Homoiconic C

Programming without a Language

ACM Reference format:
. 2017. Homoiconic C Programming without a Language . 1, 1, Article 1 (April 2017), 19 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1. INTRODUCTION

Homoiconic C is a data format for computation. It is designed to be a simple yet powerful alternative to both traditional programming languages and existing data formats.

1.1. A Brief History of Programming Languages

Early high-level languages (e.g., FORTRAN[1], COBOL[1], BASIC[1], ALGOL[1]) were strongly influenced by mathematics and artificial intelligence[1], leading to sophisticated precedence rules and complicated syntax. The primary alternative was Lisp[TBD [13];], defined – at least in theory – by a single data structure (the list) and a single rule (metacircular evaluation).

Lisp was not as efficient as, say, Fortran, but it was much more elegant and powerful. The common assumption was that eventually computers would become powerful enough that Lisp's inefficiencies would no longer matter. Sadly, the failure of hardware-optimized Lisp machines[1] marked the death of that dream.

1.2. Then Came C

Into that void stepped the C programming language[9]. C was derived from ALGOL, but borrowed (some would say butchered[1]) several powerful ideas from Lisp, such as macros[1] and function pointers[1]. In addition, the C memory model[1] mapped well onto modern processor architectures, enabling remarkably good performance. It also benefited from:

- A relatively simple and easy to read syntax
- Association with the UNIX operating system
- Self-sustaining[1], open source compilers

This led to C (along with its direct descendants, C++[1] and Objective-C[1]) completely dominating the field for systems programming; a situation which continues today, almost fifty years later. This led to the C syntax (with some borrowing from its sister language, the UNIX shell) becoming the baseline for virtually every mass-market language in use today[1].

This success was not entirely a good thing. C's memory model and type system optimized for efficiency at the cost of safety, leading to innumerable programming errors and security holes that still plague us today [1]. C's syntax and macros – a breath of fresh air when they first came out – feel clunky and dated in a world of scripting languages and functional programming, to the point where it is rarely taught to beginning programmers anymore.

1.3. Framing A New Hope

In the last decade, we have seen renewed innovation in systems programming, leading to languages such as Rust[1], Go[1], and Swift [1]. These are enough better and different than C that they have carved out their own niches (especially with the help of powerful patrons). However, they don't seem to have the factor-of-ten improvement necessary to displace or replace C, the way C did to assembly language.

Homoiconic C aspires to reach that goal by marrying the best aspects of C's syntax and memory model with the values of rigor and simplicity that characterized Lisp. The central innovation is a novel data structure we call a frame. Frames are simultaneously:

- Callable (like functions)
- Enumerable (like arrays)
- Associative (like structs)
- Inherited (like classes)

Our premise is that the multitude of structures found in programming languages boil down to just these four attributes. By encoding them in a single object, we hope to dramatically simplify both the structure of the language and the process of programming.

HC also incorporates:

- Homoiconicity, from Lisp : Frames are both code and data, making it trivial to manipulate code via higher-level abstractions.
- Dataflow, from the UNIX Shell

  Iteration and Input/Output are designed for simple pipeline-style composition.
- Effect Typing, from BitC[1] : By carefully annotating function boundaries for both constancy and immutability, we can safely alias data structures to get C-like performance while preserving the security of managed code.
- Lexical Scoping, from Scheme[1] : Everything inherits its current scope (like closures). In addition, evaluation of closures causes the result to inherit their scope, allowing them to be used as object factories.

1.4. About This Document

In the remainder of this document, we will cover:

- The syntax, operators, and access control that define the language
- Examples of using Homoiconic C to model object-oriented programming and HTML
- The current status and possible future directions

Integer		Non-Integer		Quote
0b010	binary	1/3	rational	string
0o1777	octal	123.456	float	# comment #
1234	decimal	123.456.E.-10	scientific	\length\blob
0xCAFE	hexadecimal	123.456.p123	version	'datestring'
0@Base64	Base64	+1.408.555.1212	phone	

Table 1. Literals Syntax

- Conclusions and comparisons to related work

2. LANGUAGE DEFINITION

2.1. Syntax

The core principle of Homoiconic C syntax is isomorphism: every syntactic construct refers to exactly one semantic concept. Put another way, it is modeless: syntax characters do not mean different things in different places. This enables vastly simpler parsers and a much shallower learning curve.

