C-BC – Extended BC

Version 1.4
2016 September 10

0. Introduction

C-BC is an arbitrary precision calculator language that extends BC in several significant directions. It incorporates a much larger subset of C syntax, including an almost complete set of C statements and expressions. A type system, upwardly compatible with BC, is included in order to support complex number and finite Galois Field routines, as well as arrays of arbitrary dimension and pointers. An attempt has also been made to support most of the GNU-BC extensions.

The files to be read in can be listed in the command line. This is equivalent to calling up C-BC with no input files, and then typing a consecutive series of include statements. A compiler option also exists to read in the math library, which is provided with the C-BC package.

1. Comments

C++ commenting style is accepted. These include the two commenting styles

/* .... ***/**	traditional C comments.
// ... End Of Line	trailing line BCPL comments.

The first style of comment are treated by the interpreter exactly the same as a single space (except that a valid line count is maintained for error messages). The second type of comment, including the trailing end of line, is treated as an end of line marker.

2. Names

In C-BC, names are used for a variety of purposes. They can be used to denote objects of any of the four types supported by C-BC: **number**, **complex**, **galois** and **string** (referred to collectively as scalars), as well as pointers. In addition, arrays may be denoted by array variables. There is no support, as of yet, for function pointers however.

It's important to understand the naming convention used here, since it follows standard BC practice and differs significantly from C. The following restrictions apply

- names cannot begin in uppercase letters. (because upper case is used for arbitrary precision numbers)
- variables for scalars/pointers and for arrays occupy distinct name spaces. There are also distinct name spaces for functions and labels. In a concurrent extension to C-BC, there will also be a separate name space for signal labels.

What this means, in particular, is that it is possible to give both a scalar or pointer variable and array variable the same name, *a*. The language is designed in such a way that context alone will always distinguish which is which by the following conventions

- Any name followed by **[** is understood to be an array variable
- Any name followed immediately by **(** is understood to be a function name.

A consequence of this is that even though the expressions

(p* + *n***)** and *a***[***n***]**

are equivalent, when *p* == **&***a***[0]**, one cannot use the pointer syntax with the array variable, *a*, as in: ***(***a* + *n***)**, nor the array syntax with the pointer variable, *p*, as in *p***[***n***]**.

When the context does not make it clear, it is possible to use *a***[]** to denote the array variable, *a*. Also, like in C, arrays are converted to pointers in most contexts (basically the same corresponding contexts as in C). Therefore one does have the following equivalence

(a***[]** + *n***)** and *a***[***n***]**

There are a set of pre-defined variables and built-in functions that include the BC built-ins. The BC variables **scale**, **ibase**, and **obase** are predefined all as type **number**. They have the following meanings

Variable	*Default Value*	*Meaning*

scale	0	the number of decimals that certain calculations will be carried out to.
ibase	10	the conversion base for numbers on input.
obase	10	the conversion base for numbers on output.

Other variables exist that are automatically updated to reflect the last item written out. There is one for each scalar type as if the following declarations had been made

number last; complex lastc; galois lastg; string lasts;

They can be abbreviated, respectively, as **.n**, **.c**, **.g**, and **.s**. Furthermore, for the sake of compatibility with other versions of BC, **.n** can be abbreviated simply as **.**.

Built-in functions include the BC functions **scale(***x***)**, **length(***x***)**, and **sqrt(***x***)** as well as numerous other functions exclusive to C-BC. These will be described in more detail below.

3. Values

C-BC works fundamentally with the following types of objects: arbitrary precision decimal numbers, complex numbers composed thereof, and elements of finite fields. Limited support is also provided for strings.

Numbers (and complex numbers) can have arbitrary precision both in the integer part and in the decimal part. The decimal precision, however, is truncated beyond a certain maximum (specified by a run-time variable) when doing calculations.

The two characteristic features that define a number, therefore, are the number of digits in the number (its *length*) and the number of decimals (its *scale*).

Expressions of type **number** and **complex** will have a *scale*. For complex numbers this scale applies to the real and imaginary parts equally. The maximum scale for most operations is controlled by the variable **scale** which can take on any value (but is effectively limited to 720 in the current implementation, since that is the largest number size).

4. Simple Expressions

4.1. Expression and Statement Evaluation

Since this is an interpreter, all expressions and statements are executed as soon as possible after being read in. Expressions are evaluated using a stack-based protocol.

This interpreter closely follows BC/DC tradition in that it compiles its input first into a byte code and then executes the compiled byte-code when applicable. It departs from tradition in that there is no underlying DC language, and that both the compiler and interpreter are integrated into one program.

Since it pre-compiles its own code before interpreting it, there is also plenty of room for optimization to be made on a function-by-function basis both at the source level and the byte-code level. Only a few optimizations are present in this software. Future extensions could go much further with this.

Because of this protocol, it is possible to issue function definitions and variable declarations as with any compiled language. In fact, everything that you type is considered a function definition or variable declaration. Even when you type in a stand-alone expression or statement, this is treated as a definition for an unnamed function and compiled before being executed.

The reason this is brought up is to point out that it is possible to use any statement at the command level that can be used inside function definitions, including even a **goto**. However, one should also note that a **goto** cannot jump outside a function in C-BC. Also, a return statement executed in the command-level unnamed function will return the interpreter back to command mode ready. The only place a **goto** and or **return** would be effectively used at the command level, therefore, is inside bracketed statement groups e.g.,

```
{
   for (i = 0; i < 10; i++)
      if (p[i] == x) goto x;
```

```
        <- "Not found.\n"
        x:
}
```

4.2. Number and Complex

The simplest expression is a numeric constant, galois field constant or string. There are no complex constants, but numbers can be used as complex values, and there is a conversion function: **comp(r, i)** that constructs complex numbers out of pairs of numbers.

Numeric constants consist of digits out of the set: 0-9 and A-F and may contain a radix point anywhere. A radix point by itself, however, is not recognized as a number (presumably **0.0**), but is interpreted as the numeric variable, **last**.

Numbers are read in from the source file using **ibase** to determine which input base to use. This variable can take on any value from 2 to 16. The interpreter will truncate **ibase** to 2 or 16 if it is assigned any value outside this range, and will generate an error message. Since the base is already given, the C-like prefixes (0x for base 16, 0 for base 8) are not used.

Single digits will have the value of the digit (**A** = **10**, **B** = **11**, etc) regardless of the input base. In all other cases, digits that exceed the input base are converted to the largest digit of that base **(ibase - 1)**. For instance, **FF** will be converted to **77** if **ibase = 8**, but **F** will be left as is (equal to **15** base **10**). Thus, one can use **FF** to denote the largest 2-digit number in whatever base is used. The largest one digit number is always **ibase - 1**.

