Description of a BC Implementation

Partly derived from:
Case Study 2: How to build your own stack frames.
1995 August 26 17:34:48 GMT
comp.compilers

Actually, I'm not entirely sure if either this or the previous case study is entirely relevant to the original issue. However, the same basic idea is the same: derive it, don't design it.

The general idea is you start out with the recursive specification and then systematically derive the form and operation of the stack frame by eliminating the recursion.

The following is a design reference on the C-BC interpreter. In every place where possible the actual form of the interpreter was derived, including the byte-code language. However, only one derivation is explicitly specified at the very end: that of the call-return mechanism.

References:

Description of a new BC implementation	1993 Jul 06 17:45:58 PST	comp.compilers
A new implementation of BC: C-BC	1993 Aug 14 06:33:02 PST	comp.compilers
Case Study 2: How to build your own stack frames.	1995 Aug 26 17:34:48 GMT	comp.compilers
"New" Implementation of UNIX BC – Rerelease Of C-BC	2001 Oct 27 22:41:33 GMT	comp.compilers

1. C-BC Overview

The BC language is a standard facility of UNIX brought under the umbrella of the POSIX standard. It is a stack-based typeless calculator language that works only with arbitrary precision numbers and one-dimensional arrays of numbers and a very limited subset of C's syntax.

C-BC is an extension of the UNIX BC language which includes a nearly complete superset of C's statement and expression syntax, but is compatible with BC (with a few minor exceptions).

The revised language, C-BC, is a strongly-typed dialect of BC that significantly extends the facilities present in BC (as well as those in GNU BC) with the inclusion of most of the syntax for C expressions and control statements, including even the **goto**. It has multi-dimensional arrays, both static and dynamic, pointers with dynamic allocation and pointer arithmetic analogous to that in C, and arbitrary precision arithmetic with several base types for real numbers, complex numbers, and finite fields.

It is compatible with POSIX-BC, except for minor differences in the processing of strings (to bring about more features present in ANSI-C), and restrictions, related to the strongly-typed character of the language, on the declarations of functions. The latter set of restrictions follow the evolution of C: requiring now that functions be declared with prototypes and declared before first use, thus bringing the language closer in line with standard programming practice.

In practice, most BC programs can be run in C-BC with little or no change.

All the aspects of the implementation of C-BC are described in the sections to follow. None of the notes should be considered as part of the C-BC language specification, but rather as the specification of the way that this particular version of C-BC was implemented. Other versions are free to use their own methods, of course.

The original source code may be found in the alt.sources archives from early October of 1993.

2. Rational Terms

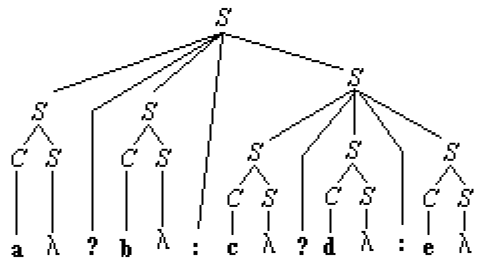
Source code is mapped into an infinitary language. The expressions in this language can be infinitely large, but in such a way that each contains only a finite number of distinct subexpressions, thus the term *rational*. This form of representation is in close keeping with the original spirit of the BC/DC language combination and is meant ultimately to serve as a front end for DC's continuation passing capability, thereby fulfilling the original intent of having BC serve as a front end to DC. That is, this form of representation will serve the end of “reprecating” the continuation-passing features that had previously been slated to be “deprecated”.

Given any grammar, one can extend the language represented by that grammar to include rational terms. The way this is done is to represent such a term by a rational tree: its parse tree. The grammar from which the C-BC object language is derived is

$$S \rightarrow \lambda$$
$$S \rightarrow C\,S$$
$$S \rightarrow C \mathbin{\textbf{?}} S \mathbin{\textbf{:}} S$$

where λ denotes the empty word (the ambiguity of the grammar will not present a problem for what follows below).

In general, a parse tree can be represented as a finite series of relations whose variables are the non-terminals of the grammar, but labeled. On the right hand side of each relation would be a copy of a one of the terms on the right-hand side of the corresponding grammar, with each non-terminal labeled. For example, one parse tree for **(a? b: c? d: e)** will be



with the corresponding system of relations

$$S_0 \rightarrow S_1 \mathbin{\textbf{?}} S_2 \mathbin{\textbf{:}} S_3$$

$$S_3 \rightarrow S_6 \mathbin{\textbf{?}} S_7 \mathbin{\textbf{:}} S_8$$

$S_1 \rightarrow C_1\, S_4$	$S_2 \rightarrow C_2\, S_5$	$S_6 \rightarrow C_3\, S_9$	$S_7 \rightarrow C_4\, S_{10}$	$S_8 \rightarrow C_5\, S_{11}$
$C_1 \rightarrow \mathbf{a}$	$C_2 \rightarrow \mathbf{b}$	$C_3 \rightarrow \mathbf{c}$	$C_4 \rightarrow \mathbf{d}$	$C_5 \rightarrow \mathbf{e}$
$S_4 \rightarrow \lambda$	$S_5 \rightarrow \lambda$	$S_9 \rightarrow \lambda$	$S_{10} \rightarrow \lambda$	$S_{11} \rightarrow \lambda$

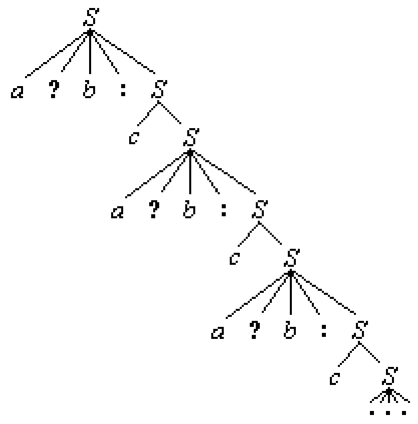
A parse tree for a finite term will not have any cycles in it. Therefore, the corresponding system of relations won’t either: no recursions.

However, when the language is extended to allow for rational infinite trees this will change. The parse trees will be infinite. But since a rational infinite tree has a finite number of distinct subterms, it can be represented as a “tree” with cycles in it. The corresponding system of relations, expressing the relations between the subtrees, will therefore be finite but cyclic.

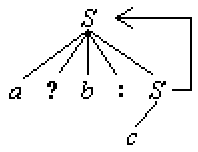
For example, the infinite expression

$$a\textbf{?}\ b\textbf{:}\ c\ a\textbf{?}\ b\textbf{:}\ c\ a\textbf{?}\ b\textbf{:}\ c\ a\textbf{?}\ b\textbf{:} \ldots$$

will have a tree that looks like this



that “folds into” (i.e. has, as a homomorphic image) the cyclic graph



which corresponds to the system of relations

$$S_0 \rightarrow^* a?\ b\!:\ S_1$$
$$S_1 \rightarrow^* c\ S_0$$

In turn, this structure corresponds to the sequence

S_0**: if (**a**) {** b**; return; } else goto** S_1
S_1**:** c**; goto** S_0

the archetypical example of a cyclic control flow structure.

In a similar way, all intra-procedural control flow in C-BC may be represented by rational terms, so that we need only translate the operators of the language. This makes the object language an order of magnitude simpler.

References:
Guy Cousineau "An Algebraic Definition for Control Structures", *Theoretical Computer Science* **12** (1980) 175-192.
Guy Cousineau "A Representation of Trees by Languages" *Theoretical Computer Science* **7** (1978) 25-55.

3. C-BC Parser

The parser handles a significant superset of BC that is close to ANSI-C. There are basically 3 components: the expression parser, the statement parser and the declarator & top level parser. For the most part, they can be isolated from each other, except for a minor hack to handle labeled statements and bracketing in return statements.

The development of the parser is an application of the Chomsky-Schuetzenberger Theorem, which states that any context-free language (or more generally: any context-free subset of a monoid, such as that described by SSDTs) can be written as a regular expression in which bracketing symbols are added, bracketing constraints separately imposed and then the brackets removed.

This is equivalent to freely extending the underlying algebra used with regular expressions to include additional symbols for brackets and subjecting the resulting algebra to the conditions that (a) the brackets commute with the input (and output) symbols and (b) that they satisfy certain algebraic identities as described below. In turn, an SSDT may, itself, be regarded as a context-free language in which the following additional relations are imposed: (c) that input and output symbols commute. Taking (a), (b) and (c) together yields, as a result, a seamless extension of the algebra of regular expressions to an algebra (actually: a calculus) that embodies context-free languages and (more generally) context-free subsets of arbitrary monoids where additional identities may be imposed.

