

7 Years Later:

Declarative, Inquisitive, *then*  
Imperative

—

Samuel A. Falvo II <[kc5tja@arrl.net](mailto:kc5tja@arrl.net)>

November, 2017

# Homework! (Optional)

You are asked to watch this presentation by Rich Hickey, the author of Clojure, an increasingly popular Lisp dialect in Enterprise computing environments.

<https://www.infoq.com/presentations/Simple-Made-Easy> (~ 1 hour.)

Specifically, pay attention to his distinction between *simple* and *easy*, and how he uses them to gauge modern programming, languages, libraries, *et. al.*

Chuck Moore once said in a video captured by Jeff Fox at iTV that he cannot say what Forth is, but he knows what it is when he sees it. This video may help shed some light on why Forth works so well for us; **but, it's not the whole story.**

# Summary of Hickey's Distinction

## Simple

One braid; involving only one concept.

*versus*

## Complex

Many braids; involving many concepts.

## Easy

Close at hand, requiring little effort.

*versus*

## Hard

Solid, requiring substantial effort.

# Is Forth *Simple*?

## Yes!

Forth cuts out the more advanced concepts you've probably come to expect in modern languages, allowing you to focus almost exclusively on your software's *operational semantics*. Concepts you **won't** find in Forth include:

1. **Heavy syntax.**
2. A long **edit, compile, run, cry, debug loop.**
3. **Packages, versions,** and their dependency hell.

## No!

A Forth environment is *simpler* than Smalltalk; yet, get *even one thing* wrong, and the whole house of cards comes tumbling down and you'll never know why. It *complects* many things that all absolutely must work right to have a functional Forth environment:

1. **Buffer management** (including stacks).
2. **Dynamically-scoped** variables.
3. **Context.**

# Is Forth *Easy*?

## Yes!

Forth is incredibly easy to use and to write software in. Learn just a few simple rules in a day or two, and a motivated programmer can debug even legacy code. Greater code locality means improved code comprehension, thanks to the lack of these things:

1. Heavy **syntax**.
2. Declared **types**.
3. **Flat** global environment.

## No!

Some of these harder points of Forth are actually *features*, not bugs. Constantly needing @ and ! is a good reminder you probably need to reduce how much or how frequently you access state, for example. Others, not so much, like + vs F+.

1. **Typed operators**.
2. **Global state**, not local state.
3. **DIY** aggregates.

# Convention over Configuration

From [https://en.wikipedia.org/wiki/Convention\\_over\\_configuration](https://en.wikipedia.org/wiki/Convention_over_configuration)

*Convention over configuration* (also known as *coding by convention*) is a software design paradigm used by software frameworks that attempt to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility.

# Convention over Configuration

## Configuration


```
MODULE Lists;
TYPE
  Node = POINTER TO NodeDesc;
  Node = RECORD
    next: Node;
    datum: INTEGER;
  END
```

```
PROCEDURE Insert*(list, node: Node);
VAR n1, n2, n3: Node;
BEGIN
```

## Convention

```
: next! ( an al - an ) @ OVER ! ;
: Insert ( al an - ) OVER next! SWAP ! ;
```

**Simple**, but relatively **hard**. 10 LOC and still no idea how Insert actually works. However, the architecture and schema is made explicit, so that's good!



# Convention over Configuration

## Configuration

```
MODULE Lists;
TYPE
  Node = POINTER TO NodeDesc;
  Node = RECORD
    next: Node;
    datum: INTEGER;
  END
PROCEDURE Insert*(list, node: Node);
VAR n1, n2, n3: Node;
BEGIN
```

## Convention

```
: next! ( an al - an ) @ OVER ! ;
: Insert ( al an - ) OVER next! SWAP ! ;
```



Just as **simple**, and far **easier** to write.

However, you must visualize layout of nodes in memory to make sense of it. **Complects** semantics with architecture, which is now implicit. Is this any better?



# Convention over Configuration

From [https://en.wikipedia.org/wiki/Convention\\_over\\_configuration](https://en.wikipedia.org/wiki/Convention_over_configuration)

*Convention over configuration* (also known as *coding by convention*) is a software design paradigm used by software frameworks that attempt to decrease the number of decisions that a developer using the framework is required to make without necessarily losing flexibility.

SO...

How do we use **convention** to minimize the intellectual cost a Forth programmer has to pay during program design, while still retaining the benefits of Forth's simplicity and facility?

# Declarative, Inquisitive, *then* Imperative!

**Convention:** use Forth words that fall within the following three categories:

- Strongly prefer writing software **declaratively**. Tell the computer what you want, *not how*.
- If you need to decide what to do based on an external factor, **inquire**. *Ask* the computer a question.
- At some point, you'll have to **impart** to the computer *how* to do something. Ideally, you want to *minimize* imperative words as much as you can.

# Declarative, Inquisitive, *then* Imperative

**Declarative** words *establish a truth*. These can be broadly broken up into two categories:

- **Preconditions.** A word can declare something that *must* already be true before continuing execution. If not, the anomaly is handled for you (somehow).