Like traditional data formats, HC is build around literals and the terminals that separate and organize them. HC simply extends that with identifiers, and a single grammatic construct, the expression. HC documents that only use literals and terminals are called data programs, or sometimes just congrams.

2.1.1. Expressions. Expressions are simply zero or more frames evaluated within the current context, then applied left to right (left fold [11]). We refer to this as the elliptical evaluator, in homage to Lisp's metacircular evaluator.

```
frame0 frame1 frame2 ...
```

Specifically:

(1) `frame0` is evaluated to `value0` (also a frame)
(2) `frame1` is evaluated to `value1`
(3) `value0` is called with the argument `value1`, producing `result1`
(4) `result1` is called with `value2` producing `result2`
(5) The final `result` is thus the value of the evaluated expression

The HC read-eval-print loop (REPL) uses ';' for the input prompt and '#' for the output prompt:

```
; frame0
# value0
```

That's pretty much all there is. There are no special forms, keywords, precedence rules, or other grammatic constructs. While we realize those may provide some efficiency gains for experienced programmers, we do not believe they are worth of cost of complicating the implementation and increasing the learning curve for new programmers. In particularly, we assert they are not actually necessary for a fully-functional programming system.

2.1.2. Literals. Like any good data format, HC has a rich set of literals, described in Table 1.

Separators		Aggregates	
,	end expression	(	begin group
;	void expression	)	end group
newline	end line	[	begin array
space	split subexpression	]	end array
		{	begin closure
		}	end closure

Table 2. Terminals Syntax

The most significant departure from traditional syntax is the use of matching smart quotes, which enables nesting and all but eliminates the need for escape sequences. With appropriate editor support (an atom package already exists), the user can type '"' as usual and still generate HC strings.

In addition, as part of HC's quest to be a universal data format, it natively supports:

- Base64 numbers
- arbitrary-length binary and hexadecimal bitstreams
- net-string style blobs
- a dedicated time type and literal (to avoid overloading integers)

By definition, literals evaluate to themselves:

```
; Hello, Quine!
# Hello, Quine!
```

This is true even at the file level. A data program consistent entirely of terminals and literals simply evaluates to (and prints out) itself, or "quines."

Because literals are also frames, and thus both code and data, they can be called like functions. Strings, for example, stringify and concatenate their argument.

```
; Hello, Homoiconicity!
# Hello, Homoiconicity!
; The Answer : 42
# The Answer: 42
```

Similarly, numbers replicate their argument that many times:

```
; 2 Repeats
# RepeatsRepeats
```

2.1.3. Terminals. Terminals in Homoiconic C look much like those in other C-based languages – see Table 2 – with a few twists.

Formally speaking, each expression should terminate in an ',' or ';', depending on whether we want to return the resulting value:

```
; My Statement;
; Self Expression,
# Self Expression
```

Variety	Example	Starts With	Contains
Label	variable	letter	letter, number, -
Operator	+	symbol	symbol
Control	$<- # return	$	any identifier
Argument	_^	_	_,^
Self	.	.	.

Table 3. Identifiers

This effectively allows us to distinguish "expressions" that return a value from "statements" that do not. However, trailing commas are optional, if neither separator is present will be inferred at the end of a line or aggregate.

Space plays an important role in binding, since we do not allow implicit precedence. Spaces create a new subexpression, so frames that have no space between them bind more tightly than those that do.

```
; Want 2 Live # evaluates left to right
# Want 2Live
; Want 2Live # evaluates ‘2Live‘ first
# Want LiveLive
```

This of course can also be done (with more visual clutter) via explicit grouping:

```
; Want (2 Live) # evaluates grouped expressions first
# Want LiveLive
```

Arrays work as you'd expect:

```
; [1, 1, 2, 3, 5] 8
# [1, 1, 2, 3, 5, 8]
```

Closures are simply lazy expressions, which evaluate their contents when invoked.

```
; {42; Life, The Universe, Everything.} ()
# Life, The Universe, Everything.
```

The result of the empty expression (`()`) is called `nil`, and represents the Boolean false. This will become important in Section 2.2 when we discuss conditionals.