For example, the syntax

$$E \rightarrow \mathbf{u}\ E \mid E\ \mathbf{p} \mid E\ \mathbf{b}\ E \mid \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{x}$$

is treated as a system of equations (or more generally: a system of inequalities where each rules A → B is corresponds to A >= B) involving regular expressions (with "actions" y_0, y_1, y_2, y_3 inserted); here just one:

$$E = \mathbf{u}\ E\ y_0 + \mathbf{(}\ E\ \mathbf{)} + \mathbf{x}\ y_1 + E\ \mathbf{p}\ y_2 + E\ \mathbf{b}\ E\ y_3$$

The right hand sides are made linear in the Qs:

$$E = Q_0$$
$$Q_0 = \mathbf{u}\ Q_1 + \mathbf{(}\ Q_3 + \mathbf{x}\ Q_5 + E\ Q_6$$
$$Q_1 = E\ Q_2$$
$$Q_2 = y_0\ Q_{10}$$
$$Q_3 = E\ Q_4$$
$$Q_4 = \mathbf{)}\ Q_{10}$$
$$Q_5 = y_1\ Q_{10}$$
$$Q_6 = \mathbf{p}\ Q_7 + \mathbf{b}\ Q_8$$
$$Q_7 = y_2\ Q_{10}$$
$$Q_8 = E\ Q_9$$

$Q_9 = y_3\ Q_{10}$

$Q_{10} = 1$

The recursive occurrences of the non-terminal symbols (E in this case) are eliminated and replaced by a set of Chomsky-Schuetzenberger bracketing symbols; and are thought of as "stack-actions"

p_n = "push symbol n"

q_n = "pop, if the symbol on top is n"

These definitions imply the following algebraic rules:

$p_n\ q_m = 1$, if $n = m$, 0 else.

The symbol, 0 is reserved exclusively for use as the bottommost symbol. After conversion, the result is

$E = p_0\ Q_0$

$Q_0 = \mathbf{u}\ Q_1 + \mathbf{(}\ Q_3 + \mathbf{x}\ Q_5 + p_4\ Q_0$

$Q_1 = p_2\ Q_0$

$Q_2 = y_0\ Q_{10}$

$Q_3 = p_3\ Q_0$

$Q_4 = \mathbf{)}\ Q_{10}$

$Q_5 = y_1\ Q_{10}$

$Q_6 = \mathbf{p}\ Q_7 + \mathbf{b}\ Q_8$

$Q_7 = y_2\ Q_{10}$

$Q_8 = p_1\ Q_0$

$Q_9 = y_3\ Q_{10}$

$Q_{10} = q_0 + q_4\ Q_6 + q_2\ Q_2 + q_3\ Q_4 + q_1\ Q_9$

Optimization rules may also be applied at this stage. For instance, left recursions can be eliminated by eliminating the stack symbol and combining states. States Q_0, Q_{10} and Q_6 would be transformed as follows

$Q_0 = \mathbf{u}\ Q_1 + \mathbf{(}\ Q_3 + \mathbf{x}\ Q_5 + Q_0$

$Q_6 = \mathbf{p}\ Q_7 + \mathbf{b}\ Q_8$

$Q_{10} = q_0 + Q_6 + q_2\ Q_2 + q_3\ Q_4 + q_1\ Q_9$

and then:

$Q_0 = \mathbf{u}\ Q_1 + \mathbf{(}\ Q_3 + \mathbf{x}\ Q_5 + Q_0$

$Q_{10} = \mathbf{p}\ Q_7 + \mathbf{b}\ Q_8 + q_0 + q_2\ Q_2 + q_3\ Q_4 + q_1\ Q_9$

with Q_6 being merged into Q_{10}.

Transitions involving actions (y_0, y_1, etc) and stack actions are treated as lambda transitions and are transformed away the same way as they would be for finite automata. The result is this

$E = p_0\ Q_0$

$Q_0 = \mathbf{u}\ p_2\ Q_0 + \mathbf{(}\ p_3\ Q_0 + \mathbf{x}\ y_1\ Q_{10}$

$Q_{10} = \mathbf{p}\ y_2\ Q_{10} + \mathbf{b}\ p_1\ Q_0 + q_0 + q_2\ y_0\ Q_{10} + q_3\ \mathbf{)}\ Q_{10} + q_1\ y_3\ Q_{10}$

with Q_1, Q_2, Q_3, Q_4, Q_5, Q_7, Q_8 and Q_9 all now also eliminated, along with the now-redundant Q_0 term on the right-hand side of the equation for Q_0.

State Q_{10} is a conflicting state. It allows you to take either one of the shift actions $\mathbf{p}\ y_2\ Q_{10}$ or $\mathbf{b}\ p_1\ Q_0$, or one of the remaining 4 actions, which are the reduce actions. Some of the reduce actions also involve shifts, such as $q_3\ \mathbf{)}\ Q_{10}$ and essentially q_0 which has an implicit shift of an end-marker.

An action table is created, which in this case amounts to a 4 × 4 table

(**p**, **b**, **)**, other) versus (q_0, q_1, q_2, q_3)

the table is filled out based whatever disambiguation rules exist. In the present case, the disambiguation rules are none other than the precedence rules of the various operators in the expression syntax.

The contexts corresponding to the stack symbols are respectively

y_0: E • (top-level)

y_1: E **b** E •

y_2: **u** E •

y_3 : **(** E • **)**

The contexts corresponding to the shift symbols are:

p: E • **p**
b: E • **b** E
): (E • **)**
other: E • (top-level)

each table entry is filled out by combining a shift context and reduce context and deciding which way to disambiguate. For example, the q_1 versus **b** entry corresponds to the context

E **b** E • **b** E

and is filled out with a shift action if **b** is right-associative, or a reduce otherwise.

Only the expression syntax will have an action table. The statement syntax and declarator syntax are close enough to being unambiguous that the conflicting states can be directly resolved without any tables.

4. C-BC Control Flow

The basis of the interpreter is therefore the rational string, which is presented as a series of items of the form:

$$(x\text{:E?y:}z)$$

Each item stands for a term/subterm relation according to one of the following schemes

- x, y, z are labels for subterms
- E is either NULL or is a rational string that contains no (S? S: S) in it.
- If E is NULL, $x \to 1$
- If $y \neq z$, $x \to E?y:z$
- If $y = z$, $x \to$ E,y

A special label is reserved to correspond to the **halt** statement, which will cause the interpreter to exit.

The way an item is interpreted corresponds directly to the meaning of the item being interpreted

(1) If E is NULL, the current execution sequence is ended, and control either resumes one level back (this is basically the **return** statement), or at the top level goes idle and waits for more commands and declarations.
(2) The sequence E is executed. Usually it is an expression to be evaluated, but may involve function calls, which recursively start up another level of execution. There are no jumps, breaks, returns or any other kind of control flow in E.
(3) If y or z is NULL, the interpreter halts and returns to the system (this is the **halt** statement).
(4) If y == z, then execution continues with the string labeled y.
(5) If y != z, then execution continues based on the value of the expression evaluated. If it is 0, execution continues at z, otherwise it continues at y.

All intraprocedural control flow can be handled. For instance, one might have a translation rule such as

while (E**)** S ↔ ... :c c) (c: E :x b) (x: S :c c) (b: ...

Inside this while-loop, breaks and continues would be translated as follows:

break; ↔... :b b) (y: ... where y is a new label
continue; ↔... :c c) (z: ... where z is a new label

where b and c are the same labels as in the translation of the while loop. Labeled statements are translated as follows:

L: S ↔... :L L) (L: S ...

and goto's referring to this label, as follows:

goto L; ↔ ... :L L) (x: S ... where x is a new label

Together, this allows one to define the entirety of a function's body. A function definition is translated as in the following example

define f**(**...**) {**	↔	(0 NULL - -) (1
E		E
while (A**)** S		2 2) (2 A 3 4) (3 S 2 2) (4
if (E**)** T		B 5 6) (5 T 6 6) (6
return C		C 0 0)
}		

which, when taken together forms the following set

(0: NULL? -: -)
(1: E? 2: 2)
(2: A? 3: 4)
(3: S? 2: 2)
(4: B? 5: 6)
(5: T? 6: 6)
(6: C? 0: 0)

This stands for the rational string E (2), which contains the infinite rational subterm

(2) = A? (S (2)): (B? (T C): C)

When the function is called, execution starts at the subterm labeled 1.

