Babel

Clayton Bauman  
June 2, 2012

Introduction

Why switch to Babel?

Babel makes common programming tasks easy while maintaining excellent performance. To open and read an entire file named ‘foo.txt’, use the following code:

“foo.txt” >>>

The file is automatically closed for you. It is now in Babel and ready to be operated on.

Babel eliminates the need for variable definitions in many instances, as the file reading example above shows. Only a few more operators are required to split the file into lines:

“\n” char\_match carve

Now, you can split each line into individual words – separating on the space character – and print each word on its own line:

{ “ ” char\_match carve cr << } foreach

So far, we have not defined a single variable. This is because Babel is a stack-based, postfix language. As long as you only need to operate on the last thing you just operated on, you can chain together any number of operators without the use of variables.

Declaring constant data is very convenient in Babel. Taking its cue from the YAML data-serialization format, Babel offers space-indented sections:

foo:  
 bar: 13  
 baz: “Hello, world”

In combination with the cp (copy) operator, you can easily use space-indented sections to define struct-style structures that can include code, thus making them not unlike a fully-public C++ class.

Babel is untyped which relieves the code of having to specify types along with every variable. This makes Babel code more readable.

Babel performs automatic memory allocation for you. For example, to assign the value 10 to the variable foo:

foo 10 set

You do not need to “declare” foo or tell Babel the type of foo.

Babel makes common string operations easy. To concatenate two strings, simply concatenate the array-8 arrays containing them:

“Hello, ” “world\n” cat8 <<

To reverse a string, convert it to a list and reverse the list:

“Babel” str2ar ar2ls reverse ls2lf ar2str << --> prints lebaB

Babel also makes it easy to containerize your data objects so they can be easily saved to and restored from file. Imagine you have a complex data structure:

net\_topology unload “topo.bbl” <<<

This will store whatever was in net\_topology out to the file named topo.bbl. You can reload it again from file sometime later:

net\_topology “topo.bbl” >>> load set

Note that it can even be saved and restored between exit and re-entry to the Babel interpreter. This makes it convenient to save your program state at any point for later resumption. You do not need to think about how to represent your objects on disk. If you want your data to be available in human-readable format, Babel also makes this simple.

Extending this containerization further, Babel makes it easy to save and restore the entire state of the interpreter itself. To save and exit:

save\_and\_exit conjure

The “conjure” operator causes the current virtual machine to call its parent. In this case, the user virtual machine is invoking the save\_and\_exit operator in the root virtual machine. The root virtual machine will use the unload operator to save the user’s virtual machine to file. You can then resume execution of the saved virtual machine in the same way you invoke any Babel program.

Babel also allows you to launch *nested* Babel programs, making it easy to manage the risks of executing arbitrary code. Built-in crypto makes it easy to enforce a "white-list" code execution policy, reducing the risks associated with executing remote code.

Babel makes it easy to perform deep-copies of your data-structures. Suppose you have a complex data structure named ‘net\_topology’, the cp operator will make a deep copy of the data structure:

net\_topology cp

If you’re finished using a data-object and you know that only one reference exists to the data structure, you can use the del operator to free all memory associated with the data structure:

net\_topology del

If there is more than one reference to the data structure, del’ing it will result in an inconsistent BVM and could result in an interpreter crash. If you are not certain that there is only one reference to your data structure, use the safer (though slower) destroy operator:

net\_topology destroy

Babel stores your data structures in an underlying, uniform data structure called a bstruct – this makes it trivial to save and restore them to and from disk, to make deep copies of them, to delete all memory associated with them, and so on.

Babel also enables you to easily compress and encrypt your program data objects, (or whole Babel programs).

Babel provides both Lisp-style lists and array structures, permitting you to organize your data in a manner to maximize both flexibility and performance. Modern computer architecture is array-based and can perform array lookups many times faster than list-traversal. But for data that is constantly changing size or continually undergoing complex permutations, lists are a clear performance winner by permitting you to perform more of your operations in-place.

Babel provides a comprehensive suite of abstract data-structures, including array, list, hash (map/dictionary) and namespace (or directory). This gives you the tools you need for rapid prototyping of very complex code even if you eventually choose to implement your solution in a lower-level language and link it in.