Note that statements inside a closure represent values that are not returned, which becomes very powerful when we add identifiers.

Importantly, the empty closure `{}` becomes the "codify" operator, which converts arrays into closures:

```
; {} [Life, The Universe, Everything.]
# { Life, The Universe, Everything. }
```

2.1.4. Identifiers. Everything else is just an identifier. Identifiers can contain letters, numbers, or non-terminal symbols. They come in several varieties as described in Table 3.

Syntactically these are equivalent, with one small piece of syntactic sugar for operators; the dot is optional when used in a binary relation:

```
; 2 .+ 2
# 4

; 2 + 2
# 4
```

Assignment. Identifiers can be referred to via three different modes:

- `value`
- `.name`
- `@reference`

Assignment is just a simple expression setting a property with that name in the current context:

```
; .x 6 * 7;
```

This avoids the subtle and confusing distinction between differing "left-hand-side" and "right-hand-side" interpretations of an identical symbol x. That enables = to always mean a test for equality, rather than also being used for assignment.

We can use the value `x` to access that property in the same or any child context:

```
; x
# 42
; [x]
# [42]
```

To set the property in the context it was defined, rather than the local context, use a reference instead of a name:

```
; .y 7;
; (.x 11; @y 3;);
; x
# 42
; y
# 3
```

Names also provide an elegant way of manipulating data structures, still in the context of simple expressions:

```
; .base {.key 42};
; base .key # gets property
# 42
; .base .key 113; # sets property
; base
# {.key 113}
```

The space between base and _key is not necessary, but we use it to emphasize the fact that this is just an ordinary expression, not a special syntax.

Controls. As part of its quest to avoid arbitrary keywords, HC introduces the concept of controls, mirroring the hardware distinction between the control plan and the data plane. Controls are special closures which modify the behavior of the runtime, rather than simply returning values.

Flow Control. As a dataflow language, there should rarely be a need for direct manipulating the flow of control. But when there is, we use a control, e.g.:

$$

Terminate the program and return the argument ("exit")

$<

Terminate the current scope and return the argument (top-level return)

$<<

Terminate the parent scope and return the argument ("inner return")

Remember that every context is a self-contained entity, and thus `.` and `$<` refer only to that scope, even if it is a clause in a conditional. That's why we need multi-level returns.

Exceptions. Controls are also used to signal exceptions and errors. Exception controls begin with `$!` and return an interrupt value that propogates back up the call chain until it encounters an appropriate handler or aborts the program. The REPL, however, simply prints out the exceptional value:

```
; .scope (.a 1, b. 2);
; scope.c
# $!missing-key .c
```

Note that in the interest of clarify, we voilate our usual rule and use English names for exceptions. We do still intend to make these fully localizable, so at least beginning programmers can experience an environment that doesn't require them to also learn a foreign language.

Arguments. Use `_` as the anonymous argument, representing everything this frame was called with:

```
; .triplicate {3 _};
; square Baby
# BabyBabyBaby
```

This is useful not just for closures, but for representing the command-line arguments for the entire script.

When you apply something to a closure, it is effectively inserting that argument into the inheritance hierarchy. Thus we can access properties of the argument directly, rather than explicitly calling `_`.

```
; .mag {(x * x) + (y * y )};
; mag (.x 1; .y 2;)
# 5
```

You can skip over the argument to access the enclosing scope (one level above) using the `_^` identifier (also known as `super`).

```
; .print-arg { var };
; .print-parent { _^.var };
; .var parent;
```

Type		Bind	
<	begin type	~~	get type
>	end type	^	bind type
~	has type	^^	bind return

Table 4. Type Operators

```
; print-arg(.var arg)
# arg
; print-parent(.var arg)
# parent
```

Since objects capture the scope where they are created, this even allows closures to be called with implicit arguments to access the enclosing scope:

```
; .x 3;
; .y 4;
; mag []
# 25
```

Implicit arguments are a code smell, and will generate a warning. However, they can be very useful when debugging or refactoring. That may seem dangerous, but the data protection rules (below) largely limit what the called function can do to the calling scope.

