Babel

Clayton Bauman  
June 2, 2012

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

* Lorem ipsum dolor sit amet
* Consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
* Ut enim ad minim veniam, quis nostrud exercitation ullamco
* Laboris nisi ut aliquip ex ea commodo consequat.
* Duis aute irure dolor in reprehenderit in voluptate velit esse cillum

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Introduction

*Culture is not your friend.*- Terence McKenna

Babel is an untyped, stack-based, postfix language with support for arrays, linked lists and hashes (dictionaries). Many languages have one or more of these features, so what makes Babel different?

Babel conscientiously reverses the usual notion of language permissions. In most languages, if I am a piece of code, the primary question is: who has the privilege to execute me? Babel reverses this by asking: who gets the privilege to be executed by me? This fundamental reversal is based on the question of whether the value of a computing system resides in its hardware or its software. Babel takes the view that the value resides within the equipment and data and that this is what requires protection. It is the software which is seeking hardware to give it life, not the other way around. Of course, data also requires protection but data resides on hardware and is threatened primarily by software. Protecting hardware from potentially malicious software is the key to protecting data.

Another key part of the Babel vision is to enable *promiscuous remote code execution*. The primary motivation for this is to make it effortless to use Babel code uploaded to the internet by any trusted party in order to create a seamless Babel library base that is directly accessible to any instance of Babel running on any machine with a connection to the internet. Babel uses a crypto-enforced white-list execution policy to ensure that only trusted code can run on the interpreter.

For any of the major languages in use today, it is easily the case that it is the library base which is the primary source of the language’s power and usability. In fact, using (and, perhaps, learning) a language might best be thought of as merely a means to get access to that language’s library base. Measured in terms of the practical usability of their library bases, the modern scripting languages such as Perl, Python and Ruby have amply demonstrated the power of distributed, reusable programming to render a fundamentally more powerful language than that designed by any corporation or even a consortium of very smart people (Haskell, Common Lisp).

But even the old-fashioned C language can be thought of as actually its library base. Think of all those open source crypto APIs written in C, for example. In fact, C is nearly a *universal computer language* as measured by hardware portability and the logical dependency of a vast swathe of superficially non-C code on C-based code – whether through interpreters, virtual machines, compilers, compiler tools, operating systems (Unix, Linux), and so on.

And if there is a universal computer language there is no doubt a universal data format. It is the octet block or *bytes*. The bit would have been a superior choice as a data format – the byte is the QWERTY of data-formats. But both bytes and bits are flat data-formats. There is no inherent structure in bytes.

There has recently been increased interest in data-serialization formats – JSON, YAML, etc. One historical and notable serialization format is the S-expression. However, the S-expression suffers precisely from the fact that it has no binding *machine specification*. JSON and YAML suffer from a similar drawback. And none of these serialization formats would be suitable for representing a DVD-length movie, for example.

The Babel Virtual Machine is Babel’s answer to encapsulation, abstraction and providing the building-blocks for a library base built around promiscuous remote code execution. The bstruct (Babel-struct) is Babel’s answer to the problem of structured data that is smarter than the byte or bit but more concrete than the S-expression as well as being suitable for storing very large binary objects such as video files.

Bipedal

Babel has one front-end, called Babel Program Description Language (or Bipedal) but any number of front-ends could be built – using different syntax or even a superficially different language – which compile down to native Babel code. The examples in this document will be given in Bipedal.

The primary design goal in Bipedal has been to make the syntax reflect the underlying data structure that is being described as closely as possible. That this data structure can be interpreted, in certain instances, as a Babel Virtual Machine that has code that does things to a stack when executed on the Babel interpreter is a separate matter.

Like any other postfix language, operands come first, then operators:

2 3 + 4 \*

The above expression, when evaluated, will result in 20.

"Hello, " "world" . <<

When this expression is evaluated, it will place the two strings on the stack, concatenate them and then print them to STDOUT.

Before discussing the internal specification of the language further, I will introduce the Bipedal front-end in order to make the examples clear.

Babel differentiates between **values** and **pointers**. Values are stored in **leaf-arrays** and pointers are stored in **interior-arrays**. In Bipedal, there are two syntactic elements that describe values: a number or a string.