The significance of all this is that now it's possible to set up a purely expression based interpreter, involving no control flow statements at all – because control flow is directly represented by the very shape of the infinitary expressions. So the actual interpreter language has relatively few operators: basically just the numeric, stack, pointer, and array operators. This separates out control from evaluation.

The same factoring idea leads to a very similar architecture, which is an extension of the SECD machine that I call SEQCD. This was described in detail in the previous case study [that this section was originally a follow-up to, in the original comp.compilers article].

5. The C-BC Byte Code Language and Execution Machine

Though C-BC is not actually a functional language its execution machine is in fact the direct ancestor of the SEQCD machine. So there is a great deal of similarity between the two.

Following is a description of the object language used internally by the C-BC interpreter. This language was not concocted out of the blue but was *derived* as the least that would be necessary to effect a complete translation of the source language *and* make the corresponding translation grammar a SSDTG. Thus, it is specifically designed in such a way that translation from C-BC could be carried out in a strictly left-to-right manner.

To recap: an SDTG (Syntax Directed Translation Grammar) is one in which for each non-terminal, a set of transformation rules are stated of the form

N: a ➔ b

where the number of non-terminals of each type in the string a is the same as the corresponding number in the string b. The grammar is called an SSDTG (Simple SDTG) if the order of the non-terminals is also the same. For instance, YACC is technically a processor for an SSDTG, not for a context free grammar.

Any SDTG can be represented by an SSDTG in many ways, including the following two standard methods:

(1) Place intermediate values on a stack.
(2) Use the SSDTG to translate to a parse tree, and accomplish the reordering by traversing the parse tree after it's fully formed.

so as a result, SSDTG's assume a primary role. YACC uses the first method.

Likewise, the corresponding reduction rules were derived systematically from the condition of being the least which would be necessary to correctly interpret the action of the corresponding high-level statement. Much of this design could have, in fact, been automated.

Each operator below is described in terms of its effect on the expression stack. Schematically, this semantic specification is a set of stack reduction rules which when systematically applied will produce the same result as the interpreter. Basically, what makes this possible is that the control flow has been isolated from the computation.

The stack reduction rules are listed in the following format

Stack contents | Operator ➔ Result

The result is listed in the form of an equivalent C-BC expression, along with the implied side effects in the case of the operators a, m, A, M, r, s, xn, and w. In all these cases, the value returned of the result is the value returned by the corresponding operation.

In the case of the P operator the side effect is listed in descriptive form and no value is returned on the stack.

The meaning of the reduction rule is as follows: if the next operator in the current continuation string is the given Operator, and the expression stack has the indicated contents, then the operations indicated will be carried out and the indicated Result will be placed on the stack. In the following, v_n, a_n, and f_n respectively denote the nth variable, nth array variable, and nth function.

Table 2a: The Semantic Specification

	`[number]`	➔	`number`
	`"string"`	➔	`"string"`
	`$galois$`	➔	`$galois$`
	`0`	➔	`0`
	`1`	➔	`1`
`A...Z`	`xn`	➔	f_n`(A, ..., Z)`
	`&n`	➔	`&`v_n
	`;n`	➔	a_n`[]`
`A B`	`:`	➔	`&A[B]`
`A`	`w`	➔	`<- A`
`A`	`P`	➔	`"write A"`
`A`	`,`	➔	
`A`	`d`	➔	`A, *A`
`A`	`n`	➔	`-A`
`A`	`!`	➔	`!A`
`A`	`r`	➔	`-> *A`
`A B`	`s`	➔	`*A = B`
`A`	`a`	➔	`(*A)++`
`A`	`m`	➔	`(*A)--`
`A`	`A`	➔	`++(*A)`
`A`	`M`	➔	`--(*A)`
`A`	`l`	➔	`*A`
`A B`	`op`	➔	`A op B    (op: + - * / % ^ < >)`
`A B`	`=`	➔	`A == B`
`A B`	`#`	➔	`A != B`
`A B`	`{`	➔	`A <= B`
`A B`	`}`	➔	`A >= B`

In the notation above, galois, number and string stand for character sequences conforming to the respective formats of constants of type galois, number and string (ignoring any leading or trailing $'s or "'s). The operators &n, ;n, xn consist of the lead character (&, ;, or x) followed by a binary code (16-bits, with low-order byte listed first) representing the associated number of the variable, array or function, respectively.

More precisely, the configuration of the C-BC machine takes on the following form

S, E, C

where S is a stack containing a set of values, E is the current environment and C is the code-string currently undergoing evaluation. The specification itself is recursive, with the recursion occurring with the specification of the function calls. The process of removing this recursion will lead to an extra component, D, which will be a dump (or display frame) for the call-return mechanism. This can be systematically derived from the recursive specification (and actually was, at a late stage in coding the interpreter).

An environment, E, contains a set of object:value pairings. The types of objects and values are as follows

Table 1: Object Types and Corresponding Values

Object	**Value**
Array Variable	Array = List of Variables
Pointer Variable	Variable
Scalar Variable	Constant
Function	String (+ parameters and local variables)

Variables are indexed and denoted &n (for scalars and pointers), ;n (for arrays) for n = 0, 1, 2, ... Functions are also indexed denoted xn for n = 0, 1, 2, ...

Given an environment the current value of a variable v will be denoted E(v). The nth component of an array variable, a, will be denoted E(a)[n]. The effect of modifying the current environment E by assigning an object x a value v will be denoted: (x <- v, E).

Technically, a configuration (S, E, C) is a function that maps input to output. This mapping will be explicitly indicated only where necessary. Otherwise, all configuration rules of the form

(S, E, C) = (S’, E’, C’)

are to be understood in the sense

<S, E, C>(Input) = <S’, E’, C’>(Input’)

where the stream Input’ is the stream Input minus whatever of its leading items were consumed by the relevant operations. This expanded listing is only used with the function call and read operation, while the reference to <S,E,C> as a list is only used with the write and print operations.

Also, list notation will be used for inputs and outputs with the basic list primitives:

x:L	The list formed from L by adding x to the front
L:x	The list formed from L by adding x to the back
[]	The empty list
[a,b, ..., z]	a: b: ... z:[]

List notation will also be used for C and S.

Table 2b: The Execution Machine

S, E, **c**:C	➔	S:c, E, C [1]
S, E, **&n**:C	➔	S:&vn, E, C
S, E, ;**n**:C	➔	S:an[], E, C
S, E, **xn**:C	➔	S:fn, E, C
S:A:B, E, ::C	➔	S:E(A)[B], E, C
S:A, E, ,:C	➔	S, E, C
S:A, E, **n**:C	➔	S:-A, E, C
S:A, E, **!**:C	➔	S:!A, E, C
S:A, E, **d**:C	➔	S:A:E(A), E, C
S:A, E, **l**:C	➔	S:E(A), E, C
S:A:B, E, op:C	➔	S:E(A) OP E(B), E, C [2]
<S:A, E, **r**:C>(x:I)	➔	<S:r,(A<-x,E),C>(I) [3]
S:A:B, E, **s**:C	➔	S:B, (A<-B,E), C
S:A, E, **a**:C	➔	S:E(A), (A<-E(A)+1,E), C
S:A, E, **m**:C	➔	S:E(A), (A<-E(A)-1,E), C
S:A, E, **A**:C	➔	S:E(A)+1, (A<-E(A)+1,E) C
S:A, E, **M**:C	➔	S:E(A)-1, (A<-E(A)-1,E), C
S:A, E, **w**:C	➔	write(A):<S, E, C> [4]
S:A, E, **P**:C	➔	print(A):<S, E, C> [4]
<S:a1:...:an:f,E,@:C>(x1:... xm:I)	➔	y1:...:yp:<S:f(a1,...,an),E’,C>(I) [5]

Notes:

[1] A constant **c** is 0, 1 or a scalar value of one of the forms [...], “...”, or $...$.

[2] The correspondences between byte codes (op) and operations (OP) are as follows:

op	+	-	*	/	%	^	<	>	=	#	{	}
A OP B	A+B	A-B	A*B	A/B	A%B	A^B	A<B	A>B	A==B	A!=B	A<=B	A>=B

with all the operations (including div, mod and exponent) defined in the C-BC reference.