TODO: Self.

2.2. Operators

Homoiconic C predefines a small number of top-level operators used by all frames. These provide functionality other languages often hard-code into their syntax. Note that specific datatypes may define their own operators (e.g., `+` for numbers) not covered in this section.

2.2.1. Type Operations. Like all modern statically typed languages, HC relies heavily on type inference. By using dataflow, we also eliminate the need to define variables for loops or temporary variables. In addition, a frame that is callable with any other frame has the generic type, and thus does not need to be declared.

However, there are times we do need to explicitly annotate what kind of frames we are expecting, especially at function boundaries. For those cases we use the type operators in Table 4.

Note that in HC, type is declared using a pair of operators, not a syntactic construct. This works because types are just expressions, and thus do not need to change the evaluation rules.

The All Type. The empty type `<>` is known as all. As the opposite of the empty expression `()` nil, it acts as the boolean `true` value.

```
; <>
# <>
; ().!
```

Content			
Role	All	Data	Metadata
Equality	=	==	===
Map	|	||	|||
Fold	&	&&	&&&

Table 5. Content Operators

```
; <>
; <>.!
; ()
```

Type Membership. The has-type operator ~ tests whether an object belongs to particular type, returning true (all) or false (nil). Every object is a member of all, while nothing is a member of nil:

```
; 1 ~ <>
# <>
; 2 ~ ()
; ()
```

Type Signatures. We use ^ to specify the type signature of a closure, which may include optional defaults:

```
; .join-name (.first Jane, .last) ^ {last , first};
; join-name (.first John, .last Doe)
# Doe, John
```

Types are generous, in that you are allowed to specify additional properties, but it is an error to omit properties that do not have a default:

```
; join-name (.middle Q, .last Doe)
# Doe, Jane
; join-name (.middle Q)
# $!invalid-argument-list (.middle Q, $!missing-required-argument .last;)
```

The return type is usually inferred, but can be specified explicitly using a trailing ^^.

Type Values. The simplest way to construct a type is to use the ~~ operator to extract the type of a known object:

```
; Q ~ ~~
# <> # true
; Q ~ ~~1
# () # false
```

2.2.2. Content. Homoiconic C defines three content operators, each of which come in three flavors as shown in Table 5.

These flavors determine whether we consider the whole object, or just the data or metadata separately.

Operator	Name	On Frame	On Nil
?	if	Call with ()	Return ()
:	else	Return ()	Call with ()

Table 6. Conditional Operators

Equality. = is the usual equality test:

```
; .a [113, .p 887];
; .b [113, .p 661];
; .c [443, .p 887];
; a = a
# <>
; a = b
# ()
; a == b
# <>
; a == c
# ()
; a === c
# <>
```

Iterators. We use | for map, in homage to the UNIX pipeline.

```
; [1, 2, 3] | { _ + 1 } # will warn, since `_` is not defined on generic frames
# [2, 3, 4]
```

Similarly, we use & for reduce:

```
; [1, 2, 3] & { . + _ }
# 6
```

The operators also work with files and network ports, reading one line (or object) at a time, greatly simplifying common I/O operations (a la the UNIX shell).

2.2.3. Conditionals. Homoiconic C's sole conditionals are a pair of binary operators analogous to C's ternary operator, albeit with slightly different semantics. These are described in Table 6.

Note that these are not special forms, but simply defined with one behavior on nil and the opposite on regular frames.

```
; 1 ? {2 + 2}
# 4
; 1 : {2 + 2}
# ()
```

```
; () ? {2 + 2}
# ()
; () : {2 + 2}
# 4
```

Which, when the first expression does not return nil, acts just like C's ternary operator:

```
; 1 > 5 ? (2 * 50) : 10
# 10
```

Note that applying nil to anything other than a closure has no effect, so conditionals work just as well with simple expressions as they do with lazy blocks.

2.2.4. Importing Modules. Importing external modules into a program has come a long way from C's text-based `#include` statement. Modern imports are typically expected to perform three roles:

(1) Match a module name with a local package on disk, often downloaded via a package manager
(2) Read information from that module
(3) Import names from that module into the current namespace

This functionality must be available as a primitive, since it is necessary for adding other functionality.