In the current implementation, numeric constants are limited by the size of the input line, which can be no larger than 256 characters (counting lines spliced onto the current line).

4.3. Galois

Expressions of type **galois** have a base and degree associated with them. These items are globally set (with default values, base = 2, degree = 1) and apply to all galois type objects at once. When doing galois calculations, one will therefore normally set these items first to define which finite field one is working in before initiating any calculations.

The current field parameters are set by the **field()** function. Only one set of field parameters can be active at any time, but galois field variables are not affected by the **field()** statement. That means two things

(1) Galois field variables assume an undefined semantics after a **field()** is executed.
(2) They retain their original value, if the original field parameters are restored and if the variable was not altered.

A galois constant consists of a leading **$**, a series of hexadecimal digits (**0-9**, **A-F**), and possibly a trailing **$**. These digits are interpreted as the coefficients of the galois number in its polynomial representation. This polynomial is processed by taking its coefficients modulo the field's base, and taking the polynomial itself modulo the field's polynomial base.

So for example, if the current field setting is base **p = 2** with base polynomial $\mathbf{g(x) = x^2 + x + 1}$, then the following is true: $\mathbf{\$10 = x}$, $\mathbf{\$33 = 3x + 3 = x + 1}$, $\mathbf{\$100 = x^2 = -(x + 1) = x + 1}$, and so on.

If the galois field base is larger than 16, then each digit is represented by its equivalent hexadecimal value, and succeeding digits are separated by spaces. In all cases, spaces can be used to separate digits in a galois constant.

4.4. String

String constants, like in ANSI-C, can consist of one or more consecutive sequences of characters, each enclosed in double quotes and separated from one another by white space. For example, the following

"this " "is "
"a " "test."

is treated the same as the string **"this is a test."** (where the backslash in the first line occurs right at the end of the line).

When processing string constants, double quotes can be quoted by preceding them with a backslash. For example, **"\""** is a string consisting of a double quote preceded by a backslash.

5. Expressions

5.1. Operators

A diversity of operators is available to create expressions from. Most of these operators are also present in ANSI-C and have the same meanings. Some are unique to BC, and some unique to this version of BC.

The description below uses the following abbreviations

e = expression, *v* = value name, *a* = array name, *f* = function name

I/O Expressions	**->** *e*, ..., *e*	Input
	<- *e*, ..., *e*	Output
Logic Expressions	*e***?** *e***:** *e*	Conditional
	e **||** *e*	Sequential Or
	e **&&** *e*	Sequential And
	!*e*	Negation
Relational Expressions	*e* **==** *e*	
	e **<=** *e*	
	e **<** *e*	
	e **!=** *e*	
	e **>=** *e*	
	e **>** *e*	
Assignment/Update and Sequential Expressions	e, e	The operators =+, =-, =*, =/, =%, =^ are recognized as valid alternatives.
	e = e	
	++e	
	--e	
	e++	
	e--	
	e += e	
	e -= e	
	e *= e	
	e /= e	
	e ^= e	
	e %= e	
Arithmetic Expressions	+e e + e e * e e ^ e	*e*^*e* is the exponentiation operator. Example: **2^4** is **16**.
	-e e – e e / e e % e	
Addressing/Pointer Expressions	*e new e	
	&e free e	
Indexed Expressions	e[e] f(e, ..., e)	
Other Expressions	(e v a[])	
Constants	number “string”	
	$galois$ 0	

The constant **0** is specially treated as a polymorphic constant. It can stand for the null pointer, the empty string (**“”**), the complex number **comp(0, 0)**, the number **0** itself, or the galois number **$0**.

5.2. Precedence

The expression precedence is, from lowest to highest

Associativity	**Operators**
	<-, ->
right	**,**
right	**? :**
left	**||**
left	**&&**
left	**!**
left	<, >, <=, >=, ==, **!=**
right	=, +=, –=, ***=**, /=, **%=**, ^=, =+, =–, =***, =/, =**%**, =^
left	(infix) +, –
left	***, /, **%**
right	^
right	(prefix) +, -
right	++, --

This precedence is different from C, and is chosen to respect POSIX BC compliance. In particular, the following

x = **0** < **1**

assigns **0** to *x*, instead of assigning **1** (= **0** < **1**) to *x*, as it would in C. Also, note that even though the I/O operators (<- and ->) have the highest precedence, it is still necessary to parenthesize comma expressions. For instance

<- *s* = 2, *s*, "\n"

will print out **22**, and

<- (*s* = 2, *s*), "\n"

will print out **2**.

To summarize, the main differences from C are the following

Differences general to BC:

- Exponents are represented with the ^ operator, as well as ^= and =^
- The alternative forms =+, =-, =*, =/, =%, =^ are recognized but are treated as single tokens as they would be in ANSI-C.
- Array variables must be explicitly followed by a **[**, and functions by a **(**, since arrays variables, functions and variables occupy disjoint name spaces in BC.
- The array indexing operation *e***[***e***]** explicitly requires the types *array***[number]**. Array indexing does not commute, as it does in C.
- Operator precedence for relationals and assignments are different than in C.

Differences specific to C-BC:

- Operators for doing I/O are added. They have a vague similarity to the C++ operators << and >>.
- Galois-field constants are part of the language.
- An array variable can be denoted as *a***[]**, to distinguish it from an arithmetic/pointer variable, *a*.
- Pointers cannot be indexed explicitly (e.g. *p***[***n***]**) because of this.
- Relational operators can be applied directly to strings, in a way compatible with C's **strcmp()** operator.
- Relational operators are defined pairwise between *all* pointers.

5.3. Types

The table below illustrates all the type restrictions involving these operators (don't you wish K&R would have done this too?). The notation is as in this example

c + c ⩳ **c** to mean **complex + complex** results in **complex**.

The following abbreviations are used

c	**complex**
n	**number**
g	**galois**
s	**string**
v	arithmetic type (**c**/**n**/**g**)
x	scalar type (**c**/**n**/**g**/**s**)
y	scalar or pointer type (i.e. everything but arrays)
p*T*	pointer to type *T*
p	pointer type
&*T*	L-value of type *T*
a*T*	array of type *T*
T	general type (except L-values **&***T*)
[*T*]	general type (except **&***T*) after type conversion

Most of these restrictions are natural since they follow from the semantics of the operators involved. For instance, *e* < *e* cannot be used with complex numbers or galois field numbers since neither of these domains are ordered.