[3] r will be equal to 0 or 1 depending on whether the item read has the correct syntax for the type of the expression A or not.

[4] write(A) and print(A) are the two “write” functions that convert the value A into an ASCII string suitable for output. Both operations are described in the C-BC reference.

[5] The function f is assumed to have arity n for some n >= 0, and when called will recursively invoke a new level of execution with respect to its corresponding String. The resulting environment is denoted E’, The values x1, x2, ..., xm are whatever the function reads in as input, and y1, y2, ..., yp whatever its outputs are. All three of these items are recursively defined by applying the specification to the function’s body.

Example:

The following example illustrates the application of the rules. To make the presentation easier, the list notation (_):(_) is replaced by concatenation (_) (_).

Continuation String: `&2 d &1 a * s ,`

Variables:

#	Name	Type	Value
2	x	number	4
1	I	number	2

Stack State: (empty)

Reduction:

```
S | (&2<-4, &1<-2, E) | &2 d &1 a * s , C
 ➔ S &x | (&2<-4, &1<-2, E) | d &1 a * s ,
c
 ➔ S &x 4 | (&2<-4, &1<-2, E) | &1 a * s ,
C
 ➔ S &x 4 &i | (&2<-4, &1<-2, E) | a * s ,
C
 ➔ S &x 4 2 | (&2<-4, &1<-3, E) | * s , C
 ➔ S &x 8 | (&2<-4, &1<-3, E) | s , C
 ➔ S 8 | (&2<-8, &1<-3, E) | , C
 ➔ S | (&2<-8, &1<-3, E) | C
```

The final effect of the continuation string is to leave the stack in its original state to multiply x by i and to increment i. This is the translation of the expression x *= i++.

6. The C-BC Translation Grammar

The source to byte-code translation is carried out with a SSDTG. A SSDTG is defined as a structure consisting of the following:

X: Input alphabet
Y: Output alphabet
N: Non-terminals
S: an element of N denoting the *start* symbol
P: a set of rules of the form n: a ➔ b, with a non-terminal n $\in$ N, a word a $\in$ (X + N)* and a word b $\in$ (Y + N)*.

All rules are constrained so that for each rule (n: a ➔ b) the number and order of non-terminals in a and b is the same. This notation is extended so that

$$\begin{array}{llll} n: & a_1 & \rightarrow & b_1 \\ & a_2 & \rightarrow & b_2 \\ & & \dots & \\ & a_n & \rightarrow & b_n \end{array}$$

will denote the set of translations { n: a_i ➔ b_i: i = 1, 2, ..., n }.

The grammar itself is ambiguous with respect to the syntax for expressions and is resolved with precedence rules. These are described in the C-BC reference.

6.1. Expressions

The description below uses the following abbreviations

f_n	Function #n
a_n	Array variable #n
v_n	Variable #n
l_x	Label denoting state #x
$e, e_1, e_2, \ldots, e_n$	Expressions e, e_1, e_2, ..., e_n
$s, s_1, s_2, \ldots, s_n$	Statements s, s_1, s_2, ..., s_n
$x, x_1, x_2, \ldots, x_n, y, z$	Continuation string labels x, x_1, x_2, ..., x_n, y, z

If a P appears in the first column, the expression following is printable if it is of scalar type. If a ? appears in the first column, the expression is printable if any only if the final sub-expression is. If an L appears in the first column, the expression is an L-value and will be printable when dereferenced if and only if it is of a scalar type.

6.1.1. I/O Expressions

E:	`-> e, e`$_1$`, ..., e`$_n$	➔	e r :x_1 x) (x_1: 1 + e_1 r :x_2 x) (x_2: 1 + ... e_n r :x_n x) (x_n: 1 + :x x) (x:
	`<- e, e`$_1$`, ..., e`$_n$	➔	e w :x_1 x) (x_1: 1 + e_1 w :x_2 x) (x_2: 1 + ... e_n w :x_n x) (x_n: 1 + :x x) (x:
	`-> e`	➔	e r
	`<- e`	➔	e w

6.1.2. Sequential Logic Expressions

E:	`e? e`$_1$`: e`$_2$	➔	e :x y)(x: e_1 :z z)(y: e_2 :z z)(z:
	`e`$_1$ `||` `e`$_2$	➔	e_1 :x z)(z: e_2 :x y)(x: 1 :w w)(y: 0 :w w)(w:
	`e`$_1$ `&&` `e`$_2$	➔	e_1 :z y)(z: e_2 :x y)(x: 1 :w w)(y: 0 :w w)(w:
	`!e`	➔	e! [≈ e :y x)(x: 1 :w w)(y: 0 :w w)(w:]
?	`e`$_1$`, e`$_2$	➔	e_1, e_2 [≈ e_1 :x x)(x: e_2]

As indicated or implied in the table, the following equivalences apply:

$$e_1 \,\|\, e_2 \approx e_1?\ 1{:}\ e_2?\ 1{:}\ 0$$
$$e_1 \mathbin{\&\&} e_2 \approx e_1?\ (e_2?\ 1{:}\ 0){:}\ 0$$
$$e \approx e?\ 0{:}\ 1$$
$$e_1, e_2 \approx e_1?\ e_2{:}\ e_2$$

The last of these does not apply to expressions e_1 of the type **void** (which is slated for addition to C-BC).

6.1.3. Relational Expressions

E: e_1 OP e_2 ➔ e_1 e_2 op

with the following correspondences

op	<	>	=	#	{	}
`OP`	`A<B`	`A>B`	`A==B`	`A!=B`	`A<=B`	`A>=B`

6.1.4. Assignment/Update Expressions

E:	`++e`	➔	e A
	`--e`	➔	e M
	`e++`	➔	e a
	`e--`	➔	e m
	`e`$_1$ `= e`$_2$	➔	e_1 e_2 s
	`e`$_1$ `OP= e`$_2$	➔	e_1 d e_2 op s
	`e`$_1$ `=OP e`$_2$	➔	e_1 d e_2 op s

with the following correspondences

op	+	-	*	/	%	^
`OP`	`A+B`	`A-B`	`A*B`	`A/B`	`A%B`	`A^B`

6.1.5. Arithmetic Operators

E:			
P	`+ e`	➔	e
P	`- e`	➔	e n
P	`e`$_1$ `OP e`$_2$	➔	e_1 e_2 op

with the same correspondences between OP and op as listed above in 6.1.4.

6.1.6. Addressing/Pointer Expressions

E:	`& e`	➔	e
L	`* e`	➔	e
	`new e`	➔	e N
	`free e`	➔	e F

6.1.7. Other Expressions

E:	`a`$_n$`[]`	➔	;n
L	`v`$_n$	➔	&n

P	`fₙ(e₁, ..., eₘ)`	➔	e_1 ... e_m xn
L	`e₁[e₂]`	➔	e_1 e_2 + (e_1 a pointer)
L	`e₁[e₂]`	➔	e_1 e_2 : (e_1 an array)
L	`aₙ[e₂]`	➔	;n e_2 : (special case)
?	`(e)`	➔	e

6.1.8. Constants

E:

P	`number`	➔	[number]
P	`“string”`	➔	“string”
P	`$galois`	➔	$galois$
P	`$galois$`	➔	$galois$
P	`0`	➔	0

6.1.9. Pre-Defined Objects

These are the mappings for the following built-in objects

Built-In Variables:

`ibase`	`obase`	`scale`	`last`	`lastc`	`lastg`	`lasts`
v_0	v_1	v_2	v_3	v_4	v_5	v_6

Built-In Functions:

`sqrt`	`scale`	`length`	`shell`	`ifile`	`ofile`	`afile`
f_0	f_1	f_2	f_3	f_4	f_5	f_6

`comp`	`re`	`im`	`field`	`gal_p`	`gal_m`	`gal_gx`	`gal_set`	`gal_get`
f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}

6.2. Automatic Conversions.

L-values are automatically dereferenced in all contexts except those specifically requiring L-values. If this operation is denoted “deref e”, then the following effective translation rule applies.

E:

P deref e ➔ e l

Arrays, likewise, are automatically converted to pointers in most contexts, as described in the C-BC documentation. If this operation is denoted as “convert e”, the following rule then applies

E: convert e ➔ e 0 :

6.3. Statements

6.3.1. Expression Statements:

S:	e;	➔	e P (if e is printable)
	e;	➔	e , (if e is not printable)
	;	➔	

6.3.2. Jump Statements:

In all these statements, a break occurs in control-flow. Therefore, when the following continuation string is created, it is created with a new label, denoted as y. The labels c and b denote the c-label and b-label of the innermost containing loop statement. The label, -, denotes the null continuation string.