**readable      undefined      -empty**

- **State transitions.** A word can define what *end state* is to be achieved. Its implementation is responsible for minimizing the amount of work necessary to realize this state.

**parsed      defined      reduced**

# Declarative, Inquisitive, then Imperative

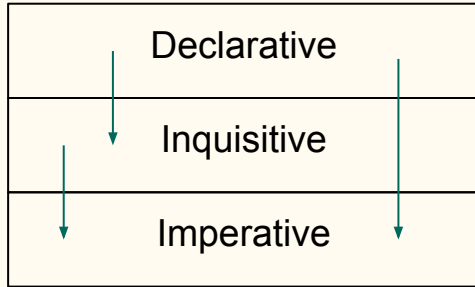
**Inquisitive** words *sense* the current state, and answers a question about it. Absence of intervening declaratives or imperatives implies consistent, repeatable answers.

`readable?`   `undefined?`   `-empty?`   `XY@`

**Imperative** words *unconditionally takes some action*. These are the normal Forth words you've been writing for years.

`read`   `insert`   `pop`   `AT-XY`

# Declarative, Inquisitive, *then* Imperative



Declarative and Inquisitive words build on top of imperative words. Both kinds of words are *idempotent by design*.

Declarative words may also use inquisitive words, but not vice versa.

Imperative words can do whatever they want.

# Hello World with DItI

**Problem statement.** We need a virtual character matrix to represent the text currently shown on the bitmapped screen, and to store characters the user types into fields on the screen. When the a new input form is shown, the whole display changes at once; we never scroll. The screen is 80 characters wide, 25 characters tall.

Write a lexicon to abstract this interface, allowing legacy “block-mode” applications to use contemporary bitmapped graphics displays.

# Hello World with DItI: Screen

```
80 CONSTANT #cols
```

```
25 CONSTANT #rows
```

```
#cols #rows * CONSTANT /matrix
```

```
CREATE matrix
```

```
/matrix ALLOT
```

```
VARIABLE X
```

```
VARIABLE Y
```

# Hello World with DItI: Cursor

```
: up          -1 Y +! ;
```

**Hmm...** This works, until we reach the top of the screen matrix. Then we risk overwriting memory which belongs to something else.



# Hello World with DItI: Cursor

```
: constrained  Y @ 0< IF Y OFF THEN ;  
: up          -1 Y +! constrained ;
```

**Note** how `constrained` establishes the truth that the cursor position remains valid at all times. It *enforces* an invariant.

# Declaration Property

**Declarations** all have this one characteristic in common:

. . . . DECLARATION . . . .



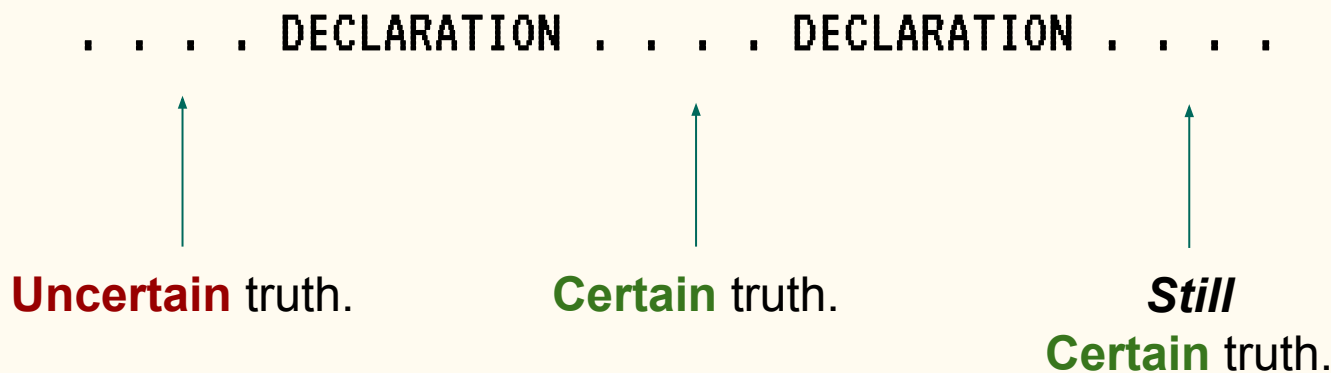
**Uncertain** truth.



**Certain** truth.

# Declaration Property

**Declarations** all have this one characteristic in common:



Declarations *may* have side-effects, but are **always idempotent**. Establishing the same truth twice or more is as good as doing so once.

# Hello World with DItI: Cursor

```
: constrained  Y @ 0< IF Y OFF THEN  Y @ #ROWS >= IF #ROWS 1- Y ! THEN  
              X @ 0< IF X OFF THEN  X @ #COLS >= IF #COLS 1- X ! THEN ;
```

```
: up          -1 Y +! constrained ;  
: dn          1 Y +! constrained ;  
: rt          1 X +! constrained ;  
: lf          -1 X +! constrained ;  
: at-xy      Y ! X ! constrained ;
```

Notice how we handle “errors” (*any* anomaly to a desired invariant) **locally**. No exceptions are thrown, no error flags are set, nor special sentinel values returned.

