`	bitwise AND of `a` and `b`
`a ∧ b`	`a bxor b`	bitwise XOR of `a` and `b`
`a | b`	`a bor b`	bitwise OR of `a` and `b`

Symbol and text synonyms like `<<` and `shl` are provided to allow the usual shorthand operators familiar to C and Java programmers and also to include a text form permissible in command-line prompts.

7.2 Relational and equality operators

The usual relational and equality operators are provided, plus text synomyms like `eq` for `==`, which are useful in command-line input (or if you just don't like the C-style ones).

Operator	Synonym	Description
`a < b`	`a lt b`	a is less than b
`a > b`	`a gt b`	a is greater than b
`a <= b`	`a le b`	a is less than or equal to b
`a >= b`	`a ge b`	a is greater than or equal to b
`a == b`	`a eq b`	a is equal to b
`a != b`	`a ne b`	a is not equal to b

7.3 Logical expressions and logical operators

An expression `A` is `false` if `A` is equal to zero; otherwise it is `true`. A logical expression returns 1 if it is true and 0 if false. To display the logical value of an expression A in text form `true` or `false`, use `??A`.

The assignment `A=1` will set `A` as true, and `A=0` will set `A` as false. In fact, setting `A` to any non-zero expression sets it as true as far as logical expressions are concerned. You can use the keywords `true` and `false` if you wish: `A=true` and `A=false`.

Operator	Description	Returns
`not A`	logical NOT	`true` if A is false; `false` if A is true
`A and B`	logical AND	`true` only if both A and B are true
`A or B`	logical OR	`true` if either A or B is true
`A xor B`	logical XOR	`true` if either A or B is true, but not both

The symbols `&&` and `||` can be used instead of `and` and `or`. There are no symbol synonyms for `not` or `xor`.

8 Saving and restoring session variables

8.1 The `save` and `restore` statements

```
save
save "filename"
restore
restore "filename"
```

The `save` statement will save the values of all current variables to disk. By default, these are saved in a file called `bdcalc_dat.txt` in the current working directory. Use the `restore` statement to restore all variables to those previously saved (you may need to hit Enter twice after using `restore`).

Alternatively, you can save to and restore from an optional named file.

```
save '\mydir\myvars.txt'
restore '\mydir\myvars.txt'
```

9 System-related statements

9.1 The `system` statement

```
system "string"
```

The `system` statement passes the string to be executed by the command processor. For example, in Windows

```
> system "start bdcalc.pdf"
```

will open the PDF manual file `bdcalc.pdf` using the default PDF viewer application.

```
> system 'dir'
```

will list the files in the current working directory. In Linux, use `system "ls"`.

9.2 The `getcwd` statement

The `getcwd` statement displays the name of the current working directory.

9.3 The `chdir` statement

```
chdir R"path"
```

The `chdir` statement changes the current directory to that specified in path.

Caution: Use the raw string format (`r'...'`) in Windows to avoid problems with backslashes and escape sequences like `\t`.

```
> getcwd
C:\ProgramData\DIManagement\bdcalc
> chdir r'C:\test'
> getcwd
C:\test
```

10 Miscellaneous statements

10.1 The `quit` statement

The `quit` statement terminates the program immediately. Use to quit interactive mode.

10.2 The `help` statement

```
help
help "keyword"
```

The help statement on its own gives you the usual generic bland statement about to get further help, with a list of keywords you can query further. `help "keyword"` outputs information on the specific keyword. Note the quotes around the keyword.

```
> help 'modexp'
`modexp(X,Y,M)` returns X raised to the power Y modulo M
```

But if you forget the quotes, you get an error

```
> help modexp
Type  help "<keyword>"  for specific help on a <keyword>.
...
ERROR: syntax error (incomplete line)
```

`help "all"` outputs help on all the keywords and `help "alls"` outputs the list with the keywords sorted.

10.3 The `version` statement

The `version` statement in interactive mode outputs information about the current version of the program. This is the same as typing `bdcalc -v` on the command line.

10.4 The `prompt` statement

`prompt "newprompt"`

Use the `prompt` statement to change the prompt in interactive mode. The default prompt is `"> "`.

```
> prompt "$ "
$ prompt '" '
" prompt "> "
>
```

11 Script files

A **script file** is a simple ASCII text file containing valid statements which are executed in order. We use the extension ".bdscr" but any valid text file is OK, including ".txt". The best language to highlight a script file in a smart text editor is Python.