S:	goto l_x;	➔	:x x) (y:
	continue;	➔	:c c) (y:
	break;	➔	:b b) (y:
	return;	➔	0 :0 0) (y:
	return();	➔	0 :0 0) (y:
	return e;	➔	e :0 0) (y:
	halt;	➔	:- -) (y:

6.3.3. Other Statements:

In the following statements, the labels c and b denote the labels that will be matched respectively to “continue” and “break” statements contained immediately inside these statements.

S:	{ s_1 ... s_n }	➔	s_1 ... s_n
	l_x: s	➔	:x x) (x: s
	if (e) s	➔	e :x z) (x: s :z z) (z:
	if (e) s_1 else s_2	➔	e :x y) (x: s_1 :z z) (y: s_2 :z z) (z:
	while (e) s	➔	:c c) (c: e :x b) (x: s :c c) (b:
	do s while (e);	➔	:x x) (x: s :c c) (c: e :x b) (b:
	for ([e_1]; [e_2]; [e_3]) s	➔	[e_1 ,] :x x) [(x: e_2 :y b)] [(c: e_3, :x x)] (y: s :c c) (b:

In the for statement, the expressions denoted in brackets are optional and the following apply to it:

- If the brackets do not occur, then the corresponding items enclosed in []’s on the right hand side will not appear either.
- If expression e_2 is absent, therefore, labels x and y will be considered identical.
- Likewise, if expression e_3 is absent, labels c and x will be considered identical.

In addition, the following applies to the do-while statement:

- In this version of C-BC, the :c c) (c: sequence in the do-while statement will be optimized away unless it is referred to by a “continue” statement.

6.4. Pseudo Statements

Some of the top-level commands (called here: pseudo-statements) are translated and processed in the interpreter. These include the following

P:	`include “string”`	➔	“string” I
	`log “string”`	➔	“string” L
	`log`	➔	“” L
	`quit`	➔	(exit)

The command “quit” is not processed at all, since the interpreter exits as soon as it is seen.

6.5. Function Bodies and Execution Sequences

A function definition consists of a set of parameter declarations, auto variable declarations, and a sequence of statements. In this version of C-BC, the declarations are processed and associated with the function name in the function name space. If the sequence of statements is s_1 ... s_n, it is translated as if by the following rule

P: `Header { s1 ... sn }` ➔ (0: - :? ?) (1: s_1 ... s_n 0 :0 0)

The - and ? denote respectively empty continuation strings and don’t-care continuation labels. In all function definitions, state 0 will be the ending state, consisting of an empty continuation string, and state 1 will be the starting state. Note the final 0 :0 0) sequence, which corresponds to an implicit return; statement at the end of the function.

An execution sequence is a single statement listed outside a function. If there is more than one statement on a given line, each one is still considered a separate execution sequence.

If the statement, s, is the execution sequence, it is translated as if by the following rule

P: `s` ➔ (0: - :? ?) (1: s :0 0)

Note that in this case, there is no value returned. Effectively, execution sequences are anonymous functions of type void that are (re)defined, and then executed.

Future extensions of C-BC may allow for the addition of “void” types, and may denote this function as “main”. In this case, the following declarations would be understood: void main();

Finally, at the top level of C-BC, one has the rule

start: `P1 P2 ... Pn` ➔ P_1 P_2 ... P_n

which states that a program is a sequence of function definitions, statements and pseudo-statements translated in the order in which they are issued.

6.6. Declarations

The syntax and semantics for declarations is described in detail in the C-BC documentation, so no further information will be provided here. Declarations may be made at the top-level (and so one has a rule of the form P: Dec $\Rightarrow$ ...) or as part of a function's header, whose syntax is not further specified here.

7. Generating the Stack Frame

In the process of removing the recursion from the specification of the C-BC Execution Machine listed in Part 5, a stack frame will be generated. In order to carry out the reduction

$$<S:a_1:...:a_n:f, E, @:C>(x_1:x_2:... x_m:I) \Rightarrow y_1:y_2:...:y_p:<S:f(a_1,...,a_n), E', C>(I) \ [5]$$

we need to temporarily save the environment. A new environment with the function's parameters and locals is created. Then the values for a_1 through a_n are stored into function's parameters and the locals are initialized (all to 0, there are no declaration initializers in C-BC). The function's string is prefixed to C, along with a return code. When the return code is reached the original environment will be restored.

An extra component, D, called the DUMP (or frame display) is then added to the configuration (S, E, C). Each of the rules previously stated will be modified by adding D to both sides.

Assume that the function's string is named Cf, its parameters are named &P1, ..., &Pn, that its locals are named &L1, ..., &Lp (the parameters and locals are the "extra information" referred to by the listing in Table 1). Also let p_i denote $E(P_i)$, and l_i denote $E(l_i)$.

The function call is accomplished inductively as follows
$<S:a_1:...:a_n:f, E, @:C, D>(x_1:...:x_m:I)$
$\Rightarrow <S, (L_1<-0, ..., L_p<-0, P_1<-a_1, ..., P_n<-a_n, E), C_f:\#:C, [P_1, p_1, ..., P_n, p_n]:[L_1, l_1,...,L_p, l_p]:D>(x_1:x_2:...:x_m:I)$
$\Rightarrow$ ... [execute C_f]
$\Rightarrow y_1:...:y_p:<S:A, E', \#:C, [P_1, p_1, ..., P_n, p_n]:[L_1, l_1, ..., L_p: l_p]:D>(I)$
$\Rightarrow y_1:...:y_p:<S:A, (P_1<-p_1, ..., P_n<-p_n, L_1<-l_1, ..., L_p<-l_p,E'), C, D>(I)$
where the x's and y's are as in note [5] under Table 2b. This has the effect of restoring the values of the locals $L_1, ..., L_p$ and parameters $P_1, ..., P_n$ after the function call. The value A is whatever is left on the stack after the function is done. Since all functions in C-BC return a single value, there will always be an extra item on the stack after the function is complete.

This listing leads to several more rules:

Function Calls:
S:a1:...:an:f, E, @:C, D $\rightarrow$ S, (L1<-0,...,Lp<-0,P1<-a1,...,Pn<-an,E), Cf:#:C, [P1,E(P1),...,Pn,E(Pn)]:[L1,E(L1),..., Lp,E(Lp)]:D

where n is the arity of the function, L1, ..., Lp its locals, P1, ..., Pn its parameters, and Cf its string. All these items are listed with the function in the environment so that f's entry in the environment will look like

E(f) = ([P1, ..., Pn], [L1, ..., Lp], Cf)

Note that such an entry is established as a result of processing a function definition. This specification, however was left out of the SSDTG.

Function Returns:
S:A, E', #:C, [P1, p1, ..., Pn, pn]:[L1, l1, ..., Lp, lp]:D $\rightarrow$ S:A, (P1<-p1, ..., Pn<-pn, L1<-l1, ..., Lp<-lp,E'), C, D

So the format of the dump is a list of items of the form

[[P1, p1, ..., Pn, pn], [L1, l1, ..., Lp, lp]]

In the C-BC interpreter, the dump doesn't really take on this form. Instead each variable is represented by name and its corresponding environment entry is actually a stack of values instead of just one. When parameters and locals are initialized the new values are pushed on the corresponding variable stacks, which are popped upon the function's return.

However, this is the process that was actually applied to the code itself to arrive at the interpreter, minus the function-call recursion. Originally, I was going to just leave the recursion in but there were some other parts of the interpreter that needed to explicitly manipulate the dump.

8. C-BC Numeric Processing Algorithms

The numeric algorithms are unique in that all the calculations are carried out in base 1,000,000,000. This is the only power of 10 smaller than 32 bits that is almost equal to a power of 2 (2^{30}), is a perfect cube (which is needed for the division algorithm), and which has factors almost equal to the square root of the power of 2 (32000, 31250 both almost equal to 2^{15}), which are needed for the multiplication algorithm.

The multiplication and division algorithms are based on 30 bit numbers. What makes the method used here unique is that it is portable in ANSI-C and does not use numbers larger than 32-bits.