A **number** can be a decimal integer, decimal floating-point, hexadecimal, binary or a p-number (or pnum – support will be added in Babel 2.0). Bipedal differentiates between integer and floating-point based on the presence of a decimal-point and the default is integer.

-42e15  
 32.7152e-12  
 0x13  
 0b1101111  
 0p101111[23]

The limits are dependent on MWORD\_SIZE (use the msize operator to get this) except for pnums which can be of arbitrary size.

A **string** is a set of characters wrapped in quotes:

“Lorem ipsum dolor sit amet”

‘Lorem ipsum dolor sit amet’

… or in a quote-block (see below). The quote character can be escaped with a backslash in the usual way.

**Line comments** in Bipedal are specified by double-dash:

-- This is a comment

Bipedal is organized at the top-level as a set of labeled text blocks which can be nested. A **label** is a string of characters that begins at the *left-edge* and is terminated by a colon:

this\_is\ a\ label!:

The colon can be escaped with a backslash. The first text in a Bipedal file is ignored until the first, left-justified label is encountered.

A label opens an indented **text block** which is terminated either by the end of file or by a dedent. The **indent** must be at least one space. Tabs or any other whitespace cannot be used as an indent character. A line that is less than one space further indented than the left-edge of the current block is a **dedent** and ends the current text block.

A leaf-array is defined by enclosing a string or one or more numbers in square-brackets:

[1 2 3]  
 [“foo”]

If the section is a leaf-array that only contains a single number or single string value, the square-brackets can be removed because they are implied:

foo: 1234  
 bar: “baz”

A leaf-array cannot nest, so the following is illegal:

[1 [2]]

An interior-array is defined by enclosing one or more leaf-arrays, labels or interior arrays in square-brackets:

[[“apples”][“bananas”][“oranges”]]

A labeled text block allows the definition of a **section**. A section contains a leaf-array, a hash-reference or an interior-array and nothing else.

the\_number\_two: 2  
main: ([the\_number\_two] [3] +)

**Comment blocks** are just unlabeled text-blocks that are discarded by Bipedal:

#  
 This is a comment block  
 It is open until de-dent

-- Any dedent closes the comment block

String **quote blocks** permit multi-line quotes:

{ #q  
 This is a quote block – everything in this block  
 is treated as if it were enclosed in double-quotes.  
<< }

foo: #qf  
 This is a folding quote block –  
 no newlines are inserted between lines of the block  
 in the final string produced by Bipedal

**Block constructors** allow you to build lists, interior-arrays, leaf-arrays, code-lists or sub-namespaces without the need to wrap the enclosed block. The following,

foo: #{}  
 2   
 3   
 +

… is equivalent to:

foo: { 2 3 + }

Use #() and #[] to build lists and arrays in the same manner. Block constructors are planned for Babel 2.0.

**Lists** are created with parentheses. A list in Babel means a linked-list in the Lisp-sense. The following are equivalent:

(1 2 3)  
[[1] [[2] [[3] nil]]]

You can verify this with the following code:

(1 2 3) [[1] [[2] [[3] nil]]] ==

… which will evaluate to 0 (memcmp equal). The **nil** element is distinct from Lisp’s in that Babel has no notion of an atomic and nil is just implemented as an entry in the symbol table – but you can take its car or cdr and you will always get nil again.

In Babel, operands in the code-stream are differentiated from operators by being nested in an interior-array. For example, to add two numbers, the following are all equivalent.

A: ( [42] [23] + )  
B: ( (42) (23) + )  
C: ( [42 nil] [23 nil] + )  
D: ( [42 nil] [23 nil] 0x38 )

A-D all produce the same result when executed but A and B do not generate exactly the same byte-code but B, C and D do.

A **code-list** is created with curly-braces. A code-list is just a list but it performs the nesting of non-operators automatically, thus reducing visual clutter. The following are identically equivalent:

( (2) (3) + (x) \* )  
 { 2 3 + x \* }

( ( “Hello, ”) (“world\n”) . << )  
{ “Hello, ” “world\n” . << }

The **hash-reference** is the third basic sub-type alongside interior-arrays and leaf-arrays that together comprise the bstruct type in which all Babel data is stored. Hash-references should not be confused with hashes. To hash something using the built-in hash function, use the %% operator. For example:

{ “nil” %% }

… yields:

[ 0x3023f4e7 0x8c2f644d 0x71cf647b 0xe974b23a ]

But a hash is still only a leaf-array. In order to create a hash-reference, use the newref operator:

{ “nil” %% newref }

**In-place expansion** allows a section or string to be expanded in a manner similar to a #define in C or string-interpolation in Perl. Note that in-place expansions do *not* take arguments and are *not* macros. For example, the following:

foo: (1 2 3)  
 bar: (foo\* 4 5)

… is equivalent to:

bar: (1 2 3 4 5)

Note that you can only use like with like – interior arrays can be expanded only into interior arrays, leaf arrays only into leaf arrays, lists only into lists and strings only into strings (similar to Perl’s string interpolation). The only exception is that all of code-lists, interior-arrays, lists and numbers (but not any other kind of leaf-array) can be in-place expanded into a code-list.

hi: “hello”  
 hw: “hi\* world” --> evaluates to “hello world”

Only double-quotes (and interpolating quote blocks) will perform in-place expansion of strings. Use single-quotes if you want to use asterisks without having to escape each one.

If you plan to use a section exclusively for in-place expansion, you can append an asterisk to the section-definition label. The following,

foo\*: [bar baz]  
 bop: [foo doo wop]

… is equivalent to:

bop: [bar baz doo wop]

Note that in-place expansions into code-lists are auto-wrapped like all other non-operators. The following,

foo: 0x13  
 bar: { foo\* 1 + }

… is equivalent to:

bar: ( (foo\*) (1) + )

When an asterisk is the trailing character of a section label it is called a **sigil** (magic symbol). There are other sigils, as well. Appending a sigil to a section label or to a symbol imparts some magical property to it and alters its default behavior.

The default behavior of a **bare symbol** varies depending on the context in which it occurs. In a leaf-array, the default (and only) meaning of a symbol is in-place expansion. You cannot use sigils in a leaf-array. For example, the following:

foo: [1 2 3]  
 bar: [foo 4 5]

… is equivalent to:

bar: [1 2 3 4 5]

If the symbol cannot be in-place expanded into the leaf-array, an error will occur.

In an interior-array or list, the default meaning of a bare symbol is **direct-reference**. A direct-reference to ‘foo’ refers directly to the thing called ‘foo’ at compile-time rather than performing a lookup in the symbol table at run-time.

foo: [1 2 3]  
 bar: [foo [4] [5]]

Equivalent to:

bar: [[1 2 3] [4] [5]]

Note that, unlike in-place expansion, a direct-reference to foo is actually a pointer to foo and, therefore, preserves the nesting but *does not make a copy*. The same is true of lists:

foo: (1 2 3)  
 bar: (foo 4 5)

Equivalent to:

bar: ((1 2 3) 4 5)

If you want a hash-reference in an interior array or a list, use the sigil, percent (%):

foo: [1 2 3 bar%]  
 bar: (1 2 3 bar%)

In a code-list, the default meaning of a bare symbol is a hash-reference. In Babel code, a hash-reference can have one of two effects – it can either auto-eval a named section or it can auto-lookup data from the symbol table and place it on the stack. The default sense is to auto-lookup data. If you want to auto-eval it, you can use the auto-eval sigil bang, (!):

add2: { 2 + }  
 foo: { 3 add2! }

Note that this has exactly the same effect as:

foo: { 3 add2 eval }

Most of the time, you will likely want to simply define the section with the auto-eval sigil:

add2!: { 2 + }  
 foo: { 3 add2 }

If you want a direct reference in a code-list, use the sigil ampersand, (&):

foo: (1 2 3)  
 bar: { foo& # }

Equivalent to:

bar: ((foo) #)

A special-case for code-lists is in-place expansion of a number. You cannot in-place expand a string into a code-list but you can in-place expand a number. The use of this special-case is for naming the built-in operator opcodes. You can use this feature to alias the built-in operators to a name of your preference:

sumar\*: 0x38  
 foo: { 2 3 sumar } --> evaluates to 5

Bipedal will check the opcode value and prevent you from using any reserved opcode or an opcode in the extended operator space above 0x1000.

Finally, an **unenclosed bare symbol** is interpreted as if it is enclosed in an interior-array and follows those rules (see above).

foo: bar

Equivalent to:

foo: [bar] --> interior array

Anatomy of Babel

The Babel language is inseparable from its implementation specification. Babel does not attempt to abstract away implementation details. Implementation matters – just look at the complexity of IEEE 754.