Signature	**Comments**
-> &x**,** ...**,** &x ⩳ n	
<- x**,** ...**,** x ⩳ n	
y**?** [t]**:** [t] ⩳ t	Coercion is applied to make the last two operand types match.
y **log_op** y' ⩳ n	**log_op** ∈ { **||**, **&&** } The types of the operands need not match.

!y ⩦ n	
[t] **rel_op** [t] ⩦ n	**rel_op** ∈ { **<=**, **>=**, **<**, **>** } t = n, s, or p only. The types must match after coercion.
[y] **equ_op** [y] ⩦ n	**equ_op** ∈ { **==**, **!=** } The types must match after coercion.
t’**,** t ⩦ t	
&t **=** [t] ⩦ t	Coercion is applied to the second operand, as need be.
&v **op=** [v] ⩦ v	op ∈ { **+**, **-**, *****, **/** }
[v] **op** [v] ⩦ v	op ∈ { **+**, **-**, *****, **/** }
&p **op=** n ⩦ p	op ∈ { **+**, **-** }
p **op** n ⩦ p	op ∈ { **+**, **-** }
n **+** p ⩦ p	
p **-** p ⩦ n	
&t **inc_op** ⩦ t	**inc_op** ∈ { **++**, **--** }, t = v or p
inc_op &t ⩦ t	**inc_op** ∈ { **++**, **--** }, t = v or p
&n **%=** n ⩦ n	
n **%** n ⩦ n	
&v **^=** n ⩦ v	
[v] **^** n ⩦ v	
sign_op v ⩦ v	sign_op ∈ { **+**, **-** }
mem_op &p ⩦ p	mem_op ∈ { **new**, **free** }
***** pt ⩦ &t	
& &t ⩦ pt	
at **[** n **]** ⩦ &t	
(t **)** ⩦ t	This includes **(** &t **)** ⩦ &t
number ⩦ n	
"string**"** ⩦ s	
$**galois**$ ⩦ g	

The types of array variables, variables, and functions are determined by how the they were declared. When a function is called, its arguments will undergo conversion, so that effectively a rule like this is in force

f(([**p**$_1$]**, ...,** [**p**$_n$]) ⩦ return type of f.

5.4. Conversions

In all contexts where conversion is required, conversion is carried out in order to make two types match. For instance

[t] **+** [t] ⩦ t

means that the two types involved on the left hand side should be made to match to a common type (t) by applying type conversions. The result then has the indicated type, t. In the following case

t **+=** [t] ⩦ t

means that the type on the right-hand side should be converted to match the one on the left hand side, if necessary. The conversions recognized are the following

convert constant 0 to null pointer,
convert constant 0 to n, c, g, or s,
convert n to c,

An additional set of conversions

convert n to s,
convert s to n

will be included in a future revision, possibly using ibase and obase.

Two other conversions are understood to apply generally, except in certain contexts

(1) Reference values are dereferenced (&t ⩦ t) except where reference types are explicitly required:

r**++**, r**--**, **++**r, **--**r, **new** r, **free** r, **&** r, **->** r**, ...,** r.

Parentheses do not override this (e.g. (v)**++** means v**++**)

(2) Arrays are automatically converted to pointers (at ≡ pt) by: a[] ≡ &a[0], except in the following contexts:

a++, a--, ++a, --a, a = e, a **op=** e

a**[e]**, r = a (when r is an array type)

function arguments, wherever the function requires arrays.

Parentheses do not override this either.

6. Operator Semantics

All the operators common to C-BC and C have similar meanings, after taking into account that C-BC works with arbitrary precision numbers. The exceptions, as mentioned above, are ^ which is used to denote exponentials, and -> and <- which are used for I/O.

The semantics for the basic arithmetic operators, when applied to type number or complex expressions will have a semantics consistent with BC, where there are differences between C and BC. All of these differences will be described in more detail below on a case-by-case basis.

6.1. L-Values

Take special note to when and where L-values are required, or where they result. This information is part of the listing above in the section on operator types. An L-value will therefore have one of the following forms

*p a[n] v (l), where l is an L-value

and is required in the following contexts

-> l, l, ..., l

l = e l op= e l++ l-- ++l --l

new l free l &l

An L-value is always equal to an expression of the form *p (and vice versa). For instance, v is equal to *&v, and a[n] is equal to *(a[] + n).

6.2. I/O Operators

Both operators are defined analogously as follows

-> e, e1, ..., en means (-> e)? 1 + (-> e1, ..., en): 0

<- e, e1, ..., en means (<- e)? 1 + (<- e1, ..., en): 0

Both -> e and <- e will return 1 or 0 depending on whether the operation was successively carried out or not. In the current implementation of C-BC, <- e will always return 1.

This definition means that evaluation is carried out strictly left-to-right in a way completely analogous to the && operator, and that both expressions will return values equal to the number of items read in or written out.

The <- operator will write out the value of the indicated expression. This expression, as indicated above, can only be of a scalar type: number, complex, galois, or string. The following escape sequences are recognized and correctly processed in a way similar to ANSI-C

\b, \t, \n, \v, \f, \r, \\, \', \" and also \e for <ESCAPE>

Galois numbers are printed out with a leading $ but no trailing $, complex numbers are printed in the format: a + i*b.

As mentioned above, the variable “obase” is used to determine which base to write out numbers in (both expressions of type number and complex are affected by this). Galois numbers are written out using the current field setting in a way analogous to obase.

The -> operator will attempt to read a scalar value into the indicated variable. Strings are read up to the end of the line (the end of line marker is not included in the resulting string). Galois numbers are read in the same format as galois constants (described previously)

$ galois digits optional-$

Numbers and complex numbers are read in using “ibase” to determine which base to use. A complex number is read in one of the following formats

number

number (spaces) number

6.3. Logical Operators

In the following expressions

w? a: b w && w' w || w' !w

the expressions w and w' are interpreted as booleans, arrays are converted to pointers first, if necessary. The interpretation is that w is false if it is equal to 0, and true otherwise. In particular, in all the contexts above the following are equivalent

w and w != 0

The constant 0 is interpreted as follows

- **Numbers:** the number 0
- **Complex:** the number whose real and imaginary parts are both 0: comp(0, 0)
- **Galois:** the galois number $0
- **String:** the empty string, ""
- **Pointers:** the null pointer.