The Babel namespace is modeled on a Unix-style file directory structure, making it possible to nest data and code in arbitrarily deep and complex structures. Manipulating namespace paths permits implementation of OO-style templates, polymorphism and more.

Arbitrary-precision arithmetic is built right into the Babel core, making it easy to implement crypto algorithms in an academic (non-optimized) fashion. Other applications such as desktop calculation are also facilitated.

Babel makes visualizing your data uniquely easy with Babel's built-in support for generating Graphviz dot files. These can be post-processed with the dot tool to generate graphical snapshots of your data. This speeds up debug and helps the programmer fully absorb the semantic significance of the syntax. We will utilize this capability in the next section.

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.

Bipedal is UTF-8 encoded – this means you can give your sections any name that can be encoded in UTF-8. You can easily alias the built-in operator names, as well.

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  
 0p(101)1.11e-5

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 followed by a colon or equal-sign:

this\_is\ a\ label\!\@ :  
 this\_is\ a\ label\_too:  
 this\_is\ a\ label =  
 and\=so\=is\=this=

Any character can be used in a label, including the colon and equal-sign. However, any ASCII character that is not an **identifier-character[[1]](#footnote-1)** must be escaped with the backslash. Non-ASCII Unicode characters can be used freely. 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 labeled text block allows the definition of a **section**.

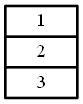
my\_section : ( 10 9 8 7 6 5 4 3 2 1 “Blast off!” )

Every section is entered into the symbol table at compile-time. Hence, every section label is a symbol. However, symbols can also be created dynamically so *not every symbol is a section*.

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

[1 1 2 3 5 8 13 21]  
 [“Smart men are named Leonardo”]

This is how the array [1 2 3] is represented internally:



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

foo : 1234  
 bar : “baz”

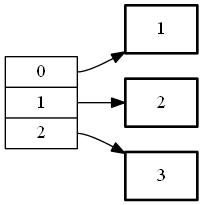
A leaf-array cannot nest and an interior-array cannot contain values, so the following are illegal:

[“speeding” [“jaywalking”]]  
 [[“littering”] “loitering”]

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

[[“the”][“cuckoo’s”][“nest”]]

This is how the array [[1] [2] [3]] is represented internally:



Note that values are represented in bold boxes, pointers are represented in standard boxes.

Because the square brackets are overloaded between leaf-arrays and interior-arrays, it is possible to have ambiguous sections:

A : [B]  
 B : [A]

Whenever Babel cannot unambiguously resolve the meaning of square-brackets in two or more sections, it will treat them as interior-arrays. So, the above example would resolve to section A being an interior-array with a single direct-reference to section B and vice-versa.

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

#:  
 I have…  
 no comment

-- Any dedent closes a text 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  
 and the left margin will be de-dented in the final  
 string produced by Bipedal.

qf#:  
 This is a folding quote block –  
 no newlines are inserted between lines of the block  
 and the left margin will be de-dented in the final   
 string produced by Bipedal.

Sub-namespaces can be created using the #: notation:

( #:   
 foo: 0x13  
 bar: 0x42

#:   
 foo: 0x777  
 bar: 0x21 )

This example represents a list of hashes. You can also use #() or #{} to reserve a list of namespace names:

foo :  
 #( a b )

≡

foo :  
 a& :  
 b& :

And:

foo :  
 #{ a b }

≡

foo :  
 a :  
 b :

An empty namespace defaults to nil:

foo :  
 ≡  
 foo : [nil]

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

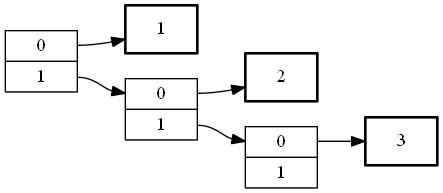
(1 2 3)  
[[1] [[2] [[3] nil]]]

You can verify this with the following code:

{ (1 2 3) [[1] [[2] [[3] nil]]] eq }

… which will evaluate to 1 (true).

This is how the list (1 2 3) is represented internally:



Entry number 1 of the final cons in the list implicitly points to nil. Any pointer which is not marked as pointing at something else is understood to be pointing at nil.