**MWORD** is short for machine-word and stands for an unsigned int or pointer whose size is the native word size of the machine. In other words, an mword is either a 32-bit or 64-bit unsigned int or unsigned int pointer that is treated like a void pointer (to put it in C terminology). It is conceivable that a Babel interpreter and bstruct could be specified with an unusual machine-word width. **MWORD\_SIZE** is the native word size (in bits) divided by 8 bits per byte. On 32-bit machines, MWORD\_SIZE is 4. On 64-bit machines, it is 8. Use the msize operator to determine the machine word size.

The data structure in which all data and code – including all execution state and meta-data – reside is called a **bstruct** which stands for Babel-struct. It is important to note that the Babel interpreter does not maintain any state which is not stored in a bstruct. This means that a bstruct at all times[[1]](#footnote-1) contains a complete image of the running program. This makes it trivial to load, save and restore Babel programs, even while they are mid-stream.

A bstruct may consist of:

* a single leaf-array OR
* a single hash-reference OR
* an interior-array that may point at one or more interior-arrays, leaf-arrays and/or hash-references

A bstruct is (intentionally) defined in such a way that it may contain any sort of data. There is nothing specific to Babel about a bstruct. For example, a large bitmap or video file can be stored as a leaf-array by simply prepending it with the appropriate s-field in memory.

To tell apart the three types of array, each array has an **s-field**. The s-field is a single mword at position 0 of an allocated array. That is, the array can be freed by passing a pointer to the s-field to the bfree() function. This permits clean destruction of a bstruct at any time. For any X in a bstruct:

* X.s > 0 X is a leaf-array
* X.s = 0 X is a hash-reference
* X.s < 0 X is an interior-array

Aside from telling the array type, the s-field also tells the array size. The size is encoded in bytes so you have to divide by MWORD\_SIZE to get the size in mwords.

* X.s > 0 X.size = X.s / MWORD\_SIZE
* X.s = 0 X.size = 1 + HASH\_SIZE
* X.s < 0 X.size = -1 \* (X.s / MWORD\_SIZE)

The size of every array in a bstruct is an even multiple of MWORD\_SIZE. The least-significant bit of the s-field is used during traversal of a bstruct, see below.

All other mwords in an array other than the s-field are called **entries**. The zeroth entry is located at array index 1, immediately following the s-field.

A **leaf-array** contains one or more values stored in the entries of an array of mwords. The contents of a leaf-array do not have to be accessed in mword-aligned fashion. For example, it is possible to access a particular byte in a leaf-array. The defining feature of a leaf-array is that it cannot contain pointers to any other arrays.

A leaf-array may also be stored as an **array-8**. Array-8 is just a convention for padding the last mword of a leaf array with a special mword that indicates the byte length of the array. Babel strings are stored as array-8 in native form, see the Strings section below.

An **interior-array** contains one or more pointers, each mword-sized. The defining feature of an interior-array is that every mword in an interior-array must contain a pointer to the *zeroth entry* of any array or hash-reference within the bstruct. A pointer in an interior-array may point at any other kind of array or hash-reference. Every pointer in an interior-array must contain a valid pointer (no dangling or misaligned pointers are allowed).

A **hash-reference** is a single hash value stored in memory suitable for fast lookup in the symbol table. Hash-references have several uses:

* By-name lookup of data
* By-name eval
* Creating soft-links that can emborder data-structures

By-name lookup of data is performed by using the hash value to probe the sym\_table. The result will be pushed on the stack. If you want to emborder a given data-structure so that the deep operators of Babel do not continue traversing into other data-structures that are pointed to by the given data-structure, you can use a hash-reference. The built-in operators will stop traversing once they reach a hash-reference.

A bstruct is a graph, not a tree. Hence, it may contain cycles. Bstruct **traversal** requires that each array be marked as it is visited. The least significant bit of the s-field is used for this. The bstruct is actually traversed twice, once to set the LSB of each s-field in the bstruct and once to clear it again.