The operators have the following interpretations, consistent with their meanings in C (the truth values "true" and "false" are used to illustrate these interpretations, but are not part of the language)

- true? a: b ⇒ a
- false? a: b ⇒ b
- w && w' ⇒ w? (w'? 1: 0): 0
- w || w' ⇒ w? 1: (w'? 1: 0)
- !w ⇒ w? 0: 1 (also equivalent to: w == 0).

Thus in the last three cases, the result is always either the number 0 or 1.

6.4. Relational Operators

In the following expressions

a < b a > b a <= b a >= b a == b a != b

the expressions a and b are assumed to be of the same type. If necessary, 0 is converted to the appropriate type, numbers are promoted to complex numbers, and arrays to pointers.

The following interpretations apply

- a > b ⇒ b < a
- a >= b ⇒ b <= a
- a <= b ⇒ a == b or a < b
- a != b ⇒ !(a == b)

It is important to note that ***a < b*** and **!(*a >= b*)** are *not* equivalent when the comparison is between pointers. Pointers are considered to constitute a partially ordered set.

Note: *In future revisions this may be generalized to allow for linear ordering, so that the language can be defined in a compiled environment, rather than just an interpreted environment.*

6.4.1. a < b

Comparisons can only occur between strings, between numbers, and between pointers. Galois and complex type objects do not lie in ordered sets and so cannot be compared.

Numbers and strings occupy linearly ordered sets, with strings ordered lexicographically in a way compatible with C's strcmp() function. Linear ordering is characterized by that fact that a < b and !(a >= b) are equivalent. Thus, either a < b, a == b or a > b will be true.

Pointers are considered to occupy a partially ordered set. This set is structured as a partition of linearly ordered sets, called segments. Each segment is a single object, corresponding to a variable defined at compile-time, or a dynamic array that is either defined at compile-time in the program text, or at run-time using the memory allocation operator,

new. Thus, every pointer will be associated with a *base* segment and will have an *offset* in that segment, corresponding to the index of the entry it points to. The *null* pointer, for purposes of definition here, is considered to lie on its own segment with offset of **0**. It is possible that none of the relations a < b, a == b, a > b will apply, if the pointers are on different segments.

The following definitions thus apply for the < operator

- numbers: a < b if a is numerically less than b
- strings: a < b if strcmp(a, b) < 0
- pointers: a < b if a and b lie in the same array segment and a's offset is less than b's offset.

6.4.2. a == b

Equality comparisons can be made between any two objects of the same type. The meanings are basically the intuitive ones, with the following observations

- Numbers can be compared to complex numbers in view of the conversions noted above. Note: n == c means comp(n, 0) == c.
- For strings, a == b means the same as strcmp(a, b) == 0 in C.
- Any object can be compared to 0, with a == 0 being equivalent to !a
- Arrays are compared as if they had been first converted to pointers
- Two pointers are equal if and only if they lie on the same segment and have the same offset. But only pointers referencing the same type can be compared.

6.5. Pointer and Array Operators

Arrays in BC are markedly different than those in C in that they are dynamic. They are not declared with an explicit size but will grow as needed. In general, most implementations will impose a maximum index in arrays, such as 65535. This is the case here as well, but is not a necessary feature of C-BC. Future revisions can eliminate this restriction or expand it. When used in conjunction with allocated dynamic arrays, this strategy, which essentially amounts to a form of language-level caching, may be called *Lazy Allocation*. Memory is allocated for objects only as needed, so it is possible to already have allocated memory for a large number of dynamic arrays which combined would overwhelm the system's memory capacity if the space were actually reserved for them on allocation.

Arrays are essentially initialized to an infinite array of 0's when first declared or allocated. Those arrays that are declared inside functions will be freed as the function ends, along with all other variables local to the function.

Memory allocation is done with the operators **new** and **free** are used to allocate or deallocate pointers. They have the following effects

Operation	**Condition**	**Result**
new *p*	*p* **== 0**	Allocates a new dynamic array, *p* points to the start of the array segment (similar to malloc)
	p **!= 0**	No effect (similar to realloc, except that realloc is redundant in C-BC).
free *p*	*p* **!= 0**	Frees the dynamic array *p* points to, sets *p* to **0**. If *p* does not point to the start of a dynamic array, a run-time error results.
	p **== 0**	no effect (similar to free(0) in C).

The result of both operations is the value of the pointer *p* after the operation takes effect. In particular, both

p = **new** *p* and *p* = **free** *p*

are redundant.

The operators * and & are defined to be inverses. The following is true

- *&l == l for all L-values l.
- &*p == p for all pointers p.

The dereferencing operator, *p, will access the object pointed to by p. If p is the 0 pointer, a run-time error results. If p points to a simple variable, v, the value of *p is v itself. Finally, if p points to a dynamic array (call it a[]) with an offset (call it n), then *p is equal to a[n].

This form of equality is known as aliasing. The reason is that *p can be assigned to like any other variable, and when it is the result is that the object that *p is equivalent to is assigned to. For this reason, the following would result in the number 1

(y = 0, x = &y, *x = 1, y)

whereas the following results in the number 0

(y = 0, x = y, x = 1, y)

The array indexing operation, a[n] applies to expressions, a, of array type and indices, n, of type number. Any decimal part in the number is ignored. The result of this operation is equivalent to *(a[] + n) and is thus likewise an L-value. The operation p[n] can also be used for expressions p of pointer type, is equivalent to *(p + n), and is also an L-value.

Note: pointer variables lie in the same name space as scalar variables, while array variables lie in a separate name space. This means it is possible to have *both* an array and pointer variable with the same name, say, x. In that case, x[n] refers to the array variable x[], not to the pointer variable x; and for the latter, one must use the expression *(x + n) instead. In the expression *(x[] + n), however, x still refers to the array variable.

6.6. Arithmetic Operators

The following operators: + - * / % and ^ are used for arithmetic operations. The following combinations are possible with these operations

n + n → n	c + c → c	g + g → g
n - n → n	c - c → c	g - g → g
+ n → n	+ c → c	+ g → g
- n → n	- c → c	- g → g
n * n → n	c * c → c	g * g → g
n / n → n	c / c → c	g / g → g
n ^ n → n	c ^ n → c	g ^ n → g
n % n → n		

where n, c and g are objects of type number, complex and galois. The usual conversions (0 promoted to appropriate type, numbers converted to complex) are applied if necessary.

The meaning of the basic arithmetic operations + - * / for objects of type n, c, and g basically match the meanings of the real number (n), complex number (c) or galois field number (g) operations. Of special note is that all galois field operations are taken modulo the field's parameters.