11.1 The `-file` command-line option

You can load a text file containing any valid statements from the command line using the `-file` option. For example:

```
bdcalc -file myscriptfile.bdscr
```

```
bdcalc -file "My Dir\anotherscript.bdscr"
```

The program will exit on successful execution of the code or on error. There is no need for a `quit` statement. Make sure there is a final newline at the end of the file.

Suppose the script file `testin.bdscr` is

```
a=3
b=4
println("a+b=",a+b)
```

Then typing `bdcalc -file testin.bdscr` on the command line will give

```
bdcalc -file testin.bdscr
a+b=7
```

11.2 The `load` statement

```
load "filename"
```

Use the `load` statement in interactive mode to load and execute a script file. For example, using the same script file as above.

```
> load 'testin.bdscr'
a+b=7
```

11.3 Example scripts

For more detailed examples of using bdcalc to perform number theoretical and cryptography calculations, see the example script files on our web site (and included in the distribution). These script files include the following.

`qr_find.bdscr`
: How to find a quadratic residue by exhaustive search

`discretelog.bdscr`
: Compute discrete logarithm

`primes1000.bdscr`
: Write out the first 1000 primes

`rsa_make.bdscr`
: Make an RSA key pair and test it

`rsa_make_exact.bdscr`
: Make an RSA key pair of exact bit length

`rsa_quint.bdscr`
: Perform RSA calculation with private key in CRT quintuple form

`rsacrack.bdscr`
: Crack RSA if used incorrectly to three recipients

`rDSA.bdscr`
: Example of the rDSA algorithm from ANSI X9.31

`dsa_test.bdscr`
: Example of DSA from Appendix 5 FIPS PUB 186-2

`rsa_quint.bdscr`
: Perform RSA calculation with private key in CRT quintuple form

`dh_gen.bdscr`
: Generate domain parameters for Diffie-Hellman

`dh_keyexch.bdscr`
: Perform Diffie-Hellman key exchange using parameters generated above

`poly1305.bdscr`
: Poly1305 Example and Test Vector

11.4 Example script: Finding a quadratic residue

The script file `qr_find.bdscr` gives an example of doing an exhaustive search for a quadratic residue. It uses an outer `for` statement to loop through two values and an inner loop to search incrementally $2, 3, \ldots, p-1$ until it finds a result.

```
################################
# SCRIPT: qr_find.bdscr [v2.0] #
################################
# 2813 is a quadratic residue mod 9907 but 1001 is not.
p=9907
print("p=",p,"\n")
# Outer loop for two values of n...
for n in (2813,1001) do
  println("Testing n=",n,"...");
  for x in (2..p-1) do q=(modexp(x,2,p)==n); breakif (q) done;
  if(q) then
    printf("%d is a QR mod %d since %d^2=%d (mod %d)\n", n, p, x, modexp(x,2,p),p);
  else
    printf("%d is NOT a QR mod %d\n", n, p);
  fi
done
? "ALL DONE"
```

This gives the output:

```
p=9907
Testing n=2813...
2813 is a QR mod 9907 since 3511^2=2813 (mod 9907)
Testing n=1001...
1001 is NOT a QR mod 9907
ALL DONE
```

This example is just meant to demonstrate how you might use a loop statement in a script to find the actual number x for which $x^2 \bmod p = n$. In practice, you find whether or not `n` is a quadratic residue modulo a prime `p` simply by using the `jacobi(n,p)` function.

```
> jacobi(2813,9907)   # A quadratic residue => 1
1
> jacobi(1001,9907)   # A non-quadratic residue => 2
2
```

11.5 The assert statement

```
assert(X)
assert(X, "message")
```

The `assert` statement will terminate the program with an error message if the expression `X` is false. An optional message can be included as a quoted string. The program will return an exit code 1 to the operating system.

11.6 The exit(N) statement

The `exit(N)` statement will unconditionally terminate the program at the next convenient opportunity and return the exit code `N` to the operating system.

12 The command line

12.1 Command-line syntax

```
Usage: BDCALC [-file filename] | "stmt[;stmts]" | [-h] | [-v]
where
-file filename  load and run script in `filename`
"stmt[;stmts]"  execute statements and exit
-h              display this syntax and exit
-v              display version info and exit
(no argument)   start in interactive mode
```

12.2 Entering statements on the command line

```
bdcalc "stmt[;stmts]"
```

You can enter a block of statements separated by semicolons directly in one line on the command line. The statements in the block will be executed and the program will quit automatically.