We use the `<-` import operator to match a name against both the online registry (TBD) and the local path, and load it into a property with the same name. Registry settings can be configured via the `.hconfig` file.

```
; <- .module;
; module
# (.prop 1, .another-prop 2)
```

We can also bind it to a different name:

```
; .alias <- .module;
; alias.prop
# 1
```

or directly into the current namespace `..`

```
; . <- .module;
; prop
# 1
```

While not recommended in general, direct import allows apparently "built-in" functionality to provided via a prologue, rather than hard-coded into the language. This reflects HC's philosophy to provide the same functionality we have as language designers to end users wherever possible.

2.3. Access Control

Arguably the primary source of bugs, clutter, and complexity in programming languages is knowing "who can perform which actions on what data," which we refer to somewhat loosely as "access control." We consider this problem so important we spend the bulk of our "syntactic budget" on giving programmers

Privacy	Constancy	
public	variable	immutable
_protected	Constant	mutable_
__private		mutating:

Table 7. Access Modifiers

an easy way to specify and detect access modes. This is done by encoding those rules in the identifiers themselves, as seen in Table {#sec-table-access}.

Importantly, these are modifiers not different identifiers. So, for example, the constant `COOL` in one scope overrides its case duplicate `cool` from a parent scope. For natural languages without separate upper and lower case, we use an initial letter `K` to denote the constant version.

2.3.1. Encapsulation. The simplest form of access control is encapsulation, restricting the ability of external objects to even see certain properties. Here we follow the informal conventions often used in C programs:

`public`
: visible to everyone

`_protected`
: not visible to parents or peers, but still visible to children

`__private`
: not visible to anyone, even children

For example:

```
; .see-me {
 .my-public-value 42;
 ._my-protected-value 21;
 .__my-private-value 7;
 .child {
 my-public-value,
 my-protected-value,
 my-private-value,
 }
};
; see-me.child()
# [42, 21, $!is-private .my-private-value]
; see-me.my-public-value
# 42
; see-me.my-protected-value
# $!is-protected .my-protected-value
; see-me.my-private-value
```

```
# $!is-private .my-private-value
```

2.3.2. Effect. Rather than specifying call-by-value or call-by-reference, HC is designed around the BitC[3] model of effect typing. Shapiro et al proved it is possible to have a sound and complete systems language if we explicitly annotate identifiers for both constancy and mutability (which together we call effect) at each context boundary. This gives the compiler enough information to know how and when to safely copy or share data structures. Put another way, effect typing makes it as easy to allow side effects for performance as it is to restrict side effects for safety.

Unfortunately, BitC could not accommodate such annotations within their Lisp-like syntax[1]. Inspired by their work, we have designed the syntax of our identifiers to explicitly support effect typing:

`variable`
: Does not begin with an uppercase letter. Can be reassigned.

`CONSTANT`
: Begins with uppercase letter. Can not be reassigned.

`immutable`
: Has no suffix. Can not be modified in place.

`mutable_`
: Trailing underscore. Can be modified in place.

`mutating_method:`
: Trailing colon. Can modify its parent context. Returns parent.

Please note that these particular conventions are preliminary, and may change in future versions based on empirical tests of readability and intuitiveness. Since HC is just a data format, we plan to explicitly version documents and enable automatic migration between incompatible versions.

Constancy. BitC's first key insight is that constancy (whether an identifier can be assigned a new value) is distinct from mutability (whether the referenced value can be modified in place). In other words, constancy is a property of the source object, whereas mutability is a property of the destination object.

```
; .variable 42;
; .Constant 21;
; .variable 113
# 113
; .Constant 7
# $error{$is-constant .Constant}
```

Mutability. The second key insight from BitC is that effect is a property of names rather than of values. Object literals have no effect restrictions on their own; this is contrast to, e.g., Apple's Cocoa frameworks[1], where mutability is an inherent part of the object's constructed type. Instead, every object is in principle mutable, but as long as it is only referenced from immutable handles the compiler can safely share a single instance between them, avoiding unnecessary copies.