There are differences, however, in that decimal precision (the "scale") may be truncated for operations on real numbers (and by extension: for complex numbers). These particulars are described below.

6.6.1. Multiplication

When numbers a and b are multiplied, the scale of the result is determined by the variable **scale**. However, it is guaranteed to be at least as large as the scale of a and b, and to be no larger than the true scale of a*b (which is scale(a) + scale(b)). So if necessary, the scale will be set to the maximum of scale(a) and scale(b), or to the true scale of a*b.

6.6.2. Division and Modulus

For numbers (and by extension, complex numbers), division and modulus are defined so as to satisfy the following conditions

- a/b = q, where q is a multiple of 10^{-scale} such that $|q\ b| \leq |a| < |q\ b| + 10^{-scale}\ |b|$ and the sign of q is the sign of a/b
- a%b = a - b*(a/b)

The scale of the quotient (a/b) is set by the variable **scale**, and the scale of the remainder (a%b) is set as if the equivalent operation a - b*(a/b) had been carried out.

6.6.3. Exponents

For real numbers, complex numbers, and galois numbers the exponent operator is essentially defined as repeated multiplication. The following conditions hold

- x^0 == 1
- x^(-n) == 1/(x^n) (or $1/(x^n)), if -n < 0

- x^(2n) == (x^n) * (x^n)
- x^(2n+1) == x * (x^n * x^n)

For numbers, the scale of the result is set as if these operations had been carried out. For negative exponents, therefore, it is set by the variable **scale**. For positive exponents, the same is true, but the resulting scale is limited below by scale(x) and above by n*scale(x).

The decimal parts of the exponent is ignored in this operation, so that

x^3.14 == x^3

6.7. Pointer Arithmetic

In addition to the combinations noted above, the following combinations are also possible

p + n ≠ p
p - n ≠ p
p - n ≠ p
p - p ≠ n

Arrays are converted to pointers in these contexts (a[] + 2 == &a[0] + 2).

The pointer operations n + p, p + n and p - n result in a pointer to the same segment as p, but whose offset is the offset of p + or - n. As with array indexing decimal parts to the number, n, are ignored.

The following restrictions apply to pointer arithmetic:

- If the pointer is the null pointer or points to a simple variable, pointer arithmetic is not valid and results in a run-time error. Other implementations of C-BC may choose to allow pointer arithmetic when the integer part of n is 0.
- If the pointer accesses a dynamic array, the result points to the same dynamic array and the offset is the result of performing the arithmetic operation on p's offset and n. Bounds checking is done and a run-time error results if the result is out of bounds. The result will fall out of bounds if the resulting offset is negative or if it exceeds the implementation-defined maximum index of dynamic arrays (which here is 65535).

The operation p - p is defined only if the two pointers access the same segment, with the following observations

- A run-time error results if they point to different segments
- Subtracting a null pointer or a pointer accessing a simple variable from itself is valid and results in 0.
- Subtracting from two pointers accessing the same array results in the corresponding differences of their offsets, e.g. &a[m] - &a[n] == m - n.

6.8. Assignment and Sequencing Operators

All of the operators

, = += -= *= /= %= ^= ++ --

will have similar meanings to the corresponding C operators, taking into account that ^ and ^= denote exponentiation, not exclusive or.

As stated in many C references

a op= b and a = a op b

are equivalent, except that a is evaluated only once. The type restrictions on these operators are the restrictions inherited by the equivalence. In particular, only the second expression, b, is converted so that the following

number op= complex

is not possible. Note that the more archaic operators =+ =- =* =/ =% =^ can still be used here, but are processed as single items (no intervening spaces allowed), so that

a =-b

subtracts b from a, but

a = -b

sets a to -b.

The assignment, a = b, will assign the value of the expression b to a. Both the types of a and b must match. If necessary, conversions are applied to b first (0 is promoted, arrays converted to pointers, numbers to complex). Note that arrays cannot be assigned to.

The resulting value of this expression is the value of b after conversion takes place, so as in C it is possible to string up assignments, like so

a = b = c *= d

The comma operator works exactly the same as in C. In the expression a, b a is evaluated first and the resulting value discarded. Then b is evaluated and the result is the resulting value of (a, b). This comes in handy in more tightly grouping related assignments, e.g.

c *= d, a = b = c

and is of essential use in the for-statement.

The increment operators ++ and -- work analogously to C increment operators, with the following notes

- Incrementing/decrementing a galois number is the same as adding/subtracting the galois number $1.
- Incrementing/decrementing a pointer is the same as adding/subtracting 1 to/from it.
- Incrementing/decrementing a complex number is equivalent to adding/subtracting comp(1, 0).
- Arrays and strings cannot be incremented or decremented.

7. Statements

All C statements are supported, except the switch/case/default statements (because I was too lazy to concoct a map strategy for switch statements).

The same syntax is accepted, except that semi-colons at the ends of lines do not need to be written. A consequence of this however, is that since C-BC will try to interpose semi-colons at the ends of lines, it will not usually be possible to split expressions or simple statements over more than one line. To remedy this, a backslash at the end of the line will explicitly override the conversion to semi-colons. More generally, as with ANSI-C, any backslash at the end of a line has the effect of joining the following line to the current one.

Because BC is an interpreter (and one of the reasons I decided to do the scanner and parser by hand), execution will proceed the very instant the final semi-colon (or end-of-line) of a stand-alone statement is read. The reason this is important is that like most compilers, C-BC will usually be one symbol ahead in where it is reading relative to where it is compiling. As an interpreter, special measures are taken to resynchronize the input.

There is one place where this can be aggravating: the if-else statement. Presumably, the reason “else” is not a part of POSIX-BC is that one symbol lookahead is always necessary to resolve “dangling-else” ambiguities. An interpreter would hang at the end of an if statement, because it is waiting to see if you’re going to type an else or not.

One way to avoid this problem in C-BC is to explicitly enclose the statement in braces

if (...) { ... }

since the semi-colon automatically placed at the end of the line is one symbol ahead of the rest of the if statement and is considered as a separate (null) statement. A consequence if this is that if you want to add an else-part to this statement, the “else” *must* be on the same line as the closing bracket. The following forms will be invalid for this reason

```
if (...) {
 ...
}
else

if (...) {
 ...
} // Comment: treated as an end-of-line
else
```

But the following will be correctly interpreted

```
if (...) {
 ...
} /* Comment: treated as a space
*/ else
```

```
if (...) {
}\
else
```

Also, the semi-colon is not required before a closing bracket, so that the following will be legal

```
{ return x }
```

Note that an expression used as a statement (similar to C expression statements) will be treated as an implicit request to print out the value of the expression, when this expression is printable. Anything other than relationals or assignments (or e?e1:e2 or e1,e2) with a scalar type is considered printable. Numeric constants are printed in the base specified by "obase", which can take on any value from 2 to 999,999,999. For bases greater than 16 each digit is written as a decimal number with each digit separated by spaces. Numbers beyond a certain length will be spliced between lines, using the \-escape.