As a convention, the zeroth entry of an interior-array is also termed the 'car' field and the first entry is also termed the 'cdr' field. This permits the construction of bstructs formally identical to **lists** in Lisp. Babel borrows the 'cons' 'car' 'cdr' etc. terminology from Lisp for this purpose. "A list" always means the portion of a bstruct that conforms to this convention. Note that **nil** behaves slightly differently in Babel than it does in Lisp. Since Babel does not have a notion of an atomic value, nil cannot be an atom and never acts like one. Its primary use in Babel is to mark the end of a list. It also serves some special functions in control-flow behavior.

Hashing is performed with a modification of Pearson's 8-bit permutation **hash** to generate 16-byte (128-bit) hash values. The hash is implemented in a manner that permits what I call **progressive hashing**, see the Appendix for more details. Babel implements **extendible hashing** (Fagin, Nievergelt, Pippenger, Strong, 1979) which means the hash table never needs to be re-hashed when items are inserted or deleted.

In Babel, **namespaces** are implemented through nested hash-tables. A namespace is just a label by which to refer to something. It is not a container, object or package. Everything lives in one namespace, so you cannot have a variable with the same name as a function, etc.

The symbol table is just a namespace but the operators for using hashes can be used by the user to maintain user namespaces. The key is to separate the idea of namespace-as-data-structure and the Babel namespace. The former will be referred to as simply *namespace* and the latter will be referred to as the symbol table.

The /babel/path namespace maintains a list of paths similar to Perl's @EXPORT variable. You can manipulate this variable using any of the applicable Babel operators.

Babel has native support for UTF-8 encoded **strings**. Babel strings are not null-terminated. However, a Babel-string stored in array-8 form is always C-string safe because the alignment-word at the end of an array-8 always contains one or more null bytes. For example, in 32-bit Babel, the alignment word is one of:

* 0x00000000, when byte-length mod 4 = 0
* 0xffffff00, when byte-length mod 4 = 1
* 0xffff0000, when byte-length mod 4 = 2
* 0xff000000, when byte-length mod 4 = 3

Babel handles strings in several different forms:

* Native form. The string is UTF-8 encoded WITHOUT a null terminator in an array8 leaf-array
* C-style. This is just a native string with a null terminator appended. Use the pad operator to append a null terminator to a native Babel string.
* String-array. This is a leaf-array such that each entry in the array contains the Unicode code-point of the encoded character. It is created from a native-form string via the str2ar operator.
* String-list. This is a string-array on which the ar2ls operator has been called.

Babel Virtual Machine

The core of Babel is the interpreter which reads and executes a Babel Virtual Machine or BVM. A BVM is a *data-structure* – it is a bstruct that contains all the data required by the Babel interpreter. This underscores the fact in elementary computer science that *a language is a machine* and vice-versa. In fact, it is impossible to fully encapsulate a virtual machine unless it has been specified as a data-structure.

A BVM is actually a hash-table stored in a bstruct which has in it entries called ‘code\_ptr’, ‘stack\_ptr’, and so on. This hash-table forms the root of the BVM. BVMs may be nested arbitrarily and are invoked with the babel operator. In fact, every Babel program executes inside of an invisible BVM that is compiled into the babel executable (see src/rt.pb). This BVM contains code for the debugger and other basic commandline and house-keeping functions.

BVM **code** is a list. Each element of the list is accessed in order. When the item pointed at by **code\_ptr** is:

* A leaf-array, it is treated as an **opcode** and a lookup is performed in jmp\_table
* A hash-reference, it is looked up in the sym\_table and control is transferred there, in a manner equivalent to eval
* An interior-array and its car is:
  + Not a hash-reference, it is pushed on the stack
  + A hash-reference, it is looked up in the sym\_table and the result is pushed on the stack

The BVM **stack** is where all operations are performed in Babel. The Babel interpreter is a stack machine. Each operator operates on the stack and returns its results on the stack. There are no registers in Babel. No internal state is maintained by the interpreter in variables except in a cache maintained in a thread-safe way.

The stack is itself a list. However, it includes some additional information for memory-management. In order to convert the stack to a list, use the take operator and use the give operator for vice-versa.

The stack is actually comprised of two components – the dstack and the ustack. The dstack is the down-stack and the ustack is the up-stack.

The **rstack** is the managed stack. It is what is used to implement the iteration, control-flow and stack nesting operators.