Conversely, objects that are referenced and passed via mutable handles are explicitly "aliases" of each other, so the compiler knows that it needs to synchronize changes between local copies, ensure serialization between different threads, etc.

In order to ensure both immutable and mutable handles support the same methods, HC explicitly tracks which methods mutate their parent scope via a trailing colon (`:`). When a mutating method is called on an immutable object, it simply performs a copy-on-write, returning a new object. To enable this, mutating methods can not explicitly return a value, but implicitly return their parent (e.g., "this"; see Section {#sec-oops} for more details).

```
; .fixed (
 .hic Object;
 .property 42;
 .accessor { property }
 .mutator: { @property _; }
);
; fixed.accessor()
# 42
; .varying_ fixed.mutator: 113;
; varying_.accessor()
# 113
; fixed.accessor()
# 42
```

3. APPLICATIONS

While the above language may seem simple to the point of simplistic, it has a surprising amount of power. Here are a few examples of what it can do.

3.1. Congrams

As mentioned above, HC programs that consist only of literals other than closures are called congrams, which are just pure data. This provides a once-in-a-lifetime opportunity to bring rigor and consistency to a variety of organic data formats developed over the Internet's brief history.

3.1.1. HCSV. Homoiconic C may finally provide a well-defined alternative to the ubiquitous CSV[1] file. A properly-structured `.hcsv` file is just as compact as CSV, with two important differences: The header row, if any, consists of a list of names Strings must be (smart) quoted

```
.first-name, .last-name, .phone-number
John, Doe, +1.408.555.1212
Jane, Smith, +1.650.555.1212
```

3.1.2. HCSON. HC can also emulate the popular JSON[1] format, or more precisely its CoffeeScript cousin CSON[1].

```
.first-name John, .last-name Doe, .phone-number +1.408.555.1212
.first-name Jane, .last-name Smith, .phone-number +1.650.555.1212
```

3.2. Object-Orientation

Perhaps surprisingly, it is possible to implement a complete object-oriented programming system using only the above primitives. All we need are our access control rules plus the super identifier `_^`.

3.2.1. Singletons. Let's start with a simple singleton object containing private data:

```
; .my-object_ (
 ._property 13;
 .getProperty { _property }
 .setProperty: { @property _}
);
; my-object_.getProperty()
# 13
; my-object_.setProperty: 42;
; my-object_.getProperty()
# 13
```

3.2.2. Classes. To turn that into a class, we simply make it a closure which returns a frame analogous to that singleton:

```
; .my-class {
 ._property _;
 .getProperty { _property }
 .setProperty: { @property _}
};
; .my-instance my-class 3;
; my-instance.getProperty()
# 3
```

3.2.3. Inheritance. Even inheritance is already accounted for, simply by explicitly specifying its parent scape:

```
; .my-subclass {
 ._^ my-base-class
};
```

There is no built-in support for multiple inheritance. However, because inheritance is just another expression, you are welcome to define your own:

```
; .my-inheritance { create your own };
; multiclass {
 ._^ my-inheritance [my-base, another-base]
};
```

4. IMPLEMENTATION

Homoiconic C is available as the `hc` interpreter via the `hclang` node.js module, written in TypeScript.

```
$ npm install -g hclang
$ hc -e "Hello, Homoiconic C!"
```

It is under active development, and available on GitHub[1] under the MIT Open Source license.

4.1. Architecture

The core runtime objects live in the `src/frames` directory, where they are implemented as TypeScript classes. Generally speaking there is one class for each primitive and aggregate frame, plus abstract classes for the generic Frame, FrameAtom, and FrameList.

The interpreter lives in the `src/execute` directory, and is also implemented using frames. Every frame knows how it is represented, and the lexer uses that information to dynamically generate a lookup table from a list of frame classes. Strings iterate as a list of symbols, so the lexer generates tokens from the input string by evaluating evaluating a specialized program. Similarly, terminals act as controls grouping tokens into expressions and aggregates.

Finally, operator bindings and currying are defined in `src/ops`, and a preliminary wrapping of HTML called MAML lives in the `src/maml` directory.