Other, special, statements included are the following

- halt gets C-BC to exit if and when executed.
- include "FILE" inserts the named file past this point as if it had been entered in by hand as C-BC input.
- log "FILE" initiates logging of input to the named file.
- log turns off file logging.
- quit gets C-BC to exit when read.

8. Declarations

8.1. Contexts

Declarations can be made globally, outside any function. Global variables and array variables can be declared in this context, and functions can be forward-referenced. Unlike ANSI-C, there are no old-style function declarations or definitions at all. Also, all parameters must be explicitly listed and named in any function declaration. Future implementations can generalize this to make it conform more to ANSI-C.

In a global context, a declaration takes on the following form

type declarator, ..., declarator ;

Local variables and array variables can be declared inside functions. For purposes of defining "inside" and "outside", parameters are considered to lie inside the function.

A parameter declaration takes on the form

optional-type declarator

and a local variable one of the following forms

type declarator, ..., declarator ;

auto optional-type declarator, ..., declarator ;

Unlike in BC, the use of the word "auto" is optional, but if it is not present a type indicator must be made explicit.

Each declarator is understood to apply to one of the following types: number, complex, galois, or string, and any of these names can be used in place of "type" above. If a type is not explicitly indicated it is understood to be number by default, like in C.

Also, as in C, any variable or array variable that is used before being declared is understood to be of type number or array of number, respectively. Functions, however, must be explicitly declared before being used. This is a point of departure from both C and BC.

8.2. Declaration-Unwinding

A declarator can take on one of the following forms

(D)	used for grouping
name	simple declarators
*D	pointer declarators
D[]	array declarators
D(P, …, P)	function declarators

where D is a declarator, and the P's denote a comma-separated list of 0, 1, 2 or more parameters.

If a declarator, D, is understood to apply to scalar type s (number, complex, galois, string), then its interpretation is obtained by taking

s :: D

and applying the following reduction rules on it

T :: (D)	⇔	T :: D
T :: *D	⇔	pointer to T :: D
T :: D[]	⇔	array of T :: D
T :: D(p_1...p_n)	⇔	function of T_1, ..., T_n returning T :: D

where T_1, …, T_n are the types respectively of parameters p_1, …, p_n.

This process is called, here, *Declaration Unwinding*. The result of this reduction is an item of the form: T :: name, which is interpreted as the statement "name is declared as a T". The item, T, is called the declarator's type. In the reduction above, T_1 ... T_n are the declarator types that result from unwinding the parameter declarations p_1 through p_n.

Examples:

(1) number *a[]
⇔ number :: *a[]
⇔ pointer to number :: a[]
⇔ array of pointer to number :: a
(2) number f(x, complex y)
⇔ number :: f(x, complex y)
⇔ function of (number, complex) returning number :: f

Note the way precedence is handled in the first example. It is not possible to declare a pointer to an array, except by using brackets (e.g. (*a)[]).

8.3. Restrictions

As in ANSI-C, functions cannot return other functions or arrays. But in addition, in the current version of C-BC, pointers to functions are not allowed. A second restriction to note is that functions can only be declared in a global context. There is, in general, nothing in C-BC analogous to global objects with static linkage (i.e. global objects that are declared and visible only in a local context).

Repeated declarations, however, are allowed as long as the types match any previous declarations.

9. Functions

9.1. Definitions

A function definition takes on the following form

define *function-declarator* **{** *auto-declarations statements* **}**

The declarator can be any declarator whose type (determined by unwinding the declaration) is a function type. Note that the type indicator is optional in this context and defaults to **number**, e.g.

define *f*(*x*) **{** ... **}**

means

define number *f*(*x*) **{** ... **}**

Unlike the case in C, a function can be defined and redefined. However, this process is more restrictive than in BC in that the function must have the same type in each definition. When a valid redefinition is encountered, the older one is discarded and the new one becomes current.

Also, as noted above, parameters are technically considered to reside "inside" the function. Therefore, no pair of parameters/locals can share the same name, as in this example

define *f*(*x*) **{ auto** *x***;** ... **}**

9.2. Parameters

When a function is called, all parameters are passed by value, except array parameters. Arrays are passed by reference. The parameter's type must match that declared for the function. Conversions will take place as needed (0 promoted to the appropriate type, numbers converted to complex, arrays to pointers).

Take note that when you call a function with an array parameter, the array parameter must be a type array expression. In particular, if an array variable is used, it must be denoted in the form, *a***[]**.

A type mismatch in the parameters of the function call, or a mismatch in the number of parameters results in a run-time error. The function must be defined before it is called, else a run-time error results here too.

Note that the variable **ibase** is restored when a function call is completed, so it is effectively a parameter.

9.3. Auto Variables

Variables declared inside a function have a similar behavior to the auto variables of C. They are created when the function is called, and removed when the function returns. For array-type variables this has the effect of allocating the array at the beginning as if (**new** *a***[0]**) had been called, and of freeing up the array at the end as if (**free** *a***[0]**) had been called.

All auto variables are initialized to **0** when first declared. All auto array variables are effectively initialized to an infinite array of **0**'s when first declared. This is also true of global variables. Function parameters, of course, are initialized by whatever parameters are used in the function call.

9.4. Function Body

The body of a function consists of a set of declarations for auto variables, followed by a set of empty statements. Either set or both can be empty. When a return statement occurs inside a function it can take on any of the following formats

return
return ( )
return *e*

The first two types of return statements are considered equivalent to a **return 0**.

At the end of each function, there is also an implicit return statement (which likewise means: return 0). Therefore, all functions will return a value. There are no void functions in BC, nor in C-BC.

The expression type in the third kind of return statement must be compatible with the function's declared return type. If necessary it will be converted (0's promoted, numbers converted to complex, arrays to pointers).

9.5. Built-Ins

Already built into C-BC is a set of basic functions that includes the BC built-in functions as a subset. These functions are listed below and have the types implied by the corresponding declarations.