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 keeping with the spirit of Lisp, you can also notate nil as empty-parentheses or empty-brackets:

[] ≡ () ≡ nil

One final difference between Bipedal syntax and Lisp syntax is that Bipedal does not use the dot-notation. In Lisp, the following syntax

(1 2 . 3)

… expresses the same list as the following written in Bipedal:

(1 [2 3])

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. 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. It also alters the meaning of labeled references. The following are identically equivalent:

( [2] [3] + )  
 ≡  
 { 2 3 + }

And:

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

You can check that this is the case:

{ { 1 2 3 “Hello, world” } ( [1] [2] [3] [“Hello, world”] ) eq }

Note that the default meaning of labeled references is different depending on whether you are in code-list or data-list context.

The **tagged-pointer** 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. A tagged-pointer is created using the angle-brackets:

< tag stuff >

The stuff that follows the tag (if any) is dependent on the tag. Certain built-ins, such as nil, are implemented internally using tagged-pointers.

**Sigils**

Babel permits the use of a **sigil** (magic symbol) to specify the meaning of a label or symbol. There are six sigils:

|  |  |  |  |
| --- | --- | --- | --- |
| **Sigil** | **Magical property** | **Bare symbol equivalent in** | **Mnemonic** |
| \* | Lexical substitution | Leaf-array | “Contents-of” (C) |
| & | Direct reference | Interior-array and list | Reference (C++) |
| $ | Indirect reference | - | Reference (Perl) |
| % | Hash-reference | Code-list | Hash-sigil (Perl) |
| ! | Auto-eval hash-reference | - | Imperative … do it! |
| @ | List-location | - | Where it’s “at” |

Appending a sigil to a section label or to a symbol imparts that magical property to it and may alter its default behavior. A sigil appended to a label **overrides** the default sense of that label in *any* context. However, you can **double-override** the symbol by appending a different sigil in a particular context:

foo% : 127  
 bar : ( foo& ) --> Overrides foo back to a direct-reference…

The default behavior of a **bare symbol** varies depending on the context in which it occurs.

The asterisk (\*) specifies that a symbol or section is to be used for **lexical substitution**. Lexical substitution allows a section or string to be expanded in a manner similar to a #define in C or string-interpolation in Perl. Note that lexical substitutions do not take arguments and are not macros.

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

≡

bar : (1 2 3 4 5)

The order in which you define sections is maintained by Bipedal. In the above example, foo has precedence over bar.

Note that you can only use like with like – interior arrays can be substituted only into interior arrays, leaf arrays only into leaf arrays, lists only into lists, code-lists only into code-lists and strings only into strings (a compile-time version of string interpolation).

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

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

If you plan to use a section primarily for lexical substitution, you can append an asterisk to the section-definition label.

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

≡

bop : [bar baz doo wop]

And:

add2 : { 2 + }  
 bar : { 1 add2\* }

≡

bar : { 1 2 + }

In a leaf-array, the default (and only) meaning of a symbol is lexical substitution.

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

≡

bar : [ 1 2 3 4 5 ]

Gratuitous use of sigils is permitted:

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

You cannot use any other sigil than the asterisk on a symbol that occurs in a leaf-array. If the symbol cannot be lexically substituted into the leaf-array, an error will occur.

A **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. There are two types of references – **direct references** (pointer) and **indirect references** (pointer-pointer).

In list context or interior-array context, the default meaning of a bare symbol is as a direct reference.

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

≡

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

The same is true of lists:

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

≡

bar : ((1 2 3) 4 5)

Note that, unlike lexical substitution, a direct-reference to foo is actually a pointer to foo and, therefore, preserves the nesting but *does not make a copy*. If you want to nest and make a copy:

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

If you want an indirect reference in list, interior-array or code-list context, use the dollar-sign sigil ($):

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

≡

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

Note the extra pair of brackets. As a general rule, you should never need to use indirect references in interior-array or list context.

foo : (1 2 3)  
 bar : { foo$ len }

≡

bar : ( [foo] len )

In code-lists, if you want a direct-reference, use the ampersand sigil (&):

foo& : 0x38  
 bar : { foo }

≡

bar : ( 0x38 )

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

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

In a code-list, the default meaning of a bare symbol is as a hash-reference.

foo : (1 2 3)  
 bar : { foo len }

Equivalent to:

bar : ([foo%] len)

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 }

The auto-eval sigil can only be used in code-list context.