8.1. 30-Bit Multiplication

The extended multiplication and division algorithms are based on the following operation $AB = QN + R \quad (Q, R < B)$.
To carry out these operations, it is assumed that the base N can be factored $N = r_0 r_1$ into two numbers $r_0, r_1 \approx \sqrt{N}$. In the case $N = 10^9$, the factoring is $10^9 = 32000 \times 31250$ with $(r_0, r_1) = (32000, 31250)$, while in the case $N = 2^{30}$, it is $2^{30} = 2^{15} \times 2^{15}$ with $(r_0, r_1) = (2^{15}, 2^{15})$.

The numbers A, B are split into two components as follows

$$A = A_1 r_0 + A_0, \quad B = B_0 r_1 + B_1 \quad (0 \le A_0, B_0 < r_0, \quad 0 \le A_1, B_1 < r_1).$$

It follows from this that

$$AB = A_1 B_0 N + (A_1 B_1) r_0 + (A_0 B_0) r_1 + A_0 B_1.$$

Now carry out the following decompositions on $A_0 B_0$ and $A_1 B_1$:

$$A_0 B_0 = C_0 r_0 + D_0, \quad A_1 B_1 = C_1 r_1 + D_1 \quad (0 \le C_0, D_0 < r_0, \quad 0 \le C_1, D_1 < r_1).$$

By assumption, $r_0, r_1 \approx \sqrt{N}$ so that $r_0^{\,2}$ and $r_1^{\,2}$ can be both represented as 32 bit numbers. Therefore, $A_0 B_0$ and $A_1 B_1$ are no larger than 32-bits. Therefore,

$$(A_0 B_0) r_1 = C_0 N + D_0 r_1, \quad (A_1 B_1) r_0 = C_1 N + D_1 r_0.$$

Thus

$$AB = (A_1 B_0 + C_0 + C_1) N + D_1 r_0 + D_0 r_1 + A_0 B_1.$$

Keeping in mind that $D_1 r_0 + D_0 r_1 + A_0 B_1 < 2N$, the remainder and quotient can be represented respectively as

$$Q = (D_1 r_0 + D_0 r_1 + A_0 B_1) \operatorname{div} N + (A_1 B_0 + C_0 + C_1), \quad R = (D_1 r_0 + D_0 r_1 + A_0 B_1) \operatorname{mod} N$$

Therefore, the result of multiplying AB can be represented as

$$AB = QN + R.$$

8.2. 30-Bit Division

Number I/O operations use base conversions, which require the following 30-bit division operation

$$NC + A = DQ + R \quad (0 \le C, R < D, \quad 0 \le Q, A < N).$$

This divides A into D, with carry digit C and yields, as a result, the quotient Q, and the remainder R as a new carry digit. In order to be able to do this, we need to assume that N is a perfect cube: $N = r^3$. Then, there are 3 cases to consider.

8.2.1. Division by small digits $(0 < D < r)$

With the following decompositions

$$N = B_1 D + B_0, \quad B_0 C + A = Q_0 D + R_0 \quad (0 \le Q_0 < C, \quad 0 \le B_0, R_0 < D)$$

we have

$$NC + A = B_1 CD + Q_0 D + R_0 = (B_1 C + Q_0)D + R_0,$$

thus leading to the following result

$$Q = Q_0 + B_1 C, \quad R = R_0.$$

8.2.2. Division by medium sized digits $(r \le D < r^2)$

Since $C < D < r^2$, the numbers A and C can be decomposed as follows

$$A = A_2 r^2 + A_1 r + A_0, \quad C = C_1 r + C_0 \quad (0 \le A_0, A_1, C_0, C_1 < r).$$

Carry out the following decompositions

$$C_1 r^2 + C_0 r + A_2 = Q_2 D + R_2, \quad R_2 r + A_1 = Q_1 D + R_1, \quad R_1 r + A_0 = Q_0 D + R_0 \quad (0 \le Q_2, Q_1, Q_0 < r, \quad 0 \le R_2, R_1, R_0 < D).$$

Then

$$CN + A = Q_2 r^2 D + Q_1 r D + Q_0 D + R_1 = (Q_2 r^2 + Q_1 r + Q_0)D + R_0 \quad (0 \le Q_2, Q_1, Q_0 < r, \quad 0 \le R_0 < D).$$

Thus, the resulting quotient and remainder are

$$Q = Q_2 r^2 + Q_1 r + Q_0, \quad R = R_0.$$

8.2.3. Division by large digits $(r^2 \le D)$

This time, decompose A into the following

$$A = A_2 r^2 + A_1 r + A_0.$$

Since the number D is large relative to N, decomposition is taken with respect essentially to $g = D/r$. In particular, select $g = 1 + (D-1) \operatorname{div} r$. Then $D \le gr < D + r$, and $r \le g \le r^2$. Carry out the following decompositions

$$C = Q_2 g + R, \quad E_2 = Q_2(gr - D) + R_2 r + A_2 \quad (0 \le R_2 < g);$$
$$E_2 = Q_1 g + R, \quad E_1 = Q_1(gr - D) + R_1 r + A_1 \quad (0 \le R_1 < g);$$
$$E_1 = Q_0 g + R_0, \quad E_0 = Q_0(gr - D) + R_0 r + A_0 \quad (0 \le R_0 < g).$$

Then the quotient and remainder are

$$Q = Q_2 r^2 + Q_1 r + Q_0 + E_0 \operatorname{div} D, \quad R = E_0 \bmod D.$$

The choice of g guarantees that all these numbers are less than 32 bits.

8.3. Base Conversion

The base conversion routines used in numeric I/O are used to convert between the numeric base 1,000,000,000 and the current input or output base. This process can be generally described as below.

Assume the more general problem of converting the array a[0]...a[A-1], with 0 <= a[i] < m to an array b[0]...b[B-1], with 0 <= b[i] < n, such that

(a[A-1] ... a[1] a[0]) base m = (b[B-1] ... b[1] b[0]) base n

Assume, also, that the bases m and n are such that the operation

Um + V -> Wn + X (0 <= U, X < n and 0 <= V, W < m)

can be carried out within the limitations of the machine's arithmetic.

The first algorithm divides the number represented by a[] and A into n and returns the remainder as U

Convert1(a[], A) -> (a[], A, U):

- Set U = 0
- For all i = A - 1 down to 0 make the following conversion:Um + a[i] -> a[i]n + U that is, assign a[i] and U to the quotient and remainder of Um + a[i] respectively

- Strip off any leading 0's and reduce A accordingly.

The second algorithm performs this conversion repeatedly, placing the successive remainders into b[], until a[] is reduced to 0.

```
Convert(a[]) -> (b[], B):
  • Set B = 0
  • while A > 0:
      ■ Convert1(a[], A) -> (a[], A, b[B])
      ■ Increment B
```

The applications of this routine are as follows

- Number input: Read input digit sequence and convert from ibase to 1,000,000,000.
- Number output: Convert from 1,000,000,000 to obase and print out resultig digit sequence.

9. Pending Extensions

9.1. Jolt Syntax for Statements, Expressions and Types

This enables subexpressions to be labelled with **goto** *x* punned into being a reference to subexpressions (as well as the other jump statements **continue**, **break** and **return**). Also: any expression with a *hole* – or missing subexpression – is counted as a statement. These are called *contexts*. Any statement followed by an expression is counted as an expression. If the statement is a context, then the result is equivalent to making the following expression a subexpression by inserting it into the hole provided by the context. Otherwise, it is equivalent to executing the statement and then evaluating the expression.