9.5.1. Numeric

number length(number *x***)**	the length of *x*
number scale(number *x***)**	the scale of *x*
number sqrt(number *x***)**	the square root of *x*

Note that square roots of negative numbers are not processed as complex numbers, but instead lead to run-time errors.

9.5.2. Complex

complex comp(number *x***, number** *y***)**	the complex number *x* + *iy*
number re(complex *z***)**	x, where *z* == *x* + *iy*
number im(complex *z***)**	*y*, where *z* == *x* + *iy*

9.5.3. I/O

number ifile(string *name***)** Defines the named file to be the file the input operator **->** will read values from; **ifile(0)** will set the input file back to standard input.

number ofile(string *name***)** Defines the named file to be the file the output operator **<-** will write values to; **ofile(0)** will set the output file back to standard output.

number afile(string *name*) The same as **ofile**, except that values are appended to the file, if it already exists.

number shell(string *cmd***)** Executes the shell command, *cmd*. Equivalent to **system(***cmd***)** in C.

All 3 functions return **0** on failure, and **1** on success.

9.5.4. Galois Field Operations

number field(*p***,** *m***,** *gx***[])** Sets the current Galois field to GF(p^m) with residue *gx***[0]**...*gx***[***m***-1]**. In this field, the polynomial x^m is equivalent to the polynomial

$$g(x) = gx[0] + x\ gx[1] + x^2\ gx[2] + ... + x^{m-1}\ gx[m-1].$$

This function will return one of the following values

0	The field conversion was successful
3	*p* was not prime
4	*p* exceeded the implementation-imposed limit (65536)
5	The polynomial x^m - *g(x)* is not irreducible.
7	p^m exceeded the implementation-imposed limit (2^{256})
8	*m* is too small (*m* < *1*)
else	Undefined error

number gal_gx(*gx***[])** Returns the degree, *m*, of the field, and the residue in *gx***[]**.

number gal_p() Returns the Galois field base, *p*.

number gal_m() Returns the Galois field degree, *m*.

The initial field setting is GF(2^1) with *gx[0] = 1*.

9.5.5. Galois Number Conversions

galois gal_set(number *d***, number** *p***[])** Returns the Galois field number obtained from the degree-d polynomial *p*[0] + *p*[1] *x* + *p*[2] x^2 + ... + *p*[*d*] x^d. Note: this polynomial is *truncated*, not converted via the field's parameters, so that a degree, *d* > **gal_m()** will will be treated as effectively as if *d* == **gal_m()**. However, the coefficients will be converted via the field's base, *p*.

number gal_get(galois *g***, number** *p***[])** Returns the coefficients of the Galois polynomial representing *g* in the array *p*[0], *p*[1],..., *p*[*d*], and returns the degree, *d*. Implementations may allow for leading **0**'s, so that *d* can be as large as **gal_m()**, itself.

9.6. Math Library

The C-BC math library can be included in your program by using the –l option. When the math library is loaded in, the default scale is set to 20. Contained in this library are the following functions:

s(*x***)**	*sine* (*x* in radians)
c(*x***)**	*cosine* (*x* in radians)
a(*x***)**	*arctangent* (result in radians)
l(*x***)**	*logarithm* (base e)
e(*x***)**	*exponential* (base e)
j(*n***,** *x***)**	*Bessel functions*

9.7. Additional Libraries

Over the course of time, with continuing use of C-BC, other libraries have emerged, including the following:

Stats.b	Basic routines for summary statistics
Matrix.b	Vector and matrix routines

These are undergoing continuing evolution. In addition, the way is being prepared with include routines compatible with those found in the AlgLib and LaPack libraries, the routines in the library for the Calc language, as well as

routines that generate scripts for doing multimedia composition. However, these will not likely be completed without first having support for multi-threaded computation and for string operations.

10. Scope and Linkage

10.1. Scoping

There is one significant difference between BC (and C-BC) and C, in the way scoping is dealt with. This difference is almost a universal characteristic that distinguishes compilers from interpreters. In BC, and C-BC *all variables are dynamically scoped*. This means that when a local variable is declared, it is accessible at *all* times while the function is active, even while another function is being called (unless it too has a local variable of the same name). Therefore, in this example

```
define f() { x = 1; }
define string g() {
   auto x
   x = 0, f()
   return x? “Dynamic scoping”: “Static scoping”
}
```

the call `g()` will return **“Dynamic scoping”**, not **“Static scoping”**.

10.2 Linkage

Another aspect of variables that basically coincides with the distinction between local and global is a variable’s linkage. In C-BC, the following observations apply

- All global variables have *static* linkage. They exist at all times, though they may be hidden during function calls by local variables of the same name.
- All local variables have *auto* linkage. They exist at all times while the function is active and (unlike C) can be accessed at any time the function is active unless another local variable of the same name has been declared and is active.

In the proposed concurrent extension to C-BC, this rule will be generalized so that all static variables will be treated as global to all active threads, and all auto variables will be treated as local to the thread it was declared in.

10.3 Dynamic versus Static Scoping and Type-Checking

C-BC, like many interpreted languages, is *dynamically scoped*, rather than *statically scoped*. This is an example in C-BC showing the effect of dynamic scoping. A similar issue arises in BC.

```
define b() { <- "i = "; return i; }
define a() { auto i; i = 1; return b(); }
i = 0;
<- "b() within a(): ",a(),"\nb() directly: ",b(),"\n";
// Result:
// b() within a(): i = 1
// b() directly: i = 0
halt;
```

Type-checking for dynamically scoped auto variables is not rigidly enforced at compile-time, in the current implementation of C-BC, but at run-time. Therefore, it is possible to obtain an *internal type-checking error*. This occurs, for instance, with the following example

```
define f() { x = 1; }
define g() {
   string x;
   x = "", f();
}
g();
```

Globals in one routine are bound to locals within the routine that called it. If the language is to be consistently typed, that would require that *all* variables sharing the same name – both local and global – either be declared to be same type or the language will *not* be strongly typed ... except at the cost of programs crashing wherever conflicts arise.

This is another example illustrating the inconsistency:

```
complex c;
define g() { <- "c + 1 = ",c + 1," in g().\n"; return 0; }
define f() { string c; c = "Testing"; void = g(); return 0; }
```

```
c = comp(0, 1);
c;
// Result: 0 + i* 1
void = g();
// Result: c + 1 = 1 + i* 1 in g();
void = f();
// Result: c + 1 = Internal type-checking error.
// *CRASH*
halt; // No need to actually halt since it already crashed.
```

Future implementations of C-BC may provide further checks to ensure type-safety, at compile-time, for dynamically scoped local variables or will add the ability to convert between numbers and strings.