The location sigil (@) can never be used as a section definition label or in array context. It allows you to create direct references within a list or code-list. For example:

foo: ( 1 2 three@ 3)  
 bar: { foo/three car %d << }  
 ≡  
 { foo cdr cdr car %d << }

You can use this facility to create a “compile-time hash” that permits your code to reach the elements of a list but without the overhead of run-time hashing. For example:

( first\_name@ “John”  
 last\_name@ “Smith”  
 ID@ “19010378” )

An **unenclosed bare symbol** is treated as a simple alias.

foo : bar

Anywhere foo is used, it will be as if bar had been used instead. You cannot use sigils on either side of an alias statement and the alias inherits any sigil attached to the aliased section. You cannot alias a location label. Aliases have global visibility, so you can alias any section by using its full name. You can use this feature to alias the built-in operators to a name of your preference:

suma : +  
 foo : { 2 3 suma }

If a symbol is not defined by a labeled section in your Bipedal file, it is treated as if it defines a symbolic section. That is, use of an otherwise undefined symbol implicitly creates an entry in the symbol-table for that symbol. This is the case regardless of whether the symbol refers to a symbolic variable or a reference variable.

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[[2]](#footnote-2) 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.~~

The data structure which forms the basis of the Babel language and all Babel data is called a bstruct. A bstruct may consist of:

* a single leaf-array OR
* a tagged-pointer that may point at an interior-array, leaf-array or another tagged-pointer or itself OR
* an interior-array that may point at one or more interior-arrays, leaf-arrays and/or tagged-pointers

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 tagged-pointer within the bstruct. Every pointer in an interior-array must contain a valid pointer (no dangling or misaligned pointers are allowed).

A **tagged-pointer** is a single hash value and pointer. Only bstructs that construct a symbol-table within Babel may contain tagged-pointers. Certain pre-defined tagged-pointers are recognized by Babel and used to construct a symbol table that may, in turn, be used by other tagged-pointers. A tagged-pointer must indicate whether it is traversable and whether the memory it points to is managed by Babel or external code.

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. Nil is implemented as a built-in tagged-pointer.

The bstruct is almost completely devoid of meta-data - the s-field is the *only* meta-data in a bstruct. This is to keep the bstruct useful for many kinds of data, even data that has nothing to do with the Babel interpreter (perhaps even your dynamic data-objects). All other meta-data is defined as a higher abstraction layer and simply *stored* in a bstruct. A bstruct has no idea what is or is not a hash-table. A hash-table is just like any other kind of data stored in a bstruct. The same goes for the Babel Virtual Machine itself.

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. In a Babel Virtual Machine (BVM), 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 array-8 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.

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. 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 jump\_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 whenever it is consumed by an operator

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 some or all of the stack to a list, use the take operator and use the give operator for vice-versa.

The BVM stack is actually comprised of two sub-stacks – the dstack and the ustack. The dstack is the down-stack and the ustack is the up-stack. When Babel auto-pushes an operand onto “the stack”, it is really pushing it onto the dstack. By using the up (->) and down (<-) operators, you can move up and down the stack. This can often reduce the need to re-arrange items on the stack, reducing stack noise.

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

The **interpreter** is a stack machine (postfix order). 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.

Each built-in opcode is an offset into the **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.

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 doc/babel\_ref.txt in the Babel source code repository 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 jump\_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.

Dynamic Environment

The bstruct and BVM provide a static, snapshot view of Babel. When a BVM is invoked, that is, when it begins to be interpreted, it enters the dynamic environment. In line with the philosophy of syntactic transparency used in the syntax of Bipedal, the Babel dynamic environment is designed to minimize behind-the-scenes magic.

The dynamic environment consists of the following elements:

* Code
* Data stacks (dstack, ustack)
* Managed stack (rstack)
* Symbolic variables
* Automatic variables
* Dynamic context (variable lifetimes)
* Dynamic scope (namespaces)
* Instantiation
* Type information, access rights, etc. (namespace meta-data)
* Interpreter environment (argv, path, permissions and limits, etc.)