An example of the kind of expression encountered in such an extended syntax is as follows

$$i = \mathbf{0},\ y = \mathbf{1},\ x\mathbf{:}\ i < n\mathbf{?}\ (y \mathrel{*}= \mathbf{2},\ i\mathbf{++},\ \mathbf{goto}\ x)\mathbf{:}\ y$$

There is a deep subtlety in this: type-checking on such expressions becomes recursive. This seems to go naturally with the idea of including an explicit **typeof** operator. The recursion in the above example occurs with

$$\mathbf{typeof}(y \mathrel{*}= \mathbf{2},\ i\mathbf{++},\ \mathbf{goto}\ x) = \mathbf{typeof}(x)$$

while

$$\mathbf{typeof}(x) = \mathbf{typeof}(i < n\mathbf{?}\ (y \mathrel{*}= \mathbf{2},\ i\mathbf{++},\ \mathbf{goto}\ x)\mathbf{:}\ y)$$

which would be the "smallest" type containing both **typeof(***x***)** and **typeof(***y***)** (i.e., the least type in the underlying inheritance hierarchy that contains both of these types). This implies the recursive equation

$$\mathbf{typeof}(x) = \text{least of } \mathbf{typeof}(x) \text{ and } \mathbf{typeof}(y).$$

If solutions to recursive type-checking systems are taken in the sense of least solutions, this would equate to

$$\mathbf{typeof}(x) = \mathbf{typeof}(y).$$

This extension with Jolt syntax allows more subtle expressions whose readings as "control-flow" need not be as obvious as the "while loop" expression above, e.g.

$$(n = \mathbf{2},\ y = \mathbf{1},\ \mathbf{goto}\ x) + (n = \mathbf{3},\ y = \mathbf{3},\ x\mathbf{:}\ i < n\mathbf{?}\ (y \mathrel{*}= \mathbf{2},\ i\mathbf{++},\ \mathbf{goto}\ x)\mathbf{:}\ y)$$

which is equivalent to the addition of **4** and **24**, yielding **28**. This is where the type-checking system really comes fully into its most subtle aspects since, here, restrictions are also placed on the allowable types for **typeof(***x***)** by the operators in the expressions containing them.

This also requires making explicit the call-return mechanism, since such labeled subexpressions are effectively in-line subprograms. A further subtlety can be seen in the following modification of the example above

$$(n = \mathbf{2},\ y = \mathbf{1},\ \mathbf{goto}\ x) + (n = \mathbf{3},\ y = \mathbf{3},\ x\mathbf{:}\ i < n\mathbf{?}\ (y \mathrel{*}= \mathbf{2},\ i\mathbf{++},\ \mathbf{goto}\ x)\mathbf{:}\ \mathbf{return})$$

If the subexpressions appear within a function body, the function itself may return to the calling function (and to whatever function, in turn, called it, so on ad infinitum) before the expression **goto** *x* is considered to be evaluated. Then it would return back into the function to evaluate the rest of the expression it is a part of. Therefore, it seems necessary to incorporate the extensions in 9.9 below, relating to concurrency.

If a function is called in the context $x = f(\ldots),\ Q$, then the corresponding **return** statement is equated to the context

return [...] $\leftrightarrow$ *x* = [...]**,** *Q*.

The corresponding type, then, of the statement **return** *e* is *not* the type of the expression *e*, but the type of the expression (*x* = *e***,** *Q*); that is: of the expression *Q*. This typifies the full extent of the subtleties both with type-checking and with the actual evaluation of subexpressions.

If void functions are supported, then corresponding to a void function called in the context *f***(**...**),** *Q* would be the equivalence

return $\leftrightarrow$ *Q*.

9.2. switch/case/default Statements, Switch Expressions and Memoized Array/Functions

There should be a combined syntax for **switch** expressions and **case**/**default** labels within expressions.

Since C-BC is an expression-based language, it is not possible to add in the **switch**, **case** and **default** statements without counterparts in the syntax for expressions. That's why they were not originally included. The extension being considered should seamlessly grandfather in the existing syntax and keep everything simple. One way to do this is to creative re-analyze the conditional expression A? B: C itself, taking the "? B" and ": C" parts separately. When given their current meaning ": C" is analogous to "case 0: C", while "? B" is analogous to "default: B". Thus, one way to expand the syntax is to allow labels with either or both; e.g. "@K_1 ...@K_n: C" would be analogous "case K_1: ... case K_n: C", while "@K_1 ...@K_n? C" might be considered as a possible way to *exclude* cases. Under this revision, one would have the following equivalences

A? B: C
A? B @0: C
A @0: C? B
A: C? B

Further possibilities may be to include relationals, e.g. @0>, @0+, @0<, @0-. The only restriction may be that the expressions listed under @ be compile-time constants so that overlaps in cases can be compile-time checked. A translation for a switch statement switch (E) S, where S included cases K_1 , ..., K_n attached with respective labels x_1 , ..., x_n and a default attached with label y_0 would be E x) (S b) (x ? x_0 @K_1: x_1 ... @K_n: x_n) (b: the label b is associated with the **break** statement. Optimizations are possible; e.g. if there is no default statement, then x_0 is replaced by b; if there is a default statement but no cases, then the translation would be E x_0) (S b) (b:.

This extension lends itself naturally over to a way to merge the syntax for arrays and functions. Here, an array expression is posed as a switch expression – possibly with a way to auto-increment the case label (e.g. by making the @ part optional). The default case is then defined by a function body. This also happens to get us one step closer to the extensions described in 9.1.

9.3. Polynomial Routines

Support has been added to C-BC to allow manipulations of polynomials through number arrays, through the galois field type.

To enhance this support, it may be desirable to add a basic integer type, such as **word**, to allow for more efficient manipulation of small numbers. If you add this type, you should include it in the type conversion scheme as follows

word < number < complex

9.4. Variable Argument Functions

A facility similar to ANSI-C's <stdarg.h> would be highly desirable. This is slated for addition as part of an integrated facility for passing parameters and returning values for function calls (including void functions); see 9.7.4 and 9.7.5.

9.5. Complex Numerals and Other Algebraic Systems

Expand the type to include quaternions and even Clifford algebras; and possibly implement the complex number arithmetic directly on a (complex) 'decimal' positional number system, we well as signed radixes (e.g. base 3 with digits -1, 0 and +1, or base 3 complex with additional digits for -1 - i, -1 + i, -i, +i, +1 – i and +1 + i. This may be done by extending the "ibase" and "obase" radix facilities to include some kind of provision for specifying the range of values of the digits.

9.6. String Operators

The most important operator that comes to mind is the sprintf conversion operator, and reverse conversion operators to translate numeric types into strings. Concatenation and truncation operators would also be nice to have, and maybe a pair of conversion operators to equate the types

string ↔ array of **word**

In general, C-BC has been conceived of as a family of related languages that differ in the set of basic types and operations they support, rather than as a single language; so that other type systems may be built into the language(s).

9.7. C-DC: Separate Compilation, Self-Sufficient Low-Level Language and DC

With the addition of a few more operators to denote symbol table additions, declarations and function definitions, the byte-code language used by the interpreter in C-BC can be made self-sufficient. This will allow files to be compiled separately. In particular, it will allow the math library to be pre-compiled.

Also, the low-level language might be made extensive enough to make possible conversion of a superset of DC into it.

There are two kinds of declarations that need to be added to a self-contained byte-code language: symbol table entries, and type declarations. Basically, the extension envisioned would look something like this (other symbols will have to be used for some of these commands):

9.7.1. Symbol Table Entries

#*p S Name* – Names symbol table entry #*p* in space *S* as indicated. *S* denotes one of the name spaces (**variable**, **array**, **function**, **label**)

9.7.2. Declarations

S*p Type* – A command that declares variable #*p* to be of the given *Type*, and pops its initial value off the stack. If the variable is of array type, it is referenced to the value popped off the stack.

C*p Type* – Declares array variable #*p* as the given type (array of *Type*), and initializes it to an empty array of **0**'s.
L*p* – Frees up the local/auto variable #*p*.
F*p* – Frees up the local/auto array variable #*p*, and deallocates the underlying array, if it was declared as block-local.

In the current implementation of C-BC, information about each variable is kept on a separate stack dedicated to that variable. This should all be moved to the general stack, so as to simplify the structure of variables used by the interpreter. In addition, support for the keyword "auto" should be repurposed to denote thread-local variables. See the comments in 9.9.2 (concurrency).

9.7.3. Denoting Types

A method for representing types in the byte code language must be defined. One way (the way used internally in C-BC) is to set each of the base types to a numeric value (n = 1, c = 2, g = 3, s = 4) and define the type-building operations as numeric operations (pointer to T = 2T + 3, array of T = 2T + 4). Owing to the internal type restrictions of the current implementation of C-BC (29 dimensions max), all types can then be represented by 32 bit integers. An alternate method is to represent the type as a string, e.g. the string "aps" to represent (array of pointer to string).

9.7.4. Parameter Passing, Local Variables and Local Scoping

A function declared with parameters $p_1 \ldots p_n$ of types $T_1 \ldots T_n$ and auto variables $a_1 \ldots a_m$ of types $U_1 \ldots U_m$ would now include both the initialization sequence and finalization sequence in the statement body. For instance, the start state (1) would be modified by prefixing its continuation string with a *Prologue*

$$\mathbf{S}\ p_1\ T_1\ \mathbf{S}\ p_2\ T_2 \ldots \mathbf{S}\ p_n\ T_n$$

and the auto variables with a corresponding sequence consisting of

S a_i T_i – if for non-array types, T_i

C a_j T – for array types T_j = array of T.

