[bottom-bracket]

What if we didn’t assume our abstractions; what if we derived them?

Abstract

Bottom-bracket (BB) is a homoiconic language designed to express the compilation of anything to anything through bottom-up abstraction via macros written in anything.

It’s intended to serve as a minimal top-down to bottom-up abstraction turnaround point at as low of a level as possible. It is designed to be as unopinionated as possible.

This is done with compilation of code to machine language in mind, but it’s open-ended.

Using BB without any libraries, you start at machine language with macros. Programming languages are just macro libraries.

Example:

```
[bb/with
 [[data my-macro-expansion [a b c]] ; Some data we'll reference

  ;; Macro - written in machine language - that expands to [a b c] by returning
  ;; a pointer to that structure.
  [macro my-macro
   [x86_64-linux
    [bb/barray-cat
     "\x48\xB8"[my-macro-expansion addr 8 LE]  ; mov rax, data
     "\xC3"]]]]

 ;; Using our macro
 [foo bar [my-macro]]]
```

Expands to

```
[foo bar [a b c]]
```

Beware: it’s not stable yet

Breaking changes should be expected for now. We need to get the core of the language right, and some iteration is inevitable.

Eventually the hope is to build a stable specification for everyone to implement.

Such that you’re not flying completely blind, here are some anticipated breaking changes:
- Parallelized macroexpansion where possible
- Macro I/O details (inputs, return value etc).
- Changes to parameters and interfaces of builtin functions
- Changes to which builtin functions are exposed

This doesn’t mean don’t build stuff with BB. This means use a pinned version of BB for anything you need to stay working, and be ready for migration work.

---

1. Introduction

When we create abstractions, one common approach is to begin with a top-level interface we’d like to have, and then work down towards the layer below working out how to make it happen. This is top-down abstraction, and it’s the default mode of operation for software development today.

There’s another way, though, one pioneered by languages like lisp. Rather than starting from an ideal interface, we start with what exists now, pick a direction we’d like to go, and start working our way up towards a particular problem we’d like to solve. The abstraction that we create is simply the abstraction that logically forms when attempting to move in that direction. This is the bottom-up approach.

Many areas of science were formed using top-down abstraction by necessity. We made high-level observations about the world (salt goes away in water!) and created abstractions for those observations. As we came to understand the underlying mechanisms, the high-level layer was already established - so we ‘make it work’ to make our abstractions logically map together as well as we can. It’s never perfect though. This approach lends itself to abstractions that don’t logically map to eachother very cleanly.

By contrast, mathematics has largely evolved in a more bottom-up fashion. Each abstraction is built upon the previous, and what resulted is a ruthlessly logical and clean system.

These examples illustrate how bottom-up abstraction lends itself to a clean, well-mapped, less leaky design.

Of course, it's never perfect. Every layer leaks to some degree - even with the bottom-up approach - and we just work to keep it to a minimum. The benefit of minimizing abstraction leakage is huge, though: the less each layer leaks, the higher we can stack abstractions without accumulating frustrating behaviors and performance issues.

Bottom-bracket embraces the bottom-up philosophy. It is built for bottom-up abstraction (enabled by macros) to minimize abstraction leakage. In contrast to most lisps, it does not start at a high-level of abstraction, but starts right at the machine-language level.

2. Bottom-bracket’s lifecycle

Upon execution, bottom bracket performs only 3 steps. Read → expand macros → print.

- Read: reads user input using reader macros
- Expand macros
- Print: Outputs result using printer macros

Bottom bracket does nothing more. All behavior of the user’s language is determined by macros.

3. Language details

3.1. The in-memory data structure

The data structure in memory is designed to represent a tree. There are only two data types, which can be differentiated by the length prefix: positive is barray, negative is parray.

The size in bytes of the length values and the size of bytes of the pointers in parrays are platform-dependent (size_t in principle, but will be made more clear in specification when that’s put together).

3.1.1. barray

Array of bytes. Prefixed with with the quantity of bytes in the barray.

3.1.2. parray

Array of pointers to other elements (other barrays or parrays). Prefixed with the **one’s complement** of the quantity of pointers. One’s complement differentiates it from barrays but can still handle the case of zero-length parrays.

3.2. The *default* syntax

Emphasis on **default** because users of bottom-bracket have control over this through reader and printer macros.

- Square brackets [] deliminate parrays.
- All other characters besides whitespace placed next to eachother represent barrays
- Double-quoted strings - byte strings - represent barrays and can use escape codes for bytes.

Examples:
- [foo bar] - parray of two barrays (foo and bar)
- [“foo” “bar”] - Exact same data structure using byte strings
- “xFFx00d042n” - Byte string using escape codes - represents barray of what’s described.
- [] - empty parray
- [foo [bar baz]] - parray containing nested parray

4. Bottom-bracket is a minimal core

Implementations of bottom-bracket itself are intended to be minimal. The version written in x86_64 assembly currently sets around 5,500 lines total.

Generally speaking, if it can be done inside the BB language and not as a builtin, it should be. The builtin macros simply serve as a bootstrapping tool.

The language has no special operators whatsoever. All functionality provided by the builtin macros can be re-created using your own macros. This also means any opinion introduced in these macros is easily changed by the user of the language.

5. What about portability?

Macros can provide multiple implementations - one per platform. Implementations of bottom bracket decide which implemention(s) they support based upon what they know how to execute. Usually that will only be the platform the implementation is running on, but it’s open to virtualization and other tricks.

Portability is not a problem solved at the bottom bracket level, as bottom bracket is intended to be the minimal abstraction turnaround point. Portable languages built using bottom bracket can reference the bb/platform macro to determine what type of machine code they should expand into.

6. Fully verifiable bootstrap is a goal

This type of language is uniquely well-suited to solving certain bootstrapping problems, and building a fully verifiable bootstrap route to the software ecosystem is a goal of this project.

The ultimate goal would be to implement C inside the language.

- The ability to slowly “walk” up abstraction levels in tiny steps makes the lower level stages of bootstrapping much easier
  - The moment you implement a tiny part of any assembler, you can use it. The x86_64 assembler currently living in this repo is a great example of this.
- Implementing C in a language of this design is particularly transparent - everything is just a library.
- Reader macros allow you to turn C syntax into a bottom bracket structure (parrays and barrays).

7. Getting started

7.1. Build an implementation of BB

This are in the impl/ subdirectory of this repository. Exact build process depends on the implementation, but usually the answer is just ‘`$ make`’.

7.2. Run some code!

- barray: `$ echo 'abc' | build/bbr`
- parray: `$ echo '[a b c]' | build/bbr`
- nested parrays: `$ echo '[a [b c]]' | build/bbr`
- builtin macro: `$ echo '[bb/platform]' | build/bbr`

Also try the example at the top of this README. Put it into a file and `$ cat my-file.bbr | build/bbr`

8. Structure of this repository

- impl - implementations of bottom-bracket.
- docs - rendered docs for github pages (not user-facing)
- notes - almost anything
- programs - misc programs written in BB