There are some important differences between the dynamic environment of Babel and other dynamic languages, such as Perl or Python. The code-list syntax in Babel provides no guarantee of “constness” of the code-stream. For example,

foo : { code\_ptr { **1** 1 += } 10 times dump << }

This code saves the code\_ptr, executes the loop 10 times, then dumps the original code to Graphviz dot format and prints it to STDOUT. When you look at the Graphviz dump of the code, you will see that the operand in bold is not 0x1 but 0xb (11 in hex) – this is because the += operator auto-assigns its result to the second operand. If that operand happens to be something in the code-stream, there is nothing in Babel that prevents this kind of code self-modification from occurring.

I know that some people will find this off-putting but there is a great deal of added complexity required to protect the user from himself which violates the design goal of transparency in both the bstruct and the dynamic environment.

The root issue here is deciding whether a particular portion of code requires by-value copies of operands in the code-stream or not. The following code *will* behave as the naïve user expects:

foo : { code\_ptr { 1 **cp** 1 += } 10 times dump << }

The loop is useless and will leave nothing on the stack but it will not modify the code-stream, either, because a by-value copy of the destination operand has been made. Each time through the loop, a pointer to the 1 in the code-stream is placed on the stack, a *copy* of that value is made (the original is removed from the stack) and then += is executed removing the top two operands from the stack, leaving the stack empty.

Both Babel and C perform the by-value copy of y dynamically but, in Babel, it must be done explicitly. Insomuch as we can speak of “passing parameters” in Babel, all parameters are either pass-by-direct-reference (pointer) or pass-by-hash-reference.

When you create a new entry on the stack using the new operator, it is allocated on the heap and it is given an entry in the symbol table which is then a **symbolic variable**. The same thing occurs for the symbolic sections that are defined in your program. The dynamic environment does not distinguish between symbolic variables that are created dynamically and symbolic variables created by sections. Both sections and symbolic variables are accessed by hash-reference.

In Babel, **instantiation** is the result of using the new operator. There are two ways to instantiate something – by direct-reference or by hash-reference.

{ nil leaf 2 new }

The nil tells Babel to instantiate by direct-reference and leave the direct-reference on the stack. You can instantiate a leaf-array, interior-array or list of any desired size:

{ nil leaf 2 new   
 nil interior 3 new  
 nil list 4 new }

Dynamic equivalent of:

{ [0 0] cp   
 [nil nil nil] cp   
 (nil nil nil nil) cp }

To instantiate by hash-reference, give the symbolic variable – instead of nil – that you want to hold the instantiated object:

{ foo leaf 2 new }

Dynamic equivalent of:

{ foo [0 0] cp set }

You can optionally skip the size:

{ foo leaf new }

Dynamically equivalent to:

{ foo [0] cp set }

You can instantiate a section:

bar : 3  
 baz : { foo bar new }

This is the dynamic equivalent of:

baz : { foo bar cp set }

You can instantiate lists:

bar : (1 2 3)  
 baz : { foo bar new }

… is the dynamic equivalent of:

baz : { foo (1 2 3) set }

And you can instantiate code:

x& :   
 bar : { 1 x + }  
 baz : { foo bar 3 new }

… is the dynamic equivalent of:

x& :  
 baz : { foo ({ 1 x + }{ 1 x + }{ 1 x + }) cp set }

Anything that can be described in a Babel section can be instantiated on the stack or as a symbolic variable. This permits you to implement something not completely unlike a public class in C++ (a class where all attributes and methods are declared in the public: section). On each instantiation, a new copy of the original namespace is created, including fresh copies of lexical variables and so on. You can also construct a symbolic variable dynamically and then instantiate that.

In contrast to symbolic variables, a **lexical variable** can be accessed by either direct-reference or hash-reference. Lexical variables are created by the let operator. The defining feature of lexical variables is that they are saved on entry to a new **lexical context** and restored on exit from that lexical context. This is the same as automatic variables in C. Note that lexical variables are *not* part of automatic memory management.

Babel diverges from most programming languages in providing no unified syntactic framework for function-calling. This is because Babel does not attempt to present a “functional syntax” to the programmer. Nevertheless, it is always possible to translate function-call code from a different language into Babel.