**It’s not always possible** to write code like this; but, do strongly prefer this approach. The more local the control flow, the easier it is to understand the code.

# Hello World with DItI: Cursor

`constrained` is an example of a *state-transitioning* declarative word, which in this case enforces a post-condition. Can we do the same with **pre-conditions** as well? Yes; but, in this particular case, post-conditions were both *easier* and *simpler*. Consider:

```
: -top      Y @ 0= IF RDROP THEN ;
: -bottom   Y @ #ROWS-1 = IF RDROP THEN ;
: up        -top      -1 Y +! ;
: dn        -bottom   1 Y +! ;
```

Clearly this is the more complex of the two code fragments: 2\* LOC *and* conditionals *and* non-local control flow are required.

Beware of conditionals of *any* kind; they're necessary, but **always increase complexity**.

# Hello World with DItI: Text Output

Drawing text to the screen requires that we both update the character matrix and adjust the pixels on the screen to match what the matrix says should be there. (Remember, we're working with a bitmapped display!)

```
: point ( - a )    Y @ #COLS * X @ + matrix + ;  
: placed ( c - )  DUP point C! EMIT ;
```

This works fine as long as `X @` and `Y @` reflect the actual console's cursor position. How can we enforce this at all times? Post-conditions won't work nearly as well here.

# Hello World with DItI: Text Output

We will now use a declaration to enforce a **pre-condition**.

```
: point ( - a )    Y @ #COLS * X @ + matrix + ;  
: positioned      X @ Y @ AT-XY ;  
: placed ( c - )  DUP point c! positioned EMIT ;
```

# Hello World with DItI: Keyboard Handling

You've already seen declarations and imperatives at work. Let's take a look at inquisitions now. But, first, I need to set the stage with more applications of declarations.

```
: handled ( k- ) ( one of... ) cursor enter graphic DROP ;  
: run          initialized BEGIN KEY handled AGAIN ;
```

We use declarations to form a **multi-way selection**. The word `handled` says that a key can be one of either a cursor key, the ENTER key, a graphic character, or is **ignored**.



# Hello World with DItI: Keyboard Handling

Note how the very nature of declarations allows us to create hierarchically organized sets of rules, trivially.

```
: cursor ( k-k ) ( one of... ) up down left right ;  
: handled ( k- ) ( one of... ) cursor enter graphic DROP ;  
: run          initialized BEGIN KEY handled AGAIN ;
```

Here, `cursor` says that the choices available are up, down, left, right; anything else is handled by `enter`, `graphic`, etc. With this, we can compose fragments of decision trees.

# Hello World with DItI: Keyboard Handling

Now we can show how to decide if one of our characters is a cursor movement key.

```
: up ( k-k )      DUP CTRL J = IF up THEN ;  
: cursor ( k-k )  ( one of... ) up down left right ;  
: handled ( k- )  ( one of... ) cursor enter graphic DROP ;  
: run            initialized BEGIN KEY handled AGAIN ;
```

Note that I freely *redefine* `up` to mean something relevant in the context of this decision tree and how I make use of Forth's hyperstatic global environment to invoke the previous declaration. Those familiar with OOP will recognize this immediately as *inheritance*.

# Hello World with DItI: Keyboard Handling

However, it would be better if we could eliminate the Forth plumbing and just focus on the intent of the code. We do that by factoring the question out into an inquisitive word:

```
: up? ( k-kf )    DUP CTRL J = ;  
: up ( k-k )      up? IF up THEN ;  
: cursor ( k-k )  ( one of... ) up down left right ;  
: handled ( k- )  ( one of... ) cursor enter graphic DROP ;  
: run            initialized BEGIN KEY handled AGAIN ;
```

This simple example doesn't do it justice; however, my time is limited.

# Inquisitive Properties

TruthX . . . . . X? . . . . .



At this point, we know  
that X must be true, so...



**X? must answer true.**

# Inquisitive Properties

TruthX . . . . . X? . . . . . X? . . . . .



At this point, we know  
that X must be true, so...



X? must answer true.



X? must **still** answer  
true.

# Did You Notice?

- **Definition size.** I used two lines of code to define a word *exactly once* in this talk. The overwhelming majority of definitions are *exactly one line* long.
- **Comprehension.** Given you are aware of the larger context and project conventions, you can *isolate any individual definition* and understand what it does.
- **Code Locality.** Relevant fragments of code tend to be co-located (or at least explicitly referenced via **INCLUDE** or **LOAD**) within a few LOC or screens of use. This avoids the massive problem of tracking down important code in larger programs.
- **Design By Contract.** Routine use of preconditions and postconditions to enforce invariants and locally handle anomalies naturally leads to more robust software since you deal with more errors sooner, rather than later.

# Kung Fu

Forth is not just a programming language. It is **kung fu** in the truest sense of the phrase: skill acquired through hard work and practice.

The rules for Declarative, Inquisitive, then Imperative are not to be taken too literally. Use them to guide your development and exploration. Break them when it's appropriate.

Thank you!

Samuel A. Falvo II <[kc5tja@arrl.net](mailto:kc5tja@arrl.net)>