4.2. Status

As of April 2017, the implementation is about 30% complete. We have implemented the core runtime frames and the parsing pipeline, and are able to evaluate simple expressions as arguments to the command-line. Supporting the remaining primitives and aggregates, as well as implementing a full REPL, should be fairly straightforward.

To date we have only implemented a single operator, iteration over properties. This is enough to serve as a proof-of-concept, though it will undoubtedly need to be refactored to support the full range of operators.

We have not yet begun to implement access control, though we have a high degree of confidence in the design based on previous prototypes. The design of the overall type system is somewhat less mature, especially where it overlaps with inheritance. This is also the case for the module system, which is still in the conceptual stage.

4.3. Next Steps

4.3.1. Documentation-Driven Development. The first priority is obviously completing the interpreter. The main tool we plan to use to get there is enabling executable documentation.

This document, for example, is written in a dialect of Markdown called Madoko[1] which uses triple back-quotes (“““”) to delimit code fragments. We could invert that by having Homoiconic C use those quotes as

the delimiters for doc-strings, so it can directly (and sequentially) evaluate the code fragments. Next, we add an input stage that tracks the raw source, and an output stage treats the semicolon-hashtag pairs as acceptance tests.

Once that is in place, we can literally run our documentation simply by prefixing this file with:

```
#!/usr/bin/env hc -doctest
\‘\‘\‘
```

The process would thus involve rewriting the documentation and the source code together until they are both complete and consistent.

4.3.2. Compilation. Once the interpreter is finished, we plan to work on a self-hosting transpiler. Written in Homoiconic C, it would convert `.hc` files directly into executable JavaScript. This may be rather heavyweight, however, since all the JavaScript objects would be Frame instances.

The second step would be compiling directly to `asm.js`[1]/ WebBinary[1]. In particular, the goal would be to first wrap asm.js as a "congram" so that it can be easily manipulated directly from Homoiconic C.

That experience would serve as a useful stepping stone towards a true native code compiler. Rather than simply using LLVM[1], we would prefer to use that opportunity to reexamine the very nature of intermediate representations and assembly language. The theory underlying Homoiconic C suggests that "switching circuits" may do a better job of abstracting the underlying electronics than traditional von Neumann architectures[1]. While admittedly a long shot, if successful it could enable dramatic breakthroughs in performance, power consumption, and hardware evolution.

5. RELATED WORK

5.1. BitC

Homoiconic C could be considered a reincarnation of the BitC[3] project, which attempted to add a "bit of C" to Lisp in order to bring rigor to systems programming. As far as we know, they are the only group to have tackled the gap between the elegance of lambda calculus and efficient use of actual computing hardware. They appeared to have stalled due to a lack of funding, and also the difficultly of grafting a rich enough type system onto Lisp syntax.

Homoiconic C hopes to avoid those problems through the pursuit of extreme simplicity. We start with a variant of the Lisp evaluation model explicitly designed to work with numbers and strings, not just lists. Next, we carefully design our identifiers to encode the critical information needed for "sound and complete type inference" across invocation boundaries. Finally, we leverage Open Source to bootstrap the project so that we are not dependent on external funding.

5.2. Julia

Though developed independently, Homoiconic C shares many goals and features with the emerging Julia[5] language for numerical computing: multiple dispatch homoiconicity, macros, and meta-programming * a type system optimized for performance

We admire and hope to emulate Julia's achievements, particularly in terms of interoperability with existing languages and creative support for concurrency. Though Julia is currently far ahead in terms of

both implementation and ecosystem, we believe that HC's use of BitC's type system plus the extreme simplicity of the syntax and implementation will still enable us to provide a compelling alternative.

5.3. Mathematica

The most widely used homoiconic language in use today is probably Mathematica [7]. However, despite several notable attempts to broaden its reach, Mathematica remains primarily a prototyping tool within Wolfram's products rather than remains primarily a prototyping tool within Wolfram's products rather than a general-purpose language[8].

5.4. STEPS

Homoiconic C was also inspired by and shares many goals with Alan Kay's ambitious STEPS [2] project for reinventing programming. From our perspective, the tragic flaw with his approach was starting from the (admittedly ubiquitous) assumption that computation should be built upon mathematics. Despite a promising start at replacing operating systems with a robust language runtime, the project appeared to eventually collapse under the weight of its leaky abstractions.