The **interpreter** is a stack machine (postfix order). This means that compilers that generate Babel code do not need to use parentheses to implement operator precedence. The precedence is encoded in the order of operations. This simplifies the parsing requirements of any Babel front-end language.

The **hidden** section contains limits and controls that restrict what the BVM can do. For example, if you are launching a BVM fetched from the web you should disable operators that can write to disk, limit the memory that it can allocate, taint data fetched locally (to prevent privacy breaches), disable system call operators, disable nested virtual machines (to prevent stack-overflow attacks) and disable operator extension.

ustack\_ptr, jump\_table, sym\_table, stack\_ptr, advance\_type, steps

----- Jump table

Each built-in opcode is an offset into a jump table. New opcodes that are

added in with the newop operator are dynamically assigned jump table offsets.

When constructing a bvm for launch, the parent bvm can restrict the built-in

operators that are available through the hidden section of the header.

Babel Operators

The active component of Babel code consists of any of a number of operators. There are two types of operators: built-in and extended. Each operator is invoked through the jmp\_table.

Babel’s **built-in operators** have a fixed numerical value below 0x1000. There are several hundred built-in operators and they can be roughly categorized as follows:

* C-style arithmetic operators
* Shift operators
* Bitwise and standard logic operators
* Comparison operators
* I/O operators
* bstruct operators
* Array operators
* String operators
* List operators
* Hash operators
* Stack operators
* Flow-control operators
* Iteration operators
* BVM operators

See the Appendix for details on specific operators.

The encodings for **extended operators** have a value greater than 0x1000 but they are not fixed. An extended operator will be installed in the next available jmp\_table entry when it is installed. Babel code should never attempt to directly rely on the encoded value of an extended operator.

An extended operator can be installed and given an encoding with the newop operator. Extended operators can also be invoked by hash-reference. If the next entry in the code list is a hash-reference, a lookup will be performed in the sym\_table and the linked code will be invoked. Naturally, this is a lower-performance alternative.

Memory Management

The memory management specification of Babel is still in flux and will settle down by Babel 0.9 release.

Babel uses a combination of automatic and manual memory management. Certain operators – such as nest or let – will create a **memory context** – all memory created within that context is subject to destruction at the end of the context unless it is promoted to the next higher memory context using the keep operator.

To signal early consumption of a stack value (before the end of the memory context), you can use the consume operator. When a value is zap’d from the stack after being marked by the consume operator, it will be del’d as well.

The scaffolding used by the stack, rstack, hashes, etc. is automatically memory-managed. When you push or pop on the stack the memory associated with the scaffolding is automatically created or destroyed.

Memory can be created and destroyed manually using the new\* and del operators.

new\* -> always returns a hash-ref?

User operators

User operators can provide a destructor function to Babel when they are registered. This function will be called whenever the operator's results are consumed by another operator (built-in or otherwise).

To signal consumption of a value (removal from the stack) to other operators, call the zap() function.

----- Namespace memory management

Memory destruction within a namespace is hierarchical. This means that if

you create /foo/bar and then destroy /foo, /foo/bar will also be

destroyed. Note that simply removing a namespace does not affect memory

allocation, it's when you use the del operator on a namespace.

The book-keeping data in a namespace is always automatically memory-

managed.

----- Hash memory management

The book-keeping data in a hash is automatically memory-managed.

----- Mortality

Each item on the stack is either mortal or immortal. Mortal items are freed

when they are consumed by an operator. Immortal items are not freed.

When you create new memory:

`10 newin

... it is mortal. When this object is zap'd, it will be freed. If you want

the object to be freed when it is zap'd, use the immortal operator:

`10 newin immortal

P-numbers

Babel 2.0 will have built-in support for arbitrary precision 2-adic numbers, called p-numbers or pnums short. The pnum operators (padd, psub, etc.) implement this functionality.

To calculate the square-root of two to five words of precision: [5] [2] psqrt

----- File I/O

Babel provides some "quick-and-dirty" I/O operators inspired by Perl's

slurp functionality.

For more robust file functionality, Babel provides memory-mapped files.

----- Operand-checking

Operands are checked for each operator. Babel 2.0 may include optimized

non-checking operators for use in well-tested code.

1. “at all times” means whenever an operator has finished executing – caching is permitted but it must be thread-safe [↑](#footnote-ref-1)