This facility has to be extended to handle functions with variable arguments (as described in 9.4).

Return statements will be considered as jumps to a common final state (the *Epilogue*). This state will execute the finalization sequence consisting of the following items:

F a_j – for each auto array, a_j, declared with **C** a_j T
L a_i – for each auto variable, a_i, declared with **S** a_i T
L p_i – for each parameter, p_i,

and will jump to state 0.

This also allows declarations to be made in-line in a block of statements.

9.7.5. void Type and Structured Return Types

A void type should be included, which means that procedures may be defined that return no values. In effect, this may be punned by using a reserved variable "void" and setting the return value to it. Right now, the same effect is achieved by using the variable "void" and the assignment void = f (…) to simulate (void)f(…).

However, a better way to do this is to simply generalize the call-return mechanism to allow 0, 1, 2 or more components to be returned. The case of 0 is the void type, and the case of 2 or more is the structured type.

9.7.6. Initializers and Local Declarations

Related to the separate compilation facility, is another extension to allow for initialized declarations (e.g. **number** *a* **= 2, ****p* **= &***a*), and local declarations inside bracketed statements (as indicated in 9.7.4). The primary issue at hand with the latter feature is how to handle jump statements that exit bracketed statements containing declarations (the declarations need to be undone before the jump). Feature 9.7.1 is prerequisite to this, and feature 9.7.2 is almost a necessary extension if (6) is to be added.

9.7.7. Type-Checking and Declarations for Statically and Dynamically Scoped Variables

C-BC, like BC and GNU-BC, is dynamically scoped. That means that variables occurring in a function that are not declared there are bound to the caller's variable, rather than the top-level variable. For a properly-typed system, this requires that a function's dynamic variables need to be matched against those of its caller, as if they were hidden parameters.

A facility should be added to allow for declaration either as statically scoped or dynamically scoped, and this should be made to mesh seamlessly with the facility for local declarations, where dynamic scoping is likewise used. The default would be dynamic scoping to the default type. The default type initially would be number (for compatibility with BC), but might be settable within individual functions or bracketed scopes, e.g.

dynamic Type;

The default may be overridden with an explicit type declaration, e.g.

dynamic type declarator, …, declarator;

to declare dynamically scoped variables. One way to do this is by combining "extern" and "auto" and using this in place of dynamic. Similarly, for statically scoped variables, one needs default settings and declarations of the form

global type
global type declarator, …, declarator

One way to do this is by combining "extern" and "static". Finally, for a multi-threaded extension, one also needs a way to declare thread-local variables. One way to do this is by combining "static" and "auto" inside a function body for thread-local variables visible only to the function (and those that call it), and using "auto" outside a function body.

9.8. Structured Types, Unions, Functions and Expressions

A syntax for structures and unions should be incorporated. The ideal setup is to incorporate the elements of bicartesian biclosed categories, rather than a system like that in C.

A syntax for in-line procedures and functions may also be added here. For the latter, this essentially means the inclusion of a λ operator. For the former (and to a degree: the latter too), it may require expanding the notion of context to enable "context" functions.

Also going along with this is a syntax for structured expressions

$$e \rightarrow \{\, e, e, \ldots, e \,\}$$

which would also provide a natural venue for concurrency.

9.9. Concurrency

9.9.1. Multi-Threaded Run-Time System

This model integrates the call-return, and halt commands into one unified scheme with concurrency.

(signals) A new name space is created for signals. Similar to goto labels, signal labels will not need to be explicitly declared before first use. A certain set of predefined signal names will exist to handle signals and interrupts. Essentially, what's being sought for here is what has since been included in the 201x version of C/C++ as *condition variables*.

spawn *Exp* Spawn a new thread to execute expression *Exp*.

- Push current thread onto Thread List.
- Initialize a new stack frame, and local variable space.
- Execute the expression, *Exp*’s, translated byte-code.

Another way to include task-splitting is (a) by the introduction of structured expressions and (b) by the explicit use of concurrency in expression-evaluation, particularly for the components of structured expressions.

exit Kill the current thread

- Free up the associated memory allocated by spawn (the stack frame, the local variable list)
- Restore the next thread from the Thread List, else go back to command mode, if the stack is empty.

pause *Sig* Stop the current thread pending receipt of signal *Sig*.

- Increment *Sig*’s count.
- If the count ≤ 0, then continue on, else:
- Save the current thread into the thread list associated with *Sig*.
- Restore the next thread off the Thread List, else go back to command mode, if the stack is empty.

resume *Sig* Start up a processing waiting receipt of signal *Sig*.

- Decrement *Sig*’s count.
- If the count < 0, the continue on, else:
- Push the current thread onto the Thread List
- Load in the next thread from the thread list associated with *Sig*.

kill *Sig* Kill all the threads associated with *Sig*.

(call) Calling a function

- Save the current address and other associated parameters on the current frame.
- Start up the new function.

This method is currently implemented.

(return) Returning from a function

- Restore the address and associated parameters from the current frame.
- If the frame was empty, however, perform an automatic **exit** instead.

Except for the auto-exit feature, this method is also currently implemented.

(signal) Signalling interrupts

With each type of interrupt/signal, associate a pre-defined signal label. When the interrupt is called, a resume is automatically carried out on the signal.

(memory) Memory model. Keywords *static* and *extern* should be added to the language, alongside *auto*. Under the expansion

- All global variables are static, and are shared by all threads, unless also declared *auto*.
- All local variables (i.e. function parameters and variables declared inside functions) are automatic, and are local to the current thread, unless declared *static* or *external* without the *auto* keyword.
- All auto variables declared inside *or outside* a function body are local to the current thread.
- Variables declared *static* are visible only in the block or file where declared; while those declared *extern* are (and refer to) variables visible everywhere.

"Sig" can be generalized to variables in order to be consistent with 9.9.3. If this is done right, then 9.9.2 can be subsumed. The recommended way is to use reads and writes from/to static or extern auto variables as the "pause" and "resume" operations. Alternatively, this might be done *alongside* the pause/resume model described here.

Also, this needs to be compared with and checked against the 2010/2011 C and C++ standards.

9.9.2. Auto, Thread-Local and Global Storage

When a task is swapped out, or restored, the underlying stack associated with the variable name, has to be appropriately updated. This requires efficiently maintaining a list of local variables that are active in the current task, among other things. When an **exit** is performed, all the local variables of the task being killed are freed up.

The easiest way to do this is to set up a calling frame to store variables. The following mechanism is then used:

- A list of global variables is maintained separately; as are global auto variables on a per-thread basis.
- Any time a local variable comes into scope, the currently active variable of the same name (if there is any) is put into the call frame, and the local variable is made active. Whatever global there exists with that name continues to be associated separately with that name.
- When a block-local auto variable exits scope, the next active variable, if any, is obtained from the call frame.
- When the current thread is swapped out, all of its auto variables are buffered, and the global variables currently active once more.
- When a thread is swapped in, all of its buffered auto variables are made active again.
- When a thread exits or is killed, all of its auto variables are released and the global variables all become active again.

All of this serves as a way to also implement the suggestion in 9.7.2. Also, as mentioned in 9.1, these extensions seem to be necessary as a prerequisite for fully incorporating the Jolt syntax.

9.9.3. Pi-Calculus, DC & Continuations

The support for continuations provided in C-BC can be expanded to full-fledged support. This will provide enough additional resources to allow the full power of DC to be embodied in C-BC. This goes hand-in-hand with the support for a multi-threaded runtime system provided in the proposed extension.

This extension, centered on the multi-threaded run-time system laid out above in 9.9.1, is an implementation of the "World's Smallest Multitasking Kernel", which originated in 1991; but also happens to embody most of the Pi-Calculus, which originated a year later in 1992. The correspondences with the fundamentals of the Pi calculus are as follows:

Concurrency	P | Q	spawn P; goto Q
Input prefixing	c(x).Q	pause(c, x); goto Q
Output prefixing	c<e>.Q	resume(c, e); goto Q
Nil thread	0	exit;
Replication	P!	while (1) spawn P;
Creation of a new name	(νx)Q	auto x; goto Q