We believe it is possible to achieve his dream by building on a different foundation. Rather than the traditional mantra of "Sequence, Selection, and Repetition", Homoiconic C is build around "Dependency, Effect, and Interrupts." We believe this choice of basis enables us to generate "leak-proof" abstractions that scale seamlessly from individual circuits to complex distributed platforms.

5.5. Water

The idea of expressing programs in a data format is not in itself new. A relatively modern attempt was Water[6], which attempted to program web applications using a more concise variant of XML. Though extremely clever, it suffered not just from the awkwardness of XML as a data format, but from the use of existing imperative programming paradigms.

HC plan to avoid that fate by adopting the more streamlined dataflow paradigm, supported by a custom-designed data format optimized for readability.

6. CONCLUSION

We believe the historical accident of confusing computation with mathematics and programming with natural languages has resulted in massive accidental complexity. We believe this is a primary cause of the buggy software, complicated tools, and steep learning curve that plague software engineering today.

By eliminated that complexity, Homoiconic C hopes to usher in a new Golden Age of software that is: Inherently and provably secure Both inherently efficient and easily optimized Easy to learn, read, and modify Trivial to statically analyze, visualize, and automatically evolve

If Homoiconic C fulfills that promise, we can finally make programming an everyday skill, used by ordinary people (even children) to solve the problems they care about, in the same way they use writing and arithmetic. Not everyone will be an expert or professional programmer – anymore than everyone who sings is an expert or professional musician – but the ability to read, understand, and personalize code will be available to all.

It may be true that there is no silver bullet[1]. But we believe Homoiconic C can at least be a "diamond sword", enabling courageous individuals to bring safety, performance, usability, and clarity to the software that impacts their corner of the world. We hope you will join us on that quest.

7. APPENDICES

7.1. Appendix A. On Turing Completeness

Turing undecidability, like Godelian incompleteness, starts by assuming "basic arithmetic" (add, subtract, multiply, divide) – i.e. the Peano Axioms. However, this glosses over the fact that division is a "type violation", and can't be fully represented using the same data structures as for addition and subtraction.

We believe that a better starting point for modeling computation are the Presburger Axioms. These give up multiplication and division as first-class operations (though you can emulate them to some extent using repeated addition and subtraction, respectively). The big win, though, is that Presburger arithmetic is both consistent and complete. This eliminates the halting problem, and massively simplifies analyses (though it may restrict what is possible).

Instead of Turing completeness, we prefer to focus on "Circuit Universality" (a la Scott Aaronson): the ability to represent the effect of any Boolean circuit, including multiple levels of abstraction above them.

REFERENCES

Anonymous. Citation Needed. N/A, 2000.

Alan Kay et al. Steps toward the reinvention of programming. TBD, 2000a. URL http://www.vpri.org/pdf/tr2008004_steps08.pdf.

Shapiro et al. Sound and complete type inference in a systems language. TBD, 2000b. URL http://www.cs.jhu.edu/~swaroop/aplas.pdf.

H. Abelson G. Sussman. Structure and Interpretation of Computer Programs. MIT Press, 1980.

The Julia Language. Julia >> docs >> metaprogramming, 2017. URL https://docs.julialang.org/en/stable/manual/metaprogramming/.

Mike Plush. Introduction to water: A new native web services programming language, 2002. URL http://www.informit.com/articles/article.aspx?p=27567.

Leonid Shifrin. Metaprogramming in mathematica, 2012a. URL https://mathematica.stackexchange.com/questions/2335/metaprogramming-in-mathematica.

Leonid Shifrin. Can mathematica be regarded as a software prototyping environment?, 2012b. URL https://mathematica.stackexchange.com/questions/1314/can-mathematica-be-regarded-as-a-software-prototyping-environment.

TBD. Clang, 2000a.

TBD. Forth, 2000b.

TBD. Leftfold, 2000c.

TBD. Liebniz, 2000d.

TBD. Lisp, 2000e.

TBD. Lua, 2000f.

TBD. Self, 2000g.

TBD. Typescript, 2000h.

TBD. Unixshell, 2000i.