Only the first argument on the command line is read, so you *must* enclose the entire block in double quotes `"..."` and avoid using any of the special Windows command-line characters like `>`, `&`, `<`, `|`, `\` and `^`. Use the text synonyms like `gt` and `band` instead. Use the apostrophe `'...'` to quote strings inside the block. In Linux, do the opposite and enclose the block in single quotes.

```
bdcalc "a=3;b=4;println('a + b =',a+b)"
a + b = 7

bdcalc "for a in (1..3) do ?a done"
1
2
3
```

```
bdcalc "a=0x10001;b=0x11110;printf('a band b=%#x', a band b)"
a band b=0x10000
```

As an example of using a loop, we can generate a random prime of 32 bits (the long way) on the command line as follows.

```
bdcalc "k=32;repeat p=randbits(k)|1<<(k-1)|0x1;
     println('Try p=',hex(p)) until isprime(p);!p"
Try p=0x988e56c5
Try p=0xf209e0d9
Try p=0xee42d1d1
Try p=0xd09edbc1
0xd09edbc1
```

Enter everything on one line between the surrounding quotes (we've broken the line above for display purposes). Note also that this example will give a different prime result each time and may take several tries before it succeeds.

Of course we could just do

```
bdcalc "!genprime(32)"
0xa543796f
```

but you get the idea.

13 Errors and error messages

13.1 Common errors

- Forgetting to close the quotes around a string, or mixing an opening quote `"` with a closing apostrophe `'`.

  ```
  > print "hello   # Forgot the closing quote

  "
  hello
  > print 'hello"  # Waiting for apostrophe (') to close

  '
  hello"
  ```

  In interactive mode the program will just wait until you have entered the closing quote, with no prompt. Any newlines you enter become part of the string. In file mode, the program will wait forever!

- `exit` doesn't work!

```
> exit
BDCALC ERROR: syntax error (incomplete line)
```

The correct statement to end the program in interactive mode is `quit` or `exit(0)`. The statement `exit(N)` (with a parameter in parentheses) is meant for use in script files to return the exit code `N` to the operating system.

14 Using on a Linux platform

14.1 Funny characters appear when using the arrow keys

To avoid "funny" characters like `^[[A ^[[B ^[[C ^[[D` when using the arrow keys on a Linux platform, install `rlwrap` and start bdcalc from the command line with

```
rlwrap bdcalc
```

This will make the arrow keys work as expected and give you a proper history of previous commands by using the 'up' arrow (just like on Windows!).

15 Reserved keywords

The following keywords, built-in function names and operator synonyms are reserved and cannot be used for variable names.

```
and assert band bin bitlen bool bor breakif bxor cbrt chdir compl define div
do done else eq exit false fi fillbytes for gcd ge genprime getbit getcwd gt
help hex if iif in isprime jacobi le load lt max min mod modexp modinv modmul
modpowof2 ne not off on or pow prime print printf println prompt putbin putbool
puthex puts quit randbits random repeat restore save setbit sha1 sha256 shl shr
showhex sqrt square start system then true until verbose version while xor
```

16 Revision history

Version 2.2.0

(17 June 2016) Changed behaviour of `print` and `println` to insert spaces between arguments. Added `setsep` command to change print separation character. Removed requirement to put quotes around arguments for `help`. Added new functions `sha256` and `fillbytes`. Made second argument optional for `revbytes`. Added `showhex` statement to switch between hexadecimal and decimal displays of the result in interactive mode.

Version 2.1.0

(19 January 2015) Added support for byte manipulation. See Byte manipulation functions.

Version 2.0.0
(24 October 2014) Version 2 is a complete rewrite of version 1 with significantly different syntax.

Version 1.0.0
(12 May 2013) bdcalc version 1 first published.

17 About this document

This document was last updated on June 17, 2016 for bdcalc version 2.2.0. It was created in LaTeX using TeXnicCenter and required the consumption of 1373 bottles of premium German lager.

```
> prime(220)
1373
> ??isprime(1373)
true
```

More information on bdcalc is available at

`http://www.di-mgt.com.au/bdcalc.html`