11. Preprocessing

A small subset of ANSI-C preprocessing is also done in C-BC. This can be significantly extended in future versions of C-BC.

11.1. Line Splicing

Any line that ends in a backslash (\) is considered to be joined onto the following line. This comes in handy when it is desired to overrule the end-of-line as semicolon rule as would be necessary if one wanted to represent long numbers in a program.

11.2. Semi-Colons

In almost every context, an end of line will be treated as a semi-colon. This is of particular importance if you like formatting statements like so

```
if (...)
{
   ...
}
else
   ...
```

You need to use backslashes to avoid an accidental cut-off if the if statement.

11.3. Comments

BCPL comments // ... <end-of-line> are processed as end-of-lines (and thus may be regarded as semi-colons).

C comments /* ... */ are processed as single spaces (except that the line count is kept up to date for error messages). A consequence of this is that line ends contained inside a C comment are not processed as semi-colons.

11.4. Pseudo-Commands

As listed above, these include the commands

include “*FILE***”** **log** [**“***FILE***”**] **quit**

The first command is processed by inserting the named file immediately after the include command. A non-existent file leads to a fatal run-time error.

The second command is used to open up or close a log file. The purpose of this log file is primarily to record an interactive session. However, it will, in fact, record anything that C-BC reads in as program text (interactive or not) while active. If no file is listed, the logging becomes inactive. If the file listed cannot be created, a fatal run-time error results.

The third command is processed immediately and unconditionally as soon as it is read. Contrast that to the **halt** command, which will only be processed if and when it is executed. Both **quit** and **halt** bring to an end the C-BC interpreter when processed. However, **quit** is the older more traditional BC command, and **halt** only appears in some BC extensions.

11.5. String Joining

Strings may be split into one or more strings, each separated by spaces, just as in ANSI-C. A slash should be used before an end of line, however, if this extends over more than one line. The maximum length of a string, in the current implementation is 4096 characters. The size of each component is limited to 256 characters (in general, no object in C-BC, be it a number, string or name cannot exceed 256 characters).

12. Limits

It is a well-known fact that all machines in the real world are finite state machines, and most likely necessarily so. A consequence of this is that only a very restricted subset of the set of computable numbers can be represented in any implementation of C-BC no matter how many subterfuges one takes. For example, if you use something related to caching or lazy allocation to represent large numbers or arrays, you will run into problems with indexing. Pretty soon, the indices themselves have to be represented as large objects, leading to a second layer of indexing. Pretty soon, the barrage of layers will overwhelm the machine's capacity. Some of you may recall an essay written in Classical Greek times where a similar situation was confronted in attempting to name the myriads of large finite numbers using the Greek numbering system of the time.

In fact, Physics itself seems to imply a fundamental fact that any region of spacetime will only have a finite information carrying capacity, owing to the existence of the Planck scale. Naively, one might expect something on the order of 10^{300} bits per cubic meter storage capacity at speeds up to 10^{40} Hz. But the surprising fact is that the actual limitation seems to be proportional to the *area* of the region's boundary, rather than to the region's volume. This limit – known as the Bekenstein bound – equates 1 bit to ¼ of a Planck area, which is about 10^{-66} square centimeters.

As such, one has to live with limits of one kind or another. Since my goals in designing C-BC were not so much to come up with a truly powerful arbitrary precision arithmetic package (there's plenty of good ones out there), so much as to explore language design in this area and finite fields, the limits are fairly modest. They are described in detail below.

ibase	2 to 16
obase	2 to 999,999,999
scale	2 to 65535
Galois Field Limit	$p^m < 2^{256}$, $2 \le p < 65536$, $0 \le m$
Array Indexes	0 to 65535
Names	65536 per name space.
Maximum Token Size	256 characters
Maximum Number Size	256 characters, counting decimal point
Internal Number Length	1152 digits
Maximum String Size	4096 characters, 256 per component (counting \'s)
Local variables	256 parameters and auto variables per function.
Declaration depth	29 levels of pointers and array dimensions total.

13. Diagnostics and Error Recovery

C-BC will attempt to process and execute as much of the input as it can, after encountering errors. In no case will it print out more than a screen's worth of errors per function definition, or in a given execution sequence.

When an error is encountered in a function definition, the rest of the definition is processed but at the end the definition is cancelled. This may spawn future errors of the type "undefined function" down the line if input is not being read in interactively.

When an error is encountered during execution of a command-level statement, this statement is cancelled, all the local variables and arrays are cleared out and the interpreter is ready for the next statement is tried, if there is any. If there are more statements on the same line past the point of the error, C-BC will still attempt to execute them.

At any time, you can escape C-BC by typing your system's break command (usually control-C), though on some systems (such as DOS) there may be processing delays. The command **quit** will make C-BC end as soon as it is encountered.

14. History

1992 September	Stand-alone interpreter/compiler, new types. Originally a class project.
1992 December	Project rescinded prior to coding and brought under exclusive personal ownership and control.
1993 Spring	First design upgrade. Expanded statement and expression syntax.
1993 Summer	Second design upgrade. Pointers, arrays, expanded declaration syntax.
1993 Oct 3	Release of C-BC; announcement to comp.compilers.
2001 Oct 27	Re-release of C-BC; announcement to comp.compilers. Began reformatting the documentation.
2012 Jul 09	Recoding/reformatting of the programs in the Examples and Test directories.
2013 Mar 04	Continued reformatting the documentation.
2013 Jun 10	Recoding/reformatting of the Source code, upgrading to C99.
2013 Oct 23	Minor corrections to Source code, raised version number to 1.2.
2015 Jan 14	Added the afile() command.
2015 Jan 15	Raised the version number to 1.3. Completed reformatting the documentation.
2015 Oct 14	Minor edits of the documentation in preparation for a major revision/expansion of the language.
2016 Feb 10	Added the equivalences &p[n] = p + n and p[n] = *(p + n) for pointers p and numbers n.
2016 Sep 9	First release to GIT as “version 1.2”.
2016 Sep 10	(Retroactively) raised version number to 1.4 for 2016 Feb 10 upgrade.

15. Bugs

Over the years I've used C-BC extensively as a primary language, even more than C. These are the main problems I've seen crop up

Numeric & File I/O

- Internal errors can crash C-BC if the data being read is not of the correct format, as mentioned in 10.3.
- String reads never return a failure, even when at the end of file.

Complex Numbers

- The arithmetic has not been heavily used until recently.