As an example, let’s say I want to call a function in C that accepts a pointer and a value:

int foo(int \*x, int y){  
 return \*x + y;  
}

This is how you would write it in Babel:

#(x y)  
 foo: { ((x stack) (y stack))   
 { y cp x + }  
 let }

It looks much uglier in Babel because the use of let here is gratuitous.

You can also declare hash-references to be local variables using the let operator. This has the same effect on hash-references as it has on direct-references – the pre-existing value is saved and restored at the end of the local context. In addition, each symbol table entry contains meta-data to track whether it is a pure symbol table entry (no other direct-references to it) or whether it is a hybrid (other direct-references exist). If it is a pure symbol table entry, then it will be freed at the end of the dynamic context before the previous value is restored.

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. Memory is created in one of two ways – directly or indirectly. Direct memory creation occurs with the use of the new operator. Indirect memory creation occurs as a by-product of executing an operator that transforms or combines existing objects, for example cut or cons.

Memory created directly can be managed manually. While reference counts are maintained for memory created by using the new operator, a zero reference count is ignored if the memory has been declared to be an **external reference** using the extern operator.

{ foo interior 10 new extern }

The symbol ‘foo’ now contains an external reference to an interior array of size 10 – a reference count is maintained but ignored since it is presumed that there may be some other program or library that is using this memory. It will not be destroyed until the user explicitly deletes it:

{ foo del }

Note that you cannot save/restore Babel programs that have live external references.

There are two levels of automatic memory management in Babel. The most basic level is stop-copy garbage collection. This is performed one of two ways – either by the user explicitly conjuring the garbage collector:

{ gc conjure }

… or by the occurrence of a memory\_limit exception, in which case, the root BVM will attempt to perform gc on the encapsulated BVM.

The next level of automatic memory management is reference counting. Reference counting is used to manage memory associated with symbolic variables. In contrast to some other languages, Babel eagerly releases reference-counted memory. The reason for this is to avoid having to do a stop-copy garbage collection which could impose a significant and unpredictable pause in execution time.

It should be noted that the internal scaffolding used by the interpreter in managing the stack, rstack, hashes, etc. is, of course, automatically memory-managed. When you push or pop on the stack, the memory associated with the scaffolding is automatically created or destroyed.

Getting Started

Babel was designed with the TIMTOWTDI[[3]](#footnote-3) philosophy in mind – there are many *right ways* to solve any particular problem in Babel. However, some solutions are a bad fit for Babel. In particular, trying to write Babel code to resemble standard function-call code – while possible – will result in sub-optimal, hard-to-read Babel code.

Babel is a stack-based language - the stack is the primary means of communicating data from one place to another. The ideal situation is to reduce your code to a set of sections that can be thought of as *combinators* where each combinator takes a small number or zero operands from the stack and returns one or zero results on the stack. In this case, you will not have “stack-noise” induced by the need to re-arrange arguments on the stack.

The next preferred situation is if your sections take multiple values on the stack and each value is used only once or twice in the section, in order or nearly so. In this case, you can simply use the up and down operators to move cleanly from operand to operand.

But if one or more operands are going to be used in more than a couple places in a section, this will create significant stack-noise which is not only a performance problem but drastically reduces the readability of your Babel code.

In this case, consider using symbolic-variables or lexical variables, depending on your needs. Symbolic variables are entries in the symbol table and are primarily intended for structural or spatial organization. Lexical variables (symbolic variables can also be treated lexically) permit you to make a section re-entrant and are primarily intended for functional or temporal organization.

Many array- and list-processing tasks are most easily done by monopolizing the entire stack with an each loop. To facilitate this, Babel provides the nest operator. Within a nested context, you have a fresh stack – when you return from the nested context, that temporary stack will be destroyed, except for TOS which is returned. You can read doc/babel\_ref.txt for more information on the nest operator.

Also, be sure to check out the many examples of Babel code on rosettacode.org.

1. An identifier-character is determined by whether it matches the following regex: [a-zA-Z\_][a-zA-Z\_0-9]\* [↑](#footnote-ref-1)
2. “at all times” means whenever an operator has finished executing – caching is permitted but it must be thread-safe [↑](#footnote-ref-2)
3. There Is More Than One Way To Do It. From “Perl, the first postmodern computer language” – speech given by Larry Wall on March 3, 1999 [↑](#footnote-ref-3)